Безопасная Работа с Каналами в Go

апреля 2, 2020 · 4 минуты · German Gorelkin

Продолжаем серию статей о проблемах многопоточности, параллелизме, concurrency и других интересных штуках.

1. Race condition и Data Race
2. Deadlocks, Livelocks и Starvation
3. Примитивы синхронизации в Go
4. **Безопасная работа с каналами в Go**
5. Goroutine Leaks

При работе с параллельным кодом существует несколько различных вариантов

безопасной работы:

- Synchronization primitives for sharing memory
- Immutable data
- Synchronization via communicating
- Data protected by confinement

Synchronization primitives for sharing memory

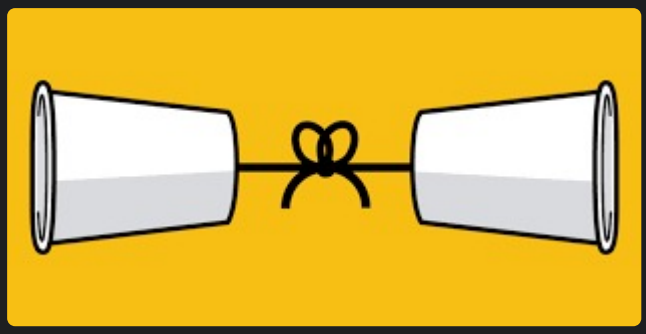
Примитивы синхронизации в Go

Immutable data

В некотором смысле неизменяемые данные идеальны, потому что они неявно безопасны. Каждый параллельный процесс может работать с одними и теми же данными, но не может их изменять.

Если процесс хочет изменить данные, он должен создать новую копию данных с желаемыми изменениями. Это обеспечивает не только более низкую когнитивную нагрузку на разработчика, но также может привести к более быстрым программам, если это приводит к меньшим критическим разделам (или полностью исключает их).

Synchronization via communicating



> **Каналы** можно использовать для синхронизации доступа к памяти, но лучше всего они подходят для обмена информацией между горутинами.

Работу с каналами условно можно разделить на две группы:

Чтение из канала

1. Чтение из nil канала блокирует навсегда
2. Чтение из пустого канала блокирует до записи в этот канал или до его закрытия
3. Чтение из закрытого канала вернет zero value типа канала и false вторым значением
4. Чтение из однонаправленного канала (<-chan) приведет к ошибке компиляции

Чтение из канала достаточно безопасная операция. Оно не может привести к panic из-за действий в других горутинах.

Operation Channel state	Channel state	Result
Read	nil	Block
	Open and Not Empty	Value
	Open and Empty	Block
	Closed	<default value> - false
	Write Only	Compilation Error
Write	nil	Block
	Open and Full	Block
	Open and Not Full	Write Value
	Closed	panic
	Receive Only	Compilation Error
close	**nil**	panic
	Open and Not Empty	Closes Channel;reads succeed until channel is drained. then reads produce default value
	Open and Empty	Closes Channel; reads produces default value
	Closed	panic
	Receive Only	Compilation Error

Запись и закрытие канала

1. Запись в заполненный или nil канал приведет к блокировки
2. **Запись в закрытый канал вызывает panic**
3. **Закрытие nil канала вызывает panic**
4. **Закрытие закрытого канала вызывает panic**

Целых три разных случая которые приведут к panic в программе.

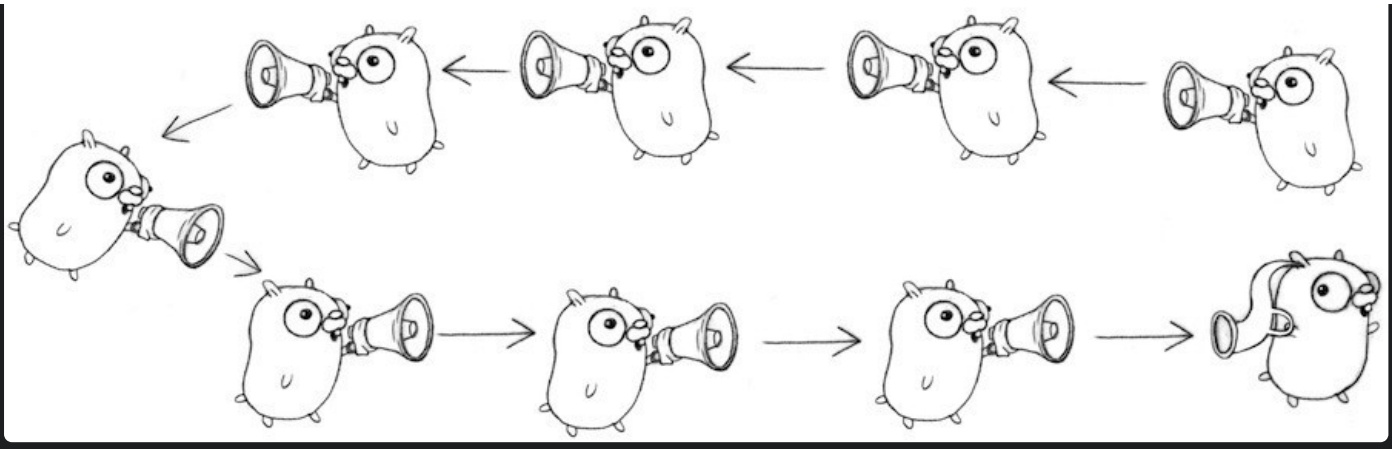
Data protected by confinement

Подход, который может сделать работу с каналами безопасней.

Это простая, но мощная идея обеспечения того, чтобы информация всегда была доступна только в одном параллельном процессе. Когда это достигается, параллельная программа неявно безопасна и не требует синхронизации.

Можно использовать так называемые **ad hoc confinement**. Это когда вы достигаете ограничения посредством соглашений. На уровне проекта, команды, сообщества. Это могут быть style guide или статические анализаторы. Придерживаться таким соглашением, как правило, сложно.

Lexical confinement предполагает использование лексической области действия для предоставления только правильных данных и примитивов синхронизации для использования несколькими параллельными процессами.



Первое, что мы должны сделать — это назначить *владельца канала*.

> Владелец канала — это горутина которая **создает, пишет и закрывает канал.**

Однонаправленные каналы — это инструмент, который позволит нам различать подпрограммы, которые владеют каналами, и те, которые только используют их.

Владельцы каналов имеют доступ для записи в канал (`chan` или `chan<-`), а утилиты канала имеют доступ только для чтения (`<-chan`).

Процедура, которой принадлежит канал, должна:

1. Создать канал
2. Передать канал потребителям
3. Выполнить запись
4. Закрыть канал

```
chanOwner := func() <-chan int {
   out := make(chan int, 5) // 1. create

   go func() {
      defer close(out) // 4. close
      for i := 0; i <= 5; i++ {
         out <- i // 3. write
      }
   }()

   return out // 2. return
}
```

Разрешая создавать, записывать и закрывать канал только владельцу мы

получаем следующие преимущества

- Поскольку мы инициализируем канал, мы исключаем риск deadlock путем записи в нулевой канал
- Поскольку мы инициализируем канал, мы исключаем риск вызвать panic, закрывая нулевой канал
- Поскольку мы сами решаем, когда канал закроется, мы исключаем риск вызвать panic, записывая в закрытый канал
- Поскольку мы сами решаем, когда канал закроется, мы исключаем риск вызвать panic, закрывая канал более одного раза
- Мы используем средство проверки типов во время компиляции, чтобы предотвратить неправильную запись в наш канал

Потребитель канала должен беспокоиться о двух вещах

- Знание, когда канал закрыт
- Обработка блокировок по любой причине

Информацию о закрытии канала можно получить из второго значения при чтении из канала. Или воспользоваться конструкцией `for range`.

Второй пункт определить гораздо сложнее, поскольку он зависит от нашей стратегии. Можно установить тайм-аут, можно прекратить чтение, когда кто-то скажет нам, или мы просто можем блокировать информацию на протяжении всего жизненного цикла процесса. Важно то, что как потребитель мы должны учитывать тот факт, что чтение может и будет блокироваться.

```
chanOwner := func() <-chan int {
   out := make(chan int, 5) // 1. create

   go func() {
      defer close(out) // 4. close
      for i := 0; i <= 5; i++ {
         out <- i // 3. write
      }
   }()

   return out // 2. return
}
```

```
consumer := func(in <-chan int) {
    for result := range in {
        fmt.Printf("Received: %d\n", result)
    }
    fmt.Println("Done receiving!")
}

results := chanOwner()
consumer(results)
```

Потребитель имеет доступ только к каналу чтения, и поэтому ему нужно знать только, как он должен обрабатывать блокировку чтения и закрытие канала.

В контексте лексической области можно писать синхронный код. Избегая большого количества проблем синхронизаций. Таким образом получается более понятный и безопасный код.

go concurrency

« ПРЕДЫДУЩАЯ
Goroutine Leaks

СЛЕДУЮЩАЯ »
Примитивы Синхронизации в Go

© 2025 German Gorelkin Blog Powered by Hugo & PaperMod