

# Go

Начальный курс по Go

**CONCURRENCY**

# Concurrency

Возможность исполнять несколько задач  
конкурентно

Ключевое: исполняются несколько задач (не  
обязательно параллельно\*)

Примечание:\* речь об истинной параллельности

# Многопоточность

Существование нескольких потоков  
исполнения кода внутри одного приложения

# Многопоточность

Потоки могут исполняться как на одном ядре процессора, так и на разных

# Многопоточность

Ключевые задачи:

- лучшее использование ресурсов
- "распараллеливание задач"

Важно: многопоточность – это не дёшево и не всегда быстрее однопоточного варианта

# Многопоточность

Ключевое: многопоточность – это сложно, т.к.  
потоки используют одни и те же объекты в  
памяти\* и состояние приложения меняется в  
зависимости от разной последовательности  
взаимодействия потоков

# Ключевые проблемы

Синхронизация:

- последовательности операций
- доступа к общим данным



**GOROUTINES**

# Варианты в Go

- Communicating Sequential Processes
- Shared Memory

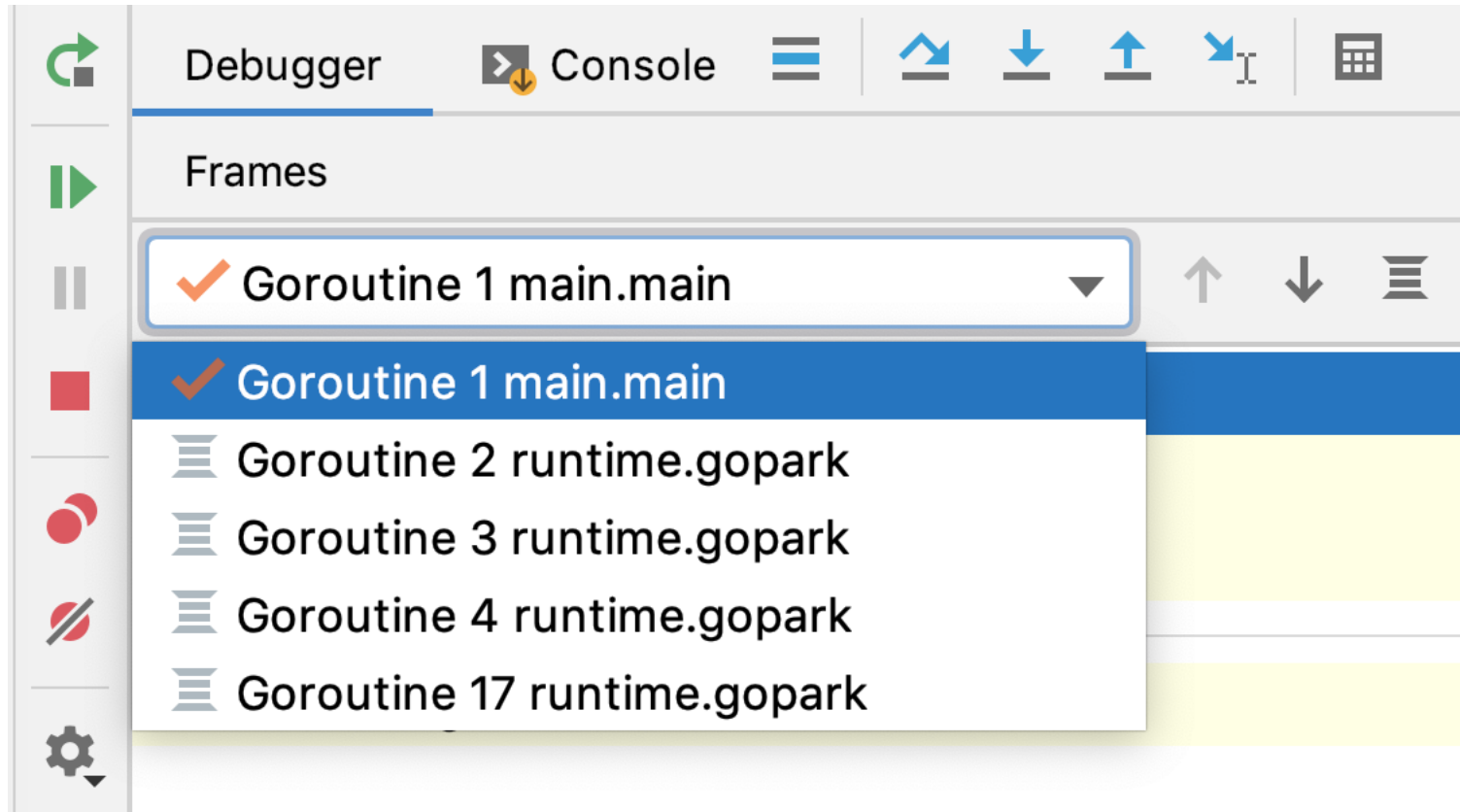
# goroutine

Конкурентная активность в Go, фактически,  
выполнение функции "независимо" от  
вызвавшей её функции

# goroutine

Как минимум, всегда существует хотя бы одна goroutine (при старте приложения вызывает main)

# goroutine



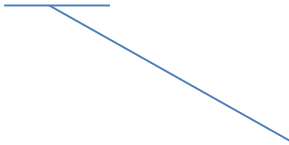
# goroutine

User-space threads (управляются самим Go), в  
отличии от OS threads

Более "легковесны" чем OS threads

# goroutine

**go** f()



вызов функции f() в новой goroutine

# Scheduler

Scheduler не гарантирует последовательности запуска goroutine и может переключать их исполнении при:

- создании новой goroutine
- операциях с каналами
- blocking syscall
- завершение цикла garbage collection



# Atomics

Современные CPU поддерживают атомарные операции сравнения и установки значения (вместо самописных синхронизированных)

<https://golang.org/pkg/sync/atomic>

# Mutex

```
mutex := sync.Mutex
```

```
mutex.Lock()
```

```
mutex.Unlock()
```

# RWMutex

```
mutex := sync.RWMutex{}
```

Позволяет устанавливать блокировки на чтение (организуя множество читателей)

- **RLock** - блокируем на чтение (читателей может быть много)
- **Lock** - на запись (писатель может быть только один)

# WaitGroups

Позволяет "ожидать" завершения серии конкурентных операций:

```
var wg sync.WaitGroup
```

```
wg.Add(2) ← Добавляем для .Wait требование ожидания двух .Done
```

```
go func() { defer wg.Done(); ...}
```

```
go func() { defer wg.Done(); ...}
```

```
wg.Wait()
```

**CHANNELS**

# Channels

"Do not communicate by sharing memory;  
instead, share memory by communicating"

Shared memory - достаточно тяжело

# Channels

Channel - канал, по которому могут передаваться данные только определённого типа:

```
ch := make(chan int)
```

```
ch <- 10 // отправка данных в канал
```

```
value := <- ch // получение данных из канала
```

# Unbuffered Channels

```
ch := make(chan int)
```

```
ch <- 10
```

```
fmt.Println(<- ch) // deadlock
```

1. Если канал пустой, receiver блокируется до получения данных
2. Отправитель может отправлять данные только в пустой канал (если канал не пустой, то блокируется)

fatal error: all goroutines are asleep - deadlock!



# Buffered Channels

```
ch := make(chan int, 3) // 3 - capacity
```

```
ch <- 10
```

```
fmt.Println(<- ch)
```

1. Если канал пустой, receiver блокируется до получения данных
2. Отправитель может отправлять данные в канал только если канал не полностью заполнен

# Unidirectional

`<-chan int` – только для приёма сообщений

`chan<- int` – только для отправки сообщений

Чаще всего указывается в аргументах функций  
(при этом передавать можно и `bidirectional`, но  
внутри функции будет как `unidirectional`)

# len & cap

builtin функции **len** и **cap** возвращают length и capacity канала соответственно

# close

builtin функция **close** закрывает канал, после чего:

- отправка данных в канал приводит к panic
- чтение из канала приводит к получению нулевого значения (и отсутствию блокировок)

# close

value, ok := <-ch

В **ok** будет **false**, если канал закрыт

# for range

```
for value := range ch {}
```

// ok писать нельзя, поэтому:

```
for {  
    value, ok := <-ch  
}
```

# multiplexing

```
select {  
    case <receive or send>:  
    case <receive or send>:  
}
```

Блокируемся, пока не сработает один из **case**  
(чаще всего заворачивают в **for {}**)

# multiplexing

```
select {  
    case <receive or send>:  
    default: ...  
}
```

Если сейчас нет ни одного доступного для чтения (или записи) канала, то выполняем **default** (без блокировки)



# multiplexing

```
select {
```

```
    case out<- value: ...
```

```
    case <-stop: return
```

```
}
```

останавливаемся,  
как в chan stop придёт значение

# timeout

```
select {
```

```
  case out<- value: ...
```

```
  case <-time.After(x * time.Second)
```

останавливаемся,  
} как пройдёт некоторое кол-во времени

# race

`go run -race app.go`

Или можно в GoLand в конфигурации запуска  
прописать этот флаг:

Environment:	<input type="text"/>
Go tool arguments:	<input type="text" value="-race"/>
	<input type="checkbox"/> Use all custom build tags
Program arguments:	<input type="text"/>
	<input type="checkbox"/> Run with sudo

# race

```
go run -race app.go
```

WARNING: DATA RACE

Read at 0x00c00009e008 by goroutine 8:  
main.main.func1()

[https://golang.org/doc/articles/race\\_detector.html](https://golang.org/doc/articles/race_detector.html)

**ПРОЕКТИРОВАНИЕ**

# Проектирование

1. Строится однопоточная модель приложения (для дальнейшего измерения производительности и валидации многопоточной модели)
2. Выделяются возможности декомпозиции по задачам и по данным
3. Реализуется многопоточная модель

См. Intel «[Threading Methodology: Principles and Practices](#)»

# **12 FACTOR APPS**

# 12 factor apps

Ознакомиться:

<https://12factor.net>



Проект 1

# ARCHIVATOR

# Archivator

Используя `sync.WaitGroup`, горютины и пакет `zip` организуйте:

1. Последовательную версию архиватора
2. Конкуррентную версия архиватора

# Archivator

Как работает архиватор:

```
archivator.exe file1.txt file2.txt file3.txt
```

Создаст архивы: file1.txt.zip, file2.txt.zip,  
file3.txt.zip

Спасибо за внимание

Ильназ Гильязов

2020г.