Параллельное программирование на Golang. Threadpool

Выполнили студенты м2-ИФСТ-11: Кулагин И.М. и Орлов В.О.

Введение

Параллельное программирование является крайне важным аспектом современной разработки ПО. Связано это, в первую очередь, с ростом числа вычислительных ядер в современных процессорах, а также с увеличением числа клиентов облачных приложений, для которых актуальны вопросы распараллеливания вычислений. Для реализации таких систем используются различные методики, которые позволяют нивелировать задержки в узких местах таких систем. Одной из таких техник является пул потоков или threadpool. Язык программирования Golang, в свою очередь, за счет своих простых в использовании механизмов организации многозадачного выполнения программы, позволяет сфокусироваться на реализации пула потоков.

Инструменты языка

Для построения сущности пула потоков необходимо ознакомиться с некоторыми концептуальными аспектами Golang, на базе которого и будет строиться реализация threadpool.

Горутины

Концепция горутин – то, что отличает Golang от многих других языков. Планировщик ресурсов, встроенный в рантайм языка, позволяет программисту запускать достаточно большое количество легковесных потоков, называемых горутинами, беря на себя планирование их исполнения. В отличии от стандартных потоков ОС, горутины имеют очень низкие накладные расходы на память и время их переключения. Для запуска используется ключевое слово go (рис. 1). Запустить таким образом можно любую функцию. Стоит помнить, что она не будет запущена мгновенно, как обычный вызов функции, а попадет в очередь планировщика. [3]

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

Рисунок 1 – Запуск анонимной функции как горутины

Каналы

Каналы – еще один достаточно необычный элемент, встроенный в язык. Он обеспечивает потокобезопасный обмен данными между разными запущенными горутинами [2]. В отличии от многих языков, использующих внутри себя стандартные потоки ОС, при использовании которых приходиться использовать общую память для передачи данных между ними, модель памяти в Golang [5] ровно противоположная – следует использовать каналы, а не память, для обмена информацией.

Они могут быть как двусторонними, так и однонаправленными, что особенно пригодиться при реализации пула потоков, поскольку попытка записи в канал только для чтения, как и попытка чтение из канала только для записи вызывает ошибку еще на этапе компиляции, что позволяет избавиться от таких ошибок во время исполнения (рис. 2).

```
func worker(input <-chan int, output chan<- int) {
    for value := range input {
        output <- value
    }
}
```

Рисунок 2 – Использование каналов только для чтения и для записи

Также каналы могут быть как небуферизированными, так и буферизированными. Первые применяются для жесткой синхронизации двух горутин, поскольку не допускают разблокировку писателя во время операции на запись, пока читатель не будет готов к чтению (рис. 3).

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

    go func() {
        ch <- 1 // <-----------------------------------------------|
    }()         //                                                 |
                //                                                 |
    <-ch        // Заблокируется, пока не будет инициирована запись |
}
```

Рисунок 3 – Использование канала без буфера

Буферизированные каналы, как и следует из названия, могут принимать в себя некоторое количество значений до момента первого чтения. Размер такого канала задается при его создании и не может быть изменен в процессе использования. Буферизированные каналы блокируют писателя только в том случае, если они полностью заполнены, а читателя – если они полностью пустые (рис. 4).

```
func main() {
    ch := make(chan int, 2)
    ch <- 1 // Попадет в канал
    ch <- 2 // Попадет в канал
    ch <- 3 // Произойдет deadlock (предыдущие значения поместятся в буфер)

}
```

Рисунок 4 – Использование канала с буфером

Группы ожидания

Группы ожидания (WaitGroup) – один из примитивов синхронизации, доступных в стандартной библиотеке языка. Он представляет из себя атомарный счетчик, с помощью которого можно отслеживать состояние разных горутин и, например, выполнять освобождение ресурсов после полного завершения работы нескольких обработчиков, чтобы избежать ошибок в рантайме, например, паники при записи в закрытый канал и т.д.

Когда внутренний счетчик становиться равным 0, горутина, которая блокировалась вызовом метода Wait(), освобождается и может продолжать работу. Однако стоит помнить, что если к моменту вызова Wait() внутренний счетчик так и останется равным 0, то горутина не заблокируется. Это может происходить, например, при инкременте счетчика внутри новой горутины обработчика, поскольку инициализация и запуск горутины занимают гораздо больше времени, чем выполнения кода от вызова горутины до вызова блокирующего метода Wait() [4]. (рис. 5)

```go
func main() {
    inputData := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    workers := 2
    input := make(chan int)

    wg := &sync.WaitGroup{}

    go func() {
        // Помещаем значения во входной канал
        defer close(input) // Не забываем закрыть по окончанию
        for _, value := range inputData {
            input <- value
        }
    }()
    for range workers {
        // Создаем обработчиков и увеличиваем счетчик
        wg.Add(1)
        go func() {
            defer wg.Done() // Уменьшаем счетчик по окончанию
            for value := range input {
                fmt.Println(value * 2)
            }
        }()
    }

    wg.Wait() // Ожидаем, когда счетчик станет равен 0
}
```

Рисунок 5 – Использование группы ожидания

Теория пула потоков

Пул потоков, призван решать 2 проблемы:

- Неэффективное использование оперативной памяти компьютера, скорость работы которой является узким местом по отношению к скорости процессора.;
- Неэффективное использование ресурсов процессора, когда создается слишком много задач, распределение которых по логическим ядрам требует слишком много процессорного времени на постоянное переключение контекста для обеспечения каждого потока некоторым количеством процессорного времени.

Несмотря на то, что рантайм go позволяет нам создавать огромное количество горутин (G согласно модели планировщика golang), физически одновременно и с максимальной эффективностью смогут исполняться не более чем runtime.GOMAXPROC(), а на практике всегда меньше, поскольку как сам рантайм создает служебные горутины, так и процессы, которые создает и использует рантайм (P согласно модели планировщика golang), сами лежат на процессах операционной системы (M согласно модели планировщика golang), планировщик которой также должен выделять процессорное время на все процессы операционной системы, коих может быть немало (>200). [1]

Несмотря на это, чаще всего узким местом реальных систем становятся именно IO-bound задачи, то есть задачи на взаимодействие с внешними устройствами, например с файловой системой или с сокетами по сети, что позволяет эффективно использовать большее количество горутин, чем runtime.GOMAXPROC().

В случае с большим количеством горутин, которые очень часто создаются и запускаются, например, при чтении из канала и запуске горутины для каждого прочитанного значения, скорость выполнения и потребления памяти могут сильно увеличиться, а эффективность планировщика снизиться. Модель потокобезопасного чтения данных из каналов в golang позволяет нескольким горутинам одновременно "висеть" на канале и безопасно читать данные из него с гарантией, что каждое значение из канала будет обработано только 1 раз, организуя такое чтение в виде цикла, что позволяет избежать неконтролируемого снижения производительности и увеличения используемого объема памяти, которое будет зависеть от количества полученных задач. Стоит также отметить, что несмотря на то, что операция запуска горутины гораздо легковеснее, чем запуск потока операционной системы, она все равно гораздо более тяжелая, чем операция чтения из канала.

Таким образом, пул потоков наиболее эффективно покажет себя при количестве обработчиков, близкому к числу логических ядер процессора, либо же к значению runtime.GOMAXPROC(), в случае если это число намеренно было ограничено.

Разбор примеров

Неэффективное использование ресурсов

Проблема: создание горутины для каждого входного значения (рис. 6).

Проявляется в значительном падении производительности, уступая даже однопоточной реализации, а также в неконтролируемом росте числа горутин, что само собой усугубляет проблему.

Происходит из за роста накладных расходов на переключение контекстов между большим числом горутин.

Решается созданием фиксированного числа горутин, ожидающих значения на входном канале и записывающие результат в выходной канал (рис. 7).

```go
func main() {
    inputData := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

    output := make(chan int)

    wg := &sync.WaitGroup{}

    for _, value := range inputData {
        wg.Add(1)
        go func() {
            defer wg.Done()
            output <- value * 2
        }()
    }
    go func() {
        wg.Wait()
        close(output)
    }()
}
```

Рисунок 6 – Проблемная реализация без использования воркеров

```go
func main() {
    inputData := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

    input := make(chan int)
    output := make(chan int)
    wg := &sync.WaitGroup{}
    workersCount := 3

    go func() {
        defer close(input)
        for _, value := range inputData {
            input <- value
        }
    }()

    for range workersCount {
        wg.Add(1)
        go func() {
            defer wg.Done()
            for value := range input {
                output <- value * 2
            }
        }()
    }
    go func() {
        wg.Wait()
        close(output)
    }()

    for value := range output {
        fmt.Println(value)
    }
}
```

Рисунок 7 – Реализация с использованием воркеров

Трудности при необходимости отмены обработки

Проблема: сложность отмены обработки значений, возможность утечки ресурсов, необходимость в поддержке graceful shutdown, необходимость в удобной интеграции кода в другие приложения.

Проявляется в фантомных критических ошибках во время исполнения, потенциальных утечках данных, необходимости написании дополнительного кода для использования в других системах или модулях.

Происходит из-за попыток записи в закрытые каналы, незакрытии ресурсов, интерфейсов, не соответствующим общепризнанным стандартам.

Решается использованием структуры, реализующей интерфейс context.Context, что позволяет выполнять сложные сценарии, например, с отменой по таймауту, без потерь данных и без утечек ресурсов (рис. 8).

```go
func main() {
    inputData := []int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

    input := make(chan int)
    output := make(chan int)
    wg := &sync.WaitGroup{}
    workersCount := 3

    ctx, cancel := context.WithCancel(context.Background())

    go func() {
        defer cancel()
        for _, value := range inputData {
            input <- value
        }
    }()

    for range workersCount {
        wg.Add(1)
        go func() {
            defer wg.Done()
            for {
                select {
                case <-ctx.Done():
                    return
                case value, ok := <-input:
                    if !ok {
                        return
                    }
                    output <- value * 2
                }
            }
        }()
    }
    go func() {
        wg.Wait()
        close(output)
    }()

    for value := range output {
        fmt.Println(value)
    }
}
```

Рисунок 8 – Реализация с использованием воркеров

Заключение

Пул потоков, построенный на удобных и эффективных механизмах: горутинах, каналах, групп ожидания, встроенных в язык Golang, представляет собой универсальный инструмент для максимально эффективной потоковой обработки данных с использованием всех возможностей современных многопроцессорных систем.

Список литературы

1. Shiina S. et al. Lightweight preemptive user-level threads //Proceedings of the 26th ACM SIGPLAN symposium on principles and practice of parallel programming. – 2021. – С. 374-388.
2. Najafizadeh A. Jump over Golang channels.
3. Shmelov O., Gubareva O. USAGE AND IMPLEMENTATION OF PARALLELISM IN GO //Collection of scientific papers «SCIENTIA». – 2023. – №. July 14, 2023; Coventry, UK. – С. 94-97.
4. Gao C., Lv H., Tan Y. Multithreading Technology Based on Golang Implementation //2023 3rd Asia-Pacific Conference on Communications Technology and Computer Science (ACCTCS). – IEEE, 2023. – С. 603-608.
5. The Go Memory Model // Golang URL: https://go.dev/ref/mem (дата обращения: 24.05.2025).