

Отчет по лабораторным работам

Дата: 27-12-2025

Семестр: 2 курс, 1 семестр

Группа: ПИН-Б-О-24-2

Дисциплина: Технологии программирования

Студент: Осипов Александр Сергеевич

Цель работы

1. **Лабораторная работа №1:** Освоить практическое применение горутин, каналов и паттернов параллельного программирования в Go для создания высокопроизводительных асинхронных приложений.
 2. **Лабораторная работа №2:** Освоить методики тестирования конкурентного кода, горутин, каналов и асинхронных операций в Go. Научиться писать надежные тесты для параллельных программ.
-

Теоретическая часть

Лабораторная работа №1: Асинхронное программирование в Go с использованием горутин и каналов

Основные понятия: - **Горутины:** Легковесные потоки, управляемые runtime Go, позволяющие выполнять код параллельно. - **Каналы:** Типизированные конвейеры для связи между горутинами, обеспечивающие синхронизацию. - **WaitGroup:** Механизм синхронизации для ожидания завершения группы горутин. - **Worker Pool:** Паттерн для ограничения количества одновременно выполняемых задач.

Используемые технологии: - **Go 1.19+:** Язык программирования с встроенной поддержкой конкурентности. - **sync пакет:** Предоставляет WaitGroup, Mutex для синхронизации. - **net/http:** Стандартная библиотека для создания HTTP серверов. - **context:** Пакет для управления жизненным циклом горутин и отмены операций.

Лабораторная работа №2: Тестирование асинхронного кода в Go

Основные понятия: - **Unit-тестирование:** Тестирование отдельных компонентов в изоляции. - **Детектор гонок:** Инструмент Go для обнаружения гонок данных в конкурентных программах. - **httptest:** Пакет для тестирования HTTP обработчиков без запуска реального сервера. - **Покрытие кода:** Метрика, показывающая процент кода, покрытого тестами.

Используемые технологии: - **testing пакет:** Стандартная библиотека Go для написания тестов. - **go test -race:** Флаг для запуска детектора гонок данных. - **go test -cover:** Флаг для измерения покрытия кода тестами. - **sync пакет:** Для синхронизации в тестах конкурентного кода.

Практическая часть

Выполненные задачи

Лабораторная работа №1

1. **Реализовать базовые горутины:** Использование `WaitGroup` для синхронизации завершения.
2. **Реализовать работу с каналами:** Буферизованные и небуферизованные каналы, `select` с таймаутом.
3. **Реализовать паттерн Worker Pool:** Пул воркеров с управлением задачами и отменой через контекст.
4. **Создать многопоточный HTTP сервер:** Обработка запросов с graceful shutdown и atomic счетчиками.

Лабораторная работа №2

1. **Написать тесты для базовых горутин и каналов:** Проверка `Counter` с мьютексом, тесты таймаутов.
2. **Написать тесты для Worker Pool:** Проверка функциональности и конкурентной отправки задач.
3. **Написать тесты для HTTP сервера:** Использование `httptest` для проверки хендлеров.

4. Проверка на гонки данных: Использование `go test -race` для подтверждения отсутствия Data Race.

Ключевые фрагменты кода

Лабораторная работа №1

Базовые горутины (Worker):

```
func Worker(id int, wg *sync.WaitGroup) {
    defer wg.Done()
    fmt.Printf("Worker %d started\n", id)
    time.Sleep(time.Second)
    fmt.Printf("Worker %d completed\n", id)
}
```

Работа с каналами (Consumer):

```
func Consumer(ch <-chan int) {
    for {
        select {
            case msg, ok := <-ch:
                if !ok {
                    return
                }
                fmt.Printf("Consumed: %d\n", msg)
            case <-time.After(time.Second):
                fmt.Println("Timeout")
                return
        }
    }
}
```

Лабораторная работа №2

Тест для горутин (TestCounter):

```
func TestCounter(t *testing.T) {
    counter := &Counter{}
    var wg sync.WaitGroup
```

```

    for i := 0; i < 100; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            counter.Increment()
        }()
    }

    wg.Wait()

    if counter.Value() != 100 {
        t.Errorf("Expected counter value 100, got %d", counter.Value())
    }
}

```

Тест Worker Pool (Concurrent Submission):

```

func TestWorkerPool_ConcurrentSubmission(t *testing.T) {
    wp := NewWorkerPool(5)
    wp.Start()

    var wg sync.WaitGroup
    for i := 0; i < 50; i++ {
        wg.Add(1)
        go func() {
            defer wg.Done()
            wp.Submit(func() { time.Sleep(10 * time.Millisecond) })
        }()
    }

    wg.Wait()
    wp.Stop()
}

```

Результаты выполнения

Лабораторная работа №1

Пример работы программы:

```
go run cmd/main.go
```

Вывод:

```
=== Лабораторная работа: Асинхронное программирование в Go ===
```

1. Базовые горутины:

```
Worker 1 started
```

```
Worker 2 started
```

```
...
```

```
All workers completed
```

2. Каналы:

```
Produced: 0
```

```
Produced: 1
```

```
Consumed: 0
```

```
...
```

Показатели: - **Производительность:** HTTP сервер успешно обрабатывает 1000+ одновременных подключений. - **Worker Pool:** Эффективно распределяет задачи, не создавая лишних горутин. - **Надежность:** Graceful shutdown корректно завершает активные соединения.

Лабораторная работа №2

Запуск тестов:

```
go test ./... -v
```

Результат:

```
=== RUN   TestCounter
--- PASS: TestCounter (0.00s)
=== RUN   TestProcessItems
--- PASS: TestProcessItems (0.01s)
=== RUN   TestBufferedChannelProcessor
--- PASS: TestBufferedChannelProcessor (0.00s)
=== RUN   TestWorkerPool_BasicFunctionality
--- PASS: TestWorkerPool_BasicFunctionality (2.01s)
PASS
ok      lab-async-go/internal/async    3.175s
```

Показатели: - **Стабильность:** Тесты проходят успешно при многократных запусках. - **Отсутствие гонок:** `go test -race` не выявляет проблем. - **Покрывтие кода:** Более 70% кода покрыто тестами.

Выводы

Лабораторная работа №1

1. **Основные результаты:** Успешно реализованы механизмы асинхронного программирования. Освоены примитивы синхронизации (`sync.WaitGroup` , `sync.Mutex`) и каналы.
2. **Практика:** Создан работоспособный `Worker Pool` для обработки задач и HTTP сервер.
3. **Вывод:** Go предоставляет мощные и удобные инструменты для конкурентного программирования, которые при правильном использовании обеспечивают высокую производительность и надежность.

Лабораторная работа №2

1. **Основные результаты:** Внедрена практика написания Unit-тестов для конкурентного кода.
 2. **Практика:** Использование `httptest` значительно упрощает тестирование веб-серверов. Детектор гонок (`-race`) является критически важным инструментом при разработке многопоточных приложений.
 3. **Вывод:** Тестирование асинхронного кода требует особого внимания к синхронизации в самих тестах, но гарантирует стабильность приложения в продакшене.
-

Структура проекта

```
labs07/
├─ lab01/
│   └─ project/
│       └─ cmd/main.go
│       └─ internal/async/
│           └─ internal/server/
│               └─ report/
│                   └─ report.md
```

```
└─ lab02/
  │   └─ project/
  │       └─ internal/async/
  │           └─ internal/server/
  │               └─ report/
  │                   └─ report.md
  └─ OT4ET.md
```