goroutine

协程

- 进程 — 进程是操作系统对运行时程序的抽象。系统运行时会同时存在多个进程，CPU按照进程的优先级来执行调度
- 线程
  - 线程是建立在进程模型上的另一种抽象，在Linux中，并不区分进程与线程，通称为任务(Task)。但是，一个进程中可有多个线程，它们共享同一进程内存空间
  - 线程的上下文切换要比进程的上下文切换效率更高，原因在于同一进程的线程间共享诸多变量
- 协程
  - 上面所述的线程通常是内核中实现的线程，而协程，有时候又称之为用户空间实现的线程
  - 协程，本意为协作的例程，而例程，通常是一个函数。协程之间通过yield来转移CPU的执行权利，并保存当前执行函数的上下文信息(如程序计数器、栈帧) — 让出CPU执行权利后，由用户空间的调度器来选择另一个例程来执行
  - 在用户空间实现的线程有诸多好处，首先是省去了用户态与内核态的切换，其次就是线程切换与调度都是用户空间控制，更为地迅捷

goroutine

Go语言中，一个协作的子例程称之为goroutine，创建一个goroutine非常简单，只需要在普通的函数或者方法调用之前添加go关键字前缀

```
foo()
go bar()  // 不会阻塞
hugo()
```

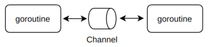
- 可增长栈实现
  - 在OS线程实现中，每个线程所用栈大小是固定的。Linux系统中为8192KB，即8M — 所以，系统内能够创建的线程是有限的
  - 但是，Go语言则采用动态栈来保存函数关键信息和执行goroutine内的函数调用 — 初始栈大小典型情况下为2KB，按需增长或收缩，最大栈大小通常为1GB
  - 正是因为动态栈的实现，所以使得goroutine的数量限制可以远超线程的数量限制，创建10万个goroutine并不是一个不可能的事情
- 调度
  - OS线程调度
    - 内核线程由内核进行调度，当线程的时间片到期或者是发起了对"慢速设备"(socket、管道等)的读取时，内核调度将停止该线程的执行
    - 将寄存器信息以及线程运行所需的所有数据保存至内存，转而调度并执行下一个线程，该过程称之为一次完整的上下文切换

    由内核完成
  - goroutine调度
    - Go在运行时包含自己的调度器，不由硬件时钟定期触发，而是使用特定的数据结构进行触发
    - 当一个goroutine执行time.Sleep()或者是被其它调用阻塞时，Go调度器将会将goroutine设置为休眠模式，并转而调度其它的goroutine

    由Go调度器完成

通道(channel)

goroutine是Go语言程序并发的执行体，而通道就是它们之间的连接 — 直白来说，通道就是goroutine之间的通信方法。为方便理解，可以看做是IPC的一种



Go语言在设计goroutine时，有一个非常重要的理念: 不要使用共享内存通信，而是应该使用通信的方式来共享内存

- 基本使用
  - 通道承载的类型
    - 每一个通道只能发送和接收同一类型的元素，称之为通道的元素类型
    - 例如一个类型元素为int的通道，通常写作为 chan int
  - 通道的创建
    - 通道通常由内置的make函数进行创建 — `ch := make(chan int)`
    - 由make所创建的通道返回结果为引用类型，当作为参数传递至函数时，函数内部复制的是该通道的引用
  - 数据的发送与接收
    - 发送数据 — 向通道中发送某一特定的元素类型的值 — `ch <- x`
    - 接收数据 — 接收通道中某一特定元素类型的值 — `x = <- ch`

    两个操作均使用 <- 操作符
- 无缓冲通道
  - 无缓冲通道最多只能容纳一个元素，当通道内未有元素进行接收操作，或者是通道内已有元素进行发送操作时，均会被阻塞 — **无缓冲通道又称之为阻塞通道，或者同步通道**

```
func main() {
    done := make(chan int)
    go func() {
        time.Sleep(2 * time.Second)
        done <- 0
    }()
    /* main函数的goroutine进行其它操作 */
    value := <- done  // 等待后台goroutine完成
    /* 执行后续的操作 */
    fmt.Printf("Receive value: %d\n", value)
}
```

  - 无缓冲通道多用于协程同步，当两个goroutine执行必须存在顺序时，就会使用无缓冲通道进行同步 — **当通道仅用于事件通知时，通常会使用 struct{} 空结构体来作为通道的元素类型，而不是其它类型，算是个约定**
- 缓冲通道
  - 缓冲通道就相当于一个队列，实际上缓冲通道的内部实现也是队列 — 缓冲通道允许在创建时指定其最大容量，当缓冲区已满时发送操作将阻塞，缓冲区无元素时接收操作将阻塞
  - 基于此特性，完全可以将缓冲通道当做是生产者-消费者之间的队列使用

```
func main() {
    done := make(chan int, 20)

    var numbers [10][0]int
    for range numbers {  // 10个消费者
        go func() {
            v := <- done
            fmt.Println(v)
        }()
    }

    for i := 0; i < 10; i++ {  // 一个生产者
        done <- i
    }

    // 确保主函数的goroutine不会退出
    time.Sleep(10 * time.Second)
}
```

使用select进行多路复用

当我们需要同时对多个通道进行数据接收时，为了避免无意义的阻塞，通常会使用select来进行多路复用

- 与Linux select()调用的相似与区别
  - 相似
    - 不管是Linux select调用，还是Go select，都是对多个感兴趣时间进行同时监控的工具
      - Linux — 对多个"慢速设备"可非阻塞读、非阻塞写事件的监控
      - Go — 对多个通道的可接收、可发送事件的监控
    - 在未有事件准备就绪时，select将会阻塞
  - 区别
    - Linux select()调用成功时，将返回至少一个可读或可写的文件描述符数量，但是Go select只关心最快发生的事件，并且只会执行该事件对应的语句
    - Go select同时支持非阻塞的多路复用(使用default)，Linux select()调用仅支持阻塞式调用(poll()调用支持非阻塞调用)
    - Linux select()调用支持超时事件，在一定事件内未有事件准备就绪则直接返回。Go select需自行实现

```
func main() {
    abort := make(chan struct{})

    go func() {
        os.Stdin.Read(make([]byte, 1))
        abort <- struct{}{}
    }()

    select {
    /* select将会一值阻塞，直到有一次channel的通信发生 */
    case <- time.After(5 * time.Second):
        fmt.Println("Rocket launched, xiu~")
    case <- abort:
        fmt.Println("Rocket launch aborted!")
    }
}
```

首先启动一个goroutine从标准输入中读取一个字节，成功后将其发送至无缓冲的abort通道

主函数进入select多路复用，要么5秒后定时器到期执行火箭发射，要么在此之前用户键入了字符，取消火箭发射

- 非阻塞的多路复用 — select允许有一种默认情况，用来指定在没有其它的通信发生时可立即执行的动作。若该动作发生，select语句块随之退出

```
select {
case <- abort:
    fmt.Println("Rocket launch aborted!")
default:
    fmt.Println("Nothing happened")
}
```

当abort通道没有数据可接收时，将会执行default下的语句，随机退出select语句块 — 多用于循环中

带有default的select多路复用只是看一眼是否有通信发生，没有就算了，也不会杵在那儿干等