Конкурентность в Go

Подготовил студент кафедры ИУ9,
Бойко Роман

Базовые сведения

- Go – язык системного программирования нового поколения.
- Был задуман в сентябре 2007 ребятами из Google и анонсирован в ноябре 2009
- Цель [illegible] – создание выразительного, высо[illegible]зного как при компиляции, так и при выполнении про[illegible]ограммирования

Немного о хороших людях

- *Робертом Грисемер*

- *Роб Пайк*

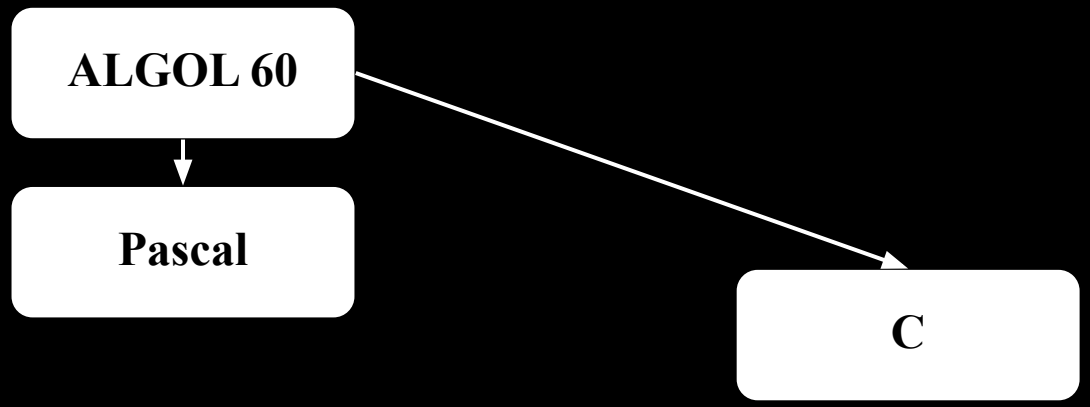
- *Кен Томпсон*

Создатели языка (слева направо): Роберт Грисемер, Роб Пайк, Кен Томпсон на Google I/O, 2012

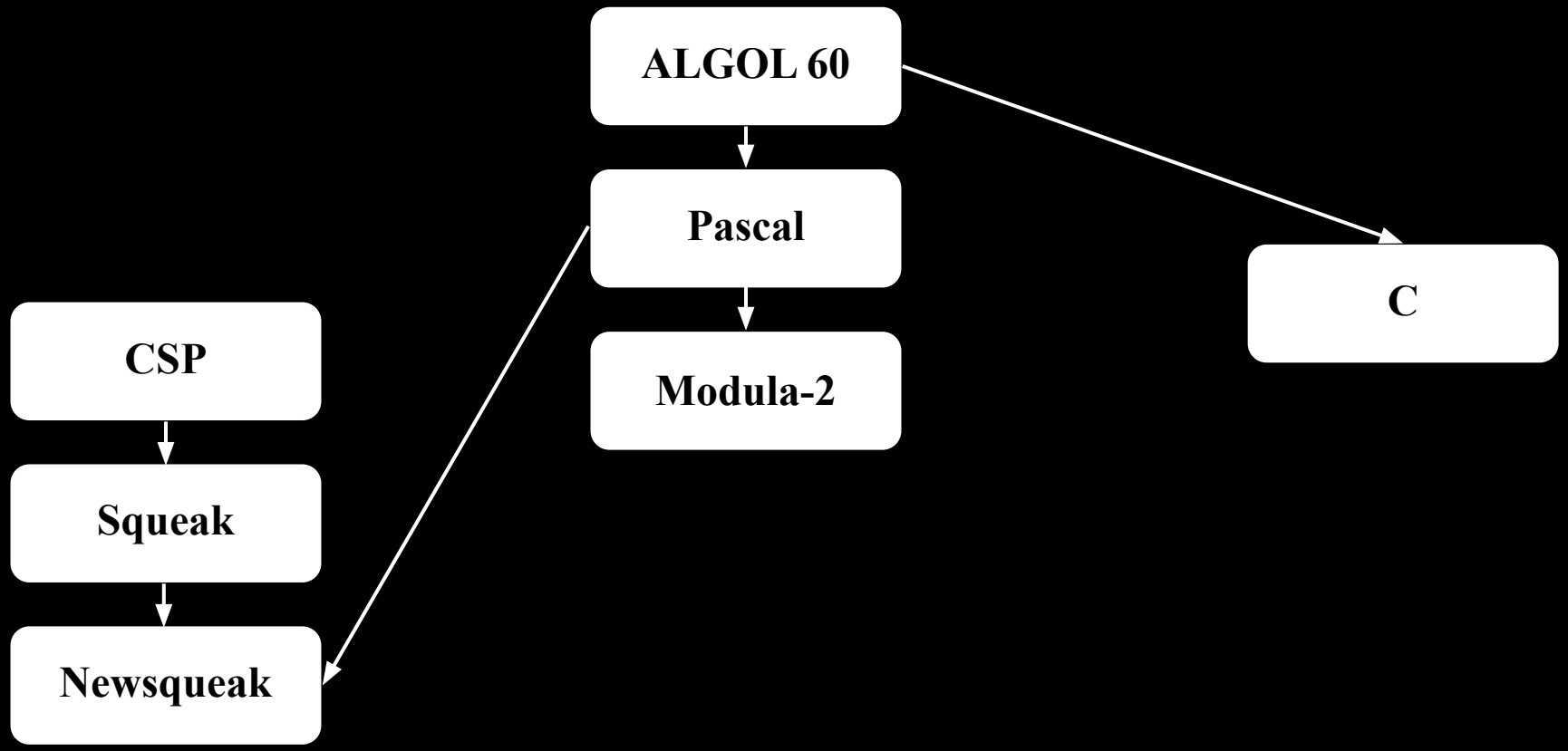
Происхождение Go

ALGOL 60

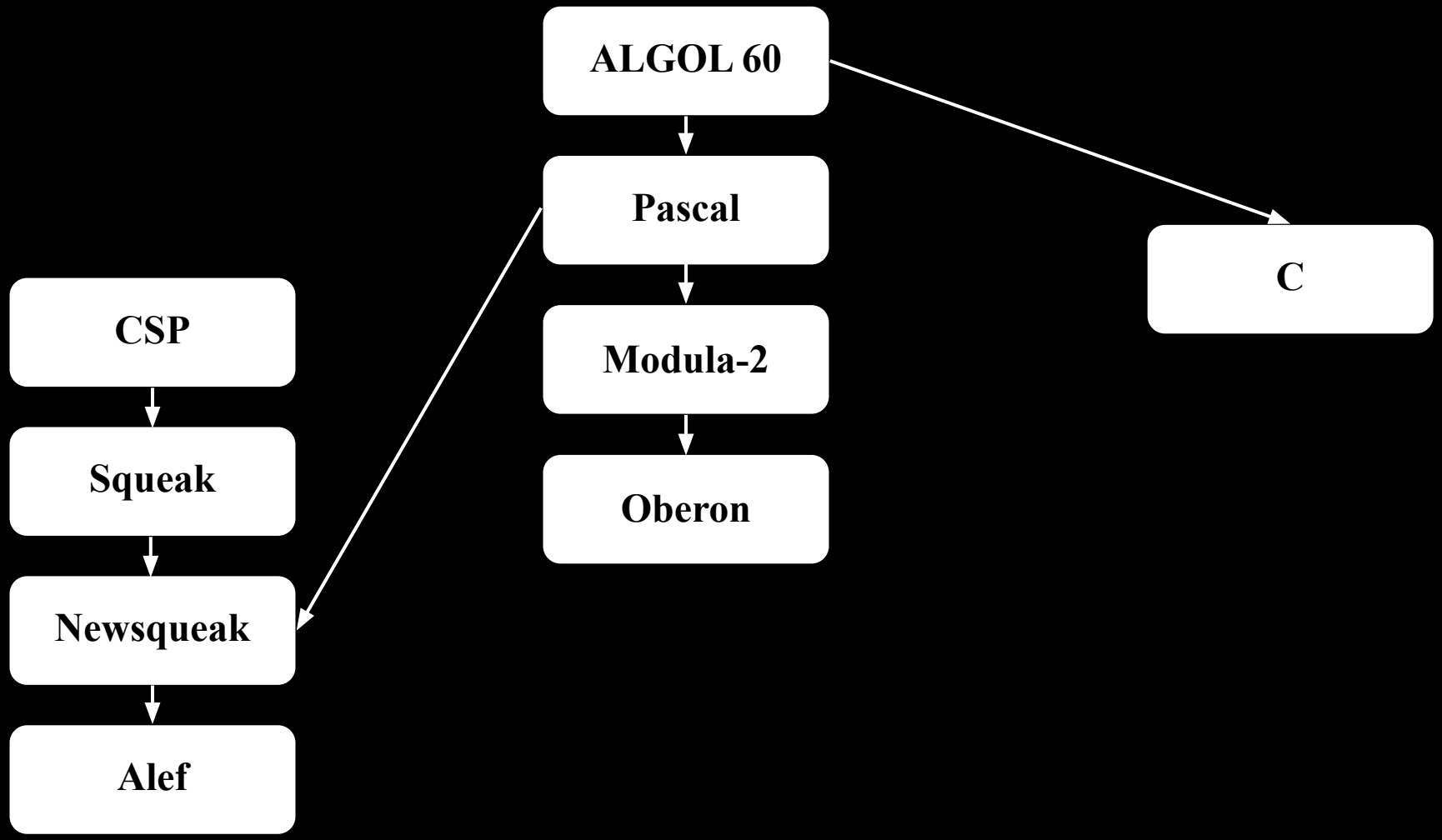
Происхождение Go



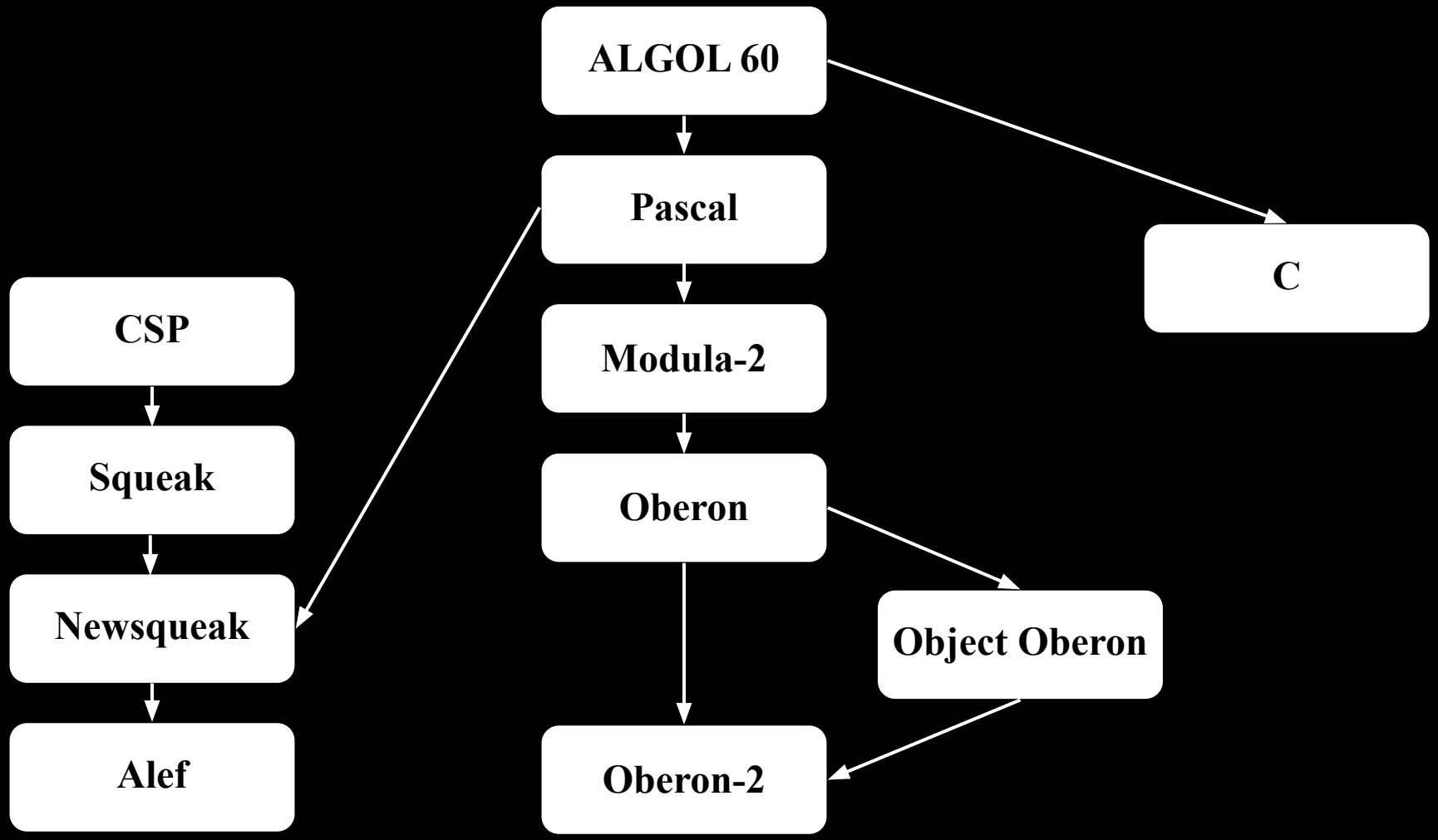
Происхождение Go



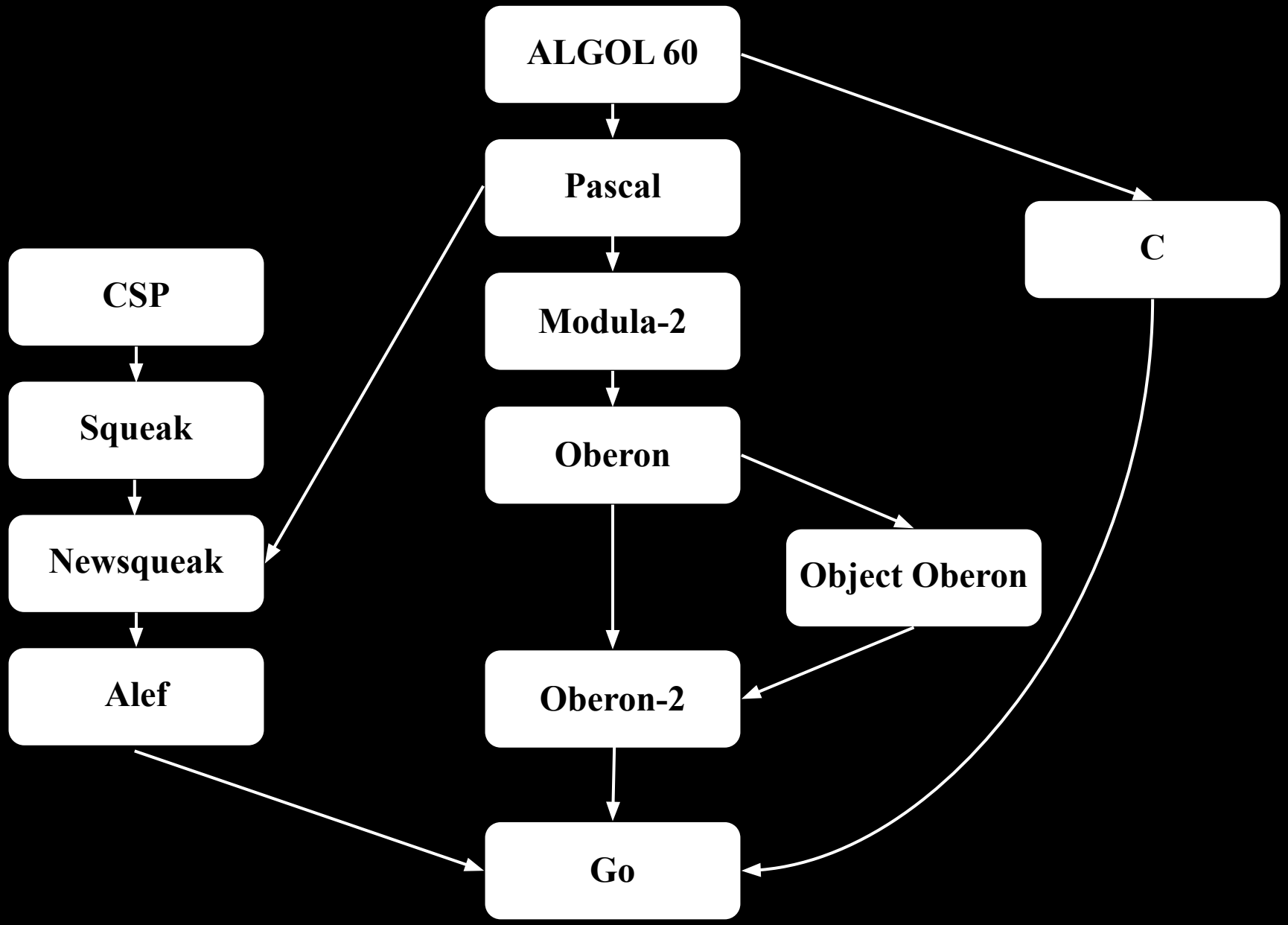
Происхождение Go



Происхождение Go



Происхождение Go

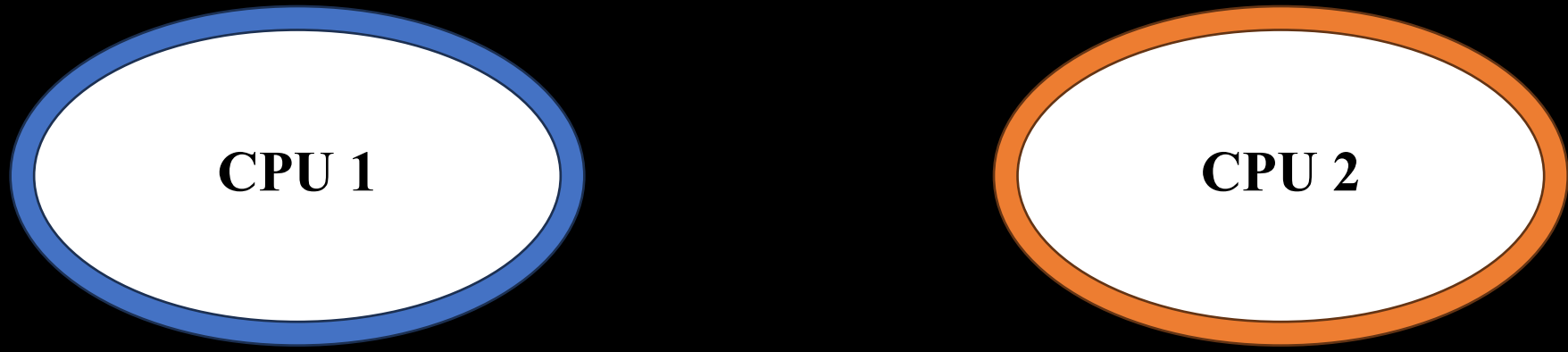


Немного подушним по теории

Давайте немного разберемся в терминах,
которые многие путают

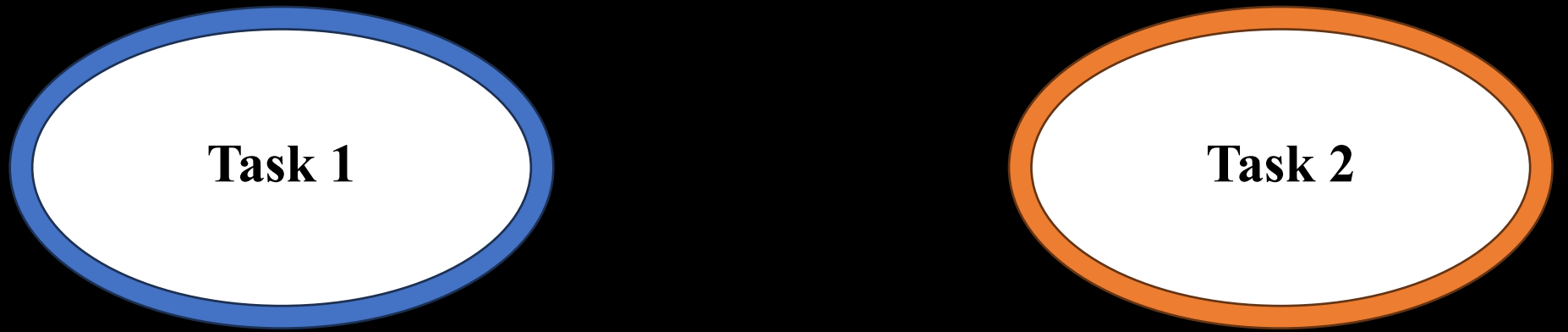
Параллелизм

- Это выполнение двух задач на двух разных исполняющих устройствах.



Конкурентность

- Это свойство системы, позволяющее ей обрабатывать несколько задач в пересекающихся периодах времени.



Многозадачность

- Это возможность операционной системы работать с несколькими задачами одновременно.

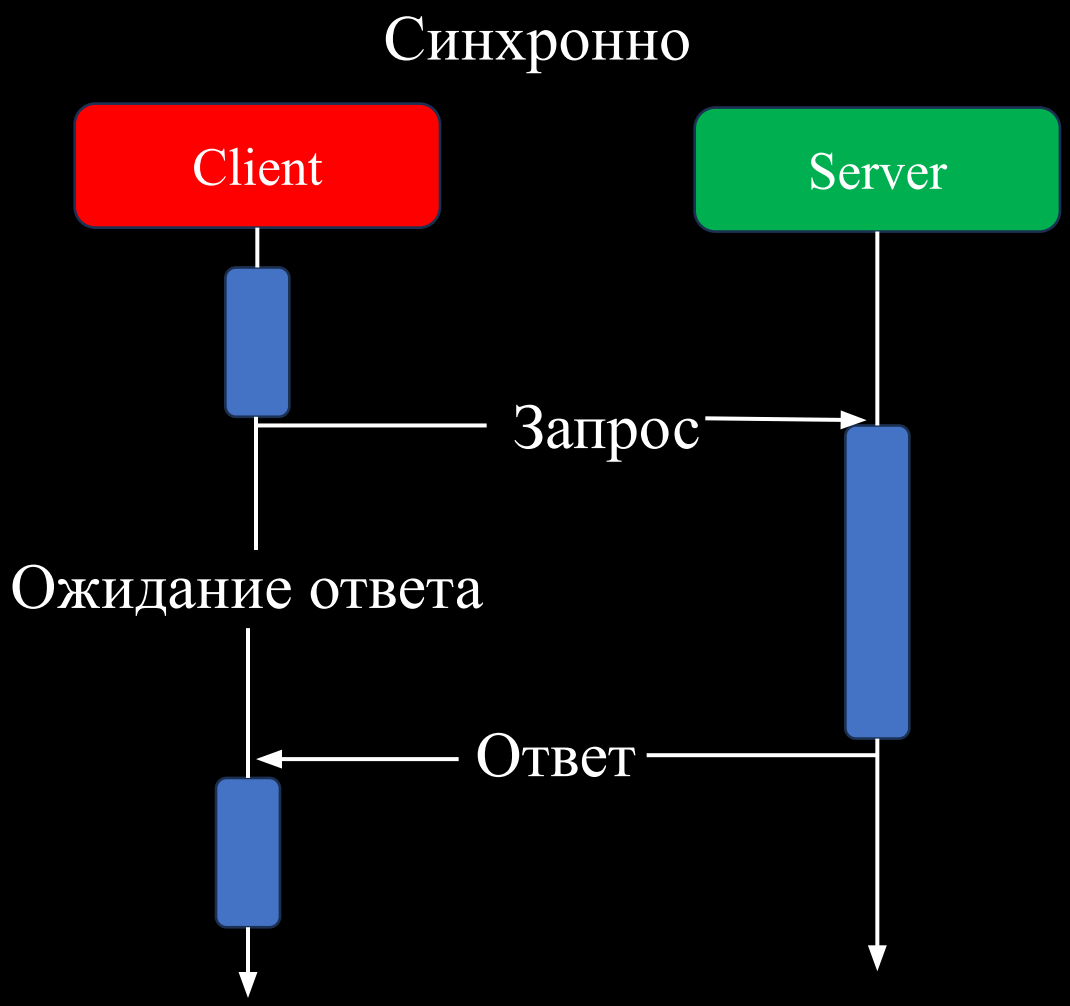
1. Вытесняющая многозадачность (Preemptive Multitasking)

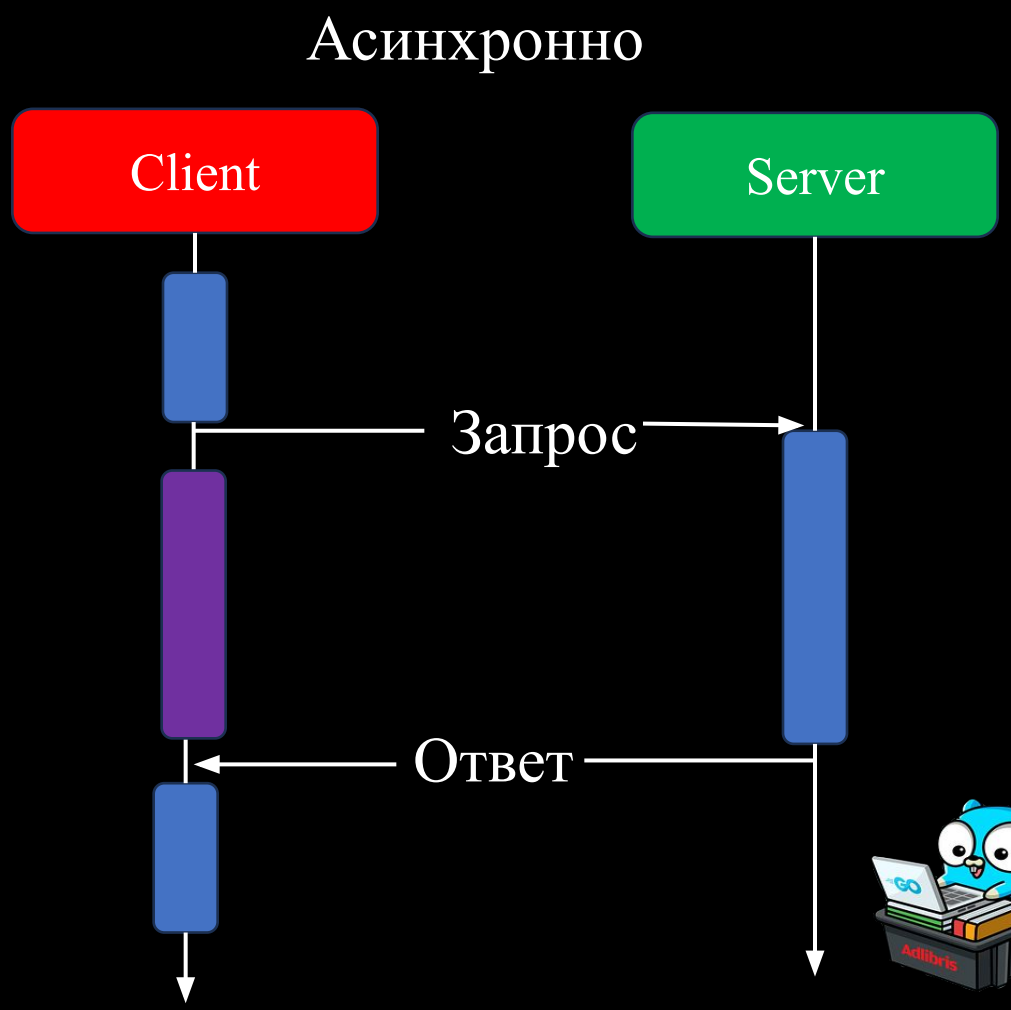
2. Кооперативная многозадачность (Cooperative Multitasking)

В Go - кооперативная многозадачность

Асинхронность

- Это концепция, которая позволяет выполнять операции или функции параллельно и не блокирует выполнение других операций.





Concurrency is not parallelism

- Concurrency is about dealing with lots of things at once.
- Parallelism is about doing lots of things at once.
- Concurrency is about structure, parallelism is about execution.
- Concurrency provides a way to structure a solution to solve a problem that may (but not necessarily) be parallelizable.

Concurrency plus communication

- Concurrency is a way to structure a program by breaking it into pieces that can be executed independently.
- Communication is the means to coordinate the independent executions.

Go поддерживает конкурентность

Go дает:

- Конкурентное выполнение (горутины)
- Синхронизацию и сообщения (каналы)
- Параллельное многозадачное управление (select)

Горутины

- **Горутиныа**— это абстракция в языке Go, которая позволяет запускать функции в асинхронном режиме.
- **Горутина** — это функция, работающая независимо в том же адресном пространстве, что и другие горутины.
- Как запуск программ с & в bash.

```
f("hello", "world") // f runs; we wait
```

```
go f("hello", "world") // f starts running
g() // does not wait for f to return
```

main – это тоже горутина

```go
package main

import (
    "fmt"
)

func doSomething() {
    fmt.Println("hello world")
}

func main() {
    go doSomething()
}
```

```go
package main

import (
    "fmt"
    "time"
)

func doSomething() {
    fmt.Println("Hello, World!")
}

func main() {
    go doSomething()
    time.Sleep(1 * time.Second)
}
```

Горутины это не потоки

- (Они чем-то похожи на потоки, но намного дешевле.)
- Горутины мультиплексируются в потоки ОС по мере необходимости.
- Когда горутина блокируется, этот поток блокируется, но никакая другая горутина не блокируется.

Планировщик в Go

GRQ

G G G

M

G

P

Core

LRQ

G G G

Каналы

- **Каналы** в Go являются мощным инструментом для обмена данными между горутинами, обеспечивая безопасность данных и избегая проблем с состоянием гонки. Каналы предоставляют средства для синхронизации и передачи данных между горутинами

```go
package main

import (
    "fmt"
)

func main() {
    ch := make(chan int)

    go func() {
        ch <- 123 // отправляем значение в канал
    }()

    val := <-ch // получаем значение из канала
    fmt.Println(val) // выводит "123"
}
```

Select

- Оператор select похож на switch, но идея основана на возможности общения, а не на равных значениях.

```
select {
case v := <-ch1:
    fmt.Println("channel 1 sends", v)
case v := <-ch2:
    fmt.Println("channel 2 sends", v)
default: // optional
    fmt.Println("neither channel was ready")
}
```

Дедлоки и Мьютексы


```go
package main

func main() {
    ch := make(chan int)
    ch <- 1 // это приведет к дедлоку,
            //так как нет другой горутины,
           //которая могла бы принять это значение
}
```

```
fatal error: all goroutines are asleep - deadlock!

goroutine 1 [chan send]:
main.main()
        /Users/nick/Desktop/VScodeProjects/Go/yandex/main.go:5 +0x34
exit status 2
```

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    data := make(map[string]int)

    go func() {
        for i := 0; i < 1000; i++ {
            data["key"] = i
        }
    }()

    go func() {
        for i := 0; i < 1000; i++ {
            fmt.Println(data["key"])
        }
    }()
    // Ожидание завершения работы горутин
    time.Sleep(time.Second)
}
```

```go
package main

import (
    "fmt"
    "sync"
    "time"
)

func main() {
    data := make(map[string]int)
    var mu sync.Mutex

    go func() {
        for i := 0; i < 1000; i++ {
            mu.Lock()
            data["key"] = i
            mu.Unlock()
        }
    }()

    go func() {
        for i := 0; i < 1000; i++ {
            mu.Lock()
            fmt.Println(data["key"])
            mu.Unlock()
        }
    }()
    // Ожидание завершения работы горутин
    time.Sleep(time.Second)
}
```

Race detector

- Race Detector (детектор гонок) в Go — это инструмент, предоставляемый компилятором Go для выявления гонок данных в параллельных программах. Гонки данных возникают, когда несколько горутин обращаются к общим данным без синхронизации, что может привести к неопределённому поведению программы.

```
go run -race имя_файла.go
```

```go
func main() {
    var wg sync.WaitGroup
    var sharedData int
    iterations := 100

    for i := 0; i < iterations; i++ {
        wg.Add(1)
        go func() {
            // Несинхронизированный доступ к общим данным
            sharedData++
            wg.Done()
        }()
    }

    wg.Wait()
    fmt.Println("Final value of sharedData:", sharedData)
}
```

```
WARNING: DATA RACE
Read at 0x00c00011e038 by goroutine 10:
  main.main.func1()
      /Users/nick/Desktop/VScodeProjects/Go/yandex/main.go:16 +0x34

Previous write at 0x00c00011e038 by goroutine 6:
  main.main.func1()
      /Users/nick/Desktop/VScodeProjects/Go/yandex/main.go:16 +0x44

Goroutine 10 (running) created at:
  main.main()
      /Users/nick/Desktop/VScodeProjects/Go/yandex/main.go:15 +0x6c

Goroutine 6 (finished) created at:
  main.main()
      /Users/nick/Desktop/VScodeProjects/Go/yandex/main.go:15 +0x6c
==================
Final value of sharedData: 97
Found 1 data race(s)
exit status 66
```

Паттерны синхронизации в Go

Паттерн	Описание
• **sync.Mutex**	Используется для защиты общих данных от конфликтов доступа.
• **sync.WaitGroup**	Используется для ожидания завершения выполнения горутин.
• **channels**	Используются для безопасной передачи данных между горутинами.

Наивный пример использования асинхронности

Таск: необходимо получить данные
с нескольких сайтов(Get)

```go
type APIResponse struct {
    URL        string // запрошенный URL
    Data       string // тело ответа
    StatusCode int    // код ответа
    Err        error  // ошибка, если возникла
}
```

```go
func GetData(
    ctxWithCancel context.Context,
    url string,
    index int,
    res []*APIResponse,
    wg *sync.WaitGroup,
) {
    defer wg.Done()
    req, err := http.NewRequestWithContext(ctxWithCancel, "GET", url, nil)
    if err != nil {
        res[index] = &APIResponse{URL: url, Err: err}
        return
    }
    client := &http.Client{}
    resp, err := client.Do(req)
    if err != nil {
        res[index] = &APIResponse{URL: url, Err: err}
        return
    }
    defer resp.Body.Close()

    body, err := io.ReadAll(resp.Body)
    if err != nil {
        res[index] = &APIResponse{url, string(body), resp.StatusCode, err}
        return
    }

    res[index] = &APIResponse{url, string(body), resp.StatusCode, nil}
}
```

```go
func FetchAPI(ctx context.Context, urls []string, timeout time.Duration) []*APIResponse {
    ctxWithCancel, cancelCtx := context.WithTimeout(ctx, timeout)
    defer cancelCtx()

    var wg sync.WaitGroup

    res := make([]*APIResponse, len(urls))

    wg.Add(len(urls))
    for i, url := range urls {
        select {
        case <-ctxWithCancel.Done():
            // Время ожидания истекло
            for j := i; j < len(urls); j++ {
                res[j] = &APIResponse{URL: urls[j], Err: ctxWithCancel.Err()}
            }
        default:
            go GetData(ctxWithCancel, url, i, res, &wg)
        }
    }

    wg.Wait()
    return res
}
```

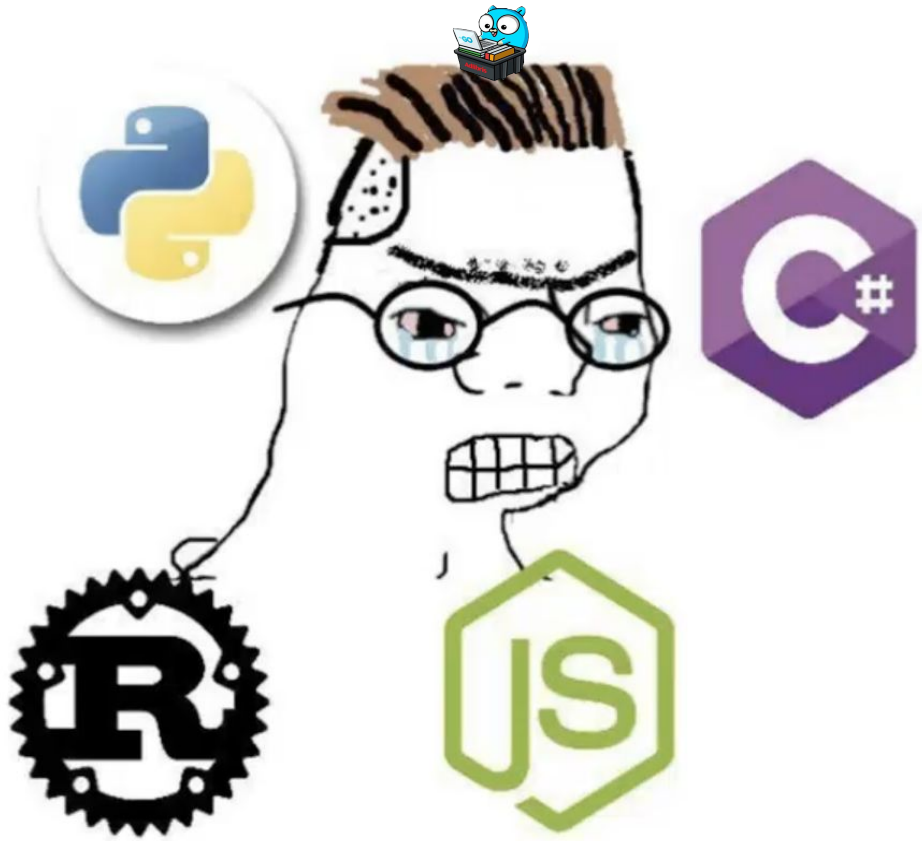
Отличие асинхронности в Go по сравнению с другими языками

- Модель Проектирования(CSP):
- Отсутствие Явного Указания Async/Await:

```python
import asyncio

async def async_example():
    print("Start")
    await asyncio.sleep(1)
    print("End")

asyncio.run(async_example())
```

```rust
use tokio::time::{sleep, Duration};

async fn async_example() {
    println!("Start");
    sleep(Duration::from_secs(1)).await;
    println!("End");
}

#[tokio::main]
async fn main() {
    async_example().await;
}
```

Goroutines


NOOOOOO!!!! YOU CAN'T JUST RUN GREEN THREADS
WITHOUT TONS OF async-awaits!!
RUNTIME MUST NOT INTERMIT INSTRUCTION STREAM!!!


haha goroutines sheduler
go brrr()

Паттерны конкурентности в Go

Паттерн	Описание
• **fan-out**	Задача делится на несколько более мелких подзадач, которые затем выполняются одновременно. Каждую подзадачу можно назначить отдельной горутине. На этом этапе рабочая нагрузка распределяется по нескольким горутинам, обеспечивая параллельную (на самом деле _concurrency) обработку.
• **fan-in**	Результаты подзадач собираются и объединяются в единый результат. На этом этапе ожидается выполнение всех подзадач и агрегирование их результатов. Этот этап также может обеспечивать синхронизацию и координацию между горутинами, чтобы обеспечить сбор всех результатов перед продолжением.
• **Worker pool**	Используется для ограничения количества одновременных задач обработки горутин. Он предполагает создание пула горутин фиксированного размера, которым можно назначать задачи. Этот шаблон полезен, когда у вас есть большое количество задач, которые необходимо обрабатывать одновременно, но вы хотите ограничить количество горутин, чтобы предотвратить истощение ресурсов.

Worker pool

```go
type WorkerPool interface {
    Start()
    Stop()
    AddWork(PoolTask)
}
```

```go
type PoolTask interface {
    Execute() error
    OnFailure(error)
}
```

```go
type MyPool struct {
    tasks       chan PoolTask
    wg          sync.WaitGroup
    isExecuting bool
    onceStart   sync.Once
    onceStop    sync.Once
    numWorkers  int
}
```

Worker pool

```go
func NewWorkerPool(numWorkers int, channelSize int) (*MyPool, error) {
    if numWorkers <= 0 {
        return nil, fmt.Errorf("incorect numWorkers")
    }
    if channelSize < 0 {
        return nil, fmt.Errorf("negative channelSize")
    }
    return &MyPool{
        tasks:       make(chan PoolTask, channelSize),
        isExecuting: false,
        numWorkers:  numWorkers,
    }, nil

}
```

Worker pool

```go
func (mp *MyPool) Start() {
    mp.onceStart.Do(func() {
        mp.wg.Add(mp.numWorkers)
        for i := 0; i < mp.numWorkers; i++ {
            go func() {
                defer mp.wg.Done()
                for pt := range mp.tasks {
                    err := pt.Execute()
                    if err != nil {
                        pt.OnFailure(err)
                    }
                }
            }()
        }
        mp.isExecuting = true
    })
}
```

```go
func (mp *MyPool) Stop() {
    mp.onceStop.Do(func() {
        mp.isExecuting = false
        close(mp.tasks)
        mp.wg.Wait()
    })
}
```

```go
func (mp *MyPool) AddWork(pt PoolTask) {
    if mp.isExecuting {
        mp.tasks <- pt
    }
}
```

Итоги

Горутины и каналы

Мультиплекси-рование ввода/вывода

Контроль над ресурсами

Преимущества асинхронности

Стандартная библиотека

Простота и понятность кода

Наши соцсети

ITS BMSTU

ITS Tech

Мой гитхаб:)