

Linux: эффективная многопроцессорность

Используем Go

Проект книги

Автор: Олег Цилюрик

Редакция 5.145

29.02.2024г.

© 2024

Оглавление

Предисловие.....	7
Предназначение и целевая аудитория.....	8
Код примеров и замеченные опечатки.....	10
Чего нет в этой книге.....	12
Соглашения и выделения, принятые в тексте.....	13
Напоминание.....	14
Источники информации.....	15
Часть 1. Инструментарий языка Go.....	16
Предыстория Go.....	17
Разворачиваем экосистему Go.....	26
Неформально о синтаксисе Go.....	67
Структура кода: пакеты, модули, импорт.....	134
Новости от последних версий.....	148
Параллельные линии развития, TinyGo.....	153
Часть 2. Конкурентность и многопроцессорность.....	162
Процессоры в Linux.....	163
Параллелизм и многопроцессорность.....	177
Масштабирование.....	204
Часть 3. Некоторые примеры и сравнения.....	226
Осваиваемся в технологии Go.....	227
Структуры данных, типы и их методы.....	243
Строки, руны и UNICODE.....	250
Объектно-ориентированное программирование.....	261
Элементы функционального программирования.....	275
Сети, Интернет и WEB.....	286
Скоростные и другие сравнения языков.....	291
Многопроцессорные параллельные вычисления.....	306
Разработка через тестирование.....	340
Заключение.....	342
Об авторе.....	343

Содержание

Предисловие.....	7
Предназначение и целевая аудитория.....	8
Код примеров и замеченные опечатки.....	10
Замечание о версиях.....	11
Чего нет в этой книге.....	12
Соглашения и выделения, принятые в тексте.....	13
Напоминание.....	14
Источники информации.....	15
Часть 1. Инструментарий языка Go.....	16
Предыстория Go.....	17
«Отцы-основатели» о целях и мотивации.....	17
Применимость: беглый взгляд.....	17
Go, C, C++, Java, Python, Rust, Oberon	18
Управление памятью и истоки ненадёжности.....	24
Источники информации.....	25
Разворачиваем экосистему Go.....	26
Создание среды.....	26
Стандартная инсталляция.....	26
Версии среды.....	28
Управление системными альтернативами.....	29
«Самая последняя» версия.....	29
Смена версий.....	33
Проверяем: простейшая программа.....	35
Простейшее приложение.....	36
Библиотеки статические и динамические.....	37
Компиляция или интерпретация.....	38
Выбор: GoLang или GCC ?.....	39
Инфраструктура GoLang.....	40
Команды go.....	40
Переменные окружение.....	42
Переменная окружения GOPATH.....	46
Переменная окружения GOTOOLDIR.....	47
Переменная окружения GOARCH и GOOS.....	47
Платформы, переносимость и кросс-компиляция.....	48
Стиль кодирования (автоформатирование — fmt).....	50
Сборка приложений (build).....	52
Сценарии на языке Go (run).....	52
Загрузка проектов из сети (get).....	53
Репозиторные системы.....	53
Установка проектов.....	55
Утилиты GoLang (tool).....	56
Утилиты компиляции.....	58
Связь с кодом C (пакет Cgo).....	60
Сторонний и дополнительный инструментарий.....	63
Интерактивный отладчик Delve.....	64
Источники информации.....	66
Неформально о синтаксисе Go.....	67
Типы данных.....	71
Переменные.....	72
Объявление переменных.....	73
Повторные декларации и переприсвоения.....	75

Константы.....	76
Агрегаты данных.....	78
Массивы и срезы.....	78
Двумерные массивы и срезы.....	82
Структуры.....	83
Таблицы (хеши).....	84
Динамическое создание переменных.....	86
Конструкторы и составные литералы.....	87
Операции.....	88
Функции.....	94
Вариативные функции.....	97
Стек процедур завершения.....	99
Обобщённые функции.....	100
Функции высших порядков.....	101
Встроенные функции.....	103
Объектно ориентированное программирование.....	104
Методы.....	105
Множество методов.....	106
Встраивание и агрегирование.....	107
Функции как объекты.....	108
Интерфейсы.....	109
Именованние интерфейсов.....	112
Контроль интерфейса.....	112
Обработка ошибочных ситуаций.....	113
Структура пакетов (библиотек) Go.....	115
Функция init.....	119
Импорт для использования побочных эффектов.....	120
Некоторые полезные и интересные стандартные пакеты.....	120
Пакет runtime.....	121
Форматированный ввод-вывод.....	121
Строки и пакет strings.....	122
Строчные литералы.....	130
Большие числа.....	131
Автоматизированное тестирование.....	132
Источники информации.....	132
Структура кода: пакеты, модули, импорт.....	134
Пакеты.....	134
Модули.....	136
Публикация модуля в Интернет.....	142
Источники информации.....	147
Новости от последних версий.....	148
Проверяем версию.....	148
Дженерики.....	148
Источники информации.....	152
Параллельные линии развития, TinyGo.....	153
Источники информации.....	161
Часть 2. Конкурентность и многопроцессорность.....	162
Процессоры в Linux.....	163
Процессоры, ядра и гипертриздинг.....	165
Загадочная нумерация процессоров.....	166
Управление процессорами Linux.....	169
Аффинити маска.....	169
Как происходит диспетчирование в Linux.....	171
Приоритеты nice.....	173

Приоритеты реального времени.....	174
Источники информации.....	176
Параллелизм и многопроцессорность.....	177
Эволюция модели параллелизма.....	177
Параллельные процессы и fork.....	177
Потоки ядра и pthread_t POSIX.....	179
Потоки C++.....	181
Сопрограммы — модель Go.....	184
Параллелизм в Go.....	185
Сопрограммы — как это выглядит.....	186
Возврат значений функцией.....	187
Ретроспектива: сопрограммы в C++.....	188
Каналы.....	189
Функциональные замыкания в сопрограммах.....	193
Примитивы синхронизации.....	194
Конкурентность и параллельность.....	199
Источники информации.....	203
Масштабирование.....	204
Планирование активности сопрограмм.....	204
Испытательный стенд.....	205
Микрокомпьютеры (Single-Board Computers).....	205
Рабочие десктопы.....	208
Серверы промышленного класса.....	209
Масштабирование в реальном мире.....	210
1-я попытка	211
2-й подход к снаряду.....	216
О числе потоков исполнения.....	222
Источники информации.....	225
Часть 3. Некоторые примеры и сравнения.....	226
Осваиваемся в технологии Go.....	227
Консольный ввод/вывод.....	227
Утилита echo.....	230
Итерационное вычисление вещественного корня.....	232
Вычисление числа π	234
Вычисления неограниченно высокой точности.....	236
Случайная последовательность и её моменты.....	237
Порядок итераций для map: сюрприз.....	239
Консольный калькулятор.....	239
Источники информации.....	242
Структуры данных, типы и их методы.....	243
Массивы и срезы.....	243
Многомерные срезы и массивы.....	247
Функции с множественным возвратом.....	247
Строки, руны и UNICODE.....	250
Символы, байты и руны.....	250
Изменение содержимого строк.....	251
Палиндромы.....	252
Регулярные выражения.....	256
Источники информации.....	259
Объектно-ориентированное программирование.....	261
Обсчёт параметров 2D выпуклых многоугольников.....	261
Активные объекты.....	265
Встраивание, агрегирование и наследование.....	268
Верификация банковских карт.....	272

Источники информации.....	274
Элементы функционального программирования.....	275
Функциональные замыкания.....	275
Карринг.....	280
Рекурсия.....	282
Рекурсия с кэшированием.....	283
Чистые функции.....	284
Источники информации.....	285
Сети, Интернет и WEB.....	286
TCP сервер и клиент.....	286
Тривиальный WEB сервер.....	289
Источники информации.....	290
Скоростные и другие сравнения языков.....	291
Алгоритмические задачи для сравнения.....	291
Некоторые известные алгоритмы.....	291
Числа Фибоначчи.....	293
Пузырьковая сортировка.....	295
Ханойская башня.....	299
Решето Эратосфена.....	302
Многопроцессорные параллельные вычисления.....	306
Скорость активации параллельных ветвей.....	306
Гонки.....	310
Защита критических данных.....	313
Неблокирующий ввод с клавиатуры.....	316
Многопроцессорный брутфорс.....	319
Каналы в сопрограмах.....	324
Таймеры.....	325
Тикеры.....	326
Майнинг.....	327
Когда не нужно злоупотреблять многопроцессорностью.....	336
Источники информации.....	339
Разработка через тестирование.....	340
Заключение.....	342
Об авторе.....	343

Предисловие

*Хорошая книга не дарит тебе
откровение, хорошая книга укрепляет
тебя в твоих самостоятельных
догадках.
Андрей Рубанов, «Хлорофилия»*

Этот текст не будет изложением того, как писать программы — это во множестве описано в литературе. И это не будет учебным руководством по языку Go, об этом тоже во множестве написано и издано в последние годы (то что понравилось — я привожу ниже в источниках информации)... хотя мы и затронем частично синтаксические основы нового языка. Здесь будет только изложение того, как добиваться максимальной производительности проекта, используя для этой цели предоставленные возможности аппаратурой («железом»). Для этого придётся углубиться (в 1-й части) в некоторые особенности использования языка Go, начиная с разворачивания среды программирования, и, далее, в те синтаксические особенности, которые понадобятся в последующем рассмотрении.

Предназначение и целевая аудитория

*Авангардная музыка, авангардная
живопись ... идеальный способ быть
музыкантом
и художником, не умея ни играть, ни
рисовать.*

А.Гаррос и А.Евдокимов «Новая жизнь»

Первоначально этот текст начал создаваться в 2012-2013 г.г. на заказ руководства крупной международной софтверной компании GlobalLogic, как учебный курс, ориентированный на программных разработчиков компании, планировавших переориентацию некоторых проектов на Go. Позже планы учебного курса потеряли актуальность, а текст был опубликован под лицензией общественного достояния для свободного доступа.

Но за прошедшее время инструментарий Go активно развивался и изменялся, выложенный текст заметно устарел, кроме того он не был ориентирован на эффективное использование многопроцессорных архитектур, по которым накопилось достаточно много нового материала. Поэтому текст послужил только основой для радикальной его переделки 2022 года.

Этот текст создавался в расчёте на более-мене опытного разработчика программного обеспечения, имеющих за плечами один или несколько завершённых проектов, предпочтительно на C или C++¹. ... или на студента, только вникающего в профессию, которому «одинаково на чём писать» и который подбирает свой любимый инструментарий для будущих побед.

Этот текст никак не может быть использован как систематический учебник или справочник по языку Go — для этого есть формализованные описания, часть из которых перечислены в указанных источниках информации. Точно так же — это не учебник программирования и того, как решать задачи на языке Go. При написании ставилась скромная цель: дать разработчику, в достаточной мере владеющему языками и C и C++ в Linux краткое руководство по адаптации этих своих знаний применительно к языку Go. Поэтому будет постоянно, везде где это возможно, приводиться сравнения кодов и конструкций языков Go с C и C++: этот путь сравнения — самый быстрый быстрый путь освоения Go. Некоторые примеры кода (в обсуждении и в архиве) будут даваться в параллельных вариантах: на C/C++ и Go. Этот метод сравнения с C и C++, будет первым принципом, на котором будет строиться всё изложение. Тем более (в смысле простоты адаптации), что Go является прямым продолжением языковой линии языка C и, в некоторой степени (скорее «от противного»), C++, а у истоков его разработки непосредственно стояли люди из числа первоначальных разработчиков C и операционной системы UNIX: Роб Пайк и Кен Томпсон.

А второй принцип: дать максимально много примеров работающих законченных приложений на Go, использующих наиболее широкий спектр возможностей языка (и дополняющей его экосистемы). На этом основан весь текст: дать максимально много примеров использования конструкций Go в разнообразных ситуациях, даже в ущерб точности и соответствию формальной документации языка. По программному коду самих этих примеров будут, даже временами без каких-либо комментариев, дополнительно рассыпаны всякие мелкие «вкусности» из языка, на которые внимательный практик сразу же обратит внимание. И отметит их себе в копилку...

Второй, и может быть главной, целью этой работы было во всех деталях изучение естественного параллелизма, вводимого синтаксисом Go, а также то, каким образом эти параллельные механизмы проявляются в многопроцессорной архитектуре. Этому, фактически, посвящена вторая половина материала.

... ну и, наконец, что может быть и не очевидно, этот текст, в теперешнем его виде, не про Go вообще, а про эффективное использование многопроцессорных платформ, а язык Go

¹ В самом первоначальном варианте этот конспект, или проект книги, так и назывался: «Go для программистов C и C++» ... он где-то так и гуляет по Интернет с 2014-го года под таким названием.

здесь видится в качестве наилучшей, по мнению автора, альтернативы для такого эффективного использования.

Наконец, последняя часть текста вообще несколько выпадает из общего развития повествования, и посвящена примерам реализации некоторых характерных и известных задач на Go. Предлагаются сравнительные решения их на Go и на других языках программирования (C и C++ главным образом) — такие сравнения позволяют отчётливо проследить различия в идеологических подходах в разрешении аналогичных задач.

Код примеров и замеченные опечатки

*Всегда пишите код так, будто
сопровождать его будет склонный к
насилию психопат, который знает, где
вы живете.
Martin Golding*

Все примеры кодов, обсуждаемые в тексте, содержатся в архиве, прилагаемом к тексту. Все примеры были испробованы и проверены, и могут быть воспроизведены из архива. Все коды (как и весь текст в целом) ориентированы на операционную систему Linux. Язык Go имеет высочайшую степень переносимости и совместимости, но под операционными системами семейства Windows (или других) может потребоваться внести изменения в примеры из архива, как правило незначительные.

Все примеры кода отрабатывались и выверялись на компьютерах 3-х групп производительности и целевого применения. Это показатель потенциала всей инфраструктуры GoLang в смысле масштабирования. Эти 3 группы, на которых отработаны все примеры:

1. от самых малых, «игрушечных» однокристальных образцов SoC (System-on-a-Chip - Система на кристалле)...
2. несколько (4-5-6 экземпляров, в разное время) традиционных десктопов архитектуры x86_64, с различающимися характеристиками по числу процессорных ядер, дистрибутивом Linux, версиями этих дистрибутивов...
3. сервер промышленного класса DELL PowerEdge R420, с 2-мя установленными физическими процессорами Intel Xeon® E5-2470 v2, по 20 ядер каждый, в итоге 40 процессоров.

Рассмотрение переносимости создаваемого программного обеспечения между различными архитектурами настолько актуально и интересно, что мы отдельно вернёмся к нему позже, к концу нашего разбирательства, в отдельной главе посвящённой масштабированию систем.

Примеры программного кода сгруппированы по разделам текста в каталоги, поэтому всегда будет указываться имя каталога в архиве (например, xxx) и имя файла примера кода в этом каталоге (например, zzz.go). Некоторые каталоги могут содержать подкаталоги, тогда указывается и подкаталог для текущего примера (например, xxx/yyy). Большинство каталогов (вида xxx) содержат одноимённые файлы (вспомогательные) вида xxx.hist — в них содержится скопированные с терминала результаты выполнения примера (журнал, протокол работы) в хронологической последовательности развития этого примера, показывающие как этот пример должен выполняться, а в более сложных случаях здесь же могут содержаться команды, показывающие порядок компиляции и сборки примеров архива.

Все примеры неоднократно проверялись и перепроверялись компиляцией и выполнением. В отношении стилистики написания кода примеров у кого-то могут возникнуть некоторые замечания. Но! ... Код писался на протяжении нескольких лет (как минимум 7-8), на протяжении которых шла работа с текстом, когда активно, когда не очень... За это время менялись даже стандарты языков, появлялись конструкции которых не было раньше, иногда примеры переписывались, иногда нет. Во-вторых, примеры написаны на разных языках: C, C++, Python, Go... Когда, в своей профессиональной работе, в периоды когда пишешь на C++ — то теряешь беглость Python, когда в другой период пишешь на POSIX API Linux — то теряешь выразительность C++, а о беглости в Go приходится забыть... Но весь код перед вами, и вы можете самостоятельно украсить стилистику на свой вкус.

Примеры листингов кода, по мнению автора, — это самая ценная часть проделанной работы. Потому что это инструмент для подтверждения сказанного и отправная точка для

дальнейших самостоятельных экспериментов. Весь остальной объем книги — это просто пространный комментарий к этим примерам и листингам кода. Архив примеров листингов кода выложен для свободного скачивания и использования по ссылке <https://zip.bhv.ru/9785977517416.zip>.

Конечно, и при самой тщательной выверке и вычитке, не исключены недосмотры и опечатки в объёмном тексте, могут проскочить мало внятные стилистические обороты и подобное. Да и в процессе вёрстки текста может быть привнесено много любопытного... О замеченных таких дефектах я прошу сообщать по электронной почте olej.tsil@gmail.com, и я был бы признателен за любые указанные недостатки рукописи, замеченные ошибки, или высказанные пожелания по её доработке.

Примечание относительно версий программных продуктов: Поскольку текст книги накапливался и обновлялся на протяжении, как минимум, 8 лет, то версии устанавливаемых и тестируемых программных продуктов, ясное дело, не один раз сменились. Я показываю во многих местах **инсталляцию** программных инструментов (пакетов), которые мы используем. После первичной инсталляции пакета Linux а). его нельзя заново проинсталлировать через некоторое время для демонстрации и б). версия уже установленных программных пакетов будет обновляться вместе с обновлением программного обеспечения всего дистрибутива Linux. Поэтому было принято решение **не пытаться обновлять** листинги инсталляций по тексту — цифры могут отличаться, и у вас быть совсем другие версии, но инсталляции и версии мы показываем только для того, чтобы у вас был путеводитель по созданию своего рабочего окружения.

Замечание о версиях

*Если на клетке слона прочтешь надпись:
буйвол, — не верь глазам своим.
«Плоды раздумья» Козьма Прутков*

Текст книги, начиная от первоначального конспекта для узкого круга изучающих, расширялся, накапливался и корректировался на протяжении достаточно многих лет. За это время сменились не одна версия любого из упоминаемых инструментов. Первоначально казалось хорошей идеей, в примерах выполняемых действий и команд, отслеживать изменения версий инструментария. Но позже практика показала что это дурная идея: невозможно по каждому «чиху» версии показать команды инсталляции инструмента (просто потому что он уже инсталлирован в предыдущей версии). Более того, попытка «угнаться» за версиями в таком обстоятельном объёме текста только приводит к путанице и ошибкам, но ничего не добавляет содержательно.

Поэтому, если вы видите в тексте цифры версий, которые, пусть даже значительно, отличаются от того что вы видите на экране своего монитора — это не должно смущать, логика поведения остаётся неизменной в зависимости от конкретики цифр.

Но там же, где поведение GoLang существенно поменялось по логике своего поведения, вследствие естественного развития и модификации инструмента, по возможности, сделаны попытки везде исправить текст в соответствии с актуальным положением вещей.

Ещё раз напомним: текст этой книги накапливался и писался более 10 лет. Некоторые описываемые аспекты, команды, опции... — могли за это время устареть и поменаться. Даже при самом тщательном подходе отследить это во всех местах и обновить — это практически невозможно! Автор был бы признателен если бы читатели сообщали ему о всех замеченных несоответствиях.

Чего нет в этой книге

*Никто не обнимет необъятного!
Козьма Прутков*

Вместе с языком Go бурно развивается номенклатура пакетов Go от разнообразных сторонних разработчиков — окружение языка Go. Если для языка C библиотечная инфраструктура (стандарт POSIX) стабилизировалась десятилетиями, как и стандарт C++ (от Б.Страуструпа), то для Go номенклатура доступных целевых пакетов меняется, расширяется от месяца к месяцу. Она настолько широка и разнообразна (каждому разработчику интересно что-то своё из областей деятельности), что пытаться охватить это разнообразие бессмысленно ... если мы углубимся в область использования А, то почему мы обойдём вниманием область В? Поэтому информация о целевых пакетах Go и сторонних инструментах полностью и сознательно исключена из рассмотрения². Текущее состояние пакетов вы можете найти на <https://pkg.go.dev/> (Discover Packages).

По мнению автора, в языке Go есть 2 предмета, выделяющих его, в положительную сторону, на фоне многих других языков. А именно:

1. Конкурентное программирование и возможности параллельного многопроцессорного выполнения;
2. Расширенные возможности обработки мультязычной текстовой информации, символьных строк.

Именно на эти двух аспектах и будут сделаны особые акценты в дальнейшем тексте.

Совершенно ожидаемо, что в ближайшее время одной из самых активно расширяемой областью проработки инструментов Go станут сферы создания графических интерфейсов (GUI) для приложений на Go, и очень близкая к ней область создания библиотек, связанных с разработкой игр. В этих областях уже и на сегодня предлагаются десятки разнообразных инструментов. Но они сознательно не рассматриваются в книге, поскольку ... это совершенно отдельный предмет, требующий очень объёмного рассмотрения, и оно увело бы нас далеко от наших намерений. Но в списке источников информации (в конце этой части) дан очень обстоятельный перечень инструментов и из этих областей, со ссылками для их детального изучения.

² Некоторые из самых употребимых (и самых старых, устоявшихся) пакетов мы будем использовать в своих примерах кода на Go без всяких дополнительных объяснений. Вы можете также использовать их: as is.

Соглашения и выделения, принятые в тексте

Для большей ясности при чтении текста, он размечен шрифтами по функциональной принадлежности выделяемого фрагмента. Применена широко используемая, устоявшаяся в других публикациях и интуитивно ясная разметка:

- Отдельные ключевые понятия и термины в тексте, на которые нужно обратить особое внимание, будут выделены курсивом.
- Тексты программных листингов, вывод в ответ на консольные команды пользователя размечен **моноширинным шрифтом**.
- Так же **моноширинным шрифтом** (прямо в тексте) могут быть выделены: имена команд, программ, файлов ... т.е. всех терминов, которые должны оставаться неизменяемыми, например: `/proc`, `clang`, `./myprog`, ...
- Ввод пользователя в консольных командах (сами команды, или ответы пользователя в диалоге ... то что мы набираем на клавиатуре), кроме того что это листинг, выделены **жирным моноширинным шрифтом**, чтобы отличать от ответного вывода программ (или системы) в диалогах (который набран просто **моноширинным шрифтом**).
- В показанных многочисленных листингах ввода-вывода терминала (как это обычно и принято в публикациях по UNIX/Linux) команды от имени ординарного пользователя и от имени администратора `root` будут различаться по предшествующему значку приглашения: `$` — это ординарный пользователь, `#` — это `root` (это достаточно традиционное соглашение, но не следует забывать, что оно настраиваемое — в вашей конкретной системе виды приглашений могут быть другими).
- Интернет-ссылки (URL) по тексту, в том числе и адреса электронной почты, выделяются **жирным написанием** (в приводимых библиографиях такое выделение не делается).

Напоминание

Язык Go — относительно новый язык со своей экосистемой (GoLang). Он до сих пор очень динамично развивается: дополняется, уточняется... и изменяется. С регулярностью в несколько месяцев выходит очередной новый релиз системы. На дату написания этого текста (начало 2024 года) самая последняя стабильная версия (если говорить о состоянии стандартных пакетов: <https://pkg.go.dev/std>):

Version: go 1.21.6 Latest Published: Jan 9, 2024, License: BSD-3-Clause

В новых версиях появляется много нового и интересного, чего не хватало в предыдущих. В частности в версии 1.21 (Релиз Go 1.21: <https://golang-blog.blogspot.com/>):

Релиз Go, версия 1.21, выходит через шесть месяцев после Go 1.20. Большинство его изменений касаются реализации цепочки инструментов, среды выполнения и библиотек. Как всегда, выпуск поддерживает обещание совместимости Go 1; на самом деле Go 1.21 улучшает это обещание.

Следите за обновлениями системы GoLang!

Источники информации

- [1] Цукалос М., Golang для профи: работа с сетью, многопоточность, структуры данных и машинное обучение с Go, изд. «Питер» Спб, 2020 г., 720 стр. — <https://rutracker.org/forum/viewtopic.php?t=5929666>
- [2] Титмус, М. А., Облачный GO: создание надежных сервисов в ненадежных окружениях, изд. «ДМК ПРЕСС», 2022 г., 417 стр. — <https://mdk-arbat.ru/book/6241942#instock> , <https://rutracker.org/forum/viewtopic.php?t=6112741>
- [3] Алан А. А. Донован, Брайан У. Керниган, Язык программирования Go, изд. «Вильямс», 2016 г., 432 стр. — <https://rutracker.org/forum/viewtopic.php?t=5222397>
- [4] Олег Цилюрик, Конспект: язык Go в Linux, 2014 г. — <http://flibusta.is/b/510170>
- [5] Блог о языке программирования Go — <https://golang-blog.blogspot.com/>
- [6] Уроки для изучения Golang — <https://golangify.com/>
- [7] The Go Programming Language Specification, Version of December 15, 2022 — <https://go.dev/ref/spec>
- [8] Standard library, Version: go1.21 — <https://pkg.go.dev/std>
- [9] Подборка проектов для разработки GUI на Go, 6 апреля 2022 — <https://dzen.ru/media/golang/podborka-proektov-dlia-razrabotki-gui-na-go-624d38f4828df173e1426ead>

Часть 1. Инструментарий языка Go

В этой части мы, предваряя основной предмет интереса, рассматриваем общие сведения о языке программирования Go, его позицию в семье многочисленных (уже) языков программирования, а также вопросы разворачивания программной инфраструктуры Go для дальнейшей нашей работы. Помимо этого, в этой части даётся не формализованное описание синтаксиса Go, достаточное для понимания всего дальнейшего изложения.

Если же вы уже «на дружеской ноге» с языком Go и имеете развёрнутую систему для работы с Go, то вы можете безболезненно пропустить эту часть и перейти к следующей.

Предыстория Go

*Существует великое множество языков
программирования, которые не
уступают или даже превосходят Си по
красоте и удобству.
Тем не менее ими никто не пользуется.
Денис Ритчи*

«Отцы-основатели» о целях и мотивации...

*Эх, знали бы бесстрашные молодые
оторвы, ужасающие своими подвигами
сеть, что активизм во все эпохи разный,
а вот старость, иконы и коты –
одинаковые во все времена...
Виктор Пелевин «iPhuck 10»*

Go — новый компилируемый язык программирования с естественными параллелизмами, разработанный компанией Google. Первоначальная разработка Go началась в сентябре 2007 года, и его непосредственным проектированием занимались Роберт Гризмер, Роб Пайк и Кен Томпсон, то есть лица непосредственно стоявшие 40 лет назад у истоков языка C и операционной системы UNIX. Официально язык был представлен в ноябре 2009 года.

Некоторыми из заявленных (и достигнутых, как увидим далее) целей разработки были:

- Создать современную альтернативу языку C (которому более 40 лет) и одновременно избежать громоздкости и тяжеловесности языка C++;
- Естественным образом отобразить в языке возможность параллельных вычислений в многопроцессорных (SMP, многоядерных) системах;
- Обеспечить высокую переносимость между операционными системами. На данный момент поддержка Go осуществляется для операционных систем Linux, FreeBSD, OpenBSD, Mac OS X, Windows. Как мы увидим вкратце, Go позволяет создавать в Linux вообще автономные приложения, вообще не использующие интерфейс системных вызовов через стандартную библиотеку C (libc.so);

Примечание: Хотя Go и реализован практически во всех операционных системах, автор ничего не может сказать о состоянии дел и особенностях Go в Mac OS X или Windows. Всё наше дальнейшее рассмотрение будет проводиться только в операционной системе Linux.

- Обеспечить высокую переносимость между аппаратной архитектурой самых различных процессорных платформ: i386, amd64, ARM, MIPS, PPC, ...

Помимо этих основных целей, явно формулировался целый ряд дополнительных, попутных, достижение которых мы будем обсуждать по ходу дальнейшего обсуждения...

Ещё одна особенность проекта, отличающего его от многих (если не всех) других языков программирования — прекрасная документированность проекта: с Go можно садиться работать не располагая никакими иными сторонними источниками справочной информации, кроме оригинальных страниц документации, размещённой на самом сайте проекта GoLang (ссылки документации указаны в конце этого раздела).

Применимость: беглый взгляд

Динамику развития Go и огромный задел доступных на сегодня библиотечных пакетов Go можно оценить, например, посмотрев в стандартный репозиторий любого дистрибутива Linux, например, (только для одной **текущей** процессорной архитектуры

X86_64) дистрибутива Fedora (далеко не самый быстро наполняемый новинками дистрибутив):

```
$ dnf list golang*
```

```
...
Имеющиеся пакеты
golang.x86_64                                1.16.13-1.fc35                updates
golang-antlr4-runtime-devel.noarch          4.9.3-1.fc35                  updates
golang-ariga-atlas.x86_64                   0.3.3-1.fc35                  updates
golang-ariga-atlas-devel.noarch              0.3.3-1.fc35                  updates
golang-bazil-fuse-devel.noarch               0-0.17.20200722gitfb710f7.fc35 fedora
golang-bin.x86_64                           1.16.13-1.fc35                updates
...
```

По количеству:

```
$ dnf list golang* | wc -l
2190
```

Это демонстрация, кроме прочего, и возможность в поддерживаемых разных аппаратных архитектурах и простота кросс-компиляции под них, что мы детально рассмотрим вскоре. Но список впечатляющий!

Несмотря на относительную молодость Go, его уже избрали в качестве инструментария авторы многих открытых публичных проектов. Здесь собран для ознакомления указатель на **несколько сот** реализаций на Go, начиная с простеньких утилит и до комплексных развиваемых и долгосрочно поддерживаемых проектов: <https://code.google.com/p/go-wiki/wiki/Projects>. Там же показаны автономные инструментальные проекты Go-инфраструктуры (от независимых разработчиков), которые мы даже не сможем упомянуть в связи с ограничениями объёмов.

На Go реализован такой уже широчайше известный и популярный проект как Docker. Как пример последнего времени: анонсирован крупнейший проект Syncthing — открытое кросс-платформенное приложение (Linux, Mac OS X, Windows, FreeBSD и Solaris, Android), строящееся по модели клиент-сервер и предназначенное для синхронизации файлов между двумя участниками (point to point). Проект реализуется на языке Go.

В 2009 Go был признан языком года по версии организации ТЮВЕ.

Go, C, C++, Java, Python, Rust, Oberon ...

*Ведь традиция, как ты понимаешь, это
не сохранение тепла, а поддержание
огня.*

Павел Крусанов «Мёртвый язык»

Вопрос вот в чём: при таком множестве языков программирования (больше сотни, несколько сот), созданных за 50-60 лет, зачем нужен ещё один язык Go. Попробуем, очень поверхностно, не акцентируясь и не теряя особенно на то своё время, взглянуть на состояние дел...

Go, так же, как C и C++ является чисто компилирующим языком: в результате компиляции и связывания (из некоторого числа отдельных объектных файлов — фрагментов будущего приложения) создаётся единый бинарный исполнимый файл (в Linux это исполнимый ELF-формат), пригодный в дальнейшем для многократного выполнения только аппаратными вычислительными средствами компьютера, без поддержки какой-либо интерпретирующей среды.

Это очень важно отметить, потому как, следуя тенденциям в развитии последних 20-30 лет, подавляющее большинство из многих десятков языковых сред требуют ту или иную интерпретацию периода выполнения — это может быть либо виртуальная языковая машина (Java, Python, Kotlin), либо просто текстуальная интерпретация программного кода (Perl, PHP, Ruby, Lua, Tcl, ...)³. Таким образом Go заполняет некоторую недостаточно

³ Это особенно бросается в глаза если сравнить ситуацию с ранним периодом становления и развития IT

заполненную нишу инструментариев.

Язык Go, непосредственно по разнообразным высказываниям его отцов-основателей, является прямым продолжением линии C, то как они же, эти же авторы, сами и спроектировали бы C — но ... «40 лет спустя», с учётом опыта прошедших десятилетий эксплуатации C. (Это как у А. Дюма: «Три мушкетёра», но только «20 лет спустя».)

В отношении C++ всё заметно сложнее (и об этом будет детальнее ниже), вот цитата из воспоминаний Роба Пайка, одного из основателей языка:

Корни Go основаны на C и, в более широком смысле, на семействе Algol. Кен Томпсон в шутку сказал, что Роб Пайк, Роберт Грейнджер и он сам собрались вместе и решили, что они ненавидят C++. Будь то шутка или нет, Go сильно отличается от C++.

В отношении скоростных показателей (помимо многих других требований, о которых уже сказано выше и о мы ещё поговорим позже) проектировщики ставили 2 скоростных требования:

1. Скорость компиляции аналогичного кода должен быть выше чем у C.
2. Скорость выполнения скомпилированного кода должна если и уступать скорости своего эквивалента на C, то незначительно.

В отношении скорости компиляции, синтаксис Go спроектирован так (на уровне лексического анализа), что позволяет осуществлять очень быструю компиляцию, и, как следствие, даже непосредственно запускать **исходный код** Go на выполнение, на манер скриптов — того, как это происходит в интерпретирующих языках, командой: `go run xxx.go`. Но это никак не интерпретация, а некая скрытая быстрая компиляция с последующим выполнением (то что называют JIT, Just-In-Time, компиляция «точно в нужное время»). К оценкам скорости компиляции Go и что это даёт в итоге — мы вернёмся позже...

Эксперименты относительно скорости **выполнения** программ ... дело, в общем, неблагодарное, потому что в зависимости от типа задачи (кто и на что «заточен») на одних задачах вы будете получать соотношение «больше чем...», а на других, для равно тех же сравниваемых языков, «меньше чем...». Можно только очень грубо говорить о разнице в порядках скорости (в 10 раз, в 100 раз, ...). Мы вернёмся к подобным сравнениям детально позже, ближе к концу книги, когда будет понятно «что к чему», но пока сделаем хотя бы поверхностные намётки. И для сравнения скорости эквивалентных приложений для Go соберём простейшее приложение (каталог `compare/fibo` архива) на нескольких языках программирования, реализующее хорошо известный рекурсивный алгоритм вычисления чисел Фибоначчи⁴, выбрав его как алгоритм с чрезвычайно высокой степенью роста трудоёмкости от порядка задачи, $O(N)$:

fibo c.c

```
#include <stdlib.h>
#include <stdio.h>

unsigned long fib(int n) {
    return n < 2 ? 1 : fib(n - 1) + fib(n - 2);
}

int main(int argc, char **argv) {
    unsigned num = atoi(argv[1]);
    printf("%ld\n", fib(num));
    return 0;
}
```

технологий (60-е, 70-е, ...), когда **практически все** языки разработки были чисто компилирующими: Algol, FORTRAN, COBOL, Pascal.

4 Такой рекурсивный алгоритм реализации «в лоб» - самый неэффективный из многих известных алгоритмов вычисления чисел Фибоначчи. Он легко трансформируется в более эффективные реализации. Но именно в такой рекурсивной форме он достаточно часто используется в оценках производительности, потому что позволяет покрыть, варьируя N, диапазон временных задержек протяжённостью во многие порядки.

fibo cc.cc

```
#include <iostream>
#include <cstdlib>

using namespace std;

unsigned long fib(int n) {
    return n < 2 ? 1 : fib(n - 1) + fib(n - 2);
}

int main(int argc, char **argv) {
    unsigned num = atoi(argv[1]);
    cout << fib(num) << endl;
    return 0;
}
```

Поскольку показатели любой языковой системы могут радикально зависеть от версии её реализации, то для каждого инструмента будем фиксировать и его версию (у вас, или по прошествии некоторого времени, это могут быть совсем другие значения).

Коды C/C++ компилируем GCC с **максимально** ему доступным уровнем оптимизации:

\$ gcc --version

gcc (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0

Copyright (C) 2021 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

\$ gcc fibo_c.c -O3 -o fibo_c

\$ ls -l fibo_c

-rwxrwxr-x 1 olej olej 16040 янв 11 19:35 fibo_c

\$ g++ --version

g++ (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0

Copyright (C) 2021 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

\$ g++ fibo_cc.cc -O3 -o fibo_cc

\$ ls -l fibo_cc

-rwxrwxr-x 1 olej olej 16584 янв 11 19:37 fibo_cc

Наконец, если мы захотим тот же C++ код компилировать таким новым, популярным (и модным) компилятором как Clang, то это будет выглядеть так (если мы компонуем код, чтоб сильно не заморачиваться, со стандартной библиотекой GCC libstdc++, а не новой родной Clang библиотекой libc++):

\$ clang++ --version

Ubuntu clang version 14.0.0-1ubuntu1.1

Target: x86_64-pc-linux-gnu

Thread model: posix

InstalledDir: /usr/bin

**\$ clang++ -O3 -I/usr/include/c++/11/ -I/usr/include/x86_64-linux-gnu/c++/11 **
-L /usr/lib/gcc/x86_64-linux-gnu/11 fibo_cc.cc -o fibo_cl

\$ ls -l fibo_cl

-rwxrwxr-x 1 olej olej 16776 янв 13 00:27 fibo_cl

А вот и первая наша программа на Go ... пока без каких-либо комментариев к синтаксису:

fibo go.go

```
package main
```

```
import ("os"; "strconv")
```

```
func fib(n int) int {
    if n < 2 {
        return 1
    } else {
        return fib(n-1) + fib(n-2)
    }
}

func main() {
    n, _ := strconv.Atoi(os.Args[1])
    println(fib(n))
}
```

Код Go компилируем средой GoLang, о развёртывании и использовании которой будет вскоре подробно рассказано:

```
$ go version
go version go1.21.3 linux/amd64
$ go build -o fibo_go fibo_go.go
$ ls -l fibo_go
-rwxrwxr-x 1 olej olej 1467733 янв 11 20:07 fibo_go
```

Далее, создадим такое же приложение, добавим язык Java:

fibo.java

```
public class fibo {
    public static long fib(int n) {
        return n < 2 ? 1 : fib(n - 1) + fib(n - 2);
    }

    public static void main(String[] args) {
        int num = Integer.valueOf(args[0]);
        System.out.println(fib(num));
    }
}
```

Компиляция этого приложения выглядит так (хотя говорить о Java «компиляция» нужно с некоторой осторожностью: код Java компилируется в некоторый промежуточный байт-код условной виртуальной машины, JVM — Java Virtual Machine, который позже, при выполнении, этой же JVM и интерпретируется ... зачастую с использованием той же упоминавшейся техники JIT):

```
$ javac --version
javac 11.0.21
$ javac fibo.java
$ ls -l fibo.class
-rw-rw-r-- 1 olej olej 618 янв 11 20:11 fibo.class
$ file fibo.class
fibo.class: compiled Java class data, version 55.0 (Java SE 11)
```

И исполняющая система (JVM) которая будет использована:

```
$ java --version
openjdk 11.0.21 2023-10-17
OpenJDK Runtime Environment (build 11.0.21+9-post-Ubuntu-0ubuntu122.04)
OpenJDK 64-Bit Server VM (build 11.0.21+9-post-Ubuntu-0ubuntu122.04, mixed mode,
sharing)
```

Далее, для сравнения, возьмём такой новый и модный язык системного программирования как Rust (который очень часто в последнее время упоминают в сравнении, а то и противопоставлении, Go):

fibo.rs.rs

```
use std::env;
21
```

```
fn fibo(x: u64) -> u64 {
    if x < 2 { 1 }
    else { fibo(x - 1) + fibo(x - 2) }
}

fn main() {
    let args: Vec<String> = env::args().collect();
    let num: u64 = args[1].trim().parse().unwrap();
    println!("{}", fibo(num));
}
```

Это приложение компилируем так:

```
$ rustc --version
rustc 1.75.0 (82e1608df 2023-12-21)
$ rustc fibo_rs.rs
$ ls -l fibo_rs
-rwxrwxr-x 1 olej olej 4445088 янв 11 19:11 fibo_rs
```

Ну и, для полноты впечатления, «вишенка на торте» — тот же эквивалентный код на интерпретирующем языке Python (версии 3):

fibo.py

```
#!/usr/bin/python3
import sys

def fib(n):
    if n < 2 : return 1
    else: return fib(n - 1) + fib(n - 2)

n = int(sys.argv[1])
print("{}".format(fib(int(sys.argv[1]))))
```

```
$ python --version
Python 3.10.12
```

И теперь, когда мы подготовили все приложения к выполнению, мы можем посмотреть их сравнительное выполнение (почему и воздерживались от запуска сразу после представления каждого кода):

```
$ time ./fibo_c 45
1836311903
real    0m2,863s
user    0m2,862s
sys     0m0,001s

$ time ./fibo_cc 45
1836311903
real    0m2,983s
user    0m2,981s
sys     0m0,001s

$ time ./fibo_cl 45
1836311903
real    0m5,065s
user    0m5,063s
sys     0m0,000s

$ time ./fibo_go 45
1836311903
real    0m7,905s
user    0m7,897s
```

```
sys      0m0,016s
```

```
$ time java fibo 45
```

```
1836311903
```

```
real    0m5,974s
```

```
user    0m5,974s
```

```
sys      0m0,036s
```

```
$ time ./fibo_rs 45
```

```
1836311903
```

```
real    0m15,917s
```

```
user    0m15,911s
```

```
sys      0m0,000s
```

Это, практически, во всех случаях величины одного порядка, в пределах точности измерения, класса задач (о чём было сказано выше), и того дефолтного уровня оптимизации кода, который установлен в каждом компиляторе и в который мы не вмешивались — разницы в результатах практически нет! На другом классе задач такая разница может нивелироваться, или даже поменяться в обратную сторону...

Ну и, наконец, запуск интерпретирующую версию программы на Python (здесь компиляция не нужна):

```
$ time ./fibo.py 45
```

```
1836311903
```

```
real    7m21,956s
```

```
user    7m15,367s
```

```
sys      0m6,109s
```

На этот раз это в 55 раз медленнее чем на Go и в 152 раз хуже чем на C. Означает ли это что Python хуже C или Go? Ни в коей мере! Это вовсе не нужно расценивать как **общее** сравнение производительности — производительность многограннее и сложнее, и радикально зависит от класса решаемых задач. Это только один частный условный пример! Но и это не главное: язык Python — это язык **быстрой разработки**, время разработки прототипа крупной программы на Python оценивают (по публикациям) в 7-10 раз меньшим (а иногда и до 50), чем такого же приложения на C или C++ (да и на Rust, нужно думать, будет то же самое). На Go оценки времени разработки, как можно предполагать, будут в несколько (немного) раз лучше C/C++.

Вот мы подошли к решению вопроса, заданного ранее: зачем нам ещё новые языки программирования? А затем, что каждый язык — для своей области применения:

- Python — для быстрой разработки приложений, особенно для приложений числовой обработки больших агрегатов данных и искусственного интеллекта;
- Rust — язык системного программирования для создания приложений экстремально высокой надёжности, максимально свободных от внутренних ошибок программиста;
- Go — язык построения прикладных систем высокой нагрузочной способности, особенно в условиях большого числа процессоров, задач естественного параллелизма.

И ... в завершении этой части разговора, относительно общности, преемственности и заимствований именно языка Go, сближающий или наоборот его с другими, популярными на дату его рождения, языками программирования (анalogии всегда во многом приносят ясность):

- Много сходственных черт роднит Go с Python, при диаметрально противоположных, вообще то говоря, их реализационных принципах и сферах применения. Из самых явных и важных заимствований (если их можно считать именно заимствованиями, на уровне идей) — это хеш-таблицы и множественные значения в присвоениях и возвратах из функций. Сами хеш-таблицы вообще являются просто базой всего Python, на которой

он сам и реализован. Включение таких возможностей многократно расширяет функциональность Go. В C++ такие возможности появились далеко не сразу, и реализовались в составе библиотек STL (Standard Template Library). А множественные значения в присвоениях и возвратах фактически позволяют воспроизвести технику кортежей из Python в Go.

- Другая «родственная» ветвь, откуда Go почерпнул идеи — это языки функционального программирования: Common Lisp, Scheme, Ocaml, Haskell... Go не является языком функционального программирования в прямом смысле. Но он рассматривает функции как объекты первого класса (равнозначные традиционным типам данных, как целые или вещественные...). Это означает что язык поддерживает передачу функций в качестве аргументов другим функциям, возврат их как результат из других функций, присваивание их переменным или сохранение в структурах данных. А это уже позволяет определять функции высших порядков, которые могут работать с функциональными значениями: принимать на вход функции или возвращать функции в качестве своего результата. Всего этого достаточно для того, чтобы использовать в коде Go, не являющимся в строгом смысле функциональным языком программирования, при желании, все приёмы и трюки функционального программирования.
- Go предлагает встроенный тип данных для представления текстовых строк — `string`. И предоставляет расширенный API для работы с текстами, вплоть до средств работы с таким мощным инструментарием как регулярные выражения, предоставляемые пакетом `regexp` (что очень интересно, и будет показано в качестве примеров в завершающей части книги). Работа с символьными строками всегда была самым слабым местом в C, и несколько смягчённым в C++ — за счёт использования шаблонов и средств STL (Standard Template Library) в создании типа `std::string`. Но даже и для C++ реализация, например, регулярных выражений искусственна и бедна ... что вообще характерно для большинства языков со статической типизацией. Go в значительной мере преодолевая этот барьер, продолжает в этой сфере традиции, идущие ещё от языка Perl.
- И, конечно, наследники языка Algol — Pascal, Modula-2 и Oberon (автор Никлаус Вирт, нем. Niklaus Emil Wirth) — оказали самое прямое влияние на Go⁵: модули для независимой компиляции и инкапсуляции, корутины и мониторы, поддержка низкоуровневого программирования...

Управление памятью и истоки ненадёжности...

Анализ ошибок возникающих в программных кодах, статистика подводимая не одно десятилетие, показали, что подавляющая доля ошибок возникает в части управления памятью в программе:

1. Разименование недействительных указателей, использование невалидных ссылок... («висячие указатели»). Результатом будет нарушение защиты памяти и аварийный крах приложения.
2. Не возвращение в пул свободной памяти динамически выделяемых участков, не освобождение памяти («утечки памяти»). Результатом будет, в конечном итоге, исчерпание доступной системе памяти.
3. Попытки повторного освобождения ранее уже освобождённых участков памяти (это частный случай п.1, но он имеет отдельную значимость).

За всю 70-летнюю историю развития языков программирования (и вообще инструментария разработки) было выдуманно не более 4-х механизмов (возможно с некоторыми вариациями) для разрешения проблемы контроля за использованием памяти (хронологически они и появлялись именно в перечисляемом порядке):

1. Автоматическое выделение в стеке. Так выделяется память под локальные

⁵ На это прямо указывают авторы языка Go неоднократно в своих интервью.

объекты программной единицы (вызываемой функции). После завершения программной единицы объект в стеке уничтожается. Это универсальный механизм, но он часто слишком узок для задач динамического создания структур данных в программе.

2. Ручное управление. Корректность работы с памятью обеспечивается дуальностью операций создания и уничтожения объектов (переменных). Классическим примером здесь могут быть а). функции `malloc()` и `free()` в языке C и б). операции `new` и `delete` в языке C++.
3. Сборка мусора. На каждый объект программы (ну, возможно, исключая переменные элементарных встроенных типов, как целые или вещественные числа) ведётся счётчик ссылок: а). при создании объекта счётчик равен 1, б). при использовании объекта ещё где-то в программном коде счётчик увеличивается, в). при освобождении объекта счётчик уменьшается. Где-то в глубине исполняющей системы работает невидимая фоновая ветвь, которая периодически, время от времени, уничтожает те динамические объекты в памяти, счётчик ссылок которых обратился в ноль. Именно так организован язык Go.
4. Управление на основе владения: операции присвоения объекта или передачи его в качестве параметра вызова — не создают копии объекта, а перемещают его к новому владельцу. Исходная переменная, имя в программе, после этого становится не валидной, не имеющей значения. Такая модель обеспечения надёжности реализована в языке Rust.

В Go, а мы далее планируем говорить только о Go, вы можете произвольно создавать требуемые объекты в памяти, но распоряжаться их временем жизни вам не только не нужно, но вы просто и не можете этого контролировать. Здесь за вас это делает система сборки мусора.⁶

Источники информации

- [1] Let's Go: объектно-ориентированное программирование на Голанге, 2016 — <https://code.tutsplus.com/ru/tutorials/lets-go-object-oriented-programming-in-golang--cms-26540>
- [2] Олег Цилюрик, Сравнение языков программирования, 2018 — <http://flibusta.is/b/512398>
- [3] Олег Цилюрик, Производительность языков программирования, 2018 — <http://flibusta.is/b/510593>
- [4] Роберт Пайк, Less is exponentially more, 2012 — <https://commandcenter.blogspot.com/2012/06/less-is-exponentially-more.html>
перевод: <https://linux-ru.ru/download/file.php?id=4930>
- [5] Начала STL и контейнеры C++ — <http://flibusta.is/b/510171>
- [6] Регулярные выражения и локализация в коде C/C++ — <http://flibusta.is/b/510176>
- [7] Постулаты Go — <https://habr.com/ru/post/272383/>

⁶ Системы со сборкой мусора не применимы в некоторых областях использования (к счастью, в достаточно узкой сфере). Например, в системах приближающихся по требованиям в «реальному времени», критическим по времени реакции.

Разворачиваем экосистему Go

*Не старайся быстро пахать -
сажай злаки, которые быстро всходят.
Андрей Рубанов «Готовься к войне»*

Для того, чтобы рассматривать использование языка Go, нам прежде нужно развернуть в своей операционной системе всё инструментальное окружение Go, позволяющее создавать разнообразные приложения и манипулировать ними.

Создание среды

Стандартная инсталляция

Как уже упоминалось, Go — это компилируемый язык. Все компиляторы полагаются полностью на собственный код — создаваемый код не является управляемым, то есть для его работы не нужна языковая виртуальная машина. По словам Роба Пайка: *получаемый после компиляции байт-код совершенно автономен*.

Существует, как минимум, два открытых проектов, развивающих инструментарий языка Go. Из них наиболее известны два:

- Основной проект развития GoLang (<https://golang.org/>). Компилятор первоначально был написан на языке C с применением Yacc/Bison в качестве парсера. Начиная примерно с версии 1.5-1.6 сам проект GoLang полностью переписан, и далее развивается сам на собственном языке Go — известный способ «раскрутки», применённый впервые ещё в реализациях языка Pascal.

- Фронтэнд Go в составе общеизвестного мультязычного GNU проекта GCC (<https://gcc.gnu.org/onlinedocs/gccgo/>), это, как понятно, относится только для UNIX-like операционных систем, базирующихся на GCC. Клиентская часть, написанная на C++ с рекурсивным парсером, совмещённым со стандартным бэкэндом GCC. Поддержка Go доступна в GCC начиная с версии 4.6.

Обе из этих реализаций могут быть установлены параллельно и независимо, но их предназначение различается...

Но то, что существуют реализации, вовсе не означает, что они уже присутствуют в вашем установленном дистрибутиве Linux по умолчанию — скорее всего наоборот, вам придётся их искать и устанавливать вручную. Но основываться мы будем в этом, как это правильно в большинстве случаев, на содержимом стандартного репозитория своего дистрибутива.

Основной проект — это GoLang. Для дистрибутива Debian и производных .deb (Ubuntu, Mint, antiX, ... и другие) это выглядит так:

```
$ aptitude search golang
```

```
p   dh-golang          - debhelper add-on for packaging software written in Go
(golang)
p   dh-make-golang     - tool that converts Go packages into Debian package source
p   golang             - Go programming language compiler — metapackage
...
```

```
$ aptitude search golang | wc -l
1377
```

```
$ sudo apt install golang
```

```
Чтение списков пакетов... Готово
```

```
Построение дерева зависимостей
```

```
Чтение информации о состоянии... Готово
```

```
Будут установлены следующие дополнительные пакеты:
```

```
golang-1.13 golang-1.13-doc golang-1.13-go golang-1.13-race-detector-runtime golang-
1.13-src golang-doc golang-go golang-race-detector-runtime golang-src
```

Предлагаемые пакеты:

bzr | brz mercurial subversion

Следующие НОВЫЕ пакеты будут установлены:

golang golang-1.13 golang-1.13-doc golang-1.13-go golang-1.13-race-detector-runtime
golang-1.13-src golang-doc golang-go golang-race-detector-runtime
golang-src

Обновлено 0 пакетов, установлено 10 новых пакетов, для удаления отмечено 0 пакетов, и 0 пакетов не обновлено.

Необходимо скачать 63,5 МВ архивов.

После данной операции объём занятого дискового пространства возрастёт на 329 МВ.

Хотите продолжить? [Д/н] у

...

\$ which go

/usr/bin/go

\$ go version

go version go1.13.8 linux/amd64

Альтернативный проект из Linux комплекта GCC:

\$ aptitude search gccgo

p gccgo - Go compiler, based on the GCC backend
...

\$ aptitude search gccgo | wc -l

239

\$ sudo apt install gccgo

Будут установлены следующие дополнительные пакеты:

gccgo

0 пакетов обновлено, 0 установлено новых, 1 переустановлено, 0 пакетов отмечено для удаления, и 4 пакетов не обновлено.

Необходимо получить 974 Б архивов. После распаковки 0 Б будет занято.

Получить: 1 <http://mirror.mirohost.net/ubuntu/jammy/universe/amd64/gccgo/amd64/4:11.2.0-1ubuntu1> [974 В]

Получено 974 Б в 0с (3.080 Б/с)

(Чтение базы данных ... на данный момент установлено 534994 файла и каталога.)

Подготовка к распаковке .../gccgo_4%3a11.2.0-1ubuntu1_amd64.deb ...

Распаковывается gccgo (4:11.2.0-1ubuntu1) на замену (4:11.2.0-1ubuntu1) ...

Настраивается пакет gccgo (4:11.2.0-1ubuntu1) ...

Обрабатываются триггеры для man-db (2.10.2-1) ...

\$ which gccgo

/usr/bin/gccgo

\$ gccgo --version

gccgo (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0

Copyright (C) 2021 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Версия gccgo у вас будет совпадать с версией основного комплекта «родных» инструментов GCC для вашего Linux:

\$ gcc --version

gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0

Copyright (C) 2021 Free Software Foundation, Inc.

This is free software; see the source for copying conditions. There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

Собственно, этого и достаточно... Обратим, попутно, внимание на рекомендацию GoLang установить сетевые репозиторные системы brz, mercurial, subversion ... ну и git, если они ещё не установлен. Эти системы активно используется GoLang

внутренним образом, для комфортной и эффективной работы со сторонними проектами на Go (для публикаций и импорта пакетов и модулей таких проектов). Правильно будет устанавливать их сразу при новых установках GoLang. Но мы, для последовательного развёртывания темы, рассмотрим их установку и использование отдельно.

Обращаем внимание, это важно, что всё, необходимое для работы с Go, в том или ином варианте из двух, присутствует и устанавливается из стандартных репозиториях пакетов вашего дистрибутива Linux — возможно, ничего другого, из сторонних источников, можно не использовать! Если вас не интересуют тонкие варианты использования среды, вы можете пропустить чтение этого раздела, и сразу перейти к созданию первого простейшего тестового приложения.

Версии среды

Но если мы поинтересуемся немного детальнее составом пакетной системы репозитория Linux, то можем увидеть довольно удивительные вещи... Например, используем дистрибутив:

```
$ lsb_release -a
```

```
No LSB modules are available.  
Distributor ID:    Linuxmint  
Description:      Linux Mint 20.3  
Release: 20.3  
Codename: una
```

```
$ aptitude search golang-1.
```

```
i A golang-1.13                - Go programming language compiler - metapackage  
i A golang-1.13-doc            - Go programming language - documentation  
i A golang-1.13-go             - Go programming language compiler, linker, compiled  
  stdlib  
i A golang-1.13-src            - Go programming language - source files  
p  golang-1.14                 - Компилятор языка программирования Go – метапакет  
p  golang-1.14-doc             - Go programming language - documentation  
p  golang-1.14-go              - Go programming language compiler, linker, compiled  
  stdlib  
p  golang-1.14-src             - Go programming language - source files  
p  golang-1.16                 - Go programming language compiler - metapackage  
p  golang-1.16-doc             - Go programming language - documentation  
p  golang-1.16-go              - Go programming language compiler, linker, compiled  
  stdlib  
p  golang-1.16-src            - Go programming language - source files
```

Здесь мы видим достаточно обычную ситуацию, типичную для многих дистрибутивов Linux: в наборе пакетов присутствует несколько версий пакета GoLang, но по умолчанию устанавливается не самая последняя, более выверенная, более полная по некоторым инструментам... Но мы можем параллельно установить и другую, более свежую версию:

```
$ sudo apt install golang-1.16
```

```
Чтение списков пакетов... Готово
```

```
Построение дерева зависимостей
```

```
Чтение информации о состоянии... Готово
```

```
Будут установлены следующие дополнительные пакеты:
```

```
  golang-1.16-doc golang-1.16-go golang-1.16-src
```

```
Следующие НОВЫЕ пакеты будут установлены:
```

```
  golang-1.16 golang-1.16-doc golang-1.16-go golang-1.16-src
```

```
Обновлено 0 пакетов, установлено 4 новых пакетов, для удаления отмечено 0 пакетов, и 0 пакетов не обновлено.
```

```
Необходимо скачать 65,7 МБ архивов.
```

```
После данной операции объём занятого дискового пространства возрастёт на 377 МБ.
```

```
Хотите продолжить? [д/н] у
```

```
...
```

Но (пока), несмотря на новую установку:

```
$ go version
```

```
go version go1.13.8 linux/amd64
```

Управление системными альтернативами

Смотрим что произошло...

```
$ ls -l `which go`
lrwxrwxrwx 1 root root 21 апр 16 2020 /usr/bin/go -> ../lib/go-1.13/bin/go

$ ls -ld /lib/go*
lrwxrwxrwx 1 root root 7 апр 16 2020 /lib/go -> go-1.13
drwxr-xr-x 4 root root 4096 янв 19 23:35 /lib/go-1.13
drwxr-xr-x 4 root root 4096 апр 30 15:06 /lib/go-1.16
drwxr-xr-x 2 root root 4096 окт 25 2021 /lib/gold-ld
```

Установлены у нас две версии GoLang (1.13 и 1.16), но ссылка вызова указывает на версию 1.13. Для использования и переключения версий в современном Linux не следует это делать вручную, для этого есть подсистема управления **альтернативами**. Добавляем две новые альтернативы для **команды go** (значение приоритетов: 60, 70 — произвольные, важно их соотношение (кто больше и кто меньше), остальные параметры команд понятны из их записи):

```
$ sudo update-alternatives --install /usr/bin/go go /lib/go-1.13/bin/go 60
update-alternatives: используется /lib/go-1.13/bin/go для предоставления /usr/bin/go
(go) в автоматическом режиме
```

```
$ sudo update-alternatives --install /usr/bin/go go /lib/go-1.16/bin/go 70
update-alternatives: используется /lib/go-1.16/bin/go для предоставления /usr/bin/go
(go) в автоматическом режиме
```

```
$ update-alternatives --list go
/lib/go-1.13/bin/go
/lib/go-1.16/bin/go
```

```
$ go version
go version go1.16.2 linux/amd64
```

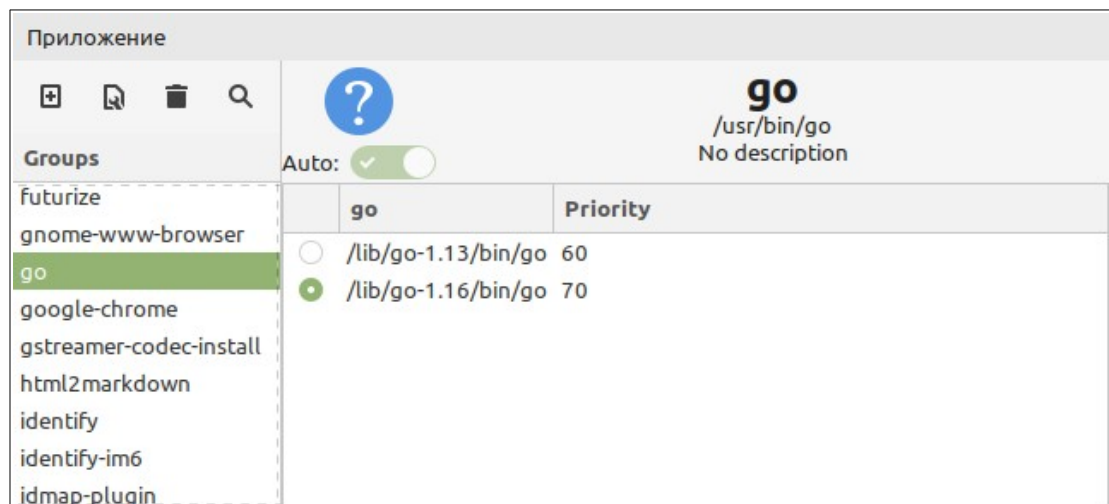


Рис. 1.1. GUI управление альтернативами

Для тех, кому комфортнее работать с графическими GUI инструментами Linux, есть альтернатива утилиты `update-alternatives` — `galternatives` (устанавливается стандартным образом из репозитория). Ней можно как создавать альтернативные записи, так и управлять их использованием, о чём будет сказано дальше.

«Самая последняя» версия

Может оказаться, что и самой свежей версии GoLang недостаточно, часто это бывает при

экспериментировании с новыми возможностями языка, например с дженериками, которые появляются в языке только начиная с версии 1.18 (март-апрель 2022 года). В этом случае нужно ... ну, конечно же, нужно идти на сам сайт проекта (<https://go.dev/>) и устанавливать последнюю (или одну из последних) версию (см. <https://go.dev/doc/install/source>) из исходных кодов GIT репозитория. (GoLang предоставляется для многих операционных систем и разных аппаратных архитектур, поэтому пакетные сборки .deb или .rpm для Linux они не создают, а ставят в Linux из бинарных архивов .tgz — это дело сомнительное.)

```
$ cd $HOME
$ git clone https://go.googlesource.com/go goroot
Клонирование в «goroot»...
remote: Sending approximately 275.40 MiB ...
remote: Counting objects: 43, done
remote: Finding sources: 100% (21/21)
remote: Total 525435 (delta 426356), reused 525433 (delta 426356)
Получение объектов: 100% (525435/525435), 274.88 МБ | 762.00 КиБ/с, готово.
Определение изменений: 100% (426356/426356), готово.
Updating files: 100% (11456/11456), готово.
```

Или, если вы обновляете (что нужно делать время от времени!) уже ранее установленную версию дерева файлов⁷:

```
$ git fetch
remote: Finding sources: 100% (6151/6151)
remote: Total 6151 (delta 3612), reused 6151 (delta 3612)
Получение объектов: 100% (6151/6151), 11.79 МБ | 3.23 МБ/с, готово.
Определение изменений: 100% (3612/3612), завершено с 642 локальными объектами.
Из https://go.googlesource.com/go
    b6a3c0273e..e9b3ff15f4  master          -> origin/master
    998fdce3ae..a95136a88c  release-branch.go1.20 -> origin/release-branch.go1.20
    434af8537e..2540b1436f  release-branch.go1.21 -> origin/release-branch.go1.21
* [новая ветка]           release-branch.go1.22 -> origin/release-branch.go1.22
* [новая метка]           go1.20.13         -> go1.20.13
* [новая метка]           go1.20.11         -> go1.20.11
* [новая метка]           go1.20.12         -> go1.20.12
* [новая метка]           go1.21.4          -> go1.21.4
* [новая метка]           go1.21.5          -> go1.21.5
* [новая метка]           go1.21.6          -> go1.21.6
* [новая метка]           go1.22rc1         -> go1.22rc1
```

```
$ ls -l
итого 72
drwxrwxr-x 2 olej olej 4096 окт 31 18:06 api
drwxrwxr-x 2 olej olej 4096 янв 11 13:17 bin
-rw-rw-r-- 1 olej olej 52 окт 31 18:06 codereview.cfg
-rw-rw-r-- 1 olej olej 1337 окт 31 18:06 CONTRIBUTING.md
drwxrwxr-x 2 olej olej 4096 окт 31 18:06 doc
-rw-rw-r-- 1 olej olej 505 окт 31 18:06 go.env
drwxrwxr-x 3 olej olej 4096 апр 30 2022 lib
-rw-rw-r-- 1 olej olej 1479 апр 30 2022 LICENSE
drwxrwxr-x 8 olej olej 4096 окт 31 18:06 misc
-rw-rw-r-- 1 olej olej 1303 апр 30 2022 PATENTS
drwxrwxr-x 4 olej olej 4096 окт 31 18:25 pkg
-rw-rw-r-- 1 olej olej 1455 янв 5 2023 README.md
-rw-rw-r-- 1 olej olej 419 апр 30 2022 SECURITY.md
drwxrwxr-x 52 olej olej 4096 окт 31 18:06 src
drwxrwxr-x 26 olej olej 12288 окт 31 18:06 test
-rw-rw-r-- 1 olej olej 35 окт 31 18:06 VERSION
olej@R420:~/goroot$ du -hs
```

⁷ Номера версий в листингах и на изображениях могут отличаться друг от друга. Это естественно, так как текст накапливался и редактировался на протяжении 10 лет... Но это обстоятельство не должно смущать — это не отражается на адекватности и актуальности обсуждаемых предметов.

В итоге мы получаем GIT-дерево исходных кодов, для долговременного последующего использования:

```
$ cd goroot
$ ls -l
итого 224
drwxrwxr-x  3 olej olej   4096 апр 30 18:01 api
-rw-rw-r--  1 olej olej  56294 апр 30 18:01 AUTHORS
-rw-rw-r--  1 olej olej    15 апр 30 18:01 codereview.cfg
-rw-rw-r--  1 olej olej   1339 апр 30 18:01 CONTRIBUTING.md
-rw-rw-r--  1 olej olej 111546 апр 30 18:01 CONTRIBUTORS
drwxrwxr-x  2 olej olej   4096 апр 30 18:01 doc
drwxrwxr-x  3 olej olej   4096 апр 30 18:01 lib
-rw-rw-r--  1 olej olej   1479 апр 30 18:01 LICENSE
drwxrwxr-x 11 olej olej   4096 апр 30 18:01 misc
-rw-rw-r--  1 olej olej   1303 апр 30 18:01 PATENTS
-rw-rw-r--  1 olej olej   1455 апр 30 18:01 README.md
-rw-rw-r--  1 olej olej    419 апр 30 18:01 SECURITY.md
drwxrwxr-x 48 olej olej   4096 апр 30 18:01 src
drwxrwxr-x 26 olej olej  12288 апр 30 18:01 test
```

С некоторых пор (начиная с довольно старой версии GoLang, которую не представляется возможным на сегодня восстановить) сборка инструментария Go делается ... с помощью компилятора для самого языка Go — метод **раскрутки**⁸:

- Начиная с некоторой, минимально зрелой, версии реализации языка, сама программная реализация этого языка **переписывается** на самом этом языке;
- А далее новая, очередная, версия N компилируется используя установленную предыдущую версию N-1 этого же языка;
- В GoLang реализована более изощрённая схема раскрутки:
- ... с использованием любой уже установленной версии Go собирается некоторая **эталонная** промежуточная версия (на самом деле даже две таких промежуточных версий: `toolchain1` и `toolchain2` — см. протокол сборки далее), чтобы исключить нюансы тонких синтаксических несоответствий...
- ... и только затем, средствами этой эталонной версии (`toolchain2`) компилируется желаемая собираемая (последняя, как правило) версия.

Поэтому, естественно, прежде чем переходить к сборке новой версии GoLang из исходных кодов, вы будете обязаны установить бинарный пакет какой-то из предыдущих версий, например из стандартного репозитория вашего дистрибутива Linux. Про это было рассказано ранее...

Для сборки (которая весьма сложная) для нас подготовлен скрипт `all.bash`, который должен сделать всё требуемое за нас. Но:

```
$ src/all.bash
all.bash must be run from $GOROOT/src
```

Поэтому, прежде этого, перейдём в сам каталог исходных кодов:

```
$ cd src
$ time ./all.bash
Building Go cmd/dist using /usr/lib/go-1.18. (go1.18.1 linux/amd64)
Building Go toolchain1 using /usr/lib/go-1.18.
Building Go bootstrap cmd/go (go_bootstrap) using Go toolchain1.
Building Go toolchain2 using go_bootstrap and Go toolchain1.
Building Go toolchain3 using go_bootstrap and Go toolchain2.
Building packages and commands for linux/amd64.
```

⁸ Метод раскрутки отчётливо сложился с времён реализации языка Pascal, где он себя успешно показал. После чего он начал использоваться для реализации целого ряда языков программирования.

```

##### Test execution environment.
# GOARCH: amd64
# CPU: Intel(R) Xeon(R) CPU E5-2470 v2 @ 2.40GHz
# GOOS: linux
# OS Version: Linux 5.15.0-91-generic #101-Ubuntu SMP Tue Nov 14 13:30:08 UTC 2023
x86_64

##### Testing packages.
ok      archive/tar      0.217s
ok      archive/zip      0.114s
ok      bufio            0.049s
ok      bytes            0.209s
ok      cmp              0.002s
ok      compress/bzip2   0.067s
ok      compress/flate   0.374s
ok      compress/gzip    2.901s
ok      compress/lzw     0.016s
ok      compress/zlib    0.026s
ok      container/heap   0.009s
ok      container/list   0.004s
ok      container/ring   0.004s
ok      context          0.022s
ok      crypto           0.003s
...
ok      cmd/pack         1.705s
ok      cmd/pprof        2.866s
ok      cmd/trace        0.273s
ok      cmd/vet          4.108s

##### os/user with tag osusergo
ok      os/user          0.008s

##### hash/maphash purego implementation
ok      hash/maphash     0.082s

##### GOMAXPROCS=2 runtime -cpu=1,2,4 -quick
ok      runtime          32.152s

##### Testing without libgcc.
ok      net              0.010s
ok      os/user          0.005s
...

##### API check
ok      cmd/api          33.355s

ALL TESTS PASSED
---
Installed Go for linux/amd64 in /home/olej/goroot
Installed commands in /home/olej/goroot/bin
*** You need to add /home/olej/goroot/bin to your PATH.

real    6m39,683s
user    60m9,799s
sys     10m47,757s

```

Значительная часть вывода здесь опущена (да и сам вид его существенно меняется от версии к версии), но вы должны видеть в итоге нечто подобное, и логика его понятна: идет последовательная сборка пакетов с хронометражем каждого из них.

Сборка весьма напряжённая: выше показана сборка на профессиональном 40 ядерном (2 процессора Intel Xeon CPU E5-2470 v2) промышленном сервере DELL PowerEdge R420

— до 10 минут, и кулера сервера воют так, что вот-вот готовы взлететь! На типовом десктопном оборудовании сборка будет ещё продолжительнее.

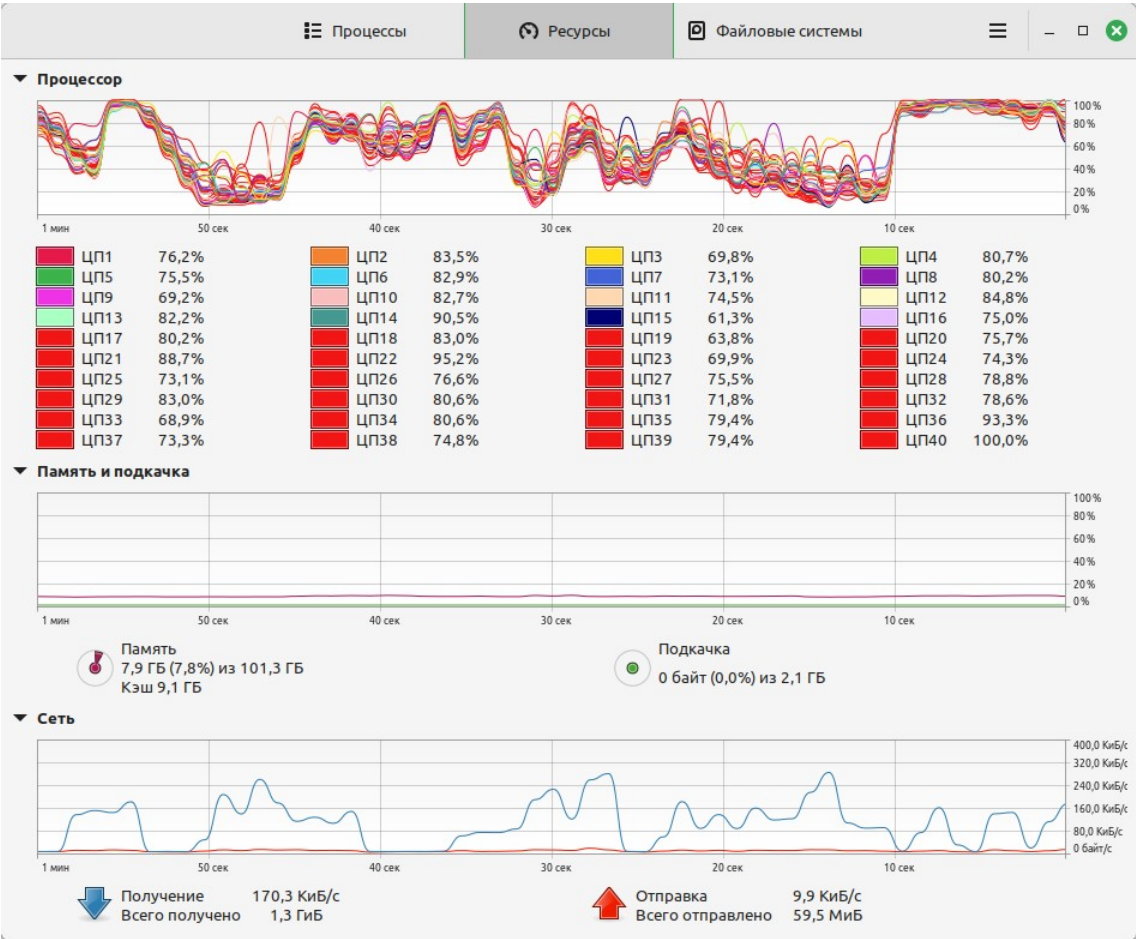


Рис. 1.2. Сборка GoLang из исходных кодов

В итоге (сразу же можем проверить):

```
$ ls -l $HOME/goroot/bin/
итого 14572
-rwxrwxr-x 1 olej olej 12403876 янв 11 13:27 go
-rwxrwxr-x 1 olej olej 2513313 янв 11 13:27 gofmt

$ $HOME/goroot/bin/go version
go version go1.21.3 linux/amd64
```

Смена версий

Наличие нескольких альтернатив может обуславливаться разными целями: проверкой совместимости и переносимости, сравнением новых возможностей, скоростных характеристик и всякое другое. Но для этих целей нам нужна возможность легко и быстро переключаться с одной версии на другую...

В комментариях к сборке, путь полученных выше исполнимых файлов, как один из вариантов, рекомендуют включить в переменную окружения \$PATH. Но ... «мы пойдём другим путём» © — мы включим этот путь в альтернативы:

```
$ sudo update-alternatives --install /usr/bin/go go /home/olej/goroot/bin/go 80

$ sudo update-alternatives --config go
Есть 3 варианта для альтернативы go (предоставляет /usr/bin/go).
```

Выбор	Путь	Приор	Состояние

0	/home/olej/goroot/bin/go	80	автоматический режим
1	/home/olej/goroot/bin/go	80	ручной режим
2	/lib/go-1.13/bin/go	60	ручной режим
* 3	/lib/go-1.18/bin/go	75	ручной режим

Нажмите «enter», чтобы не менять текущий выбор[*], или введите нужное число: 0
 update-alternatives: используется /home/olej/goroot/bin/go для предоставления /usr/bin/go (go) в автоматическом режиме

Введя номер выбранной альтернативы (в показанном выше случае: 0) мы делаем эту альтернативу умалчиваемой. И вот теперь новая собранная версия становится у нас версией по умолчанию:

\$ go version

go version go1.21.3 linux/amd64

\$ update-alternatives --list go

/home/olej/goroot/bin/go

/lib/go-1.13/bin/go

/lib/go-1.18/bin/go

И, для полноты картины и упрощения операций, напомним, что как диагностику, так и управление версиями, можно производить инструментом с графическим инструментом galternatives (для тех кто не любит командную строку):

\$ galternatives

...

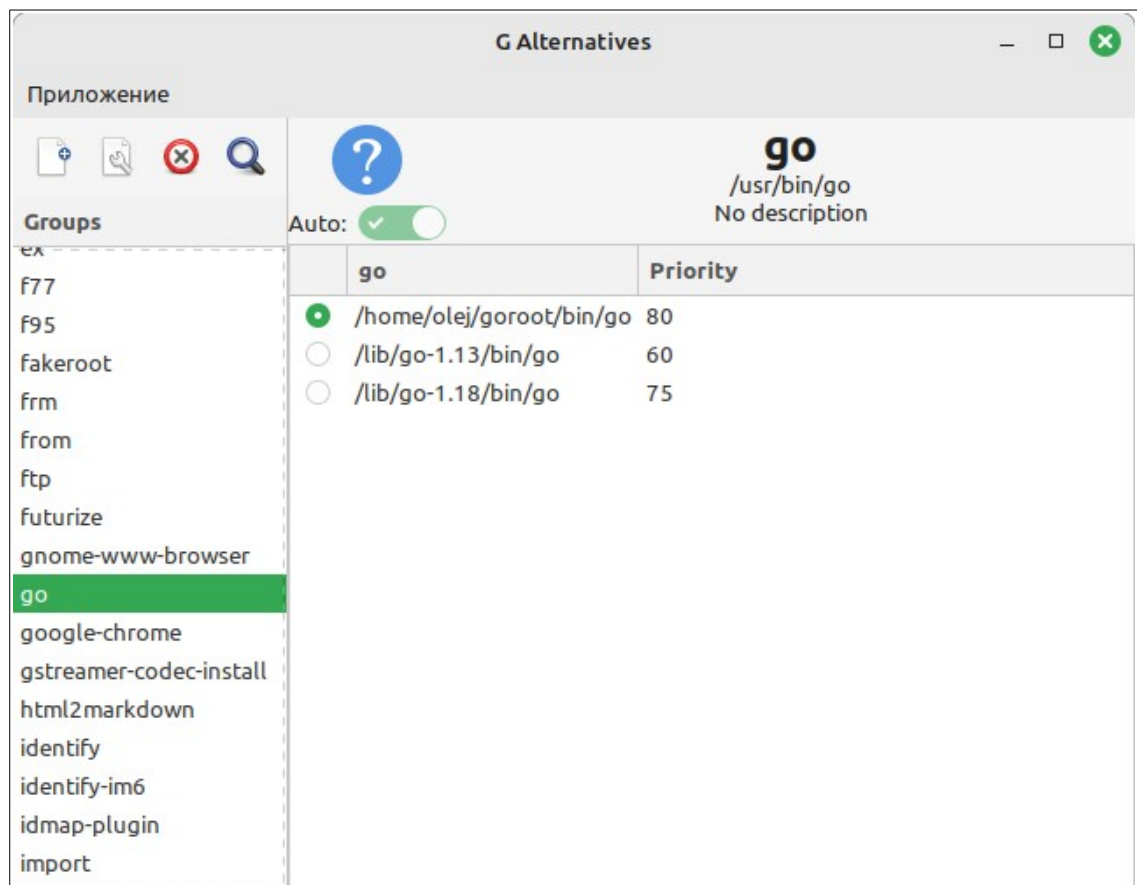


Рис. 1.3. Добавили новую альтернативную версию

Таким путём смена версии среды GoLang занимает считанные секунды.

P.S. В RPM дистрибутивах Linux (Fedora, RedHat, ...) вместо команды update-alternatives у вас будет в наличии другая команда alternatives, но она по возможностям практически не отличается, а детали её синтаксиса и использования можно легко уточнить в справочной системе.

Проверяем: простейшая программа

Первое с чего нужно начать, обустроившись со средой разработки, нам нужно бы проверить работоспособность того что мы имеем в руках. Как мы увидим очень скоро, команда `go` — это не команда компиляции или сборки (как это обычно организовано в старых языках программирования), а это некий менеджер всех широких возможностей, предоставляемых средой GoLang (такое построение вообще характерно для ряда современных сред программирования, таких как Ocaml или Haskell).

Теперь нам предстоит создать нашу первую простейшую программу, взглянуть на общую последовательность технологических действий в создании приложений, и бегло взглянуть на синтаксические подробности Go. У всякого уважающего себя программиста первая программа всегда называется «Hello world!», только мы её сделаем чуть веселее — у нас она будет чуть усложнена, будет диалоговой, и будет уже включать некоторые синтаксические «изюминки»:

hello.go

```
package main

/* первая программа
   демонстрирующая
   синтаксис языка Go */

import (
    "fmt"
    "os"
)

func main() {
    fmt.Println("ты кто будешь?")
    fmt.Printf("> ")
    буфер := make([]byte, 120)
    длина, _ := os.Stdin.Read(буфер) // возвращается 2 значения
    Ω := длина
    ответ := string(буфер[:Ω-1])      // убрали '\n'
    fmt.Printf("какое длинное имя ... целых %d байт\n", Ω)
    fmt.Printf("привет, %s\n", ответ)
}
```

Здесь показаны комментарии, в том привычном виде, как они используются и в C, и как они переключались в Go: многострочный (`/* ... */`) и однострочный (`//`) в стиле C++. И мы уже больше можем не обращаться к вопросу записи комментариев, который в любом языке программирования является вопросом совсем не последним — это гарантия читабельности кодов ваших проектов.

Здесь же сразу бросаются в глаза некоторые особенности:

- Отсутствие ограничителей точка с запятой (;), завершающих в C любой оператор (или некоторые из них)... но эти ограничители в Go могут ставиться, а могут и нет — простановка ограничителя это альтернативная возможность;
- Любой перевод строки (завершение строки) воспринимается как неявно присутствующий, не выписанный явно завершающий ограничитель (;);
- При необходимости (желания) записать несколько операторов в одну строку, это можно сделать используя точку с запятой именно в качестве разделителя, а не ограничителя, так как это понимается в языке Pascal, в отличие от языка C; использование разделителя в таком контексте уже обязательно, а не альтернативно;
- Уже из сказанного может стать понятно (мы ещё к этому вернёмся), что запись кода Go **не свободная**, как, например, в C и C++ — в некоторых случаях мы обязаны ставить перевод на новую строку (Enter) ... это особенно заметно будет в условных операторах `if ...` ;

- Функция `os.Stdin.Read()` возвращает одновременно 2 значения (в манере Python), в общем случае функции могут возвращаться столь угодно много результатов одновременно (списком), сколько нужно ... как мы увидим вскоре, то же самое относится и к оператору присвоения;

- Второе возвращаемое `os.Stdin.Read()` значение (это код ошибки) нас в данном случае не интересует, и мы в позициях не интересующих нас переменных (теряющихся) записываем специальное имя переменной «`_`» (однократный символ подчёркивания);

- Даже в именах переменных (а не только в значениях текстовых констант или комментариях) использованы символы UNICODE национальных алфавитов (буфер, длина, Ω). Кодировка UTF-8 пронизывает всю реализацию Go и мы это будем ещё не раз наблюдать (UTF — Unicode Transformation Format). Это и не удивительно, потому что Роберт Пайк, из числа разработчиков Go, и был ранее основным разработчиком способа кодирования UTF-8⁹;

Подготовим и выполним программу с помощью компилятора GoLang:

```
$ go build hello.go
```

```
$ ls -l hello
```

```
-rwxrwxr-x. 1 Olej Olej 2246096 авг 10 13:09 hello
```

```
$ ./hello
```

```
ты кто будешь?
```

```
> Вася
```

```
какое длинное имя ... целых 9 байт
```

```
привет, Вася
```

Длина русскоязычной строки в 8 байт (4 литеры * 2 байта) + перевод строки (одиночный символ «`\n`» по UNIX-традиции) — нам подтверждает что строчные данные Go также хранит в UTF-8.

Простейшее приложение

Теперь мы можем тут же проверить полученный компилятор (компиляторы) на том простейшем приложении, которое мы только-что написали (уже виденный файл `hello.go`, каталог архива `tools`) ... начнём именно с GCC:

```
$ gccgo -g hello.go -o hello.1
```

```
$ ls -l hello.1
```

```
-rwxrwxr-x 1 olej olej 52464 янв 20 00:12 hello.1
```

```
$ ldd hello.1
```

```
linux-vdso.so.1 (0x00007ffcf38f2000)
```

```
libgo.so.16 => /lib/x86_64-linux-gnu/libgo.so.16 (0x00007fd41551a000)
```

```
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007fd4154ff000)
```

```
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fd41530d000)
```

```
/lib64/ld-linux-x86-64.so.2 (0x00007fd416d85000)
```

```
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007fd4152ea000)
```

```
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007fd41519b000)
```

```
$ file hello.1
```

```
hello.1: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=6d4506b84e6f707a80e1d434c1c1b50c278cfc24, for GNU/Linux 3.2.0, with
debug_info, not stripped
```

Это традиционная для Linux (GCC) сборка, с динамическим связыванием с разделяемыми библиотеками Linux. Изменить компоновку всех, или выборочно, библиотек на статическую (при необходимости) мы можем типовым для GCC образом, не будем на этом останавливаться.

⁹ Для проекта операционной системы Plan 9.

(Выполнение этого, и всех дальше собираемых вариантов приложения hello — идентичное, и в точности соответствует тому, что было показано ранее ... не станем терять на это время — вы сможете в этом убедиться сами.)

Далее, то же самое, но с реализацией GoLang:

```
$ go build -o hello.2 hello.go
```

Или даже так — GoLang позволяет явно указать какой компилятор использовать из числа двух доступных (указываем значение опции: gccgo или gc):

```
$ go build -o hello.2 -compiler gc hello.go
```

Результат (и в том, и в другом случае он полностью идентичен):

```
$ ls -l hello.2
```

```
-rwxrwxr-x 1 olej olej 2038125 янв 20 00:14 hello.2
```

```
$ ldd hello.2
```

```
не является динамическим исполняемым файлом
```

```
$ file hello.2
```

```
hello.2: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, Go
BuildID=sDwLLc5B4-6Q08mhUwYL/wbB6y0Sh_z10-lbB4zbH/DEFipmgTjqmro_2FnANV/
Rmy_4aY1Qu3I6yGoF4Rh, not stripped
```

Различие с предыдущим вариантом (GCC) на лицо: а). размер исполнимого ELF файла (исполнимый формат Linux) в этом варианте в 39 раз больше ... б). потому что по умолчанию GoLang производит статическую сборку (линковку) приложения, включая в него все требуемые приложением библиотеки! В первом случае (GCC) собирается исполнимый файл, использующий все необходимые динамические библиотеки языка C — это приложение, «заточенное» на исполнение исключительно в Linux. Во втором случае (GoLang) собирается автономное приложение, собранное статически, могущее использоваться в любом окружении и на любой аппаратной платформе.

Проект GoLang предполагает также и альтернативное использование компилятора GCC в своём же комплексе инструментальных средств, указав это как опция командной строки:

```
$ go build -o hello.3 -compiler gccgo hello.go
```

```
$ ls -l hello.3
```

```
-rwxrwxr-x 1 olej olej 62544 янв 20 00:20 hello.3
```

```
$ file hello.3
```

```
hello.3: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 3.2.0, with debug_info, not
stripped
```

Как легко видеть, приложение, собранное средой GoLang с указанием использовать компилятор GCC, и собранное автономно компилятором GCC, отличаются очень незначительно (по размеру), очевидно за счёт различных опций компилятора GCC, используемых по умолчанию.

Библиотеки статические и динамические

Утверждается, что начиная с некоторой версии GoLang (похоже, что это версия 1.4, сейчас это трудно проверить) также «научился» собирать приложения с динамическим связыванием с библиотеками. Первоначально, с введением такой возможности, это обеспечивалось дополнительным пакетом go-lang-shared ... но сейчас его нет в составе экосистемы. А для этого предлагается использовать опцию среды -linkshared. Но... :

```
$ go build -o hello.4 -linkshared -compiler gc hello.go
```

```
# command-line-arguments
```

```
/usr/lib/go-1.13/pkg/tool/linux_amd64/link: cannot implicitly include runtime/cgo in a
shared library
```

Ошибка! Но она связана только с тем, что требуется включить в `import` пакет `Cgo` совместимости с языком `C` и его библиотеками и, в частности, с системными API Linux (мы позже отдельно поговорим о возможностях пакета `Cgo`). В исходный файл `hello.go` нужно добавить всего одну строку (я такой файл в архиве назвал `hello.c.go`):

```
...
import "C"
import (
    "fmt"
    "os"
)
...
```

И результат:

```
$ go build -o hello.4 -linkshared -compiler gc hello.c.go
```

```
$ file hello.4
```

```
hello.4: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=d041f04264d23f09a8de0c0868d8f337c34c468c, for GNU/Linux 3.2.0, not
stripped
```

```
$ ldd hello.4
```

```
linux-vdso.so.1 (0x00007fffd1b7c0000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fed577de000)
/lib64/ld-linux-x86-64.so.2 (0x00007fed57d34000)
```

Итоги различных сборок можем наглядно сравнить:

```
$ ls -o hello.{1,2,3,4}
```

```
-rwxrwxr-x 1 olej 66568 фев 22 07:13 hello.1
-rwxrwxr-x 1 olej 1918731 фев 22 07:14 hello.2
-rwxrwxr-x 1 olej 66936 фев 22 07:15 hello.3
-rwxrwxr-x 1 olej 3224296 фев 22 07:17 hello.4
```

```
$ ldd hello.{1,2,3,4}
```

```
hello.1:
```

```
linux-vdso.so.1 (0x00007ffc6c1b1000)
libgo.so.19 => /lib/x86_64-linux-gnu/libgo.so.19 (0x00007f01c0b1a000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f01c0afa000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f01c08d1000)
/lib64/ld-linux-x86-64.so.2 (0x00007f01c2535000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f01c07ea000)
```

```
hello.2:
```

```
не является динамическим исполняемым файлом
```

```
hello.3:
```

```
linux-vdso.so.1 (0x00007fff90392000)
libgo.so.19 => /lib/x86_64-linux-gnu/libgo.so.19 (0x00007f5ce06d4000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f5ce06b4000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f5ce048b000)
/lib64/ld-linux-x86-64.so.2 (0x00007f5ce20ef000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f5ce03a4000)
```

```
hello.4:
```

```
linux-vdso.so.1 (0x00007fffab1e0000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f92bdb73000)
/lib64/ld-linux-x86-64.so.2 (0x00007f92be0c9000)
```

Компиляция или интерпретация

Ну и, в завершение, для быстрой проверки приложения и отладки его вообще можно не компилировать отдельно, а проверить выполнение компиляцией «в лёт» (JIT), фактически очень напоминающее интерпретацию исходного кода, или выполнение кода как это имеет место на скриптовых языках программирования (`bash`, `Perl`, `Python`, `Ruby` ...):

```
$ go run hello.go
ты кто будешь?
> Федот
какое длинное имя ... целых 11 байт
привет, Федот
```

Такой способ непосредственного исполнения (`go run ...`) авторы Go и называют (условно) интерпретацией (что совершенно справедливо по внешнему виду). И именно в этой связи авторами ставилась ещё одна из ключевых задач проектирования Go: формализовать синтаксис нового языка в таком виде, чтобы компиляция из этого синтаксиса происходила существенно быстрее чем из C, и уж тем более быстрее чем из C++. Это позволит использовать небольшие файлы исходного кода Go в качестве скриптов, непосредственно выполняемых как команда в системе.

Выбор: GoLang или GCC ?

Как уже стало понятно, в современных (последних) дистрибутивах Linux, могут быть параллельно установлены два альтернативных компилятора, один из них будет использоваться по умолчанию для компиляции в команде: `go build ...`

Основная среда использования языка Go — это GoLang, и дальше, там где явно не оговорено обратное, в рассмотрении будет использована именно она. GCCGO предназначен использоваться, судя по всему, исключительно в Linux (или других UNIX-подобных), и при требованиях совместимости или совместного использования с другим программным обеспечением GNU, например со специфическими библиотеками. Иногда использование GCCGO может быть оправданным из аргументации применения, в некоторых условиях, хорошо описанных опций оптимизации компилятора, из великого множества их из числа доступных для GCC, особенно для особых аппаратных конфигураций.

Есть тонкая и плохо описанная особенность различия поведения окружений GoLang и GCCGO, относящийся к использованию псевдо-пакета CGo совместимости Go с традиционным кодом C и совместимости с API Linux/POSIX. Она проявляется в том, что приложение использующее CGo нормально собирается GoLang, но завершается с ошибкой при сборке GCCGO (возможно, я просто не знаю способа подключения пакета Cgo к GCCGO):

```
$ gccgo mlp.go -o mlp
mlpar.go:5:9: error: import file 'C' not found
    5 | import "C"
      |         ^
mlpar.go:30:43: error: reference to undefined name 'C'
    30 |     defer func() { ch <- msg{int(n), int64(C.tid())} }()
      |                                   ^
```

Но, с другой стороны, сам GoLang может собирать приложение, компилируя его явно указывая компилятор как собственный родной, так и компилятор GCC. Рассмотрим что при этом происходит (само приложение `mlpar.go` — это приложение, которое мы позже соберём и подробно рассмотрим, пока его существо не имеет значения):

```
$ go build -compiler gc mlp.go

$ go build mlp.go

$ ls -l mlp
-rwxrwxr-x 1 olej olej 2167496 anp  6 00:40 mlp

$ go build -compiler gccgo mlp.go

$ ls -l mlp
-rwxrwxr-x 1 olej olej 90888 anp  6 00:40 mlp
```

Как и утверждалось ранее, при указании использовать дефолтную конфигурацию,

GoLang собирает статически связанное приложение. Но при использовании компилятора GCC собирается динамически связываемое (в момент загрузки) приложение ... это без всякого анализа видно из размеров исполнимых файлов, но, тем не менее уточняем:

```
$ ldd mlpar
linux-vdso.so.1 (0x00007fffd55db5000)
libgo.so.16 => /lib/x86_64-linux-gnu/libgo.so.16 (0x00007f1a9145c000)
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f1a91441000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f1a9124f000)
/lib64/ld-linux-x86-64.so.2 (0x00007f1a92ccc000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f1a9122c000)
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f1a910dd000)
```

И проверим полученное динамически связываемое приложение на его пригодность к выполнению:

```
$ ./mlpar
число процессоров в системе: 40
число ветвей выполнения: 3
[03,7F835FD3E4C0]
[01,7F83273FC700]
[02,7F8335447700]
итоговое время выполнения: 1.000291918s
```

Инфраструктура GoLang

Сама команда `go` проекта GoLang, как об этом вскользь уже было сказано ранее, выполняет роль развитого менеджера проектов на языке Go, используя для выполнения конкретных операций набор собственных **команд**.

Команды `go`

Как уже было сказано, менеджер исходных кодов `go` имеет в арсенале много команд. Одной из таких команд, с которой всё начинается, является команда `help` (как это зачастую и принято в Linux), предоставляющая доступ к обширной и многоуровневой системе справок по командам Go:

```
$ go help
Go is a tool for managing Go source code.
```

Usage:

```
go <command> [arguments]
```

The commands are:

bug	start a bug report
build	compile packages and dependencies
clean	remove object files and cached files
doc	show documentation for package or symbol
env	print Go environment information
fix	update packages to use new APIs
fmt	gofmt (reformat) package sources
generate	generate Go files by processing source
get	download and install packages and dependencies
install	compile and install packages and dependencies
list	list packages or modules
mod	module maintenance
run	compile and run Go program
test	test packages
tool	run specified go tool
version	print Go version
vet	report likely mistakes in packages

Use "go help <command>" for more information about a command.

Additional help topics:

buildmode	build modes
c	calling between Go and C
cache	build and test caching
environment	environment variables
filetype	file types
go.mod	the go.mod file
gopath	GOPATH environment variable
gopath-get	legacy GOPATH go get
goproxy	module proxy protocol
importpath	import path syntax
modules	modules, module versions, and more
module-get	module-aware go get
module-auth	module authentication using go.sum
module-private	module configuration for non-public modules
packages	package lists and patterns
testflag	testing flags
testfunc	testing functions

Use "go help <topic>" for more information about that topic.

Справочная система названа выше многоуровневой, потому что она действительно представляет целую иерархическую древовидную справочную систему, когда на каждом уровне ниже уточняются определённые выше понятия. Выбрав на верхнем уровне, показанном выше, любую интересующую нас тему, дальше мы можем уточняться:

\$ go help mod

Go mod provides access to operations on modules.

Note that support for modules is built into all the go commands, not just 'go mod'. For example, day-to-day adding, removing, upgrading, and downgrading of dependencies should be done using 'go get'. See 'go help modules' for an overview of module functionality.

Usage:

```
go mod <command> [arguments]
```

The commands are:

download	download modules to local cache
edit	edit go.mod from tools or scripts
graph	print module requirement graph
init	initialize new module in current directory
tidy	add missing and remove unused modules
vendor	make vendored copy of dependencies
verify	verify dependencies have expected content
why	explain why packages or modules are needed

Use "go help mod <command>" for more information about a command.

\$ go help mod download

usage: go mod download [-json] [modules]

Download downloads the named modules, which can be module patterns selecting dependencies of the main module or module queries of the form path@version. With no arguments, download applies to all dependencies of the main module.

The go command will automatically download modules as needed during ordinary

execution. The "go mod download" command is useful mainly for pre-filling the local cache or to compute the answers for a Go module proxy.

By default, download reports errors to standard error but is otherwise silent. The -json flag causes download to print a sequence of JSON objects to standard output, describing each downloaded module (or failure), corresponding to this Go struct:

```
type Module struct {
    Path      string // module path
    Version   string // module version
    Error      string // error loading module
    Info      string // absolute path to cached .info file
    GoMod     string // absolute path to cached .mod file
    Zip       string // absolute path to cached .zip file
    Dir       string // absolute path to cached source root directory
    Sum       string // checksum for path, version (as in go.sum)
    GoModSum  string // checksum for go.mod (as in go.sum)
}
```

See 'go help modules' for more about module queries.

В высшей степени продуктивно использовать эту справочную систему не только в качестве онлайн подсказок непосредственно во время работы с GoLang, но и специально изучить предварительно по ней всю эту систему документирования. Потому что, в отличие от любых других источников, и поисков по Интернет, эта справочная система содержит самую свежую и актуальную информацию именно по вашей установленной версии GoLang!

Переменные окружение

Одной из важных составляющих экосистемы GoLang, позволяющих среде слаженно функционировать как единое целое, является набор переменных окружения (оригинальная разбивка строки последней переменной, GOGCCFLAGS, изменена мной для наглядности чтения):

```
$ go env
GO111MODULE=''
GOARCH='amd64'
GOBIN=''
GOCACHE='/home/olej/.cache/go-build'
GOENV='/home/olej/.config/go/env'
GOEXE=''
GOEXPERIMENT=''
GOFLAGS=''
GOHOSTARCH='amd64'
GOHOSTOS='linux'
GOINSECURE=''
GOMODCACHE='/home/olej/go/pkg/mod'
GONOPROXY=''
GONOSUMDB=''
GOOS='linux'
GOPATH='/home/olej/go'
GOPRIVATE=''
GOPROXY='https://proxy.golang.org,direct'
GOROOT='/usr/local/go'
GOSUMDB='sum.golang.org'
GOTMPDIR=''
GOTOOLCHAIN='auto'
GOTOOLDIR='/usr/local/go/pkg/tool/linux_amd64'
GOVCS=''
GOVERSION='go1.22.0'
GCCGO='/usr/bin/gccgo'
```

```

GOAMD64='v1'
AR='ar'
CC='gcc'
CXX='g++'
CGO_ENABLED='1'
GOMOD='/dev/null'
GOWORK=''
CGO_CFLAGS='-O2 -g'
CGO_CPPFLAGS=''
CGO_CXXFLAGS='-O2 -g'
CGO_FFLAGS='-O2 -g'
CGO_LDFLAGS='-O2 -g'
PKG_CONFIG='pkg-config'
GOGCCFLAGS='-fPIC -m64 -pthread -Wl,--no-gc-sections -fmessage-length=0
        -ffile-prefix-map=/tmp/go-build973272879=/tmp/go-build
        -gno-record-gcc-switches'

```

Не нужно путать (из одноимённости их звучания) переменные окружения GoLang и переменные окружения самой операционной системы Linux — это совершенно разные вещи. Убедимся в том, что среди всех системных переменных окружения системы вообще нет имён с контекстом GO:

```

$ env | grep GO
$

```

Но! Обратным порядком вы можете переопределить любую из переменных окружения GoLang (если вы хорошо понимаете последствия того что вы делаете и зачем), определив её в окружении операционной системы:

```

$ export GONOSUMDB='12345'; go env GONOSUMDB
12345

```

```

$ env | grep GONOSUMDB
GONOSUMDB=12345

```

Переменные окружения GoLang «срабатывают» только во время выполнения команд Go, они радикальным образом влияют на поведение и особенности выполнения этих команд. Некоторые переменные окружения настолько сильно влияют на поведение среды, что они заслуживают отдельного рассмотрения. (Другие из этих переменных окружения очевидны из их написания ... или менее значимы.)

Полный список предназначения переменных окружения GoLang мы можем запросить у интерактивной системы справки (вывод объёмный, но его стоит привести полностью, в виду ключевой роли, которую играют эти переменные в функционировании экосистемы GoLang вообще):

```

$ go help environment
The go command and the tools it invokes consult environment variables
for configuration. If an environment variable is unset, the go command
uses a sensible default setting. To see the effective setting of the
variable <NAME>, run 'go env <NAME>'. To change the default setting,
run 'go env -w <NAME>=<VALUE>'. Defaults changed using 'go env -w'
are recorded in a Go environment configuration file stored in the
per-user configuration directory, as reported by os.UserConfigDir.
The location of the configuration file can be changed by setting
the environment variable GOENV, and 'go env GOENV' prints the
effective location, but 'go env -w' cannot change the default location.
See 'go help env' for details.

```

General-purpose environment variables:

```

GCCGO
        The gccgo command to run for 'go build -compiler=gccgo'.
GOARCH

```

The architecture, or processor, for which to compile code.
Examples are amd64, 386, arm, ppc64.

GOBIN
The directory where 'go install' will install a command.

GOCACHE
The directory where the go command will store cached information for reuse in future builds.

GODEBUG
Enable various debugging facilities. See 'go doc runtime' for details.

GOENV
The location of the Go environment configuration file.
Cannot be set using 'go env -w'.

GOFLAGS
A space-separated list of -flag=value settings to apply to go commands by default, when the given flag is known by the current command. Each entry must be a standalone flag. Because the entries are space-separated, flag values must not contain spaces. Flags listed on the command line are applied after this list and therefore override it.

GOOS
The operating system for which to compile code.
Examples are linux, darwin, windows, netbsd.

GOPATH
For more details see: 'go help gopath'.

GOPROXY
URL of Go module proxy. See 'go help modules'.

GOPRIVATE, GONOPROXY, GONOSUMDB
Comma-separated list of glob patterns (in the syntax of Go's path.Match)
of module path prefixes that should always be fetched directly or that should not be compared against the checksum database. See 'go help module-private'.

GOROOT
The root of the go tree.

GOSUMDB
The name of checksum database to use and optionally its public key and
URL. See 'go help module-auth'.

GOTMPDIR
The directory where the go command will write temporary source files, packages, and binaries.

Environment variables for use with cgo:

AR
The command to use to manipulate library archives when building with the gccgo compiler.
The default is 'ar'.

CC
The command to use to compile C code.

CGO_ENABLED
Whether the cgo command is supported. Either 0 or 1.

CGO_CFLAGS
Flags that cgo will pass to the compiler when compiling C code.

CGO_CFLAGS_ALLOW
A regular expression specifying additional flags to allow to appear in #cgo CFLAGS source code directives.
Does not apply to the CGO_CFLAGS environment variable.

CGO_CFLAGS_DISALLOW
A regular expression specifying flags that must be disallowed from appearing in #cgo CFLAGS source code directives.

Does not apply to the CGO_CFLAGS environment variable.

CGO_CPPFLAGS, CGO_CPPFLAGS_ALLOW, CGO_CPPFLAGS_DISALLOW
Like CGO_CFLAGS, CGO_CFLAGS_ALLOW, and CGO_CFLAGS_DISALLOW,
but for the C preprocessor.

CGO_CXXFLAGS, CGO_CXXFLAGS_ALLOW, CGO_CXXFLAGS_DISALLOW
Like CGO_CFLAGS, CGO_CFLAGS_ALLOW, and CGO_CFLAGS_DISALLOW,
but for the C++ compiler.

CGO_FFLAGS, CGO_FFLAGS_ALLOW, CGO_FFLAGS_DISALLOW
Like CGO_CFLAGS, CGO_CFLAGS_ALLOW, and CGO_CFLAGS_DISALLOW,
but for the Fortran compiler.

CGO_LDFLAGS, CGO_LDFLAGS_ALLOW, CGO_LDFLAGS_DISALLOW
Like CGO_CFLAGS, CGO_CFLAGS_ALLOW, and CGO_CFLAGS_DISALLOW,
but for the linker.

CXX
The command to use to compile C++ code.

FC
The command to use to compile Fortran code.

PKG_CONFIG
Path to pkg-config tool.

Architecture-specific environment variables:

GOARM
For GOARCH=arm, the ARM architecture for which to compile.
Valid values are 5, 6, 7.

GO386
For GOARCH=386, the floating point instruction set.
Valid values are 387, sse2.

GOMIPS
For GOARCH=mips{,le}, whether to use floating point instructions.
Valid values are hardfloat (default), softfloat.

GOMIPS64
For GOARCH=mips64{,le}, whether to use floating point instructions.
Valid values are hardfloat (default), softfloat.

GOWASM
For GOARCH=wasm, comma-separated list of experimental WebAssembly
features to use.
Valid values are satconv, signext.

Special-purpose environment variables:

GCCGOTOOLDIR
If set, where to find gccgo tools, such as cgo.
The default is based on how gccgo was configured.

GOROOT_FINAL
The root of the installed Go tree, when it is
installed in a location other than where it is built.
File names in stack traces are rewritten from GOROOT to
GOROOT_FINAL.

GO_EXTLINK_ENABLED
Whether the linker should use external linking mode
when using -linkmode=auto with code that uses cgo.
Set to 0 to disable external linking mode, 1 to enable it.

GIT_ALLOW_PROTOCOL
Defined by Git. A colon-separated list of schemes that are allowed
to be used with git fetch/clone. If set, any scheme not explicitly
mentioned will be considered insecure by 'go get'.
Because the variable is defined by Git, the default value cannot
be set using 'go env -w'.

Additional information available from 'go env' but not read from the environment:

GOEXE

The executable file name suffix (".exe" on Windows, "" on other systems).

GOGCCFLAGS

A space-separated list of arguments supplied to the CC command.

GOHOSTARCH

The architecture (GOARCH) of the Go toolchain binaries.

GOHOSTOS

The operating system (GOOS) of the Go toolchain binaries.

GOMOD

The absolute path to the go.mod of the main module, or the empty string if not using modules.

GOTOOLDIR

The directory where the go tools (compile, cover, doc, etc...) are installed.

Переменная окружения GOPATH

GoLang для многих операций предполагает, что рабочие файлы проектов находятся в структуре файлового поддрева пользователя, корень которого определяет переменная окружения GOPATH. Если GOROOT — это достаточно стабильное местоположение, определяемое инсталляцией инструментария, то GOPATH может переустанавливаться под каждый сменяемый в разработке проект.

```
$ go env GOROOT
```

```
/usr/local/go
```

```
$ go env GOPATH
```

```
/home/olej/go
```

Предполагается, что каталог указанный GOPATH содержит, как минимум, подкаталоги src, pkg и bin. В каталоге src содержатся подкаталоги проектов. Например, пусть наш каталог GOPATH содержит подкаталоги проектов (не важно пока что содержательно представляет каждый из этих проектов):

```
$ go env GOPATH
```

```
/home/olej/go
```

```
$ ls `go env GOPATH`
```

```
bin pkg src
```

```
$ ls -l ~/go/src
```

```
итого 24
```

```
drwxrwxr-x 3 olej olej 4096 янв 20 02:41 github.com
```

```
drwxrwxr-x 2 olej olej 4096 янв 21 01:13 hello
```

```
drwxrwxr-x 3 olej olej 4096 янв 20 03:02 launchpad.net
```

```
drwxrwxr-x 2 olej olej 4096 фев 13 21:20 mchan
```

```
drwxrwxr-x 2 olej olej 4096 фев 13 21:20 mtime
```

```
drwxrwxr-x 2 olej olej 4096 фев 13 21:20 multy
```

Зайдём в каталог любого из проектов:

```
$ pwd
```

```
/home/olej/go/src/mtime
```

```
$ ls -l
```

```
итого 4
```

```
-rw-rw-r-- 1 olej olej 1887 фев 11 16:32 mtime.go
```

Здесь только один исходный файл, представляющий приложение. Собираем его (обращаем внимание, что если мы используем «правильное» место размещения проекта с точки зрения GoLang, то в команде нам не нужно указывать имя файла):

```
$ go build
```

```
$ ls -l
```

```
итого 2236
```

```
-rwxrwxr-x 1 olej olej 2284876 фев 22 01:41 mtime
```

```
-rw-rw-r-- 1 olej olej 1887 фев 11 16:32 mtime.go
```

И точно так же мы очищаем продукт сборки — мы таким образом знакомимся ещё с одной командой (`clean`) менеджера `go`:

```
$ go clean
$ ls -l
итого 4
-rw-rw-r-- 1 olej olej 1887 фев 11 16:32 mtime.go
```

Другая возможность — находясь в любом произвольном каталоге (рабочем) файловой системы выполнить сборку приложения можно просто указав в качестве параметра имя каталога проекта в `GOPATH/src`:

```
$ pwd
/home/olej/2022
$ go build mtime
$ ls -l mtime
-rwxrwxr-x 1 olej olej 2284876 фев 22 01:35 mtime
```

Легко видеть, что приложение проекта при таком использовании собирается именно в текущем рабочем каталоге.

Переменная окружения `GOTOOLDIR`

Переменная окружения `GOTOOLDIR` определяет путь, каталог, куда устанавливаются утилиты среды GoLang:

```
$ go env GOTOOLDIR
/usr/local/go/pkg/tool/linux_amd64
$ ls -o `go env GOTOOLDIR`
итого 86544
-rwxr-xr-x 1 root 2440215 фев 2 20:09 addr2line
-rwxr-xr-x 1 root 3836125 фев 2 20:09 asm
-rwxr-xr-x 1 root 1918355 фев 2 20:09 buildid
-rwxr-xr-x 1 root 3743834 фев 2 20:09 cgo
-rwxr-xr-x 1 root 19361697 фев 2 20:09 compile
-rwxr-xr-x 1 root 2471400 фев 2 20:09 covdata
-rwxr-xr-x 1 root 4267467 фев 2 20:09 cover
-rwxr-xr-x 1 root 3094926 фев 2 20:09 doc
-rwxr-xr-x 1 root 2593284 фев 2 20:09 fix
-rwxr-xr-x 1 root 5298429 фев 2 20:09 link
-rwxr-xr-x 1 root 2384468 фев 2 20:09 nm
-rwxr-xr-x 1 root 3479775 фев 2 20:09 objdump
-rwxr-xr-x 1 root 1649712 фев 2 20:09 pack
-rwxr-xr-x 1 root 11765645 фев 2 20:09 pprof
-rwxr-xr-x 1 root 2035878 фев 2 20:09 test2json
-rwxr-xr-x 1 root 12112935 фев 2 20:09 trace
-rwxr-xr-x 1 root 6131217 фев 2 20:09 vet
```

Или, если вам так удобнее записывать запуск дочернего процесса в командной строке:

```
$ ls -w80 $(go env GOTOOLDIR)
addr2line buildid compile cover fix nm pack test2json vet
asm cgo covdata doc link objdump pprof trace
```

Здесь перечислены дополнительные (второстепенные) инструменты (программы) экосистемы GoLang, некоторые из которых мы рассмотрим отдельно позже.

Переменная окружения `GOARCH` и `GOOS`

Ещё одна из опорных целей, которую ставили разработчики Go — это создать базу для реализации максимально возможной переносимости программного обеспечения, и возможность прозрачной кросс-компиляции и сборки приложений под разнообразные аппаратные платформы и операционные системы, с учётом их форматов, кодировок и

любых других особенностей.

Переменная среды GoLang GOARCH как раз и определяет процессорную архитектуру. А переменная GOOS — операционную систему на этой архитектуре под которую нужно собирать приложение. И прежде, чем рассматривать как это работает, интересно взглянуть на полный текущий список аппаратных и операционных платформ, реализованных на настоящее время. Это можно увидеть здесь — <https://gist.github.com/asukakenji/f15ba7e588ac42795f421b48b8aede63>:

- 17 операционных систем:

All GOOS values:

"aix", "android", "darwin", "dragonfly", "freebsd", "hurd", "illumos", "ios", "js", "linux", "nacl", "netbsd", "openbsd", "plan9", "solaris", "windows", "zos"

- под 24 процессорных платформ:

All GOARCH values:

"386", "amd64", "amd64p32", "arm", "arm64", "arm64be", "armbe", "loong64", "mips", "mips64", "mips64le", "mips64p32", "mips64p32le", "mipsle", "ppc", "ppc64", "ppc64le", "riscv", "riscv64", "s390", "s390x", "sparc", "sparc64", "wasm"

Платформы, переносимость и кросс-компиляция

А теперь мы попытаемся не выходя из своей рабочей системы Linux произвести кросс-сборку всё того же тестового приложения hello, которое мы собирали ранее, но для операционной системы Windows (совершенно несовместимой с UNIX/POSIX/Linux), причём в 2-х ипостасях Windows — 32-бит и 64-бит :

```
$ GOOS=windows GOARCH=386 go build -o hello.32.exe
```

```
$ GOOS=windows GOARCH=amd64 go build -o hello.64.exe
```

```
$ ls -l *.exe
```

```
-rwxrwxr-x 1 olej olej 1901056 янв 20 01:30 hello.32.exe  
-rwxrwxr-x 1 olej olej 2138112 янв 20 01:31 hello.64.exe
```

```
$ file *.exe
```

```
hello.32.exe: PE32 executable (console) Intel 80386 (stripped to external PDB),  
               for MS Windows  
hello.64.exe: PE32+ executable (console) x86-64 (stripped to external PDB),  
               for MS Windows
```

Теперь скопируем (любым способом, проще всего это сделать через локальную сеть) готовые бинарные исполнимые файлы в тестовую систему Windows (Windows 10), и выполним их там (как консольные приложения):

```
X:\Users\Default\Downloads>dir
```

```
Volume in drive X is Boot  
Volume Serial Number is D60A-0DC2
```

```
Directory of X:\Users\Default\Downloads
```

```
06/20/2018  02:00 AM    <DIR>          .  
06/20/2018  02:00 AM    <DIR>          ..  
01/20/2022  03:25 PM             1,054,653 drive-download-20220120T132509Z-001.zip  
01/20/2022  03:30 PM             1,901,056 hello.32.exe  
01/20/2022  03:30 PM             2,138,112 hello.64.exe  
               3 File(s)             5,093,821 bytes  
               2 Dir(s)             368,934,912 bytes free
```

```
X:\Users\Default\Downloads>hello.32.exe
```



```
ты кто будешь?  
> bastard  
какое длинное имя ... целых 9 байт  
привет, bastard
```

```
X:\Users\Default\Downloads>hello.64.exe
```

```
ты кто будешь?  
> window  
какое длинное имя ... целых 9 байт  
привет, window
```

А теперь проанализируем то, что только-что произошло:

- бинарные файлы подготовлены полностью работой Linux в исполнимом Windows-формате PE и готовы к запуску в Windows...

- символьная кодировка (кириллица) исходного файла UTF-8 согласована с консольной кодировкой Windows: CP-866, CP-1251, или что оно там у них ... даже при том, что для представления UNICODE в Windows используется кодировка UTF-16, а не UTF-8 ...

- любопытно, что длина вводимой строки (7 байт) дополнена (9-байт) 2-мя байтами (LF + CR), в то время, как в Linux строки завершаются только 1-м символом (LF).

Ну и, наконец, сборка под совершенно чужеродную операционную систему и другую архитектуру (32-бит вместо 64-бит) заняла у меня не более 10 секунд...

Это совершенно невиданная в более ранних языковых инструментах степень переносимости и мультиплатформенности! Точно с той же лёгкостью мы сможем собирать приложения под архитектуры ARM, MIPS, PPC и любую другую аппаратную платформу...

Ещё интереснее — используем аппаратные платформы одноплатных микрокомпьютеров («наладонников», SoC) на архитектуре ARM. Но компиляцию приложения сделаем в привычном десктопном окружении Linux навсё том же Intel x86_64:

```
$ GOARCH=arm go build -o hello.32.arm hello.go
```

```
$ ls -l *.arm  
-rwxrwxr-x 1 olej olej 1880350 map 23 21:59 hello.32.arm
```

```
$ file hello.32.arm  
hello.32.arm: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically  
linked, Go BuildID=ZfMUX4fipDYH_dqrGeIZ/q3f4F1Z0legZmrfeS1Qu/0zb37t0uutQShZb8-IJl/  
QT5B9gi5cCsDL02SLM5f, not stripped
```

Теперь копированием бинарного исполнимого ELF-файла переносим его (по сети) в среду одноплатного Raspberry Pi :

```
$ inxi -Cxxx
```

```
CPU:      Info: Quad Core model: ARMv7 v7l variant: cortex-a7 bits: 32  
          type: MCP arch: v7l rev: 5  
          features: Use -f option to see features bogomips: 256  
          Speed: 1000 MHz min/max: 600/1000 MHz Core speeds (MHz):  
          1: 1000 2: 1000 3: 1000 4: 1000
```

```
$ lsb_release -a
```

```
No LSB modules are available.  
Distributor ID:    Raspbian  
Description:       Raspbian GNU/Linux 11 (bullseye)  
Release: 11  
Codename: bullseye
```

```
$ uname -a
```

```
Linux raspberrypi 5.10.103-v7+ #1530 SMP Tue Mar 8 13:02:44 GMT 2022 armv7l GNU/Linux
```

```
$ ./hello.32.arm
```

```
ты кто будешь?
> вася
какое длинное имя ... целых 9 байт
привет, вася
```

И точно также, этот же исполнимый файл копируем по сети в совершенно другую архитектуру Orange Pi One (несовместимый аппаратно с Raspberry Pi ... но для полноты ощущений) :

```
$ inxi -Cxxx
```

```
CPU:      Topology: Quad Core model: ARMv7 v7l variant: cortex-a7 bits: 32
          type: MCP arch: v7l rev: 5
          features: Use -f option to see features bogomips: 0
          Speed: 1008 MHz min/max: 480/1008 MHz Core speeds (MHz):
          1: 1008 2: 1008 3: 1008 4: 1008
```

```
$ lsb_release -a
```

```
No LSB modules are available.
Distributor ID:      Ubuntu
Description:         Ubuntu 20.04.4 LTS
Release: 20.04
Codename: focal
```

```
$ uname -a
```

```
Linux orangepihone 5.15.25-sunxi #22.02.1 SMP Sun Feb 27 09:23:25 UTC 2022 armv7l armv7l
armv7l GNU/Linux
```

```
$ ./hello.32.arm
```

```
ты кто будешь?
> вася
какое длинное имя ... целых 9 байт
привет, вася
```

```
$ hostname
```

```
orangepihone
```

Примечание: В ARM-архитектуре, из-за отсутствия динамической идентификации устройств PCI стандарта, совместимость чипсетов определяется явно выписанным в сборке системы деревом устройств (DeviceTree, определенное в DeviceTree Specification), что порождает великое множество вариантов аппаратных семейств, несовместимых с точки зрения сборок образов операционной системы. Это совершенно выходит за рамки наших целей рассмотрения, но потребует, при необходимости, выверять поштучно собранные приложения для каждой отдельной конфигурации.

Стиль кодирования (автоформатирование — fmt)

В принципе, синтаксис Go достаточно свободный (но не полностью свободный), и код может быть записан достаточно произвольным образом (в смысле переносов строк и отступов), хотя это и не совсем свободное кодирование, как, например, в C/C++ (строку нельзя разорвать и перенести в произвольном месте)¹⁰. В конечном счёте, стиль записи программного кода (code style) является делом предпочтений, или, ещё чаще, предписывается корпоративными правилами, принятыми в той или иной компании по разработке программного обеспечения.

Одна из команд менеджера go — команда форматирования (fmt), которая представляет исходные коды на Go в едином стиле, как его понимают сами разработчики Go. Для демонстрации используем показанный выше файл исходного кода hello.go (то что написали вручную), скопировав его предварительно в экземпляр с именем hello.0.go:

```
$ ls -l hello.0.go
```

¹⁰ Из кода любой программы на C и C++ можно вообще полностью убрать все переносы строки, при этом код останется синтаксически верным и будет компилироваться без ошибок. Это и есть то что называется: полностью свободный синтаксис языка программирования. Другая крайность — язык Python, где уровень синтаксической вложенности определяется величиной отступа оператора в строке. Язык Go занимает некоторое промежуточное положение в этом ряду. Является ли абсолютно свободный синтаксис языка некоторым преимуществом?

```
-rw-r--r--. 1 Olej Olej 601 сен 24 21:49 hello.0.go
```

Трансформируем произвольно записанной форме (как нам самим нравится) файл кода Go командой форматирования:

```
$ go fmt hello.0.go
hello.0.go
```

```
$ ls -l hello.0.go
```

```
-rw-r--r--. 1 Olej Olej 557 сен 24 21:50 hello.0.go
```

Как легко видеть, файл программы на Go изменился (смотрим по размеру). Теперь он выглядит несколько по-иному (сравните с показанным ранее исходным вариантом):

hello.0.go

```
package main

/* первая программа
   демонстрирующая
   синтаксис языка Go */

import (
    "fmt"
    "os"
)

func main() {
    fmt.Println("ты кто будешь?")
    fmt.Printf("> ")
    буфер := make([]byte, 120)
    длина, _ := os.Stdin.Read(буфер) // возвращается 2 значения
    Ω := длина

    ответ := string(буфер[:Ω-1])      // убрали '\n'
    fmt.Printf("какое длинное имя ... целых %d байт\n", Ω)
    fmt.Printf("привет, %s\n", ответ)
}
```

Команда `fmt` при групповом форматировании файлов кода Go (каким-то странным образом — скорее всего по содержимому) отбирает и форматирует только те файлы, которые ещё не подвергались форматированию, то что не соответствует GoLang code style:

```
$ go fmt *.go
select.go
close.go
buffer.go
baze10.go
baze1.go
unblock.go
```

Это при том, что в показанном каталоге гораздо больше файлов исходного кода Go чем те, которые подверглись переформатированию:

```
$ ls *.go
baze10.go  buffer.go  closure1.go  closure3.go  once.go  result.go  sleep.go  smp2.go
baze1.go   close.go   closure2.go  mulyt.go     parm.go  select.go  smp1.go  unblock.go
```

Примечание: Строго говоря, команда `go fmt` ничего про звездочку в примере выше не узнает — ее раскрывает командный интерпретатор `bash`. Но нас здесь интересует то, что команда форматирования затрагивает (отбирает) только те файлы кода `*.go`, которые ранее не были затронуты форматированием (в каком-то смысле это отдаленно напоминает то, что делает утилита `make`).

Всё предусмотрено в системе Go! Даже эстетика представления кода Go в едином стиле...

И в завершение обсуждения этой команды, в порядке совета или пожелания: каждый день, завершая работу с кодом Go, и покидая каталог в котором вели эту работу — выполните команду:

```
$ go fmt *.go
...
```

Сборка приложений (*build*)

Большинство команд сборки приложений (*build*) мы уже неоднократно видели ранее по ходу текста, и они, в общем, интуитивно понятны. Работа этой команды сборки существенно изменилась в «новом стиле», при переходе в версии 1.11 к поддержке структуризации кодов GoLang на базе модулей (а не пакетов). Команда *build* — это одна из самых ёмких и востребованных команд при работе с Go. И самая объёмная по опционным возможностям, вот объём (строк) её справочного описания:

```
$ go help build | wc -l
201
```

По логике работе это аналогично, во многом, командам сборки систем *make* или *ninja* и, естественно, сами команды *build* могут быть помещены, в свою очередь, внутрь сценариев сборки *Makefile* (как это и сделано для большей наглядности в примерах кодов в архиве).

Команда *build* — это наша основная команда сборки исполнимых приложений! Мы с ней будем работать постоянно. Некоторые (многие) варианты выполнения *build* уже были показаны ранее, ещё другие будут показаны далее ... часть вариантов можно посмотреть в *Makefile* скриптах сборки в архивах примеров.

Сценарии на языке Go (*run*)

Одной из заявленных целей разработчиков Go была наивысшая скорость компиляции (то есть синтаксис языка, не обременяющий компилятор избыточными затратами производительности). Важной аргументацией этой цели было не только сокращение времени компиляции больших проектов (в конце концов, многократная компиляция больших проектов делается на инструментальных компьютерах самой высокой производительности, и с использованием систем инкрементальной сборки, по типу *make*, *ninja* и других). Ещё одним важным дополнительным аргументом была возможность использования кодовых файлов *.go (особенно небольшого размера) непосредственно в качестве сценариев в операционной системе Linux (в дополнение к широко применяемых в этой системе *bash/dash/zsh*, *Perl*, *Python* и др.). Идея такого сценарного использования кода состоит в том, чтобы а). быстро откомпилировать код без медленной записи результата в дисковую файловую систему и б). затем однократно выполнить откомпилированный результат.¹¹

Вот как выглядит это на примере простейшего приложения (каталог архива *tools* — ещё одна проверка UTF-8 кодировки — китайский язык) выполнение Go кода без предварительной компиляции:

tiny.go

```
package main
import "fmt"

func main() { fmt.Println("Hello, 世界") }

$ time go run tiny.go
Hello, 世界
real    0m0.257s
```

¹¹ Первоначально, в ранних версиях, для этого развивался и предлагался сторонний проект *gorun*, но на сегодня для этих целей достаточно команды GoLang *run*.

```
user      0m0,298s
sys       0m0,114s
```

Обратите внимание: мы, якобы, не делаем явно никакой компиляции, мы просто запускаем файл кода на выполнение, как это было бы, например, в Python, Ruby, или в консольном JavaScript ... да и просто Shell (bash/dash), в конце концов.

Загрузка проектов из сети (get)

Особый интерес, кроме собственно сборки, представляет команда загрузки и установки программных пакетов из сети — `get` (распределённый менеджмент программных проектов). Авторы GoLang расширяют окружение разработки из среды локального компьютера на всю сеть Интернет, когда любой проект лежащий в сети — скачивается единообразными средствами, и продолжает развития локально. В этом смысле Go продолжает и развивает идеи: пакетных систем инсталляции Linux, систем инсталлирования Python (`pip` и другие), распределённых репозиториях GIT ...

Для установки программных проектов Go (пакетов и дополнительных инструментальных пакетов) из сети используется команда `get`. Она использует, на выбор, несколько репозиторийных систем (созданных, популярных и известных задолго до появления Go).

Репозиторийные системы

Команда `get` имеет возможность использовать альтернативно одну из 4-х общеизвестных систем инсталляций и контроля версий, в зависимости от ресурса, откуда скачивается конкретный проект:

- `svn` — Subversion: <http://subversion.apache.org/packages.html>
- `hg` — Mercurial: <http://mercurial.selenic.com/wiki/Download>
- `git` — Git: <http://git-scm.com/downloads>
- `bzr` — Bazaar: <http://wiki.bazaar.canonical.com/Download>

Именно поэтому, при начальной инсталляции среды GoLang, он предлагает сразу же инсталлировать и репозиторийные системы, о чём было сказано ранее.

Естественно, что для использования репозиторийной системы соответствующее приложение менеджера репозиторийной системы должно быть предварительно установлено в вашей системе¹², иначе вы получите сообщения подобные следующим:

```
$ go get code.google.com/p/go.tools/cmd/godoc
```

```
go: missing Mercurial command. See http://golang.org/s/gogetcmd
```

```
package code.google.com/p/go.tools/cmd/godoc: exec: "hg": executable file not found in $PATH
```

Примечание: Сам проект GoLang и его ответвления используют преимущественно систему Mercurial, а проекты, ведущиеся под эгидой Ubuntu — систему Bazaar. Поэтому из 4-х репозиторийных систем лучше иметь установленными их все.

Приложения менеджеров репозиторийных управлениями версий вполне можно установить из показанных выше адресов (URL их проектов)... Но проще и целесообразнее сделать это просто проверив присутствие и, если их ещё нет в системе, просто установив их из пакетной системы своего собственного дистрибутива Linux:

```
$ aptitude search subversion
```

p	hgsubversion	- клиент Subversion как расширение
	Mercurial	
p	python-subversion	- Python bindings for Apache Subversion
v	python2.7-subversion	-

¹² Сами репозиторийные системы и их менеджеры никоим образом не относятся к проекту Go, и созданы задолго до него. Они относятся к нему как совершенно внешние проекты от сторонних разработчиков. Но GoLang изнутри, сам по себе, ориентирован на активное использование репозиторийных (вместо ручного управления кодом), поэтому предлагает их установить, даже если вы их и не используете для других целей.

```
p  subversion                      - Advanced version control system
p  subversion-tools                 - различные инструменты Apache Subversion
```

```
$ sudo apt install subversion
```

```
Чтение списков пакетов... Готово
```

```
Построение дерева зависимостей
```

```
Чтение информации о состоянии... Готово
```

```
Будут установлены следующие дополнительные пакеты:
```

```
libapr1 libaprutil1 libserf-1-1 libsvn1 libutf8proc2
```

```
Предлагаемые пакеты:
```

```
db5.3-util libapache2-mod-svn subversion-tools
```

```
Следующие НОВЫЕ пакеты будут установлены:
```

```
libapr1 libaprutil1 libserf-1-1 libsvn1 libutf8proc2 subversion
```

```
Обновлено 0 пакетов, установлено 6 новых пакетов, для удаления отмечено 0 пакетов, и 25 пакетов не обновлено.
```

```
Необходимо скачать 2.354 кВ архивов.
```

```
После данной операции объём занятого дискового пространства возрастёт на 10,3 МВ.
```

```
Хотите продолжить? [д/н] у
```

```
...
```

```
$ which svn
```

```
/usr/bin/svn
```

```
$ svn --version
```

```
svn, version 1.13.0 (r1867053)
```

```
compiled Mar 24 2020, 12:33:36 on x86_64-pc-linux-gnu
```

```
...
```

```
$ apt list mercurial
```

```
Вывод списка... Готово
```

```
mercurial/focal 5.3.1-1ubuntu1 amd64
```

```
$ sudo apt install mercurial
```

```
Чтение списков пакетов... Готово
```

```
Построение дерева зависимостей
```

```
Чтение информации о состоянии... Готово
```

```
Будут установлены следующие дополнительные пакеты:
```

```
libpython2-stdlib mercurial-common python2 python2-minimal
```

```
Предлагаемые пакеты:
```

```
kdiff3 | kdiff3-qt | kompare | meld | tkcvs | mgdiff qct python-mysqldb python-openssl  
python-pygments wish python2-doc python-tk
```

```
Следующие НОВЫЕ пакеты будут установлены:
```

```
libpython2-stdlib mercurial mercurial-common python2 python2-minimal
```

```
Обновлено 0 пакетов, установлено 5 новых пакетов, для удаления отмечено 0 пакетов, и 25 пакетов не обновлено.
```

```
Необходимо скачать 3.034 кВ архивов.
```

```
После данной операции объём занятого дискового пространства возрастёт на 15,5 МВ.
```

```
...
```

```
$ which hg
```

```
/usr/bin/hg
```

```
$ hg --help
```

```
Распределенная система контроля версий Mercurial
```

```
...
```

```
$ hg --version
```

```
Распределенная SCM Mercurial (версия 5.3.1)
```

```
...
```

```
$ sudo apt install git
```

```
Чтение списков пакетов... Готово
```

```
Построение дерева зависимостей
```

Чтение информации о состоянии... Готово
Будут установлены следующие дополнительные пакеты:
git-man liberror-perl
Предлагаемые пакеты:
git-daemon-run | git-daemon-sysvinit git-doc git-el git-email git-gui gitk gitweb git-cvs git-mediawiki git-svn
Следующие НОВЫЕ пакеты будут установлены:
git git-man liberror-perl
Обновлено 0 пакетов, установлено 3 новых пакетов, для удаления отмечено 0 пакетов, и 25 пакетов не обновлено.
Необходимо скачать 5.465 kB архивов.
После данной операции объем занятого дискового пространства возрастет на 38,4 МВ.
...

```
$ which git
/usr/bin/git
```

```
$ git --version
git version 2.25.1
```

```
$ apt list bzip3
Вывод списка... Готово
bzip3/focal,focal 2.7.0+bzip3.1+bzip3.1 all
```

```
$ sudo apt install bzip3
Чтение списков пакетов... Готово
Построение дерева зависимостей
Чтение информации о состоянии... Готово
Будут установлены следующие дополнительные пакеты:
  bzip3 python3-bzip3 python3-deprecated python3-dulwich python3-fastimport python3-github
python3-gitlab python3-gpg python3-wrapit
Предлагаемые пакеты:
  bzip3-doc python3-bzip3.tests python3-bzip3-dbg python3-kerberos python3-paramiko git-
core python-gitlab-doc
Следующие НОВЫЕ пакеты будут установлены:
  bzip3 bzip3 python3-bzip3 python3-deprecated python3-dulwich python3-fastimport python3-
github python3-gitlab python3-gpg python3-wrapit
Обновлено 0 пакетов, установлено 10 новых пакетов, для удаления отмечено 0 пакетов, и 25
пакетов не обновлено.
Необходимо скачать 2.281 kB архивов.
После данной операции объем занятого дискового пространства возрастет на 14,0 МВ.
...
```

```
$ which bzip3
/usr/bin/bzip3
```

```
$ bzip3 --version
Bzip3 (bzip3) 3.0.2
...
```

Установка проектов

Команды установки¹³ `get` и `install` будут работать только если у вас переменная окружения `GOPATH` установлена на некоторый реально существующий каталог (как было описано раньше), иначе:

```
$ go get github.com/astaxie/beego
package github.com/astaxie/beego: cannot download, $GOPATH not set. For more details
see: go help gopath
```

Если указанный в `GOPATH` каталог не содержит требуемых подкаталогов `src`, `pkg` или `bin`, то те из них, который нужны для установки, будут созданы при выполнении команды `get`.

¹³ Команда `get` претерпела существенные изменения с введением модулей в версии GoLang после 1.1 и теперь дополнена родственной ей новой командой `install`.

Несколько примеров того как в экосистеме GoLang работают команды загрузки проектов Go из сети:

```
$ go env GOPATH
/home/olej/go
```

```
$ go get github.com/astaxie/beego
```

```
$ ls `go env GOPATH`/src/github.com/astaxie/
beego
```

```
$ go get launchpad.net/gorun
```

```
$ ls -l `go env GOPATH`/src/launchpad.net/gorun
итого 44
-rw-rw-r-- 1 olej olej 35147 янв 20 03:02 COPYING
-rw-rw-r-- 1 olej olej 7597 янв 20 03:02 gorun.go
```

```
$ go get github.com/golang/lint/golint
package github.com/golang/lint/golint: code in directory
/home/olej/go/src/github.com/golang/lint/golint expects import
"golang.org/x/lint/golint"
```

```
$ ls -l `go env GOPATH`/src/github.com/golang/lint/golint
итого 20
-rw-rw-r-- 1 olej olej 3807 фев 22 10:12 golint.go
-rw-rw-r-- 1 olej olej 449 фев 22 10:12 importcomment.go
-rw-rw-r-- 1 olej olej 8459 фев 22 10:12 import.go
```

Конкретный размер и состав импортируемых проектов определяется только самим проектом, и может представлять собой от одного файла и до целого дерева каталогов-файлов.

Более подробную информацию о команде `get` вы можете получить выполнив:

```
$ go help get
...
```

Утилиты GoLang (tool)

Ещё одна команда `go`, заслуживающая комментариев — это команда `tool`, без параметров она выводит список всех доступных (установленных) внешних утилит экосистемы GoLang:

```
$ go tool
addr2line
asm
builddid
cgo
compile
covdata
cover
doc
fix
link
nm
objdump
pack
pprof
test2json
trace
vet
```

Мы уже встречали эти программные инструменты при рассмотрении переменных

56

окружения Go:

```
$ ls -w80 $(go env GOTOOLDIR)
addr2line  buildid  compile  cover   fix    nm      pack   test2json  vet
asm        cgo         covdata  doc     link  objdump pprof  trace
```

Как легко видеть — это список исполнимых утилит, находящихся в каталоге GOTOOLDIR (эта переменная окружения устанавливается командой go на основании GOROOT).

Чем наблюдаемые здесь внешние **утилиты** Go отличаются от **команд** Go? Судя по всему, в число утилиты вынесены достаточно редко требуемые команды, выполняющие инструментальные действия, более нужные разработчикам GoLang, чем пользователям Go.

Общий формат команд запуска утилит:

```
$ go help tool
usage: go tool [-n] command [args...]

Tool runs the go tool command identified by the arguments.
With no arguments it prints the list of known tools.

The -n flag causes tool to print the command that would be
executed but not execute it.

For more about each tool command, see 'go doc cmd/<command>'.
```

Как следует из этой краткой общей справки, по каждой из этих утилит может быть получена отдельная справка по её конкретному использованию, например так:

```
$ go tool nm -h
usage: go tool nm [options] file...
-n
    an alias for -sort address (numeric),
    for compatibility with other nm commands
-size
    print symbol size in decimal between address and type
-sort {address,name,none,size}
    sort output in the given order (default name)
    size orders from largest to smallest
-type
    print symbol type after name
```

```
$ go doc nm
Nm lists the symbols defined or used by an object file, archive, or
executable.
```

Usage:

```
go tool nm [options] file...
```

The default output prints one line per symbol, with three space-separated fields giving the address (in hexadecimal), type (a character), and name of the symbol. The types are:

```
T text (code) segment symbol
t static text segment symbol
R read-only data segment symbol
r static read-only data segment symbol
D data segment symbol
d static data segment symbol
B bss segment symbol
b static bss segment symbol
C constant address
```

U referenced but undefined symbol

Following established convention, the address is omitted for undefined symbols (type U).

The options control the printed output:

- n
an alias for -sort address (numeric),
for compatibility with other nm commands
- size
print symbol size in decimal between address and type
- sort {address,name,none,size}
sort output in the given order (default name)
size orders from largest to smallest
- type
print symbol type after name

\$ go tool fix -h

usage: go tool fix [-diff] [-r fixname,...] [-force fixname,...] [path ...]

- diff
display diffs instead of rewriting files
- force string
force these fixes to run even if the code looks updated
- r string
restrict the rewrites to this comma-separated list

Available rewrites are:

- cftype
Fixes initializers and casts of C.*Ref and JNI types
- context
Change imports of golang.org/x/net/context to context
- egl
Fixes initializers of EGLDisplay
- gotypes
Change imports of golang.org/x/tools/go/{exact,types} to go/{constant,types}
- jni
Fixes initializers of JNI's jobject and subtypes
- netip6zone
Adapt element key to IPAddr, UDPAddr or TCPAddr composite literals.
<https://codereview.appspot.com/6849045/>
- printerconfig
Add element keys to Config composite literals.

Утилиты компиляции

Отдельно следует обратить внимание на группу утилит, выполняющих поэтапную компиляцию и сборку приложений: `compile`, `link`, `objdump` ... В чём особый интерес? Во-первых, как понятно, для более детального анализа того что происходит при компиляции. А во-вторых, потому что Интернет переполнен вопросами: почему возникают упорные ошибки при любой компиляции вида:

\$ go tool compile tiny.go

tiny.go:2:8: could not import fmt (file not found)

Мы здесь используем в качестве эталонного приложения минимальный образец кода, который неоднократно используем по тексту (каталог `tools`):

tiny.go

```
package main
import "fmt"
```

```
func main() { fmt.Println("Hello, 世界") }
```

Мы сможем «научить» компилятор правильно подгружать информацию импорта ... но прежде посмотрим на каталог, где хранятся импорты:

```
$ du -hs `go env GOROOT`/pkg
85M      /usr/local/go/pkg
```

И теперь сделаем (команду подсказали в обсуждениях разработчики GoLang — именно так, без пробелов, и обязательно от имени root):

```
$ time sudo GODEBUG=installgoroot=all go install std
real      0m14.631s
user      0m0.017s
sys       0m0.007s
```

После чего (однократно выполненной команды) все компиляции пойдут успешно:

```
$ go tool compile tiny.go
```

```
$ ls -o tiny.o
-rw-rw-r-- 1 olej 7866 фев 22 08:52 tiny.o
```

И ещё раз посмотрим на каталог импортов:

```
$ du -hs `go env GOROOT`/pkg
209M     /usr/local/go/pkg
```

Нами выполненная команда подгрузила в каталог дополнительных 124Mb (209–85) из сети, что сделало возможным отделённые компиляции с отладочными возможностями (что конкретно подгрузила команда я не выяснял).

Теперь вернёмся к результату проделанной компиляции:

```
$ file tiny.o
tiny.o: current ar archive
```

```
$ ar -t tiny.o
___.PKGDEF
_go_.o
```

С удивлением отметим, что результатом компиляции не является объектный файл в общепринятом в Linux смысле и формате, а это AR архив Linux. Что мы можем получить дополнительно из такой компиляции? Во-первых, мы можем следующим шагом собрать приложение, так как мы делаем это и **командой** `build`, воспользовавшись **утилитой** `link`: (как и всегда и везде, выходной файл мы можем сразу определить опцией `-o ...` (а если не переопределять — то это будет `a.out`):

```
$ go tool link tiny.o
```

```
$ ls -o a.out
-rwxrwxr-x 1 olej 1895239 фев 22 09:03 a.out
```

```
$ file a.out
a.out: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, with debug_info, not stripped
```

```
$ ./a.out
Hello, 世界
```

Во-вторых, в смысле дополнительных возможностей, предоставляется ещё одна утилита этой группы `objdump`, для анализа итогов компиляции, вплоть до ассемблерного кода:

```
$ go tool objdump tiny.o
TEXT main.main(SB) /home/olej/2024/own.BOOKs/BHV.Go.3/examples/tools/tiny.go
    tiny.go:5                0x11ed                493b6610                CMPQ SP,
0x10(R14) [2:2]R_USEIFACE:type:string [2:2]R_USEIFACE:type:*os.File
```

tiny.go:5	0x11f1	764c	
JBE 0x123f			
tiny.go:5	0x11f3	55	
PUSHQ BP			
tiny.go:5	0x11f4	4889e5	
MOVQ SP, BP			
tiny.go:5	0x11f7	4883ec38	SUBQ
\$0x38, SP			
tiny.go:5	0x11fb	440f117c2428	
MOVUPS X15, 0x28(SP)			
tiny.go:5	0x1201	488d1500000000	
LEAQ 0(IP), DX			[3:7]R_PCREL:type:string
tiny.go:5	0x1208	4889542428	
MOVQ DX, 0x28(SP)			
tiny.go:5	0x120d	488d1500000000	
LEAQ 0(IP), DX			[3:7]R_PCREL:main..stmp_0<1>
tiny.go:5	0x1214	4889542430	
MOVQ DX, 0x30(SP)			
print.go:314	0x1219	488b1d00000000	
MOVQ 0(IP), BX			[3:7]R_PCREL:os.Stdout
print.go:314	0x1220	488d0500000000	
LEAQ 0(IP), AX			[3:7]R_PCREL:go:itab.*os.File,io.Writer
print.go:314	0x1227	488d4c2428	
LEAQ 0x28(SP), CX			
print.go:314	0x122c	bf01000000	
MOVL \$0x1, DI			
print.go:314	0x1231	4889fe	
MOVQ DI, SI			
print.go:314	0x1234	e800000000	
CALL 0x1239			[1:5]R_CALL:fmt.Fprintln
tiny.go:5	0x1239	4883c438	ADDQ
\$0x38, SP			
tiny.go:5	0x123d	5d	
POPQ BP			
tiny.go:5	0x123e	c3	
RET			
tiny.go:5	0x123f	e800000000	
CALL 0x1244			[1:5]R_CALL:runtime.morestack_noctxt
tiny.go:5	0x1244	eba7	
JMP main.main(SB)			

В-третьих, и это самое главное из возможностей, проделывая компиляцию с опцией `-m`, мы сразу можем увидеть какие переменные компилятор разместит в хип, а какие в локальном стеке. А для системы со сборкой мусора это очень важно знать. Делаем это так:

```
$ go tool compile -m tiny.go
```

```
/home/olej/2024/own.B00Ks/BHV.Go.3/examples/tools/tiny.go:5:6: can inline main
```

```
/home/olej/2024/own.B00Ks/BHV.Go.3/examples/tools/tiny.go:5:26: inlining call to  
fmt.Println
```

```
/home/olej/2024/own.B00Ks/BHV.Go.3/examples/tools/tiny.go:5:26: ... argument does not  
escape
```

```
/home/olej/2024/own.B00Ks/BHV.Go.3/examples/tools/tiny.go:5:27: "Hello, 世界" escapes to  
heap
```

В данном случае, это взято слишком простое приложение (чтобы не перегружать текст), но для серьёзного кода это может быть очень увлекательно!

Связь с кодом C (пакет Cgo)

Программный код Go может непосредственно использовать код, написанный на языке C. Для этого используется такой инструмент, как Cgo (<https://pkg.go.dev/cmd/cgo>), мы его только-что видели выше в составе `tool`. Для использования Cgo пишется обычный Go код, но который импортирует **псевдо-пакет "C"**. Go код после этого может ссылаться к типам как `C.size_t`, переменным как `C.stdout`, или к функциям как `C.putchar()`. Это особо актуально для операционной системы Linux, где таким образом обеспечивается низкоуровневый интерфейс ко всем **системным вызовам** POSIX API и динамическим разделяемым библиотекам `*.so` в Linux.

Сделаем простейшее тестовый приложение (каталог `tools`, пользуясь случаем проверяем ещё раз и «всесильность» UNICODE и UTF-8 для отображения на китайском языке слова «world»):

hello.go

```
package main

// #include <stdio.h>
// #include <stdlib.h>
import "C"
import "unsafe"

func main(){
    str := "Hello, 世界\n"
    cs := C.CString(str)
    C.fputs(cs, (*C.FILE)(C.stdout))
    C.free(unsafe.Pointer(cs))
}
```

Всё, что написано выше включения псевдо-пакета (строка «`import "C"`») и выглядит как комментарий (в синтаксисе Go) — является кодом на языке C, который может включать любые директивы и любой (или почти любой) код C или C++. Собираем и испытываем:

```
$ go build tiny.c.go
```

```
$ ls -l tiny.c
```

```
-rwxrwxr-x 1 olej olej 1249984 фев 22 12:30 tiny.c
```

```
$ ./tiny.c
```

```
Hello, 世界
```

Необходимые временами, и привычные, опции GCC компилятора `CFLAGS`, `CPPFLAGS`, `CXXFLAGS` и `LDFLAGS` могут быть тоже определены через псевдо-директивы `#cgo` также в виде комментария, чтобы настроить требуемое поведение C или C++ компилятора (см. переменные среды GoLang обсуждавшиеся выше: `CC="gcc"`, `CXX="g++"`, `CGO_CFLAGS="-g -O2"`, `CGO_CPPFLAGS=""`, `CGO_CXXFLAGS="-g -O2"`, `CGO_LDFLAGS="-g -O2"`) — значения, определенные в нескольких последовательных директивах, объединяются вместе, например:

```
// #cgo CFLAGS: -DPNG_DEBUG=1
// #cgo amd64 386 CFLAGS: -DX86=1
// #cgo LDFLAGS: -lpng
// #include <png.h>
import "C"
```

Альтернативно, `CPPFLAGS` или `LDFLAGS` могут быть определены через средства `pkg-config`, используя директиву `#cgo pkg-config`: со следующим за ней списком конфигурируемых пакетов:

```
// #cgo pkg-config: png cairo glu
// #include <png.h>
import "C"
```

Вспомните и сравните с системной командой Linux (относительно любой библиотеки):

```
$ pkg-config --libs glu
-lGLU
```

Любая функции C (даже возвращающая `void` функции) могут быть вызваны в контексте возврата множественных значений: возвращаемое значение (если оно есть) и привычная C переменная `errno` как код ошибка (используйте пустую переменную `_` чтобы опустить результат, если функция ничего не возвращает). Например:

```
n, err := C.sqrt(-1)
_, err := C.voidFunc()
```

Несколько специальных функций определено для преобразований между Go и C типами, которые производят копирование данных (из одного формата в другой). В псевдо-Go определениях они выглядят подобно следующему:

```
// Go string to C string
// The C string is allocated in the C heap using malloc.
// It is the caller's responsibility to arrange for it to be
// freed, such as by calling C.free (be sure to include stdlib.h
// if C.free is needed).
func C.CString(string) *C.char
// C string to Go string
func C.GoString(*C.char) string
// C string, length to Go string
func C.GoStringN(*C.char, C.int) string
// C pointer, length to Go []byte
func C.GoBytes(unsafe.Pointer, C.int) []byte
```

Поскольку строки C при копировании размещаются в хипе (C.malloc()), после использования их память должна быть освобождена C.free().

Вызов функций C по указателю на функции в настоящее время не поддерживается, однако вы можете объявить Go переменные, которые загрузить указателями на C функции и передавать их взад и вперед между Go и C. Сам C-код может вызвать функцию по указателям, полученным от Go. Например (каталог tools):

cex.go

```
package main
// typedef int (*intFunc) ();
//
// int bridge_int_func(intFunc f) {
//     return f();
// }
// int fortytwo() {
//     return 42;
// }
import "C"
import "fmt"

func main() {
    f := C.intFunc(C.fortytwo)
    fmt.Println(int(C.bridge_int_func(f)))
}
```

```
$ go build cex.go
```

```
$ ./cex
42
```

Таким образом, разрабатываемый код Go получает возможность использовать всё богатство наработанных библиотек C/C++ и произвольных фрагментов исходных кодов на этих языках, но, самое главное, все API библиотек POSIX и Linux !

Программа cgo может быть вызвана и автономно, по типу:

```
$ go tool cgo [cgo options] [-- compiler options] file.go
```

При этом cgo создаёт из входного файла file.go несколько выходных файла в подкаталоге _obj: 2 файла Go с исходным кодом, C файл для GCC и C файл GoLang. Например (уже виденный нами выше исходный hello.c.go):

```
$ go tool cgo hello.c.go
```

```
$ ls -l _obj
итого 28
-rw-rw-r-- 1 olej olej 605 фев 22 13:04 _cgo_export.c
-rw-rw-r-- 1 olej olej 1547 фев 22 13:04 _cgo_export.h
-rw-rw-r-- 1 olej olej 13 фев 22 13:04 _cgo_flags
-rw-rw-r-- 1 olej olej 819 фев 22 13:04 _cgo_gotypes.go
-rw-rw-r-- 1 olej olej 416 фев 22 13:04 _cgo_main.c
-rw-rw-r-- 1 olej olej 641 фев 22 13:04 hello.c.cgo1.go
-rw-rw-r-- 1 olej olej 1884 фев 22 13:04 hello.c.cgo2.c
```

Код на C, обратным порядком, также может использовать функции Go. Но это, как кажется, более экзотика чем необходимость, и о использовании таких возможностей можно почитать подробнее в фирменном описании cgo (<https://pkg.go.dev/cmd/cgo>).

Сторонний и дополнительный инструментарий

За последние годы на Go создано «от третьих сторон» столько интересного, полезного и разнообразного инструмента для разработчиков, что становится необозримым со всем ним ознакомиться, не только описать. На сегодня «на Go не пишет только ленивый». Я назову только несколько «лёгких» инструментов, которые позволяют по-быстрому поработать с Go.

Хороший и доступный инструмент для быстрого изучения Go, синтаксиса кода — это интерфейс WEB: <https://go.dev/play/>, предоставляемый на сайте самого проекта GoLang, позволяющий оперативно редактировать, изучать и отлаживать код непосредственно в браузере:

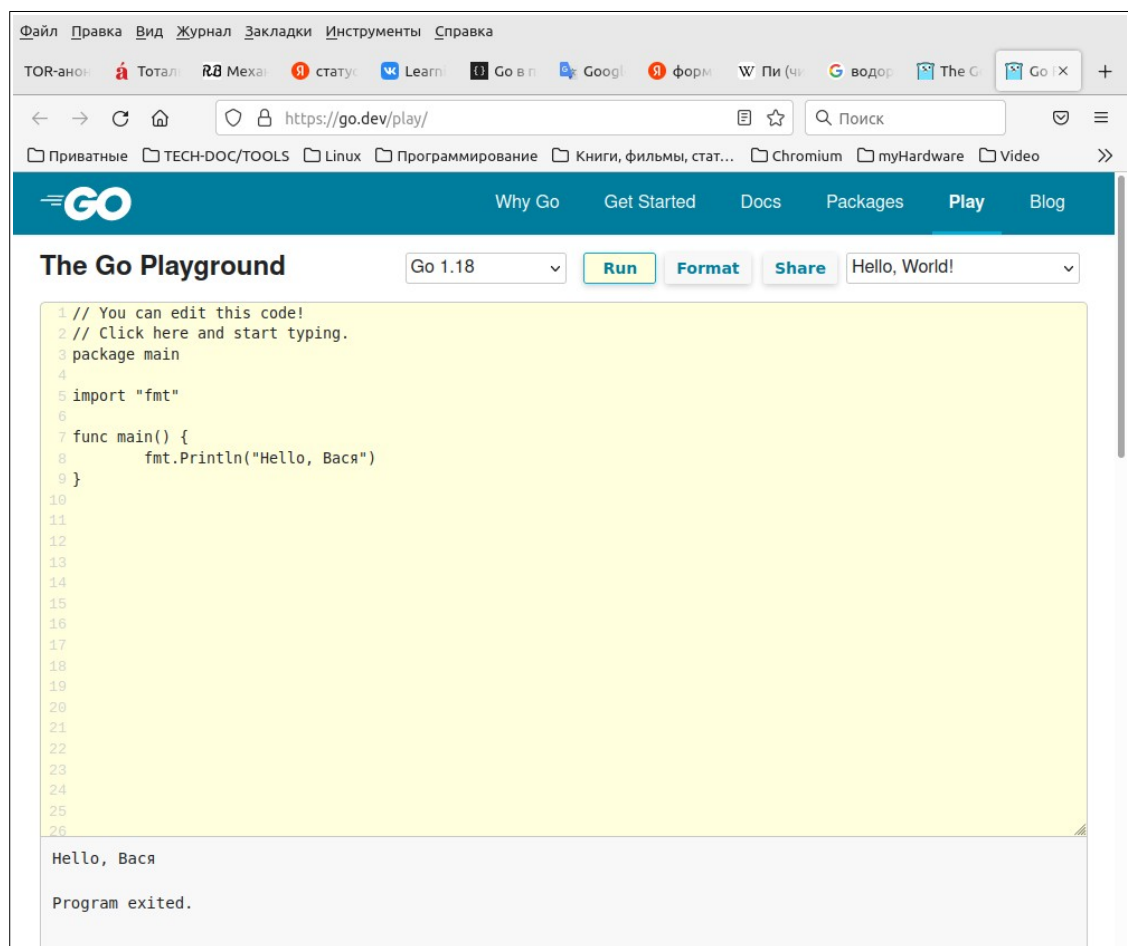


Рис. 1.4. Онлайн инструмент для изучения Go

Обратите внимание, что этот интерфейс позволяет выбрать версию GoLang, в которой вы хотите проверять свой код — вспомним, что стандарт Go динамично меняется, и то, чего в нём не было вчера, появится завтра.

Кроме того, для отработки кода хорошо бы иметь редактор с адекватной цветовой разметкой под выбранный язык. Из-за относительной новизны Go, многие из традиционных редакторов кода Linux ещё не имеют разметки под синтаксис Go (или не имели на момент написания этого текста), и её придётся настраивать под себя самостоятельно. Если вы не хотите терять время на ручную настройку цветовой разметки, то можете воспользоваться средой Geany, которая присутствует в репозиториях практически любого дистрибутива Linux. Geany не является в общепринятом смысле средой разработки (IDE), и в этом её большой плюс, а представляет собой развитый многооконный графический терминал, позволяющий «в одном флаконе» редактировать код, выполнять его сборку (make) в отдельном терминале, а также, запустив в этом терминале mc, осуществлять навигацию по файлам проекта и, возможно, оттуда же консольные тестовые запуски ... как-то так:

Очень большой набор инструментов и утилит для использования Go разработано непосредственно в рамках основного проекта GoLang, авторским коллективом Go: godoc, golint, vet, ... — этот набор активно расширяется. Ещё некоторая часть инструментария нарабатывается в качестве сторонних проектов: beego, revel ... Информация об отдельных таких средствах будет иногда упоминаться по ходу дальнейшего рассмотрения.

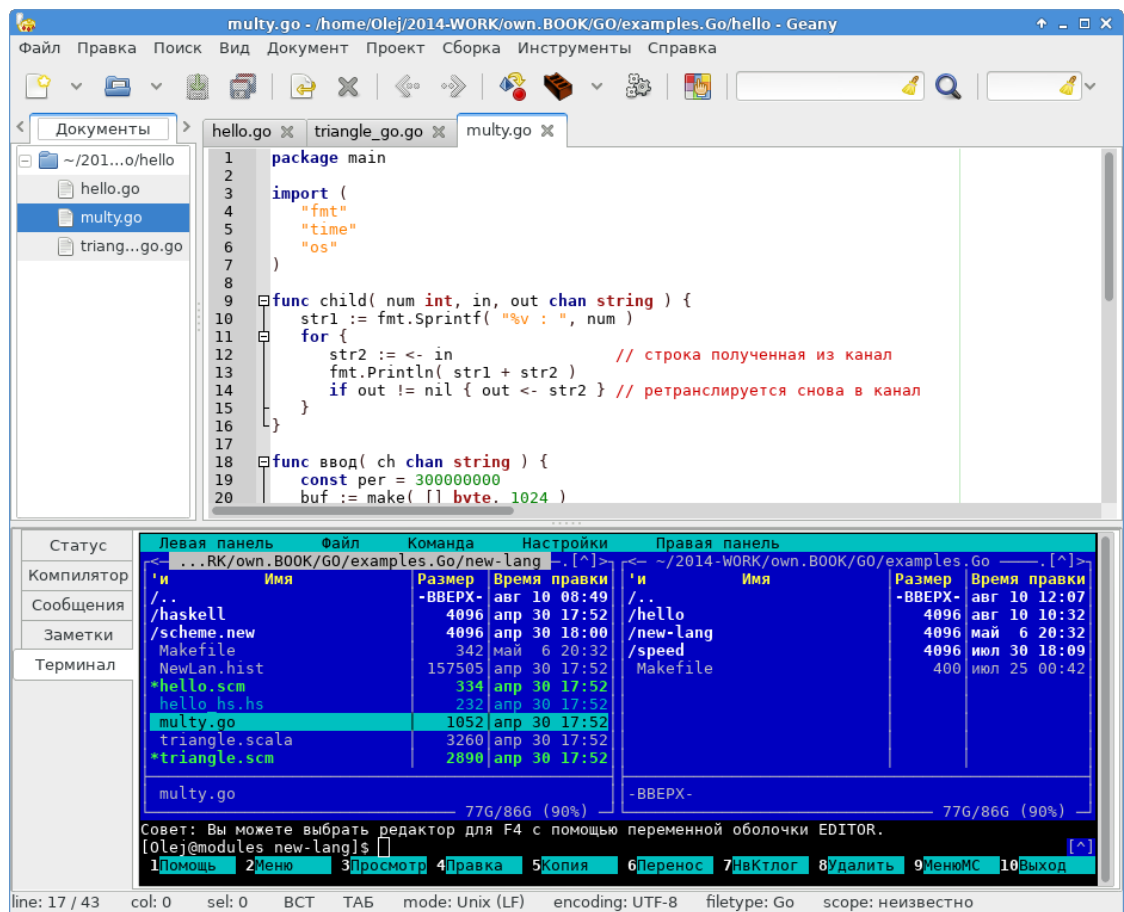


Рис. 1.5. Среда разработки Geany

Интерактивный отладчик Delve

Интерактивный отладчик Delve является отдельным от GoLang продуктом. К счастью, он входит в стандартные репозитории основных дистрибутивов Linux для стандартной установки:

```
$ aptitude search Delve | grep Go
```

```
p delve - debugger for the Go programming language
```

(Я отфильтровал вывод по Go только для того, чтобы не спутать с подобного названия инструментом JavaScript.)


```
$ sudo apt install delve
[sudo] пароль для oley:
Чтение списков пакетов... Готово
Построение дерева зависимостей... Готово
Чтение информации о состоянии... Готово
Следующие НОВЫЕ пакеты будут установлены:
  delve
Обновлено 0 пакетов, установлено 1 новых пакетов, для удаления отмечено 0 пакетов, и 4
пакетов не обновлено.
Необходимо скачать 3.865 кВ архивов.
После данной операции объём занятого дискового пространства возрастёт на 12,3 МВ.
Пол:1 http://mirror.mirohost.net/ubuntu jammy/universe amd64 delve amd64 1.8.1-1 [3.865
кВ]
Получено 3.865 кВ за 2с (1.962 кВ/с)
Выбор ранее не выбранного пакета delve.
(Чтение базы данных ... на данный момент установлен 536301 файл и каталог.)
Подготовка к распаковке .../delve_1.8.1-1_amd64.deb ...
Распаковывается delve (1.8.1-1) ...
Настраивается пакет delve (1.8.1-1) ...
Обрабатываются триггеры для man-db (2.10.2-1) ...
```

Исполнимый файл, интересующий нас отладчик, имеет имя `dlv`:

```
$ which dlv
/usr/bin/dlv
```

В общих чертах работа с интерактивным отладчиком Delve похожа на работу с хорошо известным GNU отладчиком GDB: позволяет расставлять точки останова, контролировать и изменять значение переменных в точках останова... Программа имеет развитую систему подсказок:

```
$ dlv --help
Delve is a source level debugger for Go programs.
```

```
Delve enables you to interact with your program by controlling the execution of the
process,
evaluating variables, and providing information of thread / goroutine state, CPU
register state and more.
```

```
The goal of this tool is to provide a simple yet powerful interface for debugging Go
programs.
```

```
Pass flags to the program you are debugging using `--`, for example:
```

```
`dlv exec ./hello -- server --config conf/config.toml`
```

```
Usage:
  dlv [command]
```

Available Commands:

```
  attach      Attach to running process and begin debugging.
  completion  Generate the autocompletion script for the specified shell
  connect     Connect to a headless debug server with a terminal client.
  core        Examine a core dump.
  dap         Starts a headless TCP server communicating via Debug Adaptor Protocol
(DAP).
  debug       Compile and begin debugging main package in current directory, or the
package specified.
  exec        Execute a precompiled binary, and begin a debug session.
  help        Help about any command
  run         Deprecated command. Use 'debug' instead.
  test        Compile test binary and begin debugging program.
  trace       Compile and begin tracing program.
  version     Prints version.
```

Flags:

<code>--accept-multiclient</code>	Allows a headless server to accept multiple client connections via JSON-RPC or DAP.
<code>--allow-non-terminal-interactive</code>	Allows interactive sessions of Delve that don't have a terminal as stdin, stdout and stderr
<code>--api-version int</code>	Selects JSON-RPC API version when headless. New clients should use v2. Can be reset via <code>RPCServer.SetApiVersion</code> . See <code>Documentation/api/json-rpc/README.md</code> . (default 1)
<code>--backend string</code> (default "default")	Backend selection (see 'dlv help backend').
<code>--build-flags string</code>	Build flags, to be passed to the compiler. For example: <code>--build-flags="-tags=integration -mod=vendor -cover -v"</code>
<code>--check-go-version</code>	Exits if the version of Go in use is not compatible (too old or too new) with the version of Delve. (default true)
<code>--disable-aslr</code>	Disables address space randomization
<code>--headless</code>	Run debug server only, in headless mode. Server will accept both JSON-RPC or DAP client connections.
<code>-h, --help</code>	help for dlv
<code>--init string</code>	Init file, executed by the terminal client.
<code>-l, --listen string</code> "127.0.0.1:0")	Debugging server listen address. (default
<code>--log</code>	Enable debugging server logging.
<code>--log-dest string</code>	Writes logs to the specified file or file descriptor (see 'dlv help log').
<code>--log-output string</code>	Comma separated list of components that should produce debug output (see 'dlv help log')
<code>--only-same-user</code>	Only connections from the same user that started this instance of Delve are allowed to connect. (default true)
<code>-r, --redirect stringArray</code> (see 'dlv help redirect')	Specifies redirect rules for target process
<code>--wd string</code>	Working directory for running the program.

Additional help topics:

<code>dlv backend</code>	Help about the <code>--backend</code> flag.
<code>dlv log</code>	Help about logging flags.
<code>dlv redirect</code>	Help about file redirection.

Use "dlv [command] --help" for more information about a command.

Детальное обсуждение работы с интерактивным отладчиком Delve выходит за рамки наших намерений и возможностей, но работа с ним на примерах отлично показана в книге [5].

Источники информации

[1] cgo.Documentation — https://pkg.go.dev/cmd/cgo#hdr-Go_references_to_C

[2] Andrew Gerrand, C? Go? Cgo! , 17 March 2011 — <https://go.dev/blog/cgo>

[3] Юникод — <https://docs.microsoft.com/ru-ru/windows/win32/intl/unicode>

[4] Кросс-компиляция в Go — <https://habr.com/ru/post/249449/>

[5] Михалис Цукалос, Golang для профи. Работа с сетью, многопоточность, структуры данных и машинное обучение с Go. 2020г., 720 страниц, изд. Питер, ISBN: 978-5-4461-1617-1

Неформально о синтаксисе Go

*Имейте в виду, если вы сделаете быстро
и плохо, то люди забудут, что вы
сделали быстро, и запомнят, что вы
сделали плохо. Если вы сделаете
медленно и хорошо, то люди забудут,
что вы сделали медленно, и запомнят,
что вы сделали хорошо!*
Сергей Королёв.

Синтаксис Go во многом заимствуется из классического C (у них общие авторы), что сильно снижает порог начального вхождения в работу с языком для имеющих минимальный опыт работы с C/C++. Go является прямым развитием языковой линии C/C++, но с заимствованиями многих «находок» из Oberon, Python, функциональных и скриптовых языков... мы об этом уже говорили ранее.

Именно поэтому этот раздел озаглавлен «неформально»: во-первых, потому что изложение опирается на аналогии из C, который предполагается более-менее известным¹⁴, а, во-вторых, потому что описание ниже акцентируется на тех сторонах Go, которые понадобятся дальше в рассмотрении параллельных вычислений, а некоторые другие стороны или названы вскользь, или даже совершенно опущены.

Go в значительной степени упрощает синтаксис C и делает его элегантным. Например: в Go отсутствуют обязательные ограничители операторов точкой с запятой. Совершенно понятно, что требование завершающих точки с запятой в C было вызвано только требованием простоты лексографического разбора кода, разбиение кода на лексемы, и связано это с неразвитостью инструментария лексографического разбора 40 лет назад. В Go, в большинстве случаев, достаточно просто перевода строки, который толкуется в смысле заменителя точки с запятой. Но это делает синтаксис записи Go не свободным в записи: смысл написанного кода может зависеть от его размещения по строкам. Такой отход от свободного стиля записи кода характерен для целого ряда современных языков: Python, Haskell. (Стремление к «свободе» синтаксиса, модное пару-тройку десятилетий назад, сменилось отказом от неё в новых разработках.)

Go трактует конец любой не пустой линии, как неявную точку с запятой. В результате этого в ряде случаев нельзя произвольно использовать перенос строки. Например, вы не можете написать:

```
func g()  
{  
    // НЕВЕРНО  
}
```

Точка с запятой будет поставлена после `g()`, и это приведет к тому, что данный код будет являться объявлением функции, а не её определением. Аналогично вы не можете написать:

```
if x {  
}  
else {  
    // НЕВЕРНО  
}
```

Точка с запятой будет поставлена после первой `}`, полностью заканчивает оператор `if` (укороченная форма), и перед `else` вызовет синтаксическую ошибку.

Если вы сомневаетесь в том, как Go толкует код, или он создаёт непонятные сообщения о синтаксических ошибках — **расставьте** (временно) точки с запятой в конце сомнительных строк, и всё станет ясно.

Так как точка с запятой явно обозначает конец выражения, вы вполне можете продолжать использовать такой ограничитель точно так же, как и в C и C++. Тем не менее, это не

¹⁴ Первоначально этот конспект так и назывался: «Язык Go для программистов C», и предполагает предпочтительным хотя бы поверхностное знание языка C.

рекомендуется так как засоряет текст. Идеиоматически Go опускает ненужные точки с запятой, и на практике использование точки с запятой ограничивается циклом `for` и случаем, когда вы хотите разместить на одной строке несколько коротких выражений.

Вместо того, чтобы беспокоиться о расположении точек с запятой и скобок, форматируйте ваш код с помощью команды `fmt` или программы `gofmt` (это не совсем одно и то же). Они дают единый стандартный стиль Go и позволяет вам волноваться за содержательную часть своего кода, а не его форматирование:

```
$ which gofmt
```

```
/usr/bin/gofmt
```

```
$ gofmt --help
```

```
usage: gofmt [flags] [path ...]
```

```
-cpuprofile string
```

```
    write cpu profile to this file
```

```
-d      display diffs instead of rewriting files
```

```
-e      report all errors (not just the first 10 on different lines)
```

```
-l      list files whose formatting differs from gofmt's
```

```
-r string
```

```
    rewrite rule (e.g., 'a[b:len(a)] -> a[b:]')
```

```
-s      simplify code
```

```
-w      write result to (source) file instead of stdout
```

```
$ go help fmt
```

```
usage: go fmt [-n] [-x] [packages]
```

Fmt runs the command 'gofmt -l -w' on the packages named by the import paths. It prints the names of the files that are modified.

For more about gofmt, see 'go doc cmd/gofmt'.

For more about specifying packages, see 'go help packages'.

The -n flag prints commands that would be executed.

The -x flag prints commands as they are executed.

To run gofmt with specific options, run gofmt itself.

See also: go fix, go vet.

Но Go не только заимствует из C, но и минимизирует набор допустимых конструкций, устраняя дублирование и избыточность. В Go доступны только управляющие конструкции `if`, `for` и `switch`. Первое, что бросается в глаза, это отсутствие круглых скобок:

```
Loop: for i := 0; i < 10; i++ {
    switch f(i) {
        case 0, 1, 2: break Loop
    }
    g(i)
}
```

При этом конструкция `goto` и метки сохранились (как бы кто на них не ругался), а операции инкремента и декремента более не являются выражениями, а являются операторами, и их нельзя подставлять непосредственно в вычисления выражений. Причём префиксная форма (`++i`) этих операций вообще отсутствует, используется только постфиксная (`i++`).

А в сравнении с C++ язык Go значительно упростил громоздкость и витиеватость последнего, но сохранил возможность реализации объектно-ориентированной парадигмы, хотя делает это совсем по-другому... и очень необычно, но об этом подробно позже.

Проще всего при беглом знакомстве с синтаксисом Go отталкиваться от правил C, от которого Go во многом происходит, и, отчасти, от C++, а в сравнении обращать внимание на самые принципиальные отличия. Вот какие основные отличия упоминаются в документации проекта Go и обсуждениях по языку:

— в Go есть указатели, но нет арифметики для них. Вы не сможете использовать переменную указатель для прохода по массиву, байтам или строке.

— в Go используется динамическая сборка мусора. Нет необходимости (и даже нет возможности!) освобождать память прямым указанием. Память объекта освобождается только когда число ссылок на объект становится нулевым. Сборка мусора инкрементная и высоко эффективна на современных процессорах.

— в Go нигде не используется неявное преобразование типов. Операции, которые сочетают разные типы, требуют явного приведения (называемого преобразованием в Go), даже если это, например, всего лишь целочисленные представления с разной разрядностью, например: `int8` и `int16`, или `int64` и `uint64` — это попарно перечислены разные типы, их смешение не допустимо.

— Go не поддерживает спецификаторы `const` или `volatile` для переменных, но есть не типизированный `const` как **описание** константных данных без типа (напоминающий препроцессорный `#define` в C).

— в Go используется `nil` для не инициализированных указателей, в то время, как в C в тех же случаях используются `NULL` или даже просто `0`, хотя это, конечно, вопрос только наименований.

— в Go нет классов с конструкторами или деструкторами. Вместо методов класса, иерархии наследования классов и виртуальных функций, в Go имеются методы и интерфейсы. Интерфейсы также используются там, где в C++ задействуются шаблоны.

— в Go отсутствует наследование типов (для похожей, но не идентичной, конструкции используется анонимное вложение типов, агрегирование).

— в Go не допускается переопределение методов (перегрузки функций) и нет определяемых пользователем операций.

— массивы в Go являются предопределёнными типами языка (а не агрегатами из существующих типов). Когда массив используется в качестве параметра функции, функция получает **копию** массива, а не указатель на него. Тем не менее, на практике функции часто используют **срезы** образуемые из массивов, для передачи параметров.

— в языке предусмотрены строки (`string`) как **предопределённый** тип (а не производный из существующих). Будучи один раз созданными, они не могут изменяться (строчным переменным, конечно, могут быть присвоены новые строчные данные, но это будут уже совсем другие данные, размещённые в другой области памяти — это подход Python). Такой код вполне корректен, но в первом и во втором присвоениях переменной присваиваются разные литеральные константы (неизменяемые) типа `string`, размещаемые в разных областях памяти:

```
func main() {  
    var x string  
    x = "first"  
    fmt.Println(x)  
    x = "second"  
    fmt.Println(x)  
}
```

— в Go предусмотрены хеш-таблицы (ассоциативные массивы) как предопределённый тип. Они ещё называются: таблицами, словарями, хешами (`map`).

— внутри языка предусмотрены разделенные параллельные ветви исполнения (`go`-процедуры, горутины, сопрограммы) и каналы связи (`channel`) между ветвями (сознательно не называю их «потокami», чтобы не путать с потоками ядра операционной системы `pthread_t` — это совершенно другие вещи).

— некоторые типы (словари и каналы) передаются по ссылке, а не по значению. Например, передача словаря в функцию не копирует словарь, и если функция изменяет словарь, то изменение будет видно там, откуда её вызвали.

— в Go не используются заголовочные файлы. Вместо этого, любой файл с исходным кодом — всегда составная часть определенного **пакета** (не может быть кода вне пакета). Когда пакет определяет объект (тип, константу, переменную, функцию) с именем, начинающимся с буквы в верхнем регистре (заглавной), этот объект виден для всех других файлов, которые импортирую (`import`) пакет в котором определён объект. Если имя начинается с буквы в нижнем регистре (малой, строчной), то этот объект не доступен, не виден за пределами пакета (это некоторый эквивалент видимости `public` и `private` из классов C++).

— Go не требует предшествующего описания используемых функций (либо полного описания, либо прототипа определения). Важно чтобы функция вообще только присутствовала в данном пакете (или файле). Предшествующее описание в C/C++ — это определённо рудимент, возникший только из требований простоты компиляции кода. В Go вполне допустима такая структура программы:

```
func main() {
    own_func()
    // ...
}

func own_func() {
    // ...
}
```

— объявление какого-либо объекта (имени переменной, пакета в списке импорта) в коде Go, и его дальнейшее не использование в коде — трактуется компилятором как грубая ошибка, прекращающая компиляцию. Вот что авторы языка сами пишут по этому поводу:

Ошибкой является импорт пакета или объявление переменной без их использования. Неиспользование импорта приводит к раздуванию программы и медленной её компиляции, в то время как переменная, которая инициализируется, но не используется, по крайней мере, растрчивает ресурсы, потраченное на её вычисление и, возможно, свидетельствует о серьёзной ошибке. Когда программа находится в стадии активной разработки, неиспользованные импорт и переменные часто возникают, и может быть раздражающим их удаление просто для того, чтобы продолжить компиляцию, тем более, что они снова могут потребоваться позже. Пустой идентификатор предоставляет временное решение.

И тут же предлагается решение: во всех таких местах использовать «пустой идентификатор» имени, обозначаемый символом одиночного подчёркивания («`_`»). Вот пример, предлагаемый для иллюстрации этого решения:

```
package main
import ("fmt" /*неиспользуемый*/; "io"; "log"; "os")

var _ = fmt.Printf // For debugging; delete when done.
var _ = io.Reader  // For debugging; delete when done.

func main() {
    fd, err := os.Open("test.go")
    if err != nil {
        log.Fatal(err)
    }
    // TODO: use fd.
    _ = fd
}
```

Язык чрезвычайно изящный: синтаксис во многом повторяющий C (без необходимости лишних разделителей ';' завершающих каждый оператор), дополненный своеобразным

механизмом классов (типов) и объектов, но без их громоздкости и тяжеловесности из C++.

В документации утверждается, что: *Компиляция выполняется очень быстро – намного быстрее, чем в некоторых других языках, особенно в сравнении с языками C и C++*. Это может оказаться существенным при работе над крупными проектами. Мы ещё вернёмся к этому факту.

Типы данных

В описаниях Go разделяются фундаментальные, предопределённые типы данных (first class type) и производные типы данных. К фундаментальным данным относятся, например: скалярные числовые типы, логические значения, символьные строки, массивы, хеш-таблицы, функции, интерфейсы, ... (достаточно обширный перечень и не всегда очевидный для программиста привыкшего к C/C++).

C/C++ и Go предоставляют подобные, но не идентичные, предопределённые типы данных: знаковые и беззнаковые целые числа разной (8, 16, 32, 64) разрядности, 32-разрядные и 64-разрядные, числа с плавающей точкой (вещественные и комплексные), структуры, указатели и др. В Go `uint8`, `int64` и подобно именованные целочисленные типы являются частью языка, а не построены на вершине иерархии целых чисел, размер которых зависит от реализации (например, `long long` в C). В языке существует большое количество различных типов простых скалярных данных. Например, имеются пять вариантов целочисленного типа `int`: `int`, `int8`, `int16`, `int32`, `int64`. Такие же типы данных, но с префиксом `u`, представляют беззнаковые значения.

Числа с плавающей точкой представлены тремя типами: `float`, `float32` и `float64`. Имеется даже два типа данных для комплексных переменных: `complex64` и `complex128`. Операции над комплексными значениями вводятся пакетом `math/cmplx`. Существует тип `byte` для представления коротких целых, и часто применяющийся для побайтового представления символов.

В Go допускается использовать имя `byte` как синоним беззнакового типа `uint8` и приветствуется использование имени `rune` как синонима типа `int32`, там где этим значением представляются отдельные символы UNICODE в кодировке UTF-32 (чтобы отличать их от собственно числовых значений, предназначенных для арифметических вычислений).

Помимо этого, стандартная библиотека (пакет `math/big` — о пакетах будет подробно рассказано далее) добавляет поддержку больших чисел: целых значений типа `big.Int` и рациональных значений типа `big.Rat`, которые имеют вообще неограниченный размер (то есть их размеры ограничиваются вообще только доступным объемом машинной памяти).

В языке Go нет нигде **неявного** приведения типов, поэтому смешение даже этих родственных типов между собой при компиляции вызывает терминальные ошибки.

Из простых скалярных типов Go предоставляет ещё логический тип `bool` — 1-битовый целочисленный тип, представляющий значение истинности в логических выражениях. Для представления логических значений предназначены логические константы в таком написании: `true` и `false`. Логические значения могут объединяться логическими операциями:

`&&` - операция «и»

`||` - операция «или»

`!` - операция отрицания (инверсии)

Логические значения могут быть выведены на печать как логические константы:

```
fmt.Println(true && true)
fmt.Println(true && false)
fmt.Println(true || true)
fmt.Println(true || false)
```

```
fmt.Println(!true)
```

Будет выведено (C/C++ в подобном контексте вывели бы целочисленное значение переменных, 0 или 1):

```
true
false
true
true
false
```

Из числа агрегатных типов данных Go дополнительно обеспечивает: встроенный тип строки (string), хеш-таблицы (map), каналы (channel), а также базовые массивы и их срезы. Символьные данные (string) представляются в UNICODE, а не ASCII. Строки, ввиду их важности, будут подробно рассмотрены далее.

Go гораздо более строго типизированным, чем даже C++. В частности, нет никакого **неявного** приведения типов в Go, только явное преобразование типа (int16 и int32 будут уже разными типами, не говоря уже о float64). Это обеспечивает дополнительную безопасность и свободу от целого класса ошибок, но за счет некоторой дополнительной строгости типизации. Также нет типа union, поскольку это позволило бы создание системы подтипов. Однако Go интерфейс, описанный как interface{} (пустой интерфейс) предоставляет собой тип-безопасную альтернативу — такой тип совместим со значением любого типа данных. Выражение T(v) преобразовывает значение v к типу T, пример некоторых численных преобразований:

```
var i int = 42
var f float64 = float64(i)
var u uint = uint(f)
```

Оба, и C++ и Go поддерживают псевдонимы (синонимы, алиасы) типа (typedef в C++ и type в Go). Однако, в отличие от C++, Go трактует новые объявленные типы как разные типы (строгая именная типизация). Следовательно, следующий код вполне допустим в C++:

```
// C++
typedef double position;
typedef double velocity;
position pos = 218.0;
velocity vel = -9.8;
pos += vel;
```

Но эквивалент такого кода недопустим в Go без явного приведения типа:

```
type position float64
type velocity float64
var pos position = 218.0
var vel velocity = -9.8
// pos += vel           // INVALID: mismatched types position and velocity
pos += position(vel)    // Valid
```

Go не позволяет указателям быть преобразованными в целые числа (или сконструированными из них), в отличие от C/C++. Однако, пакет Go unsafe позволяет явным образом обойти этот механизм безопасности в случае необходимости (например, при использовании кода для систем низкого уровня).

Описание всех предопределённых типов (в документации Go они называются типами 1-го уровня) производится в пакете builtin. Полное перечисление всех типов с их определениями можно получить из описания этого пакета: <https://pkg.go.dev/builtin>.

Переменные

Go в символьных представлениях везде последовательно использует UTF-8 кодировку

для представления UNICODE кодов символов. Поэтому даже и в именах переменных допускаются символы национальных алфавитов (русские, греческие, китайские, математически символы и др.). В этом нет ничего удивительного — ведь Роб Пайк, один из первоначальных архитекторов Go, и был разработчиком системы кодирования UTF-8 для UNICODE представления.

Вот как это выглядит на тестовом примере (каталог `hello`), здесь демонстрируется использование символов греческого алфавита как в именах переменных, так и в составе символьных константных строк для вывода:

circle.go

```
package main
import("fmt"; "os"; "strconv")

var π float64 = 3.1415926;

func main() {
    bufer := make([]byte, 80)
    for {
        fmt.Printf("радиус вашего круга? : ")
        длина, _ := os.Stdin.Read(bufer)
        str := string( bufer[:длина - 1 ])
        радиус, err := strconv.ParseFloat(str, 64)
        if err != nil {
            fmt.Println("ошибка ввода!")
            continue
        }
        fmt.Printf("длина окружности 2*π*радиус = %f\n",
                    2*π*радиус)
    }
}
```

Выполнение такого приложения:

```
$ ./circle
радиус вашего круга? : 11
длина окружности 2*π*радиус = 69.115037
радиус вашего круга? : .789
длина окружности 2*π*радиус = 4.957433
радиус вашего круга? : asd
ошибка ввода!
радиус вашего круга? : ^C
```

Объявление переменных

В сравнении с С или с С++, синтаксис объявления типов переменных «перевернут» (там где он вообще требуется), в стиле языков Pascal, Modula-2 или Oberon. Ниже показаны примерно эквивалентные объявления как они приведены в документации:

Go	C++
var v1 int	// int v1;
var v2 string	// const std::string v2; (примерно)
var v3 [10]int	// int v3[10];
var v4 []int	// int* v4; (примерно)
var v5 struct { f int }	// struct { int f; } v5;
var v6 *int	// int* v6; (но нет арифметики для указателей)
var v7 map[string]int	// unordered_map* v7; (примерно)
var v8 func(a int) int	// int (*v8)(int a);

Объявления переменных можно группировать:

```
var (
    i int
```

```
m float64
)
```

Но явно объявлять тип переменных, при такой строгости типизации, приходится, как ни странно, достаточно редко — язык Go поддерживает автоматический **вывод типов**: переменная может быть инициализирована при объявлении, её тип при этом можно не указывать, типом переменной становится (выводится) тип присваиваемого ей значения (примерно то, что появилось в C++ только со стандарта C++11/14):

```
var v = *p
```

Но если переменная не инициализирована **явно**, должен быть явно указан её тип. В таком случае переменной (не инициализированной) будет **неявно** присвоено нулевое значение, предусмотренное **для этого типа** данных (0 для целочисленных переменных, nil для указателей, пустая строка для string, свободное состояние для мютекса и так далее — для **каждого** типа обязательно существует **своё** значение, которое толкуется как нулевое). В Go вообще **не существует** и не может быть **не инициализированных** переменных.

Внутри функции короткий синтаксис присвоения локальным переменным значения с автоматическим выводом типов напоминает обычное присваивание в Pascal:

```
v1 := v2 // аналог var v1 = v2
```

Вне функции (в глобальной области), каждая конструкция (данных) начинается с ключевого слова (var, func и т.д.), а конструкция := является недопустимой:

```
package main
import "fmt"
var i, j int = 1, 2
func main() {
    k := 3
    c, python, java := true, false, "no!"
    fmt.Println(i, j, k, c, python, java)
}
```

Так же, как множественные присвоения или множественные возвраты из функций (см. далее), допускается и множественная инициализация переменных:

```
var v1, v2 uint32 = 10, 20
```

В Go имеется некоторое ограниченное число ключевых зарезервированных слов, которые могут употребляться только в свойственном им контексте, и не могут быть использованы в качестве имён переменных (или любых других объектов). Это обычная практика для большинства языков программирования. Вот **ключевые** слова (их очень немного):

break	case	chan	const	continue
default	defer	else	fallthrough	for
func	go	goto	if	import
interface	map	package	range	return
select	struct	switch	type	var

В языке Go есть, кроме того, много **предопределённых** идентификаторов (имена типов, логические константы, встроенные функции...). В программах допускается создавать собственные идентификаторы с этими именами (в отличие от ключевых слов), совпадающими с именами предопределённых идентификаторов, хотя это не всегда может быть целесообразным. Вот предопределённые имена (их перечень может расширяться с развитием версии):

append	bool	byte	cap	close
complex	complex64	complex128	copy	delete
error	false	float32	float64	imag
int	int8	int16	int32	int64
iota	len	make	new	nil
panic	print	println	real	recover

rune string

Повторные декларации и переприсвоения

Показанная чуть выше запись `v1 := v2` вводит **объявление** новой переменной `v1`. В то время, как запись `v1 = v2` **присваивает** значение ранее существующей переменной `v1`:

`v1 = v2` // присвоить существующей переменной `v1` значение переменной `v2`

Посмотрим пример, написанный по мотивам обсуждений на сайте GoLang:

revar.go

```
package main

import (
    "fmt"
    "os"
)

func main() {
    f, err := os.Open("revar")
    if err != nil {
        fmt.Println(err)
        os.Exit(1)
    }
    print(&err, "\n")
    d, err := f.Stat()
    if err != nil {
        println(err)
        f.Close()
        os.Exit(2)
    }
    println(&err)
    print("файл открыт:\n")
    fmt.Println(d)
    f.Close()
    os.Exit(0)
}

$ ./revar
0xc200004160
0xc200004160
файл открыт:
&{revar 27963 509 {63544826375 172386509 0x7fbbe4d83c00} 0xc200023000}
```

Посмотрим как работает оператор «:=» — краткая форма декларации переменной. Вызов `os.Open()` объявляет две новые переменные `f` и `err`. А немногими строками ниже следует оператор:

```
    d, err := f.Stat()
```

Он выглядит так, как если бы объявляются новые переменные `d` и `err`. Хотя, обратите внимание, что имя `err` фигурирует в обеих декларациях. Такое дублирование является законным: `err` объявляется первым оператором, но только вновь переприсваивается вторым. Это означает, что вызов `f.Stat()` использует существующую переменную `err`, объявленную ранее, и просто присваивает ей новое значение. (Это подчёркивает и специально сделанный вывод адреса размещения переменной `err` до и после присвоения.)

В «:=» декларации, переменная `v` может появляться повторно (даже если она уже была объявлена) в случаях если:

- это предыдущее объявление `v` сделано в той же программной единице, что и новое

объявление (если `v` была уже объявлена во внешней области, то новая декларация позволит создать новую переменную с тем же именем!);

- использовано соответствующее значение в инициализации присваиваем для `v` (по типу);

- существует по крайней мере ещё одна другая переменная в декларации, которая объявляется заново.

Это необычное свойство — это чистый прагматизм, который делает легким использование одной единственной переменной `err`, например, в длинной цепочке утверждений `if-else`. В Go мы видим это часто.

Некоторое представление об этих правилах повторных присвоениях даёт такой пример (каталог `types`, требует GoLang версии не ниже 1.22):

revar2.go

```
package main

func main() {
    var sum int;
    v := "v - это есть строка!"
    for v := range 5 {           // это становится 2-я переменная
        print(v, " ")
        v := v / 2              // это становится 3-я переменная
        sum += v
    }
    println()
    println(v)
    println(sum)
}
```

Проанализируйте — значения каких переменных выводятся в каждом случае вывода:

```
$ go run revar2.go
0 1 2 3 4
v - это есть строка!
4
```

Константы

В Go константы должны не иметь типа. Это применимо даже для констант, объявленных с помощью `const`, если в объявлении не указано типа, а инициализирующее выражение использует только запись выражений без типа. Значение константы без типа становится типизированным при использовании в контексте, который требует типизированное значение (это очень напоминает препроцессорные константы C, макросы), например, присвоение константы переменной. Это позволяет пользоваться константами относительно свободно и не требует явного преобразования типов:

```
var a uint
f(a + 1) // Численная константа без типа - 1 становится типа uint
```

Язык не налагает ограничений по размеру численных констант без типа или константных выражений. Ограничение применяется только в том месте, где при использовании константы потребуется для неё тип.

```
const huge = 1 << 100
f(huge >> 98)
```

Всё сказанное, об отсутствии типа у константных значений, соответствует документации ... но в реализации вызывает некоторые сомнения. Вот фрагмент кода (каталог `types` а архиве примеров):

const.go

```

package main

import "fmt"

const угол = 30

func main() {
    fmt.Printf("%T\n", угол)
    fmt.Printf("%f\n", угол)
    var x float32 = угол
    fmt.Printf("%f\n", x)
}

```

Выполнение этого фрагмента выглядит так:

```

$ go run const.go
int
%!f(int=30)
30.000000

```

Компилятор по виду записи константного значения 30 делает предположение о типе `int` для константы `угол`, которая позже участвует в присвоении вещественной переменной, что, вообще то говоря, должно быть недопустимо без преобразования типа для целочисленных значений.

Константные значения могут быть записаны также и с явным указанием типа этого значения:

```
const y float64 = 1.0
```

Go не поддерживает перечислений `enum`. Вместо этого можно использовать специальное зарезервированное имя `iota` в одном объявлении `const`, чтобы получить набор увеличивающихся значений. Когда в `const` опущено инициализирующее выражение, повторно используется предыдущее выражение.

```

const (
    red = iota // red == 0
    blue      // blue == 1
    green     // green == 2
)

```

Имя `iota` может использоваться и в любом другом произвольном контексте в определении констант (но не в каком-то другом) — это специальный счетчик, значение которого увеличивается при каждом его последующем упоминании в пределах **файла** кода:

```

const {
    a, b = iota, iota;
}

```

Константы `a` и `b` получают значения 0 и 1 соответственно. Так как `iota` может быть неявно повторяемой для одного или нескольких выражений, то легко можно строить сложные наборы значений. Вот совсем не очевидный пример (каталог `types` в архиве примеров), который приводят непосредственно авторы Go:

iota.go

```

package main
import "fmt"

type ByteSize float64

const (
    _ = iota // ignore first value by assigning to blank identifier

```

```

        KB ByteSize = 1 << (10 * iota)
        MB
        GB
        TB
        PB
        EB
        ZB
        YB
    )

func main() {
    fmt.Println(KB, MB, GB, TB, PB, EB, ZB, YB)
}

```

И вот результат (который может казаться неожиданным):

```

$ ./iota
1024 1.048576e+06 1.073741824e+09 1.099511627776e+12 1.125899906842624e+15
1.152921504606847e+18 1.1805916207174113e+21 1.2089258196146292e+24

```

Агрегаты данных

Go — это язык со сборкой мусора. Поэтому в нем можно динамически выделять память под объекты, но освобождать ее никогда не нужно (да и невозможно), так как этим занимается сборщик мусора.

Массивы и срезы

Главное отличие массивов в Go от большинства популярных языков — это то, что они являются значениями, то есть имя массива не является ссылкой. Массив может быть объявлен, создан и инициализирован так (варианты):

```

var a0 [7]int;
type arr [7]int
a1 := arr {0:1, 2:2, 4:3, 6:4}
a2 := *new(arr)
a2 = a1

```

Go имеет два примитива выделения памяти: встроенные функции `new()` и `make()`. Они делают разные вещи и применяются для различных типов, могут вводить в заблуждение, но правила их просты. Прежде всего о `new()`. Это встроенная функция, которая выделяет память, но в отличие от своего тезки в некоторых других языках, она никак не инициализирует память, а только обнуляет её (в соответствии с правилами того, что является нулевым значением для каждого типа). То есть, `new(T)` выделяет для использования и обнуляет новый элемент данных типа `T` и возвращает его адрес, значение типа `*T`. В терминологии Go функция `new()` возвращает указатель на вновь выделенных обнулённое значение типа `T`. Поэтому показанное выше выражение создаст массив и вернёт на него указатель:

```
a2 := *new([7]int)
```

Массивы популярны когда точно известно необходимое количество памяти, чтобы не делать излишних перераспределений, но в первую очередь они являются составной частью для срезов. Какие основные отличия между обращением с массивами между языками Go и C:

- массивы — это значения. Присвоение одно массива другому копирует все элементы.
- если вы передаёте массив в функцию, то передаётся копия массива, а не указатель на него.
- размер массива является частью типа массива. Типы `[10]int` и `[20]int` совершенно разные.

Длина массива является составной частью его типа, поэтому размер массива не может быть изменён. И массив размерностью, скажем, 7 не может быть присвоен переменной массива размерностью 9. Всё это может показаться существенным ограничением (хотя это привычная практика C/C++), но Go предлагает гибкий путь работы с массивами, используя такое понятие как **срез** (slice).

Срезы являются обёртками для массивов, дающими более общий, мощный и удобный интерфейс для последовательностей данных. За исключением задач с явными фиксированными измерениями, таких как преобразование матриц, в большинстве случаев программирование с массивами делается со срезами, а не напрямую с массивами.

Срез всегда делается как наложение, надстройка некоторой структуры последовательных (индексируемых) элементов над базовым массивом. В некоторых случаях срез создаётся даже без явного создания и указания базового массива, но массив всегда присутствует и может создаваться неявно.

Длина среза может меняться, пока не исчерпает размер внутреннего массива. С помощью встроенной функции `cap()` можно узнать ёмкость среза, представляющий максимальную длину среза (и это размер базового массива). В противоположность, встроенная функция `len()` возвращает текущую длину среза (`cap()` и `len()` - это не одно и то же!).

Встроенная функция `make(T, args)` служит целям отличающимся от `new(T)`. Такой вызов создаёт только срез, таблицу или канал типа `T` (но не `*T`), но в инициализированном (а не обнулённом) состоянии. Смысл такого различия состоит в том, что эти три типа представляют собой (во внутреннем представлении) ссылки на некоторые структуры данных, которые должны быть инициализированы перед их использованием. Срез, например, является 3-компонентным дескриптором, содержащим: а). указатель на данные (внутри базового массива), б). текущую длину данных и в). ёмкость созданного среза (объём среза всегда больше или равен длине, и определяется параметрами базового массива). И до тех пор, пока эти элементы не инициализированы, значением дескриптора является `nil`. Для срезов, таблиц и каналов `make()` прорабатывает инициализацию внутренних структур данных и подготавливает их к использованию. Например:

```
b := make([]int, 10, 100) // len(b) == 10, cap(b) == 100
```

Это определение разместит (неявно) массив из 100 значений `int`, и затем создаст структуру среза с длиной 10 и ёмкостью (объёмом) 100 (совпадающей с длиной базового массива), срез будет представлять собой первые 10 элементов базового массива.

```
b := make([]int, 10) // len(b) == 10, cap(b) == 10
```

Когда создаётся срез, ёмкость может быть опущена, и тогда она будет равна требуемой длине;

```
b := *new([]int) // len(b) == 0, cap(b) == 0
```

В противоположность предыдущим случаям, создаёт срез, нулевой длины и ёмкости, и возвращает указатель на структуру среза нулевой длины (это `nil` срез), позже срез может быть перемещён используя `make()`;

Получить адрес создаваемого массива можно через `&arr`. Но из-за отсутствия адресной арифметики, никакой пользы из знания адреса извлечь нельзя, разве что передавать в функцию адрес массива вместо его копии, что положительно влияет на производительность. Массив при создании может быть явно инициализирован или не инициализирован.

Инициализации перечисляются через запятую. Для инициализации выделенных (не последовательных) элементов можно указать индекс элемента и далее через двоеточие его значение:

```
var arr [10]int {2:1, 3:1, 5:1, 7:1}
```

Если просто указать `n` значений через запятую, то будут инициализированы последовательно только первые `n` элементов. Так как в Go вся память переменных инициализируется, то все элементы, значения для которых не были заданы явно, получают нулевые значения по умолчанию (в Go нет не инициализированных переменных!).

Пример некоторых объявлений и инициализации массивов показан ниже:

array2.go

```
package main

func show(p *[]int) {
    print("len=", len(*p), " cap=", cap(*p), " : ")
    for _, y := range *p {
        print(y, " ")
    }
    print("\n")
}

var a1 []int

func main() {
    a2 := []int{1, 2, 3, 4, 5}
    a3 := []int{2: 1, 4: 1, 7: 1, 9: 1}
    a := a1
    show(&a)
    a = a2
    show(&a3)
    a = a3
    show(&a)
}

$ ./array2
len=0 cap=0 :
len=5 cap=5 : 1 2 3 4 5
len=10 cap=10 : 0 0 1 0 1 0 0 1 0 1
```

Здесь показано много деталей, отличающих Go от традиций C/C++:

- массивы могут присваиваться как значения (`a = a2`), при этом происходит их копирование;
- не инициализированное описание массива (вида `var a1 []int`) создаёт пустой массив размерности 0, позже он может быть расширен использованием функции `make()`;
- операция `a := ...` является описанием новой переменной и её инициализацией, повторная запись такого оператора вызовет синтаксическую ошибку;
- операция `a = ...` означает копирование массива с сохранением его типа (типы `[]int` и `[10]int` — различные!);
- массивы как параметры вызова функций передаются копированием, по значению, что полностью противоположно подходу C/C++;
- в программе специально показана (искусственно сделанная) передача указателей массивов в функцию по ссылке, но даже при этом размерность массива не теряется;
- всякий массив и срез имеют характеристики, которые возвращаются встроенными функциями `len()` — его текущая длина и `cap()` — объём, ёмкость;
- для массивов значения `len()` и `cap()` совпадают, для срезов `len()` меньше или равно `cap()`, а, как мы увидим далее, `cap()` — это и есть `len()` массива, над которым надстроен срез, плюс начальное смещение этого среза.

Задать значение среза можно не только при неявном создании базового массива функцией `make()`, но и адресом или фрагментом («срезом» — отсюда и название) базового массива, например:

```
sl = arr[7:9];
sl = &arr
```


Предположим, что дан массив или его срез `a`. Новый срез `b` (над массивом или срезом) создается с помощью выражения `a[i:j]`. Будет создан новый срез, ссылающийся на `a`, начинающийся с индекса `i` и заканчивающийся перед индексом `j`. Он будет иметь длину (`len()`) равную `j - i`. Новый срез ссылается на тот же массив, на который ссылается `a`, но начиная с элемента `a[i]`. Так, все изменения, сделанные с помощью нового среза `b`, можно увидеть, используя исходный `a`. Объем нового среза (`cap()`) — это просто объем `a` минус `i`: `cap(a) - i`. Также вы можете присвоить указатель на массив переменной типа среза. Дано: `var s []int; var a[10]int`, присвоение `s = &a` эквивалентно `s = a[0:len(a)]`.

Также, к примеру:

```
s[lo:lo] // пустой срез длиной 0
s[lo:lo+1] // срез из одного элемента
```

Как уже должно быть понятно, в Go срезы используются часто именно для тех случаев, где в C/C++ используются указатели (на начало массива). Если вы создадите массив типа `[100]byte` (массив из 100 байт, — возможно, буфер) и захотите передать его в функцию без копирования, вы должны объявить параметр функции типа `[]byte` и передать адрес массива. В отличие от C/C++, нет необходимости передавать длину буфера, она просто доступна через `len()`.

Работу с массивами и срезами, их перерасмещением с изменением размера показывает тестовая программа:

array1.go

```
package main

import (
    "fmt"
    "os"
    "strconv"
)

var arr []int

func main() {
    buf := make([]byte, 120)
    for {
        print("len=", len(arr), " cap=", cap(arr), "\n")
        fmt.Printf("+/- ? : ")
        n, _ := os.Stdin.Read(buf)
        n, _ = strconv.Atoi(string(buf[:n-1]))
        n = n + len(arr) // новый размер
        if n < 1 {
            n = 1
        }
        if n >= cap(arr) { // недостаточно места
            nar := make([]int, n*2) // выделение вдвое больше
            arr = arr[0:cap(arr)]
            for i := range arr {
                nar[i] = arr[i]
            }
            arr = nar
        }
        arr = arr[0:n]
        arr[n-1] = n
        for _, a := range arr {
            print(a, " ")
        }
        print(" : ")
    }
}
```

```

}

$ ./array1
len=0 cap=0
+/- ? : 1
1 : len=1 cap=2
+/- ? : 3
1 0 0 4 : len=4 cap=8
+/- ? : 4
1 0 0 4 0 0 0 8 : len=8 cap=16
+/- ? : -5
1 0 3 : len=3 cap=16
+/- ? : 8
1 0 3 4 0 0 0 8 0 0 11 : len=11 cap=16
+/- ? : ^C

```

Синтаксис среза может быть также использован со строками. Вернётся новая строка, чье значение будет подстрокой исходной. Так как строки не изменяемы, строковые срезы могут быть реализованы без выделения новой памяти для содержимого среза.

Двумерные массивы и срезы

Массивы и срезы Go одномерные. Для создания эквивалентов 2D массивов и срезов необходимо определить массив-массивов или срез-срезов, подобно следующим:

```

type Transform [3][3]float64 // a 3x3 array, really an array of arrays.
type LinesOfText [][]byte    // a slice of byte slices.

```

Поскольку срезы имеют переменную длину, то возможно иметь для каждого отдельного среза различную длину. Это может быть достаточно общей ситуацией, как в таком вот варианте, где каждая строка переменной типа `LinesOfText` (срез срезов) имеет независимую длину:

```

text := LinesOfText{
    []byte("Now is the time"),
    []byte("for all good gophers"),
    []byte("to bring some fun to the party."),
}

```

Иногда бывает необходимо разместить 2D срез, в ситуациях которые могут возникнуть при обработке сканированных линий пикселей, например. Существует два способа достижения этой цели. Один — это выделить каждый срез самостоятельно. Другой — это выделить единый массив и указывать отдельные срезы из него. Что использовать зависит от конкретики приложения. Если срезы должны увеличиваться или уменьшаться, то они должны быть выделены независимо, чтобы избежать перезаписи следующей строки. Если нет, то может быть более эффективным создать объект с единым распределением. Для справки, вот эскизы двух методов.

Во-первых, строка за строкой (срез срезов):

```

// Allocate the top-level slice.
// One row per unit of y.
picture := make([][]uint8, Ysize)
// Loop over the rows, allocating the slice for each row.
for i := range picture {
    picture[i] = make([]uint8, Xsize)
}

```

А во-вторых, как альтернатива, теперь как единое выделение, нарезанное линиями:

```

// Allocate the top-level slice, the same as before.
// One row per unit of y.
picture := make([][]uint8, Ysize)
// Allocate one large slice to hold all the pixels.

```

```
// Has type []uint8 even though picture is [][]uint8.
pixels := make([]uint8, Xsize * Ysize)
// Loop over the rows, slicing each row from the front of the remaining pixels slice.
for i := range picture {
    picture[i], pixels = pixels[Xsize:], pixels[Xsize:]
}
```

Структуры

Структуры трактуются как набор полей — это общее место всех языков программирования:

struct1.go

```
package main

import "fmt"

type Vertex struct {
    X int
    Y int
}

func main() { fmt.Println(Vertex{1, 2}) }
```

```
$ gccgo -g struct.go -o struct

$ ./struct 1
{1 2}
```

Доступ к полям структуры производится через точку, «.». Go имеет указатели, но не допускает арифметики указателей (в таком случае уместнее было бы указатели называть ссылками, как делают в Java, например, но авторы Go используют наименование «указатель»). При использовании указателя на структуру также применяется «.», вместо «->» или «*» используемых в C и C++. С точки зрения синтаксиса, структура и указатель на структуру используются в Go одинаково:

```
type myStruct struct { i int }
var v9 myStruct           // v9 является структурой
var p9 *myStruct          // p9 указатель на структуру
f(v9.i, p9.i)             // поле структуры, указатель на поле структуры
```

Всё это хорошо видно на примере:

struct2.go

```
package main

import "fmt"

type Vertex struct {
    X int
    Y int
}

func main() {
    p := Vertex{}
    fmt.Println(p)
    q := &p
    q.X = 1e9
    p.Y = -3
    fmt.Println(p)
}
```

```
$ ./struct2
```

```
{0 0}  
{1000000000 -3}
```

Выражение `new(T)` размещает новый обнулённый экземпляр типа `T` (в Go не бывает не инициализированных значений) и возвращает указатель на него:

```
var t *T = new(T)  
t := new(T)
```

Как это происходит показано на примере:

struct3.go

```
package main  
  
import "fmt"  
  
type Vertex struct {  
    X, Y int  
}  
  
func main() {  
    v := new(Vertex)  
    fmt.Println(v)  
    v.X, v.Y = 11, 9  
    fmt.Println(v)  
}  
  
$ ./struct3  
&{0 0}  
&{11 9}
```

Структуры (и типы образуемые из структур) могут иметь не именованные поля. Такие поля называются встраиваемыми, в отличие от именованных, называемых агрегированными. Например:

```
import "color"  
...  
type ColoredPoint struct {  
    color.Color // Анонимное (безымянное) поле (встраивание)  
    x, y int    // Именованные поля (агрегирование)  
}
```

Все операции, допустимые для типа встраиваемого поля, автоматически допустимы непосредственно и для значений типа структуры, в которую встроено поле. Если создать значение типа `ColoredPoint` (например, как: `point := ColoredPoint{}`), то его поля будут именоваться, соответственно, как: `point.Color`, `point.x` и `point.y`.

Это имеет существенное значение для объектно-ориентированной модели Go, из которой исключено понятие наследования типов, и это будет показано при рассмотрении этой объектной модели. Это, фактически, альтернативная замена понятия наследования.

Таблицы (хеши)

Хеш-таблицы хорошо известны, например, из языка Python или привнесены библиотеками STL в стандарт C++ (со стандарта C++11 перенесено как составная часть языка). В Go таблицы вводятся как встроенный тип языка. Ниже приведен пример создания хеш-таблицы:

```
var mp = map[string] float {"first":1, "second":2.0001}
```

Таблицы могут инициализироваться как и структуры, но нужно обязательное указание ключа для каждого элемента:

map1

```

package main

import "fmt"

type Vertex struct {
    Lat, Long float64
}

var m = map[string] Vertex{
    "Bell Labs": Vertex{
        40.68433, -74.39967,
    },
    "Google": Vertex{
        37.42202, -122.08408,
    },
}

func main() {
    fmt.Println(m)
}

$ ./map1
map[Google:{37.42202 -122.08408} Bell Labs:{40.68433 -74.39967}]

```

Хеш-таблицы можно и не инициализировать, тогда у нас образуется пустая таблица не содержащая (ещё) элементов. Обращаться к элементам можно как в ассоциативном массиве в PHP: `mp["second"]`. Понятно, что массив подобен хеш-таблице с типом ключа `int`, а хеш-таблицы можно условно рассматривать как массивы, индекс которых может иметь произвольный тип.

В Go существует удобная форма `for: range` для итерации по значениям строки, массива или хеш-таблицы (то есть по любым перечислимым, индексруемым типам), как показано ниже (`range` — это одно, из не так многих, ключевых слов в Go, которое не может быть использовано ни в каком другом качестве):

```

for key, value := range mp {
    fmt.Printf("key %s, value %g\n", key, value)
}

```

Основные операции над таблицами (назовём таблицу условно `m`):

1. Вставить новый или обновить значение уже существующего элемента:
`m[key] = elem`
2. Возвратить значение элемента:
`elem = m[key]`
3. Удалить элемент из таблицы:
`delete(m, key)`
- 4) проверить присутствие элемента с заданным ключом — присвоение 2-х значений:
`elem, ok = m[key]`

В последней операции: если ключ присутствует в таблице, `ok` возвращается `true`. Если нет — то `ok` устанавливается `false`, а значение `elem` — нулевое значение в соответствии с типом элементов таблицы. Вообще, когда читается из таблицы элемент с отсутствующим ключом, возвращается нулевое значение в соответствии с типом элементов:

map2

```

package main

import "fmt"

```

```
func main() {
    m := make(map[string]int)
    m["Answer"] = 42
    fmt.Println("The value:", m["Answer"])
    m["Answer"] = 48
    fmt.Println("The value:", m["Answer"])
    delete(m, "Answer")
    fmt.Println("The value:", m["Answer"])
    v, ok := m["Answer"]
    fmt.Println("The value:", v, "Present?", ok)
}

$ ./map2
The value: 42
The value: 48
The value: 0
The value: 0 Present? false
```

Динамическое создание переменных

В Go есть встроенная функция `new()`, которая принимает тип и выделяет пространство в куче под объект такого типа. Выделяемое пространство будет инициализировано **нулем** для данного типа. Например, `new(int)` выделит новый `int` в куче (а не в стеке как при объявлении локальной переменной), инициализирует его значением 0 и вернет его адрес, который имеет тип `*int`. В отличие от C++, `new()` это функция, а не оператор, поэтому запись `new int` приведет к синтаксической ошибке.

Покажется удивительным, но `new()` не часто используется в Go программах. В Go взятие адреса переменной всегда безопасно и не может создать висячий указатель. Если программа где-то использует адрес переменной, она будет размещаться в хипе столь долго, сколько это будет необходимо (пока сохраняется последняя ссылка на эту переменную). Поэтому вот такие функции эквивалентны:

```
type S { I int }
func f1() *S {
    return new(S)
}
func f2() *S {
    var s S
    return &s
}
func f3() *S {
    // More idiomatic: use composite literal syntax.
    return &S{}
}
```

В противоположность этому, в C/C++ всегда опасно возвращать адрес локальной переменной (размещённой в стеке, и уничтожаемой вместе со стеком):

```
// C++
S* f2() {
    S s;
    return &s; // INVALID -- contents can be overwritten at any time
}
```

Значения словарей (`map`) и каналов (`channel`) должны выделяться с помощью встроенной функции `make()`. Переменная с типом `map` или `channel` без инициализации будет автоматически инициализирована `nil`. Вызов `make(map[int]int)` вернет новую переменную типа `map[int] int` (таблица целых значений, индексируемых целочисленным индексом). Отметим, что `make()` возвращает значение, а не указатель. Это согласуется с тем фактом, что значения `map` и `channel` передаются по ссылке. Вызов `make()` с типом `map` принимает необязательный аргумент, обозначающий

зарезервированный объем словаря. Вызов `make()` с типом `channel` принимает необязательный аргумент, который устанавливает объем буфера канала (по умолчанию равен 0 — не буферизируемый канал, что будет рассмотрено позднее):

```
mp = map([string]int, 200)
ch = map(chan int, 3)
```

Функция `make()` может также использоваться сразу для выделения среза. В таком случае, будет выделена память и под соответствующий базовый массив, и возвращен срез, ссылающийся на массив. Требуется один аргумент — количество элементов среза. Второй, необязательный, это объем среза. Например, `make([]int, 10, 20)` аналогично `new([20]int)[0:10]`. Так как в Go реализована сборка мусора, новый выделенный массив будет уничтожен только (и сразу) после того, как не останется ссылок на возвращаемый таким `make()` срез.

Конструкторы и составные литералы

Функция `new()` создает объект, инициализированный нулями. Не всегда это подходящий вариант. Но в Go не предоставляется конструкторов и деструкторов для структур и объектов класса. Если нужно что-то вроде конструктора, то следует создать конструирующую функцию. Например, как это делается в стандартном пакете с именем `os`:

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := new(File);
    f.fd = fd;
    f.name = name;
    f.dirinfo = nil;
    f.nepipe = 0;
    return f;
}
```

Но можно сократить количество лишних операций за счет составных литералов:

```
func NewFile(fd int, name string) *File {
    if fd < 0 {
        return nil
    }
    f := File{fd, name, nil, 0};
    return &f;
}
```

Две последние строки этого варианта можно еще подсократить. Они эквивалентны следующей строке:

```
return &File{fd, name, nil, 0};
```

В составном литерале все атрибуты должны указываться в порядке их перечисления в исходной структуре (позиционное указание полей). Но можно использовать и перечисления вида `field:value` — указываются только необходимые атрибуты (ключевое указание полей), а остальные обнуляются:

```
return &File{fd:fd, name:name}
```

В предельном случае запись `new(File)` эквивалентна записи `&File{}` — все атрибуты получают нулевые значения.

Составные литералы могут использоваться для инициализации массивов, слайсов и `map`-ов:

```
// Массив, чей размер определяется автоматически по содержимому
```

```

a := [...]string {Enone: "no error", Eio: "Eio", Eival: "invalid argument"}
// Это слайс
s := []string {Enone: "no error", Eio: "Eio", Eival: "invalid argument"}
// Это map (хеш-таблица)
m := map[int]string {Enone: "no error", Eio: "Eio", Eival: "invalid argument"}

```

Операции

Go допускает множественные инициализации и присваивания, выполняемые параллельно для нескольких переменных:

```
i, j := k, m // Инициализировать новые переменные
```

Оператор «:=» определяет **новую** переменную, вводит её в пространство имён задачи и обязательно инициализирует её указанным значением (в Go не бывает неинициализированных переменных!). Оператор «=», в отличие, выполняет присвоение значения (от простого типа до сколь угодно структурированного объекта) уже ранее существующей переменной, или только-что объявленной в var с явным указанием типа (что тоже означает уже существующую переменную к этому моменту):

```
var β, ω complex128 = 0.0 + 1i, math.Pi
```

Это позволяет, например, обменивать значения переменных в одном операторе:

```
i, j, k = j, k, i // Циклически обменять местами значения i, j и k
```

В Go не требуются круглые скобки вокруг условия в выражениях if, условий для выражения for или значения выражения в switch:

```

for θ := 0; θ < π; θ++ {
    if β == 0 {
        ...
    }
}

```

Но, с другой стороны, именно из-за этого требуется **обязательно** заключать в фигурные скобки **тело** выражений if, else и for, даже если это тело состоит всего из одного оператора:

```

if a < b {f()}           // Корректно
if(a < b) {f()}          // Корректно
if a < b f()             // НЕКОРРЕКТНО
for i = 0; i < 10; i++ {} // Корректно
for(i = 0; i < 10; i++) {} // НЕКОРРЕКТНО

```

Подобно циклу for, оператор if также может начинаться с короткого утверждения, выполняемого раньше проверки условия. Переменные, объявленные (:=) в таких утверждениях имеют область существования только до конца if оператора. Переменные, объявленные в ветке if, могут также использоваться и в ветке else:

```

func pow(x, n, lim float64) float64 {
    if v := math.Pow(x, n); v < lim {
        return v
    } else {
        fmt.Printf("%g >= %g\n", v, lim)
    }
    // can't use v here, though
    return lim
}

```

Циклы for в Go — это вообще совсем особая песня... В Go вообще нет никаких форм цикла кроме for: ни выражения while, ни выражения do { ... } while или do { ... } until (подобно другим языкам программирования). Любые мыслимые формы циклов выражаются только формами for!

Выражение `for` может быть использовано с одним условием, что делает его аналогичным выражению `while`:

```
sum := 1
for sum < 1000 {
    sum += sum
}
```

Если же условия вообще опущены, то будет создан бесконечный цикл, из которого выходим по оператору `break`:

```
for {
    ...
    if ... { break }
}
```

Для любых структурных регулярных объектов (массивы, срезы, хеши ...) существует очень популярная форма цикла, использующая итератор `range` (встроенное ключевое слово языка). Например (фрагмент из архива `types/array3`):

```
a3 := []int{2:9, 4:8, 7:7, 9:1}
for i, y := range a3 {
    fmt.Printf("%02d:%d ", i, y)
}
println()
...
```

```
$ go run array3.go
```

```
00:0 01:0 02:9 03:0 04:8 05:0 06:0 07:7 08:0 09:1
...
```

Примечание: Обратите внимание, что в объявлении `a3` инициализируется только некоторый выборочный набор элементов, с номерами: 2, 4, 7, 9 (индексация массивов, как и в других языках, начинается с 0). Тем не менее, инициализируются все элементы, общий размер определяется из максимального значения индекса, а все не указанные явно (пропущенные) элементы получают нулевое значение, 0 (для каждого типа в Go предполагается своё индивидуальное нулевое значение). В Go не бывает не инициализированных переменных!

То же самое для хеш-таблиц (пример `types/map4rand.go`):

map4rand.go

```
package main
```

```
import "fmt"
```

```
func main() {
    m := map[int]int{
        1: 19, 2: 18, 3: 17,
        4: 16, 5: 15, 6: 14,
    }
    fmt.Printf("%v\n", m)
    for i := 0; i < 5; i++ {
        for key, value := range m {
            fmt.Printf("%1d:%2d ", key, value)
        }
        println()
    }
}
```

Выполняется это так:

```
$ go run map4rand.go
```

```
map[1:19 2:18 3:17 4:16 5:15 6:14]
3:17 4:16 5:15 6:14 1:19 2:18
```

```
5:15 6:14 1:19 2:18 3:17 4:16
3:17 4:16 5:15 6:14 1:19 2:18
5:15 6:14 1:19 2:18 3:17 4:16
6:14 1:19 2:18 3:17 4:16 5:15
```

Обратите внимание (это очень важно!), что, начиная с некоторой версии GoLang (раньше это было не так), итерация в цикле по элементам таблицы производится **в случайном порядке**, который меняется не только от одного запуска приложения к другому, но даже при последовательных вызовах одного и того же фрагмента кода. Это сделано (после достаточно обстоятельных обсуждений) сознательно, для того чтобы программист-разработчик неявно не предполагал последовательный метод перебора элементов таблиц.

Начиная с версии 1.22 GoLang (декабрь 2023г.) разрешена в цикле итерация по целочисленному параметру (на манер Python), вот как это выглядит (в архиве это `compare/new.vers`):

for.go

```
package main

func main() {
    for i := range 10 {
        print(i, " ")
    }
    println()
}

$ go run for.go
0 1 2 3 4 5 6 7 8 9
```

Более того, разрешены (с версии 1.22) итерации по функциям-итераторам, определяемым в любом пользовательском коде. Как частный пример (но, как вы понимаете, широта охвата такого нововведения может быть много шире):

forit.go

```
package main
import "github.com/go-board/std/slices"

func main() {
    arr := []int{1, 2, 3, 4, 5}
    for x := range slices.Backward(arr) {
        print(x, " ")
    }
    println()
}
```

Но такая возможность (и ряд других) определена (пока) как экспериментальная, и компилироваться такое приложение должно с определением переменной окружения GoLang `GOEXPERIMENT` (в противном случае вы поймаете терминальную ошибку компиляции):

```
$ GOEXPERIMENT=rangefunc go build forit.go

$ ls -o forit
-rwxrwxr-x 1 olej 1792609 фев  9 20:46 forit
```

И в итоге:

```
$ ./forit
5 4 3 2 1
```

Поскольку роль цикла `for` в Go максимально перегружена, и цикл может оперировать данными разнообразных типов (и это не всегда как очевидно выразить в записи такого оператора) то приведём в завершение «сборный» пример (каталог `hello`):

for.go

```
package main

import (
    "fmt"
    "time"
)

func main() {
    sum := 1
    for sum < 1000 {                                // 1
        sum += sum
        print(sum, " ")
    }
    println()
    const limit = 100
    var sum1 uint64 = 0
    for i := uint64(0); i < 100; i++ { sum1++ }      // 2
    println("сумма =", sum1)
    sum1 = 0
    var sum2 rune = 0
    for i := 'a'; i < 'z'; i++ { sum2 += i }        // 3
    println("сумма =", sum2)
    for i := 'я'; i >= 'a'; i-- { sum2 += rune(i)}  //4
    println("сумма =", sum2)
    for i := 'я'; i >= 'a'; i-- { print(string(i)) } //5
    println()
    s := "строка"
    for i := 0; i < len(s); i++ {                    //6
        fmt.Printf("%q | ", s[i])
    }
    println()
    for i, c := range s {                             //7
        fmt.Printf("%d: %q | ", i, c)
    }
    println()
    ch := make(chan int)
    go func() {
        time.Sleep(300 * time.Millisecond)
        ch <- 1
        time.Sleep(200 * time.Millisecond)
        ch <- 2
        time.Sleep(100 * time.Millisecond)
        close(ch)
    }()
    fmt.Printf("before:\t\t%s\n", time.Now())
    for v := range ch {                                //8
        fmt.Printf("tick #%d:\t%s\n", v, time.Now())
    }
    fmt.Printf("after:\t\t%s\n", time.Now())
    ch = make(chan int)
    close(ch)
    for range ch {                                     //9
        fmt.Println("tick")
    }
    fmt.Println("done!")
}
```

Некоторые формы использования не совсем очевидные, но их поведение становится понятным из исполнения, и вряд ли требует дополнительных комментариев:

\$./for

2 4 8 16 32 64 128 256 512 1024

сумма = 100

```

сумма = 2725
сумма = 37525
яюэыьщцхфутсрпонмлкйизждгвба
'Ñ' | '\u0081' | 'Ñ' | '\u0082' | 'Ñ' | '\u0080' | 'Ð' | '¼' | 'Ð' | 'º' | 'Ð' | 'º' |
0: 'с' | 2: 'т' | 4: 'р' | 6: 'о' | 8: 'к' | 10: 'а' |
before:          2024-02-27 06:13:25.3020501 +0200 EET m=+0.000509455
tick #1: 2024-02-27 06:13:25.602680683 +0200 EET m=+0.301140038
tick #2: 2024-02-27 06:13:25.803148485 +0200 EET m=+0.501607827
after:          2024-02-27 06:13:25.90343262 +0200 EET m=+0.601891950
done!

```

Следующим в ряду рассмотрения важных операторов Go должен быть `switch`. В выражении `switch`, метки `case` не являются проходными (здесь имеет место радикальное отличие от C/C++, Python ... да и большинства других языков программирования — за этим нужно внимательно следить). Вы можете сделать их проходными с помощью ключевого слова `fallthrough`. Это применяется даже для смежных альтернатив:

```

switch i {
case 0: // пустое тело case, ничего не делать
case 1:
    f() // f не вызовется, когда i == 0!
}

```

Но:

```

switch i {
case 0:
    y = -1
    fallthrough
case 1:
    f() // f не вызовется с предварительным действием, когда i == 0!
}

```

(Такое, как показано выше, форматирование оператора `switch` использует родной форматор GoLang: `go fmt ...` — поэтому и мы постараемся придерживаться такого написания.)

Также в `case` может быть объявлено несколько значений.

```

switch i {
case 0, 1:
    f() // f будет вызвана если i == 0 || i == 1.
}

```

Значения переменной `i` в `case` не обязательно должны быть константами, или даже целыми числами. Любые виды типов и выражений, которые поддерживает оператор сравнения, — такие, как строки или указатели, — могут быть использованы. И если значение для `switch` вообще опущено, по умолчанию типом условия становится `true`, а значениями в ветках `case` становятся логические условия:

```

switch {
case i < 0:
    f1()
case i == 0:
    f2()
case i > 0:
    f3()
}

```

Оператор `switch` может использоваться и для динамической диагностики **типа** интерфейсной переменной (run-time рефлексия). Такой `type switch` использует синтаксис `type` утверждения типа с помощью ключевого слова `type` внутри скобок (и это

единственное использование такого ключевого слова). Если switch объявляет переменную в выражении, то эта переменная будет иметь соответствующий свой тип в каждой альтернативе. Если использовать это имя в ветвях выбора case, то эффектом будет объявление новой переменной с тем же именем, но с разным типом в каждом конкретном случае:

```
var t interface{}
t = functionOfSomeType()
switch t := t.(type) {
default:
    fmt.Printf("unexpected type %T", t)          // %T prints whatever type t has
case bool:
    fmt.Printf("boolean %t\n", t)                // t has type bool
case int:
    fmt.Printf("integer %d\n", t)                // t has type int
case *bool:
    fmt.Printf("pointer to boolean %t\n", *t)    // t has type *bool
case *int:
    fmt.Printf("pointer to integer %d\n", *t)    // t has type *int
}
```

Достаточно часто в программном коде Go (в виду распространённости и лёгкости реализации конкурентности в этом языке) можно видеть характерные конструкции подобные следующему:

```
select {
case <-ctx.Done():
    return
default:
    ...
}
```

Это означает проверку того, что некоторая параллельная ветвь закрыла канал ctx.Done() как сигнал окончания некоторого своего действия. В этой, часто используемой конструкции взаимодействия, канал вообще не используется для передачи сообщения (прямое назначение канала), но только для фиксации факта его закрытия, как признака завершения некоторого действия в параллельной ветви.

Следующая предмет наших интересов — это особенности и приоритеты операций Go в выражениях...

Постфиксные операторы ++ и -- могут быть в Go вообще использованы только в утверждениях, но никак не в выражениях. Вы не можете написать c = *p++. Выражение *p++ воспринимается, как (*p)++, а не как *(p++). Префиксных операций ++p и --p в языке Go вообще не существует.

Приоритеты операций и правила ассоциативности в Go и C++ различаются! Как итог, вычисление одного и того же выражения 7 & 3 << 1 — будет давать 6 в Go и 4 в C++.

В Go приоритеты операций (показаны численные значения приоритетов):

1. * / % << >> & & ^
2. + - | ^
3. == != < <= > >=
4. &&
5. ||

В C++ приоритеты операций (показаны только релевантные операции, имеющие эквиваленты в Go):

1. * / %
2. + -
3. << >>
4. < <= > >=
5. == !=
6. &

7. ^
8. |
9. &&
10. ||

Как и везде: во всех случаях, когда у вас возникают сомнения в порядке вычислений выражений — вы можете явно задать требуемый порядок вычислений использованием скобок.

Функции

Функции объявляются при помощи ключевого слова `func`. После (списка) параметров в скобках указываются типы возвращаемых значений. В случае с одним возвращаемым значением скобки не используются. Параметры функций и методов, и возвращаемые ими результаты объявляются таким образом:

```
func f(i, j, k int, s, t string) string { ... }
```

Однотипные параметры, следующие друг за другом, могут объявляться списком (как показано выше), но могут объявляться и индивидуально (что не типично для стиля Go):

```
func f(i int, j int, k int, s string, t string) string { ... }
```

Поскольку функция рассматривается Go как и любой объект данных, то функция может объявляться присвоением (функциональной) именованной переменной тела функции. Позже по этому имени будет вызываться функция, или передаваться в вызов других функций:

```
newfunc = func (n int) string { ... }
...
s := newfunc(1000)
...
other(newfunc)
...
```

Функции могут возвращать несколько (два или более) значений, типы таких множественных значений в определении функции заключаются в скобки:

```
func f(a, b int) (int, string) {
    return a + b, "сложение"
}
```

Подобный подход устраняет необходимость передавать указатель на возвращаемое значение, чтобы имитировать ссылочный параметра (в C++ это решается объявлением аргумента-ссылки: `func(int& x)`). Вот простая функция, которая ищет цифровое значение, начиная с указанной позиции в байтовом срезе, и возвращает значение многозначного преобразованного числа, и следующую за числом позицию в срезе (пример из документации):

```
func nextInt(b []byte, i int) (int, int) {
    for ; i < len(b) && !isDigit(b[i]); i++ {}
    x := 0
    for ; i < len(b) && isDigit(b[i]); i++ {
        x = x * 10 + int(b[i]) - '0'
    }
    return x, i
}
```

Такую функцию можно использовать для сканирования числовых значений во входном срезе `b` подобно следующему:

```
for i := 0; i < len(b); {
    x, i = nextInt(b, i)
    fmt.Println(x)
}
```

```
}
```

Если несколько значений, возвращаемых функцией, должны присваиваться переменным, то их перечисляем в присвоении через запятую:

```
first, second := incTwo(1, 2) // first = 2, second = 3
```

Если какое-то из множества возвращаемых значений не представляет интереса в контексте конкретного вызова функции (игнорируется), то для этой переменной в списке присвоения используется специальное имя «_» (символ подчёркивания):

```
длина, _ := os.Stdin.Read(буфер) // 2-е значения (ошибка) игнорируется
```

Подобные выражения часто приходится видеть в реальном коде, поскольку 1). мы не можем для функции возвращающей N значений принять только N-1, и 2). мы не можем присвоить не нужное нам значение некоторой именованной переменной, которая дальше никак не будет использована: не используемая переменная в Go квалифицируется как терминальная ошибка.

В присвоении может использоваться больше одной таких не используемых переменных, например, только для проверки потенциальной возможности (безошибочности) выполнения операции:

```
_, _ := os.Stdin.Read(буфер) // оба значения игнорируются
```

В Go нет возбуждаемых исключений. Множественные возвращаемые функцией значения являются в Go базовым механизмом для реакции на ошибки вызова:

```
func MySqrt(f float) (v float, ok bool) {
    if f >= 0 { v, ok = math.Sqrt(f), true }
    else { v, ok = 0, false }
    return v, ok
}
...
result, ok := MySqrt(...)
if !ok {
    // Something bad happened.
    return nil
}
// Continue as normal.
...
```

Результаты возвращаемые функцией (хоть одиночный, хоть множественные), также как и входные аргументы, могут быть именованными, тогда нет необходимости (но можно) их указывать при возврате результатов из функции:

```
func incTwo(a, b int) (c, d int) {
    c = a + 1
    d = b + 1
    return
}
```

Как и во всех языках семейства C, всё в Go передается по значению. То есть, функция всегда получает копию того, что передавалось, так как если бы оператор присвоения присваивал значение параметру перед вызовом. Например, при передаче значение `int` в функцию делается копия `int`, а при передаче указателя делает копию указателя, но не данных, на которые он указывает.

Таблицы и срезы ведут себя подобно указателям: они являются дескрипторами, которые содержат указатели на базовую таблицу или срез. Копирование объекта типа таблицы или среза не копирует данные, на которые они указывают. Копирование объекта типа интерфейса делает копию всего того, что загружено в интерфейс. Если интерфейсный объект содержит структуру, то копируя объект интерфейса — делаете копию структуры. Если интерфейсный объект содержит указатель, то копируя значение интерфейса —

делаете копию указателя, но опять-таки, не данных, на которые он указывает.

Функция является таким же объектом как и любые другие данные, она может присваиваться другим именованным переменным:

func1.go

```
package main

import (
    "fmt"
    "math"
)

func main() {
    hypot := func(x, y float64) float64 {
        return math.Sqrt(x*x + y*y)
    }
    fmt.Println(hypot(3, 4))
}

$ ./func1
5
```

Этот пример попутно показывает, что в Go функции могут быть на произвольную глубину вложены друг в друга в объявлениях (не видимы за пределами обрамляющей функции)¹⁵. Это очень частая практика при параллельном запуске сопрограмм (оператор `go`), когда функция определяется локально в момент её запуска.

Более того, функция может быть создана даже без имени, анонимно, выполнена и тут же утилизирована сборкой мусора¹⁶:

func2.go

```
package main

import "fmt"

func main() {
    sum := func(a, b int) int { return a + b }(3, 4)
    fmt.Println(sum)
}

$ ./func2
7
```

Или даже так:

```
func main() {
    fmt.Println(func(a, b int) int { return a + b }(3, 4))
}
```

Поскольку функция рассматривается Go как объект данных, язык позволяет реализовать многие из приёмов функционального программирования (хотя анонимные функции — это уже сам по себе трюк из функционального программирования). Один из таких приёмов, из области функций высших порядков (выше первого, явно определяемого), является функциональное замыкание, или просто замыкание (closure). Замыкание — это функциональный объект, который ссылается к переменным вне тела самих этих функций (в этом смысле функция «привязана» к переменным).

Примечание: Дэвид Мертц приводит следующее определение замыкания: "Замыкание - это процедура вместе с привязанной к ней совокупностью данных" (в противовес объектам в объектном программировании, которые по его же словам: "данные вместе с привязанным к ним совокупностью процедур").

¹⁵ Это обычная практика для языков Pascal, Modula-2, или Oberon, но такое не допустимо в стандарте языков C и C++.

¹⁶ То что определяется как lambda-выражение в языках функционального программирования.

func3.go

```
package main

import "fmt"

func adder() func(int) int {
    sum := 0
    return func(x int) int {
        sum += x
        return sum
    }
}

func main() {
    pos, neg := adder(), adder()
    for i := 0; i < 10; i++ {
        fmt.Println(pos(i), neg(-2 * i))
    }
}
```

В этом примере функция `adder()` возвращает замыкание (возвращает **функцию!**). Используя это замыкание функциональные переменные `pos()` и `neg()` работают каждый со своим экземпляром накапливающей переменной `sum`:

```
$ ./func3
0 0
1 -2
3 -6
6 -12
10 -20
15 -30
21 -42
28 -56
36 -72
45 -90
```

Вариативные функции

Вариативная функция – это функция, которая в качестве единого аргумента принимает ноль, одно или несколько значений¹⁷. Хотя вариативные функции используются не так часто, в отдельных случаях они могут сделать код чище и читабельнее.

Функция с параметром, типу которого **предшествует** многоточие (... — или можно так считать, что за именем параметра следует многоточие), считается вариативной функцией. Многоточие означает, что предоставленный параметр может принимать ноль, одно или несколько значений.

В функции может быть только один вариативный параметр – он обязательно должен быть последним параметром, из определенных в функции. Определение параметров в вариативной функции без учета порядка приведет к ошибке компиляции.

Функция Go может принимать сразу список аргументов определённого типа (во многих случаях это эквивалентно по возможностям функциям с переменным числом параметров в C, или возможности C++ определения параметров вызова со значениями по умолчанию, которые могут опускаться при вызове). Вот синтаксический пример такой записи (все примеры этой главы находятся в каталоге `function`):

arglist.go

```
package main

func Min(a ...int) int {
```

¹⁷ В некоторых языках программирования это называют «функция с переменным числом параметров».

```

    min := int(^uint(0) >> 1) // largest int
    for _, i := range a {
        print(i, " ")
        if i < min {
            min = i
        }
    }
    print(" => ", min, "\n")
    return min
}

func main() {
    Min(-11)
    Min(11, 7, 3)
    Min(-1, 2, -3, 4, -5, 6)
}

$ ./arglist
-11 => -11
11 7 3 => 3
-1 2 -3 4 -5 6 => -5

```

Более того, можно указать, что функция может принять произвольное число аргументов произвольного типа (arbitrary type). А затем уже внутри такой функции динамически определить последовательно тип каждого из параметров вызова, и произвести адекватные действия. Вот как подобным образом определяется функция `Printf()` в пакете `fmt`:

```
func Printf(format string, v ...interface{}) (n int, err error)
```

Интерфейсному типу `interface{}` может быть присвоено любое значение. В теле функции `Printf()` переменная `v` действует как переменная типа `[]interface{}`, но если её нужно дальше передать следующей другой функции с переменным числом аргументов (variadic), то эта переменная выступает как обычный список аргументов. Вот возможная реализация вашей функции `xxx.Println()`. Она передает свои аргументы непосредственно в `fmt.Sprintln()` на фактическое форматирование:.

```

// Println осуществляет вывод на стандартный регистратор в манере fmt.Println.
func Println(v ...interface{}) {
    std.Output(2, fmt.Sprintln(v...)) // Output принимает параметры (int, string)
}

```

Мы пишем здесь `...` после параметра `v` во вложенном вызове `fmt.Sprintln()`, чтобы сообщить компилятору, что нужно рассматривать `v` как **список** аргументов. Иначе `v` должен рассматриваться как единичный параметр типа среза. (именно поэтому мы в примере выше применяли к вариативному параметру оператор `range`).

Чтобы не усложнять объяснения сложным значащим примером, я приведу здесь пример, заимствованный из одной из публикаций:

typelist.go

```

package main

import (
    "fmt"
    "reflect"
)

func main() {
    variadicExample(1, "red", true, 10.5, []string{"foo", "bar", "baz"},
        map[string]int{"apple": 23, "tomato": 13})
}

```

```
func variadicExample(i ...interface{}) {
    for _, v := range i {
        fmt.Println(v, "--", reflect.ValueOf(v).Kind())
    }
}

$ ./typelist
1 -- int
red -- string
true -- bool
10.5 -- float64
[foo bar baz] -- slice
map[apple:23 tomato:13] -- map
```

Здесь в коде всё ясно без объяснений. Но этот фрагмент приведен для того, чтобы он мог быть использован в качестве образца того, как а). посредством пакета `reflect` (рефлексия периода выполнения) диагностировать **тип** переменной и б). затем использовать этот тип как переключатель ветвей `switch`, как это показывалось немногим ранее.

Стек процедур завершения

Еще одна возможность Go, стоящая особняком, но непосредственно связанная с вызовами некоторых функций — это оператор `defer`:

```
defer foo();
```

Здесь конструкция `defer` регистрирует функцию завершения, и функция `foo()` будет вызвана по достижении `return` в вызывающей единице (это может быть и главная функция `main()` или любая другая функция). Если `defer` используется в функции несколько раз, то соответствующие методы выстраиваются в стек и будут выполняться при завершении функции в порядке, обратном их помещению.

Утверждение `defer` может быть использовано для указания финальных действий, которые нужно выполнить при завершении вызывающей единицы кода (функции, главной программы):

```
fd := open("filename")
defer close(fd)           // fd будет закрыт после завершения функции
```

Некоторые авторы считают, что польза от подобного нововведения не очевидна, особенно при большом объеме кода. Так программисту, незнакому с кодом, придется держать в памяти весь этот стек вызовов со всеми параметрами. При использовании `defer` структурное программирование не обеспечивается.

С другой стороны (противоположное мнение), это механизм, непосредственно повторяющий логику стека процедур завершения потока `pthread_t` из стандарта POSIX 1003.b: `pthread_cleanup_push()` и `pthread_cleanup_pop()`. И стандарт рекомендует к использованию именно эти механизмы. При нескольких `return` в теле функции `defer` будет выполняться по любому из них, и тем самым гарантируется единообразие действий завершения. Как завершающее действие в `defer` должен обязательно указываться вызов функции (это не может быть оператор или последовательность операторов). Но это может быть и непосредственный фрагмент кода, оформленный как анонимная функция (мы это уже умеем), по типу такого:

```
defer func() { ... } ()
```

Аргументы отложенной функции (в `defer`) вычисляются когда **определяется** `defer`, а не когда **вызывается** функция завершения. В документации приводится такой любопытный пример:

```
for i := 0; i < 5; i++ {
    defer fmt.Printf("%d ", i)
```

```
}
```

Откладывание функции в LIFO очередь (в каждом прохождении цикла), приведет к следующей работе функции при завершении фрагмента кода и печати на экран: 4 3 2 1 0.

Обобщённые функции

Достаточно часто нужно создать «видовую» функцию (generic), которая выполняла бы единообразные действия, но над данными разного типа. Простейшим примером такой функции может быть `Minimum()` — поиск минимального значения среди полученных параметров вызова. Но для различных типов данных смысл сравнения ($>$, $<$) должен быть различным. В C++ для таких целей используют шаблоны (template) и шаблонные функции, параметризуемые типом параметров, а компилятор сгенерирует все необходимые версии функции (то есть по одной для каждого используемого типа).

В языке Go не поддерживалась параметризация по типу параметров¹⁸, поэтому, чтобы добиться того же эффекта, требуется вручную создать все необходимые функции (например, `MinimumInt()`, `MinimumFloat()`, `MinimumString()`). Но это оказывается громоздко, особенно не столько на этапе написания, сколько при использовании таких функций.

Язык Go предлагает несколько альтернативных подходов, позволяющих избежать необходимости создавать функции, отличающиеся только типами данных, которыми они оперируют, хотя и за счет некоторой потери эффективности во время выполнения. Ниже приводится пример использования обобщенной функции `Minimum()`:

`minimum.go`

```
package main

import (
    "fmt"
)

func Minimum(first interface{}, rest ...interface{}) interface{} {
    minimum := first
    for _, x := range rest {
        switch x := x.(type) {
        case int:
            if x < minimum.(int) {
                minimum = x
            }
        case float64:
            if x < minimum.(float64) {
                minimum = x
            }
        case string:
            if x < minimum.(string) {
                minimum = x
            }
        }
    }
    return minimum
}

func main() {
    i := Minimum(4, 3, 8, 2, 9).(int)
    fmt.Printf("%T : %v\n", i, i)
    f := Minimum(9.4, -5.4, 3.8, 17.0, -3.1, 0.0).(float64)
    fmt.Printf("%T : %v\n", f, f)
    s := Minimum("K", "X", "B", "C", "CC", "CA", "D", "M").(string)
```

¹⁸ До определённого времени, до версии 1.18, до введения в Go дженериков ... но об этом поговорим отдельно.

```

    fmt.Printf("%T : %q\n", s, s)
}

```

Здесь использовано то обстоятельство, что интерфейсному типу `interface{}` может быть присвоено любое значение:

```

$ ./minimum
int : 2
float64 : -5.4
string : "B"

```

Функции высших порядков

*Альпинист защищает крутизну и сложность подъема. Компьютерный энтузиаст защищает непрозрачность и затрудненность взаимодействия с программным обеспечением.
Алан Купер, «Психбольница в руках пациентов»*

Функцией высшего порядка называется функция, принимающая в аргументах одну или более внешних функций и использующая их в своём теле:

index.go

```

package main

import (
    "fmt"
)

func SliceIndex(limit int, predicate func(i int) bool) int {
    for i := 0; i < limit; i++ {
        if predicate(i) {
            return i
        }
    }
    return -1
}

func main() {
    xs := []int{2, 4, 6, 8, 10}
    ys := []string{"f", "fd", "B", "вы", "выб", "выбор"}
    fmt.Println(
        SliceIndex(len(xs), func(i int) bool { return xs[i] == 7 }),
        SliceIndex(len(xs), func(i int) bool { return xs[i] == 8 }),
        SliceIndex(len(ys), func(i int) bool { return ys[i] == "z" }),
        SliceIndex(len(ys), func(i int) bool { return ys[i] == "выб" }))
}

```

Здесь анонимные функции предикаты, передаваемые функции `SliceIndex()` в качестве второго параметра, являются замыканиями, поэтому срезы, на которые они ссылаются (`xs` и `ys`), должны находиться в области видимости там, где создаются эти функции:

```

$ ./index
-1 3 -1 4

```

В этом примере функция высшего порядка `SliceIndex()` является универсальной функцией, которая вообще не имеет дело со срезами, и даже «не знает» ничего о срезах:

mindex .go

```

package main

```

```

import "math"

func SliceIndex(limit int, predicate func(i int) bool) int {
    for i := 0; i < limit; i++ {
        if predicate(i) {
            return i
        }
    }
    return -1
}

func main() {
    print(SliceIndex(math.MaxInt32,
        func(i int) bool {
            return i != 0 && i%27 == 0 && i%51 == 0
        }),
        "\n")
}

```

В таком варианте Функция `SliceIndex()` выполняет итерации по натуральным числам от 0 до максимально возможного значения, и на каждой итерации вызывает анонимную функцию (предикат) для натурального числа. Функция предикат прерывает поток итераций при нахождении наименьшего натурального числа, кратного числам 27 и 51:

```

$ ./mindex
459

```

Выше показан поиск по не отсортированным срезам. Часто бывает необходимо фильтровать их, отбрасывая элементы, не удовлетворяющие некоторому условию. Ниже приводится простой пример функции, здесь функция `Filter()` «не знает» **типа данных**, составляющих фильтруемый срез:

filter.go

```

package main

import (
    "fmt"
)

func Filter(limit int, predicate func(int) bool, appender func(int)) {
    for i := 0; i < limit; i++ {
        if predicate(i) {
            appender(i)
        }
    }
}

func main() {
    data := []int{4, -3, 2, -7, 8, 19, -11, 7, 18, -6}
    even := make([]int, 0, len(data))
    Filter(len(data),
        func(i int) bool { return data[i]%2 == 0 },
        func(i int) { even = append(even, data[i]) })
    fmt.Println(even)
}

$ ./filter
[4 2 8 18 -6]

```

С одинаковым успехом функция `Filter()` может применяться к вещественной последовательности, или текстовой последовательности слов.

Встроенные функции

В Go существует достаточно обширный набор **встроенных** функций, не требующих определений и импортирования пакетов их содержащих. Вот их перечень (он может меняться в зависимости от версии):

```
func append(slice []Type, elems ...Type) []Type
func cap(v Type) int
func close(c chan<- Type)
func complex(r, i FloatType) ComplexType
func copy(dst, src []Type) int
func delete(m map[Type]Type1, key Type)
func imag(c ComplexType) FloatType
func len(v Type) int
func make(Type, size IntegerType) Type
func new(Type) *Type
func panic(v interface{})
func print(args ...Type)
func println(args ...Type)
func real(c ComplexType) FloatType
func recover() interface{}
```

Большинство из этих встроенных функций уже встречались неоднократно в примерах, или будут ещё разобраны детально далее. Есть смысл остановиться только на некоторых деталях функции `append()`:

append.go

```
package main

import "fmt"

func main() {
    x := []int{1, 2, 3}
    println(x)
    fmt.Println(x)
    x = append(x, 4, 5, 6)
    println(x)
    fmt.Println(x)
    y := []int{7, 8, 9}
    x = append(x, y...)
    println(x)
    fmt.Println(x)
}
```

Обратим внимание на то, как записан второй параметр вызова (с «...») когда этим параметром является срез элементов, а не список отдельных элементов (как в 1-м вызове), без этого ... компилятор напишет ошибку, так как у не имеет тип `int`.

```
$ ./append
[3/3]0xc000018400
[1 2 3]
[6/6]0xc00001a240
[1 2 3 4 5 6]
[9/12]0xc0000cc060
[1 2 3 4 5 6 7 8 9]
```

Попутно показано как встроенные функции `print()` и `println()` выводят **срез**: в порядке напоминания того, что срез в любую функцию в качестве параметра передаётся **по ссылке** (как массивы в языке C). А вот **массив** в качестве параметра будет передан по значению (копированием), и будет показан `print()` и `println()` поэлементно.

Функция `append()` — **встроенная**. Формально её сигнатура должна бы выглядеть так:

```
func append(slice []*T, elements ...*T) []*T
```

Где T любой тип. Но мы не можем написать в языке Go функцию в которой T определена вызывающей стороной. Поэтому это обеспечивается поддержкой компилятора индивидуально для функции `append()`.

Причина того, что `append()` по сигнатуре должна возвращать результат состоит в том, что (как это и происходит в примере выше) базовый массив среза во время вызова может измениться (получить новое размещение, адрес).

Детальное описание всех встроенных функций вы найдёте в пакете `builtin`. Полный перечень встроенных функций с их краткой аннотацией можно получить из описания этого пакета: <https://pkg.go.dev/builtin>.

Объектно ориентированное программирование

*Я изобрел понятие «объектно-ориентированный»,
и могу заявить, что не имел в виду C++.
Alan Kay*

На вопрос: «является ли Go объектно-ориентированным языком?», сами авторы Go отвечают в документации: «И да и нет». Авторы Go говорят так:

Объектно-ориентированное программирование, по крайней мере в известных языках, включает в себя слишком много разговоров на тему взаимоотношений между типами, взаимоотношений, которые часто могут быть выведены автоматически. Go использует другой подход.

Вместо того, чтобы требовать от программиста объявлять заранее, что два типа связаны, в Go тип автоматически удовлетворяет любому интерфейсу, который специфицирует подмножество его методов. Помимо простого рутинного учёта, такой подход имеет реальные преимущества. Типы могут удовлетворить многим интерфейсам одновременно, без сложностей традиционного множественного наследования. Интерфейсы могут быть очень легковесными — интерфейсом с одним единственным методом, или даже вообще без методов, можно выразить весьма полезные концепции. Интерфейсы могут быть добавлены уже после того, когда новая идея приходит после, или во время тестирования, без существенного изменения оригинальных типов. Потому что нет отчётливой связи между типами и интерфейсами, нет выраженной иерархии типов, которой нужно управлять или которую обсуждать.

Это подобно использованию этих идей для создания чего-то аналогичного типабезопасных UNIX pipes. Например, посмотрите как `fmt.Fprintf()` позволяет форматировать печать на любой вывод, а не просто в файл, или как пакет `bufio` может быть полностью отделен от файла в вводе-выводе, или как пакеты обработки изображений позволяют генерировать сжатые файлы изображений. Все эти возможности происходят из одного интерфейса (`io.Writer`), представляющего единственный метод (`Write()`). И это только самые поверхностные примеры. Интерфейсы Go имеют самое глубокое влияние на то, как структурированы программы.

Это требует некоторого привыкания, но этот неявный стиль зависимостей типов является одним из наиболее продуктивных сторон Go.

А теперь попробуем перевести всё это на нормальный человеческий язык, иллюстрируя и подтверждая сказанное примерами кода...

Авторы Go в объяснениях тщательно избегают терминов класс и объект. Понятие класса и ключевое слово `class` в Go отсутствует, для любого именованного типа (включая структуры и даже синонимы базовых типов, вроде `int`) можно определить методы работы с ним. Ключевое слово `type` вводит определение нового **типа**, который и

является некоторым эквивалентом определения нового класса:

```
type newInt int
```

Как заместитель понятию «объект» (класса) авторы повсеместно используют термин значение (типа). В русскоязычном контексте более уместным кажется термин переменная (типа). Мы будем использовать оба термина как эквиваленты.

Объектно-ориентированная архитектура Go весьма своеобразна и с большими и совсем не очевидными возможностями. Но она радикально отличается от той привычной модели, которая используется в C++ или Java. Ниже описываются только базовые принципы этой техники, детальней она будет проиллюстрирована примерами приложений в конце книги.

Методы

В Go для **любого** типа, описываемого в текущем пакете, могут быть определены **методы**. Определение метода отличается от автономного определения функции только тем, что отдельным полем в определении указывается **получатель** (receiver). Это похоже на указатель `this` в методе класса C++, передаваемый первым скрытым параметром метода. Но в Go это может быть разнообразнее:

```
type myType struct {i int}
func (p *myType) get() int { return p.i }
func (p *myType) set(i int) { p.i = i }
```

Или так:

```
func (p myType) get() int { return p.i }
    func (p myType) set(i int) { p.i = i }
```

Здесь запись в определении функций `(p *myType)` или `(p myType)` и указывает получателя.

Фактически, вы можете определить методы для **любого** типа определяемого в вашем пакете, не только структуры. Не могут быть определены методы для типов из других пакетов, или для базовых типов. Но вы можете определять методы для **синонима** базового типа (все примеры кода относительно объектно-ориентированного программирования сведены в архиве примеров в каталог `oop`):

object1.go

```
package main

import (
    "fmt"
    "math"
)

type MyFloat float64

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}

func main() {
    f := MyFloat(-math.Sqrt2)
    fmt.Println(f.Abs())
}
```

```
$ ./object1
1.4142135623730951
```

В предыдущем примере получателем указывался объект типа (MyFloat) для которого реализуется метод. В качестве получателя может указываться не только сам тип, но и указатель на этот тип (это как-раз вариант более близкий к C++):

object2.go

```
package main

import (
    "fmt"
    "math"
)

type Vertex struct {
    X, Y float64
}

func (v *Vertex) Scale(f float64) {
    v.X = v.X * f
    v.Y = v.Y * f
}

func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

func main() {
    v := &Vertex{3, 4}
    v.Scale(5)
    fmt.Println(v, v.Abs())
}

$ ./object2
&{15 20} 25
```

Существует два резона использовать указателей на получателя (передачу получателя по ссылке). Первый состоит в предотвращении копирования структуры получателя на каждый вызов метода, что может быть неэффективно для крупных структур. Второй проявляется в том случае, когда методу необходимо модифицировать значение, которое он принимает указателем (передача по значению, копированием, препятствует этому).

В примере выше метод Scale() модифицирует состояние объекта. А методу Abs() требуется доступ только по чтению, и передача ему указателя может оказаться избыточной или опасной (в смысле побочных эффектов). Обратите внимание на то, что варианты альтернативных реализаций func (v Vertex) Abs() float64 {...} и func (v *Vertex) Abs() float64 {...} будут иметь текстуально идентичную запись кода реализации, потому что и обращение к полю структуры, и разыменование указателя записывается в Go одинаково точечной нотацией (это хорошо видно в записи v.X и v.X в реализациях двух методов Scale() и Abs(), где они имеют принципиально различный смысл).

Множество методов

Множество методов типа – это множество всех методов, которые могут быть вызваны относительно значения этого типа.

Множество методов указателя на значение пользовательского типа включает в себя все методы этого типа, независимо от того, принимают ли они значение или указатель на него. Если относительно указателя вызвать метод, принимающий значение, компилятор

Go автоматически разыменует указатель и в качестве приемника передаст методу значение. Мы вполне можем определить:

```
func (v Vertex) Abs() float64 {
    return math.Sqrt(v.X * v.X + v.Y * v.Y)
}

func main() {
    v := &Vertex{3, 4}
    r := v.Abs()
    ...
}
```

Множество методов значения пользовательского типа включает в себя все методы этого типа, принимающие приемник по значению, — методы, принимающие указатель, не входят в это множество. Однако это не такое большое ограничение, так как для вызова метода, принимающего указатель на приемник, достаточно просто вызвать этот метод относительно адресуемого значения (то есть относительно переменной, разыменованного указателя, элемента массива или среза, или адресуемого поля структуры: запись `value.Method()`, где `Method()` требует указатель на приемник, а `value` — адресуемое значение, компилятор Go будет интерпретировать как `(&value).Method()`).

Встраивание и агрегирование

Мы уже употребляли эти термины при рассмотрении структур ранее. В Go, как уже сказано, нет наследования типов. Но новый тип может содержать в себе поля определённых ранее типов. Причём сделано это может быть двумя разными способами:

1. Агрегирование, когда содержащиеся в структуре поля именованы. Это полностью напоминает структуры из других языков программирования:

```
import "math/cmplx"
type point struct {
    xy complex128 /* агрегирование */
}
var p point
p.xy = complex(3, 3)
r, θ := cmplx.Polar(p.xy)
```

Это традиционно, привычно, как и в других языках программирования. Обращение к отдельным агрегированным полям происходит по имени поля, например: `p.xy`. Имена всех агрегированных полей должны, естественно, быть уникальными, различаться.

2. Встраивание, когда содержащиеся в структуре поля не именованы. Тогда все методы встроенного типа применимы **непосредственно** к новому определяемому типу (охватываемому). Например:

```
import "math"
type FuzzyBool struct { float32 } /* встраивание */
var fb FuzzyBool
θ := math.Sin(fb)
```

Обращение к встроенным полям, если это необходимо, происходит по имени их типа, например: `p.float32`. Естественно, что встроенные типы должны различаться по именованию, поэтому недопустимо определить одновременно поля `float32` и `*float32`.

Объемлющий тип может переопределить (с тем же именем) метод, определённый во встраиваемом типе. Как это происходит показано на примере:

object5.go

```
package main

import "fmt"
```

```

type Item struct {
    id      string // агрегирование
    price   float64 // агрегирование
    quantity int    // агрегирование
}

func (item *Item) Cost() float64 {
    return item.price * float64(item.quantity)
}

type SpecialItem struct {
    Item          // анонимное поле - встраивание
    catalogId int // именованное поле - агрегирование
}

type LuxuryItem struct {
    Item          // анонимное поле - встраивание
    markup float64 // именованное поле - агрегирование
}

func (item *LuxuryItem) Cost() float64 { // переопределение в производном типе
    return item.Item.Cost() * item.markup
}

func main() {
    special := SpecialItem{Item{"Green", 3, 5}, 207}
    fmt.Println(special, " => ", special.Cost())
    luxury := LuxuryItem{Item{"Green", 3, 5}, 2}
    fmt.Println(luxury, " => ", luxury.Cost())
}

```

Типы `SpecialItem` и `LuxuryItem` — производные от типа `Item` (имеют встроенным поле такого типа).

Вызов `special.Cost()` компилятор Go будет интерпретировать как вызов метода `Item.Cost()`, потому что тип `SpecialItem` не имеет собственного метода `Cost()`, и передаст ему встроенное значение типа `Item`, а не всё значение типа `SpecialItem`, относительно которого метод вызывался первоначально.

А вот вызов `luxury.Cost()` будет вызывать переопределённый в типе `LuxuryItem` его собственный метод `Cost()`. Если имя какого-либо поля или метода во встроенном типе `Item` совпадает с именем поля в производном типе `SpecialItem` или `LuxuryItem`, обратиться к полю или методу во встроенном типе `Item` тоже можно, указав имя этого типа как часть полного имени поля, например `special.Item.price`. Этим и пользуется реализация метода `LuxuryItem.Cost()`, вызывая одноимённый (переопределяемый) метод встроенного типа: `item.Item.Cost()`.

В итоге:

```

$ ./object5
{{Green 3 5} 207} => 15
{{Green 3 5} 2}  => 30

```

Механизм встраивания, дополненный возможностями интерфейсов, обеспечивает полную функциональность наследования классов в C++ или Java.

Функции как объекты

Из того, что функции Go это производные типы объектов первого класса (точно так же как элементарные типы данных) следует очень интересное свойство (вряд ли представленное в других языках): к функции как к типу можно определить методы. И чем это объяснять, это проще показать на предельно упрощённом примере:

objfun.go

```
package main

import "fmt"

type Multiply func(...int) string

func (f Multiply) Apply(i int) Multiply {
    return func(values ...int) string {
        values = append([]int{i}, values...)
        return f(values...)
    }
}

func main() {
    var multiply Multiply = func(values ...int) string {
        var total int = 1
        for _, value := range values {
            total *= value
        }
        return fmt.Sprintf("%v => %d", values, total)
    }

    fmt.Println("multiply:", multiply(3, 4), "(expect 12)")
    var times2 Multiply = multiply.Apply(2)
    fmt.Println("times 2:", times2(3, 4), "(expect 24)")
    // ... и можно даже каскадно применять к производным от Multiply объектам
    times6 := times2.Apply(3)
    fmt.Println("times 6:", times6(2, 3, 5, 10), "(expect 1800)")
}
```

В этом примере метод возвращает функцию, внутри которой вызывается функция базового типа `Multiply` (в данном случае `Apply()` трансформирует параметры передаваемые вызову, а затем в конце передаёт их на выполнение оригинальному `Multiply`, но это не обязательно и взаимодействие может быть самым замысловатым):

```
$ ./objfun
multiply: [3 4] => 12 (expect 12)
times 2: [2 3 4] => 24 (expect 24)
times 6: [2 3 2 3 5 10] => 1800 (expect 1800)
```

Этот пример условный и упрощён. Но за ним стоит возможность построить целое семейство родственных функций с модификацией поведения.

Интерфейсы

Там где в некоторых традиционных объектно ориентированных языках используются классы, в Go задействованы интерфейсы. Интерфейсы Go, по своей сути, похожи на интерфейсы в языке Java. При объявлении интерфейса в нем объявляется набор методов, и все (любые) типы, реализующие этот интерфейс, совместимы с переменной этого интерфейса. Интерфейсы также похожи и на абстрактные классы C++.

В Go каждый тип, предоставляющий все методы, обозначенные в интерфейсе, может трактоваться как реализация интерфейса, никакого явного объявления для этого не требуется.

```
type myInterface interface {
    get() int
    set(i int)
}
```

Объявленный в предыдущем обсуждении (глава «Методы» выше) тип `myType` также реализует интерфейс `myInterface`, хотя это нигде и не указано явно (сам тип `myType` об

109

этом «не знает», и когда мы там писали пример с типом `myType`, мы сами ещё не подозревали о существовании интерфейса `myInterface`).

Интерфейс к конкретному типу определяет для него набор методов. Переменная интерфейсного типа может принимать любое значение (переменную, объект), которое реализует эти методы:

object3.go

```
package main

import (
    "fmt"
    "math"
)

type Abser interface {
    Abs() float64
}

func main() {
    var a Abser
    f := MyFloat(-math.Sqrt2)
    v := Vertex{3, 4}
    a = f // a MyFloat implements Abser
    fmt.Println(a.Abs())
    a = &v // a *Vertex implements Abser
    fmt.Println(a.Abs())
    // In the following line, v is a Vertex (not *Vertex)
    // and does NOT implement Abser.
    // a = v
}

type MyFloat float64

func (f MyFloat) Abs() float64 {
    if f < 0 {
        return float64(-f)
    }
    return float64(f)
}

type Vertex struct {
    X, Y float64
}

func (v *Vertex) Abs() float64 {
    return math.Sqrt(v.X*v.X + v.Y*v.Y)
}

$ ./object3
1.4142135623730951
5
```

Переменная типа интерфейс `Abser` благополучно принимает как объект типа `MyFloat` (оператор: `a = f`), так и указатель на объект `*Vertex` (оператор: `a = &v`). Но 3-й (комментированный) случай присвоения не пройдёт компиляцию из-за синтаксической ошибки: тип `Vertex` не удовлетворяет интерфейсу `Abser` потому что `Abs()` метод определен только для `*Vertex`, но не для `Vertex`.

Тип реализует интерфейс путем реализации методов этого интерфейса. Нет никакой явной декларации о намерениях. Подобным образом интерфейсы отделяют пакеты реализации от пакетов, которые определяют интерфейсы: одни не зависят от других. Этот

механизм также вынуждает определять точные интерфейсы, потому что вы не должны искать все реализации и помечать их с новым именем интерфейса. В примере ниже:

- определяются типы интерфейсов Reader и Writer;
- интерфейсы объявляют методы Read() и Write();
- тем самым, переменные типов Reader и Writer (или объединённого интерфейса ReadWriter) наследуют методы от типа File из пакета с именем os (реализующим штатный в составе Go платформенно независимый интерфейс к операционным системам: <https://pkg.go.dev/os>)

- потому как именно тип File реализует объявленные интерфейсами операции:

```
func (f *File) Read(b []byte) (n int, err error)
func (f *File) Write(b []byte) (n int, err error)
```

- это позволяет присваивать переменным таких интерфейсных типов значения переменных из пакета OS:

```
var (
    Stdin  = NewFile(uintptr(syscall.Stdin), "/dev/stdin")
    Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")
    Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")
)
```

- и для этих переменных интерфейсных типов применимы впоследствии все операции из пакета fmt, например:

```
func Fprint(w io.Writer, a ...interface{}) (n int, err error)
```

- которые используют аргументы интерфейсного типа io.Writer (пакет io — базовые примитивы ввода-вывода: <https://pkg.go.dev/io>), который также наследует (является обёрткой) для базового типа File и его метода Write() (который был показан 3-мя пунктами выше):

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Вот так замыкается вся цепочка связей. И нам нет необходимости определять полную реализацию операций ввода-вывода для переменных своего нового определяемого типа ReadWriter:

object4.go

```
package main

import (
    "fmt"
    "os"
)

type Reader interface {
    Read(b []byte) (n int, err error)
}

type Writer interface {
    Write(b []byte) (n int, err error)
}

type ReadWriter interface {
    Reader
    Writer
}
```

```
func main() {
    var w ReadWriter
    // os.Stdout implements Writer
    w = os.Stdout
    fmt.Fprintf(w, "hello, writer\n")
}
```

```
$ ./object4
hello, writer
```

Именованние интерфейсов

По соглашению (разработчиков Go), интерфейсы объявляющие единственный метод называются по имени этого метода плюс суффикс `...er`, или аналогичный простой модификатор, позволяющий построить существительное: `Reader`, `Writer`, `Formatter`, `CloseNotifier` и др.

Уже существует ряд таких имен, произведенных в честь имен функций, которые они реализуют. `Read`, `Write`, `Close`, `Flush`, `String` и так далее имеют каноническое написание и смысл. Чтобы избежать путаницы, не давайте своему методу одно из таких имён, если только он не имеет ту же самую сигнатуру и смысл. И наоборот, если ваш тип реализует метод с таким же смыслом, что и метод широко известного типа, дайте ему то же имя и сигнатуру: вызывайте для конвертера вашего типа в строку `String()`, но не `ToString()`.

Контроль интерфейса¹⁹

Как мы видели выше, тип не нуждается в том, чтобы декларировать явно, что он реализует какой-то интерфейс. Вместо этого, тип реализует интерфейс только с помощью того, что он реализует методы для этого интерфейса. На практике, большинство интерфейсных преобразований статические и поэтому проверяются во время компиляции. Например, передача `*os.File` функции, ожидающей `io.Reader` не будет компилироваться до тех пор, пока `*os.File` не реализует интерфейс `io.Reader`.

Но иногда проверка интерфейса требуется во время выполнения, тем не менее. Одним из демонстрирующих экземпляров является пакет `encoding/json`, определяющий интерфейс `Marshaler`. Когда JSON кодировщик получает значение, которое реализует этот интерфейс, кодировщик вызывает метод `Marshaler` чтобы преобразовать его в формат JSON вместо того, чтобы делать стандартное преобразование. Кодер проверяет это свойство во время выполнения используя проверку типа, подобно следующему:

```
m, ok := val.(json.Marshaler)
```

Если необходимо только опросить является ли некоторый тип реализацией интерфейса, фактически не используя сам интерфейс, то возможно только проверять ошибку, используя пустой идентификатор для игнорирования результата проверки:

```
if _, ok := val.(json.Marshaler); ok {
    fmt.Printf("value %v of type %T implements json.Marshaler\n", val, val)
}
```

В одном из тонких случаев, в такой ситуации возникает проблема, когда необходимо гарантировать, в рамках реализации пакета, что тип на самом деле удовлетворяет интерфейсу. Если тип — например, `json.RawMessage` — нуждается в индивидуальном представлении JSON, то он должен реализовать `json.Marshaler`, но нет статических преобразований, которые мог бы вызвать компилятор, чтобы это проверить автоматически. Если тип случайно не удовлетворяет интерфейсу, JSON кодировщик по-прежнему будет работать, но не будет использовать индивидуальную реализацию. Чтобы гарантировать что реализация корректная, глобальная декларация с помощью пустого идентификатора может быть использована в пакете:

¹⁹ Заимствовано из книги «Effective Go».


```
var _ json.Marshaler = (*RawMessage)(nil)
```

В этой декларации присвоение с участием преобразования от `*RawMessage` к `Marshaler` требует, чтобы `*RawMessage` реализовывал `Marshaler`, и что это обстоятельство будет проверено во время компиляции. В случае, если `json.Marshaler` интерфейс изменится со временем, то этот пакет больше не будет компилироваться, и мы будем знать, что он нуждается в обновлении.

Использование пустого идентификатора в этой конструкции указывает на то, что декларация существует только для проверки типа, но не для создания переменной. Не делайте этого для каждого типа, который удовлетворяет интерфейсу, тем не менее. По соглашению, такие объявления используются только когда отсутствуют статические преобразования, уже присутствующие в коде, что является довольно редким событием.

Обработка ошибочных ситуаций

В Go нет возбуждаемых исключений. Это связано не со сложностями реализации, или минималистичностью языка, но является твёрдой позицией его разработчиков. Вот их позиция:

Мы уверены, что сопряжение возбуждаемых исключений со структурами управления, таких как `try-catch-finally` идиома, имеет своим результатом запутанный код. Оно также имеет тенденцию стимулировать программистов к оформлению слишком многих самых элементарных ошибок (например, из-за невозможности открыть файл) как возбуждение исключения.

Go использует другой подход. Для простоты обработки ошибок, Go предлагает множественные возвращаемые значения, что делает лёгкой возможность сообщить об ошибке не переписывая само возвращаемое значение. Встроенный тип `error`, в сочетании с другими возможностями Go, делает обработку ошибок приятной, но и весьма отличающейся от других языков.

Go также имеет несколько встроенных функций для сигнализации и восстановления из действительно экстремальных ситуаций. Механизм восстановления выполняется только как часть состояния функции, которая будет прервана в результате ошибки, которая достаточно катастрофична. Но это не требует создания каких-то дополнительных структур управления и, при правильном использовании, может привести в итоге к чистому коду обработки ошибок.

Ошибкой (<https://pkg.go.dev/errors>) в Go может быть всё, что может описать себя как строка ошибки (это смешная фраза из документации). Идея реализуется предопределённым встроенным интерфейсным типом `error`, с его **единственным** (требуемым к реализации) методом `Error()`, который должен формировать и возвращать строку описания ошибки (каталог `errors` архива примеров):

error1.go

```
package main

import (
    "fmt"
    "time"
)

type MyError struct {
    When time.Time
    What string
}

func (e *MyError) Error() string {
    return fmt.Sprintf("at %v, %s", e.When, e.What)
}
```

```

func run() error {
    return &MyError{
        time.Now(),
        "it didn't work",
    }
}

func main() {
    if err := run(); err != nil {
        fmt.Println(err)
    }
}

$ ./error1
at 2022-04-13 20:45:23.565002072 +0300 EEST m=+0.000083744, it didn't work

```

Из упоминавшихся, в мотивации принятой в Go идеологии обработки ошибок, «встроенных функций для сигнализации и восстановления» имеются в виду уже упоминавшийся (при рассмотрении функций) оператор `defer` и встроенные функции `panic()` и `recover()`, описываемые ниже.

```

func panic(v interface{})
func recover() interface{}

```

Встроенная функция `panic()` прекращает нормальное выполнение текущей сопрограммы (goroutine). Когда функция `F()` вызывает `panic()`, нормальное исполнение `F()` немедленно прекращается. После прекращения выполняются какие-либо функции, исполнение которых было отложено (`defer`) в функции `F()`, и затем `F()` возвращает управление её вызвавшему. Абонент `G()`, вызывавший ранее `F()`, также ведёт себя точно так, как при прямом вызове `panic()`: прекращение исполнения `G()` и выполнение любых ею отложенных (`defer`) функций. Это продолжается (распространяется вверх) до тех пор, пока все функции в исполняемой сопрограмме последовательно не будут остановлены в обратном порядке. В этот момент программа (сoproграмма) завершается и регистрируется код ошибка, включающий, в том числе, и значение аргумента первоначального вызова `panic()`. Эта последовательность завершений называется паникованием, и может управляться с помощью встроенной функции `recover()`.

Встроенная функция `recover()` позволяет программе управлять поведением паникующей сопрограмы. Выполнение вызова `recover()` внутри любой из отложенных (`defer`) функций (но не в какой либо функции в коде непосредственно) прерывает распространение панической последовательности вверх, восстанавливает нормальное исполнение и возвращает значение `error`, переданное вызовом `panic()`. Если `recover()` вызовется вне отложенных функций, он не сможет остановить паникующую последовательность. В этом случае, или когда сопрограмма не паникует, или если аргумент вызова `panic()` был `nil`, `recover()` возвращает `nil`. Таким образом, возвращаемое `recover()` значение является индикатором того, паникует ли сопрограмма.

Не программисту объяснить такое невозможно, а программисту синтаксис всех таких взаимодействий гораздо проще показать на примере кода, который заимствован из публикаций по Go:

panic.go

```

package main

import "fmt"

func main() {
    f()
    fmt.Println("Returned normally from f.")
}

```

```

func f() {
    defer func() {
        if r := recover(); r != nil {
            fmt.Println("Recovered in f", r)
        }
    }()
    fmt.Println("Calling g.")
    g(0)
    fmt.Println("Returned normally from g.")
}

func g(i int) {
    if i > 3 {
        fmt.Println("Panicking!")
        panic(fmt.Sprintf("%v", i))
    }
    defer fmt.Println("Defer in g", i)
    fmt.Println("Printing in g", i)
    g(i + 1)
}

```

```
$ go build panic.go
```

```
$ ./panic
```

```

Calling g.
Printing in g 0
Printing in g 1
Printing in g 2
Printing in g 3
Panicking!
Defer in g 3
Defer in g 2
Defer in g 1
Defer in g 0
Recovered in f 4
Returned normally from f.

```

Это чем-то похоже на структурную обработку исключений (SEH) в Windows, но в гораздо более изящном исполнении.

Структура пакетов (библиотек) Go

Каждая программа и каждый подключаемый программный компонент в Go является пакетом (см. объявление: `package main` в листингах всех примеров), а точнее — набором пакетов. Программы собираются из пакетов, и то, что обычно называется библиотеками, в Go называется набором пакетов.

Исполнимая программа должна иметь пакет `main` и метод `main()`, с которого и начинается выполнение программы, и по завершении которого программа закрывается.

Каждая программа и каждый подключаемый программный компонент в Go является **пакетом**. Программы начинают выполнение из пакета `main`.

Программа может использовать (импортировать) функциональность других пакетов (программных компонент) импортируя пакет, объявляя оператор `import`:

```

package main

import (
    "fmt"
    "math/rand"
)

```

```
func main() {
    fmt.Println("My favorite number is", rand.Intn(10))
}
```

Эта программа использует пакеты с импортируемых путей "fmt" и "math/rand" (заданы символьными константами). По соглашению, имя пакета совпадает с последним элементом файлового пути импорта.

Выражение `import "package"` дает доступ к методам, переменным и константам пакета `package` (только **видимым** вне пакета, имена которых начинаются с большой буквы). Обращаться к ним следует через оператор `.` (точка), например, `package.Foo()`.

Предыдущая показанная запись импорта, так как в этой записи каждая строка пути пакета завершается неявным `';`, эквивалента такой форме её записи (с явным разделителем `';`):

```
import("fmt"; "math/rand")
```

Также можно записывать множественные описатели импорта:

```
import "fmt"
import "math/rand"
```

Область видимости в Go заметно отличается от таковой в языке C. Внутри пакета все переменные, функции, константы и типы имеют глобальную видимость, даже если пакет представлен несколькими файлами. Но для обращения к ним из клиента, импортирующего этот пакет, имена последних должны начинаться с большой буквы. Например, клиент импортировал пакет `package`, в котором объявлены следующие переменные:

```
const hello = "Im not visible in client, just in package" //видна только в пакете
const Hello = "I can say hello to client" //видна также при импорте
```

Вторая константа будет доступна клиенту по имени `package.Hello`. Первая будет недоступна вне рамок самого пакета `package` вовсе.

Вы можете как создавать свои собственные целевые пакеты как составные части крупного разрабатываемого проекта, так и, наверняка, использовать API предоставляемый набором стандартных пакетов, предоставляемых с системой Go.

Полную иерархия стандартных пакетов (необходимую для указания путевых имён в импорте) в вашей установленной версии GoLang, и откомпилированных в форму объектных архивов (статических библиотек вида `*.a`) вы можете найти в своей системе по полному путевому имени `/lib/go/pkg/linux_amd64` (здесь компонент имени `go` — это просто ссылка на актуальную установленную версию, например, `go-1.18` в моём случае ... но при корректной инсталляции GoLang номер версии не имеет значения в этом путевом имени):

```
$ ls -l /lib/go
lrwxrwxrwx 1 root root 7 map 23 2022 /lib/go -> go-1.18
```

Полная иерархия файлов (пакетов) необходимая для правильного указания путевых имён в строках импорта — очень важна, она очень объёмна, но стоит того, чтобы её изучить детально приступая к работе:

```
$ pwd
/lib/go/pkg/linux_amd64
```

```
$ ls -w80
archive  database  go        io        net.a     regexp.a  testing
bufio.a  debug     hash      io.a     os        runtime   testing.a
bytes.a   embed.a   hash.a    log      os.a      runtime.a text
cmd       encoding  html      log.a    path      sort.a    time
compress encoding.a html.a    math     path.a    strconv.a time.a
container errors.a   image     math.a   plugin.a  strings.a unicode
context.a expvar.a   image.a   mime     reflect   sync      unicode.a
```

```
crypto    flag.a    index    mime.a    reflect.a  sync.a    vendor
crypto.a  fmt.a    internal net      regexp    syscall.a
```

```
$ tree | grep \.a$ | wc -l
442
```

Вот такое число пакетов (терминальные файлы *.a этого дерева) вы получаете в распоряжение своей исполнимой системой (с учётом версии, естественно):

```
$ tree | head -n25
```

```
.
├── archive
│   ├── tar.a
│   └── zip.a
├── bufio.a
├── bytes.a
├── cmd
│   ├── asm
│   │   └── internal
│   │       ├── arch.a
│   │       ├── asm.a
│   │       ├── flags.a
│   │       └── lex.a
│   └── compile
│       └── internal
│           ├── abi.a
│           ├── amd64.a
│           ├── arm64.a
│           ├── arm.a
│           ├── base.a
│           ├── bitvec.a
│           ├── deadcode.a
│           ├── devirtualize.a
│           ├── dwarfgen.a
│           └── escape.a
```

```
$ tree | tail
```

```
└── cpu.a
    └── text
        ├── secure
        │   └── bidirule.a
        ├── transform.a
        └── unicode
            ├── bidi.a
            └── norm.a
```

```
119 directories, 442 files
```

И относительно каждой терминальной вершины мы можем видеть что это статический библиотечный архив (и именно в таком путевом виде пакеты должны указываться в строках описания импорта):

```
$ file math.a
```

```
math.a: current ar archive
```

```
$ file net/rpc/jsonrpc.a
```

```
net/rpc/jsonrpc.a: current ar archive
```

Подобная же иерархия пакетов, но для реализации `gssgo` (если вы используете Go проект GCC) находится совсем в другом месте, и она, в основном, соответствует тому что показано выше:

```
$ cd /lib/x86_64-linux-gnu/go/1.11/x86_64-linux-gnu
```

Здесь элемент пути «11» определяется версией:

```
$ gccgo --version
```

```
gccgo (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0
```

```
Copyright (C) 2021 Free Software Foundation, Inc.
```

```
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Вы можете убедиться, что дефолтная версия gccgo будет та же, что и для всех продуктов проекта GCC в вашей системе:

```
$ gcc --version
```

```
gcc (Ubuntu 11.3.0-1ubuntu1~22.04) 11.3.0
```

```
Copyright (C) 2021 Free Software Foundation, Inc.
```

```
This is free software; see the source for copying conditions. There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Состав здесь (gccgo) несколько отличается (от GoLang), отличается и формат представления пакетов:

```
$ ls -w80
```

archive	embed.gox	html	math	reflect.gox	syscall.gox
bufio.gox	encoding	html.gox	math.gox	regexp	testing
bytes.gox	encoding.gox	image	mime	regexp.gox	testing.gox
compress	errors.gox	image.gox	mime.gox	runtime	text
container	expvar.gox	index	net	runtime.gox	time
context.gox	flag.gox	internal	net.gox	sort.gox	time.gox
crypto	fmt.gox	io	os	strconv.gox	unicode
crypto.gox	go	io.gox	os.gox	strings.gox	unicode.gox
database	hash	log	path	sync	
debug	hash.gox	log.gox	path.gox	sync.gox	

```
$ tree | wc -l
```

```
188
```

```
$ tree | tail
```

```
|   └─ template.gox
└─ time
|   └─ tzdata.gox
└─ time.gox
└─ unicode
|   └─ utf16.gox
|   └─ utf8.gox
└─ unicode.gox
```

```
35 directories, 150 files
```

Как мы уже видели относительно GoLang, число терминальных файлов — это и есть число доступных в этом релизе пакетов Go. Относительно формата пакетов (здесь это *.gox):

```
$ file math.gox
```

```
math.gox: ELF 64-bit LSB relocatable, x86-64, version 1 (SYSV), stripped
```

Можно предположить (?) что по формату это динамические (разделяемые) библиотеки, в отличие от статических, используемых GoLang.

Наконец, основные оригинальные пакеты проекта GoLang в виде самых свежих **исходных кодов**, если возникает необходимость, вы можете найти для изучения на сайте разработчиков проекта — <https://go.dev/src/>. Например, в каталоге <https://go.dev/src/fmt/> мы находим все необходимые файлы из уже неоднократно использовавшегося пакета fmt:

```
doc.go  export_test.go  fmt_test.go  format.go
```

print.go scan.go scan_test.go stringer_test.go

И можете текстуально изучить реализационную часть пакета чтобы разрешить тонкие детали:

print.go

```
...
func (b *buffer) Write(p []byte) (n int, err error) {
    *b = append(*b, p...)
    return len(p), nil
}
func (b *buffer) WriteString(s string) (n int, err error) {
    *b = append(*b, s...)
    return len(s), nil
}
func (b *buffer) WriteByte(c byte) error {
    *b = append(*b, c)
    return nil
}
}
```

doc.go

```
/*
...
    General:
        %v          the value in a default format.
                    when printing structs, the plus flag (%+v) adds field
names
        %#v         a Go-syntax representation of the value
        %T          a Go-syntax representation of the type of the value
        %%          a literal percent sign; consumes no value
...
*/
```

Поскольку пакетная система Go недостаточно полно описана (всё находится в динамике и развитии), то показанные выше расширенные «целеуказания» будут отнюдь не лишними — информацию по использованию пакетов Go вам предстоит отчасти извлекать самостоятельно. Но показанных источников информации достаточно, чтобы при их тщательном изучении полностью использовать все возможности пакетов Go.

Примечание: При этом не забываем, что по правилам видимости Go импортироваться для использования могут только объекты, имена которых начинаются с большой литеры (`WriteString` и т.д.). Эта памятка **на порядок** сокращает объём просматриваемой информации.

Функция *init*

Каждый исходный файл **может** (но вовсе не обязательно) определить свою собственную `init`-функцию без аргументов, чтобы настроить все требуемые начальные состояния. (На самом деле каждый файл может иметь даже несколько функций `init()`.) И выглядит это так: `init()` вызывается после того, как все объявленные переменные в пакете пройдут инициализацию, что может произойти только после того, как все импортированные пакеты будут инициализированы.

Помимо этого, все инициализации, которые не могут быть выражены в виде деклараций — обычное использование `init`-функции: проверить и восстановить корректность состояний программы перед тем, как начнётся реальное выполнение.

Вот пример из документации использования `init`-функции:

```
func init() {
    if user == "" {
        log.Fatal("$USER not set")
    }
    if home == "" {
```

```

    home = "/home/" + user
}
if GOPATH == "" {
    GOPATH = home + "/go"
}
// GOPATH may be overridden by --GOPATH flag on command line.
flag.StringVar(&GOPATH, "GOPATH", GOPATH, "override default GOPATH")
}

```

Если в пакете исполнимой программы `main`, или в экспортируемых им пакетах, имеется одна или более функций `init()`, то они автоматически будут все вызваны **прежде** вызова функции `main()` в пакете `main` (раньше фактического начала выполнения кода программы).

Импорт для использования побочных эффектов

Ранее объяснялось, что неиспользование в коде импортируемых пакетов — грубая ошибка, прерывающая компиляцию. Временно эту ситуацию решают «пустые идентификаторы» (имя — символ подчёркивания), как объяснялось, но их наличие маркирует код как «находящийся в развитии, черновой», и, в конце концов, они должны быть удалены.

Но иногда полезно импортировать пакет только для использования его побочных эффектов, без какого-либо прямого использования. Например, в коде своей функции `init()`, пакет `net/http/pprof` регистрирует обработчики HTTP-данных, которые содержат информацию для отладки. Пакет имеет экспортируемый API, но большинству клиентов нужна только регистрация обработчика и доступ к данным через веб-страницу. Чтобы импортировать пакет только для использования его побочных эффектов, предлагается переименовать пакет в пустой идентификатор:

```
import _ "net/http/pprof"
```

Эта форма импорта указывает ясно, что пакет импортируется только из-за его побочных эффектов, потому нет никакой возможности сослаться для применения на пакет: в этом файле, он не имеет имени. (Если бы это было не так, и мы не использовали бы это имя, то компилятор должен был бы отклонить программу как ошибочную.)

Некоторые полезные и интересные стандартные пакеты

Выше описан состав и иерархия **некоторых** стандартных пакетов системы Go и алгоритм поиска информации в ней. Как легко видеть, это очень объёмная (и всё расширяющаяся по ходу развития) система. Обзор пакетов системы поимённо вы найдёте в документации: Standard library — <https://pkg.go.dev/std>.

Пакеты GoLang, конкретно установленные в вашей системе, находятся в каталоге (иерархии каталогов) `/usr/lib/go/pkg/linux_amd64`. Их число рано:

```
$ tree /usr/lib/go/pkg/linux_amd64 | grep \.a | wc -l
494
```

Изучите предварительно, перед работой, состав стандартных пакетов Go, предоставляемых вашей версией GoLang.

Ниже будут бегло затронуты только **отдельные** пакеты Go из этой иерархии, по минимуму, которые автору показались особо необходимыми для целей экспериментирования с Go. Этот выбор очень субъективный.

По затрагиваемым пакетам показываются лишь ключевые понятия, позволяющие представить назначение, состав и направления использования пакетов. Детальное описание было бы слишком объёмным. Но по каждому пакету даётся URL страницы с полным и детальным описанием.

Пакет runtime

Пакет времени выполнения содержит переменные, константы и функции, которые повязаны с взаимодействием кода с исполняющей системой Go, такие, например, как функции управления сопрограмами.

Например, переменная `GOGC` устанавливает начальный процент для срабатывания сборщика мусора Go. Сборка мусора инициируется когда соотношение свежих выделенных данных к данным, оставшихся в живых после предыдущей сборки мусора, превысит этот процент. По умолчанию `GOGC=100`. Установка `GOGC=off` вообще запрещает работу сборщика мусора. Изменить величину `GOGC` позволяет функция из пакета `runtime/debug`:

```
func SetGCPercent(percent int) int
```

Переменная `GOMAXPROCS` ограничивает (или перепределяет) число потоков операционной системы, которые могут выполняться на уровне пользовательского кода Go одновременно. Нет ограничения на число потоков, которые могут быть заблокированы в системных вызовах от имени Go кода: такие не учитываются в `GOMAXPROCS` ограничении. Функция из этого же пакета `GOMAXPROCS()` запрашивает и изменяет этот лимит (функция будет рассмотрена позже, при рассмотрении механизма сопрограмм Go).

Функция из пакета `runtime/debug`:

```
func SetMaxStack(bytes int) int
```

Эта функция задает максимальный объем памяти, который может использоваться одной сопрограммой под стек. Если любая сопрограмма превышает этот предел при росте её стека, то программа завершает работу. Функция возвращает предыдущее установленное значение. Установка по умолчанию составляет: 1 Гб на 64-битных системах, 250 Мб на 32-битных системах. Функция `SetMaxStack()` полезна, главным образом, для ограничения ущерба, наносимого сопрограммами, когда они входят в бесконечную рекурсию.

Это были только некоторые примеры из пакета `runtime`. Полную информацию вы найдёте по ссылке: <https://pkg.go.dev/runtime>.

Форматированный ввод-вывод

Без форматированного ввода вывода не обходится ни одна программа, начиная с «Hello World!». Поэтому именно такой пакет мы рассматриваем в первую очередь. Эти возможности собраны в пакет `fmt`, и содержат функции аналогичные `printf()`, `sprintf()` и `scanf()` из C. Само понятие форматной строки заимствовано из C, но сделано проще, и расширено функционально.

Большинство элементов формата следует привычным из C (`%d`, `%b`, `%x`, `%c`, `%s`, `%f`, `%e`, `%p` ...). Но есть и весьма специфичные:

- `%v` — форматирование в умалчиваемом (default) представлении для каждого **типа** данных (для структур добавление флага плюс: `%+v` — будет добавлять имена полей)
- `%t` — представление логических значений как `true` или `false`
- ...

Пакет определяет достаточно много функций форматированного ввода-вывода. Вот только некоторые (для примера):

- вывод в формате по умолчанию:

```
func Print(a ...interface{}) (n int, err error)
func Println(a ...interface{}) (n int, err error)
```

- вывод в явно определяемом формате:

```
func Printf(format string, a ...interface{}) (n int, err error)
```

- сканирование текста со стандартного ввода:

```
func Scan(a ...interface{}) (n int, err error)
func Scanf(format string, a ...interface{}) (n int, err error)
```

- форматированный вывод в строку:

```
func Sprint(a ...interface{}) string
func Sprintf(format string, a ...interface{}) string
```

Обращаем внимание на то, что функции `Print()` и `Printf()` близки к своим аналогам в C, но вот функции `Sprint()` и `Sprintf()` имеют совершенно другую семантику: они не модифицируют строку указанную 1-м параметром (как в C за счёт побочного эффекта), а возвращают новую строку как результат. Это связано с тем, что в Go строки неизменяемые. Чтобы пояснить это не очень внятное объяснение, процитирую одну строку из примеров кода находящихся в архиве (там множество подобных строк для того чтобы их рассмотреть):

```
return fmt.Sprintf("[%02d,%x]", p.n, p.t)
```

Кроме большого набора функций форматного ввода-вывода, пакет содержит целый ряд интересных определений (интерфейсных) типов, например:

```
type Stringer interface {
    String() string
}
```

Интерфейс `Stringer` реализуется любым новым типом (вашим собственным!), который имеет определение метода `String()`, и именно этот метод будет определять «родной» формат (%v, см. выше) для значений (объектов) этого типа. Метод `String()` используется для печати значений, передаваемых в качестве операнда в любой формат, который принимает строку, или в не форматированный вывод, такие как `Print()` или `Println()`. Пример реализации интерфейса `Stringer` для нового определяемого типа `point` (2D точка) показан в примере `triangle.go` (находится в каталоге архива `compare/triangle`):

```
type point struct {
    xy complex128
}
func (p *point) String() string {
    return fmt.Sprintf("[%0.2f,%0.2f] ", real( p.xy ), imag( p.xy))
}
```

Вот как может выглядеть индикация типа элемента данных, что может оказаться очень полезным при отладке:

```
fmt.Printf("%T | %T | %T\n ", "строка", 1, 2.2)
```

Результат:

```
string | int | float64
```

Примеры использования функция форматного ввода-вывода раскиданы во множестве по всем кодам архива примеров, поэтому не будут показываться отдельно.

Детальную информацию по пакету `fmt` вы найдёте по ссылке: <https://pkg.go.dev/fmt>.

Строки и пакет *strings*

Практически ни одно современное приложение не обходится без использования символьных строк и разного уровня глубины обработки этих строк. В ранних языках программирования, FORTRAN, а позже C и далее C++, акцент техники программирования делался на вычислительных аспектах, а текстовым строкам отводилась роль, главным образом, для представления строчных литералов, украшающих результаты вывода на терминал или печать, форматирующих их, и делающих более понятными. Поэтому обработка символьной информации — это самое слабое место языков C и C++ (в C++ оно как-то обходится использованием шаблонной реализации

типа `std::string` из STL — внешними относительно языка средствами). Язык Go предоставляет встроенный (или как они это называют типы 1-го уровня) тип `string`.

Но с годами актуальность текстовой обработки возрастала всё более и более, её доля становилась преобладающей относительно самих вычислительных аспектов, а способы обработки становились всё изощрённее: поиск, замена, регулярные выражения... Что вызвало даже появление языков программирования заметно ориентированных на текстовую обработку: Perl, Python...

Разработчики Go это учли, и язык предоставляет разнообразнейшие средства обработки текстовой информации. Как уже отмечалось, в Go всё представляется в UNICODE в кодировке UTF-8: сама запись программного кода, имена переменных, содержимое символьных строк. А раз это так, то средства символьной обработки Go решают все проблемы мультязычности. Важно отметить, что это относится не только к текстам на самых разнообразных языках мира, но, с тем же успехом и инструментарием, к любым последовательностям из разных областей деятельности...

Вот как это формулируется в статье относительно последовательностей-палиндромов (примеры которых мы будем рассматривать в последнем разделе, посвящённом примерам кодов):

В молекулах ДНК присутствует от 100 тысяч до 1 млн коротких палиндромных последовательностей. Принцип их формирования несколько отличается от того, как это происходит для слов и предложений. Поскольку молекула ДНК состоит из двух комплементарных цепочек нуклеотидов, а нуклеотиды всегда соединяются одним и тем же образом (аденин (А) с тиминами (Т), цитозин (С) с гуанином (G)), считается, что одноцепочечная последовательность ДНК является палиндромом, если она равна своей комплементарной последовательности, прочитанной задом наперёд. Например, последовательность ACC TAGGT является палиндромной, так как комплементарной ей будет последовательность TGG ATCCA, которая совпадает с исходной, прочитанной задом наперёд.

Или (там же) относительно последовательностей музыкальных нотных знаков:

Пьесе играют «как обычно», но после того, как она заканчивается, ноты переворачивают и произведение играют заново, причём музыка не изменится. Итераций может быть сколько угодно и неизвестно, что является верхом, а что низом. Такие произведения можно играть вдвоём, читая ноты с разных сторон. Примерами таких музыкальных палиндромов могут являться произведения «Застольная мелодия для двоих» Моцарта и «Путь Мира» Мошелеца, а также «Прелюдия и постлюдия» из фортепианного цикла Пауля Хиндемита «Ludus tonalis».

Наконец, строка может быть набор просто символов-значков (каталог примеров `strings/runes`), по типу смайликов, не относящихся ни к какому языковому алфавиту (последний UNICODE символ в 1-й строке, — это вообще символ расширения имён файлов исходных кодов на новом языке программирования Mojo):

smail.go

```
package main
import "fmt"

func main() {
    fmt.Println(" ")
    fmt.Println(string([]rune{0x1F468, 0x1F9B0, 0x2318})) // ☺
}
$ go build smile.go
$ ./smile

☺
```

В большинстве своём, все (осмысленные) строки Go встроенного типа `string`, с которыми мы зачастую будем иметь дело — это последовательность символов UNICODE в кодировке UTF-8 неограниченной длины. Но, в самом общем случае, строка Go может

содержать **любую** последовательность байт, которая может не соответствовать вообще никаким символам UNICODE / UTF-8. Строго говоря, строки Go представляют собой **срезы** (слайсы) произвольных **байт**, и доступны эти срезы только для **чтения** (это ещё одна, и более точная формулировка неизменяемости строк Go).

Любопытства ради, и в подтверждение сказанного, можем рассмотреть такое маленькое приложение (каталог strings/runes):

nulstr.go

```
package main
import "fmt"

func main() {
    buf := make([]byte, 10)
    fmt.Printf("%v\n", buf)
    str := string(buf)
    fmt.Printf("%x\n", []rune(str))
}
```

```
$ go run nulstr.go
[0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0]
```

Здесь мы создали строку, состоящую исключительно из байт с нулевыми значениями! Это может быть большой неожиданностью для программистов привыкших к правилам C/C++!

К строкам применима встроенная функция `len()` (которая вообще очень широко используется в языке). Но функция `len()` работает с байтами, но не символами UTF-8 строки, это очень важно помнить!

Над значениями типа `string` выполнимы основные операции:

`+` — конкатенация, объединение содержимого 2-х строк в одну новую строку

`[]` — индексация, выборка из строки по порядковому номеру, индексация начинается с номера 0. Операция индексации строки производится по байтам. Но выборка вовсе не означает, что этот отдельный байт строке можно изменить — строки в Go неизменяемые! Пример этого (каталог примеров strings/runes):

chars.go

```
package main
import "fmt"

func main() {
    sample := "русская строка"
    fmt.Printf("[%d]: %s", len(sample), sample); println()
    fmt.Printf("%x", sample); println()
    for i := 0; i < len(sample); i++ {
        fmt.Printf("%x ", sample[i])
    }
    println()
    for i := 0; i < len(sample); i++ {
        fmt.Printf("%c ", sample[i])
    }
    println()
}
```

```
$ ./chars
[27]: русская строка
d180d183d181d181d0bad0b0d18f20d181d182d180d0bed0bad0b0
d1 80 d1 83 d1 81 d1 81 d0 ba d0 b0 d1 8f 20 d1 81 d1 82 d1 80 d0 be d0 ba d0 b0
Ñ Ñ Ñ Ñ Ð ° Ð ° Ñ   Ñ Ñ Ñ Ð % Ð ° Ð °
```

Пересчитав символы в текстовой строке (14) мы убеждаемся что 27 (len()) — это число байт в строке, но не символов. Ниже мы видим типичное 2-байтовое представление (d1 80 ...) символов кириллических шрифтов.

Не следует смешивать понятие строки string и байтового среза массива, в который обычно считывается последовательность вводимых символов (из файлов или терминала). Но они легко преобразуются один в другой (фрагмент из примера goproc/multy.go в архиве):

```
buf := make([]byte, 1024)
for {
    fmt.Printf("> ")
    n, _ := os.Stdin.Read(buf)
    str := string(buf[:n - 1])
    ...
}
```

Для приведения в соответствие байтового и символьного представления Go вводит целочисленное представление символов, тип rune. Стандарт UNICODE использует термин «кодировочная точка» («code point») для обозначения элемента, представленного одним значением. Кодировочная точка U+2318 с шестнадцатеричным значением 2318, например, представляет символ японского алфавита ㊿. Но «кодировочная точка» — это нечто вроде труднопроизносимого слова, поэтому стандарт Go вводит более короткое понятие: руна (rune). Именно rune соответствует одному мультиязычному символу UNICOD, независимо от того, сколькими байтами в этом языке представляется каждый символ. Но rune это несколько не символьный тип, это числовой тип, над которым можно производить, например, арифметические операции (что мы увидим вскоре). При выводе в виде содержимого строки срезы rune нужно приводить к типу string, что мы и видим в примере (каталог strings/runes):

str rune.go

```
package main
import "fmt"
import "bufio"
import "log"
import "os"

func main() {
    scanner := bufio.NewScanner(os.Stdin)
    for scanner.Scan() {
        s := scanner.Text()
        fmt.Println(s)
        r := []rune(s)
        fmt.Println(r)           // hex int32
        fmt.Printf("%U\n", r) // UNICODE
        fmt.Println(string(r))
        fmt.Println("-----")
    }
    if err := scanner.Err(); err != nil {
        log.Println(err)
    }
}

$ ./str rune
ASCII string
ASCII string
[65 83 67 73 73 32 115 116 114 105 110 103]
[U+0041 U+0053 U+0043 U+0049 U+0049 U+0020 U+0073 U+0074 U+0072 U+0069 U+0065 U+0067]
ASCII string
-----
русская строка
русская строка
```

```
[1088 1091 1089 1089 1082 1072 1103 32 1089 1090 1088 1086 1082 1072]
[U+0440 U+0443 U+0441 U+0441 U+043A U+0430 U+044F U+0020 U+0441 U+0442 U+0440 U+043E
U+043A U+0430]
русская строка
-----
```

В пакете `strings`, который мы рассматриваем, представлено великое множество операций и методов над строками, практически на все случаи жизни (это один из самых больших пакетов Go). В плотном контакте с пакетом `strings` находятся и ряд других пакетов стандартной библиотеки Go (<https://pkg.go.dev/std>), с которыми приходится взаимодействовать при написании кода строчной обработки:

`bytes` — опер операции над байтовыми срезами;

`bufio` — строчный ввод-вывод (файлы, теримнал...);

`regexp` — регулярные выражения;

`strconv` — константы разнообразные преобразования типов в строки и наоборот;

`unicode` — константы и предикаты отдельных категорий символов UNICODE;

Все эти пакеты (их использование) будут показаны в последней части книги, когда мы перейдём к конкретным примерам решения задач.

Некоторое насальное представление о строчных операциях Go, соотношении строк с байтовыми массивами и операциях вывода строк даёт следующий пример (каталог `types`):

string1.go

```
package main

import (
    "fmt"
)

func main() {
    var (
        s1    string = "это первая русскоязычная строка "
        s2    string = "и вторая строка"
        s5     = "it is a short english string"
        sfmt   = "[%d]: %s\n"
    )
    fmt.Printf(sfmt, len(s1), s1)
    fmt.Printf(sfmt, len(s5), s5)
    buf := make([]byte, 120)
    for i := range s1 {
        buf[i] = s1[i]
    }
    fmt.Printf("[%d]: ", len(buf))
    fmt.Println(buf)
    buf = []byte(s1)
    fmt.Printf("[%d]: ", len(buf))
    fmt.Println(buf)
    s3 := string(buf)
    fmt.Printf(sfmt, len(s3), s3)
    fmt.Printf("[%d]: %s \n", len(s1+s2), s1+s2)
    fmt.Printf("%c => %d\n", s5[0], s5[0])
    fmt.Printf("%c => %d\n", s1[0], s1[0])
    //  s5[ 0 ] = '+'
    buf = []byte(s5)
    buf[0] = byte('+')
    fmt.Printf("%s\n", buf)
}
```

```
[60]: это первая русскоязычная строка
[28]: it is a short english string
[120]: [209 0 209 0 208 0 32 208 0 208 0 209 0 208 0 208 0 209 0 32 209 0 209 0 209 0
209 0 208 0 208 0 209 0 208 0 209 0 209 0 208 0 208 0 209 0 32 209 0 209 0 209 0 208 0
208 0 208 0 32 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[60]: [209 141 209 130 208 190 32 208 191 208 181 209 128 208 178 208 176 209 143 32 209
128 209 131 209 129 209 129 208 186 208 190 209 143 208 183 209 139 209 135 208 189 208
176 209 143 32 209 129 209 130 209 128 208 190 208 186 208 176 32]
[60]: это первая русскоязычная строка
[88]: это первая русскоязычная строка и вторая строка
i => 105
Ñ => 209
+t is a short english string
```

Как показывает этот пример:

Хотя и декларируется представление UTF-8 для символьной информации (и с ним отлично работает операция конкатенации '+'), функция `len()` (длина строки) и индекс символа в строке ('[]') оперируют с **байтами**, но не **символами** UNICODE. Это аналогично ситуации с `char[]` в языке C, и использованием мультибайтовых функций типа `mblen()` и других `mb*()`, и требует перехода к представлению в широких символах `wchar_t`.

Если вы попытаете изменить символ в строке (попробуйте!), оператором типа:

```
s5[0] = 22
```

То в таком случае получите ошибку вида:

```
./string1.go:25: cannot assign to s5[0]
```

Потому, что строки `string` в Go неизменяемые. Это в точности напоминает решение той же проблемы в Python: или строки должны представляться простым нуль-терминальным массивом в манере C со всеми вытекающими «удобствами», либо строки должны быть неизменяемыми в своём внутреннем содержании.

Но вы вполне можете сделать:

```
buf = []byte(s5)
buf[0] = byte('+')
fmt.Printf("%s\n", buf)
```

И иметь в итоге в выводе (но это уже тип не `string!`):

```
...
+t is a short english string
```

Для посимвольной работы со строками может с успехом использоваться цикл в форме итератора, как и для всяких агрегатных данных Go (каталог `types`):

string3.go

```
package main

import "fmt"

func main() {
    for pos, char := range "строка+\x80+ф" { // \x80 is an illegal UTF-8 encoding
        fmt.Printf("символ %#U в байтовой позиции %d\n", char, pos)
    }
}
```

```
$ go build -o string3 string3.go
```

```
$ ./string3
символ U+0441 'с' в байтовой позиции 0
```

символ U+0442 'т' в байтовой позиции 2
 символ U+0440 'р' в байтовой позиции 4
 символ U+043E 'о' в байтовой позиции 6
 символ U+043A 'к' в байтовой позиции 8
 символ U+0430 'а' в байтовой позиции 10
 символ U+002B '+' в байтовой позиции 12
 символ U+FFFD '␣' в байтовой позиции 13
 символ U+002B '+' в байтовой позиции 14
 символ U+0424 'ф' в байтовой позиции 15

Для строк итератор `range` делает для вас существенно больше работы, чем просто перебор байтов строки — в он выбирает отдельные символы в кодировке UNICODE, анализируя UTF-8. Если встречается байт, содержащий ошибочный код, не представимый в UTF-8, то цикл-итератор заменяет его на фиксированное значение U+FFFD, ассоциированное с встроенным типом `rune` (в терминологии Go `rune` — это числовое значение одного символа UNICODE, UTF-32, см. ниже).

Но кроме непосредственных возможностей встроенных операций для типа `string` (достаточно обширных), язык Go сопровождается ещё пакетом символьной обработки `strings`, содержащем много функций для операций со строками. Здесь содержатся все эквиваленты библиотеки C `<string.h>` и ещё более (показаны для иллюстрации только некоторые из более 40 прототипов):

```

func Contains(s, substr string) bool
func ContainsAny(s, chars string) bool
func ContainsRune(s string, r rune) bool
func Count(s, sep string) int
...
func Fields(s string) []string
func FieldsFunc(s string, f func(rune) bool) []string
...
func Index(s, sep string) int
func IndexByte(s string, c byte) int
...
func Join(a []string, sep string) string
...
func Repeat(s string, count int) string
func Replace(s, old, new string, n int) string
func Split(s, sep string) []string
...
func Trim(s string, cutset string) string
...
func TrimSpace(s string) string
...

```

В символьных операциях фигурирует тип `rune`. По определению — это псевдоним для `int32` и эквивалент `int32` во всех отношениях. Этот тип используется, по соглашению, чтобы различать символьные (UNICODE) значения (в каком-то смысле это эквивалентно широкому, 4-байтному типу `wchar_t` в C) и целочисленного значения для вычислительных операций.

Несколько примеров использования функций из пакета `strings`, заимствованные из документации пакета и собранные «под одной крышей» (каталог `types`), показаны в примере (все коды Go показываются не в том виде, как они нам, возможно, «нравятся», а так как они выглядят после форматирования их командой `go fmt ...`):

string2.go

```

package main

import (
    "fmt"
    "strings"

```



```

        "unicode"
    )

func main() {
    fmt.Printf("Fields are: %q\n", strings.Fields(" foo bar baz "))
    f := func(c rune) bool {
        return !unicode.IsLetter(c) && !unicode.IsNumber(c)
    }
    fmt.Printf("Fields are: %q\n", strings.FieldsFunc(" foo1;bar2,baz3...", f))
    fmt.Println(strings.Index("chicken", "ken"))
    fmt.Println(strings.Index("chicken", "dmr"))
    f = func(c rune) bool {
        return unicode.Is(unicode.Han, c)
    }
    fmt.Println(strings.IndexFunc("Hello, 世界", f))
    fmt.Println(strings.IndexFunc("Hello, world", f))
    s := []string{"foo", "bar", "baz"}
    fmt.Println(strings.Join(s, ", "))
    rot13 := func(r rune) rune {
        switch {
        case r >= 'A' && r <= 'Z':
            return 'A' + (r-'A'+13)%26
        case r >= 'a' && r <= 'z':
            return 'a' + (r-'a'+13)%26
        }
        return r
    }
    fmt.Println(strings.Map(rot13, "'Twas brillig and the slithy gopher..."))
    fmt.Println(strings.Replace("oink oink oink", "k", "ky", 2))
    fmt.Println(strings.Replace("oink oink oink", "oink", "moo", -1))
    fmt.Printf("%q\n", strings.Split("a,b,c", ","))
    fmt.Printf("%q\n", strings.Split("a man a plan a canal panama", "a "))
    fmt.Printf("%q\n", strings.Split(" xyz ", ""))
    fmt.Printf("%q\n", strings.Split("", "Bernardo O'Higgins"))
    fmt.Printf("[%q]\n", strings.Trim(" !!! Achtung! Achtung! !!! ", "! "))
    fmt.Println(strings.TrimSpace(" \t\n a lone gopher \n\t\r\n"))
}

```

Рассмотрение кодов этих примеров использования без дополнительных комментариев показывает общие принципы использования функций пакета лучше любых объяснений:

```
$ go build -o string2 string2.go
```

```
$ ./string2
```

```

Fields are: ["foo" "bar" "baz"]
Fields are: ["foo1" "bar2" "baz3"]
4
-1
7
-1
foo, bar, baz
'Gjnf oevyyvt naq gur fyvgul tbcure...
oinky oinky oink
moo moo moo
["a" "b" "c"]
["" "man " "plan " "canal panama"]
[" " "x" "y" "z" " "]
[""]
["Achtung! Achtung"]
a lone gopher

```

Попутно в примере показано использование некоторых функций-предикатов из пакета `unicode`, имеющего непосредственное отношение к символьной обработке.

Детальную информацию по пакету вы найдёте по ссылке: <https://pkg.go.dev/strings>.

Строчные литералы

Выше, в записях строчных значений (символьных строк, строчных литералов) мы пользовались традиционным для большинства языков программирования форматом, традицией сложившейся за десятилетия — содержимое строки заключается в двойные кавычки. Но в Go, для обеспечения разнообразными форматами представления, этого оказывается мало.

- Традиционная запись строк в двойных кавычках допускает управляющие последовательности в строке, в потоке, отмечаемые обратным слэшем, мы их уже неоднократно видели, как например: `"\n"`, `"\t"`, `"\r"`, `"\x"` (этот формат предоставляет возможность записать прямое 16-ричное значение в байт)...
- Сырое (raw) представление строки, в котором любые символы записываются «as is», без всякого управляющего смысла, записываются в одиночных кавычках, только не в тех одиночных кавычках, как в языке Python и некоторых других, а в обратных кавычках (на клавиатуре соответствующие букве Ё). Например: ``это знак переноса строки: \n``.
- А обычные одиночные кавычки в Go зарезервированы для записей литералов типа `rune`, которые тоже имеют некоторый смысл в контексте строчных представлений, но являются целочисленными значениями `int32`. Например: `'Ю'`.

Всё это разнообразие не там легко для осознания, но покрывает все потребности работы с многобайтными символами UTF-8. Это разнообразие лучше увидеть на примере (каталог `/strings/runes`):

raw.go

```
package main

import "fmt"

func main() {
    s := " ЮЁЩ"
    r := []rune(s)
    fmt.Printf("%x : байт %d символов %d\n", s, len(s), len(r))
    fmt.Printf("%#v\n", []byte(s))
    for p, c := range s {
        fmt.Printf("%d[%#U] ", p, c)
    }
    println()

    for i := 0; i < len(r); i++ {
        fmt.Printf("%x ", r[i])
    }
    println()

    //      f0 9f 94 a5 ec 9a 94 c2 a3 c2 b5 d0 a9
    s = "\xf0\x9f\x94\xa5\xec\x9a\x94\xc2\xa3\xc2\xb5\xd0\xa9"
    fmt.Printf("%s\n", s)
    s = ` \xf0\x9f\x94\xa5\xec\x9a\x94\xc2\xa3\xc2\xb5\xd0\xa9 `
    fmt.Printf("%s\n", s)
    s = string([]rune{0x1f525, 0xc694, 0xA3, 0xB5, 0x429})
    fmt.Printf("%s\n", s)
    s = string([]rune{' ', 'Ю', 'Ё', 'µ', 'Щ'})
    fmt.Printf("%s\n", s)
}
```

И выглядит это так:

```
$ ./raw
f09f94a5ec9a94c2a3c2b5d0a9 : байт 13 символов 5
130
```

```

[]byte{0xf0, 0x9f, 0x94, 0xa5, 0xec, 0x9a, 0x94, 0xc2, 0xa3, 0xc2, 0xb5, 0xd0, 0xa9}
0[U+1F525 ' ' ] 4[U+C694 'ϣ'] 7[U+00A3 '£'] 9[U+00B5 'μ'] 11[U+0429 'Щ']
1f525 c694 a3 b5 429
ϣ£μЩ
\xf0\x9f\x94\xa5\xec\x9a\x94\xc2\xa3\xc2\xb5\xd0\xa9
ϣ£μЩ
ϣ£μЩ

```

У нас здесь есть строка из пяти многобайтных символов, суммарной длиной 13 байт: знак-символ языка программирования Mojo — 4 байта, иероглиф корейского языка — 3 байта, знак английского фунта стерлингов — 2 байта, литера греческого алфавита, часто фигурирующая в математических формулах — 2 байта, буква русского алфавита, которые, как мы уже знаем — 2 байта.

Сначала (первые 4 строки вывода) представлена внутренняя структура строки в разных формах. Дальше (последние 4 строки вывода) показан вывод такой же строки, записанной в разных формах строчных литералов.

Большие числа

Для вычислений с точностью или разрядностью, превышающей стандартные машинные представления, пакет `math/big` вводит типы больших чисел: `big.Int` (знаковое большое целое) и `big.Rat` (большое вещественное). Теоретически их разрядность определяется только размером доступной оперативной памяти (то есть практически безгранична).

Естественно (поскольку Go не допускает переопределение операций) для таких типов данных вводится очень широкий набор методов, выполняющих весь набор арифметических операций на все случаи, описанные по типу:

```

func (z *Int) Add(x, y *Int) *Int
func (z *Rat) Add(x, y *Rat) *Rat

```

Пример использования, непосредственно из документации этого пакета:

bigint.go

```

package main

import (
    "fmt"
    "log"
    "math/big"
)

func main() {
    // The Scan function is rarely used directly;
    // the fmt package recognizes it as an implementation of fmt.Scanner.
    i := new(big.Int)
    _, err := fmt.Sscan("18446744073709551617", i)
    if err != nil {
        log.Println("error scanning value:", err)
    } else {
        fmt.Println(i)
    }
}

```

```
$ go build -o bigint bigint.go
```

```
$ ls -l bigint
```

```
-rwxrwxr-x 1 olej olej 2307888 фев 14 22:30 bigint
```

```
$ ./bigint
```

```
18446744073709551617
```

Ещё один пример использования больших чисел будет показан позже в разделе, посвящённом примерам кода.

Детальную информацию по пакету вы найдёте по ссылке: <https://pkg.go.dev/math/big>.

Автоматизированное тестирование

В состав стандартной библиотеки GoLang входит пакет для подготовки кода автоматизированного тестирования — `testing`. Этот пакет создаёт все необходимые предпосылки для единообразной организации разработки проекта, в технологии известной как «разработка через тестирование», позволяет одновременно с разработкой кода, развитием проекта, создавать набор unit-тестов для его функций и методов.

- Вы создаёте новый файл, заканчивающийся именем на `*_test.go`, в том же каталоге, где находятся исходные коды вашего пакета.

- Внутри этого файла импортируйте пакет `testing` ...

- И пишите для функций вида `Foo()` соответствующие тестовые функции, имеющие вид `func TestFoo(t *testing.T) { ... }`.

- По команде `go test` в этом каталоге запускается сценарий, который находит тестовые функции, создает тестовый двоичный файл и запускает его.

Пример построения тестов будет рассмотрен позже, в завершающей части книги. А детальное описание можно найти в описании самого пакета: <https://pkg.go.dev/testing> (и дополнительные разъяснения в FAQ по Go: https://go.dev/doc/faq#Packages_Testing).

Источники информации

[1] Спецификация языка Go на официальной странице проекта:

The Go Programming Language Specification, Version of March 10, 2022 — <https://go.dev/ref/spec>

[2] Miek Gieben : Learning Go, стр. 112 — <http://archive.miek.nl/files/go/Learning-Go-latest.pdf>

[3] Caleb Doxsey : An Introduction to Programming in Go, 2012, ISBN: 978-1478355823 —

<http://www.golang-book.com/>

<http://www.golang-book.com/assets/pdf/gobook.pdf>

[4] Go Language Community WiKi — <https://code.google.com/p/go-wiki/w/list>

[5] Евгений Охотников : Краткий пересказ «Effective Go» на русском языке, 2009.11.19 —

http://eao197.narod.ru/desc/short_effective_go.html

[6] Effective Go — http://golang.org/doc/effective_go.html

[7] Effective Go (RU) (Эффективный Go), Русскоязычный текст на сайте проекта GoLang —

https://github.com/Konstantin8105/Effective_Go_RU/blob/master/README.md

[8] Go for C++ Programmers — <https://code.google.com/p/go-wiki/wiki/GoForCPPProgrammers>

[9] The GNU Go Compiler — <https://gcc.gnu.org/onlinedocs/gccgo/>

[10] Frequently Asked Questions (FAQ) — <https://go.dev/doc/faq>

[11] Directory `src/pkg/` — <https://go.dev/src/>

[12] Andrew Gerrand : The Go Blog. Defer, Panic, and Recover, 4 August 2010 —

<http://blog.golang.org/defer-panic-and-recover>

[13] Калейб Докси : Введение в программирование на Go — <http://golang-book.ru/>

[14] Let's Go: объектно-ориентированное программирование на Голанге —

<https://code.tutsplus.com/ru/tutorials/lets-go-object-oriented-programming-in-golang--cms-26540>

[15] Go Язык программирования. Standard library — <https://runebook.dev/ru/docs/go/>

Структура кода: пакеты, модули, импорт...

Всё ранее рассказанное хорошо до тех пор, пока размер кода проекта не превысит некоторый критический размер, и остаётся обозримым в полном своём объёме (это, обычно, до 1-2 страниц текста максимум). Дальше код предстоит делить на части, и в игру вступают правила взаимодействия этих частей: импорт, видимость имён...

Пакеты

Долгое время от начала развития до версии 1.11 (примерно конец 2018 года) единственным средством структуризации кода в Go было понятие пакета, [1]:

Пакет Go состоит из Go-файлов, расположенных в одной и той же директории, в начале которых имеется одинаковое выражение `package`. Пакеты, подключаемые к программам, позволяют расширять их возможности. Некоторые пакеты входят в состав стандартной библиотеки Go. А это значит, что они, если вы пользуетесь Go, уже у вас установлены.

Вы можете в одном каталоге разместить сколь угодно много файлов, фрагментов **одного и того же** пакета, с одинаковым именем пакета в выражении `package`. Все имена объектов, независимо в каком фрагменте пакета они находятся, в каком они написании, и в каком порядке расположены (внутри файлов, например) видны во всём пакете. Смотрим пример `private` (все примеры этого раздела находятся в каталоге `looking.for архива`):

```
$ ls -o *.go
-rw-rw-r-- 1 olej 66 фев 19 12:27 test1.go
-rw-rw-r-- 1 olej 41 фев 19 12:27 test2.go
```

Пример очень условный, но нас здесь интересует не его содержательная часть (синтаксис и конструкции языка), а то как код разложен по разным файлам:

test2.go

```
package main
```

```
func main() {
    demo()
}
```

test1.go

```
package main
```

```
import "fmt"
```

```
func demo() {
    fmt.Println("HI")
}
```

Это два фрагмента одного и того же пакета `main` — так всегда должен называться пакет главной программы, с которой начинается её выполнение. Если же пакет называется любым другим именем, то это будет пакет выполняющий по смыслу функции библиотеки: из него будут импортироваться функции, типы и методы...

Сразу же здесь отметим, что (вопреки широко существующему предубеждению) что имена пакетов никак не соотносятся с именами файлов, в которых они помещены!

Вспоминаем (уже говорилось ранее), что в Go имена, начинающиеся с заглавной (большой) буквы видны (могут быть импортированы) за пределами пакета (из других пакетов). А имена, начинающиеся с прописной (малой) буквы (`demo` в показанном примере) видны только внутри пакета ... но во всех `.go` файлах этого пакета. (По аналогии с C++ такие имена могут быть названы как приватные, `private`, почему мы так и назвали пример.)

Для использования любых объектов (имен) извне пакета, из других пакетов, имена таких пакетов должны обязательно быть **импортированы**, что прописывается в ключевом слове `import ...` что мы уже неоднократно видели ранее, и что будет иметь место практически в любом пакете кода. В импорте (списке импорта) могут быть указаны либо простые имена пакетов (например, `"math"` или `"math/cmplx"`), либо имена с путевыми (сетевыми URL, как правило) префиксами (в архиве примеров, например, мы можем видеть `"github.com/gSpera/meval"`). Простые имена указывают, по именам, импорт многочисленных пакетов из стандартной библиотеки GoLang [2]. На сайте GoLang можно, также, контекстным поиском [6] найти нужный из числа сотен публичных пакетов, опубликованных на сетевых GIT репозиториях.

В коде имена объектов из импортируемых пакетов должны указываться с полным квалифицированным именем (с указанием имени пакета через точку), например, в одном из примеров в архиве (`compare/triangle`):

```
import "math/cmplx"
...
r1, θ1 := cmplx.Polar(slice[i+1].xy - slice[0].xy)
```

Импорт может указываться с определением алиаса (синонима), для замены длинного имени коротким и, иногда, более удобным. Например, в ближайшем примере далее (`public.ali` в архиве) мы сможем увидеть:

```
import L "public/lib"
...
L.Demo()
```

Уже упоминалось, что в Go любое объявление, которое не используется далее в коде, по решению авторов языка квалифицируется как критическая ошибка. Иногда может понадобиться импорт пакета только для использования побочного эффекта такого импорта (например, для выполнения функции инициализации `init()`, которая может присутствовать в любом пакете). В таком случае следует в качестве алиаса пакета указать временную переменную, обозначающуюся символом подчёркивания, который мы уже неоднократно упоминали:

```
import _ "github.com/ziutek/mymysql/godrv"
```

Но вернёмся к оставленному нами примеру с разделённым пакетом `main` на два файла `test1.go` и `test2.go` ...

Сборку приложения можно делать так:

```
$ go build test1.go test2.go
$ ls -o
-rwxrwxr-x 1 olej 1895359 фев 19 12:31 test1
-rw-rw-r-- 1 olej      66 фев 19 12:27 test1.go
-rw-rw-r-- 1 olej      41 фев 19 12:27 test2.go
```

Или так:

```
$ go build test2.go test1.go
$ ls -o
-rw-rw-r-- 1 olej      66 фев 19 12:27 test1.go
-rwxrwxr-x 1 olej 1895351 фев 19 12:34 test2
-rw-rw-r-- 1 olej      41 фев 19 12:27 test2.go
```

В обоих случаях мы получим равнозначные по функциональности исполнимые файлы ... но, обратим внимание, любопытно, что, в зависимости от указания порядка сборки, их размеры несколько отличаются.

Или можем просто выполнять написанный код (в режиме JIT, по типу интерпретатора):

```
$ go run test2.go test1.go
HI
```

Или даже так:

```
$ go run .  
HI
```

В завершение пара вопросов относительно пакетов Go, которые часто задают...

Можно ли в одном каталоге разместить файлы *.go двух, или более, разных пакетов? Нет, этого делать нельзя! (Возможно, это было допустимо ранее, до версии 1.11 и введения модулей — на это указывают некоторые из обсуждений в Интернет). Поменяем имя пакета в файле test1.go (на lib1, например, в архиве каталог looking.for/public.dbl), и при компиляции получим сообщение подобного вида:

```
$ go run .  
found packages lib1 (test1.go) and main (test2.go) in ...looking.for/public.dbl
```

Можно ли в одном каталоге расположить несколько разнообразных *.go приложений, каждое из которых располагается в своём собственном пакете с именем main (ведь исполнимое приложение всегда должно начинать исполнение из такого пакета)? Да, вполне — и так сделано во многих каталогах нашего архива примеров: для компактности и вашего удобства, например (gorpoc/concurrent):

```
$ ls *.go  
racea.go race.go racem.go races.go
```

И вы можете собрать (или выполнить: go run ...) каждое из таих приложений **индивидуально** (или из Makefile, когда утилита make станет прилагать правило сборки поочерёдно к каждому приложению отдельно):

```
$ go build race.go  
$ ls -o race* | grep rwx  
-rwxrwxr-x 1 olej 1937897 фев 20 14:44 race
```

Но если вы попытаетесь выполнить групповую сборку, то потерпите терминальную ошибку (потому что компилятор попытается объединить воедино main пакеты из разных приложений):

```
$ go build *.go  
# command-line-arguments  
./race.go:10:6: main redeclared in this block  
    ./racea.go:11:6: other declaration of main  
./racem.go:11:6: main redeclared in this block  
    ./racea.go:11:6: other declaration of main  
./races.go:10:6: main redeclared in this block  
    ./racea.go:11:6: other declaration of main
```

Модули

В экосистеме GoLang создано, и пролжает нарабатываться, множество сторонних пакетов, пригодных для свободного и всестороннего использования. Они, главным образом, располагаются в сетевых репозиториях, о которых упоминалось выше (и в заключительной части книги мы ещё попрактикуемся в том, как искать нужные пакеты в публичных сетевых GIT репозиториях, для использования вместо того, чтобы писать такой код вручную). Но с некоторых пор (и из-за такого обилия, во многом), для поддержания всего этого хозяйства потребовалась система поддержания версий — взамен или в дополнение концепции пакетов. И такая система поддержки модулей появляется в GoLang начиная с версии 1.11 (2018 год). Определение модуля может быть такое [3]:

Модуль определяется деревом исходных файлов Go с файлом go.mod в корневом каталоге дерева. Каталог, содержащий файл go.mod, называется корнем модулем (module root). Обычно корень модуля также будет соответствовать корню хранилища исходного

кода (но в общем случае это не обязательно). Модуль представляет собой набор всех пакетов Go в корневом каталоге модуля и его подкаталогах, но исключая поддеревья с собственными файлами `go.mod`.

Система модулей требуется, главным образом, для поддержания версионности ваших модулей в сетевых репозиториях. Но задача отслеживания и коррекции версий намного шире наших намерений рассмотрения, и это следует изучать по документации. Тем не менее, если наши пакеты, даже локально обрабатываемые, потребуют использования (библиотечных) кодов из Интернет GIT репозитория, то тут мы не сможем обойтись без поддержки модулей (мы просто станем получать непреодолимые ошибки компиляции).

В связи с введением модулей у команды `go` (а мы уже говорили ранее, что это не индивидуальная команда, а менеджер системы GoLang) появляется целый подкласс команд: `go mod ...` И создание модуля начинается с варианта:

```
$ go mod init <имя_модуля>
```

```
...
```

Такая команда создаёт в каталоге выполнения файл описания модуля (некоторые авторы называют его манифест модуля): `go.mod`, дальнейшие манипуляции будут изменять содержимое этого файла.

Имя модуля в этой команде (параметр), достаточно произвольное, обозначает только то, под каким именем из этого модуля будут осуществлять импорт другие фрагменты кода Go. Вот несколько примеров команд из реальной практики работы с модулями:

```
$ go mod init github.com/olej-tsil-51/inout
```

```
...
```

```
$ go mod init gitlab.com/go1195807/inout.git
```

```
...
```

```
$ go mod init bitbucket.org/oleg-tsiliuric/inout.git
```

```
...
```

Каждая из этих команд, ссылающихся на популярные в Интернет публичные GIT репозитории GitHub, GitLab, или Bbitbucket (или ещё какой), вовсе не означают, что они будут что-то создавать и размещать в репозиториях. Они только **намекают**, что по этому путевому имени (URL) вы имеете намерение разместить модуль, из пакетов которого будет осуществляться публичный импорт объектов (функций, типов, методов...). Всё фактическое размещение модуля (кодов) в GIT репозитории вы обязаны осуществить сами, вручную, в соответствии с техникой размещения проектов в сетевых репозиториях (с некоторой спецификой конкретно для каждого из них).

Более того, если вы планируете работать с модулем индивидуально, в своей локальной файловой системе, то вы можете вообще дать модулю произвольно выбранное имя (иногда его выбирают по имени размещения кода модуля, но это совершенно не обязательно). Мы, для начала, порепетируем с командами модулей на пакетах, которые мы станем позже обрабатывать в следующем разделе относительно дженериков (каталог `generic`). Выполним (с произвольным именем модуля `generic` — пусть он у нас просто совпадает с именем каталога):

```
$ go mod init generic
```

```
go: creating new go.mod: module generic
```

```
go: to add module requirements and sums:
```

```
    go mod tidy
```

После чего у нас появится новый файл манифеста:

```
$ cat go.mod
```

```
module types
```

```
go 1.20
```

Пока он определяет только версию компилирующей системы, для которой предназначен пакет. И если пакет (или ни один пакет в этом модуле) не требует импорта пакетов из иных сетевых репозитория, то условия (безошибочность) компиляции не изменятся. Но в нашем каталоге примера один из пакетов (файл `max.go`) хочет импортировать пакет

constraints из **сетевого** репозитория:

```
$ grep import max.go
import "fmt"
import "golang.org/x/exp/constraints"
...
```

И раньше (до версии 1.11) такой номер проходил бы! Но не сейчас. Вы получите ошибку типа:

```
$ go build max.go
max.go:4:8: no required module provides package golang.org/x/exp/constraints; to add it:
    go get golang.org/x/exp/constraints
```

И выполнив рекомендуемую в сообщении команду мы можем так и решить эту новую проблему (производим загрузку недостающего пакета из Интернет):

```
$ go get golang.org/x/exp/constraints
go: downloading golang.org/x/exp v0.0.0-20230310171629-522b1b587ee0
go: added golang.org/x/exp v0.0.0-20230310171629-522b1b587ee0
```

Но проще и лучше (и я думаю что правильнее ... и, тем более, разобраться что же всё-таки происходит) сразу после инициализации файла `go.mod` (как это показано выше: `go mod init ...`) выполнить рекомендуемое там же (в подсказке) действие вот такой командой:

```
$ go mod tidy
go: finding module for package golang.org/x/exp/constraints
go: found golang.org/x/exp/constraints in golang.org/x/exp v0.0.0-20230310171629-522b1b587ee0
```

Эта команда проанализировала списки импорта текущего пакета, и подготовила (загрузила) **все** требуемые зависимости в нужной (адекватной пакету!) версиях. В итоге чего получилось:

```
$ ls -l go.*
-rw-rw-r-- 1 olej olej 85 map 11 15:34 go.mod
-rw-rw-r-- 1 olej olej 207 map 11 15:34 go.sum

$ cat go.mod
module generic
go 1.20
require golang.org/x/exp v0.0.0-20230310171629-522b1b587ee0
```

Содержимое `go.mod` (как мы видим) изменилось. Но это не всё! Произошла загрузка требуемых пакетов для импорта, которые мы находим здесь:

```
$ go env GOPATH
/home/olej/go
```

```
$ ls `go env GOPATH`/pkg/mod/golang.org/x/exp@v0.0.0-20230310171629-522b1b587ee0 -w80
apidiff      CONTRIBUTING.md  go.mod  LICENSE  rand      slog
cmd          devtools        go.sum  maps     README.md utf8string
codereview.cfg  ebnf           inotify mmap     shootout
constraints   ebnflint        io      PATENTS  slices
```

```
$ tree `go env GOPATH`/pkg/mod/golang.org/x/exp@v0.0.0-20230310171629-522b1b587ee0/
constraints
├── constraints.go
└── constraints_test.go
```

```
0 directories, 2 files
```

И вот (только) теперь пакет, требующий импорта из сети, замечательно компилируется ...

чем мы и займёмся в следующем разделе относительно дженериков:

```
$ go build max.go
$ ls -l max
-rwxrwxr-x 1 olej olej 1839159 мар 11 16:17 max
```

До этого места мы рассмотрели (очень по верхам) как организовать в модуле своего проекта импорт потребных API из сетевых GIT. А дальше мы посмотрим как организовать пакеты в рамках модуля своего проекта, и как происходит импорт сквозь границы пакетов...

Как было отмечено ранее, в одном каталоге файловой системы мы не можем сформировать ещё один отдельный пакет, отличный от main. Тогда рассмотрим такую последовательность ()

- В корневом каталоге проекта определим модуль, с самым произвольным именем (мы не планируем публиковать модуль для сетевого использования):

```
$ go mod init public
go: creating new go.mod: module public
go: to add module requirements and sums:
    go mod tidy
```

- ... более ничего интересного там нет смотреть...

- Создаём дочерний каталог под отдельный пакет модуля:

```
$ mkdir lib
```

- Во вновь созданном каталоге создаём исходный код отдельного пакета (собственно, это тот же, перенесенный, код, который рассмотрен выше, но пакет переименован в lib1):

```
test1.go
package lib1
import "fmt"

func Demo() {
    fmt.Println("HI")
}
```

- Наконец, в корневом каталоге проекта (модуля) формируем код пакета main:

```
test2.go
package main
import "public/lib"

func main() {
    lib1.Demo()
}
```

- В конечном итоге, у нас получилась структура проекта (модуля):

```
$ tree
.
├── go.mod
├── lib
│   └── test1.go
└── test2.go

1 directory, 3 files
```

Теперь мы прежде испытаем то что у нас получилось, а только затем проанализируем детали. Запуск (или, стем же успехом, сборку исполнимого бинарного файла можно производить только из каталога, содержащего пакет main — корневого):

```
$ go run test2.go
HI
```

Или даже так:

```
$ go run .  
HI
```

Пример сознательно упрощён до нельзя, до тривиального состояния, но позволяет нам вычлениить основные выводы из происходящего:

- Имена пакетов Go никак не соотносятся с именами файлов *.go в которых они содержатся.
- Импорт в утверждении в ключевом слове `import` нужно указать как `"<имя_модуля>/<каталог_пакета>"` ...
- Но уже в указании полных квалифицированных имен (как правило имён вызываемых функций) мы должны формировать их как `<имя_пакета>.<имя_объекта>`.
- И только имена, начинающиеся с заглавной (большой) буквы, повторимся, могут быть использованы (импортированы) за пределами пакета их определения: сравните `Demo()` с `demo()` в предыдущем примере (с разделённым между файлами единым пакетом) ... и всё это вне зависимости кто где и в каком файле находится — всё дело в видимости пакета в точке использования.

Этот же пример может быть представлен с использованием алиасных имён пакетов в импорте (каталог `looking.for/public.ali` архива), здесь изменения претерпевает только импортирующий пакет `main`:

```
package main  
import L "public/lib"  
  
func main() {  
    L.Demo()  
}
```

И, в завершение, более реалистичный пример (каталог `looking.for/angels` архива), закрепляющий показанное: главный пакет, импортирующий два тригонометрических пакета, один из которых вычисляет значения от аргументов заданных в угловых градусах, а другой — в радианах. Структура этого проекта:

```
$ tree angels  
angels  
├── digree  
│   └── углы.go  
├── go.mod  
├── head.go  
└── radian  
    └── рад1.go
```

2 directories, 4 files

Файлы проекта (в общем, код теперь говорит сам за себя любых объяснений):

go.mod

```
module angels
```

```
go 1.22.0
```

head.go

```
package main
```

```
import (  
    "angels/digree"  
    "angels/radian"
```

```

        "fmt"
        "math"
    )

    func main() {
        const угол1, угол2 = 30., 30. / 180 * math.Pi
        fmt.Printf("синус %f градусов = %f\n", угол1, углы.Sin(угол1))
        fmt.Printf("синус %f радиан   = %f\n", угол2, радианы.Sin(угол2))
    }

```

рад1.go

```

package радианы

import "math"

func Cos(rd float64) float64 {
    return math.Cos(rd)
}

func Sin(rd float64) float64 {
    return math.Sin(rd)
}

```

углы.go

```

package углы

import "math"

var a2r func(dg float64) float64

func Cos(dg float64) float64 {
    return math.Cos(a2r(dg))
}

func Sin(dg float64) float64 {
    return math.Sin(a2r(dg))
}

func init() {
    a2r = func(dg float64) float64 {
        return dg / 180 * math.Pi
    }
}

```

И в результате:

```

$ go run head.go
синус 30.000000 градусов = 0.500000
синус 0.523599 радиан   = 0.500000

```

Или так:

```

$ go build head.go
$ ./head
синус 30.000000 градусов = 0.500000
синус 0.523599 радиан   = 0.500000

```

Или даже так:

```

$ go run .
синус 30.000000 градусов = 0.500000
синус 0.523599 радиан   = 0.500000

```

Примечание: В этом примере, да и в других тоже, использовано наименование многих объектов русскоязычными именами. Чем каждый раз проверяется утверждение что GoLang весь насквозь пронизан UTF-8... Так вот в этом примере, в единственном месте за много лет, проявилась несовместимость, состоящая в том, что если подкаталоги пакетов (digree и radian) назвать русскоязычными (или китайскими) именами, то на утверждении `import` компилятор GoLang сходит с ума...

Публикация модуля в Интернет

Это и есть конечная цель разработки вашего модуля: разместить его для публичного доступа по сети на каком-то из сетевых репозиториях GIT²⁰ (возможно и не публичных, а развёрнутых корпоративных или персональных). Именно за счёт этого механизма, и его простоты и прозрачности, в среде языка Go доступны многие сотни (если не тысячи) пакетов от разнообразных разработчиков!

Мы развёрнём сетевой репозиторий на GitHub, но с таким же успехом (и точно так же в основных шагах) может быть выбран GitLab, Bitbucket ... или любой другой. Прежде всего мы создадим пустой репозиторий (заготовка будущего модуля). Для этого зайдём через WEB интерфейс под своим логином на GitHub, и жмём кнопку New для создания нового репозитория:

Рис. 1.6 Создание нового репозитория GitHub

²⁰ GoLang при инсталляции сразу предлагает установить и репозиторийные системы: `brz`, `mercurial`, `subversion`, `git` (это обсуждалось ранее при рассмотрении инсталляции среды). Код Go умеет импортировать сторонние пакеты не только из `git`, но и из всех остальных. А это означает, что публиковать свои модули вы можете, очевидно, и этих системах. Но это автором не проверялось...

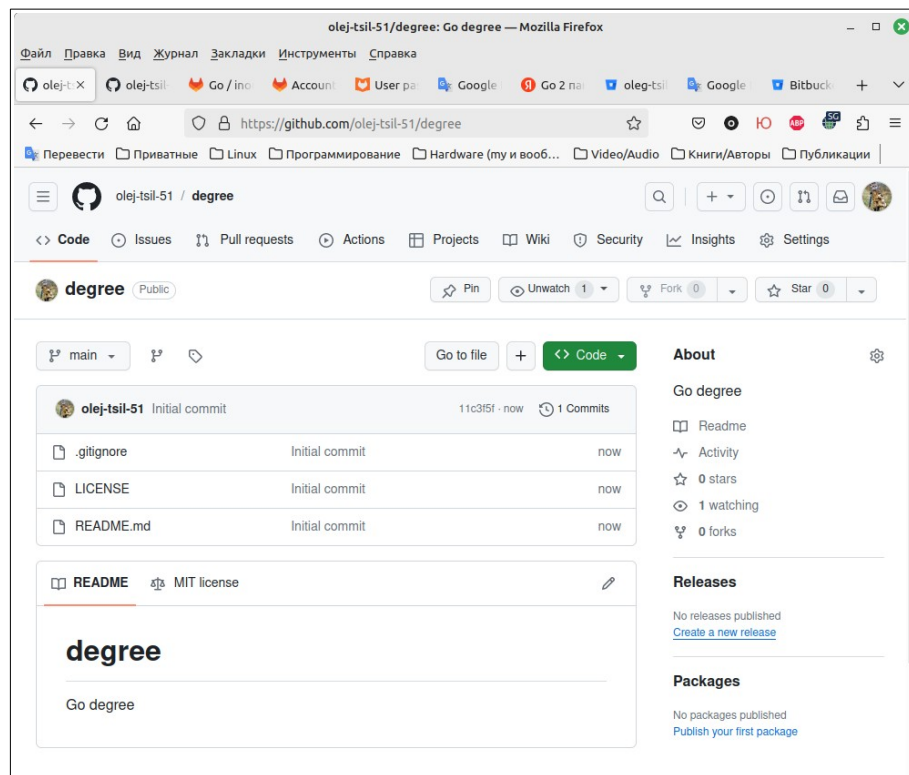


Рис. 1.7 Создан пустой репозиторий

Определяем параметры нового проекта: имя репозитория (очень важно нам для ссылок), лицензия, и, может, некоторые другие в зависимости от специфики. В итоге мы получаем пустой репозиторий в качестве шаблона будущего проекта.

Примечание: Я не знаю способа, и существует ли он вообще, создать репозиторий нового проекта на GitHub средствами консольной команды `git` с локального рабочего компьютера... поэтому для создания буду пользоваться браузером и WEB интерфейсом к GitHub. Аналогично создаётся шаблон будущего репозитория и в системах GitLab или Bitbucket.

Клонирую этот пустой репозиторий в локальную файловую систему, как каталог для будущего Go модуля `degree`:

```
$ git clone https://github.com/olej-tsil-51/degree.git
Клонирование в «degree»...
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
Получение объектов: 100% (5/5), готово.
$ ls -a degree/
. .. .git .gitignore LICENSE README.md
$ du -hs degree/
196K  degree/
```

Добавляю в полученный каталог файл кода модуля (это иллюстративный модуль тригонометрических функций, но получающих аргумент не в радианах, как в стандартном библиотечном модуле `math`, а в угловых градусах ... чисто тестовая задача):

```
$ cdt degree
$ cat degree.go
package degrees

import "math"

func d2r(dg float64) float64 {
    return dg / 180 * math.Pi
}
```

```
func Cos(dg float64) float64 {
    return math.Cos(d2r(dg))
}

func Sin(dg float64) float64 {
    return math.Sin(d2r(dg))
}

func Tan(dg float64) float64 {
    return math.Tan(d2r(dg))
}
```

Обратите внимание, что имя пакета `degrees` сознательно сделано отличающимся от имени файла, чтоб не порождать ненужных иллюзий. Помимо этого файла, стандартным образом создаю описание модуля:

```
$ go mod init github.com/olej-tsil-51/degree
go: creating new go.mod: module github.com/olej-tsil-51/degree
go: to add module requirements and sums:
    go mod tidy
```

На этом наполнение локального GIT завершено. Завожу вновь добавленные файлы под контроль GIT:

```
$ git status
Текущая ветка: main
Эта ветка соответствует «origin/main».
```

Изменения, которые не в индексе для коммита:

```
(используйте «git add <файл>...», чтобы добавить файл в индекс)
(используйте «git restore <файл>...», чтобы отменить изменения в рабочем каталоге)
    изменено:      degree.go
    изменено:      go.mod
```

индекс пуст (используйте «git add» и/или «git commit -a»)

```
$ git add .
$ git status
Текущая ветка: main
Эта ветка соответствует «origin/main».
```

Изменения, которые будут включены в коммит:

```
(используйте «git restore --staged <файл>...», чтобы убрать из индекса)
    изменено:      degree.go
    изменено:      go.mod
```

Зафиксируем произведенные изменения:

```
$ git commit -m 'module degrees'
[main 6b3352a] module degrees
 2 files changed, 2 insertions(+)
```

И остаётся только синхронизировать с сетевым GitHub:

```
$ git push
Username for 'https://github.com': olej-tsil-51
Password for 'https://olej-tsil-51@github.com':
Перечисление объектов: 7, готово.
Подсчет объектов: 100% (7/7), готово.
При сжатии изменений используется до 40 потоков
Сжатие объектов: 100% (4/4), готово.
Запись объектов: 100% (4/4), 380 байтов | 380.00 КиБ/с, готово.
Всего 4 (изменений 2), повторно использовано 0 (изменений 0), повторно использовано пакетов 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
```


To <https://github.com/olej-tsil-51/degree.git>
1d5fd2c..6b3352a main -> main

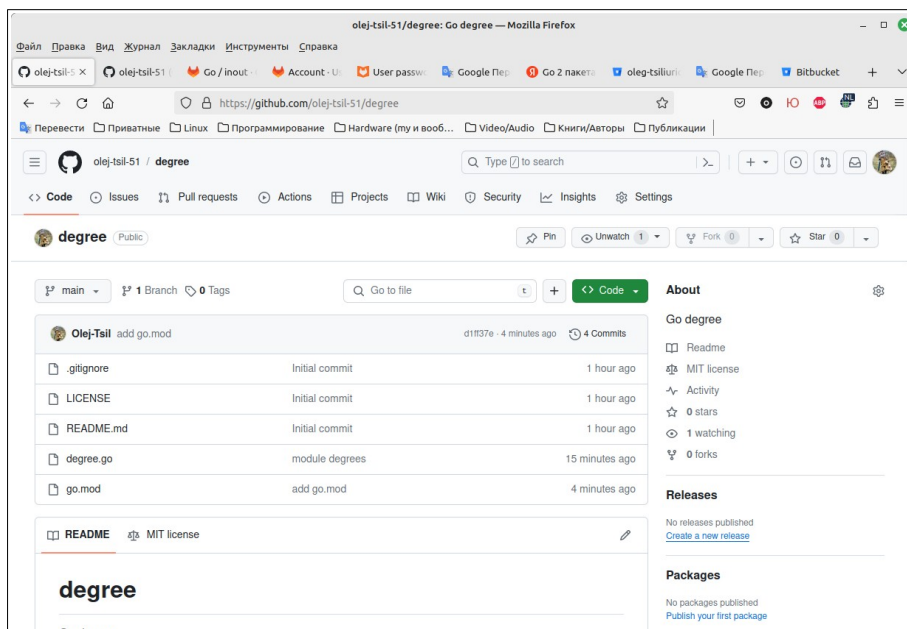


Рис. 1.8 Заполненный репозиторий

На этом работу с модулем размещённым в сети закончили, и готовим исполнимое приложение, которое будет импортировать и использовать возможности нашего модуля: рядом (это нам ещё пригодится) с локальным каталогом GIT создаём каталог `main` (собственно, с произвольным именем, повторно делаю акцент на этом), и файл исходного кода в нём:

```
$ tree -L 2
```

```
.
├── degree
│   ├── degree.go
│   ├── go.mod
│   ├── LICENSE
│   └── README.md
└── main
    ├── go.mod
    ├── go.sum
    └── head.go
```

2 directories, 7 files

(Структура файлов показана в том виде, как она получится уже после завершения манипуляций по подготовке пакета `main`).

head.go

package main

```
import (
    "fmt"
    "github.com/olej-tsil-51/degree"
)

func main() {
    const угол = 30.
    fmt.Printf("синус %f градусов = %f\n", угол, degrees.Sin(угол))
}
```

Дальше нужно подготовить манифест модуля `main` (инициализировать):

```
$ go mod init degree
go: creating new go.mod: module degree
go: to add module requirements and sums:
    go mod tidy
$ cat go.mod
module degree

go 1.22.0
```

... и осуществить импорт используемого модуля из сетевого репозитория (который мы только-что подготовили). Вручную это делается так (прямым указанием URL):

```
$ go get github.com/olej-tsil-51/degree
go: downloading github.com/olej-tsil-51/degree v0.0.0-20240221053336-1d5fd2c9133f
go: added github.com/olej-tsil-51/degree v0.0.0-20240221053336-1d5fd2c9133f
```

После чего импортируемый модуль прописался в манифест вызывающего модуля (файл поменялся):

```
$ cat go.mod
module degree

go 1.22.0
```

```
require github.com/olej-tsil-51/degree v0.0.0-20240221053336-1d5fd2c9133f // indirect
```

А ещё лучше это делать так, чтобы исполняющая система Go сама нашла все требуемые из кода импорты:

```
$ go mod tidy
```

Файл манифеста при этом опять слегка меняется:

```
$ cat go.mod
module degree

go 1.22.0
```

```
require github.com/olej-tsil-51/degree v0.0.0-20240221053336-1d5fd2c9133f
```

Импортированному модулю присвоена отладочная версия: v0.0.0-20240221053336-1d5fd2c9133f

И в том и в другом случае указанных действий, модуль скачивается из сети (GitHub) локально вот

```
$ ls -o `go env GOPATH`/pkg/mod/github.com/olej-tsil-51/*133f
итого 16
-r--r--r-- 1 olej 272 фев 21 08:54 degree.go
-r--r--r-- 1 olej 49 фев 21 08:54 go.mod
-r--r--r-- 1 olej 1071 фев 21 08:54 LICENSE
-r--r--r-- 1 olej 19 фев 21 08:54 README.md
```

Дальше нам остаётся только проверить исполнение созданного приложения:

```
$ go run head.go
синус 30.000000 градусов = 0.500000
```

Или так:

```
$ go run .
синус 30.000000 градусов = 0.500000
```

Или так:

```
$ go build head.go
$ ./head
```

синус 30.000000 градусов = 0.500000

А это означает, что помещённый на GitHub модуль, теперь может использоваться любым пакетом `main` в любом конце мира!

И в завершение... Как вносить изменения и отлаживать код модуля? Неужели по каждому чиху проделывать `commit/push` в локальном GIT? Нет, мы можем на время выполнения отладочных работ перенаправить импорт из сетевого репозитория на локальный, выполнив (в каталоге вызывающего модуля):

```
$ go mod edit -replace github.com/olej-tsil-51/degree.git=./degree
```

После чего содержимое файла манифеста этого модуля `go.mod` ещё раз изменится:

```
$ cat go.mod
module degree
```

```
go 1.22.0
```

```
require github.com/olej-tsil-51/degree v0.0.0-20240221053336-1d5fd2c9133f
```

```
replace github.com/olej-tsil-51/degree.git => ../degree
```

И теперь импорт будет идти из локального GIT репозитория.

Источники информации

[1] Как писать Go-пакеты, 2019 — <https://habr.com/ru/companies/ruvds/articles/464289/>

[2] Standard library, справочник стандартной библиотеки GoLang — <https://pkg.go.dev/std>

[3] Определение модуля, файл `go.mod` —

<https://golang-blog.blogspot.com/2019/07/defining-module.html?ysclid=lsszemuura106305447>

[4] Введение в систему модулей Go, 27 авг 2018 — <https://habr.com/ru/post/421411/>

[5] Введение в модули, 03.12.2021 — <https://metanit.com/go/tutorial/5.2.php>

[6] Search for a package, контекстный поиск пакетов Go — <https://pkg.go.dev/>

[7] Публикация модулей —

https://ru.hexlet.io/courses/go-setup-environment/lessons/publishing/theory_unit?ysclid=lsvfoxgs91132724395

Новости от последних версий...

Почему нововведения последних версий GoLang вынесены сюда отдельным разделом, а не «вкраплены» естественным образом в основной текст? На то есть несколько оснований...

Текст этой книги складывался (дополнялся, расширялся) на протяжении более 10 лет... Поэтому попытаться «вшить» нововведения последних версий GoLang (особенно 1.11, 1.18, 1.22 ...) было бы эклектично, неразумно. Все такие нововведения и рассматриваются здесь отдельным разделом. Нововведения не отменяют прежних возможностей, в меру сил разработчики языка сохраняют преемственность снизу вверх. Нововведения накладываются сверх прежних возможностей, дополняя их.

Во-вторых, проект GoLang очень активно развивается от версии к версии. Некоторые новинки, которые коротко рассматриваем в этой части, находятся в динамике и могут ещё существенно меняться в ближайших версиях: некоторый аспект появляется в версии 1.N, а в версии 1.N+1 он, глядишь, а выглядит заметно по-другому ...

Примечание: В дополнение, и к первому и ко второму аргументам, могу уточнить, что нововведения попадающие в этот раздел, со временем, и при их стабилизации по форме в языке, всё-таки переносятся в соответствующие разделы книги.

Проверяем версию...

Новые возможности в Go нарастают с такой стремительностью, что порой за ними уже сложно следить... Иногда хотелось бы проследить совместимость своего кода с различными версиями Go. Посмотрим как это делается на простом примере возможности, которая уже упоминалась ранее — с версии 1.22 разрешена итерация `range` по целочисленному параметру (каталог архива `compare/new.vers`):

for.go

```
package main
```

```
func main() {  
    for i := range 10 {  
        print(i, " ")  
    }  
    println()  
}
```

В версии 1.22 этот пример, естественно, благополучно выполняется:

```
$ go run for.go  
0 1 2 3 4 5 6 7 8 9
```

Но совместимо ли это с исполняющей системой некоторой предыдущей версии? Проверим это:

```
$ go run -gcflags=-lang=go1.20 for.go  
# command-line-arguments  
./for.go:4:17: cannot range over 10 (untyped int constant): requires go1.22 or later (-lang was set to go1.20; check go.mod)
```

Отчётливо несовместимо!

Такая же проверка может делаться не только для выполнения приложения, но и для его компиляции:

```
$ go build -gcflags=-lang=go1.20 for.go  
# command-line-arguments  
./for.go:4:17: cannot range over 10 (untyped int constant): requires go1.22 or later (-lang was set to go1.20; check go.mod)
```

Дженерики

Go выпустили в свет 10 ноября 2009 года, и уже меньше чем через 24 часа появился
148

первый комментарий про использование дженериков (родовых определений). Но реально дженерики появляются в GoLang появляются только в версии 1.18, март 2022 года (через 13 лет). Но даже в документации к этому релизу было указано, что что их реализация ещё может претерпевать заметные изменения.

Дженерики — это способ написания кода, который не зависит от используемых конкретных типов. Функции и типы теперь могут быть написаны для использования любого набора типов.

Отношение к дженерикам остаётся противоречивым, цитата из [16]:

Нужно ли усложнять язык дженериками?

Вопрос дискуссионный. Мнения разделились. Как известно, язык Go изначально был заточен под максимальную простоту, и обобщение типов может усложнить читабельность кода. Многие противопоставляют Go языку Java, традиционно наполненному обобщениями различного рода, и дженерики — это как первый шаг в эту сторону.

...

С другой стороны, если надо написать универсальную библиотеку для каких-то универсальных целей, то придётся использовать `interface{}` или кодогенерацию, а это тоже в общем-то читабельности и надёжности не добавляет. Также необходимо отметить, что разработчики языка сделали всё возможное, чтобы дженерики выглядели и использовались как можно проще. Намного проще, чем в других языках.

Как бы там ни было, это ещё не устоявшаяся техника в языке ... А мы, тем временем, создадим наше первое приложение использующее дженерики: печать срезов массивов, независимо от типа элементов этого массива (каталог `generic`):

print_slice.0.go

```
package main

import "fmt"

func PrintSlice[T any](s []T) {
    for _, v := range s {
        fmt.Printf("%v ", v)
        //fmt.Printf("площадь = %.2f\n", многоугольник.square())
    }
    fmt.Println()
}

func main() {
    cpez_str := []string{"Hello", "world"}
    PrintSlice[string](cpez_str)
    cpez_int8 := []int8{9, 8, 7, 6, 5, 4}
    PrintSlice[int8](cpez_int8)
    cpez_float32 := []float32{9.0, 8.1, 7.2, 6.3, 5.4, 4.5}
    PrintSlice[float32](cpez_float32)
    cpez_complex128 := []complex128{9 + 0i, 8 + 1i, 7 + 2i, 6 + 3i}
    PrintSlice[complex128](cpez_complex128)
}
```

Параметризируемое определение функции теперь выглядит так: `func PrintSlice[T any](s []T) {...}` — где `T` это обобщённое обозначение типа, а `any` при этом типе указывает, что здесь может быть любой тип (новое встроенное определение `any` — это синоним пустого интерфейса `interface{}`)

Готовим исполнимое приложение:

```
$ go fmt print_slice.0.go
```

```
print_slice.0.go
$ go build -o print_slice print_slice.0.go
```

И его выполнение:

```
$ ./print_slice.0
Hello world
9 8 7 6 5 4
9 8.1 7.2 6.3 5.4 4.5
(9+0i) (8+1i) (7+2i) (6+3i)
```

Синтаксически корректный способ вызовов `PrintSlice` следовало бы записывать так как показано выше: `PrintSlice[string]()`, `PrintSlice[float32]()` ... Но GoLang идёт дальше, и во многих случаях (и здесь как-раз такой случай) компилятор может сам сделать вывод типа из переданных функции аргументов (это сильно напоминает то, как делается вывод типов в C++ в стандартах C++11/C++14). При этом вызовы дженерик функции могут быть записаны проще:

print_slice.go

```
...
func main() {
    cpez_str := []string{"Hello", "world"}
    PrintSlice(cpez_str)
    cpez_int8 := []int8{9, 8, 7, 6, 5, 4}
    PrintSlice(cpez_int8)
    cpez_float32 := []float32{9.0, 8.1, 7.2, 6.3, 5.4, 4.5}
    PrintSlice(cpez_float32)
    cpez_complex128 := []complex128{9 + 0i, 8 + 1i, 7 + 2i, 6 + 3i}
    PrintSlice(cpez_complex128)
}
```

Но вывод типов не всегда может быть сделан компилятором. Вот что пишут ([7]) относительно вывода типов на сегодня сами разработчики языка:

Механизм вывода типа сложен, но применять его просто: вывод типа либо происходит, либо нет. Если тип выводится, типы-аргументы можно опустить — тогда вызов параметризованных функций ничем не отличается от вызова обычных функций. Если вывод типа не происходит, в компиляторе выдаётся сообщение об ошибке — тогда мы можем просто указать необходимые типы-аргументы.

Параметризация типов также может использоваться и с ново образуемыми типами:

```
type Tree[T interface{}] struct {
    left, right *Tree[T]
    value      T
}

func (t *Tree[T]) Lookup(x T) *Tree[T] { ... }
```

То, что во всех примерах выше указывалось в скобках [...] за обозначением типа T — это ограничения типа (constraints, констрейнты), условия которым должен удовлетворять обобщённый тип T. Это ограничение типа может быть как любым привычным интерфейсом Go, а может быть интерфейсом, перечисляющим полный список типов, для которых интерфейс может быть использован:

```
type MyConstraint interface {
    int | int8 | int16 | int32 | int64
}
```

Смотрим следующий пример: поиск присутствия элемента в срезе (массиве):

exist.go

```

package main

import "fmt"

func existsInSlice[T comparable](val T, values []T) bool {
    for _, v := range values {
        if val == v {
            return true
        }
    }
    return false
}

func showIn[T comparable](val T, values []T) {
    fmt.Printf("%v in %v => %v\n", val, values, existsInSlice(val, values))
}

func main() {
    showIn("worlds", []string{"Hello", "world"})
    showIn(7, []int8{9, 8, 7, 6, 5, 4})
    showIn(8.3, []float32{9.0, 8.1, 7.2, 6.3, 5.4, 4.5})
    showIn(7 + 2i, []complex128{9 + 0i, 8 + 1i, 7 + 2i, 6 + 3i})
}

$ go build exist.go
$ ./exist
worlds in [Hello world] => false
7 in [9 8 7 6 5 4] => true
8.3 in [9 8.1 7.2 6.3 5.4 4.5] => false
(7+2i) in [(9+0i) (8+1i) (7+2i) (6+3i)] => true

```

Здесь в качестве ограничения типа указан встроенный интерфейс Go `comparable` — ограничивающий типы, для которых определены операторы сравнения на равенство и неравенство.

И, наконец, пример который мы уже упоминали выше при рассмотрении модулей — поиск большего из 2-х значений:

max.go

```

package main

import "fmt"
import "golang.org/x/exp/constraints"

func Max[T constraints.Ordered](a T, b T) T {
    if a > b {
        return a
    }
    return b
}

func main() {
    fmt.Println(Max("Hello", "world"))
    fmt.Println(Max(9, 4))
    fmt.Println(Max(4.5, 5.4))
    // fmt.Println(Max(8 + 1i, 7 + 2i))
}

```

Здесь ограничение типа (констрейнт) импортируется из соответствующего пакета, содержащего достаточно много частных констрейнтов на разные случаи жизни (как импортировался сам пакет обсуждалось в предыдущем разделе).

И в итоге:

```
$ go fmt max.go
max.go
$ go build max.go
$ ./max
world
9
5.4
```

А вот если вы раскомментируете последний вызов и попытаетесь сравнить комплексные числовые значения (стандартный пакет `math.cmplx`), то получите ошибку компиляции вида:

```
$ go build max.go
# command-line-arguments
./max.go:17:17: complex128 does not implement constraints.Ordered (complex128 missing in
~int | ~int8 | ~int16 | ~int32 | ~int64 | ~uint | ~uint8 | ~uint16 | ~uint32 | ~uint64 |
~uintptr | ~float32 | ~float64 | ~string)
```

... которая (ошибка) напоминает нам о том, что для комплексных числовых значений не определены соотношения больше-меньше (эти операции имеют смысл, например, для модулей комплексных векторов `cmplx.Abs()`, но не самих значений). И это убедительно показывает как работает механизм ограничения типа в дженериках.

Наконец, в завершение, рассмотрим какие ограничения типов предоставляет этот импортируемый пакет (его местонахождение в файловой системе мы нашли в предыдущем разделе):

```
$ grep ^type `go env GOPATH`/pkg/mod/golang.org/x/exp@v0.0.0-20230310171629-522b1b587ee0/constraints/constraints.go
type Signed interface {
type Unsigned interface {
type Integer interface {
type Float interface {
type Complex interface {
type Ordered interface {
```

Источники информации

- [1] Tutorial: Getting started with generics — <https://go.dev/doc/tutorial/generics>
- [2] Дженерики в языке Go, 2 июн 2021 — <https://habr.com/ru/company/karuna/blog/552944/>
- [3] Введение в использование дженериков в Golang, 2 мая 2022 — <https://golang-blog.blogspot.com/2022/05/generics-intro.html>
- [4] Дженерики в Go — подробности из блога разработчиков, 29.03.22 — <https://itnan.ru/post.php?c=1&p=657853>
- [5] Golang пощупаем дженерики, 13 фев 2022 — <https://habr.com/ru/post/651229/>
- [6] Дженерики могут замедлить ваш код на Go, 9 апр 2022 — <https://habr.com/ru/post/660007/>
- [7] Брифинг по дженерикам Go 1.18, 27 фев 2023 — <https://habr.com/ru/company/otus/blog/719262/>
- [8] Пакет constraints — <https://pkg.go.dev/golang.org/x/exp/constraints>

Параллельные линии развития, TinyGo

Всё предыдущее рассмотрение строилось вокруг основного мейнстрима развития языка Go — проекта GoLang, инициированного корпорацией Google, в рамках которого и был создан стандарт языка. (Упомянутый ранее проект GCCGO в рамках GCC тоже является, в некотором смысле, фронтэндом проекта GoLang). И всё дальнейшее рассмотрение позже, и примеры кода, мы будем проводить в рамках именно этой основной реализации.

Но это не единственная линия развития, и, несомненно, будут ещё появляться новые со своими особенностями и специальными сферами применения. Учитывая ту динамику, с которой нарастается объём общедоступных целевых пакетов в сетевых репозиториях Go.

Одним из таких, молодых²¹ но уже достаточно зрелых, проектов является TinyGo.

TinyGo — это свободный проект, с открытыми исходными кодами, компилятор языка Go, предназначенный для изготовления приложений для небольших систем, таких как микроконтроллеры, WebAssembly (wasm/wasi), и построения инструментов командной строки (CLI). Он реализован с использованием технологий проекта LLVM, а для поддержки языка используются библиотеки, которые предоставляются базовым проектом GoLang. TinyGo предоставляет альтернативный способ компиляции программ написанных на языке программирования Go.

Инсталляции TinyGo предоставляются для разных платформ: Linux, Windows, MacOS (все необходимые ссылки представлены в источниках информации). Для Linux установка крайне проста:

Скачиваем инсталляционный DEB пакет:

```
$ wget
https://github.com/tinygo-org/tinygo/releases/download/v0.30.0/tinygo_0.30.0_amd64.deb
--2024-02-05 16:25:21--
https://github.com/tinygo-org/tinygo/releases/download/v0.30.0/tinygo_0.30.0_amd64.deb
Распознаётся github.com (github.com)... 140.82.112.4
Подключение к github.com (github.com)|140.82.112.4|:443... соединение установлено.
HTTP-запрос отправлен. Ожидание ответа... 302 Found
...
Сохранение в: 'tinygo_0.30.0_amd64.deb'
...
```

```
$ ls -l tinygo_0.30.0_amd64.deb
-rw-rw-r-- 1 olej olej 126469036 сен 21 16:38 tinygo_0.30.0_amd64.deb
```

Инсталляция делается стандартным способом:

```
$ sudo dpkg -i tinygo_0.30.0_amd64.deb
Выбор ранее не выбранного пакета tinygo.
(Чтение базы данных ... на данный момент установлено 587543 файла и каталога.)
Подготовка к распаковке tinygo_0.30.0_amd64.deb ...
Распаковывается tinygo (0.30.0) ...
Настраивается пакет tinygo (0.30.0) ...
```

В итоге получаем работающий компилятор:

```
$ which tinygo
/usr/local/bin/tinygo
$ tinygo version
tinygo version 0.30.0 linux/amd64 (using go version go1.21.3 and LLVM version 16.0.1)
```

Как уже упоминалось, TinyGo — это проект с открытыми исходными кодами, и те, кто не доверяют бинарным инсталляциям, или кто хочет проверить свои силы, могут собрать компилятор самостоятельно из исходных кодов (это, в данном случае, не совсем так

²¹ Судя по GIT-репозиторию исходных кодов проекта, он ведёт начало от 2018-2019 годов.

просто, но все инструкции предоставляются в репозитории):

```
$ git clone https://github.com/tinygo-org/tinygo.git
```

```
Клонирование в «tinygo»...
```

```
remote: Enumerating objects: 30678, done.
```

```
remote: Counting objects: 100% (534/534), done.
```

```
remote: Compressing objects: 100% (288/288), done.
```

```
remote: Total 30678 (delta 274), reused 395 (delta 228), pack-reused 30144
```

```
Получение объектов: 100% (30678/30678), 10.44 МБ | 1.71 МБ/с, готово.
```

```
Определение изменений: 100% (22400/22400), готово.
```

```
$ du -hs tinygo
```

```
21M    tinygo
```

TinyGo не является автономным или альтернативным к GoLang проектом — если вы рассмотрите исходные коды, загруженные показанным образом, то обнаружите, что весь проект TinyGo сам написан на языке Go и компилируется с помощью GoLang.

TinyGo экспериментально показал высокую совместимость по синтаксису языка, на кодах тех же примеров, которые мы раньше рассматривали в тексте. Мы можем произвести сравнительное рассмотрение GoLang и TinyGo на одних и тех же (для чистоты эксперимента) файлах исходных кодов своих приложений:

```
$ time go build -o fibo_golang fibo_go.go
```

```
real    0m0,206s
```

```
user    0m0,328s
```

```
sys     0m0,135s
```

```
$ time tinygo build -o fibo_gotiny fibo_go.go
```

```
real    0m3,441s
```

```
user    0m5,627s
```

```
sys     0m0,467s
```

Компиляция на TinyGo идёт значительно медленнее (здесь конкретно — в 16 раз ... не всегда эта разница столь разительна, но всегда это медленнее, и медленнее в несколько раз):

```
$ ls -l fibo_go*
```

```
-rw-r--r-- 1 olej olej    192 апр 21  2022 fibo_go.go
```

```
-rwxrwxr-x 1 olej olej 1467701 фев  5 20:17 fibo_golang
```

```
-rwxrwxr-x 1 olej olej  300744 фев  5 20:17 fibo_gotiny
```

```
$ file fibo_go*
```

```
fibo_go.go:  C source, ASCII text
```

```
fibo_golang: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, Go
BuildID=lfvyDkw_SN3g9gWgfJqL/Q_t_waY0gzNz_XeM9DY3/zTVAf3fYMeEcS9rJLm9t/aZ31lfKu-
MxDGvkSpOmd, with debug_info, not stripped
```

```
fibo_gotiny: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked,
with debug_info, not stripped
```

И тот и другой производят статически скомпонованные бинарные файлы, пригодные для непосредственного исполнения и проверки... но, обращаем сразу внимание, что исполнимый файл произведенный TinyGo короче здесь аналогичного результата GoLang в 4,88 раза. Более того, при относительно небольших объёмах вычислений TinyGo ... и заметно быстрее GoLang, что несколько приводит в недоумение:

```
$ time ./fibo_golang 40
```

```
165580141
```

```
real    0m1,202s
```

```
user    0m1,201s
```

```
sys     0m0,005s
```

```
$ time ./fibo_gotiny 40
```

```
165580141
```

```
real    0m0,743s
```

```
user    0m0,742s
```

```
sys     0m0,000s
```

Но при возрастании объёма вычислений эта разница заметно нивелируется:

```
$ time ./fibo_golang 45
1836311903
real    0m7,673s
user    0m7,662s
sys     0m0,020s
$ time ./fibo_gotiny 45
1836311903
real    0m6,995s
user    0m6,987s
sys     0m0,005s
```

Можно предположить, что на коротких программах TinyGo получает дополнительные преимущества за счёт размера загружаемого кода и скорости загрузки программы. Тем не менее, TinyGo подтверждает тем самым своё основное предназначение: создание небольших приложений для малых архитектурных конфигураций.

И здесь мы подходим к самому главному обстоятельству, без которого можно было бы и не заводить эту отдельную главу о альтернативном компиляторе TinyGo — поддерживаемые им аппаратные платформы. Вот тот полный список множества аппаратных платформ, в бинарный код для которых на сегодня может компилировать ваш исходный код на языке Go компилятор TinyGo²²:

```
$ tinygo targets
ae-rp2040
arduino-leonardo
arduino-mega1280
arduino-mega2560
arduino-mkr1000
arduino-mkrwifi1010
arduino-nano-new
arduino-nano
arduino-nano33
arduino-zero
arduino
atmega1284p
atsame54-xpro
attiny1616
badger2040
bluemicro840
bluepill-clone
bluepill
challenger-rp2040
circuitplay-bluefruit
circuitplay-express
clue-alpha
clue
cortex-m-qemu
d1mini
digispark
esp32-c3-devkit-rust-1
esp32-coreboard-v2
esp32-mini32
esp32
esp32c3-12f
esp32c3
esp8266
feather-m0-express
feather-m0
```

²² Здесь, рискуя навлечь на себя гнев и редактора и издателя, я всё-таки рискну привести полный и впечатляющий список микроконтроллеров, для того, чтобы заинтересованный читатель, возможно, мог найти (или, напротив, не найти) интересующую его платформу без необходимости заранее устанавливать TinyGo.

feather-m4-can
feather-m4
feather-nrf52840-sense
feather-nrf52840
feather-rp2040
feather-stm32f405
gameboy-advance
gemma-m0
gnse
gobadge
gopher-badge
gopherbot
gopherbot2
grandcentral-m4
hifive1b
itsybitsy-m0
itsybitsy-m4
itsybitsy-nrf52840
kb2040
lgt92
lorae5
m5stack-core2
m5stack
m5stamp-c3
m5stick-c
macropad-rp2040
maixbit
makerfabs-esp32c3spi35
matrixportal-m4
mch2022
mdbt50qrx-uf2
metro-m4-airlift
microbit-s110v8
microbit-v2-s113v7
microbit-v2
microbit
nano-33-ble-s140v6-uf2
nano-33-ble-s140v7-uf2
nano-33-ble-s140v7
nano-33-ble
nano-rp2040
nicenano
nodemcu
nrf52840-mdk-usb-dongle
nrf52840-mdk
nrf52840-s140v6-uf2
nrf52840-s140v7-uf2
nucleo-f103rb
nucleo-f722ze
nucleo-l031k6
nucleo-l432kc
nucleo-l552ze
nucleo-wl55jc
p1am-100
particle-3rd-gen
particle-argon
particle-boron
particle-xenon
pca10031
pca10040-s132v6
pca10040
pca10056-s140v6-uf2
pca10056-s140v7

```
pca10056
pca10059
pico
pinetime
pybadge
pygamer
pyportal
qtpy-rp2040
qtpy
reelboard-s140v7
reelboard
riscv-qemu
rp2040
simavr
stm32f469disco
stm32f4disco-1
stm32f4disco
swan
teensy36
teensy40
teensy41
thingplus-rp2040
trinket-m0
trinkey-qt2040
tufty2040
wasi
wasm
waveshare-rp2040-zero
wioterminal
x9pro
xiao-ble
xiao-esp32c3
xiao-rp2040
xiao
```

Всего же их:

```
$ tinygo targets | wc -l
132
```

И этот обширный список расширяется от версии к версии проекта...

Для компиляции в код нужной вам платформы используется такая форма команды:

```
$ tinygo build -o main.wasm -target arduino-nano main.go
```

Или, например:

```
$ tinygo build -o main.wasm -target wasm main.go
```

(Если вы вспомните, то GoLang обеспечивает кросс-компиляцию, в поддерживаемые ним платформы, совсем другим образом — переопределением переменных окружения компилятора: GOARCH и GOOS.)

Среди поддерживаемых платформ присутствуют и совсем микроминиатюрные конструкции, типа семейства широко распространённых контроллеров Arduino, с процессорами семейства AVR! Что особенно интересно для учебных целей. Для загрузки полученного кода в контроллеры Arduino вам, правда, понадобится дополнительная программа avrdude, но она присутствует в стандартном репозитории Linux, и её установка не составит проблем:

```
$ aptitude search avrdude
```

p	avrdude	- утилита для программирования микроконтроллеров
Atmel	AVR	
p	avrdude-doc	- документация для avrdude

Большим и несомненным преимуществом TinyGo будет то, что вы можете сначала

выполнять и отлаживать свою программу на инструментальном Linux десктопе, а только потом компилировать её в бинарный код целевой платформы. Вторым преимуществом будет та компактность кода, который генерируется для микроминиатюрных конфигураций.

У проекта TinyGo есть онлайн платформа для разработки прототипов приложений и непосредственно в браузере, для разных аппаратных платформ (<https://play.tinygo.org/>). Это может быть хорошим стартом в освоении программирования на Go. Вот как выглядит в этом инструменте программирование для AVR контроллеров Arduino:

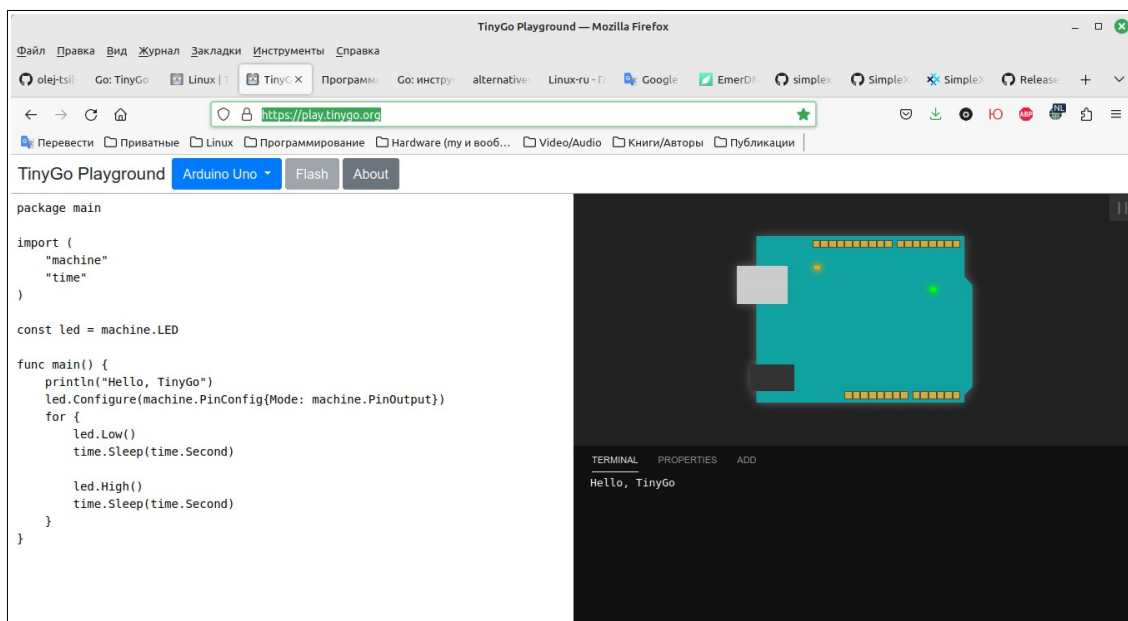


Рис. 1.9 Онлайн разработка для Arduino

Однако ..., при самом начальном ознакомлении с TinyGo, при компиляции примеров кода, предлагаемых во множестве по ресурсам TinyGo для микроминиатюрных конфигураций, вас могут ожидать досадные сюрпризы. Для их разрешения есть смысл остановиться подробнее. Вот самый распространённый образец из их примеров кода:

led.go

```
package main

import (
    "machine"
    "time"
)

func main() {
    led := machine.LED
    led.Configure(machine.PinConfig{Mode: machine.PinOutput})
    for {
        led.Low()
        time.Sleep(time.Millisecond * 1000)
        led.High()
        time.Sleep(time.Millisecond * 1000)
    }
}
```

При компиляции возникает такого сорта ошибка (у вас в деталях она может отличаться, но это всё-же будет терминальная ошибка):

```
$ tinygo build led.go
# command-line-arguments
led.go:9:20: undefined: machine.LED
```

И напрашивается естественный вопрос: «Откуда запрашивается импорт пакета `import`

"machine", если TinyGo, как уже отмечалось, пользуется пакетами GoLang, а во всём огромном множестве пакетов последнего нет такого пакета с таким именем?». Ответы на этот и родственные вопросы, а также ещё много любопытного, вам может подсказать изучение исходных кодов, которые мы загрузили ранее из сетевого репозитория исходных кодов проекта TinyGo:

1. TinyGo в стандартной инсталляции (из .deb пакета) устанавливается вот сюда (переменные окружения среды Go нам уже известны, кроме последней TINYGOROOT):

```
$ tinygo env
GOOS="linux"
GOARCH="amd64"
GOROOT="/usr/local/go"
GOPATH="/home/olej/go"
GOCACHE="/home/olej/.cache/tinygo"
CGO_ENABLED="1"
TINYGOROOT="/usr/local/lib/tinygo"
$ ls -o /usr/local/lib/tinygo/
итого 28
drwxr-xr-x  2 root  4096 фев  5 16:31 bin
drwxr-xr-x 11 root  4096 фев  5 16:31 lib
drwxr-xr-x  5 root  4096 фев  5 16:31 pkg
drwxr-xr-x 14 root  4096 фев  5 16:31 src
drwxr-xr-x  2 root 12288 фев  5 16:31 targets
```

2. Пакеты описания целевых архитектур "machine" (для разных поддерживаемых конфигураций) он импортирует вот отсюда:

```
$ ls -o /usr/local/lib/tinygo/src/machine | head
итого 1696
-rw-r--r--  1 root   571 сен 21 10:35 adc.go
-rw-r--r--  1 root  2057 сен 21 10:35 board_ae_rp2040.go
-rw-r--r--  1 root   651 сен 21 10:35 board_arduino.go
-rw-r--r--  1 root   544 сен 21 10:35 board_arduino_leonardo.go
-rw-r--r--  1 root  1544 сен 21 10:35 board_arduino_mega1280.go
-rw-r--r--  1 root  2758 сен 21 10:35 board_arduino_mega2560.go
-rw-r--r--  1 root  1715 сен 21 10:35 board_arduino_mkr1000.go
-rw-r--r--  1 root  2228 сен 21 10:35 board_arduino_mkrwifi1010.go
-rw-r--r--  1 root  2408 сен 21 10:35 board_arduino_nano33.go
$ ls -o /usr/local/lib/tinygo/src/machine | wc -l
236
```

3. Помимо прочего, там же, недалеко, вы найдёте множество примеров кодов — образцов для подражания, от авторов проекта:

```
$ ls -o /usr/local/lib/tinygo/src/examples
итого 148
drwxr-xr-x  2 root  4096 фев  5 16:31 adc
drwxr-xr-x  2 root  4096 фев  5 16:31 blinkm
drwxr-xr-x  2 root  4096 фев  5 16:31 blinky1
drwxr-xr-x  2 root  4096 фев  5 16:31 blinky2
drwxr-xr-x  2 root  4096 фев  5 16:31 button
drwxr-xr-x  2 root  4096 фев  5 16:31 button2
drwxr-xr-x  2 root  4096 фев  5 16:31 can
drwxr-xr-x  2 root  4096 фев  5 16:31 caninterrupt
drwxr-xr-x  2 root  4096 фев  5 16:31 dac
drwxr-xr-x  2 root  4096 фев  5 16:31 echo
drwxr-xr-x  2 root  4096 фев  5 16:31 echo2
drwxr-xr-x  2 root  4096 фев  5 16:31 empty
drwxr-xr-x  2 root  4096 фев  5 16:31 flash
drwxr-xr-x  2 root  4096 фев  5 16:31 gba-display
drwxr-xr-x  2 root  4096 фев  5 16:31 hid-joystick
drwxr-xr-x  2 root  4096 фев  5 16:31 hid-keyboard
drwxr-xr-x  2 root  4096 фев  5 16:31 hid-mouse
```

```

drwxr-xr-x 2 root 4096 фев 5 16:31 i2c-target
drwxr-xr-x 2 root 4096 фев 5 16:31 i2s
drwxr-xr-x 2 root 4096 фев 5 16:31 mcp3008
drwxr-xr-x 2 root 4096 фев 5 16:31 memstats
drwxr-xr-x 2 root 4096 фев 5 16:31 microbit-blink
drwxr-xr-x 2 root 4096 фев 5 16:31 pdm
drwxr-xr-x 2 root 4096 фев 5 16:31 pininterrupt
drwxr-xr-x 2 root 4096 фев 5 16:31 pwm
drwxr-xr-x 2 root 4096 фев 5 16:31 ram-func
drwxr-xr-x 2 root 4096 фев 5 16:31 rand
drwxr-xr-x 2 root 4096 фев 5 16:31 rtcinterrupt
drwxr-xr-x 2 root 4096 фев 5 16:31 serial
drwxr-xr-x 2 root 4096 фев 5 16:31 systick
drwxr-xr-x 2 root 4096 фев 5 16:31 temp
drwxr-xr-x 2 root 4096 фев 5 16:31 test
drwxr-xr-x 2 root 4096 фев 5 16:31 time-offset
drwxr-xr-x 2 root 4096 фев 5 16:31 uart
drwxr-xr-x 2 root 4096 фев 5 16:31 usb-midi
drwxr-xr-x 7 root 4096 фев 5 16:31 wasm
drwxr-xr-x 2 root 4096 фев 5 16:31 watchdog
$ ls -o /usr/local/lib/tinygo/src/examples | wc -l
38

```

Но далеко не каждый из этих примеров может быть скомпилирован для любой из этих архитектур — каждая из них имеет свои аппаратные особенности! И если вы придирчиво выберите пример, точно соответствующий архитектуре указанной в опции `-target` компилятора, то вы можете проделать компиляцию прямо из своего рабочего (произвольного) каталога:

```
$ tinygo build -target arduino `tinygo env TINYGOROOT`/src/examples/blink1/blink1.go
```

(Хотя то же самое вы можете проделать и откопировав исходный код используемого примера в свой рабочий каталог.)

Теперь вам будет сопутствовать успех:

```

$ ls -l blink1
-rwxr-xr-x 1 olej olej 87872 фев 12 21:22 blink1
$ file blink1
blink1: ELF 32-bit LSB executable, Atmel AVR 8-bit, version 1 (SYSV), statically
linked, with debug_info, not stripped

```

И вот мы получили бинарное приложение для микропроцессора AVR, без виртуализации памяти (MMU), с совершенно индивидуальной системой команд, про который мы, возможно, до сегодняшнего дня и не слышали... Дальше этот исполнимый код нужно только загружать в микропроцессор и тестировать. (Но это уже аппаратные подробности, которые выходят за пределы темы «программирование в Go».)

И последняя особенность TinyGo, которая может ввести в смущение при попытках его использования. При компиляции самого заурядного кода Go вы можете получить совершенно неожиданную ошибку:

```

$ tinygo build hello.go
error: requires go version 1.18 through 1.21, got go1.22
$ go version
go version go1.22.0 linux/amd64

```

О плотной интеграции проекта TinyGo с проектом GoLang, на базе которого он и построен, уже отмечено выше. Но TinyGo «готов» работать только с некоторым ограниченным подмножеством версий GoLang, которые он охотно и сообщает нам сообщением: от 1.18 до 1.21. На сегодня (начало февраля 2024 года), это соответствует версии TinyGo:

```

$ tinygo version
tinygo version 0.30.0 linux/amd64 (using go version go1.22.0 and LLVM version 16.0.1)

```


Это обстоятельство отмечается мельком в сопутствующей документации на GIT репозитории проекта TinyGo. И эти границы допустимых версий GoLang (как нижняя, так и верхняя) время от времени (при смене версий TinyGo) смещаются.

Но и сменить текущую версию GoLang под пригодную, на время работы с TinyGo, из числа установленных, не представляет труда с помощью управления альтернативами Linux, о чём было подробно рассказано ранее.

Источники информации

- [1] TinyGo - A Go Compiler For Small Places, ресурс проекта — <https://tinygo.org/>
- [2] Linux install guide — <https://tinygo.org/getting-started/install/linux/>
- [3] GIT-репозиторий исходного кода проекта — <https://github.com/tinygo-org/tinygo>
- [4] Онлайн платформа TinyGo для разработки кода — <https://play.tinygo.org/>
- [5] Go для встраиваемых систем и WebAssembly; русскоязычный ресурс проекта — <https://tinygo.ru/>
- [6] Документация на русском языке — <https://tinygo.ru/docs/>
- [7] Разрабатываем для Arduino и других микроконтроллеров — <https://habr.com/ru/companies/otus/articles/663118/>

Часть 2. Конкурентность и многопроцессорность

На момент создания Go и 30-40 лет до того подавляющее большинство находящихся в обиходе компьютеров были однопроцессорные, и редкие серверные образцы представляли собой конструкции с немногими (2-4) отдельными процессорными расположенными раздельно на системной плате... Относительно архитектуры таких редких серверных экземпляров и сложился термин и обозначение SMP (Symmetric MultiProcessing). В этот период превалирования однопроцессорных архитектур, на протяжении нескольких десятков лет, параллелизм (в практических, не теоретических целях) рассматривался как модель выполнения, когда параллельные ветви развития вычисления чередовались во времени, конкурируя за доступ к единственному процессу, и вытесняя друг-друга в пассивное состояние ожидания процессора (квази-параллельное выполнение). Квази-параллельное выполнение не увеличивает (и даже, зачастую, несколько несколько снижает) эффективность использования вычислительных мощностей оборудования, но часто благотворно отображается на логической ясности решаемой проблемы.

История развитие многоядерных процессоров (несколько автономных процессоров, на едином кристалле, под единой крышкой чипа) начинается с 2000-х годов (в 1999 году анонсирован первый двухъядерный процессор в мире — IBM Power4, предназначенный для промышленных серверов). А массовый выпуск 2-х ядерных и начало их широкой доступности относится только к 2005 году. Собственно, история многоядерных процессоров начинается тогда, когда их производители, из своих коммерческих интересов, поняли (опережая понимание этого рынком), что эра повышения производительности за счёт повышения тактовой частоты — закончилась. Закончилась потому, что при постоянном уменьшении масштаба литографии, в технологическом процессе начали сказываться фундаментальные квантовомеханические ограничения. И единственным путём увеличения производительности остался путь увеличения числа ядер на одной процессорной подложке.

Но если у вас будет даже 200 или 1000 процессоров (ядер) — это вовсе ещё не значит, что ваша программа механически станет хоть на йоту быстрее. Для этого программное обеспечение должно быть написано в специальных техниках, позволяющих использовать под задачу более одного процессора. Таким образом, стремительный прогресс в технологии железа потребовал радикально пересмотреть принципы проектирования программного обеспечения. Но осознание этого пришло не сразу, и в среде практиков относится, пожалуй, к 2010-2012 годам (лет на 5-7 позже начала производства многоядерных процессоров).

Но прежде нужно внимательно посмотреть на то, как и во что отдельные процессоры отображаются в операционной системе Linux ... раз уж мы говорим конкретно о Linux, хотя всё то же, в основной мере, будет относиться и ко всем POSIX, UNIX-like операционным системам.

Процессоры в Linux

Основная информация о процессоре формируется ядром Linux в квази-файловой системе `procfs`, рассмотрим что там есть для простейшего (который на сегодня можно найти) процессора (для более развитых моделей вывода может быть во много раз больше):

```
$ cat /proc/cpuinfo
```

```
processor : 0
vendor_id : GenuineIntel
cpu family      : 6
model           : 158
model name      : Intel(R) Celeron(R) CPU G3930 @ 2.90GHz
stepping       : 9
microcode      : 0xea
cpu MHz         : 800.058
cache size     : 2048 KB
physical id    : 0
siblings       : 2
core id        : 0
cpu cores      : 2
apicid         : 0
initial apicid : 0
fpu            : yes
fpu_exception  : yes
cpuid level    : 22
wp             : yes
flags          : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm
constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid
aperfperf pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg cx16 xtpr pdcm
pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave rdrand lahf_lm abm
3dnowprefetch cpuid_fault invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi
flexpriority ept vpid ept_ad fsgsbase tsc_adjust smep erms invpcid mpx rdseed smap
clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm arat pln pts hwp hwp_notify
hwp_act_window hwp_epp md_clear flush_l1d
bugs           : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds
swapgs itlb_multihit srbds
bogomips       : 5799.77
clflush size   : 64
cache_alignment : 64
address sizes   : 39 bits physical, 48 bits virtual
power management:
```

```
processor : 1
vendor_id : GenuineIntel
cpu family      : 6
model           : 158
model name      : Intel(R) Celeron(R) CPU G3930 @ 2.90GHz
stepping       : 9
microcode      : 0xea
cpu MHz         : 800.062
cache size     : 2048 KB
physical id    : 0
siblings       : 2
core id        : 1
cpu cores      : 2
apicid         : 2
initial apicid : 2
fpu            : yes
fpu_exception  : yes
cpuid level    : 22
wp             : yes
flags          : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat
pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm
constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid
```

```

aperfmpperf pni pclmulqdq dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg cx16 xtptr pdcm
pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave rdrand lahf_lm abm
3dnowprefetch cpuid_fault invpcid_single pti ssbd ibrs ibpb stibp tpr_shadow vnmi
flexpriority ept vpid ept_ad fsgsbase tsc_adjust smep erms invpcid mpx rdseed smap
clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm arat pln pts hwp hwp_notify
hwp_act_window hwp_epp md_clear flush_l1d
bugs : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds
swapgs itlb_multihit srbds
bogomips : 5799.77
clflush size : 64
cache_alignment : 64
address sizes : 39 bits physical, 48 bits virtual
power management:

```

Видим, что объём информации более чем исчерпывающий в полноте, и именно оттуда (/proc/cpuinfo) черпают информацию команды-утилиты Linux (да и любой желающий в своём программном коде), которые предоставляют её в более компактном, читабельном виде:

\$ lscpu

```

Архитектура: x86_64
CPU op-mode(s): 32-bit, 64-bit
Порядок байт: Little Endian
Address sizes: 39 bits physical, 48 bits virtual
CPU(s): 2
On-line CPU(s) list: 0,1
Потоков на ядро: 1
Ядер на сокет: 2
Сокетов: 1
NUMA node(s): 1
ID производителя: GenuineIntel
Семейство ЦПУ: 6
Модель: 158
Имя модели: Intel(R) Celeron(R) CPU G3930 @ 2.90GHz
Степпинг: 9
CPU МГц: 800.067
CPU max MHz: 2900,0000
CPU min MHz: 800,0000
BogoMIPS: 5799.77
Виртуализация: VT-x
L1d cache: 64 KiB
L1i cache: 64 KiB
L2 cache: 512 KiB
L3 cache: 2 MiB
NUMA node0 CPU(s): 0,1
Vulnerability Itlb multihit: KVM: Vulnerable
Vulnerability L1tf: Mitigation; PTE Inversion
Vulnerability Mds: Mitigation; Clear CPU buffers; SMT disabled
Vulnerability Meltdown: Mitigation; PTI
Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass disabled via prctl and seccomp
Vulnerability Spectre v1: Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2: Mitigation; Full generic retpoline, IBPB conditional, IBRS_FW, STIBP disabled, RSB filling
Vulnerability Srbds: Mitigation; Microcode
Vulnerability Tsx async abort: Not affected
Флаги: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb
rdt scp lm constant_tsc art arch_perfmon pebs bts rep_good
nopl xtopology nonstop_tsc cpuid aperfmpperf pni pclmulqdq dtes64 monitor ds_cpl vmx est
tm2 ssse3 sdbg cx16 xtptr pdcm pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer
aes xsave rdrand lahf_lm abm 3dnowprefetch cpuid_fault invpcid_single pti ssbd ibrs ibpb
stibp tpr_shadow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust smep erms invpcid
mpx rdseed smap clflushopt intel_pt xsaveopt xsavec xgetbv1 xsaves dtherm arat pln pts
hwp hwp_notify hwp_act_window hwp_epp md_clear flush_l1d

```

Или так, кратко²³:

```
$ inxi -Cxxx
```

```
CPU:      Topology: Dual Core model: Intel Celeron G3930
          bits: 64 type: MCP arch: Kaby Lake rev: 9 L2 cache: 2048 KiB
          flags: lm nx pae sse sse2 sse3 sse4_1 sse4_2 ssse3 vmx bogomips: 11599
          Speed: 800 MHz min/max: 800/2900 MHz Core speeds (MHz): 1: 800 2: 800
```

Все 3 показанные команды диагностики относятся, естественно, к одному и тому же процессору, про который мы узнаём то, что важно для нашего дальнейшего рассмотрения: 64-битная архитектура x86_64 (AMD), 2 физических ядра (2 процессора), с размером кеш-памяти верхнего уровня (L3 cache) 2 MiB (важно для задач высокой производительности).

Процессоры, ядра и гипертриэдинг

Вообще то, каждое физическое ядро является полностью автономным независимым процессором, но промышленные сервера, например, могут ещё иметь конструктивно несколько (2-4) установленных чипов (микросхем, сокетов...) процессоров. Для целей нашего рассмотрения многоядерные и многопроцессорные системы не представляют существенной разницы²⁴.

Но если мы посмотрим диагностику чуть более развитого (чем выше) процессора, то увидим (почти наверняка для подавляющего большинства современных процессоров) что-то подобное следующему:

```
$ inxi -Cxxx
```

```
CPU:      Topology: Dual Core model: Intel Core i5 660
          bits: 64 type: MT MCP arch: Nehalem rev: 5 L2 cache: 4096 KiB
          flags: lm nx pae sse sse2 sse3 sse4_1 sse4_2 ssse3 vmx bogomips: 26601
          Speed: 1451 MHz min/max: N/A Core speeds (MHz): 1: 1451 2: 1502 3: 1478 4:
1716
```

Всё в порядке? 4 процессора (ядра)? Ура! ... Не совсем так. При более внимательном рассмотрении:

```
$ lscpu
```

```
Архитектура:          x86_64
CPU op-mode(s):      32-bit, 64-bit
Порядок байт:        Little Endian
Address sizes:        36 bits physical, 48 bits virtual
CPU(s):               4
On-line CPU(s) list: 0-3
Потоков на ядро:      2
Ядер на сокет:        2
Сокетов:              1
NUMA node(s):         1
ID производителя:     GenuineIntel
Семейство ЦПУ:        6
Модель:               37
Имя модели:           Intel(R) Core(TM) i5 CPU           660  @ 3.33GHz
Степпинг:             5
CPU МГц:              1996.993
BogoMIPS:             6650.38
Виртуализация:        VT-x
L1d cache:            64 KiB
L1i cache:            64 KiB
L2 cache:             512 KiB
L3 cache:             4 MiB
```

²³ Этой утилиты `inxi` в системе по умолчанию нет, но устанавливается стандартным способом из пакетной системы.

²⁴ Вообще то, многоядерные (несколько ядер процессора под одной крышкой) и многопроцессорные (конструктивно различные процессоры) отличаются с точки зрения «обвязки», работы с прерываниями, использования контроллеров прерываний APIC др. Но, с интересующей нас точки зрения производительности, эти отличия не вносят такой уж существенной разницы.

...

Здесь ядер (процессоров) только 2 («Ядер на сокет»), но каждый из этих процессоров имеет 2 потока выполнения (гипертрэддинг, hyper-threading, HT). Это второе, сопутствующее **физическому** ядру, **логическое** ядро при некоторых условиях может выполнять поток команд параллельно основному физическому ядру. Технологию HT производитель Intel впервые применил в Pentium 4, но по настоящему широко стали применять (после некоторого перерыва) только в линии Core i3/i5/i7 и в серверных процессорах. У производителя AMD есть своя, отличающаяся, технология подобная HT.

Но особенность HT состоит в том, что более-менее существенного повышения суммарной производительности пары (физическое + логическое ядра) наблюдается только при соблюдении определённых условий ... грубо состоящих в том, чтобы ядра в паре выполняли как можно более **разнородные** задачи. Но и в этом случае, максимальный выигрыш производительности 2-х ядер с HT оценивается самой Intel по максимуму в 10-30%, и то это наблюдается чаще всего в серверных задачах²⁵.

На некоторых задачах распределение вычислительной работы на все ядра (логические и физические) может не только не увеличивать, а снижать итоговую производительность, причём существенно, до 70% от максимальной, при условии использования только физических ядер (реальных процессоров)! (Такое наблюдается, в качестве тестинга, в майнинге криптовалют, в частности, проверено для Монего ... и, в общем случае, это, похоже, кроме самого характера задач, радикально зависит от размеров кэш-памяти верхних уровней и числа процессоров, которые эту память разделяют).

В связи с вышесказанным возникает потребность: а). выяснить какие процессоры (по порядковому номеру в `inxi` и аналогичных утилитах) могут быть отобраны как физические (реальные) ядра и б). как указать задаче какой набор процессоров (по номерам) использовать и как это сделать?

Примечание: Зачем отбирать только физические ядра? На некоторых задачах, как показывает эксперимент, использование числа CPU сверх числа физических ядер не увеличивает, а только уменьшает суммарную производительность — вот экспериментальные данные производительности (число хеш в секунду) майнинга криптовалюты Монего (только одна из наугад выбранных задач высокой нагрузки) от числа CPU на сервере с 20 физическими и 40 логическими ядрами:

10 CPU - 4666 H/s - 58.9%
 12 CPU - 5564 H/s - 70.2%
 16 CPU - 7174 H/s - 90.5%
 20 CPU - 7920 H/s - 100%
 40 CPU - 5754 H/s - 72.7%

Пик отчётливо наблюдается при при числе CPU совпадающем с числом физических ядер! Это связано по-видимому, с конкуренцией (при экстремально высокой нагрузке) CPU за другие ресурсы, в частности за кэш-память различных уровней: L3, L4 (если он есть) ...

Загадочная нумерация процессоров

*Если знаешь, где искать, то
 найдёшь скелет в любом шкафу.
 Чак Паланик «Бойцовский клуб»*

Смотрим ещё один серверный процессор, Xeon E3-1240 v3:

`$ inxi -Cxxx`

```
CPU:      Topology: Quad Core model: Intel Xeon E3-1240 v3
          bits: 64 type: MT MCP arch: Haswell rev: 3 L2 cache: 8192 KiB
          flags: avx avx2 lm nx pae sse sse2 sse3 sse4_1 sse4_2 ssse3 vmx bogomips:
54275
          Speed: 3592 MHz min/max: 800/3800 MHz Core speeds (MHz):
          1: 3592 2: 3592 3: 3592 4: 3592 5: 3592 6: 3592 7: 3592 8: 3592
```

²⁵ Во многих случаях гипертрэддинг — это только иллюстрация удачного способа от производителей продавать воздух.

Здесь системой диагностируются 8 процессоров. Какие из этих 8-ми могут быть отобраны так, чтобы использовать только физические (реальные) ядра? Здесь нам может помочь:

```
$ lscpu -e
CPU NODE SOCKET CORE L1d:L1i:L2:L3 ONLINE   MAXMHZ   MINMHZ
  0    0      0      0 0:0:0:0      да 3800,0000 800,0000
  1    0      0      1 1:1:1:0      да 3800,0000 800,0000
  2    0      0      2 2:2:2:0      да 3800,0000 800,0000
  3    0      0      3 3:3:3:0      да 3800,0000 800,0000
  4    0      0      0 0:0:0:0      да 3800,0000 800,0000
  5    0      0      1 1:1:1:0      да 3800,0000 800,0000
  6    0      0      2 2:2:2:0      да 3800,0000 800,0000
  7    0      0      3 3:3:3:0      да 3800,0000 800,0000
```

Пары ядер (логические относительно физических — колонка CORE) здесь группируются в пары так: 0 + 4, 1 + 5, 2 + 6, 3 + 7.

Важно: Может сложиться ложное представление что в оборудовании существуют некие различающиеся «физические» и некие «логические» ядра, которые, и те и другие Linux воспринимает как доступные ему процессоры. Это не так! Есть только симметричные пары «логических» ядер (за счёт гипертрэдинга), принадлежащие одному аппаратному «физическому» ядру. Это же хорошо можно видеть на диагностике сенсоров температурного нагрева процессоров (потому что их здесь только 4, но не 8):

```
$ sensors coretemp-isa-*
coretemp-isa-0000
Adapter: ISA adapter
Package id 0:  +80.0°C (high = +80.0°C, crit = +100.0°C)
Core 0:        +78.0°C (high = +80.0°C, crit = +100.0°C)
Core 1:        +80.0°C (high = +80.0°C, crit = +100.0°C)
Core 2:        +79.0°C (high = +80.0°C, crit = +100.0°C)
Core 3:        +75.0°C (high = +80.0°C, crit = +100.0°C)
```

Поэтому, если на этом процессоре мы захотим использовать только его физически разделённые ядра, то мы можем с одинаковым успехом взять либо 0, 1, 2, 3 либо 4, 5, 6, 7... но точно также можно использовать и 0, 1, 6, 7 или 4, 5, 2, 3 и т.д.

По показанному выше выводу `lscpu` может показаться что порядок нумерации процессоров следующий: сначала одна половинка пары логических ядер, относящихся к физическим, а только затем соответствующие им вторые половинки.

Но посмотрим уже рассматривавшийся ранее 2-ядерный процессор Intel Core i5:

```
$ inxi -Cxxx
CPU:           Topology: Dual Core model: Intel Core i5 660
               bits: 64 type: MT MCP arch: Nehalem rev: 5 L2 cache: 4096 KiB
               flags: lm nx pae sse sse2 sse3 sse4_1 sse4_2 ssse3 vmx bogomips: 26601
               Speed: 1324 MHz min/max: N/A Core speeds (MHz): 1: 1324 2: 1961 3: 1702 4:
1879
```

Здесь:

```
$ lscpu -e
CPU NODE SOCKET CORE L1d:L1i:L2:L3 ONLINE
  0    0      0      0 0:0:0:0      да
  1    0      0      0 0:0:0:0      да
  2    0      0      1 1:1:1:0      да
  3    0      0      1 1:1:1:0      да
```

Большая неожиданность! Здесь сначала нумеруется пара для 1-го физического ядра, а только затем — для 2-го, это полная противоположность тому что мы только что видели выше.

Точно такая же разносортица, от модели к модели, и у процессоров AMD, поэтому приводить детально её здесь избыточно (это замечено и обсуждается в обсуждениях в

сообществах).

И, наконец, посмотрим процессоры (2 физически отдельных сокета, процессорных чипов) сервера промышленного уровня DELL PowerEdge R420:

```
$ inxi -Cxxx
```

```
CPU:      Topology: 2x 10-Core model: Intel Xeon E5-2470 v2
          bits: 64 type: MT MCP SMP arch: Ivy Bridge rev: 4 L2 cache: 50.0 MiB
          flags: avx lm nx pae sse sse2 sse3 sse4_1 sse4_2 ssse3 vmx bogomips: 192104
          Speed: 2800 MHz min/max: 1200/3200 MHz Core speeds (MHz): 1: 2800 2: 2800
          3: 2800 4: 2800 5: 2800 6: 2800 7: 2800 8: 2800 9: 2800 10: 2800 11: 2801
          12: 2800 13: 2800 14: 2804 15: 2804 16: 2800 17: 2797 18: 2800 19: 2803
          20: 2800 21: 2801 22: 2801 23: 2800 24: 2800 25: 2802 26: 2801 27: 2800
          28: 2800 29: 2800 30: 2800 31: 2800 32: 2801 33: 2800 34: 2800 35: 2800
          36: 2800 37: 2800 38: 2800 39: 2800 40: 2800
```

Здесь 40 процессоров (ядер), которые нумеруются так (SOCKET — конструктивно отдельный чип SMP, CORE — номер ядра):

```
$ lscpu -e
```

CPU	NODE	SOCKET	CORE	L1d:L1i:L2:L3	ONLINE	MAXMHZ	MINMHZ
0	0	0	0	0:0:0:0	да	3200,0000	1200,0000
1	1	1	1	1:1:1:1	да	3200,0000	1200,0000
2	0	0	2	2:2:2:0	да	3200,0000	1200,0000
3	1	1	3	3:3:3:1	да	3200,0000	1200,0000
4	0	0	4	4:4:4:0	да	3200,0000	1200,0000
5	1	1	5	5:5:5:1	да	3200,0000	1200,0000
6	0	0	6	6:6:6:0	да	3200,0000	1200,0000
7	1	1	7	7:7:7:1	да	3200,0000	1200,0000
8	0	0	8	8:8:8:0	да	3200,0000	1200,0000
9	1	1	9	9:9:9:1	да	3200,0000	1200,0000
10	0	0	10	10:10:10:0	да	3200,0000	1200,0000
11	1	1	11	11:11:11:1	да	3200,0000	1200,0000
12	0	0	12	12:12:12:0	да	3200,0000	1200,0000
13	1	1	13	13:13:13:1	да	3200,0000	1200,0000
14	0	0	14	14:14:14:0	да	3200,0000	1200,0000
15	1	1	15	15:15:15:1	да	3200,0000	1200,0000
16	0	0	16	16:16:16:0	да	3200,0000	1200,0000
17	1	1	17	17:17:17:1	да	3200,0000	1200,0000
18	0	0	18	18:18:18:0	да	3200,0000	1200,0000
19	1	1	19	19:19:19:1	да	3200,0000	1200,0000
20	0	0	0	0:0:0:0	да	3200,0000	1200,0000
21	1	1	1	1:1:1:1	да	3200,0000	1200,0000
22	0	0	2	2:2:2:0	да	3200,0000	1200,0000
23	1	1	3	3:3:3:1	да	3200,0000	1200,0000
24	0	0	4	4:4:4:0	да	3200,0000	1200,0000
25	1	1	5	5:5:5:1	да	3200,0000	1200,0000
26	0	0	6	6:6:6:0	да	3200,0000	1200,0000
27	1	1	7	7:7:7:1	да	3200,0000	1200,0000
28	0	0	8	8:8:8:0	да	3200,0000	1200,0000
29	1	1	9	9:9:9:1	да	3200,0000	1200,0000
30	0	0	10	10:10:10:0	да	3200,0000	1200,0000
31	1	1	11	11:11:11:1	да	3200,0000	1200,0000
32	0	0	12	12:12:12:0	да	3200,0000	1200,0000
33	1	1	13	13:13:13:1	да	3200,0000	1200,0000
34	0	0	14	14:14:14:0	да	3200,0000	1200,0000
35	1	1	15	15:15:15:1	да	3200,0000	1200,0000
36	0	0	16	16:16:16:0	да	3200,0000	1200,0000
37	1	1	17	17:17:17:1	да	3200,0000	1200,0000
38	0	0	18	18:18:18:0	да	3200,0000	1200,0000
39	1	1	19	19:19:19:1	да	3200,0000	1200,0000

Чередование достаточно сложное (но понятное, чтобы его описывать словесно). Из него можно выделить 2 подмножества (из многих возможных, вообще то говоря) номеров
168

процессоров, когда будут использовать только физические ядра: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 или, с тем же успехом, могут быть использованы 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39.

Предварительные итоги по выбору процессоров для максимально производительного выполнения могут выглядеть так:

- Последовательная нумерация процессоров меняется в зависимости от производителя, семейства и даже модели процессорного чипа;
- Вместо того, чтобы предсказывать порядок нумерации догадками (что не всегда просто), или искать по документации (мне не удалось этого найти), её нужно, при необходимости, проверить экспериментально как показано выше;
- Хорошим решением может оказаться вообще отключение гипертрэдинга в процессорах, что можно сделать, зачастую, в установках BIOS вашего компьютера;
- Для истинно многопроцессорных конструкций (с несколькими процессорными чипами на плате) равномерное распределение нагрузки по процессорам может быть существенным ещё и с точки зрения наличия или отсутствия перегрева, что контролируется утилитой `sensors` (или другой из того же класса: `psensor`, `xsensor`, ...).

Управление процессорами Linux

*Два самых известных продукта,
созданных в Университете Беркли —
это UNIX и LSD. Это не может быть
просто совпадением.
Jeremy S. Anderson*

Описанная выше дифференциация CPU по принадлежности процессорам и ядрам имеют какой-либо смысл только тогда, когда мы умеем распределить вычислительную нагрузку между ними. Для этого предназначается аффинити маска.

Аффинити маска

Аффинити маска (маска сродства) — это битовая маска допустимых к использованию CPU, в POSIX API определяемая в `<bits/sched.h>` отдельным типом `cpu_set_t`: 0 бит — 1-й CPU, 1 бит — 2-й CPU и т.д. последовательно. Точный вид `cpu_set_t` определённый в `<bits/sched.h>` зависит от версий... но это и не особенно важно, хотя бы потому, что использовать структурность `cpu_set_t` в коде непосредственно присвоениями нельзя, для этого определено (`<sched.h>`) большое множество макросов такого вот вида (семантика каждого из них понятна из названия):

`CPU_SET`, `CPU_CLR`, `CPU_ISSET`, `CPU_ZERO`, `CPU_COUNT`, `CPU_AND`, `CPU_OR`,
`CPU_XOR`,

`CPU_EQUAL`, `CPU_ALLOC`, `CPU_ALLOC_SIZE`, `CPU_FREE`, `CPU_SET_S`, `CPU_CLR_S`,
`CPU_ISSET_S`, `CPU_ZERO_S`, `CPU_COUNT_S`, `CPU_AND_S`, `CPU_OR_S`, `CPU_XOR_S`,
`CPU_EQUAL_S`

Примечание: Включение макроопределения имени `_GNU_SOURCE` в **начало** любого файла исходного кода, использующего макросы `CPU_*` — **обязательно!**: `#define _GNU_SOURCE`

Аффинити маска может применяться (относиться) либо ко всему процессу в целом, либо к отдельному потоку ядра (`pthread_t`) этого процесса. В отношении процесса используются API `sched_getaffinity()` и `sched_setaffinity()` из `<sched.h>`. В отношении потока используются `pthread_getaffinity_np()` и `pthread_setaffinity_np()`, соответственно. Напишем 2 игрушечных приложения,

иллюстрирующих сказанное (эти приложения полезны для экспериментов далее, каталог архива `gorproc/concurrent`):

how-many-p.c

```
#define _GNU_SOURCE
#include <sched.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    cpu_set_t mask;
    if(sched_getaffinity(getpid(), sizeof(cpu_set_t), &mask) != 0 )
        printf("ошибка sched_getaffinity() %m\n", exit(1);
    printf("в системе процессоров: %d\n", CPU_COUNT(&mask));
    return 0;
}
```

how-many-t.c

```
#define _GNU_SOURCE
#include <sched.h>
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

int main(int argc, char *argv[]) {
    cpu_set_t mask;
    if(pthread_getaffinity_np(pthread_self(), sizeof(cpu_set_t), &mask) != 0)
        printf("ошибка pthread_setaffinity_np() %m\n", exit(1);
    printf("в системе процессоров: %d\n", CPU_COUNT(&mask));
    return 0;
}
```

И то, как это выполняется:

```
$ gcc how-many-p.c -Wall -o how-many-p
```

```
$ ./how-many-p
```

в системе процессоров: 40

```
$ gcc how-many-t.c -Wall -lpthread -o how-many-t
```

```
$ ./how-many-t
```

в системе процессоров: 40

Но в первом случае число процессоров (взведенных бит в слове аффинити маске) мы определяем из информации о **процессах**, то во втором — исходя из информации о **потоках ядра** системы.

И, наконец, для сравнения — что это полностью соответствует независимой диагностике утилитами системы Linux:

```
$ inxi -C
```

```
CPU:      Topology: 2x 10-Core model: Intel Xeon E5-2470 v2
          bits type: MT MCP SMP L2 cache: 50.0 MiB
          Speed: 2800 MHz min/max: 1200/3200 MHz Core speeds (MHz): 1: 2800 2: 2800
          3: 2800 4: 2800 5: 2801 6: 2800 7: 2800 8: 2800 9: 2800 10: 2800 11: 2800
          12: 2800 13: 2800 14: 2799 15: 2804 16: 2800 17: 2799 18: 2800 19: 2798
          20: 2798 21: 2795 22: 2802 23: 2802 24: 2800 25: 2800 26: 2802 27: 2800
          28: 2802 29: 2800 30: 2800 31: 2802 32: 2802 33: 2799 34: 2802 35: 2800
          36: 2800 37: 2800 38: 2800 39: 2800 40: 2800
```

На этом мы оставим вопрос интерфейса POSIX к аффинити маскам из программного кода (это предмет совсем другого разговора, касающийся модели параллелизма C/C++) и обратим внимание на то, что аффинити маской (по крайней мере для процесса в целом) можно управлять при запуске любого приложения в Linux — команда `taskset`:

```
$ taskset --help
```

```
Usage: taskset [options] [mask | cpu-list] [pid|cmd [args...]]
```

Show or change the CPU affinity of a process.

Options:

-a, --all-tasks	operate on all the tasks (threads) for a given pid
-p, --pid	operate on existing given pid
-c, --cpu-list	display and specify cpus in list format
-h, --help	показать эту справку
-V, --version	показать версию

The default behavior is to run a new command:

```
taskset 03 sshd -b 1024
```

You can retrieve the mask of an existing task:

```
taskset -p 700
```

Or set it:

```
taskset -p 03 700
```

List format uses a comma-separated list instead of a mask:

```
taskset -pc 0,3,7-11 700
```

Ranges in list format can take a stride argument:

```
e.g. 0-31:2 is equivalent to mask 0x55555555
```

Для более детальной информации смотрите `taskset(1)`.

Причём, как видно из этой краткой справки, команда `taskset` может как заказать аффинити маску любого приложения при его запуске, так и переопределить маску для уже выполняющегося (очевидно, долговременного) приложения по его PID.

Выглядит это как-то так:

```
$ taskset -c 0-3 ./how-many-p
```

в системе процессоров: 4

```
$ taskset -c 1,3,5,7,9 ./how-many-t
```

в системе процессоров: 5

Аналогично можно **изменять** аффинити маску и выполняющихся потоков ядра в рамках единого процесса. Но делается это изнутри программного кода используя POSIX API (вызов `pthread_getaffinity_np()`) ... но это не понадобится нам в дальнейшем рассмотрении. Отметим только то (не совсем очевидная вещь), что в многопоточном приложении каждому из потоков может быть предписано индивидуальной список дозволенных ему к использованию список физических процессоров, причём при этом разные потоки выполнения можно, в частности, «развести» на разные процессоры.

Как происходит диспетчирование в Linux

Для того, чтобы экспериментировать с многопроцессорностью, эффективностью выполнения, наблюдения временных отметок выполнения, и других подобных вещей и их толкований — необходимо вспомнить и хорошо понимать как процессы и потоки диспетчируются в операционной системе Linux. Далее следует краткое изложение (повторение) основ диспетчирования в многозадачной системе Linux ... кому это известно или не интересно — вы можете вполне опустить эту часть без ощутимого ущерба для дальнейшего обсуждения.

Примечание: внутри ядра Linux диспетчируются на самом деле только потоки, а для процесса это относится именно к главному или единственному его потоку (возникающем при запуске функции `main()`), но для пользователя это выглядит именно как диспетчирование запускаемых им процессов.

Планировщик Linux (диспетчер, шедюлер) — это часть ядра, отвечающая за распределение процессорного времени между процессами и потоками в многозадачной операционной системе. Диспетчер ядра предоставляет процессам три алгоритма планировщика: один для обычных процессов и два для потоков (процессов) реального времени.

Примечание: Словосочетание «реальное время» применительно к Linux не имеет никакого разумного отношения к понятию реальному времени или к выполнению с соблюдением требований реального времени. Здесь он означает только то, что при таких дисциплинах потоки и процессы планируются по более строгим алгоритмам, предусмотренным расширением стандарта POSIX для требований реального времени POSIX 1003.b.

Подавляющее большинство процессов, выполняющихся в Linux, выполняются как обычные процессы (так запускаются процессы по умолчанию), для них политика планирования обозначается константой политики `SCHED_OTHER`. Многие пользователи Linux твёрдо убеждены (из литературы), что их операционная система обеспечивает для обычных процессов (`SCHED_OTHER`), а таких 99.9% в работающей системе, вытесняющую мультизадачность (`preemptive multitasking`). На самом деле это не совсем так! В Linux для обычных процессов используется модифицированный (изобретённый в Linux) упрощённый метод, основанный на массиве всех выполняющихся процессов, каждому из которых выделяются различающиеся квоты времени на выполнение (зависящие от характера их активности) в течении одного периода диспетчирования ... о чём будет сказано чуть ниже.

Потоки и процессы реального времени могут иметь политики планирования `SCHED_FIFO` (обслуживание в порядке очереди поступления, кооперативная многозадачность) и `SCHED_RR` (`round-robin`, круговое обслуживание с вытеснением по таймеру, вытесняющая многозадачность). Любой поток или процесс имеет статический приоритет (приоритет реального времени), определяемый структурой:

```
struct sched_param {
    int sched_priority;
};
```

Примечание: Стандарт POSIX 1003.b (расширение реального времени) предусматривает для `struct sched_param` более сложное определение, но в Linux структура выродилась именно в такое единичное значение.

Приоритет и политику планирования для потока можно изменить и диагностировать системными вызовами (POSIX API):

```
#include <sched.h>
int sched_setscheduler( pid_t pid, int policy, const struct sched_param *p );
int sched_getscheduler( pid_t pid );
int sched_setparam( pid_t pid, const struct sched_param *p );
int sched_getparam( pid_t pid, struct sched_param *p );
int getpriority( int which, int who);
int setpriority( int which, int who, int prio);
```

Если `pid` в вызовах равен 0, то вызов относится к текущему процессу. Последние два вызова могут работать с процессом, группой процессов, или процессами конкретного пользователя.

Для потока аналогичные действия делаются вызовами:

```
#include <pthread.h>
int pthread_setschedparam( pthread_t __target_thread, int __policy,
                           const struct sched_param *__param );
int pthread_getschedparam( pthread_t __target_thread,
                           int *__restrict __policy,
                           struct sched_param *__restrict __param );
int pthread_setschedprio( pthread_t __target_thread, int __prio);
```

Для обычных процессов и потоков (`SCHED_OTHER`) статический приоритет может иметь только единственное значение 0, попытка установить другой значение будет приводить к

ошибке. Для процессов и потоков с планированием реального времени (`SCHED_FIFO` или `SCHED_RR`) статический приоритет может иметь значение в диапазоне 1 ... 99 (значение 0 недопустимо, в противовес `SCHED_OTHER`).

Статический приоритет, больший, чем 0, может быть установлен только у суперпользовательских процессов (выполняющихся с правами `root`), то есть только эти процессы могут иметь алгоритм планировщика `SCHED_FIFO` или `SCHED_RR` (но здесь вы можете воспользоваться установкой флага `SUID` для разрешений ординарному пользователю выполнять такие программы).

Если на процессоре выполняется активный процесс или поток с планированием реального времени (со статическим приоритетом больше 0), то ни один обычный процесс и никогда не получит вообще кванта времени на этом процессоре, до освобождения его выполняющимся потоком (завершения или переходом в заблокированное состояние). То же самое (не получит никогда кванта) относится и потокам реального времени, но с меньшим статическим приоритетом.

В свою очередь, обычные процессы, которых, как упоминалось, в системе подавляющее большинство, имеют дополнительный приоритет (`nice`-приоритет), на основе которых и производится их взаимное планирование (потоки не могут иметь самостоятельный `nice`-приоритет, а потоки с планированием `SCHED_OTHER` будут все иметь `nice`-приоритет своего процесса). Допускается 40 значений `nice`-приоритетов для `SCHED_OTHER` диспетчеризации, в диапазоне от -20 до +19 — максимальный приоритет -20 (см. далее).

Таким образом, в Linux может быть 140 (препроцессорная константа `MAX_PRIO`) приоритетов: 100 приоритетов реального времени и 40 `nice`-приоритетов.

Планирование `SCHED_OTHER` процессов в Linux выполняется строго по системному таймеру, на основании динамически пересчитываемых приоритетов. Каждому процессу с сформированным приоритетом `nice` на каждом периоде диспетчирования, в зависимости от этого значения приоритета процесса, назначается период активности (`timeslice`, квант) — 10-200 системных тиков, который динамически в ходе выполнения этого процесса может быть ещё расширен в пределах 5-800, в зависимости от характера интерактивности процесса (процессам, активно загружающим процессор, `timeslice` задаётся ниже, а активно взаимодействующим с пользователем, диалоговым — повышается). На этом построена схема диспетчеризации процессов в Linux сложности $O(1)$ - не зависящая по производительности от числа подлежащих планированию процессов, которой очень гордятся разработчики ядра Linux (возможно, что и вполне оправдано). Но это совсем другая система планирования, не имеющая прямого отношения к вытеснению!

Примечание: Новая система диспетчеризации $O(1)$ построена на основе 2-х очередей: очередь ожидающих выполнения процессов, и очередь отработавших свой квант процессов. Из первой из них выбирается поочередно следующий процесс на выполнение, и после выработки им своего кванта, он сбрасывается во вторую. Когда очередь ожидающих опустошается, очереди просто меняются местами: очередь отработавших становится новой очередью ожидающих, а пустая очередь ожидающих — становится очередью отработавших. Но всё это происходит так только при отсутствии процессов с установленной реалтайм диспетчеризацией (`RR` или `FIFO`), с ненулевым приоритетом реального времени. До тех пор, пока в системе будет находиться хотя бы один реалтайм процесс в состоянии готовности к выполнению (активный), ни один процесс нормального приоритета не будет выбираться на исполнение (на данном процессоре!).

Период системного тика определяется символьной препроцессорной константой ядра `HZ`, которая для большинства аппаратных процессорных архитектур равна 1000, а период системного тика, соответственно — 1 миллисекунда. Таким образом период активности (максимальный интервал непрерывного выполнения) для различных обычных процессов может находиться в диапазоне 10-1000 миллисекунд.

Описанная процедура приводит к тому, что, рано или поздно, любой процесс, с самым малым приоритетом (`nice=19`), планируемый по стандартному алгоритму планировщика с разделением времени (`SCHED_OTHER`) получит некоторый квант процессорного времени (не менее 10 системных тиков, 10 миллисекунд).

Приоритеты `nice`

Приоритеты `nice`, вообще то говоря, приоритетами в общепринятом смысле вовсе и не

является, а, напротив, означает «уступчивость»: ветви с большими числовыми значениями `nice` уступают большую часть времени диспетчирования соседям с меньшими (или отрицательными) значениями `nice`.

Приоритеты `nice` имеют смысл и значение только для обычных процессов. Изменить приоритет обычного процесса можно командой `nice` (с консоли, терминала), или программным вызовом (из кода):

```
#include <unistd.h>
int nice(int inc);
```

Диапазон параметра (значение `nice` процесса) находится в пределах -19 ... +20 — чем выше, тем выше «уступчивость», тем ниже приоритет выполнения. И в командном, и в API варианте, отрицательные значения параметра, для повышения приоритета, допускаются только с правами суперпользователя `root`. Ещё для работы с `nice`-приоритетами используются упоминавшиеся уже программные вызовы `getpriority()` и `setpriority()`.

Команда `nice` изменяет приоритет запускаемого вами приложения не путём установки значения, а корректировкой его относительно умалчиваемого (равновесного) значения:

\$ nice --help

Использование: `nice` [ПАРАМЕТР] [КОМАНДА [АРГ]...]

Запускает КОМАНДУ с изменённым значением `nice`, что влияет на приоритет при планировании. Если КОМАНДА не задана, печатает текущее значение `nice`. Значения `nice` лежат в диапазоне от -20 (наибольший приоритет) до 19 (наименьший).

Аргументы, обязательные для длинных параметров, обязательны и для коротких.

- n, --adjustment=N увеличить `nice` на целое число N (по умолчанию 10)
- help показать эту справку и выйти
- version показать информацию о версии и выйти

ЗАМЕЧАНИЕ: ваша оболочка может включать свою версию `nice`, которая, обычно, заменяет версию, описанную здесь. Пожалуйста, обратитесь к документации по оболочке, чтобы узнать, какие параметры она поддерживает.

Когда вы запускаете привычным путём (командой терминала, или мышкой из графического окружения рабочего стола) любую программу обычного приоритета (а запустить так программу с приоритетом реального времени вы не имеете возможности), то программа запускается с умалчиваемым приоритетом `nice` равным 10 (выбрано некоторое среднее значение диапазона).

Приоритеты реального времени

Для того, чтобы узнать возможный диапазон значений статических приоритетов (приоритетов реального времени) данного алгоритма планировщика, можно использовать функции:

```
#include <sched.h>
int sched_get_priority_max(int __algorithm);
int sched_get_priority_min(int __algorithm);
```

Это может понадобиться в переносимых в другие системы программах для того, чтобы они соответствовали стандарту POSIX.1b.

Период времени квантования (переключений), установленный для планирования с дисциплиной `SCHED_RR`, можно узнать вызовом:

```
int sched_rr_get_interval(__pid_t __pid, struct timespec *__t);
```

Зачастую период квантования установлен (для Intel x86) в 1 миллисекунду (параметр ядра `HZ=1000`), но это очень сильно меняется в зависимости от аппаратной платформы.

Приложению можно установить приоритет реального времени только переведя его в режим планирования по схеме реального времени: SCHED_FIFO или SCHED_RR. Это делается командой chrt:

```
$ chrt --help
```

Show or change the real-time scheduling attributes of a process.

Set policy:

```
chrt [options] <priority> <command> [<arg>...]
```

```
chrt [options] --pid <priority> <pid>
```

Get policy:

```
chrt [options] -p <pid>
```

Параметры политики:

-b, --batch	set policy to SCHED_BATCH
-d, --deadline	set policy to SCHED_DEADLINE
-f, --fifo	set policy to SCHED_FIFO
-i, --idle	set policy to SCHED_IDLE
-o, --other	set policy to SCHED_OTHER
-r, --rr	set policy to SCHED_RR (default)

Scheduling options:

-R, --reset-on-fork	set SCHED_RESET_ON_FORK for FIFO or RR
-T, --sched-runtime <ns>	runtime parameter for DEADLINE
-P, --sched-period <ns>	period parameter for DEADLINE
-D, --sched-deadline <ns>	deadline parameter for DEADLINE

Другие параметры:

-a, --all-tasks	operate on all the tasks (threads) for a given pid
-m, --max	show min and max valid priorities
-p, --pid	operate on existing given pid
-v, --verbose	display status information
-h, --help	показать эту справку
-V, --version	показать версию

Как уже должно быть понятно, сделать изменение политики можно только с административными правами root.

Так же, как и рассматриваемая раньше команда taskset, команда chrt позволяет либо изменить статический приоритет при запуске процесса, либо изменить приоритет динамически, «по ходу», для уже выполняющегося процесса (по его PID):

```
# chrt -r 50 bash
```

```
# ps
```

PID	TTY	TIME	CMD
3068	pts/2	00:00:00	sudo
3074	pts/2	00:00:00	bash
3100	pts/2	00:00:00	ps

```
# chrt -p 3074
```

```
pid 3074's current scheduling policy: SCHED_RR
```

```
pid 3074's current scheduling priority: 50
```

```
# chrt -r -p 5 3074
```

```
# chrt -p 3074
```

```
pid 3074's current scheduling policy: SCHED_RR
```

```
pid 3074's current scheduling priority: 5
```

Для запущенного процесса таким же образом (при наличии соответствующих прав)

можно произвольно произвольно менять и политику и приоритеты:

```
# chrt -f 20 bash
```

```
# ps
```

PID	TTY	TIME	CMD
3288	pts/2	00:00:00	sudo
3294	pts/2	00:00:00	bash
3320	pts/2	00:00:00	ps

```
# chrt -p 3294
```

```
pid 3294's current scheduling policy: SCHED_FIFO
```

```
pid 3294's current scheduling priority: 20
```

```
# chrt -r -p -r 10 3294
```

```
# chrt -p 3294
```

```
pid 3294's current scheduling policy: SCHED_RR
```

```
pid 3294's current scheduling priority: 10
```

```
# chrt -r -p -o 0 3294
```

```
# chrt -p 3294
```

```
pid 3294's current scheduling policy: SCHED_OTHER
```

```
pid 3294's current scheduling priority: 0
```

```
# exit
```

Источники информации

[1] Краткий экскурс в историю десктопных многоядерных процессоров, 11 января 2022 —

<https://i2hard.ru/publications/29369/>

[2] Олег Цильорик, «Параллелизм, конкурентность, многопроцессорность в Linux», 2014 —

<http://mylinuxprog.blogspot.com/2014/09/linux.html>

<http://flibusta.is/b/523510>

Параллелизм и многопроцессорность

Эволюция модели параллелизма

*– Мы не крысы ... Мы музыканты.
«Титаник» тонет, а мы – сидим на
палубе
и играем на виолончелях.
Андрей Рубанов, «Патриот»*

Параллельные процессы и fork

Исторически первые модели параллелизма были созданы на уровне процесса многозадачной операционной системы как единицы параллельного выполнения. И ранее использовалась возможность запуска из кода дочерних процессов — несколькими родственными системными вызовами группы `exec`: `execl()`, `execvp()`, `execle()`, `execv()`, `execvp()`, `execvre()`, а в некоторых операционных системах (QNX) и группы `spawn()`. Но настоящий «разгул» параллелизма начался с появления системного вызова `fork()` в операционных системах класса UNIX (POSIX совместимых).

Концепция ветвления процессов впервые описана в 1962 году Мелвином Конвеем, а к 1964 году относятся первые реализации, которые были заимствованы Томпсоном при реализации операционной системы UNIX, и позже была включена в стандарты POSIX как обязательное требование для систем этого класса совместимости.²⁶

Вызов `fork()` разветвляет текущий процесс на родительский (текущий) и дочерний (вновь созданный). В системах с виртуальной памятью (а это практически любая аппаратная архитектура на сегодня), за счёт механизма *copy-on-write* (COW), создание полной копии адресного пространства родительского процесса происходит без фактического копирования (просто переотражением нового виртуального адресного пространства на уже существующее физическое). А поэтому происходит создание нового процесса очень быстро.

Модель ветвления процессов `fork()` породила целую новую парадигму построения программных систем: клиент-серверную, причём с параллельными серверами. Она была использована во множестве крупнейших информационных продуктов.

Логика работы `fork()` следующая: сразу же после вызова у нас возникает дубликат вызвавшего (родительского) процесса. Единственная разница между ними (родительским и дочерним процессом) в этой точке ветвления в том, что вызов `fork()` возвратит: 0 — в дочернем процессе, и значение PID (process ID) нового дочернего процесса — в родительском процессе, и <0 — если в выполнении вызова произошли ошибки (неудача, не состоялся). Вот так (каталог `goproc/concurrent`), например, мы можем создать сколь угодно много процессов из одного запущенного командой:

forks.cc

```
#include <cstdlib>
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>
using namespace std;

int main(int argc, char *argv[]) {
    int numpar = (argc > 1 && atoi(argv[1]) > 0) ?
        atoi(argv[1]) : 1;
    pid_t pid[numpar];
    for(int i = 0; i < numpar; i++)
        switch (pid[i] = fork()) {
            case -1: perror("ошибка fork"), exit( EXIT_FAILURE );
            case 0: // дочерний процесс
```

²⁶ Автор считает, что именно включение системного вызова `fork()` явилось той «серебряной пулей», которая обеспечила системам UNIX/POSIX процветание на протяжении 50 лет до наших дней.

```

        cout << getpid() << " стартовал" << endl;
        sleep(1);
        cout << getpid() << " завершился" << endl;
        exit( EXIT_SUCCESS );
    default: // родительский процесс
        cout << "запустил: " << pid[i] << endl;
        continue;
    }
    for(int i = 0; i < numpar; i++)
        waitpid(pid[i], NULL, 0);
    cout << getpid() << " завершился" << endl;
    exit(EXIT_SUCCESS);
}

```

Приложение специально и сознательно написано с C++, хотя `fork()` — это стандарт POSIX языка C, но здесь мы подчёркиваем, что в этой части разницы нет, и решения переносимы. Посмотрим как это работает — 2 последовательных запуска (предпоследней строкой мы выводим идентификатор родительского процесса, который не совпадает с идентификатором ни одного дочернего):

```

$ ./forks 3
запустил: 95502
95502 стартовал
запустил: 95503
95503 стартовал
запустил: 95504
95504 стартовал
95502 завершился
95503 завершился
95504 завершился
95501 завершился

```

```

$ ./forks 3
запустил: 95506
запустил: 95507
95507 стартовал
95506 стартовал
запустил: 95508
95508 стартовал
95506 завершился
95507 завершился
95508 завершился
95505 завершился

```

Принципиально важно: во всех механизмах параллелизмов, и здесь тоже мы это видим в первый раз, и это принципиально, что порядок в котором стартуют и завершаются параллельные ветки непредсказуем. При повторяющихся запусках одного и того же приложения вы можете получать самую различающуюся последовательность сообщений! Иногда вперёд после точки ветвления может проскочить дочерний процесс, иногда — родительский. И любые предположения о порядке активации параллельных ветвей недопустимы — они ведут к грубым и, главное, крайне трудно диагностируемым ошибкам!

Иногда приходится слышать что механизм, основанный на параллельных процессах (механизм `fork()`), тяжеловесный в сравнении с другими (например, потоками ядра). Это не совсем так. Уже запущенный параллельный процесс автономно выполняется того же порядка эффективности как, например, и запущенный поток — и тот и другой диспетчируются (вытесняется и активируется) ядром Linux совершенно идентично. Разница состоит в том, что при переключении процессов, кроме переключения контекста (сохранения и восстановления регистров и указателей стека), происходит перераспределение страниц виртуальной памяти, и кэши становятся не синхронизированными, начинают заполняться заново.

Но главная причина этой отмечаемой тяжеловесности проявляется в другом. Особенностью этого способа организации параллелизма является то, что каждый из ветвящихся процессов (родительский и дочерний) выполняются в собственном изолированном (защищённом) адресном пространстве. Любой объект программы, например, та же переменная `int pumrag` в коде выше, представляет абсолютно иной объект в каждом процессе; логический (виртуальный) адрес этой переменной будет один и тот же и в дочернем и в родительском процессах, но отображаться они будут на совершенно разные переменные с физическими адресами памяти.

P.S. Если подходить с особой придирчивостью, то из-за механизма `copy-on-write` (реализуется практически в каждой современной операционной системе) сразу после выполнения `fork()` переменные `pumrag` в обоих процессах будут вообще отображаться в один и тот же физический адрес памяти — на чтение совпадающих значений это не повлияет. Но как только произойдёт 1-я попытка записи значения `pumrag` в любом из этих двух процессов — виртуальная страница памяти (размером 4Kb или 64Kb), в которую попадает изменяемая переменная `pumrag`, будет перерасмещена (аппаратными механизмами) в физической памяти, и скопирована из предыдущего размещения, и только после этого значение `pumrag` будет изменено. Но этот тонкий механизм реализации не влияет на поведение описываемой схемы.

Из-за размещения в этой модели каждой параллельной копии задачи в изолированном адресном пространстве, взаимодействие параллельных ветвей крайне затруднено — переменные одной ветви недоступны в другой (это и отмечают как «тяжеловесность» этой модели). С другой стороны это — залог повышенной надёжности: критические ошибки в коде одной ветви не могут повредить выполнение другой. Для преодоления этой сложности разработано множество механизмов межпроцессного взаимодействия (IPC, InterProcess Communication), стандартизованных стандартами POSIX и детально описанных во многих толстых книгах. Но эта детализация уже выходит за пределы намерений этой книги...

Такая модель параллелизма очень хорошо для серверных систем массового обслуживания, когда каждому подключающемуся клиенту создаётся его индивидуальная копия серверного процесса, а отдельные серверные процессы разных клиентов практически никак не взаимодействуют друг с другом. Именно в этом классе задач эта модель успешно развивалась не один десяток лет.

Потоки ядра и `pthread_t` POSIX

Потоки ядра появились, как утверждают, впервые, под названием «задача», ещё в IBM OS/360 где-то на уровне 1967 года. Но только с 2003 года популярность потоков радикально возросла, когда стало понятно, что дальнейший рост производительности будет происходить не за счёт частоты процессора, а за счёт числа ядер процессора (или числа процессоров в системе). С 1995 года весь API `pthread_t` вошёл в стандарты POSIX (Стандарт POSIX.1c, Threads extensions, IEEE Std 1003.1c-1995). С этого времени это (именно в таком виде) обязательная и неотъемлемая часть всякой UNIX-совместимой операционной системы, в том числе, естественно, Linux. В Windows существует подобный, несколько отличающийся по форме, механизм. Таким образом, механизм потоков ядра нашёл своё активное применение лет на 20 позже параллелизма процессов.

Поток, часто утверждают — это «лёгкая» единица планирования ядра. Переключение потоков (диспетчирование, шедулирование) осуществляется относительно легко, оно требует переключение контекста (сохранения и восстановления регистров и указателей стека), но не затрагивает виртуальную память, перераспределение страниц, являясь кэш-дружественным.

Как минимум один поток существует всегда в каждом процессе (даже если это сугубо последовательный процесс по типу HelloWorld). Если внутри процесса создаётся несколько потоков ядра, то они все совместно используют выделенную процессу память (а значит, доступны все переменные в области видимости потоковой функции), файловые дескрипторы, и другие ресурсы. Таким образом, ресурсами владеет процесс, а потоки обеспечивают активность.

P.S. Многие операционные системы, Linux в частности, даже в мультизадачном планировании (вытеснении-восстановлении) классических однопоточных приложений, на самом деле осуществляют диспетчирование не процессов, а потоков — тех главных и единственных потоков таких процессов. Во внутренних структурах ядра Linux просто нет структур данных, ответственных за активность процессов!

Для создаваемого параллельного процесса, рассмотренного в предыдущей главе, источником исполнимого кода при выполнении `fork()` является копия адресного пространства выполняющегося процесса (или, в гораздо более редких случаях запуска дочерним другим приложением вызовами группы `exec()`, загружаемый образ приложения из файла). Для создания параллельного потока ядра в качестве его источника исполнимого кода должна быть указана потоковая функция. Функции потока при создании (запуске) потока могут быть переданы параметры. Прототип функции потока строго регламентирован: `void* threadfunc (void* data)`, где `data` — указатель на блок данных любой сложности, но структурность которого должна быть согласована с создающей поток единицей (этот блок данных не типизирован).

Выпишем пример кода, максимально повторяющий (по функциональности) аналогичный параллельности процессов выше:

threads.cc

```
#include <iostream>
#include <pthread.h>
#include <unistd.h>
#include <cstdlib>
using namespace std;

void* threadfunc (void *data) {
    char msg[80];
    pthread_t tid = pthread_self();
    sprintf(msg, "%lu стартовал\n", tid);
    cout << msg;
    sleep(*(int*)data);
    sprintf(msg, "%lu завершился\n", tid);
    cout << msg;
    pthread_exit( NULL );
}

int main( int argc, char *argv[] ) {
    int numpar = (argc > 1 && atoi(argv[1]) > 0) ?
        atoi(argv[1]) : 1;
    pthread_t *tids = new pthread_t[numpar];
    const int delay = 1;
    for(int i = 0; i < numpar; i++) {
        pthread_create(&tids[i], NULL, threadfunc, (void*)&delay);
        cout << "запустил: " << tids[i] << endl;
    }
    for(int i = 0; i < numpar; i++)
        pthread_join(tids[i], NULL);
    cout << pthread_self() << " завершился" << endl;
    exit( EXIT_SUCCESS );
}
```

Отличие от аналогичного кода для процессов состоит в том, что здесь добавлена передача параметров функции потока, в данном случае константы длительности выполнения потока (что, в принципе, не нужно по смыслу, но иллюстративно).

Многопоточные приложения C/C++, использующие API `pthread_t` должны при сборке обязательно указываться с явным указанием библиотеки `libpthread.so`, иначе сборка завершится ошибкой:

```
$ g++ -O3 threads.cc -lpthread -o threads
```

```
$ ./threads 3
запустил: 140431306901248
140431306901248 стартовал
запустил: 140431298508544
140431298508544 стартовал
запустил: 140431290115840
140431290115840 стартовал
140431306901248 завершился
140431298508544 завершился
140431290115840 завершился
140431306905408 завершился
```

```
$ ./threads 3
запустил: 140685794973440
140685794973440 стартовал
запустил: 140685786580736
запустил: 140685778188032
140685778188032 стартовал
140685786580736 стартовал
140685794973440 завершился
140685786580736 завершился
140685778188032 завершился
140685794977600 завершился
```

Вопрос: зачем мы в потоковой функции предварительно формируем строку сообщения, и только затем выводим эту строку на терминал:

```
sprintf(msg, "%lu завершился\n", tid);
cout << msg;
```

Вместо того, чтобы просто записать вывод в поток, как мы это делаем для главного потока приложения в конце выполнения (и как это вообще привычно в C++ программировании):

```
cout << pthread_self() << " завершился" << endl;
```

Ответ: это ещё одна особенность параллельного программирования (и их ещё будет много дополнительно проявляться), когда точки переключения потоков непредсказуемы во времени (но совпадают с завершением одиночной операции). Во 2-м случае в строке записано 3 последовательных операции вывода в поток, они могут прерваться на любой из этих операций, и вы не получите построчного вывода, а получите смесь строк из разных потоков.

Вопрос: а при параллельном запуске процессов ранее это не происходит и не нужно учитывать?

Ответ: там всё точно так же, но с процессами всё масштабируется заметно медленнее, и вероятность таких наложений гораздо ниже. Но и там, по хорошему, это нужно делать!

Любые операции в параллельном исполнении требуют синхронизации!

Потоки C++

Потоки ядра в C++ реализованы точно так же как и в C — «под капотом» там находится всё тот же механизм `pthread_t` с тем же POSIX API. И, в принципе, в коде C++ можно использовать всё тот же код, использующий API `pthread_t`. Но с некоторых пор в C++ над этой моделью введена надстройка, основывающаяся на классе `std::thread`, заимствованная из очень популярного проекта Boost и апробированная там много лет.

Объект класса представляет собой один поток выполнения. Новый поток ядра начинает выполнение сразу же после построения объекта `std::thread`. В конструкторе объекта 1-м параметром указывается функция потока, или функтор — объект любого класса, релизующего метод `void operator()() const`. Гибкость этого подхода состоит в том, что если функция потока предполагает несколько параметров, то эти параметры можно передать указав их 2-м, 3-м, 4-м... и далее параметрами конструктора объекта `std::thread` (в подходе C для передачи набора параметров приходится конструировать

структуру из этих параметров, а затем передавать в `pthread_create()` растипизированный указатель `void*` на эту структуру). Это проще показать в коде — сделаем приложение в общих чертах похожее на предыдущие:

threads+.cc

```
#include <iostream>          // std::cout
#include <string>             // std::string
#include <sstream>            // std::ostringstream
#include <thread>             // std::thread
#include <vector>             // std::vector
#include <unistd.h>
using namespace std;

void run(string p) {
    ostringstream msg;
    auto tid = this_thread::get_id();
    msg << tid << " стартовал: " << p << endl;
    cout << msg.str();
    sleep(1);
    msg << tid << " завершился" << endl;
    cout << msg.str();
}

void fun1() {
    string t = __FUNCTION__;
    run(t);
}

void fun2(int x) {
    ostringstream msg;
    msg << __FUNCTION__ << " : " << x;
    run(msg.str());
}

void fun3(int x, float y, string z) {
    ostringstream msg;
    msg << __FUNCTION__ << " : " << x << " | " << y << " | " << z ;
    run(msg.str());
}

class object_task{
public:
    void operator()() const {
        string t = __FUNCTION__;
        run(t);
    }
};

int main( int argc, char *argv[] ) {
    int numpar = (argc > 1 && atoi(argv[1]) > 0) ?
        atoi(argv[1]) : 1;
    vector<thread> vthr;
    for(int i = 0; i < numpar; i++) {
        switch(i%4) {
            case 0: vthr.push_back(thread(fun1));
                    break;
            case 1: vthr.push_back(thread(fun2, 123));
                    break;
            case 2: vthr.push_back(thread(fun3, 321, 0.123, string("строка")));
                    break;
            case 3: vthr.push_back(thread(object_task()));
                    break;
        }
    }
}
```

```

    }
    cout << "запустил: " << vthr[i].get_id() << endl;
}
for(auto &v : vthr) v.join();
exit(EXIT_SUCCESS);          // синхронизация ожиданием
}

```

Мы добавили тривиальные варианты с по-разному переданными параметрами, но в остальном всё осталось так же:

```

$ ./threads+ 6
запустил: 140117162264320
140117162264320 стартовал: fun1
запустил: 140117153871616
140117153871616 стартовал: fun2 : 123
запустил: 140117145478912
запустил: 140117066905344
запустил: 140117058512640
140117066905344 стартовал: operator()
140117145478912 стартовал: fun3 : 321 | 0.123 | строка
140117058512640 стартовал: fun1
запустил: 140117050119936
140117050119936 стартовал: fun2 : 123
140117162264320 стартовал: fun1
140117162264320 завершился
140117153871616 стартовал: fun2 : 123
140117153871616 завершился
140117066905344 стартовал: operator()
140117066905344 завершился
140117058512640 стартовал: fun1
140117058512640 завершился
140117145478912 стартовал: fun3 : 321 | 0.123 | строка
140117145478912 завершился
140117050119936 стартовал: fun2 : 123
140117050119936 завершился

```

```

$ ./threads+ 6
запустил: 140199567718144
запустил: 140199559325440
140199567718144 стартовал: fun1
140199559325440 стартовал: fun2 : 123
запустил: 140199476590336
запустил: 140199468197632
140199476590336 стартовал: fun3 : 321 | 0.123 | строка
140199468197632 стартовал: operator()
запустил: 140199459804928
140199459804928 стартовал: fun1
запустил: 140199451412224
140199451412224 стартовал: fun2 : 123
140199567718144 стартовал: fun1
140199567718144 завершился
140199559325440 стартовал: fun2 : 123
140199559325440 завершился
140199468197632 стартовал: operator()
140199468197632 завершился
140199476590336 стартовал: fun3 : 321 | 0.123 | строка
140199476590336 завершился
140199459804928 стартовал: fun1
140199459804928 завершился
140199451412224 стартовал: fun2 : 123
140199451412224 завершился

```

Мы очередной раз наблюдаем то очень важное обстоятельство, что запуск-завершения

параллельных ветвей может изменяться самым причудливым от одного запуска к другому одного и того же приложения! Но в остальном, по показанному коду можно легко проследить что на уровне идеологии вариант C++ ничем не отличается от модели POSIX и C. Детальнее мы не будем углубляться в тему, потому что наша задача вовсе не детализация модели параллельных потоков C++.

Сопрограммы — модель Go

Go — один из самых удивительных языков, появившихся в последние 15 лет, и первый, нацеленный на программистов и компьютеры XXI века.
Марк Саммерфильд

Язык Go приятно сочетает в себе лаконизм и ясность C с такими вещами (например из Python) как множественные возвраты из функций, простота представления и лёгкость работы со строками и другие. Но главная «фишка» Go вовсе не в этом. Язык предназначен для поддержания параллельного выполнения (реального, а не квази-параллельного) на нескольких процессорах, ядрах (а правильнее на многих). Для этого любую функцию Go можно запустить выполняться в отдельной ветке (я буду стараться в этой части рассмотрения тщательно избегать термина «поток», чтобы избежать смешивания с понятием потока ядра операционной системы). Параллельно выполняющиеся ветви выполняются как сопрограммы, и могут обмениваться между собой синхронными сообщениями через двунаправленные каналы. Через каналы могут передаваться данные любых типов.

Таким образом, язык Go предвосхитил (к началу разработки в 2007 году это было вовсе не очевидным) тотальный переход всего компьютерного железа на многоядерность (многопроцессорность) и возможности параллельной обработки на многих процессорах. Заканчивалась 30-летняя эпоха наращивания производительности процессоров за счёт тактовой частоты, оставалась единственная (на сегодня понятная) возможность делать это за счёт числа ядер в процессоре. И если на сегодня уже не такая редкость у вас на столе 16 или 32 ядра, то в ближайшее время совершенно ординарной архитектурой для настольного компьютера может стать десятки или сотня ядер.

Параллелизм и многопоточное программирование традиционно имеют репутацию вещей сложных (и заслуженно). Авторы Go утверждают, что это происходит, во многом, из-за сложных конструкций, таких как `pthread_t`, и излишнего внимания к низкоуровневым деталям, таким как мьютексы, условные переменные, барьеры памяти. Интерфейсы более высокого уровня гораздо проще кодировать, и даже если при этом всё ещё остаются мьютексы и другие, но они остаются «под крышкой».

Одна из самых успешных моделей для обеспечения более высокого уровня языковой поддержкой параллелизма происходит от модели Хоара (Чарльза Энтони Ричарда Хоар, C. A. R. Hoare) взаимодействующих последовательных процессов (Communicating Sequential Processes, или CSP). Occam и Erlang — вот два хорошо известных языков программирования, которые происходят из CSP.

Язык Go, в этом смысле, в чём то наследует философию процедурного языка параллельного программирования Оссам, разработанному в начале 1980-х годов для программирования транспьютеров, но слишком сильно опередившим (как и сама идея транспьютера) своё время. Но ничто разумное не проходит бесследно...

Конкурентные примитивы Go представляют другую часть генеалогического дерева CSP, и здесь главный вклад — это мощное понятие каналов в качестве объектов первого уровня (first class objects). Опыт эксплуатации нескольких более ранних языков показал, что модель CSP хорошо вписывается в среду процедурного языка.

Параллелизм в Go

Одна из целей создания Go, формулируемая его авторами, как уже отмечалось выше — это эффективное выполнение многих конкурирующих ветвей (сопрограмм, горутин) в многопроцессорной среде. В GoLang, начиная с версии 1.1, встроен и совершенствуется новый механизм планировщика параллельных горутин, который реализовал Дмитрий Вьюков (в перечне литературы показаны подробные описания). Это планировщик по схеме M:N, где M — это **потoki ядра** (pthread_t в терминологии POSIX API), а N — это число горутин, которые на схематическом рисунке показаны как кружки, обозначенные G. Число M зачастую (по умолчанию) равно числу **процессоров**, но это вовсе не обязательно. А N параллельных горутин ($N > M$ или $N \gg M$) реализуются как лёгкие сопрограммы пространства пользователя (не ядра), реализующих модель **кооперативной** многозадачности. Горутини прикреплены к потокам, но закреплены не «глухо» — время от времени, при возникновении разбалансировки, горутини могут перераспределяться между потоками ядра.

Такая схема позволяет:

- Осуществлять лёгкое (быстрое) переключение между контекстами горутин, без вовлечения в эти процессы механизмов ядра и без аппаратного переключения в супервизорный режим работы процессора (кольцо защиты 0).
- Реализовать очень большое число параллельных горутин — порядка десятков или даже сотен тысяч (вопреки бытующим представлениям, в одном процессе-приложении Linux могут успешно работать и до нескольких тысяч параллельных потоков ядра, но с горутини это число возрастает ещё на 1-2 порядка).
- Обеспечить оптимальную балансировку нагрузки между всеми доступными процессорами на оборудовании, на котором выполняется приложение — при переносе приложения на другое оборудование, нагрузка балансируется автоматически.
- Предоставить (на будущее) возможность работы одновременно на весьма большом числе процессоров, исчисляемых десятками, или даже сотнями.

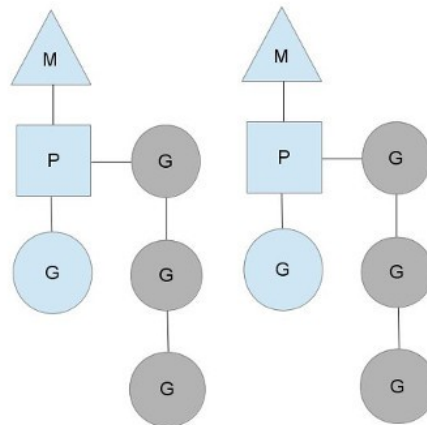


Рис. 2.1. Планировщик по схеме M:N

Мы не будем дальше углубляться в тонкости планирования, они подробно описаны в источниках, приведенных в конце текста, но вот такое описанное поверхностное понимание специфики планирования Go необходимо для эффективного использования горутин.

Детальнее эффекты планирования сопрограмм в Go мы будем наблюдать во всех последующих частях текста, связанных с понятием **масштабирования**.

Сопрограммы — как это выглядит

Go дает возможность создать новую ветвь (чтобы не говорить: поток) выполнения программы (goroutine — go-процедуру) с помощью выражения `go`. Выражение `go` запускает функцию в другой, заново созданной, go-процедуре (сопрограмме). Все go-процедуры в одной программе используют одно и то же адресное пространство.

Изнутри, go-процедуры действуют как подпрограммы, которые размножены по разным потокам в операционной системе и могут выполняться на различных процессорах (ядрах) SMP (все примеры этой главы размещаются в каталоге `gorgos` архива), практически тривиальный первый пример:

parm.go

```
package main

import (
    "time"
)

func test_parm(s string, i int, f float64) {
    print(i, " : ", s, " -> ", f, "\n")
}

func main() {
    матрица := [...]string{"первый", "второй", "третий"}
    for i := range матрица {
        go test_parm(матрица[i], i+1, float64(i))
    }
    time.Sleep(1000000000)
}
```

Сразу обращаем внимание на то, что в отличие от сложной семантики создания потока `pthread_create()` в POSIX API (C/C++), накладывающей жёсткие ограничений на прототип функции потока (вида `void* (*)(void*)`), с передачей блока параметров (`void*`) в функцию, механизм go-процедур не накладывает никаких ограничений на выполняемую сопрограммой функцию: параметры, их число, типизация, ...

Примечание: Попутно обратите внимание на полезный вызов `print()`, который не экспортируется ни из какого пакета, `print()` — это встроенная (builtin) функция Go, такая же как, например, `len()`. Но она может работать только с встроенными типами языка, в отличие, например, от `fmt.Printf()` и `fmt.Sprintf()`.

\$ cat Makefile

```
...
all: $(TASK)

%: %.go
    gccgo -g $< -o $@
    go build -o $@.gl -compiler gc $<
...
```

\$ make parm

```
gccgo -g parm.go -o parm
go build -o parm.gl -compiler gc parm.go
```

\$./parm

```
1 : первый -> +0.000000e+000
2 : второй -> +1.000000e+000
3 : третий -> +2.000000e+000
```

Указывается, что go-процедуры не потребляют много ресурсов (так предполагается). Примером простейшего, но законченного приложения Go, реализующего сопрограммы, может быть:

sleep.go

```
package main

import (
    "fmt"
    "time"
)

var ch1 chan int = make(chan int)

func foo(ch chan int) {
    for {
        fmt.Println(<-ch)
    }
}

func bar(ch chan int) {
    for i := 0; ; i++ {
        time.Sleep(1000000000)
        ch <- i
    }
}

func main() {
    go bar(ch1)
    go foo(ch1)
    time.Sleep(1000000000)
}

$ make sleep
gccgo -g sleep.go -o sleep
go build -o sleep.gl -compiler gc sleep.go

$ ./sleep
0
1
2
3
4
5
6
7
8
```

Функциональный литерал (которые в Go реализует как замыкания) могут быть полезны при использовании выражения `go`.

```
var g int
go func(i int) {
    s := 0
    for j := 0; j < i; j++ { s += j }
    g = s
} (1000)
```

Это уже демонстрирует некоторые минимальные элементы, заимствованные Go из функционального программирования ... дальше будет больше.

Возврат значений функцией

Вы, естественно, не можете вернуть результат выполнения функции сопрограммы, просто потому, что оператор `go` запускает выполнение функции отдельной параллельной сопрограммой, но никак не ожидает результата её завершения — продолжает своё

выполнение далее... И если эта запускающая функция `main()`, главная функция программы, то, если не предусмотреть какие-то механизмы синхронизации и ожидания, эта главная функция может так и завершиться не ожидая вообще выполнения или завершения запущенных нею сопрограмм. И вместе с ней завершится и вся программа, вместе со всеми выполняющимися в это время сопрограммами ... и это, наверное, далеко не то, что вы ожидали. Но об этом отдельно, позже...

Тем не менее функция сопрограммы вполне может вернуть результат своего выполнения, как одна из возможностей — в качестве побочного эффекта изменения своих параметров, передаваемых ей по ссылке. Простейший пример того показан в примере ниже:

result.go

```
package main

import (
    "fmt"
)

func main() {
    текст := [...]string{"первый", "второй", "третий"}
    done := make(chan bool)
    for _, v := range текст {
        print(v, " ")
    }
    print("\n")
    for i, _ := range текст {
        go func(s *string, f float64) {
            *s = fmt.Sprintf("%f", f)
            done <- true
        }(&текст[i], float64(i))
    }
    // wait for all goroutines to complete before continue:
    for _ = range текст {
        <-done
    }
    for _, v := range текст {
        print(v, " ")
    }
    print("\n")
}

$ ./result
первый второй третий
0.000000 1.000000 2.000000
```

Для подобных вещей очень успешно могут использоваться функциональные замыкания (closure) в качестве функции тела сопрограммы. Некоторые особенности использования замыканий в сопрограммах будут рассмотрены ниже.

Ретроспектива: сопрограммы в C++

Хотя это и не входит непосредственно в предмет наших интересов, в таком классическом языке как C++, в последних его стандартах (C++14, C++17, C++20) возник большой интерес к сопрограммам, о которых в контексте языка C++, в русскоязычной транскрипции, принято говорить как о корутинах. В более ранних редакциях C++, на протяжении предыдущих 30 лет, интерес к реализации корутин не наблюдалась, и механизмы реализации не предусматривались. Объяснить рост интереса к реализации корутин совершенно объясним широким распространением многопроцессорного (многоядерного) «железа» и недостаточной производительностью механизмов параллелизма, основанных на процессах или потоках ядра. Последний факт настолько интересен и не очевиден, что мы о нём будем говорить позже отдельно и обстоятельно...

Каналы

Логическое понятие канала (тип данных `channel`) используются для связи между любыми фрагментами кода, в частности между `go`-процедурами. Значения любого типа (включая другие каналы!) могут быть передано через канал. Каналы своими значениями: они могут быть сохранены в переменных и передаваться в функции, как и любые другие значения. При вызове функции, каналы, как параметры вызова, передаются по ссылке. Каналы как и любые данные типизированы: `chan int` отличается от `chan string`.

Каналы могут быть не буферизированные или с буферизацией: использование буфера определённой длины указывается во время создания канала:

```
канал1 = make(chan string)
канал2 = make(chan string, 100)
```

Каналы эффективны и потребляют мало ресурсов. Чтобы передать значение в канал, используется `<-` в качестве бинарного оператора (вида `chan <- data`). Чтобы получить сообщение из канала, используется тот же `<-` в качестве унарного оператора (вида `data = <- chan`). При вызове функций, каналы, в качестве параметров вызова, передаются **по ссылке**.

При использовании каналов в качестве параметров функции можете указать, предназначен ли канал только для отправки или получения значений. Эта может существенно повышать безопасность в крупных программах. Выглядит это указание направления при определении функции примерно так:

```
func retrans(pings <-chan string, pongs chan<- string) {
    ...
    msg := <-pings
    pongs <- msg
}
```

Здесь функция принимает один канал только для приема (`pings`) и второй только для отправки (`pongs`). При попытке получения значений, например, из канала в процессе `pongs` при компиляции возникнет ошибка. Но каналы могут быть и двунаправленными, в коде одной функции может и записываться и читаться один и тот же канал.

Библиотека `Go` (импортируемый пакет `sync`) предоставляет и мютексы (и некоторые другие примитивы синхронизации), но вы также можете использовать единую `go`-процедуру с открытым каналом для защиты данных. Вот пример использования управляющей функции для контроля доступа к единственной переменной:

```
type cmd struct {get bool; val int}
func manager(ch chan cmd) {
    var val int = 0
    for {
        c := <- ch
        if c.get { c.val = val; ch <- c }
        else { val = c.val }
    }
}
```

В этом примере один канал использован и на вход, и на выход. Это некорректно, если несколько `go`-процедур общаются с управляющей функцией одновременно: `go`-процедура, ждущая ответа от управляющей функции, может вместо ответа получить запрос от другой `go`-процедуры. Решением будет передать канал в качестве аргумента:

```
type cmd2 struct {get bool; val int; ch <- chan int}
func manager2(ch chan cmd2) {
    var val int = 0
    for {
        c := <- ch
        if c.get { c.ch <- val }
        else { val = c.val }
    }
}
```

```

    }
}

```

Для использования `manager2()`, дается канал:

```

func f4(ch <- chan cmd2) int {
    myCh := make(chan int)
    c := cmd2{ true, 0, myCh }    // Composite literal syntax.
    ch <- c
    return <-myCh
}

```

Канал может создаваться как буферизированный, с указанием ёмкости буфера (числовой параметр создания `N`). Отправитель в буферизированный канал блокируется, когда буфер заполнен (`N` сообщений). Отправитель в не буферизированный канал блокируется сразу же после единственной записи (`N` ведь равно `1`), и до тех пор, пока канал не будет считан (из другой сопрограммы, потому что текущая заблокирована). Получатель, напротив, при чтении блокируется, когда буфер пуст. Этот набор правил необходим и достаточен²⁷ в своей полноте для реализации любых синхронизаций параллельных ветвей!

buffer.go

```

package main

import "fmt"

func main() {
    c := make(chan int, 2)
    c <- 1
    c <- 2
    fmt.Println(<-c)
    fmt.Println(<-c)
}

```

\$./buffer

```

1
2

```

Только отправитель может закрыть канал, чтобы обозначить, что у него нет больше данных для передачи. Получатель может проверить не был ли канал закрыт присвоением второму параметру считывающего выражения (код завершения):

```

v, ok := <-ch

```

Здесь `ok` примет значение `false` если нет больше данных для приёма и канал закрыт.

Вот такая форма цикла `for` будет циклически принимать данные из канала пока он не будет закрыт (здесь `c` — это переменная канала):

```

for i := range c { ... }

```

Только отправитель имеет право закрыть канал. И никогда получатель. Отправка данных в закрытый канал приведёт к аварийному завершению. Каналы вовсе не подобны файлам — вы не обязаны заботиться о их закрытии. Закрываете канал только когда получатель должен быть уведомлен что данных больше не будет, и ему нужно уходить из бесконечного цикла считывания.

closen.go

```

package main

import "fmt"

func fibonacci(n int, c chan int) {

```

²⁷ Где-то у «титанов» (у Хоара?) было опубликовано и формальное доказательство этого. Но это уже вопросы теории, и вопросы достаточно сложной теории.

```

        x, y := 0, 1
        for i := 0; i < n; i++ {
            c <- x
            x, y = y, x+y
        }
        close(c)
    }

func main() {
    c := make(chan int, 10)
    go fibonacci(cap(c), c)
    for i := range c {
        fmt.Println(i)
    }
}

$ ./close
0
1
1
2
3
5
8
13
21
34

```

Оператор `select` позволяет go-сопрограмме ожидать (блокируясь) событий от нескольких коммуникационных каналов одновременно. Операция `select` блокируется до тех пор, пока его не разблокирует какая-то из коммуникационных веток, в этом случае эта ветка выполняется. Если несколько ветвей оказываются готовы к выполнению, то ветвь к выполнению выбирается случайным образом:

select.go

```

package main

import "fmt"

func fibonacci(c, quit chan int) {
    x, y := 0, 1
    for {
        select {
        case c <- x:
            x, y = y, x+y
        case <-quit:
            fmt.Println("quit")
            return
        }
    }
}

func main() {
    c := make(chan int)
    quit := make(chan int)
    go func() {
        for i := 0; i < 10; i++ {
            fmt.Println(<-c)
        }
        quit <- 0
    }()
    fibonacci(c, quit)
}

```

```

}

$ ./select
0
1
1
2
3
5
8
13
21
34
quit

```

Ветвь `default` в `select` операторе позволяет осуществить действие если ни одна другая ветвь не готова. Такое использование `default` позволяет осуществить попытку записи или чтения канала без блокирования:

```

select {
case i := <-c:
    // use i
default:
    // receiving from c would block
}

```

Это показано на следующем примере (который, заодно, показывает как в Go реализуются асинхронные таймеры):

unblock.go

```

package main

import (
    "fmt"
    "time"
)

func main() {
    tick := time.Tick(100 * time.Millisecond)
    boom := time.After(500 * time.Millisecond)
    for {
        select {
        case <-tick:
            fmt.Println("tick.")
        case <-boom:
            fmt.Println("BOOM!")
            return
        default:
            fmt.Println("    .")
            time.Sleep(50 * time.Millisecond)
        }
    }
}

$ ./unblock
.
.
tick.
.
.
tick.
.
.

```



```
tick.  
.  
.  
tick.  
.  
.  
BOOM!
```

Функциональные замыкания в сопрограмах

Сюрпризом может стать результат использования функциональных замыканий (closure, см. выше) в сопрограмах (каталог `functionals/closure` архива):

closure1.go

```
package main  
  
import (  
    "fmt"  
)  
  
func main() {  
    done := make(chan bool)  
  
    values := []string{"a", "b", "c"}  
    for _, v := range values {  
        go func() {  
            fmt.Println(v)  
            done <- true  
        }()  
    }  
  
    // wait for all goroutines to complete before exiting  
    for _ = range values {  
        <-done  
    }  
}
```

Ошибочно ожидать увидеть здесь a, b, c в качестве вывода результата. Но то, что вы увидите вместо этого, будет: c, c, c:

```
$ ./closure1  
c  
c  
c
```

Это происходит потому, что каждая итерация цикла использует один и тот же экземпляр переменной `v`, так что все экземпляры замыкания работают с одной и той же переменной. Когда замыкание запускается (оператором `go`) оно печатает значение `v` на тот момент когда `fmt.Println()` выполняется, а значение `v`, возможно, было изменено с того момента, как сопрограмма была запущена.

Чтобы привязать текущее значение `v` для каждого замыкания, на тот момент как оно будет запущено, необходимо изменить внутренний цикл для создания новой переменной на каждой итерации.

Один путь состоит в том, чтобы передать переменную в качестве аргумента для замыкания:

closure2.go

```
package main  
  
import (  
    "fmt"
```

```

)

func main() {
    done := make(chan bool)

    values := []string{"a", "b", "c"}
    for _, v := range values {
        go func(u string) {
            fmt.Println(u)
            done <- true
        }(v)
    }

    // wait for all goroutines to complete before exiting
    for _ = range values {
        <-done
    }
}

$ ./closure2
a
b
c

```

Но ещё проще — это просто создание новой переменной, используя декларационный стиль, который может показаться странным, но отлично работает в Go²⁸:

closure3.go

```

package main

import (
    "fmt"
)

func main() {
    done := make(chan bool)

    values := []string{"a", "b", "c"}
    for _, v := range values {
        v := v // create a new 'v'.
        go func() {
            fmt.Println(v)
            done <- true
        }()
    }

    // wait for all goroutines to complete before exiting
    for _ = range values {
        <-done
    }
}

$ ./closure3
a
b
c

```

Примитивы синхронизации

Авторы проекта Go неодобрительно высказываются относительно `pthread_t` модели потоков в POSIX, справедливо указывая на её перегруженность деталями и громоздкость.

²⁸ Это хорошо известная проблема, и так было до версии 1.22 (до 2024 года). В версии 1.22 эта особенность исправлена, и всё будет работать как и хотелось бы без дополнительной переменной.

Go предлагает модель высокого уровня взаимодействия — сопрограммы, идущую от техники Communicating Sequential Processes Хоара. Тем не менее, Go предоставляет и весь набор примитивов синхронизации, и предоставляются все эти «вкусности» пакетом `sync` (<https://pkg.go.dev/sync>). Пакет предоставляет базовые примитивы синхронизации, такие, например, как блокировки взаимного исключения (мютексы). Исключая типы `Once` и `WaitGroup`, большинство из них предназначено для использования в библиотеках низкого уровня. Высокоуровневые синхронизации лучше выражать через каналы и коммуникации.

Переменные, принадлежащие к типам, определяемым в этом пакете, не могут быть скопированы.

Простейшим примитивом синхронизации является мютекс (mutual exclusion lock):

```
type Mutex
    func (m *Mutex) Lock()
    func (m *Mutex) Unlock()
```

Мютексы могут создаваться как составная часть других структур. Нулевым значением для мютекса является разблокированный мютекс.

Метод `Lock()` захватывает мютекс. Если мютекс уже захвачен, то вызвавшая `Lock()` go-сопрограмма блокируется до тех пор, пока мютекс не будет освобождён.

Метод `Unlock()` разблокирует захваченный мютекс. Если `Unlock()` вызывается для не захваченного мютекса, то возникает ошибка времени исполнения (run-time error).

Заблокированный мютекс, не ассоциируется с определенной go-сопрограммой. Допускается, что одна сопрограмма захватит мютекс, а затем другая сопрограмма разблокирует его.

Примечание: Такое поведение, вообще то говоря, в терминологии POSIX соответствует бинарному семафору, а POSIX-мютекс всегда имеет владельца, его захватившего, и только поток-владелец может его освободить.

Тип `Locker` представляет обобщённый интерфейс, который представляет объект, который может захватываться и освобождаться:

```
type Locker interface {
    Lock()
    Unlock()
}
```

Тип `Cond` реализует условную переменную, точку встречи сопрограмм, где они ожидают наступления или уведомляют о наступлении определённого события. Каждая условная переменная имеет ассоциированный с ней объект блокирования `Locker` (зачастую это `*Mutex` или `*RWMutex`), который должен захватываться когда изменяется состояние и когда вызывается метод `Wait()`. Объект `Cond` может создаваться как составная часть других структур. Объекты `Cond` не могут быть скопированы после первого использования.

```
type Cond
    func NewCond(l Locker) *Cond
    func (c *Cond) Broadcast()
    func (c *Cond) Signal()
    func (c *Cond) Wait()
```

`NewCond()` создаёт новую условную переменную с элементом синхронизации `l` (должен быть создан предварительно).

`Wait()` атомарно разблокирует `c.L` и блокирует выполнение вызвавшей сопрограммы до наступления условия. Позже, после возобновления выполнения, `Wait()` блокирует `c.L` перед началом последующего выполнения. В отличие от других систем, `Wait()` не может разблокироваться иначе как вызовами `Signal()` или `Broadcast()`.

Поскольку `c.L` не блокирован в начале освобождения `Wait()`, вызывающий, как правило, не в состоянии предположить, что условие истинно, когда `Wait()` возвратится. Вместо этого ожидающей стороне необходимо будет ждать в цикле:

```
c.L.Lock()
for !condition() {
    c.Wait()
}
... make use of condition ...
c.L.Unlock()
```

Вызов `Signal()` освобождает одну сопрограмму из числа ожидающих на `c`, если таковые имеются. Разрешено, но не требуется, чтобы вызывающий держал заблокированной блокировку `c.L` во время вызова.

Вызов `Broadcast()` освобождает все сразу сопрограммы, ожидающих на `c`.

Объект `RWMutex` является вариантом `Mutex`, но с отдельными уровнями блокирования для читателей (сопрограмм, которые не будут изменять защищаемые данные) и писателей (которые намереваются их изменять):

```
type RWMutex
func (rw *RWMutex) Lock()
func (rw *RWMutex) RLock()
func (rw *RWMutex) RLocker() Locker
func (rw *RWMutex) Runlock()
func (rw *RWMutex) Unlock()
```

Блокировка `RWMutex` позволяет доступ к критической секции (структуре данных) одновременно сколь угодно многим читателям, или только одному писателю. Не допускается одновременный доступ читателей и писателей. Как и мютекс, `RWMutex` может быть составной частью другой структуры, нулевым значением для `RWMutex` является разблокированное состояние.

Метод `Lock()` захватывает `RWMutex` для записи. Если блокировка уже кем-то захвачена (независимо для чтения или для записи) сопрограмма блокируется до её освобождения. После освобождения блокировки и продолжения выполнения, любые другие попытки и `Lock()` и `RLock()` будут приводить к блокировке вызвавших сопрограмм (исключение доступа новых читателей и писателей).

Метод `RLock()` захватывает `RWMutex` для чтения. Это не препятствует присоединению к блокировке новых читателей (выполняющих в свою очередь `RLock()`), но выполнение `Lock()` блокирует вызвавшую сопрограмму, что препятствует доступу писателей.

`RLocker()` возвращает интерфейс `Locker`, что реализует `Lock()` и `Unlock()` методы вызовами `rw.RLock()` и `rw.Unlock()`.

`Runlock()` отменяет один ранее выполненный вызов `RLock()`, он не влияет на состояния других одновременных читателей. Возбуждается ошибка времени выполнения, если `rw` вообще не заблокирована по чтению к моменту вызова `Runlock()`.

`Unlock()` разблокирует блокировку, захваченную на запись. Если блокировка не захвачена на запись, возбуждается ошибка времени выполнения.

Следующий объект `Once` — это объект, который обеспечивает исключительно однократное выполнение действия.

```
type Once
func (o *Once) Do(f func())
```

Если `Once.Do(f)` вызывается многократно, то только при первом вызове будет вызываться `f()`, даже если `f()` и имеет различные значения в каждом вызове. Новый экземпляр `Once` требуется для выполнения каждой отдельной функции.

Метод `Do()` предназначен для инициализации, которая должна выполняться только один

раз.

Так как прототип функции `f()` без параметров, то может оказаться необходимым использовать явный функциональный литерал (анонимную функцию), чтобы захватить аргументы функции в вызове `Do()`:

```
config.once.Do(func() { config.init(filename) })
```

Поскольку никакой вызов `Do()` не возвращается пока не завершиться `f()`, то в случае если `f()` вынуждает повторно `Do()` быть вызванным — это порождает бесконечный дэдлок.

Вот как выглядит пример использования:

once.go

```
package main

import (
    "fmt"
    "sync"
)

func main() {
    var once sync.Once
    onceBody := func() {
        fmt.Println("Only once")
    }
    done := make(chan bool)
    for i := 0; i < 10; i++ {
        go func() {
            once.Do(onceBody)
            done <- true
        }()
    }
    for i := 0; i < 10; i++ {
        <-done
    }
}

$ ./once
Only once
```

Ещё тип данных из этого пакета `Pool`:

```
type Pool struct {
    // New optionally specifies a function to generate
    // a value when Get would otherwise return nil.
    // It may not be changed concurrently with calls to Get.
    New func() interface{}
    // contains filtered or unexported fields
    func (p *Pool) Get() interface{}
    func (p *Pool) Put(x interface{})
}
```

`Pool` — это набор временных объектов хранения, которые могут быть индивидуально сохраняться и вновь извлекаться. Любой элемента, хранящийся в `Pool`, может быть автоматически удалён в любое время без уведомления. Если `Pool` содержит только одну последнюю ссылку на элемент, то когда происходит его удаление, этот экземпляр `Pool` может быть утилизирован (сборкой мусора).

`Pool` безопасен для использования множественными сопрограммами одновременно.

`Pool` предназначен в качестве кэша выделенных, но временно не используемых элементов для использования в дальнейшем, снимая нагрузку со сборщика мусора. Таким образом облегчается создание эффективных потокобезопасных списков. Однако это

подходит не для всех свободных списков.

Соответствующее использование Pool для управления группой временных объектов умалчивая разделяет переиспользование объектов между конкурентными независимыми клиентами пакета. Pool предоставляет возможность амортизировать затраты на размещение из многих клиентов.

Хорошим примером использования Pool является пакет fmt, который поддерживает динамического размера область временного размещения выходных буферов. Область возрастает под нагрузкой (когда многие программы активно печатают) и уменьшается в покое.

С другой стороны, свободный список, поддерживаемый для короткоживущих объектов не есть подходящей областью для использования Pool, так как издержки на поддержание не будут хорошо покрываться в такой ситуации. Более эффективно иметь для таких объектов свои собственные реализованные списки свободных элементов.

Get() выбирает произвольный элемент из Pool, удаляет его из Pool, и возвращает его для использования вызывающей стороне. Get() может выбрать игнорирование Pool и рассматривать его как пустой. Вызывающие не должны предполагать какую-либо взаимосвязь объектов, переданных Put() и объектов возвращаемые Get(). Если Get() в противоположность возвращает nil, а p.New() не nil, то Get() возвращает результат вызова p.New().

Put() добавляет элемент в Pool.

И ещё один механизм синхронизации:

```
type WaitGroup
func (wg *WaitGroup) Add( delta int )
func (wg *WaitGroup) Done()
func (wg *WaitGroup) Wait()
```

WaitGroup ожидает завершения некоторого набора программ. Главная программа вызывает Add() чтобы установить число программ в наборе, которых следует ожидать. Затем каждая программа выполняется, и вызывает метод Done() когда она завершается. В то же самое время метод Wait() может быть вызван для ожидания того, что все программы (инициализированные по числу в Add()) завершатся (в некотором смысле это напоминает POSIX барьеры pthread_barrier_t).

Следующий пример (из документации Go) извлекает несколько URL одновременно (в программах), а используя WaitGroup вызывающий поток блокируется до тех пор, пока все выборки будут выполнены:

```
var wg sync.WaitGroup
var urls = []string{
    "http://www.golang.org/",
    "http://www.google.com/",
    "http://www.somestupidname.com/",
}
for _, url := range urls {
    // Increment the WaitGroup counter.
    wg.Add(1)
    // Launch a goroutine to fetch the URL.
    go func(url string) {
        // Decrement the counter when the goroutine completes.
        defer wg.Done()
        // Fetch the URL.
        http.Get(url)
    }(url)
}
// Wait for all HTTP fetches to complete.
wg.Wait()
```

Законченный пример использования барьерных операций будет показан далее, при обсуждении выполнения на многих процессорах в SMP.

Ещё один пакет, в подкаталоге `sync/atomic`, содержит широкий спектр атомарных операций над целочисленными значениями, которые выполняют неделимые операции тестирования и модификации значений своих операндов. Смысл этих вызовов понятен, в общем виде, из их наименований:

```
func AddInt32(addr *int32, delta int32) (new int32)
func AddInt64(addr *int64, delta int64) (new int64)
func AddUint32(addr *uint32, delta uint32) (new uint32)
func AddUint64(addr *uint64, delta uint64) (new uint64)
func AddUintptr(addr *uintptr, delta uintptr) (new uintptr)
func CompareAndSwapInt32(addr *int32, old, new int32) (swapped bool)
func CompareAndSwapInt64(addr *int64, old, new int64) (swapped bool)
func CompareAndSwapPointer(addr *unsafe.Pointer, old, new unsafe.Pointer) (swapped
bool)
func CompareAndSwapUint32(addr *uint32, old, new uint32) (swapped bool)
func CompareAndSwapUint64(addr *uint64, old, new uint64) (swapped bool)
func CompareAndSwapUintptr(addr *uintptr, old, new uintptr) (swapped bool)
func LoadInt32(addr *int32) (val int32)
func LoadInt64(addr *int64) (val int64)
func LoadPointer(addr *unsafe.Pointer) (val unsafe.Pointer)
func LoadUint32(addr *uint32) (val uint32)
func LoadUint64(addr *uint64) (val uint64)
func LoadUintptr(addr *uintptr) (val uintptr)
func StoreInt32(addr *int32, val int32)
func StoreInt64(addr *int64, val int64)
func StorePointer(addr *unsafe.Pointer, val unsafe.Pointer)
func StoreUint32(addr *uint32, val uint32)
func StoreUint64(addr *uint64, val uint64)
func StoreUintptr(addr *uintptr, val uintptr)
func SwapInt32(addr *int32, new int32) (old int32)
func SwapInt64(addr *int64, new int64) (old int64)
func SwapPointer(addr *unsafe.Pointer, new unsafe.Pointer) (old unsafe.Pointer)
func SwapUint32(addr *uint32, new uint32) (old uint32)
func SwapUint64(addr *uint64, new uint64) (old uint64)
func SwapUintptr(addr *uintptr, new uintptr) (old uintptr)
```

Простейший пример использования атомарных переменных будет показан в следующей главе при рассмотрении выполнения на многих процессорах SMP.

Конкурентность и параллельность

Авторы проекта несколько раз обращаются к обсуждению этой проблематики, поэтому и мы не можем её оставить за рамками рассмотрения. Кроме того, это потребует от нас построить несколько более реалистичных примеров, использующих примитивы синхронизации.

Итак ... Авторы проекта Go неоднократно подчёркивают, что конкурентность исполнения (обеспечиваемая механизмами сопрограмм и каналов) и параллельность — это две совершенно разные вещи, не коррелирующие между собой. Конкурентно выполняющиеся сопрограммы «не знают» на скольких процессорах SMP они выполняются. А число процессоров SMP и способность программы задействовать ресурсы этих процессоров — это уже больше из области аппаратной поддержки вычислений.

Первоначально, в документации GoLang утверждалось, что по умолчанию исполняющая система Go использует один процессор!²⁹ Для того, чтобы сопрограммы могли распределяться на N процессоров нужно:

²⁹ Так было только в ранних версиях, похоже, что до версии 1.1 GoLang. В последующих версиях значение `GOMAXPROCS`, если вы не изменяете его принудительно, напротив устанавливается равным числу процессоров в системе.

- установить переменную окружения shell: GOMAXPROCS=N

- или вызвать в задаче функцию из пакета runtime :

```
func GOMAXPROCS(n int) int
```

Такая функция устанавливает максимальное число системных потоков, которое может задействовать задача Go под свои сопрограммы. Функция возвращает предыдущий установленный лимит. Если вызов производится с параметром $n \leq 0$, то функция не изменяет лимит, а только возвращает его установленное значение. Естественно, что для компилирующей автономной системы (GoLang), установки GOMAXPROCS() действительны только для текущего процесса.

Посмотрим на практике как в программном коде Go можно задействовать SMP. Первый пример (каталог архива `gorproc`) проделает это на уровне атомарных примитивов низкого уровня (`sync.WaitGroup`), о которых говорилось ранее:

smp1.go

```
package main

import (
    "fmt"
    "os"
    "runtime"
    "strconv"
    "sync"
    "sync/atomic"
    "time"
)

var count uint64 = 0
var wg sync.WaitGroup

func main() {
    повторы, потоки := 10000000, 1
    if len(os.Args) > 1 {
        потоки, _ = strconv.Atoi(os.Args[1])
        if потоки > 1 {
            runtime.GOMAXPROCS(потоки)
        }
    }
    if len(os.Args) > 2 {
        повторы, _ = strconv.Atoi(os.Args[2])
    }
    fmt.Printf("число процессоров в системе: %v\n", runtime.NumCPU())
    fmt.Printf("число потоков исполнения: %v\n", runtime.GOMAXPROCS(-1))
    повторы = повторы / потоки
    fmt.Printf("циклов на поток: %v\n", повторы)
    t0 := time.Now()
    for i := 0; i < потоки; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            for i := 0; i < повторы; i++ {
                atomic.AddUint64(&count, 1)
            }
        }()
    }
    wg.Wait()
    fmt.Printf("выполнено циклов: %v\n", count)
    t1 := time.Now()
    fmt.Printf("время выполнения: %v\n", t1.Sub(t0))
}
```


Здесь несколько (или много) сопрограмм (1-й параметр командной строки запуска) совместно выполняют некоторый объём (2-й параметр команды) работы — инкремент счётчика `count`, выполняемый атомарной операцией `atomic.AddUint64()`. Главная программа терпеливо ожидает завершения работы всех сопрограмм на барьерной переменной типа `sync.WaitGroup`.

Но, поскольку Go реализует высокоуровневый механизм взаимодействия и синхронизации сопрограмм через каналы, а авторы проекта настоятельно рекомендуют использовать именно этот механизм высокого уровня, то сделаем другой вариант той же задачи:

smp2.go

```
package main

import (
    "fmt"
    "os"
    "runtime"
    "strconv"
    "sync"
    "time"
)

var ch1 chan uint64 = make(chan uint64)
var count uint64 = 0

func counter(ch chan uint64) {
    for {
        count = count + <-ch
    }
}

var wg sync.WaitGroup

func main() {
    повторы, потоки := 10000000, 1
    if len(os.Args) > 1 {
        потоки, _ = strconv.Atoi(os.Args[1])
        if потоки > 1 {
            runtime.GOMAXPROCS(потоки)
        }
    }
    if len(os.Args) > 2 {
        повторы, _ = strconv.Atoi(os.Args[2])
    }
    fmt.Printf("число процессоров в системе: %v\n", runtime.NumCPU())
    fmt.Printf("число потоков исполнения: %v\n", runtime.GOMAXPROCS(-1))
    повторы = повторы / потоки
    fmt.Printf("циклов на поток: %v\n", повторы)
    go counter(ch1)
    t0 := time.Now()
    for i := 0; i < потоки; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            for i := 0; i < повторы; i++ {
                ch1 <- 1
            }
        }()
    }
    wg.Wait()
    fmt.Printf("выполнено циклов: %v\n", count)
```

```

        t1 := time.Now()
        fmt.Printf("время выполнения: %v\n", t1.Sub(t0))
    }

```

Здесь синхронизация циклящихся сопрограмм происходит при записи в канал ch1, а инкремент счётчика выполняет отдельная сопрограмма-получатель с функцией counter().

И несколько результатов...

\$./smp2

```

число процессоров в системе: 4
число потоков исполнения: 1
циклов на поток: 10000000
выполнено циклов: 10000000
время выполнения: 1.534208534s

```

\$./smp2 2

```

число процессоров в системе: 4
число потоков исполнения: 2
циклов на поток: 5000000
выполнено циклов: 10000000
время выполнения: 3.560645615s

```

\$./smp2 3

```

число процессоров в системе: 4
число потоков исполнения: 3
циклов на поток: 3333333
выполнено циклов: 9999999
время выполнения: 2.615792178s

```

\$./smp2 4

```

число процессоров в системе: 4
число потоков исполнения: 4
циклов на поток: 2500000
выполнено циклов: 10000000
время выполнения: 1.792243896s

```

Вас, возможно, смущает то, что 4-х процессорах время выполнения хуже чем на 1-м? А это не должно удивлять: «полезная» работа сопрограмм (инкремент целочисленной переменной) на порядки менее трудоёмкая, чем работа по синхронизации, в данном случае отправке данных в канал и их получение оттуда. Львиную долю своего времени каждая сопрограмма находится в заблокированном состоянии, в ожидании возможности доступа к переменной. К подобным сюрпризам нужно быть готовым в среде SMP и не переносить тупо объём работы на несколько процессоров (да ещё при этом можно радикально ухудшить условия кэширования данных).

А теперь вот так:

\$./smp2 100

```

число процессоров в системе: 4
число потоков исполнения: 100
циклов на поток: 100000
выполнено циклов: 10000000
время выполнения: 2.860770862s

```

Система из 4-х процессоров (все разрешены) выполняет 100 параллельных сопрограмм ничуть не хуже, чем при согласованных числах потоков и процессоров.

Примечание: К измерению временных интервалов в многозадачных операционных системах нужно относиться с очень большой осторожностью. Да ещё при стандартном уровне приоритета выполнения задачи в системе. Да ещё при том, что и изменение приоритета (в сторону увеличения) командой nice в Linux носит очень сомнительный характер (ввиду его уж очень специфического планирования для обычных задач). По-хорошему, нужно было бы выполнять задачу с дисциплиной планирования реального времени (FIFO или

RR), например, командной `chrt`. В итоге, в измерении временных интервалов выполнения в учёт может приниматься только порядок значений, но не их величина. И величины, отличающиеся, скажем, вдвое, должны расцениваться как «равно».

И ещё одно сравнение:

\$./smp1

число процессоров в системе: 4
число потоков исполнения: 1
циклов на поток: 10000000
выполнено циклов: 10000000
время выполнения: 115.828692ms

\$./smp2

число процессоров в системе: 4
число потоков исполнения: 1
циклов на поток: 10000000
выполнено циклов: 10000000
время выполнения: 1.514249613s

Это плата (на порядок) за использование высокоуровневых механизмов синхронизации. В такой задаче это, конечно, граничный случай, в более реальных ситуациях разрыв будет ниже. Но, с другой стороны, в сложных реальных проектах высокоуровневые механизмы приносят ясность и лаконизм. А механизмы низкого уровня приносят трудно локализуемые ошибки, которые нивелируют любые выигрыши в скоростных показателях.

Всё это, помимо практики написания параллельного кода в Go, подводит к заключению, что ко всему, что относится к параллелизму и использованию нескольких процессоров, нужно относиться с очень большой настороженностью — здесь нас ожидают много сюрпризов. И это относится не только к использованию Go, но следует из фундаментальных принципов организации вычислений.

Источники информации

[1] Daniel Morsing, The Go scheduler — <https://morsmachine.dk/go-scheduler>

перевод: <https://linux-ru.ru/download/file.php?id=4925>

[2] У. Ричард Стивенс, Стивен Раго, UNIX. Профессиональное программирование, 3-е издание, 2013, СПб. Символ-Плюс

[3] Уильям Стивенс, Unix. Взаимодействие процессов, 2003, СПб. Питер —

<https://rutracker.org/forum/viewtopic.php?t=31889>

[4] Олег Цилюрик, Параллелизм, конкурентность, многопроцессорность в Linux —

<http://flibusta.is/b/523510>

<http://mylinuxprog.blogspot.com/2014/09/linux.html>

[5] Олег Цилюрик, Егор Горошко, QNX/UNIX: анатомия параллелизма, декабрь 2005, СПб. Символ-Плюс

<http://www.flibusta.net/a/36261>

[6] Многопоточность в C++. Управление потоками — <https://radioprogram.ru/post/1403/>

[7] Подробно о корутинах в C++ — <https://habr.com/ru/company/piter/blog/491996/>

[8] Корутины в C++20. Часть 1 — <https://itnan.ru/post.php?c=1&p=520756>

[9] Proposal: Non-cooperative goroutine preemption, Author(s): Austin Clements, Last updated: 2019-01-18 —

<https://github.com/golang/proposal/blob/master/design/24543-non-cooperative-preemption.md>

Масштабирование

*«Чудище обло, озорно, огромно, стозевно
и лаяй»*

*А.Н.Радищев «Путешествие из
Петербурга в Москву»*

Здесь мы подошли к месту, где разрозненные фрагменты, мозаично разбросанные по всему предыдущему тексту, должны сложиться в единую картину. То, собственно, для чего и писалась эта книга, в некотором смысле...

Под масштабированием будем понимать способность исполнимого программного кода к переносу между различными архитектурами радикально отличающимися производительностью, с сохранением возможности эффективно использовать все имеющиеся ресурсы конкретной архитектуры, в первую очередь — оптимально распараллеливать вычислительную нагрузку между имеющимися процессорами.

Мы сейчас добрались до самого интересного места нашего повествования: каким образом код Go имеет возможность масштабирования на весь доступный спектр оборудования? Код Go, за счёт оригинального механизма диспетчирования горутин, даже без перекомпиляции позволяет легко обеспечить масштабирование в широчайшем диапазоне: от однокристальных ARM микрокомпьютеров (SBC — single-board computer, SoC — System on Chip) и до многопроцессорных серверов промышленной линии, с многими десятками процессоров. Это рассмотрение сводит воедино все аспекты многопроцессорности и параллелизма, рассмотренные по отдельности ранее...

Планирование активности сопрограмм

Для уяснения того как GoLang позволяет производить масштабирование, нам нужно вернуться и вспомнить как GoLang осуществляет планирование (диспетчирование) активности сопрограмм.³⁰ Принципы в общих чертах:

- Программе на Go позволено выполняться в числе потоков ядра Linux (pthread_t) равному внутренней переменной окружения GOMAXPROCS пакета runtime.
- Это значение может быть считано вызовом `runtime.GOMAXPROCS(-1)`, и может быть изменено вызовом с положительным значением параметра `runtime.GOMAXPROCS(N)` (но вряд ли это стоит делать без хорошего понимания зачем вы это делаете).
- При старте приложения GOMAXPROCS устанавливается в дефолтное значение равное реальному числу процессоров (ядер) в "железе" — `runtime.NumCPU()`.
- Для эффективного использования процессоров нет смысла выполнять приложение Go на числе потоков ядра превышающем число процессоров — чтобы избежать вытесняющего (preemptive multitasking) переключения контекста механизмами ядра: каждому потоку — свой процессор.
- Такое число потоков поддерживается даже при очень большом числе горутин приложения, намного превышающих число процессоров: сотни и даже тысячи сопрограмм...
- Сами горутины «прикрепляются» к очередям сопрограмм каждого из GOMAXPROCS потоков ядра (при числе параллельных ветвей больше числа процессоров) и активируются между собой в пределах одной очереди последовательно, на принципах кооперативной многозадачности, под управлением исполняющей системы Go, а не операционной системы.

³⁰ Механизм планирования активности горутин, кажется, укрупнённо заложен авторами Go ещё в сам синтаксис языка предвосхищая любую его реализацию. Но по реализации сам такой механизм сложен, поэтому в деталях он менялся от версии к версии GoLang. В том виде, который описывается в тексте, этот механизм реализован, похоже по публикациям, начиная с версии 1.5. Но даже в последующих версиях он частично меняется, что отражается и в тонкостях поведения описываемых тестов.

Примечание: Многозадачность горутин, «прикрепленных» к одному ядру, тоже не является кооперативной: планировщик Go производит их вытеснение и контролирует безопасные точки, зачастую совпадающие с входом в функции. Планировщик Go умудряется разрывать даже плотные циклы. Таким образом, происходит вытеснение горутин, даже если они не выполняют явно `yield()`, что требуется для чистой кооперативности. Механизм вытеснения планировщика подробно описан в [9] перечня информационных источников предыдущего раздела. Но эти механизмы сложны и очень существенно изменяются от версии к версии. Мне они до конца не понятны, но их наличие и работу нужно иметь во внимании даже если механизм их до конца не понятен.

Обратим внимание, что в этой модели потоки ядра, выполняющие горутин, представляются не как механизмы параллелизма ядра (которые можно создавать и уничтожать произвольно), а как зеркальные отображения аппаратных процессоров. На каждом процессоре выполняется в точности один выполняющий поток, так чтобы он не мог прерываться, вытесняться другими такими же выполняющими потоками. А значит каждая горутина выполняется на одном процессоре, без перезагрузки контекста и без восстановления, что ещё важнее, синхронного состояния кэш-памяти.

Это — схема. Но жизнь разнообразнее любых схем, и эта схема может дополняться существенными деталями:

- Кроме очереди горутин у каждого потока (процессора), может возникать глобальная общая очередь горутин, через которую они перераспределяются между процессорами.
- Вообще то, доминирующее время горутина прикреплена к очереди «своего» фиксированного потока. Но при возникновении в ходе работы значительной разбалансировки числа горутин между разными процессорами, уже выполняющиеся горутин могут изредка переноситься из одной очереди в другую, а значит продолжать выполнение на другом процессоре.

Детали реализации и поведение этой укрупнённой схемы могут довольно существенно отличаться, в первую очередь, в зависимости от той дисциплины вытесняющего диспетчирования потоков (`pthread_t`) по таймеру, которая принята по умолчанию в той или другой операционной системе.

Испытательный стенд

Для испытания эффективности приложений на Go в многопроцессорной архитектуре (что постулировалось одной из главных целей разработчиками языка) нам необходим стенд из разных конфигураций «железа». Как можно более различающихся масштабами. Это отличает поставленную задачу от других в IT — здесь нельзя смоделировать разные окружения конфигурациями и программными настройками, здесь нужно тестировать эквивалентные программные коды на физически различающихся экземплярах оборудования.

Мы будем тестировать 3 принципиально отличающихся **группы** оборудования... Дальше, при тестировании и обсуждении результатов, я смогу ссылаться на экземпляр оборудования просто по наименованию, не отвлекаясь каждый раз на аппаратные особенности.

Микрокомпьютеры (Single-Board Computers)

Первым представителем группы микрокомпьютеров будет широко известный и популярный в народе Raspberry Pi:

```
$ inxi -MCxxx
```

```
Machine:   Type: ARM Device System: Raspberry Pi 2 Model B Rev 1.1 details: BCM2835
           rev: a21041 serial: 00000000f57e2ca8
CPU:       Info: Quad Core model: ARMv7 v7l variant: cortex-a7 bits: 32
           type: MCP arch: v7l rev: 5
           features: Use -f option to see features bogomips: 256
           Speed: 1000 MHz min/max: 600/1000 MHz Core speeds (MHz):
           1: 1000 2: 1000 3: 1000 4: 1000
```

```
$ free
```

	total	used	free	shared	buff/cache	available
Mem:	945300	194196	160504	6644	590600	680608
Swap:	102396	0	102396			



Рис. 2.2. Raspberry Pi

С установленной (на SD карту 8Gb как носитель) операционной системой:

```
$ lsb_release -a
```

```
No LSB modules are available.
```

```
Distributor ID: Raspbian
```

```
Description:    Raspbian GNU/Linux 11 (bullseye)
```

```
Release:        11
```

```
Codename:       bullseye
```

```
$ cat /etc/debian_version
```

```
11.3
```

```
$ uname -a
```

```
Linux raspberrypi 5.15.30-v7+ #1536 SMP Mon Mar 28 13:43:34 BST 2022 armv7l GNU/Linux
```

Даже такой малыш позволяет установить на него (стандартным образом из стандартного репозитория системы) **нативную** среду GoLang одной из самых свежих версий:

```
$ go version
```

```
go version go1.15.15 linux/arm
```

Следующий представитель этой группы ещё миниатюрнее (как в натуральном физическом смысле, так и в смысле электронных ресурсов) — это гораздо менее известное, но многочисленное семейство китайцев (оригинальная разработка) Orange Pi. Я специально использовал самую минимальную модель этого семейства Orange Pi One (512Mb памяти), и тем она ещё изумительнее³¹:

```
$ inxi -MCxxx
```

```
Machine:    Type: ARM Device System: Xunlong Orange Pi One
            details: Allwinner sun8i Family rev: N/A
            serial: 02c000815fd5e717
```

```
CPU:        Info: Quad Core model: ARMv7 v7l variant: cortex-a7 bits: 32
            type: MCP arch: v7l rev: 5
            features: Use -f option to see features bogomips: 0
            Speed: 1008 MHz min/max: 480/1008 MHz Core speeds (MHz):
```

³¹ Такую игрушку вы вполне можете купить (заказать из Китая) по цене около 9 USD.

1: 1008 2: 1008 3: 1008 4: 1008

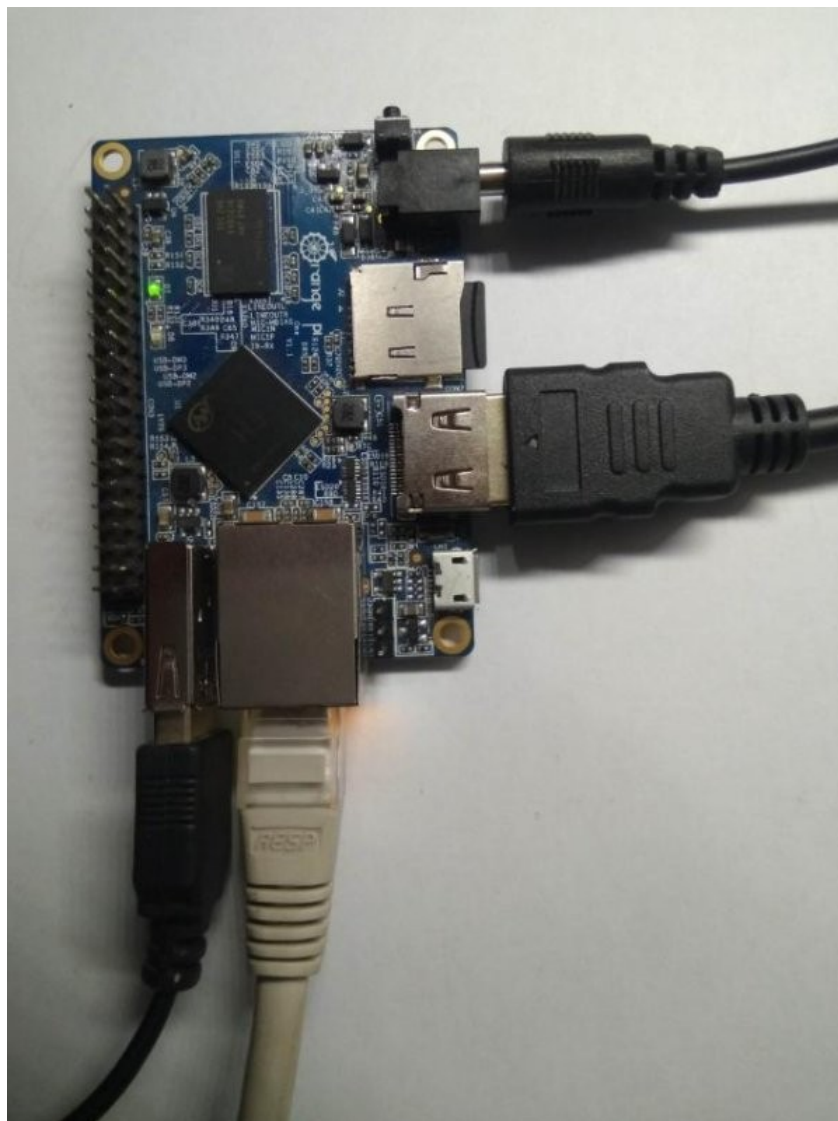


Рис. 2.3. Orange Pi One

```
$ free
```

	total	used	free	shared	buff/cache	available
Mem:	503532	104388	175920	740	223224	386924
Swap:	251764	0	251764			

С установленной (на SD карту всего 4Gb как носитель) операционной системой сборки Armbian, основанной на Debian (эти сборки обновляются с большой регулярностью):

```
$ lsb_release -a
```

```
No LSB modules are available.
Distributor ID: Debian
Description:    Debian GNU/Linux 11 (bullseye)
Release:       11
Codename:      bullseye
```

```
$ cat /etc/debian_version
```

```
11.2
```

```
$ uname -a
```

```
Linux orangepione 5.15.25-sunxi #22.02.1 SMP Sun Feb 27 09:23:25 UTC 2022 armv7l
GNU/Linux
```

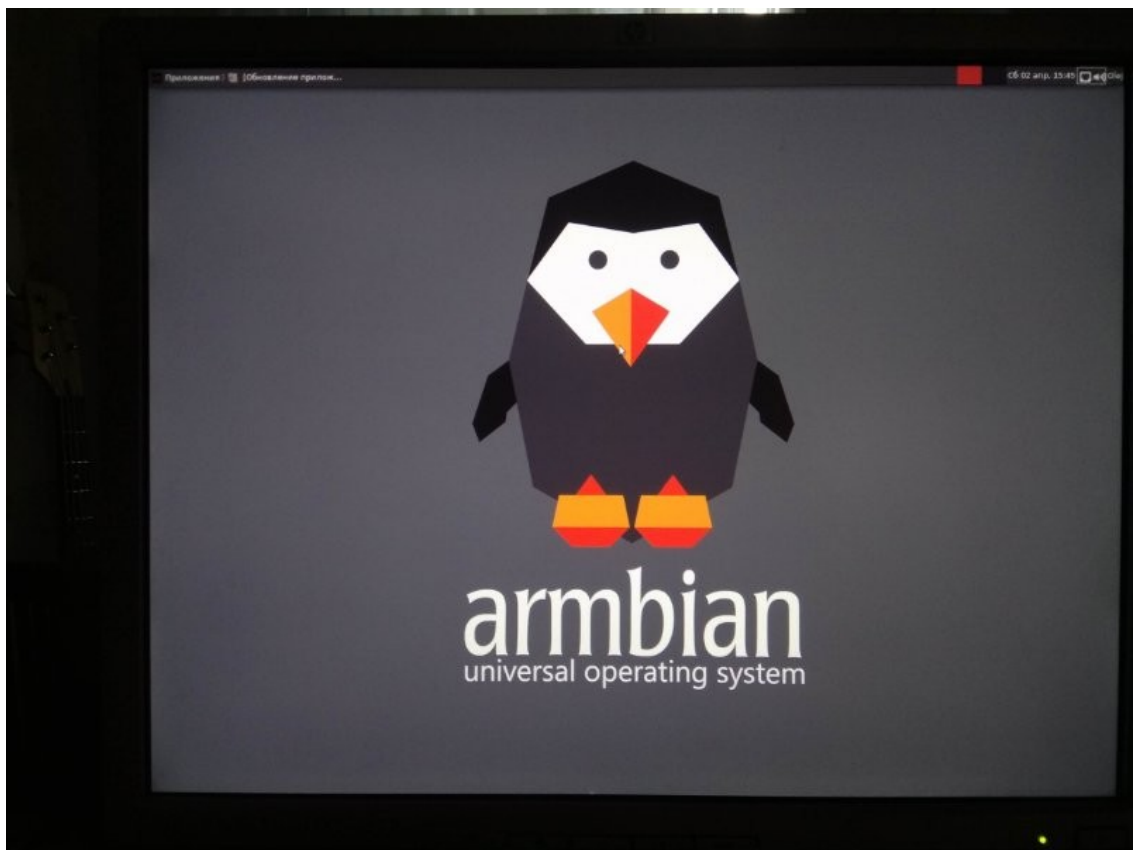


Рис. 2.4. Сборка дистрибутива Armbian

И даже на такую «игрушечную» платформу можно установить полномерный GoLang, одной из самых последних версий:

```
$ go version
go version go1.17.8 linux/arm
```

И, более того, даже на такой минималистичной архитектуре (как первой, так и второй) можно вполне, как будет показано вскоре, нативно компилировать достаточно изощрённые приложения. Это при том, что экосистема GoLang вполне позволяет легко, как показано было ранее, выполнять кросс-компиляцию проектов под самые разные платформы на типовых, привычных десктопах Intel/AMD.

Рабочие десктопы

Это группа наиболее понятна, доступна для использования и не нуждается в особых пояснениях. Я в этом качестве использовал 4-5 подручных экземпляров компьютеров, главное требование к которым состояло в различающемся числе ядер (процессоров)³². Это могут быть любые процессоры Intel или AMD, конкретный тип процессора или его производительность для наших целей не имеет значения.

Например так — вот такой минимальной информации о процессорах уже достаточно для поставленных целей:

- минимальная конфигурация: 2 ядра без гипертренинга:

```
$ inxi -Cxxx
CPU:      Topology: Dual Core model: Intel Celeron G3930 bits: 64
          type: MCP arch: Kaby Lake rev: 9 L2 cache: 2048 KiB
          flags: lm nx pae sse sse2 sse3 sse4_1 sse4_2 ssse3 vmx bogomips: 11599
          Speed: 2900 MHz min/max: 800/2900 MHz Core speeds (MHz): 1: 2900 2: 2900
```

³² Мне так и не удалось найти на сегодня работающий компьютер с одноядерным процессором для проверки некоторых тестов. Его интересно было бы испытать для (квази) параллельного использования. С другой стороны, это — яркое подтверждение провидческого предсказания авторов Go в их ориентации в будущем именно на многопроцессорную (десятки и сотни процессоров) обработку.


```
$ lscpu | grep -i яд
```

```
Порядок байт:           Little Endian
Потоков на ядро:         1
Ядер на сокет:           2
```

- промежуточная конфигурация: 2 ядра с гипертредингом:

```
$ inxi -Cxxx
```

```
CPU:      Topology: Dual Core model: Intel Core i5 660 bits: 64
          type: MT MCP arch: Nehalem rev: 5 L2 cache: 4096 KiB
          flags: lm nx pae sse sse2 sse3 sse4_1 sse4_2 ssse3 vmx bogomips: 26600
          Speed: 1891 MHz min/max: N/A Core speeds (MHz): 1: 1891 2: 1535 3: 1684 4:
1597
```

```
$ lscpu | grep -i яд
```

```
Порядок байт:           Little Endian
Потоков на ядро:         2
Ядер на сокет:           2
```

- максимальная конфигурация: 4 ядра с гипертредингом:

```
$ inxi -Cxxx
```

```
CPU:      Topology: Quad Core model: Intel Xeon E3-1240 v3 bits: 64
          type: MT MCP arch: Haswell rev: 3
          L2 cache: 8192 KiB
          flags: avx avx2 lm nx pae sse sse2 sse3 sse4_1 sse4_2 ssse3 vmx bogomips:
54275
          Speed: 3592 MHz min/max: 800/3800 MHz Core speeds (MHz):
          1: 3592 2: 3592 3: 3592 4: 3592 5: 3592 6: 3592 7: 3592 8: 3592
```

```
$ lscpu | grep -i яд
```

```
Порядок байт:           Little Endian
Потоков на ядро:         2
Ядер на сокет:           4
```

Понятно, что для экземпляров этой группы нас должно интересовать только число процессоров, и ссылаться на них можно по этому числу.

Серверы промышленного класса

В этой группе, в качестве старшего образца в рассматриваемом диапазоне платформ я использовал DELL PowerEdge R420 с 2-мя установленными физическими процессорами Intel Xeon E5-2470 v2 (старшие из допускаемых для серверов этого поколения) по 10 ядер каждый (20 с учётом гипертрединга), 96Gb оперативной памяти:

```
$ sudo inxi -MCxxx
```

```
Machine:   Type: Server System: Dell product: PowerEdge R420 v: N/A
          serial: 9DDFKY1 Chassis: type: 23 serial: 9DDFKY1
          Mobo: Dell model: 0CN7CM v: A06 serial: ..CN1374035400R0.
          BIOS: Dell v: 2.9.0 date: 01/09/2020
CPU:       Topology: 2x 10-Core model: Intel Xeon E5-2470 v2 bits: 64
          type: MT MCP SMP arch: Ivy Bridge rev: 4
          L1 cache: 640 KiB L2 cache: 50.0 MiB L3 cache: 50.0 MiB
          flags: avx lm nx pae sse sse2 sse3 sse4_1 sse4_2 ssse3 vmx bogomips: 192104
          Speed: 2964 MHz min/max: 1200/3200 MHz Core speeds (MHz):
          1: 2950 2: 2963 3: 2800 4: 2801 5: 2810 6: 2802 7: 2807 8: 2801 9: 2883 10:
2806
          11: 2822 12: 2872 13: 2952 14: 2793 15: 2804 16: 2799 17: 2800 18: 2799 19:
2799
          20: 2825 21: 2817 22: 2801 23: 2800 24: 2963 25: 2801 26: 2814 27: 2853 28:
2800
          29: 2802 30: 2800 31: 2825 32: 2803 33: 2813 34: 2854 35: 2952 36: 2962 37:
2827
          38: 2803 39: 2857 40: 2801
```

\$ free

	всего	занято	свободно	общая	буф./врем.	доступно
Память:	98936044	16594916	71548804	876696	10792324	80603408
Подкачка:	2097148	0	2097148			



Рис. 2.5. DELL PowerEdge R420

Конкретная модель сервера, для изучения результатов, не суть важно, можно использовать любую доступную модель, но желательно с большим числом ядер/процессоров.

Установлена операционная система:

```
$ lsb_release -a
```

```
No LSB modules are available.  
Distributor ID: Linuxmint  
Description:   Linux Mint 20.3  
Release:      20.3  
Codename:     una
```

```
$ uname -a
```

```
Linux R420 5.4.0-107-generic #121-Ubuntu SMP Thu Mar 24 16:04:27 UTC 2022 x86_64 x86_64  
x86_64 GNU/Linux
```

Установлена из стандартного репозитория версия GoLang:

```
$ go version
```

```
go version go1.13.8 linux/amd64
```

Масштабирование в реальном мире

*Тестирование программы может
быть очень эффективным
способом показать наличие
ошибок, но оно безнадежно
неподходяще для
доказательства их отсутствия.
Дейкстра Эдсгер Вибе
«The Humble Programmer»*

Для того, чтобы оценить степень эффективности («умения») использования нескольких (P) процессоров в системе, необходима адекватная тестовая задача. Нужно отследить как N параллельных ветвей (N может изменяться в широких пределах), выполняющих одну и ту же работу, могут распараллелиться и загрузить на 100% все имеющиеся P процессоров. Это можно контролировать по времени выполнения задачи для разных N : T(N).

Это не так просто, потому что в теле рабочей функции каждой параллельной ветви не должно быть действий, переводящих ветвь в блокированное состояние, сколь бы кратковременным это состояние не было (имеются в виду избежать операций, которые в разных языках программирования называются `sleep()`, `delay()`, `pause()`, `wait()` и т. д., или вызовов, которые косвенно, скрыто создают блокированное состояние ... в том числе и примитивов синхронизации). В мультизадачной операционной системе (Linux) такое блокированное состояние инициирует переключение контекста на уровне ядра операционной системы, перекоммутацию процессоров, и разрушение адекватности нашего теста. Понятно, что в качестве рабочей функции ветви (а у нас это будут сопрограммы) можно попытаться использовать некоторые фиктивные (результат нас не интересует) циклические вычисления, многократно повторяемые и не провоцирующие блокированных состояний.

Далее... Вспомнив как координируется взаимное исполнение сопрограмм в Go (о чём подробно рассказано ранее по тексту), нам нужно наблюдать как N сопрограмм приложения распределятся между потоками ядра (`pthread_t`) операционной системы, к которым они «прикреплены», и число которых (потоков) M должно, в соответствии с теорией, поддерживаться исполнимой системой Go равным числу процессоров в «железе» («каждому потоку — по своему процессору»). Теперь, определившись с принципами, можно приступить к конструированию тестов (каталог `scheduler` архива).

А заодно, на таких задачах мы можем уже поупражняться в достаточно изощрённом программировании Go, использующем в совокупности всё, о чём по частностям говорилось ранее: а). горютины, б). функциональные литералы и анонимные функции, в). использование межъязыкового интерфейса `Cgo` к API Linux/POSIX, г). объектное программирование, д). работа со службой времени Go, е). форматирование разнообразных типов для вывода во внутренние строки или на терминал, ж). `defer` действия завершения, з). взаимодействие и синхронизация сопрограмм через каналы ... и другое...

1-я попытка ...

Здесь рабочая функция сопрограммы тупо выполняет пустой цикл, в конце каждого цикла проверяя не истекла ли ещё 1 секунда, отпущенная на выполнение каждой горютины. Попутно мы выясняем TID идентификатор потока ядра, в котором выполнялась горютина. (TID типа `pthread_t` потока присваивается потоку при его создании вызовом API ядра Linux `pthread_create()` и остаётся не изменяемым всё время жизни до завершения. TID уникальны в рамках процесса, приложения, и не могут дублироваться при следующем создании новых потоков.)

m1par.go

```
package main
```

```

#include <pthread.h>
//unsigned long tid(void) { return (unsigned long)pthread_self(); }
import "C"
import (
    "fmt"; "os"; "runtime"
    "strconv"; "time"
)

```

```
type msg struct { // структура сообщения
```

```

        n int          // номер ветви
        t int64        // системный pthread_t
    }
    func (p msg) String() string {
        return fmt.Sprintf("[%02d,%X]", p.n, p.t)
    }

    func main() {
        fmt.Printf("число процессоров в системе: %v\n", runtime.NumCPU())
        debug := true
        ветви := 3
        if len(os.Args) > 1 {
            ветви, _ = strconv.Atoi(os.Args[1])
            if ветви < 0 {
                debug = false
                ветви = -ветви
            }
        }
        fmt.Printf("число ветвей выполнения: %v\n", ветви)
        ch := make(chan msg)
        t0 := time.Now()
        for i := 1; i <= ветви; i++ {
            go func(n int) {
                defer func() { ch <- msg{int(n), int64(C.tid())} }()
                t0, t1 := time.Now(), time.Now()
                for t1.Sub(t0).Seconds() < 1 {
                    t1 = time.Now()
                }
            }(i)
        }
        for i := 1; i <= ветви; i++ {
            m := <-ch
            if debug { fmt.Printf("%s\n", m.String()) }
        }
        fmt.Printf("итоговое время выполнения: %v\n", time.Now().Sub(t0))
    }
}

```

Число параллельных гопрограмм, которые будут выполняться параллельно, задаётся числовым параметром запуска приложения в командной строке.

Для начала сервер R420 с 40 процессорами:

```

$ ./mlpar
число процессоров в системе: 40
число ветвей выполнения: 3
[01,7F284DE61740]
[03,7F284AC0D700]
[02,7F284A40C700]
итоговое время выполнения: 1.000147569s

```

```

$ ./mlpar 10
число процессоров в системе: 40
число ветвей выполнения: 10
[10,7F2011D1F700]
[04,7F200BFFF700]
[01,7F20097FA700]
[02,7F2009FFB700]
[03,7F201151E700]
[07,7F200AFFD700]
[05,7F200A7FC700]
[08,7F2014772740]
[09,7F2008FF9700]
[06,7F200B7FE700]

```

итоговое время выполнения: 1.000812088s

\$./m1par -20

число процессоров в системе: 40

число ветвей выполнения: 20

итоговое время выполнения: 1.011463233s

\$./m1par -50

число процессоров в системе: 40

число ветвей выполнения: 50

итоговое время выполнения: 1.095768786s

\$./m1par -100

число процессоров в системе: 40

число ветвей выполнения: 100

итоговое время выполнения: 1.491287599s

\$./m1par -1000

число процессоров в системе: 40

число ветвей выполнения: 1000

итоговое время выполнения: 3.917394145s

Пока, не углубляясь в детали, отметим только (запуск 10 параллельных горутин): когда процессоров-потоков в избытке (больше чем параллельных ветвей) каждая ветвь выполняется в своём **индивидуальном** потоке.

Перейдём сразу к самому слабому десктопному компьютеру нашего стенда — хоть и свежая модель, но Celeron с 2-мя ядрами:

\$ inxi -Cxxx

CPU: Topology: Dual Core model: Intel Celeron G3930 bits: 64

type: MCP arch: Kaby Lake rev: 9 L2 cache: 2048 KiB

flags: lm nx pae sse sse2 sse3 sse4_1 sse4_2 ssse3 vmx bogomips: 11599

Speed: 2900 MHz min/max: 800/2900 MHz Core speeds (MHz): 1: 2900 2: 2900

\$./m1par 2

число процессоров в системе: 2

число ветвей выполнения: 2

[02,7FCF8CA50740]

[01,7FCF88FEB700]

итоговое время выполнения: 1.008687755s

\$./m1par 4

число процессоров в системе: 2

число ветвей выполнения: 4

[04,7F859BFFF700]

[02,7F85A3CC5740]

[01,7F85A3CC5740]

[03,7F859BFFF700]

итоговое время выполнения: 1.056497493s

\$./m1par -10

число процессоров в системе: 2

число ветвей выполнения: 10

итоговое время выполнения: 1.095401087s

\$./m1par -100

число процессоров в системе: 2

число ветвей выполнения: 100

итоговое время выполнения: 2.041495128s

Ну и, для полноты картины - миниатюрные одноплатники ARM.

Raspberry Pi 2 :

```
$ inxi -mCxxx
```

```
Memory:    RAM: total: 999.1 MiB used: 337.7 MiB (33.8%) gpu: 76 MiB
            RAM Report: unknown-error: Unknown dmidecode error. Unable to generate data.
CPU:       Info: Quad Core model: ARMv7 v7l variant: cortex-a7 bits: 32
            type: MCP arch: v7l rev: 5
            features: Use -f option to see features bogomips: 256
            Speed: 1000 MHz min/max: 600/1000 MHz Core speeds (MHz):
            1: 1000 2: 1000 3: 1000 4: 1000
```

Компиляция под ARM нативная:

```
$ time go build mlpars.go
```

```
real      0m8,938s
user      0m6,353s
sys       0m1,956s
```

Компиляция не мгновенная, но совершенно приемлемо быстрая... И исполнение:

```
$ ./mlpar
```

```
число процессоров в системе: 4
число ветвей выполнения: 3
[03,76F881C0]
[02,657FF440]
[01,64FFE440]
итоговое время выполнения: 1.001151413s
```

```
$ ./mlpar 5
```

```
число процессоров в системе: 4
число ветвей выполнения: 5
[05,660FF440]
[02,76F441C0]
[01,656FF440]
[03,76F441C0]
[04,64EFE440]
итоговое время выполнения: 1.013559188s
```

```
$ ./mlpar 8
```

```
число процессоров в системе: 4
число ветвей выполнения: 8
[06,76FCC1C0]
[08,661FF440]
[05,657FF440]
[04,64FFE440]
[07,661FF440]
[01,64FFE440]
[02,657FF440]
[03,76FCC1C0]
итоговое время выполнения: 1.053898134s
```

```
$ ./mlpar -10
```

```
число процессоров в системе: 4
число ветвей выполнения: 10
итоговое время выполнения: 1.056158051s
```

```
$ ./mlpar -100
```

```
число процессоров в системе: 4
число ветвей выполнения: 100
итоговое время выполнения: 2.887051111s
```

... и, наконец, совсем «малыш» Orange Pi One :

```
$ inxi -mCxxx
Memory:    RAM: total: 491.7 MiB used: 110.8 MiB (22.5%)
           RAM Report: missing: Required program dmidecode not available
CPU:       Info: Quad Core model: ARMv7 v7l variant: cortex-a7 bits: 32
           type: MCP arch: v7l rev: 5
           features: Use -f option to see features bogomips: 0
           Speed: 1008 MHz min/max: 480/1008 MHz Core speeds (MHz):
           1: 1008 2: 1008 3: 1008 4: 1008
```

Компиляция нативная (и это при 512Mb памяти на всю операционную систему!):

```
$ go version
go version go1.17.8 linux/arm

$ time go build mlpar.go
real    0m3.706s
user    0m3.990s
sys     0m0.975s

$ ls -l mlpar
-rwxr-xr-x 1 olej olej 1674272 апр  6 23:14 mlpar
```

И выполнение:

```
$ ./mlpar
число процессоров в системе: 4
число ветвей выполнения: 3
[01,A61FF450]
[03,B6F1AD40]
[02,A4FFE450]
итоговое время выполнения: 1.000566416s
```

```
./mlpar 5
число процессоров в системе: 4
число ветвей выполнения: 5
[05,B6F9FD40]
[01,A43FF450]
[04,A43FF450]
[03,A4FFE450]
[02,A61FF450]
итоговое время выполнения: 1.026225244s
```

```
$ ./mlpar -10
число процессоров в системе: 4
число ветвей выполнения: 10
итоговое время выполнения: 1.097000481s
```

```
$ ./mlpar -100
число процессоров в системе: 4
число ветвей выполнения: 100
итоговое время выполнения: 2.948944264s
```

Как и раньше, мы воочию наблюдаем экспериментальную картину планирования, совпадающую с описанной выше моделью:

- Исполняющая система Go при старте приложения создаёт системные рабочие потоки ядра Linux в количестве GOMAXPROCS, совпадающим с числом процессоров;
- До тех пор, пока число горутин приложения не превышает GOMAXPROCS, каждая новая запускаемая горутин распределяется на очередной индивидуальный свободный поток ядра;
- Как только число параллельных горутин превысит GOMAXPROCS, новые горутин

прикрепляются к очередям горутин при потоках;

Это качественная модель. А теперь обратимся к абсолютным цифрам, которые мы пока игнорировали. До 10 параллельных горутин на 4 процессорах (и даже на 2-х) увеличивают суммарное время (относительно 1-й последовательной ветви) не более 5-7%. И даже 100 горутин увеличивает время выполнения 100-кратной работы не более чем в 2.8-2.9 раз!

И так, или близко к тому, будет действительно, с большой степенью вероятности, распределяться работа вашего реального приложения между доступными процессорами аппаратуры, если, естественно, приложение написано основываясь на горутинах. Более того, распределение нагрузки между процессорами будет происходить автоматически, без вашего вмешательства (изменений в код, перекомпиляции и др.): на 2-х процессорах работа будет распределяться на 2, на 8-ми процессорах та же работа будет распределяться на 8 ... на 100 процессорах работа будет распределяться на 100.

Но! «Дьявол кроется в деталях»! Это слишком хорошо чтобы быть так... В нашей рабочей функции горутин всё-таки каким-то образом происходит переключения контекста ядра (вытесняющее переключение между системными потоками), хотя и не происходит никаких блокированных ожиданий, который бы искажали и аннулировали полученные нами результаты. И происходит это, как я почти уверен в этом, в точках системных вызовов Linux, обращений к ядру операционной системы, и у нас это системный вызов получения текущего времени, укрытый в вызовах `time.Now()` внутри горутин³³. Это не отменяет полученных нами результатов, не делает их незначительными... , потому что в любом вашем реальном коде в горутине всегда будут иметь место точки системных вызовов Linux. Но это понуждает нас поискать и ещё один, более тщательный результат.

2-й подход к снаряду...

Изменим рабочую функцию потока так, чтобы во время выполнения цикла не было необходимости никаким образом обращаться к системным вызовам Linux, всё что необходимо от системы нужно получить либо до запуска, либо после завершения горутин. Тогда в цикле рабочей функции можно положить любой более-менее трудоёмкий вычислительный процесс. Я в таком качестве возьму вычисление из комплексной математики $\exp(-j\omega t)$ — базовая основа теоретической электротехники переменного тока ... но вы можете использовать что вам ближе. А для фиксации временного интервала (1 секунда) используем не службой времени, а произведём предварительную калибровку, которая даст число циклов требующих такого времени.

Пользуясь случаем, добавим подсчёт гистограммы числа горутин, приходящихся на каждый видимый Go системе поток ядра — как, с какой плотностью горутин распределяются по **очередям** исполняющих их потоков. (Заодно мы содержательно поработаем с хеш-таблицами Go как встроенным типом данных — очень могучим типом данных на котором построена, в частности вся гибкость языка Python).

m1par2.go

```
package main
```

```
//#include <pthread.h>
//unsigned long tid(void) { return (unsigned long)pthread_self(); }
import "C"
import (
    "fmt"; "os"; "runtime"
    "math"; "math/cmplx"
    "strconv"; "time"
)
```

```
type msg struct { // структура сообщения
    n int           // номер ветви
```

³³ Это сильно напоминает квантово-механические эффекты, когда присутствие наблюдателя (попытка зафиксировать положение объекта) тут-же вызывает, в силу принципа неопределённости, огромную неопределённость в его (объекта) скорости.


```

    t int64      // системный pthread_t
    d int        // задержка в ms
    s time.Time  // время старта
}
func (p msg) String() string {      // метод форматирования сообщения
    return fmt.Sprintf("[%02d,%X,%03d]", p.n, p.t, p.d)
}

func main() {
    fmt.Printf("число процессоров в системе: %v\n", runtime.NumCPU())
    ветви := 3
    debug := true
    if len(os.Args) > 1 {
        ветви, _ = strconv.Atoi(os.Args[1])
        if ветви < 0 {
            debug = false
            ветви = -ветви
        }
    }
    fmt.Printf("число ветвей выполнения: %v\n", ветви)

    активность := func(n int64) {    // функция активной паузы
        var β, ω, t complex128 = 0.0, math.Pi / 5, 0.0
        const step = 1000
        n *= step
        for n != 0 {
            β += cmplx.Exp(-1i * ω * t)
            t += 0.1
            n--
        }
    }

    count := func() int64 {
        t0, t1 := time.Now(), time.Now()
        var count_1sec int64 = 0
        for t1.Sub(t0).Seconds() < 1 { // калибровка паузы 1s
            активность(1)
            count_1sec++
            t1 = time.Now()
        }
        return count_1sec
    }()

    delay := func() { активность(count) }

    t0 := time.Now()
    delay()
    if debug {
        fmt.Printf("%s\n", msg{0, int64(C.tid()),
            int(time.Now().Sub(t0).Milliseconds()), t0}.String())
    }
    var table = map[int64] int {int64(C.tid()) : 1}

    ch := make(chan msg)
    t0 = time.Now()
    for i := 1; i <= ветви; i++ {    // параллельные ветки
        t1 := time.Now()
        go func(n int) {
            delay()
            ch <- msg{int(n), int64(C.tid()), 0, t1}
        }(i)
    }
    for i := 1; i <= ветви; i++ {

```

```

        m := <-ch
        if debug {
            m.d = int(time.Now().Sub(m.s).Milliseconds())
            fmt.Printf("%s\n", m.String())
        }
        elem, ok := table[m.t]
        if ok {
            table[m.t] = elem + 1
        } else { table[m.t] = 1 }
    }
    fmt.Printf("итоговое время параллельного выполнения: %v\n",
        time.Now().Sub(t0))
    for key, value := range table { // гистограмма использования потоков
        fmt.Printf("%X => [%d]\n", key, value)
    }
}

```

Начинаем проверку опять с сервера R420, потенциально до 40 протоколов...

\$./mlpar2

```

число процессоров в системе: 40
число ветвей выполнения: 3
[00,7F535E64B740,988]
[02,7F535E64B740,1001]
[03,7F535B3F7700,1005]
[01,7F535BBF8700,1008]
итоговое время параллельного выполнения: 1.008215142s
7F535E64B740 => [2]
7F535B3F7700 => [1]
7F535BBF8700 => [1]

```

\$./mlpar2 7

```

число процессоров в системе: 40
число ветвей выполнения: 7
[00,7F284BD9E740,992]
[07,7F284BD9E740,1007]
[02,7F2843FFF700,1016]
[04,7F284934B700,1019]
[05,7F28427FC700,1020]
[01,7F2848B4A700,1020]
[06,7F2842FFD700,1022]
[03,7F2841FFB700,1026]
итоговое время параллельного выполнения: 1.027025995s
7F2842FFD700 => [1]
7F2841FFB700 => [1]
7F284BD9E740 => [2]
7F2843FFF700 => [1]
7F284934B700 => [1]
7F28427FC700 => [1]
7F2848B4A700 => [1]

```

\$./mlpar2 -20

```

число процессоров в системе: 40
число ветвей выполнения: 20
итоговое время параллельного выполнения: 1.373806236s
7F61F3FFF700 => [1]
7F620AC30700 => [1]
7F61CEFFD700 => [1]
7F620D683740 => [2]
7F61CCFF9700 => [1]
7F6208BEC700 => [1]
7F6209C2E700 => [1]
7F620A42F700 => [1]

```

```
7F61F0FF9700 => [1]
7F61CDFFB700 => [1]
7F61CD7FA700 => [1]
7F61F17FA700 => [1]
7F61F2FFD700 => [1]
7F61CF7FE700 => [1]
7F61F27FC700 => [1]
7F61CE7FC700 => [1]
7F61AFFFF700 => [1]
7F61F37FE700 => [1]
7F61CFFFF700 => [1]
7F61F1FFB700 => [1]
```

Пока число горутин меньше числа процессоров (потоков) — горутин распределяются по одной на каждый процессор (одна, [2] в каждом выводе — это искусственно добавленная в эту хеш-таблицу записи о потоке главной программы `main()`, чтобы убедиться что она не занимает отдельный поток).

Цифры производительности теперь стали гораздо реалистичнее: 40 параллельных ветвей увеличивают время относительно одиночной ветви на 37%.

Посмотрим как поведёт себя приложение при заказанном числе параллельных ветвей больше числа доступных процессоров:

```
$ ./mlpar2 -50
```

```
число процессоров в системе: 40
```

```
число ветвей выполнения: 50
```

```
итоговое время параллельного выполнения: 2.318548612s
```

```
7F10AB7FE700 => [1]
7F10B366740 => [2]
7F1006FFD700 => [2]
7F10AA7FC700 => [1]
7F1067FFF700 => [1]
7F1045FFB700 => [1]
7F10657FA700 => [1]
7F10257FA700 => [1]
7F10667FC700 => [3]
7F10477FE700 => [2]
7F1026FFD700 => [2]
7F10877FE700 => [1]
7F10A97FA700 => [1]
7F1007FFF700 => [1]
7F1085FFB700 => [2]
7F10B0C13700 => [1]
7F10ABFFF700 => [1]
7F1086FFD700 => [1]
7F1065FFB700 => [3]
7F1064FF9700 => [2]
7F10267FC700 => [1]
7F1046FFD700 => [1]
7F1087FFF700 => [1]
7F10457FA700 => [1]
7F1084FF9700 => [1]
7F10067FC700 => [2]
7F1044FF9700 => [1]
7F10677FE700 => [1]
7F10A8FF9700 => [1]
7F1066FFD700 => [1]
7F10867FC700 => [1]
7F1025FFB700 => [1]
7F10277FE700 => [1]
7F1027FFF700 => [2]
7F10857FA700 => [2]
7F1047FFF700 => [1]
```

```
7F1024FF9700 => [1]
7F10467FC700 => [1]
```

\$./mlpar2 -100

число процессоров в системе: 40

число ветвей выполнения: 100

итоговое время параллельного выполнения: 4.547764482s

```
7F3B07FFF700 => [1]
7F3B857FA700 => [2]
7F3AE7FFF700 => [3]
7F3B06FFD700 => [2]
7F3B66FFD700 => [1]
7F3B877FE700 => [1]
7F3B657FA700 => [3]
7F3B677FE700 => [3]
7F3B47FFF700 => [3]
7F3B84FF9700 => [4]
7F3B457FA700 => [4]
7F3B9D685700 => [2]
7F3B64FF9700 => [3]
7F3B267FC700 => [2]
7F3B667FC700 => [2]
7F3B067FC700 => [3]
7F3B65FFB700 => [2]
7F3B277FE700 => [1]
7F3B44FF9700 => [1]
7F3B87FFF700 => [4]
7F3B86FFD700 => [1]
7F3B077FE700 => [1]
7F3B24FF9700 => [4]
7F3B9E687700 => [2]
7F3B04FF9700 => [1]
7F3BA10DA740 => [3]
7F3B257FA700 => [3]
7F3B467FC700 => [3]
7F3B05FFB700 => [3]
7F3B867FC700 => [2]
7F3B67FFF700 => [1]
7F3B27FFF700 => [5]
7F3B057FA700 => [3]
7F3B25FFB700 => [1]
7F3B85FFB700 => [5]
7F3B46FFD700 => [2]
7F3B9DE86700 => [3]
7F3B26FFD700 => [7]
7F3B477FE700 => [3]
7F3B45FFB700 => [1]
```

Хорошо видно более-менее равномерное разбрасывание порождаемых горутин по системным потокам их выполняющим. А время выполнения 100 «порций» работы превышает единичное (всего) в 4.5 раз.

И снова обратимся к другой крайности - миниатюрные микрокомпьютеры ARM.

Raspberry Pi 2 :

\$./mlpar2

число процессоров в системе: 4

число ветвей выполнения: 3

```
[00,76FE01C0,995]
[03,76FE01C0,998]
[02,64FFE440,1007]
[01,661FF440,1024]
```

итоговое время параллельного выполнения: 1.024868641s

```
76FE01C0 => [2]
64FFE440 => [1]
661FF440 => [1]
```

\$./mlpar2 7

```
число процессоров в системе: 4
число ветвей выполнения: 7
[00,76FA11C0,966]
[01,76FA11C0,1529]
[07,76FA11C0,1648]
[04,661FF440,1673]
[02,657FF440,1714]
[03,76FA11C0,1741]
[06,64FFE440,1757]
[05,661FF440,1767]
итоговое время параллельного выполнения: 1.768285034s
661FF440 => [2]
657FF440 => [1]
64FFE440 => [1]
76FA11C0 => [4]
```

\$./mlpar2 -20

```
число процессоров в системе: 4
число ветвей выполнения: 20
итоговое время параллельного выполнения: 4.826756735s
76F971C0 => [5]
661FF440 => [4]
657FF440 => [6]
64FFE440 => [6]
```

\$./mlpar2 -100

```
число процессоров в системе: 4
число ветвей выполнения: 100
итоговое время параллельного выполнения: 25.274911046s
76FA21C0 => [26]
64FFE440 => [30]
661FF440 => [21]
657FF440 => [24]
```

Здесь картина T(N) ещё более реалистичная! А распределение горутин по процессорам ещё нагляднее.

И самый минималистичный вариант — Orange Pi One (и это всё в 512Mb операционной системы):

\$./mlpar2

```
число процессоров в системе: 4
число ветвей выполнения: 3
[00,B6F47D40,957]
[01,A43FF450,937]
[03,B6F47D40,937]
[02,A57FF450,937]
итоговое время параллельного выполнения: 937.670202ms
B6F47D40 => [2]
A43FF450 => [1]
A57FF450 => [1]
```

\$./mlpar2 -9

```
число процессоров в системе: 4
число ветвей выполнения: 9
итоговое время параллельного выполнения: 2.141070976s
A57FF450 => [1]
B6F7AD40 => [5]
```

A4FFE450 => [3]

A61FF450 => [1]

\$./mlpar2 -20

число процессоров в системе: 4

число ветвей выполнения: 20

итоговое время параллельного выполнения: 4.435031013s

A43FF450 => [5]

A4FFE450 => [6]

A57FF450 => [5]

B6F50D40 => [5]

\$./mlpar2 -100

число процессоров в системе: 4

число ветвей выполнения: 100

итоговое время параллельного выполнения: 23.508637053s

B6FD0D40 => [31]

A4FFE450 => [24]

A57FF450 => [29]

A61FF450 => [17]

Но отметим одну не очевидную вещь: как бы мы не стремились избежать прерываний и перевода в блокированные состояния потоков ядра, выполняющих горутины, мы не можем 100% гарантировать это — помимо своего приложения на Go, мы всё ещё находимся в мультизадачной операционной системе Linux с вытесняющей многозадачностью, где рабочие потоки нашего приложения конкурируют за ресурсы с потоками других (в том числе и системных) процессов.

0 числе потоков исполнения

Выше была описана схема в которой, по умолчанию, как было описано:

- При запуске приложения Go запускается скрытым образом GOMAXPROCS рабочих потоков ядра `pthread_t`, которые выполняют горутину, и к каждому из которых прикреплен очередь исполняющихся горутин.
- При этом каждый поток ядра оказывается прикрепленным к одному из физических процессоров системы, число которых `runtime.NumCPU()`.
- При такой схеме каждый их выполняющихся потоков не вытесняемый (в мультизадачной операционной системе Linux), по крайней мере, не вытесняемый GOMAXPROCS-1 конкурирующими потоками этого приложения (но могут вытесняться другими приложениями Linux, что может несколько нарушать оптимизм всей этой модели).
- Оптимистичность такой модели зиждется на том, что отдельные горутины выполняются в контексте своих фиксированных потоков, и при их переключении нет необходимости ни перегружать контекст (регистры процессора, стек, ...), ни перезагружать содержимое кэш-памяти.

Но отсюда может возникнуть мнение (и оно неявно не опровергалось описаниями предыдущих глав), что число потоков выполнения Go приложения GOMAXPROCS а). всегда устанавливается равным числу физических процессоров `runtime.NumCPU()` и б). это жёсткая связь между числом процессоров и числом потоков диспетчирования горутин. Но это не совсем так!

Если вы хорошо понимаете цели с которыми это делаете, вы можете всегда указать своему приложению Go использовать число потоков выполнения (горутин) меньше или больше чем существующее число процессоров в аппаратуре³⁴. Для подтверждения этого утверждения сделаем почти игрушечное **диагностическое** приложение (каталог `scheduler`):

³⁴ Но без понимания цели такое лучше не делать — у Go весьма совершенная схема использования физических аппаратных ресурсов!

numproc.go

```
package main
import ("fmt"; "os"; "runtime"; "strconv")

func main() {
    потоки := 1
    if len(os.Args) > 1 {
        потоки, _ = strconv.Atoi(os.Args[1])
        if потоки > 1 {
            runtime.GOMAXPROCS(потоки)
        }
    }
    fmt.Printf("число процессоров в системе: %v\n", runtime.NumCPU())
    fmt.Printf("число потоков исполнения: %v\n", runtime.GOMAXPROCS(-1))
}
```

Значение переменной окружения GOMAXPROCS (из пакета runtime) можно изменить, и сделать это можно разными способами:

- Из программного кода вызовом runtime.GOMAXPROCS() с положительным значением единственного параметра (при отрицательном значении вызов только возвращает текущее установленное значение GOMAXPROCS, при положительном параметре возвращается предыдущее установленное значение).
- Изменением переменной окружения GOMAXPROCS системы при запуске Go приложения (при этом сам код приложения совершенно ничего «не знает» об изменении числа потоков исполнения).

А теперь иллюстрируем сказанное... На процессоре (системное приложение nproc Linux даёт нам, для контроля, физическое число процессорных ядер):

```
$ nproc
40
```

В данном случае, мы действительно экспериментируем на сервере промышленного уровня. А теперь наше созданное приложение:

```
$ ./numproc
число процессоров в системе: 40
число потоков исполнения: 40
```

```
$ ./numproc 5
число процессоров в системе: 40
число потоков исполнения: 5
```

```
$ export GOMAXPROCS=7; ./numproc
число процессоров в системе: 40
число потоков исполнения: 7
```

```
$ ./numproc
число процессоров в системе: 40
число потоков исполнения: 7
```

```
$ GOMAXPROCS=9; ./numproc
число процессоров в системе: 40
число потоков исполнения: 9
```

Но это ещё далеко не всё! Это мы переопределяли число доступных потоков средствами самой исполнимой системы Go. Это так и будет работать в любой операционной системе. Но в операционной системе Linux мы можем гибко управлять распределением любых приложений по процессорам средствами самой операционной системы. Делается это малоизвестной консольной Linux-командой taskset, которая управляет аффинити маской любого процесса.

Начнём с того, что вспомним как «попросить» у операционной системы её информацию о числе доступных физических процессоров. Вот несколько простейших способов:

```
$ cat /proc/cpuinfo | grep processor | wc -l
40
```

```
$ nproc
40
```

```
$ lscpu | grep 'CPU(s)'
CPU(s):                                40
On-line CPU(s) list:                  0-39
NUMA node0 CPU(s):                    0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38
NUMA node1 CPU(s):                    1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39
```

Каждый процесс в Linux имеет собственную индивидуальную аффинити маску — слово в котором каждый установленный бит отмечает процессор (в показанной выше нумерации), который этот процесс может использовать. (Заметим, что и каждому потоку в рамках процесса также соответствует своя индивидуальная аффинити маска, но нас в данном контексте интересуют только процессы.) Команда `taskset` позволяет посмотреть или изменить (что нас интересует гораздо больше) аффинити маску процесса:

```
$ taskset --help
```

```
Usage: taskset [options] [mask | cpu-list] [pid|cmd [args...]]
```

Show or change the CPU affinity of a process.

Options:

-a, --all-tasks	operate on all the tasks (threads) for a given pid
-p, --pid	operate on existing given pid
-c, --cpu-list	display and specify cpus in list format
-h, --help	показать эту справку
-V, --version	показать версию
...	

Например, в простейшем применении — диагностика:

```
$ ps
  PID TTY          TIME CMD
 8148 pts/5        00:00:00 bash
11759 pts/5        00:00:00 ps
```

```
$ taskset -p 8148
```

```
pid 8148's current affinity mask: ffffffff
```

Как видно, в обычном состоянии процессам разрешено распределяться на все физические процессоры. Но это можно изменить, причём как по числу, так и по выборочному подмножеству конкретных процессоров.

```
$ taskset -c 0-3 ./numproc
```

```
число процессоров в системе: 4
```

```
число потоков исполнения: 4
```

```
$ taskset -c 32,34,36,38 ./numproc
```

```
число процессоров в системе: 4
```

```
число потоков исполнения: 4
```

Вспомните что мы обсуждали ранее относительно гипертрэдинга и странностей нумерации ядер процессоров... Используя возможности `taskset` мы можем запускать свои приложения (и не только тестовые) так, чтобы они выполнялись только на физических ядрах без их гипертрэдинговых пар, или в многопроцессорных серверах (SMP) так, чтобы приложения выполнялись на выбранном чипе. Примеры (выполняемое приложение мы сделали и обсуждали ранее, а процессоры сервера DELL описывались

при рассмотрении нумерации), только для образца:

- выполнение на всех процессорах без ограничений:

```
$ ./mlpar -40
```

```
число процессоров в системе: 40
```

```
число ветвей выполнения: 40
```

```
итоговое время выполнения: 1.066720489s
```

- выполнение 40 горутин на 4-х процессорах выбранных без пересечений в гипертрэдинге:

```
$ taskset -c 0-3 ./mlpar -40
```

```
число процессоров в системе: 4
```

```
число ветвей выполнения: 40
```

```
итоговое время выполнения: 1.399913902s
```

- выполнение 40 горутин на 3-х процессорах только одного (из 2-х) физического чипа:

```
$ taskset -c 0,2,4 ./mlpar -40
```

```
число процессоров в системе: 3
```

```
число ветвей выполнения: 40
```

```
итоговое время выполнения: 1.370372032s
```

- выполнение 40 горутин на 2-х процессорах составляющих гипертрэдинговую пару общего ядра (специально выбранные):

```
$ taskset -c 0,20 ./mlpar -40
```

```
число процессоров в системе: 2
```

```
число ветвей выполнения: 40
```

```
итоговое время выполнения: 1.407447393s
```

Предполагаю, что этого разъяснения вполне достаточно для того, чтобы вы легко могли варьировать (и изучать) число процессоров (потоков) на которых станет выполняться ваше Go приложение. Мы проделывали это для экспериментирования, но такие возможности могут оказаться крайне востребованными а). во встраиваемых системах на уровне оптимизации железа, так и б). в сложных высоко нагруженных системах непрерывного функционирования, 365x24.

Источники информации

[1] Механизм планирования сопрограмм Golang и настройка производительности GOMAXPROCS —

<https://russianblogs.com/article/9411826814/>

[2] Armbian. Linux for ARM development boards — <https://www.armbian.com/orange-pi-one/>

[3] Raspberry Pi Documentation — <https://www.raspberrypi.com/documentation/>

[4] Raspberry Pi OS — <https://www.raspberrypi.com/documentation/computers/os.html>

[5] DELL PowerEdge R420. Technical Guide — <https://content.etalize.com/User-Manual/1024095456.pdf>

[6] О.Цилиурик, Параллелизм, конкурентность, многопроцессорность в Linux, 2014 — <http://mylinuxprog.blogspot.com/2014/09/linux.html>

Часть 3. Некоторые примеры и сравнения

*В восточных странах жара и праздная
жизнь порождает у обывателей
досужие и смертоносные фантазии.
Роберт Грэм Ирвин «Арабский кошмар»*

Эта часть менее формализованная чем предыдущие... В этой части будет показана некоторая подборка приложений, которые автору, из совершенно разных соображений, показались интересными (по причине жары и праздной жизни, как сказано в эпиграфе). Все они присутствуют в архиве примеров, и даже с детальными журналами (файлами вида *.hist) сборки и выполнения. Для многих примеров будут параллельно показаны сравнительные реализации на Go и на C или C++ (иногда для сравнения скоростных показателей, иногда в качестве аналогии, и того как это выглядит).

Осваиваемся в технологии Go

Язык, который не меняет вашего представления о программировании, достоин изучения.
Alan J. Perlis

В первом разделе этой части текста мы рассмотрим несколько хорошо известных и понятных задач, выраженных в языке Go. Здесь мы просто в конкретном контексте посмотрим на использование тех синтаксических возможностей, которые обсуждали ранее.

После этого, в следующих разделах, мы рассмотрим более тонкие вещи, возвращающие нас к главной цели работы — параллельное и многопроцессорное выполнение.

Консольный ввод/вывод

Первейшей задачей, которую станет решать всяк знакомящийся с Go, по крайней мере следующей попыткой вослед за «Hello World», это будет ввод исходных данных и вывод итоговых результатов выполнения для своей консольной (CLI) программы (потому что иные, графические, GUI, интерфейсы программ в этой книге мы не рассматриваем).

В отношении вывода в терминал мы уж достаточно подробно рассматривали ранее. Разве что, в порядке напоминания, подытожим — у вас есть две основные группы API:

1. Встроенные функции языка `print()` и `println()`, которые не требуют импорта никаких программных пакетов Go. Это простейшая форма, функции эти принимают список параметров (объектов вывода) произвольной длины. Каждый элемент вывода выводится в том своём единственном «нативном» формате, который для типа этого элемента представляется методом `String()`, реализующим интерфейс:

```
type Stringer interface {
    String() string
}
```
2. Функции из пакета импорта `fmt` (<https://pkg.go.dev/fmt@go1.22.0>): `Print()`, `Println()`, `Printf()`. Этот пакет позволяет форматировать вывод самым замысловатым образом, в зависимости от предполагаемых по ситуации форматов, что подробно описывалось ранее при описании языка.

Всё гораздо сложнее обстоит с вводом данных. Ниже мы приведём несколько способов ввода данных с клавиатуры, используя разные импортируемые пакеты Go. Это представляется полезным в разных аспектах:

- предоставляет готовые образцы, с которыми можно сразу приступить к написанию своего пробного кода, чтобы не искать требуемые для этого пакеты;
- демонстрирует как в Go одни и те же потребности могут обеспечиваться самыми разнообразными способами, имеется в виду не различия в написании кода, а за счёт импорта разных пакетов, используя их специфику;
- попутно демонстрируется ряд трансформаций над строчными переменными...

Итак... Мы построим несколько схем ввода с клавиатуры строчных данных (данные вводимые с клавиатуры всегда строчные, а что с ними уже делать дальше — это вы решите сам затем в программном коде). Но производим ввод так, чтобы иметь «цивильный» способ завершения этого ввода, а не так чтобы грубо «рубить» всю программу по `Ctrl+C`. Завершать ввод мы будем по традиционному в UNIX «конец ввода» (EOF): клавиатурной комбинации `Ctrl+D`, а ещё, альтернативно, по вводу пустой строки по `Enter`. (Способ завершения ввода на практике оказывается не менее важен чем само его осуществление). Наши короткие примеры в качестве результата производят эхо-повтор

введенной строки (устраняя, где надо хвостовой символ перевода строки!) ... а ваш код, соответственно, будет передавать их куда-то на осмысленную обработку. Эти примеры находятся в каталоге `strings/inout` архива (потому что ввод, повторюсь — это прежде всего строки).

Первым способом у нас будет использование функций `Scan()`, `Scanln()`, `Scanf()`, всё из того же пакета `fmt`. Это первое что просится «под руку» при изучении Go, но это ... пожалуй, и самый неудачный из показанных способов:

inputf.go

```
package main

import "fmt"

func main() {
    for {
        var line string
        n, _ := fmt.Scanf("%s", &line)
        if 0 == n {      // ввод Enter - пустая строка
            break // ввод ^D - EOF
        }
        println(line)
    }
}
```

Если в строке ввода будет несколько слов (элементов), разделённых пробелами или табуляцией, то функции группы `fmt.Scan()` за один вызов вводят один последовательный элемент такой строки, и часто это не лучший вариант того, что мы хотели бы иметь:

\$ go run inputf.go

это одна строка

это

одна

строка

1 23 456

1

23

456

Это не совсем то, что соответствует духу и привычкам консольного Linux, сравним:

\$ echo "а это цельная фраза"

а это цельная фраза

\$ echo "а это цельная фраза" | go run inputf.go

а

это

цельная

фраза

Следующий пример будет использовать пакет `bufio` (буферизованный ввод/вывод) — это один из наиболее часто используемых пакетов для таких целей:

inputs.go

```
package main

import "bufio"
import "os"

func main() {
    scanner := bufio.NewScanner(os.Stdin)
    for scanner.Scan() {      // ввод ^D - EOF
        line := scanner.Text()
    }
}
```

```

        if 0 == len(line) {
            break // ввод Enter - пустая строка
        }
        println(line)
    }
}

```

(В 3-й строке показанного ввода числа разделены табуляцией — при тестировании должны проверяться все возможности):

\$ go run inputs.go

это одна строка

это одна строка

строка

строка

1 2 3

1 2 3

1 23 456

1 23 456

Далее мы воспользуемся из того же пакета `bufio`, но другими методами:

inputs2.go

```
package main
```

```
import "strings"
```

```
import "bufio"
```

```
import "io"
```

```
import "os"
```

```

func main() {
    reader := bufio.NewReader(os.Stdin)
    for {
        line, err := reader.ReadString('\n')
        if err == io.EOF {
            println() // ввод ^D - EOF
            break
        }
        line = strings.TrimRight(line, " \n");
        if 0 == len(line) { // ввод Enter - пустая строка
            break
        }
        println(line)
    }
}

```

\$ go run inputs2.go

это будет строка

это будет строка

дальше ^D

дальше ^D

И последний наш вариант использует непосредственно возможности консольного ввода нашей операционной системы (пакет `os`, здесь может выявиться системная зависимость, но мы обсуждаем Linux). Обратите внимание, что здесь мы оперируем не строками `string`, как в предыдущих вариантах, а массивами байт (`byte`). Взаимные преобразования UTF-8 строк и представляющих их потоков байт — это очень широко применяемая практика в Go:

inputr.go

```
package main
```

```
import "os"
```

```
import "io"

func main() {
    line := make([]byte, 160)
    for {
        n, err := os.Stdin.Read(line)
        if err == io.EOF { // ввод ^D - EOF
            break
        }
        s := string(line[:n-1])
        if 0 == len(s) { // ввод Enter - пустая строка
            break
        }
        println(s)
    }
}
```

```
$ go run inputr2.go
```

```
1 23
1 23
34 567
34 567
это строка
это строка
```

Ещё одна потребность ввода, которую приходится решать для консольных программ — это, в традициях UNIX/Linux, ввод параметров (данных) и опций (ключей) в командной строке запуска программы. Мы это и сделаем в следующем примере...

Утилита echo

В качестве простого примера использования Go посмотрим пример утилиты `echo` (каталог `hello` архива примеров), близкий к тому прототипу, который приводимый в документации:

echo.go

```
package main

import (
    "flag" // парсер параметров командной строки
    "os"
)

var omitNewLine = flag.Bool("n", false, "не печатать знак новой строки")

const (
    Space    = " "
    NewLine  = "\n"
)

func main() {
    flag.Parse() // Сканирование списка аргументов и установка флагов
    var s string
    for i := 0; i < flag.NArg(); i++ {
        if i > 0 {
            s += Space
        }
        s += flag.Arg(i)
    }
    if !*omitNewLine {
        s += NewLine
    }
    os.Stdout.WriteString(s)
```

```
}
```

Как и следовало ожидать:

```
$ ./echo повторяет то что я пишу
повторяет то что я пишу
```

```
$ ./echo -n 12345
12345$
```

Пакет `flag` (импортируемый программой) отрабатывает ошибочные ситуации — ввод не описанных в коде задачи опций командной строки приводит к ошибке и генерации (автоматической) подсказки:

```
$ ./echo -z 12345
flag provided but not defined: -z
Usage of ./echo:
  -n      не печатать знак новой строки
```

Здесь попутно задействована функциональность пакета `flag`, который позволяет организовать обработку параметров и опций командной строки запуска в том стиле, который принят в UNIX/Linux (известный POSIX вызов `getopt()`). Это будет приятным подарком пишущим в Linux консольные приложения в едином принятом стиле, поэтому мы останавливаемся на нём так подробно.

Есть и небольшое отличие в поведении от привычного `getopt()` — все опции командной строки пакета `flag` должны обязательно предшествовать параметрам командной строки. В противном случае они рассматриваются уже как последующие параметры:

```
$ ./echo 12345 -n
12345 -n
```

Но такая особенность (обязательное предшествование опций параметрам в командной строке) — это общее поведение всех вообще команд и самого GoLang (в отличие от привычных команд-утилит Linux), это их специфика.

Ещё одно отличие от привычек командной строки Linux (если вы уже успели к ней привыкнуть) состоит в записи значений параметров (чисел, строк...) тем опциям, которые требуют значений. На примере одной из задач, которую мы будем подробно рассматривать позже (сейчас нас интересует только синтаксис записи опций):

- Нормально:

```
$ ./mining -w=20
горутин 1; размер блока 1000
...
```

- Нормально:

```
$ ./mining -w 20
горутин 1; размер блока 1000
...
```

- А вот это, что традиционно для Linux, да и вообще для UNIX, для `getopt()` API, ошибочно:

```
$ ./mining -w20
flag provided but not defined: -w20
Usage of ./mining:
  -b int           размер блока (default 1000)
  -g int           число горутин (default 1)
  -v uint
```

```
        лимит поиска (default 32767)
-w int
        ширина вывода
```

P.S. Дефолтные опции подсказки (-h, --help), как это принято в Linux, пакет `flag` создаёт за вас сам, исходя из определений списка используемых опций (`flag.Bool` у нас, как пример, и подобные ему), вам этого делать не надо (это ещё один приятный бонус):

```
$ ./echo -h
Usage of ./echo:
  -n      не печатать знак новой строки

$ ./echo --help
Usage of ./echo:
  -n      не печатать знак новой строки
```

Эти особенности пакета `flag` (отличия от `getopt()`) нужно держать в уме, иначе они подготовят вам немало удивительных сюрпризов при отлаке ваших программных кодов. Мы ещё не раз встретимся с использованием пакета `flag` в последующих примерах кода (включённых в архив примеров к книге).

Подробную информацию (с примерами использования) того, как воспользоваться возможностями пакета `flag` см. здесь: <https://pkg.go.dev/flag>.

Итерационное вычисление вещественного корня

Здесь будет показана реализация функции вычисления квадратного корня $z = \sqrt{x}$, как она может быть выражена на C и на Go (каталог `compare/sqrt`). Вычисление делается методом Ньютона, когда на каждой итерации вычисляется: $z_{i+1} = z_i - (z_i^2 - x) / (2 * z_i)$.

Реализация (всё в каталоге `compare/sqrt`) на C:

```
sqrt.c.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

long double eps = 1e-9;
int itr = 0;

long double Sqrt(long double arg) {
    long double z = 1.;
    while(1) {
        long double z1 = z - (z * z - arg) / 2. / z;
        if(fabs(z1 - z) / z < eps) return z;
        z = z1;
        itr++;
    }
}

int main(int argc, char **argv) {
    long double sqr = Sqrt(atof(argv[1]));
    printf("[%d]: %.16Lf\n", itr, sqr);
    return 0;
}
```

Реализация на Go:

```
sqrt.go.go
package main
```

```
import (
```



```

        "fmt"
        "math"
        "os"
        "strconv"
    )

    const eps = 1e-9

    func sqrt(x float64) (float64, int) {
        z := float64(1)
        var i int = 0
        for {
            z1 := z - (z*z-x)/2/z
            if math.Abs(z1-z)/z < eps {
                return z, i
            }
            z = z1
            i++
        }
    }

    func main() {
        v, _ := strconv.ParseFloat(os.Args[1], 64)
        var n int
        v, n = sqrt(v)
        fmt.Printf("[%v]: %v\n", n, v)
    }

```

Каждый файл исходного кода (для сравнений) собирается дважды: `sqrt_c.c` компиляторами GCC и Clang, а `sqrt_go.go` — с помощью команд `gccgo` и `go`. Файл сборки:

Makefile

```

BASE = sqrt
TASK = $(BASE)_c $(BASE)_cl $(BASE)_go $(BASE)_gc
all: $(TASK)
%: %.c
    gcc -O0 -lm $< -o $@
%: %.go
    gccgo $< -g -O0 -o $@
$(BASE)_cl: $(BASE)_c.c
    clang -O0 $< -o $@
$(BASE)_gc: $(BASE)_go.go
    go build -o $@ -compiler gc $<
clean:
    rm -f $(TASK)

```

В итоге, после сборки получим 4 приложения:

```

$ ls -l | grep x
-rwxrwxr-x. 1 olej olej   8721 авг 14 02:13 sqrt_c
-rwxrwxr-x. 1 olej olej   8769 авг 14 02:13 sqrt_cl
-rwxrwxr-x. 1 olej olej 2245728 авг 14 02:13 sqrt_gc
-rwxrwxr-x. 1 olej olej   28827 авг 14 02:13 sqrt_go

```

Результаты сравнительного выполнения (в скобках выводится число итераций, посредством которых достигается сходимость 1E9):

```

$ ./sqrt_c 10
[6]: 3.1622776601683793

```

```

$ ./sqrt_cl 10
[6]: 3.1622776601683793

```

```
$ ./sqrt_go 10
[6]: 3.1622776601683795
```

```
$ ./sqrt_gc 10
[6]: 3.1622776601683795
```

Вычисление числа π

В этой части мы посмотрим как в Go организована работа с большими числами — числами произвольной разрядности, пакет `math/big`. В качестве примера использования больших чисел рассмотрим (каталог архива `types`) пример приводимый Марком Саммерфильдом в его книге (в заметно упрощённом виде). Где вычисляется значение числа π с произвольно большим (100, 1000, ...) числом значащих цифр по формуле Мэчина (1706г.), формула выглядит так:

$$\pi = 4 * (4 * \operatorname{arccot}(5) - \operatorname{arccot}(239))$$

$$\text{где: } \operatorname{arccot}(x) = 1/(x) - 1/(3*x^3) + 1/(5*x^5) - 1/(7*x^7) + \dots$$

Естественно, в форме вещественного числа мы проводить вычисления со столькими значащими цифрами не можем, поэтому будем вычислять его в целочисленном эквиваленте, сдвинутом на соответствующее (требуемому) число десятичных позиций (плюс несколько позиций во избежание округления при вычислениях). Пример на Go базируется на реализации в Python, где хитрое хранение целочисленных значений в этом языке допускает естественным образом их представление с произвольным числом значащих цифр, без всяких сторонних пакетов. Поэтому в качестве идеи разумно посмотреть сначала вариант Python, который проще и понятнее на уровне алгоритмики:

pi.py

```
#!/usr/bin/python3
import sys

def arccot(x, unity):
    sum = xpower = unity // x
    n = 3
    sign = -1
    while 1:
        xpower = xpower // (x*x)
        term = xpower // n
        if not term:
            break
        sum += sign * term
        sign = -sign
        n += 2
    return sum

def pi(digits):
    unity = 10 ** (digits + 10)
    pi = 4 * (4 * arccot(5, unity) - arccot(239, unity))
    return pi // 10 ** 10

print(pi(int(sys.argv[1])))
```

Для вычисления N десятичных знаков числа π после запятой вычисляется большое целочисленное значение π сдвинутое на N позиций влево (умноженное на 10^{N+10}). Функция `pi()` начинается с вычисления значения переменной `unity` (10^{N+10}), которое используется как коэффициент масштабирования для вычислений с использованием целых чисел. Слагаемое `+10` увеличивает дополнительно на 10 число цифр, заказанное пользователем, чтобы избежать ошибок округления. Затем используется формула Мэчина с модифицированной версией функции `arccot()`, которой во втором аргументе передается переменная `unity`. После вычисления возвращается результат, деленный на 234

10^{10} , чтобы устранить эффект увеличения коэффициента масштабирования `unity`. Все эти операции, естественно, целочисленные с неограниченным числом разрядов.

Вариант на Go демонстрирует работу с целочисленными значениями неограниченно большого размера — ещё один пакет: `math/big`. Этот вариант реализует в точности ту же схему вычислений (обе реализации в каталоге примеров `types`):

pi.go

```
package main

import (
    "fmt"
    "math/big"
    "os"
    "strconv"
)

func main() {
    x, _ := strconv.Atoi(os.Args[1])
    fmt.Println( $\pi$ (x))
}

func  $\pi$ (places int) *big.Int {
    digits := big.NewInt(int64(places))
    unity := big.NewInt(0)
    ten := big.NewInt(10)
    exponent := big.NewInt(0)
    unity.Exp(ten, exponent.Add(digits, ten), nil)
    pi := big.NewInt(4)
    left := arccot(big.NewInt(5), unity)
    left.Mul(left, big.NewInt(4))
    right := arccot(big.NewInt(239), unity)
    left.Sub(left, right)
    pi.Mul(pi, left)
    return pi.Div(pi, big.NewInt(0).Exp(ten, ten, nil))
}

func arccot(x, unity *big.Int) *big.Int {
    sum := big.NewInt(0)
    sum.Div(unity, x)
    xpower := big.NewInt(0)
    xpower.Div(unity, x)
    n := big.NewInt(3)
    sign := big.NewInt(-1)
    zero := big.NewInt(0)
    square := big.NewInt(0)
    square.Mul(x, x)
    for {
        xpower.Div(xpower, square)
        term := big.NewInt(0)
        term.Div(xpower, n)
        if term.Cmp(zero) == 0 {
            break
        }
        addend := big.NewInt(0)
        sum.Add(sum, addend.Mul(sign, term))
        sign.Neg(sign)
        n.Add(n, big.NewInt(2))
    }
    return sum
}
```

В итоге:

```
$ ./pi.py 80
```

```
314159265358979323846264338327950288419716939937510582097494459230781640628620899
```

```
$ go run pi.go 80
```

```
314159265358979323846264338327950288419716939937510582097494459230781640628620899
```

Вычисления неограниченно высокой точности

Если целочисленные вычисления огромной разрядности — это всё-таки, в некоторой мере, экзотика, то тот же пакет `math/big` предоставляет и другую возможность: вещественных вычислений неограниченно высокой точности. Вещественные величины здесь представляются с мантиссой произвольной длины (в принципе, представление определяется только доступной памятью компьютера).

Для иллюстрации этой техники мы сделаем вычисление квадратного корня из числа 2.0 методом Ньютона (нас совершенно не интересует сам выбор алгоритма, выбираем что проще). Вычисление — итерационное:

- Используем произвольное начальное значение, мы станем использовать $x[0] = 1.0$;
- На каждом шаге итерации: $x[n+1] = 1/2 * (x[n] + (2.0 / x[n]))$;
- Утверждается (в литературе), что метод Ньютона удваивает количество правильных цифр при каждой итерации, поэтому мы можем заранее вычислить требуемое число итераций.

Поскольку язык Go не допускает переопределение операций (+, -, *, / ...), как C++, то для математических операций пакет предоставляет соответствующие методы: `big.Rat.Add()`, `big.Rat.Sub()`, `big.Rat.Mul()`, `big.Rat.Quo()` и ещё много других, обеспечивающих все потребности операций и преобразований.

Код вычисления (каталог `types`):

newton.go

```
package main

import (
    "fmt"
    "math"
    "math/big"
)

func main() {
    const prec = 250 // точностью 250 бит в
    мантиссе
    steps := int(math.Log2(prec)) // число итераций
    two := new(big.Float).SetPrec(prec).SetInt64(2) // инициализации
    half := new(big.Float).SetPrec(prec).SetFloat64(0.5)
    x := new(big.Float).SetPrec(prec).SetInt64(1) // начальное значение
    t := new(big.Float).SetPrec(prec) // результат
    for i := 0; i <= steps; i++ { // итерации
        t.Quo(two, x) // t = 2.0/x[n]
        t.Add(x, t) // t = x[n] + (2.0/x[n])
        x.Mul(half, t) // x[n+1] = 0,5 * t
    }
    // Могут использовать форматы fmt.Printf: big.Float реализует fmt.Formatter
    fmt.Printf("sqrt(2) = %.85f\n", x)
    t.Mul(x, x)
    fmt.Printf("error = %.85f\n", t.Sub(two, t)) // ошибка
}
```

[illegible]

Подобная задача была нужна мне не один раз (я разных языках) в прагматических целях — для усреднения нескольких последовательных значений временного ряда, например при оценивании временных характеристик выполнения приложения. Но эта задача может оказаться очень полезной в весьма широком круге иных применений. Итак...

```
mean = 1 / N * Σ( X[i] )
disp = 1 / N * Σ( X[i] - mean )2
```

Если раскрыть выражение (квадрат разности) для дисперсии и произвести дальнейшие упрощения, то можно прийти к выражению: $\text{disp} = 1 / N * \sum (X[i]^2) - \text{mean}^2$. Теперь мы можем накапливать сумму квадратов последовательности чисел в потоке, а вычисление итоговых характеристик отложить на завершение.

В этом и состоит вторая цель такой задачи — мы рассматриваем как (и где) генераторы случайных представлены в Go, и для сравнения возьмём C++ (каталог `compare/sco`).

```
#include <iostream>
#include <iomanip>
#include <random>
#include <cmath>

int main(int argc, char **argv ) {
    long n = 1000;
    if(argc > 1)
        n = atoi(argv[1]);
    std::random_device rd{};
    std::mt19937 gen{rd()};
    std::normal_distribution<> d{0,1};

    double s1 = 0., s2 = 0.;
    for(int i = 0; i < n; i++) {
        double r = d(gen);
        s1 += r;
        s2 += r * r;
    }
    s1 /= n;
    s2 = s2 / n - s1 * s1;
```

```

    std::cout << n << " чисел: среднее=" << s1
              << ", СК0=" << sqrt(s2) << std::endl;
    return 0;
}

```

Тот же функционально код на Go:

skoc.go

```

package main

import (
    "fmt"
    "math"
    "math/rand"
    "os"
    "strconv"
)

func main() {
    n := 1000
    if len(os.Args) > 1 {
        n, _ = strconv.Atoi(os.Args[1])
    }
    s1, s2 := 0., 0.
    for i := 0; i < n; i++ {
        r := rand.Float64()
        s1 += r
        s2 += r * r
    }
    s1 /= float64(n)
    s2 = s2/float64(n) - s1*s1
    fmt.Printf("%d чисел: среднее=%f, СК0=%f\n", n, s1, math.Sqrt(s2))
}

```

Любопытно видеть насколько близки два показанных примера! Сборка:

```

$ make
go build -o sko -compiler gc skoc.go
g++ -o skoc -lm skoc.cc

```

И выполнение, для последовательности из 100 миллионов случайных значений:

```

$ time ./sko 100000000
100000000 чисел: среднее=0.000017, СК0=0.999985
real    0m2,725s
user    0m2,728s
sys     0m0,004s

$ time ./skoc 100000000
100000000 чисел: среднее=-3.48074e-05, СК0=1.00006
real    0m13,505s
user    0m13,505s
sys     0m0,000s

```

Достаточно интересное соотношение времён выполнения для Go и C++ ... но это, скорее всего, никак не связано с вычислениями, а определяется используемыми генераторами случайных чисел и тем, с какой тщательностью они проводят генерацию.

С другой стороны, оцените как бы вы вычисляли дисперсию пользуясь классическим формульным вычислением «в лоб» как это делают математики, для последовательности в 100 миллионов вещественных значений, для хранения которых нужен объём памяти порядка 800Mb.

Порядок итераций для map: сюрприз

Ещё одна интересная особенность Go, которая обсуждается, как оказалось, в Интернет и активно и не первый год... (каталог types) — пример удивительно прост:

map4rand.go

```
package main

import "fmt"

func main() {
    m := map[int]int{
        1: 19, 2: 18, 3: 17,
        4: 16, 5: 15, 6: 14,
    }
    fmt.Printf("%v\n", m)
    for i := 0; i < 5; i++ {
        for key, value := range m {
            fmt.Printf("%1d:%2d ", key, value)
        }
        println()
    }
}
```

А вот его выполнение оказывается не таким простым, и может вызвать удивление (смотрим два последовательных запуска непосредственно один за другим):

```
$ ./map4rand
map[1:19 2:18 3:17 4:16 5:15 6:14]
3:17 4:16 5:15 6:14 1:19 2:18
1:19 2:18 3:17 4:16 5:15 6:14
3:17 4:16 5:15 6:14 1:19 2:18
4:16 5:15 6:14 1:19 2:18 3:17
1:19 2:18 3:17 4:16 5:15 6:14
```

```
$ ./map4rand
map[1:19 2:18 3:17 4:16 5:15 6:14]
6:14 1:19 2:18 3:17 4:16 5:15
5:15 6:14 1:19 2:18 3:17 4:16
4:16 5:15 6:14 1:19 2:18 3:17
4:16 5:15 6:14 1:19 2:18 3:17
1:19 2:18 3:17 4:16 5:15 6:14
```

Порядок перебора в итерациях map-ов (карт, отображений, таблиц, хешей, hashmap ... как только их не обзывают) меняется случайным образом не только от запуска к запуску, но и при повторных последовательных итерациях в уже запущенной программе)! (Точнее, меняется хаотически порядок выборки, а начальные элемент итератора range ... , но и это может смениться в любой новой версии).

Это не случайность, и не дефект реализации — это сделано разработчиками Go сознательно! См. [4,5] и в документации [6] в обоснование такого решения ... Коротко (в моём переводе) эта мотивация выглядит так:

Начиная с версии Go 1.0, среда выполнения использует рандомизированный порядок итерации карт. Программисты начали полагаться на стабильный порядок итераций ранних версий Go, который варьировался в зависимости от реализации, что приводило к ошибкам переносимости. Если вам требуется стабильный порядок итераций, вы должны поддерживать отдельную структуру данных, определяющую этот порядок.

Консольный калькулятор

Для проверки приоритетов операций в Go (о чём говорилось в первой части книги) ... ну, и для других некоторых целей — возникло намерение сделать консольный (CLI)

калькулятор: ты ему задаёшь арифметическое выражение (строку), а он тебе возвращает результат вычисления этого выражения. Только сделать это так, чтобы не выписывать вручную для каждой операции код — таких реализаций готовых по Интернет великое множество — на многие десятки, а то и сотни строк кода. А так чтобы найти в огромных, на сегодня уже, сетевых репозиториях GoLang и импортировать **пакет**, реализующий метод Eval() (по аналогии того как это делается в Lisp или Python). Так что эта глава будет не о том, как написать калькулятор, а о том, как в сетевых репозиториях GoLang искать необходимое. И делать всё это быстро... А калькулятор у нас выпадет как побочный продукт этого занятия.

Сказано — сделано! Поиском на ресурсах GoLan (<https://pkg.go.dev/>) ищем по контексту «math eval» (<https://pkg.go.dev/search?q=math+eval&m=package>) и находим несколько удовлетворяющих, например:

meval (github.com/gSpera/meval)

Package Math Eval implements a math evaluator.

Imported by 1 | v0.1.1 published on Nov 16, 2019 | MIT

Обычно даже крайне сухих комментариев к пакетам на GoLang достаточно, чтобы начать их полноценно использовать... Быстренько проверив пакет на показанных в нём Examples (что делать каждый раз нужно обязательно для проверки на пригодность своим целям!), набрасываю себе код (каталог types):

math.eval.go

```
package main
```

```
import (
    "bufio"
    "fmt"
    "github.com/gSpera/meval"
    "io"
    "os"
    "strings"
)

func main() {
    e := meval.New()
    e.SetFn("sum", sum)
    e.SetFn("prod", prod)
    r := bufio.NewReader(os.Stdin)
    for {
        print("? ")
        str, err := r.ReadString('\n')
        if err == io.EOF {
            println()    // ввод ^D - EOF
            break
        }
        if err != nil {
            fmt.Printf("%v\n", err)
            panic(err)
        }
        str = strings.TrimRight(str, " \n")
        if 0 == len(str) {    // ввод Enter – пустая строка
            break
        }
        print(str, " = ")
        value, err := e.Eval(str)
        if err == nil {
            fmt.Printf("%v\n", value)
        } else {
            fmt.Printf("ошибочное выражение: %v\n", err)
        }
    }
}
```



```

    }
}

var sum = func(args ...float64) (float64, error) {
    var sum float64
    for _, arg := range args {
        sum += arg
    }
    return sum, nil
}

var prod = func(args ...float64) (float64, error) {
    var prod float64 = 1.0
    for _, arg := range args {
        prod *= arg
    }
    return prod, nil
}

```

В этом коде гораздо больше, чем сформулировано в первоначальной постановке: там определены две новых собственных функции калькулятора: `sum()` и `prod()` (раз пакет позволяет лёгкими средствами делать такие вещи). А это значит, что демонстрируется возможность добавить туда обработку всего что только не вздумается...

Совершенно естественно, что это пока не собирается, поскольку требует импорта из сетевого репозитория:

```

$ go build math.eval.go
math.eval.go:6:5: no required module provides package github.com/gSpera/meval; to add
it:
    go get github.com/gSpera/meval

```

Как это и обсуждалось ранее, в первой части книги, и что легко решается:

```

$ go mod tidy
go: finding module for package github.com/gSpera/meval
go: downloading github.com/gSpera/meval v0.1.1
go: found github.com/gSpera/meval in github.com/gSpera/meval v0.1.1

```

И в итоге:

```

$ go build math.eval.go
$ ls -o math.eval*
-rwxrwxr-x 1 olej 2558703 фев 10 17:03 math.eval
-rw-rw-r-- 1 olej 1035 фев 10 16:44 math.eval.go

```

И вот что из этого получилось:

```

$ go run math.eval.go
? 2 + 3
2 + 3 = 5
? 2/3
2/3 = 0.6666666666666666
? (3+4)/(6-2)
(3+4)/(6-2) = 1.75
? pi
pi = 3.141592653589793
? log(11,15)
log(11,15) = 1.1293446514625658
? log(2,256)
log(2,256) = 8
? sin(0.5*pi)
sin(0.5*pi) = 1
? 2<<1
2<<1 = ошибочное выражение: cannot evaluate "2<<1": Unkown Token: token.Token(<<)

```

```
? sum(1, 2, 3, 4)
sum(1, 2, 3, 4) = 10
? prod(1,2,3)
prod(1,2,3) = 6
? prod(sin(pi/4) * log(2,8))
prod(sin(pi/4) * log(2,8)) = 2.1213203435596424
?
```

Но глава эта не от том хорош или плох получился калькулятор, а о технике **быстрого** нахождения по огромным сетевым репозиториям GoLang необходимых целевых пакетов, уже выполненных кем-то до вас. На всю разработку этого калькулятора, считая от начала поиска по репозиториям, ушло не больше 3-4 часов. А это уже напоминает технику использования Python, который заслуженно называют «язык быстрой реализации проектов».

Источники информации

- [1] Go в примерах — <https://gobyexample.com.ru/>
- [2] Pi with Machin's formula (Python) — https://literateprograms.org/pi_with_machin_s_formula_python_.html
- [3] Подборка проектов для разработки GUI на Go, 6 апреля 2022 — <https://dzen.ru/media/golang/podborka-proektov-dlia-razrabotki-gui-na-go-624d38f4828df173e1426ead>
- [4] Hashmap (map) по версии Golang вместе с реализацией на дженериках — <https://habr.com/ru/post/704796/>
- [5] Why are iterations over maps random? — <https://stackoverflow.com/questions/55925822/why-are-iterations-over-maps-random>
- [6] Iterating in maps — <https://go.dev/doc/go1#iteration>

Структуры данных, типы и их методы

Более-менее формально сами типы данных были уже рассмотрены при рассмотрении синтаксиса Go (так сказать «прямолинейное перечисление»). Здесь же мы вернёмся к рассмотрению **на примерах** тех тонких особенностей, которые могут не очевидным образом вытекать из этих определений.

Массивы и срезы

Массивы и срезы представляют несколько необычные для программиста на C/C++ конструкции в языке Go. Поэтому вспомним (каталог `compare/valadr` архива), для начала, что из себя представляют массивы C (да и C++ тоже):

array.c.c

```
#include <stdio.h>

void ptrans(int p[], int size) {          // для указателя массива
    int i;
    for(i = 0; i < size; i++) p[i]++;
}

int main(int argc, char **argv) {
    int a1[] = {1, 0, 2, 0, 3, 0, 4},
        size = sizeof(a1) / sizeof(a1[0]);
    void show(int p[]) { // для массива
        int i;
        printf("[ %d ]: ", size);
        for(i = 0; i < size; i++) printf("%d ", p[i]);
        printf("\n");
    }
    show(a1);
    ptrans(a1, size);
    show(a1);
    return 0;
}
```

Массив `a1` описан в функции (в данном случае в `main()`, но это не важно), и в программной единице, где непосредственно находится его определение, он видится как **массив**, и для него может быть вычислен размер как:

```
size = sizeof(a1) / sizeof(a1[0]);
```

Но при передаче в качестве параметра вызова любой функции (и всей последующей, возможно, передаче по цепочке вызовов) массив передаётся по его адресу, как **указатель** первого элемента массива, и информация собственно о массиве теряется. Любые изменения параметра, переданного по адресу, сделанные внутри вызванной функции, отображаются и в вызывающей единице (побочный эффект):

```
$ ./array_c
[ 7 ]: 1 0 2 0 3 0 4
[ 7 ]: 2 1 3 1 4 1 5
```

Примечание: Пример сознательно выписан так, чтобы он был максимально похож на свой эквивалент на языке Go, который показан далее. В примере использовано такое **расширения компилятора GCC** (но не допускаемой поздними стандартами C89 и C99) как вложенное (в `main()`) описание функция `show()`. В GCC C++ такое расширение не допускается.

Теперь рассмотрим аналогичную ситуацию в языке Go:

array.go.go

```
package main
```

```

type arr [7]int                                // тип массива

func atrans(v arr) arr {                      // для массива
    for i, x := range v {
        v[i] = x + 1
    }
    return v
}

func ptrans(p *arr) {                         // для указателя массива
    for i, x := range *p {
        (*p)[i] = x + 1
    }
}

func strans(v []int) []int {                 // для среза
    for i, x := range v {
        v[i] = x + 1
    }
    return v
}

func main() {
    show := func(p arr) {                    // для массива
        print("[ ", len(p), " ]: ")
        for _, y := range p {
            print(y, " ")
        }
        print("\n")
    }
    shows := func(p []int) { // для среза
        print("[ ", len(p), " ]: ")
        for _, x := range p {
            print(x, " ")
        }
        print("\n")
    }
    a1 := arr{0: 1, 2: 2, 4: 3, 6: 4}
    show(a1)
    a2 := *new(arr)
    a2 = a1                                // присвоение массива
    show(a2)
    b1 := atrans(a1)                       // возврат массива значением
    show(a1)                               // массив передаётся по значению!
    show(b1)
    ptrans(&a1)                             // передача массива адресом!
    show(a1)
    a3 := a2[0:len(a2)]                    // срез образованный из массива
    shows(a3)
    strans(a3)                             // срез передаётся по адресу
    shows(a3)
}

```

Переменные a1, a2, b1 — это **массивы** Go. Они имеют тип [7]int (размерность 7 — составная часть типа массива! — [7]int и [8]int это разные и **несовместимые** типы). Массивы передаются в функции show() (описана как вложенная **функциональная переменная**) и atrans() **по значению**, копированием. Поэтому никакие изменения в переданном массиве, производимые в функции atrans(), **не отражаются** позже в вызывающей функцию единице. Более того, массив может так же копированием присваиваться (a2 = a1) и возвращаться копированием как результат выполнения функции atrans() (b1 := atrans(a1)). Если нам нужно иметь побочный эффект

изменений переданного массива функцией, следует передавать в функцию (`ptrans()`) **адрес** массива.

Но **срез** `a3`, образованный над массивом `a2`, передаются **по ссылке**, поэтому изменения, произведенные функцией `strans()` видны позже в вызвавшей единице.

```
$ ./array_go
[ 7 ]: 1 0 2 0 3 0 4
[ 7 ]: 1 0 2 0 3 0 4
[ 7 ]: 1 0 2 0 3 0 4
[ 7 ]: 2 1 3 1 4 1 5
[ 7 ]: 2 1 3 1 4 1 5
[ 7 ]: 1 0 2 0 3 0 4
[ 7 ]: 2 1 3 1 4 1 5
```

Мы не сможем вызвать функции `show()`, `atrans()` и `ptrans()`, ожидающие параметром **массив** типа `[7]int`, для **среза**, имеющего тип `[]int`. Симметрично, так же невозможно вызвать и функции `shows()` и `strans()`, ожидающие параметр `[]int`, для массивов `[7]int`. Это следствие жёсткой типизации Go.

Отсюда можно наблюдать, что именно срезы в Go ведут себя во многом похоже на массивы C и C++, и именно срезы наиболее употребимы в практике Go. Отличие их (от C/C++) состоит в том, что за ними нет необходимости «тянуть» дополнительным параметром их длину — длина всегда доступна вызовом `len()` в вызываемой единице.

Ещё одним результатом сравнения может быть то, что срезы Go, являющиеся **наложением** на массивы, можно легко изменять в размерах (в пределах `len()` базового массива!) без накладных расходов перерасмещения в памяти (типа `realloc()` в C). В чём-то, и в ограниченных пределах, это напоминает динамические типы STL в C++, например, `vector<int>`.

Для того, чтобы более наглядно представить что из себя представляют массивы Go, реализуем **аналогичный тип данных** в C:

garray.c.c

```
#include <stdio.h>

#define size 7
typedef struct {
    int data[size];
} garrey_t;

void show(garrey_t a) {
    int i;
    printf("[ %d ]: ", size);
    for(i = 0; i < sizeof(a) / sizeof(*a.data); i++)
        printf("%d ", a.data[i]);
    printf("\n");
}

void ptrans(garrey_t* p) {          // для указателя массива
    int i;
    for(i = 0; i < size; i++) p->data[i]++;
    show(*p);
}

void atrans(garrey_t a) {          // для массива по значению
    int i;
    for(i = 0; i < sizeof(a) / sizeof(*a.data); i++)
        a.data[i]++;
    show(a);
}
```

```

int main(int argc, char **argv) {
    garrey_t a1 = {{1, 0, 2, 0, 3, 0, 4}};
    show(a1);
    atrans(a1);
    show(a1);
    ptrans(&a1);
    show(a1);
    return 0;
}

```

Это (переменные типа `garrey_t`) также будут массивы фиксированного размера, которые передаются в качестве параметров в функции **по значению** (3-я строка вывода):

```

$ ./garray_c
[ 7 ]: 1 0 2 0 3 0 4
[ 7 ]: 2 1 3 1 4 1 5
[ 7 ]: 1 0 2 0 3 0 4
[ 7 ]: 2 0 3 0 4 0 5
[ 7 ]: 2 0 3 0 4 0 5

```

В завершение рассмотрения массивов C, C++ и Go, и передачи параметров вызова в функцию, вернёмся к уже высказанному ранее утверждению, что **во всех** этих языках этой группы параметры **любых типов** (простых и агрегатных) передаются **только по значению**, то есть **копированием** переданного значения. И для полной ясности незначительно модифицируем уже показанный выше пример `array_c.c`:

cpptr.c

```

#include <stdio.h>

void ptrans(int *p, int size) {           // для указателя массива
    printf("%p ... ", p);
    while(size-- > 0) (*p++)++;
    printf("%p ... \n", p);
}

int main(int argc, char **argv) {
    int a1[] = {1, 0, 2, 0, 3, 0, 4},
        size = sizeof(a1) / sizeof(a1[0]),
        *pa1 = &a1[0];
    void show(int *p) {
        int i;
        printf("[ %d ]: ", size);
        for(i = 0; i < size; i++) printf("%d ", p[i]);
        printf("\n");
    }
    printf("%p ... \n", pa1);
    show(a1);
    ptrans(a1, size);
    show(a1);
    return 0;
}

$ ./cpptr
0x7fffd1623ab0 ...
[ 7 ]: 1 0 2 0 3 0 4
0x7fffd1623ab0 ... 0x7fffd1623acc ...
[ 7 ]: 2 1 3 1 4 1 5

```

Здесь в вызванной функции `ptrans()` модифицируется как указатель начала переданного массива, так и его длина. Но вызвавшая функция `main()` после вызова благополучно продолжает работать с не испорченным массивом. Это происходит потому, что в `ptrans()` передавались **копия** указателя массива (в 3-й строке вывода видно как

эта копия меняется) и **копия** длины массива.

Поэтому в корне неверно говорить, что **массивы** C/C++ и **срезы** Go передаются при вызове по ссылке (адресу). Просто по правилам C/C++, из соображений **эффективности**, при передаче массива в качестве параметра вызова функции **вместо** него передаётся указатель начала массива, и этот указатель передаётся **по значению**.

Многомерные срезы и массивы

Массивы и срезы Go, так же как практически во всех других языках программирования, могут быть многомерными. Чаще всего на практике используются 2-мерные массивы (матрицы), но могут быть и большей размерности. Элементы многомерного массива индексируются последовательностью индексов по измерениям, например: `A[i][j]`. Здесь пока прямые аналогии с другими языками программирования.

А теперь перейдём к особенностям... Многомерные **срезы** (но не **массивы**) могут быть не прямоугольными! Например (каталог `types`), треугольная матрица (достаточно широко применяется в математических расчётах):

3slice.go

```
package main
import ("fmt")

func main() {
    const dim = 5
    twoD := make([][]int, dim)
    for i := 0; i < dim; i++ {
        innerLen := i + 1
        twoD[i] = make([]int, innerLen)
        for j := 0; j < innerLen; j++ {
            twoD[i][j] = i + j
        }
    }
    fmt.Println("2D:", twoD)
    fmt.Printf("2D: %v\n", twoD)
}
```

Обращаем внимание как функции вывода пакета `fmt` форматируют (пытаются представить) при выводе данные сложно структурируемых типов: `Println()` — массивы, а `Printf()` — данные в предопределённом для них формате (`%v`).

```
$ ./3slice
```

```
2D: [[0] [1 2] [2 3 4] [3 4 5 6] [4 5 6 7 8]]
```

```
2D: [[0] [1 2] [2 3 4] [3 4 5 6] [4 5 6 7 8]]
```

Это возможность происходит из того, что массивы и срезы **реализуют** интерфейс `Stringer`, определяя метод `String()` в своих реализациях. (О чём самому интерфейсу `Stringer` вовсе не обязательно «знать» — он только описывает как должен выглядеть прототип метода `String()`) А все эти функции пакета `fmt`, для **любого** типа, реализующего интерфейс `Stringer`, должны только вызывать метод `String()`. Это подробно описывалось ранее, и составляет фундамент объектно-ориентированной техники в Go.

Функции с множественным возвратом

Функции Go могут возвращать не одно значение, и не несколько значений запакованных в единую составную структуру (как в некоторых более старых языках, C/C++), а сколь угодно много различных значений, в возвращающем `return` эти значения просто перечисляются через запятую (это напоминает Python). Такие функции они называют: функции с множественным возвратом. Самый частый (но не самый хитроумный в

использовании) случай их использования вы будете встречать в Go **повсеместно**, когда функция своим 2-м возвращаемым значением возвращает код завершения (ошибки). Основной способ **реакции на ошибочные ситуации** в Go.

Но область использования такого способа может быть много шире, и определяется только вашей фантазией... Небольшой, но показательный пример (в каталоге `function`):

marg.go

```
package main
import "fmt"

func vals(n int) (int, float64, []string) {
    i := n
    f := float64(n) / 3
    s := []string{"g", "h", "i"}
    return i, f, s
}

func main() {
    i, f, s := vals(5)
    fmt.Println(i, f, s)
    _, f, _ = vals(7)
    fmt.Printf("%.3f\n", f)
}

$ ./marg
5 1.6666666666666667 [g h i]
2.333
```

Показательный тем что:

- Прежде всего, конечно, иллюстрацией множественного возврата: число возвращаемых значений может быть сколь угодно, типы возвращаемых значений могут **различаться**, типы могут быть произвольны и любых структурных типов; число и типы функции перечисляются списком в скобках.
- Функция, в данном случае, возвращает (специально) значения **локальных** переменных, описанных в области видимости внутри тела функции. В C/C++ это абсолютно **недопустимо**, но компилировалось бы без ошибок, а затем порождало бы крайне трудно локализуемые («блуждающие») ошибки. Но Go — это язык с динамической сборкой мусора, и локально порождённые переменные `i`, `f`, `s` имеют право быть уничтоженными только тогда, когда исчезнет последняя ссылка на переменные (в вызывающей функции `main()`).
- Если какие-то из значений, возвращаемые функцией с множественным возвратом, вам не нужны, то в качестве целей их присвоения следует указать неиспользуемую переменную с именем `_`. Бывает, что и все возвращаемые значения могут оказаться не нужными, функция может вызываться только для побочных эффектов вызова.

У Go очень вариативный синтаксис, отдельные конструкции языка ортогональные (независимо альтернативные). Имеет смысл для сравнения записать совсем другой вариант записи только-что показанной функции:

marg2.go

```
...
func vals(n int) (i int, f float64, s []string) {
    i = n
    f = float64(n) / 3
    s = []string{"g", "h", "i"}
    return;
}
...
```


Здесь возвращаемые значения (**из** функции) **именованы**, подобно передаваемым параметрам вызова (**в** функцию). Обращаем внимание на обязательную смену := на =, в первом варианте локальные переменные **объявляются** (с попутной инициализацией), а во втором варианте уже объявленным (и именованным) переменным **присваиваются** значения. В return в этом варианте перечислять значения возврата не надо, они уже перечислены в заголовке функции. Иногда, в длинных функциях, такой вариант оказывается более удобным.

Строки, руны и UNICODE

Обработка символьной, текстовой информации — это стало на сегодня настолько важной частью любого языка программирования, что она заслуживает отдельного рассмотрения в примерах. Мы рассмотрим различные возможности обработки мультязычных строк: не латинских, русскоязычных для конкретности, но это может быть и иероглифический китайский, и иврит, с записью слева-направо, и справа-налево... То, что в традиционных языках (C и C++) делается совсем не так просто.

Символы, байты и руны

Строки `string` (зачастую) состоят из последовательности многобайтных (1, 2, 3, 4 ... потенциально до 6 байт) символов UTF-8, представляющих кодовых точек UNICODE (int32 значения, `rune`). Структура `string` в Go позволяет лучше представить такое простое приложение (каталог `strings/runes`):

`rune_len.go`

```
package main

import "fmt"

func main() {
    английский := "Hello"
    русский := "Привет"
    японский := "こんにちは"
    show := func(s string) {
        fmt.Print(s, " : байт ", len(s),
                  " символов ", len([]rune(s)), " : ")
        for i := 0; i < len(s); i++ {
            fmt.Printf(" %02X", s[i])
        }
        fmt.Print(" [")
        for p, c := range s {
            fmt.Printf(" %d:%q", p, c)
        }
        println("]")
    }
    show(английский)
    show(русский)
    show(японский)
}
```

И видим:

```
$ ./rune_len
Hello : байт 5 символов 5 :  48 65 6C 6C 6F [ 0:'H' 1:'e' 2:'l' 3:'l' 4:'o']
Привет : байт 12 символов 6 :  D0 9F D1 80 D0 B8 D0 B2 D0 B5 D1 82 [ 0:'П' 2:'р' 4:'и'
6:'в' 8:'е' 10:'т']
こんにちは : байт 18 символов 6 :  E2 8C 98 E3 81 93 E3 82 93 E3 81 AB E3 81 A1 E3 81 AF
[ 0:'こ' 3:'ん' 6:'ん' 9:'に' 12:'ち' 15:'は']
```

В итоге:

- итерация по позиции и встроенная функция `len()` оперируют с отдельными составляющими байтами UTF-8 символов...
- для того, чтобы иметь дело с последовательными символами, тип `string` нужно преобразовывать к срезу `[]rune`
- функция `len()` для типа `string` возвращает число байт, эта же функция для преобразованного среза `[]rune` возвращает число UTF-8 символов в исходной строке;
- итерация, использующая `range`, ведёт нас по многобайтным символам UTF-8: по

порядковым индексам начальных байт каждого символа и позволяет получить rune каждого символа;

Нужно хорошо понимать, что массив (срез) rune не является каким-то «ещё одним» представлением строки — это числовой int32 массив (2-я строка вывода), и для его вывода (терминал, печать, графическое окно вывода...), или любого другого использования в качестве строки, срез rune должен преобразовываться к типу string:

rune_out.go

```
package main

import "fmt"
import "unicode/utf8"

func main() {
    j := "世界" // "мир", яп.
    fmt.Println("Hello" + j, ": байт", len(j),
               ", символов", utf8.RuneCountInString(j))
    r := []rune(j)
    fmt.Printf("%X [%d]\n", r, len(r))
    fmt.Printf("%q [%d]\n", r, len(r))
    fmt.Println(string(r))
}

$ ./rune_out
Hello 世界 : байт 6 , символов 2
[4E16 754C] [2]
['世' '界'] [2]
世界
```

(Хотя срез []rune и является срезом именно целочисленных значений int32, для его форматированного вывода в качестве среза отображаемых символов предоставлен специальный формат %q).

Изменение содержимого строк

Неоднократно повторялось, и это специально акцентируется в документации Go, что переменные типа string — **не изменяемые**! Но не изменяемость эта выражается в том, что нельзя взять в строке отдельный символ и поменять его на что-то другое. Но содержимое переменных string можно сколь угодно изменять дозволенными функциями и методами, которые при этом берут такую переменную в одном месте памяти и, после её модификации, размещают новую модифицированную копию в совершенно другом месте памяти. В этом смысле строки Go совершенно подобны строкам Python, для которых точно таким же образом декларируется неизменяемость.

Следующее простое приложение демонстрирует изменение любого цифрового символа (0 ... 9) его инкрементом «по кругу» (9 превращается в 0) ... приложение бессмысленное по содержанию, но очень хорошо иллюстрирует общую схему изменения содержимого строки:

map.go

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "strings"
    "unicode"
)

func main() {
```

```

scanner := bufio.NewScanner(os.Stdin)
for scanner.Scan() {
    инкремент := func(r rune) rune {
        if unicode.IsNumber(r) {
            return '0' + (r-'0'+1)%10
        } else {
            return r
        }
    }
    fmt.Println(strings.Map(инкремент, scanner.Text()))
}
}

```

Первая строка здесь в каждой паре — это исходная строка вводимая нами с терминала, а следующая — строка изменённая кодом:

```

$ ./map
12 + 13 = 25
23 + 24 = 36
В 1913 году 1 пуд зерна стоил 34 руб.
В 2024 году 2 пуд зерна стоил 45 руб.

```

В пакете `strings` очень много методов подобных `strings.Map()` — на все потребности и вкусы! Часть из них выполняют фиксированные действия, по типу `strings.TrimLeft()` или `strings.TrimRight()`, их смысл ясен из названия. Но много и таких, которые включают в свой прототип функцию преобразования, или предикат фильтрующий символы подлежащие преобразованию. Для `strings.Map()`, которую мы только-что видели, прототип записан так:

```
func Map(mapping func(rune) rune, s string) string
```

Важно: прототипы функций-операторов в таких вызовах описываются как `func(rune) rune` или как `func(rune) boolean` (это когда логический предикаты отбора для операций) — в операцию вовлекается одиночный символ UNICODE. (Очень часто, в силу специфики синтаксиса Go, такие функции-параметры определяются как инлайн лямбда-функции (анонимные), определяемые непосредственно в месте вызова.) Таким образом нельзя заменить, например, один символ на 2, или на пустую строку (удалить символ), но вполне возможно 1-байтовый UTF-8 сменить на 4-байтовый. Но подобные методы вполне могут и изменять число символов в строке, например, функции тримингования:

```
func TrimLeftFunc(s string, f func(rune) bool) string
func TrimRightFunc(s string, f func(rune) bool) string

```

Палиндромы

Палиндромы — это слова или целые фразы, которые читаются одинаково в обоих направлениях: слева направо и справа налево. Самым длинным в мире палиндромом, состоящим из одного слова, принято считать финское слово `säpöruukivakivi` (торговец щелоком — мыльным камнем). В английском языке самый длинный палиндром считают — `redivider` (своего рода перегорода).

В русском языке довольно много палиндромов. Нахождение их было модной забавой в литературных кругах XIX-XX веков, но популярной и на сейчас. Но это и отличная возможность практики использования строчных функций Go, особенно если применительно не к ASCII (латинским) наборам символов.

Первой, более простой, задачей мы сделаем тестер слов-палиндромов. Это может выглядеть так (всё в каталоге примеров `strings/palindrom`):

palindrow.go

```

package main
import (
    "fmt"
    "bufio"

```

```

        "log"
        "os"
    )

func test(scanner *bufio.Scanner) {
    for scanner.Scan() {
        ustr := scanner.Text() // введенная строка
        rstr := []rune(ustr)
        pali := true;
        for i := 0; i < len(rstr) / 2; i++ {
            pali = (rstr[i] == rstr[len(rstr) - i - 1])
            if !pali {
                break
            }
        }
        fmt.Printf("%v : %s\n", pali, ustr)
    }
}

func main() {
    if len(os.Args) > 1 {
        file, err := os.Open(os.Args[1])
        if err != nil {
            log.Fatal(err)
        }
        defer file.Close()
        test(bufio.NewScanner(file))
    } else {
        test(bufio.NewScanner(os.Stdin))
    }
}

```

В главной функции `main()` только определяется источник исходных строк: либо терминал (если в командной строке не указан параметр), либо файл со строками (если параметром указано имя файла). Проверка строк осуществляется в функции `test()`, которая тут же и выводит вердикт, является ли строка палиндромом: `true` или `false`.

С не-ASCII текстами мы не имеем права перебирать символы в `string` — это приведёт к перебору байт в строке, а не многобайтных символов UTF-8 ... так же как и функция `len()` возвращает длину строки не в символах, а в байтах. Для работы с символами мы должны перейти от `string` к срезам из `rune`!

Консольный режим:

```

$ ./palindrow
123454321
true : 123454321
123321
true : 123321
123322
false : 123322
радар
true : радар
дерево
false : дерево

```

Или при чтении из файла (файлы тестовых строк находятся там же в каталоге):

```

$ ./palindrow words.txt
true : доход
true : шалаш
true : топот
true : радар
true : комок

```

```
true : saippuakivikauppias
true : redivider
```

А вот что происходит, если имя файла указать неправильно:

```
$ ./palindrow worde.txt
2024/01/17 00:04:09 open worde.txt: no such file or directory
```

Для простейшего случая, для палиндромов-слов это работает достаточно прилично. Но если мы перейдём к фразам, словам разделённым пробелами, тут нас будут ожидать неприятности ... симитируем слова числовыми значениями:

```
$ ./palindrow
123 321
true : 123 321
12 33 21
true : 12 33 21
1 23 321
false : 1 23 321
```

По правилам (достаточно формализованным) составления палиндромов-слов:

- пробелы и знаки препинания в фразе не принимаются во внимание;
- большие и малые литеры не различаются;
- буквы вот в этих парах считаются идентичными: е-ё, и-й, ь-ъ;

Для начала мы исключим пробелы и различия в строчных и прописных буквах ... и это нам даст возможность попрактиковаться в изменениях содержимого `string` (нас ведь не интересуют сами палиндромы, а интересуют техники символьной обработки):

palindrob.go

```
package main
import (
    "fmt"
    "bufio"
    "log"
    "os"
    "strings"
)

func test(scanner *bufio.Scanner) {
    for scanner.Scan() {
        ustr := scanner.Text() // введенная строка
        rstr := []rune(strings.ReplaceAll(strings.ToLower(ustr), " ", ""))
        pali := true;
        for i := 0; i < len(rstr) / 2; i++ {
            if pali = rstr[i] == rstr[len(rstr) - i - 1]; !pali {
                break
            }
        }
        fmt.Printf("%v : %s\n", pali, ustr)
    }
}

...
```

Вызывающая функция `main()` остаётся неизменной, поэтому смотрим только тестирующую функцию и список импорта, который поменялся.

Мы использовали метод `strings.ToLower()` — превратить все символы в малые, и `strings.ReplaceAll()` — заменить все найденные подстроки в строке, в данном, случае заменить пробелы на пустую строку, но в общем случае — на что угодно. При этом длина строки, естественно, изменится! Это очень показательно ... это ещё одно разъяснение смысла неизменяемости строк Go.

Ещё здесь дополнительно показана инициализация значений непосредственно в теле условия оператора `if` перед их использованием, после этого эти значения (или даже вновь объявленные переменные) могут использоваться в теле оператора `{...}` или ветке `else {...}` (но здесь это не понадобилось). Это ещё одна, непривычная но приятная, особенность условного оператора `Go`.

Ручное выполнение:

```
$ ./palindrob
saippuakivikauppias
true : saippuakivikauppias
Лев осовел
true : Лев осовел
А роза упала на лапу Азора
true : А роза упала на лапу Азора
```

Здесь всё хорошо, но если мы перейдём к заготовленному файлу фраз, то там не всё так хорошо:

```
$ ./palindrob palindrom.txt
true : А роза упала на лапу Азора
true : Я иду с мечем судия
true : На в лоб, болван
true : Лев осовел
false : Да, гневен гад
true : Мат и тут и там
false : Лев с ума ламу свёл
true : Кирилл лирик
true : Уж редко рукою окурок держу
true : Коту скоро сорок суток
false : А муза рада музе без ума да разума.
true : Веер веял для евреев
false : Madam, I'm Adam
false : Муха! О, муха! Велика аки лев! Ах, ум! О ах, ум!
true : Sum summus mus
false : Νίψον ανομήματα μη μόναν όψιν
true : Sator Arepo tenet opera rotas
false : Уверена я, а не реву
```

Здесь нас сбивают знаки препинания и, например, буква ё ... И последний вариант, и ещё один импортированный пакет возможностей: `regex` — регулярные выражения!

palindrob.go

```
package main
import (
    "fmt"
    "bufio"
    "log"
    "os"
    "strings"
    "regexp"
)

var nonAlphaNum = regexp.MustCompile(`^[a-zA-я0-9 ]+`)

func test(scanner *bufio.Scanner) {
    for scanner.Scan() {
        ustr := scanner.Text() // введенная строка
        s1 := strings.ReplaceAll(strings.ToLower(ustr), " ", "")
        s1 = strings.ReplaceAll(s1, "ё", "е")
        s1 = strings.ReplaceAll(s1, "й", "и")
        rstr := []rune(nonAlphaNum.ReplaceAllString(s1, ""))
        pali := true;
```

```

        for i := 0; i < len(rstr) / 2; i++ {
            if pali = rstr[i] == rstr[len(rstr) - i - 1]; !pali {
                break
            }
        }
        fmt.Printf("%v : %s\n", pali, ustr)
    }
}
...

```

Методы замены `strings.ReplaceAll()` мы уже видели раньше ... а вот пакет `regexp` стандартной библиотеки Go покрывает все возможности регулярных выражений Perl, `egrep`, Python, или (альтернативно) в синтаксисе POSIX. Метод `regexp.MustCompile()` компилирует шаблон регулярного выражения, а метод `ReplaceAllString()`, применением этого скомпилированного шаблона, замещает удовлетворяющие строки (в данном случае удаляет — замещая на пустую строку).

В итоге:

```

$ ./palindrom palindrom.txt
true : А роза упала на лапу Азора
true : Я иду с мечем судия
true : На в лоб, болван
true : Лев осовел
true : Да, гневен гад
true : Мат и тут и там
true : Лев с ума ламу свёл
true : Кирилл лирик
true : Уж редко рукою окурок держу
true : Коту скоро сорок суток
true : А муза рада музе без ума да разума.
true : Веер веял для евреев
true : Madam, I'm Adam
true : Муха! О, муха! Велика аки лев! Ах, ум! О ах, ум!
true : Sum summus mus
true : Νίψον ανομήματα μη μόναν όψιν
true : Sator Arepo tenet opera rotas
true : Уверена я, а не реву

```

Мы на этих 3-х примерах затронули, мельком, все основные техники работы с символьными строками. Любопытное наблюдение: если для разбора с содержимым отдельных UTF-8 символов в строке нам требуются типы `rune`, то библиотечные функции и методы, согласованно меняющие содержимое строк, работают непосредственно с самими типами `string` без требования ручных преобразований.

Регулярные выражения

Регулярные выражения — это отдельно стоящая, очень мощная техника работы с текстовой информацией. Регулярные выражения являются способом описания текстовых шаблонов (образцов) для сопоставления и выполнения последующих действий по итогам этих сопоставления. Регулярные выражения являются фундаментальной базой таких инструментов GNU как `grep`, `egrep`, `sed`, `awk` и редакторов `ed`, `vi`, `vim`, `Emacs`. В некотором смысле, регулярные выражения являются альтернативой функциональным или объектным API строчной обработки в разных языках программирования. Некоторые авторы считают эту технику излишне сложной, с высоким порогом вхождения в тему, но все соглашаются с её высокой мощностью.

В Go поддержка регулярных выражений поддерживается пакетом стандартной библиотеки `regexp`. В синтаксисе регулярных выражений есть несколько вариантов, различий (диалектов), синтаксис `grep` несколько отличается от `egrep`, поэтому говорят о синтаксисе Perl, Python или POSIX. Синтаксис принятых в Go регулярных выражений — это тот же общий синтаксис, который используется в Perl, Python ...

строго, это синтаксис, принятый RE2. Но может использоваться и синтаксис POSIX, если для подготовки (компиляции) регулярных выражений использовать `regexp.CompilePOSIX()` вместо `regexp.Compile()` (или `regexp.MustCompilePOSIX()` вместо `regexp.MustCompile()`, соответственно).

Техника применения регулярных выражений, хотя знание это и не имеет принципиального значения для применителя, состоит в том, что символьная строка регулярного выражения предварительно компилируется в небольшой фрагмент внутреннего кода, который потом применяется к анализируемым текстам (возможно многократно).

Синтаксис записи образцов (шаблонов) сопоставления содержит много спецсимволов, поэтому для записи строк шаблона документация Go рекомендует использовать не обычную запись (в двойных кавычках), а сырой (raw) формат, заключаемый в обратные одиночные кавычки (где спецсимволы никак не интерпретируются).

Вот теперь у нас всё готово чтобы приступить к коду (каталог `strings/regexp`) ... и первый пример будет выделять нам числовые значения (в примере использовано применение к строкам, но регулярные выражения могут, и часто, применяться к большим фрагментам обрабатываемого текста целиком).

finddig.go

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "regexp"
)

func main() {
    re, _ := regexp.Compile(`\d+`)
    scanner := bufio.NewScanner(os.Stdin)
    for scanner.Scan() {
        s := scanner.Text() // A123 AA455 AAA2 A89
        r := re.FindAllString(s, -1)
        fmt.Println(r)      // [123 455 2 89]
        a := re.FindAllStringIndex(s, -1)
        fmt.Println(a)      // [[1 4] [8 11] [15 16] [18 20]]
        m := make(map[int]string)
        for _, e := range a {
            m[int(e[0])] = s[e[0]:e[1]]
        }
        fmt.Println(m)      // [1:123 8:455 15:2 18:89]
    }
}
```

Особо меня интересовала здесь возможность выделения числовых (ASCII) полей внутри мультитайтных UTF-8 строк UNICODE, отличных от латиницы:

\$ go run finddig.go

A123 AA455 AAA2 A89

[123 455 2 89]

[[1 4] [8 11] [15 16] [18 20]]

map[1:123 8:455 15:2 18:89]

13 + 987 **Ɱ** + 5£ - 543μ / 4щ

[13 987 5 543 4]

[[0 2] [9 12] [18 19] [24 27] [32 33]]

map[0:13 9:987 18:5 24:543 32:4]

При Петре I средний заработок неквалифицированного работника составлял 5-8 копеек в день, 1 пуд мяса тогда стоил 30 копеек, 1 пуд хлеба - 10 копеек.

[5 8 1 30 1 10]

[[133 134] [135 136] [163 164] [203 205] [220 221] [242 244]]

```
map[133:5 135:8 163:1 203:30 220:1 242:10]
в одном километре 1000 метров, или 100000 сантиметров, или 1000000 миллиметров
[1000 100000 1000000]
[[33 37] [59 65] [97 104]]
map[33:1000 59:100000 97:1000000]
```

Это же пример (при объёмном тестировании) можно запускать из файла предварительно заготовленных тестовых строк (файл `dig.txt` в архиве). Так:

```
$ go run finddig.go < dig.txt
...
```

Или так:

```
$ cat dig.txt | go run finddig.go
...
```

Следующий пример чуть сложнее, и показывает не только поиск по шаблону, но и разнообразные трансформации исходной строки:

replw.go

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "regexp"
    "strings"
)

func main() {
    // re, _ := regexp.Compile(`\w+`)
    // re, _ := regexp.Compile(`([Ё-Яа-ё]|[a-zA-Z])+`)
    // re, _ := regexp.Compile(`\S+`)
    re, _ := regexp.Compile(`\pL+`)
    scanner := bufio.NewScanner(os.Stdin)
    for scanner.Scan() {
        s := scanner.Text()
        r1 := re.FindAllString(s, -1)
        fmt.Printf("[%d] => ", len(r1))
        for _, c := range r1 {
            fmt.Printf("<%s> ", string(c))
        }
        fmt.Printf("\nfunc: ")
        r2 := re.ReplaceAllStringFunc(s, rep2up)
        println()
        fmt.Printf("[%d] => %v\n", len(r2), string(r2))
        println("-----")
    }
}

func rep2up(s string) string {
    fmt.Printf("<%s> ", s)
    return strings.ToLower(s)
}
```

Здесь упор сделан на операции над мультиязычными UTF-8 строками UNICODE. Здесь шаблон выбран так, чтобы любую строку-фразу разбить на отдельные слова. Но часто упоминаемый в публикациях шаблон ``\w+`` здесь не работает ... то есть он работает, но только на ASCII (англоязычных) строках. Остальные 3 показанных формы будут работать для разных языков. Здесь некоторые комментарии...

Если мы хотим определить диапазон символов конкретного языка, то прежде должны пересмотреть кодовые таблицы UNICODE этого языка, потому что символы не всегда следуют национальным алфавитам. Для русского языка, например, большая буква `Ё` (U+0401) предшествует `А` (U+0410), а малая `ё` (U+0451) следует за `я` (U+04FF), поэтому шаблон будет: `([Ё-Яа-ё][a-zA-Z])+`.

Метасимвол в шаблоне `S+` соответствует «любой не пробельный символ». А вот шаблон из числа новых `rL+` означает любой символ UNICODE. Поскольку это всё может заметно отличаться от традиционного синтаксиса регулярных выражений описываемого в литературе, настоятельно рекомендуется изучить краткое руководство по синтаксису для Go, датированное 2023 годом, и ссылка на которое даётся в конце текста.

Но вернёмся к коду... Результатом `re.FindAllString()` будет массив слов составляющих исходную фразу — для убедительности выводится размер этого массива. Далее мы для каждого слова в тестируемой строке (тот же шаблон) делаем замену `re.ReplaceAllStringFunc()`, но не просто фиксированную замену, а для каждого найденного слова вызываем нами же определённую собственную функцию вида `func rep2up (s string) string`, которая, вообще то говоря, может делать со строкой всё что угодно! В нашем случае она довольно тривиально преобразует слово к малым буквам.

И вот как это выглядит (для контроля, функция замены `rep2up` многократно выводит значения получаемых ею параметров на обработку):

```
$ ./replw
Ёлки пойдут на палки
[4] => <Ёлки> <пойдут> <на> <палки>
func: <Ёлки> <пойдут> <на> <палки>
[37] => ёлки пойдут на палки
-----
世界
[2] => <世> <界>
func: <世> <界>
[7] => 世界
-----
В начале было Слово, и Слово было у Бога, и Слово было Бог
[13] => <В> <начале> <было> <Слово> <и> <Слово> <было> <у> <Бога> <и> <Слово> <было>
<Бог>
func: <В> <начале> <было> <Слово> <и> <Слово> <было> <у> <Бога> <и> <Слово> <было>
<Бог>
[102] => в начале было слово, и слово было у бога, и слово было бог
-----
It Is A Short English String
[6] => <It> <Is> <A> <Short> <English> <String>
func: <It> <Is> <A> <Short> <English> <String>
[28] => it is a short english string
-----
```

Поскольку для отладки утомительно повторять на вводе тестовые строки, то вы их можете заготовить заранее в файл (образец такого файла прилагается в архиве), и выполнять приложение так:

```
$ ./replw < wrd.txt
...
```

Вариант не со строками `string`, но с байтовыми срезами `[]byte` (с функцией замены `func rep2up (b []byte) []byte`) очень похож, и приводится в архиве примеров как `replwb.go`.

Источники информации

[1] Strings, bytes, runes and characters in Go — <https://go.dev/blog/strings>

[2] Syntax, Aug 11, 2023 — Синтаксис регулярных выражений Go —

<https://github.com/google/re2/wiki/Syntax>

Объектно-ориентированное программирование

Язык Go объектно-ориентирован (хотя авторы и избегают термина называть его именно так) не в меньшей степени, чем более привычные C++ или Java, только объектность здесь совсем другой природы. Об этом уже было, и достаточно подробно, рассказано ранее, при описании общих принципов языка (в первой части).

Теперь же мы посмотрим несколько примеров, выполненных в объектно-ориентированной технике исполнения. Безусловно, показать технику объектов (пользовательских типов и их методов) нельзя на таких игрушечных примерах, которые представлялись ранее. Поэтому и примеры эти будут более обстоятельными...

Обсчёт параметров 2D выпуклых многоугольников

Для демонстрационной реализации была выбрана задача (каталог oop/triangle архива примеров) расчёта параметров 2D (на плоскости) треугольника, заданного координатами своих вершин, а именно: расчёт периметра и площади. По ходу развития эта задача была переформулирована как расчёт тех же параметров, но для произвольных выпуклых 2D многоугольников. При этом N-угольник просто представляется как N - 2 составляющих его треугольников.

Координаты вершин будут выражаться как комплексное значение — это естественно для физического мира, так как комплексные величины это и есть отображение точек 2D-плоскости. Но самое главное, что такой подход с самого начала потребует работы со структурными объектами (2-х компонентные комплексные значения). В Go работа с комплексными значениями обеспечивает пакет `math/cmplx`. А геометрическая фигура (треугольник, многоугольник) естественным образом подталкивает к использованию понятий класса и объекта. Есть где разгуляться!

Но прежде, чем приступить к реализациям, нужно сделать минимальный экскурс в теорию комплексных вычислений. Каждое комплексное число представляется суммой вещественной и мнимой компонент:

$$z = \text{real} + i * \text{image}$$

Здесь `real` — это вещественная часть числа, а `image` — мнимая его часть (`real` и `image` здесь конкретный числовые, вещественные значения, компоненты для данного конкретного комплексного числа)... (я мог бы рассказать ещё, что i — это величина, равная $\sqrt{-1}$, и что это означает ... но это ровно ничего не добавит к целям нашего рассмотрения).

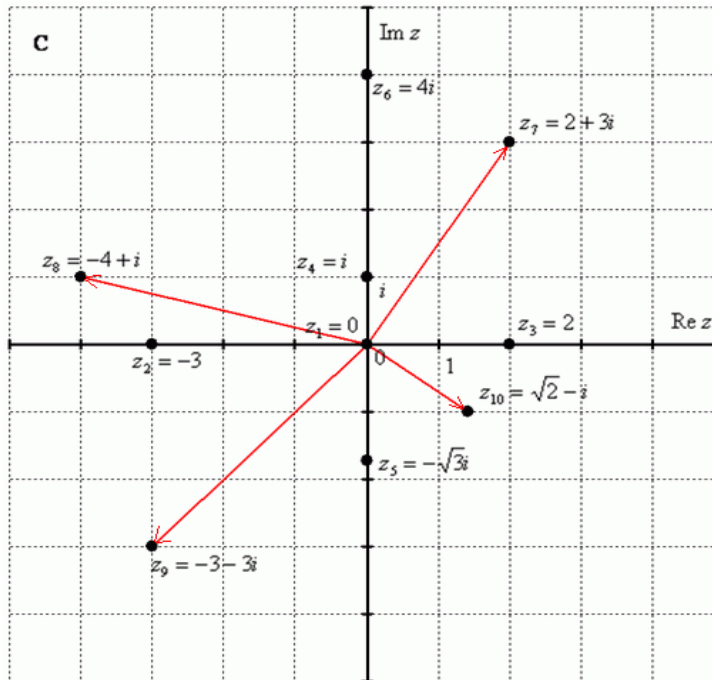
На вещественной плоскости (2D) комплексное число z отображается точкой, для которой: `real` — это координата точки по горизонтали (ось X), а `image` — это координата точки по вертикали (ось Y). Также каждое комплексное число имеют другую форму представления, так называемую экспоненциальную, вида:

$$z = \text{abs} * \exp(i * \text{arg})$$

Здесь `abs` — это длина вектора z (от точки 0,0), а `arg` — фазовый угол наклона (против часовой стрелки) вектора относительно положительного направления оси X, выраженный в радианах.

Эти две формы представления описывают одну и ту же точку плоскости, и между ними существуют взаимно однозначные соответствия. Они связаны соотношениями (все показанные математические функции присутствуют в библиотеке любого языка программирования):

$$\text{real} = \text{abs} * \cos(\text{arg})$$
$$\text{image} = \text{abs} * \sin(\text{arg})$$
$$\text{abs} = \sqrt{\text{real}^2 + \text{image}^2}$$
$$\text{arg} = \text{atan2}(\text{image}, \text{real})$$
$$z = \text{abs} * \exp(i * \text{arg}) = \text{abs} * (\cos(\text{arg}) + i * \sin(\text{arg}))$$



```

Z1 = 0. + 0. * i
Z2 = -3. + 0 * i
Z3 = 2. + 0 * i
Z5 = 0 - sqrt( 3. ) * i
Z6 = 0 + 4 * i
Z7 = 2. + 3. * i
Z8 = -4 + i
Z9 = -3 - 3 * i
Z10 = sqrt( 2. ) - i

```

Рис. 3.1. Геометрическая интерпретация множества комплексных чисел: двумерная координатная плоскость

В каждый момент вычислений мы используем ту форму, одну из 2-х, представления комплексного числа, которая нам удобнее в данный момент. Математические библиотеки манипуляции с комплексными числами содержат встроенные функции преобразования из одной формы в другую. Например, для показанных на рисунке некоторых чисел (векторов) имеет место соотношения (угол \arg показан в радианах, долях π и в угловых градусах для наглядности — это одно и то же значение):

```

z1  =  (+2.0 , +3.0i)  <=>  abs = 3.606 , arg = 0.983 = 0.31*π = 56°
z5  =  (-0.0 , -1.7i)  <=>  abs = 1.732 , arg = -1.571 = -0.50*π = -90°
z8  =  (-4.0 , +1.0i)  <=>  abs = 4.123 , arg = 2.897 = 0.92*π = 166°
z9  =  (-3.0 , -3.0i)  <=>  abs = 4.243 , arg = -2.356 = -0.75*π = -135°
z10 =  (+1.4 , -1.0i)  <=>  abs = 1.732 , arg = -0.615 = -0.20*π = -35°

```

Зачем нам такие сложности? А затем, что дальше всё становится очень просто:

- вектор, замыкающий точки z_9 и z_8 будет вычисляться просто как $(z_8 - z_9)$;
- его длина (нужная нам как составляющая периметра) — как $\text{abs}(z_8 - z_9)$;
- а площадь треугольника, построенного на сторонах z_9 и z_8 будет вычисляться как:

$$\text{abs}(z_8) * \text{abs}(z_9) * \sin(\arg(z_8) - \arg(z_9)) / 2.$$

Мы можем пойти и далее (что и сделано в примере): любой произвольный выпуклый N -угольник с вершинами $[1 \dots N]$ может быть представлен как последовательность $N-2$ треугольников (рассекаемых из единой вершины, мы выберем 1), где K -й треугольник составят вершины $[1, K, K+1]$ исходного многоугольника. Тогда площадь произвольного многоугольника может быть найдена как сумма в цикле площадей K составляющих треугольников (всё это легко видеть далее по коду).

Реализация описанной задачи на языке Go (каталог `triangle`):

triangle.go

```
package main
```

```
import (
```

```
    "errors"
```

```
    "fmt"
```

```

        "io"
        "math"
        "math/cmplx"
        "os"
        "strconv"
        "strings"
    )

    // ----- класс точки вершины -----
    type point struct {
        xy complex128
    }

    func (p *point) String() string { // формат вывода
        return fmt.Sprintf("%.2f,%.2f ", real(p.xy), imag(p.xy))
    }

    func (p *point) inpoint() (ok bool, err error) { // ввод координат
        buf := make([]byte, 1024)
        ok = false
        n, err := os.Stdin.Read(buf)
        if err == io.EOF || n == 0 || buf[n-1] != '\n' { // конец ввода
            err = io.EOF
            return
        }
        as := strings.Split(string(buf[:n-1]), string(" "))
        if len(as) != 2 {
            err = errors.New("число параметров")
            return
        }
        x, err := strconv.ParseFloat(as[0], 64)
        if err != nil {
            return
        } // ошибка преобразования
        y, err := strconv.ParseFloat(as[1], 64)
        if err != nil {
            return
        } // ошибка преобразования
        p.xy = complex(x, y)
        ok, err = true, nil
        return
    }

    // ----- класс многоугольника -----
    type shape []point

    func (p *shape) append(data point) {
        slice := *p
        l := len(slice)
        if l+1 > cap(slice) { // недостаточно места
            newSlice := make([]point, (l+1)*2) // выделение вдвое большего
буфера
            if l > 0 { // скопировать данные
                for i, c := range slice {
                    newSlice[i] = c
                }
            }
            slice = newSlice
        }
        slice = slice[0 : l+1]
        slice[l] = data
        *p = slice
    }

    func (p *shape) String() string { // формат вывода

```

```

        slice := *p
        var s string = ""
        for _, c := range slice {
            s += c.String()
        }
        return s
    }
}

func (p *shape) perimeter() float64 {
    summa := 0.0
    slice := *p
    for i, c := range slice {
        if i == 0 {
            summa += cmplx.Abs(c.xy - slice[len(slice)-1].xy)
        } else {
            summa += cmplx.Abs(c.xy - slice[i-1].xy)
        }
    }
    return summa
}

func (p *shape) square() float64 {
    summa := 0.0
    slice := *p
    for i := 0; i < len(slice)-2; i++ {
        r1, θ1 := cmplx.Polar(slice[i+1].xy - slice[0].xy)
        r2, θ2 := cmplx.Polar(slice[i+2].xy - slice[0].xy)
        summa += r1 * r2 * math.Abs(math.Sin(θ2-θ1)) / 2.
    }
    return summa
}

}

func main() {
    for {
        fmt.Println("координаты вершин в формате: X Y (^D конец ввода)")
        многоугольник := new(shape)
        i := 0
        точка := new(point)
        for {
            fmt.Printf("вершина № %v: ", i+1)
            ok, err := точка.inpoint() // ввод координат вершины
            if !ok {
                if err == io.EOF {
                    fmt.Printf("\r")
                    break
                } // конец ввода вершин
                fmt.Printf("ошибка ввода: %s!\n", err)
                continue
            }
            многоугольник.append(*точка)
            i++
        }
        fmt.Printf("вершин %d : %v\n", len(*многоугольник), многоугольник)
        fmt.Printf("периметр = %.2f\n", многоугольник.perimeter())
        fmt.Printf("площадь = %.2f\n", многоугольник.square())
        fmt.Println("-----")
    }
}

```

Сборка:

```
$ go build -o triangle triangle.go
```

И вот как выполняется только-что собранное нами приложение и, в частности:

- показан ввод 3-угольника, 4-угольника — конец ввода точек вершин завершается по набору обычной для UNIX комбинации EOF : ^D (Ctrl + D);

- показано несколько вариантов реакции на ошибки ввода с терминала пользователем;

\$./triangle

координаты вершин в формате: X Y (^D конец ввода)

вершина № 1: 1 1

вершина № 2: 1. 2

вершина № 3: 2. 1

вершин 3 : [1.00,1.00] [1.00,2.00] [2.00,1.00]

периметр = 3.41

площадь = 0.50

координаты вершин в формате: X Y (^D конец ввода)

вершина № 1: 1 2

вершина № 2: 1.0 2

вершина № 3: 2 2.

вершина № 4: 2 1

вершин 4 : [1.00,2.00] [1.00,2.00] [2.00,2.00] [2.00,1.00]

периметр = 3.41

площадь = 0.50

координаты вершин в формате: X Y

вершина № 1: 3.3

ошибка ввода: число параметров!

вершина № 1: 1. 2. 3. 4.

ошибка ввода: число параметров!

вершина № 1: 2.2 4.r

ошибка ввода: strconv.ParseFloat: parsing "4.r": invalid syntax!

вершина № 1: k 5

ошибка ввода: strconv.ParseFloat: parsing "k": invalid syntax!

вершина № 1: ^C

Комплексная математика — наиважнейший раздел для всех, кто занимается теоретической электротехникой и радиотехникой. И здесь Go оказывает им естественную и легко используемую поддержку.

Активные объекты

Одна из самых интересных тем из области объектно-ориентированного программирования — активные объекты. Особенно активно эти темы пошли в обсуждение с языка программирования Oberon Никлауса Вирта. Активные объекты — это такие объекты, которые после своего создания содержат в себе постоянно действующую активность (поток POSIX `pthread_t`, горутину Go ... всё что угодно), активность объекта всё время его жизни не прекращается, выполняет какую-то полезную работу. А взаимодействие с внешним окружением (программным кодом) активный объект осуществляет через какие-то механизмы взаимодействия и синхронизации параллельных ветвей.

В качестве примера, чтобы не усложнять, рассмотрим реализацию в технике активного объекта логер некоего проекта:

- Код проекта, в различных его точках, может генерировать сообщения логирования разных классов грубости...
- Физическое логирование (после формирования самого сообщения) может быть достаточно продолжительным ... это может быть, например, отправка сообщения через Интернет с помощью электронной почты ... а то ещё и с ожиданием подтверждения доставки, если это критическая область применения.
- Основной код приложения после отправки сообщения лога не должен беспокоиться о его дальнейшей судьбе, и не должен нисколько задерживаться после отправки сообщения.

Схема упрощена относительно деталей реальной жизни, но вот её мы и станем реализовывать (каталог oop/logger, базовый прототип этой реализации взят из [3]) :

logger.go

```
package main

import (
    "fmt"
    "sync"
    "time"
)

type LogSeverityType string

const (
    Info      LogSeverityType = "Info"
    Warning   LogSeverityType = "Warning"
    Error     LogSeverityType = "Error"
    None      LogSeverityType = "None"
)

type LogMessage struct {           // структура сообщений для логера
    Severity LogSeverityType
    Message   string
}

type ActiveLogger struct {         // данные Active Object
    logQueue chan *LogMessage // канал логирования
    stopEvent chan struct{}   // канал завершения
    tmStart   time.Time       // время старта
    wg        sync.WaitGroup
}

func createActiveLogger() *ActiveLogger {
    return &ActiveLogger{
        logQueue: make(chan *LogMessage, 10),
        stopEvent: make(chan struct{}),
        tmStart:   time.Now(),
    }
}

func (l *ActiveLogger) Log(message *LogMessage) {
    l.logQueue <- message
}

func (l *ActiveLogger) StartLogProcessor() {
    l.wg.Add(1)
    go func() {
        // loop Active Object
        defer l.wg.Done()
        for {
            // цикл обслуживания
            select {
            case message := <-l.logQueue:
                l.processMessage(message)
            case <-l.stopEvent:
                return
            }
        }
    }()
}

// фактическое логирование: тут может быть любая длинная работа...
func (l *ActiveLogger) processMessage(m *LogMessage) {
    out := func(m string) {
```

```

        dur := func() time.Duration {
            return time.Now().Sub(l.tmStart)
        }
        fmt.Printf("% 14v : %s", dur(), m)
    }
    l.wg.Add(1)
    go func() {
        defer l.wg.Done()
        out(fmt.Sprintf("логируется... [%s]:\t%s\n", m.Severity, m.Message))
        time.Sleep(500 * time.Millisecond)
        out(fmt.Sprintf("залогировано! [%s]:\t%s\n", m.Severity, m.Message))
    }()
}

func (l *ActiveLogger) StopLogProcessor() {
    message := &LogMessage{
        Message:  fmt.Sprintf("завершение логера"),
        Severity:  None,
    }
    l.Log(message) // сообщение завершения логера
    close(l.stopEvent) // закрыть канал завершения
    l.wg.Wait() // ... и ожидать завершения рабочего цикла
}

func main() {
    logger := createActiveLogger() // создание Active Object
    logger.StartLogProcessor() // запуск активности ...
    const N = 5
    for i := range N {
        var Severity LogSeverityType
        if i < N-1 {
            Severity = Info
        } else {
            Severity = Error
        }
        message := &LogMessage{
            Message:  fmt.Sprintf("сообщение номер %d", i),
            Severity:  Severity,
        }
        logger.Log(message) // сообщение логеру ...
    }
    time.Sleep(2 * time.Second)
    logger.StopLogProcessor() // завершение логера
}

```

Очень коротко о том как это работает:

- После создания объекта `createActiveLogger()` для него тут же выполняется вызов метода `createActiveLogger()`, по которому в объекте начинает выполняться горутина с бесконечным циклом ожидания взаимодействия (метод `StartLogProcessor()`);
- Горутина ожидает взаимодействия на двух каналах:
- ... по каналу `stopEvent` приходит сообщение на завершение работы — один и последний раз;
- ... по каналу `logQueue` приходят регулярно сообщения для логирования;
- Само логирование достаточно продолжительное (метод `processMessage()`), представлено оно пассивной задержкой (в примере обработка лога требует 0.5 секунды);
- При логировании к сообщению добавляется префикс временной метки —

временная задержка отсчитываемая от момента запуска активного объекта;

- Дополнительная синхронизация `sync.WaitGroup` требуется для того, чтобы запретить завершение задачи (и активности объекта) до того момента (возможно, в некотором будущем, до 0.5 секунд), когда полностью завершится фактическая отправка всех переданных до этого сообщений логов.

Все остальные детали хорошо видны из кода. И вот как это работает:

```
$ go run logger.go
199.844µs : логируется... [Error]: сообщение номер 4
226.752µs : логируется... [Info]:  сообщение номер 3
236.534µs : логируется... [Info]:  сообщение номер 1
170.442µs : логируется... [Info]:  сообщение номер 2
331.345µs : логируется... [Info]:  сообщение номер 0
500.956811ms : залогировано! [Info]:  сообщение номер 0
501.002193ms : залогировано! [Info]:  сообщение номер 1
501.027609ms : залогировано! [Error]: сообщение номер 4
501.035735ms : залогировано! [Info]:  сообщение номер 2
501.046721ms : залогировано! [Info]:  сообщение номер 3
2.001640435s : логируется... [None]:  завершение логера
2.502352703s : залогировано! [None]:  завершение логера
```

```
$ go run logger.go
100.011µs : логируется... [Error]: сообщение номер 4
123.31µs : логируется... [Info]:  сообщение номер 3
133.566µs : логируется... [Info]:  сообщение номер 0
155.837µs : логируется... [Info]:  сообщение номер 2
174.337µs : логируется... [Info]:  сообщение номер 1
500.855252ms : залогировано! [Info]:  сообщение номер 1
500.873195ms : залогировано! [Info]:  сообщение номер 2
500.90007ms : залогировано! [Info]:  сообщение номер 0
500.903716ms : залогировано! [Info]:  сообщение номер 3
500.944625ms : залогировано! [Error]: сообщение номер 4
2.001708059s : логируется... [None]:  завершение логера
2.502475414s : залогировано! [None]:  завершение логера
```

Специально показаны два последовательных запуска, чтобы было видно, что последовательность выполнения параллельных ветвей случайная — это, обычно, не достаточный, но необходимый (грубый) признак того, что в параллельном коде всё действительно играет параллельно. (Если у вас соблюдается строгий детерминизм между запусками, значит с параллелизмом у вас что-то не так!)

Встраивание, агрегирование и наследование

Объектность Go вовсе не такая как в C++ или Java, в Go отсутствует наследование типов, а для похожей, но не идентичной, конструкции используется анонимное вложение типов, встраивание. Вот как-то так (повторим то что уже было сказано в первой части):

```
...
type ColoredPoint struct {
    color.Color          // Анонимное (безымянное) поле (встраивание)
    x, y int              // Именованные поля (агрегирование)
}
```

Такое анонимное встраивание может делаться многократно, в несколько уровней, и все поля и методы встроенных типов при этом доступны напрямую в охватывающем типе (без именованного промежуточных уровней вложения).

Именно иллюстрация таких производных типов и, что ещё посложнее, их именованная и инициализация — это и будет 1-й задачей этого примера.

А 2-й задачей, чтобы код попусту не переводить — это будет использование такой группы пакетов стандартной библиотеки GoLang, которая именуется как контейнеры (container). И в эту группу входят 3 разных пакета для импорта:

1. `heap` (<https://pkg.go.dev/container/heap>) — пакет предоставляет операции с кучей памяти для объектов любого типа, реализующих интерфейс `heap.Interface`.
2. `list` (<https://pkg.go.dev/container/list>) — пакет реализует операции для двусвязных линейных списков.
3. `ring` (<https://pkg.go.dev/container/ring>) — пакет реализует операции для двунаправленных циклических списков.

И использовать в своём коде мы будем именно пакет для реализации циклических списков. Это хорошая идея, и хороша она вот чем:

- Это именно тот кольцевой двусвязный список (в структурном а не кодовом смысле), на котором построены подавляющее большинство динамических структур ядра Linux (причём, именно начиная с некоторой версии ядра 2.6 были перестроены заново реализации все другие «ручные» реализации, применявшиеся ранее).
- На таком базисе позже можно строить практически всё что вздумается из области динамических структур: списки, стеки, очереди, деревья, сбалансированные деревья, красно-чёрные деревья ...
- Наконец, просто потому, что с таким кольцевым списком нельзя никогда по ошибке разименовать указатель `nil`, с тяжелейшими последствиями этого действия ... просто потому что в таком списке вообще нет концевых `nil` указателей.

Утилитарной полезности, осмысленности и функциональности в проектируемом примере будет мало, но зато будет во множестве на практике манипуляций с данными.... и встраивание, и агрегирование... именно то, к чему мы здесь и стремились:

ring.go

```
package main

import (
    "container/ring"
    "fmt"
    "github.com/eiannone/keyboard"
    C "image/color"
    "unicode"
)

/* type RGBA struct { // структура цвета (импортируемая)
 *     R, G, B, A uint8
 * }
 */

type point struct { // структура координат
    x, y float32
}

type pixel struct {
    point
    C.RGBA
}

func px(x, y float32, r, g, b, a uint8) pixel { // сокращённая форма инициализации
    return pixel{point{x, y}, C.RGBA{r, g, b, a}}
}

func (p pixel) String() string { // формат вывода
    return fmt.Sprintf("[%5.2f,%5.2f]{%03v,%03v,%03v,%03v}",
        p.x, p.y, p.R, p.G, p.B, p.A)
}
```

```

type shape []pixel

func main() {
    s := shape {
        px(2, 0, 255, 0, 0, 255),
        px(0, 2, 127, 127, 0, 255),
        px(-2, 0, 0, 255, 0, 255),
        px(0, -2, 0, 127, 127, 255),
    }
    for i, x := range s {
        fmt.Printf("%d -> %v\n", i, x)
    }
    var r *ring.Ring // голова циклического списка
    for i, z := range s {
        x := ring.New(1)
        x.Value = z
        if i == 0 {
            r = x
        } else {
            r := r.Link(x)
        }
    }
    show := func() {
        // Итерация поэлементно по всем элементам и их вывод
        r.Do(func(p any) { fmt.Print(p.(pixel), " ") })
        println()
    }
    show()

    keyboard.Open() // неблокирующий ввод с клавиатуры
    defer keyboard.Close()
    chChan := make(chan rune) // канал ввода с клавиатуры
    go func(chChan chan<- rune) { // горутина ввода клавиши
        for {
            ch, _, _ := keyboard.GetKey()
            chChan <- ch
        }
    }(chChan)

    for {
        ch := <-chChan // вводимые символы
        if ch == 0 {
            break
        } else if unicode.IsControl(ch) {
            break
        } else if unicode.IsLetter(ch) {
            r = r.Next()
        } else if unicode.IsDigit(ch) {
            r = r.Prev()
        } else {
            break
        }
        show()
    }
    println()
}

```

В отношении объектов данных:

- Там `type shape` — это есть представление любой 2D геометрической фигуры, массив объектов типа `pixel`, обладающих координатой и цветом...

- ... в который **встраиваются**:
 - импортируемый тип `color.RGBA` ...
 - самописный тип `point` ...
 - в который, в свою очередь, **агрегированы** 2 `float32` координаты со своими именами: `X` и `Y` ...
- Для типа `pixel` показан **метод** `String()`, реализующий интерфейс `Stringer`, чем и обеспечивается единообразный формат вывода для всего семейства показанных типов.
- И 1-я задача: начально инициализировать всю эту иерархию, начиная от `type shape` фигуры ... что и делается одним объявлением (пока я лучше не придумал), это такой вот 4-угольник:


```
s := shape { px( 2,  0, 255,  0,  0, 255),
              px( 0,  2, 127, 127,  0, 255),
              px(-2,  0,  0, 255,  0, 255),
              px( 0, -2,  0, 127, 127, 255),
            }
```
- Далее элементы этого массива поочерёдно переносятся (копируются) в циклический (круговой) список с указателем его текущего положения «головы» `r`.
- В последней части организуется неблокирующий ввод символа (нажатия с клавиатуры), если нажимается алфавитная клавиша, то указатель «головы» `r` сдвигается на `Next()` элемент по списку, если цифровая клавиша — на `Prev()` элемент, а если вводится непечатный символ (`Ctrl+D`, `Enter`, пробел ...), то происходит выход из цикла и завершение задачи.
- При завершении нужно чтобы обязательно выполнилось: `defer keyboard.Close()`, иначе клавиатура останется в неканоническом режиме ввода (посимвольном), и вкладка терминала, в которой производится отладка, будет испорчена.

Приступаем к сборке... Но тут нас поджидает некоторый облом:

```
$ go build ring.go
```

```
ring.go:9:4: no required module provides package github.com/eiannone/keyboard: go.mod
file not found in current directory or any parent directory; see 'go help modules'
```

Но это дело известное, оно рассматривалось в первой части книги, и приводится здесь только в порядке напоминания: начиная с версии Go 1.11 и введения системы поддержки модулей, для благополучного импорта пакетов из сети нам нужно сделать предварительно, как минимум:

```
$ go mod init ring
```

```
go: creating new go.mod: module ring
go: to add module requirements and sums:
  go mod tidy
```

Сформируется файл манифеста текущего (вашего разрабатываемого) модуля `go.mod`, и далее либо реально загрузится внешний пакет из Интернет, либо отработается ссылка на такой загруженный пакет, если он ранее закачивался при отработке ваших других проектов на этом компьютере:

```
$ go mod tidy
```

```
go: finding module for package github.com/eiannone/keyboard
go: found github.com/eiannone/keyboard in github.com/eiannone/keyboard v0.0.0-20220611211555-0d226195f203
```

И теперь можно благополучно произвести сборку исполнимого файла приложения:

```
$ go build ring.go
```

```
$ ls -o ring
```

И можем приступить к выполнению и изучению полученного (на терминале вывод, естественно, выглядит в одну строку без переносов):

```
$ go run ring.go
0 -> [ 2.00, 0.00]{255,000,000,255}
1 -> [ 0.00, 2.00]{127,127,000,255}
2 -> [-2.00, 0.00]{000,255,000,255}
3 -> [ 0.00, -2.00]{000,127,127,255}
[ 2.00, 0.00]{255,000,000,255} [-2.00, 0.00]{000,255,000,255} [ 0.00, 2.00]
{127,127,000,255} [ 0.00, -2.00]{000,127,127,255}
[ 0.00, -2.00]{000,127,127,255} [ 2.00, 0.00]{255,000,000,255} [-2.00, 0.00]
{000,255,000,255} [ 0.00, 2.00]{127,127,000,255}
[ 2.00, 0.00]{255,000,000,255} [-2.00, 0.00]{000,255,000,255} [ 0.00, 2.00]
{127,127,000,255} [ 0.00, -2.00]{000,127,127,255}
[ 0.00, -2.00]{000,127,127,255} [ 2.00, 0.00]{255,000,000,255} [-2.00, 0.00]
{000,255,000,255} [ 0.00, 2.00]{127,127,000,255}
[ 2.00, 0.00]{255,000,000,255} [-2.00, 0.00]{000,255,000,255} [ 0.00, 2.00]
{127,127,000,255} [ 0.00, -2.00]{000,127,127,255}
[ 0.00, -2.00]{000,127,127,255} [ 2.00, 0.00]{255,000,000,255} [-2.00, 0.00]
{000,255,000,255} [ 0.00, 2.00]{127,127,000,255}
[ 2.00, 0.00]{255,000,000,255} [-2.00, 0.00]{000,255,000,255} [ 0.00, 2.00]
{127,127,000,255} [ 0.00, -2.00]{000,127,127,255}
```

Хорошо видно чередование одинаковых строк — это чередующиеся нажатия клавиш «вперёд» и «назад»...

От этого примера происходит мало практической пользы, но много понимания об объектах и структурах данных в Go.

Верификация банковских карт

Алгоритм Луна [5] — этот алгоритм не имеет никакого отношения к астрономии, к ночному небесному светилу, или к вопросу «а были ли американцы на Луне вообще?»... Это проверочный алгоритм цифровой последовательности, которому, кроме всего прочего, удовлетворяю номера всех ваших банковских карт!

Алгоритм Луна (Luhn algorithm) — алгоритм вычисления контрольной цифры номера пластиковой карты в соответствии со стандартом ISO/IEC 7812. Не является криптографическим средством, а предназначен в первую очередь для выявления ошибок, вызванных непреднамеренным искажением данных (например, при ручном вводе номера карты, при приёме данных о номере социального страхования по телефону). ... Алгоритм разработан сотрудником фирмы IBM Хансом Питером Луном, описан в США в 1954 году, патент получен в 1960 году. В настоящее время алгоритм является публичным достоянием.

U.S. Patent 2 950 048 Computer for Verifying Numbers, Hans P. Luhn, August 23, 1960.

Реализуем этот алгоритм (каталог oop/luna архива примеров):

luna.go

```
package main
```

```
import (
    "bufio"
    "errors"
    "fmt"
    "io"
    "os"
)
```



```

const LEN = 16

type card struct {
    num [LEN]byte
}

func (c card) valid() (val bool, err error) {
    val = false
    err = nil
    var s byte = 0
    for i, d := range c.num {
        if !(d >= byte('0') && d <= byte('9')) {
            s := fmt.Sprintf("ошибка: не число!")
            err = errors.New(s)
            return
        } else {
            c.num[i] = c.num[i] - byte('0')
            if i%2 == 0 {
                c.num[i] = c.num[i] * 2
                if c.num[i] > 9 {
                    c.num[i] -= 9
                }
            }
            s += c.num[i]
        }
    }
    return s % 10 == 0, nil
}

func (c card) String() string { // формат вывода
    s := ""
    for i := range LEN {
        s += string(c.num[i])
    }
    return s
}

func main() {
    for {
        crd, err := incard()
        if err == nil {
            v, err := crd.valid()
            if err != nil {
                fmt.Println(err)
                continue
            }
            fmt.Printf("номер %s : ", crd)
            if v {
                fmt.Printf("верифицирован\n")
            } else {
                fmt.Printf("не верифицирован\n")
            }
        } else if err == io.EOF {
            break
        } else {
            fmt.Println(err)
            break
        }
    }
}

```

```

func incard() (*card, error) {
    scanner := bufio.NewScanner(os.Stdin)
    if !scanner.Scan() {
        return nil, io.EOF // ввод ^D - EOF
    }
    line1 := scanner.Text()
    if 0 == len(line1) {
        return nil, io.EOF // ввод Enter - пустая строка
    }
    line2 := ""
    for _, c := range line1 {
        if c != ' ' {
            line2 += string(c)
        }
    }
    if len(line2) != LEN {
        err := errors.New(fmt.Sprintf("длина %d должна быть %d",
            len(line2), LEN))
        return nil, err
    }
    crd := new(card)
    for i, c := range line2 {
        crd.num[i] = byte(c)
    }
    return crd, nil
}

```

Собственно, алгоритмический интерес здесь представлен в методе `valid()` типа `card` который и представляет 16-цифровой номер банковской карты.

Проверяем:

```

$ ./luna
4561 2612 1234 5464
номер 4561261212345464 : не верифицирован
4561 2612 1234 5467
номер 4561261212345467 : верифицирован
5106 2110 7031 8501
номер 5106211070318501 : верифицирован

```

Или так:

```

$ echo 4561261212345467 | go run luna.go
номер 4561261212345467 : верифицирован

```

Источники информации

- [1] Андрей Шагин, Объектно-ориентированное программирование в Golang, 2022 — <https://medium.com/nuances-of-programming/объектно-ориентированное-программирование-в-golang-52f36f2fa837>
- [2] Understanding Object Orientation in Golang, Aug 16, 2022 — <https://aysf.github.io/p/go-oo/>
- [3] Iede Snoek, Easy Concurrency: Active Object Pattern in Go Explained, Nov 25, 2023 — <https://www.hackingwithgo.nl/2023/11/25/easy-concurrency-active-object-pattern-in-go-explained/>
- [4] ring — <https://pkg.go.dev/container/ring>
- [5] Алгоритм Луна — https://ru.wikipedia.org/wiki/Алгоритм_Луна

Элементы функционального программирования

Функциональный стиль программирования может показаться, по внешнему виду порождаемых кодов, искусственным и усложнённым. Но у него много сторонников. Вопрос: так в чём же особенно хорош функциональный стиль? Да в том, что при многочисленных вызовах **чистых** (в идеале) функций не возникает побочных эффектов, изменений данных промежуточных переменных. А это оберегает от тонких ошибок, с трудом подлежащих локализации и исправлению.

Язык Go рассматривает функции как объекты первого класса. В частности, это означает, что язык поддерживает передачу функций в качестве аргументов другим функциям, возврат их как результат других функций, присваивание их переменным или сохранение в структурах данных. Здесь имена функций не имеют никакого специального статуса, они рассматриваются как обычные значения, тип которых является **функциональным**. А значит это позволяет определять функции **высшего порядка**, которые могут работать с функциональными значениями (принимать на вход функции и возвращать их в качестве результата). А это является основой функционального программирования. Дополнительную гибкость предоставляют возможности анонимных функций и функциональных литералов.

И если Go не является и не позиционируется как язык функционального программирования (как Common Lisp, Scheme, Ocaml, или Haskell), тем не менее он позволяет в очень существенной степени использовать приёмы функционального программирования.

Основные приёмы функционального стиля программирования кажутся очень необычными, непонятными ... и, временами, даже неприятными для тех, кто работает в императивном программировании (традиционном). Поэтому в этой части рассмотрения мы будем особо детально разбирать каждый из элементов и приёмов функционального программирования.

Функциональные замыкания

О функциональных замыканиях (closure) было уже сказано вскользь при рассмотрении функций ранее. Это одно из самых полезных понятий функционального программирования. Эта идея оказалась настолько заманчивой для многих разработчиков, что была реализована даже в некоторых совершенно не функциональных языках программирования (Perl). Девид Мертц приводит следующее определение замыкания:

Замыкание - это процедура вместе с привязанной к ней совокупностью данных (в противовес объектам в объектном программировании, как: «данные вместе с привязанным к ним совокупностью процедур»).

Смысл замыкания состоит в том, что определение функции «замораживает» окружающий её контекст на **момент определения** этой функции (это очень важно, что не на момент **вызова**).

Функциональные замыкания — это такая мощная техника, которую заимствуют самые разнообразные языки, не являющиеся языками функционального программирования. Широко приёмы функционального программирования используются (или предоставляются средства для реализации) в языке Python (степень «предрасположенности» здесь практически та же, что мы увидим в Go). Здесь есть уже широта выбора метода реализации для всё того же замыкания, вариантность — это может делаться разными способами, например, за счёт параметризации создания функции. Для начала парочка элементарнейших первый пример для понимания того чем является замыкание, потому что это не так просто как кажется:

clos1.py

```
#!/usr/bin/python3
```

```
def multiplier(n):    # multiplier возвращает функцию умножения на n
    def mul(k):
```

```

        return n * k
    return mul

mul3 = multiplier(3) # mul3 - функция, умножающая на 3

print(mul3(3), mul3(5))

```

Выполнение:

```

$ ./clos1.py
9 15

```

Аналогичная «функциональная конкретизаций» посредством замыкания на Go могла бы выглядеть так:

clos1.go

```

package main

import "fmt"

func makeAdder(x int) func(int) int {
    return func(y int) int {
        return x + y
    }
}

func main() {
    add5 := makeAdder(5)
    add10 := makeAdder(10)
    fmt.Printf("%d %d\n", add5(2), add10(2)) // 7 12
}

```

С ожидаемым результатом:

```

$ go run clos1.go
7 12

```

Другой, по смыслу, способ создания замыкания — это использование значения параметра по умолчанию в точке определения функции, как показано в следующем листинге:

clos3.py

```

#!/usr/bin/python3

n = 3
def mult(k, mul = n):
    return mul * k

n = 7
print(mult(3))
n = 13
print(mult(5))

n = 10
mult = lambda k, mul = n: mul * k
print(mult(3))

$ ./clos3.py
9
15
30

```

К этой точке мы могли увидеть только некоторые из вариантов создания и использования замыканий, и убедились, что они могут быть разнообразными и даже очень не похожими.

Общим остаётся то что: в функция на момент её **определения** «замораживается» окружающий контекст (некоторый набор значений переменных), и на момент **вызова** используется именно этот сохранённый контекст, не взирая на то, что к моменту вызова этот контекст мог поменяться самым радикальным образом.

И теперь же мы, для полноты освещения предмета, мы можем сравним как эта возможность может быть реализована в различных языках программирования (каталог всех примеров `functionals/closure`).

В C++ ранее версии C++11 (стандарт 2011г.) наиболее распространенный способ создания функции с скрытым состоянием было использование класса, который перегружает оператор `()`, чтобы сделать его экземпляры внешне похожими на вызов функции (функторы). Например, следующий далее код определяет `my_transform()` функцию (упрощенная версия STL `std::transform()`), которая применяет заданный унарный оператора (`op`) к каждому элементу массива (`in`), сохраняя результат действия в другой массив (`out`). Для накапливающего сумматора (т.е., { `x[0]`, `x[0]+x[1]`, `x[0]+x[1]+x[2]`, ...}) код создает функтор (`MyFunctor`), который отслеживает сохраняемое состояние (`total`) и передает его экземпляр функтору для выполнения `my_transform()`:

clos_cc.cc

```
#include <iostream>
#include <cstdint>

template <class UnaryOperator>
void my_transform(size_t n_elts, int* in, int* out, UnaryOperator op) {
    for(size_t i = 0; i < n_elts; i++)
        out[i] = op(in[i]);
}

class MyFunctor {
public:
    int total;
    int operator()(int v) {
        return total += v;
    }
    MyFunctor() : total(0) {}
};

int main( void ) {
    int data[7] = {8, 6, 7, 5, 3, 0, 9};
    const int len = sizeof(data) / sizeof(data[0]);
    int result[len];
    MyFunctor accumulate;
    my_transform(len, data, result, accumulate);
    std::cout << "Result is [ ";
    for(size_t i = 0; i < len; i++)
        std::cout << result[i] << (i == len - 1 ? " ]\n" : " ");
    return 0;
}
```

В стандарте C++11 появились **анонимные** функции («лямбда» функции — название идёт из языка Lisp где они именно таким ключевым словом и определялись), которые могут храниться в качестве значений переменных, передаваемых функциям. (В Go подобные функции часто называют функциональными литералами.) Они, помимо прочего, могут служить в качестве замыканий — могут ссылаться на состояния (значения), определяемые в их родительской области. Эта функциональность значительно упрощает `my_transform()`:

clos_cc11.cc

```
#include <iostream>
#include <cstdint>
```

```

#include <functional>

void my_transform(size_t n_elts, int* in, int* out,
                  std::function<int(int)> op) {
    for( size_t i = 0; i < n_elts; i++ )
        out[i] = op(in[i]);
}

int main( void ) {
    int data[7] = {8, 6, 7, 5, 3, 0, 9};
    const int len = sizeof(data) / sizeof(data[0]);
    int result[len];
    int total = 0;
    my_transform(len, data, result,
                 [&total](int v) {
                     return total += v;
                 });
    std::cout << "Result is [ ";
    for(size_t i = 0; i < len; i++)
        std::cout << result[i] << (i == len - 1 ? " ]\n" : " ");
    return 0;
}

```

Теперь возвратимся несколько «назад», и взглянем как некоторая подобная функциональность, в упрощённом виде, могла бы реализовываться в классическом С — пример накапливающего сумматора:

clos c.c

```

#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

int accumulate(int in) {
    static int total;
    if(in == INT_MIN) total = 0;
    else total += in;
    return total;
}

void my_transform(size_t n_elts, int* in, int* out, int (*op)(int)) {
    size_t i;
    for(i = 0; i < n_elts; i++)
        out[i] = op(in[i]);
}

int main( void ) {
    int data[7] = {8, 6, 7, 5, 3, 0, 9}, i;
    const int len = sizeof(data) / sizeof(data[0]);
    int* result = (int*)calloc(len, sizeof(int));
    accumulate(INT_MIN);
    my_transform(len, data, result, accumulate);
    printf("Result is [ ");
    for(i = 0; i < len; i++)
        printf("%d%s", result[i], (i == len - 1 ? " ]\n" : " "));
    free(result);
    return 0;
}

```

Здесь сохраняемое между вызовами состояние (*total*) образуется за счёт переменной объявленной *static* внутри функции. Ещё некоторое упрощение связывания значения с функцией может быть достигнуто в С за счёт расширения компилятора GCC (но не

стандартов C89 и C99!³⁵) - встроенного определения функции:

clos gcc.c

```
#include <stdio.h>
#include <stdlib.h>

void my_transform(size_t n_elts, int* in, int* out) {
    int total = 0;
    size_t i;
    int accumulate(int in) {
        return total += in;
    }
    for(i = 0; i < n_elts; i++)
        out[i] = accumulate(in[i]);
}

int main(void) {
    int data[7] = {8, 6, 7, 5, 3, 0, 9}, i;
    const int len = sizeof(data) / sizeof(data[0]);
    int* result = (int*)calloc(len, sizeof(int));
    my_transform(len, data, result);
    printf("Result is [ ");
    for(i = 0; i < len; i++)
        printf("%d%s", result[i], (i == len - 1 ? " ]\n" : " "));
    free(result);
    return 0;
}
```

Типичная реализация того же на Go выглядит в чём-то похожей на версию C++11 (с анонимным определением функции-операции):

clos go.go

```
package main
import "fmt"

func my_transform(in []int, xform func(int) int) (out []int) {
    out = make([]int, len(in))
    for idx, val := range in {
        out[idx] = xform(val)
    }
    return
}

func main() {
    data := []int{8, 6, 7, 5, 3, 0, 9}
    total := 0
    fmt.Printf("Result is %v\n", my_transform(data, func(v int) int {
        total += v
        return total
    })))
}
```

И посмотрим, например, как подобная задача может решаться на Python, поскольку с него мы и начинали:

clos.py

```
#!/usr/bin/python3

def accumulate(acc):
    global total
    total = acc
    def summa(inp):
```

³⁵ Этот вариант пройдёт с компилятором GCC, но не пройдёт, к примеру, с Clang.

```

        global total
        total = total + inp
        return total
    return summa

def my_transform(inp, out, op):
    for x in inp: out.append(op(x))

data = [ 8, 6, 7, 5, 3, 0, 9 ]
result = []
my_transform(data, result, accumulate(0))
print("Result is", result)

```

Мощность и привлекательность такой техники (на любом языке реализации) состоит в том, что в качестве структуры данных, отображающей внутреннее **состояние** функции замыкания, могут быть структуры произвольной степени сложности (но в наших простейших примерах — это целочисленная переменная `total`).

И теперь мы можем посмотреть сравнительное выполнение всех рассмотренных вариантов реализации (каталог `compare/closure`):

```

$ make
gcc -Wall -O0 clos_c.c -o clos_c
gcc -Wall -O0 clos_gcc.c -o clos_gcc
g++ -Wall -std=gnu++11 -O0 clos_cc.cc -o clos_cc
g++ -Wall -std=gnu++11 -O0 clos_cc11.cc -o clos_cc11
gccgo -Wall clos_go.go -g -O0 -o clos_go.gcc
go build -o clos_go -compiler gc clos_go.go

$ ./clos_c
Result is [ 8 14 21 26 29 29 38 ]

$ ./clos_gcc
Result is [ 8 14 21 26 29 29 38 ]

$ ./clos_cc
Result is [ 8 14 21 26 29 29 38 ]

$ ./clos_cc11
Result is [ 8 14 21 26 29 29 38 ]

$ ./clos_go
Result is [8 14 21 26 29 29 38]

$ ./clos_go.gcc
Result is [8 14 21 26 29 29 38]

$ ./clos.py
Result is [8, 14, 21, 26, 29, 29, 38]

```

В Go функции всегда являются полными замыканиями, что эквивалентно `[&]` в C++11. Важным отличием является то, что является недопустимым в C++11 для замыкания ссылаться на переменную, вне области её определения (что может быть вызвано, например, при `funarg problem` — передаче для использования лямбда-выражения, которое ссылается на локальные переменные). В Go это является совершенно допустимым (за счёт подсчёта ссылок использования для локально определённых переменных и работы сборщика мусора).

Карринг

Карринг (или каррирование, *curring*), ещё один приём из функционального программирования — преобразование функции от многих переменных в функцию, берущую свои аргументы по одному.

Примечание. Это преобразование было введено М. Шейнфинкелем и Г. Фреге и получило своё название в честь математика Хаскелла Карри, в честь которого также назван и язык программирования Haskell.

Карринг не относится к уникальным особенностям функционального программирования, так карринговое преобразование может быть записано, например, и на языках Perl или C++. Оператор каррирования даже встроен в некоторые языки программирования (ML, Ocaml, Haskell), что позволяет многоместные функции приводить к каррированному представлению. Но все языки, поддерживающие замыкания, позволяют записывать каррированные функции, и Python не является исключением в этом плане.

Ещё одна формулировка карринга, которую приходится слышать: *карринг функции* — это изменение функции от вида $func(a,b,c)$ до вида $func(a)(b)(c)$.

Элементарный пример использования карринга в Python представлен таким листингом (каталог `functionals/currying`) простейший пример :

curry1.py

```
#!/usr/bin/python3

def show(x, y):
    print('param1={}, param2={}'.format(x, y))

spam1 = lambda x : lambda y : show(x, y)

def spam2(x) :
    def new_spam(y) :
        return show(x, y)
    return new_spam

spam1(2)(3)      # карринг
spam2(2)(3)
```

Здесь нет ничего необычного ... за исключением того, что и в первом и во втором случае здесь вызов `spam(2)` возвращает **функцию**, которая уже затем, в свою очередь, может быть вызвана с **одним** параметром:

```
$ ./curry1.py
param1=2, param2=3
param1=2, param2=3
```

Практически аналогичный по смыслу код мы можем записать и на Go:

curry1.go

```
package main

import ("fmt")

func mkAdd(a int) func(int) int {
    return func(b int) int {
        return a + b
    }
}

func main() {
    add2 := mkAdd(2)
    add3 := mkAdd(3)
    fmt.Println(add2(5), add3(6))
}

$ go build curry1.go

$ ./curry1
7 9
```

Рекурсия

Итерация *свойственна*
человеку, *рекурсия*
божественна.
L. Peter Deutsch

Рекурсивные вычисления являются мощнейшим механизмом! В функциональной парадигме принято отдавать предпочтение рекурсии — для чистоты и прозрачности, вместо использования циклического перебора элементов в циклах через: `for`, `while`, `until...`

По рекурсивным вычислениям уже приводилось достаточно много примеров (попутно с другими целями), но ещё одним хорошим дополнительным примером, с большим практическим смыслом, и особенностями выразительности Go — будет представление и обход бинарного дерева (каталог `function`):

btree.go

```
package main

import "fmt"

type Node struct {
    value    int
    left, right *Node
}

func (n *Node) Traverse(f func(int)) {
    if n == nil {
        return
    }
    n.left.Traverse(f)
    f(n.value)
    n.right.Traverse(f)
}

func main() {
    root := &Node{
        value: 3,
        left: &Node{
            value: 2,
            left: &Node{value: 1}},
        right: &Node{
            value: 5,
            left: &Node{value: 4},
            right: &Node{value: 6}}}
    var values []int
    root.Traverse(func(v int) { values = append(values, v) })
    fmt.Println(values)
}
```

Здесь показан рекурсивный обход всего дерева:

```
$ go fmt btree.go
$ go run btree.go
[1 2 3 4 5 6]
```

По аналогии, легко могут быть реализованы и любые другие, традиционные для деревьев операции: добавление и удаление ветвей и терминальных вершин ... и другие.

А то что выше названо «особенностями выразительности Go», это имелась в виду простота статической инициализации дерева, которая программисту с опытом C/C++

может показаться необычной... Здесь устраняется потенциальная возможность «самой страшной ошибки» C/C++ — не инициализированные (значениями NULL) терминальные ветви, с последующим разименованием не инициализированных указателей. Но в Go не может быть не инициализированных значений, и поэтому не инициализированные явно указатели получают неявно «нулевые значения» для переменных этого типа (указателей), равные nil.

Рекурсия с кэшированием

Обратимся снова к нашей многострадальной рекурсивной функции вычисления чисел Фибоначчи. Огромная степень её роста связана с тем, что на вычислении каждого последующего значения нам приходится вычислять значения двух предыдущих (уже раньше вычисленных) **заново**.

Но вычисленные ранее значения мы можем кэшировать в памяти переменных. Сделаем это в технике функционального программирования (всё тот же каталог `compare/fibo`), добавив туда точное измерение времени:

fibom.go

```
package main
import ("os"; "strconv"; "fmt"; "time")

func Memoize(mf Memoized) Memoized {
    cache := make(map[int]int)
    return func(key int) int {
        if val, found := cache[key]; found {
            return val
        }
        temp := mf(key)
        cache[key] = temp
        return temp
    }
}

type Memoized func(int) int
var fibMem Memoized

func FibMemoized(n int) int {
    n += 1
    fibMem = Memoize(func(n int) int {
        if n == 0 || n == 1 {
            return n
        }
        return fibMem(n - 2) + fibMem(n - 1)
    })
    return fibMem(n)
}

func main() {
    n, _ := strconv.Atoi(os.Args[1])
    t := time.Now()
    fmt.Printf("%d [%v]\n", FibMemoized(n), time.Since(t))
}
```

Здесь функция `Memoize()` получает в качестве параметра **функцию** типа `Memoized`, дополняет таблицу и возвращает модифицированную **функцию**. Это также рекурсивная функция. Но сравним скорость её выполнения с исходной простой рекурсивной функцией (чуть модифицировав её, добавив точное измерение времени):

fibos.go

```
package main
import ("os"; "strconv"; "fmt"; "time")
```

```

func fib(n int) int {
    if n < 2 { return 1
    } else { return fib(n-1) + fib(n-2) }
}

func main() {
    n, _ := strconv.Atoi(os.Args[1])
    t := time.Now()
    fmt.Printf("%d [%v]\n", fib(n), time.Since(t))
}

```

Сравниваем:

```

$ ./fibos 15
987 [10.285µs]

$ ./fibom 15
987 [23.747µs]

$ ./fibos 25
121393 [1.130834ms]

$ ./fibom 25
121393 [39.568µs]

$ ./fibos 40
165580141 [899.897626ms]

$ ./fibom 40
165580141 [66.951µs]

$ ./fibos 45
1836311903 [9.818835496s]

$ ./fibom 45
1836311903 [68.785µs]

```

И далее... При таких значениях время прямого рекурсивного вычисления возрастает так, что дожидаться его бессмысленно:

```

$ ./fibom 50
20365011074 [51.876µs]

$ ./fibom 60
2504730781961 [59.945µs]

$ ./fibom 70
308061521170129 [105.399µs]

```

На что обращаем внимание? На то, что в этом случае радикальное ускорение достигнуто не за счёт параллельности и эффективного использования многих процессоров, а просто за счёт разумной реструктуризации алгоритма вычислений.

Чистые функции

Ещё один элемент функционального программирования — это чистые функции. Как уже говорилось, чистые функции это те, которые возвращают значения, которые связаны только с **аргументами**, приходящими на вход, и не влияющие на глобальное состояние (значения переменных).

Говоря о чистых функциях, часто употребляют термин немутабельность: концепция, которая позволяет создавать в результате любых действий неизменяемые значения, такие

как строки или числа, или их комбинации. Это позволяет избежать различных проблем, связанных с изменяемыми значениями, например, состоянием гонок в параллельных ветвях выполнения:

Например:

nomut.go

```
package main

import "fmt"

type Point struct {
    x, y int
}

func (p Point) Add(q Point) Point {
    return Point{p.x + q.x, p.y + q.y}
}

func main() {
    p := Point{1, 2}
    q := Point{3, 4}
    r := p.Add(p).Add(q).Add(p)
    fmt.Println(r.x, r.y)
}
```

Сколько бы, и какой глубин не происходили бы функциональные вызовы — никаких изменений промежуточных значений не происходит.

```
$ go fmt nomut.go
nomut.go
$ go run nomut.go
6 10
```

Это становится особенно важным когда функциональные фрагменты кода выполняются параллельно во многих (десятках и сотнях) независимых горутин!

Источники информации

[1] Функциональная парадигма на Go: основные техники, 18 ноября 2019 —

<https://habr.com/ru/company/otus/blog/476346/>

[2] Lex Sheehan, Learning Functional Programming in Go

Copyright ©2017 Packt Publishing, 973 стр. —

https://vk.com/wall-54530371_223302?ysclid=l280vn3lp8

Сети, Интернет и WEB

TCP сервер и клиент

Go, как язык позднего времени, изначально предназначен для создания сетевых инструментов, и изначально включал в свою пакетную инфраструктуру развитые сетевые инструменты. Чтобы не нагромождать детали и не усложнять напрасно, мы сделаем простейший эхо-сервер, который будет принимать сообщения от клиента и ретранслировать принятое обратно клиенту (такой шаблон может быть успешной базой для последующего наполнения его требуемой функциональностью клиент-сервера).

Итак (каталог архива `net/client-server`) ... простой ретранслирующий эхо-сервер:

echo_serv.go

```
package main
import "net"
import "fmt"
import "bufio"

func main() {
    ln, _ := net.Listen("tcp", ":51500") // Устанавливаем прослушивание порта
    for {
        fmt.Println("waiting on port 51500")
        conn, _ := ln.Accept()          // Открываем серверный порт
        for {                             // Запускаем цикл прослушивания
            // Будем прослушивать все сообщения разделенные \n
            message, err := bufio.NewReader(conn).ReadString('\n')
            if err != nil {                // Клиент разорвал коннект
                conn.Close()
                break
            }
            conn.Write([]byte(message))    // Отправить обратно клиенту
        }
    }
}
```

В принципе, для TCP-сервера можно не писать соответствующего ему клиента (мы это сделаем, но чуть позже), а использовать в качестве клиента существующие Linux сетевые приложения. И такой способ тестирования даже предпочтительнее, потому что он подтверждает совместимость нашего продукта со стандартами и традициями POSIX.

```
$ go build -o echo_serv echo_serv.go
```

```
$ ls -l echo_serv*
```

```
-rwxrwxr-x 1 olej olej 2758757 янв 12 15:56 echo_serv
-rw-rw-r-- 1 olej olej    812 дек 17 18:26 echo_serv.go
```

Тестируем (TCP порт 51500 мы для этих примеров выбрали произвольно) — 3 различных тестера (естественно, что сервер при этом должен быть запущен, а приложения клиентов можно запускать как с другого терминала того же компьютера — как это и показано в примерах — так и с другого хоста локальной сети):

```
$ nc 127.0.0.1 51500
```

```
123
123
3456
3456
^C
```

```
$ socat tcp4:127.0.0.1:51500 STDOUT
```

```
123
123
0987

286
```

0987

^C

\$ telnet 127.0.0.1 51500

Trying 127.0.0.1...

Connected to 127.0.0.1.

Escape character is '^]'.

123

123

4568

4568

^]

telnet> quit

Connection closed.

Что тестовые клиенты серверу отсылают (наш ввод с терминала) — то он им и возвращает!

Для полноты картины и комплементарный серверу TCP-клиент:

echo cli.go

```
package main
import "net"
import "fmt"
import "bufio"
import "os"
import "io"
import "strconv"

func main() {
    port := "51500"
    if len(os.Args) > 1 {
        _, err := strconv.Atoi(os.Args[1])
        if err != nil {
            fmt.Println("Ошибочный порт")
            return
        } else {
            port = os.Args[1]
        }
    }
    fmt.Println("connect to port " + port)
    conn, err := net.Dial("tcp", "127.0.0.1:" + port) // Подключаемся к сокету
    if err != nil {
        fmt.Println("no server available"); return
    }
    for {
        reader := bufio.NewReader(os.Stdin) // Чтение данных из stdin
        fmt.Print("> ")
        text, err := reader.ReadString('\n')
        if err != nil {
            if err != io.EOF { fmt.Print(err) }
            fmt.Println()
            return
        }
        if len(text) == 1 { break }
        fmt.Fprintf(conn, text + "\n") // Отправляем в socket
        message, _ := bufio.NewReader(conn).ReadString('\n') // Чтение ответа
        fmt.Print(" " + message)
    }
}
```

\$ go build -o echo_cli echo_cli.go

\$ ls -l echo_cli*

```
-rwxrwxr-x 1 olej olej 2764929 янв 12 15:49 echo_cli
-rw-rw-r-- 1 olej olej 1122 дек 20 11:11 echo_cli.go
```

Тестируем только-что собранный клиент:

```
$ ./echo_cli
connect to port 51500
> 123
123
> 2345
2345
> русская строка
русская строка
> ^C
```

Здесь же мы, попутно, проверяем то, как мультязычные строки UNICODE передаются по IP сети, и как с ними работает код Go³⁶.

Сервер же наш, как мы его построили, будет только приглашением (с напоминанием номера TCP порта) подтверждать подключение очередного клиента:

```
$ ./echo_serv
waiting on port 51500
waiting on port 51500
waiting on port 51500
waiting on port 51500
waiting on port 51500
^C
```

Выше был показан **последовательный** TCP-сервер, который обслуживает только одного клиента, следующий клиент обязан будет ожидать завершения работы с предыдущим. По самой поразительной чертой языка Go (и его исполнимой системы!) является то, с какой лёгкостью последовательный код превращается в **параллельный**, могущий одновременно и параллельно обслуживать многих клиентов:

echo_servm.go

```
package main
import "net"
import "fmt"
import "bufio"

func main() {
    ln, _ := net.Listen("tcp", ":51501") // Устанавливаем прослушивание порта
    for {                                // Цикл прослушивания
        fmt.Println("waiting on port 51501")
        conn, _ := ln.Accept()          // Открываем серверный порт
        go func(cn net.Conn) {
            defer func() { cn.Close() }()
            for {                          // Запускаем цикл подключений
                // Будем прослушивать все сообщения разделенные \n
                message, err := bufio.NewReader(cn).ReadString('\n')
                if err != nil {            // Клиент разорвал коннект
                    return
                }
                cn.Write([]byte(message))  // Отправить обратно клиенту
            }
        }(conn)
    }
}
```

³⁶ Здесь же отметим, вскользь, что клиент `telnet`, как один из самых старых сетевых инструментов Linux, не обладает такой способностью, и при передаче/приёме искажает или «проглатывает» некоторые кириллические буквы в UTF-8, воспринимая их байты как элементы управляющих последовательностей в потоке; `telnet` будет безупречно работать только с ASCII символами.


```
$ go build -o echo_servm echo_servm.go
$ ls -l echo_servm*
-rwxrwxr-x 1 olej olej 2759199 янв 12 16:38 echo_servm
-rw-rw-r-- 1 olej olej 944 дек 17 18:26 echo_servm.go
```

Теперь все наши клиенты работают параллельно (на одном и том же TCP-порту), никак не взаимодействуя и не интерферируя по данным (специально по каждому из одновременно подключенных клиентов сделано по два ввода-подтверждения, чтобы убедиться, что каждый из них работает со своим экземпляром сервера):

```
$ ./echo_cli 51501
connect to port 51501
> A: 123 456
    A: 123 456
> A: русская строка
    A: русская строка
>
```

```
$ nc 127.0.0.1 51501
B: asdf ghjk
B: asdf ghjk
B: 12 34 567 890
B: 12 34 567 890
^C
```

```
$ socat tcp4:127.0.0.1:51501 STDOUT
C: русская строка
C: русская строка
C: one more string
C: one more string
^C
```

```
$ telnet 127.0.0.1 51501
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
D: latin string
D: latin string
D: 9876
D: 9876
^]
telnet> quit
Connection closed.
```

Тривиальный WEB сервер

Пакет `net/http` обслуживает HTTP-запросы, используя объект (переменную) любого типа, который реализует интерфейс `http.Handler`:

```
package http

type Handler interface {
    ServeHTTP(w ResponseWriter, r *Request)
}
```

В показанном примере (каталог `net/http.server`) новый, определённый нами, тип `Hello` реализует интерфейс `http.Handler`:

http.go

```
package main
```

```
import (
    "fmt"
```

```

    "net/http"
)

type Hello struct{}

func (h Hello) ServeHTTP(
    w http.ResponseWriter,
    r *http.Request) {
    fmt.Fprint(w, "Hello!")
}

func main() {
    var h Hello
    http.ListenAndServe("localhost:4000", h)
}

$ ./http
...
^C

```

Для наблюдения эффекта выполнения нашей программы `./http` (которая находится после запуска в блокированном состоянии) заходим браузером по адресу `http://localhost:4000/`:

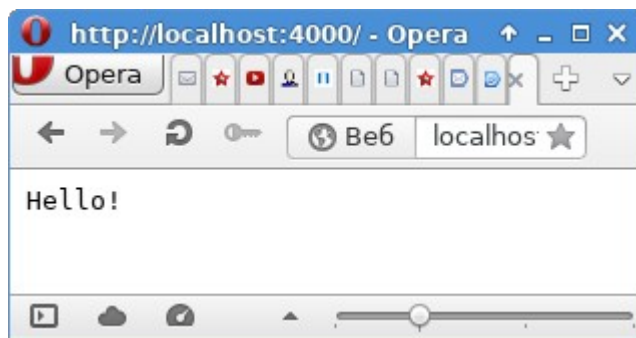


Рис. 3.2. Локальный веб-сервер

Это тривиальный пример... Однако на использовании пакета `net/http` в Go легко строят GUI/WEB интерфейсы к локальным (и не только локальным) приложениям.

Источники информации

[1] `net`. Documentation — пакет сетевых операций стандартной библиотеки — <https://pkg.go.dev/net>

Скоростные и другие сравнения языков

*Всё дело в естестве - если стриж
полетит медленнее, он упадёт.
Павел Крусанов «О людях и ангелах»*

Сравнение скорости выполнения эквивалентных программных проектов, но реализованных на разных языках программирования — занятие неблагодарное: результат будет зависеть от **класса** сравниваемых задач, уровня машинной **оптимизации**, допускаемого компилятором-интерпретатором, ... и ещё от множества других факторов. Но можно и необходимо ориентироваться в численном различии **порядков скорости** выполнения, для того, чтобы выбирать адекватный инструментарий для реализации того или иного программного проекта. Поскольку мы хотим иметь оценки в **порядке** скорости (различия в единицы, десятки, сотни, или тысячи раз), то для сравнений годится почти любая формулировка вычислительной задачи.

Алгоритмические задачи для сравнения

На алгоритмических (вычислительных) задачах нам предстоит реализовать линейку идентичных приложений на разных языках, на которых можно провести такого сравнения. Задачи мы хотели бы использовать различных сортов (вычислительные и не только), в **простейших формах** для формулирования и понимания.

Для наших целей очень подходят задачи, которые имела бы очень **высокую степень роста** вычислительной сложности от размерности (например экспоненциальную) $O(n)$, чтобы можно было в самых широких пределах изменять интегральную потребность в вычислительных операциях. Иногда спрашивают: зачем такое условие? Всё очень просто и прозаично:

- Эти тесты (и подобные им) могут выполняться разными людьми на разном оборудовании, по производительности отличающемся в тысячи и даже более раз...
- Но выполняться они, для сравнений, должны в разумный интервал времени — в несколько секунд, не сотые доли секунды (когда невозможно интерпретировать куда они истрачены) и не десятки или сотни секунд, которые нужно выждать для каждого прогона.
- Хотелось бы иметь возможность параметризовать тесты, так, чтобы их не приходилось перекомпилировать под каждый экземпляр оборудования.
- Но кроме всего прочего, параметризация (и даже перекомпиляция) на задачах низкой степени роста вычислительной сложности (эффективных алгоритмах) может требуемую выводить размерность (n) задачи за пределы фиксированной ёмкости элементов данных, таких как `int32`, `int64`, ...

Все задачи (примеры кода и журнал результатов) этого раздела находятся в подкаталогах каталога `compare` архива — каждая задача в своём подкаталоге.

Но скоростные оценки — это только попутная цель этого раздела изложения, здесь мы на конкретных задачах смотрим аналогичные (в меру возможности) сравнительные реализации на различных языках.

Некоторые известные алгоритмы

*Программированию обычно учат на
примерах.
Niklaus Wirth*

Сравнивать мы станем реализации на Go с реализациями на C и C++ эквивалентных кодов, для C/C++ вариантов компиляция будет делаться с помощью компиляторов GCC и

Clang. Уровень оптимизации всех компиляторов будет, по возможности, сброшен в минимальное значение, потому что эффективность генерируемого кода — это вопрос компиляции с языка, а вопрос оптимизации генерированного кода — это вопрос умений конкретного компилятора (да ещё и радикально зависящий от версии его реализации).

Подготовку тестовых задач будем вести в таком виде, чтобы запуск команд на хронометраж (пусть и грубый) мы могли делать **консольными командами** вида:

```
$ time nice -9 <команда> <размерность>
```

- хронометраж выполняется системной командой `time` (не будем вмешиваться в процесс временных измерений);
- команда выполняется от `root`, чтобы позволить повысить приоритет (`nice -9`) задачи выше нормального, снизить дисперсию результатов;
- числовой параметр определяет размерность задачи, объём вычислений нарастает в зависимости от него.

Но при относительно большой размерности (время выполнения в несколько секунд), на не сильно загруженном быстром процессоре с несколькими ядрами (SMP), достаточно адекватные оценки получаются и простым:

```
$ time <команда> <размерность>
```

Вышесказанное относительно хронометража относится к тем случаям далее, когда мы сравниваем эквивалентные реализации на **разных** языках и с разными компиляторами — для того чтобы мы не вносили от себя в измерения специфику измерения временных интервалов в разных средах. Когда же мы будем сравнивать варианты на Go, будем пользоваться внутренней службой времени Go, **пакетом** `time`, точность, а главное дискретность которого гораздо лучше.

По каждой реализации будут показаны один-два характерный результата, но на самом деле прогонов нужно делалось несколько (серией, до 10 и более), показанный же в тексте — это средний, самый устойчивый вариант (при измерении **временных интервалов** повторяемость чисел всегда является проблемой). Не используем результаты 1-го запуска в серии, чтобы обеспечить для разных запусков серии идентичные условия кэширования.

Вариант кода C/C++ будет компилироваться, для сравнения, двумя компиляторами: стандартным GCC и относительно новым, популярным и динамично развивающимся Clang из проекта LLVM. Clang вам, возможно, придётся доустановить в системе дополнительно:

```
$ sudo apt install clang
```

```
...
```

```
Распаковывается clang (1:10.0-50~exp1) ...
```

```
Настраивается пакет clang (1:10.0-50~exp1) ...
```

```
Обрабатываются триггеры для man-db (2.9.1-1) ...
```

Показанные ниже результаты получены на реализациях:

```
$ gcc --version
```

```
gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
```

```
...
```

```
$ clang --version
```

```
clang version 10.0.0-4ubuntu1
```

```
Target: x86_64-pc-linux-gnu
```

```
Thread model: posix
```

```
...
```

```
$ go version
```

```
go version go1.13.8 linux/amd64
```

Числа Фибоначчи

Для грубых оценок вполне пригодна задача рекурсивного вычисления чисел Фибоначчи. Эту задачу часто используют для подобных оценок. Мы на протяжении предыдущего текста несколько раз уже возвращались к реализациям этой функции — настолько она вариативная и показательная. Эта функция крайне проста, её формулировка будет ещё раз показана в изложении кода на языке C.

Примечание (для дотошной публики): Существуют 2 определения последовательности чисел Фибоначчи: а). $F_1=0, F_2=1, F_3=1, F_N=F_{N-1}+F_{N-2}$ и б). $F_1=1, F_2=1, F_N=F_{N-1}+F_{N-2}$. Как легко видеть, эти последовательности сдвинуты на 1 член, так что не стоит ломать копья по этому поводу: можно использовать любую форму. Мы будем использовать 2-ю (выбор не имеет значения, он только должен быть одинаков для всех сравниваемых вариантов кодов).

Существуют более-мене **эффективные** алгоритмы вычисления последовательности чисел Фибоначчи (циклические, слева направо). Мы же сознательно будем использовать самую **неэффективную** рекурсивную реализацию (справа налево), именно в той форме, как выражения записаны выше. При таком алгоритме задача как-раз удовлетворяет требованию высокой степени роста вычислительной сложности, о чём упоминалось выше.

Реализации этой задачи размещены в каталоге `compare/fibo` архива примеров.

Реализация задачи на языке C:

fibo.c.c

```
#include <stdio.h>
#include <stdlib.h>

unsigned long fib(int n) {
    return n < 2 ? 1 : fib(n - 1) + fib(n - 2);
}

int main(int argc, char **argv) {
    unsigned long num = atoi(argv[1]);
    printf("%ld\n", fib(num));
    return 0;
}
```

Реализация на языке C++:

fibo.cc.cc

```
#include <iostream>
#include <cstdlib>
using namespace std;

unsigned long fib( int n ) {
    return n < 2 ? 1 : fib(n - 1) + fib(n - 2);
}

int main(int argc, char **argv) {
    unsigned long num = atoi(argv[1]);
    cout << fib(num) << endl;
    return 0;
}
```

Из этого единого кода будет создано 2 приложения — компиляцией GCC и компиляцией Clang.

Сравнительная реализация того же теста на языке Go:

fibo.go.go

```
package main
import ("os"; "strconv")
```

```

func fib(n int) int {
    if n < 2 { return 1
    } else { return fib(n-1) + fib(n-2)
    }
}

func main() {
    n, _ := strconv.Atoi(os.Args[1])
    println(fib(n))
}

```

Из этого кода также будет произведено 2 приложения — компиляцией `gccgo` (из реализации GCC) и компиляцией `go` (из проекта GoLang).

В итоге, выполнение сценария сборки (`Makefile`) выглядит так:

```

$ make
gcc -O0 fibo_c.c -o fibo_c
g++ -O0 fibo_cc.cc -o fibo_cc
clang++ -O0 fibo_cc.cc -o fibo_cl
gccgo fibo_go.go -g -O0 -o fibo_go
go build -o fibo_gc -compiler gc fibo_go.go

```

Результаты сборки (обращаем внимание на размеры файлов):

```

$ ls -l | grep rwx
-rwxrwxr-x. 1 Olej Olej      8632 авг 12 17:33 fibo_c
-rwxrwxr-x. 1 Olej Olej     9208 авг 12 17:33 fibo_cc
-rwxrwxr-x. 1 Olej Olej     9212 авг 12 17:33 fibo_cl
-rwxrwxr-x. 1 Olej Olej  2245608 авг 12 17:33 fibo_gc
-rwxrwxr-x. 1 Olej Olej    28066 авг 12 17:33 fibo_go

```

Вот такие детали тоже любопытны:

```

$ file fibo_go
fibo_go: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked
(uses shared libs), for GNU/Linux 2.6.32,
BuildID[sha1]=76eebf02004cdbc2978a40c396c15d234df6c037, not stripped

```

```

$ ldd fibo_go
linux-vdso.so.1 => (0x00007ffff1c1fe000)
libgo.so.4 => /lib64/libgo.so.4 (0x00007fd3a95c2000)
libm.so.6 => /lib64/libm.so.6 (0x0000003494200000)
libgcc_s.so.1 => /lib64/libgcc_s.so.1 (0x0000003ad8a00000)
libc.so.6 => /lib64/libc.so.6 (0x0000003493200000)
/lib64/ld-linux-x86-64.so.2 (0x0000003492a00000)
libpthread.so.0 => /lib64/libpthread.so.0 (0x0000003493600000)

```

```

$ file fibo_gc
fibo_gc: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, not
stripped

```

```

$ ldd fibo_gc
      не является динамическим исполняемым файлом

```

Прделаем сравнительное выполнение 5-ти полученных примеров:

```

$ time ./fibo_c 42
433494437
real    0m2.026s
user    0m2.020s
sys     0m0.000s

```

```

$ time ./fibo_cc 42
433494437

```

```

real    0m2.181s
user    0m2.178s
sys     0m0.000s

$ time ./fibonacci 42
433494437
real    0m2.921s
user    0m2.914s
sys     0m0.006s

$ time ./fibonacci_gc 42
433494437
real    0m2.564s
user    0m2.550s
sys     0m0.005s

$ time ./fibonacci_cl 42
433494437
real    0m2.314s
user    0m2.311s
sys     0m0.000s

```

Это практически везде одна и та же цифра в пределах статистической погрешности, которая для оценки временных интервалов всегда велика. И этим мы убеждаемся, что такого рода вычислений язык Go своей реализацией и семантикой нисколько не ухудшает условий компиляции, и не уступает эквивалентным программам на C и C++, которые являются самыми быстрыми реализациями из всех языков программирования, предлагаемых в Linux.

Пузырьковая сортировка

Сортировки также являются высокочисленными вычислительными алгоритмами. Пузырьковая сортировка (последовательной перестановкой соседних элементов) является самым неэффективным алгоритмом (степень роста $O(n^2)$), применяется только в учебных заданиях, для практики существуют быстрые рекурсивные алгоритмы. Но для наших целей оценивания именно низкая эффективность делает привлекательной пузырьковую сортировку.

В каждой из сред есть свои готовые инструменты сортировки — это очень уж востребованная потребность — в C++ это `<algorithm>`, в GoLang это отдельный пакет `sort` и т.д. Но наша цель — только сравнение, а поэтому мы не станем их использовать и создадим простенькие «ручные» прототипы.

Ниже показаны 3 файла реализации (в архиве каталог `compare/sort`) на языках C, C++ и Go. Но из них произведено 5 исполнимых программ: компилируя C вариант посредством GCC и Clang, а Go вариант — с помощью `gccgo` и компилятора проекта GoLang. Вот Makefile сценарий сборки:

```

TASK = sort_c sort_cc sort_cl sort_go sort_gc
all: $(TASK)
%: %.c
    gcc -Wall -O0 $< -o $@
%: %.cc
    g++ -Wall -O0 $< -o $@
%: %.go
    gccgo -Wall $< -g -O0 -o $@
sort_cl: sort_c.c
    clang -O0 $< -o $@
sort_gc: sort_go.go
    go build -o $@ -compiler gc $<
clean:
    rm -f $(TASK)

```

Разноязыкие варианты специально «подогнаны» так, чтобы они были подобны в исполнении. Итак:

Реализация задачи на языке C:

sort.c.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

typedef long long data_t;

data_t sort(data_t arr[], data_t num) {
    data_t i, j, k, m = 0;
    for(i = 0; i < num; i++)
        for(j = 0; j < num - 1; j++)
            if(arr[j] > arr[j + 1]) {
                k = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = k;
                m++;
            }
    return m;
}

void show(data_t arr[], data_t num) { // отладка-контроль
    data_t i = 0;
    printf("[");
    for(; i < num; i++) printf("%llu ", arr[i]);
    printf("]\n");
}

int main(int argc, char **argv) {
    if(argc != 2)
        printf("usage: %s [-]<number>\n", argv[0]), exit(1);
    data_t size = atoll(argv[1]), i, n, *vect;
    char debug = 0;
    if(size < 0) size = -size, debug = 1;
    if(!(vect = (data_t*)calloc(size, sizeof(data_t))))
        perror("allocate"), exit(1);
    for(i = 0; i < size; i++) vect[i] = size - i;
    if(debug) show(vect, size);
    n = sort(vect, size);
    printf("%llu\n", n);
    if(debug) show(vect, size);
    free(vect);
    return 0;
}
```

Реализация задачи на языке C++:

sort.cc.cc

```
#include <stdlib.h>
#include <iostream>
#include <vector>

using namespace std;

typedef long long data_t;

data_t sort(vector<data_t>& v) {
    data_t m = 0;
    for(vector<data_t>::iterator i = v.begin(); i != v.end(); i++)
```



```

        for(vector<data_t>::iterator j = v.begin(); j + 1 != v.end(); j++)
            if(*j > *(j + 1)) {
                data_t k = *j;
                *j = *(j + 1);
                *(j + 1) = k;
                m++;
            }
        return m;
    }

void show(vector<data_t> v) { // отладка-контроль
    cout << "[";
    vector<data_t>::iterator i = v.begin();
    while(i != v.end()) cout << *i++ << " ";
    cout << "]" << endl;
}

int main(int argc, char **argv) {
    if(argc != 2)
        cout << "usage: " << argv[0] << " [-]<number>" << endl, exit(1);
    bool debug = false;
    data_t size = atoll(argv[1]), n;
    if(size < 0) size = -size, debug = true;
    vector<data_t> vect = vector<data_t>(size);
    n = size;
    for(vector<data_t>::iterator i = vect.begin(); i != vect.end(); i++)
        *i = n--;
    if(debug) show(vect);
    n = sort(vect);
    cout << n << endl;
    if(debug) show(vect);
    return 0;
}

```

Реализация задачи на языке Go:

sort.go.go

```

package main

import (
    "fmt"
    "os"
    "strconv"
)

type data_t int64

func sort(p []data_t) data_t {
    var m data_t = 0
    for i := range p {
        i = i
        for j := range p {
            if j == len(p)-1 {
                break
            }
            if p[j] > p[j+1] {
                p[j], p[j+1] = p[j+1], p[j]
                m++
            }
        }
    }
    return m
}

```

```

func main() {
    show := func(p []data_t) { // диагностика для среза
        fmt.Println(p)
    }
    if len(os.Args) != 2 {
        fmt.Printf("usage: %v [-]<number>\n", os.Args[0])
        return
    }
    var длина data_t
    n, _ := strconv.Atoi(os.Args[1])
    длина = data_t(n)
    debug := false
    if длина < 0 {
        длина = -длина
        debug = true
    }
    срез := make([]data_t, длина) // len( b ) == 10, cap( b ) == 10
    for i := range срез {
        срез[i] = data_t(len(срез) - i)
    }
    if debug {
        show(срез)
    }
    var := sort(срез)
    fmt.Println(var)
    if debug {
        show(срез)
    }
}

```

Численным параметром командной строки каждой задачи будет длина сортируемой последовательности чисел. Если это число указано со знаком минус, то будет выводиться индикация исходной и отсортированной последовательности (для отладки и контроля):

```

$ ./sort_go
usage: ./sort_go [-]<number>

$ ./sort_go -20
[20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1]
190
[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20]

```

Исходные последовательности подготовлены так, чтобы условия сортировки были наихудшими — в обратной расстановке. Число-результат выполнения программ — количество потребовавшихся перестановок для данной длины последовательности (для контроля идентичности):

```

$ time ./sort_c 20000
199990000
real    0m1.500s
user    0m1.494s
sys     0m0.000s

$ time ./sort_cc 20000
199990000
real    0m19.007s
user    0m18.946s
sys     0m0.009s

$ time ./sort_cl 20000
199990000
real    0m1.451s

```

```

user      0m1.447s
sys       0m0.001s

$ time ./sort_go 20000
199990000
real      0m2.303s
user      0m2.295s
sys       0m0.006s

$ time ./sort_gc 20000
199990000
real      0m1.141s
user      0m1.140s
sys       0m0.000s

```

В итоге:

- То, что C++ отстаёт от остальных участников практически **на порядок** — это указывает только на нечестность такого сравнения: код на C++ написан с использованием средств STL, шаблонного типа `vector<long long>` и итераторов, в то время, как все остальные варианты — прямой адресацией элементов массива. Такая реализация в C++ сделана специально, чтобы показать плюсы (гибкость) и минусы STL механизмов.
- Удивила компиляция кода C компилятором Clang: результаты лучше, чем у GCC (по крайней мере, без вовлечения оптимизации).
- Язык Go (в варианте GCC) если и уступает C по скорости, то только порядка 50%, при том предоставляя гибкость и выразительную мощность соизмеримую с C++ с использованием STL (см. код).
- Окончательно удивил результат, показанный Go из проекта GoLang (родной проект развития Go) — такой код оказался **быстрее**, чем код C компилированный GCC (базовый компилятор для проектов GNU до последнего времени)!

Ханойская башня

Хорошо известная задача, которую часто приводят в пример плохо формализуемым алгоритмам, и простоты их рекурсивного описания. Утверждается, что эту задачу (для размерности $N=10$ колец) решают уже несколько столетий монахи тибетских монастырей, и когда они её разрешат, то наступит конец света, Армагедон в европейской нотации (но это, скорее всего, красивая легенда из среды IT-йяпи):



Рис. 3.3. Ханойская башня

- Имеется 3 нумерованных стержня (1, 2, 3), на один из которых нанизаны N колец, на манер детской пирамидки, так что кольца, снизу-вверх, уложены в порядке убывания диаметра ...

- Нужно последовательно переложить всю пирамидку со стержня 1 на стержень 3, используя как промежуточный стержень 2 при условиях: а). перекладываем с любого стержня за раз только одно верхнее кольцо; б). это кольцо можно перекладывать либо на пустой стержень, либо на стержень только поверх лежащего на нём кольца большего размера (можно класть только меньшее поверх большего).

```

=====>
- | -   |   |
--|--   |   |
---|---  |   |
  1     2   3

```

- Исходное состояние задачи для $N = 3$ показано на схематическом рисунке выше. Целью является перенесение всей пирамидки со стержня 1 на стержень 3.

И вот как это делает в 7 переносов колец задача, которую мы напишем далее (как и в предшествующем примере, параметр запуска определяет размерность задачи, а когда он указан со знаком минус, то выводится промежуточная контрольно-отладочная информация):

```

$ ./hanoi_c -3
размер пирамиды: n=3
1 => 3,   1 => 2,   3 => 2,   1 => 3,   2 => 1,
2 => 3,   1 => 3,
число перемещений 7

```

Реализация задачи на языке C:

hanoi_c.c

```

#include <stdio.h>
#include <stdlib.h>

char debug = 0;
ulong nopr = 0;

void put(int from, int to) {
    ++nopr;
    if(!debug) return;
    printf("%d => %d,   ", from, to);
    if(0 == (nopr % 5)) printf("\n");
}

int temp(int from, int to) {    // промежуточная позиция
    int i = 1;
    for(; i <= 3; i++)
        if(i != from && i != to)
            return i;
    return 0;                    // ошибка
}

void move(int from, int to, int n) {
    if(n > 1) move(from, temp(from, to), n - 1);
    put(from, to);                // единичное перемещение
    if(n > 1) move(temp(from, to), to, n - 1);
}

int main(int argc, char **argv, char **envp) {
    if(argc != 2)
        printf("usage: %s [-]<number>\n", argv[0]), exit(1);
    int size = atoi(argv[1]);    // число переносимых фишек
    if(size < 0) size = -size, debug = 1;
    if(debug) printf("размер пирамиды: n=%d\n", size);
    move(1, 3, size);            // вот и всё решение!
}

```

```

    if(debug && (nopr % 5) != 0) printf("\n");
    printf("число перемещений %ld\n", nopr);
    return 0;
}

```

Как можно видеть, всю работу выполняет рекурсивная функция `move(int from, int to, int n)` - переместить верхнюю под-пирамидку размером `n` колец со штыря номером `from` [1, 2, или 3] на штырь `to` [1, 2, или 3]:

- если требуется переместить только одно верхнее кольцо (`n == 1`), то просто взять его и переложить;

- а вот если `n > 1`, то а). переложить всю верхнюю под-пирамидку размерностью (`n - 1`) на оставшийся свободным промежуточный штырь (не `from` и не `to`), б). переместить одно оставшееся (последнее) кольцо на место назначения `to`, в). после чего опять же всю под-пирамидку размерностью (`n - 1`) с промежуточного штыря также перенести поверх кольца, уложенного на место назначения `to`.

Теперь то же самое, выраженное на языке Go:

hanoi.go

```

package main
import("os"; "strconv")

var debug bool = false
var nopr uint64 = 0

func move(from, to, сколько int) {
    put := func(from, to int) {
        nopr++
        if !debug { return }
        print(from, " => ", to, ", ")
        if 0 == (nopr % 5) { print("\n") }
    }
    temp := func(from, to int) int { // промежуточная позиция
        for i := 1; i <= 3; i++ {
            if i != from && i != to { return i }
        }
        panic(0); // ошибка
    }
    if сколько > 1 { move(from, temp(from, to), сколько - 1) }
    put(from, to) // единичное перемещение
    if сколько > 1 { move(temp(from, to), to, сколько - 1) }
}

func main() {
    if len(os.Args) != 2 {
        print("usage: ", os.Args[0], " [-]<number>\n")
        return
    }
    debug = false
    размер, _ := strconv.Atoi(os.Args[1])
    if размер < 0 {размер, debug = -размер, true}
    if debug { print("размер пирамиды: n=", размер, "\n") }
    move(1, 3, размер) // вот и всё решение!
    if debug && (nopr % 5) != 0 { print("\n") }
    print("число перемещений ", nopr, "\n")
}

```

Скомпилируем всё это (для широты сравнения) различными компиляторами:

```

$ make
gcc -O3 -Wall hanoi_c.c -o hanoi_c

```

```
clang -O3 -xc -Wall hanoy_c.c -o hanoy_cl
gccgo -O3 -Wall hanoy_go.go -g -o hanoy_go
go build -o hanoy_gol -compiler gc hanoy_go.go
```

В результате:

```
$ time ./hanoy_c 30
```

```
число перемещений 1073741823
```

```
real    0m3,991s
```

```
user    0m3,991s
```

```
sys     0m0,000s
```

```
$ time ./hanoy_cl 30
```

```
число перемещений 1073741823
```

```
real    0m3,564s
```

```
user    0m3,560s
```

```
sys     0m0,004s
```

```
$ time ./hanoy_go 30
```

```
число перемещений 1073741823
```

```
real    0m4,577s
```

```
user    0m4,556s
```

```
sys     0m0,032s
```

```
$ time ./hanoy_gol 30
```

```
число перемещений 1073741823
```

```
real    0m8,455s
```

```
user    0m8,442s
```

```
sys     0m0,028s
```

На таком типе задач (при максимальном уровне оптимизации доступном GCC) GoLang реализация на таком объеме вычислений показывает результат только вдвое хуже GCC, что можно считать вполне приемлемым. Отличные результаты показывает и относительно новый компилятор C/C++ динамично развивающегося проекта Clang, немногим даже лучше GCC.

Решето Эратосфена

Ещё один хорошо известный алгоритм: поиск всех простых чисел (меньше N) прореживанием натурального ряда чисел $[1 \dots N]$, который приписывают древнегреческому математику Эратосфену Киренскому. В схематичном описании это выглядит так: нахождение всех простых чисел не больше заданного числа n , нужно выполнить следующие шаги:

1. Выписать подряд все целые числа от двух до n (2, 3, 4, ..., n).
2. Пусть переменная p изначально равна 2 — первому простому числу.
3. Зачеркнуть в списке числа от $p+1$ до n кратные p : $2p$, $3p$, $4p$, ...).
4. Найти первое ещё не вычеркнутое число в списке, большее чем p , и присвоить значению переменной p это число.
5. Повторять шаги 3 и 4, пока p не достигнет n .

Есть некоторые алгоритмические улучшения этой схемы (в коде далее), но мы не будем сосредотачиваться на этих деталях (каталог `compare/erastof`).

На языке C реализация может выглядеть так:

erastof.c

```
#include <stdio.h>
```

```
#include <stdlib.h>
```

```
typedef long long data_t;
```

```
void eratos(char *arr, ulong size) {
```

```

        ulong i, j;
        for(i = 2; i < size; i++)                // цикл по всему массиву от первого простого
числа
            if(1 == arr[i])
                for(j = i + i; j < size; j += i) // вычеркивание всех чисел кратных i
                    arr[j] = 0;
    }

int main(int argc, char **argv) {
    long n;
    char debug = 0;
    if(argc != 2)
        printf("usage: %s [-]<number>\n", argv[0]), exit(1);
    n = atol(argv[1]);                          // максимальное число
    if(n < 0) n = -n, debug = 1;
    ulong k, j;
    char *a = calloc(n + 1, sizeof(char));
    a[0] = a[1] = 0;                             // вычёркиваем "0" и "1"
    for(k = 2; k < n; k++) a[k] = 1;             // остальные размечаем как простые
    eratos(a, n);
    for(k = 0, j = 0; k < n; k++)
        j += (a[k] != 0 ? 1 : 0);
    printf("простых чисел %lu\n", j)
    if(debug) {
#define INLINE 10
        for(k = 0, j = 0; k < n; k++)
            if(1 == a[k]) {
                j++;
                printf("%lu%s", k, (0 == j % INLINE ? "\n" : "\t"));
            }
        if(j % INLINE != 0) printf("\n");
    }
    free(a);
    return 0;
}

```

Эквивалент на языке Go может выглядеть так:

erastof go.go

```

package main

import (
    "os"
    "strconv"
)

func main() {
    type data_t int64
    var cpez []bool
    debug := false
    eratos := func() {
        count := func() data_t {
            var j data_t
            for i := range cpez {
                if cpez[i] {
                    j++
                }
            }
            return j
        }
        for i := range cpez {
            if cpez[i] {
                for j := i + i; j < len(cpez); j += i {

```

```

// вычеркивание всех чисел кратных i
срез[j] = false
    }
}
}
print("простых чисел ", count(), "\n")
}
show := func(s []bool) { // диагностика среза
    const inlin = 10
    var j data_t
    for i := range s {
        if s[i] {
            j++
            print(i, "\t")
            if 0 == (j % inlin) {
                print("\n")
            }
        }
    }
    if j%inlin != 0 {
        print("\n")
    }
}
if len(os.Args) != 2 {
    print("usage: ", os.Args[0], " [-]<number>\n")
    return
}
n, _ := strconv.Atoi(os.Args[1])
var длина data_t = data_t(n)
if длина < 0 {
    длина, debug = -длина, true
}
срез = make([]bool, длина)
for i := range срез {
    if i > 1 {
        срез[i] = true
    }
}
eratos()
if debug {
    show(срез)
}
}

```

Помимо прочего, здесь показана вложенность описаний функций глубиной больше единичной (функция `count()` вложена в функцию `eratos()`, которая, в свою очередь вложена в функцию `main()`). Такую иерархию вложенных описаний можно строить на произвольную глубину. И здесь же показано то, как эти вложенные функции используют глобальные по отношению к их собственным описаниям переменные (описанные вне тела функций): функции `count()` и `eratos()` работают с переменными `срез` и `debug`, описанными на внешнем по отношению к функциям уровне. Это напоминает области видимости имён, как они определены, например, в языках Н.Вирта Pascal и Modula-2, и открывает весьма широкие перспективы использования.

Но вернёмся к сравнению реализаций...

\$ make

```

gcc -O3 -Wall erastof_c.c -o erastof_c
clang -O3 -xc -Wall erastof_c.c -o erastof_cl
gccgo -O3 -Wall erastof_go.go -g -o erastof_go
go build -o erastof_gol -compiler gc erastof_go.go

```

И вот сравнительное выполнение полученных бинарных исполнимых файлов:


```
$ time ./erastof_c 50000000
```

```
простых чисел 3001134
```

```
real    0m1,232s
```

```
user    0m1,196s
```

```
sys     0m0,036s
```

```
$ time ./erastof_cl 50000000
```

```
простых чисел 3001134
```

```
real    0m1,237s
```

```
user    0m1,205s
```

```
sys     0m0,032s
```

```
$ time ./erastof_go 50000000
```

```
простых чисел 3001134
```

```
real    0m1,339s
```

```
user    0m2,453s
```

```
sys     0m0,104s
```

```
$ time ./erastof_gol 50000000
```

```
простых чисел 3001134
```

```
real    0m1,334s
```

```
user    0m1,436s
```

```
sys     0m0,029s
```

На этом классе задач (многократное сканирование массивов) языки (и C и Go) и используемые компиляторы (GCC, Clang, GoLang) показывают практически идентичные цифры производительности (но заметим при этом, что мы здесь из GCC выжимаем максимально возможный уровень оптимизации кода: -O3, без оптимизации это время станет порядка 8.5s).

Многопроцессорные параллельные вычисления

Скорость активации параллельных ветвей

Скорость имеет значение.

Павел Дуров.

Прежде чем рассматривать примеры конкретных задач, зададимся таким интересным вопросом — раньше уже речь шла о 3-х моделях параллелизма, исторически внедрившиеся именно в такой последовательности: 1). параллельные процессы, вызов `fork()`, 2). параллельные потоки ядра `pthread_t`, 3). лёгкие go-рутины (сопрограммы). Попробуем оценить (каталог архива `goroc/gotime`) время активации для N параллельных ветвей в каждой из моделей. И начнём именно с go-рутин, потому что именно это есть основной предмет нашего интереса...

gotim1.go

```
package main
import ("os"; "strconv"; "fmt"; "time")

func sequgo(n int, c chan bool) {
    if n == 0 {
        c <- true
    } else {
        go sequgo(n - 1, c)
        <-c
    }
}

func main() {
    n := 1000
    if len(os.Args) > 1 {
        n, _ = strconv.Atoi(os.Args[1])
    }
    c := make(chan bool)
    t0 := time.Now()
    sequgo(n, c)
    fmt.Printf("[%d] время выполнения: %v\n", n, time.Now().Sub(t0))
}
```

Здесь N параллельных go-рутин (N можно указать параметром команды запуска) запускают друг-друга рекурсивно, по цепочке, и каждая k -я go-рутина, запускающая $(k+1)$ -ю — ожидает её завершения. Активировавшись все N ветвей, они начинают так же, по цепочке, в обратном порядке завершаться.

Можно сделать проще (но как мне кажется, не настолько корректно) — запускать все требуемые N go-рутин из главной программы (одна за одной), и они завершаются сразу после запуска, но главная программа ожидает завершения всех N запущенных ветвей:

gotim2.go

```
package main
import ("os"; "strconv"; "fmt"; "time")

func main() {
    n := 1000
    if len(os.Args) > 1 {
        n, _ = strconv.Atoi(os.Args[1])
    }
    c := make(chan bool)
    t0 := time.Now()
    for i := 0; i < n; i++ {
```

```

        go func() { c <- true }()
    }
    for i := 0; i < n; i++ { <-c }
    fmt.Printf("[%d] время выполнения: %v\n", n, time.Now().Sub(t0))
}

```

Результаты запусков мы посмотрим чуть позже, а сейчас сконструируем некоторые подобию показанной выше модели `gotim2.c`, но относительно запуска параллельных процессов вызовом `fork()` и запуска потоков ядра вызовом `pthread_create()`. Естественно, что эти аналоги будут написаны на C, потому что это нативные механизмы POSIX API.

forktim.c

```

#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/time.h>

/* struct timeval {
    time_t      tv_sec;        // seconds
    suseconds_t tv_usec;      // microseconds
};
*/

int main(int argc, char *argv[]) {
    ulong n = argc > 1 ? atol(argv[1]) : 1000;
    pid_t pid[n];
    struct timeval t0, t1;
    gettimeofday(&t0, NULL);
    for(int i = 0; i < n; i++)
        if((pid[i] = fork()) != 0)
            continue;
        else
            return 0;
    for(int i = 0; i < n; i++)
        waitpid(pid[i], NULL, 0);
    gettimeofday(&t1, NULL);
    double t = (t1.tv_usec - t0.tv_usec) + (t1.tv_sec - t0.tv_sec) * 1000000;
    if(t < 1000)
        printf("время выполнения: %.3fµs\n", t);
    else
        printf("[%ld] время выполнения: %.3fms\n", n, t / 1000.);
    return 0;
}

```

И, наконец, последний вариант, относительно `pthread_t`:

thretim.c

```

#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <pthread.h>

/* struct timeval {
    time_t      tv_sec;        // seconds
    suseconds_t tv_usec;      // microseconds
};
*/

void* tfunc(void* data) {
    pthread_exit(NULL);
}

```

```

}

int main(int argc, char *argv[]) {
    ulong n = argc > 1 ? atol(argv[1]) : 1000;
    pthread_t tid[n];
    struct timeval t0, t1;
    gettimeofday(&t0, NULL);
    for(int i = 0; i < n; i++)
        pthread_create(&tid[i], NULL, tfunc, NULL);
    for(int i = 0; i < n; i++)
        pthread_join(tid[i], NULL);
    gettimeofday(&t1, NULL);
    double t = (t1.tv_usec - t0.tv_usec) + (t1.tv_sec - t0.tv_sec) * 1000000;
    if(t < 1000)
        printf("время выполнения: %.3fµs\n", t);
    else
        printf("[%ld] время выполнения: %.3fms\n", n, t / 1000.);
    return 0;
}

```

Сборка:

```

$ make
go build -o gotim1 -compiler gc gotim1.go
go build -o gotim2 -compiler gc gotim2.go
gcc -Wall forktim.c -lpthread -o forktim
gcc -Wall thretim.c -lpthread -o thretim

```

И теперь можно запустить и сравнить все варианты сразу:

```

$ ./gotim1
[1000] время выполнения: 2.364592ms

$ ./gotim2
[1000] время выполнения: 2.689867ms

$ ./forktim
[1000] время выполнения: 59.344ms

$ ./thretim
[1000] время выполнения: 36.257ms

$ ./gotim1 10
[10] время выполнения: 66.441µs

$ ./gotim2 10
[10] время выполнения: 114.853µs

$ ./forktim 10
время выполнения: 670.000µs

$ ./thretim 10
время выполнения: 508.000µs

$ ./gotim1 10000
[10000] время выполнения: 24.849198ms

$ ./gotim2 10000
[10000] время выполнения: 28.417926ms
$ ./thretim 10000
[10000] время выполнения: 354.031ms

$ ./forktim 10000
[10000] время выполнения: 625.808ms
308

```

Результаты говорят само за себя, впечатляют и не требуют, думаю, комментариев. При существенно больших значениях N (100000 и далее, в зависимости от ресурсов конкретного компьютера) варианты и для `fork()` и для `pthread_create()` просто сходят с ума, а компьютер в целом ведёт себя загадочным образом... вплоть до неустойчивости операционной системы, потому что это механизмы ядра системы. Но ... наблюдаем сюда:

```
$ ./gotim1 1000000
[1000000] время выполнения: 2.242563217s
```

```
$ ./gotim2 1000000
[1000000] время выполнения: 5.352434703s
```

```
$ ./gotim1 10000000
[10000000] время выполнения: 22.5808485s
```

```
$ ./gotim2 10000000
[10000000] время выполнения: 53.33367164s
```

P.S. Тем не менее, чтобы быть совсем честным и корректным в сравнениях, сейчас и на будущее, нужно отметить здесь, для объективности, что в системе Linux установлены ограничения на а). число **процессов** которое может одновременно создать один пользователь и б). общее число **потоков** ядра, которые могут существовать в системе.

Число процессов на пользователя:

```
$ ulimit -u
386094
```

И этот предел может быть программно изменён (обращаем внимание, что все эти, принципиальные для системы, установки мы можем делать только от имени `root`, о чём и напоминает показанный значок приглашения командной строки):

```
# ulimit -u 500000
# ulimit -u
500000
```

Общее число потоков ядра в системе:

```
$ cat /proc/sys/kernel/threads-max
772189
```

Что так же может быть изменено программно:

```
# echo 1100000 > /proc/sys/kernel/threads-max
# cat /proc/sys/kernel/threads-max
1100000
```

Но, кроме этих системных ограничений (для чего они и введены) число и процессов и потоков может ограничиваться конечностью физических ресурсов (в первую очередь оперативной памятью) вашего конкретного компьютера. Нужно учитывать, что каждый поток (и процесс, представленный хотя бы единичным потоком) резервирует блок под стек вызовов (и блок под обработку ожидающих сигналов):

```
$ ulimit --help
...
-i   the maximum number of pending signals
...
-s   the maximum stack size
$ ulimit -i
386094
$ ulimit -s
8192
```

Если же вы хорошо знаете, что ваш код не делает глубоких вызовов (например рекурсивных) и не использует больших объёмов стека, то вот такие установки, например, позволяют выполнять до 100000 pthread_t:

```
# ulimit -s 256
# ulimit -i 120000
```

Гонки

*Безумие — это точное повторение
одного и того же действия. Раз за
разом, в надежде на получение другого
результата.
Альберт Эйнштейн*

Даже квази-параллельное выполнение на одном процессоре создаёт серьёзные проблемы, неизвестные «последовательностным» программистам, и проявляющееся как серьёзные ошибки, с трудом поддающиеся диагностике и локализации. А в многопроцессорной среде их частота их возникновения тысячекратно возрастает. Это проблемы связанные с доступом и модификация значений переменных из различных ветвей одновременно. В общем виде весь класс таких ошибок, а они могут возникать и выявляться по-разному называют как **гонки**, или состояние гонок. Общая природа их в том, что, хотя каждая параллельная ветвь развивается последовательно и детерминировано, в параллельной среде мы не можем сказать событие x в ветви A предшествует событию y в ветви B, происходит с ним одновременно, или позже него.

Частота выявления гонок тем выше, чем легче механизм переключения ветвей. Поэтому в Go в параллельных горутинках они **выявляются** чаще, их проще наблюдать. Но выявляются или не выявляются состояния гонок — от этого скрытых ошибок в коде остаётся ровным счётом столько же. Поэтому в многопроцессорном исполнении места потенциальных возникновения гонок нужно прогнозировать заранее и их предотвращать.

Первым примером (всё в каталоге `goproc/concurent`) мы смоделируем ошибочную ситуацию, которая в параллельной среде выполнения будет создавать состояние гонок:

race.go

```
package main
import ("fmt"; "os"; "strconv"; "time")

func main() {
    ветви := 3
    if len(os.Args) > 1 {
        ветви, _ = strconv.Atoi(os.Args[1])
    }
    fmt.Printf("число ветвей выполнения: %v\n", ветви)
    ch := make(chan bool)
    var counter int64 = 0
    const циклы = 1000000;
    t := time.Now()
    for i := 0; i < ветви; i++ {
        go func() {
            defer func() { ch <- true }()
            for j := 0; j < циклы; j++ {
                counter++
            }
        }()
    }
    for i := 0; i < ветви; i++ {
        <-ch
    }
    delta := (float64(counter) - float64(ветви) * циклы) /
```

```

        float64(ветви) / циклы * 100
        fmt.Printf("счётчик насчитал: %d(%2.1f%s) [%v]\n",
            counter, delta, "%", time.Since(t))
    }

```

Элементарно! Мы инкрементируем счётчик counter 1000000 раз в каждой ветке, общее число которых можно указать параметром запуска процесса в командной строке. Итак:

```

$ ./race 1
число ветвей выполнения: 1
счётчик насчитал: 1000000(0.0%) [2.189858ms]

```

Это было традиционное последовательное выполнение. И всё предсказуемо — всё чисто и быстро. Начинаем параллельное выполнение (это сопрограммы Go и мы уже знаем, что каждая из них будет выполняться на **отдельном** процессоре):

```

$ ./race 2
число ветвей выполнения: 2
счётчик насчитал: 1004757(-49.8%) [2.678934ms]

```

```

$ ./race 4
число ветвей выполнения: 4
счётчик насчитал: 1220800(-69.5%) [3.738663ms]

```

```

$ ./race 10
число ветвей выполнения: 10
счётчик насчитал: 2107868(-78.9%) [10.241183ms]

```

```

$ ./race 20
число ветвей выполнения: 20
счётчик насчитал: 2759453(-86.2%) [17.886448ms]

```

```

$ ./race 40
число ветвей выполнения: 40
счётчик насчитал: 5163669(-87.1%) [27.535044ms]

```

О-ба-на... Ожидаемое значение счётчика радикально отличается (отстаёт) от ожидаемого значения $1000000 \cdot N$, где N — это число процессоров (ветвей). Более того, это отставание тем больше (относительно ожидаемого) чем больше процессоров задействовано в операциях ... и чем больше процессоров, тем медленнее этот ошибочный результат производится.

Причина в том, что даже простейшая операция (инкремент), **модифицирующая** общую переменную, не является **неделимой** (атомарной), хотя нам она и кажется по записи в коде элементарной и неделимой. (В некоторых ранних, лет 40 тому, книгах даже утверждалось, что операция инкремента, особенно в низкоуровневом ассемблерном коде, является именно атомарной, но и это не совсем так.) В нашем примере «пространством» гонок является один единственный оператор **кажущийся** атомарным. В реальной жизни такими фрагментами, которые требуют защиты от совместного доступа, зачастую будет целая группа последовательных операторов языка.

Обратим внимание ещё на одно очень важное обстоятельство: при **последовательных** запусках одного и того же приложения в котором возникают состояния гонок (в одних и тех же условиях и оборудовании, с тем же аргументом), мы будем каждый раз получать **случайные** различающиеся значения. И это **самый верный знак** возникновения гонок — поведение приложения в условиях гонок **не детерминировано!**

```

$ ./race 10
число ветвей выполнения: 10
счётчик насчитал: 2031599(-79.7%) [9.418046ms]

```

```

$ ./race 10
число ветвей выполнения: 10
счётчик насчитал: 2072683(-79.3%) [10.402179ms]

```

```
$ ./race 10
```

```
число ветвей выполнения: 10
```

```
счётчик насчитал: 1961887(-80.4%) [10.089151ms]
```

Прежде чем «лечить» показанный фрагмент кода от гонок, рассмотрим ещё одну крайне необходимую возможность среды GoLang, нужную для локализации места возникновения ситуации гонок: программу можно скомпилировать (собрать) с опцией `-race`:

```
$ go build race.go
```

```
$ ls -l race
```

```
-rwxrwxr-x 1 olej olej 2140502 anp 23 15:59 race
```

Пересоберём:

```
$ go build -race race.go
```

```
$ ls -l race
```

```
-rwxrwxr-x 1 olej olej 2810226 anp 23 16:01 race
```

Убеждаемся (размеры существенно различаются), что это уже другой исполнимый файл. И станем внимательно контролировать время выполнения:

```
$ ./race 1
```

```
число ветвей выполнения: 1
```

```
счётчик насчитал: 1000000(0.0%) [23.212622ms]
```

```
$ ./race 2
```

```
число ветвей выполнения: 2
```

```
=====
```

```
WARNING: DATA RACE
```

```
Read at 0x00c0000140e8 by goroutine 8:
```

```
    main.main.func1()
```

```
        /home/olej/2022/Go/goproc/concurent/race.go:19 +0x75
```

```
Previous write at 0x00c0000140e8 by goroutine 7:
```

```
    main.main.func1()
```

```
        /home/olej/2022/Go/goproc/concurent/race.go:19 +0x8b
```

```
Goroutine 8 (running) created at:
```

```
    main.main()
```

```
        /home/olej/2022/Go/goproc/concurent/race.go:15 +0x1b8
```

```
Goroutine 7 (running) created at:
```

```
    main.main()
```

```
        /home/olej/2022/Go/goproc/concurent/race.go:15 +0x1b8
```

```
=====
```

```
счётчик насчитал: 1780728(-11.0%) [2.661255012s]
```

```
Found 1 data race(s)
```

```
$ ./race 40
```

```
число ветвей выполнения: 40
```

```
=====
```

```
WARNING: DATA RACE
```

```
...
```

```
=====
```

```
счётчик насчитал: 2287076(-94.3%) [24.562311625s]
```

```
Found 1 data race(s)
```

Пока это было последовательное выполнение в одну ветвь — всё нормально. Но как только начинается параллельное выполнение и **возникают** гонки — программа даёт диагностику, даже с указанием строк кода где это состояние гонок возникает.

Наконец, сравним время выполнения выше, с выполнением той же программы, но

312

собранный традиционным простейшим способом (ещё раз пересоберём приложения):

\$./race 1

число ветвей выполнения: 1

счётчик насчитал: 1000000(0.0%) [2.217342ms]

\$./race 2

число ветвей выполнения: 2

счётчик насчитал: 1001812(-49.9%) [2.697164ms]

\$./race 40

число ветвей выполнения: 40

счётчик насчитал: 4972064(-87.6%) [27.599415ms]

Отметим, что даже для чисто последовательного выполнения время выполнения с контролем гонок становится больше в 10 раз, а для 40 процессоров разница возрастает до 1000 раз! Это плата, и так часто бывает, за подключение отладочных средств с диагностикой.

Защита критических данных

Теперь мы готовы перебрать **некоторые** способы защитить критические данные, к которым происходит параллельный доступ, так, чтобы предотвратить возникновение гонок (библиотека пакетов Go предоставляет ещё много разных не упоминающихся элементов синхронизации).

Первый вариант, когда нам нужно защищать отдельные, «штучные» элементы данных, как counter в нашем коде — это атомарные переменные. Операции над ними осуществляется не обычными операциями, а специальными операциями-функциями (пакет sync/atomic), гарантирующими **атомарность** действия:

racea.go

```
package main
import ("fmt"; "os"; "strconv"; "sync/atomic"; "time")

func main() {
    ветви := 3
    if len(os.Args) > 1 {
        ветви, _ = strconv.Atoi(os.Args[1])
    }
    fmt.Printf("число ветвей выполнения: %v\n", ветви)
    ch := make(chan bool)
    var counter int64 = 0
    const циклы = 1000000;
    t := time.Now()
    for i := 0; i < ветви; i++ {
        go func() {
            defer func() { ch <- true }()
            for j := 0; j < циклы; j++ {
                atomic.AddInt64(&counter, 1)
            }
        }()
    }
    for i := 0; i < ветви; i++ {
        <-ch
    }
    delta := (float64(counter) - float64(ветви) * циклы) /
        float64(ветви) / циклы * 100
    fmt.Printf("счётчик насчитал: %d(%2.1f%) [%v]\n",
        counter, delta, "", time.Since(t))
}
```

Я все запуски выполнения буду показывать с контролем времени выполнения, чтобы вы

сами могли оценить временные издержки (а они значительные) каждого из методов, которые нужно нести для корректного использования параллелизма:

\$./racea 1

число ветвей выполнения: 1

счётчик насчитал: 1000000(0.0%) [7.902583ms]

\$./racea 2

число ветвей выполнения: 2

счётчик насчитал: 2000000(0.0%) [40.384083ms]

\$./racea 4

число ветвей выполнения: 4

счётчик насчитал: 4000000(0.0%) [111.776466ms]

\$./racea 10

число ветвей выполнения: 10

счётчик насчитал: 10000000(0.0%) [225.367499ms]

\$./racea 20

число ветвей выполнения: 20

счётчик насчитал: 20000000(0.0%) [375.476059ms]

\$./racea 40

число ветвей выполнения: 40

счётчик насчитал: 40000000(0.0%) [638.591344ms]

Здесь всё безупречно с точки зрения корректности результата, но теперь время затрачиваемое на вычисления возрастает в 15-25 раз — это плата за **синхронизацию** доступа к разделяемой переменной.

Следующий способ ограничивает целую **критическую секцию** кода используя мютекс (хорошо известное понятие из POSIX API, пакет sync):

racem.go

```
package main
import ("fmt"; "os"; "strconv"; "sync"; "time")

func main() {
    ветви := 3
    if len(os.Args) > 1 {
        ветви, _ = strconv.Atoi(os.Args[1])
    }
    fmt.Printf("число ветвей выполнения: %v\n", ветви)
    ch := make(chan bool)
    var mu sync.Mutex
    var counter int64 = 0
    const циклы = 1000000;
    t := time.Now()
    for i := 0; i < ветви; i++ {
        go func() {
            defer func() { ch <- true }()
            for j := 0; j < циклы; j++ {
                mu.Lock()
                counter++
                mu.Unlock()
            }
        }()
    }
    for i := 0; i < ветви; i++ {
        <-ch
    }
    delta := (float64(counter) - float64(ветви) * циклы) /
```

```

        float64(ветви) / циклы * 100
    fmt.Printf("счётчик насчитал: %d(%2.1f%s) [%v]\n",
        counter, delta, "%", time.Since(t))
}

```

\$./racem 1

число ветвей выполнения: 1

счётчик насчитал: 1000000(0.0%) [18.340369ms]

\$./racem 2

число ветвей выполнения: 2

счётчик насчитал: 2000000(0.0%) [70.844342ms]

\$./racem 4

число ветвей выполнения: 4

счётчик насчитал: 4000000(0.0%) [499.137989ms]

\$./racem 10

число ветвей выполнения: 10

счётчик насчитал: 10000000(0.0%) [1.582758437s]

\$./racem 20

число ветвей выполнения: 20

счётчик насчитал: 20000000(0.0%) [2.84947313s]

\$./racem 40

число ветвей выполнения: 40

счётчик насчитал: 40000000(0.0%) [5.376668294s]

Здесь временные накладные расходы заметно возрастают. Более того, время обработки начинает очень сильно расти при большом числе параллельных ветвей ... что, в общем, объяснимо: всё больше горутин начинают «толпиться» перед оператором `Lock()` на мютексе.

И, наконец, вариант семафорами (это самый, пожалуй, известный из всех примитив синхронизации, у всех на слуху, множественно описан в литературе). Это нужно рассмотреть обязательно, хотя бы просто потому что ... семафоров в инструментарии Go просто нет! Но при внимательном рассмотрении встроенный тип **канал**, буферизированный со счётчиком 1, по существу и является искомым **бинарным** семафором:

racem.go

```

package main
import ("fmt"; "os"; "strconv"; "time")

func main() {
    ветви := 3
    if len(os.Args) > 1 {
        ветви, _ = strconv.Atoi(os.Args[1])
    }
    fmt.Printf("число ветвей выполнения: %v\n", ветви)
    ch := make(chan bool)
    var counter int64 = 0
    const циклы = 1000000;
    t := time.Now()
    for i := 0; i < ветви; i++ {
        go func() {
            defer func() { ch <- true }()
            for j := 0; j < циклы; j++ {
                counter++
            }
        }()
    }
}

```

```

    for i := 0; i < ветви; i++ {
        <-ch
    }
    delta := (float64(counter) - float64(ветви) * циклы) /
        float64(ветви) / циклы * 100
    fmt.Printf("счётчик насчитал: %d(%2.1f%s) [%v]\n",
        counter, delta, "%", time.Since(t))
}

```

Здесь, как и следует прогнозировать, всё ещё медленнее, ещё затратнее (это механизм синхронизации более высокого уровня). Но это (каналы) — **универсальный** механизм, базовый для Go, который того стоит:

```

$ ./races 1
число ветвей выполнения: 1
счётчик насчитал: 1000000(0.0%) [67.962886ms]

$ ./races 2
число ветвей выполнения: 2
счётчик насчитал: 2000000(0.0%) [463.224757ms]

$ ./races 4
число ветвей выполнения: 4
счётчик насчитал: 4000000(0.0%) [897.443302ms]

$ ./races 10
число ветвей выполнения: 10
счётчик насчитал: 10000000(0.0%) [2.242805833s]

$ ./races 20
число ветвей выполнения: 20
счётчик насчитал: 20000000(0.0%) [4.686267027s]

$ ./races 40
число ветвей выполнения: 40
счётчик насчитал: 40000000(0.0%) [9.133310498s]

```

В завершение отметим ещё одну важную вещь: точно так же как в случае с **бинарным** семафором, буферизированный канал, с значением (при создании) счётчика >1 , является, по существу, семафором общего вида — **счётным** семафором!

Неблокирующий ввод с клавиатуры

Первый иллюстрирующий пример, который мы рассмотрим, казалось бы и не имеет прямого отношения к параллелизму... но всё же...

Часто хотелось бы иметь, о об этом часто спрашивают «как?», ввод с клавиатуры который не ожидал бы ввода целой строки, а вводил бы посимвольно в темпе нажатия клавиш пользователем. Это классическая ситуация, например, в компьютерных играх, где нажатием клавиш в темпе игры мы управляем ходом этой игры.

Традиционно в терминальной системе UNIX (и Linux как наследника UNIX) всегда подразумевалось два различающихся режима ввода с клавиатуры:

1. **канонический** (режим по умолчанию): построчный ввод (строка заканчивая Enter), блокирующийся до ввода строки, есть эхо-отображение вводимых символов на экран, в потоке ввода обрабатываются специальные символы, например такие как: BackSpace, Ctrl+C, Ctrl+D, стрелки перемещения по строке и другие...
2. **неканонический**: ввод посимвольный, неблокирующий (каждый введенный символ тут же поступает в обработку), эхо-отображение вводимых символов не производится (если это нужно, то это должен делать сам программный код), никакие спецсимволы не имеют никакого особого управляющего назначения, и

обрабатываются в потоке ввода как соответствующие числовые коды (как и все алфавитно-цифровые клавиши).

Не отвлекаясь на то, как устанавливается тот или другой режим, отметим только следующее:

это платформенно зависимое решение, которое актуально только для POSIX / UNIX совместимых операционных систем (Linux, все BSD, Solaris, Minix ...), но абсолютно теряют смысл в других системах (все Windows).

С активным внедрением параллелизма (конкурентности) неблокирующий ввод можно реализовать по-другому, и это будет универсальным решением:

- ввод (опрос) с клавиатуры осуществляется в отдельной ветке (потоке или горутине)...
- при поступлении символа (rune) он передаётся его в обрабатывающий код (вне своей горутины), используя для этого какой-то из механизмов коммуникаций кооперирующихся ветвей (но чаще всего это бывает канал)...
- опрос клавиатуры при этом не обязательно должен быть неблокирующим — отдельная ветвь ввода, в период отсутствия активности пользователя, не блокирует обрабатывающий код.

Вот теперь можно перейти к реализующему примеру (каталог `goproc/errgrp` архива) ... Если пройтись поиском с контекстом «keyboard» по справочной системе GoLang (<https://pkg.go.dev/search?q=keyboard&m=>), то мы получим **несколько десятков** пакетов и модулей во внешних сетевых репозиториях, для операций чтения с клавиатуры ... реализациями от 2012 до 2024 года! Выберем любой подходящий нам пакет, и проделаем весь процесс подготовки сетевого импорта модулей (описывалось ранее, начиная с версии 1.11):

```
$ go mod init errgrp
go: creating new go.mod: module errgrp
go: to add module requirements and sums:
    go mod tidy
```

После чего у нас сгенерируется файл описания модуля:

```
$ ls -o *.mod
-rw-rw-r-- 1 olej  25 фев  7 14:42 go.mod
```

Пока у него такое вот скромное содержание:

```
$ cat go.mod
module errgrp
```

```
go 1.21.3
```

Теперь мы пропишем в код примера строку импорта из сетевого репозитория ... вот полный код примера который мы создадим:

keyb.go

```
package main

import (
    "github.com/eiannone/keyboard"
    "fmt"
)

func main() {
    keyboard.Open()
    defer keyboard.Close()
    chChan := make(chan rune)           // канал ввода с клавиатуры
    go func(chChan chan <- rune) {     // горутина ввода клавиши
        for {
```

```

        ch, _, _ := keyboard.GetKey()
        chChan <- ch
    }
}(chChan)

for {
    ch := <- chChan           // вводимые символы
    if ch == 0 { break }
    fmt.Printf("<%=>%03x ", string(ch), ch)
}
println()
}

```

Теперь попытка компиляции Go-кода завершится ошибкой, потому что мы должны отразить импорт в файле описания нашего модуля:

```

$ go mod tidy
go: finding module for package github.com/eiannone/keyboard
go: downloading github.com/eiannone/keyboard v0.0.0-20220611211555-0d226195f203
go: found github.com/eiannone/keyboard in github.com/eiannone/keyboard v0.0.0-20220611211555-0d226195f203
go: downloading golang.org/x/sys v0.0.0-20220520151302-bc2c85ada10a

```

Такая команда (синтаксическая форма её неизменная — она не включает в себя упоминание ни имени модуля, ни имени пакета, ни имени каких-то файлов) ищет строки сетевого импорта во всех исходных .go файлах модуля (корневого каталога проекта) и осуществляет фактическую загрузку упоминаемых пакетов из сети:

```

$ ls -o `go env GOPATH`/pkg/mod/github.com/eiannone*
итого 4
dr-xr-xr-x 2 olej 4096 фев  8 21:05 keyboard@v0.0.0-20220611211555-0d226195f203
$ tree `go env GOPATH`/pkg/mod/github.com/eiannone/keyboard*
/home/olej/go/pkg/mod/github.com/eiannone/keyboard@v0.0.0-20220611211555-0d226195f203
├─ AUTHORS
├─ go.mod
├─ go.sum
├─ keyboard_common.go
├─ keyboard.go
├─ keyboard_windows.go
├─ LICENSE
├─ README.md
├─ syscalls.go
├─ syscalls_linux.go
└─ terminfo.go

0 directories, 11 files

```

И требуемый (и произведенный) импорт будет записан в файл описания модуля:

```

$ cat go.mod
module errgrp

go 1.21.3

require (
    github.com/eiannone/keyboard v0.0.0-20220611211555-0d226195f203
    golang.org/x/sys v0.6.0
)

require golang.org/x/sys v0.0.0-20220520151302-bc2c85ada10a // indirect

```

Вот теперь мы можем успешно компилировать приложение:

```

$ go build keyb.go

```

```
$ ls -o keyb.*
-rw-rw-r-- 1 olej 1013 фев  9 04:04 keyb.1.go
-rw-rw-r-- 1 olej  588 фев  9 04:10 keyb.go
```

Или даже просто начать выполнять (ввиду его незначительного объёма) непосредственно выполнять исходный код в режиме JIT (подобно интерпретатору — компиляция «в лёт» в момент вызова):

```
$ go run keyb.go
<q>071 <w>077 <e>065 <r>072 <t>074 <y>079 <й>439 <ц>446 <y>443 <к>43а <е>435 <н>43d
<1>031 <2>032 <3>033
```

(Хорошо видно, что ввод — UNICODE UTF-8, на любом национальном языке).

И, наконец, убедившись что это работает, очень краткие комментарии к коду:

- горутина (анонимная, лямбда-функция) вида `func(chChan chan <- rune)` ожидает одиночного нажатия клавиши...
- при получении символа эта горутина передаёт его по каналу `chChan` в обрабатывающий код.

Примечание: Очень важно при завершении приложения обязательно вызвать `keyboard.Close()`, в противном случае (например, при завершении приложения командой `kill`) мы испортим этот терминал (похоже, что при этом он остаётся в неканоническом режиме ввода). За нас в этом примере делает оператор `defer`.

Многопроцессорный брутфорс

*Чтобы искупать кошку, нужна грубая сила,
настойчивость, мужество убеждений — и кошка.
Последний ингредиент обычно труднее всего найти.
Stephen L. Baker, американский журналист, писатель-фантаст*

Брутфорс (brute force — грубая сила) — метод угадывания пароля (или ключа, используемого для шифрования), предполагающий систематический перебор **всех** возможных комбинаций символов до тех пор, пока не будет найдена правильная комбинация.

Одна из идея скрытия информации основывается на шифровании без возможности декодирования: любое сообщение произвольной длины (короткий текст пароля или длинный двоичный файл) пропускается через хеш-функцию, которая превращает поток байт в битовую последовательность фиксированной длины, зависящую от конкретного вида хеш-функции. Такой принцип используется для идентификации паролей, контроля целостности файлов (контрольные суммы), на этом основана генерация всех криптовалют (когда значение хеша должно не превышать некоторую фиксированную «сложность»).

Существует довольно много алгоритмов хеш-функций для вычисления хешей. MD5 является одним из применяемых алгоритмов хеширования на 128-битной основе. Полученный при этом в ходе вычислений результат представлен в шестнадцатеричной системе исчисления — он называется хешем, хеш-суммой или хеш-кодом. То есть хеш, полученный от функции, работа которой основана на этом алгоритме, выдает строку в 16 байт (128) бит. И эта строка включает в себя 16 шестнадцатеричных чисел.

Свойством хорошей хеш-функции является то, что она не гладкая — малость изменения во входной информации не гарантирует что изменения хеш-кода будет тоже малым, оно может скачкообразно изменяться произвольно. Вторым свойством хеширования является то, что в общем случае (согласно принципу Дерихле) нет однозначного соответствия между хеш-кодом и исходными данными. Возвращаемые хеш-функцией значения менее разнообразны, чем значения входного массива. Поэтому восстановить исходный массив невозможно по хеш-коду, и **единственным** способом гарантировано 100% восстановить исходную информацию является **полный перебор всего** множества

возможных входных последовательностей, выполнение для них хеш-функции, и сравнение итогов. Понятно, что это **чудовищно большое** число повторяемых вычислений! Даже для паролей (Linux), например, длиной 8 символов (и даже 6) это практически не реализуемый процесс в разумное время вычислений.

Сам же термин brute-force обычно используется в контексте хакерских атак, когда злоумышленник пытается подобрать логин/пароль к какой-либо учетной записи или сервису. Понятно, что такая задача замечательно распараллеливается (когда вычисления одной ветви не зависят от результатов другой). Сейчас мы уподобимся злым хакерам и будем подбирать исходный пароль по его MD5 кодированном хеш-коду. Я здесь использую идею изложенную Владимиром Солониным в статье (показана в источниках информации), но радикально изменив её реализацию. Вместо того, чтобы, для **сравнения**, иметь 2 разных приложения — последовательного выполнения и в виде параллельных сопрограмм, я делаю общее приложение, в котором можно указать число сопрограмм которыми ведётся обработка, и если это число указывается как 1, то это и будет эквивалент простого последовательного исполнения (сравнение с отдельно выполненным последовательным приложением показало, что разница при этом не превышает 5-10%).

Словарём (множество символов которые могут входить в искомый ключ) в этом приложении мы выберем '0'...'9' и 'a'...'z'. Это достаточно узкий словарь, реально он включает большие литеры латиницы, спецсимволы ASCII... — расширение словаря резко расширяет сложность (объём) перебора. В этом приложении мы будем «угадывать» **5-символьные** парольные последовательности (уже даже на 6-символьных последовательностях и не очень могучем процессоре мы можем просто не дождаться тестовых результатов).

Итак, код приложения (каталог goproc/bruteforce архива):

bruteforce.go

```
package main
import (
    "crypto/md5"; "encoding/hex"
    "fmt"; "log"; "flag"
    "runtime"; "time"
)

var hash [16]byte
var branch int

func init() {
    flag.IntVar(&branch, "b", -1, "number of parallel branches")
    multip := flag.Bool("m", false, "multiproc evaluation")
    flag.Parse()
    if branch > 0 { return }
    if !*multip {
        branch = 1
    } else {
        if branch < 0 { branch = runtime.GOMAXPROCS(-1) }
    }
}

func main() {
    hashString := "95ebc3c7b3b9f1d2c40fec14415d3cb8" // "zzzzz"
    if len(flag.Args()) > 0 {
        hashString = flag.Arg(0)
    } else {
        fmt.Scanf("%s", &hashString)
    }
    h, err := hex.DecodeString(hashString)
    if err != nil {
        log.Fatal(err)
    }
}
```



```

    }
    copy(hash[0:len(hash)], h)
    fmt.Printf("число процессоров в системе: %v\n", runtime.NumCPU())
    fmt.Printf("число ветвей вычисления: %v\n", branch)
    fmt.Printf("хеш-строка: %v\n", hashString)

    in, out := make(chan string), make(chan string)
    t := time.Now()
    for i := 0; i < branch; i++ {
        go worker(in, out)
    }
    go generator(in)
    fmt.Println("пароль:", <-out)
    fmt.Println("время поиска:", time.Since(t))
}

// следующий символ словаря
var nextByte = func (b byte) byte {
    if 'z' == b {return '0'}
    if '9' == b {return 'a'}
    return b + 1
}

// worker по порядку сравнивает хеш каждой строки пароля с искомым
func worker(in <-chan string, out chan<- string) {
    var p string
    var b []byte
    for {
        p = <-in
        b = []byte(p)
        e := b[0]
        for b[0] == e {
            if md5.Sum(b) == hash {
                out <- string(b)      // хеш совпал - успех!
                return
            }
            for i := len(b) - 1; i >= 0; i-- { // следующая строка
перебора
                b[i] = nextByte(b[i])
                if b[i] != '0' {
                    break // return
                }
            }
        }
    }
}

// generator создаёт начальную строку группы перебора вида "X...."
func generator(in chan<- string) {
    start := []byte("000000")
    var b byte
    for {
        b = nextByte(start[0])
        in <- string(start)
        start[0] = b
        if b == '0' { return } // все возможные комбинации перебраны
    }
}

```

Здесь функция `generator` создаёт поочерёдно начальное значение группы ключей для пробы. Эти начальные значения имеют "00000", "10000" и далее, соответственно, до "z0000" и «перебрасывает» через канал каждую такую группу очередной горутине

worker, которая обчисляет все ключи в этой группе, например от "00000" до "0zzzz", и дальше будет ожидать получения следующей группы от generator.

Прежде чем наблюдать за поведением, отметим некоторые особенности приложения:

- Программа может получить исходные данные (хеш-код который подлежит восстановлению) из разных источников: а). из параметра командной строки, б). из ручного ввода с терминала после запуска (если параметр в командной строке не указан) в). из конвейера командной оболочки, что будет нам особенно полезно.
- Если значение хеш-код не указан в команде запуска, и на ручном вводе (сразу же ввод Enter — пустая строка), то используется хеш-код 95ebc3c7b3b9f1d2c40fec14415d3cb8, соответствующий исходному паролю zzzzz — это самый продолжительный, не оптимальный вариант по времени поиска (но это конкретика частной реализации, и узнать это можно только инсайдерно из анализа кода).
- Опция запуска -m в команде запуска предписывает параллельную многопроцессорную обработку; при этом используется число горутин равное числу физических процессоров в системе.
- Опция запуска -b <число> в команде запуска определяет вести обработку в указанное число параллельных сопроцессов, если: -b 1 — то это означает последовательную обработку в одну ветвь. Указание опции -b с параметром отличным от 1 уже устанавливает параллельную обработку (-m можно не указывать).
- Если ни -m ни -b не указаны, то выполняется последовательная обработка в одну ветвь.

Для подсчёта хеш-кода произвольной символьной последовательности в Linux существует утилита так и называемая: md5sum.

Итак, в том порядке как варианты запуска перечислялись:

```
$ ./bruteforce -m 95ebc3c7b3b9f1d2c40fec14415d3cb8
```

```
число процессоров в системе: 40
```

```
число ветвей вычисления: 40
```

```
хеш-строка: 95ebc3c7b3b9f1d2c40fec14415d3cb8
```

```
пароль: zzzzz
```

```
время поиска: 456.363862ms
```

```
$ ./bruteforce -m
```

```
95ebc3c7b3b9f1d2c40fec14415d3cb8
```

```
число процессоров в системе: 40
```

```
число ветвей вычисления: 40
```

```
хеш-строка: 95ebc3c7b3b9f1d2c40fec14415d3cb8
```

```
пароль: zzzzz
```

```
время поиска: 392.610396ms
```

Пустой ввод (Enter):

```
$ ./bruteforce -m
```

```
число процессоров в системе: 40
```

```
число ветвей вычисления: 40
```

```
хеш-строка: 95ebc3c7b3b9f1d2c40fec14415d3cb8
```

```
пароль: zzzzz
```

```
время поиска: 425.520938ms
```

Ну, и главный вариант — сравнение, из за которого и делалось приложение:

```
$ printf xyz98 | md5sum | ./bruteforce
```

```
число процессоров в системе: 40
```

```
число ветвей вычисления: 1
```

```
хеш-строка: 48450731a4fccb081dd2c4ad3bdfc1c9
```

```
пароль: xyz98
```

время поиска: 9.523057834s

```
$ printf xyz98 | md5sum | ./bruteforce -m  
число процессоров в системе: 40  
число ветвей вычисления: 40  
хеш-строка: 48450731a4fccb081dd2c4ad3bdfе1c9  
пароль: xyz98  
время поиска: 381.871574ms
```

Это 2 предельных случая: последовательное выполнение в одну ветвь, и параллельное выполнение в 40 сопрограмм на 40 процессорах. Время выполнения, производительность, отличаются в 25-30 раз! И можем посмотреть как время задачи зависит от числа задействованных процессоров:

```
$ printf xyzyw | md5sum | ./bruteforce -b 2  
число процессоров в системе: 40  
число ветвей вычисления: 2  
хеш-строка: ab29709ae83c06f43539ac375bd9e298  
пароль: xyzyw  
время поиска: 5.562063176s
```

```
$ printf xyzyw | md5sum | ./bruteforce -b 5  
число процессоров в системе: 40  
число ветвей вычисления: 5  
хеш-строка: ab29709ae83c06f43539ac375bd9e298  
пароль: xyzyw  
время поиска: 2.615070934s
```

```
$ printf xyzyw | md5sum | ./bruteforce -b 10  
число процессоров в системе: 40  
число ветвей вычисления: 10  
хеш-строка: ab29709ae83c06f43539ac375bd9e298  
пароль: xyzyw  
время поиска: 1.387552526s
```

```
$ printf xyzyw | md5sum | ./bruteforce -b 20  
число процессоров в системе: 40  
число ветвей вычисления: 20  
хеш-строка: ab29709ae83c06f43539ac375bd9e298  
пароль: xyzyw  
время поиска: 820.228105ms
```

Собственно, результаты этой задачи (и её код) являются полноценной иллюстрацией всего того, для чего писалась эта книга. Ещё может оказаться показательным, в том смысле, что мы правильно понимает планирование гопрограмм между рабочими потоками ядра исполняющей системы GoLang, запуск этого же приложения с заказанным числом горутин **много больше** чем число имеющихся физических процессоров:

```
$ printf xyzyw | md5sum | ./bruteforce -b 200  
число процессоров в системе: 40  
число ветвей вычисления: 200  
хеш-строка: ab29709ae83c06f43539ac375bd9e298  
пароль: xyzyw  
время поиска: 409.793847ms
```

Видим, что увеличение числа вычислительных горутин **до** числа процессоров (выше) почти линейно уменьшало время вычислений. Но дальнейшее превышение числа горутин **свыше** числа реальных процессоров практически не улучшает вычислительные показатели.

P.S. Попутно, мы с этой задачей лишний раз успокаиваемся относительно целостности паролей в Linux — даже при наших 5 символах в пароле и очень ограниченном словаре, поиск полным перебором становится тоскливым ожиданием. Уже при 6 символах в

пароле из полного словаря (с некоторыми дополнительными средствами сокрытия в Linux, такими как salt пароля) время перебора делает его бессмысленным. А при 8-ми символах — принципиально невозможным, как говорится: «так долго не живут».

Каналы в сопрограммах

Пример использования каналов для взаимного обмена информацией между несколькими go-процедурами показывает следующий пример (каталог `gorproc` архива):

multy.go

```
package main
import (
    "fmt"
    "os"
    "time"
)

func child(num int, in <-chan string, out chan<- string) {
    str1 := fmt.Sprintf("%v : ", num)
    for {
        str2 := <-in // строка полученная из канал
        fmt.Println(str1 + str2)
        if out != nil {
            out <- str2
        } // ретранслируется снова в канал
    }
}

func ввод(ch chan<- string) {
    const per = 3000000000
    buf := make([]byte, 1024)
    for {
        fmt.Printf("> ")
        n, _ := os.Stdin.Read(buf)
        str := string(buf[:n-1])
        fmt.Println(str)
        ch <- str
        time.Sleep(per)
    }
}

func main() {
    канал := [...]chan string{
        make(chan string, 100), make(chan string),
        make(chan string), make(chan string)}
    for i := range канал {
        if i != len(канал)-1 {
            go child(i, канал[i], канал[i+1])
        } else {
            go child(i, канал[i], nil)
        }
    }
    ввод(канал[0])
}
```

Здесь запускается 4 экземпляра go-процедур (они определяются динамически), параллельно выполняющих код функции `child()`. В этом примере демонстрируется, попутно, как каналы передаются в качестве параметров в функции. Специально показано как в определении функции каналы маркированы как входной и выходной (альтернативная, необязательная возможность, в противном случае канал — двунаправленный). Каждый экземпляр (горутинка) имеет входной и выходной канал для передачи символьных сообщений типа `string`. 1-й экземпляр получает информацию по

входному каналу от ввода с терминала, а дальнейшие параллельные сопрограммы передают эту информацию последовательно от экземпляра к экземпляру, по цепочке: 1 -> 2 -> 3 -> 4 :

```
$ ./multy
> 1 2 3 4 5
1 2 3 4 5
0 : 1 2 3 4 5
1 : 1 2 3 4 5
2 : 1 2 3 4 5
3 : 1 2 3 4 5
> new string
new string
0 : new string
1 : new string
2 : new string
3 : new string
> ^C
```

Таймеры

Важнейшая часть любой языковой системы, особенно если это связано с встраиваемыми системами, управляющими реальным оборудованием — это таймеры, обеспечивающие возможность асинхронных реакций на разнообразные события. Даже если код с использованием таймеров внешне не содержит параллелизма, он по смыслу является реализацией параллельно выполняющихся действий.

Таймеры реализованы в пакете `time` и позволяют запланировать и выполнить одно событие в будущем. Мы сообщаем таймеру как долго хотим ждать, и он создаёт канал (типа `chan Time`), по которому будет уведомление в заказанное время. И тип `Timer`, и рассматриваемый дальше тип `Ticker`, описаны в пакете (документации) так:

```
type Timer struct {
    C <-chan Time
    // contains filtered or unexported fields
}

type Ticker struct {
    C <-chan Time // The channel on which the ticks are delivered.
    // contains filtered or unexported fields
}
```

Простейший пример схемы использования таймеров (каталог `goproc/timers`):

timer.go

```
package main

func main() {
    timer := time.NewTimer(2 * time.Second)
    t := time.Now()
    // go func() { выполняем некоторую работу }()
    <-timer.C // ожидание уведомления о срабатывании таймера
    fmt.Println("The timer expired:", time.Since(t))
}

$ ./timer
The timer expired: 2.000085092s
```

Таймер может и не использовать уведомление через канал, а вызывать зарегистрированную в нём функцию реакции. В этом случае в коде вообще внешне не фигурирует понятие таймера как такового, хотя действие реализуется именно через таймер:

timera.go

```
package main

import (
    "fmt"
    "sync/atomic"
    "time"
)

func main() {
    var flag int32 = 0
    t := time.Now()
    time.AfterFunc(2*time.Second, // пауза вызова
        func() {                  // регистрируемая функция
            atomic.StoreInt32(&flag, 1)
        })
    for {
        if atomic.LoadInt32(&flag) != 0 {
            break
        }
    }
    fmt.Println("The timer has worked:", time.Since(t))
}

$ ./timera
The timer has worked: 2.000227289s
```

Тикеры

Ещё один вариант таймеров — периодический таймер, или тикер: посылает (через канал) строго периодические импульсы для выполнения периодически повторяющихся действия. Это классическая схема программного опроса системы ввода-вывода GPIO системы Linux в самых разнообразных системах АСУТП (системах автоуправления). По такой алгоритмике работают управляющие программируемые контроллеры (PLC — Programmable Logic Controller), крайне популярные в промышленной автоматизации на протяжении не одного десятилетия.

tickers.go

```
package main

import (
    "fmt"
    "time"
)

func main() {
    ticker := time.NewTicker(500 * time.Millisecond)
    done := make(chan bool)
    t0 := time.Now()
    go func() {
        for {
            select {
            case <-done:
                return
            case t := <-ticker.C:
                fmt.Println("Tick at ", t.Sub(t0))
                // сосчитать GPIO, обработать и записать в GPIO
            }
        }
    }()
    time.Sleep(3 * time.Second)
```

```

        ticker.Stop()
        done <- true
        fmt.Println("Ticker stopped")
    }

$ ./tickers
Tick at 500.072523ms
Tick at 1.000090135s
Tick at 1.500098076s
Tick at 2.000091364s
Tick at 2.50009904s
Tick at 3.000100251s
Ticker stopped

```

Майнинг

Термин, который за последние 10 лет стал у всех на слуху, и дословно означающий «горнодобыча», но не имеющий никакого отношения к горнорудному ремеслу. Этот жаргонный термин возник в связи с появлением криптовалют, а если ещё шире — с техникой блокчейнов. На сегодня техника майнинга, кроме генерации блоков во всех криптовалютах (блокчейнах), используется гораздо шире:

- генерация «высоких» IPv6 адресов в пиринговой сети Yggdrasil;
- альтернативная система DNS разрешения EmerDNS (на базе криптовалюты EmerDNS);
- альтернативная система DNS разрешения Alfis;

Процедура майнинга настолько широко (и с каждым днём всё шире) применяется в различных проектах, что и сама по себе представляет интерес, безотносительно тому в каких целях она использована. Мы её рассмотрим в несколько упрощённой модели (но везде это происходит именно так) и применительно к многопроцессорной реализации на языке Go:

- Существует некоторый блок, наполняемый произвольными данными (природа этих данных нас не занимает)...
- Для его «закрытия» (фиксации его содержимого) формируется его хеш-сумма по какому-то алгоритму криптографирования (из числа достаточно многих известных);
- В блоке, помимо осмысленных данных, резервируется некоторое небольшое (uint32, uint64, ...) поле (некоторые авторы называют его salt, соль), для которого последовательным перебором подбирается значение так, чтобы для всего блока хеш-сумма была минимальной, или ниже некоторого порогового значения, которое называют «сложностью» (ещё говорят о максимальном числе ведущих нулей в записи хеш-суммы);

Примечание: Смысл здесь состоит в том, чтобы из всего потенциального диапазона хеш-сумм выделить небольшой интервал значений, скажем его 5% или 2%. И считать «успешными» только те хеш-суммы, которые попадают в этот малый участок — чем он уже, тем выше и будет «сложность». В принципе, такой участок «успешных» хеш-сумм мог бы находиться в любой части возможного диапазона значений. Но контролировать попадание значения в интервал технически наиболее просто (а главное быстрее) если разместить этот интервал в нижней части диапазона, начиная от нулевой отметки. А это и значит, что контроль идёт по числу ведущих нулей (двоичных) в записи хеш-суммы.

- Последовательный перебор (и никак иначе) нужен потому, что для хорошего алгоритма хеширования значения хеш-функции не «гладкие»: малейшие изменения в хешируемом блоке приводят к радикальному, хаотическому и скачкообразному, изменению значения итоговой хеш-суммы.

Такой, несколько упрощённой в деталях, модели вполне достаточно чтобы объяснить практически все применения техники майнинга!

Сначала я приведу код примера (каталог `gorproc/mining` архива), а только затем будут даны короткие к нему комментарии. В примере для конкретизации использован крипто алгоритм SHA-512 [6] (использующийся также, например, в формировании IPv6 адресов пиринговой сети Yggdrasil), точно так же можно использовать SHA-256, применяемый, по традиции (идущей ещё от Bitcoin), в блокчейнах практически всех криптовалют. Но с той же лёгкостью вы сможете перестроить такой код под полтора десятков крипто алгоритмов из стандартной библиотеки [7], или ещё под любой из более трёх десятков дополнительных алгоритмов, из импортируемых пакетов [8] ("golang.org/x/crypto") репозитория GoLang.

mining.go

```
package main

import (
    "crypto/sha512"
    "flag"
    "fmt"
    "encoding/binary"
    "time"
    "sync/atomic"
)

const Size512 = 64 // Size is the size, in bytes, of a SHA-512 checksum.

var (branch,      // число горутин
    size,          // размер блока
    width int)
var limit uint

type mesage struct {
    solt uint32
    hash [Size512] byte
}

func (m mesage) String() string { // формат вывода
    s := fmt.Sprintf("%07d | ", m.solt)
    for i, b:= range(m.hash) {
        if width > 0 && i > width {
            s += "... "
            break
        }
        s += fmt.Sprintf("%02X", b)
    }
    return s
}

func (m1 mesage) less(m2 mesage) bool {
    for i := 0; i < Size512; i++ {
        if m2.hash[i] < m1.hash[i] { return true }
        if m2.hash[i] > m1.hash[i] { return false }
    }
    return true
}

func main() {
    exemplar := make(chan mesage, 30) // канал результата
    var count uint32 = ^uint32(0)    // -1
    var finish uint32 = 0
```



```

fmt.Printf("горутин %d; размер блока %d\n", branch, size)
data := make([]byte, size)
for i, _ := range(data) {
    data[i] = byte(i % 256)
}

for _ = range branch {
    go func(ch chan message) {          // горутина хеш-функции
        var m message
        for {
            m.solt = atomic.AddUint32(&count, 1)
            if m.solt > uint32(limit) { break }
            // заполняем 4 начальных байта блока:
            binary.LittleEndian.PutUint32(data, m.solt)
            m.hash = sha512.Sum512(data)
            ch <- m
        }
        if atomic.AddUint32(&finish, 1) == uint32(branch) {
            close(ch)
        }
    }(exemplar)
}

tmStart := time.Now()
dur := func() time.Duration {          // временная метка задержки
    return time.Now().Sub(tmStart)
}
first := true
var m2 message
for m1 := range exemplar {            // получить данные из канала
    if first {
        m2 = m1
        first = false
    }
    if m2.less(m1) {
        fmt.Printf("% 14v : %s\n", dur(), m1)
        m2 = m1
    }
}
fmt.Printf("% 14v : завершение опроса\n", dur())
}

func init() {
    flag.IntVar(&branch, "g", 1, "число горутин")
    flag.IntVar(&size, "b", 1000, "размер блока")
    flag.UintVar(&limit, "v", 32767, "лимит поиска")
    flag.IntVar(&width, "w", 0, "ширина вывода")
    flag.Parse()
}

```

С обработкой опций командной строки запуска посредством пакета `flag` мы уже познакомились. Эту обработку можно сделать и в `main()`, но у нас она вынесена в функцию инициализации `init()`. При запуске могут переопределяться 3 значения:

```

$ ./mining -h
Usage of ./mining:
  -b int
        размер блока (default 1000)
  -g int
        число горутин (default 1)
  -v uint
        лимит поиска (default 32767)

```

```
-w int
    ширина вывода
```

Или точно то же самое можно увидеть так:

```
$ ./mining --help
Usage of ./mining:
  -b int
        размер блока (default 1000)
  ...
```

Две первые опции (-b и -g) понятны из названия (размер хешируемого блока и число горутин выполнения, соответственно). Третий «лимит поиска» означает диапазон перебора подбираемого значения: от 0 и до указанной величины (использование всего диапазона значений uint32 уж слишком большого времени потребовало бы). Последняя опция -w введена в последний миг, «в уходящий поезд», и только для удобства: для того чтобы ограничить ширину вывода хеш-сумм только несколькими (десятками) начальных позиций, для публикации в текстах, Интернет... Нас ведь (когда нет сомнений в корректности и не производится отладка) в значениях хеш-сумм интересуют несколько начальных нулевых позиций, что и есть критерием «сложности».

Обрабатывающие -g горутин берут для пробы последовательные значения count (инкрементируется как атомарное значение atomic.AddUint32()) и отправляют результат хеширования, по буферизированному каналу exemplar, в виде сообщений type message в главную вызывающую функцию main(). Принимающая функция из всей последовательности приходящих из разных источников хешей отбирает минимальный, для чего к объектам типа message определён метод сравнения less.

Примечание: То место в блоке где располагается варьируемое поле (у меня это 4 байта со значением count) в теле хешируемого блока не имеет абсолютно никакого значения: в некоторых проектах это конец блока, в некоторых фиксируемая позиция в структуре блока ... в показанном примере это первые 4 байта блока, только из-за простоты и сокращения кода реализации.

Здесь применена синхронизация отличная от всего что использовалось ранее: принимающая программа (главная) засинхронизирована именно на канале передачи exemplar. Когда последняя из branch отработавших горутин закроет этот канал (и канал для этого должен быть двунаправленным), цикл обработки завершится.

И теперь можем оценить результаты (на терминале вы увидите гораздо более красивый и наглядный вывод, но показать его на странице без переносов невозможно):

```
$ ./mining -b 8000 -v 1000000 -w 20
горутин 1; размер блока 8000
201.692µs : 0000000 | 29516625AB81DCBDC359B0D0767FCE4CA9068D9426...
260.586µs : 0000001 | 1630B8A0062DF2282D00EC3083748DC7E32636A218...
275.151µs : 0000002 | 110649121CB89CDF23BBDEF0A182D521D9C95C99D0...
783.843µs : 0000013 | 0A70DE125FDF60C82600AA5F7E787D4EF60CED4C63...
1.693275ms : 0000031 | 0066A9BBA0F4149C5423D136D92484D58AE62EA353...
51.461271ms : 0001490 | 003AC6A27060ACBCFBEC6D7AB123AF80AAF6CB60D...
57.818574ms : 0001670 | 001443A6F720352EC00B335F0060DA1701E52C3744...
341.379842ms : 0008708 | 000FED88A9BE6D729B70589C5F34884B24ABB3538C...
442.883926ms : 0011400 | 0008CCC1EBE1AFAC852097E6162A77A61B80B2234D...
513.695822ms : 0013631 | 0007B65302A8EF35D6AC335879CA65FBCECB5BFD7E...
552.271948ms : 0014874 | 0002D2DA3F63FB9805B47609C6225191989B49E59D...
1.211426755s : 0028765 | 000269355591B2C2EAD19F7F77F86F93445F10FDC3...
1.449270231s : 0036214 | 00012E5924E21796CB3E2B9CBBA6495A90D093836...
2.45740637s : 0063781 | 000038E088D0320665B3EC72000E0250CCB55F9B4C...
3.953867565s : 0112182 | 00001797F86B7E769526228A1E636BD8B3EA34EBCA...
32.850807613s : завершение опроса
```

Это было последовательное выполнение (в одну ветвь)... и выполнение довольно тоскливое... Хорошо видно, как последовательно во времени увеличивается число ведущих нулей в значениях вычисленных хеш-сумм.

А теперь то же самое, но со всё возрастающим числе ветвей параллельного выполнения:

\$./mining -b 8000 -v 1000000 -w=20 -g=5

горутин 5; размер блока 8000

```
669.669µs : 0000001 | 1630B8A0062DF2282D00EC3083748DC7E32636A218...
874.229µs : 0000000 | 110649121CB89CDF23BBDEF0A182D521D9C95C99D0...
907.066µs : 0000013 | 0A70DE125FDF60C82600AA5F7E787D4EF60CED4C63...
928.083µs : 0000031 | 0066A9BBA0F4149C5423D136D92484D58AE62EA353...
13.654409ms : 0001490 | 003AC6A27060ACBCFBEC6D7AB123AF80AAFB6CB60D...
14.801917ms : 0001670 | 001443A6F720352EC00B335F0060DA1701E52C3744...
60.285762ms : 0008708 | 000FED88A9BE6D729B70589C5F34884B24ABB3538C...
79.566047ms : 0011400 | 0008CCC1EBE1AFAC852097E6162A77A61B80B2234D...
93.381098ms : 0013631 | 0007B65302A8EF35D6AC335879CA65FBCECB5BFD7E...
101.162274ms : 0014874 | 0002D2DA3F63FB9805B47609C6225191989B49E59D...
204.616196ms : 0028765 | 000269355591B2C2EAD19F7F77F86F93445F10FDC3...
262.54894ms : 0036214 | 00012E5924E21796CB3E2B9CBBBA6495A90D093836...
457.735363ms : 0063781 | 000038E088D0320665B3EC72000E0250CCB55F9B4C...
774.432353ms : 0112182 | 00001797F86B7E769526228A1E636BD8B3EA34EBCA...
6.251561417s : завершение опроса
```

\$./mining -b 8000 -v 1000000 -w=20 -g=10

горутин 10; размер блока 8000

```
740.682µs : 0000002 | 110649121CB89CDF23BBDEF0A182D521D9C95C99D0...
833.261µs : 0000000 | 110649121CB89CDF23BBDEF0A182D521D9C95C99D0...
918.637µs : 0000001 | 110649121CB89CDF23BBDEF0A182D521D9C95C99D0...
965.187µs : 0000013 | 0A70DE125FDF60C82600AA5F7E787D4EF60CED4C63...
987.167µs : 0000031 | 0066A9BBA0F4149C5423D136D92484D58AE62EA353...
8.705156ms : 0001490 | 003AC6A27060ACBCFBEC6D7AB123AF80AAFB6CB60D...
9.351802ms : 0001670 | 001443A6F720352EC00B335F0060DA1701E52C3744...
35.925814ms : 0008708 | 000FED88A9BE6D729B70589C5F34884B24ABB3538C...
45.522858ms : 0011400 | 0008CCC1EBE1AFAC852097E6162A77A61B80B2234D...
53.511255ms : 0013631 | 0007B65302A8EF35D6AC335879CA65FBCECB5BFD7E...
58.006366ms : 0014874 | 0002D2DA3F63FB9805B47609C6225191989B49E59D...
105.798054ms : 0028765 | 000269355591B2C2EAD19F7F77F86F93445F10FDC3...
133.613908ms : 0036214 | 00012E5924E21796CB3E2B9CBBBA6495A90D093836...
232.379986ms : 0063781 | 000038E088D0320665B3EC72000E0250CCB55F9B4C...
408.527962ms : 0112182 | 00001797F86B7E769526228A1E636BD8B3EA34EBCA...
3.625750993s : завершение опроса
```

\$./mining -b 8000 -v 1000000 -w=20 -g=20

горутин 20; размер блока 8000

```
664.13µs : 0000000 | 1630B8A0062DF2282D00EC3083748DC7E32636A218...
826.513µs : 0000001 | 1630B8A0062DF2282D00EC3083748DC7E32636A218...
853.143µs : 0000002 | 110649121CB89CDF23BBDEF0A182D521D9C95C99D0...
884.313µs : 0000013 | 0A70DE125FDF60C82600AA5F7E787D4EF60CED4C63...
905.52µs : 0000031 | 0066A9BBA0F4149C5423D136D92484D58AE62EA353...
922.097µs : 0000027 | 0066A9BBA0F4149C5423D136D92484D58AE62EA353...
6.615353ms : 0001490 | 003AC6A27060ACBCFBEC6D7AB123AF80AAFB6CB60D...
7.185671ms : 0001670 | 001443A6F720352EC00B335F0060DA1701E52C3744...
27.907083ms : 0008708 | 000FED88A9BE6D729B70589C5F34884B24ABB3538C...
35.926656ms : 0011400 | 0008CCC1EBE1AFAC852097E6162A77A61B80B2234D...
42.138362ms : 0013631 | 0007B65302A8EF35D6AC335879CA65FBCECB5BFD7E...
45.802609ms : 0014874 | 0002D2DA3F63FB9805B47609C6225191989B49E59D...
86.149212ms : 0028765 | 000269355591B2C2EAD19F7F77F86F93445F10FDC3...
106.299527ms : 0036214 | 00012E5924E21796CB3E2B9CBBBA6495A90D093836...
184.786312ms : 0063781 | 000038E088D0320665B3EC72000E0250CCB55F9B4C...
321.035921ms : 0112182 | 00001797F86B7E769526228A1E636BD8B3EA34EBCA...
2.704524837s : завершение опроса
```

\$./mining -b 8000 -v 1000000 -w=20 -g=40

горутин 40; размер блока 8000

```
1.604µs : 0000000 | 1630B8A0062DF2282D00EC3083748DC7E32636A218...
65.492µs : 0000001 | 1630B8A0062DF2282D00EC3083748DC7E32636A218...
```

```

83.329µs : 0000002 | 110649121CB89CDF23BBDEF0A182D521D9C95C99D0...
706.025µs : 0000013 | 0A70DE125FDF60C82600AA5F7E787D4EF60CED4C63...
866.943µs : 0000031 | 0066A9BBA0F4149C5423D136D92484D58AE62EA353...
6.390164ms : 0001670 | 001443A6F720352EC00B335F0060DA1701E52C3744...
23.418431ms : 0008708 | 000FED88A9BE6D729B70589C5F34884B24ABB3538C...
29.478607ms : 0011400 | 0008CCC1EBE1AFAC852097E6162A77A61B80B2234D...
34.037804ms : 0013631 | 0007B65302A8EF35D6AC335879CA65FBCECB5BFD7E...
36.840543ms : 0014874 | 0002D2DA3F63FB9805B47609C6225191989B49E59D...
65.095478ms : 0028765 | 000269355591B2C2EAD19F7F77F86F93445F10FDC3...
80.027013ms : 0036214 | 00012E5924E21796CB3E2B9CBBBA6495A90D093836...
140.50309ms : 0063781 | 000038E088D0320665B3EC72000E0250CCB55F9B4C...
240.513194ms : 0112182 | 00001797F86B7E769526228A1E636BD8B3EA34EBCA...
2.029330374s : завершение опроса

```

\$./mining -b 8000 -v 1000000 -w=20 -g=100

горутин 100; размер блока 8000

```

1.763µs : 0000000 | 1630B8A0062DF2282D00EC3083748DC7E32636A218...
85.927µs : 0000002 | 110649121CB89CDF23BBDEF0A182D521D9C95C99D0...
201.614µs : 0000001 | 110649121CB89CDF23BBDEF0A182D521D9C95C99D0...
237.443µs : 0000013 | 0A70DE125FDF60C82600AA5F7E787D4EF60CED4C63...
275.222µs : 0000031 | 0066A9BBA0F4149C5423D136D92484D58AE62EA353...
5.357094ms : 0001490 | 003AC6A27060ACBCFBEC6D7AB123AF80AAF6CB60D...
5.797881ms : 0001670 | 001443A6F720352EC00B335F0060DA1701E52C3744...
18.240994ms : 0008708 | 000FED88A9BE6D729B70589C5F34884B24ABB3538C...
22.794165ms : 0011400 | 0008CCC1EBE1AFAC852097E6162A77A61B80B2234D...
26.512151ms : 0013631 | 0007B65302A8EF35D6AC335879CA65FBCECB5BFD7E...
59.065388ms : 0028765 | 000269355591B2C2EAD19F7F77F86F93445F10FDC3...
73.726334ms : 0036214 | 00012E5924E21796CB3E2B9CBBBA6495A90D093836...
128.75394ms : 0063781 | 000038E088D0320665B3EC72000E0250CCB55F9B4C...
226.454647ms : 0112182 | 00001797F86B7E769526228A1E636BD8B3EA34EBCA...
1.768752404s : завершение опроса

```

\$./mining -b 8000 -v 1000000 -w=20 -g=1000

горутин 1000; размер блока 8000

```

1.848µs : 0000000 | 1630B8A0062DF2282D00EC3083748DC7E32636A218...
115.931µs : 0000002 | 110649121CB89CDF23BBDEF0A182D521D9C95C99D0...
135.766µs : 0000001 | 110649121CB89CDF23BBDEF0A182D521D9C95C99D0...
161.356µs : 0000013 | 0A70DE125FDF60C82600AA5F7E787D4EF60CED4C63...
347.502µs : 0000087 | 06AF3918ACD6F1B25C98EBEE82A2AB63220483C823...
631.74µs : 0000314 | 02D3187BE1EF07F056ABBC2858F85BCCF4AE14C7EA...
1.036444ms : 0000453 | 0215C820E323C6B30486B2AE95335D90A3DC2411E9...
1.575007ms : 0000541 | 009689102B4179F39E537B0E39607A7DC1E7DBE4AA...
1.712514ms : 0000540 | 009689102B4179F39E537B0E39607A7DC1E7DBE4AA...
3.627501ms : 0001146 | 00836430BF88A6FF93C676AD47FC369B930B8EFE08...
5.046854ms : 0001283 | 001443A6F720352EC00B335F0060DA1701E52C3744...
5.277489ms : 0001670 | 001443A6F720352EC00B335F0060DA1701E52C3744...
23.665708ms : 0008708 | 000FED88A9BE6D729B70589C5F34884B24ABB3538C...
28.781342ms : 0011400 | 0008CCC1EBE1AFAC852097E6162A77A61B80B2234D...
32.44628ms : 0013631 | 0007B65302A8EF35D6AC335879CA65FBCECB5BFD7E...
35.227234ms : 0014874 | 0002D2DA3F63FB9805B47609C6225191989B49E59D...
59.086869ms : 0028765 | 000269355591B2C2EAD19F7F77F86F93445F10FDC3...
75.135977ms : 0036214 | 00012E5924E21796CB3E2B9CBBBA6495A90D093836...
125.498503ms : 0063781 | 000038E088D0320665B3EC72000E0250CCB55F9B4C...
219.189608ms : 0112182 | 00001797F86B7E769526228A1E636BD8B3EA34EBCA...
1.803812843s : завершение опроса

```

Мы наблюдаем почти линейный рост производительности от числа задействованных процессоров, до тех пор, пока число горутин не приближается к реальному числу физических процессоров (ядер), после чего идёт насыщение роста (но даже и здесь ещё наблюдается некоторый рост производительности ... а воспроизводимость результатов лишний раз подтверждает корректность реализации).

Примечание: Это на заметку тем авторам, отечественным по преимуществу и многочисленным, кто

противопоставляет термины «параллельность» и «конкурентность»... Так вот: до тех пор пока у системы диспетчера Go число горутин не превышает число процессоров (ядер), выполнение истинно параллельное; а при дальнейшем росте числа горутин начинается их конкурентное диспетчирование.

Такие вычисления весьма интенсивно загружают все доступные процессоры. Для того, чтобы успеть зафиксировать это, повторим эксперимент заметно увеличив (опциями) объём вычислений, и (в соседнем терминале) наблюдаем:

```
$ ./mining -b 20000 -v 2000000 -w=20 -g=40
горутин 40; размер блока 20000
1.273365ms : 0000000 | 99192222FF03932BB65AA26EB6B34E2D5BAEFDB4F2...
1.520345ms : 0000001 | 99192222FF03932BB65AA26EB6B34E2D5BAEFDB4F2...
1.563411ms : 0000002 | 479396C36AAB11AEDFD9A081F2B922D1E2F5332A8D...
1.602877ms : 0000004 | 0EE2506A3B3CB5B74CD4B547E2B65C1F03301079C8...
1.644867ms : 0000005 | 0E5A67EB53B59289DC4DF1495F3FDB6F2C3382384F...
1.687613ms : 0000012 | 08327462F845948E01476FCEB9F088A72AD4AB6C24...
1.730579ms : 0000010 | 08327462F845948E01476FCEB9F088A72AD4AB6C24...
1.788965ms : 0000038 | 081539BC10D7489B78B3795882AF6FF42881109BC3...
2.797338ms : 0000094 | 0121CED09BD7357751A83D153C4476C5480618E0FA...
4.235868ms : 0000309 | 0099D9AF8BDBBCBC37A54456FD99F56ED9DF107684...
4.60353ms : 0000385 | 005E3A3863010D14C2B4A1EAFB0293090876F05D79...
5.272215ms : 0000517 | 000C4E3CC4EF171A6B10FD50B6B10F30689DB93C61...
21.045149ms : 0003899 | 0003EA780FF32D79C8350C16A89CCE7A21403EBCD5...
110.006528ms : 0023257 | 00021880C5A75BE0FC80076405873E14C898E084D7...
174.851004ms : 0037375 | 000081A783B9E13B8AA32C611CE76F6060D2C1D994...
600.375804ms : 0136242 | 00001FBB7C33518329D68D657121EB670519BB6ABA...
3.570281554s : 0831405 | 000002FCD2F66101A7EF453BB62959D0A3FECFB4E6...
8.515154044s : завершение опроса
```

\$ uptime

07:30:07 up 48 min, 1 user, load average: 5,06, 2,80, 1,76

Обратите внимание:

- На возрастание числа ведущих нолей в хеше (большая «сложность»), что и является целью майнинга.
- На экспоненциальный рост времени майнинга для нахождения каждой следующей, более оптимальной, записи. В реальных проектах (упоминались ранее), которые используют майнинг для нахождения некоторых оптимальных значений, для нахождения приемлемых значений «сложности» часто требуются часы, а иногда и дни, и даже временами недели непрерывной процессорной работы. А поскольку в реальных задачах такие данные (блоки данных) подшумливаются некоторым случайным генератором, то воспроизведение результатов при повторных запусках не возникает (весь отбор начинается с начала).
- Именно в силу показанного, особенно эффективно криптографические алгоритмы реализовывать на Go, учитывая ту лёгкость, с которой в Go можно задействовать под такую задачу вычислительную мощность всех наличных процессоров. (И из года в год это преимущество Go будет только возрастать из-за технологического роста числа ядер в общедоступных процессорах).

На этом мы заканчиваем рассмотрение майнинга... Но, поскольку вопросы криптографии крайне важны и значимость их постоянно возрастает, алгоритмы трудоёмкие, а Go замечательно пригоден для таких алгоритмах ... то я злоупотреблю вашим некоторым временем для краткого взгляда на состояние дел с криптографией в Go. А те, кого совершенно не занимают вопросы шифрования и защищённости, могут переходить к следующей главе...

В поддержку Go включено множество известных алгоритмов, и отчётливо уделяется

333

внимание новым. Если мы взглянем на пакетные репозитории дистрибутивов Linux, в части, например, новых алгоритмов семейства BLAKE (появившихся только к началу 2020-х годов и находящихся в развитии и пополнении), то только в 2-х языках программирования уже есть легализованная их поддержка, и эти языки — Go и Rust:

```
$ aptitude search blake | grep golang | wc -l
3
$ aptitude search blake | grep rust | wc -l
104
```

В Go «раскрутка» программной поддержки криптографии для вашего кода начинается от пакета `crypto`, где мы найдём [7] любопытный комментарий, который подскажет вам где и как использовать поддерживаемые методы:

```
type Hash uint
```

```
const (
    MD4          Hash = 1 + iota // import golang.org/x/crypto/md4
    MD5          // import crypto/md5
    SHA1         // import crypto/sha1
    SHA224       // import crypto/sha256
    SHA256       // import crypto/sha256
    SHA384       // import crypto/sha512
    SHA512       // import crypto/sha512
    MD5SHA1      // no implementation; MD5+SHA1 used for TLS RSA
    RIPEMD160    // import golang.org/x/crypto/ripemd160
    SHA3_224     // import golang.org/x/crypto/sha3
    SHA3_256     // import golang.org/x/crypto/sha3
    SHA3_384     // import golang.org/x/crypto/sha3
    SHA3_512     // import golang.org/x/crypto/sha3
    SHA512_224   // import crypto/sha512
    SHA512_256   // import crypto/sha512
    BLAKE2s_256  // import golang.org/x/crypto/blake2s
    BLAKE2b_256  // import golang.org/x/crypto/blake2b
    BLAKE2b_384  // import golang.org/x/crypto/blake2b
    BLAKE2b_512  // import golang.org/x/crypto/blake2b
)
```

И сделаем совсем маленькое тестовое приложение (в архиве всё тот же каталог `goros/mining`), которое подскажет нам куда двигаться в направлении конкретного использования любого из методов:

cripto+.go

```
package main

import (
    "crypto"
    "fmt"
    _ "golang.org/x/crypto/md4"
    _ "crypto/md5"
    _ "crypto/sha1"
    _ "crypto/sha256"
    _ "crypto/sha512"
    _ "golang.org/x/crypto/ripemd160"
    _ "golang.org/x/crypto/sha3"
    _ "golang.org/x/crypto/blake2s"
)

func main() {
    for hash := crypto.MD4; hash < crypto.BLAKE2b_512; hash++ {
        fmt.Printf("%-15s\t%v\t%d\n",
            hash, hash.Available(), hash.Size())
    }
}
```

```
}
```

Мы всего лишь хотим посмотреть доступность каждого из методов для использования в своём коде, и длину создаваемой каждым из этих методов хеш-суммы. При компиляции приложения нас ждёт, ожидаемая впрочем, неприятность:

```
$ go build crypto+.go
```

```
crypto+.go:13:2: no required module provides package golang.org/x/crypto/blake2s; to add it:
```

```
    go get golang.org/x/crypto/blake2s
```

```
crypto+.go:6:2: no required module provides package golang.org/x/crypto/md4; to add it:
```

```
    go get golang.org/x/crypto/md4
```

```
crypto+.go:11:2: no required module provides package golang.org/x/crypto/ripemd160; to add it:
```

```
    go get golang.org/x/crypto/ripemd160
```

```
crypto+.go:12:2: no required module provides package golang.org/x/crypto/sha3; to add it:
```

```
    go get golang.org/x/crypto/sha3
```

Но мы уже неоднократно работали с модулями ранее по тексту, и знаем как побороть такую напасть (и не ручным импортом, как говорят подсказки: `go get ...` — хотя и так можно):

```
$ go mod tidy
```

```
go: finding module for package golang.org/x/crypto/sha3
```

```
go: finding module for package golang.org/x/crypto/blake2s
```

```
go: finding module for package golang.org/x/crypto/ripemd160
```

```
go: finding module for package golang.org/x/crypto/md4
```

```
go: downloading golang.org/x/crypto v0.20.0
```

```
go: found golang.org/x/crypto/blake2s in golang.org/x/crypto v0.20.0
```

```
go: found golang.org/x/crypto/md4 in golang.org/x/crypto v0.20.0
```

```
go: found golang.org/x/crypto/ripemd160 in golang.org/x/crypto v0.20.0
```

```
go: found golang.org/x/crypto/sha3 in golang.org/x/crypto v0.20.0
```

```
go: downloading golang.org/x/sys v0.17.0
```

В результате чего (после некоторой паузы связанной с реальной загрузкой из Интернет) файл манифеста текущего модуля (всего что мы отрабатываем в этом каталоге) поменяется:

```
$ cat go.mod
```

```
module mining
```

```
go 1.22.0
```

```
require golang.org/x/crypto v0.20.0
```

```
require golang.org/x/sys v0.17.0 // indirect
```

После чего — сборка и выполнение:

```
$ go build crypto+.go
```

```
$ ./crypto+
```

MD4	true	16
MD5	true	16
SHA-1	true	20
SHA-224	true	28
SHA-256	true	32
SHA-384	true	48
SHA-512	true	64
MD5+SHA1	false	36
RIPEMD-160	true	20
SHA3-224	true	28
SHA3-256	true	32
SHA3-384	true	48

SHA3-512	true	64
SHA-512/224	true	28
SHA-512/256	true	32
BLAKE2s-256	true	32
BLAKE2b-256	false	32
BLAKE2b-384	false	48

Вот это приложение, бесцельное по своему выполнению, в общем, можно использовать как заготовку, которую наполняем целевым кодом при намерении использовать любой из алгоритмов хеширования (я, по крайней мере, так пользуюсь).

Когда не нужно злоупотреблять многопроцессорностью

Существуют задачи, которые сложно или невозможно выразить в форма параллельных вычислений. Обычно это бывает когда, например, последующие ветви вычислений активно используют, а для этого ожидают результаты предыдущих ветвей. Так было, например, при вычислении чисел Фибоначчи.

В других случаях задача просто «просится», на первый взгляд, разложить её составляющие компоненты, слабо зависящие между собой, между различными ветвями и процессорами. Такими кажутся, например, сходящиеся ряды, максимально широко применяющиеся в классической математике для вычисления специальных функций или фундаментальных констант.

Поэкспериментируем с достаточно известной Базельской проблемой, которую Леонард Эйлер решил в 1735, речь идёт о сходимости ряда и нахождения точного значения числа π . Сумма такого ряда $1/1^2 + 1/2^2 + 1/3^2 + 1/4^2 + \dots$ оказывается равной $\pi^2/6$. (Как будет понятно вскоре, я сознательно выбрал из множества известных разложений **не** знакопеременный ряд и ряд известный своей медленной сходимостью.)

Если мы собираемся сравнивать скоростные характеристики, то первым шагом мы сделаем приложение, которое вычисляет с высокой точностью сумму такого ряда чисто последовательностным образом в стиле классического программирования:

bazel0.go

```
package main
import ("fmt"; "math"; "os"; "strconv"; "time")

//  $\pi^2/6 = 1/1^2 + 1/2^2 + 1/3^2 + 1/4^2 + \dots$ 
func main() {
     $\epsilon$  := 1e-17
    if len(os.Args) > 1 {
        if e, err := strconv.ParseFloat(os.Args[1], 64); err == nil {
             $\epsilon$  = e
        } else {
            println("ошибка в параметрах")
            return
        }
    }
    bazel, arg := 0.0, 1.0
    t := time.Now()
    for {
        delta := 1 / (arg * arg)
        if delta <  $\epsilon$  { break }
        arg += 1
        bazel += delta
    }
    const result = math.Pi * math.Pi / 6
    fmt.Printf("[%0.2e:%d] %f -> %f (%e) [%v]\n",
         $\epsilon$ , int(arg),
        bazel, result, bazel - result, time.Since(t))
}
```


Программе при запуске можно указать точность (вещественную) затребованного результата, точность сходимости. Результат программы будет выглядеть примерно так:

```
$ ./bazel0 1e-17
[1.00e-17:316227767] 1.644934 -> 1.644934 (-9.013651e-09) [1.581889477s]
```

Здесь самое важное: напоминание затребованной точности 1.00e-17 (чтобы не запутаться в множестве выводов на терминал), число членов ряда для достижения такой точности 316227767, и время вычисления этого числа членов ряда 1.581889477s.

На этом пока остановимся, и сделаем некоторый эквивалент, но который будет вычислять то же, но «разбрасывая» вычисление членов ряда между последовательными сопrogramмами (и как мы уже знаем, между ядрами, процессорами):

bazel.go

```
package main
import ("fmt"; "math"; "os"; "runtime"; "sync"; "strconv"; "time")

//  $\pi^2/6 = 1/1^2 + 1/2^2 + 1/3^2 + 1/4^2 + \dots$ 
func main() {
     $\epsilon$  := 1e-4
    if len(os.Args) > 1 {
        if b, err := strconv.Atoi(os.Args[1]); err == nil {
            runtime.GOMAXPROCS(b + 1)
        } else {
            println("ошибка в параметрах")
            return
        }
    }
    if len(os.Args) > 2 {
        if e, err := strconv.ParseFloat(os.Args[2], 64); err == nil {
             $\epsilon$  = e
        } else {
            println("ошибка в параметрах")
            return
        }
    }
    var fi, mu sync.Mutex
    bazel, n, yet := 0.0, 1, true
    fi.Lock()
    t := time.Now()
    for yet {
        go func(x float64) {
            delta := 1. / (x * x)
            mu.Lock()
            bazel += delta
            if delta <  $\epsilon$  {
                if yet { fi.Unlock() }
                yet = false
            }
            mu.Unlock()
        }(float64(n))
        n++
    }
    fi.Lock()
    const result = math.Pi * math.Pi / 6
    fmt.Printf("[%2e:%d:%d] %f -> %f (%e) [%v]\n",
         $\epsilon$ , runtime.GOMAXPROCS(-1) - 1, int(n - 1),
        bazel, result, bazel - result, time.Since(t))
    return
}
```

Здесь добавился один (1-й) параметр запуска — число параллельных ветвей, которые

337

будут суммировать числовой ряд. Несколько запусков, для начала в **одну** сопрограмму:

```
$ ./bazel 1 1e-5
[1.00e-05:1:321] 1.641814 -> 1.644934 (-3.120122e-03) [342.929µs]

$ ./bazel 1 1e-7
[1.00e-07:1:3228] 1.644618 -> 1.644934 (-3.165077e-04) [1.452241ms]

$ ./bazel 1 1e-13
[1.00e-13:1:3162363] 1.644934 -> 1.644934 (-3.162191e-07) [1.157085669s]

$ ./bazel 1 1e-15
[1.00e-15:1:32196335] 1.644934 -> 1.644934 (-3.104706e-08) [13.209902424s]
```

А теперь, для сравнения, такая же серия запусков последовательной программы, которую мы сделали раньше:

```
$ ./bazel0 1e-7
[1.00e-07:3163] 1.644618 -> 1.644934 (-3.162055e-04) [15.934µs]

$ ./bazel0 1e-13
[1.00e-13:3162278] 1.644934 -> 1.644934 (-3.162277e-07) [15.894647ms]

$ ./bazel0 1e-15
[1.00e-15:31622777] 1.644934 -> 1.644934 (-3.155929e-08) [159.351051ms]

$ ./bazel0 1e-15
[1.00e-15:31622777] 1.644934 -> 1.644934 (-3.155929e-08) [159.351051ms]
```

Вот так! Последовательный вариант медленнее в 100 раз! Этого следовало ожидать, но не в такой степени... Но мы можем рассчитывать, что нас, как в предыдущих задачах ранее, спасёт большое число параллельных ветвей обработки:

```
$ ./bazel 2 1e-13
[1.00e-13:2:3162342] 1.644934 -> 1.644934 (-3.162212e-07) [1.917860121s]

$ ./bazel 10 1e-13
[1.00e-13:10:3162322] 1.644934 -> 1.644934 (-3.162232e-07) [2.539155868s]

$ ./bazel 20 1e-13
[1.00e-13:20:3162314] 1.644934 -> 1.644934 (-3.162240e-07) [3.09384969s]

$ ./bazel 40 1e-13
[1.00e-13:40:3162284] 1.644934 -> 1.644934 (-3.162270e-07) [3.068690143s]
```

Результат на многих процессорах ещё хуже чем на одном! И, более того, чем больше процессоров — тем медленнее итоговая работа.

И только теперь мы можем попытаться проанализировать то что мы видели, и сделать выводы на будущее:

- Параллельные программы **всегда** требуют синхронизации доступа к данным (ну, разве что исключая случаи совершенно автономных серверов с не взаимодействующими ветвями обслуживающих независимых клиентов).
- Для синхронизации используем **специальные** примитивы синхронизации, обычно известные как элементы из области IPC (Inter Process Communication): атомарные переменные, мютексы, условные переменные, семафоры, каналы и др.
- Работа примитивов синхронизации (даже самых быстрых атомарных переменных) **очень медленная** в сравнении с обычными операторами языка.
- Если мы сильно «мельчим» с той долей работы, которая будет выполняться в каждой ветви параллельной программы, то работа по **синхронизации** может намного превысить часть содержательно работы по **обработке данных** в

параллельных ветвях.

- И увеличение числа параллельных ветвей, процессоров, не только улучшает, но даже несколько ухудшает цифры, потому что всё больше ветвей (потокосов ядра системы, по существу говоря) «толкуются» у мютексов, ограждающих критическую секцию.

Выводы:

- Механическое распараллеливание задачи, даже хорошо и грамотно выполненное, не всегда улучшает временные характеристики задачи, может быть и наоборот.
- В ограждаемых примитивами синхронизации фрагментах кода (критических секциях) должно содержаться достаточно много целевых операций задачи, чтобы на их фоне нивелировать издержки на работу примитивов синхронизации.

Источники информации

[1] Простой пример использования goroutines в языке Go, Владимира Солонин, 3 апреля 2015 —

<https://eax.me/go-goroutines/>

[2] Использование пакета Flag в Go, January 24, 2020 —

<https://www.digitalocean.com/community/tutorials/how-to-use-the-flag-package-in-go-ru>

[3] ПЛК — что это такое? — <https://habr.com/ru/post/139425/>

[4] Бикрамадитья Сингхал, Гаутам Дамеджа, Панда Приянсу Сехар

Блокчейн. Руководство для начинающих разработчиков. BHV, С-Петербург, 2019, 288 страниц, ISBN: 978-5-9775-4052-0

[5] Нараян Прасти, Блокчейн. Разработка приложений. BHV, С-Петербург, 2019, 256 страниц, ISBN 978-5-9775-3976-0

[6] sha512. Documentation — <https://pkg.go.dev/crypto/sha512>

[7] crypto. Documentation — <https://pkg.go.dev/crypto>

[8] Go Cryptography — <https://pkg.go.dev/golang.org/x/crypto>

Разработка через тестирование

Разработка через тестирование — это одна из современных высоко продуктивных методик создания программных проектов. Ранее уже упоминалось, что GoLang предоставляет всё необходимое для быстрого создания тестов параллельно с созданием целевых пакетов. Мы рассмотрим эту технику на частном примере простого пакета (подкаталоги `tests` архива примеров кода), но с такой же лёгкостью вы можете создать тесты для любого создаваемого пакета. Наш исходный программный пакет:

degree.go

```
package degr

import "math"

func Sin(dg float64) float64 {
    return math.Sin(dg / 180 * math.Pi)
}
```

Это некоторый условный пакет, который вычисляет тригонометрические функции, но от аргумента заданного не в радианах, а в угловых градусах.

Создаём для него парный пакет для тестирования. Ранее было рассказано, что для тестирования функций в файле `xxx.go` тестовый файл будет иметь вид `xxx_test.go`:

degree_test.go

```
package degr

import "testing"
import "math"

func TestSin(t *testing.T) {
    got, want := Sin(30.0), 0.5
    if math.Abs(got - want) > 1E-7 {
        t.Errorf("Sin(30) = %f, должно быть %f", got, 0.5)
    }
}
```

Здесь код относится к тому же пакету (`degr`), что и тестируемый код (имя пакета сознательно выбрано отличным от имён файлов, чтобы оттенить разницу). Если мы попытаемся выполнить тестирование в таком составе, то (в современных версиях GoLang) получим:

```
$ go test
```

```
go: go.mod file not found in current directory or any parent directory; see 'go help modules'
```

А это означает, что нам нужно создать манифест модуля `go.mod` ... подобное мы уже неоднократно проделывали:

```
$ go mod init degree
```

```
go: creating new go.mod: module degree
go: to add module requirements and sums:
    go mod tidy
```

(Отметим здесь, что помимо имени файлов и имени пакета возникла ещё одна сущность: имя модуля, которое не связано с двумя предыдущими; для наших экспериментов мы можем выбрать его произвольным.)

После чего:

```
$ go test
```

```
PASS
ok      degree  0.003s
```

Мы теперь можем успешно контролировать адекватность целевого модуля прямо по ходу развития его кода. Покажем попутно как выглядит ошибочное завершение теста ... перепишем файл так (подкаталог `/tests/degree.err`):

degree_test.go

```
package degr

import "testing"
import _ "math"

func TestSin(t *testing.T) {
    got, want := Sin(30.0), 0.5
    if got != want {
        t.Errorf("Sin(30) = %f, должно быть %f", got, 0.5)
    }
}
```

Теперь тестирование завершается так (достаточно неожиданно):

```
$ go test
--- FAIL: TestSin (0.00s)
    degree_test.go:9: Sin(30) = 0.500000, должно быть 0.500000
FAIL
exit status 1
FAIL    degree  0.004s
```

Но в данном случае — это нам лишний раз напоминание что в машинном формате (независимо от языка программирования) нельзя сравнивать вещественные значения! Вещественные числа — это приближённые представления значений в машинных форматах.

По этой схеме, совершенно единообразной, вы можете организовать параллельное тестирование любых своих разрабатываемых программных модулей.

Заключение

В Go ещё множество интересных «фишек»... Но книга эта не столько про Go, сколько про то, как в своём программном коде использовать множество процессоров одновременно, предоставляемых оборудование. И о том, что именно Go изначально «заточен» на такое использование — его авторы предвосхитили именно такое направление развитие компьютерных тенденций.

Эра повышения производительности компьютерных систем за счёт **скорости** (тактовой частоты) процессора, протяжённостью в 50-60 лет, завершилась. Началась эра повышения производительности за счёт **числа** процессоров. И вопрос того как использовать эту изменившуюся реальность ещё требует осмысления...

Об авторе

Цилиурик Олег Иванович — практик-разработчик архитектуры (системотехники) и, главным образом, программного обеспечения крупных компьютерных проектов со стажем практических разработок больше 40 лет. Ещё в последние десятилетие существования СССР участвовал в больших проектах области ВПК СССР, в составе крупнейших инженерно-научных центров: ВНИИ РТ, КБ ПМ г.Москва.

С того времени непрерывно работал в самых разнообразных разработческих проектах. Как обобщение этого опыта издал 2 книги в издательствах Москвы и С.-Петербурга, и около 20 книг в электронном виде, достаточно известных в программистских кругах.

Параллельно работал 5-6 лет научным редактором переводной компьютерной литературы издательства «Символ-Плюс» — хорошо известная серия компьютерной литературы «со зверушками на обложке».

Автор известного курса по разработке модулей ядра, драйверов Linux, гуляющего по Интернет в виде конспекта на протяжении 10 лет, который разрабатывался для проведения учебных тренингов программистов-разработчиков известной международной софтверной фирмы Global Logic.

Работу с языком Go начал с 2013 года, по заказу той же компании Global Logic учебный курс, известный (гуляющий по Интернет) в электронных публикациях как «Go конспект» — адаптация в Go практикующих программистов C/C++, и хронологически первый в русскоязычных публикациях обстоятельный курс Go.

Настоящий текст является радикальной переработкой того, что начато писаться в 2012-2014 годах, с, более того, очень значительным «уклоном» в другую сторону — использование Go в многопроцессорных системах для их эффективного использования.