Go 并发元件

黎相敏

上海观源信息科技有限公司
上海市闵行区紫竹科技园 4 号楼 303B

12/01/2019

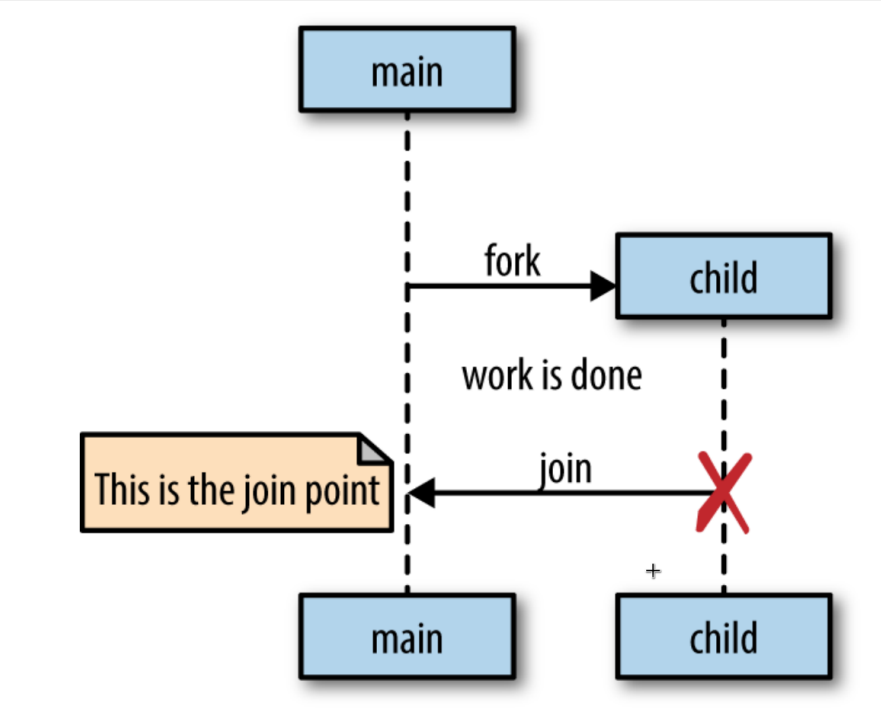
Focus

大纲

并发模型

分叉–汇合（Fork-Join）模型

- 程序执行过程中，父线程可以分叉出与其并发执行的子线程
- 汇聚点：子线程从独立执行到汇合回父线程的时间点



图：分叉–汇合模型

错误设置的汇聚点

代码片段 1: 不正确的汇聚点引发竞态

```
1 sayHello := func() {
2   fmt.Println("hello")
3 }
4 go sayHello()
5 // continue doing other things
```

以上程序的输出结果是不确定的。因为协程由 Go 的运行时创建和调度，sayHello 协程没能通过合适的汇聚点和主协程进行汇合，因此，如果被调度之前主协程已退出，sayHello 就无法获得执行的机会

温馨提示

正确设置的汇聚点能确保程序正确性并消除潜在的竞态

纠正的汇聚点

为上述程序设置正确的汇合点，sayHello 必须和主协程同步，使自己能在主协程退出之前与其汇合，解决方案之一如代码片段 2

代码片段 2: 利用同步确保 sayHello 在主协程退出之前与其汇合

```
1  var wg sync. WaitGroup
2  sayHello := func() {
3    defer wg.Done()
4    fmt.Println("hello")
5  }
6  wg.Add(1)
7  go sayHello()
8  wg.Wait()  // 汇聚点
9
10 // 输出:
11 // hello
```

可见，分叉–汇合模型下并发编程的正确性依赖于数据同步

数据同步方式

go 语言实现数据同步操作提供了两种方式

- 传统地，共享内存型同步模式，常用元件分布在 `sync` 包
- 基于顺序进程通信 (CSP)的消息传递实现数据同步，主要元件为通道 `channel` 及 `select` 语句

sync 包

基本元件

sync 包维护着用于同步底层内存访问的元件，包括

- WaitGroup
- Mutex 和 RWMutex
- Cond
- Once
- Pool

WaitGroup

用途: 等待一批并发操作结束，操作的结果不是关心的重点或者能够通过其他途径收集

```go
var wg sync.WaitGroup

wg.Add(1)
go func() {
  defer wg.Done()
  fmt.Println("Alice sleeping...")
  time.Sleep(1)
}()

wg.Add(1)
go func() {
  defer wg.Done()
  fmt.Println("Bob sleeping...")
  time.Sleep(2)
}()
wg.Wait()
fmt.Println("Both are awaken.")

// 输出:
// Bob sleeping...
// Alice sleeping...
// Both are awaken.
```

WaitGroup

WaitGroup 可看作一个线程安全的计数器

- 通过 Add(x) 加上特定数 x
- 通过 Done() 使其减 1
- Wait() 使其一直阻塞直至计数器置零

温馨提示

Add() 需要在 WaitGroup 协调的协程之外 (一般使主协程) 调用，否则将引入竞态

基本介绍

- `Mutex` 是 “Mutual Exclusion” 的缩写
- 用途: 保护程序的关键区域—需要排他地存取共享资源的程序片段
- `Mutex` 和 `RWMutex` 要求开发者必须以特定的方式访问内存以保证数据的同步

示例

代码片段 3: Mutex 使用示例

```
1  var count int
2  var lock sync.Mutex
3
4  increment := func() {
5    lock.Lock()
6    defer lock.Unlock()
7    count++
8    fmt.Printf("Incrementing: %d\n", count)
9  }
10 decrement := func() {
11   lock.Lock() // 请求对关键区域 --count 变量的存取
12   defer lock.Unlock()  // 放弃对关键区域的排他存取权利
13   count--
14   fmt.Printf("Decrementing: %d\n", count)
15 }
```

示例 (续)

代码片段 4: Mutex 使用示例 (续)

```
16 // Increment
17 var arithmetic sync.WaitGroup
18 for i := 0; i <= 2; i++ {
19   arithmetic.Add(1)
20   go func() {
21     defer arithmetic.Done()
22     increment()
23   }()
24 }
25 // Decrement
26 for i := 0; i <= 2; i++ {
27   arithmetic.Add(1)
28   go func() {
29     defer arithmetic.Done()
```

代码片段 5: Mutex 使用示例 (续)

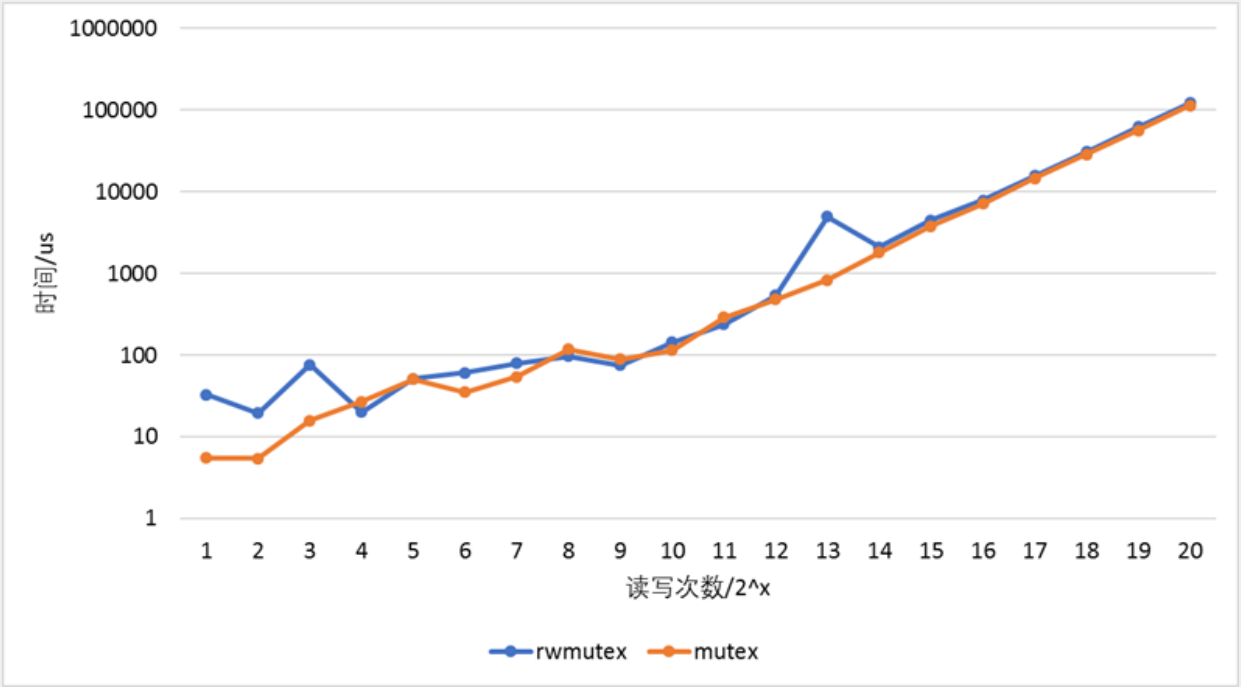
```
30     decrement()
31   }()
32 }
33 arithmetic.Wait()
34 fmt.Println("Arithmetic complet
35
36 // 输出:
37 // Decrementing: -1
38 // Incrementing: 0
39 // Incrementing: 1
40 // Decrementing: 0
41 // Decrementing: -1
42 // Incrementing: 0
43 // Arithmetic complete.
```

温馨提示

- 锁 `m` 的 `Lock()` 调用之后应有配对的 `defer m.UnLock()` 语句
- 进出关键区域的代价是昂贵的

读写锁 RWMutex

在写锁未被锁定之前，读写锁能够满足任意共存的读锁请求
示例代码参见Mutex 和 RWMutex 性能对比，具体结果如下图所示，
可见，性能替身没有想象中的那么明显



图：Mutex 和 RWMutex 性能对比结果

基本介绍

- 用途: 协程用于等待或通知特定事件发生的中心
- 这里的事件只是在两个或更多协程之间传递事件发生的事实本身，无法承载其他信息

使用场景

没有借助条件信号量实现事件通知的两种方式

1. for 循环等到为止: 无限占用 CPU 内核，无法被调度

```
1 for conditionTrue() == false { }
```

2. 利用主动睡眠提供可被抢占机会: 效率不高，而且难以准确设置睡眠的时长：太长降低性能，太短消耗太多 CPU 时间

```
1 for conditionTrue() == false {
2   time. Sleep(1*time.Millisecond)
3 }
```

使用场景

借助 Cond 实现更优的事件通知

```
1 c := sync.NewCond(&sync.Mutex{})
2
3 c.L.Lock()
4 for conditionTrue() == false {
5   c.Wait()
6 }
7 c.L.Unlock()
```

- Wait() 促使协程进入阻塞状态，挂起当前协程，为其他协程腾出在系统级线程执行的机会
- Wait() 内部执行过程会先调用所绑定锁的 Unlock()，而在收到需要的信号后退出执行前再次执行 Lock() 重新获得对锁的控制权

另一个例子是玩具级的限定容量队列

两种通知方式：Signal 和 Broadcast

- Go 运行时维护着等待信号的协程队列 (先进先出)
- `Signal()` 调用后，Go 运行时只会通知等待最久的那个协程并将其出列
- `Broadcast()` 的调用则会 (逐一? 并发?) 通知所有等待的协程
- `Cond` 的性能要比 `channel` 的高

某些需求场景下，基于 `Cond` 的实现会比基于通道 `channel` 简洁很多。例如，按钮点击事件通知

Once

用途: 确保某个操作最多只被执行一次

代码片段 6: Once 使用样例

```
var count int
increment := func() {
  count++
}

var once sync.Once

var increments sync.WaitGroup
increments.Add(100)
for i := 0; i < 100; i++ {
  go func() {
    defer increments.Done()
    once.Do(increment)
  }()
```

代码片段 7: Once 使用样例

```
}

increments.Wait()
fmt.Printf("Count is %d\n", coun

// 输出:
// Count is 1
```

注意事项

- Once 确保的目标操作是通过自己的 Do() 函数注册的第一个回调函数
- Once 内部使用锁，在回调函数调用前请求锁，而在调用返回前释放锁，需要留意不同回调函数之间的依赖性，避免死锁

代码片段 8: Once 的使用不当导致死锁

```
1 var onceA, onceB sync. Once
2 var initB func()
3 initA := func() { onceB. Do(initB) }
4 initB = func() { onceA. Do(initA) }
5
6 onceA. Do(initA)
7
8 // 输出:
9 // fatal error: all goroutines are asleep - deadlock!
```

池 Pool

- 用途: 池模式创建并导出特定数目或一批资源以被第三方使用
- 3 个基本方法

New 创建资源实例

Get 调用时，如果池有可用的实例，则将其直接返回给请求方，否则调用 New 方法实时创建一个然后将其返回给请求方

Put 将服务完毕的实例放回池缓存以被其他进程使用

实例

利用池创建对象，使用完毕后重新放回池中，从而节省对象的创建

```
myPool := &sync.Pool{
  New: func() interface{} {
    fmt.Println("Creating new instance.")
    return struct{}{}
  },
}
myPool.Get()
instance := myPool.Get()
myPool.Put(instance)
myPool.Get()

// 输出:
// Creating new instance.
// Creating new instance.
```

另一个例子是使用池缓存资源以节省内存空间

更多使用场景

预存一批能够快速响应 (初始化比较漫长) 的服务提供方，相关代码参见

- 没有使用池预存对象的方式
- 使用池预存对象的方式

对比结果是：使用缓存池的方式快了近 1000 倍

注意事项

适用条件

- 为并发的进程提供服务对象，这些对象需要满足
  - 在初始化后很快就被抛弃
  - 或创建会对内存影响不小

不适用条件

所需服务对象不是同态的—部分时间会被浪费在对象的类型转换上

Pool 小结

- 初始化 Pool 时，创建一个 New 成员
- 不要对 Get() 返回的对象状态做任何前提假设
- 不要忘记利用 Put() 将服务对象再次缓存以重用
- 池中对象应基本同态，即类型相同

通道 channel

用途和种类

- 用途: 以线程安全的方式实现不同协程之间的通信
- 通道在声明时指定其数据流向
  - 读写　默认，例如，`xStream := make(chan int)`
  - 只读　将 <-箭头放在chan关键字的左边，例如，
    `yStream := make(<-chan int)`
  - 只写　将 <-箭头放在chan关键字的右边，例如，
    `zStream := make(chan<- int)`

温馨提示

go 会在必要时隐式地将单向通道 (只读/只写) 转为双向通道 (读写)

数据类型与传递

- 通道传递的数据类型可以被指定
- 对于一个可读通道`c`，可以利用表达式`c <- x`将值`x`放到通道里
- 对于一个可写通道`c`，可以利用表达式`y <- c`从通道里取出值`y`

代码片段 9: 通道数据传递样例

```
1 stringStream := make(chan string)
2 go func() {
3   stringStream <- "Hello channels!"
4 }()
5 fmt.Println(<-stringStream)
6
7 // 输出:
8 // Hello channels!
```

导致编译错误的操作

- 对只读通道的写操作

```
1 readStream := make(<- chan interface{})
2 readStream <- struct{}{}
3
4 // 输出:
5 // invalid operation: readStream <- struct {} literal
6 // (send // to receive-only type <-chan interface {})
```

- 对只写通道的读操作

```
1 writeStream := make(chan<- interface{})
2 <- writeStream
3
4 // 输出:
5 // invalid operation: <-writeStream (receive from
6 // send-only type chan<- interface {})
```

阻塞性

- 任何尝试向容量已满的通道发送数据的协程会等待直至通道有剩余空间
- 任何尝试向空通道读数据的协程都会等待直至有一个值进入通道

读操作的两种方式

- 只返回一个值: 从通道传输下来的数值
- 返回一个值x和一个布尔值closed
  - x表示从通道传输下来的数据
  - closed=false表示x是由于通道已关闭其容量为空而返回的所声明类型的默认值
  - closed=false则表示x是从别处写入通道的值

代码片段 10: 通道的两种读取操作

```
1  stringStream := make(chan string)
2  go func() {
3    stringStream <- "Hello channels!"
4  }()
5
6  salutation, ok := <-stringStream
7  fmt.Printf("(%v): %v", ok, salutation)
8
9  // 输出:
10 // (true): Hello channels!
```

通道的关闭

- `close(c)`函数调用可关闭通道`c`
- 可读通道能够满足无限的读请求 (即使通道已被关闭)

设计模式 1：FOR...RANGE遍历

range和for关键字组合可以通道作为实参，并在通道关闭时自动跳出循环

```
1  intStream := make(chan int)
2
3  go func() {
4      defer close(intStream)
5      for i := 1; i <= 5; i++ {
6      intStream <- i
7      }
8  }()
9
10 for integer := range intStream {
11     fmt.Printf("%v ", integer)
12 }
13
14 // 输出：
15 // 1 2 3 4 5
```

设计模式 2: 作为取消信号

以通道的关闭为信号通知多个协程

```
1  begin := make(chan interface{})
2  var wg sync.WaitGroup
3  for i := 0; i < 3; i++ {
4      wg.Add(1)
5      go func(i int) {
6          defer wg.Done()
7          <-begin
8          fmt.Printf("%v has begun\n", i)
9      }(i)
10 }
11 fmt.Println("Unblocking goroutines...")
12 close(begin)
13 wg.Wait()
14
15 // Unblocking goroutines...
16 // 1 has begun
17 // 0 has begun
18 // 2 has begun
```

容量

- 通道根据容量可划分为 2 种类型：非缓存型和缓存型
- 非缓存型：声明时不指定容量或指定容量为 0
- 缓存型：通道在初始化时被赋予大于 0 的容量
  - 给定容量为n，对通道c可同时支持n个写操作
  - 可作为一个内存型消息队列实现并发进程间的通信
  - 通道为空且有挂起的等待协程时，新值的传入会绕过通道直接传给等待协程
  - 这类通道容易导致过早优化问题和隐藏死锁问题

通道的满和空都是相对与通道的容量而言的
在预知通道需要支持的并发写操作次数n时，容量大小为n的缓存型通道可以使得写操作尽可能快地执行，例如，缓存型通道作为消息队列

NIL 通道

给定nil通道c

- 对c的读操作会阻塞进程 (尽管并不一定会导致死锁??)

```
1    fatal error: all goroutines are asleep - deadlock!
```

- 对c的写操作也会阻塞进程

```
1 fatal error: all goroutines are asleep - deadlock!
```

- 关闭c会触发panic

```
1    panic: close of nil channel
```

对通道不同操作的后果

Table 3-2. Result of channel operations given a channel's state

Operation	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

图：对通道不同操作的后果

由表可知，

- 阻塞协程的操作有 3 种
- 导致协程恐慌的操作有 3 种

上下文

通道的正确使用需要有合理的上下文环境，而为通道赋予正确的所有权协助创造这种环境

通道的所有权应该属于负责初始化、写入数据和关闭通道的协程

通过这种规则限定

- 所有者持有对通道的读写权限，并负责
  - 初始化通道
  - 执行写操作或将通道所有权转移给其他协程
  - 关闭通道
  - 将以上 3 项封装为一个只读通道暴露给消费者
- 消费者则对通道只有读权限，只需关心
  - 监听通道的关闭
  - 负责处理任何情况下的阻塞

基于通道实现的所有者—消费者样例

```go
chanOwner := func() <-chan int {
    resultStream := make(chan int, 5)
    go func() {
        defer close(resultStream)
        for i := 0; i <= 5; i++ {
        resultStream <- i
        }
    }()
    return resultStream
}

resultStream := chanOwner()
for result := range resultStream {
    fmt.Println("Received:", result)
}
fmt.Println("Done!")
```

```
// 输出:
// Received: 0
// Received: 1
// Received: 2
// Received: 3
// Received: 4
// Received: 5
// Done!
```

select 语句

SELECT 语句

- select 语句负责绑定通道到
  - 局部的类型或单个函数
  - 全局范围内多个系统组件
- 借助 select 语句安全地利用通道实现诸如取消、超时(代码片段 11)、等待和默认值等概念
- 空 select 语句 (select {}) 等价于一个死循环，会永远阻塞

代码片段 11: 基于 select 实现超时退出

```
1 var c <-chan int
2 select {
3 case <-c:
4 case <-time.After(1 * time.Second):
5     fmt.Println("Timed out.")
6 }
7
8 // 输出:
9 // Timed out.
```

CASE 子句的选择公平性

- 不同于switch语句，对于 select 语句
  - case子句不会线性地从上往下执行
  - 在没有任何满足条件的case子块时会一直阻塞
  - Go 运行时会随机均匀地选择非阻塞的case子句执行

case语句指明的所有通道读写操作会被同时检查是否已被满足，即

- 读操作的可读通道非空，或通道已关闭
- 写操作的可写通道非满

CASE 子句的选择公平性证明

```go
c1 := make(chan interface{})
close(c1)
c2 := make(chan interface{})
close(c2)

var c1Count, c2Count int
for i := 1000; i >= 0; i-- {
  select {
  case <-c1:
    c1Count++
  case <-c2:
    c2Count++
  }
}
fmt.Printf("c1Count: %d\nc2Count: %d\n", c1Count, c2Count)

// 输出
// c1Count: 495
// c2Count: 506
```

DEFAULT 默认子句

- 用途: default子句负责执行其他case子句都不满足时的操作

```
start := time.Now()
var c1, c2 <-chan int

select {
case <-c1:
case <-c2:
default:
  fmt.Println("In default after", time.Since(start))
}

// 输出:
// In default after 1.482µs
```

- 通常default子句会与for-select循环结合使用。其中一个用例为：利用一个通道done作为终止循环的信号，而default子句负责执行终止之前的常规任务