目　录

前序

本书转自:https://www.kancloud.cn/mutouzhang/go/596804

Go是门很棒的语言。当它首次被宣告给这个世界时，所包含的特性令人疯狂：语法简洁，编译速度非常快，执行效率高，支持鸭子类型，用它的并发原语处理问题非常直观。在第一次使用go关键字建立goroutine时(这就是本书将要介绍的东西)，我太高兴了，与其他多种语言相比，我从未有过如此爽快的感觉。

多年来，我把Go代码一点一点加入个人项目中，直到有一天，猛然发觉已经积累了上万行之多。随着社区的不断发展，在Go中使用并发的最佳实践话题越来越多。 有些开发者就他们的使用经验进行了讨论。 但在社区中关于如何较好的进行并发操作，并没有一本完善的综合指南。

正是基于这一点，我决定写这本书。以希望大家能够了解并掌握有关Go中并发性的高质量、全面的信息：如何使用它，如何将最佳实践和模式整合到系统中，以及它们如何在所有系统中运行。 我尽力在这些考量之间取得平衡。

衷心希望这本书能给你带来帮助！

谁适合读这本书

本书适用于有Go经验的开发人员; 我不会试图解释语言的基本语法。关于如何在其他语言中呈现并发性的知识是有用的，但并非必需。

在本书中，我们将讨论整个Go并发系统：常见的并发问题，Go并发设计背后的思考，Go并发原语的基本语法，常见的并发模式，以及各种工具，以帮助你应对日常工作中遇到的问题。

书中介绍的主题非常广泛。你可以根据个人需求随时查看章节导读，以帮助浏览本书。

章节导读

在我阅读技术书籍时，通常会跳到那些能够激起我兴趣的部分，或者，如果我试图提高新技术的效率，我会疯狂地剔除与当前工作直接相关的部分 。 无论你的使用初衷是什么，这里是本书的路线图，希望它能帮助你，引导你到需要的地方。

第一章 并发编程介绍

本章介绍了并发的历史，阐述并发是一个重要的概念，并讨论了一些使并发难以正确的基本问题。 同时简要介绍了Go如何缓解这些负担。

如果你有相关的知识，或者只是想了解如何使用Go的并发原语技术，可以跳过本章。

第二章 代码建模：序列化交互处理

本章介绍了Go的设计的一些初衷。 这将有助于为你提供与Go社区中其他人交谈的背景信息，并帮助你了解为什么代码会按照的这样那样的思考方式运行。

第三章 Go的并发构建模块

这一章我们将开始深入研究Go并发原语的语法。 还会介绍标准库中的sync包，它负责处理Go的内存访问同步。 如果你之前没有在Go中使用过并发，但希望立即了解正式内容，那么这里就是开始的地方。

在Go中编写并发代码的所需基础知识很少，在这里主要是将Go的概念与其他语言和并发模型进行比较。 严格地说，没有必要理解这些东西，但这些概念可以帮助你全面了解Go语言的并发特性。

第四章 Go的并发编程范式

在本章中，我们将着眼于如何把Go的并发原语组合在一起来形成有用的模式。 这些模式既可以帮助我们解决问题，也可以避免在使用中常见的错误。

如果你已经使用Go编写了一些并发代码，本章仍然是有用的。

第五章 大规模并发

在本章中，我们将学习过的模式组合成更大的程序，展示服务和分布式系统中常用的经验。

第六章 Goroutines和Go运行时

本章介绍Go运行时如何处理调度goroutines。 这对于那些想了解Go的运行时内部机制的人来说会很有意思。

附录
附录列举了各种工具和命令，可以帮助你更轻松地编写和调试并发程序。

所有示例皆运行于main函数下，译者程序基于go 1.10.1运行于windows7系统下，且全部验证无误

在线资源

Go有一个非常活跃和热情的社区！ 对于那些喜欢Go的人来说，很容易找到友善，乐于帮助的人。 以下是我最喜欢的面向社区阅读资料，去获取帮助和同伴之间的互动吧：

- https://golang.org/
- https://golang.org/play
- https://go.googlesource.com/go
- https://go.googlesource.com/go
- https://github.com/golang/go/wiki

并发编程介绍

并发是一个有趣的词，因为它对编程领域中的不同人员意味着不同的事情。除了”并发”之外，你可能听说过”异步”，”并行”或”线程化”等字眼。有些人认为这些词意思相同，而其他人则非常特别地在每个词之间划定界限。既然我们整本书都会讨论并发，那么首先花一些时间讨论我们说”并发”时指代的含义是非常重要的。

我们将在第2章花费一些时间讨论并发原理，但现在让我们采用一个实际的定义，作为我们理解的基础。

当大多数人使用“并发”这个词时，他们通常指的是与一个或多个进程同时发生的进程。 通常也暗示所有这些进程都在同一时间取得进展。 在这个定义下，一个简单的方法来思考这个问题是——人。 你正在阅读这篇文章，而世界上的其他人则同时过着自己的生活。 它们与你同时存在。

并发是计算机科学中的一个广泛话题，从这个定义中可以得出各种各样的主题：理论，并发建模方法，逻辑正确性，实际问题甚至理论物理。 我们将在本书中涉及一些辅助话题，但我们主要将坚持涉及Go内容中涉及并发的部分，具体为：Go如何选择对并发进行建模，由此产生哪些问题和模型，以及我们如何在这个模型中构造原语来解决这些问题。

在本章中，我们将详细介绍并发在计算机科学中占有重要地位的原因，为什么并发操作很困难并且值得仔细研究，最重要的是，尽管存在这些挑战，Go可以使 通过使用它的并发原语使程序更加清晰和快速。

与大多数理解问题的方式一样，我们将从一些历史开始，解读并发操作是如何成为一个重要话题的。

摩尔定律，可伸缩网络和我们所处的困境

1965年，美国科学家，企业家，英特尔公司创始人之一的戈登·摩尔撰写了一篇长达三页的论文，阐述了电子产品市场对集成电路的整合影响，并预测集成电路中元件数量每年至少增加一倍，而这个过程将持续至少十年。1975年，他修正了这一预测，指出集成电路中的元件数量每两年翻一番。 并持续到2012年左右。

一些公司预见摩尔定律预测的速度会放缓，开始研究增加计算能力的替代方法。 俗话说，必要性是创新之母，所以多核处理器就是这样诞生了。

这看起来像是解决摩尔定律边界问题的一种聪明方式，但是计算机科学家很快就发现自己面临着另一个定律的局限：以IBM360系列机的主要设计者阿姆达尔命名的阿姆达尔定律。

阿姆达尔定律指出，系统中对某一部件采用更快执行方式所能获得的系统性能改进程度，取决于这种执行方式被使用的频率，或所占总执行时间的比例。于是阿姆达尔致力于并行处理系统的研究。

例如，假设你正在编写一个基于GUI的程序：用户将看到一个界面，点击某些按钮，并发生一些事情。 这种类型的程序受到管道中一个非常大的连续部分的限制：人员交互。 无论为此程序提供多少内核，它总是受限于用户与界面进行交互的速度。

现在考虑一个不同的例子，计算pi的数字。多亏了spigot算法，我们可以很容易地将其划分为并行任务。在这种情况下，通过为程序提供更多的内核可以获得显著的收益，并且新问题变成了如何组合和存储结果。

阿姆达尔定律帮助我们理解这两个问题之间的差异，并帮助我们确定并行化是否是解决系统性能问题的正确方法。

对于这种类型的并行问题，建议编写可以水平扩展的应用程序。 这意味着可以使用程序的实例，在更多的CPU或机器上运行它，虽然这会导致系统的运行时间提高。 以这种方式构建程序非常简单，你可以将大块问题发送给应用程序的不同实例。

在21世纪初，当一种新的范式开始出现后，横向扩展变得容易得多：云计算。 尽管有迹象表明这个词早在20世纪70年代就已经被使用过了，但是21世纪初，这个概念真正在时代精神中扎根。 云计算意味着一种新的应用程序部署及扩展的规模和方法。云计算替你配置物理设备，替你安装软件，替你进行维护，这意味着你可以访问庞大的资源池，这些资源池将按需提供到机器中供工作负载使用。 物理设备对使用者变得可有可无，并且配备了特别适合他们将要运行的程序的特性。 通常（但不总是）这些资源池被托管在其他公司拥有的数据中心中。

这种改变引发了一种新的思考。 突然之间，开发人员相对低成本的具备了大规模的计算能力，他们可以用它来解决大型问题。解决方案现在可以轻松跨越许多机器甚至访问到全球。 云计算为以前那些只有技术巨头才有资格解决的问题提供了一套全新的低成本解决方案。
2021-01-02 14:08:53 星期六
但云计算也带来了许多新的挑战。 调配资源，在设备之间进行通信以及汇总和存储结果都成为需要解决的问题。 但其中最困难的是弄清楚如何同时对代码进行建模。 部分解决方案可能在不同的机器上运行，这一情况加剧了建模问题时常见的一些问题。这些问题很快就造就了新的解决方案，可伸缩网络。

网络通常希望能够通过添加更多应用程序实例来处理数十万（或更多）的同步工作负载，这使得滚动升级，弹性水平可伸缩体系结构和地理描述等成为必备属性，与此同时，这种解决方案还在编译和容错两方面引入了新的复杂度。

因此，我们发现，现代开发人员可能有点不知所措。2005年，ISO C++标准委员会主席，C++/CLI首席架构师Herb Sutter为Dobb博士撰写了一篇文章，标题为“免费午餐结束：软件并发的根本转向”。 标题贴切，文章有先见之明。 Sutter最后表示，“我们迫切需要一种更高层次的并发性编程模型，而非当前语言所能提供给我们的。”

如果想明白为什么Sutter有如此的“迫切感”，我们必须研究为什么并发很难得到正确的结果。

为什么并发编程如此困难

编写并发代码的难度有目共睹。通常需要几次迭代才能按预期工作，即使如此，在某些时间发生变化（较重的磁盘利用率，更多用户登录系统等）之前，代码中存在多年的bug也并不罕见，预先发现并发代码错误就像看自己的后脑勺一样困难。 事实上，对于本书来说，我已经在代码上尽可能多地试图缓解这种情况。

幸运的是，在使用并发代码时，每个人都会遇到同样的问题。 正因为如此，计算机科学家已经能够标记常见问题，这使我们能够讨论它们如何产生，为什么产生以及如何解决它们。

所以让我们开始吧。 以下是一些最常见的问题，使得并发编程既令人沮丧又有令人着迷。

数据竞争

当两个或更多的操作必须以正确的顺序执行时，就会出现竞争状态，但如果程序没有写入，无法使操作顺序得到保持。

大多数时候，这出现在所谓的数据竞争中，其中一个并发操作尝试在某些未确定的时间读取变量，而另一个并发操作尝试写入同一个变量。

这里有一个简单的例子：

```
1 var data int
2 go func() { // 1
3     data++
4 }()
5 if data == 0 {
6     fmt.Printf("the value is %v.\n", data)
7 }
```

1. 在Go中，可以使用go关键字同时运行一个函数。 这样做创建了所谓的goroutine。

在第3行和第5行都试图访问名为data的变量，但是并没有施行任何措施保证执行的顺序。运行此代码有三种可能的结果：

- 没有输出。在这种情况下，第3行是在第5行之前执行的。
- 输出 the value is 0。在这种情况下，第5行和第6行在第3行之前执行。
- 输出 the value is 1。在这种情况下，第5行在第3行之前执行，但第3行在第6行之前执行。

正如你所看到的，仅仅几行不确定的代码会在你的程序中引入巨大的变化。

大多数情况下，数据竞争是由于开发人员按顺序思考问题而引入的。 他们认为，上一行代码会先于下一行代码执行。 他们假设在if语句中读取数据变量之前，上面的goroutine将被调度并执行。

在编写并发代码时，你必须仔细地遍历所有可能出现的场景。 除非你正在使用本书稍后部分介绍的一些技巧，否则保证代码将按其在源代码中列出的顺序运行。 我有时会发现在操作之间等待很长一段时间会很有帮助。 想象一下，在调用goroutine的时间和运行的时间之间要经过一个小时。 该程序的其余部分如何运作？ 如果在goroutine成功执行和程序到达if语句之间花了一个小时呢？ 以这种方式思考对我有所帮助，因为对于计算机而言，规模可能不同，但相对时间差异差不多。

事实上一些开发者确实这么干并发现看起来解决了并发上的问题，我们修改上个例子看看：

```
1 var data int
2 go func() { // 1
3     data++
4 }()
5 time.Sleep(1*time.Second) // 这种做法实在太烂了!
6 if data == 0 {
7     fmt.Printf("the value is %v.\n", data)
8 }
```

我们解决了数据竞争问题吗吗？没有。事实上，从这个方案中产生的所有三个结果仍然是可能的。我们在调用我们的goroutine和检查数据值之间的让程序休眠的时间越长，程序越接近实现正确性——但这只是在概率上渐近地接近逻辑正确而已。

除此之外，这样做已经在算法中引入了低效率。 程序现在必须休眠一秒钟才能使我们更有可能看不到的数据竞争。如果我们使用正确的方式来编写代码，我们可能无需等待，或者等待时间可能只有1微秒。

这里要说的是，你应该总是以逻辑的正确性为目标。 在你的代码中引入休眠可以是一种调试并发程序的方便方式，但不是解决方案。

数据竞争的产生条件是最隐秘的并发错误类型之一，因为它们可能在代码投入生产后才会展现出来。 它们通常是由代码执行环境发生变化或前所未有的突发事件引起的。 在这些情况下，代码看起来行为正确，但实际上，这些操作按顺序执行的出现不确定性的几率非常高。

原子性

当某种东西被认为是原子性的或者具有原子性的时候，这意味着在它运行的环境中，它是不可分割的或不可中断的。

那么这到底意味着什么，为什么在使用并发代码时知道这很重要？

第一件非常重要的事情就是了解“上下文(context)”这个词。在某个特定的上下文中，有的操作可能是原子的，有的可能不是。在你的流程环境中，原子状态的操作在操作系统环境中可能不是原子操作； 在操作系统环境中是原子的操作在你的机器环境中可能不是原子的; 并且在机器上下文中是原子的操作在应用程序上下文中可能不是原子的。 换句话说，操作的原子性可以根据当前定义的范围而改变。 清醒的认识到这个事实对你程序的利弊是非常重要的！

在考虑原子性时，经常需要做的第一件事是定义上下文或作用域，这个操作将被视为原子性的。这是思考程序的基础。

> 2006年，游戏公司Blizzard成功起诉了MDY Industries 600万美元源于其开发的名为“滑翔机”的程序，该程序可在没有用户干预的情况下自动操作他们的游戏“魔兽世界”。 这些类型的程序通常被称为“机器人外挂”。
> 当时，魔兽世界有一个反作弊程序叫做“守望者“，它可以在你玩游戏的任何时候运行。除此之外，守望者将扫描主机的内存并运行启发式查找似乎用于作弊的程序。
> 利用原子上下文的概念，滑翔机成功避免了守望者的这种检查。 守望者认为扫描机器上的内存是一种原子操作，但在开始扫描之前，滑翔机利用硬件中断来隐藏自己！守望者守护进程的内存扫描在进程的上下文中是原子的，但不在操作系统的上下文中。

现在让我们看一下术语“不可分割”和“不可中断”。这些术语意味着在你定义的上下文中，原子的东西将在整个过程中发生，而不会同时发生任何事情。 这让人有点糊涂，所以我们来看一个例子：

```
i++
```

这是一个任何人都可以明白的简单代码，但它很容易证明原子性的概念。 它可能看起来很原子，但是一个简单的分析揭示了几种操作：

- 检索i的值。
- 增加i的价值。
- 存储i的值。

尽管这些操作中的每一个都是原子的，但三者的组合可能不是，取决于你的上下文。 这揭示了原子操作的一个有趣特性：将它们结合并不一定会产生更大的原子操作。 创建操作原子取决于你希望它在哪个上下文中处于原子状态。 如果你的上下文是一个没有并发进程的程序，那么这个代码在该上下文中是原子的。 如果你的上下文是一个不会将 i 暴露给其他goroutine的goroutine，那么这个代码是原子的。

为什么我们如此在意原子性？原子性非常重要，因为如果说某些东西是原子的，那么就隐式地意味着在并发环境中是安全的。这使我们能够编写逻辑上正确的程序，并且——我们稍后将看到——甚至可以用作优化并发程序的一种方式。

大多数语句不是原子的，更不用说函数，方法和程序了。如果原子性是组成逻辑上正确的程序关键，并且大多数语句不是原子的，那么我们如何调和这两个问题？稍后我们会深入探讨，但总之，我们可以通过采用各种技术来强制原子性。至于如何决定你的代码的哪些部分需要是原子的，以及需要划分到什么样的粒度，我们在下一节继续讨论。

内存访问同步

假设我们有一个数据竞争：两个并发进程试图访问同一个内存区域，并且它们访问内存的方式不是原子的。 我们对之前的例子进行一些修改：

```
var data int
go func() { data++}()
if data == 0 {
    fmt.Println("the value is 0.")
} else {
   fmt.Printf("the value is %v.\n", data)
}
```

我们在这里添加了一个else子句，以便不管数据的值如何，总会得到一些输出。 请记住，正如它所写的那样，存在数据竞争，并且程序的输出将完全不确定。

程序中有一些操作需要独占访问共享资源。在这个例子中，我们找到三处：

- goroutine正在增加数据变量。
- if语句，它检查数据的值是否为0。
- fmt.Printf语句，用于检索输出数据的值。

有很多方法可以保护这些访问，Go有很好的方式来处理这个问题，解决这个问题的方法之一是让这些操作同步访问内存。 让我们看看该怎样做到这一点。

下面的代码不是Go的惯用法（我不建议你像这样解决数据竞争问题），但它很简单地演示了内存访问同步。如果这个例子中的任何类型，函数或方法对你来说都很陌生，那没问题。跟踪注释，关注同步访问内存的概念就行。

```
var memoryAccess sync.Mutex //1
var value int
go func() {
    memoryAccess.Lock() //2
    value++
    memoryAccess.Unlock() //3
}()

memoryAccess.Lock() //4
if value == 0 {
    fmt.Printf("the value is %v.\n", value)
} else {
    fmt.Printf("the value is %v.\n", value)
}
memoryAccess.Unlock() //5
```

1. 这里我们添加一个变量，它允许我们的代码同步对数据变量内存的访问。第三章的sync包会介绍sync.Mutex类型的细节。
2. 在这里我们声明，除非解锁，否则我们的goroutine应该独占访问此内存。
3. 在这里，我们声明这个对该内存的访问已经完成了。
4. 在这里，我们再次声明接下来的条件语句应该独占访问数据变量的内存。
5. 在这里，我们声明对内存的访问已经完成。

在这个例子中，我们为开发者制定了一个约定。任何时候开发人员都想访问data变量的内存，必须首先调用Lock，当完成访问操作时，必须调用Unlock。这两个语句之间的代码可以假定它拥有对数据的独占访问权; 我们已经成功地同步了对内存的访问。注意，如果开发者不遵循这个约定，我们就没有保证独占访问的权利！

你可能已经注意到，虽然我们已经解决了数据竞争，但我们并没有真正解决竞争条件！这个程序的操作顺序仍然不确定。 我们刚刚只是缩小了非确定性的范围。在这个例子中，仍然不确定goroutine是否会先执行，或者我们的if和else块是否都会执行。稍后，我们将探索正确解决这类问题的工具。

从表面上看，这似乎很简单：如果你发现你有这样的需求，添加点来同步访问内存！ 很简单，对吧？

但是！

确实，你可以通过同步访问内存来解决一些问题，但正如我们刚刚看到的，它不会自动解决数据竞争或逻辑正确性问题。 此外，它还可能导致维护和性能问题。

请注意，之前我们提到已经创建了一个声明需要对某些内存进行独占访问的约定。约定本身是没问题的，但实际开发中我们总是会丢三落四。更别说开发组里总会有这样或那样不遵守约定的人。值得庆幸的是，随后我们还将介绍一些更有效的方法。

以这种方式同步对内存的访问会导致性能下降。每次我们执行其中一项操作时，程序会暂停一段时间。 这带来了两个问题：

- 加锁的程序部分是否重复进入和退出？
- 加锁的程序对内存占用到底有多大？

要说清这两个问题简直是门艺术。

同步对内存的访问也与其他并发建模存在关联，我们将在下一节讨论这些问题。

死锁，活锁和锁的饥饿问题

前面的章节都是关于程序正确性的讨论，如果这些问题得到正确处理，程序永远不会给出错误的答案。不幸的是，即使你成功处理了这些类别的问题，还有另一类问题需要解决：死锁，活锁和饥饿。这些问题都涉及确保您的程序在任何时候都能够有效执行。如果处理不当，您的程序可能会进入某个状态中，最终停止运行。

死锁

死锁是所有并发进程都在彼此等待的状态。 在这种情况下，如果没有外部干预，程序将永远不会恢复。
如果这听起来很严峻，那是因为它确实很严峻！ Go运行时会检测到一些死锁（所有的例程必须被阻塞或“休眠”），但这对于帮助你防止死锁产生没有多大帮助。

为了帮助你更直观的认识死锁，我们先来看一个例子。同样的，跟着注释走，任何变量、函数、语句都不重要：

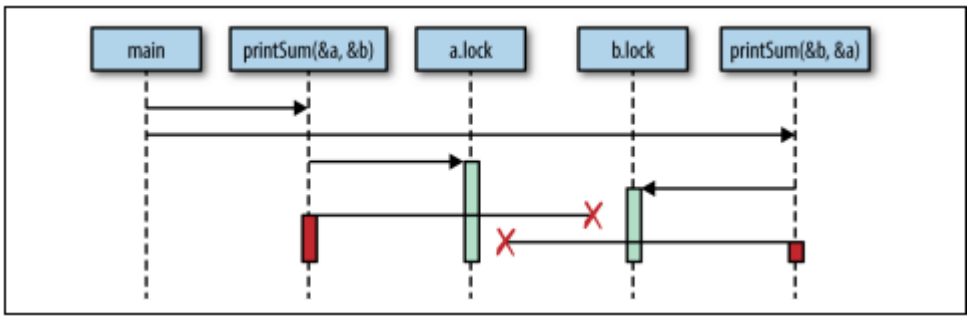
```
type value struct {
    mu    sync.Mutex
    value int
}

var wg sync.WaitGroup
printSum := func(v1, v2 *value) {
    defer wg.Done()
    v1.mu.Lock()          //1
    defer v1.mu.Unlock() //2

    time.Sleep(2 * time.Second) //3
    v2.mu.Lock()
    defer v2.mu.Unlock()

    fmt.Printf("sum=%v\n", v1.value+v2.value)
}

var a, b value
wg.Add(2)
go printSum(&a, &b)
go printSum(&b, &a)
wg.Wait()
```

1. 这里我们试图访问带锁的部分
2. 这里我们试图调用defer关键字释放锁
3. 这里我们添加休眠时间 以造成死锁

如果你试着运行这段程序，应该会看到这样的输出：

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

为什么？ 如果仔细观察，你将在此代码中看到计时问题。下面的时序图能清晰的展现问题所在：



实质上，我们创建了两个不能一起运转的齿轮：我们的第一个打印总和调用a锁定，然后尝试锁定b，但与此同时，我们打印总和的第二个调用锁定了b并尝试锁定a。 两个goroutine都无限地等待着彼此。

> 为了保持这个例子简单，我使用time.Sleep来触发死锁。 但是，这引入了竞争条件！ 你能找到它吗？
> 一个逻辑上“完美”的死锁将需要正确的同步。

这似乎很明显，为什么当我们以这种方式绘制图表时出现这种僵局，但我们会从更严格的定义中受益。事实证明，出现僵局时必定存在一些条件，1971年，埃德加科夫曼在一篇论文中列举了这些条件。这些条件现在称为科夫曼条件，是帮助检测，防止和纠正死锁的技术基础。

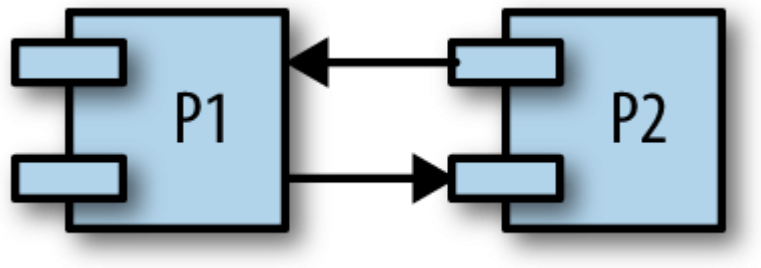
科夫曼条件如下：

相互排斥
并发进程在任何时候都拥有资源的独占权。

等待条件
并发进程必须同时持有资源并等待额外的资源。

没有抢占
并发进程持有的资源只能由该进程释放，因此它满足了这种情况。

循环等待
并发进程（P1）等待并发进程（P2），同时P2也在等待P1，因此也符合”循环等待”这一条件。



让我们来看看我们的设计程序，并确定它是否符合所有四个条件：

1. printSum函数确实需要a和b的独占权，所以它满足了这个条件。
2. 因为printSum保持a或b并等待另一个，所以它满足这个条件。
3. 我们没有任何办法让我们的goroutine被抢占。
4. 我们第一次调用printSum正在等待我们的第二次调用，反之亦然。

很好，我们亲手实现了死锁。

科夫曼条件同样有助于我们规避死锁。如果我们确保至少有一个条件不成立，就可以防止发生死锁。不幸的是，实际上这些条件很难推理，因此难以预防。网上大量充斥着被死锁困扰的开发人员的求助，一旦有人指出它就很明显，但通常需要另一双眼睛。

活锁

活锁是正在主动执行并发操作的程序，但这些操作无法向前移动程序的状态。

你有没有在走廊走向另一个人？ 她移动到一边让你通过，但你也是这样做的。 所以你转移到另一边，但她也是这样做的。 想象这会永远持续下去，这就是活锁。

接下来的这个例子，我不建议试图了解它的细节，直到你牢牢掌握sync包。 相反，我建议遵循代码标注来理解高亮，然后将注意力转移到包含示例核心的第二个代码块。

```
cadence := sync.NewCond(&sync.Mutex{})
go func() {
    for range time.Tick(1 * time.Millisecond) {

        cadence.Broadcast()
    }
}()

takeStep := func() {
    cadence.L.Lock()
    cadence.Wait()
    cadence.L.Unlock()
}

tryDir := func(dirName string, dir *int32, out *bytes.Buffer) bool { //1
    fmt.Fprintf(out, " %v", dirName)
    atomic.AddInt32(dir, 1) //2
    takeStep()              //3
    if atomic.LoadInt32(dir) == 1 {
        fmt.Fprint(out, ". Success!")
        return true
    }
    takeStep()
    atomic.AddInt32(dir, -1) //4
    return false
}

var left, right int32
tryLeft := func(out *bytes.Buffer) bool { return tryDir("left", &left, out) }
tryRight := func(out *bytes.Buffer) bool { return tryDir("right", &right, out) }
```

1. tryDir 允许一个人尝试向某个方向移动并返回，无论他们是否成功。 每个方向都表示为试图朝这个方向移动的次数。
2. 首先，我们通过将该方向递增1来朝着某个方向移动。 我们将在第3章详细讨论atomic包。现在，你只需要知道这个包的操作是原子操作。
3. 每个人必须以相同的速度或节奏移动。 takeStep模拟所有动作之间的恒定节奏。
4. 在这里，这个人意识到他们不能在这个方向上放弃。 我们通过将该方向递减1来表示这一点。

```
walk := func(walking *sync.WaitGroup, name string) {
    var out bytes.Buffer
    defer func() { fmt.Println(out.String()) }()
    defer walking.Done()
    fmt.Fprintf(&out, "%v is trying to scoot:", name)
    for i := 0; i < 5; i++ { //1
        if tryLeft(&out) || tryRight(&out) { //2
            return
        }
    }
    fmt.Fprintf(&out, "\n%v tosses her hands up in exasperation!", name)
```

```
}

var peopleInHallway sync.WaitGroup //3
peopleInHallway.Add(2)
go walk(&peopleInHallway, "Alice")
go walk(&peopleInHallway, "Barbara")
peopleInHallway.Wait()
```

1. 我对尝试次数进行了人为限制，以便该程序结束。 在一个有活锁的程序中，可能没有这种限制，这就是为什么它是一个现实工作中的问题。
2. 首先，这个人会试图向左走，如果失败了，会尝试向右走。
3. 这个变量为程序提供了等待，直到两个人都能够相互通过或放弃。

程序会产生如下输出：

```
Alice is trying to scoot: left right left right left right left right left right Alice tosses her hands up in exasperation!
Barbara is trying to scoot: left right left right left right left right left right
Barbara tosses her hands up in exasperation!
```

你可以看到Alice和Barbara在最终放弃之前持续交互。

这个例子演示了一个非常常见的活锁写入原因：两个或多个并发进程试图在没有协调的情况下防止死锁。 如果走廊里的人们一致认为只有一个人会移动，那么就不会有活锁：一个人静止不动，另一个人移动到另一边，他们会继续走路。

在我看来，活锁比死锁更难以发现，因为它看起来好像程序正在工作。 如果活锁程序在你的机器上运行，并且你查看了CPU利用率以确定它是否在执行任何操作，那么你可能会认为它是。 根据活锁的不同，它甚至可能会发出其他信号，使你认为它正在工作。 然而，一直以来，你的程序都扮演着走廊洗牌的永恒游戏。

活锁是饥饿的问题的一个子集。 接下来我们会讨论。

饥饿

饥饿是指并发进程无法获得执行工作所需的任何资源的情况。

当我们讨论活锁时，每个goroutine所缺乏的资源就是一个共享锁。 活锁需要与饥饿分开讨论，因为在活锁过程中，所有并发进程都是平等的，并且没有任何任务可以被完成。 更广泛地说，饥饿通常意味着有一个或多个贪婪的并发进程不公平地阻止一个或多个并发进程尽可能有效地完成工作，或者根本不可能完成工作。

下面这个例子展示了一个贪婪的goroutine和一个知足的goroutine：

```
var wg sync.WaitGroup
var sharedLock sync.Mutex
const runtime = 1*time.Second

greedyWorker := func() {
    defer wg.Done()

    var count int
    for begin := time.Now(); time.Since(begin) <= runtime; {
        sharedLock.Lock()
        time.Sleep(3*time.Nanosecond)
        sharedLock.Unlock()
        count++
    }

    fmt.Printf("Greedy worker was able to execute %v work loops\n", count)
}

politeWorker := func() {
    defer wg.Done()

    var count int
    for begin := time.Now(); time.Since(begin) <= runtime; {
        sharedLock.Lock()
        time.Sleep(1*time.Nanosecond)
        sharedLock.Unlock()

        sharedLock.Lock()
        time.Sleep(1*time.Nanosecond)
        sharedLock.Unlock()

        sharedLock.Lock()
        time.Sleep(1*time.Nanosecond)
        sharedLock.Unlock()

        count++
    }
    fmt.Printf("Polite worker was able to execute %v work loops.\n", count)
}

wg.Add(2)
go greedyWorker()
go politeWorker()

wg.Wait()
```

这个代码段会输出：

```
Polite worker was able to execute 289777 work loops. Greedy worker was able to execute 471287 work loops
```

greedy 贪婪地持有整个工作循环的共享锁，而polite 试图只在需要时才锁定。 二者都进行了相同数量的模拟工作（休眠时间为三纳秒），但正如你在相同的时间内看到的那样，greedy 几乎完成了两倍的工作量！

但在这里我们要清楚的了解到，greedy不必要的扩大了对共享锁的控制，并且(通过饥饿)阻碍了polite有效的执行。

我们在例子中使用计数的方式识别饥饿，在记录和抽样度量指标时这是一个很不错的方法。检测和解决饥饿的方法之一就是就是记录程序完成的时间，然后确定你的程序执行速度是否与预期的一样高。

> 值得一提的是，前面的代码示例也可以作为进行同步内存访问的性能分支示例。 因为同步访问内存的代价很高，所以扩大我们的锁定范围可能会产生额外的代价。 另一方面，正如我们所看到的那样，我们冒着令其他并发进程挨饿的风险。 如果你利用内存访问同步，你必须在性能粗粒度同步和公平性细粒度同步之间找到平衡点。 当开始调试应用程序时，我强烈建议你将内存访问同步仅限于程序的关键部分; 如果同步成为性能问题，则可以扩大范围。 除此之外，其他的解决方式可能会更难以操作。

因此，饥饿可能会导致程序无效或不正确。 前面的例子表明了执行效率是如何被降低的，如果你有一个非常贪婪的并发进程，以至于完全阻止另一个并发进程完成工作，那么你的问题就大了。

我们还需要考虑来自程序之外导致的饥饿问题。请记住，饥饿还可以产生于于CPU，内存，文件句柄和数据库连接：任何必须共享的资源都是饥饿的候选对象。

并发安全性

最后，我们谈到了开发并发代码的最困难的方面，它是所有其他问题的基础：人。 每行代码的编写者至少有一个人。

正如我们发现的，并发代码的难题由很多原因产生。 如果你是一名开发人员，并且在引入新功能时尝试解决所有这些问题，或修复程序中的错误，确定什么样的操作是正确的确实很困难。

如果你从零开始构建程序，需要建立一个合理的方式来模拟问题，但如果涉及到并发，就可能很难找到合适的抽象级别。你该如何向调用者暴露并发接口？应该使用什么样的技术使之简单而有效？应该支持什么样的并发规模？有不同的结构化方式来思考这些问题，但这些问题的解决方案有时候更接近艺术而不是技术。

作为一个对已有代码改造的开发人员，哪些代码利用并发并不总是很明显，如何安全地使用前人的代码有时更与智力无关。考虑下面的函数声明：

```
// CalculatePi 会在开始和结束位置之间计算Pi的数字
func CalculatePi(begin, end int64, pi *Pi)
```

以较高精度计算pi是最好的方法，但这个例子引发了很多问题：

1. 我该如何调用这个函数？
2. 我是否负责实例化此函数的多个并发调用？
3. 看起来函数的所有实例都将直接在我传入地址的Pi实例上运行；是由我负责同步对内存的访问，还是函数为我处理？

仅此一个函数就引发了这些问题。 想象一下任何规模适中的程序，你就可以开始理解并发可能带来的复杂性。

注释可以在这里创造奇迹。如果函数是这样写的呢？

```
// CalculatePi 会在开始和结束位置之间计算Pi的数字
//
// 在内部，CalculatePi会创建FLOOR（（end-begin）/ 2）递归调用
// CalculatePi的并发进程。 写入pi的同步锁由Pi结构内部处理。
func CalculatePi(begin, end int64, pi *Pi)
```

我们现在明白，调用者可以简单地调用该函数，而不必担心访问控制或同步问题。 重要的是，注释涵盖了这些方面：

1. 谁负责并发？
2. 问题空间如何映射到并发基元？
3. 谁负责同步？

当需要暴露涉及并发问题的函数、方法和变量时，请尽可能让你的同事和未来的自己受益：不一定非要写出冗长的注释，但请尽量覆盖上面的三个要素。

还要考虑到函数命名在含义上的模糊。也许我们应该让函数看起来没有副作用：

```
func CalculatePi(begin, end int64) []uint
```

这个函数的签名本身就消除了任何同步问题的疑问，但仍然留下了是否使用并发的问题。 我们可以再次修改签名，以明确的告诉调用者我们要返回什么：

```
func CalculatePi(begin, end int64) <-chan uint
```

现在我们首次看到了被称为channel(通道)的用法。随后在第三章会有更详细的介绍。修改后的函数签名表明CalculatePi将至少有一个goroutine，我们不应该为创建自己的goroutine而烦恼。

然后，这些修改会产生性能影响，必须予以考虑，我们又回到了平衡清晰度与性能之间的问题。 清晰性非常重要，因为我们希望将来尽可能使用此代码的人能够做正确的事情，并且由于显而易见的原因，性能很重要。 两者不是相互排斥的，但它们很难同时被处理的很好。

体会下我们在上面遇到的各种困难，并尝试将它们扩展到团队规模。

喔，真是个相当可怕的情景。

好消息是，Go已经逐步的给出了简单实用的解决方案。语言本身就具备了较强的可读性而又不失简约。Go鼓励并发建模的正确性，可组合性和可伸缩性。事实上，Go处理并发的方式实际上可以帮助你更清楚地表达问题。 让我们来看看为什么这么说。

优雅的面对复杂性

到目前为止，我描绘了一个非常严酷的场景。 并发在计算机科学中当然是一个困难的领域，但我希望你不要失去希望：这些问题不是无解的，使用Go的并发原语，可以更安全，更清晰地表达并发算法。 我们讨论的运行时长和沟通困难并不是Go能够完全解决的，但它们确实变得更加容易处理了。在这里，我们花一点时间来探索Go的并发原语，以助于更容易地对问题域进行建模，并更清楚地表达算法。

Go的运行时完成了大部分繁重的工作，并为Go的大部分并发优势提供了基础。详细部分我们放到第6章再讲，但现在让我们讨论下它是如何让人生变得轻松的。

我们先看看Go的并发，低延迟，自动执行垃圾收集器(GC)。开发者经常讨论GC是否是语言中必备的东西。批评者认为，GC阻止任何需要实时性能或确定性能特征的问题领域的工作——暂停程序中的所有活动来清理垃圾简直是不可接受的。但Go的GC所做的出色工作大大减少了开发者的心智负担。从Go 1.8开始，垃圾收集暂停一般在10到100微秒之间。

这对你有什么帮助？ 内存管理可能是计算机科学领域的另一个难题，如果与并发性结合使用，编写正确的代码变得非常困难。如果大多数开发人员不需要担心低至10微秒的暂停时间，那么Go通过不强迫你管理内存而更容易使用并发程序，更不用说跨并发进程了。

Go的运行时也会自动处理并发操作到操作系统线程上。这么说有些笼统，我们将在第三章的“Goroutines”一节中确切地说明这意味着什么。为了理解这对开发者的帮助，你需要知道的一点是它允许开发者直接将并发问题映射到结构体，而不是处理启动和管理线程的细节，并在可用线程间均匀映射逻辑。

例如，假设你编写了一个Web服务器，并且你希望每个连接都可以与其他连接同时处理。 在某些语言中，在Web服务器开始接受连接之前，你可能必须创建一个线程集合（通常称为线程池），然后将传入连接映射到线程。 然后，在你创建的每个线程中，你需要循环该线程上的所有连接，以确保它们都获得了一些CPU时间。 另外，你必须编写你的连接处理逻辑，使它可以与其他连接公平地共享。

相比之下，在Go中你可以编写一个函数，然后用go关键字预先调用它。 运行时会处理我们自动调用的所有内容！ 当你正在经历设计程序的过程时，你认为在哪种模型下你更有可能达到预期的并发效果？ 你认为哪个更可能是正确的？

Go的并发原语也使得解决更大的问题变得更容易。 正如我们将在第三章的“Channels”部分中看到的，Go的channel原语为并发进程之间的通信提供了可组合，并发安全的方式。

我已经掩盖了大部分细节，但我想给阐述一些关于Go如何在程序中处理并发，以帮助你以清晰和高效的方式解决问题。 这将是下一张的内容。 如果渴望立刻开始讨论代码，你可能会想转到第三章。

代码建模：序列化交互处理

并发与并行

什么是CSP

CSP在Go中的衍生物

Go的并发哲学

并发与并行

并发与并行是不同的，这一事实常常被忽视或误解。 在许多开发人员之间的对话中，这两个术语经常互换使用，意思是“与其他东西同时运行的东西”。有时在这种情况下使用“并行”这个词是正确的，但通常如果开发人员正在讨论代码， 他们真的应该使用“并发”这个词。

并发和并行之间的差异导致在建模代码时，演变成非常显著的抽象区别，Go充分利用了这一点。 让我们来看看这两个概念是如何不同的，以便我们能够理解这种抽象的力量。 我们将从一个非常简单的陈述开始：

> 并发是代码的一个属性; 并行是正在运行的程序的一个属性。

这是一个有趣的区别。我们编写我们的代码，以便它可以并行执行。 对？

那么，我们再来考虑一下。 如果所编写的代码的意图是两个程序块并行运行，那么当程序运行时，我是否有任何保证？ 如果我在只有一个内核的机器上运行代码会发生什么？ 你们中有些人可能会想，它会并行运行，但事实并非如此。

程序块可能表现为并行运行，但实际上他们是以一种连续的方式执行，而不是不可区分的。(单核)CPU上下文切换为在不同程序之间共享时间，在足够长的时间间隔内，这些任务表现为并行运行。如果我们要在两个内核的机器上运行相同的二进制文件，那么程序块可能确实是并行运行的。

这揭示了一些有趣且重要的事情。我们的代码可能不是并行的，而表现出来却有可能是并行的。并行是我们程序运行时的一个属性，而非代码。

第二个有趣的地方在于，我们发现运行时不知道我们的并发代码是否实际并行运行。对程序模型的抽象可以使我们区分并发和并行，并最终赋予程序力量和灵活性。

第三，并行性是时间或环境的函数。我们在之前的“原子性”中讨论了上下文的概念。在那里，上下文被定义为一个操作被认为是原子的边界。 在这里，它被定义为两个或多个操作可以被认为是并行的边界。

例如，如果我们的上下文是五秒钟的空间，并且我们运行了两个每秒需要运行一次的操作，那么我们会认为这些操作是并行运行的。 如果我们的情况是一秒钟，我们会认为这些操作是按顺序运行的。

就时间片而言，重新定义我们的上下文可能并不是很好，但记住上下文不受时间限制。 我们可以将上下文定义为程序运行的过程，操作系统线程或其机器。 这很重要，因为定义的上下文与并发性和正确性的概念密切相关。 就像原子操作根据上下文可以被视为原子操作一样，根据定义的上下文，并发操作是正确的。当然 这都是相对的。

这有点抽象，所以我们来看一个例子。 假设我们正在讨论的环境是你的电脑。 除了理论物理外，我们可以合理地预期在我的机器上执行的进程不会影响机器上进程的逻辑。 如果我们都启动计算器过程并开始执行一些简单的算术运算，那么我执行的计算不应该影响你执行的计算。

这个例子有点傻。但是如果我们把它分解，我们会看到所有的部分在起作用：我们的机器是上下文，进程是并发操作。 在这种情况下，我们选择通过独立的计算机，操作系统和流程来思考世界并行操作。 这些抽象使我们能够确认这一的思考是正确的。

> 使用单独的计算机似乎是一个有意义的例子，但个人计算机并不总是如此无处不在！ 直到20世纪70年代末，大型机才是常态，开发人员在同时考虑问题时使用的常见上下文是程序的过程。
> 现在许多开发人员正在使用分布式系统，它正在向另一种方向转移！ 我们现在开始考虑虚拟机管理程序，容器和虚拟机作为我们的并发环境。

我们可以合理地期望一台机器上的一个进程不受另一台机器上的进程的影响（假设它们不属于同一个分布式系统），但是我们可以期望同一台机器上的两个进程不会影响另一台机器上的逻辑吗？进程A可能会覆盖进程B正在读取的某些文件，或者在不安全的操作系统中，进程A甚至可能会破坏正在读取的进程B。

尽管如此，在流程层面上，事情仍然相对容易考虑。 如果我们回到我们的计算器示例，那么期望在同一台计算机上运行两个计算器进程的两个用户合理地期望他们的操作在逻辑上彼此隔离是合理的。 幸运的是，过程边界和操作系统帮助我们以合理的方式思考这些问题。 但是我们可以看到，开发人员开始担心并发问题，并且这个问题只会变得更糟。

如果我们再向下移动到操作系统线程边界，“为什么是并发编程如此困难”一节中列举的所有问题才真正出现：竞争条件，死锁，活锁和饥饿。 如果我们有一台机器上的所有用户都可以查看的计算器进程，那么并发逻辑就会变得更加困难。 我们不得不开始担心同步对内存的访问的影响并为正确的用户检索正确结果烦恼。

当我们开始向下移动抽象层时，对事物的建模变得更加难以推理，抽象对我们来说变得越来越重要。 换句话说，获得并发权越困难，访问容易编写的并发原语就越重要。 不幸的是，我们行业中的大多数并发逻辑都是以最高抽象层次之一编写的：系统线程。

在Go出现前，这是大多数流行编程语言的抽象链的最终解决方案。如果你想编写并发代码，你可以用线程来建模你的程序并同步它们之间的内存访问。 如果你有很多事情需要同时建模，并且你的机器不能处理那么多的线程，你会创建一个线程池并将你的操作复用到线程池中。

Go在该链中添加了另一个链接：goroutine。 此外，Go借鉴了着名计算机科学家托尼霍尔的着作中的几个概念，并引入了我们使用的新原语，即通道(channel)。

继续我们的推理，会发现在系统线程之下引入另一个抽象层次会带来更多困难，但有趣的是，事实并非如此。 它实际上使事情变得更容易 这是因为我们没有在操作系统线程的顶部添加另一个抽象层，我们(在用Go的时候)已经取代了他们。

当然，线程仍然存在，但是我们发现很少需要再从操作系统线程的角度考虑我们的问题空间。 相反，我们在goroutines和channel中建模，偶尔共享内存。 这会产生一些有趣的属性，我们会逐步探讨。但首先，让我们了解下Go哲学的基石：Tony Hoare的开创性论文“序列化交互”。

什么是CSP

当讨论Go时，你会经常听到人们围绕CSP进行争论。它会被称为Go成功的原因，或者是并发编程的灵丹妙药。虽然CSP使事情变得更容易，而且程序更加强大，但不幸的是这不是一个奇迹。 那它是什么？

CSP代表”Communicating Sequential Processes”，它既是一种技术，也是引入它的论文的名称。 1978年，Charles Antony Richard Hoare在计算机械协会（更通俗地称为ACM）上发表了这篇论文。

在该论文中，Hoare认为输入和输出是两个被忽视的编程原语，特别是在并发代码中。在Hoare撰写本文时，关于如何构造程序的研究仍在进行中，但大部分工作都是针对连续代码的技术：goto语句的使用正在讨论中，面向对象的思想开始萌发。并发并没有得到太多关注。 Hoare开始纠正这个问题，于是他的论文和CSP诞生了。

在1978年的论文中，CSP只是一个简单的编程语言，仅仅是为了展示顺序过程的交流能力; 实际上，他甚至在论文中说过：

> 因此，本论文中介绍的概念和符号……不应被视为适合用作编程语言，无论是抽象的还是具体的编程语言。

Hoare非常担心他所提供的技术没有进一步研究程序的正确性，而且这些技术可能不是以他自己的真实语言来表达的。 在接下来的六年里，CSP的概念被提炼成一种被称为过程演算的形式化表示，以便采取交流顺序过程的思想，并实际开始推理程序的正确性。 过程演算是数学建模并发系统的一种方式，也提供了代数法则来对这些系统进行转换，以分析它们的各种属性，例如效率和正确性。 虽然过程计算本身就是一个有趣的话题，但它们超出了本书的范围。 而且由于关于CSP的原始文件和从其演变而来的语言在很大程度上是Go的并发模型的灵感，所以我们将重点关注这些。

为了支持他的观点，Hoare的CSP程序设计语言包含原型来正确模拟输入和输出或进程之间的通信。Hoare将术语”进程”应用于逻辑的所有封装部分，这些部分需要输入来运行并产生其他过程将消耗的输出。当他写论文时，Hoare可能用“功能”这个词来描述如何构建社区中的程序。

为了进行流程之间的沟通，Hoare创建了输入和输出命令！ 用于将输入发送到进程中，以及？ 用于读取进程的输出。 每个命令都必须指定一个输出变量（在从流程中读取变量的情况下）或目标（在将输入发送到进程的情况下）。 有时候这两个过程会引用相同的东西，在这种情况下，这两个过程将被认为是相对应的。 换句话说，一个进程的输出将直接流入另一个进程的输入。 下表给出了几个例子。

表达式	说明
cardreader?card image	从cardreader读取卡并将其值（字符数组）分配给变量cardimage
lineprinter!line image	对于lineprinter，发送lineimage的值进行打印
X?(x, y)	从名为X的进程中，输入一对值并将它们分配给x和y
DIV!(3*a+b, 13)	从进程DIV输出2个指定的值
*[c:character; west?c → east!c]	从west读取所有字符并逐个放入east

Go的通道与之相似之处很明显。注意表格的最后一个例子中，来自west的输出是如何发送给变量c的，并且输入为east的输入来自同一个变量，这两个过程相对应。 在Hoare关于CSP的第一篇论文中，进程只能通过指定的来源和目的地进行通信。 他承认，这会导致代码作为库的嵌入问题，因为代码的使用者必须知道输入和输出的名称。 他同时提到注册所谓的“端口名称”的可能性，其中该名称可以在并行命令的头部声明，我们可能会将其命名为命名参数并命名为返回值。

该语言还利用了所谓的守护命令，Edgar Dijkstra在1974年撰写的一篇文章“Guarded commands, nondeterminacy and formal derivation of programs”中介绍了这一命令。 守护命令由→分割。 左侧是右测的有条件的守护，如果左测是错误

的，或者在命令的情况下返回假或退出，则右测永远不会执行。 将这些与Hoare的I/O命令结合起来为Hoare的通信进程奠定了基础，从而为Go的通道奠定了基础。

通过使用这些原语，Hoare演示了几个例子，并演示了一种支持通信建模的语言如何使解决问题变得更简单，更容易理解。 他使用的一些符号有点简单（perl程序员可能不同意），但他提出的问题有非常明确的解决方案。Go中的类似解决方案稍长一些，但也带有这种清晰度。

历史证明了Hoare是正确的; 然而，有趣的是，在Go发布之前，很少有语言确实将这些原语支持到语言中。 大多数流行的语言都倾向于共享和同步对CSP的信息传递风格的访问。也有例外，但不幸的是这些仅限于没有广泛采用的语言。 Go是第一批将CSP原理融入其核心的语言之一，并将这种并发编程风格带给了大众。 它的成功使得其他语言也试图添加这些原语。

内存访问同步本质上并不坏。 我们将在后面的章节中（在“Go的并发哲学”一节）中指出，有时在某些情况下共享内存是合适的，即使在Go中也是如此。 但是，共享内存模型可能难以正确使用——特别是在大型或复杂的程序中。 正是因为这个原因，并发才被认为是Go的优势之一：它从一开始就以CSP的原则为基础，因此很容易阅读，编写和推理。

CSP在Go中的衍生物

你可能会也，可能不会发现所有这些引人入胜的东西，但如果你正在阅读本书，你就有难以解决的问题，并且你想知道为什么这些问题很重要。 Go的做法有何不同，使它在并发性方面与其他流行语言不同？

我们在”并发与并行”章节提到，语言在操作系统线程和内存访问同步级别结束其抽象链是很常见的。Go采用不同的路线，用goroutines和channel的概念取而代之。

如果我们在抽象并发代码的两种方式中比较概念，我们可能会将goroutine与线程和通道相比较（这些基元只具有相似性，但希望能够进行比较帮助你获得更深入的理解）。 这些不同的抽象为我们做了什么？

Goroutines使我们不必从并行的角度思考我们的问题，而是让我们对问题进行模拟，使其更接近自然。 虽然我们讨论了并发和并行之间的区别，但这种差异如何影响我们对解决方案的建模可能并不明确。 我们来看一个例子。

比方说，我需要构建一个Web服务器，在客户端提交访问请求。 暂时搁置框架，用一种只提供线程抽象的语言，我可能会反思下列问题：

- 我的语言是否自然地支持线程，还是必须选择一个库？
- 我的线程限制边界应该在哪里？
- 此操作系统中的线程有多重？
- 我的程序将如何在处理线程中运行不同的操作系统？
- 我应该创建一个线程池来限制我创建的线程数量。 我如何找到最佳的数字？

所有这些都是需要考虑的重要事情，但是没有一个直接关注你正在尝试解决的问题。 你被放到了如何解决并行性问题的技术上。

如果我们退后一步并考虑原始问题，我们可以这样说：个人用户正在连接到我的终端并开启会话。 会话应该对他们的请求返回响应。 在Go中，我们几乎可以用代码直接表示这个问题：我们将为每个传入连接创建一个goroutine，在那里将请求放在那里（可能与其他数据/服务的goroutines进行通信），然后从 goroutine的函数返回信息。使用Go可以让我们自然地思考这个问题并直接映射到编码。

这是通过Go对我们的承诺实现的：goroutines是轻量级的，我们通常不必担心创建一个导致耗费很多资源。有时候需要考虑系统中有多少个goroutines正在运行，但是这么做是一个过早的优化。 将此与线程进行对比，你可以预先考虑这些问题。

不过这并不意味着这种对并发问题建模不重要。在使用Go的情况下，语言是围绕并发设计的，所以这种语言与它提供的并发原语是一致的。 这意味着更少的摩擦和更少的错误。

对问题的更直观自然的映射处理好处是巨大的，它也有一些有益的副作用。Go的运行时自动将goroutine多路复用到OS线程，并为我们管理其调度。 这意味着可以在不需要改变我们如何模拟问题的情况下对运行时进行优化; 这是经典的关注点分离。 随着并行性的进步，Go的运行时在更新，程序的性能也在提高。 留意Go的发布说明，偶尔你会看到类似的东西：

> 在Go 1.5中，goroutines的调度顺序已经改变。

并发和并行的分离还有另一个好处：由Go的运行时为你管理goroutines的调度，它可以检查像阻塞等待I/O的goroutines和智能地重新分配OS线程到未阻塞的goroutines之类的事情。 这也增加了你的代码的性能。我们将在第六章中更多地讨论Go的运行时为你做什么。

现实问题和Go代码之间更自然映射的另一个好处是提高了以并发方式建模的问题数量。 由于我们作为开发人员所面临的问题在现实中是并发的，使用Go可以在更细的粒度级别上编写并发代码，而不是我们在其他语言中可能会使用的思考方式；例如，回到我们的Web服务器示例，我们现在将为每个用户建立一个goroutine，而不是将多个连接复用到一个线程池中。 这种更精细的粒度使程序能够在主机上实现友好的并行扩展。

goroutine是Go提供的解决方案的一部分，从CSP概念衍生出的通道——channel和select语句同样有用。

例如，通道本身可与其他通道合并。 这使得编写大型系统变得更简单，因为莫可以通过组合输出来协调多个子系统的输入。 可以将输入通道与超时，执行取消或把消息组合到其他子系统。 与之相对应的，协调互斥是一个值得关注的话题。

select语句是Go的通道的补充，并且是赋予通道巨大威力的直接因素。 select语句允许你有效地等待事件，以统一的随机方式从竞争渠道中选择消息，如果没有消息等待，则继续操作。

这些由CSP和支持它的运行时所激发的奇妙基元就是Go的动力。我们将在这本书的其余部分来发现这些东西是如何工作的，为什么以及如何使用它们来编写出色的代码。

Go的并发哲学

CSP过去和现在都是Go设计的重要组成部分; 然而，Go还支持通过内存访问同步和遵循该技术的基元来编写并发代码的更传统手段。 同步和其他软件包中的结构和方法允许你执行锁定，创建资源池，抢占goroutine等。

这些能力对你来说将非常有用，因为它可以让你对选择编写哪种类型的并发代码来解决问题拥有更大的自由度，但它也可能有点混乱。 该语言的新手经常会得到这样的印象：并发CSP风格被认为是在Go中编写并发代码的唯一方法。 例如，在同步软件包的文档中，它说：

> sync包提供基本的同步原语，如互斥锁。 除了Once和WaitGroup类型之外，大部分类型都是供底层库例程使用的。
> 通过通道和通信可以更好地完成更高级别的信息交互。

在官方FAQ文档中提到：

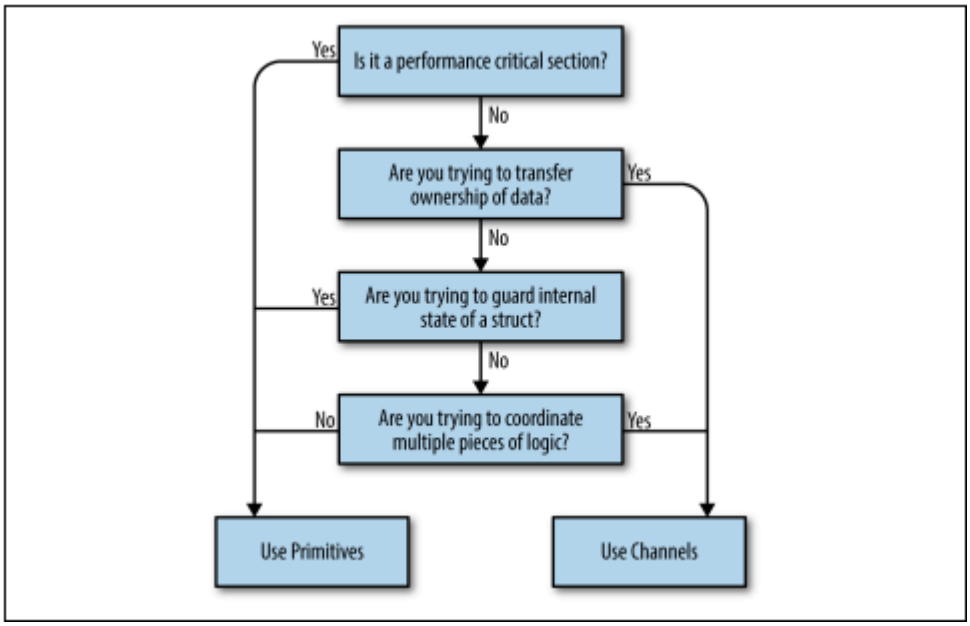
> 关于互斥锁，sync包实现它们，但我们希望Go编程风格将鼓励人们尝试更高级的技术。 特别是，考虑构建你的程序，
> 这样一次只有一个goroutine负责某个特定的数据。
> 不要通过共享内存进行通信。 相反，通过通信共享内存。

还有很多文章，讲座和采访，Go核心库大多都支持CSP风格，比如sync.Mutex。

因此，Go团队为什么选择公开内存访问同步原语会感到困惑是完全可以理解的。 更令人困惑的是，你会看到通常会出现的同步原语，看到人们抱怨过度使用通道，并且还听到一些Go团队成员表示可以使用它们。 这是来自Go Wiki的关于此事的引文：

> Go的格言之一是“通过沟通共享内存，不要通过共享内存进行通信”。
> 也就是说，Go确实在sync包中提供了传统的锁定机制。 大多数锁定问题都可以使用通道或传统锁来解决。
> 那么你应该使用哪个？
> 使用最具表现力和/或最简单的。

这是很好的建议，这是你在使用Go时经常看到的指导方针，但它有点含糊。 我们如何理解什么更具表现力和/或更简单？ 我们可以使用什么标准？ 幸运的是，可以使用一些指导来帮助我们做正确的事情。 正如我们将会看到的那样，主要区分的方式来自于试图管理并发性的地方：从内部到紧密的范围，或者在整个系统中。 下图列举了这些指标：



让我们一步接一步的看这幅图：

你想转移数据的所有权吗？
如果你有一些代码能够产生结果，并希望与另一部分代码共享这个结果，那么你真正在做的是转移那些数据的所有权。 如果你

熟悉不支持垃圾回收的语言的内存所有权概念，那么也可以称之为：数据所有者，使并发程序安全的一种方法是确保只有一个并发上下文拥有数据的所有权 。 通道可以帮助我们来传达这一概念。

这样做的一大好处是你可以创建缓冲通道来实现资源廉价的内存队列，从而将你的生产者与消费者分离。 另一个是通过使用通道，你可以隐式地将你的并发代码与其他并发代码组合在一起。

你是否试图保护结构的内部状态？
这是内存访问同步原语的一个很好的选择，也是一个非常强大的指示器，你不应该使用通道。 通过使用内存访问同步原语，你可以隐藏从呼叫者锁定关键部分的实现细节，但不会给调用者带来复杂性。 这是一个线程安全类型的小例子：

```
type Counter struct {
    mu sync.Mutex value int
}
func(c *Counter) Increment()
{
    c.mu.Lock()
    defer c.mu.Unlock()
    c.value++
}
```

如果你回想一下原子性的概念，我们可以说在这里所做的是定义了Counter类型的原子性范围。 调用增量可以被认为是原子的。

记住这里的关键词是”内部的”。 如果你发现自己超出锁定范围，这应该会引起重视。 尽量将锁限制在一个小的范围内。

你是否想要协调多个逻辑？
请记住，通道本质上比内存访问同步基元更具可组合性。 将锁分散在各个结构中听起来像是一场噩梦。

如果你因Go的选择语句而使用通道，且能够充当队列安全地传递，你会发现控制软件中出现的紧急复杂性要容易得多。 如果你发现自己在努力了解并发代码的工作原理，为什么会发生死锁或竞争，并且你正在使用基元，这可能是你需要切换到通道的一个很好的信号。

这是一个性能的关键部分吗？
这绝对不意味着，“我希望我的程序是高性能的，因此我只会使用互斥锁。”相反，如果你有一部分程序是已经分析过的，并且事实证明它是一个主要的瓶颈，当你发现这里比程序的其余部分慢一些，使用内存访问同步原语可以帮助这个关键部分在负载下执行。由于通道使用内存访问同步来操作，因此它们只能更慢。

希望这可以清楚地说明是否利用CSP风格的并发或内存访问同步。 还有其他一些模式和做法在使用操作系统线程作为抽象并发的方式的语言中很有用。 例如，像线程池这样的东西经常出现。 因为这些抽象的大部分是针对OS线程的优点和缺点的，所以使用Go时的一个很好的经验法则是放弃这些模式。

这并不是说它们根本没有用处，而是Go中的用例受到了更多限制。 坚持用goroutines为你的问题建模，用它们来代表你的工作流程的并发部分，并且不要害怕在启动它们时变得自由。 你很可能需要重新构建你的程序，而不是关注你的硬件可以支持多少个goroutines的上限。

Go的并发理念可以这样概括：为了简单起见，在可能的情况下使用通道，并且像免费资源一样处理goroutine(而不需要过多过早的考虑资源占用情况)。

Go的并发构建模块

在这一章，我们会讨论Go丰富的并发支持。到本章结束时，你将对相关语法，函数和包以及它们的功能有较为清晰的理解。

Goroutines

Goroutine是Go中最基本的组织单位之一，所以了解它是什么以及它如何工作是非常重要的。事实上，每个Go程序至少拥有一个：main gotoutine，当程序开始时会自动创建并启动。在几乎所有Go程序中，你都可能会发现自己迟早加入到一个gotoutine中，以帮助自己解决问题。那么它到底是什么？

简单来说，gotoutine是一个并发的函数（记住：不一定是并行）和其他代码一起运行。你可以简单的通过将go关键字放在函数前面来启动它：

```
func main() {
    go sayHello()
    // continue doing other things
}

func sayHello() {
    fmt.Println("hello")
}
```

对于匿名函数，同样也能这么干，从下面这个例子你可以看得很明白。在下面的例子中，我们不是从一个函数建立一个goroutine，而是从一个匿名函数创建一个goroutine：

```
go func() {
fmt.Println("hello")
}()// 1
// continue doing other things
```

1. 注意这里的()，我们必须立刻调用匿名函数来使go关键字有效。

或者，你可以将函数分配给一个变量，并像这样调用它：

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

看起来很简单，对吧。我们可以用一个函数和一个关键字创建一个并发逻辑块，这就是启动goroutine所需要知道的全部。当然，关于如何正确使用它，对它进行同步以及如何组织它还有很多需要说明的内容。本章接下来的部分会深入介绍goroutine及它是如何工作的。如果你只想编写一些可以在goroutine中正确运行的代码，那么可以考虑直接跳到下一章。

那么让我们来看看发生在幕后的事情：goroutine实际上是如何工作的？ 是OS线程吗？ 绿色线程？ 我们可以创建多少个？

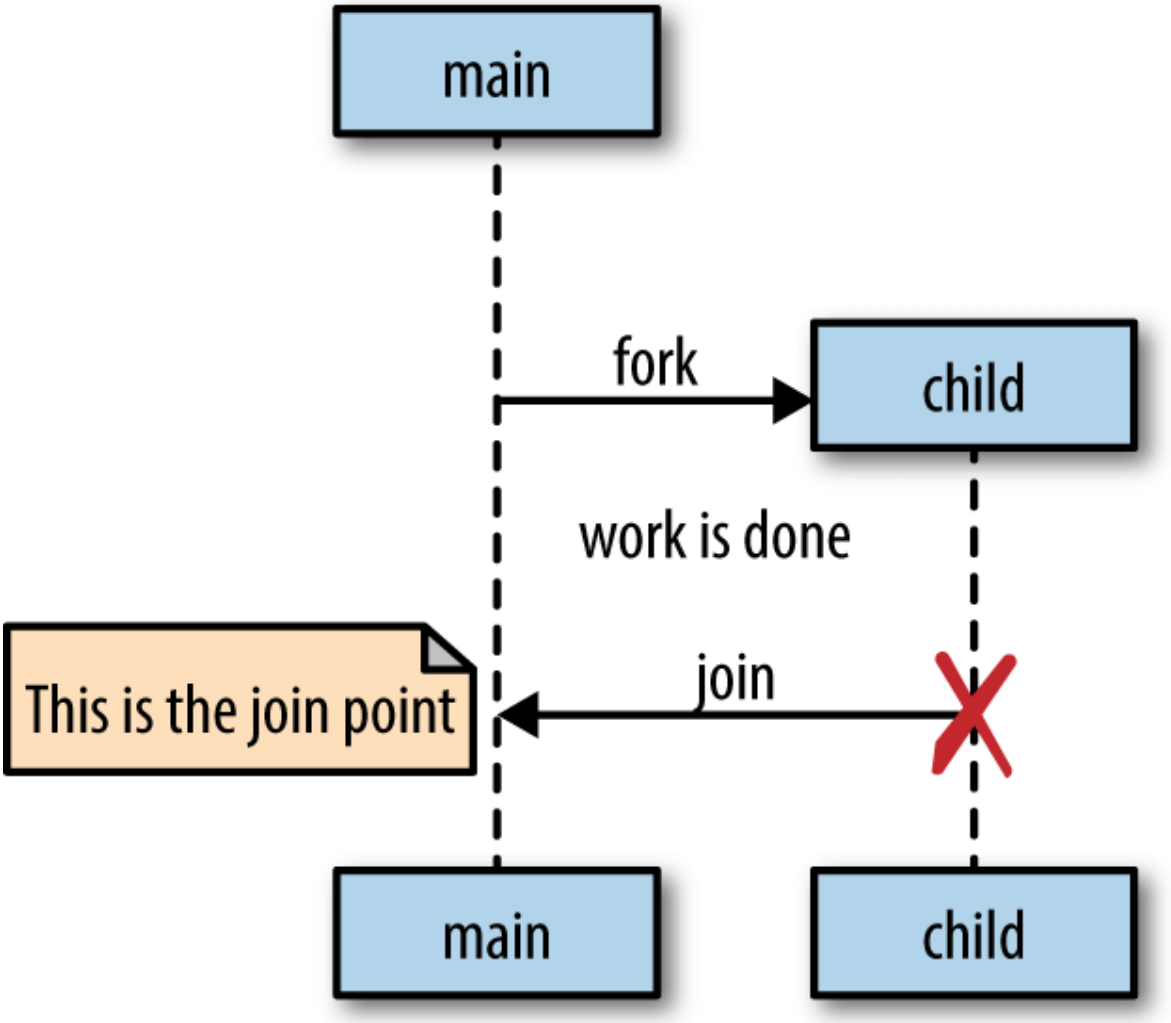
Goroutines对Go来说是独一无二的（尽管其他一些语言有类似的并发原语）。它们不是操作系统线程，它们不完全是绿色的线程(由语言运行时管理的线程)，它们是更高级别的抽象，被称为协程(coroutines)。协程是非抢占的并发子程序，也就是说，它们不能被中断。

Go的独特之处在于goutine与Go的运行时深度整合。Goroutine没有定义自己的暂停或再入点; Go的运行时观察着goroutine的行为，并在阻塞时自动挂起它们，然后在它们变畅通时恢复它们。在某种程度上，这使得它们可以抢占，但只是在goroutine被阻止的地方。它是运行时和goroutine逻辑之间的一种优雅合作关系。 因此，goroutine可以被认为是一种特殊的协程。

协程，因此可以被认为是goroutine的隐式并发构造，但并发并非协程自带的属性：某些东西必须能够同时托管几个协程，并给每个协程执行的机会，否则它们无法实现并发。当然，有可能有几个协程按顺序执行，但看起来就像并行一样，在Go中这样的情况比较常见。

Go的宿主机制实现了所谓的M：N调度器，这意味着它将M个绿色线程映射到N个系统线程。 Goroutines随后被安排在绿色线程上。 当我们拥有比绿色线程更多的goroutine时，调度程序处理可用线程间goroutines的分布，并确保当这些goroutine被阻塞时，可以运行其他goroutines。我们将在第六章讨论所有这些机制是如何工作的，但在这里我们将介绍Go如何对并发进行建模。

Go遵循称为fork-join模型的并发模型.fork这个词指的是在程序中的任何一点，它都可以将一个子执行的分支分离出来，以便与其父代同时运行。join这个词指的是这样一个事实，即在将来的某个时候，这些并发的执行分支将重新组合在一起。子分支重新加入的地方称为连接点。这里有一个图形表示来帮助你理解它：



go关键字为Go程序实现了fork，fork的执行者是goroutine，让我们回到之前的例子：

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

sayHello函数会在属于它的goroutine上运行，与此同时程序的其他部分继续执行。在这个例子中，没有连接点。执行sayHello的goroutine将在未来某个不确定的时间退出，并且该程序的其余部分将继续执行。

然而，这个例子存在一个问题：我们不确定sayHello函数是否可以运行。goroutine将被创建并交由Go的运行时安排执行，但在main goroutine退出前它实际上可能没有机会运行。

事实上，由于我们为了简单而省略了其他主要功能部分，所以当我们运行这个小例子时，几乎可以肯定的是，程序将在主办sayHello调用的goroutine开始之前完成执行。 因此，你不会看到打印到标准输出的单词“hello”。 你可以在创建goroutine之后为main goroutine添加一段休眠时间，但请记住，这实际上并不创建一个连接点，只是一个竞争条件。如果你记得第一章，你会增加退出前goroutine将运行的可能性，但你无法保证它。加入连接点是确保程序正确性并消除竞争条件的保证。

为了创建一个连接点，你必须同步main goroutine和sayHello goroutine。 这可以通过多种方式完成，但我将使用sync包中提供的一个解决方案：sync.WaitGroup。现在了解这个示例如何创建一个连接点并不重要，只是需要清楚它在两个goroutine之间创建了一个连接点。 这是我们的示例版本：

```
var wg sync.WaitGroup
sayHello := func() {
    defer wg.Done()
    fmt.Println("hello")
}
wg.Add(1)
go sayHello()
wg.Wait() //1
```

1. 在这里加入连接点。

这会输出：

```
hello
```

这个例子明确的阻塞了main goroutine，直到承载sayHello函数的main goroutine终止。你将在随后的sync包章节了解到更详细的内容。

我们在示例中使用了匿名函数。让我们把注意力转移到闭包。闭包围绕它们创建的词法范围，从而捕捉变量。如果在goroutine中使用闭包，闭包是否在这些变量或原始引用的副本上运行？让我们试试看：

```
var wg sync.WaitGroup
salutation := "hello"
wg.Add(1)
go func() {
    defer wg.Done()
    salutation = "welcome" // 1
}()
wg.Wait()
fmt.Println(salutation)
```

你认为salutation的值是”hello”还是”welcome”？运行后会看到：

```
wlecome
```

有趣！事实证明，goroutine在它创建的同一地址空间内执行，因此我们的程序打印出“welcome”。让我们再来尝试一个例子。 你认为这个程序会输出什么？

```
var wg sync.WaitGroup
for _, salutation := range []string{"hello", "greetings", "good day"} {
    wg.Add(1)
    go func() {
        defer wg.Done()
        fmt.Println(salutation) // 1
    }()
}
wg.Wait()
```

1. 这里我们测试打印字符串切片创建的循环变量salutation。

答案比大多数人所预期的不同，而且是Go中为数不多的令人惊讶的事情之一。 大多数人直觉上认为这会以某种不确定的顺序打印出“hello”，”greeting”和“good day”，但实际上：

```
good day
good day
good day
```

这有点令人惊讶。让我们来看看这里发生了什么。 在这个例子中，goroutine正在运行一个已经关闭迭代变量salutation的闭包，它有一个字符串类型。 当我们的循环迭代时，salutation被分配给切片中的下一个字符串值。 由于运行时调度器安排的goroutine可能会在将来的任何时间点运行，因此不确定在goroutine内将打印哪些值。 在我的机器上，在goroutines开始之前，循环很可能会退出。 这意味着salutation变量超出了范围。 然后会发生什么？ goroutines仍然可以引用已经超出范围的东西吗？ 这个goroutine会访问可能已经被回收的内存吗？

这是关于Go如何管理内存的一个有趣的侧面说明。Go运行时足够敏锐地知道对salutation变量的引用仍然保留，因此会将内存传输到堆中，以便goroutine可以继续访问它。

在这个例子中，循环在任何goroutines开始运行之前退出，所以salutation转移到堆中，并保存对字符串切片“good day”中最后一个值的引用。所以会看到“good day”打印三次 。 编写该循环的正确方法是将salutation的副本传递给闭包，以便在运行goroutine时，它将对来自其循环迭代的数据进行操作：

```
var wg sync.WaitGroup
for _, salutation := range []string{"hello", "greetings", "good day"} {
    wg.Add(1)
    go func(salutation string) { // 1
        defer wg.Done()
        fmt.Println(salutation)
    }(salutation) // 2
}
wg.Wait()
```

1. 在这里我们声明了一个参数，和其他的函数看起来差不多。我们将原始的salutation变量映射到更加明显的位置。
2. 在这里，我们将当前迭代的变量传递给闭包。 一个字符串的副本被创建，从而确保当goroutine运行时，我们引用正确的字符串。

正如我们所看到的，我们得到的输出看起来没那么奇怪了：

```
good day
hello
greetings
```

这个例子的行为和我们预期的一样，只是稍微更冗长。多运行几次，输出顺序可能不同。

goroutine在相同的地址空间内运行，Go的编译器很好地处理了内存中的固定变量，因此goroutine不会意外地访问释放的内存，这允许开发人员专注于他们的问题而不是内存管理。

由于多个goroutine可以在相同的地址空间上运行，我们仍然需要担心同步问题。正如我们已经讨论过的，可以选择同步访问共享内存的例程访问，也可以使用CSP原语通过通信共享内存。

goroutines的另一个好处是它们非常轻巧。 这是官方FAQ的摘录：

> 新建立一个goroutine有几千字节，这样的大小几乎总是够用的。 如果出现不够用的情况，运行时会自动增加（并缩小）用于存储堆栈的内存，从而允许许多goroutine存在适量的内存中。CPU开销平均每个函数调用大约三个廉价指令。 在相同的地址空间中创建数十万个goroutines是可以的。如果goroutines只是执行等同于线程的任务，那么系统资源的占用会更小。

每个goroutine几kb，那根本不是个事儿。让我们来亲手试着确认下。在此之前，我们必须了解一个关于goroutine的有趣的事：垃圾收集器不会收集以下形式的goroutines。如果我写出以下代码：

```
go func() {
// <操作会在这里永久阻塞>
}()
// Do work
```

这个goroutine将一直存在，直到整个程序退出。我们会在第四章的“防止Goroutine泄漏”中详细的聊一聊这个话题。

接下来，让我们回来看看该怎么写个例子来衡量一个goroutine的实际大小。

我们将goroutine不被垃圾收集的事实与运行时的自省能力结合起来，并测量在goroutine创建之前和之后分配的内存量：

```
memConsumed := func() uint64 {
    runtime.GC()
    var s runtime.MemStats
    runtime.ReadMemStats(&s)
    return s.Sys
}

var c <-chan interface{}
var wg sync.WaitGroup
noop := func() { wg.Done(); <-c } // 1

const numGoroutines = 1e4 // 2
wg.Add(numGoroutines)
before := memConsumed() // 3
for i := numGoroutines; i > 0; i-- {
    go noop()
}
wg.Wait()
after := memConsumed() // 4
fmt.Printf("%.3fkb", float64(after-before)/numGoroutines/1000)
```

1. 我们需要一个永不退出的goroutine，以便我们可以将它们中的一部分保存在内存中进行测量。 不要担心我们目前如何实现这一目标。 只知道这个goroutine不会退出，直到这个过程结束。
2. 这里我们定义要创建的goroutines的数量。 我们将使用大数定律渐近地逼近一个goroutine的大小。
3. 这里测量创建分区之前所消耗的内存量。
4. 这里测量创建goroutines后消耗的内存量。

在控制台会输出：

```
2.817kb
```

2017年7月这本书出版，go1.9发布于2017年8月24日，那么假设作者用的是当时最新的1.8版。译者用windows系统，go 1.10.1版，这个数字在8.908kb~9.186kb上下浮动。在centos6.4上测试，这个数字为2.748kb。

看起来文档是正确的。这个例子虽然有些理想化，但仍然让我们了解可能创建多少个goroutines有了大致的了解。

在我的笔记本上，我有8G内存，这意味着理论上我可以支持数百万的goroutines。当然，这忽略了在电脑上运行的其他东西。但这个快速估算的结果表明了goroutine是多么的轻量级。

存在一些可能会影响我们的goroutine规模的因素，例如上下文切换，即当某个并发进程承载的某些内容必须保存其状态以切换到其他进程时。如果我们有太多的并发进程，上下文切换可能花费所有的CPU时间，并且无法完成任何实际工作。在操作系统级别，使用线程，这样做代价可能会非常高昂。操作系统线程必须保存寄存器值，查找表和内存映射等内容，才能在操作成功后切换回当前线程。 然后它必须为传入线程加载相同的信息。

在软件中的上下文切换代价相对小得多。在软件定义的调度程序下，运行时可以更具选择性地进行持久检索，例如如何持久化以及何时发生持续化。我们来看看操作系统线程和goroutines之间上下文切换的相对性能。 首先，我们将利用Linux内置的基准测试套件来测量在同一内核的两个线程之间发送消息需要多长时间：

```
taskset -c 0 perf bench sched pipe -T
```

这会输出：

```
# Running 'sched/pipe' benchmark:
# Executed 1000000 pipe operations between two threads

     Total time: 2.935 [sec]
       2.935784 usecs/op
          340624 ops/sec
```

这个基准测量实际上是衡量在一个线程上发送和接收消息所需的时间，所以我们将把结果分成两部分。 每个上下文切换1.467微秒。 这看起来不算太坏，但让我们先别急着下判断，再来比较下goroutine之间的上下文切换。

我们将使用Go构建一个类似的基准测试。下面的代码涉及到一些尚未讨论过的东西，所以如果有什么困惑的话，只需根据注释关注结果即可。 以下示例将创建两个goroutine并在它们之间发送消息：

```
func BenchmarkContextSwitch(b *testing.B) {
    var wg sync.WaitGroup
    begin := make(chan struct{})
    c := make(chan struct{})

    var token struct{}
    sender := func() {
        defer wg.Done()
        <-begin //1
        for i := 0; i < b.N; i++ {
            c <- token //2
        }
    }
    receiver := func() {
        defer wg.Done()
        <-begin //1
        for i := 0; i < b.N; i++ {
            <-c //3
        }
    }

    wg.Add(2)
    go sender()
    go receiver()
    b.StartTimer() //4
    close(begin)   //5
    wg.Wait()
}
```

1. 这里会被阻塞，直到接受到数据。我们不希望设置和启动goroutine影响上下文切换的度量。
2. 在这里向接收者发送数据。struct{}{}是空结构体且不占用内存；这样我们就可以做到只测量发送信息所需要的时间。
3. 在这里，我们接收传递过来的数据，但不做任何事。
4. 开始启动计时器。
5. 在这里我们通知发送和接收的goroutine启动。

我们运行该基准测试，指定只使用一个CPU，以便与之前的Linux基准测试想比较，我们来看看结果：

```
go test -bench=. -cpu=1 /src/gos-concurrency-building-blocks/goroutines/fig-ctx-switch_test.go
```

BenchmarkContextSwitch	5000000	225ns/op
PASS		
ok	command-line-arguments	1.393s

每个上下文切换225 ns，哇！ 这是0.225µs，比我机器上的操作系统上下文切换快92%，如果你记得1.467µs的话。很难说有多少goroutines会导致过多的上下文切换，但我们可以很自然地说上限可能不会成为使用goroutines的障碍。

sync包

sync包包含对低级别内存访问同步最有用的并发原语。 如果你使用的是主要通过内存访问同步处理并发的语言，那么这些类型可能已经很熟悉了。Go与这些语言的区别在于，Go在内存访问同步基元的基础上构建了一组新的并发基元，并为使用者提供扩展的内容。 正如我们在之前“Go的并发哲学”中讨论的那样，这些操作都有其用处——主要体现在小的作用域中，例如结构体。 你可以自行决定何时内存访问同步是最适当的。 接下来，让我们开始看看sync包暴露的各种基元。

WaitGroup

如果你不关心并发操作的结果，或者有其他方式收集结果，那么WaitGroup是等待一组并发操作完成的好方法。如果这两个条件都不成立，我建议你改用channel和select语句。WaitGroup非常有用，我先介绍它，以便在后续章节中使用它。以下是使用WaitGroup等待goroutine完成的基本示例：

```
var wg sync.WaitGroup

wg.Add(1) //1
go func() {
    defer wg.Done() //2
    fmt.Println("1st goroutine sleeping...")
    time.Sleep(1)
}()

wg.Add(1) //1
go func() {
    defer wg.Done() //2
    fmt.Println("2nd goroutine sleeping...")
    time.Sleep(2)
}()

wg.Wait() //3
fmt.Println("All goroutines complete.")
```

1. 这里我们调用Add并传入参数1来表示一个goroutine正在开始。
2. 在这里我们使用defer关键字来调用Done，以确保在退出goroutine的闭包之前，向WaitGroup表明了我们已经退出。
3. 在这里，我们调用Wait，这将main goroutine，直到所有的goroutine都表明它们已经退出。

这会输出：

```
2nd goroutine sleeping...
1st goroutine sleeping...
All goroutines complete.
```

你可以把WaitGroup视作一个安全的并发计数器：调用Add增加计数，调用Done减少计数。调用Wait会阻塞并等待至计数器归零。

请注意，Add的调用是在goroutines之外完成的。 如果没有这样做，我们会引入一个数据竞争条件，因为我们没有对goroutine做任何调度顺序上的保证; 我们可能在任何一个goroutines开始前触发Wait调用。 如果Add的调用被放置在goroutines的闭包中，对Wait的调用可能完全没有阻塞地返回，因为Add没有被执行。

通常情况下，尽可能与要跟踪的goroutine就近且成对的调用Add，但有时候会一次性调用Add来跟踪一组goroutine。我通常会做这样的循环：

```
hello := func(wg *sync.WaitGroup, id int) {
    defer wg.Done()
    fmt.Printf("Hello from %v!\n", id)
}

const numGreeters = 5
var wg sync.WaitGroup
wg.Add(numGreeters)
for i := 0; i < numGreeters; i++ {
    go hello(&wg, i+1)
```

```
}
wg.Wait()
```

这会输出：

```
Hello from 5!
Hello from 4!
Hello from 3!
Hello from 2!
Hello from 1!
```

Mutex和RWMutex

如果你对通过内存访问同步处理并发的语言很熟悉，那么你可能会立即明白Mutex的使用方法。如果你没有这样的经验，没关系，Mutex很容易理解。Mutex代表”mutual exclusion(互斥)”。互斥提供了一种并发安全的方式来表示对共享资源访问的独占。下面是一个简单的两个goroutine，它们试图增加和减少一个公共值;，并使用Mutex来同步访问：

```
var count int
var lock sync.Mutex

increment := func() {
    lock.Lock() // 1
    defer lock.Unlock() // 2
    count++
    fmt.Printf("Incrementing: %d\n", count)
}

decrement := func() {
    lock.Lock() // 1
    defer lock.Unlock() // 2
    count--
    fmt.Printf("Decrementing: %d\n", count)
}

// Increment
var arithmetic sync.WaitGroup
for i := 0; i <= 5; i++ {
    arithmetic.Add(1)
    go func() {
        defer arithmetic.Done()
        increment()
    }()

}

// Decrement
for i := 0; i <= 5; i++ {
    arithmetic.Add(1)
    go func() {
        defer arithmetic.Done()
        decrement()
    }()
}

arithmetic.Wait()
fmt.Println("Arithmetic complete.")
```

1. 在这里，我们要求独占使用关键部分 - 在这种情况下，count变量由互斥锁保护。
2. 这里表明我们已经完成了对共享部分的锁定。

这会输出：

```
Decrementing: -1
Incrementing: 0
Decrementing: -1
Incrementing: 0
Decrementing: -1
Decrementing: -2
Decrementing: -3
```

```
Incrementing: -2
Decrementing: -3
Incrementing: -2
Incrementing: -1
Incrementing: 0 Arithmetic complete.
```

你会注意到我们总是使用defer在延迟声明中调用解锁。 使用互斥锁时，这是一个非常常见的习惯用法，以确保调用始终执行，即使在发生恐慌时也是如此。否则一旦未能解除锁定，可能会导致你的程序陷入死锁。

被锁定部分是程序的性能瓶颈，进入和退出锁定的成本有点高，因此人们通常尽量减少锁定涉及的范围。

可能在多个并发进程之间共享的内存并不是都要读取和写入，出于这样的考虑，你可以使用另一个类型的互斥锁：sync.RWMutex。

sync.RWMutex与Mutex在概念上是一样的：它保护对内存的访问；不过，RWMutex可以给你更多地控制方式。 你可以请求锁定进行读取，在这种情况下，你将被授予读取权限，除非锁定正在进行写入操作。 这意味着，只要没有别的东西占用写操作，任意数量的读取者就可以进行读取操作。 下面是一个演示生产者的示例：

```
producer := func(wg *sync.WaitGroup, l sync.Locker) { //1
    defer wg.Done()
    for i := 5; i > 0; i-- {
        l.Lock()
        l.Unlock()
        time.Sleep(1) //2
    }
}

observer := func(wg *sync.WaitGroup, l sync.Locker) {
    defer wg.Done()
    l.Lock()
    defer l.Unlock()
}

test := func(count int, mutex, rwMutex sync.Locker) time.Duration {
    var wg sync.WaitGroup
    wg.Add(count + 1)
    beginTestTime := time.Now()
    go producer(&wg, mutex)
    for i := count; i > 0; i-- {
        go observer(&wg, rwMutex)
    }

    wg.Wait()
    return time.Since(beginTestTime)
}

tw := tabwriter.NewWriter(os.Stdout, 0, 1, 2, ' ', 0)
defer tw.Flush()

var m sync.RWMutex
fmt.Fprintf(tw, "Readers\tRWMutext\tMutex\n")
for i := 0; i < 20; i++ {
    count := int(math.Pow(2, float64(i)))
    fmt.Fprintf(
        tw, "%d\t%v\t%v\n", count,
        test(count, &m, m.RLocker()), test(count, &m, &m),
    )
}
```

1. producer函数的第二个参数是类型sync.Locker。 该接口有两种方法，锁定和解锁，互斥和RWMutex类型都适用。

2. 在这里，我们让producer休眠一秒钟，使其不那么活跃。

这会输出：

```
Readers  RWMutext  Mutex
1        5ms       5ms
2        5ms       5ms
4        5ms       5ms
8        5ms       5ms
16       5ms       5ms
32       5ms       5ms
64       5ms       5ms
128      5ms       5ms
256      5ms       5ms
512      5ms       5ms
1024     5ms       5ms
2048     5ms       5ms
4096     6ms       7ms
8192     8ms       8ms
16384    7ms       8ms
32768    9ms       11ms
65536    12ms      15ms
131072   29ms      31ms
262144   61ms      68ms
524288   121ms     137ms
```

你可以通过这个例子看到，RWMutext在大量级上相对于Mutex是有性能优势的，不过这同样取决于你在锁住的部分做了什么。通常建议在逻辑上合理的情况下使用RWMutex而不是Mutex。

Cond

Cond的文档很好的描述了其存在的目的：

> Cond实现了一个条件变量，用于等待或宣布事件发生时goroutine的交汇点。

在这个定义中，“事件”是指两个或更多的goroutine之间的任何信号，仅指事件发生了，不包含其他任何信息。 通常，你可能想要在收到某个goroutine信号前令其处于等待状态。 如果我们要在不使用Cond的情况下实现这一点，那么一个粗暴的方法就是使用无限循环：

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

然而这会导致消耗一个内核的所有周期。我们可以引入time.sleep来改善这一点：

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

这样就看起来好点了，但执行效率依然很低效，而且你需要显示标明需要休眠多久：太长或太短都会不必要的消耗无谓的CPU时间。如果有一种方法可以让goroutine有效地睡眠，直到唤醒并检查其状态，那将会更好。这种需求简直是为Cond量身定制的，使用它我们可以这样改造上面的例子：

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

1. 这里我们实例化一个新的Cond。NewCond函数传入的参数实现了sync.Locker类型。Cond类型允许以并行安全的方式与其他goroutines协调。
2. 在这里我们进行锁定。这一步很必要，因为Wait的调用会执行解锁并暂停该goroutine。
3. 在这里我们进入暂停状态，这是阻塞的，直到接收到通知。
4. 这里执行解锁，这一步很必要，因为当调用退出时，它会c.L上调用Lock。

这个例子相对之前的效率就比较高了。请注意，对Wait的调用不仅仅是阻塞，它暂停当前的goroutine，允许其他goroutine在操作系统线程上运行。当你调用Wait时，还会发生其他一些事情：进入Wait后，Cond的变量Locker将调用Unlock，并在退出Wait时，Cond变量的Locker上会调用Lock。 在我看来，这有点让人不习惯； 这实际上是该方法的隐藏副作用。 看起来我们在等待条件发生的整个过程中都持有这个锁，但事实并非如此。 当你检查代码时，需要留意这一点。

让我们扩展这个例子，来看看等待信号的goroutine和发送信号的goroutine该怎么写。假设我们有一个固定长度为2的队列，并且我们要将10个元素放入队列中。 我们希望一有空间就能放入，所以在队列中有空间时需要立刻通知：

```
c := sync.NewCond(&sync.Mutex{})    //1
queue := make([]interface{}, 0, 10) //2

removeFromQueue := func(delay time.Duration) {
    time.Sleep(delay)
    c.L.Lock()        //8
    queue = queue[1:] //9
    fmt.Println("Removed from queue")
    c.L.Unlock() //10
    c.Signal()   //11
}
```

```
for i := 0; i < 10; i++ {
    c.L.Lock() //3
    for len(queue) == 2 { //4
        c.Wait() //5
    }
    fmt.Println("Adding to queue")
    queue = append(queue, struct{}{})
    go removeFromQueue(1 * time.Second) //6
    c.L.Unlock()                        //7
}
```

1. 首先，我们使用一个标准的sync.Mutex作为Locker来创建Cond。
2. 接下来，我们创建一个长度为零的切片。 由于我们知道最终会添加10个元素，因此我们将其容量设为10。
3. 在进入关键的部分前调用Lock来锁定c.L。
4. 在这里我们检查队列的长度，以确认什么时候需要等待。由于removeFromQueue是异步的，for不满足时才会跳出，而if做不到重复判断，这一点很重要。
5. 调用Wait，这将阻塞main goroutine，直到接受到信号。
6. 这里我们创建一个新的goroutine，它会在1秒后将元素移出队列。
7. 这里我们退出条件的关键部分，因为我们已经成功加入了一个元素。
8. 我们再次进入该并发条件下的关键部分，以修改与并发条件判断直接相关的数据。
9. 在这里，我们移除切片的头部并重新分配给第二个元素，这一步模拟了元素出列。
10. 我们退出操作关键部分，因为我们已经成功移除了一个元素。
11. 这里，我们发出信号，通知处于等待状态的goroutine可以进行下一步了。

这会输出：

```
Adding to queue Adding to queue Removed from queue Adding to queue Removed from queue Adding to queue Removed from queue Adding to queue Removed from queue Adding to queue Removed from queue Adding to queue Removed from queue Adding to queue Removed from queue Adding to queue
```

强烈建议将代码中for换成if感受下，作者考虑的真周全

正如你所看到的，程序成功地将所有10个元素添加到队列中（并且在它有机会在最后两项出队之前退出）。 它也会持续等待，直到至少有一个元素在放入另一个元素之前出列。

在这个例子中，我们使用了一个新的方法，Signal。这是Cond类型提供的两种通知方法之一，用于通知在等待调用上阻塞的goroutines条件已被触发。另一种方法是Broadcast。在内部，运行时维护一个等待信号发送的goroutines的FIFO列表; Signal寻找等待时间最长的goroutine并通知，而Broadcast向所有处在等待状态的goroutine发送信号。Broadcast可以说是两种方法中最有趣的方式，因为它提供了一种同时与多个goroutine进行通信的解决方案。 我们可以通过通道轻松地再现Signal（随后我们会看到这个例子），但是再现对Broadcast重复呼叫的行为将很困难。另外，Cond类型比使用通道更高效。

为了了解Broadcast是如何使用的，假设我们正在创建一个带有按钮的GUI程序，该程序需要注册任意数量的函数，当点击按钮时运行这些函数。可以使用Cond的Brocast来通知所有已注册函数，让我们看看该如何实现：

```
type Button struct {
    //1
    Clicked *sync.Cond
}
button := Button{Clicked: sync.NewCond(&sync.Mutex{})}

subscribe := func(c *sync.Cond, fn func()) { //2
    var tempwg sync.WaitGroup
    tempwg.Add(1)
    go func() {
        tempwg.Done()
        c.L.Lock()
        defer c.L.Unlock()
        c.Wait()
```

```
        fn()
    }()
    tempwg.Wait()
}

var wg sync.WaitGroup //3
wg.Add(3)
subscribe(button.Clicked, func() { //4
    fmt.Println("Maximizing window.")
    wg.Done()
})
subscribe(button.Clicked, func() { //5
    fmt.Println("Displaying annoying dialog box!")
    wg.Done()
})
subscribe(button.Clicked, func() { //6
    fmt.Println("Mouse clicked.")
    wg.Done()
})

button.Clicked.Broadcast() //7

wg.Wait()
```

1. 我们定义一个Button类型，包含了sync.Cond指针类型的Clicked属性，这是goroutine接收通知的关键条件。
2. 这里我们定义了一个较为简单的函数，它允许我们注册函数来处理信号。每个注册的函数都在自己的goroutine上运行，并且在该goroutine不会退出，直到接收到通知。
3. 在这里，我们为按钮点击设置了一个处理程序。 它反过来在Clicked Cond上调用Broad cast以让所有注册函数知道按钮已被点击。
4. 这里我们创建一个WaitGroup。 这只是为了确保我们的程序在写入标准输出之前不会退出。
5. 在这里我们注册一个处理函数，模拟点击时最大化窗口。
6. 在这里我们注册一个处理函数，模拟点击时显示对话框。
7. 接下来，我们模拟按钮被点击。

这会输出：

```
Mouse clicked.
Maximizing window.
Displaying annoying dialog box!
```

可以看到，通过调用Broadcast，三个处理函数都运行了。如果不是wg WaitGroup，我们可以多次调button.Clicked.Broadcast（），并且每次都将运行这三个处理函数。 这是通道难以做到的，也是使用Cond类型的优势之一。

Once

考虑下面这段代码会打印出什么：

```
var count int

increment := func() {
    count++
}

var once sync.Once

var wg sync.WaitGroup

wg.Add(100)
for i := 0; i < 100; i++ {
    go func() {
        defer wg.Done()
        once.Do(increment)
    }()
}

wg.Wait()
fmt.Printf("Count is %d\n", count)
```

你肯定已经注意到了sync.Once类型的变量，没错，这段代码将打印以下内容：

```
Count is 1
```

顾名思义，sync.Once确保了即使在不同的goroutine上，调用Do传入的函数只执行一次。

看起来将多次调用一个函数但执行一次的能力封装并放入标准库是一件奇怪的事情，但事实证明，对这种模式的需求相当频繁。为了好玩，让我们来检查Go的标准库，看看Go本身使用这个原语的频率。 这是一个将执行搜索的grep命令：

```
grep -ir sync.Once $(go env GOROOT)/src |wc -l
```

这会输出：

```
70
```

关于利用sync.Once有几点需要注意。我们来看看另一个例子。 你认为它会打印什么？

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

var once sync.Once
once.Do(increment)
once.Do(decrement)

fmt.Printf("Count: %d\n", count)
```

这会输出：

```
Count: 1
```

输出显示1而不是0令人惊讶吗？ 这是因为sync.Once只计算Do被调用的次数，而不是调用传入Do的唯一函数的次数。 通过这种方式，sync.Once的副本与被用于调用的函数紧密耦合；我们再次看到sync包内如何在一个紧密的范围内发挥最佳效果。 我建议你通过在一个小的词法块中包装sync.Once的来形式化这种耦合：一个小型函数，或者通过包装在一个类型中。 那么下面这个例子呢？ 你认为会发生什么？

```
var onceA, onceB sync.Once
var initB func()
initA := func() { onceB.Do(initB) }
initB = func() { onceA.Do(initA) } // 1
onceA.Do(initA) // 2
```

1. 这里的调用无法执行，直到2被返回。

这段程序会发生死锁，因为在1处对Do的调用不会执行直到2执行完毕，而2处无法结束执行——这是一个标准的死锁示例。这可能有点反直觉，看起来好像我们正在使用sync.Once来防止多个初始化。有时候程序出现死锁正是由于逻辑中出现了循环引用。

Pool

Pool是对象池模式的并发安全实现。关于对象池模式的完整解释最好留给有关设计模式的文献(如 Head First Design Patterns)。不过既然Pool出现在了标准库中，就让我们简要讨论下为什么你可能有兴趣使用它。

在较高的层次上，池模式是一种创建和提供固定数量可用对象的方式。它通常用于约束创建资源昂贵的事物（例如数据库连接）。Go的sync.Pool可以被多个例程安全地使用。

Pool的主要接口是它的Get方法。 被调用时，Get将首先检查池中是否有可用实例返回给调用者，如果没有，则创建一个新成员变量。使用完成后，调用者调用Put将正在使用的实例放回池中供其他进程使用。 这里有一个简单的例子来演示：

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

myPool.Get()            //1
instance := myPool.Get() //1
myPool.Put(instance)    //2
myPool.Get()            //3
```

1. 这里我们调用Get方法，将调用在池中定义的New函数，因为实例尚未实例化。
2. 在这里，我们将先前检索到的实例放回池中。 这时实例的可用数量为1个。
3. 执行此调用时，我们将重新使用先前分配的实例。New函数不会被调用。

我们可以看到，这回调用2次New函数：

```
Creating new instance.
Creating new instance.
```

那么为什么要使用一个池，而不是实例化对象呢？ Go有一个垃圾收集器，所以实例化的对象将被自动清理。 重点是什么？ 考虑这个例子：

```
var numCalcsCreated int
calcPool := &sync.Pool{
    New: func() interface{} {
        numCalcsCreated += 1
        mem := make([]byte, 1024)
        return &mem // 1
    },
}

// 将池扩充到4KB
calcPool.Put(calcPool.New())
calcPool.Put(calcPool.New())
calcPool.Put(calcPool.New())
calcPool.Put(calcPool.New())
const numWorkers = 1024 * 1024
var wg sync.WaitGroup
wg.Add(numWorkers)

for i := numWorkers; i > 0; i-- {
    go func() {
        defer wg.Done()
```

```
        mem := calcPool.Get().(*[]byte) // 2
        defer calcPool.Put(mem)

    }()
}

// 假设内存中执行了一些快速的操作

wg.Wait()
fmt.Printf("%d calculators were created.", numCalcsCreated)
```

1. 注意，我们存储了字节切片的指针。
2. 这里我们断言此类型是一个指向字节切片的指针。

这会输出：

```
8 calculators were created.
```

如果我没有使用sync.Pool运行此示例，那么结果将是非确定性的，但在最坏的情况下，我可能需要分配千兆字节的内存。但正如你从输出中看到的那样，我只分配了4 KB 。

Pool有用的另一种常见情况是预热分配对象的缓存，用于必须尽快运行的操作。 在这种情况下，我们不是通过限制创建对象的数量来保护主机的内存，而是通过预先加载获取对另一个对象的引用来减少消费者的时间消耗。在编写高吞吐量网络服务器时，这是非常常见的。我们来看看这种情况。

首先，我们来创建一个模拟创建服务连接的函数。 我们会让这个连接花费很长时间：

```
func connectToService() interface{} {
    time.Sleep(1 * time.Second)
    return struct{}{}
}
```

接下来，让我们看看如果对于每个请求都开启一个新的服务连接，网络服务的性能如何。我们将编写一个网络处理程序，为了简化基准测试，我们一次只允许一个连接：

```
func startNetworkDaemon() *sync.WaitGroup {
    var wg sync.WaitGroup
    wg.Add(1)

    go func() {
        server, err := net.Listen("tcp", "localhost:8080")
        if err != nil {
            log.Fatalf("cannot listen: %v", err)
        }
        defer server.Close()
        wg.Done()
        for {
            conn, err := server.Accept()
            if err != nil {
                log.Printf("cannot accept connection: %v", err)
                continue
            }

            connectToService()
            fmt.Fprintln(conn, "")
            conn.Close()
```

```
        }
    }()

    return &wg
}
```

现在我们可以着手进行基准测试了：

```
func init() {
    daemonStarted := startNetworkDaemon()
    daemonStarted.Wait()
}

func BenchmarkNetworkRequest(b *testing.B) {
    for i := 0; i < b.N; i++ {
        conn, err := net.Dial("tcp", "localhost:8080")
        if err != nil {
            b.Fatalf("cannot dial host: %v", err)
        }
        if _, err := ioutil.ReadAll(conn); err != nil {
            b.Fatalf("cannot read: %v", err)
        }
        conn.Close()
    }
}
```

在命令行执行：

```
cd src/gos-concurrency-building-blocks/the-sync-package/pool/ && \
go test -benchtime=10s -bench=.
```

这会输出：

BenchmarkNetworkRequest-8	10	1000385643 ns/op
PASS ok	command-line-arguments	11.008s

就性能而言这看起来挺合理。让我们加上sync.Pool再试试：

```
func warmServiceConnCache() *sync.Pool {
    p := &sync.Pool{
        New: connectToService,
    }
    for i := 0; i < 10; i++ {
        p.Put(p.New())
    }
    return p
}

func startNetworkDaemon() *sync.WaitGroup {
    var wg sync.WaitGroup
    wg.Add(1)
```

```
    go func() {
        connPool := warmServiceConnCache()

        server, err := net.Listen("tcp", "localhost:8080")
        if err != nil {
            log.Fatalf("cannot listen: %v", err)
        }
        defer server.Close()
        wg.Done()
        for {
            conn, err := server.Accept()
            if err != nil {
                log.Printf("cannot accept connection: %v", err)
                continue

            }
            svcConn := connPool.Get()
            fmt.Fprintln(conn, "")
            connPool.Put(svcConn)
            conn.Close()
        }
    }()

    return &wg
}
```

同样执行基准测试，

```
cd src/gos-concurrency-building-blocks/the-sync-package/pool && \
go test -benchtime=10s -bench=.
```

我们可以看到：

BenchmarkNetworkRequest-8	5000	2904307 ns/op
PASS ok	command-line-arguments	32.647s

整整快了三个数量级！你可以看到我们是如何利用这种模式如何大大缩短响应时间的。

正如这个例子所展现的，池模式非常适合于这种需要并发进程，或者构建这些对象可能会对内存产生负面影响的应用程序。

但是，在确定是否应该使用池时有一点需要注意：如果使用池子里东西在内存上不是大致均匀的，则会花更多时间将从池中检索，这比首先实例化它要耗费更多的资源。例如，你的程序需要随机和可变长度的切片，在这种情况下Pool不会为你提供太多的帮助。

因此，在使用Pool时，请记住以下几点：

- 实例化sync.Pool时，给它一个新元素，该元素应该是线程安全的。
- 当你从Get获得一个实例时，不要假设你接收到的对象状态。
- 当你从池中取得实例时，请务必不要忘记调用Put。否则池的优越性就体现不出来了。这通常用defer来执行延迟操作。
- 池中的元素必须大致上是均匀的。

Channels

Channel，即通道，衍生自Charles Antony Richard Hoare的CSP并发模型，是Go的并发原语，在Go语言中具有极其重要的地位。虽然它可用于同步内存的访问，但更适合用于goroutine之间传递信息。就像我们在之前“Go的并发哲学”章节所提到的那样，通道在任何规模的程序编码中都非常有用，因为它足够灵活，能够以各种方式组合在一起。我们在随后的“select语句”章节会进一步探讨其是如何构造的。

可以把通道想象为一条河流，通道作为信息流的载体；数据可以沿着通道被传递，然后从下游被取出。基于这个原因，我通常用单词“Stream”为我的通道变量命名。在使用通道时，你把一个值传递给chan类型的变量，然后在程序的其他位置再把这个值从通道中读取出来。该值被传递的入口和出口不需要知道彼此的存在，你只需使用通道的引用进行操作就好。

建立一个通道是非常简单的。下面这个例子展现了如何对通道声明和如何实例化。与Go中的其他类型一样，你可以使用 := 来简化通道的创建。不过在程序中通常会先声明通道，所以了解该如何将二者分成单独的步骤是很有用的：

```
var dataStream chan interface{} // 1
dataStream = make(chan interface{})  // 2
```

1.这里声明了一个通道。我们说该通道的“类型”是interface{}的。
2.这里我们使用内置函数make实例化通道。

这个示例定义了一个名为dataStream的通道，在该通道上可以写入或读取任意类型的值（因为我们使用了空的接口）。通道也可以声明为仅支持单向数据流——即你可以定义仅支持发送或仅支持接收数据的通道。我将在本节的末尾解释单向数据流的重要性。

要声明一个只能被读取的单向通道，你可以简单的使用 <- 符号，如下所示：

```
var dataStream <-chan interface{}
dataStream := make(<-chan interface{})
```

与之相对应，要声明一个只能被发送的单向通道，把 <-放在 chan关键字的右侧即可：

```
var dataStream chan<- interface{}
dataStream := make(chan<- interface{})
```

你不会经常看到实例化的单向通道，但你会经常看到它们被用作函数参数和返回类型，这是非常有用的，因为Go可以在需要时将双向通道隐式转换为单向通道，比如这样：

```
var receiveChan <-chan interface{}
var sendChan chan<- interface{}
dataStream := make(chan interface{})

// 这样做是有效的
receiveChan = dataStream
sendChan = dataStream
```

要注意通道是有“类型”的。 在这个例子中，我们创建了一个接口“类型”的chan，这意味着我们可以在其上放置任何类型的数据，但是我们也可以给它一个更严格的类型来约束它可以传递的数据类型。 这是一个整数通道的例子：

```
intStream := make(chan int)
```

为了操作通道，我们再一次使用 <- 符号。我们看一个实际的例子：

```
stringStream := make(chan string)
go func() {
    stringStream <- "Hello channels!" //1
}()
fmt.Println(<-stringStream) //2
```

1. 这里我们将字符串放入通道stringStream。
2. 这里我们从通道中取出字符串并打印到标准输出流。

这会输出：

```
Hello channels!
```

很简单，是吧。你所需要做的只是建立一个通道变量，然后将数据传递给它并从中读取数据。但是，尝试将值写入只读通道或从只写通道读取值都是错误的。如果我们尝试编译下面的例子，Go的编译器会报错：

```
writeStream := make(chan<- interface{})
readStream := make(<-chan interface{})

<-writeStream
readStream <- struct{}{}
```

这会输出：

```
invalid operation: <-writeStream (receive from send-only type chan<- interface {})
invalid operation: readStream <- struct {} literal (send to receive-only type <-chan interface {})
```

这是Go类型系统的一部分，即使在处理并发原语时也为我们保证类型安全。稍后我们会看到，这对构建易于推理的可组合逻辑程序提供了强大的保证。

回想一下，在之前我们强调过，仅简单的定义一个goroutine并不能保证它在main goroutine退出之前运行。为此我们介绍了对sync包的各种使用案例。那么在使用通道的情况下该如何呢？看下面这个例子：

该示例之所以产生这样的结果，是因为在Go中，通道是包含有阻塞机制的。这意味着试图写入已满的通道的任何goroutine都会等待直到通道被清空，并且任何尝试从空闲通道读取的goroutine都将等待，直到至少有一个元素被放置 。在这个例子中，我们的fmt.Println包含一个对通道stringStream的读取，并且将阻塞在那里，直到通道上被放置一个值。同样，匿名goroutine试图在stringStream上放置一个字符串，然后阻塞住等待被读取，所以goroutine在写入成功之前不会退出。因此，main goroutine和匿名的goroutine发生阻塞是毫无疑问的。

```
stringStream := make(chan string)
go func() {
    if 0 != 1 { //1
        return
    }
    stringStream <- "Hello channels!"
}()

fmt.Println(<-stringStream)
```

1. 在这里 我们确保通道stringStream永远不会获得值。

这会产生错误：

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

```
goroutine 1 [chan receive]:
main.main()

/tmp/babel-23079IVB/go-src-230795Jc.go:15 +0x97
exit status 2
```

main goroutine等待着stringSteam通道被放上一个值，而且由于我们的if条件，导致这不会发生。当匿名goroutine退出时，Go发现并报告死锁。在本节后面，我将解释如何构建我们的程序，以防止这种死锁。与此同时，让我们回到从通道读取数据的讨论。

< - 运算符的接收形式也可以选择返回两个值，如下所示：

```
stringStream := make(chan string)
go func() {
    stringStream <- "Hello channels!"
}()
salutation, ok := <-stringStream //1
fmt.Printf("(%v): %v", ok, salutation)
```

1. 我们在这里接收一个字符串salutation和一个布尔值ok。

这会输出：

```
(true): Hello channels!
```

有意思吧。那么布尔值代表什么呢？这个值是读取操作的一个标识，用于指示读取的通道是由过程中其他位置的写入生成的值，还是由已关闭通道生成的默认值。 等一下; 一个已关闭的通道，那是什么？

在程序中，能够指示还有没有更多值将通过通道发送是非常有用的。 这有助于下游流程知道何时移动，退出，或在新的通道上重新开启通信等。我们可以通过为每种类型提供特殊的标识符来完成此操作，但这会开发人员的工作产生巨大的重复性，如果能够内置将产生极大的便利，因此关闭通道就像是一个万能的哨兵，它说：“嘿，上游不会写更多的数据啦，做你想做的事吧。”要关闭频道，我们使用close关键字，就像这样：

```
valueStream := make(chan interface{})
close(valueStream)
```

有趣的是，我们也可以从已关闭的通道读取。 看这个例子：

```
intStream := make(chan int)
close(intStream)
integer, ok := <- intStream // 1
fmt.Printf("(%v): %v", ok, integer)
```

1. 这里我们从已关闭的通道读取。

这会输出：

```
(false): 0
```

注意我们在关闭通道前并没有把任何值放入通道。即便如此我们依然可以执行读取操作，而且尽管通道处在关闭状态，我们依然可以无限期地在此通道上执行读取操作。这是为了支持单个通道的上游写入器可以被多个下游读取器读取(在第四章我们会看到这是一种常见的情况)。第二个返回值——即布尔值ok——表明收到的值是int的零值，而非被放入流中传递过来。

这为我们开辟了一些新的模式。首先是通道的range操作。与for语句一起使用的range关键字支持将通道作为参数，并且在通道关闭时自动结束循环。这允许对通道上的值进行简洁的迭代。 我们来看一个例子：

```
intStream := make(chan int)
go func() {
    defer close(intStream) // 1
    for i := 1; i <= 5; i++ {
        intStream <- i
    }
}()

for integer := range intStream { // 2
    fmt.Printf("%v ", integer)
}
```

1. 在这里我们在通道退出之前保证正常关闭。这是一种很常见的Go惯用法。
2. 这里对intStream进行迭代。

正如你所看到的，所有的值被打印后程序退出：

```
1 2 3 4 5
```

注意循环退出并没有设置条件，并且range也不返回第二个布尔值。对通道进行关闭的处理被隐藏了起来，以此保证循环的简洁。

关闭某个通道同样可以被作为向多个goroutine同时发生消息的方式之一。如果你有多个goroutine在单个通道上等待，你可以简单的关闭通道，而不是循环解除每一个goroutine的阻塞。由于一个已关闭的通道可以被无限次的读取，因此其中有多少goroutine在阻塞状态并不重要，关闭通道(以解除所有阻塞)消耗的资源又少执行的速度又快。以下是一次解除多个goroutine的示例：

```
begin := make(chan interface{})
var wg sync.WaitGroup
for i := 0; i < 5; i++ {
    wg.Add(1)
    go func(i int) {
        defer wg.Done()
        <-begin //1
        fmt.Printf("%v has begun\n", i)
    }(i)
}

fmt.Println("Unblocking goroutines...")
close(begin) //2
wg.Wait()
```

1. 这里对begin通道进行读取，由于通道中没有任何值，会产生阻塞。
2. 这里我们关闭通道，这样所有goroutine的阻塞会被解除。

你可以看到，在我们关闭begin 通道之前，没有任何一个goroutine开始运行：

```
**Unblocking goroutines...
4 has begun
2 has begun
3 has begun
0 has begun
1 has begun**
```

回想一下在“sync包”中我们讨论过sync.Cond实现类似功能的例子，你当然可以使用Single或者Brocast来做，不过通道是可组合的，所以这也是我最喜欢的同时解除多个goroutine阻塞的方法。

接下来，我们来讨论“缓冲通道”，这种通道实在实例化时候提供可携带元素的容量。这意味着，即使没有对通道进行读取操作，goroutine仍然可以执行n次写入，这里的n即缓冲通道的容量。下面是一个实例化的例子：

```
var dataStream chan interface{}
dataStream = make(chan interface{}, 4)
```

1. 这里我们创建一个容量为4的缓冲通道。 这意味着我们可以将4个元素放在通道上，而不管它是否被读取（在数量达到上限之前，写入行为都不会发生阻塞）。

我们再一次把初始化分为了两行，这样你可以清楚的发现，一个缓冲通道和一个非缓冲通道
在声明上上没有区别的(区别只在实例化部分)。有趣的地方在于，我们可以在实例化的位置对通道是否是缓冲的进行控制。这表明，通道的建立应该与goroutine紧密结合，这样我们可以极大的提高代码的可读性。

无缓冲的通道也可以按缓冲通道定义：无缓冲的通道可以视作一个容量为0的缓冲通道。就像下面这样：

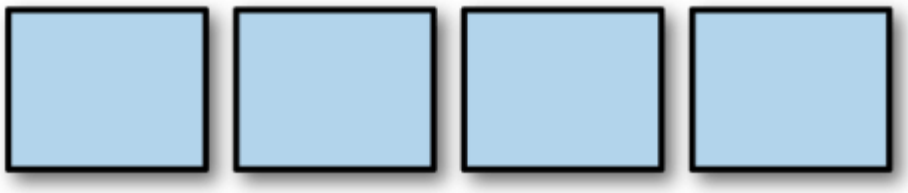
```
a := make(chan int)
b := make(chan int, 0)
```

这两个通道都是int“类型”的。请记住我们在讨论“阻塞”时所代表的含义，我们说向一个已满的通道写入，会出现阻塞，从一个已空的通道读取，也会出现阻塞。这里的“满”和“空”是针对容量或缓冲区大小而言的。无缓冲的通道所拥有的容量为0，所以任何写入行为之后它都会是满的。一个容量为4的缓冲通道在4次写入后会是满的，并且会在第5次写入时出现阻塞，因为它已经没有其他位置可以放置第5个元素，这时它表现出的行为与无缓冲通道一样：由此可见，缓冲通道和无缓冲通道的区别在于，通道为空和满的前提条件是不同的。通过这种方式，缓冲通道可以在内存中构建用于并发进程通信的FIFO队列。

为了帮助理解这一点，我们来举例说明缓冲通道容量为4的示例中发生了什么。 首先，让我们初始化它：

```
c := make(chan rune, 4)
```

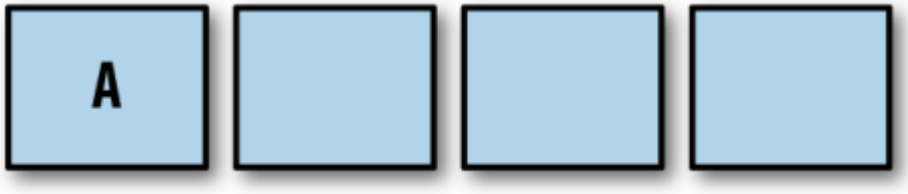
从逻辑上讲，这会创建一个带有四个空位的缓冲区的通道：



现在，让我们向通道写入：

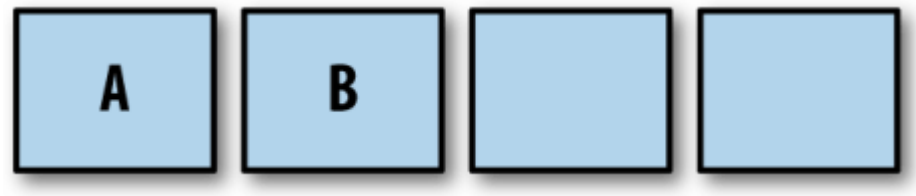
```
c <- 'A'
```

当这个通道没有被读取时，A字符将被放置在通道缓冲区的第一个空位中，像这样：

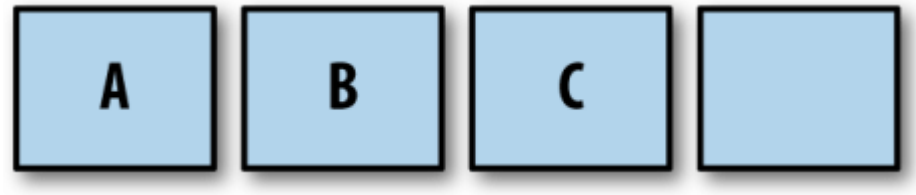


随后每次写入缓冲通道(同样假设没有被读取)，将填充缓冲通道中的剩余空位，像这样：

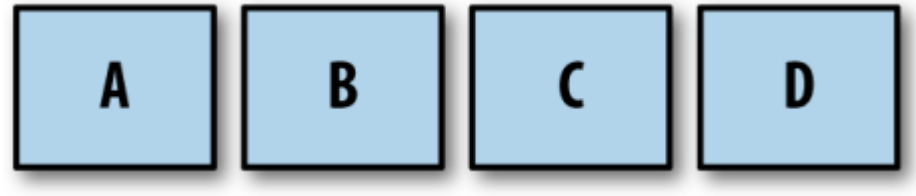
```
c <- 'B'
```




```
c <- 'C'
```

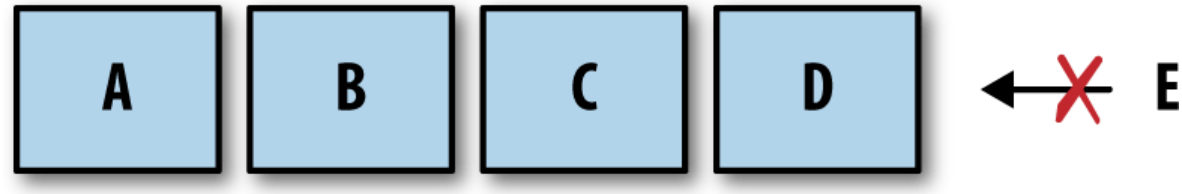



```
c <- 'D'
```



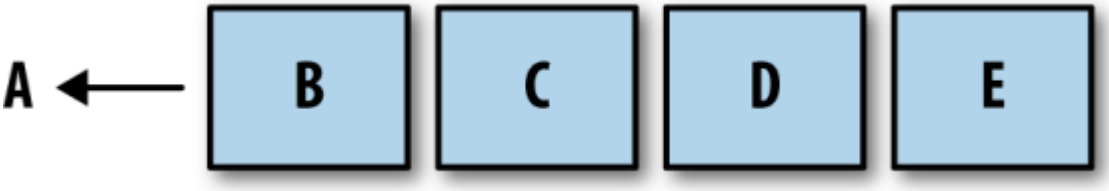
经过四次写入，我们的缓冲通道已经装满了4个元素。如果我们再向通道中进行写入的话：

```
c <- 'E'
```



当前的goroutine会表现为阻塞！并且goroutine将一直保持阻塞状态，直到由其他的goroutine执行读取操作在缓冲区中空出了位置。 让我们看看是什么样子的

```
<- c
```



正如你所看到的那样，读取时会接收到放在通道上的第一个字符A，被阻塞的写入阻塞解除，E被放置在缓冲区的末尾。

如果，如果缓冲通道为空且有接收器读取，则缓冲器将被绕过，并且该值将直接从发送器传递到接收器（存疑）。实际上，这是透明地发生的，但值得了解。

缓冲通道在某些情况下很有用，但你应该小心使用。正如我们将在下一章中看到的那样，缓冲通道很容易成为不成熟的优化，并且通过使用它们死锁会变得更为隐蔽。我猜你宁愿在第一次编写代码时找到一个死锁，而不是在半夜系统停机的时候。

让我们来看看另一个更完整的代码示例，以便更好地了解缓冲通道的工作方式：

```
var stdoutBuff bytes.Buffer          //1
defer stdoutBuff.WriteTo(os.Stdout) //2

intStream := make(chan int, 4) //3
go func() {
    defer close(intStream)
    defer fmt.Fprintln(&stdoutBuff, "Producer Done.")
    for i := 0; i < 5; i++ {
        fmt.Fprintf(&stdoutBuff, "Sending: %d\n", i)
        intStream <- i
    }
}()

for integer := range intStream {
    fmt.Fprintf(&stdoutBuff, "Received %v.\n", integer)
}
```

1. 这里我们创建一个内存缓冲区来帮助缓解输出的不确定性。 它不会给带来我们任何保证，但比直接写stdout要快一些。
2. 在这里，我们确保在进程退出之前将缓冲区内容写入标准输出。
3. 这里我们创建一个容量为4的缓冲通道。

在这个例子中，写入stdout的顺序是不确定的，但你仍然可以大致了解匿名goroutine是如何工作的。 如果你检查输出结果，可以看到我们的匿名goroutine能够将所有五个结果放在intStream中，并在主要goroutine将一个结果关闭之前退出。

```
Sending: 0
Sending: 1
Sending: 2
Sending: 3
Sending: 4
Producer Done.
Received 0
Received 1
Received 2
Received 3
Received 4
```

这是一个在正确条件下可以使用的优化示例：如果写入通道的goroutine明确知道将会写入多少信息，则创建相对应的缓冲通道容量会很有用，就可以尽可能快地进行读取。 当然，这样做是有限制的，我们将在下一章中介绍。

我们已经讨论了无缓冲的频道，缓冲频道，双向频道和单向频道。目前还没有讨论到的还有通道的默认值：nil。程序是如何处理处理nil通道的呢？首先，让我们试着从一个nil通道中读取：

```
var dataStream chan interface{}
<-dataStream
```

这会输出：

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

```
goroutine 1 [chan receive (nil chan)]:
main.main()
    F:/code/gospcace/src/myConcurrency/11introduction/101/main.go:6 +0x30
```

死锁出现了。这说明从一个nil通道进行读取会阻塞程序(注意，这段代码的前提是在main函数中执行，所以会导致死锁。如果是运行在单个gouroutine中，那么就不会是死锁而是阻塞)。让我们再试试写入：

```
var dataStream chan interface{}
dataStream <- struct{}{}
```

这会输出：

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

goroutine 1 [chan send (nil chan)]:
main.main()
    F:/code/gospcace/src/myConcurrency/11introduction/101/main.go:6 +0x53
```

看来对一个nil通道进行写入操作同样会阻塞。
我们再试试关闭操作：

```
var dataStream chan interface{}
close(dataStream)
```

这会输出：

```
panic: close of nil channel

goroutine 1 [running]:
main.main()
    F:/code/gospcace/src/myConcurrency/11introduction/101/main.go:6 +0x31
```

程序挂掉了，这也许是最符合你预期的结果。无论如何，请务必确保你的通道在工作前已经完成了初始化。

我们已经了解了很多与通道互动的规则。现在你已经了解了在通道上执行操作的方式和为什么这样做的原因。 下表列举了通道上的操作对应状态的通道会发生什么。

注意：表中的用词都很简短，为了减少不必要的歧义或混乱，并未对该表进行不必要的翻译，此外，正如上面例子所展现的，该表的操作结果默认都是在main函数下操作。请以批判的眼光审视下表。

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

如果我们查看该表，可以察觉到在操作中可能产生问题的地方。这里有三个可能导致阻塞的操作，以及三个可能导致程序恐慌的操作。乍看之下，通道的使用上限制很多，但在检查了这个限制产生的动机并熟悉了通道的使用后，它变得不那么可怕并开始具有很大意义。让我们讨论如何组织不同类型的通道来构筑稳健的程序。

我们应该做的第一件事是将通道置于正确的环境中，即分配通道所有权。我将所有权定义为goroutine的实例化，写入和关闭。就像在那些没有垃圾回收的语言中使用内存一样，重要的是要明确哪个goroutine拥有该通道，以便从逻辑上推理我们的程序。单向通道声明是一种工具，它可以让我们区分哪些gouroutine拥有通道，哪些goroutine仅使用通道：通道所有者对通道具有写入访问权（chan或chan<- ），而通道使用者仅具有读取权（<-chan）。一旦我们对通道权责区分，上表的结果自然就会出现。我们可以开始对拥有通道和不拥有通道的goroutine赋予不同的责任并给予对应的检查(以增强程序和逻辑的健壮性)。

让我们从通道的所有者说起。当一个goroutine拥有一个通道时应该：

1. 初始化该通道。
2. 执行写入操作，或将所有权交给另一个goroutine。
3. 关闭该通道。
4. 将此前列出的三件事情封装在一个列表中，并通过订阅通道将其公开。

通过将这些责任分配给通道所有者，会发生一些事情：

- 因为我们是初始化频道的人，所以我们要了解写入nil通道会带来死锁的风险。
- 因为我们是初始化频道的人，所以我们要了解关闭ni通道会带来恐慌的风险。
- 因为我们是决定频道何时关闭的人，所以我们要了解写入已关闭的通道会带来恐慌的风险。
- 因为我们是决定何时关闭频道的人，所以我们要了解多次关闭通道会带来恐慌的风险。
- 我们在编译时使用类型检查器来防止对通道进行不正确的写入。

现在我们来看看读取时可能发生的那些阻塞操作。 作为一个通道的消费者，我只需要担心两件事情：

- 通道什么时候会被关闭。
- 处理基于任何原因出现的阻塞。

解决第一个问题，通过检查读取操作的第二个返回值就可以。第二点很难，因为它取决于你的算法(和业务逻辑)：你可能想要超时，当获得通知时你可能想停止读取操作，或者你可能只是满足于在整个生命周期中产生阻塞。 重要的是，作为一个消费者，你应该明确这样一个事实，即读取操作可以并且必将产生阻塞。我们将在下一章中探讨如何实现select语句解决这个棘手的问题。

现在，让我们用一个例子来总结以上的思考结果。我们建立一个goroutine，它拥有一个通道，一个消费者，它会处理阻塞问题：

```
chanOwner := func() <-chan int {

    resultStream := make(chan int, 5)//1
    go func() {//2
        defer close(resultStream)//3
        for i := 0; i <= 5; i++ {
            resultStream <- i
        }
    }()
    return resultStream//4

}

resultStream := chanOwner()
for result := range resultStream {//5
    fmt.Printf("Received: %d\n", result)
}
fmt.Println("Done receiving!")
```

1. 这里我们实例化一个缓冲通道。 由于我们知道我们将产生六个结果，因此我们创建了五个缓冲通道，以便该goroutine可以尽快完成操作。
2. 在这里，我们启动一个匿名的goroutine，它在resultStream上执行写操作。 请注意，我们是如果创建goroutines的，它现在被封装在函数中。
3. 这里我们确保resultStream在操作完成后关闭。作为通道所有者，这是我们的责任。
4. 我们在这里返回通道。由于返回值被声明为只读通道，resultStream将隐式转换为只读的。

这里我们消费了resultStream。 作为消费者，我们只关心阻塞和通道的关闭。
这会输出：

```
Received: 0
Received: 1
Received: 2
Received: 3
Received: 4
Received: 5
Done receiving!
```

注意resultStream通道的生命周期如何封装在chanOwner函数中。很明显，写入不会发生在nil或已关闭的频道上，并且关闭总是会发生一次。这消除了我们之前提到的部分风险。我强烈建议你在自己的程序中尽可能做到保持通道覆盖范围最小，以便这些事情保持明显。如果你将一个通道作为一个结构体的成员变量，并且有很多方法，它很快就会把你自己给绕进去（**虽然很多库和书中都这么干，但只有这本书的作者将这一点给明确提出来了**）。

消费者功能只能读取通道，因此只需知道应该如何处理阻塞读取和通道关闭。 在这个小例子中，我们采取了这样的方式：在通道关闭之前阻塞程序是完全没问题的。

如果你设计自己的代码时来遵循这个原则，那么对你的系统进行推理就会容易得多，而且它很可能会像你期望的那样执行。我不能保证你永远不会引入阻塞或恐慌，但是当你这样遇到这样的情况时，我认为你会发现你的通道所有权范围要么太大，要么

所有权不清晰。

通道是首先吸引我使用Go的原因之一。 结合goroutines和闭包的简约性，编写干净、正确的并发代码是比较容易的。在很多方面，通道是将goroutine绑在一起的胶水。 本节为你概述了什么是通道以及如何使用它们。当我们开始编写通道以形成更高阶的并发设计模式时，真正的乐趣就开始了。我们将在下一章中体会到这一点。

select语句

通道是将goroutine的粘合剂，select语句是通道的粘合剂。后者让我们能够在项目中组合通道以形成更大的抽象来解决实际中遇到的问题。凸显select语句在Go并发上的地位绝对不是言过其实。你可以在单个函数或类型定义中找到将本地通道绑定在一起的select语句，也可以在全局范围找到连接系统级别两个或多个组件的使用范例。除了连接组件外，在程序中的关键部分，select语句还可以帮助你安全地将通道与业务层面的概念（如取消，超时，等待和默认值）结合在一起。

既然select语句在Go中占有如此重要的地位——专门用于处理通道，那么你认为程序的组件应该如何协调？ 我们将在第五章专门研究这个问题（提示：更喜欢使用频道）。

那么这些强大的select语句是什么？ 我们如何使用它们，它们是如何工作的？ 我们先从一个简单的示例开始：

```
var c1, c2 <-chan interface{}
var c3 chan<- interface{}
select {
case <-c1:
    // Do something
case <-c2:
    // Do something
case c3 <- struct{}{}:
    // Do something
}
```

看着跟switch有点像，是吧。跟switch相同的是，select代码块也包含一系列case分支。跟switch不同的是，case分支不会被顺序测试，如果没有任何分支的条件可供满足，select会一直等待直到某个case语句完成。

所有通道的读取和写入都被同时考虑，以查看它们中的任何一个是否准备好： 如果没有任何通道准备就绪，则整个select语句将会阻塞。当一个通道准备好时，该操作将继续，并执行相应的语句。 我们来看一个简单的例子：

```
start := time.Now()
c := make(chan interface{})
go func() {
    time.Sleep(5 * time.Second)
    close(c) // 1
}()

fmt.Println("Blocking on read...")
select {
case <-c: // 2
    fmt.Printf("Unblocked %v later.\n", time.Since(start))
}
```

1. 这里我们在5秒后关闭通道。
2. 在这里我们尝试读取通道。注意，尽管我们可以不使用select语句而直接使用<-c，但我们的目的是为了展示select语句。

这会输出：

```
Blocking on read...
Unblocked 5s later.
```

如你所见，在5秒的阻塞后我们进入select代码块。这是一种简单而有效的方式来实现阻塞等待，但如果反思一下，我们可以发现一些问题：

- 当多个通道需要读取时会发生什么？
- 如果所有通道都尚未初始化完成，该怎么办？

- 如果我们想做点什么，但当前通道还没准备好呢？

第一个问题很有趣，让我们试试看会发生什么：

```
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)
```

这会输出：

```
c1Count: 505
c2Count: 496
```

c1和c2被关闭后，可以读到其承载的类型的零值。你可以看到，在1001次循环中，大约有一半的时间从c1读取，有一半是从c2读取的。这似乎很有趣，也许有点太巧合。 事实上，是由Go的运行时导致的。Go运行时对一组case语句执行伪随机统一选择。这意味着在同样的条件下，每个case被选中的机会几乎是一样的。

乍一看这似乎并不重要，但其背后的理由是令人难以置信的有意思。我们先陈述一个事实：Go运行时无法知道你的select语句的意图；也就是说，它不能推断出你的问题所在，或者你为什么将一组通道放在一个select语句中。正因为如此，Go运行时所能做的最好的事情就是在任何情况下运行良好。一个好的方法是在你的程序中中引入一个随机变量——以决定选择哪个case执行。通过加权平均使用每个通道的机会，使得所有使用select语句的Go程序表现良好。

第二个问题呢？如果所有通道都尚未初始化完成会发生什么？如果所有的通道都处在阻塞状态，你无法进行处理，但你又不能就这样持续阻塞下去，你可能希望程序能够执行超时。Go的time包提供了一个很好的方式来完成这个功能，这些功能完全符合选择语句的范式。 这里有一个例子：

```
var c <-chan int
select {
case <-c: //1
case <-time.After(1 * time.Second):
    fmt.Println("Timed out.")
}
```

1. 这个case分支会永久阻塞，因为我们从一个nil通道读取。

这会输出：

```
Timed out.
```

time.After接收一个类型为time.After的参数，并返回一个通道，该通道将在你提供该通道的持续时间后发送当前时间。这提供了在选择语句中超时的简洁方法。我们将在第4章中重新讨论这种模式，并讨论针对此问题的更强大的解决方案。

我们还有最后的一个问题：如果我们想做点什么，但当前通道还没准备好呢？select语句中允许我们添加default条件，以便你在所有分支都不符合执行条件的时候执行。这里有个例子：

```
start := time.Now()
var c1, c2 <-chan int
select {
case <-c1:
case <-c2:
default:
    fmt.Printf("In default after %v\n\n", time.Since(start))
}
```

这会输出：

```
In default after 1.421µs
```

你可以看到它几乎是瞬间运行默认语句。这允许你在不阻塞的情况下退出选择块。 通常你会看到for-select循环结合使用。这使得goroutine可以在等待另一个goroutine报告结果的同时取得进展。 这是一个例子：

```
done := make(chan interface{})
go func() {
    time.Sleep(5 * time.Second)
    close(done)
}()

workCounter := 0
loop:
for {
    select {
    case <-done:
        break loop
    default:
    }

    // Simulate work
    workCounter++
    time.Sleep(1 * time.Second)
}

fmt.Printf("Achieved %v cycles of work before signalled to stop.\n", workCounter)
```

这会输出：

```
Achieved 5 cycles of work before signalled to stop.
```

在这个例子中，我们有一个循环正在做某种工作，偶尔检查它是否应该停止。

最后，空选择语句有一个特殊情况：select语句没有case子句。 这些看起来像这样：

```
select {}
```

这条语句将永久阻塞。

在第6章中，我们将深入研究select语句的工作原理。 从更高层面来看，它可以帮助你安全高效地将各种概念和子系统组合在一起。

GOMAXPROCS

在runtime包中，有一个方法是GOMAXPROCS。在我看来，这个函数名有点误导人：人们经常认为这个函数与主机上逻辑处理器的数量有关， 但是这个函数真正控制着将要托管所谓的“工作队列”的操作系统线程的数量。在第6章我们会讨论关于它的更多详细信息。

在Go 1.5之前，GOMAXPROCS总是设置为1，通常你会在大多数Go程序中找到这个片段：

```
runtime.GOMAXPROCS(runtime.NumCPU())
```

几乎所有的开发人员都希望利用其进程能够使用计算机上的所有内核。因此，在随后的Go版本中，它被自动设置为主机上的逻辑CPU数量。

那么，如果你想调整这个值呢？ 大多数时候你尽量别这么想。Go的调度算法在大多数情况下足够好，即增加或减少工作队列和线程的数量可能会造成更多的伤害，但仍然有些情况下可能会更改此值。

例如，我曾经参与过一个项目，该项目有一个受竞争条件困扰的测试套件。事实上，该团队有一些测试包有时会失败。我们运行测试的基础架构只有四个逻辑CPU，因此在任何一个时间点，我们都有四个goroutines同时执行。 通过增加GOMAXPROCS超过我们所拥有的逻辑CPU数量，我们能够更频繁地触发竞态条件，从而更快地纠正它们。

有人可能会通过实验发现，他们的程序在一定数量的工作队列和线程下运行得更好，但我敦促谨慎行事。如果你通过调整CPU来压缩性能，请务必在每次提交后，使用不同硬件以及使用不同版本的Go时执行此操作。调整这个值是以更高的抽象和长期性能稳定性的降低为代价的。

结论

在本章中，我们介绍了Go提供的所有基本并发基元。 如果你已阅读并理解了它们，恭喜！你将编写出高性能，可读和逻辑正确的程序。 你知道什么时候使用sync包中的内存访问同步原语，以及何时使用通道和select语句来“通过通信共享内存”。

编写并发执行代码时仍需要了解的一点是如何以结构化的方式来组合这些原语，这些原语是可以扩展并易于理解的。在本书的后半部分，我们将着眼于如何做到这一点。下一章将讲述如何使用社区发现的模式来组合这些原语。

Go的并发编程范式

我们已经探索了Go的并发原语的基础知识，并讨论了如何正确使用这些原语。 在本章中，我们将深入探讨如何将这些基元组合成模式，以帮助保持系统的可扩展性和可维护性。

但是，在我们开始之前，我们需要谈谈本章所包含的一些模式的格式。 在很多示例中，我们将使用传递空接口（接口）的通道。 在Go中使用空接口是有争议的；不过，我出于几个原因这样做。首先，它使得在书的其余部分编写简洁的例子变得更容易。 其次，在某些情况下，我认为这更能代表该模式努力达成的目标。 我们将在“管道”部分更直接地讨论这一点。

接下来，让我们深入了解Go的一些并发模式。

访问范围约束

在使用并发代码时，安全操作有几种不同的选项。 我们已经使用并了解了其中两个：

- 用于共享内存的同步原语(例如sync.Mutex)
- 通过通信同步(例如channel)

此外，有多个其他选项在多个并发进程中隐式安全：

- 不可变数据
- 受限制条件保护的数据

从某种意义上讲，不可变数据是最理想的，因为它隐式地是并行安全的。每个并发进程可以在同一条的数据上运行，但不能修改它。如果要创建新数据，则必须创建所需修改数据的副本。 这不仅可以减轻开发人员认知负担，还可以让程序执行的更快(在某些情况下)。在Go中，可以通过使用值的副本而非该值的指针来实现此目的。 有些语言支持使用明确不变的值的指针； 然而，Go不在其中。

不可变数据的使用依赖于约定——在我看来，坚持约定很难在任何规模的项目上进行协调，除非你有工具在每次有人提交代码时对代码进行静态分析。这里就有一个例子：

```
data := make([]int, 4)

loopData := func(handleData chan<- int) {
    defer close(handleData)
    for i := range data {
        handleData <- data[i]
    }
}

handleData := make(chan int)
go loopData(handleData)

for num := range handleData {
    fmt.Println(num)
}
```

我们可以看到，loopData函数和对handleData通道的循环都使用了整数切片data，但只有loopData对其进行了直接访问。但想想看，随着代码被其他的开发人员触及和修改，明显的，不明显的问题都有可能会被加入其中，并最终产生严重的错误（因为我们没有对data切片做显示的访问和操作约束）。正如我所提到的，一个静态分析工具可能会发现这类问题，但如此灵活的静态分析并不是很多团队能够实现的。 这就是为什么我更喜欢词汇约束，使用编译器来执行对变量的操作进行约束是非常好的。

词法约束涉及使用词法作用域仅公开用于多个并发进程的正确数据和并发原语。 这使得做错事情变得不可能。 实际上，我们在第3章已经谈到了这个话题。回想一下通道部分，它讨论的只是将通道的读或写操作暴露给需要它们的并发进程。 我们再来看看这个例子：

```
chanOwner := func() <-chan int {
    results := make(chan int, 5) //1
    go func() {
        defer close(results)
        for i := 0; i <= 5; i++ {
            results <- i
        }
    }()
    return results
}
```

```
consumer := func(results <-chan int) { //3
    for result := range results {
        fmt.Printf("Received: %d\n", result)
    }
    fmt.Println("Done receiving!")
}

results := chanOwner() //2
consumer(results)
```

- 这里我们在chanOwner函数的词法范围内实例化通道。这将导致通道的写入操作范围被限制在它下面定义的闭包中。 换句话说，它限制了这个通道的写入使用范围，以防止其他goroutine写入它。
- 在这里，我们接受到一个只读通道，我们将它传递给消费者，消费者只能从中读取信息。
- 这里我们收到一个int通道的只读副本。通过声明该函数的唯一用法是读取访问，我们将通道用法限制为只读。

这样的设计方式就可以把通道的读取写入限制在一定的范围内。这个例子可能不是非常的有趣，因为通道是并发安全的。我们来看一个对非并发安全的数据结构约束的示例，它是一个bytes.Buffer实例：

```
printData := func(wg *sync.WaitGroup, data []byte) {
    defer wg.Done()

    var buff bytes.Buffer
    for _, b := range data {
        fmt.Fprintf(&buff, "%c", b)
    }
    fmt.Println(buff.String())
}

var wg sync.WaitGroup
wg.Add(2)
data := []byte("golang")
go printData(&wg, data[:3]) // 1
go printData(&wg, data[3:]) // 2

wg.Wait()
```

1. 这里我们传入包含前三个字节的data切片。
2. 这里我们传入包含剩余三个字节的data切片。
   在这个例子中，你可以看到，我们不需要通过通信同步内存访问或共享数据。

那么这样做有什么意义呢？ 如果我们有同步功能，为什么要给予约束？ 答案是提高了性能并降低了开发人员的认知负担。同步带来了成本，如果你可以避免它，你就不必支付同步它们的成本。 你也可以通过同步回避所有可能的问题。利用词法约束的并发代码通常更易于理解。

话虽如此，建立约束可能很困难，所以有时我们必须回到使用并发原语的开发思路上去。

fo-select循环

在Go程序中你会一遍又一遍地看到for-select循环：

```
for { // 无限循环或遍历
    select {
    // 对通道进行操作
    }
}
```

比较常见的有以下几种不同的情况：

在通道上发送迭代变量

通常情况下，你需要将可迭代的内容转换为通道上的值。

```
for _, s := range []string{"a", "b", "c"} {
    select {
    case <-done:
        return
    case stringStream <- s:
    }
}
```

无限循环等待停止
创建无限循环直到停止的例子很常见。这有一些变化。 你选择哪一个纯粹是一种偏好。

第一种变体保持select语句尽可能短：

```
for {
    select {
    case <-done:
        return
    default:
    }

    // 执行非抢占任务
}
```

如果done通道没有关闭，我们会退出select语句并执行循环体剩下的部分。

第二种变体将任务嵌入到select语句的默认子句中：

```
for {
    select {
    case <-done:
        return
    default:
    // 执行非抢占任务
    }
}
```

当我们进入select语句时，如果done通道尚未关闭，我们将执行default子句。

这种模式没有什么高深的地方，但它展示了最常见的使用方式，所以值得一提。

防止Goroutine泄漏

正如我们在“Goroutines”一节中介绍的那样，goroutines占用资源较少且易于创建。运行时将多个goroutine复用到任意数量的操作系统线程，以便我们不必担心抽象级别。但是他们会花费成本资源，并且goroutine不会被运行时垃圾收集，所以无论内存占用多少，我们都不想让他们对我们的进程撒谎。 那么我们如何去确保他们被清理干净？

让我们从头开始，一步一步思考：为什么会有一个goroutine？ 在第二章中，我们确定，goroutines代表可能并行或不可以并行运行的工作单元。 该goroutine有几条路径终止：

- 当它完成任务。
- 当它遇到不可恢复的错误无法继续它的任务。
- 当它被告知停止当前任务。

前两条我们已经知晓，可以通过算法实现。但如何取消当前任务？由于网络效应，这最重要的一点是：如果你已经开始了一个goroutine，那么它很可能以某种有组织的方式与其他几个goroutines合作。我们甚至可以把这种相互连接表现为一张图表，这时该goroutine能否停下来还取决于处在交互的其他goroutines。我们将在下一章中继续关注大规模并发产生的相互依赖关系，但现在让我们考虑如何确保保证单个goroutine得到清理。 让我们从一个简单的goroutine泄漏开始：

```
doWork := func(strings <-chan string) <-chan interface{} {
    completed := make(chan interface{})
    go func() {
        defer fmt.Println("doWork exited.")
        defer close(completed)
        for s := range strings {
            fmt.Println(s)
        }
    }()
    return completed
}

doWork(nil)
// 这里还有其他任务执行
fmt.Println("Done.")
```

我们看到doWork被传递了一个nil通道。所以strings通道永远无法读取到其承载的内容，而且包含doWork的goroutine将在这个过程的整个生命周期中保留在内存中（如果我们在doWork和主goutoutine中加入了goroutine，我们甚至会死锁）。

在这个例子中，整个进程的生命周期很短，但是在一个真正的程序中，goroutines可以很容易地在一个长期生命的程序开始时启动，导致内存利用率下降。

解决这种情况的方法是建立一个信号，按照惯例，这个信号通常是一个名为done的只读通道。父例程将该通道传递给子例程，然后在想要取消子例程时关闭该通道。 这是一个例子：

```
doWork := func(done <-chan interface{}, strings <-chan string) <-chan interface{} { //1
    terminated := make(chan interface{})
    go func() {
        defer fmt.Println("doWork exited.")
        defer close(terminated)
        for {
            select {
            case s := <-strings:
                // Do something interesting
                fmt.Println(s)
            case <-done: //2
                return
            }
```

```
        }
    }()
    return terminated
}

done := make(chan interface{})
terminated := doWork(done, nil)

go func() { //3
    // Cancel the operation after 1 second.
    time.Sleep(1 * time.Second)
    fmt.Println("Canceling doWork goroutine...")
    close(done)
}()

<-terminated //4
fmt.Println("Done.")
```

1. 这里我们传递done通道给doWork函数。作为惯例，这个通道被作为首个参数。
2. 这里我们看到使用了for-select的使用模式之一。我们的目的是检查done通道 有没有发出信号。如果有的话，我们退出当前goroutine。
3. 在这里我们创建另一个goroutine，一秒后就会取消doWork中产生的goroutine。
4. 这是我们在main goroutine中调用doWork函数返回结果的地方。

这会输出：

```
Canceling doWork goroutine...
doWork exited.
Done.
```

你可以看到尽管向doWork传递了nil给strings通道，我们的goroutine依然正常运行至结束。与之前的例子不同，本例中我们把两个goroutine连接在一起之前，我们建立了第三个goroutine以取消doWork中的goroutine，并成功消除了泄漏问题。

前面的例子很好地处理了在通道上接收goroutine的情况，但是如果我们正在处理相反的情况：在尝试向通道写入值时阻塞goroutine会怎样？

```
newRandStream := func() <-chan int {
    randStream := make(chan int)
    go func() {
        defer fmt.Println("newRandStream closure exited.") // 1
        defer close(randStream)
        for {
            randStream <- rand.Int()
        }
    }()

    return randStream
}

randStream := newRandStream()
fmt.Println("3 random ints:")
for i := 1; i <= 3; i++ {
    fmt.Printf("%d: %d\n", i, <-randStream)
}
```

1. 当goroutine成功执行时我们打印一行消息。

这会输出：

```
3 random ints:
1: 5577006791947779410
2: 8674665223082153551
3: 6129484611666145821
```

你可以看到注释1所在的打印语句并未执行。在循环的第三次迭代之后，我们的goroutine块试图将下一个随机整数发送到不再被读取的通道。我们无法告知它停下来，解决方案是为生产者提供一条通知它退出的通道：

```
newRandStream := func(done <-chan interface{}) <-chan int {
    randStream := make(chan int)
    go func() {
        defer fmt.Println("newRandStream closure exited.")
        defer close(randStream)

        for {
            select {
            case randStream <- rand.Int():
            case <-done:
                return
            }
        }

    }()

    return randStream
}

done := make(chan interface{})
randStream := newRandStream(done)
fmt.Println("3 random ints:")
for i := 1; i <= 3; i++ {
    fmt.Printf("%d: %d\n", i, <-randStream)
}

close(done)
//模拟正在进行的工作
time.Sleep(1 * time.Second)
```

这会输出：

```
3 random ints:
1: 5577006791947779410
2: 8674665223082153551
3: 6129484611666145821
newRandStream closure exited.
```

我们现在看到该goroutine被妥善清理。
现在我们知道如何确保goroutine不泄漏，我们可以制定一个约定：如果goroutine负责创建goroutine，它也负责确保它可以停止goroutine。

这个约定有助于确保程序在组合和扩展时可用。我们将在“管道”和“context包”中重新讨论这种技术和规则。我们该如何确保goroutine能够被停止根据goroutine的类型和用途而有所不同，但是它们 所有这些都是建立在传递done通道基础上的。

go

or-channel

有时你可能会发现自己考虑将一个或多个done通道合并到一个done通道中，该通道在任何组件通道关闭时关闭。编写一个执行这种耦合度较高的select语句是可行的，尽管很冗长；但是有时你无法知道运行状态下done通道的数量。在这种情况下，或者你如果喜欢单线操作，你可以使用or通道模式将这些通道组合在一起（如果你对done通道看着有点懵，可以先看看上一节）。

这种模式使用递归和goroutine创建一个复合done通道。 我们来看一下：

```
var or func(channels ...<-chan interface{}) <-chan interface{}

or = func(channels ...<-chan interface{}) <-chan interface{} { //1

    switch len(channels) {
    case 0: //2
        return nil
    case 1: //3
        return channels[0]
    }

    orDone := make(chan interface{})
    go func() { //4
        defer close(orDone)

        switch len(channels) {
        case 2: //5
            select {
            case <-channels[0]:
            case <-channels[1]:
            }
        default: //6
            select {
            case <-channels[0]:
            case <-channels[1]:
            case <-channels[2]:
            case <-or(append(channels[3:], orDone)...): //6
            }
        }
    }()
    return orDone
}
```

1. 这里我们建立了名为or的函数，接收数量可变的通道并返回单个通道。
2. 由于这是个递归函数，我们必须设置终止条件。第一个条件是，如果传入的切片是空的，我们简单的返回一个nil通道。这与不传递通道的想法一致：我们不希望复合通道做任何事。
3. 第二个递归终止条件是，如果切片只含有一个元素，我们就返回给元素。
4. 这是该函数最重要的部分，也是递归产生的地方。我们建立一个goroutine，以便可以不受阻塞地等待我们通道上的消息。
5. 由于我们这里是递归的，每次递归调用将至少有两个通道。作为保持goroutine数量受到限制的优化方法，们在这里为仅使用两个通道的时设置了一个特殊情况。
6. 在这里，我们递归地在第三个索引之后，从我们切片中的所有通道中创建一个or通道，然后从中选择。递归操作会逐层累计直到取到第一个通道元素。我们在其中传递了orDone通道，这样当该树状结构顶层的goroutines退出时，结构底层的goroutines也会退出。

这是一种奇妙的做法，你可以将任意数量的通道组合到单个通道中，只要任何作为组件的通道关闭或被写入，整个通道就会关闭。让我们来看看该如何进行实际操作。下面这个例子将经过一段时间后关闭通道，然后使用or函数将这些通道合并到一个关闭的通道中：

```
sig := func(after time.Duration) <-chan interface{} { //1
    c := make(chan interface{})
    go func() {
        defer close(c)
        time.Sleep(after)
    }()
    return c
}

start := time.Now() //2
<-or(sig(2*time.Hour), sig(5*time.Minute), sig(1*time.Second), sig(1*time.Hour), sig(1*time.Minute))
fmt.Printf("done after %v", time.Since(start)) //3
```

1. 此功能只是创建了一个通道，当后续时间中指定的时间结束时将关闭该通道。
2. 在这里，我们设置追踪自or函数的通道开始阻塞的起始时间。
3. 在这里我们打印阻塞发生的时间。

这会输出：

```
done after 1.000216772s
```

请注意，尽管在我们的调用中放置了多个通道需要多个时间才能关闭，但我们在一秒钟后关闭的通道会导致由该d调用创建的整个通道关闭。 这是因为它位于树或函数构建的树中，它将始终第一个关闭，因此依赖于其关闭的通道也将关闭。

我们以额外创建 f(x)=x/2 个goroutine以”简洁的”实现该目的，其中x是goroutine的数量。请记住Go的一个优点是能够快速创建，调度和运行goroutines，并且 该语言积极鼓励使用goroutines来正确建模问题。无需在前期太担心在这里创建的分支太多。如果在编译时你不知道自己正在使用多少个done通道，那么恐怕就没有其他更好的方法来合并done通道了。

这种模式适用于系统中模块的交叉2021-01-02 15:56:03 星期六点。在这些交叉点，有多种条件通过你的调用堆栈取消goroutines树。 使用or函数，你可以简单地将它们组合在一起并将其传递给堆栈。 我们将在“context包”中看到另一种更具描述性的做法。

我们也将看到这种模式的变体在第五章“重复请求”中形成更复杂的模式。

错误处理

在并发编程中，错误处理可能难以正确运行。有时候，我们花了很多时间思考我们的各种流程将如何共享信息和协调，却忘记考虑如何优雅地处理错误。Go避开了流行的错误异常模型，Go认为错误处理非常重要，并且在开发程序时，我们应该像关注算法一样关注它。本着这种精神，让我们来看看在处理多个并发进程时我们如何做到这一点。

思考错误处理时最根本的问题是，“应该由谁负责处理错误？”在某些情况下，程序需要停止传递堆栈中的错误，并将它们处理掉，这样的操作应该何时执行呢？

在并发进程中，这样的问题变得愈发复杂。因为一个并发进程独立于其父进程或兄弟进程运行，所以可能很难推断出错误是如何产生的。
下面的就展示了这样的问题：

```
checkStatus := func(done <-chan interface{}, urls ...string, ) <-chan *http.Response {

    responses := make(chan *http.Response)
    go func() {
        defer close(responses)
        for _, url := range urls {
            resp, err := http.Get(url)
            if err != nil {
                fmt.Println(err) //1
                continue
            }
            select {
            case <-done:
                return
            case responses <- resp:
            }
        }
    }()

    return responses
}

done := make(chan interface{})
defer close(done)

urls := []string{"https://www.baidu.com", "https://badhost"}
for response := range checkStatus(done, urls...) {
    fmt.Printf("Response: %v\n", response.Status)
}
```

- 这个我们看到goroutine尽其最大努力展示错误信号。但也仅仅是展示出来，它还能做什么？ 它无法传回！ 如果错误种类太多怎么办？ 再请求一遍吗？

这会输出：

```
Response: 200 OK
Get https://badhost: dial tcp: lookup badhost on 127.0.1.1:53: no such host
```

我们看到代码中并没有给goroutine更多的选择以处理可能出现的错误。它不能简单的把这个错误不加任何处理的抛弃掉，所以当前唯一明智的做法是：它会打印错误并希望受到程序使用者的关注。别把你的goroutine像这样放到如此尴尬的处境之下。我建议你把程序的关注点分离：一般来说，你的并发进程应该把错误发送到你的程序的另一部分，这样程序状态的完整信息就被保留下来，并留出余地让使用者可以做出更明智的决定来处理它。我们对上面的例子做了一点点修改：

```
type Result struct { //1
    Error    error
    Response *http.Response
}
checkStatus := func(done <-chan interface{}, urls ...string) <-chan Result { //2

    results := make(chan Result)
    go func() {
        defer close(results)

        for _, url := range urls {
            var result Result
            resp, err := http.Get(url)
            result = Result{Error: err, Response: resp} //3
            select {
            case <-done:
                return
            case results <- result: //4
            }
        }
    }()

    return results
}
done := make(chan interface{})
defer close(done)

urls := []string{"https://www.baidu.com", "https://badhost"}
for result := range checkStatus(done, urls...) {
    if result.Error != nil { //5
        fmt.Printf("error: %v", result.Error)
        continue
    }
    fmt.Printf("Response: %v\n", result.Response.Status)
}
```

1. 这里我们创建一个包含*http.Response和goroutine循环迭代中可能出现的错误类型。
2. 该行返回一个可读取的通道，以检索循环迭代的结果。
3. 在这里，我们创建一个Result实例，并设置Error和Response字段。
4. 这是我们将结果写入通道。
5. 在这里，在我们的main goroutine中，我们能够自行处理由checkStatus中出现的错误，并获取详细的响应信息。

这会输出：

```
Response: 200 OK
error: Get https://badhost: dial tcp: lookup badhost on 127.0.1.1:53: no such host
```

这里要注意的关键是我们如何将潜在的结果与潜在的错误结合起来。我们已经成功地将错误处理的担忧从生产者中分离出来。这是可取的，因为生成goroutine的goroutine（在这种情况下是我们的main goroutine）拥有更多关于正在运行的程序的上下文，并且可以做出关于如何处理错误的更明智的决定。

在前面的例子中，我们只是将错误写入stdio，但我们可以做其他事情。 让我们稍微修改我们的程序，以便在发生三个或更多错误时停止错误检查：

```
done := make(chan interface{})
defer close(done)

errCount := 0
```

```
urls := []string{"a", "https://www.baidu.com", "b", "c", "d"}
for result := range checkStatus(done, urls...) {
    if result.Error != nil {
        fmt.Printf("error: %v\n", result.Error)
        errCount++
        if errCount >= 3 {
            fmt.Println("Too many errors, breaking!")
            break
        }
        continue
    }
    fmt.Printf("Response: %v\n", result.Response.Status)
}
```

这会输出：

```
error: Get a: unsupported protocol scheme ""
Response: 200 OK
error: Get b: unsupported protocol scheme ""
error: Get c: unsupported protocol scheme ""
Too many errors, breaking!
```

你可以看到，因为错误是从checkStatus返回的而不是在goroutine内部处理的，所以错误处理遵循熟悉的Go规范。 这是个简单的例子，但不难想象在更大更复杂的的程序下是什么样子。这里的主要内容是，在构建从goroutines返回的价值时，应将错误视为一等公民。 如果你的goroutine可能产生错误，那么这些错误应该与你的结果类型紧密结合，并且通过相同的通信线路传递——就像常规的同步函数一样。

管道

当你编写一个程序时，你可能不会坐下来写一个长函数——至少我希望你不要！ 你会以函数，结构体，方法等形式构造抽象。为什么要这样做？ 部分是为了抽象出无关的细节，另一部分是为了能够在不影响其他区域的情况下处理某个代码区域。你没有必要改变整个系统，当发现自己必须调整多个区域才能做出一个合乎逻辑的改变时，说明该系统的抽象实在是很糟糕。

pipline，又名管道，或者叫流水线，是你可以用来在系统中形成抽象的另一种工具。特别是当你的程序需要处理流或批处理数据时，它是一个非常强大的工具。管道这个词被认为是在1856年首次使用的，指将液体从一个地方输送到另一个地方的一系列管道。计算机科学借用了这个术语，因为我们也在从一个地方向另一个地方传输某些东西：数据。管道是个系统，它将一系列数据输入，执行操作并将数据传回。我们称这些操作都是管道的一个阶段。

通过使用管道，你可以分离每个阶段的关注点，这提供了许多好处。你可以独立于彼此修改模块，你可以混合搭配模块的组合方式，而无需修改模块，你可以让每个模块同时处理到上游或下游模块，并且可以扇出或限制你的部分管道。 我们将在"扇出，扇入"一节中介绍扇出，我们将在第5章介绍速率限制。你现在不必担心这些奇怪的术语； 让我们从最简单的开始，尝试构建一个管道。

如前所述，一个阶段只是类似于执行将数据输入，对其进行转换并将数据发回这样的功能。 这是一个可以被视为管道某阶段的例子：

```
multiply := func(values []int, multiplier int) []int {
    multipliedValues := make([]int, len(values))
    for i, v := range values {
        multipliedValues[i] = v * multiplier
    }
    return multipliedValues
}
```

这个函数用取整数切片，循环遍历它们，然后返回一个新的切片。看起来很无聊的功能，对吧？ 让我们创建管道的另一个阶段：

```
add := func(values []int, additive int) []int {
    addedValues := make([]int, len(values))
    for i, v := range values {
        addedValues[i] = v + additive
    }
    return addedValues
}
```

跟上个函数类似，只不过把乘法变成了加法。接下来，让我们尝试将它们合并：

```
ints := []int{1, 2, 3, 4}
for _, v := range add(multiply(ints, 2), 1) {
    fmt.Println(v)
}
```

这会输出：

```
3
5
7
9
```

看看我们是如何将他们结合起来的。这些函数就像你每天工作使用的函数一样，但是我们可以将它们组合起来形成一个流水线。 那么我们怎么定义管道的"阶段"呢？

- 一个阶段消耗并返回相同的类型。
- 一个阶段必须通过语言来体现，以便它可以被传递。 Go的函数已经实现并很好地适用于这个目的。

那些熟悉函数式编程的人可能会点头并思考像高阶函数和monad这样的术语。 事实上，管道的各阶段与函数式编程密切相关，可以被认为是monad的一个子集。 我不会在这里明确地讨论Monad或函数式编程，但它们本身就是有趣的主题，在尝试理解管道时，对这两个主题的工作知识虽然没有必要，但是对于加强理解是有用的。

在这里，我们的add和multiply满足管道的阶段属性：它们都消耗int切片并返回int切片，并且因为Go支持函数传递，所以我们可以传递add和multiply。 这些属性很有趣：在不改变阶段本身的情况下，我们可以很容易地将我们的阶段结合到更高层次。

例如，如果我们现在想为管道添加一个额外的阶段：乘以2，只需将我们以前的管道包装在一个新的阶段内，就像这样：

```
ints := []int{1, 2, 3, 4}
for _, v := range multiply(add(multiply(ints, 2), 1), 2) {
    fmt.Println(v)
}
```

这会输出：

```
6
10
14
18
```

注意我们是如何做到这一点的。也许你已经开始看到使用管道模式的好处了。 当然，我们也可以在程序上编写此代码：

```
ints := []int{1, 2, 3, 4}
for _, v := range ints {
    fmt.Println(2 * (v*2 + 1))
}
```

虽然这看起来简单得多，但正如我们接下来会看到的，程序在处理数据流时不会提供与管道相同的好处。

请注意每个阶段如何获取数据切片并返回数据的。这些阶段的行为我们称为批处理。这意味着它们一次性对大块数据进行操作，而不是一次一个单独进行。还有另一种类型的管道，模块每次仅接收和返回单个元素。

批处理和流处理各有优点和缺点，我们稍微讨论一下。现在，请注意，为使原始数据保持不变，每个阶段都必须创建一个等长的新切片来存储其计算结果。这意味着我们的程序在任何时候的内存占用量都是我们发送到管道开始处的切片大小的两倍。 让我们转换为面向流操作，看看会有什么效果：

```
multiply := func(value, multiplier int) int {
    return value * multiplier
}

add := func(value, additive int) int {
    return value + additive
}

ints := []int{1, 2, 3, 4}
for _, v := range ints {
    fmt.Println(multiply(add(multiply(v, 2), 1), 2))
}
```

这会输出：

```
6
10
14
18
```

每个阶段都接收并返回一个值，程序的内存占用空间将回落到管道输入数据的大小。但是我们不得不将管道放入到for循环的体内，这让我们的操作变“重”了。这不仅限制了我们对管道的重复使用，而且我们稍后会在本节中看到，这也限制了我们的扩展能力。 实际上，我们因为循环而在每次迭代中实例化我们的管道。尽管进行函数调用耗费的资源很少，但函数的调用次数确实增加了。那么如果涉及到并发性又如何？我之前说过，使用管道的好处之一是能够同时处理数据的各个阶段，并且我提到了一些关于扇出的内容。这些我们在接下来会进一步了解到。

我会扩展add和multiply来介绍这些概念。现在开始学习在Go中构建管道的最佳实践的时候了，先从并发原语通道开始。

构建管道的最佳实践

通道非常适合在Go中构建管道，因为它们满足了我们所有的基本要求。它们可以接收并传递值，它们可以在并发中安全的使用，它们可以被遍历，而且它们被语言给予了完美的支持。让我们用通道将之前的例子改造一下：

```
generator := func(done <-chan interface{}, integers ...int) <-chan int {
    intStream := make(chan int)
    go func() {
        defer close(intStream)
        for _, i := range integers {
            select {
            case <-done:
                return
            case intStream <- i:
            }
        }
    }()
    return intStream
}

multiply := func(done <-chan interface{}, intStream <-chan int, multiplier int) <-chan int {
    multipliedStream := make(chan int)
    go func() {
        defer close(multipliedStream)
        for i := range intStream {
            select {
            case <-done:
                return
            case multipliedStream <- i * multiplier:
            }
        }
    }()

    return multipliedStream
}

add := func(done <-chan interface{}, intStream <-chan int, additive int) <-chan int {
    addedStream := make(chan int)
    go func() {
        defer close(addedStream)
        for i := range intStream {
            select {
            case <-done:
                return
            case addedStream <- i + additive:
            }
        }
    }()
    return addedStream
}

done := make(chan interface{})
defer close(done)

intStream := generator(done, 1, 2, 3, 4)
pipeline := multiply(done, add(done, multiply(done, intStream, 2), 1), 2)

for v := range pipeline {
```

```
    fmt.Println(v)
}
```

这会输出：

```
6
10
14
18
```

看起来我们得到了期望的输出结果，但代价是代码更多了。先，我们来看看我们写的是什么。 现在有三个函数，而不是两个。他们都看起来像是在内部开启一个通道，并使用我们在“防止Goroutine泄漏”中建立的模式，通过一个done通道表示该通道应该退出。他们都看起来像返回通道，其中一些看起来像他们也采用了额外的通道。让我们开始进一步分解：

```
done := make(chan interface{})
defer close(done)
```

我们的程序首先创建了done通道，并调用close通过defer延迟执行。正如前面所讨论的那样，这可以确保我们的程序干净地退出，而不泄漏goroutines。没有什么新鲜的。接下来，我们来看看函数generator：

```
generator := func(done <-chan interface{}, integers ...int) <-chan int {
    intStream := make(chan int)
    go func() {
        defer close(intStream)
        for _, i := range integers {
            select {
            case <-done:
                return
            case intStream <- i:
            }
        }
    }()
    return intStream
}

// ...

intStream := generator(done, 1, 2, 3, 4)
```

generator接收整数类型的切片，构造整数类型的通道，启动一个goroutine并返回构造的通道。然后，在创建的goroutine通道上发送切片的值。

请注意，通道上的发送与done通道上的选择共享一条select语句。这是我们在“防止Goroutine泄漏”中建立的模式。

简而言之，generator函数将一组离散值转换为一个通道上的数据流。这种操作的函数我们称之为生成器。在使用管道时，你会经常看到这一点，因为在管道开始时，你总是会有一些需要转换为通道的数据。我们将稍微介绍一些有趣的生成器的几个例子，但我们先来完成对这个程序的分析。 接下来，我们构建管道：

```
pipeline := multiply(done, add(done, multiply(done, intStream, 2), 1), 2)
```

这与本节之前的流程相同：对于一串数字，我们将它们乘以2，加1，然后将结果乘以2。这条管道与我们前面例子中使用函数的管道相似，但它在很重要的方面有所不同。

首先，我们正在使用通道。 这是显而易见的，因为它允许两件事：在我们的管道的末尾，可以使用range语句来提取值，并且在每个阶段我们可以安全地并发执行，因为我们的输入和输出在并发上下文中是安全的。

这给我们带来了第二个区别：管道的每个阶段都在同时执行。 这意味着任何阶段只需要等待其输入，并且能够发送其输出。 事实证明，这会产生巨大的影响，我们将在“扇出，扇入”一节中发现，但现在我们可以简单地注意到它允许各阶段相互独立地执行一段时间。

最后，在我们的例子中，我们对这个管道进行了遍历取值：

```
for v := range pipeline {
    fmt.Println(v)
}
```

下面是一个表格，演示系统中的每个值如何进入每个通道，以及通道何时关闭。

Iteration	Generator	Multiply	Add	Multiply	Value
0	1				
0		1			
0	2		2		
0		2		3	
0	3		4		6
1		3		5	
1	4		6		10
2	(closed)	4		7	
2		(closed)	8		14
3			(closed)	9	
3				(closed)	18

让我们更仔细地研究一下这个模式来标示goroutines退出。当处理多个相互依赖的goroutines时，这种模式如何起作用？ 如果我们在程序完成执行之前在完成的通道上调用close，会发生什么情况？

要回答这些问题，再来看看管道是构建的这一行：

```
pipeline := multiply(done, add(done, multiply(done, intStream, 2), 1), 2)
```

管道的各阶段通过两种方式连接在一起：通过默认的done通道，和被传递给后续阶段的通道。换句话说，multiply函数创建的通道被传递给add函数。让我们重新审视前面的表格，并在完成之前，关闭done通道，看看会发生什么：

Iteration	Generator	Multiply	Add	Multiply	Value
0	1				
0		1			
0	2		2		
0		2		3	
1	3		4		6
close(done)	(closed)	3		5	
		(closed)	6		
			(closed)	7	
				(closed)	
					(exit range)

看到关闭done通道是如何影响到管道的了么？这是通过管道每个阶段的两件事情实现的：

- 对传入的频道进行遍历。当输入通道关闭时，遍历操作将退出。
- 发送操作与done通道共享select语句。

无论流水线阶段处于等待数据通道的状态，还是处在等待发送通道关闭的状态，都会强制管道各阶段终止。
这里有一个复发关系。在管道开始时，我们已经确定必须将传入的切片值转换为通道。在这个过程中有两点必须是可抢占的：

- 在生成器通道上创建值。
- 在其频道上发送离散值。

在我们的例子中，在生成器函数中，离散值是通过遍历切片生成的，它足够快，不需要被抢占。 第二个是通过我们的select语句和done通道处理的，它确保发生器即使被阻塞试图写入intStream也是可抢占的。

在管道的另一端，同样我们可以确保最终阶段的可抢占性。因为我们正在操作的通道在抢占时会被关闭，所以当这种情况发生时，通道将会中断。 最后阶段是可抢占的，因为我们依赖的流是可抢占的。

在管道开始和结束之间，代码总是在一个通道上遍历，并在包含done通道的select语句内的另一个通道上发送。

如果某个阶段在传入通道检索到值时被阻塞，则该通道关闭时它将变为未阻塞状态。 如果某个阶段在发送值时被阻塞，则由于select语句而可抢占。

因此，我们的整个管道始终可以通过关闭done通道来抢占。

便利的生成器

我早些时候承诺会演示一些可能广泛使用的有趣的生成器。我们来看看一个名为repeat的生成器：

```
repeat := func(done <-chan interface{}, values ...interface{}) <-chan interface{} {

    valueStream := make(chan interface{})
    go func() {
        defer close(valueStream)
        for {
            for _, v := range values {
                select {
                case <-done:
                    return
                case valueStream <- v:
                }
            }
        }
    }()
    return valueStream
}
```

这个函数会重复你传给它的值，直到你告诉它停止。 让我们来看看另一个函数take，它在与repeat结合使用时很有用：

```
take := func(done <-chan interface{}, valueStream <-chan interface{}, num int, ) <-chan interface{} {

    takeStream := make(chan interface{})
    go func() {
        defer close(takeStream)
        for i := 0; i < num; i++ {
            select {
            case <-done:
                return
            case takeStream <- <-valueStream:
            }
        }
    }()
    return takeStream
}
```

这个函数会从其传入的valueStream中取出第一个元素然后退出。二者组合起来会怎么样呢？

```
done := make(chan interface{})
defer close(done)

for num := range take(done, repeat(done, 1), 10) {
    fmt.Printf("%v ", num)
}
```

这会输出：

```
1 1 1 1 1 1 1 1 1 1
```

在这个基本的例子中，我们创建了一个repeat生成器来生成无限数量的重复生成器，但是只取前10个。repeat生成器由take接收。虽然我们可以生成无线数量的流，但只会生成n+1个实例，其中n是我们传入take的数量。

我们可以扩展这一点。让我们创建另一个生成器，但是这次我们创建一个重复调用函数的生成器repeatFn：

```
repeatFn := func(done <-chan interface{}, fn func() interface{}) <-chan interface{} {

    valueStream := make(chan interface{})
    go func() {
        defer close(valueStream)
        for {
            select {
            case <-done:
                return
            case valueStream <- fn():
            }
        }
    }()
    return valueStream
}
```

我们用它来生成10个随机数：

```
done := make(chan interface{})
defer close(done)

rand := func() interface{} {
    return rand.Int()
}

for num := range take(done, repeatFn(done, rand), 10) {
    fmt.Println(num)
}
```

这会输出：

```
5577006791947779410
8674665223082153551
6129484611666145821
4037200794235010051
3916589616287113937
6334824724549167320
605394647632969758
1443635317331776148
894385949183117216
2775422040480279449
```

您可能想知道为什么所有这些发生器通道类型都是interface{}。

Go中的空接口有点争议，但我认为处理interface的通道方便使用标准的管道模式。 正如我们前面所讨论的，管道的强大来自可重用的阶段。当阶段以适合自身的特异性水平进行操作时，这是最好的。在repeat和repeatFn生成器中，我们需要关注的是通过在列表或运算符上循环来生成数据流。这些操作都不需要关于处理的类型，而只需要知道参数的类型。

当需要处理特定的类型时，可以放置一个执行类型断言的阶段。有一个额外的管道阶段和类型断言的性能开销可以忽略不计，正如我们稍后会看到的。 以下是一个介绍toString管道阶段的小例子：

```
toString := func(done <-chan interface{}, valueStream <-chan interface{}, ) <-chan string {
    stringStream := make(chan string)
    go func() {
        defer close(stringStream)
```

```
        for v := range valueStream {
            select {
            case <-done:
                return
            case stringStream <- v.(string):
            }
        }
    }()
    return stringStream
}
```

可以这样使用它：

```
done := make(chan interface{})
defer close(done)

var message string
for token := range toString(done, take(done, repeat(done, "I", "am."), 5)) {
    message += token
}

fmt.Printf("message: %s...", message)
```

这会输出：

```
message: Iam.Iam.I...
```

现在让我们证明刚才提到的性能问题。我们将编写两个基准测试函数：一个测试通用阶段，一个测试类型特定阶段：

```
func BenchmarkGeneric(b *testing.B) {
    done := make(chan interface{})
    defer close(done)

    b.ResetTimer()
    for range toString(done, take(done, repeat(done, "a"), b.N)) {
    }
}

func BenchmarkTyped(b *testing.B) {

    repeat := func(done <-chan interface{}, values ...string) <-chan string {

        valueStream := make(chan string)
        go func() {

            defer close(valueStream)
            for {
                for _, v := range values {
                    select {
                    case <-done:
                        return
                    case valueStream <- v:
                    }
                }
            }
        }()
        return valueStream
    }
```

```
    take := func(done <-chan interface{}, valueStream <-chan string, num int, ) <-chan string {
        takeStream := make(chan string)
        go func() {
            defer close(takeStream)
            for i := num; i > 0 || i == -1; {
                if i != -1 {
                    i--
                }
                select {
                case <-done:
                    return
                case takeStream <- <-valueStream:
                }
            }
        }()

        return takeStream
    }

    done := make(chan interface{})
    defer close(done)

    b.ResetTimer()
    for range take(done, repeat(done, "a"), b.N) {
    }
}
```

这会输出：

BenchmarkGeneric-4	1000000	2266 ns/op
BenchmarkTyped-4	1000000	1181 ns/op
1181 ns/op	command-line-arguments	3.486s

可以看到，特定类型的速度是接口类型的2倍。一般来说，管道上的限制因素将是生成器，或者是密集计算的某个阶段。如果生成器不像repeat和repeatFn生成器那样从内存中创建流，则可能会受I/O限制。从磁盘或网络读取数据可能会超出此处显示的性能开销。

那么，如果真是在计算上存在性能瓶颈，我们该怎么办？基于这种情况，让我们来讨论扇出扇入技术。

扇入扇出

那么你已经建立了一条管道。 数据在你的系统中欢畅地流动，并在莫连接在一起的各个阶段发生变化。 它就像一条美丽的溪流； 一个美丽的，缓慢的溪流，哦，我的上帝为什么这需要这么久？

有时候，管道中的各个阶段可能在计算上特别耗费资源。当发生这种情况时，管道中的上游阶段可能会在等待完成时被阻塞。不仅如此，管道本身可能需要很长时间才能整体执行。 我们如何解决这个问题？

管道的一个有趣属性是它的各个阶段相互独立，方便组合。你可以多次重复使用管道的各个阶段。因此，在多个goroutine上重用管道的单个阶段实现并行化，将有助于提高管道的性能。

事实上，这种模式被称为扇入扇出。

扇出（Fan-out）是一个术语，用于描述启动多个goroutines以处理来自管道的输入的过程，并且扇入（fan-in）是描述将多个结果组合到一个通道中的过程的术语。

那么在什么情况下适用于这种模式呢？如果出现以下两种情况，你就可以考虑这么干了：

- 不依赖模块之前的计算结果。
- 运行需要很长时间。

运行的独立性是非常重要的，因为你无法保证各阶段的并发程序以何种顺序运行，也无法保证其返回的顺序。

我们来看一个例子。在下面的例子中，构建了一个寻找素数的方法。我们将使用在“管道”中的经验，创建各个阶段，并将它们拼接在一起：

```
rand := func() interface{} { return rand.Intn(50000000) }

done := make(chan interface{})
defer close(done)

start := time.Now()

randIntStream := toInt(done, repeatFn(done, rand))
fmt.Println("Primes:")
for prime := range take(done, primeFinder(done, randIntStream), 10) {
    fmt.Printf("\t%d\n", prime)
}

fmt.Printf("Search took: %v", time.Since(start))
```

这会输出：

```
Primes:
24941317
36122539
6410693
10128161
25511527
2107939
14004383
7190363
45931967
2393161
Search took: 23.437511647s
```

我们生成一串随机数，最大值为50000000，将数据流转换为整数流，然后将其传入primeFinder。pri meFinder会尝试将输入流提供的数字除以比它小的每个数字。如果不成功，会将该值传递到下一个阶段。当然，这个方法很低效，但它符合我们程序运行时间较长的要求。

在我们的for循环中，搜索找到的素数，在进入时将它们打印出来，并且take在找到10个素数后关闭管道。然后，我们打印出搜索需要多长时间，完成的通道被延迟声明关闭，管道停止 。

为了避免结果中出现重复，我们可以把已找到的素数缓存起来，但为了简单起见，我们将忽略这些。

你可以看到大概需要23秒才能找到10个素数，这实在是有点慢。通常遇到这种情况，我们首先看一下算法本身，也许是拿一本算法书籍，然后看看我们是否能在哪个阶段改进。但是，由于目的是通过扇出来解决该问题，所以算法我们暂时先不去管它。

我们的程序现在有两个阶段：生成随机数和筛选素数。在更大的程序中，你的管道可能由更多的阶段组成，那我们该对什么样的阶段使用扇出模式进行改进？请记住我们之前提出的标准：执行顺序的独立性和执行时间。我们的随机数生成器肯定是与顺序无关的，但运行起来并不需要很长的时间。PrimeFinder阶段也是顺序无关的，因为我们采用的算法效率非常低下，它需要很长时间才能运行完成。因此，我们可以把关注点放在PrimeFinder身上。

为此，我们可以将其操作拆散，就像这样：

```
numFinders := runtime.NumCPU()
finders := make([]<-chan int, numFinders)
for i := 0; i < numFinders; i++ {
    finders[i] = primeFinder(done, randIntStream)
}
```

在我的电脑上，runtime.NumCPU()返回8，在生产中，我们可能会做一些经验性的测试来确定CPU的最佳数量，但在这里我们将保持简单，并且假设只有一个findPrimes阶段的CPU会被占用。

这就好像一个班级的作业，原本由1位老师批改，现在变成了8位老师同时批改。

接下来我们遇到的问题是，如何将结果汇总到一起。为此，我们开始考虑使用扇入(fan-in)。

正如我们前面所提到的，扇入意味着将多个数据流复用或合并成一个流。 这样做相对简单：

```
fanIn := func(done <-chan interface{}, channels ...<-chan interface{}) <-chan interface{} { // 1

    var wg sync.WaitGroup // 2
    multiplexedStream := make(chan interface{})

    multiplex := func(c <-chan interface{}) { // 3
        defer wg.Done()
        for i := range c {
            select {
            case <-done:
                return
            case multiplexedStream <- i:
            }
        }
    }

    // 从所有的通道中取数据
    wg.Add(len(channels)) // 4
    for _, c := range channels {
        go multiplex(c)
    }

    // 等待所有数据汇总完毕
    go func() { // 5
        wg.Wait()
```

```
        close(multiplexedStream)
    }()

    return multiplexedStream
}
```

1. 一如既往，我们使用done通道来关闭衍生的goroutine，并接收接口类型的通道切片来汇总数据。
2. 这里我们使用sync.WaitGroup以等待全部通道读取完成。
3. 我们在这里建立函数multiplex，它会读取传入的通道，并把该通道的值放入multiplexedStream。
4. 这里增加等待计数。
5. 这里我们建立一个goroutine等待汇总完毕。**这样函数块可以快速return，不必等待wg.Wait()。这种用法不多见，但在这里很符合场景需求。**

简而言之，扇入涉及读取多路复用通道，然后为每个传入通道启动一个goroutine，以及在传入通道全部关闭时关闭复用通道。由于我们要创建一个等待N个其他goroutine完成的goroutine，因此创建sync.WaitGroup来协调处理是有意义的。multiplex还通知WaitGroup它已执行完成。

额外提醒，在对返回结果的顺序有要求的情况下扇入扇出可能工作的不是很好。我们没有做任何事情来保证从randIntStream中读取数据的顺序。稍后，我们将看一个维护顺序的例子。

让我们把所有这些改进放在一起，看看运行时长是否有所减少：

```
done := make(chan interface{})
defer close(done)

start := time.Now()

rand := func() interface{} { return rand.Intn(50000000) }

randIntStream := toInt(done, repeatFn(done, rand))

numFinders := runtime.NumCPU()
fmt.Printf("Spinning up %d prime finders.\n", numFinders)
finders := make([]<-chan interface{}, numFinders)
fmt.Println("Primes:")
for i := 0; i < numFinders; i++ {
    finders[i] = primeFinder(done, randIntStream)
}

for prime := range take(done, fanIn(done, finders...), 10) {
    fmt.Printf("\t%d\n", prime)
}

fmt.Printf("Search took: %v", time.Since(start))
```

这会输出：

```
Spinning up 8 prime finders. Primes:
6410693
24941317
10128161
36122539
25511527
2107939
14004383
7190363
2393161
45931967
Search took: 5.438491216s
```

最大降幅23秒，这简直是个壮举。运用扇入扇出可以在不大幅改变程序结构的前提下将运行时间缩短了大约78%。

or-done-channel

有时你会与来自系统不同部分的通道交互。与管道不同的是，当你使用的代码通过done通道取消操作时，你无法对通道的行为方式做出判断。也就是说，你不知道正在执行读取操作的goroutine现在是什么状态。出于这个原因，正如我们在“防止Goroutine泄漏”中所阐述的那样，需要用select语句来封装我们的读取操作和done通道。可以简单的写成这样：

```
for val := range myChan {
    // 对 val 进行处理
}
```

展开后可以写成这样：

```
loop:
    for {
        select {
        case <-done:
            break loop
        case maybeVal, ok := <-myChan:
            if ok == false {
                return // or maybe break from for
            }
            // Do something with val
        }
    }
```

这样做可以快速退出嵌套循环。继续使用goroutines编写更清晰的并发代码，而不是过早优化的主题，我们可以用一个goroutine来解决这个问题。 我们封装了细节，以便其他人调用更方便：

```
orDone := func(done, c <-chan interface{}) <-chan interface{} {

    valStream := make(chan interface{})
    go func() {
        defer close(valStream)
        for {
            select {
            case <-done:
                return
            case v, ok := <-c:
                if ok == false {
                    return
                }
                select {
                case valStream <- v:
                case <-done:
                }
            }
        }
    }()

    return valStream
}
```

这样做允许我们回到简单的循环方式：

```
for val := range orDone(done, myChan) {
    // Do something with val
}
```

你可能会在代码中发现需要使用一系列select语句的循环代码，但我会鼓励你先尝试提高可读性，并避免过早优化。

tee-channel

有时候你可能想分割来自通道的多个值，以便将它们发送到两个独立区域。想象一下：你可能想要在一个通道上接收一系列操作指令，将它们发送给执行者，同时记录操作日志。

与Unix系统的tee命令功能类似，我们用tee-channel来实现同样的功能。你可以传递给它一个用作读取的通道，它会返回两个单独的通道：

```
tee := func(
    done <-chan interface{},
    in <-chan interface{},
) (_, _ <-chan interface{}) { <-chan interface{}) {

    out1 := make(chan interface{})
    out2 := make(chan interface{})

    go func() {

        defer close(out1)
        defer close(out2)

        for val := range orDone(done, in) {
            var out1, out2 = out1, out2 //1
            for i := 0; i < 2; i++ { //2
                select {
                case <-done:
                case out1 <- val:
                    out1 = nil //3
                case out2 <- val:
                    out2 = nil //3
                }
            }
        }
    }()

    return out1, out2
}
```

注意：原文例子就是这样，反复确认没有贴错。大家就当伪码看吧

1. 我们希望使用使用本地的变量，所以建立了他们的副本。
2. 我们将使用一条select语句，以便写入out1和out2不会彼此阻塞。 为了确保两者都顺利写入，我们将执行select语句的两个迭代。
3. 一旦我们写入了通道，我们将其副本设置为零，这样继续写入将阻塞，而另一个通道可以继续执行。

注意写入out1和out2是紧密耦合的。 直到out1和out2都被写入，迭代才能继续。 通常这不是问题，因为无论如何，处理来自每个通道的读取流程的吞吐量应该是tee之外的关注点，但值得注意。 这是一个快速调用示例：

```
done := make(chan interface{})
defer close(done)

out1, out2 := tee(done, take(done, repeat(done, 1, 2), 4))

for val1 := range out1 {
    fmt.Printf("out1: %v, out2: %v\n", val1, <-out2)
}
```

利用这种模式，很容易使用通道作为系统数据的连接点。

bridge-channel

在某些情况下，你可能会发现自己想要使用一系列通道的值：

```
<-chan <-chan interface{}
```

这与将某个通道的数据切片合并到一个通道中稍有不同，这种调用方式意味着一系列通道有序的写入操作。这与管道的单个“阶段”类似，其生命周期是间歇性的。按“访问范围约束”章节所提到的，通道由写入它们的goroutine所拥有，每当在新的goroutine中启动一个管道的“阶段”时，就会创建一个新的通道——这意味着我们会得到一个通道队列。我们会在第五章“Goroutines异常行为修复”中详细讨论。

作为消费者，代码可能不关心其值来自于一系列通道的事实。在这种情况下，处理一系列通道中的单个通道可能很麻烦。如果我们定义一个函数，可以将一系列通道拆解为一个简单的通道——我们成为通道桥接(bridge-channle)，这使得消费者更容易关注手头的问题：

```
bridge := func(done <-chan interface{}, chanStream <-chan <-chan interface{}) <-chan interface{} {

    valStream := make(chan interface{}) // 1
    go func() {
        defer close(valStream)
        for { // 2
            var stream <-chan interface{}
            select {
            case maybeStream, ok := <-chanStream:
                if ok == false {
                    return
                }
                stream = maybeStream
            case <-done:
                return
            }
            for val := range orDone(done, stream) { // 3
                select {
                case valStream <- val:
                case <-done:
                }
            }
        }
    }()
    return valStream
}
```

1. 这个通道会返回所有传入bridge的通道。
2. 该循环负责从chanStream中提取通道并将其提供给嵌套循环以供使用。
3. 该循环负责读取已经给出的通道的值，并将这些值重复到valStream中。当前正在循环的流关闭时，我们跳出执行从该通道读取的循环，并继续下一次循环来选择要读取的通道。 这为我们提供了一个不间断的流。

这段代码非常直白。接下来我们来使用它。下面这个例子创建了10个通道，每个通道都写入一个元素，并将这些通道传递给bridge：

```
genVals := func() <-chan <-chan interface{} {

    chanStream := make(chan (<-chan interface{}))

    go func() {
        defer close(chanStream)
```

```
        for i := 0; i < 10; i++ {
            stream := make(chan interface{}, 1)
            stream <- i
            close(stream)
            chanStream <- stream
        }
    }()
    return chanStream
}

for v := range bridge(nil, genVals()) {
    fmt.Printf("%v ", v)
}
```

这会输出：

```
0 1 2 3 4 5 6 7 8 9
```

通过使用bridge，我们可以专注于解构之外的逻辑，而无需去关心大量的通道处理问题。

队列

我们在之前的章节列举了管道的各种优点，但有时候，尽管管道没有准备好，我们的程序依然还是要干活的，这种处理方式，被称为“队列”。

这意味着，一旦管道的某个阶段完成了工作，将其存储在内存中的临时位置，以便其他阶段可以稍后检索它，而你无需保持其引用。在“Channels”章节，我们讨论了带缓冲的通道，你可以把它视作队列的一种。

虽然在系统中引入队列功能非常有用，但它通常是优化程序时希望采用的最后一种技术之一。过早地添加队列会隐藏同步问题，例如死锁和活锁，并且，随着程序不断重构，你可能会发现需要更多或更少的队列。

那么使用队列有什么好处呢？队列通常用来尝试解决性能问题。队列几乎不会减少程序的总运行时间，它只会让程序的行为有所不同。

让我们看个简单的管道例子：

```
done := make(chan interface{})
defer close(done)

zeros := take(done, 3, repeat(done, 0))
short := sleep(done, 1*time.Second, zeros)
long := sleep(done, 4*time.Second, short)
pipeline := long
```

这个管道链共有4个阶段：

1. 间隔0s，不间断生成数据流。
2. 在接收到3条数据后取消前置操作。
3. 休眠1秒，短耗时阶段。
4. 休眠3秒，长耗时阶段。

我们假设阶段1和阶段2是即时的，那么需要关注的是休眠如何影响管道的运行时间。

Time(t)	i	Long stage	Short stage
0	0		1s
1	0	4s	1s
2	0	3s	(blocked)
3	0	2s	(blocked)
4	0	1s	(blocked)
5	1	4s	1s
6	1	3s	(blocked)
7	1	2s	(blocked)
8	1	1s	(blocked)
9	2	4s	(close)
10	2	3s	
11	2	2s	
12	2	1s	
13	3	(close)	

你可以看到，这个管道耗时13秒。短耗时阶段花费了大约9秒。

如果我们给管道加入缓存会怎么样？让我们试试在长短耗时阶段之间添加个缓冲：

```
done := make(chan interface{})
defer close(done)

zeros := take(done, 3, repeat(done, 0))
short := sleep(done, 1*time.Second, zeros)
buffer := buffer(done, 2, short)     // Buffers sends from short by 2
long := sleep(done, 4*time.Second, short)
pipeline := long
```

Time(t)	i	Long stage	Buffer	Short stage
0	0		0/2	1s
1	0	4s	0/2	1s
2	0	3s	1/2	1s
3	0	2s	2/2	(close)
4	0	1s	2/2	
5	1	4s	1/2	
6	1	3s	1/2	
7	1	2s	1/2	
8	1	1s	1/2	

Time(t)	i	Long stage	Buffer	Short stage
9	2	4s	0/2	
10	2	3s	0/2	
11	2	2s	0/2	
12	2	1s	0/2	
13	3	(close)		

整个管道依然是13秒，但短耗时阶段时长降低到了3秒，看来加入缓存是有效的。但是如果整个管道仍然需要13秒来执行，这对我们有什么帮助？

我们来看看下面这个操作：

```
p := processRequest(done, acceptConnection(done, httpHandler))
```

这条管道会持续运行直到被取消，并且在取消之前会持续接受连接。在这期间，你肯定不希望处理连接的processRequest因acceptConnection接受连接而阻塞，你会希望processRequest是持续可用的，否则程序的用户可能会发现连接请求被拒绝。

因此，队列的价值并不是减少了某个阶段的运行时间，而是减少了它处于阻塞状态的时间。 这可以让程序继续工作。 在这个例子中，用户可能会在他们的请求中感受到延迟，但不会被拒绝服务。

通过这种方式，队列的真正用途是将操作流程分离，以便一个阶段的运行时间不会影响另一个阶段的运行时间。以这种方式解耦来改变整个系统的运行时行为，这取决于你的程序，产生的结果可能是好的也可能是不好的。

接下来我们回到关于队列的讨论。 队列应该放在哪里？ 缓冲区大小应该是多少？ 这些问题的答案取决于管道的性质。

我们首先分析队列提高系统整体性能适用于哪些情况：

- 如果某个阶段执行批处理能够节省时间。
- 如果推迟某个阶段产生结果可以在程序中循环执行。

适用于第一种情况的一个例子是，将输入缓冲到内存而非硬盘中。实际上bufio包就是这么干的。下面这个例子比较了使用缓冲与非缓冲进行写操作：

```
func BenchmarkUnbufferedWrite(b *testing.B) {
    performWrite(b, tmpFileOrFatal())
}

func BenchmarkBufferedWrite(b *testing.B) {
    bufferredFile := bufio.NewWriter(tmpFileOrFatal())
    performWrite(b, bufio.NewWriter(bufferredFile))
}

func tmpFileOrFatal() *os.File {
    file, err := ioutil.TempFile("", "tmp")
    if err != nil {
        log.Fatal("error: %v", err)
    }
    return file
}

func performWrite(b *testing.B, writer io.Writer) {
    done := make(chan interface{})
    defer close(done)

    b.ResetTimer()
    for bt := range take(done, repeat(done, byte(0)), b.N) {
        writer.Write([]byte{bt.(byte)})
    }
}
```

执行命令行：

```
go test -bench=. src/concurrency-patterns-in-go/queuing/buffering_test.go
```

这会输出：

```
BenchmarkUnbufferedWrite-8      500000      3969 ns/op
BenchmarkBufferedWrite-8        1000000     1356 ns/op
PASS
ok      command-line-arguments      3.398s
```

如预期的那样，有缓冲的写入比无缓冲更快。这是因为在bufio.Writer中，写入操作在内部缓冲区中进入队列，直到已经积累了足够长的数据块，该块才被写出。这个过程通常称为分块。

分块速度更快，因为bytes.Buffer必须增加其分配的内存以容纳存储的字节数据。出于各种原因，内存扩张操作代价高昂； 因此，我们需要增长的时间越少，整个系统的整体效率就越高。 于是，队列提高了整个系统的性能。

这只是一个简单的内存分块示例，但是你可能会频繁地进行分块。通常，执行任何操作都存在开销，分块可能会提高系统性能。例如打开数据库事务，计算消息校验和以及分配内存连续空间。

除了分块之外，如果程序算法支持向后查找或排序优化，队列也可以起到帮助作用。

第二种情况，某个阶段的延迟执行导致更多的数据进入管道，但没有被发现，这更加致命，因为它可能导致上游系统的崩溃。

这个想法通常被称为负反馈循环，甚至是死亡螺旋。 这是因为管道与上游系统之间存在经常性关系；上游系统提交新请求的速度在某种程度上与管道的有效性有关。

如果管道的效率降低到某个临界阈值以下，则管道上游的系统开始增加其对管道的输入，这导致管道损失更多效率，并且死亡螺旋开始。 如果没有安全防护，该系统将无法恢复。

通过在管道入口处引入队列，你可以延迟请求来打破反馈循环。从调用者的角度来看，请求似乎正在处理中，但需要很长时间。只要调用者不超时，管道将保持稳定。如果调用方超时，则需要确保你在出列时支持安全检查。如果不这样做，可能会无意中通过处理无效请求创建了另一个负馈循环，从而降低管道的效率。

> 如果你曾尝试过一些热门的新系统（例如，新游戏服务器，用于产品发布的网站等），并且尽管开发人员尽了最大的努力，但该网站一直处于不稳定状态，恭喜！你可能目睹了一个负反馈循环。
> 开发团队总是在尝试不同的解决方案，直到有人意识到他们需要一个队列，并且匆忙地将其实现。
> 然后客户开始抱怨排队时间。

从以上的例子中，我们可以看到一种模式慢慢浮出水面，队列应该满足以下情况：

- 在管道的入口处。
- 在某个阶段进行批处理会更高效。

您可能会试图在其他地方添加队列，例如，某个阶段会执行密集计算。要避免这种诱惑！正如我们所知道的那样，只有少数情况下，对立会减少管道的运行时间。为排除干扰而尝试队列会产生灾难性后果。

为了理解为什么，我们必须讨论管道的吞吐量。别担心，这并不困难，这也将帮助我们回答关于如何确定队列应该多大的问题。

在队列理论中，有一条定律（进行足够的抽样）可以预测管道的吞吐量。这就是所谓的”最小原则”，你只需要知道几点就可以理解和利用它。

我们以代数的方式定义“最小原则”，它通常表示为：L = λW，其中

- L = 系统中的平均单位数。
- λ = 单位的平均到达率。
- W = 单位在系统中花费的平均时间。

这个等式仅适用于所谓的稳定系统。 在一条管道中，一个稳定的系统就是数据进入管道或入口的速率等于它退出系统或出口的速率。 如果进入速率超过出口速度，那么你的系统就不稳定，并且已经进入死亡螺旋。 如果入口速率小于出口速率，则系统仍然不稳定，因为你的资源没有被完全利用。这不是世界上最糟糕的情况，但是如果发现资源利用严重不足（例如，集群或数据中心），也许你会关心这一点。

假设我们的管道是稳定的。如果我们想要减少单位花费在系统中的平均时间n，只有一个选择：减少系统中平均单位数：L/n = λW / n。 如果提高出口率，我们只能减少系统中的平均单位数量。还要注意，如果我们将队列添加到阶段，我们增加L，这会增加单位的到达率（nL = nλ* W）或增加单位在系统中的平均时间（nL = λ* nW）。通过最小原则，我们可以证明，队列对于减少系统花费的时间帮助不大。

同时请注意，由于我们正在观察整个管道，因此将W减少n倍将分布在我们管道的所有阶段。在我们的案例中，最小原则应该是这样定义的：

```
L = λΣiWi
```

不分青红皂白的优化，可能导致你的管道完全被最慢的执行阶段影响。

这个原则可以帮助我们分析管道的各个阶段。假设我们的管道有三个阶段。

让我们尝试确定管道每秒可以处理多少个请求。假设我们在管道上启用了采样，发现1个请求（r）需要约1秒才能通过管道。让我们向公式放入这些数字：

```
3r = λr/s * 1s
3r/s = λr/s
```

```
λ r/s = 3r/s
```

我们将L设置为3，因为我们管道中的每个阶段都在处理请求。然后我们将W设置为1秒，做一个小代数计算，瞧！在这个管道中，我们每秒可以处理三个请求。

假设采样表明请求需要1 ms来处理。 我们的队列需要处理每秒100000次请求的大小是多少？

```
Lr-3r = 100,000r/s * 0.0001s
Lr-3r = 10r
Lr = 7r
```

我们的管道有三个阶段，所以我们将L递减3。将λ设置为100000 r/s，我们发现如果想要处理很多请求，我们的队列应该有7个容量。请记住， 如果增加队列的大小，它需要更长的时间才能完成！ 你实际上在延迟降低系统利用率。

但这个公式也存在缺陷，它无法观察对失败的处理。请记住，如果由于某种原因管道发生混乱，你将丢失队列中的所有请求。为了缓解这种情况，你可以将队列大小保持为零，也可以将其移至持久队列中，该队列是一个持续存在的队列，可以在需要时再读取。

队列在你的系统中可能很有用，但由于它的复杂性，它通常是我建议实现的最后优化手段之一。

context包

在并发程序中，由于连接超时，用户取消或系统故障，往往需要执行抢占操作。我们之前使用done通道来在程序中取消所有阻塞的并发操作，虽然取得了不错的效果，但同样也存在局限。

如果我们可以给取消通知添加额外的信息：例如取消原因，操作是否正常完成等，这对我们进一步处理会起到非常大的作用。

在社区的不断推动下，Go开发组决定创建一个标准模式，以应对这种需求。在Go 1.7中，context包被引入标准库。

如果我们浏览一下context包，会发现其包含的内容非常少：

```
var Canceled = errors.New("context canceled")
var Canceled = errors.New("context canceled")

type CancelFunc
type Context

func Background() Context
func TODO() Context
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)
func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)
func WithValue(parent Context, key, val interface{}) Context
```

我们稍后会讨论这些类型和函数，现在让我们把关注点放到Context类型上。这个类型会贯穿你的整个系统，就跟done通道一样。如果你使用context包，从上游衍生出来的每个下游函数都可以使用Context作为参数。其类型定义是这样的：

```
type Context interface {
        // Deadline 返回任务完成时（该 context 被取消）的时间。
        // 如果deadline 未设置，则返回的ok值为false。
        // 连续调用该函数将返回相同的结果。
        Deadline() (deadline time.Time, ok bool)

        // Done 返回任务完成时（该 context 被取消）一个已关闭的通道。
        // 如果该context无法被取消，Done 将返回nil。
        // 连续调用该函数将返回相同的结果。
        //
        // 当cancel被调用时，WithCancel 遍历 Done以执行关闭；
        // 当deadline即将到期时，WithDeadline 遍历 Done以执行关闭；
        // 当timeout时，WithTimeout 遍历 Done以执行关闭。
        //
        // Done 主要被用于 select 语句：
        //
        //  // Stream 使用DoSomething生成值，并将值发送出去
        //  // 直到 DoSomething 返回错误或 ctx.Done 被关闭
        //  func Stream(ctx context.Context, out chan<- Value) error {
        //      for {
        //          v, err := DoSomething(ctx)
        //          if err != nil {
        //              return err
        //          }
        //          select {
        //          case <-ctx.Done():
        //              return ctx.Err()
        //          case out <- v:
        //          }
        //      }
        //  }
```

```
        //
        // 查看 https://blog.golang.org/pipelines更多示例以了解如何使用
        // Done通道执行取消操作。
        Done() <-chan struct{}

        // 如果 Done 尚未关闭, Err 返回 nil.
        // 如果 Done 已关闭, Err 返回值不为nil的error以解释为何关闭:
        // 因 context 的关闭导致
        // 或 context 的 deadline 执行导致。
        // 在 Err 返回值不为nil的error之后, 连续调用该函数将返回相同的结果。
        Err() error

        // Value 根据 key 返回与 context 相关的结果,
        // 如果没有与key对应的结果，则返回nil。
        // 连续调用该函数将返回相同的结果。
        //
        // 该方法仅用于传输进程和API边界的请求数据,
        // 不可用于将可选参数传递给函数。
        //
        // 键标识着上Context中的特定值。
        // 在Context中存储值的函数通常在全局变量中分配一个键,
        // 然后使用该键作为context.WithValue和Context.Value的参数。
        // 键可以是系统支持的任何类型;
        // 程序中各包应将键定义为未导出类型以避免冲突。
        //
        // 定义Context键的程序包应该为使用该键存储的值提供类型安全的访问器:
        //
        //     // user包 定义了一个User类型，该类型存储在Context中。
        //     package user
        //
        //     import "context"
        //
        //     // User 类型的值会存储在 Context中。
        //     type User struct {...}
        //
        //     // key是位于包内的非导出类型。
        //     // 这可以防止与其他包中定义的键的冲突。
        //     type key int
        //
        //     // userKey 是user.User类型的值存储在Contexts中的键。
        //     // 它是非导出的;  clients use user.NewContext and user.FromContext
        //     // 使用 user.NewContext 和 user.FromContext来替代直接使用键。
        //     var userKey key
        //
        //     // NewContext 返回一个新的含有值 u 的 Context。
        //     func NewContext(ctx context.Context, u *User) context.Context {
        //         return context.WithValue(ctx, userKey, u)
        //     }
        //
        //     // FromContext 返回存储在 ctx中的 User类型的值(如果存在的话)。
        //     func FromContext(ctx context.Context) (*User, bool) {
        //         u, ok := ctx.Value(userKey).(*User)
        //         return u, ok
        //     }
        Value(key interface{}) interface{}
```

这看起来挺简单。有一个Done方法返回当我们的函数被抢占时关闭的通道。还有一些新鲜的但不难理解的方法：一个Deadline函数，用于指示在一定时间之后goroutine是否会被取消，以及一个Err方法，如果goroutine被取消，将返回非零值。 但Value方法看起来有点奇怪。它是干嘛用的？

goroutines的主要用途之一是为请求提供服务。通常在这些程序中，除了抢占信息之外，还需要传递特定于请求的信息。这是Value函数的意义。我们会稍微谈一谈这个问题，但现在我们只需要知道context包有两个主要目的：

- 提供取消操作。
- 提供用于通过调用传输请求附加数据的数据包。

让我们看看第一个目的：取消操作。

正如我们在“防止Goroutine泄漏”中所学到的，函数中的取消有三个方面：

- goroutine的生成者可能想要取消它。
- goroutine可能需要取消其衍生出来的goroutine。
- goroutine中的任何阻塞操作都必须是可抢占的，以便将其取消。

Context包可以帮助我们处理这三个方面的需求。

前面提到，Context类型将是函数的第一个参数。如果你查看了Context接口的方法，会发现没有任何东西可以改变底层结果的状态。更进一步的说，没有任何东西被系统允许把Context本身干掉。这保护了Context调用堆栈的功能。因此，结合接口中的Done方法，Context类型可以安全的管理取消操作。

这就产生了一个问题：如果Context是一成不变的，那我们如何影响调用堆栈中当前函数的子函数中的取消行为？

context包提供的一些函数回答了这个问题：

```
func WithCancel(parent Context) (ctx Context, cancel CancelFunc)
func WithDeadline(parent Context, deadline time.Time) (Context, CancelFunc)
func WithTimeout(parent Context, timeout time.Duration) (Context, CancelFunc)
```

你会发现，这些函数都传入了Context类型的值，同时返回了Context类型的值，它们都使用与这些函数相关的选项生成Context的新实例。

WithCancel返回一个新的Context，它在调用返回的cancel函数时关闭done通道。
WithDeadline返回一个新的Context，当机器的时钟超过给定的最后期限时，它关闭done通道。
WithTimeout返回一个新的Context，它在给定的超时时间后关闭done通道。

如果你的函数需要以某种方式在调用中取消它的子函数，可以调用这三个函数中的一个并传递给它的上下文，然后将返回的上下文传递给它的子函数。 如果你的函数不需要修改取消行为，那么函数只传递给定的上下文。

通过这种方式，调用者可以创建符合其需求的上下文，而不会影响其创建者。这为管理调用分支提供了一个可组合的优雅的解决方案。

通过这样的方式，Context的实例可以贯穿你的整个程序。在面向对象的范例中，通常将对经常使用的数据的引用存储为成员变量，但重要的是不要使用context.Context的实例来执行此操作。context.Context的实例可能与外部看起来相同，但在内部它们可能会在每个堆栈帧处发生变化。出于这个原因，总是将Context的实例传递给你的函数是很重要的。通过这种方式，函数具有用于它的上下文，而不是把堆栈里的上下文随意取出来用。

在异步调用链的顶部，你的代码可能不会传递Context。要启动链，context包提供了两个函数来创建Context的空实例。

我们来看一个使用done通道模式的例子，并比较下切换到使用context文包获得什么好处。 这是一个同时打印问候和告别的程序：

```
func main() {
    var wg sync.WaitGroup
    done := make(chan interface{})
    defer close(done)

    wg.Add(1)
```

```
    go func() {
        defer wg.Done()
        if err := printGreeting(done); err != nil {
            fmt.Printf("%v", err)
            return
        }
    }()

    wg.Add(1)
    go func() {
        defer wg.Done()
        if err := printFarewell(done); err != nil {
            fmt.Printf("%v", err)
            return
        }
    }()

    wg.Wait()
}

func printGreeting(done <-chan interface{}) error {
    greeting, err := genGreeting(done)
    if err != nil {
        return err
    }
    fmt.Printf("%s world!\n", greeting)
    return nil
}

func printFarewell(done <-chan interface{}) error {
    farewell, err := genFarewell(done)
    if err != nil {
        return err
    }
    fmt.Printf("%s world!\n", farewell)
    return nil
}

func genGreeting(done <-chan interface{}) (string, error) {
    switch locale, err := locale(done); {
    case err != nil:
        return "", err
    case locale == "EN/US":
        return "hello", nil
    }
    return "", fmt.Errorf("unsupported locale")
}

func genFarewell(done <-chan interface{}) (string, error) {
    switch locale, err := locale(done); {
    case err != nil:
        return "", err
    case locale == "EN/US":
        return "goodbye", nil
    }
    return "", fmt.Errorf("unsupported locale")
}

func locale(done <-chan interface{}) (string, error) {
    select {
    case <-done:
        return "", fmt.Errorf("canceled")
```

```
    case <-time.After(5 * time.Second):
    }
    return "EN/US", nil
}
```

这会输出：

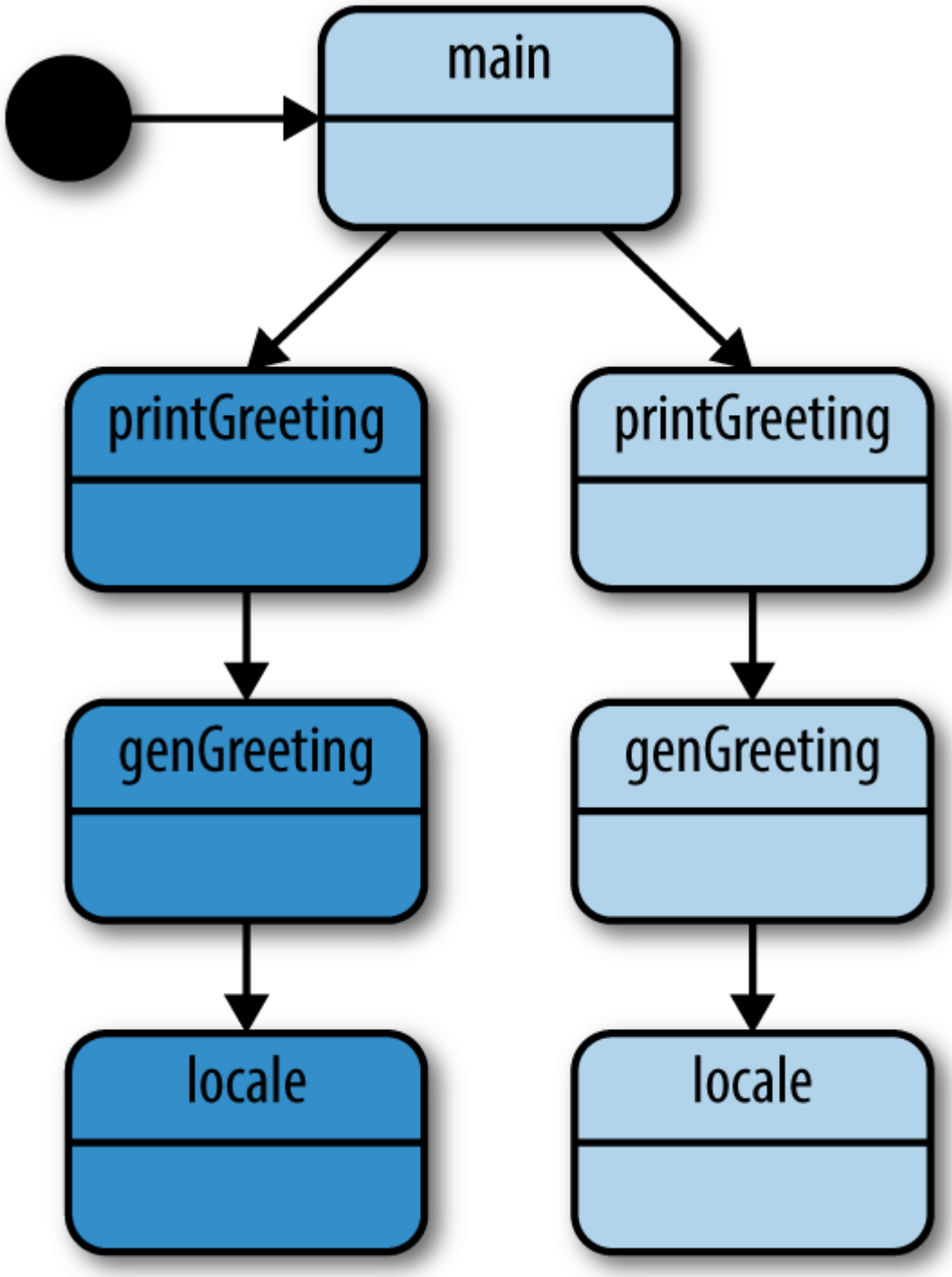
```
hello world!
goodbye world!
```

忽略竞争条件，我们可以看到程序有两个分支同时运行。通过创建done通道并将其传递给我们的调用链来设置标准抢占方法。如果我们在main的任何一点关闭done频道，那么两个分支都将被取消。

我们可以尝试几种不同且有趣的方式来控制该程序。也许我们希望genGreeting如果花费太长时间就会超时。也许我们不希望genFarewell调用locale——在其父进程很快就会被取消的情况下。在每个堆栈框架中，一个函数可以影响其下的整个调用堆栈。

使用done通道模式，我们可以通过将传入的done通道包装到其他done通道中，然后在其中任何一个通道启动时返回，但我们不会获得上下文给的deadline和错误的额外信息。

为了将done通道模式与使用context包进行比较，我们将该程序表示为树状图。 树中的每个节点代表一个函数的调用。



让我们使用context包来修改该程序。由于现在可以使用context.Context的灵活性，所以我们引入一个有趣的场景。

假设genGreeting在放弃调用locale之前等待一秒——超时时间为1秒。如果printGreeting不成功，我们想取消对printFare的调用。 毕竟，如果我们不打声招呼，说再见就没有意义了：

```
func main() {
    var wg sync.WaitGroup
    ctx, cancel := context.WithCancel(context.Background()) //1
    defer cancel()
    wg.Add(1)
    go func() {
        defer wg.Done()
```

```
        if err := printGreeting(ctx); err != nil {
            fmt.Printf("cannot print greeting: %v\n", err)
            cancel() //2
        }
    }()

    wg.Add(1)
    go func() {
        defer wg.Done()
        if err := printFarewell(ctx); err != nil {
            fmt.Printf("cannot print farewell: %v\n", err)
        }
    }()

    wg.Wait()
}

func printGreeting(ctx context.Context) error {
    greeting, err := genGreeting(ctx)
    if err != nil {
        return err
    }
    fmt.Printf("%s world!\n", greeting)
    return nil
}

func printFarewell(ctx context.Context) error {
    farewell, err := genFarewell(ctx)
    if err != nil {
        return err
    }
    fmt.Printf("%s world!\n", farewell)
    return nil
}

func genGreeting(ctx context.Context) (string, error) {
    ctx, cancel := context.WithTimeout(ctx, 1*time.Second) //3
    defer cancel()

    switch locale, err := locale(ctx); {
    case err != nil:
        return "", err
    case locale == "EN/US":
        return "hello", nil
    }
    return "", fmt.Errorf("unsupported locale")
}

func genFarewell(ctx context.Context) (string, error) {
    switch locale, err := locale(ctx); {
    case err != nil:
        return "", err
    case locale == "EN/US":
        return "goodbye", nil
    }
    return "", fmt.Errorf("unsupported locale")
}

func locale(ctx context.Context) (string, error) {
    select {
    case <-ctx.Done():
```

```
        return "", ctx.Err() //4
    case <-time.After(1 * time.Minute):
    }
    return "EN/US", nil
}
```

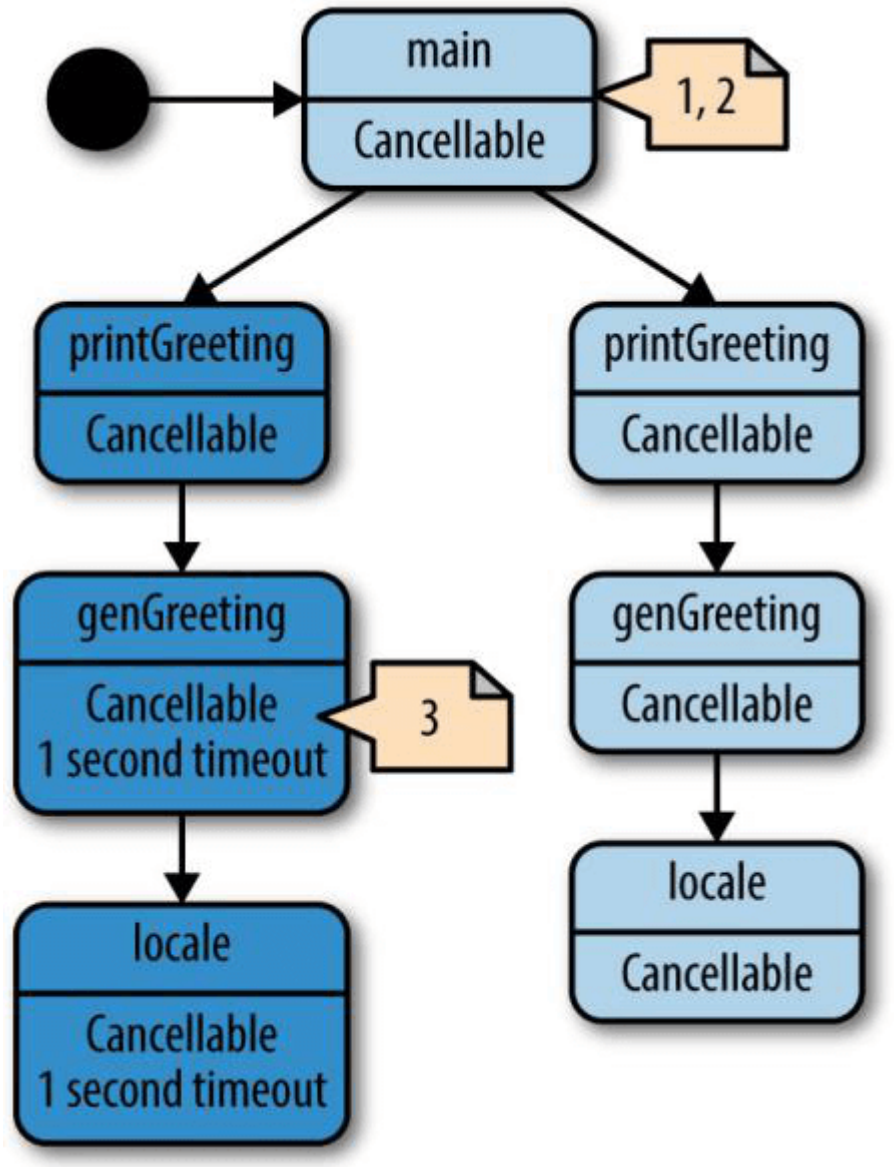
1. 在main函数中使用context.Background()建立个新的Context，并使用context.WithCancel将其包裹以便对其执行取消操作。
2. 在这一行上，如果从 printGreeting返回错误，main将取消context。
3. 这里genGreeting用context.WithTimeout包装Context。这将在1秒后自动取消返回的context，从而取消它传递context的子进程，即语言环境。

这一行返回为什么Context被取消的原因。 这个错误会一直冒泡到main，这会导致注释2处的取消操作被调用。

这会输出：

```
cannot print greeting: context deadline exceeded
cannot print farewell: context canceled
```

下面的图中数字对应例子中的代码标注。



我们可以看到系统输出工作正常。由于local设置至少需要运行一分钟，因此genGreeting将始终超时，这意味着main会始终取消printFarewell下面的调用链。

请注意，genGreeting如何构建自定义的Context.Context以满足其需求，而不必影响父级的Context。如果genGreeting成功返回，并且printGreeting需要再次调用，则可以在不泄漏genGreeting相关操作信息的情况下进行。这种可组合性使你能够编写大型系统，而无需在整个调用链中费劲心思解决这样的问题。

我们可以在这个程序上进一步改进：因为我们知道locale需要大约一分钟的时间才能运行，所以可以在locale中检查是否给出了deadline。下面这个例子演示了如何使用context.Context的Deadline方法：

```
func main() {
    var wg sync.WaitGroup
    ctx, cancel := context.WithCancel(context.Background())
    defer cancel()

    wg.Add(1)
    go func() {
        defer wg.Done()

        if err := printGreeting(ctx); err != nil {
```

```
            fmt.Printf("cannot print greeting: %v\n", err)
            cancel()
        }
    }()
    wg.Add(1)
    go func() {
        defer wg.Done()
        if err := printFarewell(ctx); err != nil {
            fmt.Printf("cannot print farewell: %v\n", err)
        }
    }()

    wg.Wait()
}

func printGreeting(ctx context.Context) error {
    greeting, err := genGreeting(ctx)
    if err != nil {
        return err
    }
    fmt.Printf("%s world!\n", greeting)
    return nil
}

func printFarewell(ctx context.Context) error {
    farewell, err := genFarewell(ctx)
    if err != nil {
        return err
    }
    fmt.Printf("%s world!\n", farewell)
    return nil
}

func genGreeting(ctx context.Context) (string, error) {
    ctx, cancel := context.WithTimeout(ctx, 1*time.Second)
    defer cancel()

    switch locale, err := locale(ctx); {
    case err != nil:
        return "", err
    case locale == "EN/US":
        return "hello", nil
    }
    return "", fmt.Errorf("unsupported locale")
}

func genFarewell(ctx context.Context) (string, error) {
    switch locale, err := locale(ctx); {
    case err != nil:
        return "", err
    case locale == "EN/US":
        return "goodbye", nil
    }
    return "", fmt.Errorf("unsupported locale")
}

func locale(ctx context.Context) (string, error) {
    if deadline, ok := ctx.Deadline(); ok { //1
        if deadline.Sub(time.Now().Add(1*time.Minute)) <= 0 {
            return "", context.DeadlineExceeded
        }
    }
```

```
    select {
    case <-ctx.Done():
        return "", ctx.Err()
    case <-time.After(1 * time.Minute):
    }
    return "EN/US", nil
}
```

1. 我们在这里检查是否Context提供了deadline。如果提供了，而且我们的程序时间已经越过这个时间线，这时简单的返回一个context包预设的错误——DeadlineExceeded。

虽然修改的部分很小，但它允许locale函数快速失败，而不必像之前那样等待一分钟。在调用资源耗费较高的程序中，这样做会节省大量时间。唯一的问题是，你得考虑deadline设置多久合适——这需要不断的尝试。

接下来我们讨论context包的另一个用处：存储和检索附加于请求的数据包。请记住，当一个函数创建一个goroutine和Context时，它通常会启动一个为请求提供服务的进程，并且子函数可能需要相关的请求信息。下面是一个示例：

```
func main() {
    ProcessRequest("jane", "abc123")
}

func ProcessRequest(userID, authToken string) {
    ctx := context.WithValue(context.Background(), "userID", userID)
    ctx = context.WithValue(ctx, "authToken", authToken)
    HandleResponse(ctx)
}

func HandleResponse(ctx context.Context) {
    fmt.Printf("handling response for %v (%v)",
        ctx.Value("userID"),
        ctx.Value("authToken"),
    )
}
```

这会输出：

```
handling response for jane (abc123)
```

很简单的用法。不过也是有限制的：

- 你使用的key必须在Go中是可比较的，也就是说，== 和 != 必须能返回正确的结果。
- 返回值必须是并发安全的，这样才能从多个goroutine访问。

由于Context的键和值都被定义为interface{}，所以当试图检索值时，我们会失去其类型安全性。基于此，Go建议在context中存储和检索值时遵循一些规则。

首先，推荐你在包中自行定义key的类型，这样无论是否其他包执行相同的操作都可以防止context中的冲突。看下面这个例子：

```
type foo int
type bar int

m := make(map[interface{}]int)
m[foo(1)] = 1
m[bar(1)] = 2
```

```
fmt.Printf("%v", m)
```

这会输出：

```
map[1:2 1:1]
```

可以看到，虽然基础值是相同的，但不同类型的信息会在map中区分它们。由于你为包定义的key类型未导出，因此其他包不会与你在包中生成的key冲突。

由于用于存储数据的key是非导出的，因此我们必须导出执行检索数据的函数。这很容易做到，因为它允许这些数据的使用者使用静态的，类型安全的函数。

当你把所有这些放在一起时，你会得到类似下面的例子：

```
func main() {
    ProcessRequest("jane", "abc123")
}

type ctxKey int

const (
    ctxUserID    ctxKey = iota
    ctxAuthToken
)

func UserID(c context.Context) string {
    return c.Value(ctxUserID).(string)
}

func AuthToken(c context.Context) string {
    return c.Value(ctxAuthToken).(string)
}

func ProcessRequest(userID, authToken string) {
    ctx := context.WithValue(context.Background(), ctxUserID, userID)
    ctx = context.WithValue(ctx, ctxAuthToken, authToken)
    HandleResponse(ctx)
}

func HandleResponse(ctx context.Context) {
    fmt.Printf(
        "handling response for %v (auth: %v)",
        UserID(ctx),
        AuthToken(ctx),
    )
}
```

这会输出：

```
handling response for jane (auth: abc123)
```

在本例中，我们使用类型安全的方法来从Context获取值，如果消费者在不同的包中，他们不会知道或关心用于存储信息的key。 但是，这种技术会造成隐患。

在前面的例子中，我们假设HandleResponse存在于另一个名为response的包中，假设ProcessRequest包位于名为pross的包中。 pross包必须导入response包才能调用HandleResponse，但HandleResponse无法访问pross包中定义的访问函数，

因为导入会形成循环依赖关系。由于用于Context中存储的key类型对于process包来说是私有的，所以response包无法检索这些数据！

这迫使我们创建以从多个位置导入的数据类型为中心的包。 虽然这不是一件坏事，但它是需要注意的。

context包非常简洁，但依然褒贬不一。在Go社区中一直存在争议。该包的取消操作功能相当受欢迎，但是在Context中存储任意数据的能力以及存储数据的类型不安全的造成了一些分歧。 虽然我们已经部分减缓了访问函数缺乏类型安全性的问题，但是仍然可以通过存储不正确的类型来引入错误。然而，更大的问题在于，开发人员到底应该在Context的实例中存储什么样的数据。

在context包的文档中这样写到：

> 使用context存储值仅适用于传输进程和API的请求附加数据，而不用于将可选参数传递给函数。

该说明十分含糊，“传输进程和API的请求”实在太过宽泛。我认为最好的解读方法是与开发组一起提出一些约定，并在代码评审中检查它们：

1. 数据应该是由进程传递的或与API相关。如果你在进程的内存中生成数据，那么除非你也通过API传递数据，否则可能不是一个很好的候选应用程序。
   数据应该是不可变的。如果可变，那么根据定义，你存储的内容肯定不是来自请求。
2. **数据应指向系统简单类型。**我们在上面已经讨论了关于使用安全类型的包导入问题，这个结论是很明显的。
3. **数据应该是纯粹的数据，而不是某种类型的函数。**消费者的逻辑应该是消耗这些数据。
4. **数据应该有助于操作，而不是驱动操作。**如果你的算法根据context中包含或不包含的内容而有所不同，那么就违背了“不用于将可选参数传递”的初衷。

这些不是硬性规定，但如果你发现自己的程序与以上约定由冲突，则可能需要考虑是否存在隐患或使用context是否必要。

另一个需要考虑的方面是该数据在使用之前可能需要经过多少层。如果在接受数据的位置和使用位置之间有几个框架和几十个函数，你可以考虑使用日志，并将数据添加为参数；或者你更愿意将它放在Context中，从而创建一个不可见的依赖关系。每种方法都有优点，最终这由你和你的团队做出决定。

我将这5条约定做成了表格，你可以将之作为参考：

Data	1	2	3	4	5
Request ID	✓	✓	✓	✓	✓
User ID	✓	✓	✓	✓	
URL	✓	✓			
API Server Connection					
Authorization Token	✓	✓	✓	✓	
Request Token	✓	✓	✓		

对于是否有必要使用context存储值，这里并没有简单的答案，具体取决于你的业务、算法和团队。

我留给你的最后建议是Context提供的取消功能非常有用，这样轻便的功能如果不用实在是太可惜了。

小结

本章介绍了很多内容。 我们结合了Go的并发原语来介绍各种模式，以帮助编写可维护的并发代码。如果你熟悉了这些模式，接下来我们就来讨论如何将这些模式合并到其他模式中，以帮助你编写大型系统。下一章将会对这样的技术进行一个概述。

可伸缩并发设计

你已经学习了在Go中使用并发的一些常见模式，现在让我们将注意力集中在将这些模式组合成一系列实践，这些实践将使你能够编写可扩展的大型可组合系统。

在本章中，我们将讨论在单个进程中扩展并发操作的方法，并开始研究处理多个进程时如何发挥并发性。

错误传递

使用并发代码，特别是分布式系统，在系统中很容易出现问题，而且很难确认发生这种问题的原因。仔细考虑问题是如何通过系统传播的，以及如何最终呈现给用户，你会为自己，团队和用户减少很多痛苦。 在“错误处理”一节中，我们讨论了如何从goroutine处理错误，但我们没有花时间讨论这些错误应该是什么样子，或者错误应该如何流经一个庞大而复杂的系统。让我们花点时间来讨论错误传递的哲学。

许多开发人员认为错误传递是不值得关注的，或者，至少不是首先需要关注的。 Go试图通过强制开发者在调用堆栈中的每一帧处理错误来纠正这种不良做法。

首先让我们看看错误的定义。错误何时发生，以及错误会提供什么。

错误表明您的系统已进入无法完成用户明确或隐含请求的操作的状态。 因此，它需要传递一些关键信息：

发生了什么
这是错误的一部分，其中包含有关所发生事件的信息，例如“磁盘已满”，“套接字已关闭”或“凭证过期”。尽管生成错误的内容可能会隐式生成此信息，你可以用一些能够帮助用户的上下文来完善它。

何时何处发生
错误应始终包含一个完整的堆栈跟踪，从调用的启动方式开始，直到实例化错误。

此外，错误应该包含有关它正在运行的上下文的信息。 例如，在分布式系统中，它应该有一些方法来识别发生错误的机器。当试图了解系统中发生的情况时，这些信息将具有无法估量的价值。

另外，错误应该包含错误实例化的机器上的时间，以UTC表示。

有效的信息说明
显示给用户的消息应该进行自定义以适合你的系统及其用户。它只应包含前两点的简短和相关信息。 一个友好的信息是以人为中心的，给出一些关于这个问题的指示，并且应该是关于一行文本。

如何获取更详细的错误信息
在某个时刻，有人可能想详细了解发生错误时的系统状态。提供给用户的错误信息应该包含一个ID，该ID可以与相应的日志交叉引用，该日志显示错误的完整信息：发生错误的时间（不是错误记录的时间），堆栈跟踪——包括你在代码中自定义的信息。包含堆栈跟踪的哈希也是有帮助的，以帮助在bug跟踪器中汇总类似的问题。

默认情况下，没有人工干预，错误所能提供的信息少得可怜。 因此，我们可以认为：在没有详细信息的情况下传播给用户任何错误的行为都是错误的。因为我们可以使用搭建框架的思路来对待错误处理。可以将所有错误归纳为两个类别：

- Bug。
- 已知业务及系统意外（例如，网络连接断开，磁盘写入失败等）。

  Bug是你没有为系统定制的错误，或者是“原始”错误。有时这是故意的，如果在系统多次迭代时出现的错误，尽快不可避免的传递给了用户，但接受到用户反馈后对提高系统健壮性并不是坏处。有时这是偶然的。在确定如何传播错误，系统随着时间的推移如何增长以及最终向用户展示什么时，这种区别将证明是有用的。

想象下一个巨大的系统，包含了很多模块：

假设在“Low Level Component”中发生错误，并且我们已经制作了一个格式良好的错误，并传递给堆栈。 在“Low Level Component”的背景下，这个错误可能被认为是合理的，但在我们的系统中，它可能不是。 让我们看看在每个组件的边界处，所有传入的错误都必须包含在我们代码所在组件的格式错误中。 例如，如果我们处于“Intermediary Component”，并且我们从“Low Level Component”调用代码，这可能会出错，我们可以使用：

```
func PostReport(id string) error {
    result, err := lowlevel.DoWork()
    if err != nil {
        if _, ok := err.(lowlevel.Error); ok { //1
            err = WrapErr(err, "cannot post report with id %q", id) //2
        }
        // ...
    }
}
```

1. 我们在这里断言以确定是我们自定义的错误。如果不是，我们会简单的把err传递给堆栈表明这里发生的错误是个bug。
2. 在这里，我们使用函数将传入的错误与我们模块的相关信息进行封装，并给它一个新类型。请注意，包装错误可能隐藏一些底层细节。

在错误最初被实例化时，错误发生时的底层细节是存在于错误信息中的。在我们的示例中，模块的边界处我们将错误包装起来，不属于我们定义的错误类型的错误都被视为格式错误。请注意，实际工作中建议只以你自己的模块边界（公共函数/方法）或代码添加有价值的上下文时以这种方式包装错误。

采取这种立场可以让我们的系统有机地发展。 我们可以确定传入的错误是正确的，反过来可以确保考虑错误如何离开模块。错误正确性成为我们系统的一个新特性。通过这样做，我们给出了一个思想上的框架，通过呈现给用户的内容明确划分了错误的类型。

所有的错误都应该记录下尽可能多的信息。 但是，当向用户显示错误时，就需要尽可能的清晰明了。

当我们的代码发现到一个格式良好的错误时，我们可以确信，在代码中的所有级别上，都意识到了该错误的存在，而且已经将其记录下来并打印出来供用户查看。

当错误传播给用户时，我们记录错误，同时向用户显示一条友好的消息，指出发生了意外事件。如果我们的系统中支持自动错误报告，那是最好不过的事情。如果不支持，应当建议用户提交一个错误报告。请注意，任何微小的错误都会包含有用的信息，即使我们无法保证面面俱到。

请记住，在任何一种情况下，如果出现错误或格式错误，我们将在邮件中包含一个日志ID，以便在需要更多信息时可以参考。

我们来看一个完整的例子。 这个例子不会非常健壮（例如，错误类型可能是简单化的），并且调用堆栈是线性的，但不妨碍大家来理清思路：

```
type MyError struct {
    Inner      error
    Message    string
    StackTrace string
    Misc       map[string]interface{}
}

func wrapError(err error, messagef string, msgArgs ...interface{}) MyError {
    return MyError{
        Inner:      err, //1
        Message:    fmt.Sprintf(messagef, msgArgs...),
        StackTrace: string(debug.Stack()),        //2
        Misc:       make(map[string]interface{}), //3
    }
}

func (err MyError) Error() string {
    return err.Message
}
```

1. 在这里存储我们正在包装的错误。 如果需要调查发生的事情，我们总是希望能够查看到最低级别的错误。
2. 这行代码记录了创建错误时的堆栈跟踪。
3. 这里我们创建一个杂项信息存储字段。可以存储并发ID，堆栈跟踪的hash或可能有助于诊断错误的其他上下文信息。

接下来，我们建立一个名为 lowlevel 的模块：

```
// "lowlevel" module

type LowLevelErr struct {
    error
}

func isGloballyExec(path string) (bool, error) {
    info, err := os.Stat(path)
    if err != nil {
        return false, LowLevelErr{wrapError(err, err.Error())} // 1
    }
    return info.Mode().Perm()&0100 == 0100, nil
}
```

- 在这里，我们用自定义错误来封装os.Stat中的原始错误。在这种情况下，我们不会掩盖这个错误产生的信息。

然后我们建立另一个名为 intermediate 的模块，它会调用 lowlevel 所在的包：

```
// "intermediate" module

type IntermediateErr struct {
    error
}

func runJob(id string) error {
    const jobBinPath = "/bad/job/binary"
    isExecutable, err := isGloballyExec(jobBinPath)
    if err != nil {
        return err //1
    } else if isExecutable == false {
        return wrapError(nil, "job binary is not executable")
    }

    return exec.Command(jobBinPath, "--id="+id).Run() //1
}
```

1. 我们传递来自 lowlevel 模块的错误，由于我们接收从其他模块传递的错误而没有将它们包装在我们自己的错误类型中，这将会产生问题。

最后，让我们创建一个调用intermediate包函数的顶级main函数：

```
func handleError(key int, err error, message string) {
    log.SetPrefix(fmt.Sprintf("[logID: %v]: ", key))
    log.Printf("%#v", err) //3
    fmt.Printf("[%v] %v", key, message)
}

func main() {
    log.SetOutput(os.Stdout)
    log.SetFlags(log.Ltime | log.LUTC)

    err := runJob("1")
    if err != nil {
        msg := "There was an unexpected issue; please report this as a bug."
        if _, ok := err.(IntermediateErr); ok { //1
            msg = err.Error()
```

```
        }
        handleError(1, err, msg) //2
    }
}
```

1. 在这里我们检查是否错误是预期的类型。 如果是，可以简单地将其消息传递给用户。
2. 在这一行中，将日志和错误消息与ID绑定在一起。我们可以很容易增加这个增量，或者使用一个GUID来确保一个唯一的ID。
3. 在这里我们记录完整的错误，以备需要深入了解发生了什么。

我们在运行后会在日志中发现：

```
[logID: 1]: 21:46:07 main.LowLevelErr{error:main.MyError{Inner: (*os.PathError)(0xc4200123f0),
Message:"stat /bad/job/binary: no such file or directory", StackTrace:"goroutine 1 [running]: runtime/debug.S
tack(0xc420012420, 0x2f, 0xc420045d80)
/home/kate/.guix-profile/src/runtime/debug/stack.go:24 +0x79 main.wrapError(0x530200, 0xc4200123f0, 0xc420012
420, 0x2f, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, ...)
/tmp/babel-79540aE/go-src-7954NTK.go:22 +0x62 main.isGloballyExec(0x4d1313, 0xf, 0xc420045eb8, 0x487649, 0xc4
20056050)
/tmp/babel-79540aE/go-src-7954NTK.go:37 +0xaa main.runJob(0x4cfada, 0x1, 0x4d4c35, 0x22)
/tmp/babel-79540aE/go-src-7954NTK.go:47 +0x48 main.main()
/tmp/babel-79540aE/go-src-7954NTK.go:67 +0x63 ", Misc:map[string]interface {}{}}}
```

并且标准输出会打印：

```
[1]There was an unexpected issue; please report this as a bug.
```

我们可以看到，在这个错误路径的某处，它没有正确处理，并且因为我们无法确定错误信息是否适合用户自行处理，所以我们输出一个简单的错误信息，指出意外事件发生了。如果回顾 lowlevel 模块，我们会发现错误发生的原因：我们没有包装来自 lowlevel 模块的错误。让我们纠正它：

```
// "intermediate" module

type IntermediateErr struct {
    error
}

func runJob(id string) error {
    const jobBinPath = "/bad/job/binary"
    isExecutable, err := isGloballyExec(jobBinPath)
    if err != nil {
        return IntermediateErr{wrapError(err,
            "cannot run job %q: requisite binaries not available", id)} //1
    } else if isExecutable == false {
        return wrapError(
            nil,
            "cannot run job %q: requisite binaries are not executable", id,
        )
    }

    return exec.Command(jobBinPath, "--id="+id).Run()
}
```

1. 在这里，我们现在使用自定义错误。我们想隐藏工作未运行原因的底层细节，因为这对于用户并不重要。

```
func handleError(key int, err error, message string) {
    log.SetPrefix(fmt.Sprintf("[logID: %v]: ", key))
```

```
    log.Printf("%#v", err)
    fmt.Printf("[%v] %v", key, message)
}

func main() {
    log.SetOutput(os.Stdout)
    log.SetFlags(log.Ltime | log.LUTC)

    err := runJob("1")
    if err != nil {
        msg := "There was an unexpected issue; please report this as a bug."
        if _, ok := err.(IntermediateErr); ok {
            msg = err.Error()
        }
        handleError(1, err, msg)
    }
}
```

现在，当我们运行更新后的代码，会得到类似的日志：

```
[logID: 1]: 22:11:04 main.IntermediateErr{error:main.MyError
{Inner:main.LowLevelErr{error:main.MyError{Inner:(*os.PathError) (0xc4200123f0), Message:"stat /bad/job/binar
y: no such file or directory", StackTrace:"goroutine 1 [running]:
runtime/debug.Stack(0xc420012420, 0x2f, 0x0)
/home/kate/.guix-profile/src/runtime/debug/stack.go:24 +0x79 main.wrapError(0x530200, 0xc4200123f0, 0xc420012
420, 0x2f, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, ...)
/tmp/babel-79540aE/go -src-7954DTN.go:22 +0xbb main.isGloballyExec(0x4d1313, 0xf, 0x4daecc, 0x30, 0x4c5800)
/tmp/babel-79540aE/go -src-7954DTN.go:39 +0xc5 main.runJob(0x4cfada, 0x1, 0x4d4c19, 0x22)
/tmp/babel-79540aE/go-src-7954DTN.go:51 +0x4b
main.main()
/tmp/babel-79540aE/go -src-7954DTN.go:71 +0x63
", Misc:map[string]interface {}{}}, Message:"cannot run job \"1\": requisite binaries not available", StackT
race:"goroutine 1 [running]: runtime/debug.Stack(0x4d63f0, 0x33, 0xc420045e40)
/home/kate/.guix-profile/src/runtime/debug/stack.go:24 +0x79 main.wrapError(0x530380, 0xc42000a370, 0x4d63f0,
0x33, 0xc420045e40, 0x1, 0x1, 0x0, 0x0, 0x0, ...)
/tmp/babel-79540aE/go -src-7954DTN.go:22 +0xbb main.runJob(0x4cfada, 0x1, 0x4d4c19, 0x22)
/tmp/babel-79540aE/go -src-7954DTN.go:53 +0x356 main.main()
/tmp/babel-79540aE/go -src-7954DTN.go:71 +0x63 ", Misc:map[string]interface {}{}}
```

错误信息变得十分明白：

```
[1]cannot run job "1": requisite binaries not available
```

这种实现方法与标准库的错误包兼容，此外你可以用你喜欢的任何方式来进行包装，并且自由度非常大。

超时和取消

在编写并发代码时，超时和取消会频繁出现。我们将在本节中看到，超时对于创建一个可以健壮易读的程序至关重要。取消是对超时的回应。我们还将探讨在并发进程中引发取消的其他原因。

那么，我们为什么需要并发程序支持超时呢？

系统饱和
正如我们在“队列”部分所讨论的那样，如果系统已经达到最大负荷（即，它的处理请求的能力达到了极限），我们可能希望系统的请求超时而不是花很长时间等待。你选择哪条路线取决于你的实际业务，但这里有一些关于何时触发超时的一般性指导：

- 如果请求在超时情况下不太可能重复发送。
- 如果没有资源来存储请求（例如，临时队列的内存，持久队列的磁盘空间）。
- 如果请求或其发送的数据的过期（我们将在下面讨论）。 如果一个请求可能会重复发生，那么系统将会接受并超时请求。如果开销超过我们系统的容量，这可能导致死亡螺旋。 但是，如果我们缺乏将请求存储在队列中所需的系统资源，这是一个有争议的问题。 即使我们符合这两条准则，将该请求加入队列也没有什么意义，只要我们可以处理请求，请求就会过期。 这给我们带来了超时的下一个理由。

数据过期
有时数据有一个窗口，在这个窗口中必须优先处理部分相关数据，或者处理数据的需求已过期。如果一个并发进程比这个窗口花费更长的时间来处理数据，我们希望超时并取消该进程。例如，如果某个并发进程在长时间等待后发起请求，则在队列中请求或其数据可能已过期。
如果这个窗口已经事先知晓，那么将context.WithDeadline或context.WithTimeout创建的context.Context传递给我们的并发进程是有意义的。 如果不是，我们希望并发进程的父节点能够在需求不再需要时取消并发进程。 context.WithCancel完美适用于此目的。

防止死锁
在大型系统中，尤其是分布式系统中，有时难以理解数据流动的方式或系统边界可能出现的情况。这并非毫无道理，甚至有人建议将超时放置在所有并发操作上，以确保系统不会发生死锁。 超时时间不一定要接近执行并发操作所需的实际时间。 设置超时时间的目的仅仅是为了防止死锁，所以它只需要足够短以满足死锁系统会在合理的时间内解锁即可。
我们在“死锁，活锁和锁的饥饿问题”章节中提到过，通过设置超时来避免死锁可能会将问题从死锁变为活锁。在大型系统中，由于存在更多的移动部件，因此与死锁相比，系统遇到不同的时序配置文件的可能性更大。因此，最好有机会锁定并修复进程，而非直接让系统死锁最终不得不重启。
请注意，这不是关于如何正确构建系统的建议。而是建议你考思考在开发和测试期间的时间、时序问题。我建议你使用超时，但是目标应该集中在一个没有死锁的系统上，在这种系统中，超时基本不会触发。

现在我们了解了何时使用超时，让我们将注意力转向取消，以及如何构建并发进程以优雅地处理取消。并发进程可能被取消的原因有很多：

超时
超时是隐式的取消操作。

用户干预
为了获得良好的用户体验，通常建议如果启动长时间运行的进程时，向服务器做轮询将状态报告给用户，或允许用户查看他们的状态。当面向用户的并发操作时，有时需要允许用户取消他们已经开始的操作。

父节点取消
如果作为子节点的任何父节点停止，我们应当执行取消。

重复请求
我们可能希望将数据发送到多个并发进程，以尝试从其中一个进程获得更快的响应。 当收到响应时，需要取消其余的处理。 我们将在“重复请求”一节中详细讨论。

此外，也可能有其他的原因。然而，“为什么”这个问题并不像“如何”这样的问题那么困难或有趣。在第4章中，我们探讨了两种取消并发进程的方法：使用done通道和context.Context类型。 但这里我们要探索更复杂的问题：当一个并发进程被取消时，这对正在执行的算法及其下游消费者意味着什么？在编写可随时终止的并发代码时，需要考虑哪些事项？

为了回答这些问题，我们需要探索的第一件事是并发进程的可抢占性。下面是一个简单的例子：

```
var value interface{}
select {
case <-done:
    return
case value = <-valueStream:
}

result := reallyLongCalculation(value)

select {
case <-done:
    return
case resultStream <- result:
}
```

我们已经将valueStream的读取和resultStream的写入耦合起来，并检查done通道，看看goroutine是否已被取消，但这里存在问题。reallyLongCalculation看起来并不会执行抢占操作，而且根据名字，它看起来可能需要很长时间。这意味着，如果在reallyLongCalculation正在执行时某些事件试图取消这个goroutine，则可能需要很长时间才能确认取消并停止。让我们试着让reallyLongCalculation抢占进程，看看会发生什么：

```
reallyLongCalculation := func(done <-chan interface{}, value interface{}) interface{} {
    intermediateResult := longCalculation(value)
    select {
    case <-done:
        return nil
    default:
    }

    return longCaluclation(intermediateResult)
}
```

我们已经取得了一些进展：reallyLongCalculation现在可以抢占进程。但问题依然存在：我们只能对调用该函数的地方进行抢占。为了解决这个问题，我们需要继续调整

```
reallyLongCalculation := func(done <-chan interface{}, value interface{}) interface{} {
    intermediateResult := longCalculation(done, value)
    return longCaluclation(done, intermediateResult)
}
```

如果将这一推理结果归纳一下，我们会看到当前必须做两件事：定义并发进程可抢占的时间段，并确保任何花费比此时间段更多时间的函数本身是可抢占的。一个简单的方法就是将你的goroutine分解成更小的部分。 你应该瞄准所有不可抢占的原子操作，以便在更短的时间内完成。

这里还存在另一个问题：如果goroutine恰好修改共享状态（例如，数据库，文件，内存数据结构），那么当goroutine被取消时会发生什么？goroutine是否尝试回滚？ 这项工作需要多长时间？既然goroutine已经开始运行，它应该停下来，所以这个时间不应该花太长的时间来执行回滚，对吧？

如何处理这个问题很难给出一般性的建议，因为算法的性质决定了你如何处理这种情况；如果在较小的范围内保留对任何共享状态的修改，无论是否需要确保这些修改回滚，通常都可以很好地处理。如果可能的话，在内存中建立临时存储，然后尽可能快地修改状态。作为一个例子，这是错误的做法：

```
result := add(1, 2, 3)
writeTallyToState(result)
result = add(result, 4, 5, 6)
writeTallyToState(result)
```

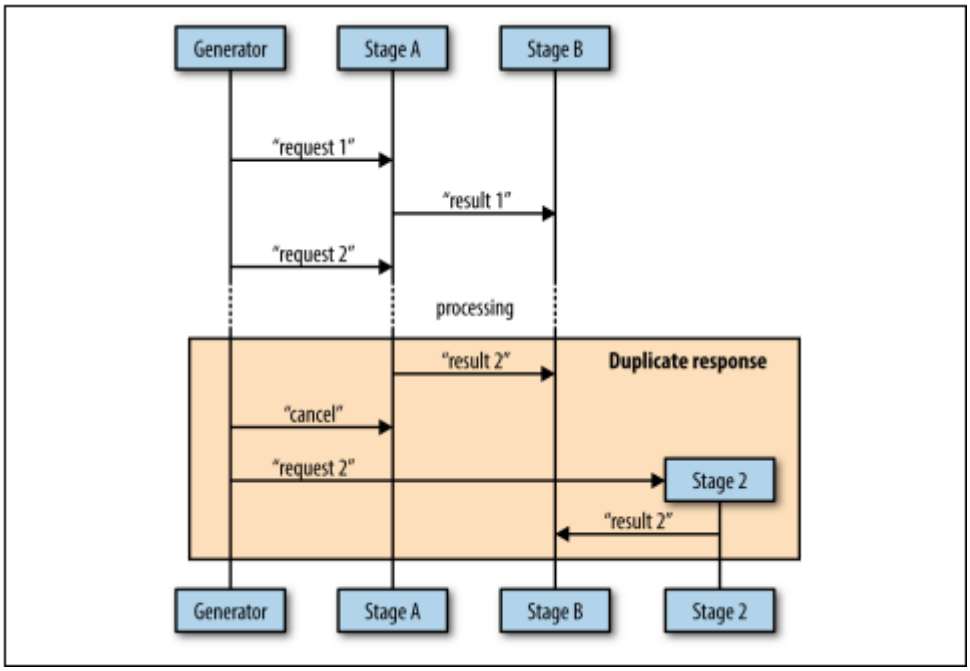
```
result = add(result, 7, 8, 9)
writeTallyToState(result)
```

我们在这里向state写如三次。如果运行此代码的goroutine在最终写入之前被取消，我们需要以某种方式回滚之前的写入。对比这种方法：

```
result := add(1, 2, 3, 4, 5, 6, 7, 8, 9)
writeTallyToState(result)
```

这里需要担心的回滚范围要小得多。 如果在我们调用writeToState之后取消，仍然需要一种方法来退出更改，但发生这种情况的可能性会很小，因为我们只修改一次状态。

你需要关心的另一个问题是消息重复。假设你的管道有三个阶段：generator阶段，A阶段和B阶段。generator阶段监控A阶段，跟踪自上次从其通道读取以来的时间长度，如果当前实例不正常，就创建一个新实例 A2。如果发生这种情况，阶段B可能会收到重复的消息（图5-1）。

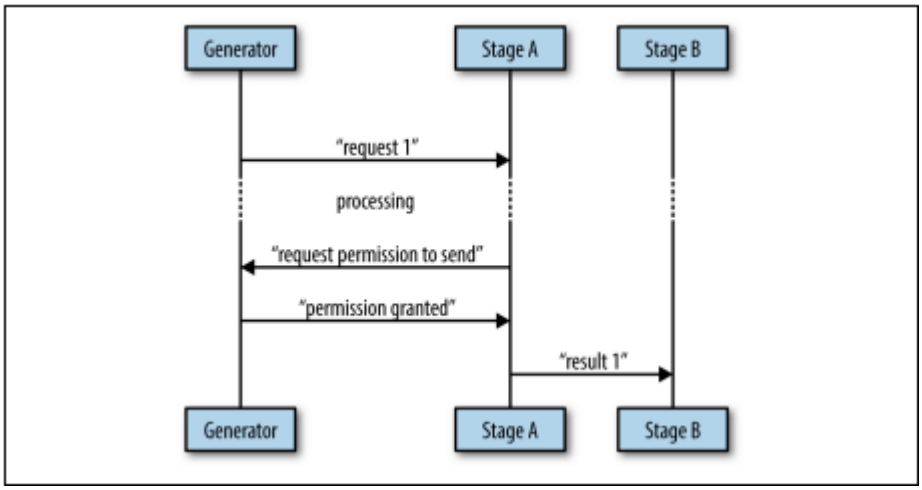


可以在此看到，如果在阶段A已经将阶段B的结果发送到阶段B后取消消息进入，则阶段B可能会收到重复的消息。

有几种方法可以避免这样的情况发生。最简单的方法（以及我推荐的方法）是，在子例程已经报告结果后，父例程不再发送取消信号。这需要各个阶段之间的双向通信，我们将在“心跳”一节中详细介绍。其他方法是：

接受返回的第一个或最后一个结果
如果你的算法允许，或者你的并发进程是幂等的，那么可以简单地在下游进程中允许重复消息，并选择是否接受你收到的第一条或最后一条消息。

检查goroutine许可
可以使用与父节点的双向通信来明确请求发送消息的权限。这种方法类似于心跳。它看起来像这样。



因为我们明确要求许可执行写入B的通道，所以这是比心跳更安全的方法。然而在实践中很少这样做，因为它比心跳更复杂，所以我建议你只是使用心跳。

在设计并发进程时，一定要考虑超时和取消。像软件工程中的许多其他技术问题一样，如果在项目初期忽略超时和取消，然后尝试将它们放在项目后期加入，有点像试图在蛋糕烘烤后再将蛋添加到蛋糕中。

心跳

心跳是并发进程向外界发出信号的一种方式。命名者从人体解剖学中受到启发，使用心跳一词表示被观察者的生命体征。心跳在Go语言出现前就已被广泛使用。

在并发中使用心跳是有原因的。心跳能够让我们更加深入的了解系统，并且在系统存在不确定性的时候对其测试。

我们将在本节中讨论两种不同类型的心跳：

- 以固定时间间隔产生的心跳。
- 在工作单元开始时产生的心跳。

固定时间间隔产生的心跳对于并发来说很有用，它可能在等待处理某个工作单元执行某个任务时发生。由于你不知道这项工作什么时候会进行，所以你的goroutine可能会持续等待。心跳是一种向监听者发出信号的方式，即一切都很好，当前静默是正常的。

以下代码演示了会产生心跳的goroutine：

```
doWork := func(done <-chan interface{}, pulseInterval time.Duration) (<-chan interface{}, <-chan time.Time) {

    heartbeat := make(chan interface{}) //1
    results := make(chan time.Time)

    go func() {
        defer close(heartbeat)
        defer close(results)

        pulse := time.Tick(pulseInterval)       //2
        workGen := time.Tick(2 * pulseInterval) //3

        sendPulse := func() {
            select {
            case heartbeat <- struct{}{}:
            default: //4
            }
        }
        sendResult := func(r time.Time) {
            for {
                select {
                case <-done:
                    return
                case <-pulse: //5
                    sendPulse()
                case results <- r:
                    return
                }
            }
        }

        for {
            select {
            case <-done:
                return
            case <-pulse: //5
                sendPulse()
            case r := <-workGen:
                sendResult(r)
            }
```

```
        }
    }()
    return heartbeat, results
}
```

1. 在这里，我们设置了一个发送心跳信号的通道。doWork会返回该通道。
2. 我们按传入的pulseInterval值定时发送心跳，每次心跳都意味着可以从该通道上读取到内容。
3. 这只是用来模拟进入的工作的另一处代码。我们选择一个比pulseInterval更长的持续时间，以便我们可以看到来自goroutine的心跳。
4. 请注意，我们包含一个default子句。我们必须考虑如果没有人接受到心跳的情况。从goroutine发出的结果是至关重要的，但心跳不是。
5. 就像done通道，无论何时执行发送或接收，你都需要考虑心跳发送的情况。

请注意，由于我们可能在等待输入时发送多个pulse，或者在等待发送结果时发送多个pulse，所有select语句都需要在for循环内。 目前看起来不错; 我们如何利用这个函数并消费它发出的事件？ 让我们来看看：

```
done := make(chan interface{})
time.AfterFunc(10*time.Second, func() { close(done) }) //1

const timeout = 2 * time.Second                //2
heartbeat, results := doWork(done, timeout/2) //3
for {
    select {
    case _, ok := <-heartbeat: //4
        if ok == false {
            return
        }
        fmt.Println("pulse")
    case r, ok := <-results: //5
        if ok == false {
            return
        }
        fmt.Printf("results %v\n", r.Second())
    case <-time.After(timeout): //6
        return
    }
}
```

1. 我们设置done通道并在10秒后关闭它。
2. 我们在这里设定超时时间 我们将用它将心跳间隔与超时时间相耦合。
3. 我们向dowork传入超时时间的一半。
4. 我们将hearbeat的读取放入select语句中。每间隔 timeout/2 获取一次来自心跳通道的消息。如果我们没有收到消息，那就说明该goroutine存在问题。
5. 我们从result通道获取数据，没有什么特别的。
6. 如果我们没有收到心跳或result，程序就会超时结束。

这会输出：

```
pulse
pulse
results 52
pulse
pulse
results 54
pulse
pulse
results 56
pulse
pulse
```

```
results 58
pulse
```

和预期的一样，每次从result中接收到信息，都会收到两次心跳。

我们可能会使用这样的功能来收集系统的统计参数，当你的goroutine没有像预期那样运行，那么基于固定时间的心跳信号的作用会非常明显。

考虑下一个例子。 我们将在两次迭代后停止goroutine来模拟循环中断，然后不关闭任何一个通道；

```
doWork := func(done <-chan interface{}, pulseInterval time.Duration) (<-chan interface{}, <-chan time.Time) {
    heartbeat := make(chan interface{})
    results := make(chan time.Time)
    go func() {
        pulse := time.Tick(pulseInterval)
        workGen := time.Tick(2 * pulseInterval)

        sendPulse := func() {
            select {
            case heartbeat <- struct{}{}:
            default:
            }
        }
        sendResult := func(r time.Time) {
            for {
                select {
                case <-pulse:
                    sendPulse()
                case results <- r:
                    return
                }
            }
        }

        for i := 0; i < 2; i++ { //1
            select {
            case <-done:
                return
            case <-pulse:
                sendPulse()
            case r := <-workGen:
                sendResult(r)
            }
        }
    }()
    return heartbeat, results
}

done := make(chan interface{})
time.AfterFunc(10*time.Second, func() { close(done) })

const timeout = 2 * time.Second
heartbeat, results := doWork(done, timeout/2)
for {
    select {
    case _, ok := <-heartbeat:
        if ok == false {
            return
        }
        fmt.Println("pulse")
```

```
    case r, ok := <-results:
        if ok == false {
            return
        }
        fmt.Printf("results %v\n", r)
    case <-time.After(timeout):
        fmt.Println("worker goroutine is not healthy!")
        return
    }
}
```

1. 这里我们简单模拟循环中断。前面的例子中，未收到通知会无限循环。这里我们只循环两次。

这会输出：

```
pulse
pulse
worker goroutine is not healthy!
```

效果很不错。在两秒钟之内，我们的系统意识到goroutine未能正确读取，并且打破了for-select循环。通过使用心跳，我们已经成功地避免了死锁，并且不必通过依赖较长的超时而保持稳定性。 我们将在“Goroutines异常行为修复”中进一步理解这个概念。

另外请注意，心跳会帮助处理相反的情况：它让我们知道长时间运行的goroutine依然存在，但花了一段时间才产生一个值并发送至通道。

接下来让我们看看另一个场景：在工作单元开始时产生的心跳。这对测试非常有用。下面是个例子：

```
doWork := func(done <-chan interface{}) (<-chan interface{}, <-chan int) {

    heartbeatStream := make(chan interface{}, 1) //1
    workStream := make(chan int)
    go func() {
        defer close(heartbeatStream)
        defer close(workStream)

        for i := 0; i < 10; i++ {
            select { //2
            case heartbeatStream <- struct{}{}:
            default: //3
            }

            select {
            case <-done:
                return
            case workStream <- rand.Intn(10):
            }
        }
    }()

    return heartbeatStream, workStream
}

done := make(chan interface{})

defer close(done)

heartbeat, results := doWork(done)
for {
```

```
    select {
    case _, ok := <-heartbeat:
        if ok {
            fmt.Println("pulse")
        } else {
            return
        }
    case r, ok := <-results:
        if ok {
            fmt.Printf("results %v\n", r)
        } else {
            return
        }
    }
}
```

1. 这里我们用一个缓冲区创建心跳通道。这确保即使没有人及时监听发送，也总会发送至少一个pulse。
2. 在这里，我们为心跳设置了一个单独的select块。我们不希望将它与发送结果一起包含在同一个select块中，因为如果接收器未准备好，它们将接收到一个pulse，而result的当前值将会丢失。我们也没有为done通道提供case语句，因为我们有一个default可以处理这种情况。
3. 我们再次处理如果没有人监听到心头。因为我们的心跳通道是用缓冲区创建的，如果有人在监听，但没有及时处理第一个心跳，仍会被通知。

这会输出：

```
pulse
results 1
pulse
results 7
pulse
results 7
pulse
results 9
pulse
results 1
pulse
results 8
pulse
results 5
pulse
results 0
pulse
results 6
pulse
results 0
```

如预期一致，每个结果都会有一个心跳。

至于测试的编写。考虑下面的代码：

```
func DoWork( done <-chan interface {}, nums ...int ) (<-chan interface{}, <-chan int) {

    heartbeat := make(chan interface{}, 1)
    intStream := make(chan int)

    go func () {

        defer close(heartbeat)
        defer close(intStream)
```

```
        time.Sleep(2*time.Second) // 1

        for _, n := range nums {

            select {
            case heartbeat <- struct{}{}:
            default:
            }

            select {
            case <-done:
            return
            case intStream <- n:
            }
        }
    }()

    return heartbeat, intStream
}
```

1. 我们在goroutine开始工作之前模拟延迟。在实践中，延迟可以由各种各样的原因导致，例如CPU负载，磁盘争用，网络延迟和bug。

DoWork函数是一个相当简单的生成器，它将传入的数字转换为它返回通道上的数据流。我们来试试这个函数。下面提供了一个测试的反例：

```
func TestDoWork_GeneratesAllNumbers(t *testing.T) {

    done := make(chan interface{})
    defer close(done)

    intSlice := []int{0, 1, 2, 3, 5}
    _, results := DoWork(done, intSlice...)

    for i, expected := range intSlice {
        select {
        case r := <-results:
            if r != expected {
                t.Errorf(
                    "index %v: expected %v, but received %v,", i,
                    expected, r,
                )
            }
        case <-time.After(1 * time.Second): // 1
            t.Fatal("test timed out")
        }
    }
}
```

1. 在这里，我们设置超时，以防止goroutine出现问题导致死锁。

运行结果为：

```
go test ./bad_concurrent_test.go
--- FAIL: TestDoWork_GeneratesAllNumbers (1.00s) bad_concurrent_test.go:46: test timed out
FAIL
FAIL    command-line-arguments 1.002s
```

这个测试之所以不好，是因为它的不确定性。如果移除time.Sleep情况会变得更糟：这个测试会有时通过，有时失败。

我们之前提到过程中的外部因素可能会导致goroutine花费更长的时间才能完成第一次迭代。关键在于我们不能保证在超时之前第一次迭代会完成，所以我们开始考虑：这时候超时会有多大意义？我们可以增加超时时间，但这意味着测试时失败也需要很长时间，从而减慢我们的测试效率。

这种情况很可怕，项目组甚至会对测试的正确性及必要性产生怀疑。

幸运的是这种情况并非无解。这是一个正确的测试：

```
func TestDoWork_GeneratesAllNumbers(t *testing.T) {

    done := make(chan interface{})
    defer close(done)

    intSlice := []int{0, 1, 2, 3, 5}
    heartbeat, results := DoWork(done, intSlice...)

    <-heartbeat //1

    i := 0
    for r := range results {
        if expected := intSlice[i]; r != expected {
            t.Errorf("index %v: expected %v, but received %v,", i, expected, r)
        }
        i++
    }
}
```

1. 在这里，我们等待goroutine发出信号表示它正在开始处理迭代。 运行此测试会产生以下输出

```
ok    command-line-arguments    2.002s
```

使用心跳我们可以安全地编写该测试，而不会超时。运行的唯一风险是我们的一次迭代花费了过多的时间。 如果这对我们很重要，我们可以利用更安全的、基于间隔的心跳。

以下是使用基于间隔的心跳的测试示例：

```
func DoWork(done <-chan interface{}, pulseInterval time.Duration, nums ...int) (<-chan interface{}, <-chan in
t) {

    heartbeat := make(chan interface{}, 1)
    intStream := make(chan int)

    go func() {
        defer close(heartbeat)
        defer close(intStream)
        time.Sleep(2 * time.Second)
        pulse := time.Tick(pulseInterval)
    numLoop: //2
        for _, n := range nums {
            for { //1
                select {
                case <-done:
                    return

                case <-pulse:
                    select {
```

```
                    case heartbeat <- struct{}{}: default:
                    }
                case intStream <- n:
                    continue numLoop //3
                }
            }
        }
    }()

    return heartbeat, intStream
}

func TestDoWork_GeneratesAllNumbers(t *testing.T) {
    done := make(chan interface{})
    defer close(done)

    intSlice := []int{0, 1, 2, 3, 5}
    const timeout = 2 * time.Second
    heartbeat, results := DoWork(done, timeout/2, intSlice...)

    <-heartbeat //4

    i := 0
    for {
        select {
        case r, ok := <-results:
            if ok == false {
                return
            } else if expected := intSlice[i]; r != expected {
                t.Errorf(
                    "index %v: expected %v, but received %v,", i,
                    expected, r,
                )
            }
            i++
        case <-heartbeat: //5
        case <-time.After(timeout):
            t.Fatal("test timed out")
        }
    }
}
```

1. 我们需要两个循环：一个用来覆盖我们的数字列表，并且这个内部循环会运行直到intStream上的数字成功发送。
2. 我们在这里使用一个标签来使内部循环继续更简单一些。
3. 这里我们继续执行外部循环。
4. 我们仍然等待第一次心跳出现，表明我们已经进入了goroutine的循环。
5. 我们在这里获取心跳以实现超时。

运行此测试会输出：

```
ok    command-line-arguments    3.002s
```

你可能已经注意到这个版本的逻辑有点混乱。如果你确信goroutine的循环在启动后不会停止执行，我建议只阻塞第一次心跳，然后进入循环语句。你可以编写单独的测试，专门来测试如未能关闭通道，循环迭代耗时过长以及其他与时间相关的情况。

在编写并发代码时，心跳不是绝对必要的，但本节将展示其的实用性。对于任何需要测试的长期运行的goroutines，我强烈推荐这种模式。

请求并发复制处理

对于大部分应用，尽可能快地响应请求是首要任务。例如，应用程序可能正在服务用户的HTTP请求，或者检索复制的数据块。在这些情况下，你需要做出权衡：是将请求复制到多个处理程序（无论是goutoutine，进程还是服务器），并且其中一个将比其他处理程序返回更快呢，还是立即返回结果——缺点是必须考虑如何高效利用资源来保持处理程序的多个副本同时运行。

如果这种复制是在内存中完成的，它耗费的资源可能并不是那么昂贵，但如果处理程序需要复制进程，服务器甚至数据中心，这就完全不一样了。你必须考虑这样做成本是否值得。

我们来看看如何在单个进程中复制请求。我们将使用多个goroutines作为请求处理程序，并且goroutine将在1到6纳秒之间随机休眠一段时间以模拟负载。 这将使处理程序在不同时间返回结果，并让我们看到这样做能否更高效。

下面这个例子通过10个处理程序复制模拟请求：

```
doWork := func(done <-chan interface{}, id int, wg *sync.WaitGroup, result chan<- int) {

    started := time.Now()
    defer wg.Done()

    // 模拟随机加载
    simulatedLoadTime := time.Duration(1+rand.Intn(5)) * time.Second
    select {
    case <-done:
    case <-time.After(simulatedLoadTime):
    }

    select {
    case <-done:
    case result <- id:
    }

    took := time.Since(started)
    // 显示处理程序将花费多长时间
    if took < simulatedLoadTime {
        took = simulatedLoadTime
    }
    fmt.Printf("%v took %v\n", id, took)
}

done := make(chan interface{})
result := make(chan int)

var wg sync.WaitGroup
wg.Add(10)

for i := 0; i < 10; i++ { //1
    go doWork(done, i, &wg, result)
}

firstReturned := <-result //2
close(done)               //3
wg.Wait()

fmt.Printf("Received an answer from #%v\n", firstReturned)
```

1. 我们开启10个处理程序以处理请求。
2. 抓取处理程序第一个返回的值。
3. 取消所有剩余的处理程序。这确保他们不会继续做不必要的工作。

这会输出：

```
4 took 1s
3 took 1s
6 took 2s
8 took 1s
2 took 2s
0 took 2s
7 took 4s
5 took 5s
1 took 3s
9 took 3s
Received an answer from #3
```

在输出中，我们显示每个处理程序花费了多久，以便你了解此技术可以节省多少时间。

唯一需要注意的是，所有的处理程序都需要有相同且平等的请求。换句话说，不会出现从无法处理请求的处理程序接收响应时间。正如我所提到的那样，所有处理者用来完成工作的资源都需要复制。

如果你的处理程序太相似了，那么任何一个出现异常的几率都会更小。你只应该将这样的请求复制到具有不同运行时条件的处理程序：不同的进程，计算机，数据存储路径或完全访问不同的数据存储区。

这样做的代价可能是昂贵且难以维护。如果速度是你的目标，那这个技术就很有价值了。当然你还需要考虑容错和可扩展性。

速率限制

如果你曾经使用过API来获取服务，那么你可能经受过与速率限制相抗衡。速率限制使得某种资源每次访问的次数受限。资源可以是任何东西：API连接，磁盘I/O，网络包，错误。

你有没有想过为什么会需要制定速率限制？为什么不允许无限制地访问系统？最明显的答案是，通过对系统进行速率限制，可以防止整个系统遭受攻击。如果恶意用户可以在他们的资源允许的情况下极快地访问你的系统，那么他们可以做各种事情。

例如，他们可以用日志消息或有效的请求填满服务器的磁盘。如果你的日志配置错误，他们甚至可能会执行恶意的操作，然后提交足够的请求，将任何活动记录从日志中移出并放入/dev/null中导致日志系统完全崩溃。他们可能试图暴力访问资源，或者执行分布式拒绝服务攻击。如果你没有对系统进行请求限制，你的系统就成了一个走在街上不穿衣服的大美女。

更糟糕的是，非恶意请求有时也会造成上述结果。恶意使用并不是唯一的原因。 在分布式系统中，如果合法用户正在以足够高的速度执行操作，或者他们正在运行的代码有问题，则合法用户可能会降低系统的性能。这甚至可能导致我们之前讨论的死亡螺旋。 从产品的角度来看，这太糟糕了！通常情况下，你希望向用户提供某种类型的性能保证，以确保他们可以在一致的基础上达到不错的性能。如果一个用户可以影响该协议，那么你的日子就不好过了。用户对系统的访问通常被沙盒化，既不会影响其他用户的活动也不会受其他用户影响。如果你打破了这种思维模式，你的系统会表现出糟糕的设计使用感，甚至会导致用户生气或离开。

> 我曾经在一个分布式系统上工作，通过启动新的进程来并行扩展（这允许它水平扩展到多台机器）。每个进程都会打开数据库连接，读取一些数据并进行一些计算。一段时间以来，我们在以这种方式扩展系统以满足客户需求方面取得了巨大成功。 但是，一段时间后，我们发现大量的数据库中读取数据超时。
> 我们的数据库管理员仔细查看了日志，试图找出问题所在。最后他们发现，由于系统中没有设定任何速率限制，所以进程彼此重叠。由于不同的进程试图从磁盘的不同部分读取数据，因此磁盘竞争将达到100％并一直保持在这个水平。这反过来导致了一系列的循环——超时——重试——循环。工作永远不会完成。
> 我们设计了一个系统来限制数据库上可能的连接数量，并且速率限制被放在连接可以读取的秒级单位，问题消失了。虽然客户不得不等待更长的时间，但毕竟他们的工作完成了，我们能够在接下来的时间里进行适当的容量规划，以系统化的方式扩展容量。

速率限制使得你可以通过某个界限来推断系统的性能和稳定性。如果你需要扩展这些边界，可以在大量测试后以可控的方式进行。

在密集用户操作的情况下，速率限制可以使你的系统与用户之间保持良好的关系。你可以允许他们在严格限制的速率下尝试系统的特性。Google的云产品证明了这一点，这种限制在某种意义上是可以保护用户的。

速率限制通常由资源的构建者角度来考虑，但对于用户来说，能够减少速率限制的影响会让其感到非常欣慰。

那么我们怎样用Go来实现呢？

大多数速率限制是通过使用称为令牌桶的算法完成的。这很容易理解，而且相对容易实施。 我们来看看它背后的理论。

假设要使用资源，你必须拥有资源的访问令牌。没有令牌，请求会被拒绝。想象这些令牌存储在等待被检索以供使用的桶中。该桶的深度为d，表示它一次可以容纳d个访问令牌。

现在，每次你需要访问资源时，你都会进入存储桶并删除令牌。如果你的存储桶包含五个令牌，那么您可以访问五次资源。在第六次访问时，没有访问令牌可用，那么必须将请求加入队列，直到令牌变为可用，或拒绝请求。

这里有一个时间表来帮助观察这个概念。time表示以秒为单位的时间增量，bucket表示桶中请求令牌的数量，并且请求列中的tok表示成功的请求。（在这个和未来的时间表中，我们假设这些请求是即时的。）

time	bucket	request
0	5	tok
0	4	tok
0	3	tok
0	2	tok
0	1	tok
0	0	
1	0	
	0	

可以看到，在第一秒之前，我们能够完成五个请求，然后我们被阻塞，因为没有更多的令牌可供使用。

到目前为止，这些都很容易理解。那么如何补充令牌呢？在令牌桶算法中，我们将r定义为将令牌添加回桶的速率。 它可以是一纳秒或一分钟。它就是我们通常认为的速率限制：因为我们必须等待新的令牌可用，我们将操作限制为令牌的刷新率。

以下是深度为1的令牌桶示例，速率为1令牌/秒：

time	bucket	request
0	1	
0	0	tok
1	0	
2	1	
2	0	tok
3	0	
4	1	
4	0	tok

可以看到我们立即可以提出请求，但是只能隔一秒钟将请求一次。速率限制执行的非常好！

所以我们现在有两个设置可以摆弄：有多少令牌可用于立即使用，即桶的深度d，以及它们补充的速率r。另外我们还可以考虑下当存储桶已满时可以进行多少次请求。

以下是深度为5的令牌桶示例，速率为0.5令牌/秒：

time	bucket	request
0	5	
0	4	tok
0	3	tok
0	2	tok
0	1	tok
0	0	tok
1	0 (0.5)	
2	1	
2	0	tok
3	0 (0.5)	
4	1	
4	0	tok

在这里，我们能够立即提出五个请求，在这之后，每两秒限制一次请求。请求的爆发是在桶最初装满的时候。

请注意，用户可能不会消耗一个长数据流中的整个令牌桶。桶的深度只控制桶的容量。这里有一个用户爆发两次请求的例子，然后四秒钟后爆发了五次：

time	bucket	request
0	5	
0	4	tok
0	3	tok
1	3	
2	4	
3	5	
4	5	
5	4	tok
time	bucket	request
5	3	tok
5	2	tok
5	1	tok
5	0	tok

虽然用户有可用的令牌，但这种爆发性仅允许根据调用者的能力限制访问系统。对于只能间歇性访问系统但希望尽快往返的用户来说，这种爆发性的存在是很好的。你只需要确保系统能够同时处理所有用户的爆发操作，或者确保操作上不可能有足够的用户同时爆发操作以影响你的系统。无论哪种方式，速率限制都可以让你规避风险。

让我们使用这个算法来看看一个Go程序在执行令牌桶算法的时如何表现。

假设我们可以访问API，并且已经提供了客户端来使用它。该API有两个操作：一个用于读取文件，另一个用于将域名解析为IP地址。为了简单起见，我将忽略任何参数并返回实际访问服务所需的值。 所以这是我们的客户端：

```
func Open() *APIConnection {
    return &APIConnection{}
}

type APIConnection struct{}

func (a *APIConnection) ReadFile(ctx context.Context) error {
    // Pretend we do work here
    return nil
}

func (a *APIConnection) ResolveAddress(ctx context.Context) error {
    // Pretend we do work here
    return nil
}
```

由于理论上这个请求正在通过网络传递，所以我们将context.Context作为第一个参数，以防我们需要取消请求或将值传递给服务器。 通过前面的章节讨论，想必大家已经熟悉了这样的用法。

现在我们将创建一个简单的程序来访问这个API。程序需要读取10个文件并解析10个地址，但这些文件和地址彼此没有关系，因此程序可以并发调用。稍后这将有助于添加利率限制。

```
func main() {

    defer log.Printf("Done.")

    log.SetOutput(os.Stdout)
    log.SetFlags(log.Ltime | log.LUTC)

    apiConnection := Open()
    var wg sync.WaitGroup
    wg.Add(20)

    for i := 0; i < 10; i++ {
        go func() {
            defer wg.Done()
            err := apiConnection.ReadFile(context.Background())
            if err != nil {
                log.Printf("cannot ReadFile: %v", err)
            }
        }()

    }
    log.Printf("ReadFile")

    for i := 0; i < 10; i++ {
        go func() {
            defer wg.Done()
            err := apiConnection.ResolveAddress(context.Background())
            if err != nil {
                log.Printf("cannot ResolveAddress: %v", err)

            }
        }()
    }
```

```
    log.Printf("ResolveAddress")

    wg.Wait()
}
```

这会输出：

```
20:13:13 ResolveAddress
20:13:13 ReadFile
20:13:13 ResolveAddress
20:13:13 ReadFile
20:13:13 ReadFile
20:13:13 ReadFile
20:13:13 ReadFile
20:13:13 ResolveAddress
20:13:13 ResolveAddress
20:13:13 ReadFile
20:13:13 ResolveAddress
20:13:13 ResolveAddress
20:13:13 ResolveAddress
20:13:13 ResolveAddress
20:13:13 ResolveAddress
20:13:13 ResolveAddress
20:13:13 ReadFile
20:13:13 ReadFile
20:13:13 ReadFile
20:13:13 ReadFile
20:13:13 Done.
```

我们可以看到所有的API请求几乎同时进行。由于没有设定速率限制，所以用户可以随意随意访问系统。现在我需要提醒你，当前个可能导致无限循环的错误。

好的，让我们来介绍一个限速器。我打算在APIConnection中这样做，但通常会在服务器上运行速率限制器，这样用户就无法绕过它。生产系统还可能包括一个客户端速率限制器，以防止用户因不必要的调用而被拒绝，但这是一种优化，并不是必需的。就我们的目的而言，限速器使事情变得简单。

我们将看看 golang.org/x/time/rate 中的令牌桶速率限制器实现的示例。我选择了这个包，因为这跟我所能得到的标准库较为相近。当然还有其他的软件包可以做更多的花里胡哨的工作。 golang.org/x/time/rate 非常简单，而且它适用于我们的目的。

我们将与这个包交互的前两种方法是Limit类型和NewLimiter函数，在这里定义：

```
// Limit 定义了事件的最大频率。
// Limit 被表示为每秒事件的数量。
// 值为0的Limit不允许任何事件。
type Limit float64

// ewLimiter返回一个新的Limiter实例，
// 件发生率为r，并允许至多b个令牌爆发。
func NewLimiter(r Limit, b int) *Limiter
```

在NewLimiter中，我们看到了两个熟悉的参数：r ，以及 b.r。b是我们之前讨论过的桶的深度。

rate 包还定义了一个有用的函数，Every，帮助将time.Duration转换为Limit：

```
// Every将事件之间的最小时间间隔转换为Limit。
func Every(interval time.Duration) Limit
```

Every函数是有意义的，但我想讨论每次rate限制了操作次数，而不是请求之间的时间间隔。 我们可以将其表述如下：

```
rate.Limit(events/timePeriod.Seconds())
```

但是我不想每次都输入这个值，Every函数有一些特殊的逻辑会返回rate.Inf——表示没有限制——如果提供的时间间隔为零。 正因为如此，我们将用这个词来表达我们的帮助函数：

```
func Per(eventCount int, duration time.Duration) rate.Limit {
    return rate.Every(duration/time.Duration(eventCount))
}
```

在我们建立rate.Limiter后，我们希望使用它来阻塞我们的请求，直到获得访问令牌。 我们可以用Wait方法来做到这一点，它只是用参数1来调用WaitN：

```
// Wait 是 WaitN(ctx, 1)的简写。
func (lim *Limiter) Wait(ctx context.Context)

// WaitN 会发生阻塞直到 lim 允许的 n 个事件执行。
// 它返回一个 error 如果 n 超过了 Limiter的桶大小,
// Context会被取消, 或等待的时间超过了 Context 的 Deadline。
func (lim *Limiter) WaitN(ctx context.Context, n int) (err error)
```

我们已经集齐了限制API请求的所有要素。让我们修改APIConnection类型并尝试一下：

```
func Open() *APIConnection {
    return &APIConnection{
        rateLimiter: rate.NewLimiter(rate.Limit(1), 1) //1
    }
}

type APIConnection struct {
    rateLimiter *rate.Limiter
}

func (a *APIConnection) ReadFile(ctx context.Context) error {
    if err := a.rateLimiter.Wait(ctx); err != nil { //2
        return err
    }
    // Pretend we do work here
    return nil
}

func (a *APIConnection) ResolveAddress(ctx context.Context) error {
    if err := a.rateLimiter.Wait(ctx); err != nil { //2
        return err
    }
    // Pretend we do work here
    return nil
}
```

1. 我们将所有API连接的速率限制设置为每秒一个事件。
2. 我们等待速率限制器有足够的权限来完成我们的请求。

这会输出：

```
22:08:30 ResolveAddress
22:08:31 ReadFile
22:08:32 ReadFile
22:08:33 ReadFile
22:08:34 ResolveAddress
22:08:35 ResolveAddress
22:08:36 ResolveAddress
22:08:37 ResolveAddress
22:08:38 ResolveAddress
22:08:39 ReadFile
22:08:40 ResolveAddress
22:08:41 ResolveAddress
22:08:42 ResolveAddress
22:08:43 ResolveAddress
22:08:44 ReadFile
22:08:45 ReadFile
22:08:46 ReadFile
22:08:47 ReadFile
22:08:48 ReadFile
22:08:49 ReadFile
22:08:49 Done.
```

在生产中，我们可能会想要一些更复杂的东西。我们可能希望建立多层限制：细粒度控制以限制每秒请求数，粗粒度控制限制每分钟，每小时或每天的请求数。

在某些情况下，可以通过速率限制器来实现；然而，在所有情况下都这么干是不可能的，并且通过尝试将单位时间的限制语义转换为单一层，你会因速率限制器而丢失大量信息。出于这些原因，我发现将限制器分开并将它们组合成一个限速器来管理交互更容易。 为此，我创建了一个简单的聚合速率限制器multiLimiter。 这是定义：

```
type RateLimiter interface { //1
    Wait(context.Context) error
    Limit() rate.Limit
}

func MultiLimiter(limiters ...RateLimiter) *multiLimiter {
    byLimit := func(i, j int) bool {
        return limiters[i].Limit() < limiters[j].Limit()
    }
    sort.Slice(limiters, byLimit) //2
    return &multiLimiter{limiters: limiters}

}

type multiLimiter struct {
    limiters []RateLimiter
}

func (l *multiLimiter) Wait(ctx context.Context) error {
    for _, l := range l.limiters {
        if err := l.Wait(ctx); err != nil {
            return err
        }
    }
    return nil
}

func (l *multiLimiter) Limit() rate.Limit {
    return l.limiters[0].Limit() //3
}
```

1. 这里我们定义一个RateLimiter接口，以便MultiLimiter可以递归地定义其他MultiLimiter实例。
2. 这里我们实现一个优化，并按照每个RateLimiter的Limit()行排序。
3. 因为我们在multiLimiter实例化时对子RateLimiter实例进行排序，所以我们可以简单地返回限制性最高的limit，这将是切片中的第一个元素。

Wait方法遍历所有子限制器，并在每一个子限制器上调用Wait。这些调用可能会也可能不会阻塞，但我们需要通知每个子限制器，以便减少令牌桶内的令牌数量。通过等待每个限制器，我们保证最长的等待时间。这是因为如果我们执行时间较小的等待，那么最长的等待时间将被重新计算为剩余时间。在较早的等待被阻塞时，后者会等待令牌桶的填充。

经过思考，让我们重新定义APIConnection，对每秒和每分钟都进行限制：

```
func Open() *APIConnection {
    secondLimit := rate.NewLimiter(Per(2, time.Second), 1)    //1
    minuteLimit := rate.NewLimiter(Per(10, time.Minute), 10) //2
    return &APIConnection{
        rateLimiter: MultiLimiter(secondLimit, minuteLimit) //3
    }
}

type APIConnection struct {
    rateLimiter RateLimiter
}

func (a *APIConnection) ReadFile(ctx context.Context) error {
    if err := a.rateLimiter.Wait(ctx); err != nil {
        return err
    }
    // Pretend we do work here
    return nil
}

func (a *APIConnection) ResolveAddress(ctx context.Context) error {
    if err := a.rateLimiter.Wait(ctx); err != nil {
        return err
    }
    // Pretend we do work here
    return nil
}
```

1. 我们定义了每秒的极限。
2. 我们定义每分钟的突发极限为10，为用户提供初始池。每秒的限制将确保我们1不会因请求而使系统过载。
3. 我们结合这两个限制，并将其设置为APIConnection的主限制器。

这会输出：

```
22:46:10 ResolveAddress
22:46:10 ReadFile
22:46:11 ReadFile
22:46:11 ReadFile
22:46:12 ReadFile
22:46:12 ReadFile
22:46:13 ReadFile
22:46:13 ReadFile
22:46:14 ReadFile
22:46:14 ReadFile
22:46:16 ResolveAddress
22:46:22 ResolveAddress
22:46:28 ReadFile
22:46:34 ResolveAddress
22:46:40 ResolveAddress
```

```
22:46:46 ResolveAddress
22:46:52 ResolveAddress
22:46:58 ResolveAddress
22:47:04 ResolveAddress
22:47:10 ResolveAddress
22:47:10 Done.
```

正如您所看到的，我们每秒发出两个请求，直到请求＃11，此时我们开始每隔六秒发出一次请求。 这是因为我们耗尽了我们可用的每分钟请求令牌池，并受此限制。

为什么请求＃11在两秒内发生而不是像其他请求那样发生，这可能有点违反直觉。 请记住，尽管我们将API请求限制为每分钟10个，但一分钟是一个动态的时间窗口。 当我们达到第十一个要求时，我们的每分钟限制器已经累积了另一个令牌。

通过定义这样的限制，我们可以清楚地表达了粗粒度限制，同时仍然以更详细的细节水平限制请求数量。

这种技术也使我们能够开始思考除时间之外的其他维度。 当你对系统进行限制时，你可能会限制不止一件事。 也可能对API请求的数量有一些限制，也会对其他资源（如磁盘访问，网络访问等）有限制。让我们稍微充实一下示例并设置磁盘和网络限制

```
func Open() *APIConnection {
    return &APIConnection{
        apiLimit: MultiLimiter( //1
            rate.NewLimiter(Per(2, time.Second), 2),
            rate.NewLimiter(Per(10, time.Minute), 10)),
        diskLimit:    MultiLimiter(rate.NewLimiter(rate.Limit(1), 1)),      //2
        networkLimit: MultiLimiter(rate.NewLimiter(Per(3, time.Second), 3)) //3
    }
}

type APIConnection struct {
    networkLimit,
    diskLimit,
    apiLimit RateLimiter
}

func (a *APIConnection) ReadFile(ctx context.Context) error {
    err := MultiLimiter(a.apiLimit, a.diskLimit).Wait(ctx) //4
    if err != nil {
        return err
    }
    // Pretend we do work here
    return nil
}

func (a *APIConnection) ResolveAddress(ctx context.Context) error {
    err := MultiLimiter(a.apiLimit, a.networkLimit).Wait(ctx) //5
    if err != nil {
        return err
    }
    // Pretend we do work here
    return nil
}
```

1. 我们为API调用设置了一个限制器。 每秒请求数和每分钟请求数都有限制
2. 我们为磁盘读取设置一个限制器。将其限制为每秒一次读取。
3. 对于网络，我们将设置每秒三个请求的限制。
4. 当我们读取文件时，将结合来自API限制器和磁盘限制器的限制。
5. 当我们需要网络访问时，将结合来自API限制器和网络限制器的限制。

这会输出：

```
01:40:15 ResolveAddress
01:40:15 ReadFile
01:40:16 ReadFile
01:40:17 ResolveAddress
01:40:17 ResolveAddress
01:40:17 ReadFile
01:40:18 ResolveAddress
01:40:18 ResolveAddress
01:40:19 ResolveAddress
01:40:19 ResolveAddress
01:40:21 ResolveAddress
01:40:27 ResolveAddress
01:40:33 ResolveAddress
01:40:39 ReadFile
01:40:45 ReadFile
01:40:51 ReadFile
01:40:57 ReadFile
01:41:03 ReadFile
01:41:09 ReadFile
01:41:15 ReadFile
01:41:15 Done.
```

让我们关注一下这样的事实，即我们可以将限制器组合成对每个调用都有意义的组，并且让APIClient执行正确的操作。如果我们想随便观察一下它是如何工作的，会注意到涉及网络访问的API调用似乎以更加规律的方式发生，并在前三分之二的调用中完成。这可能与goroutine调度有关，也是我们的限速器正在执行各自的工作。

我还应该提到的是，rate.Limiter类型有一些其他的技巧来优化不同的用例。这里只讨论了它等待令牌桶接收另一个令牌的能力，但如果你有兴趣使用它，可以查阅文档。

在本节中，我们考察了使用速率限制的理由，构建了令牌桶算法的Go实现，以及如何将令牌桶限制器组合为更大，更复杂的速率限制器。这应该能让你很好地了解速率限制，并帮助你开始使用它们。

Goroutines异常行为修复

在诸如守护进程这样的长期进程中，拥有一组长生命周期的goroutines非常普遍。这些goroutines通常被阻塞，等待被某种方式唤醒以继续工作。有时候，这些例程依赖于你没有很好控制的资源。也许一个goroutine会接收到Web服务中希望获取数据的请求，或者它正在监视一个临时文件。 如果程序处理不够健壮，goroutine会很容易陷入一个糟糕的状态。在长期运行的过程中，如果能创建一种机制来确保goroutine的健康状况良好，并在健康状况不佳时重新启动，那么我们的项目想必能活得久一点。 我们将在本节讨论对goroutines异常行为进行修复的话题。

我们将使用心跳来检查正在监测的goroutine的活跃程度。心跳的类型将取决于你想要监控的内容，但是如果你的goroutine可能会产生活锁，请确保心跳包含某种信息，以表明该goroutine不仅没死掉，而且还可以正常执行任务。在本节中，为了简单起见，我们只会考虑goroutines是活的还是死的。

下面这段代码建立一个管理者监视一个goroutine的健康状况，以及它的子例程。如果例程变得不健康，管理者将重新启动子例程。为此，它需要引用一个可以启动goroutine的函数。让我们看看管理程序是什么样子的：

```
type startGoroutineFn func(done <-chan interface{},
        pulseInterval time.Duration) (heartbeat <-chan interface{}) //1

    newSteward := func(timeout time.Duration, startGoroutine startGoroutineFn) startGoroutineFn { //2

        return func(done <-chan interface{}, pulseInterval time.Duration) <-chan interface{} {

            heartbeat := make(chan interface{})

            go func() {
                defer close(heartbeat)

                var wardDone chan interface{}
                var wardHeartbeat <-chan interface{}
                startWard := func() { //3

                    wardDone = make(chan interface{}) //4
                    wardHeartbeat = startGoroutine(or(wardDone, done), timeout/2) //5
                }
                startWard()
                pulse := time.Tick(pulseInterval)

            monitorLoop:

                for { //6
                    timeoutSignal := time.After(timeout)

                    for {

                        select {
                        case <-pulse:

                            select {
                            case heartbeat <- struct{}{}:

                            default:

                            }
                        case <-wardHeartbeat: //7
                            continue monitorLoop

                        case <-timeoutSignal: //8

                            log.Println("steward: ward unhealthy; restarting")
```

```
                            close(wardDone)
                            startWard()
                            continue monitorLoop
                        case <-done:

                            return
                        }

                    }

                }
            }()

            return heartbeat

        }

    }
```

1. 这里我们定义一个可以监控和重新启动的goroutine的函数签名。 我们看到熟悉的done通道，以及熟悉的心跳模式写法。
2. 在这里我们设置了超时时间，并使用函数startGoroutine来启动它正在监控的goroutine。有趣的是，监控器本身返回一个startGoroutineFn，表示监控器自身也是可监控的。
3. 在这里我们定义一个闭包，它以同样的的方式来启动我们正在监视的goroutine。
4. 这是我们创建一个新通道，我们会将其传递给监控通道，以响应发出的停止信号。
5. 在这里，我们开启对目标goroutine的监控。如果监控器停止工作，或者监控器想要停止被监控区域，我们希望监控者也停止，因此我们将两个done通道都包含在逻辑中。我们传入的心跳间隔是超时时间的一半，但正如我们在“心跳”中讨论的那样，这可以调整。
6. 这是我们的内部循环，它确保监控者可以发出自己的心跳。
7. 在这里我们如果接收到监控者的心跳，就会知道它还处于正常工作状态，程序会继续监测循环。
8. 这里如果我们发现监控者超时，我们要求监控者停下来，并开始一个新的goroutine。然后开始新的监测。

我们的for循环有点杂乱，但如果你阅读过前面的章节，熟悉其中的模式，那么理解起来会相对简单。 接下来让我们试试看如果监控一个行为异常的goroutine，会发生什么：

```
log.SetOutput(os.Stdout)
log.SetFlags(log.Ltime | log.LUTC)

doWork := func(done <-chan interface{}, _ time.Duration) <-chan interface{} {

    log.Println("ward: Hello, I'm irresponsible!")

    go func() {
        <-done // 1
        log.Println("ward: I am halting.")
    }()
    return nil
}

doWorkWithSteward := newSteward(4*time.Second, doWork) // 2

done := make(chan interface{})
time.AfterFunc(9*time.Second, func() { // 3
    log.Println("main: halting steward and ward.")
    close(done)
})

for range doWorkWithSteward(done, 4*time.Second) { // 4
}

log.Println("Done")
```

1. 可以看到这个goroutine什么都没干，持续阻塞等待被取消，它同样不会发出任何表明自己正常信号。
2. 这里开始建立被监控的例程，其4秒后会超时。
3. 这里我们9秒后向done通道发出信号停止整个程序。
4. 最后，我们启动监控器并在其心跳范围内防止示例停止。

这会输出：

```
18:28:07 ward: Hello, I'm irresponsible!
18:28:11 steward: ward unhealthy; restarting 18:28:11 ward: Hello, I'm irresponsible!
18:28:11 ward: I am halting.
18:28:15 steward: ward unhealthy; restarting 18:28:15 ward: Hello, I'm irresponsible!
18:28:15 ward: I am halting.
18:28:16 main: halting steward and ward.
18:28:16 ward: I am halting.
18:28:16 Done
```

看起来工作正常。我们的监控器比较简单，除了取消操作和心跳所需信息之外不接收也不返回任何参数。我们可以用闭包强化一下：

```
doWorkFn := func(done <-chan interface{}, intList ...int) (startGoroutineFn, <-chan interface{}) {//1

    intChanStream := make(chan (<-chan interface{}))//2
    intStream := bridge(done, intChanStream)

    doWork := func(done <-chan interface{}, pulseInterval time.Duration) <-chan interface{} {//3

        intStream := make(chan interface{})//4
        heartbeat := make(chan interface{})

        go func() {
            defer close(intStream)
            select {
            case intChanStream <- intStream://5
            case <-done:
                return
            }

            pulse := time.Tick(pulseInterval)

            for {
            valueLoop:
                for _, intVal := range intList {
                    if intVal < 0 {
                        log.Printf("negative value: %v\n", intVal)//6
                        return
                    }

                    for {
                        select {
                        case <-pulse:
                            select {
                            case heartbeat <- struct{}{}: default:
                            }
                        case intStream <- intVal:
                            continue valueLoop
                        case <-done:
                            return
                        }
                    }
                }
```

```
            }
        }()
        return heartbeat
    }
    return doWork, intStream
}
```

1. 我们将监控器关闭的内容放入返回值，并返回所有监控器用来交流数据的通道。
2. 我们建立通道的通道，这是我们在前面章节中"bridge"模式的应用。
3. 这里我们建立闭包控制监控器的启动和关闭。
4. 这是各通道与监控器交互数据的实例。
5. 这里我们向起数据交互作用的通道传入数据。
6. 这里我们返回负数并从goroutine返回以模拟不正常的工作状态。

由于我们可能会启动监控器的多个副本，因此我们使用"bridge"模式来帮助向doWorkFn的调用者呈现单个不间断的通道。通过这样的方式，我们的监控器可以简单地通过组成模式而变得任意复杂。让我们看看如何调用：

```
log.SetFlags(log.Ltime | log.LUTC)
log.SetOutput(os.Stdout)

done := make(chan interface{})
defer close(done)

doWork, intStream := doWorkFn(done, 1, 2, -1, 3, 4, 5) //1
doWorkWithSteward := newSteward(1*time.Millisecond, doWork) //2
doWorkWithSteward(done, 1*time.Hour) //3

for intVal := range take(done, intStream, 6) { //4
    fmt.Printf("Received: %v\n", intVal)
}
```

1. 这里我们调用该函数，它会将传入的不定长整数参数转换为可通信的流。
2. 在这里，我们创建了一个检查doWork关闭的监视器。我们预计这里会极快的进入失败流程，所以将监控时间设置为一毫秒。
3. 我们通知 steward 开启监测。
4. 最后，我们使用该管道，并从intStream中取出前六个值。

这会输出：

```
Received: 1
23:25:33 negative value: -1
Received: 2
23:25:33 steward: ward unhealthy; restarting Received: 1
23:25:33 negative value: -1
Received: 2
23:25:33 steward: ward unhealthy; restarting Received: 1
23:25:33 negative value: -1
Received: 2
```

我们可以看到监控器发现错误并重启。你可能还会注意到我们只接收到了1和2，这证明了重启功能正常。如果你的系统对重复值很敏感，一定要考虑对其进行处理。你也可以考虑在一定次数的失败后退出。比如在这样的位置：

```
valueLoop:
for _, intVal := range intList {
// ...
}
```

稍作修改：

```
valueLoop:
for {
    intVal := intList[0]
    intList = intList[1:]
    // ...
}
```

尽管我们依然停留在返回的无效负数上，尽管我们的监控器将继续失败，但这会记录在重新启动前的位置，你可以在这个思路上扩展。

使用这样的方式可以确保你的系统保持健康，此外，相信系统崩溃的减少也能大幅度降低开发过程中猝死的几率。

本章小结

在本章中，我们介绍了一些在日常开发中保持系统稳定性和可读性的方法。本章还演示了Go的并发基元如何创建更高阶的抽象。凡事皆有利弊，这些模式也可能会更麻烦，或降低系统健壮性，需要使用者仔细权衡。

在最后一章中，我们将探索Go的一些运行时的内部结构，以帮助你深入理解事情的工作方式。 我们还将探索一些有用的工具，使开发和调试变得更容易一些。

Goroutines和Go运行时

在使用Go进行开发时，直接利用并发是很有趣的，因为这种语言让它变得如此简单。我很少需要了解运行时如何将所有内容联系在一起。尽管如此，有些时候了解这些信息是有益处的。第2章中讨论的所有内容都是由运行时实现的，所以值得花点时间来看看运行时的工作方式。

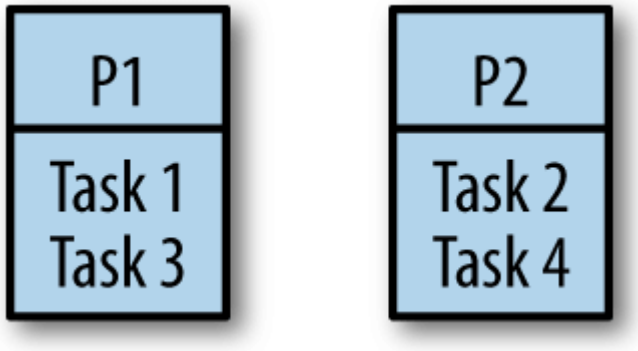
Go运行时为你做的所有事情中，生成和管理goroutines可能是对你和你的软件最有利的一件事。谷歌有将计算机科学理论和白皮书运用于工作的历史，所以Go包含来自学术界的一些想法并不令人惊讶。令人惊讶的是每个goroutine背后的复杂程度。Go使用了一些强大的算法，使你的程序更加高效，同时Go抽象出了这些细节，为开发人员提供了非常简单的接口。

任务调度

就像我们在”Goroutines”章节中提到了，Go语言会帮你处理goroutine到系统线程上。它使用的算法被称为工作窃取策略(work stealing strategy)。这是什么意思？

首先，让我们看看在许多处理器之间共享工作的天然策略，有时称为公平调度。为了确保所有处理器的平均利用率，我们可以在所有可用处理器之间平均分配负载。想象一下，有n个处理器和x个任务需要执行。在公平调度策略中，每个处理器都会得到x/n个任务：

```
<Schedule Task 1>
<Schedule Task 2>
<Schedule Task 3>
<Schedule Task 4>
```



不幸的是，这种方法存在问题。 如果你还记得“Goroutines”章节中我们提到Go使用fork-join模型来并发建模。在fork-join范例中，任务可能依赖于另一个，并且事实证明，在处理器之间分裂它们可能会导致其中一个处理器未充分利用。不仅如此，它还可能导致局部性缓存较差，因为在其他处理器上调度需要相同的数据任务。我们来看一个例子。

考虑一个程序，可以产生前面所述的工作分配。 如果第2项任务比第1项和第3项结合需要更长的时间，会发生什么？

Time	P1	P2
	T1	T2
n+a	T3	T2
n+a+b	(idle)	T4

无论a和b之间的时间有多久，处理器一会闲置。

如果任务之间存在相互依存关系，如果分配给一个处理器的任务需要分配给另一个处理器的任务的结果，会发生什么情况？ 例如，如果任务一依赖任务4呢？

Time	P1	P2
	T1	T2
n+a	(blocked)	T2
n+a+b	(blocked)	T4
n+a+b+c	T1	(idle)
n+a+b+c+d	T3	(idle)

在这种情况下，处理器1完全空闲，而任务2和4正在计算中。 虽然处理器1在任务1中被阻塞，处理器2在任务2中被占用，但处理器1可能已经在处理任务4以解除其自身阻塞。