第 5 章

并发

欢迎来到第 5 章！在学习 Go 的过程中，你已经取得了很大的进步，现在你已经可以用 Go 实现简单和中等复杂的应用程序了。然而，要真正掌握 Go 并解锁其全部潜能，你还需要熟悉现代计算的基石——并发。通过让一台计算机同时做多件事，或者通过在多个执行流之间切换，你可以完成过去的计算机永远无法完成的任务。

阅读本章后，可以得到以下问题的答案。

- 什么是线程和进程？
- 并发和并行之间的区别是什么？
- 绿色线程和用户空间线程是如何工作的？
- 为什么 Goroutine(Go 协程)比线程更好？什么时候不应该使用它们？
- 如何使用通道在不同的 Goroutine 之间进行通信？
- 一个 Goroutine 如何做到同时等待来自多源的输入？

5.1 并发、线程和并行

并发是现代计算的基础，能使计算机同时执行独立的指令流，从而实现同时运行多个应用程序的功能。

想象一下，如果计算机一次只能做一件事，这将是令人难以接受的限制，此时甚至不具备图形界面的技术条件，我们只能使用命令行界面来处理所有事情。

在一些操作系统上，并发仅仅意味着 CPU 会非常快速地从一个进程的运行指令切换到另一个进程的运行指令，使它看起来好像同时执行了多件事情。

然而，这种技术有其局限性。如果试图运行需要大量计算的多个进程，并且它们都需要同时运行，那么 CPU 将花费大量的时间来运行这些进程，这是因为 CPU 不仅需要对每个进程进行计算，还需要在进程之间不断切换，这本身就需要一定的计算能力。

这就是现代 CPU 需要“内核”的原因。CPU 拥有的内核越多，它可以同时运行的独立执行流就越多。这意味着，在同一个“时钟周期”内，当一个进程的指令在一个“内核”上执行时，另一个完全不同的进程可以在另一个“内核”上运行它的指令。

并发的真正价值在新旧系统共存时期开始显现。想象一下，拥有多个物理内核就能够同时运行多个不同的进程，并且能够让它们在进程之间快速切换，这样我们就可以构建像 GUI 这样的系统了。这就是现代计算系统的体系架构——从 CPU 到操作系统，甚至到编程语言。

我们知道，单个 CPU 和操作系统实例可以同时运行多个（甚至数千个）不同的进程。但是，如果一个独立的进程需要同时运行它自己的多个执行流，又该怎么办呢？假设有一个网页浏览器，你打开了多个选项

卡。从技术上讲，浏览器是一个单独的进程，但有多个选项卡都在运行自己的代码，这就是线程发挥作用的地方。线程是进程的轻量级部分，有自己的“堆栈指针”和寄存器。一个进程可以产生数百个线程，操作系统会将这些线程映射到 CPU，就像对其他进程一样。

需要注意的是，我们给出的浏览器这个例子并不是一成不变的，也有一些值得注意的例外。例如，在 Chrome 浏览器上，每个标签都有自己成熟的进程，这使得 Chrome 更容易达到性能基准，但这是非常耗费资源的，这就是 Chrome 在笔记本计算机上消耗的电量和内存比 Safari 和 Firefox 等其他使用线程的浏览器更多的原因。

你现在可能在想“我们可以运行数千个线程和进程，而我们的操作系统可以有效地处理这一切吗?”不幸的是，当一切看起来好得令人难以置信时，它往往就不是真的了——使用可伸缩的线程有很大的限制性。

可以看到，与成熟的进程相比，线程可能相对轻量级，但是启动并加入它们的代价仍然非常高，它需要一个完整的系统调用，并与内核(kernel)进行交互。这意味着在需要数千个线程的情况下，只启动数十或数百个线程通常更有效，然后让每个线程执行多个任务。

对于今天的大多数软件而言，实际上有两种线程：原生线程(native thread)，这是我们已经讨论过的类型(在内核级别处理)；绿色线程(green thread)，我们稍后将讨论它(在用户级别处理)。

考虑一下 Web 服务器。一个系统可能会同时收到数千甚至数万个需要处理的请求。在这种情况下，线程是不够的，因为线程太多，所以负责在线程之间切换的调度器将执行比线程本身更长的时间。

这就是 Web 服务器使用“绿色线程”的原因，这是由服务器自己处理的，因此，在单个线程中，我们可以在代码的某处实现多个不同绿色线程之间的切换。因为我们对“上下文切换”发生的位置有更细粒度的控制，所以 Web 服务器类型的应用程序性能通常会得到提升。

Go 有一个类似概念的独特实现，它遵循一个称为 M：N 线程模型的范式，也称为混合线程。M 代表用户模式或绿色线程，N 代表内核模式或原生线程，类似于将完全内核模式（1：1）和完全用户模式（N：1）线程结合后所得到的结果。

混合线程的复杂性在于，你需要修改内核级和用户级的代码才能使系统正常工作。然而，大多数现代内核都已经内置了我们需要做的所有准备工作。我们只需要通过用户级软件来处理绿色线程即可，而这正是由 Go 提供的。

5.2 Goroutine（Go 协程）

Go 有一个 Goroutine 的概念，它可以将许多不同的执行流（我们可以直接称之为函数）映射到一个本地线程池。Goroutine 是非常轻量级的，它们只需要 2KB 的内存空间，而原生线程需要 1MB 的内存空间再加上一个保护页面，是 Goroutine 的 500 倍。

在深入研究 Goroutine 的工作原理之前，让我们先看一个示例。

创建 Goroutine 的语法非常简单：像往常一样调用函数，只需要在调用之前添加标记“go”，例如下面的代码。

代码清单 5.1　一个简单的单线程程序求平方数

```
package main

import (
    "fmt"
)

func squareIt(x int) {
    fmt.Println(x * x)
```

```
}
func main() {
    squareIt(2)
}
```

可以想到，当编译并运行这段代码时，它会打印出数字 4。但是，如果 squareIt 是一个需要长时间运行的操作，并希望在单独的 Goroutine 中运行它，那么主函数就可以在 squareIt 运行时继续做它需要做的其他事情，这该怎么办呢？你所需要做的只是在主函数的 squareIt 前面加上“go”标记。

```
func main() {
    go squareIt(2)
}
```

如果编译程序，你会发现没有错误。然而，当运行代码时，你将看到它什么都没有打印（大多数系统通常都是这样），这是为什么呢？

当运行 Goroutine 时，我们将 squareIt 的执行置于“后台”，另一种说法是，它是“与我们当前运行的 Goroutine 分开的一个程序”。然后，就在我们这样做的时候，main 函数结束了。main 函数在主 Goroutine 中执行，该 Goroutine 执行该程序所在进程的主线程。当 main 函数结束时，整个程序将会退出，这意味着在其他 Goroutine 打印出平方数之前，进程就退出了。

因此我们需要告诉主线程，它需要在退出之前等待，这样另一个 Goroutine 才有机会完成工作。这可以通过令 main 函数“休眠”（空闲）来完成，例如只休眠 1ms。为此，首先需要导入 time 包，并将 import 修改如下。

```
import (
```

```
    "fmt"
    "time"
)
```

然后在 main 函数中增加一个 Sleep 函数调用。

```
func main() {
    go squareIt(2)
    time.Sleep(1 * time.Millisecond)
}
```

现在，如果编译并运行程序，则它会像往常一样输出 4。如果没有，可以试着增加休眠时间，处理器可能需要更多的时间来进行计算。

同样，你应该记住一件事：在现实情境中，我们实际上并不是像上面这样等待 Goroutine 完成的，我们可以使用一个称为通道（channel）的功能，这一点稍后会讲到。

就效率而言，Goroutine 背后的思想非常巧妙。当程序启动时，Go 只运行一个 Goroutine，因此只有一个原生线程在运行。当同时运行多个 Goroutine 时，Go 会将每个 Goroutine 映射到多个线程。如果它需要启动更多的线程来尝试并行运行 Goroutine，而不是仅仅并发运行，它可能会这样做——在本例中，当你运行 go squareIt(2)时，它将启动另一个线程。

因此，就像操作系统能够将多个线程映射到 CPU 内核一样，Go 运行时能够将多个 Goroutine 映射到原生线程。并且，在默认情况下，原生线程可以在不同的 CPU 内核之间“移动”（例如，内核 1 在几分钟前正在执行线程 1，但经过几次上下文切换后，内核 2 现在正在执行线程 1），这可以用特定标志锁定，告诉 kernel 只在特定内核（core）上运行一个线程。类似地，你可以告诉 Go“将此 Goroutine 固定在特定线程上”。同样，对于这个 kernel 特性，大多数 kernel（Linux、BSD、XNU 等）都支

持它。

假设当 Go 在线程上多路复用 Goroutine 时，在它处理调度时，某个 Goroutine 会阻塞，因为它需要执行系统调用，这通常是程序中最慢的部分。在这种情况下，Go 将创建一个新线程并将 Goroutine 转移到该线程中，这意味着当执行一个长时间运行的操作时，我们不需要空闲 CPU 来等待，Go 可以保持其他 Goroutine 运行。此外，该操作只需要存储 3 个寄存器，这表明它是超轻量级的，你的程序可以运行得非常快。当你将其与需要存储每个寄存器（在 x86-64 上超过 50 个寄存器）的线程之间的切换成本进行比较时，你就会明白 Goroutine 的价值。

Goroutine 有一个缺点，那就是当生成一个新的 Goroutine 时，你不能够再通过其他 Goroutine 对它进行任何控制。例如，Goroutine 不能“终止”另一个 Goroutine 或与另一个 Goroutine“合并”，而只能在返回（到达函数的末尾）时“完成执行”或“退出”。

这一限制还意味着，作为 Goroutine 调用的函数不能返回任何值，必须使用通道（channel）来共享信息。

至此，你应该已经很好地理解了 Goroutine 的工作原理。下面让我们看一下使用 Goroutine 实现并发的另一个组成部分——通道。

5.3　通道

利用通道，我们可以以非常高性能的方式将数据发送给不同的 Goroutine。此外，它们的语法非常简单，就像使用 Goroutine 一样，因此代码不会因为 Goroutine 之间的通信而变得更复杂。

让我们将前面求数的平方的示例扩展为使用通道进行通信。更新 squareIt 函数以接收两个通道的参数，一个通道是让这个函数知道它需要对哪些数求平方，另一个通道是 Goroutine 在对这些数求平方后放置

结果的地方。

下面是 squareIt 函数的代码。

```
func squareIt(inputChan, outputChan chan int) {
    for x := range inputChan {
    outputChan <-x * x
    }
}
```

在查看主函数之前，让我们进一步解读一下 squareIt 函数。正如你所看到的，通道的类型标注是唯一的，因为它由两个标记（chan int）组成，而大多数类型标注只有一个标记 int。然而，标注的含义并不模糊——告诉 Go 我们想要一个通道，并且该通道中的数据是 int 类型的。你可以在通道中存储任何类型的数据，包括指针和自定义结构体。

该函数有两个独立的通道：一个输入通道和一个输出通道。在函数内部，可以循环遍历输入通道。当通道中没有值时，for 循环将会阻塞并开始等待新的数据；当它找到一个数据时，for 循环将继续迭代，并在结束时跳回循环的顶部。

在 for 循环中，我们只需要做一件简单的事情：对从输入通道获取的整数 x 求平方，然后使用“<-”操作符将该平方值放入输出通道。

可以看出，将此函数修改为使用通道非常简单，main 函数却需要做更多的修改。

```
func main() {
    inputChannel := make(chan int)
    outputChannel := make(chan int)
    go squareIt(inputChannel, outputChannel)
    for i := 0; i <10; i++ {
        inputChannel <-i
    }
    for i := range outputChannel {
```

```
            fmt.Println(i)
        }
    }
```

让我们再看一遍主函数。

我们从前两行代码开始，这两行代码负责使用 make 函数创建输入通道和输出通道，第 2 章曾使用这个函数创建数组。

```
    inputChannel := make(chan int)
    outputChannel := make(chan int)
```

然后，在一个单独的 Goroutine 中运行 squareIt 函数。

```
    go squareIt(inputChannel, outputChannel)
```

现在，函数在后台运行，由于 for 循环的存在，因此它正在等待数据进入输入通道。

为了向输入通道提供数据，我们仅从 0 循环到 9，并在每次迭代中将数放入输入通道。

```
    for i := 0; i <10; i++ {
        inputChannel <-i
    }
```

此时，因为已经将数据放入了通道，所以我们期望 squareIt 函数获取数据并将其放入输出通道。然后，我们需要通过循环输出通道从 squareIt 函数获取数据并将其打印出来，以便看到输出。

```
    for i := range outputChannel {
        fmt.Println(i)
    }
```

如果你一直在关注这一点，可能已经发现了程序中的一个 bug。同时，如果你已经了解了通道的工作原理，那么你可能已经注意到了程序中的另一个 bug。不过，让我们先来看看这段代码，以确定出现了什么 bug。

下面是最终的代码。

代码清单 5.2　求平方应用程序的并发实现

```
package main

import (
    "fmt"
)

func squareIt(inputChan, outputChan chan int) {
    for x := range inputChan {
        outputChan <- x * x
    }
}

func main() {
    inputChannel := make(chan int)
    outputChannel := make(chan int)
    go squareIt(inputChannel, outputChannel)
    for i := 0; i <10; i++ {
        inputChannel <- i
    }
    for i := range outputChannel {
        fmt.Println(i)
    }
}
```

编译程序，注意到没有出现错误。然而，当运行代码时，在大多数系统中会出现这样的错误：

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