

ЛЕКЦИЯ 5. АСИНХРОННЫЕ СУЩНОСТИ И ПАТТЕРНЫ В GO

OZON

МОСКВА, 2021

ЛЕКЦИИ














1. Введение. Рабочее окружение. Структура программы. Инструментарий.
2. Базовые конструкции и операторы. Встроенные типы и структуры данных.
3. Структуры данных, отложенные вызовы, обработка ошибок и основы тестирования
4. Интерфейсы, моки и тестирование с ними
5. Асинхронные сущности и паттерны в Go
6. Protobuf и gRPC
7. Работа с БД в Go
8. Брокеры сообщений. Трассировка. Метрики.

ТЕМЫ

Сегодня мы поговорим про:

1. Параллелизм и конкурентность
2. Горутины
3. Каналы
4. Примитивы синхронизации
5. Гонки приоритетов
6. Контексты

ОБОЗНАЧЕНИЯ

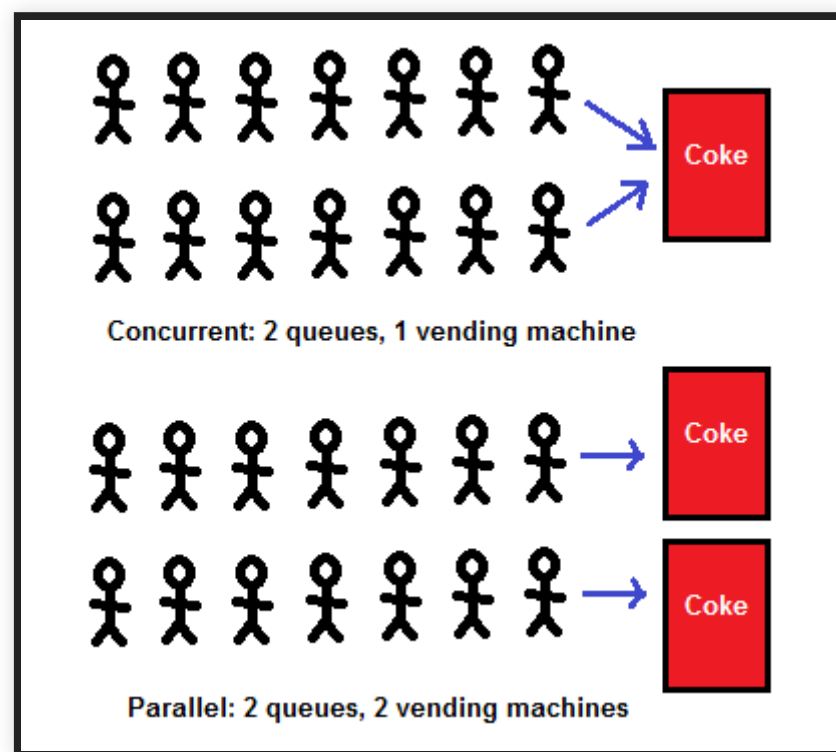
-  - посмотри воркшоу
-  - проведи эксперимент
-  - изучи внимательно
-  - прочитай документация
-  - подумай о сложности
-  - запомни ошибку
-  - запомни решение
-  - обойди камень предковенья
-  - сделай перерыв
-  - попробуй дома
-  - обсуди светлые идеи
-  - задай вопрос
-  - запомни панику

ПАРАЛЛЕЛИЗМ И КОНКУРЕНТНОСТЬ

- Параллелизм – это исполнение задач параллельно на физическом уровне, то есть когда у каждого параллельного процесса есть свой набор ресурсов, которые не разделяются. Аналогия: параллельные прямые.
- Конкурентность – это исполнение задач параллельно на логическом уровне. То есть когда мы можем управлять общими ресурсами так, чтобы задачи эффективно их использовали.

⚡ В русскоязычной литературе эти два термина путаются. Смотрите на определения выше.

ПАРАЛЛЕЛИЗМ И КОНКУРЕНТНОСТЬ



source: <https://joearms.github.io/#Index>

ГОРУТИНЫ

Goroutine — это, по сути, Coroutine, но это Go, поэтому мы заменяем букву C на G и получаем слово Goroutine.

Как и в ОС в Go есть свой планировщик, который определяет когда какая горутина выполняется:

1. В Go ≤ 1.13 был кооперативный планировщик, т.е. каждая горутина исполнялась, пока не подходила к месту, где можно переключить контекст исполнения на другую горутину (системные вызовы, мьютексы и т.д.)
2. С Go 1.14 планировщик стал вытесняющим. То есть он может переключить контекст тогда, когда захочет. Его поведение недетерминировано.

 <https://habr.com/ru/post/489862/>

ГОРУТИНЫ

🏠 Сравни как будет исполняться на Go 1.13 и Go \geq 1.14.

```
package main

import (
    "fmt"
    "runtime"
    "time"
)

func main() {
    runtime.GOMAXPROCS(1)

    fmt.Println("The program starts ...")

    go func() {
        for {
        }
    }()

    time.Sleep(time.Second)
    fmt.Println("I got scheduled!")
}
```

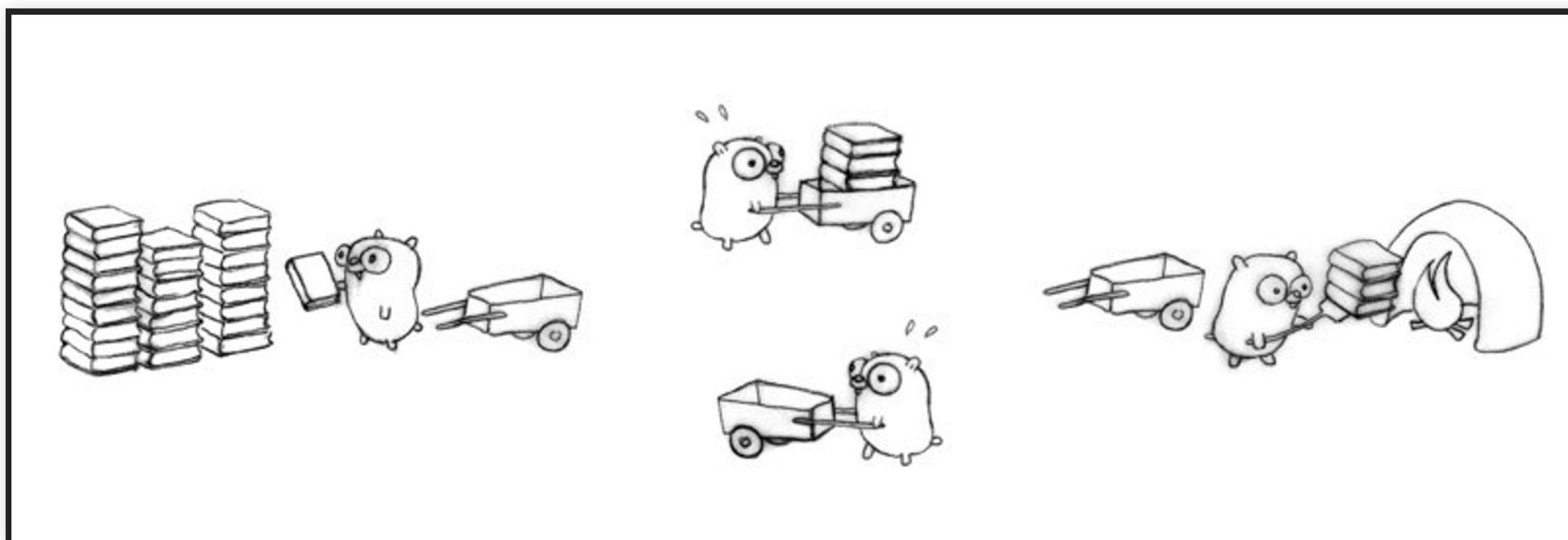

ГОРУТИНЫ

Планировщик может переключать контекст в следующих ситуациях:

- Использование ключевого слова `go`
- Сборщик мусора
- Системные вызовы
- Синхронизация

Но не обязательно, что он это сделает.

ГОРУТИНЫ



ЗАПУСК ГОРУТИНЫ

Каждая горутина имеет свой стек, но всё остальное она делит с системой.

```
go func(trolley Trolley) {  
    burnBooks(trolley)  
}(trolley)
```

```
go func() {  
    burnBooks(trolley)  
}()
```

```
go burnBooks(trolley)
```

```
go trolley.Load(pile)
```

СКОЛЬКО ТУТ ГОРУТИН?

```
func main() {  
    fmt.Printf(  
        "Goroutines: %d",  
        runtime.NumGoroutine(),  
    )  
}
```

  Когда количество горутин будет меняться?

ЧТО НАПЕЧАТАЕТ ПРОГРАММА?


N.b.: “что напечатает программа?”, это примерно то же самое, что и “что хотел сказать автор?”.

```
func main() {  
    go fmt.Printf("Hello")  
}
```

ЗАМЫКАНИЕ

Замыкание это когда внутренняя функция имеет доступ к переменным родительской функции, даже после того как родительская функция выполнена.

```
func main() {  
    for i := 0; i < 5; i++ {  
        go func() {  
            fmt.Print(i)  
        }()  
    }  
    time.Sleep(2 * time.Second)  
}
```

 А что если `runtime.GOMAXPROCS(1)`?

КАКОЙ АДРЕС БУДЕТ У I?

```
func main() {  
    for i := 0; i < 5; i++ {  
        go func() {  
            fmt.Printf("%v, %T, %v\n", i, i, &i)  
        }()  
        time.Sleep(2 * time.Second)  
    }  
}
```

КАНАЛЫ

`chan T`

- работают с конкретным типом
- потокобезопасны
- похожи на очереди FIFO

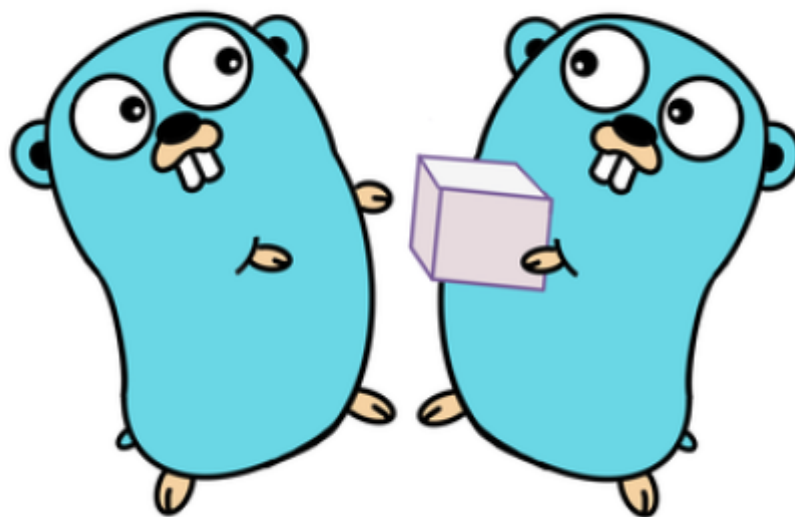
ОПЕРАЦИИ НАД КАНАЛАМИ

```
ch := make(chan int) // создать канал
ch <- 10 // записать в канал
v := <-ch // прочитать из канала
close(ch) // закрыть канал
```

 <https://habr.com/ru/post/308070/>

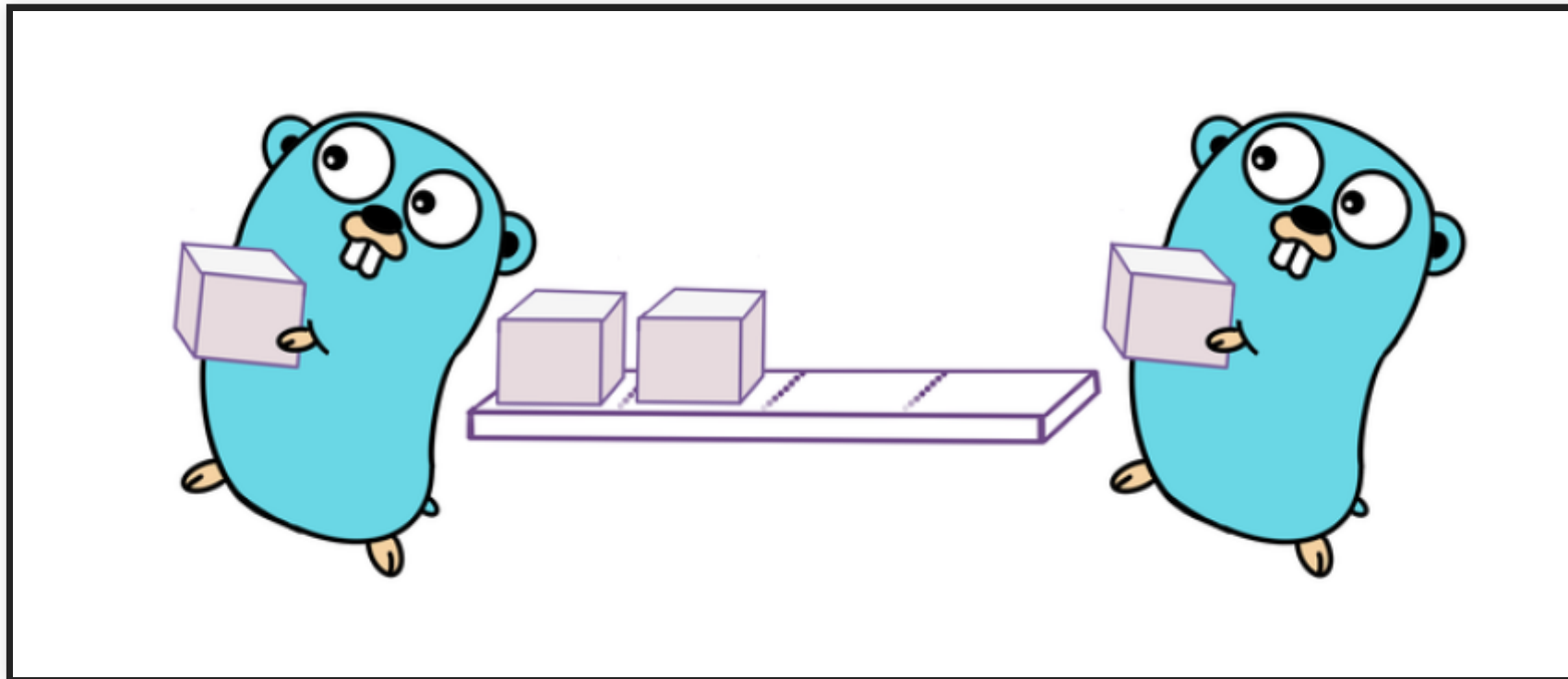
НЕБУФЕРИЗИРОВАННЫЕ КАНАЛЫ

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



БУФЕРИЗИРОВАННЫЕ КАНАЛЫ

```
ch := make(chan int, 4)
```

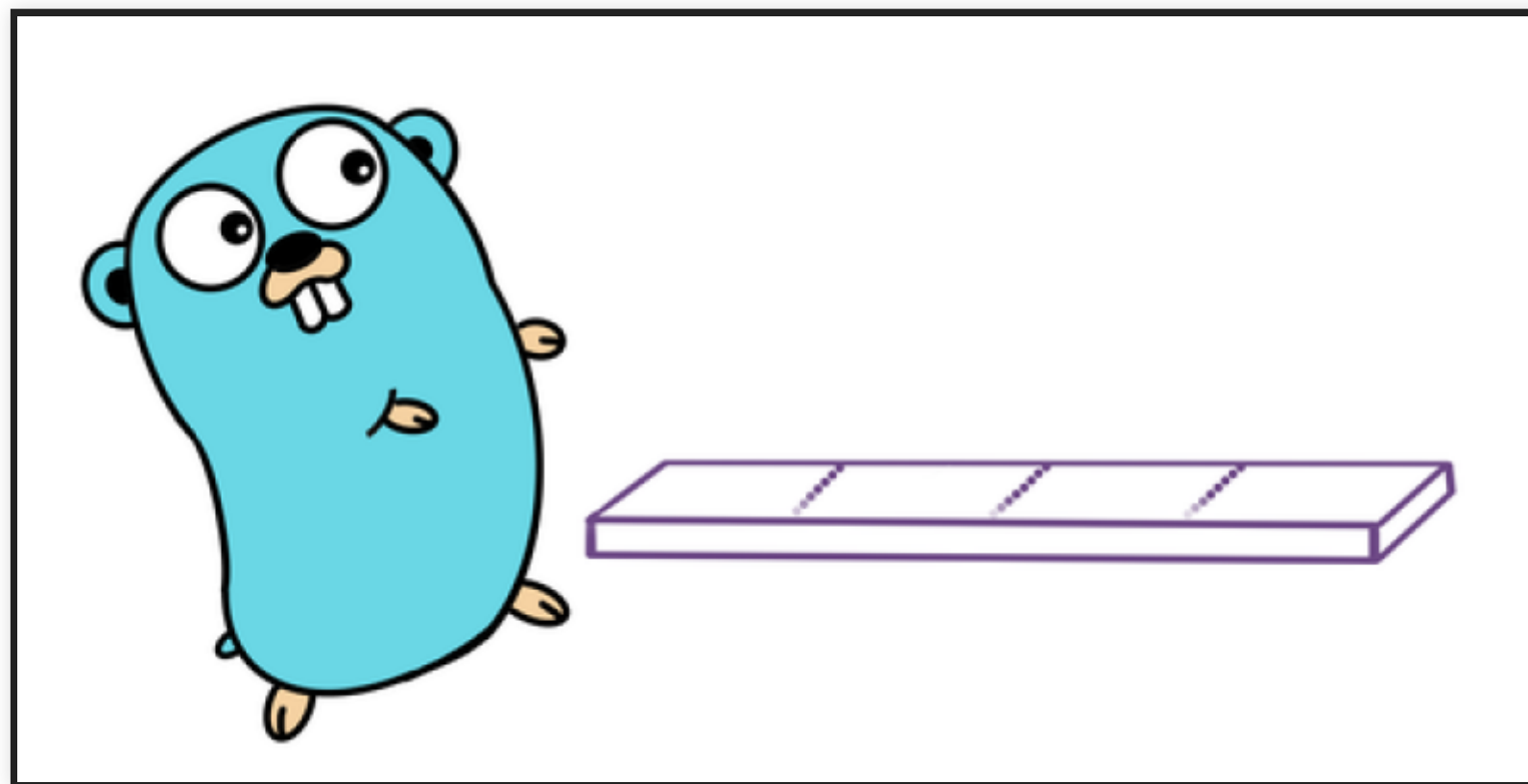


ЧЕМУ РАВЕН БУФЕР НЕБУФЕРИЗИРОВАННОГО КАНАЛА?

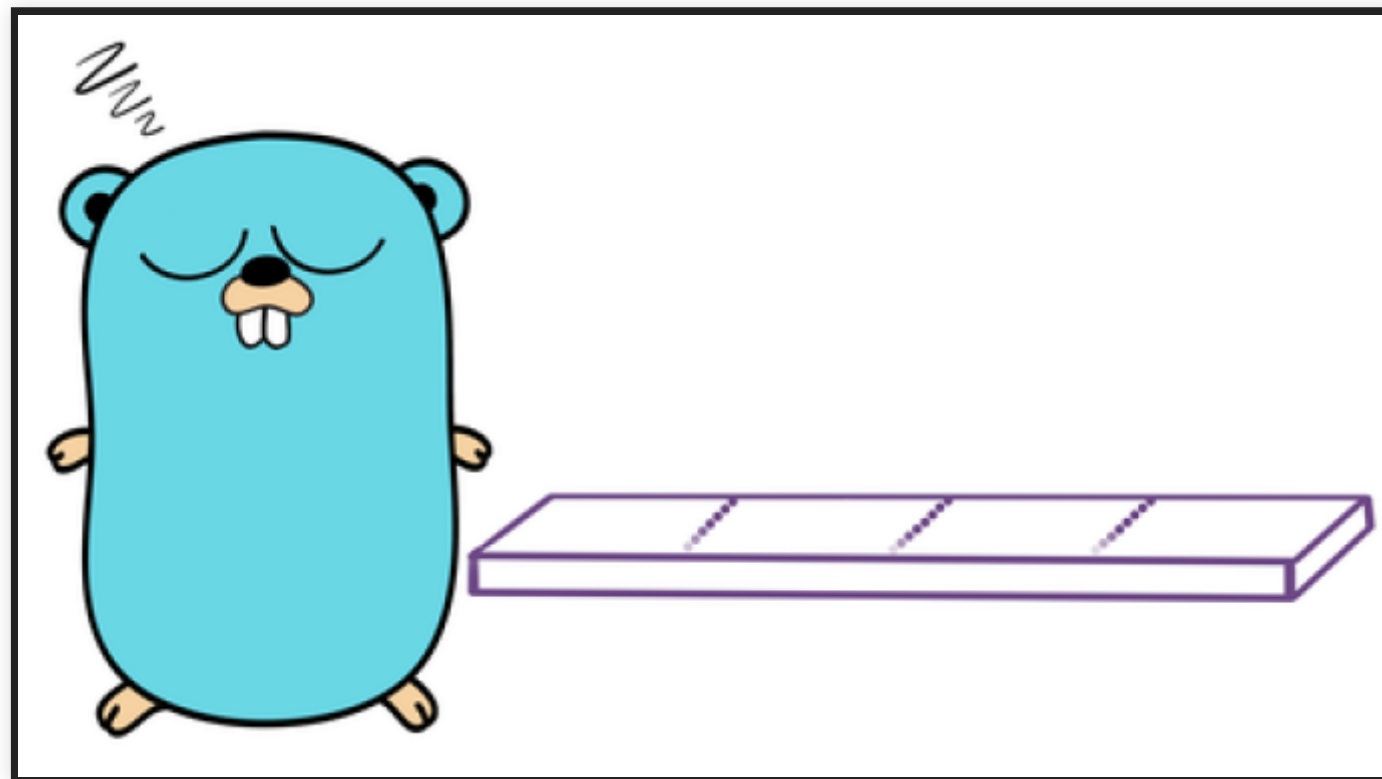
```
ch := make(chan int, ?)
```

```
func main() {  
    ch := make(chan int)  
    fmt.Println(cap(ch))  
}
```

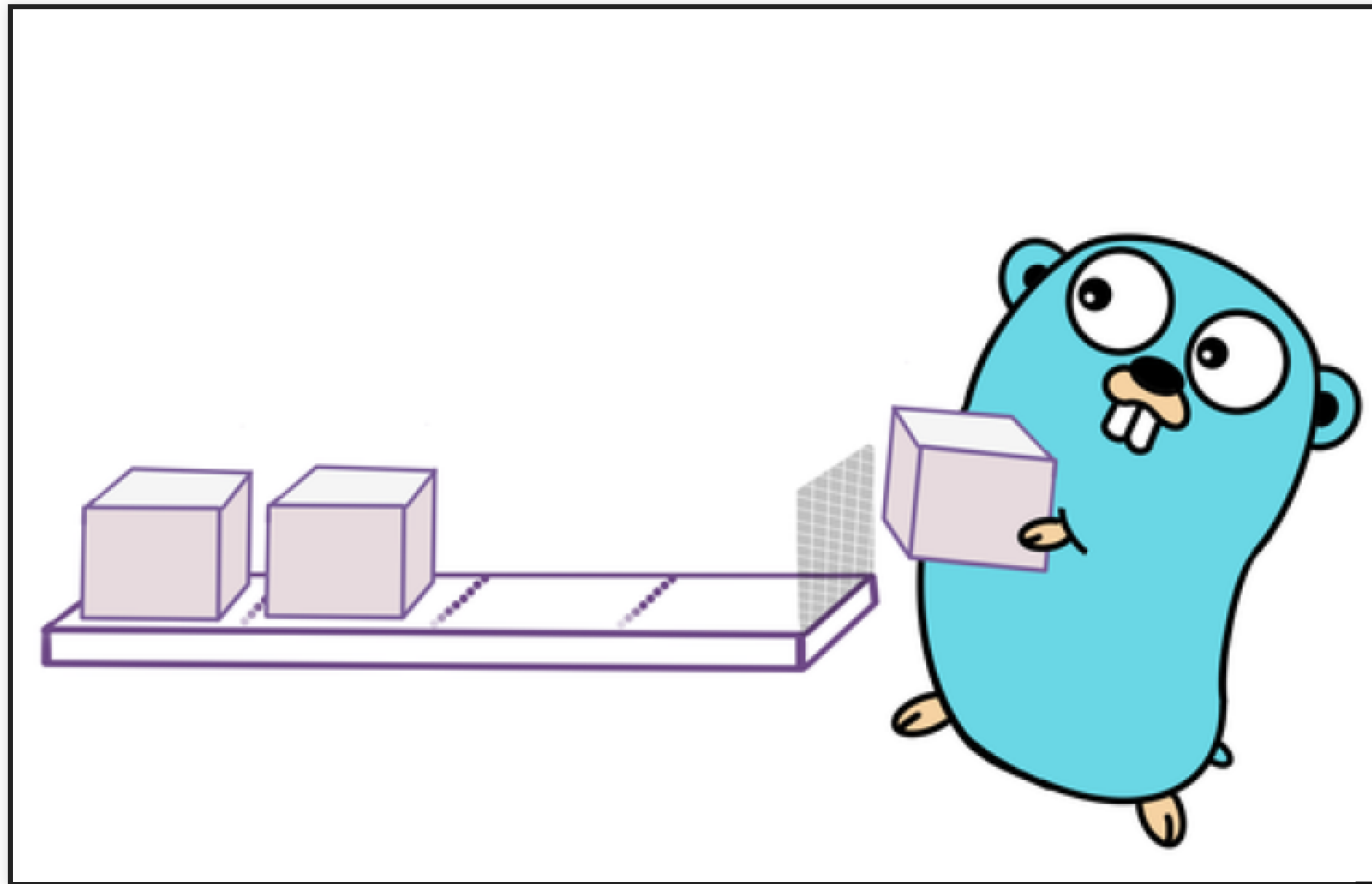
ЧТО БУДЕТ, ЕСЛИ ЧИТАТЬ ИЗ ПУСТОГО КАНАЛА?



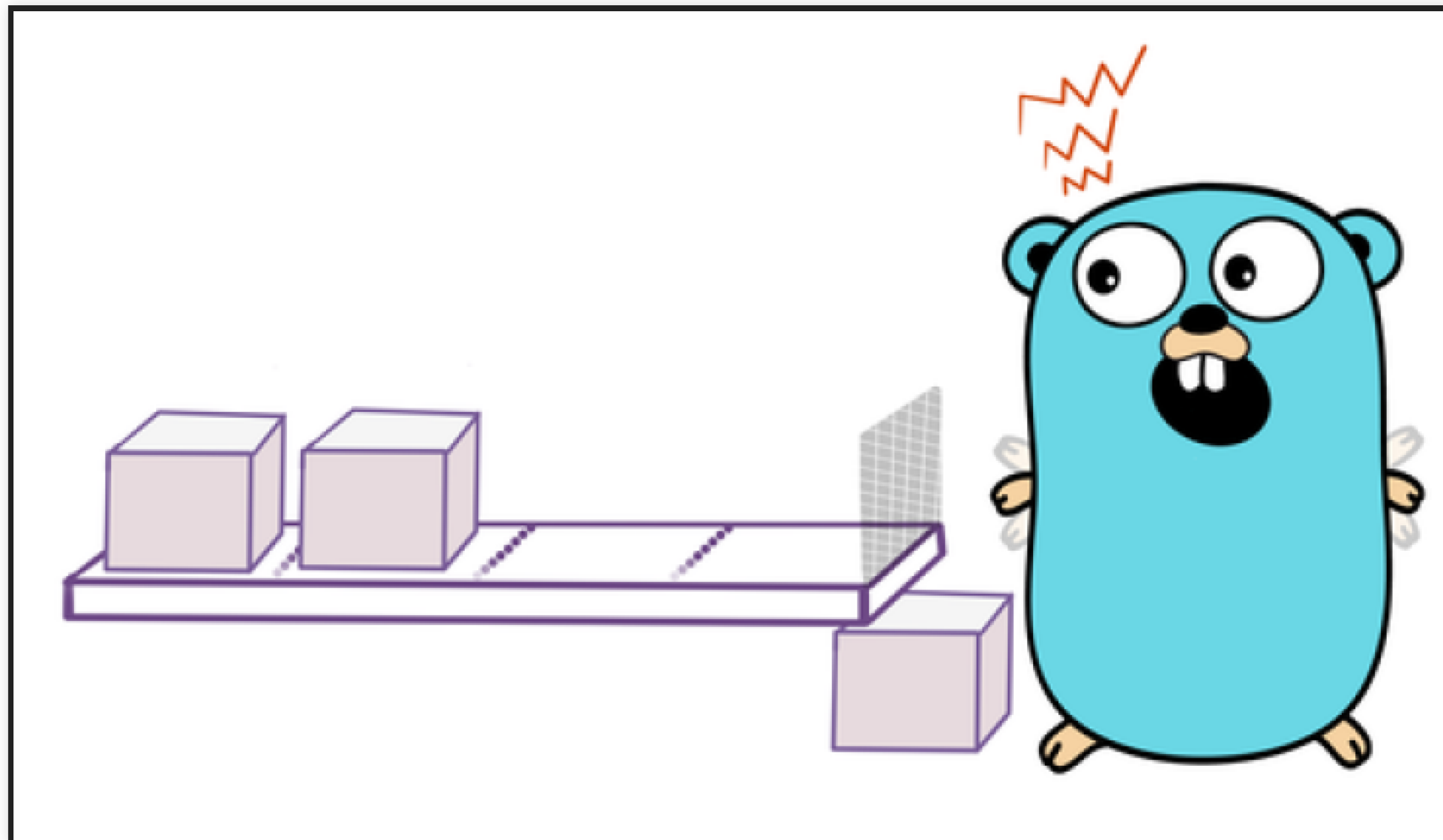
ЧТО БУДЕТ, ЕСЛИ ЧИТАТЬ ИЗ ПУСТОГО КАНАЛА?



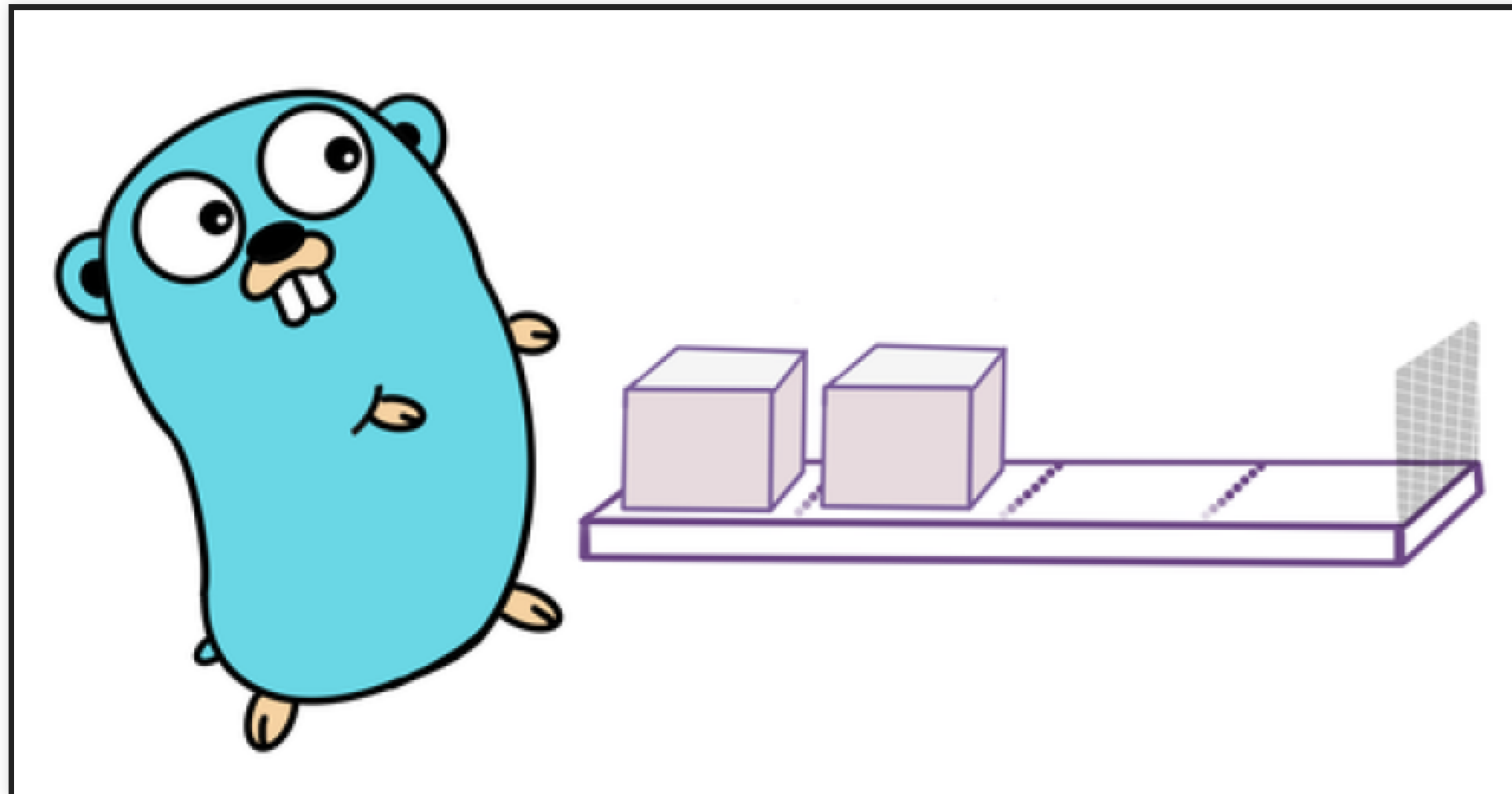
ЧТО БУДЕТ, ЕСЛИ ПИСАТЬ В ЗАКРЫТЫЙ КАНАЛ?



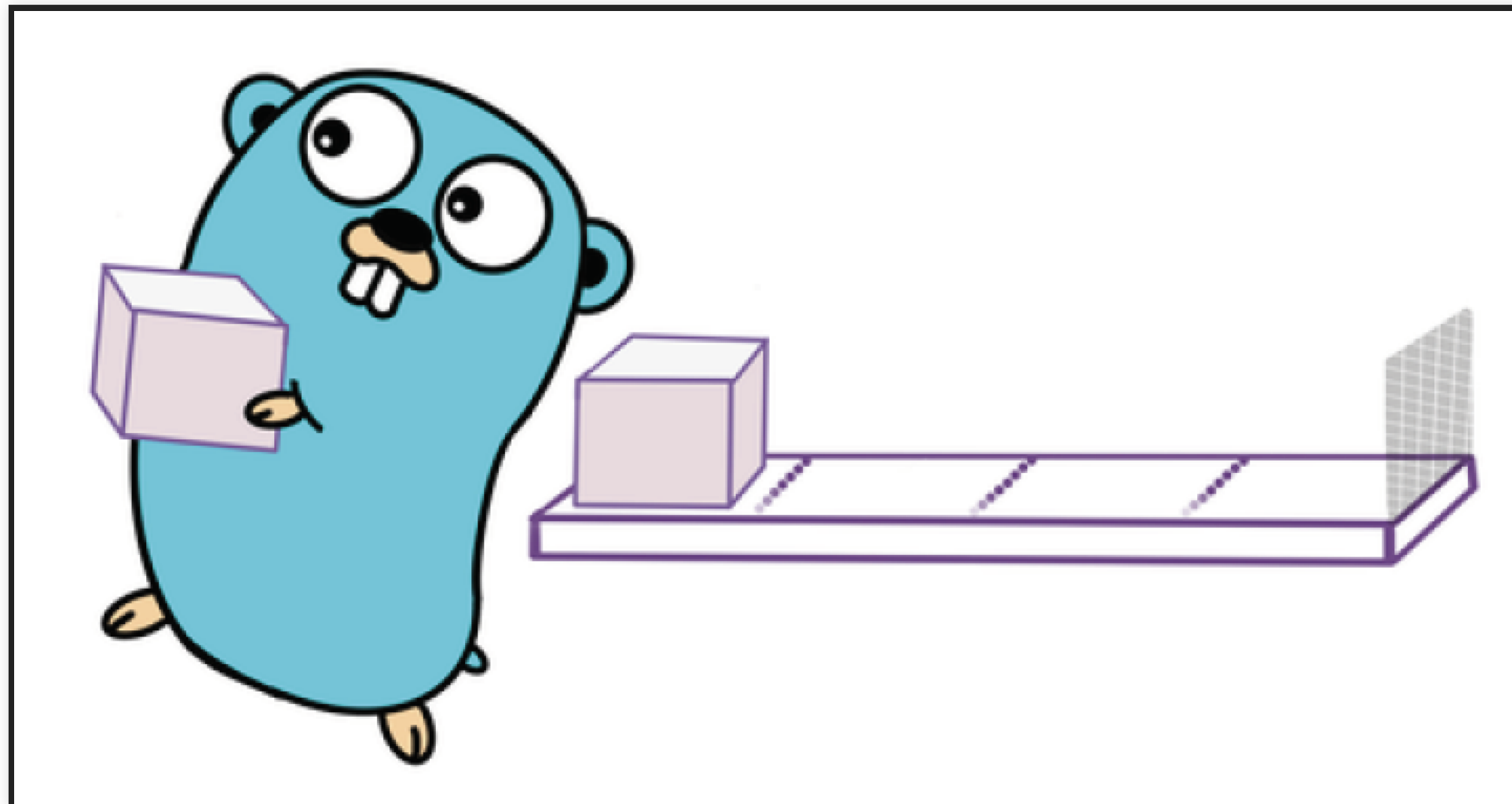
ЧТО БУДЕТ, ЕСЛИ ПИСАТЬ В ЗАКРЫТЫЙ КАНАЛ?



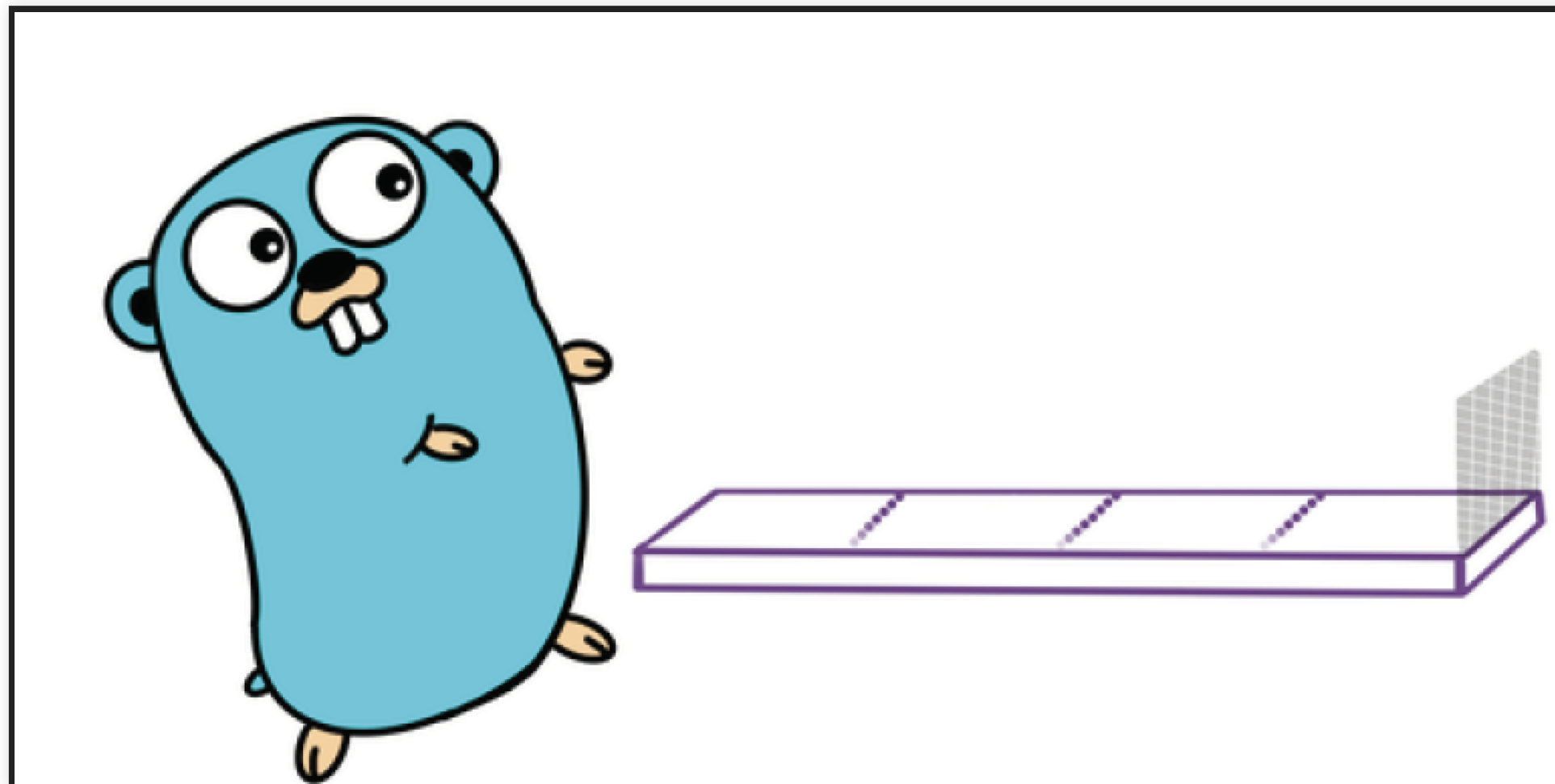
ЧТО БУДЕТ, ЕСЛИ ЧИТАТЬ ИЗ ЗАКРЫТОГО КАНАЛА?



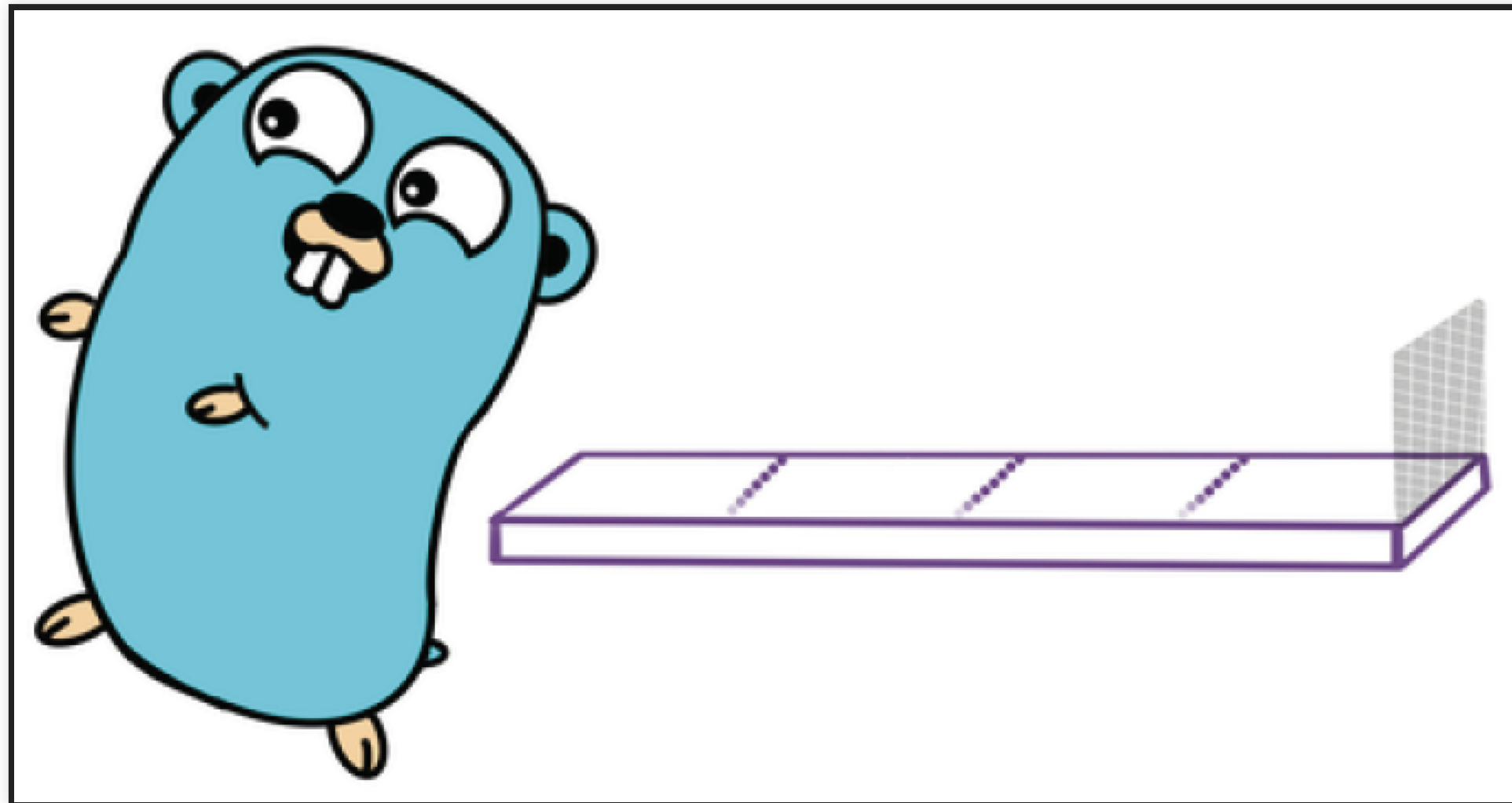
ЧТО БУДЕТ, ЕСЛИ ЧИТАТЬ ИЗ ЗАКРЫТОГО КАНАЛА?



**ЧТО БУДЕТ, ЕСЛИ ЧИТАТЬ ИЗ ПУСТОГО ЗАКРЫТОГО
КАНАЛА?**



**ЧТО БУДЕТ, ЕСЛИ ЧИТАТЬ ИЗ ПУСТОГО ЗАКРЫТОГО
КАНАЛА?**



СИНХРОНИЗАЦИЯ ГОРУТИН КАНАЛАМИ

```
func main() {  
    var ch = make(chan struct{})  
    go func() {  
        fmt.Printf("Hello")  
        ch <- struct{}{}  
    }()  
    <-ch  
}
```

```
func main() {  
    var ch = make(chan struct{})  
    go func() {  
        fmt.Printf("Hello")  
        <-ch  
    }()  
    ch <- struct{}{}  
}
```

РАБОТА С КАНАЛАМИ

- Чтение из канала, пока он не закрыт

```
v, ok := <-ch // значение и флаг "открытости" канала
```

Producer:

```
for _, t := range tasks {  
    ch <- t  
}  
close(ch)
```

Consumer:

```
for {  
    x, ok := <-ch  
    if !ok {  
        break  
    }  
    fmt.Println(x)  
}
```

РАБОТА С КАНАЛАМИ

Producer:

```
for t := range tasks {  
    ch <- t  
}  
close(ch)
```

Consumer:

```
for x := range ch {  
    fmt.Println(x)  
}
```

ПРАВИЛА ЗАКРЫТИЯ КАНАЛА

Кто закрывает канал?

- Канал закрывает тот, кто в него пишет.
- Если несколько писателей, то тот, кто создал писателей и канал.

КАНАЛЫ: ОДНОНАПРАВЛЕННЫЕ

```
chan<- T // только запись  
<-chan T // только чтение
```

Что произойдет?

```
func f(out chan<- int) {  
    <-out  
}  
func main() {  
    var ch = make(chan int)  
    f(ch)  
}
```

КАНАЛЫ: ТАЙМАУТ

```
timer := time.NewTimer(10 * time.Second)
select {
case data := <-ch:
    fmt.Printf("received: %v", data)
case <-timer.C:
    fmt.Printf("failed to receive in 10s")
}
```

КАНАЛЫ: ПЕРИОДИК

```
ticker := time.NewTicker(10 * time.Second)
defer ticker.Stop()
for {
    select {
    case <-ticker.C:
        fmt.Printf("tick")
    case <-doneCh:
        return
    }
}
```

КАНАЛЫ: КАК СИГНАЛЫ

```
make(chan struct{}, 1)
```

Источник сигнала:

```
select {  
    case notifyCh <- struct{}{}:  
    default:  
}
```

Приемник сигнала:

```
select {  
    case <-notifyCh:  
    case ...  
}
```

КАНАЛЫ: GRACEFUL SHUTDOWN

```
interruptCh := make(chan os.Signal, 1)
signal.Notify(interruptCh, os.Interrupt, syscall.SIGTERM)
fmt.Printf("Got %v...\n", <-interruptCh)
```

ПРИМИТИВЫ СИНХРОНИЗАЦИИ

1. `sync.WaitGroup`
2. `sync.Once`
3. `sync.Mutex`
4. `sync.RWMutex`

SYNC.WAITGROUP: КАКУЮ ПРОБЛЕМУ РЕШАЕТ?

```
func main() {
    const goCount = 5
    ch := make(chan struct{})
    for i := 0; i < goCount; i++ {
        go func() {
            fmt.Println("go-go-go")
            ch <- struct{}{}
        }()
    }
    for i := 0; i < goCount; i++ {
        <-ch
    }
}
```

SYNC.WAITGROUP: ОЖИДАНИЕ ГОРУТИН

```
func main() {  
    const goCount = 5  
    wg := sync.WaitGroup{}  
    wg.Add(goCount) // <===  
    for i := 0; i < goCount; i++ {  
        go func() {  
            fmt.Println("go-go-go")  
            wg.Done() // <===  
        }()  
    }  
    wg.Wait()  
}
```


SYNC.WAITGROUP: ОЖИДАНИЕ ГОРУТИН

```
func main() {  
    wg := sync.WaitGroup{}  
    for i := 0; i < 5; i++ {  
        wg.Add(1) // <===  
        go func() {  
            fmt.Println("go-go-go")  
            wg.Done() // <===  
        }()  
    }  
    wg.Wait()  
}
```

SYNC.WAITGROUP: API

```
type WaitGroup struct {  
}  
  
func (wg *WaitGroup) Add(delta int) // увеличивает счетчик WaitGroup.  
func (wg *WaitGroup) Done() // уменьшает счетчик на 1.  
func (wg *WaitGroup) Wait() // блокируется, пока счетчик WaitGroup не обнулится.
```

SYNC.ONCE: КАКУЮ ПРОБЛЕМУ РЕШАЕТ?

```
func main() {  
    var once sync.Once  
    onceBody := func() {  
        fmt.Println("Only once")  
    }  
    var wg sync.WaitGroup  
    for i := 0; i < 10; i++ {  
        wg.Add(1)  
        go func() {  
            once.Do(onceBody)  
            wg.Done()  
        }()  
    }  
    wg.Wait()  
}
```

SYNC.ONCE: ЛЕНИВАЯ ИНИЦИАЛИЗАЦИЯ (ПРИМЕР)

```
type List struct {
    once sync.Once
    ...
}

func (l *List) PushFront(v interface{}) {
    l.init()
    ...
}

func (l *List) init() {
    l.once.Do(func() {
        ...
    })
}
```

SYNC.ONCE: СИНГЛТОН (ПРИМЕР)

```
type singleton struct {  
}  
  
var instance *singleton  
var once sync.Once  
  
func GetInstance() *singleton {  
    once.Do(func() {  
        instance = &singleton{}  
    })  
    return instance  
}
```

SYNC.MUTEX: КАКУЮ ПРОБЛЕМУ РЕШАЕТ?

```
func main() {  
    wg := sync.WaitGroup{}  
    v := 0  
    for i := 0; i < 1000; i++ {  
        wg.Add(1)  
        go func() {  
            v++  
            wg.Done()  
        }()  
    }  
    wg.Wait()  
    fmt.Println(v)  
}
```

SYNC.MUTEX

```
$ GOMAXPROCS=1 go run mu.go  
1000  
$ GOMAXPROCS=4 go run mu.go  
947  
$ GOMAXPROCS=4 go run mu.go  
956
```

SYNC.MUTEX

```
func main() {  
    wg := sync.WaitGroup{}  
    mu := sync.Mutex{}  
    v := 0  
    for i := 0; i < 1000; i++ {  
        wg.Add(1)  
        go func() {  
            mu.Lock() // <===  
            v++  
            mu.Unlock() // <===  
            wg.Done()  
        }()  
    }  
    wg.Wait()  
    fmt.Println(v)  
}
```


SYNC.MUTEX: ПАТТЕРНЫ ИСПОЛЬЗОВАНИЯ

Помещайте мьютекс выше тех полей, доступ к которым он будет защищать

```
var sum struct {  
    mu sync.Mutex // <=== этот мьютекс защищает  
    i int // <=== поле под ним  
}
```

Используйте defer, если есть несколько точек выхода

```
func doSomething() {  
    mu.Lock()  
    defer mu.Unlock()  
    err := ...  
    if err != nil {  
        return // <===  
    }  
    err = ...  
    if err != nil {  
        return // <===  
    }  
    return // <===  
}
```

КОПИРОВАНИЕ МЬЮТЕКСОВ

```
type Container struct {
    sync.Mutex
    counters map[string]int
}

func (c Container) inc(name string) {
    c.Lock()
    defer c.Unlock()
    c.counters[name]++
}

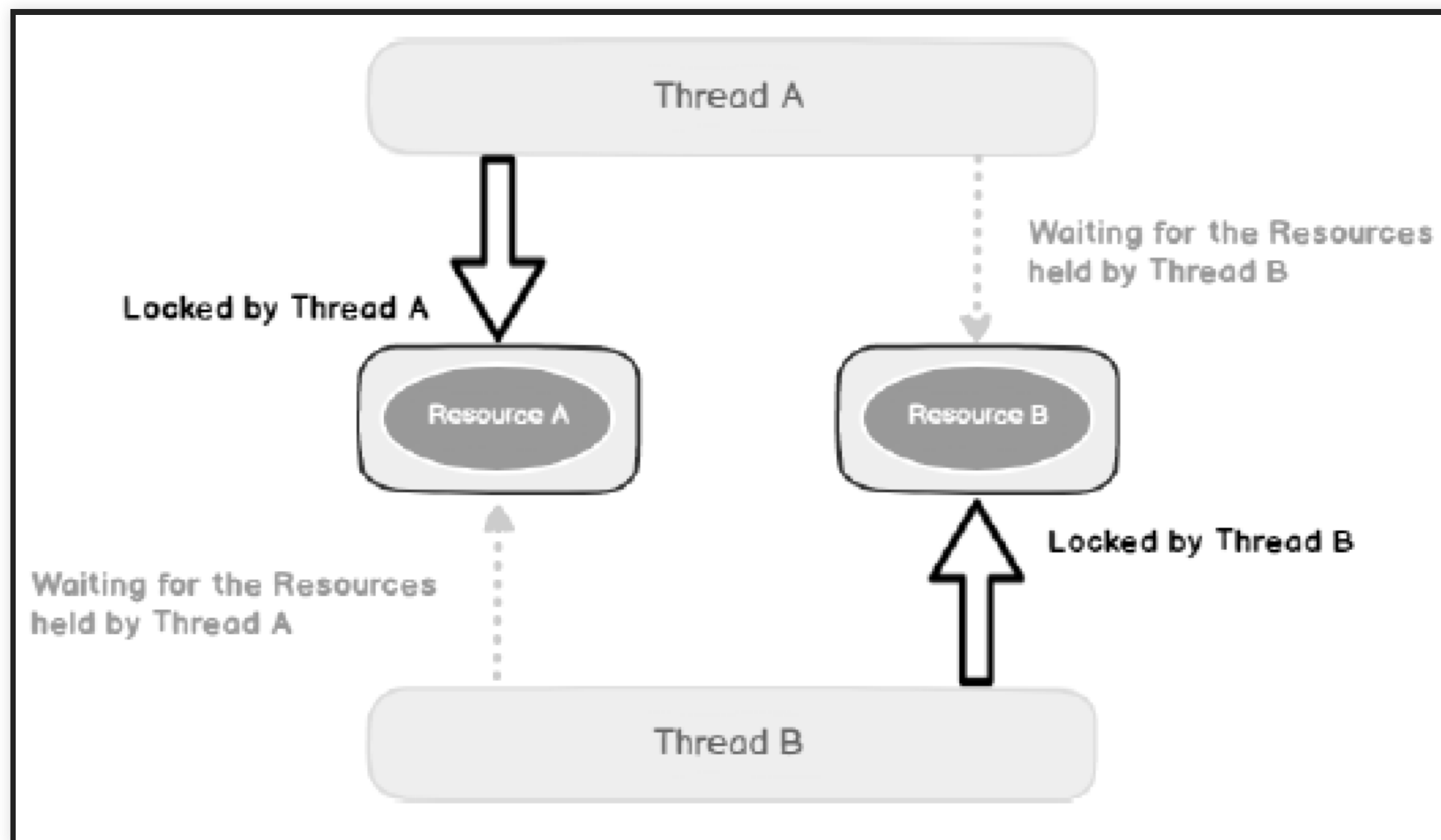
func main() {
    c := Container{counters: map[string]int{"a": 0, "b": 0}}

    doIncrement := func(name string, n int) {
        for i := 0; i < n; i++ {
            c.inc(name)
        }
    }

    go doIncrement("a", 100000)
    go doIncrement("a", 100000)

    // Wait a bit for the goroutines to finish
    time.Sleep(300 * time.Millisecond)
    fmt.Println(c.counters)
}
```

ДЕДЛОКИ



ГОНКИ ПРИОРИТЕТОВ

В чем проблема, кроме неопределенного поведения?

```
func main() {  
    wg := sync.WaitGroup{}  
    text := ""  
    wg.Add(2)  
    go func() {  
        text = "hello world"  
        wg.Done()  
    }()  
    go func() {  
        fmt.Println(text)  
        wg.Done()  
    }()  
    wg.Wait()  
}
```

ОПРЕДЕЛЕНИЕ ГОНОК

```
$ go test -race mypkg  
$ go run -race mysrc.go  
$ go build -race mycmd  
$ go install -race mypkg
```

ОПРЕДЕЛЕНИЕ ГОНОК

Ограничение race детектора:

```
func main() {  
    for i := 0; i < 10000; i++ {  
        go func() {  
            time.Sleep(time.Second)  
        }()  
    }  
    time.Sleep(time.Second)  
}
```

Можно исключить тесты:

```
// +build !race  
package foo  
// The test contains a data race. See issue 123.  
func TestFoo(t *testing.T) {  
    // ...  
}
```

CONTEXT

A Context carries a deadline, a cancellation signal, and other values across API boundaries

```
type Context interface {
    Deadline() (deadline time.Time, ok bool)
    Done() <-chan struct{}
    Err() error
    Value(key interface{}) interface{}
}
```

```
package context

var Canceled = errors.New("context canceled")

var DeadlineExceeded error = deadlineExceededError{}
// Stream generates values with DoSomething and sends them to out
// until DoSomething returns an error or ctx.Done is closed.
func Stream(ctx context.Context, out chan<- Value) error {
    for {
        v, err := DoSomething(ctx)
        if err != nil {
            return err
        }
        select {
        case <-ctx.Done():
            return ctx.Err()
        case out <- v:
        }
    }
}
```

CONTEXT

Контексты

```
context.Background()  
context.TODO()
```

- Background: возвращает ненулевой пустой контекст. Он никогда не отменяется, не имеет никаких значений и не имеет крайнего срока. Обычно он используется основной функцией, инициализацией и тестами, а также в качестве контекста верхнего уровня для входящих запросов.
- TODO: возвращает ненулевой пустой контекст. Код должен использовать `context.TODO()`, когда неясно, какой контекст использовать, или он еще недоступен (поскольку окружающая функция еще не была расширена для приема параметра контекста).

CONTEXT: WITHTIMEOUT

WithTimeout:

```
ctx, cancel := context.WithTimeout(ctx, timeout)
defer cancel()

resp, err := client.DescribeTaskV1(ctx, req)
```

CONTEXT: WITHCANCEL

```
gen := func(ctx context.Context) <-chan int {
    dst := make(chan int)
    n := 1
    go func() {
        for {
            select {
            case <-ctx.Done():
                return // returning not to leak the goroutine
            case dst <- n:
                n++
            }
        }
    }()
    return dst
}

ctx, cancel := context.WithCancel(context.Background())
defer cancel() // cancel when we are finished consuming integers

for n := range gen(ctx) {
    fmt.Println(n)
    if n == 5 {
        break
    }
}
```

CONTEXT: WITHDEADLINE

```
return WithDeadline(parent, time.Now().Add(timeout))
```

CONTEXT: WITHVALUE

```
package user
import "context"
type User struct {...}
type key int
var userKey key
func NewContext(ctx context.Context, u *User) context.Context {
    return context.WithValue(ctx, userKey, u)
}
func FromContext(ctx context.Context) (*User, bool) {
    u, ok := ctx.Value(userKey).(*User)
    return u, ok
}
```

CONTEXT: METADATA

Метаданные - частный случай для WithValue

Получение:

```
import "google.golang.org/grpc/metadata"

var user string
meta, found := metadata.FromIncomingContext(ctx)
if found {
    for key, values := range meta {
        if !strings.Contains(key, "ocp") {
            continue
        }
        if len(values) == 0 {
            continue
        }
        value := values[0]
        span.SetTag(key, value)
    }
}
```

CONTEXT: METADATA

Отправка:

```
md := metadata.New(map[string]string{
    "key1": "val1",
    "key2": "val2",
})
md := metadata.Pairs(
    "key1", "val1",
    "key1", "val1-2", // "key1" will have map value []string{"val1", "val1-2"}
    "key2", "val2",
)
ctx := metadata.NewOutgoingContext(context.Background(), md)
```

CONTEXT: METADATA

Что из себя представляет метаданные 

```
type MD map[string][]string
func FromIncomingContext(ctx context.Context) (md MD, ok bool) {
    md, ok = ctx.Value(mdIncomingKey{}).(MD)
    return
}
```

HTTP Authorization header is added as authorization gRPC request header

HTTP headers that start with 'Grpc-Metadata-' are mapped to gRPC metadata (prefixed with grpcgateway-)