

# Сравнение: Мьютексы и Каналы в Go

Go предлагает два подхода к синхронизации и обмену данными между горутинами:

- **Мьютексы (Mutex)** — синхронизация доступа к общей памяти
- **Каналы (Channels)** — передача данных без разделяемой памяти

## ❏ Мьютекс `sync.Mutex`

Мьютекс позволяет **гарантировать, что только одна горутина** в момент времени имеет доступ к критической секции кода.

## ❏ Пример:

```
import "sync"

type Counter struct {
    m sync.Mutex
    v int
}

func (c *Counter) Inc() {
    c.m.Lock()
    c.v++
    c.m.Unlock()
}
```

## ❏ Когда использовать мьютекс:

- Множественные горутины читают/пишут переменную
- Работа с `map`, `slice` без гонки данных

- Критический код, который нельзя выполнять одновременно
- 

## □□ Канальchan)

Каналы передают данные между горутинами. Это позволяет **избежать прямого доступа к разделяемой памяти**.

## □ Пример:

```
func worker(jobs <-chan int, results chan<- int) {
    for job := range jobs {
        results <- job * 2
    }
}

func main() {
    jobs := make(chan int, 5)
    results := make(chan int, 5)

    go worker(jobs, results)

    jobs <- 1
    jobs <- 2
    close(jobs)

    fmt.Println(<-results)
    fmt.Println(<-results)
}
```

## □□ Когда использовать каналы:

- Поточковая обработка данных
  - Логика “продюсер/консьюмер”
  - Синхронизация начала/окончания работы
  - Архитектура “fan-in”, “fan-out”
-

# ⚖ Сравнение

Критерий	Мьютексы	Каналы
Принцип	Общая память	Сообщения
Сложность	Низкая, но с рисками	Чище, но требует дизайна
Поддержка <code>pprof</code>	Да (через <code>mutex</code> )	Частично (горутины)
Предотвращение гонок	Да	Да
Применение	Быстрый доступ	Асинхронные операции

## ☐☐ Рекомендация от Go-разработчиков

“Не общайся через общую память — делись памятью через общение.”

Используй **каналы**, если можешь выразить логику через них, и **мьютексы**, когда нужна низкоуровневая производительность.

## ☐☐ Итог

Хочешь максимальную скорость	Используй `sync.Mutex`
Хочешь безопасный обмен	Используй `chan`
Хочешь лучшее из двух миров	Используй `sync.Map`, `sync.Once`, `sync.WaitGroup` при необходимости

# □□ Дополнительные инструменты

## из `sync`

Go также предоставляет удобные примитивы для управления конкурентностью:

---

### `[sync.Once` — выполнить только один раз

Позволяет гарантировать, что определённый код будет выполнен **только один раз**, даже если вызывается из нескольких горутин.

```
var once sync.Once

func initConfig() {
    once.Do(func() {
        fmt.Println("Конфигурация инициализирована")
    })
}
```

Используется для ленивой инициализации, подключения к БД, загрузки конфигов.

---

### `[sync.WaitGroup` — дождаться завершения горутин

Позволяет дождаться, пока все горутин завершатся.

```
var wg sync.WaitGroup

for i := 0; i < 5; i++ {
    wg.Add(1)
    go func(id int) {
        defer wg.Done()
        fmt.Printf("Горутин %d завершена", id)
    }(i)
}
```

```
}

wg.Wait()

fmt.Println("Все горютины завершились")
```

## [sync.Map — потокобезопасная map

Альтернатива обычной `map`, не требующая ручной синхронизации.

```
var sm sync.Map

sm.Store("foo", 42)
val, ok := sm.Load("foo")
if ok {
    fmt.Println("Значение:", val)
}

sm.Range(func(key, value any) bool {
    fmt.Printf("%v = %v\n", key, value)
    return true
})
```

Подходит для кэширования, счётчиков, безопасной общей памяти между горютинами.

## 📦 Вывод:

Инструмент	Назначение
<code>sync.Mutex</code>	Защита разделяемых данных
<code>sync.Once</code>	Инициализация кода только один раз
<code>sync.WaitGroup</code>	Ожидание завершения группы горютин
<code>sync.Map</code>	Потокобезопасный ассоциативный массив

Revision #1

Created 27 June 2025 19:21:54 by Admin

Updated 27 June 2025 19:25:51 by Admin