GO

- Информация по append в Go
- Как устроен планировщик горутин в Go: G-M-P модель
- Сравнение: Мьютексы и Каналы в Go
- Поведение for range с каналами в Go
- Пример потока данных в Clean Architecture на Go

Информация по append в Go

`append` — одна из самых важных встроенных функций в Go для работы со слайсами. Ниже рассмотрены все возможные способы использования, примеры и пояснения.

□□ Базовый синтаксис

```
append(slice, elems...)
```

- `slice` — срез, к которому добавляем
- `elems...` — один или несколько элементов **или другой срез**

□ Варианты использования

1. Добавить один элемент

```
nums := []int{1, 2}
nums = append(nums, 3)
// → [1 2 3]
```

---

2. Добавить несколько элементов

```
nums = append(nums, 4, 5, 6)
// → [1 2 3 4 5 6]
```

---

3. Добавить другой срез

```
more := []int{7, 8}
nums = append(nums, more...) // обязательно `...`!
```

---

4. Добавить пустой срез

```
nums = append(nums, []int{}...) // всё ещё [1 2 3 4 5 6 7 8]
```

---

5. Добавить к `nil`-срезу

```
var s []int
s = append(s, 10) // работает, даже если s == nil
```

---

6. Удалить элемент (по индексу)

```
i := 2
s = append(s[:i], s[i+1:]...)
```

□□ Как работае`append`

- Если у слайса **есть свободная capacity** — элементы просто добавляются.
- Если нет — Go **создаёт новый массив**, копирует старые значения и добавляет новые.
- Поэтому `append` может **изменить underlying array**.

□□ Трюки `append`

Объединение слайсов

```
a := []string{"a", "b"}
b := []string{"c", "d"}
a = append(a, b...) // → [a b c d]
```

Удаление последнего элемента

```
s = s[:len(s)-1]
```

⚠ Важно помнить

- `append` **всегда возвращает новый слайс** (возможно тот же, возможно новый)
- Обязательно `...`, если добавляешь другой срез: `append(s, other...)`
- Работает **только со слайсами**, не с массивами

Сводка

Использование	Пример
Один элемент	`append(s, 10)`
Несколько элементов	`append(s, 1, 2, 3)`
Добавить срез	`append(s, other...)`
Удалить элемент	`append(s[:i], s[i+1:]...)`
Работать с `nil`-срезом	`append(nil, 1)`

Как устроен планировщик горутин в Go: G-M-P модель

Go использует уникальную модель управления конкурентностью — **G-M-P** (Goroutine, Machine, Processor), которая обеспечивает масштабируемость, высокую производительность и лёгкость работы с параллелизмом.

□□ Компоненты Go-планировщика

Обозначение	Название	Что это такое
G	Goroutine	Логическая единица выполнения (код + стек)
M	Machine (Thread)	Системный поток (OS thread)
P	Processor	Планировщик, управляющий выполнением G

□□ Как это работает?

- **M (Thread)** — физический поток, выполняющий код
- **P (Processor)** — абстрактный процессор: даёт M-у задачу (G)
- **G (Goroutine)** — логическая задача, которую нужно выполнить

> M нужен P, чтобы выполнять G. Без P поток (M) простаивает.

□□ Модель в действии

При запуске:

```
runtime.GOMAXPROCS(4)
```

Ты создаёшь 4 **P (процессора)**. Это означает, что Go может **одновременно** запускать до 4 горутин **в реальном параллелизме**.

□□ Визуально:

```
+-------------+       +----------+       +------------------+
| Goroutine G |   -->  |   P      |  -->   | Thread M (OS)    |
+-------------+       +----------+       +------------------+
      ^                 |
      |_________________|
        Work stealing (если P простаивает)
```

□ Work-stealing

Если один P (процессор) простаивает, он может “украсть” G из очереди другого P. Это делает систему ещё более гибкой и **минимизирует простои**.

□□ Примеры поведения

Сценарий	Как себя ведёт Go
1 млн горутин	Go спокойно справится
Ожидание по `time.Sleep`	P отходит от M и отдаёт другим
Блокировка I/O	M блокируется, но P берёт новый M

Сценарий	Как себя ведёт Go
`runtime.GOMAXPROCS(n)`	Ограничивает кол-во одновременно исполняемых G

Вывод:

Go-планировщик:

- работает по **G-M-P** модели
- очень эффективно управляет тысячами горутин
- масштабируется лучше, чем системы, построенные на потоках ОС
- делает конкурентное программирование простым и надёжным

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

Мьютексы (`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

Канал (`chan`)

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:

- Потоковая обработка данных
- Логика “продюсер/консюмер”
- 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 `pprof`	Да (через `mutex`)	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 горутины завершатся.

```
var wg sync.WaitGroup

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

```
}

wg.Wait()
fmt.Println("Все горутины 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
", key, value)
    return true
})
```

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

□□ Вывод:

Инструмент	Назначение
`sync.Mutex`	Защита разделяемых данных
`sync.Once`	Инициализация кода только один раз
`sync.WaitGroup`	Ожидание завершения группы горутин
`sync.Map`	Потокобезопасный ассоциативный массив

Поведение for range с каналами в Go

В Go `for v := range ch` используется для последовательного чтения из канала `ch`. Цикл завершится **только** после того, как:

- канал будет **закрыт**, и
- все значения из него будут **прочитаны**

□□ Сценарии поведения

1. Канал пуст и **не закрыт**

```
ch := make(chan string)

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

□□ Итог**блокировка навсегда**, цикл ждёт значения, но никто не пишет.

---

2. В канале есть одно значение, канал **не закрыт**

```
ch := make(chan string)

go func() {
    ch <- "hello"
}()
```

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

Итог: `hello` выведется, затем цикл **зависнет**, ожидая следующего значения.

---

3. В канале есть значение, потом ещё одно, затем закрытие

```
ch := make(chan string)

go func() {
    ch <- "one"
    time.Sleep(1 * time.Second)
    ch <- "two"
    close(ch)
}()

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

Итог:

- `one` будет прочитан
- цикл подождёт
- `two` будет прочитан
- после `close(ch)` — цикл завершится

Правильное завершение горутины через канал и WaitGroup

```
import (
    "fmt"
    "sync"
)

func worker(id int, jobs <-chan int, wg *sync.WaitGroup) {
    defer wg.Done()
    for job := range jobs {
        fmt.Printf("Worker %d processing job %d\n", id, job)
    }
}

func main() {
    jobs := make(chan int)
    var wg sync.WaitGroup

    for w := 1; w <= 2; w++ {
        wg.Add(1)
        go worker(w, jobs, &wg)
    }

    for j := 1; j <= 5; j++ {
        jobs <- j
    }

    close(jobs)  // □□ важно закрыть канал
    wg.Wait()    // □□ ждём завершения всех воркеров
}
```

□□ Общее поведени`range` по каналам

Сценарий	Поведение
Канал пуст и не закрыт	□ Блокировка
Есть 1 значение, канал не закрыт	□ Прочтёт 1, потом зависнет

Сценарий	Поведение
Есть значения, канал закрыт после них	Прочтёт все, завершит цикл
Канал закрыт сразу	Завершит цикл (если пуст)

Рекомендации

- **Закрывай канал**, если больше не будет отправок
- Используй `sync.WaitGroup`, чтобы ждать завершения горутин
- ⚠ Избегай чтения из **открытых, но не используемых** каналов

Пример потока данных в Clean Architecture на Go

Предисловие

Clean Architecture (Чистая Архитектура), предложенная Робертом Мартином (Uncle Bob), — это подход к проектированию ПО, который подчеркивает разделение на слои с четкими зависимостями.

Основная идея: сделать систему независимой от фреймворков, UI, баз данных и внешних сервисов. Зависимости направлены “внутрь” — внешние слои зависят от внутренних, но не наоборот.

Ключевые слои Clean Architecture:

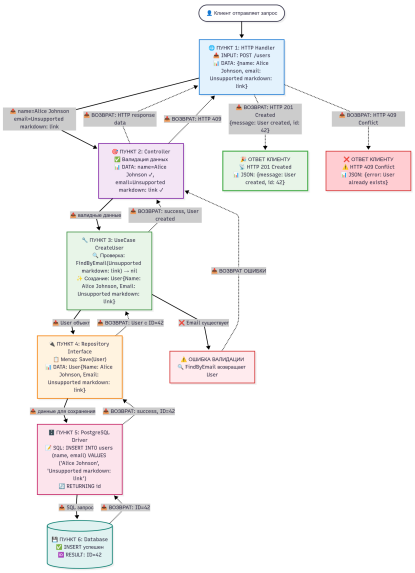
- **Доменный слой (Domain/Entities)**: Содержит бизнес-сущности и бизнес-правила. Это “ядро” системы, не зависящее от внешнего мира.
- **Прикладной слой (Application/Use Cases)**: Описывает сценарии использования — бизнес-логику, которая оркеструет сущности. Зависит от домена, но не от инфраструктуры.
- **Слой адаптеров (Adapters/Controllers, Presenters)**: Обеспечивает взаимодействие с внешним миром (например, HTTP, CLI). Преобразует внешние запросы в вызовы Use Cases.
- **Инфраструктурный слой (Infrastructure/Frameworks & Drivers)**: Реализует детали, такие как доступ к БД, API-клиенты. Зависит от внутренних слоев через интерфейсы (инверсия зависимостей).

Практический пример: Создание пользователя

Сценарий: Пользователь регистрируется, отправляя HTTP POST-запрос на `/users` с JSON-телом:

```
{
  "name": "Alice Johnson",
  "email": "alice@example.com"
}
```

Приложение использует PostgreSQL как БД. Рассмотрим пошаговый поток данных:



Входящий поток: От запроса к сохранению в БД

1. HTTP запрос → Адаптер (HTTP Handler)

Что происходит: Адаптер получает сырой HTTP запрос и парсит его.

- **Входные данные**:
  - Метод: POST
  - Путь: `/users`
  - Тело: `{"name": "Alice Johnson", "email": "alice@example.com"}`
- **Логирование**:

```
[Адаптер] Получен POST-запрос на /users
[Адаптер] Парсинг JSON: name="Alice Johnson", email="alice@example.com"
```

```
[Адаптер] Вызываю контроллер с name="Alice Johnson", email="alice@example.com"
```

- **Роль в архитектуре**: Внешний слой адаптеров — точка входа, зависящая от HTTP.

2. Адаптер → Контроллер

Что происходит: Контроллер валидирует данные.

- **Входные данные**: `name = "Alice Johnson"`, `email = "alice@example.com"` (валидные)
- **Логирование**:

```
[Контроллер] Получены данные: name="Alice Johnson", email="alice@example.com"
[Контроллер] Валидация пройдена
[Контроллер] Вызываю UseCase для создания пользователя
```

- **Роль в архитектуре**: Часть прикладного слоя — координирует, но без глубокой логики.

3. Контроллер → UseCase

Что происходит: UseCase применяет бизнес-правила и проверяет уникальность.

- **Входные данные**:
  - Проверка: `FindByEmail("alice@example.com")` → `nil` (не существует)
  - Создание: `User{Name: "Alice Johnson", Email: "alice@example.com"}`
- **Логирование**:

```
[UseCase] Запуск бизнес-логики для создания пользователя: name="Alice Johnson"
[UseCase] Проверка существования: вызываю repo.FindByEmail("alice@example.com")
[UseCase] Пользователь не найден - можно создавать
[UseCase] Создаю модель User: {Name: "Alice Johnson", Email: "alice@example.com"}
[UseCase] Вызываю repo.Save(User)
```

- **Роль в архитектуре**: Доменный/прикладной слой — чистая бизнес-логика, независимая от БД.

4. UseCase → Repository Interface

Что происходит: Абстрактный вызов сохранения.

- **Входные данные**: `User{Name: "Alice Johnson", Email: "alice@example.com"}`

- **Логирование**:

```
[Repository-интерфейс] Вызов Save для User: {Name: "Alice Johnson", Email:
"alice@example.com"}
[Repository-интерфейс] Перенаправляю в реализацию (PostgresUserRepository)
```

- **Роль в архитектуре**: Доменный слой — интерфейс для инверсии зависимостей.

5. Repository Interface → Driver (PostgreSQL)

Что происходит: Формирование и выполнение SQL-запроса.

- **Реальные значения**:

```
INSERT INTO users (name, email)
VALUES ('Alice Johnson', 'alice@example.com')
RETURNING id
```

  Результат: `ID=42`
- **Логирование**:

```
[Драйвер] Получен вызов Save: User {Name: "Alice Johnson", Email: "alice@example.com"}
[Драйвер] Формирую SQL: INSERT INTO users (name, email) VALUES ($1, $2) RETURNING id
[Драйвер] Выполняю запрос с параметрами: $1="Alice Johnson", $2="alice@example.com"
[Драйвер] Результат: Новый ID=42, ошибок нет
```

- **Роль в архитектуре**: Инфраструктурный слой — детали реализации.

6. Database → Успешный ответ

Что происходит: Сохранение в PostgreSQL.

- **Реальные значения**: Вставка записи с `id=42`
- **Логирование**:

```
[БД] Выполнена INSERT: Добавлена запись с id=42, name="Alice Johnson",
email="alice@example.com"
```

- **Роль в архитектуре**: Внешняя зависимость в инфраструктуре.

Визуализация полного цикла на диаграмме

□□ Прямые стрелки (сплошные): Входящий поток

Движение данных ОТ клиента К базе данных

- `→` Обычные стрелки показывают передачу запроса и данных вниз по архитектуре
- На каждой стрелке подписано, какие именно данные передаются

□□ Обратные стрелки (пунктирные): Исходящий поток

Движение результата ОТ базы данных К клиенту

- `-.->` Пунктирные стрелки показывают возврат результата вверх по архитектуре
- На каждой обратной стрелке подписано, какой результат возвращается

???????? ???????????? ???????????

1. **Изоляция слоев**: Смена БД не затрагивает бизнес-логику
2. **Тестируемость**: UseCase можно тестировать с mock-репозиториями
3. **Понятность**: Каждый слой имеет четкую ответственность
4. **Гибкость**: Легко добавлять новые адаптеры (GraphQL, gRPC)

Заключение

Этот пример показывает, как Clean Architecture делает поток данных предсказуемым и устойчивым. Каждый слой выполняет свою роль, а зависимости направлены правильно.