ПАРАЛЛЕЛЬНОЕ ПРОГРАММИРОВАНИЕ: Channel & WaitGroups (golang)

Выполнили студенты м2-ИФСТ-11: Успанов Н.Н., Жаворонков А.О.

1 Введение

Параллельное программирование является одним из ключевых аспектов современной разработки программного обеспечения, особенно в контексте высокопроизводительных и распределенных систем [2, Concurrency]. Язык программирования Go (Golang) был спроектирован с учетом необходимости простой и эффективной реализации параллельных вычислений [2, Goroutines]. В данной работе рассматриваются два основных механизма управления параллелизмом в Go: каналы (Channels) и группы ожидания (WaitGroups).

2 Теоретические основы параллельного программирования в Go

2.1 Горутины (Goroutines)

Прежде чем углубиться в изучение каналов и групп ожидания, важно понять концепцию горутин в Go. Горутина — это легковесный поток выполнения, управляемый планировщиком Go. В отличие от стандартных системных потоков, горутины имеют очень низкие накладные расходы, что позволяет создавать тысячи горутин в рамках одного приложения [3, раздел 7.5, стр. 218-220].

Горутины запускаются с использованием ключевого слова `go`, за которым следует вызов функции:

```
func main() {
    go func() {
        // Код, который будет выполняться в горутине
```

```
        fmt.Println("Это выполняется в горутине")
    }()

    // Код основной функции продолжает выполняться параллельно
    fmt.Println("Это выполняется в основной функции")
}
```

Важно отметить, что при запуске горутины основная программа не ждёт её завершения. Если основная функция завершит выполнение до горутины, программа завершится, а горутина может не успеть выполниться. Именно здесь на помощь приходят механизмы синхронизации, такие как каналы и группы ожидания [1, Channel Synchronization].

2.2 Каналы (Channels)

Каналы в Go представляют собой типизированные конвейеры, через которые можно отправлять и получать значения. Они являются основным механизмом коммуникации и синхронизации между горутинами [3, раздел 8.4, стр. 267-270]. Каналы позволяют горутинам безопасно обмениваться данными без необходимости явной блокировки.

Создание канала осуществляется с помощью встроенной функции `make`:

```
ch := make(chan int) // Создание канала для передачи целых чисел
```

Существует два основных типа каналов в Go:

2.2.1 Небуферизованные каналы

Небуферизованные каналы не имеют внутренней ёмкости для хранения данных. Это означает, что операция отправки данных в такой канал блокирует отправителя до тех пор, пока другая горутина не будет готова получить это значение. Аналогично, операция получения из канала блокирует получателя, пока отправитель не отправит значение [5, Unbuffered Channels].

```
ch := make(chan int) // Небуферизованный канал

go func() {
    // Отправка значения в канал блокирует горутину
    // до получения этого значения
    ch <- 42
}()

```

```
// Получение значения из канала блокирует главную горутину
// до отправки значения
value := <-ch
fmt.Println(value) // Выведет: 42
```

Такой подход обеспечивает строгую синхронизацию между отправителем и получателем.

2.2.2 Буферизованные каналы

Буферизованные каналы имеют внутреннюю ёмкость для хранения определённого количества значений без необходимости немедленного получения [4, раздел 3.3, стр. 85-87]. Операция отправки в такой канал блокируется только в том случае, если буфер полностью заполнен, а операция получения блокируется, если буфер пуст.

```
ch := make(chan int, 2) // Буферизованный канал с ёмкостью 2

ch <- 1 // Не блокируется
ch <- 2 // Не блокируется
// ch <- 3 // Блокируется, пока из канала не будет прочитано
   значение

value1 := <-ch // Получить первое значение (1)
value2 := <-ch // Получить второе значение (2)
```

Буферизованные каналы позволяют некоторую асинхронность в коммуникации между горутинами.

2.2.3 Направление каналов

Go позволяет определять каналы с ограниченным направлением потока данных [3, раздел 8.4, стр. 270-271]:

```
func send(ch chan<- int) {
    // Канал только для отправки
    ch <- 42
}

func receive(ch <-chan int) {
    // Канал только для получения
    value := <-ch
    fmt.Println(value)
}
```

Такое ограничение позволяет более чётко выразить намерения и предотвратить ошибки, связанные с неправильным использованием каналов.

2.2.4 Закрытие каналов

Каналы в Go можно закрывать, указывая, что больше значений отправлено не будет [1, Channels]:

```
ch := make(chan int)

go func() {
    for i := 0; i < 5; i++ {
        ch <- i
    }
    close(ch) // Закрытие канала
}()

// Перебор значений канала до его закрытия
for value := range ch {
    fmt.Println(value)
}
// После закрытия канала цикл range завершится
```

Закрытие канала особенно полезно при использовании цикла `range` для итерации по значениям канала, так как он позволяет получателю узнать, когда больше нет значений для получения.

2.3 Группы ожидания (WaitGroups)

Группы ожидания представляют собой механизм синхронизации, который позволяет дождаться завершения группы горутин [4, раздел 4, стр. 123-128]. Они предоставляются пакетом `sync` и особенно полезны, когда необходимо точно знать, когда все параллельные задачи завершили свою работу.

Группы ожидания используют счётчик, который увеличивается перед запуском каждой горутины и уменьшается при её завершении. Основная горутина может ждать, пока счётчик не достигнет нуля, что будет означать завершение всех горутин.

```
func main() {
    var wg sync.WaitGroup

    // Увеличиваем счетчик на количество горутин
    wg.Add(2)

    go func() {
        // Уменьшаем счетчик при завершении
        defer wg.Done()
        fmt.Println("Горутина 1 выполняется")
    }()

```

```
13      go func() {
14          defer wg.Done()
15          fmt.Println("Горутина 2 выполняется")
16      }()
17
18      // Ожидаем завершения всех горутин
19      wg.Wait()
20      fmt.Println("Все горутины завершили работу")
21  }
```

Группы ожидания предоставляют три основных метода [3, раздел 9.2, стр. 306-307]:

- `Add(delta int)` — увеличивает счётчик на указанное значение
- `Done()` — уменьшает счётчик на единицу
- `Wait()` — блокирует выполнение до тех пор, пока счётчик не станет равным нулю

2.4 Сравнение Channels и WaitGroups

Каналы и группы ожидания решают разные, но взаимодополняющие задачи [5, Comparing Channels and Wait Groups]:

- **Каналы** предназначены в первую очередь для коммуникации между горутинами, хотя также могут использоваться для синхронизации. Они позволяют передавать данные и сигналы между параллельно выполняющимися частями программы.
- **Группы ожидания** фокусируются исключительно на синхронизации, позволяя основной горутине дождаться завершения всех запущенных горутин. Они не предоставляют механизмов для передачи данных.

Во многих случаях эти механизмы используются совместно для создания эффективных параллельных программ.

3 Практические примеры

Рассмотрим несколько практических примеров, демонстрирующих использование каналов и групп ожидания для решения типичных задач параллельного программирования.

3.1 Проблема гонки данных и её решение с помощью каналов

Одной из распространённых проблем в параллельном программировании является гонка данных (race condition), когда несколько горутин пытаются одновременно читать и изменять одну и ту же переменную [4, раздел 4.1, стр. 130-135].

3.1.1 Пример с гонкой данных

```
func main() {
    counter := 0

    for i := 0; i < 1000; i++ {
        go func() {
            counter++ // Проблема: гонка данных
        }()
    }

    time.Sleep(time.Second)
    fmt.Println("Итоговое значение:", counter)
    // Результат будет непредсказуемым и почти всегда меньше
    1000
}
```

В этом примере несколько горутин пытаются инкрементировать одну и ту же переменную `counter`, что приводит к непредсказуемым результатам из-за гонки данных.

3.1.2 Решение с использованием каналов

```
func main() {
    counter := 0
    ch := make(chan int, 1000)

    for i := 0; i < 1000; i++ {
        go func() {
            ch <- 1 // Отправляем сигнал увеличения
        }()
    }

    // Дожидаемся всех горутин и увеличиваем счетчик
    for i := 0; i < 1000; i++ {
        <-ch // Получаем сигнал
        counter++
    }
```

```
16
17      fmt.Println("Итоговое значение:", counter)
18      // Результат всегда будет 1000
19  }
```

В данном решении мы используем канал для сигнализации о необходимости увеличения счётчика [3, раздел 9.1, стр. 303-304]. Основная горутина последовательно обрабатывает все сигналы и инкрементирует счетчик без гонки данных.

3.2 Проблема ожидания завершения и её решение с использованием WaitGroups

Другой распространённой проблемой является необходимость дождаться завершения всех параллельных задач перед продолжением выполнения программы.

3.2.1 Пример с использованием ненадежного ожидания

```
1  func main() {
2      results := make([]int, 0)
3
4      for i := 0; i < 5; i++ {
5          go func(id int) {
6              // Имитация длительной работы
7              time.Sleep(time.Duration(rand.Intn(100)) * time.
       Millisecond)
8              results = append(results, id)
9          }(i)
10     }
11
12     // Ненадежное ожидание
13     time.Sleep(time.Second)
14
15     fmt.Println("Полученные результаты:", results)
16     // Результат непредсказуем, может не все горутины успеют
       выполниться
17 }
```

Использование `time.Sleep` для ожидания завершения горутин — это ненадежное решение, которое может привести к ошибкам, если какие-то горутины выполняются дольше, чем предполагалось.

3.2.2 Решение с использованием WaitGroups

```go
func main() {
    var wg sync.WaitGroup
    var mu sync.Mutex // Мьютекс для безопасного доступа к слайсу
    results := make([]int, 0)

    for i := 0; i < 5; i++ {
        wg.Add(1)
        go func(id int) {
            defer wg.Done()

            // Имитация длительной работы
            time.Sleep(time.Duration(rand.Intn(100)) * time.Millisecond)

            // Безопасное добавление результата
            mu.Lock()
            results = append(results, id)
            mu.Unlock()
        }(i)
    }

    // Надежное ожидание завершения всех горутин
    wg.Wait()

    fmt.Println("Полученные результаты:", results)
    // Все результаты гарантированно будут получены
}
```

В этом примере мы используем `WaitGroup` для надежного ожидания завершения всех горутин и мьютекс для безопасного доступа к разделяемому ресурсу (слайсу результатов) [1, WaitGroups].

3.3 Комбинированное использование Channels и WaitGroups

На практике часто наиболее эффективные решения получаются при комбинировании каналов и групп ожидания [5, Choosing the Right Mechanism].

```go
func main() {
    var wg sync.WaitGroup
    jobs := make(chan int, 100)
    results := make(chan int, 100)

    // Запускаем воркеров
    for w := 1; w <= 3; w++ {
        wg.Add(1)
        go worker(w, jobs, results, &wg)
    }

```

```
12      // Отправляем задачи
13      for j := 1; j <= 10; j++ {
14          jobs <- j
15      }
16      close(jobs) // Больше заданий не будет
17
18      // Запускаем горутину для закрытия канала результатов
19      // после завершения всех воркеров
20      go func() {
21          wg.Wait()
22          close(results)
23      }()
24
25      // Собираем результаты
26      for r := range results {
27          fmt.Println("Результат:", r)
28      }
29  }
30
31  func worker(id int, jobs <-chan int, results chan<- int, wg *
        sync.WaitGroup) {
32      defer wg.Done()
33      for j := range jobs {
34          fmt.Printf("Воркер %d начал обработку задания %d\n",
        id, j)
35          time.Sleep(time.Second) // Имитация работы
36          fmt.Printf("Воркер %d завершил обработку задания %d\n
        ", id, j)
37          results <- j * 2 // Отправляем результат
38      }
39  }
```

В этом примере мы используем [4, раздел 4.2, стр. 140-145]:

- Каналы `jobs` и `results` для передачи заданий и результатов соответственно
- Группу ожидания `wg` для отслеживания завершения всех воркеров
- Закрытие канала `jobs` для сигнализации об отсутствии новых заданий
- Дополнительную горутину, которая ждёт завершения всех воркеров и затем закрывает канал результатов

Такой подход обеспечивает эффективную параллельную обработку заданий с надежным сбором результатов.

3.4 Базовый пример использования WaitGroups

Рассмотрим простой пример использования WaitGroups для синхронизации горутин [3, раздел 9.2, стр. 305-306]:

```
func main() {
    var wg sync.WaitGroup

    // Увеличиваем счетчик на 2
    wg.Add(2)

    go func() {
        // Уменьшаем счетчик при завершении
        defer wg.Done()
        fmt.Println("Горутина 1")
    }()

    go func() {
        defer wg.Done()
        fmt.Println("Горутина 2")
    }()

    // Ожидаем завершения всех горутин
    wg.Wait()
    fmt.Println("Все горутины завершены")
}
```

В этом примере мы запускаем две горутины и ожидаем их завершения с помощью WaitGroup. Каждая горутина выводит сообщение и уменьшает счетчик WaitGroup. Главная горутина блокируется на вызове `wg.Wait()` до тех пор, пока обе горутины не завершатся.

4 Заключение

Каналы (Channels) и группы ожидания (WaitGroups) представляют собой мощные инструменты для параллельного программирования в Go, которые позволяют разработчикам создавать эффективные и надежные многопоточные приложения [2, Concurrency].

Каналы предоставляют элегантный способ коммуникации и синхронизации между горутинами, обеспечивая безопасную передачу данных и сигналов. Группы ожидания, в свою очередь, обеспечивают простой и надежный механизм для ожидания завершения группы горутин [3, раздел 9.1, стр. 303-304].

Комбинированное использование этих инструментов открывает широкие возможности для эффективного параллельного программирова-

ния в Go, позволяя создавать масштабируемые приложения с высоким уровнем конкурентности.

При написании параллельных программ на Go важно понимать основные принципы работы этих механизмов и выбирать подходящие инструменты для конкретных задач, учитывая их особенности и ограничения [4, раздел 4, стр. 123-165].

Список литературы

[1] Mark McGranaghan, Eli Bendersky. *Go by Example: Channels and WaitGroups*. [Электронный ресурс] // Go by Example. - Режим доступа: https://gobyexample.com

[2] The Go Authors. *Concurrency Patterns in Go*. [Электронный ресурс] // The Go Programming Language. - Режим доступа: https://golang.org/doc/effective_go.html

[3] Alan A. A. Donovan, Brian W. Kernighan. *The Go Programming Language*. Addison-Wesley Professional, 2015, раздел 9.1, стр. 303-304.

[4] Katherine Cox-Buday. *Concurrency in Go: Tools and Techniques for Developers*. O'Reilly Media, 2017, раздел 4, стр. 123-165.

[5] Emre Ayberk Kaymaz. *Concurrency in Go: Channels and WaitGroups*. [Электронный ресурс] // Medium. - Режим доступа: https://medium.com/goturkiye/concurrency-in-go-channels-and-waitgroups-25dd43064d1