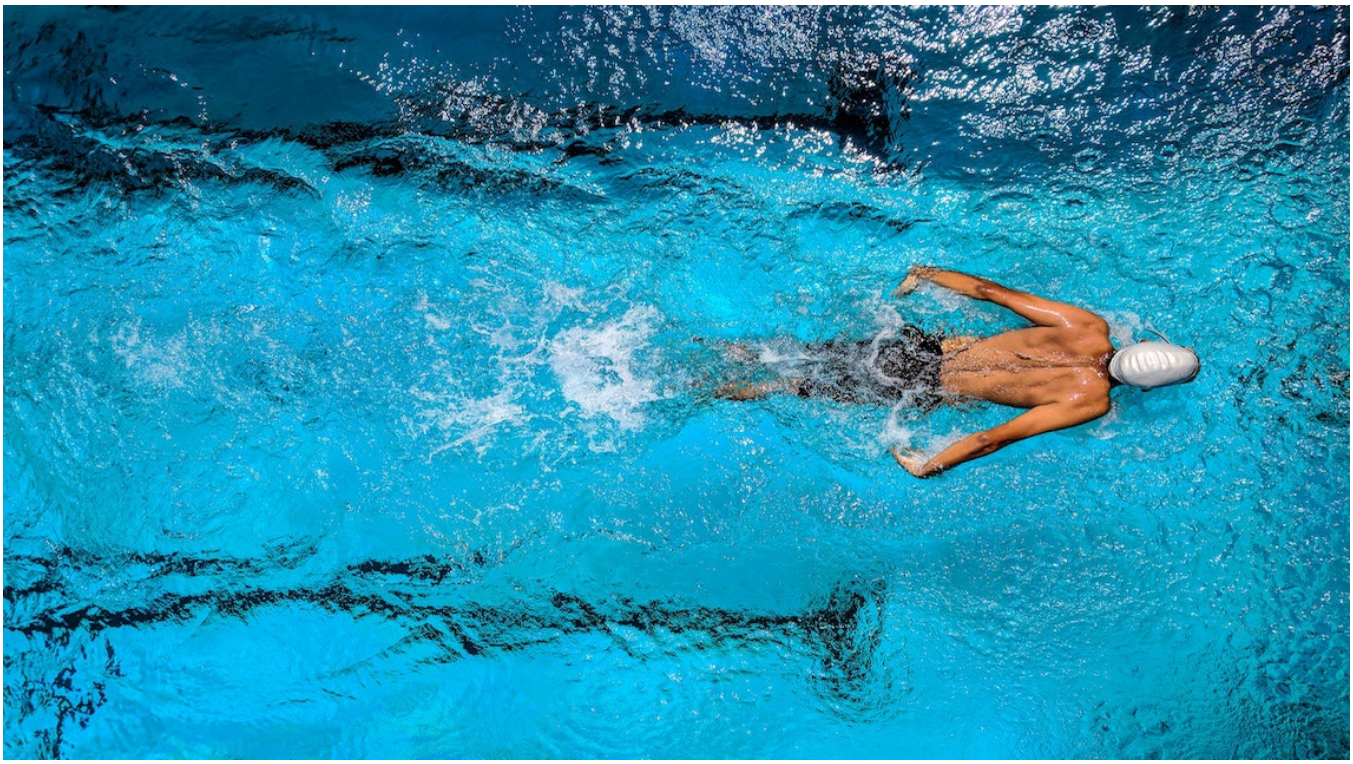
45 | CSP模型：Golang的主力队员

2019-06-11 王宝令



Golang是一门号称从语言层面支持并发的编程语言，支持并发是Golang一个非常重要的特性。在上一篇文章《44 | 协程：更轻量级的线程》中我们介绍过，Golang支持协程，协程可以类比Java中的线程，解决并发问题的难点就在于线程（协程）之间的协作。

那Golang是如何解决协作问题的呢？

总的来说，Golang提供了两种不同的方案：一种方案支持协程之间以共享内存的方式通信，Golang提供了管程和原子类来对协程进行同步控制，这个方案与Java语言类似；另一种方案支持协程之间以消息传递（Message-Passing）的方式通信，本质上是要避免共享，Golang的这个方案是基于**CSP**（Communicating Sequential Processes）模型实现的。Golang比较推荐的方案是后者。

什么是CSP模型

我们在《42 | Actor模型：面向对象原生的并发模型》中介绍了Actor模型，Actor模型中Actor之间就是不能共享内存的，彼此之间通信只能依靠消息传递的方式。Golang实现的CSP模型和Actor模型看上去非常相似，Golang程序员中有句格言：“**不要以共享内存方式通信，要以通信方式共享内存**（Don’t communicate by sharing memory, share memory by communicating）。”虽然Golang中协程之间，也能够以共享内存的方式通信，但是并不推荐；而推荐的以通信的方式共享内存，实际上指的就是协程之间以消息传递方式来通信。

下面我们先结合一个简单的示例，看看Golang中协程之间是如何以消息传递的方式实现通信的。我们示例的目标是打印从1累加到100亿的结果，如果使用单个协程来计算，大概需要4秒多的时间。单个协程，只能用到CPU中的一个核，为了提高计算性能，我们可以用多个协程来并行计算，这样就能发挥多核的优势了。

在下面的示例代码中，我们用了4个子协程来并行执行，这4个子协程分别计算[1, 25亿]、(25亿, 50亿]、(50亿, 75亿]、(75亿, 100亿]，最后再在主协程中汇总4个子协程的计算结果。主协程要汇总4个子协程的计算结果，势必要和4个子协程之间通信，**Golang中协程之间通信推荐的是使用channel**，channel你可以形象地理解为现实世界里的管道。另外，calc()方法的返回值是一个只能接收数据的channel ch，它创建的子协程会把计算结果发送到这个ch中，而主协程也会将这个计算结果通过ch读取出来。

```
import (
 "fmt"
 "time"
)

func main() {
   // 变量声明
 var result, i uint64
   // 单个协程执行累加操作
 start := time.Now()
 for i = 1; i <= 10000000000; i++ {
  result += i
 }
 // 统计计算耗时
 elapsed := time.Since(start)
 fmt.Printf("执行消耗的时间为:", elapsed)
 fmt.Println(", result:", result)

   // 4个协程共同执行累加操作
 start = time.Now()
 ch1 := calc(1, 2500000000)
 ch2 := calc(2500000001, 5000000000)
 ch3 := calc(5000000001, 7500000000)
 ch4 := calc(7500000001, 10000000000)
   // 汇总4个协程的累加结果
 result = <-ch1 + <-ch2 + <-ch3 + <-ch4
```

```
 // 统计计算耗时
 elapsed = time.Since(start)
 fmt.Printf("执行消耗的时间为:", elapsed)
 fmt.Println(", result:", result)
}
// 在协程中异步执行累加操作，累加结果通过channel传递
func calc(from uint64, to uint64) <-chan uint64 {
    // channel用于协程间的通信
 ch := make(chan uint64)
    // 在协程中执行累加操作
 go func() {
  result := from
  for i := from + 1; i <= to; i++ {
   result += i
  }
        // 将结果写入channel
  ch <- result
 }()
    // 返回结果是用于通信的channel
 return ch
}
```

CSP模型与生产者-消费者模式

你可以简单地把Golang实现的CSP模型类比为生产者-消费者模式，而channel可以类比为生产者-消费者模式中的阻塞队列。不过，需要注意的是Golang中channel的容量可以是0，容量为0的channel在Golang中被称为**无缓冲的channel**，容量大于0的则被称为**有缓冲的channel**。

无缓冲的channel类似于Java中提供的SynchronousQueue，主要用途是在两个协程之间做数据交换。比如上面累加器的示例代码中，calc()方法内部创建的channel就是无缓冲的channel。

而创建一个有缓冲的channel也很简单，在下面的示例代码中，我们创建了一个容量为4的channel，同时创建了4个协程作为生产者、4个协程作为消费者。

```
// 创建一个容量为4的channel
ch := make(chan int, 4)
// 创建4个协程，作为生产者
for i := 0; i < 4; i++ {
 go func() {
  ch <- 7
 }()
}
// 创建4个协程，作为消费者
for i := 0; i < 4; i++ {
    go func() {
     o := <-ch
     fmt.Println("received:", o)
    }()
}
```

Golang中的channel是语言层面支持的，所以可以使用一个左向箭头（<-）来完成向channel发送数据和读取数据的任务，使用上还是比较简单的。Golang中的channel是支持双向传输的，所谓双向传输，指的是一个协程既可以通过它发送数据，也可以通过它接收数据。

不仅如此，Golang中还可以将一个双向的channel变成一个单向的channel，在累加器的例子中，calc()方法中创建了一个双向channel，但是返回的就是一个只能接收数据的单向channel，所以主协程中只能通过它接收数据，而不能通过它发送数据，如果试图通过它发送数据，编译器会提示错误。对比之下，双向变单向的功能，如果以SDK方式实现，还是很困难的。

CSP模型与Actor模型的区别

同样是以消息传递的方式来避免共享，那Golang实现的CSP模型和Actor模型有什么区别呢？

第一个最明显的区别就是：**Actor模型中没有channel**。虽然Actor模型中的 mailbox 和 channel 非常像，看上去都像个FIFO队列，但是区别还是很大的。Actor模型中的mailbox对于程序员来说是“透明”的，mailbox明确归属于一个特定的Actor，是Actor模型中的内部机制；而且Actor之间是可以直接通信的，不需要通信中介。但CSP模型中的 channel 就不一样了，它对于程序员来说是“可见”的，是通信的中介，传递的消息都是直接发送到 channel 中的。

第二个区别是：Actor模型中发送消息是**非阻塞**的，而CSP模型中是**阻塞**的。Golang实现的CSP模型，channel是一个阻塞队列，当阻塞队列已满的时候，向channel中发送数据，会导致发送消息的协程阻塞。

第三个区别则是关于消息送达的。在《42 | Actor模型：面向对象原生的并发模型》这篇文章中，我们介绍过Actor模型理论上不保证消息百分百送达，而在Golang实现的**CSP模型中，是能保证消息百分百送达的**。不过这种百分百送达也是有代价的，那就是有可能会导致**死锁**。

比如，下面这段代码就存在死锁问题，在主协程中，我们创建了一个无缓冲的channel ch，然后从ch中接收数据，此时主协程阻塞，main()方法中的主协程阻塞，整个应用就阻塞了。这就是Golang中最简单的一种死锁。

```
func main() {
    // 创建一个无缓冲的channel
    ch := make(chan int)
    // 主协程会阻塞在此处，发生死锁
    <- ch
}
```

总结

Golang中虽然也支持传统的共享内存的协程间通信方式，但是推荐的还是使用CSP模型，以通信的方式共享内存。

Golang中实现的CSP模型功能上还是很丰富的，例如支持select语句，select语句类似于网络编程里的多路复用函数select()，只要有一个channel能够发送成功或者接收到数据就可以跳出阻塞状态。鉴于篇幅原因，我就点到这里，不详细介绍那么多了。

CSP模型是托尼·霍尔（Tony Hoare）在1978年提出的，不过这个模型这些年一直都在发展，其理论远比Golang的实现复杂得多，如果你感兴趣，可以参考霍尔写的Communicating Sequential Processes这本电子书。另外，霍尔在并发领域还有一项重要成就，那就是提出了霍尔管程模型，这个你应该很熟悉了，Java领域解决并发问题的理论基础就是它。

Java领域可以借助第三方的类库JCSP来支持CSP模型，相比Golang的实现，JCSP更接近理论模型，如果你感兴趣，可以下载学习。不过需要注意的是，JCSP并没有经过广泛的生产环境检验，所以并不建议你在生产环境中使用。

欢迎在留言区与我分享你的想法，也欢迎你在留言区记录你的思考过程。感谢阅读，如果你觉得这篇文章对你有帮助的话，也欢迎把它分享给更多的朋友。

精选留言

walkingonair 3

最后一篇了，结束打卡，内容很丰富，还要复习两遍

2019-06-11

zhangtnty 3

王老师辛苦了

2019-06-11

作者回复

感谢

2019-06-11

Sunqc 2

会反复看这些内容，每次看感觉都不一样，另外老师推荐的 并发编程的艺术，还有图解java多线程模式，结合这些书看加深理解。持续关注并发编程

2019-06-11

作者回复

感谢感谢

2019-06-11

QQ怪 2

更本看不够啊，不想老师停更

2019-06-11

作者回复

赶紧见好就收

2019-06-11

我的腿腿

0

老师辛苦了，我也打卡，像我们底层的程序员只能好好学习技术前行了

2019-06-13

作者回复

劳动不分高低贵贱

2019-06-13

ack

0

今天默默打开才发现已经更完了，谢谢老师，写得真的通俗易懂

2019-06-13

作者回复

多谢夸奖

2019-06-13

JackJin

0

第一个看完的专栏，老师叫的跟细致，以前只会简单的用一下线程，现在稍微会诊断一下多线程引发的问题了，课程虽然讲完了，内容还未消化，会反复阅读。

2019-06-13

作者回复

多谢夸奖

2019-06-13

朱延云

0

谢谢老师，终于补完了，对并发编程有了更多认识。特备棒的课程，辛苦了

2019-06-12

作者回复

也感谢你的支持

2019-06-12

刘晓林

0

全程跟完的，飞机起飞关机前，打卡。感谢老师

2019-06-12

作者回复

感谢支持

2019-06-12

nimil

0

最后一篇了？谢谢老师，收获很多

2019-06-12

作者回复

感谢支持

2019-06-12

张三 0

打卡！这是最后一期了吗？

2019-06-12

作者回复

还有一篇不谈技术的总结

2019-06-12

圆小白 0

老师，幸苦了。

2019-06-12

作者回复

彼此彼此

2019-06-12

空知 0

感谢 老师 ヾ(°°)!

2019-06-11

作者回复

客气了，感谢支持

2019-06-12

刘章周 0

个人感觉把前三个模块重点学完，后面的框架、其他模型有空了解下，不是重点。对于框架部分，有阅读源码能力有必要深入研究。

2019-06-11

onepieceJT2018 0

终于追完老师的专栏 虽然golang的语法看上去有点不适应 但是了解到CSP这种模型的思想还是很有收获的

2019-06-11

西行寺咕哒子 0

全程追更下来， 老师辛苦了

2019-06-11

作者回复

你也辛苦了

2019-06-11

kaixiao7 0

在这个专栏中学到了很多，尤其是并发编程的原理，老师讲的非常棒。

非常感谢老师，老师辛苦了！

2019-06-11

作者回复

感谢信任

2019-06-11

Vincent

0

谢谢

2019-06-11

广训

0

这个专栏是完整读完的，比虚拟机那个容易多了。虚拟机那个专栏，看的非常辛苦才读完，还很多未能理解，往后还要继续重读。读完对并发编程理解又近了一步，非常赞。

2019-06-11

cricket1981

0

"Actor 模型中发送消息是非阻塞的"--->如果actor mailbox满了呢？是否发送者阻塞？另外，如果引入actor persistence是否能保证消息百分百送达？

2019-06-11