

Into Go from the inside



The background is a dark, almost black, 3D-rendered scene. It features a grid of rectangular blocks or platforms of varying heights. Each block is outlined with a bright blue, glowing light that creates a sense of depth and perspective. The lighting is directional, coming from the upper left, which casts soft shadows and highlights the edges of the blocks. The overall aesthetic is futuristic and technological.

Часть 2

Эффективная многозадачность в Go



{Channel}

Устройство каналов в Go

Встроенные операции

Чтобы найти реализацию встроенной в язык операцию, обычно достаточно просмотреть сгенерированный код.

make(chan T)

```
// chan_example.go
func makeUnbufferedIntChan() chan int {
    return make(chan int)
}

// $ go tool compile -S chan_example.go
// CALL runtime.makechan(SB)
```

Пакет runtime

Реализацию пакета runtime можно найти по пути ``$(go env GOROOT/src/runtime)``.

Далее вы можете выполнить `grep` по известной функции, в нашем случае это `makechan`.

make(chan T)

```
$ cd $(go env GOROOT)/src/runtime  
$ grep -nr 'func makechan'  
chan.go:63:func makechan64(t *chantype, size  
int64) *hchan {  
chan.go:71:func makechan(t *chantype, size  
int) *hchan {
```

hchan (runtime/chan.go)

```
type hchan struct {  
    qcount    uint           // total data in the queue  
    dataqsiz  uint           // size of the circular queue  
    buf       unsafe.Pointer // points to an array of dataqsiz elements  
    elemsize  uint16  
    closed    uint32  
    elemtype  *_type // element type  
    sendx     uint       // send index  
    recvx     uint       // receive index  
    recvq     waitq      // list of recv waiters  
    sendq     waitq      // list of send waiters  
  
    // lock protects all fields in hchan, as well as several  
    // fields in sudogs blocked on this channel.  
    lock mutex  
}
```


Mutex в канале?

```
type hchan struct {  
    // ...  
  
    // lock protects all fields in hchan,  
    // as well as several fields in  
    // sudogs blocked on this channel.  
    lock mutex  
}
```

Send & Receive

```
func send(ch chan int) {  
    ch <- 10  
}  
// CALL runtime.chansend1(SB)  
  
func recv(ch chan int) {  
    <-ch  
}  
// CALL runtime.chanrecv1(SB)
```

Send & Receive

```
// entry point for c <- x from compiled code
func chansend1(c *hchan, x unsafe.Pointer) {
    chansend(c, x, true, getcallerpc())
}
```

```
// entry points for c <- c from compiled code
func chanrecv1(c *hchan, x unsafe.Pointer) {
    chanrecv(c, x, true)
}
```

chansend & chanrecv

Внутри реализации `chansend` и `chanrecv` можно найти `lock(&c.lock)` и `unlock(&c.lock)`.

Атомарность получений и отправлений данных в общем случае не является lock-free.

=> Каналы не быстрее традиционной синхронизации для произвольной задачи.

Особенности каналов в Go

- Встроенный типобезопасный контейнер
- Позволяют организовать синхронизацию
- Чтение/запись сериализованы
- Поддержка `select`, `for`/`range`

Каналы vs “всё остальное”

Чистая CSP (communicating sequential processes) модель подразумевает коммуникацию вычислителей только через каналы, но на практике в Go сложно обойтись одними лишь каналами и это вопрос не только производительности.



{Channel: performance}

Когда канал не самый эффективный
выбор

Постановка задачи

- Обновление разделяемого счётчика
- Запись из N горутин
- Синхронизация через каналы не требуется

Можно представить эту задачу в базах данных, где встречаются последовательности (sequence) с возрастающими ID.

Варианты решения

1. Каналы
2. Мьютексы (sync package)
3. Атомики (sync/atomic package)

Реализация бенчмарков:

<https://bit.ly/2DtR4iz>

Результаты бенчмарков

BenchmarkChannel-8	429	ns/op
BenchmarkMutex-8	204	ns/op
BenchmarkAtomic-8	165	ns/op

// Atomic > Mutex > Channel для простейшей
// операции атомарного обновления значения.
// Вы можете ускорить бенчмарк с каналами,
// но быстрее мьютексов он не станет.

{Синхронизация в Go}

Каналы для синхронизации, а также
остальные механизмы синхронизации

Синхронизация через каналы

- **1-to-N**: закрытие канала через `close` (**once**)
- **1-to-1**: отправка и получение сообщения

Примером 1-to-N является **`context.WithCancel`** из стандартной библиотеки. Метод **`Done()`** позволяет всем горутинам получить событие вызова функции **`CancelFunc`**.

sync.Cond vs channel/select

sync.Cond может использоваться для эффективной и многократной 1-to-N отправки (метод Broadcast).

См. Также:

- > [sync: add example for Cond](#)
- > [go2: proposal: remove sync.Cond](#)

Синхронизация через группы

- `sync.WaitGroup` (`stdlib`)
- `errgroup.Group` (golang.org/x/sync/errgroup)
- `run.Group` (github.com/oklog/run)

`run.Group` также позволяет определить как завершить выполнение горуты из группы.

(Не)рекурсивные мьютексы

```
// sync.Mutex не рекурсивен.  
func main() {  
    var mu sync.Mutex  
    mu.Lock()  
    mu.Lock() // Deadlock  
    mu.Unlock()  
    mu.Unlock()  
}
```


Атомики и spinlock

В горячих путях кода spinlock может быть эффективнее, чем блокировка со сном.

С помощью `sync/atomic` можно реализовать свой spinlock цикл ожидания.

<https://bit.ly/2qBE95>

Data race

Без необходимой синхронизации, в программе могут появляться гонки данных.

Попробуйте запустить код, доступный по ссылке, с параметром “-race”.

<https://bit.ly/2zCrtQ5>

go -race

Параметр “-race” может использоваться совместно с тестами (и бенчмарками).

К сожалению, race detector не находит те гонки данных, которые возникают в коде, который не был исполнен.

=> Пишите больше тестов и бенчмарков.

The background is a dark, almost black, space filled with a complex, three-dimensional grid of glowing blue lines. These lines form various geometric shapes, including cubes and rectangular prisms, which are interconnected to create a sense of depth and perspective. The lines themselves have a soft, ethereal glow, and the overall composition suggests a high-tech or digital environment.

{Cooperative vs preemptive}

Обзор некоторых особенностей
работы планировщика

1 CPU, но несколько горутин 👍

```
func main() {  
    runtime.GOMAXPROCS(1)  
    go func() {  
        for { time.Sleep(1) }  
    }()  
    time.Sleep(100 * time.Millisecond)  
    fmt.Println("hello world")  
}  
// => Печатает "hello, world"
```

1 CPU, но несколько горутин 🙅

```
func main() {  
    runtime.GOMAXPROCS(1)  
    go func() {  
        for {}  
    }()  
    time.Sleep(100 * time.Millisecond)  
    fmt.Println("hello world")  
}  
// => Программа уходит в бесконечный цикл
```

Кооперативная многозадачность

```
go func() {  
    for {  
        // В этом цикле нет точки вытеснения.  
        // Когда OS thread заходит в него,  
        // он никогда не может вернуть  
        // управление рантайму.  
    }  
}()  
// См. https://golang.org/issue/10958.
```

Пример блокировки планировщика

Код, вызывающий блокировку, можно найти по ссылке, указанной ниже.

<https://bit.ly/2ASCHBO>

Мониторинг работы планировщика

С помощью пакета `runtime/trace` можно собирать профиль работы планировщика. API очень близок к CPU/memory профилированию.

Просмотреть trace профиль можно с помощью команды “go tool trace”.



{goroutines}

Green threads в Go

Создание горутины

```
func spawnGoroutine() {  
    go func() {}()  
}  
  
// CALL runtime.newproc(SB)  
  
// А вот код внутри proc.go довольно сложен.  
// См. Файл $GOROOT/src/runtime/HACKING.md,  
// который описывает терминологию.
```

Сравнение с OS threads

	Posix thread	Goroutine
Stack size	2 MB	2 KB (1/1000)
Locking/sync	Kernel space	User space (channels)
Switching	Kernel space	User space
Managed	Нет	Да (can steal work, etc.)

Состояния горутины

- Выполняется (runs)
- В ожидании (runnable)
- В блокировке (blocked)

Горутина заблокирована, если её выполнение остановлено системным вызовом (ввод/вывод) или операцией над каналом.

Блокировка горютины

Если выполнение достигает блокирующей операции, то блокируется OS thread, на котором исполняется горютина.

Сама горютина помечается как blocked. Работа, ассоциированная с исполняющим потоком будет переназначена на другой.

Handoff

Детектирование заблокированных потоков выполняется “монитором”, который работает в отдельном потоке.

Рантайм может порождать новые потоки, если их начинает не хватать в связи с I/O блокировками.

Переключение контекста

Горутини управляются самим Go, а не операционной системой.

Переключение с одной горутини на другую не требует большого количества вычислений, поскольку вся работа выполняется внутри одного процесса, внутри user space.

Unstoppable

Горутины не представляют прямого способа прервать их выполнение.

Чтобы управлять временем жизни горутины (и прервать длительную операцию по истечению времени) можно использовать, например, пакет **context**.

Thread parking

Когда горутина завершает исполнение (умирает), исполняющий поток возвращается в пул потоков планировщика.

Потоки в пуле планировщика находятся в пассивном ожидании и не расходуют большого количества CPU ресурсов.

The background is a dark, almost black, space filled with a complex network of glowing blue lines. These lines form a grid-like pattern with various rectangular and square shapes, some of which are slightly offset or layered, creating a three-dimensional effect. The lines are bright blue and have a soft glow, contrasting sharply with the dark background.

{Go scheduler}

Высокоуровневый обзор
планировщика

Go scheduler

- Не полностью вытесняющий
- N:M:P модель (или G:M:P)
- Переиспользует “настоящие” потоки

Часть материала ниже взята из статьи JBD:

<https://rakyll.org/scheduler>

N:M:P

- N горутин (В Go именуется G)
- M нативных тредов (OS threads)
- P процессоров

Обычно, $N > M$.

P может быть ограничено через `GOMAXPROCS`.

Заимствование работы (work stealing)

Если у M (OS thread) не осталось runnable горутин в runqueue, то этот M может забрать некоторую часть работы у другого треда, который загружен работой.

Помимо локальных для треда runqueue, существуют также глобальные очереди.