# Go语言并发编程

本章重点为大家介绍如下的内容：

* 什么是并发与并行
* 进程和线程
* 轻量级线程——coroutine
* go语言中的协程——goroutine
* 多个goroutine间通信的通道channel
* time包中跟通道相关的函数
* 缓冲通道和定向通道
* select分支语句
* sync包中的WaitGroup同步等待组
* 共享数据安全问题及sync包中的Mutex互斥锁
* sync包中的RWMutex读写互斥锁及Cond条件变量

## 并发与并行

### Go是并发语言，而不是并行语言

在讨论如何在Go中进行并发处理之前，我们首先必须了解什么是并发，以及它与并行性有什么不同。Go是并发语言，而不是并行语言

并发性Concurrency是指把任务在不同的时间点交给处理器进行处理。在同一个时间点，任务不会同时运行。并发性是同时处理许多事情的能力。

举个例子，打电话和吃饭。吃饭时，电话来了，需要停止吃饭去接电话。电话接完后回来继续吃饭，这个过程是并发执行。

并行性parallelism是把一个任务分配给每一个处理器独立完成。在同一个时间点，任务一定是同时运行。并行就是同时做很多事情。这听起来可能与并发类似，但实际上是不同的。

用同样打电话和吃饭的例子去理解。吃饭时，电话来了，边吃饭边接电话，这个过程是并行执行。这就是所谓的并行性(parallelism)。

Go在Gomaxprocs数量与任务数量相等时，可以做到并行执行，但是一般情况下都是并发执行。

并发性和并行性——一种技术上的观点。

假设我们正在编写一个web浏览器。web浏览器有各种组件，其中两个是web页面呈现区域和从internet下载文件的下载器。当这个浏览器运行在单核处理器中时，处理器将在浏览器的两个组件之间进行上下文切换。它可能会下载一个文件一段时间，然后它可能会切换到呈现用户请求的网页的html，这就是所谓的并发性。并发进程从不同的时间点开始，它们的执行周期重叠。在这种情况下，下载和呈现从不同的时间点开始，它们的执行重叠。

假设同一浏览器运行在多核处理器上。在这种情况下，文件下载组件和HTML呈现组件可能同时在不同的内核中运行，这就是所谓的并行性。并行性Parallelism不会总是导致更快的执行时间。这是因为并行运行的组件可能需要相互通信。例如，在我们的浏览器中，当文件下载完成时，应该将其传递给用户，比如使用弹出窗口。这种通信发生在负责下载的组件和负责呈现用户界面的组件之间。当组件在多个内核中并行运行时，这种通信开销很高。因此，并行程序并不总是导致更快的执行时间!这种通信开销在并发concurrent 系统中很低。在单CPU上，是os代码强制把一个进程或者线程挂起，换成另外一个，所以实际上是并发的，只是“概念上的并行”。在现在的多核的cpu上，线程可能是“真正并行的”。

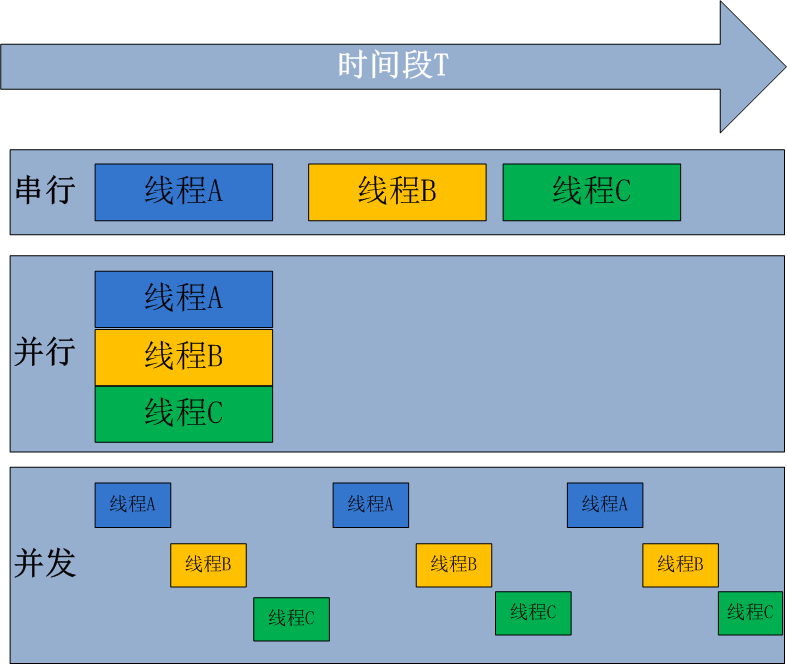


图 1

### 什么是并发？

并发指在同一时间内可以执行多个任务。

并发编程含义广泛，可以是多线程编程，也可以是多进程编程，还可以是编写分布式程序。并发编程的思想来自于多任务操作系统，它允许同时运行多个程序。

早期单用户操作系统，任务是一个接一个运行，单个任务的执行完全是串行的。只有在一个任务运行完成之后，另一个任务才会被读取；多任务操作系统则允许终端用户同时运行多个程序。当一个程序暂时不需要使用CPU的时候，系统会把该程序挂起或中断，以使其他程序可以使用CPU。最早支持并发编程的计算机编程语言是汇编语言。

串行程序特指只能被顺序执行的指令列表，并发程序则是可以被并发执行的多个串行程序的综合体。因此并发程序内部会被划分为多个部分，每个部分都可以看做一个串行程序。串行程序中的所有代码的先后顺序都是固定的，而并发程序中只有部分代码是有序的。这一特性被称为不确定性。这导致并发程序每次运行的代码执行路径都是不同的，即便输入数据相同也会如此。

Go语言通过编译器运行时runtime，从语言上支持并发的特性。Go语言的并发通过Goroutine特性完成。

## 进程和线程

### 基本概念

进程（process）和线程（thread）是操作系统级别的两个基本概念。计算机的核心是CPU，它承担了所有的计算任务。它就像一座工厂，时刻在运行。

进程就好比工厂的车间，它代表CPU所能处理的单个任务。进程是一个容器。线程就好比车间里的工人。一个进程可以包括多个线程。线程是容器中的工作单位。一个进程的内存空间是共享的，每个线程都可以使用这些共享内存。就如同车间的空间是工人们共享的，许多房间是每个工人都可以进出的。车间的每个房间大小不同，有些房间最多只能容纳一人，比如厕所。里面有人的时候，其他人就不能进去了。这代表一个线程使用某些共享内存时，其他线程必须等它结束，才能使用这一块内存。防止他人进入的简单方法，就是门口加一把锁。先到的人锁上门，后到的人看到上锁，就在门口排队，等锁打开再进去。这就叫"互斥锁"（Mutual exclusion，缩写 Mutex），防止多个线程同时读写某一块内存区域。还有些房间，可以同时容纳n个人，比如厨房。如果人数大于n，多出来的人只能在外面等着。这好比某些内存区域，只能供给固定数目的线程使用。这时的解决方法，就是在门口挂n把钥匙。进去的人就取一把钥匙，出来时再把钥匙挂回原处。后到的人发现钥匙架空了，就知道必须在门口排队等着了。这种做法用来保证多个线程不会互相冲突。

进程是程序的一次动态执行过程；进程需要经历从代码加载、代码执行到执行完毕的一个完整过程，这个过程也就是进程本身从产生、发展到消亡的过程；每个运行中的程序就是一个进程；进程是一个实体，每一个进程都有它自己的地址空间，一般情况下，包括文本区域（text region）、数据区域（data region）和堆栈（stack region）；进程在系统中独立存在，拥有自己独立的资源，多个进程可以在单个处理器上并发执行且互不影响；操作系统可以同时执行多个任务，每个任务就是进程。

进程有三个状态：就绪、运行和阻塞。就绪状态其实就是获取了除CPU之外的所有资源，只要处理器分配资源就可以马上执行。就绪状态有排队序列什么的，排队原则不再赘述；运行态就是获得了处理器分配的资源，程序开始执行；阻塞态，当程序条件不够时候，需要等待条件满足时候才能执行，如等待i/o操作时候，此刻的状态就叫阻塞态。

线程是CPU的基本调度单位；线程是在进程之后发展出来的概念，线程也叫轻量级进程；通常一个进程包含若干个线程，当然一个进程中至少有一个线程，不然没有存在的意义；线程可以利用进程所拥有的资源，在引入线程的操作系统中，通常都是把进程作为分配资源的基本单位，而把线程作为独立运行和独立调度的基本单位；由于线程比进程更小，基本上不拥有系统资源，故对它的调度所付出的开销就会小得多，能更高效的提高系统多个程序间并发执行的程度。

例如：杀毒软件程序是一个进程，该程序在为计算机进行检查的同时，可以清理垃圾文件，并进行日志记录。检查计算机、清理垃圾文件、日志记录这就是线程。

例如：启动word文档就是运行了一个进程，文字录入时，可以进行拼写错误检查，并使首字母大写。文字录入、拼写错误检查、首字母大写就属于word操作进程中的线程。

### 进程与线程的区别

进程和线程的主要差别在于它们是不同的操作系统资源管理方式。

进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。

一个程序至少有一个进程，一个进程至少有一个线程。线程的划分尺度小于进程，使得多线程程序的并发性高。进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。

线程在执行过程中与进程还是有区别的。每个独立的进程有一个程序运行的入口、顺序执行程序。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。

线程执行开销小，但不利于资源的管理和保护；而进程正相反。同时，线程适合于在SMP(多核处理机)机器上运行，而进程则可以跨机器迁移。

### 相关背景知识

进程和线程都是一个时间段的描述，是CPU工作时间段的描述，不过是颗粒大小不同。

CPU+RAM+各种资源（比如显卡、GPU）构成我们的电脑，但是电脑的运行，实际就是CPU和相关寄存器以及RAM之间的事情。

一个最最基础的事实：CPU太快，太快，太快了，寄存器仅仅能够追的上他的脚步，RAM和别的挂在各总线上的设备完全是望其项背。那当多个任务要执行的时候怎么办呢？轮流着来?或者谁优先级高谁来？不管怎么样的策略，一句话就是在CPU就是轮流着来。

一个必须知道的事实：执行一段程序代码，实现一个功能的过程，当得到CPU的时候，相关的资源必须也已经就位，就是显卡啊，GPU什么的必须就位，然后CPU开始执行。这里除了CPU以外所有的就构成了这个程序的执行环境，也就是我们所定义的程序上下文。当这个程序执行完了，或者分配给他的CPU执行时间用完了，那它就要被切换出去，等待下一次CPU的临幸。在被切换出去的最后一步工作就是保存程序上下文，因为这个是下次他被CPU临幸的运行环境，必须保存。

串联起来的事实：前面讲过在CPU看来所有的任务都是一个一个的轮流执行的，具体的轮流方法就是：先加载程序A的上下文，然后开始执行A，保存程序A的上下文，调入下一个要执行的程序B的程序上下文，然后开始执行B,保存程序B的上下文。。。。

进程就是包括上下文切换的程序执行时间总和 = CPU加载上下文 + CPU执行 + CPU保存上下文

线程是什么呢？

进程的颗粒度太大，每次都要有上下的调入，保存，调出。如果我们把进程比喻为一个运行在电脑上的软件，那么一个软件的执行不可能是一条逻辑执行的，必定有多个分支和多个程序段，就好比要实现程序A 实际分成 a，b，c等多个块组合而成。那么这里具体的执行就可能变成：程序A得到CPU -> CPU加载上下文，开始执行程序A的a小段，然后执行A的b小段，然后再执行A的c小段，最后CPU保存A的上下文。

这里a，b，c的执行是共享了A的上下文，CPU在执行的时候没有进行上下文切换的。这里的a，b，c就是线程，也就是说线程是共享了进程的上下文环境的更为细小的CPU时间段。

操作系统的设计，可归结为三点：

（1）以多进程形式，允许多个任务同时运行；

（2）以多线程形式，允许单个任务分成不同的部分运行；

（3）提供协调机制，一方面防止进程之间和线程之间产生冲突，另一方面允许进程之间和线程之间共享资源。

## 轻量级线程——协程coroutine

### 什么是协程？

协程（Coroutine），又称为微线程。是一种比线程更加轻量级的存在。正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。协程最初在1963年被提出。

协程不是进程或线程，其执行过程更类似于子程序，或者说不带返回值的函数调用。

最重要的是，协程不是被操作系统内核所管理，而完全是由程序所控制。这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。Coroutine是编译器级的，Process和Thread是操作系统级的。

协程与传统的系统级线程和进程相比，协程的最大优势在于其轻量级，可以轻松创建上百万个，而不会导致系统资源衰竭；线程和进程通常最多也不能超过一万个。这也是协程叫做轻量级线程的原因。

协程的执行效率极高。因为子程序切换不是线程切换，而是由程序自身控制，因此，没有线程切换的开销。和多线程比，线程数量越多，协程的性能优势就越明显。

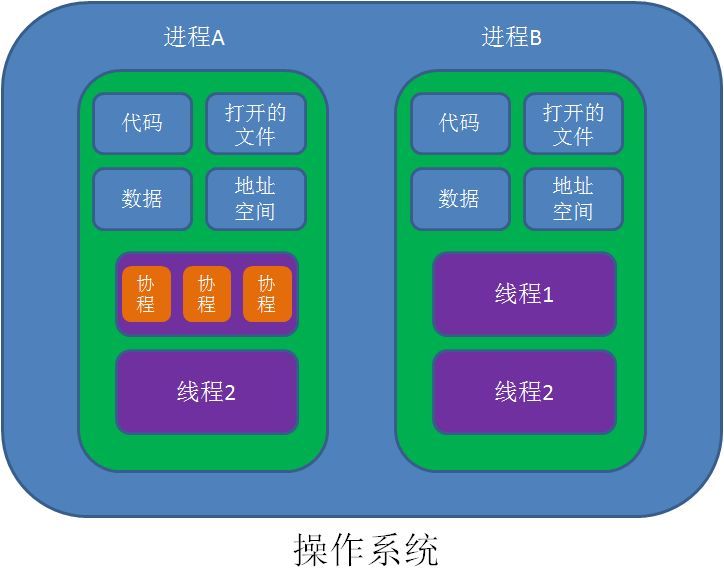


图 2

### 协程的应用

有哪些编程语言应用到了协程呢？

Lua语言

Lua从5.0版本开始使用协程，通过扩展库coroutine来实现。

Python语言

python可以通过 yield/send 的方式实现协程。在python 3.5以后，async/await 成为了更好的替代方案。

Go语言

Go语言对协程的实现非常强大而简洁，可以轻松创建成百上千个协程并发执行。

Java语言

Java语言并没有对协程的原生支持，但是某些开源框架模拟出了协程的功能，例如：Kilim框架。

## Go语言中协程——goroutine

### 什么是Goroutine？

在执行多任务时，为了保证每个任务能及时被分配到CPU上运行处理，同时避免多个任务频繁地在线程间切换执行而损失效率，常使用线程池；但是如果面对随时随地可能发生的并发和线程处理需求，线程池不是非常直观和方便；

使用者分配足够多的任务，系统能自动帮助使用者把任务分配到CPU上，让这些任务尽量并发运行。这种机制在GO语言中被称为Goroutine。Goroutine类似于线程，Goroutine由GO程序运行时runtime调度和管理，GO程序会智能地将Goroutine中的任务合理地分配给每个CPU。

Go程序从main包的main()函数开始，在程序启动时，Go程序会为main()函数创建一个默认的Goroutine。

### Goroutine在线程上的优势

Go中使用Goroutine来实现并发concurrently。Goroutine是与其他函数或方法同时运行的函数或方法。

与线程相比，Goroutine非常便宜。它们只是堆栈大小的几个kb，堆栈可以根据应用程序的需要增长和收缩，而在线程的情况下，堆栈大小必须指定并且是固定的。由于创建Goroutine的成本很小。因此，Go应用程序可以并发运行数千个Goroutine。

当使用Goroutine访问共享内存时，通过设计的通道可以防止竞态条件发生。通道可以被认为是Goroutine通信的管道。

Coroutine与Goroutine比较：

都可以将函数或者语句在独立的环境中运行，但是他们之间存在两点不同：

（1）goroutine可能发生并行执行；但是coroutine始终顺序执行；goroutine可能发生在多线程环境下，coroutine始终发生在单线程，coroutine程序需要主动交出控制权，宿主才能获得控制权并将控制权交给其他coroutine。coroutine的运行机制属于协作式任务处理，应用程序在不需要使用CPU时，需要主动交出CPU使用权。如果开发者无意间让应用程序长时间占用CPU，操作系统也无能为力，计算机很容易失去响应或者死机。

goroutine属于抢占式任务处理，已经和现有的多线程和多进程任务处理非常类似。应用程序对CPU的控制最终需要有操作系统来管理，操作系统如何发现一个应用程序常时间占用CPU，那么用户有权终止这个任务。

（2）goroutine间使用channel通信；coroutine使用yield和resume操作。

### 使用普通函数创建goroutine

在函数或方法调用前面加上关键字go，将会同时运行一个新的Goroutine。

使用go关键字创建Goroutine时，被调用的函数往往没有返回值，如果有返回值也会被忽略。

如果需要在Goroutine中返回数据，必须使用通道channel，通过通道把数据从Goroutine中作为返回值传出。

例：

package main

import (

"fmt"

)

func hello() {

fmt.Println("Hello world goroutine")

}

func main() {

go hello()

//time.Sleep(50 \* time.Microsecond)

fmt.Println("main function")

}

运行结果：

main function

Go程序的执行过程是：创建和启动主goroutine，初始化操作，执行main()函数，当main()函数结束，主goroutine随之结束，程序结束。

被启动的goroutine叫做子goroutine。

如果main的goroutine终止了，程序将被终止，而其他Goroutine将不再运行。换句话说，所有goroutine在main()函数结束时会一同结束。

修改代码

func hello() {

fmt.Println("Hello world goroutine")

}

func main() {

go hello()

time.Sleep(50 \* time.Microsecond)

fmt.Println("main function")

}

运行结果：

Hello world goroutine

main function

在上面的程序中，我们已经调用了时间包的Sleep方法，它会在执行过程中睡觉。在这种情况下，main的goroutine被用来睡觉50毫秒。现在调用go hello()有足够的时间在main Goroutine终止之前执行。这个程序首先打印Hello world goroutine，等待50毫秒，然后打印main函数。

例2：

func main() {

go running()

var input string

fmt.Scanln(&input)

}

func running() {

var times int

for {

times++

fmt.Println("tick", times)

time.Sleep(time.Second)

}

}

程序运行效果。控制台不断输出tick，同时还可以接收用户输入。两个环节同时运行。

该案例中，Go程序在启动时，运行时runtime默认为main()函数创建一个goroutine。在main()函数的goroutine执行到go running()语句时，归属于running()函数的goroutine被创建，running()函数开始在自己的goroutine中执行。

此时，main()继续执行，两个goroutine通过GO程序的调度机制同时运行。

### 使用匿名函数创建goroutine

go关键字后也可以是匿名函数或闭包。

例：

func main() {

go func() {

var times int

for {

times++

fmt.Println("tick" , times)

time.Sleep(time.Second)

}

}()

var input string

fmt.Scanln(&input)

}

### 启动多个Goroutines

func numbers() {

for i := 1; i <= 5; i++ {

time.Sleep(250 \* time.Millisecond)

fmt.Printf("%d ", i)

}

}

func alphabets() {

for i := 'a'; i <= 'e'; i++ {

time.Sleep(400 \* time.Millisecond)

fmt.Printf("%c ", i)

}

}

func main() {

go numbers()

go alphabets()

time.Sleep(3000 \* time.Millisecond)

fmt.Println("\n main terminated")

}

运行结果：

1 a 2 3 b 4 c 5 d e

main terminated

时间轴分析：

### 调整并发的运行性能Gomaxprocs

在Go程序运行时runtime实现了一个小型的任务调度器。这套调度器的工作原理类似于操作系统调度线程，Go程序调度器可以高效地将CPU资源分配给每一个任务。

传统逻辑中，开发者需要维护线程池中的线程与CPU核心数量的对应关系。在Go语言中可以通过runtime.GOMAXPROCS()函数做到。

语法为：runtime.GOMAXPROCS(逻辑CPU数量)

逻辑CPU数量有如下几种数值：

<1 ，不修改任何数值；

=1，单核执行；

>1，多核并发执行。

Go1.5版本之前，默认使用单核执行。Go1.5版本开始，默认执行：runtime.GOMAXPROCS(逻辑CPU数量)，让代码并发执行，最大效率地利用CPU。

## 多个goroutine间通信的通道channel

### 通道的概述

单纯地讲函数并发执行没有意义。函数与函数间需要交换数据才能体现出并发执行的意义。虽然可以使用共享内存进行数据交换，但是共享内存在不同goroutine中容易发生竞态问题，必须使用互斥对内存进行加锁，所以会造成性能问题。

Go语言中提倡使用通道channel的方式代替共享内存。也就是说，Go语言中主张，应该通过数据传递来实现共享内存，而不是通过共享内存来实现消息传递。

排队的目的是避免拥堵、插队造成的资源使用和交换过程低效问题。多个goroutine为了争抢数据，势必造成低效，使用队列的方式是最高效的，channel就是一种队列结构。

Go语言中的通道channel是一种特殊的类型。通道像一个传送带或者队列，总是遵循先入先出first in first out的规则，保证收发数据的顺序。通道可以被认为是Goroutine通信的管道。类似于管道中的水从一端到另一端的流动，数据可以从一端发送到另一端，通过通道接收。每个通道都有与其相关的类型。该类型是通道允许传输的数据类型。(通道的零值为nil。nil通道没有任何用处，因此通道必须使用类似于map和slice的方法来定义。

### 声明通道类型

var 通道变量 chan 通道类型

chan类型的空值是nil，声明后需要配合make才能使用。

通道是引用类型，需要使用make进行创建

语法：通道示例 := make(chan 数据类型)

例如：

ch1 := make(chan int) //创建一个整数类型通道

ch2 := make(chan interface{}) //创建一个空接口类型的通道，可以存放任意数据

type Equip struct {/\* 属性 \*/}

ch3 := make(chan \*Equip) //创建一个Equip指针类型的通道，可以存放Equip指针

### 通道发送数据

通道发送使用特殊的操作符"<-"，将数据通过通道发送的语法为：

通道变量 <- 值

通道发送的值可以是变量、常量、表达式或函数返回值等。值的类型必须与ch通道的元素类型一致。

把数据往通道中发送，如果接收方一直没有接收，那么发送操作将持续阻塞。此时所有的goroutine，包括main的goroutine都处于等待状态。

运行会提示报错：fatal error: all goroutines are asleep - deadlock!

死锁deadlock

使用通道时要考虑的一个重要因素是死锁。

如果Goroutine在一个通道上发送数据，那么预计其他的Goroutine应该接收数据。如果这种情况不发生，那么程序将在运行时出现死锁。

类似地，如果Goroutine正在等待从通道接收数据，那么另一些Goroutine将会在该通道上写入数据，否则程序将会死锁。

例：

func main() {

//创建一个空指针型通道

ch := make(chan interface{})

//将0通过通道发送

ch <- 0

//发送字符串

ch <- "StevenWang"

}

### 阻塞

一个通道发送和接收数据，默认是阻塞的。

当一个数据被发送到通道时，在发送语句中被阻塞，直到另一个Goroutine从该通道读取数据。类似地，当从通道读取数据时，读取被阻塞，直到一个Goroutine将数据写入该通道。

这些通道的特性是帮助Goroutines有效地进行通信，而无需像使用其他编程语言中非常常见的显式锁或条件变量。

例：

func main() {

var ch1 chan int

fmt.Println(ch1) //<nil>,

fmt.Printf("%T\n", ch1) //chan int

ch1 = make(chan int)

fmt.Println(ch1) //0xc4200200c0

ch2 := make(chan bool)

go func() {

fmt.Println("子goroutine。。。")

data, ok := <-ch1 //阻塞式，从通道中读取数据

time.Sleep(1 \* time.Second)

fmt.Println("子goroutine从通道中读取到main传来的数据是：", ok, data)

ch2 <- true //向通道中写入数据，表示结束

}()

ch1 <- 100 //阻塞式，main goroutine向通道中写入数据

<-ch2 //目的是防止main goroutine先执行完毕后退出。因为如果main的goroutine终止了，程序将被终止，而其他Goroutine将不再运行。

fmt.Println("main..over")

}

运行结果：

<nil>

chan int

0xc420076060

子goroutine。。。

子goroutine从通道中读取到main传来的数据是： true 100

main..over

### 通道接收数据

通道接收同样使用特殊的操作符"<-"。

通道变量 <- 值

通道收发操作在不同的两个goroutine间进行。接收操作将持续阻塞，直到发送方发送数据。每次接收一个元素。

阻塞接收数据

data := <-ch

执行该语句时将会阻塞，直到接收到数据并赋值给data变量。

阻塞接收数据的完整写法

data , ok := <-ch

data ：表示接收到的数据。未接收到数据时，data为通道类型的零值。

ok：表示是否接收到数据。

通过ok值可以判断当前通道是否被关闭。

<-ch表示接收任意数据，忽略接收的数据。执行该语句时将会阻塞。其目的不在于接收通道中数据，而是为了阻塞goroutine。

循环接收数据，需要配合使用关闭通道。借助普通for循环和for ... range语句循环接收多个元素，遍历通道，遍历的结果就是接收到的数据，数据类型就是通道的数据类型。

普通for循环接收通道数据，需要有break循环的条件；for range会自动判断出通道已关闭，而无需通过判断来终止循环。

循环接收数据的三种方式

func main() {

ch1 := make(chan string)

go sendData(ch1)

//循环接收数据方式1

for {

data := <-ch1

//如果通道关闭，通道中传输的数据则为各数据类型的默认值。chan int 默认值为0，chan string默认值为"" 等。

if data == "" {

break

}

fmt.Println("从通道中读取数据方式1：", data)

}

//循环接收数据方式2

for {

data, ok := <-ch1

//通过多个返回值的形式来判断通道是否关闭，如果通道关闭，则ok值为false。

if !ok {

break

}

fmt.Println("从通道中读取数据方式2：", data)

}

//循环接收数据方式3

//for range循环会自动判断通道是否关闭，自动break循环。

for value := range ch1 {

fmt.Println("从通道中读取数据方式3：", value)

}

}

func sendData(ch1 chan string) {

for i := 1; i <= 10; i++ {

ch1 <- fmt.Sprintf("发送数据%d\n", i)

}

fmt.Println("发送数据完毕。。")

//显式调用close()实现关闭通道

close(ch1)

}

### 关闭通道

发送方如果数据写入完毕，需要关闭通道，用于通知接受方数据传递完毕。一般都是发送方关闭通道。如何判断一个channel是否已经关闭？可以在读取的时候使用多重返回值的方式。如果返回值是false，则表示通道已经被关闭。如果往关闭的通道中写入数据，会报错：panic: send on closed channel。但是可以从关闭后的通道中取数据，不过返回数据默认值和false。

例：

func main() {

//通道关闭后是否可以写入和读取呢？

ch1 := make(chan int)

go func() {

ch1 <- 100

ch1 <- 200

close(ch1)

//ch1 <- 10 //关闭的通道，无法写入数据

}()

data, ok := <-ch1

fmt.Println("main读取数据：", data, ok)

data, ok = <-ch1

fmt.Println("main读取数据：", data, ok)

data, ok = <-ch1

fmt.Println("main读取数据：", data, ok)

data, ok = <-ch1

fmt.Println("main读取数据：", data, ok)

data, ok = <-ch1

fmt.Println("main读取数据：", data, ok)

}

返回结果：

main读取数据： 100 true

panic: send on closed channel

main读取数据： 200 true

main读取数据： 0 false

main读取数据： 0 false

main读取数据： 0 false

goroutine 5 [running]:

main.main.func1(0xc42006e060)

/Users/steven/Documents/go\_project/src/ch10\_2/demo01\_test1.go:12 +0x79

created by main.main

/Users/steven/Documents/go\_project/src/ch10\_2/demo01\_test1.go:8 +0x70

## 缓冲通道和定向通道

### 缓冲通道

默认创建的通道都是非缓冲通道，读写都是即时阻塞；缓冲通道是自带一块缓冲区，可以暂时存储数据，如果缓冲区满了，那么才会阻塞。

例：

func main() {

//1.非缓冲通道

ch1 := make(chan int)

fmt.Println("非缓冲通道：", len(ch1), cap(ch1)) //0, 0

go func() {

data := <-ch1 //阻塞

fmt.Println("获取数据：", data)

}()

ch1 <- 100 //阻塞

//time.Sleep(1)

fmt.Println("写入数据ok\n-----------------------")

//2.缓冲通道,缓冲区满了才会阻塞

//ch2 := make(chan int, 5)

//fmt.Println("缓冲通道：", len(ch2), cap(ch2)) //0,5

//go func() {

// for data := range ch2 {

// //time.Sleep(1)

// fmt.Println("获取数据：", data)

// }

//}()

//

//ch2 <- 1

//fmt.Println(len(ch2), cap(ch2))

//ch2 <- 2

//fmt.Println(len(ch2), cap(ch2))

//ch2 <- 3

//fmt.Println(len(ch2), cap(ch2)) //3,5

//ch2 <- 4

//fmt.Println(len(ch2), cap(ch2))

//ch2 <- 5

//fmt.Println(len(ch2), cap(ch2)) //5, 5

//ch2 <- 6 //阻塞

//fmt.Println(len(ch2), cap(ch2))

//close(ch2)

//fmt.Println("main。。。over。。")

//3.缓冲通道

ch3 := make(chan string, 5)

fmt.Printf("%T\n", ch3)

go sendData(ch3)

for data := range ch3 {

//time.Sleep(1 \* time.Second)

fmt.Println("\t读取数据：", data)

}

fmt.Println("读取完毕。。")

}

func sendData(ch3 chan string) {

for i := 1; i <= 10; i++ {

ch3 <- fmt.Sprintf("data%d", i) //1,2,3,4,5,6,7

fmt.Println("写入数据：", i) //1,2,3,4,5

fmt.Println(len(ch3), cap(ch3))

}

close(ch3)

}

缓冲通道模拟生产者和消费者

func main() {

ch1 := make(chan int , 5)

ch2 := make(chan bool)//判断结束

rand.Seed(time.Now().UnixNano())

//写入数据：生产者

go func() {

for i := 1; i <= 20; i++ {

ch1 <- i

fmt.Println("写入数据：", i)

time.Sleep(time.Duration(rand.Intn(1000))\*time.Millisecond)

}

close(ch1)

}()

//读取数据：消费者

go func() {

for data := range ch1 {

fmt.Println("\t1号消费者：", data)

time.Sleep(time.Duration(rand.Intn(1000))\*time.Millisecond)

}

ch2 <- true

}()

go func() {

for data := range ch1 {//1

fmt.Println("\t2号消费者：", data)

time.Sleep(time.Duration(rand.Intn(1000))\*time.Millisecond)

}

ch2 <- true

}()

<- ch2

fmt.Println("main...over...")

}

### 定向通道

通道默认都是双向通道。即可写入数据，又可读取数据。定向通道：也叫单向通道，只读或只写。

只读：make(<- chan Type)，只能读取数据，不能写入数据

<- chan

只写：make(chan <- Type)，只能写入数据，不能读取数据

chan <- data

将通道作为参数传递的时候使用单向通道。定义函数，只有写入数据功能，或定义函数，只有读取数据功能。定向通道的意义：在语法级别，保证通道的操作安全

例：

func main() {

//1.双向通道

ch1 := make(chan string)

go fun1(ch1)

data := <-ch1

fmt.Println("main，接受到数据：", data)

ch1 <- "Go语言好学么？"

ch1 <- "区块链好学么?"

go fun2(ch1)

go fun3(ch1)

time.Sleep(1 \* time.Second)

fmt.Println("main over!")

}

func fun1(ch1 chan string) {

ch1 <- "我是Steven老师"

data := <-ch1

data2 := <-ch1

fmt.Println("回应：", data, data2)

}

//功能：只有写入数据

func fun2(ch1 chan<- string) {

//只能写入

ch1 <- "How are you?"

//<- ch1 //invalid operation: <-ch1 (receive from send-only type chan<- string)

}

//功能：只有读取数据

func fun3(ch1 <-chan string) {

data := <-ch1

fmt.Println("只读：", data)

//ch1 <- "hello" //invalid operation: ch1 <- "hello" (send to receive-only type <-chan string)

}

## time包中跟通道相关的函数

### Timer结构体

计时器类型表示单个事件。当计时器过期时，当前时间将被发送到C上(C是一个只读通道<-chan time.Time，该通道中放入的是Time结构体)，除非计时器是AfterFunc创建的。计时器必须使用NewTimer或AfterFunc创建。

Timer结构体的源码

// The Timer type represents a single event.

// When the Timer expires, the current time will be sent on C,

// unless the Timer was created by AfterFunc.

// A Timer must be created with NewTimer or AfterFunc.

type Timer struct {

C <-chan Time

r runtimeTimer

}

### NewTimer函数

NewTimer创建一个新的计时器，它会在至少持续时间d之后将当前时间发送到其通道上。

NewTimer()函数的源码

*// NewTimer creates a new Timer that will send*

*// the current time on its channel after at least duration d.*

**func** NewTimer(d Duration) \*Timer

例：

*//.创建计时器*

timer1 := time.NewTimer(5 \* time.***Second***)

fmt.Printf(**"%T\n"**, timer1) *//\*time.Timer*

fmt.Println(time.Now())

data := <-timer1.C *//<-chan time.Time*

fmt.Printf(**"%T\n"**,timer1.C) *//<-chan time.Time*

fmt.Printf(**"%T\n"**,data) *//time.Time*

fmt.Println(data)

### After()函数

After()函数相当于NewTimer(d). C。

After()函数的源码

*// After waits for the duration to elapse and then sends the current time*

*// on the returned channel.*

*// It is equivalent to NewTimer(d).C.*

*// The underlying Timer is not recovered by the garbage collector*

*// until the timer fires. If efficiency is a concern, use NewTimer*

*// instead and call Timer.Stop if the timer is no longer needed.*

**func** After(d Duration) <-**chan** Time {

**return** NewTimer(d).C

}

例：

*//使用After(),返回值<- chan Time,同Timer.C*

ch1 := time.After(5 \* time.***Second***)

fmt.Println(time.Now())

data := <-ch1

fmt.Printf(**"%T\n"**,data) *//time.Time*

fmt.Println(data)

## select分支语句

select 语句类似于 switch 语句，但是select会随机执行一个可运行的case。如果没有case可运行，它将阻塞，直到有case可运行。每个case都必须是一个通信，所有channel表达式都会被求值，所有被发送的表达式都会被求值。

如果任意某个通信可以进行，它就执行；其他被忽略。如果有多个case都可以运行，select会随机公平地选出一个执行，其他不会执行；如果有default子句，则执行该语句；如果没有default字句，select将阻塞，直到某个通道可以运行；Go不会重新对channel或值进行求值。

例1：

func main() {

ch1 := make(chan int)

ch2 := make(chan int)

go func() {

time.Sleep(1 \* time.Second)

ch1 <- 100

}()

go func() {

time.Sleep(1 \* time.Second)

ch2 <- 200

}()

select {

case data := <-ch1:

fmt.Println("ch1中读取数据了:", data)

case data := <-ch2:

fmt.Println("ch2中读取数据了：", data)

default:

fmt.Println("执行了default。。。")

}

}

运行结果：

执行了default。。。

例2：

func main() {

ch1 := make(chan int)

ch2 := make(chan int)

go func() {

time.Sleep(1 \* time.Second)

data := <-ch1

fmt.Println("ch1：", data)

}()

go func() {

time.Sleep(2 \* time.Second)

data := <-ch2

fmt.Println("ch2：", data)

}()

select {

case ch1 <- 100: //阻塞

close(ch1)

fmt.Println("ch1中写入数据。。")

case ch2 <- 200: //阻塞

close(ch2)

fmt.Println("ch2中写入数据。。")

case <-time.After(2 \* time.Second): //阻塞

fmt.Println("执行延时通道")

//default:

// fmt.Println("default..")

}

time.Sleep(4 \* time.Second)

fmt.Printf("main over ")

}

运行结果：

ch1： 100

ch1中写入数据。。

main over

## sync包

sync包提供了基本的同步基元，如互斥锁。除了Once和WaitGroup类型，大部分都是适用于低水平程序线程，高水平的同步使用channel通信更好一些。本包的类型的值不应被拷贝。

之前的案例中，我们都使用time.Sleep()函数，通过睡眠将主线程阻塞至所有线程结束。而更好的做法是使用WaitGroup来实现。

### WaitGroup同步等待组

同步：sync是串行执行

异步：async是同时执行

WaitGroup：同步等待组

type WaitGroup struct {

noCopy noCopy

state1 [12]byte

sema uint32

}

WaitGroup用于等待一组线程的结束。父线程调用Add方法来设定应等待的线程的数量。每个被等待的线程在结束时应调用Done方法。同时，主线程里可以调用Wait方法阻塞至所有线程结束。

WaitGroup中的方法

func (wg \*WaitGroup) Add(delta int)

Add方法向内部计数加上delta，delta可以是负数；如果内部计数器变为0，Wait方法阻塞等待的所有线程都会释放，如果计数器小于0，方法panic。注意Add加上正数的调用应在Wait之前，否则Wait可能只会等待很少的线程。一般来说本方法应在创建新的线程或者其他应等待的事件之前调用。

func (wg \*WaitGroup) Done()

Done方法减少WaitGroup计数器的值，应在线程的最后执行。

func (wg \*WaitGroup) Wait()

Wait方法阻塞直到WaitGroup计数器减为0。

例：

func main() {

var wg sync.WaitGroup

fmt.Printf("%T\n", wg) //sync.WaitGroup

fmt.Println(wg) //{{} [0 0 0 0 0 0 0 0 0 0 0 0] 0}

wg.Add(3)

rand.Seed(time.Now().UnixNano())

go printNum1(&wg)

go printNum2(&wg)

go printNum3(&wg)

wg.Wait() //main goroutine进入阻塞状态，当计数器为0后解除阻塞

fmt.Println("main解除阻塞，main over...")

}

func printNum1(wg \*sync.WaitGroup) {

for i := 1; i <= 10; i++ {

fmt.Println("子goroutine1,i：", i)

time.Sleep(time.Duration(rand.Intn(1000)))

}

wg.Done() //计数器减1

}

func printNum2(wg \*sync.WaitGroup) {

for i := 1; i <= 10; i++ {

fmt.Println("\t 子goroutine1,i：", i)

time.Sleep(time.Duration(rand.Intn(1000)))

}

wg.Done() //计数器减1

}

func printNum3(wg \*sync.WaitGroup) {

for i := 1; i <= 10; i++ {

fmt.Println("\t\t 子goroutine1,i：", i)

time.Sleep(time.Duration(rand.Intn(1000)))

}

wg.Done() //计数器减1

}

### 互斥锁

Mutex是一个互斥锁，可以创建为其他结构体的字段；零值为解锁状态。Mutex类型的锁和线程无关，可以由不同的线程加锁和解锁。

type Mutex struct {

state int32

sema uint32

}

func (m \*Mutex) Lock()

Lock方法锁住m，如果m已经加锁，则阻塞直到m解锁。

func (m \*Mutex) Unlock()

Unlock方法解锁m，如果m未加锁会导致运行时错误。锁和线程无关，可以由不同的线程加锁和解锁。

互斥锁实现售票示例代码

var tickts = 100 //全局变量，仅一份

var wg sync.WaitGroup

var mutex sync.Mutex //互斥锁

func main() {

/\*

练习题：模拟火车站卖票

火车票100张，4个售票口出售(4个goroutine)。

\*/

var wg sync.WaitGroup

wg.Add(4)

go saleTickets("售票口1", &wg) //g1

go saleTickets("售票口2", &wg) //g2

go saleTickets("售票口3", &wg) //g3

go saleTickets("售票口4", &wg) //g4

wg.Wait()

fmt.Println("所有车票已售空。程序结束！")

}

func saleTickets(name string, wg \*sync.WaitGroup) {

for {

mutex.Lock()

if tickts > 0 {

time.Sleep(1 \* time.Second)

fmt.Println(name, "：", tickts) //1

tickts--

} else {

fmt.Println(name, ",结束卖票。。")

mutex.Unlock()

break

}

mutex.Unlock() //解锁

}

wg.Done()

}

### 读写互斥锁RWMutex

RWMutex是读写互斥锁。该锁可以被同时多个读取者持有或唯一个写入者持有。RWMutex可以创建为其他结构体的字段；零值为解锁状态。RWMutex类型的锁也和线程无关，可以由不同的线程加读取锁/写入和解读取锁/写入锁。

type RWMutex struct {

w Mutex // held if there are pending writers

writerSem uint32 // semaphore for writers to wait for completing readers

readerSem uint32 // semaphore for readers to wait for completing writers

readerCount int32 // number of pending readers

readerWait int32 // number of departing readers

}

锁定的规则：

读写锁的使用中：写操作都是互斥的、读和写是互斥的、读和读不互斥。

理解为：

可以多个goroutine同时读取数据，但是写只允许一个goroutine写数据。

Mutex中的方法

func (rw \*[RWMutex](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "RWMutex)) Lock()

Lock方法将rw锁定为写入状态，禁止其他线程读取或者写入。

func (rw \*[RWMutex](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "RWMutex)) Unlock()

Unlock方法解除rw的写入锁状态，如果m未加写入锁会导致运行时错误。

func (rw \*[RWMutex](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "RWMutex)) RLock()

RLock方法将rw锁定为读取状态，禁止其他线程写入，但不禁止读取。

func (rw \*[RWMutex](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "RWMutex)) RUnlock()

Runlock方法解除rw的读取锁状态，如果m未加读取锁会导致运行时错误。

func (rw \*[RWMutex](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "RWMutex)) RLocker() [Locker](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "Locker)

Rlocker方法返回一个互斥锁，通过调用rw.Rlock和rw.Runlock实现了Locker接口。

例

func main() {

var rwm sync.RWMutex

for i := 1; i <= 3; i++ {

go func(i int) {

fmt.Printf("goroutine %d，尝试读锁定。。\n", i)

rwm.RLock()

fmt.Printf("goroutine %d，已经读锁定了。。\n", i)

time.Sleep(5 \* time.Second)

fmt.Printf("goroutine %d,读解锁。。\n", i)

rwm.RUnlock()

}(i)

}

time.Sleep(1\*time.Second)

fmt.Println("main..尝试写锁定。。")

rwm.Lock()

fmt.Println("main。。已经写锁定了。。")

rwm.Unlock()

fmt.Printf("main。。写解锁。。。")

}

### 条件变量Cond

Cond实现了一个条件变量，一个线程集合地，供线程等待或者宣布某事件的发生。

每个Cond实例都有一个相关的锁（一般是\*Mutex或\*RWMutex类型的值），它必须在改变条件时或者调用Wait方法时保持锁定。Cond可以创建为其他结构体的字段，Cond在开始使用后不能被拷贝。条件变量：sync.Cond,多个goroutine等待或接受通知的集合地

type Cond struct {

noCopy noCopy

// L is held while observing or changing the condition

L Locker

notify notifyList

checker copyChecker

}

Cond中的方法

func NewCond(l [Locker](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "Locker)) \*[Cond](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "Cond)

使用锁l创建一个\*Cond。Cond条件变量，总是要和锁结合使用。

func (c \*[Cond](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "Cond)) Broadcast()

Broadcast唤醒所有等待c的线程。调用者在调用本方法时，建议（但并非必须）保持c.L的锁定。

func (c \*[Cond](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "Cond)) Signal()

Signal唤醒等待c的一个线程（如果存在）。调用者在调用本方法时，建议（但并非必须）保持c.L的锁定。发送通知给一个人。

func (c \*[Cond](https://www.studygolang.com/static/pkgdoc/pkg/sync.htm" \l "Cond)) Wait()

Wait自行解锁c.L并阻塞当前线程，在之后线程恢复执行时，Wait方法会在返回前锁定c.L。和其他系统不同，Wait除非被Broadcast或者Signal唤醒，不会主动返回。广播给所有人。

因为线程中Wait方法是第一个恢复执行的，而此时c.L未加锁。调用者不应假设Wait恢复时条件已满足，相反，调用者应在循环中等待：

例：

func main() {

var mutex sync.Mutex

cond := sync.Cond{L:&mutex}

condition := false

go func() {

time.Sleep(1\*time.Second)

cond.L.Lock()

fmt.Println("子goroutine已经锁定。。。")

fmt.Println("子goroutine更改条件数值，并发送通知。。")

condition = true//更改数值

cond.Signal() //发送通知：一个goroutine

fmt.Println("子gorutine。。。继续。。。")

time.Sleep(5\*time.Second)

fmt.Println("子groutine解锁。。")

cond.L.Unlock()

}()

cond.L.Lock()

fmt.Println("main..已经锁定。。。")

if !condition{

fmt.Println("main.。即将等待。。。")

//wait()

// 1.wait尝试解锁，

// 2.等待--->当前的groutine进入了阻塞状态，等待被唤醒：signal(),broadcast()

// 3.一旦被唤醒后，又会锁定

cond.Wait()

fmt.Println("main.被唤醒。。")

}

fmt.Println("main。。。继续")

fmt.Println("main..解锁。。。")

cond.L.Unlock()

}

[第12章 Go语言并发编程 1](#_Toc857487387)

[12.1 并发与并行 1](#_Toc27758292)

[12.1.1 Go是并发语言，而不是并行语言 1](#_Toc529662245)

[12.1.2 什么是并发？ 2](#_Toc713634900)

[12.2 进程和线程 3](#_Toc365595805)

[12.2.1 基本概念 3](#_Toc617980568)

[12.2.2 进程与线程的区别 4](#_Toc1168489484)

[12.2.3 相关背景知识 4](#_Toc64805773)

[12.3 轻量级线程——协程coroutine 5](#_Toc416417782)

[12.3.1 什么是协程？ 5](#_Toc84456501)

[12.3.2 协程的应用 6](#_Toc2121205287)

[12.4 Go语言中协程——goroutine 6](#_Toc721234762)

[12.4.1 什么是Goroutine？ 6](#_Toc1394941266)

[12.4.2 Goroutine在线程上的优势 7](#_Toc698883363)

[12.4.3 使用普通函数创建goroutine 7](#_Toc1544616498)

[12.4.4 使用匿名函数创建goroutine 9](#_Toc1587156950)

[12.4.5 启动多个Goroutines 9](#_Toc1452479263)

[12.4.6 调整并发的运行性能Gomaxprocs 10](#_Toc1372357792)

[12.5 多个goroutine间通信的通道channel 10](#_Toc1243041364)

[12.5.1 通道的概述 10](#_Toc1075286732)

[12.5.2 声明通道类型 10](#_Toc1269215219)

[12.5.3 通道发送数据 11](#_Toc745120082)

[12.5.4 阻塞 11](#_Toc1256072517)

[12.5.5 通道接收数据 12](#_Toc1046543209)

[12.5.6 关闭通道 14](#_Toc1360644733)

[12.6 缓冲通道和定向通道 15](#_Toc1950154275)

[12.6.1 缓冲通道 15](#_Toc1347479411)

[12.6.2 定向通道 17](#_Toc1871403062)

[12.7 time包中跟通道相关的函数 18](#_Toc625769072)

[12.7.1 Timer结构体 18](#_Toc1073373745)

[12.7.2 NewTimer函数 19](#_Toc1329897415)

[12.7.3 After()函数 19](#_Toc576055929)

[12.8 select分支语句 20](#_Toc915718027)

[12.9 sync包 21](#_Toc1605065387)

[12.9.1 WaitGroup同步等待组 21](#_Toc1791869342)

[12.9.2 互斥锁 23](#_Toc1784849113)

[12.9.3 读写互斥锁RWMutex 24](#_Toc1907460895)

[12.9.4 条件变量Cond 25](#_Toc1059379849)