Go语言特色编程

一、课前准备

> 说明：提前需要让学生做的课前准备，比如环境安装部署、工具安装、插件安装等，需要学生提前做的都放这，并且给出下载链接或信息源；

1. Golang开发环境环境安装就绪；
2. Golang-IDE开发环境安装就绪；

3. 练习Go语言基础语法代码；
4. 预习Go语言并发与同步知识；

二、课堂主题

> 说明：本堂课的总体概述，明确课堂主题和主体；

本节主要介绍Go语言当中的特色编程，主要包含Go语言的并发特性，并发调度算法了解，同步机制，sync包的使用，Go语言高效测试，以及包的管理等。

三、课堂目标

> 说明：主要是让学生了解，学了本堂课后，能达到的一个期望值，要量化；

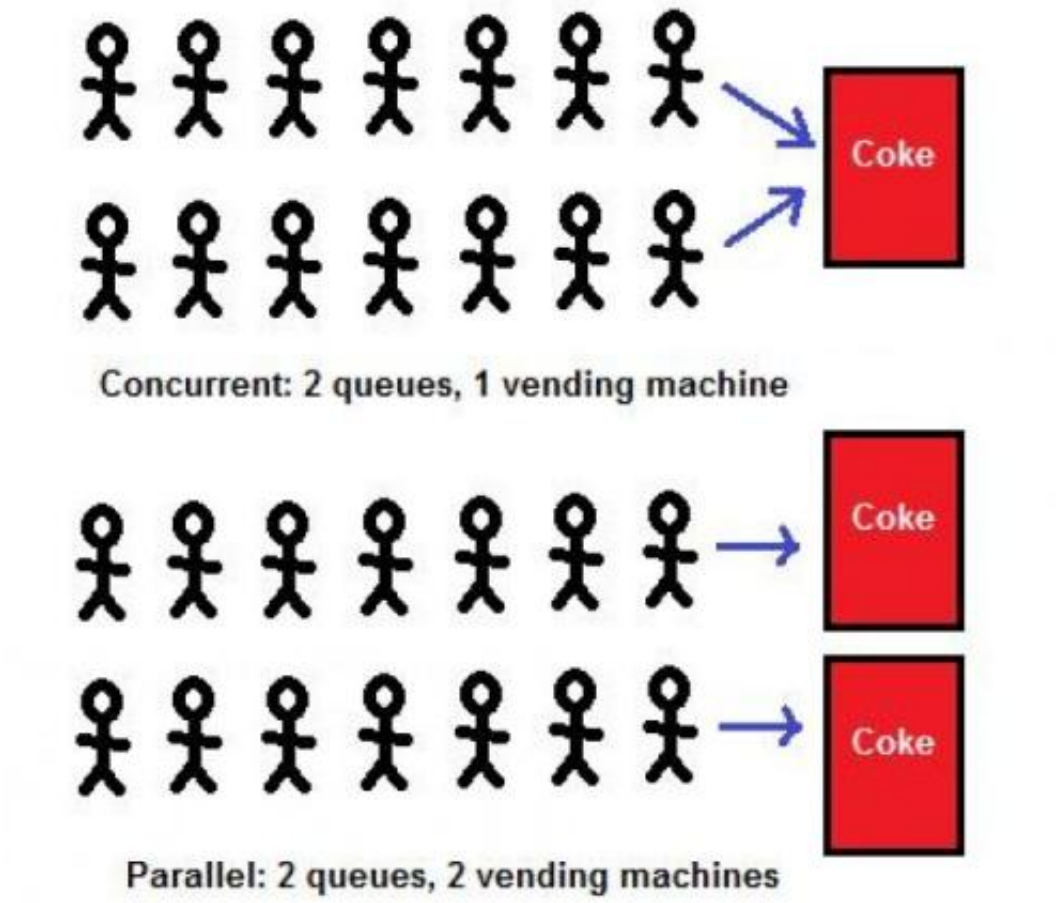
1. 掌握Go语言并发编程；
2. 掌握Go语言多种同步方式；
3. 掌握Go语言生产者与消费者模型；
4. 掌握Go语言测试代码的编写原则；
5. 掌握Go语言包的创建与管理；

四、知识点

Go语言作为很新的一门语言，我们本堂主要介绍的都是Go语言的特性，并发，channel，测试，包管理这些都可以代表Go语言的优秀品质。

1. 并发编程

1.1 并发与并行



并行与并发是两个不同的概念，普通解释：

- 并发：交替做不同事情的能力
- 并行：同时做不同事情的能力

如果站在程序员的角度去解释是这样的：

- 并发：不同的代码块交替执行
- 并行：不同的代码块同时执行

并发和并行都是为了提高机器硬件利用率，提升应用运行效率。并行和并发就是为达目的的两个手段。并行可以提升CPU核心数，并发则是通过系统设计，比如分时复用系统，多进程，多线程的开发方法，不过在对并发的支持上，Go语言是天生最好的，因为它设计的理念就是并发，而且是在语言层面实现的并发。

1.2 Goroutine

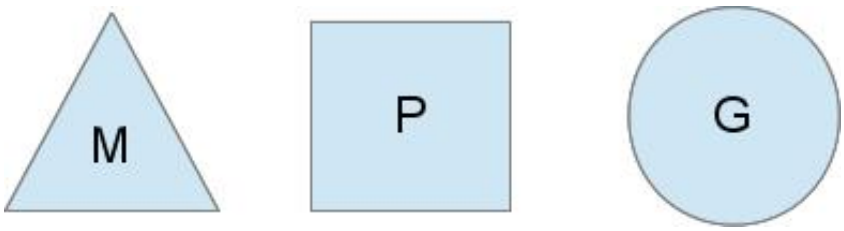
如果对线程和进程有所了解的话，我们可以这样给进程和线程下一个专业点的定义。

- 线程是最小的执行单位
- 进程是最小的资源申请单位

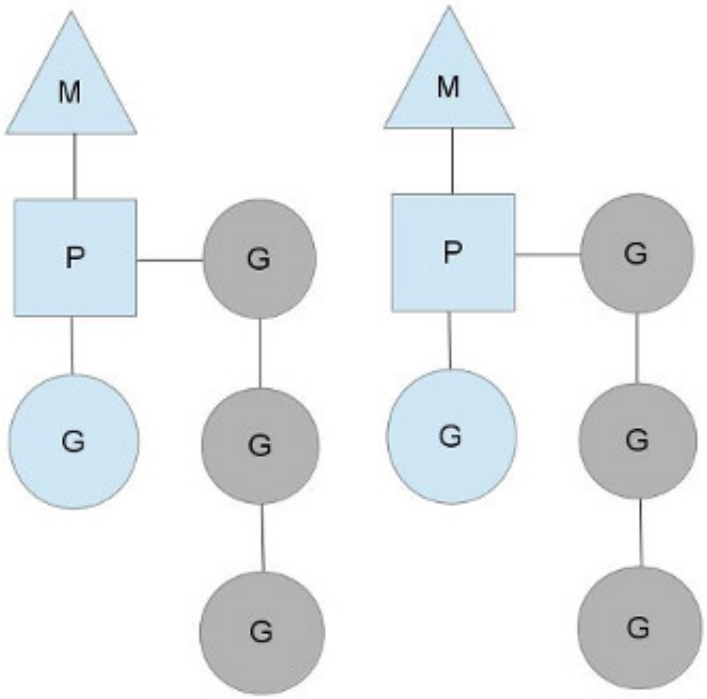
在Go语言当中，有一个存在是比线程还要小的执行单位，那就是Goroutine，翻译上习惯叫例程或协程，不过我们遵循原汁原味还是直接用Goroutine来称呼它。Go语言开发者为了实现并支持Goroutine，特意花了大力气来开发一个语言层面的调度算法。

具体调度算法详情介绍可以查看英文原文：调度算法链接

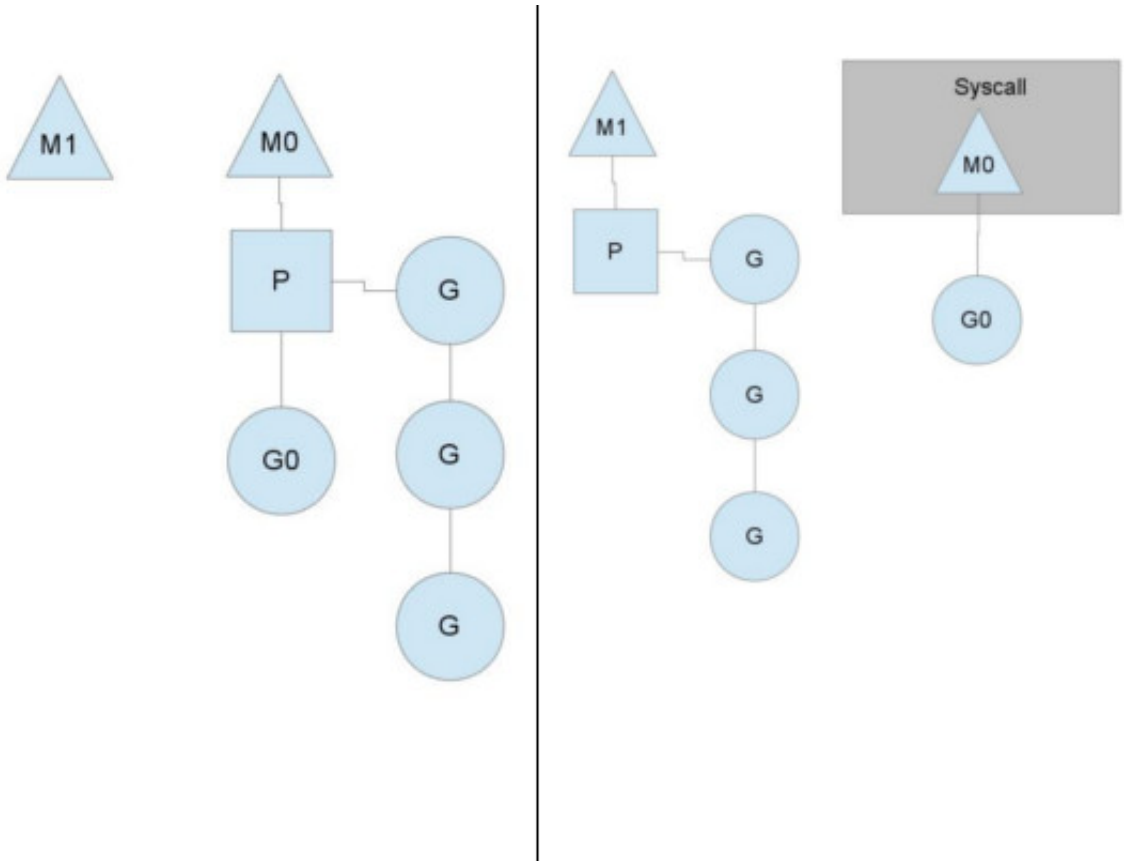
简单理解的话，首先要明确Go语言调度算法中提到的三个字母，M，P，G，分别代表线程，上下文以及Goroutine。



在操作系统层面线程仍然是最小的执行单位，在进程调度的时候，CPU需要在不同进程间切换，此时需要保留运行的上下文信息，也就是前面提到的P，至于G则是代表了Go语言当中的Goroutine。



每个线程M有一个自己的上下文P，同一时刻，每个线程内部可以执行一个Groutine代码，如上图所示，每个线程有一个自己的调度队列，这个队列里存放的就是若干个Goroutine。



当线程发生系统调用时，该线程会被阻塞，此时为了提高运行效率，Goroutine调度算法会将该阻塞的线程Goroutine队列转移到其他线程上。当线程执行完系统调用后，又会从其他线程的队列中“借”一些Goroutine过来。

1.3 Goroutine启动

在一台主机上，线程启动的数量是有上限的，这个上限并非可以启动的上限，而是不影响系统性能的上限。Goroutine在并发上则没有这方面的顾虑，可以随心所欲的启动，不用担心数量的问题，当然这归功于Go语言的调度算法。

那么Goroutine如何启动呢？其实在我们之前写的代码中，都存在一个Goroutine，就是我们的主函数，我们习惯叫他main-goroutine。如果我们想再启动一个Goroutine，也非常容易，直接关键字go再加上函数调用就行了。

```
go call_func()
```

为了见识一下这个Goroutine，我们来简单尝试一下。

```
package main

import (
  "fmt"
)
```

```
func main() {
  fmt.Println("begin call goroutine")
  //启动goroutine
  go func() {
    fmt.Println("I am a goroutine!")
  }()
  fmt.Println("end call goroutine")
}
```

执行这个代码，我们大概率是看不到“I am a goroutine!”这句话的，因为go func()这里创建的Goroutine或者尚未创建成功时，main-goroutine已经结束执行了，main-goroutine结束执行，也就代表着进程退出，那么不会有任何代码被执行了！那么大家考虑一下，如何解决这个问题呢?

1.4 Goroutine执行案例

接下来我们来做一个小练习，求斐波那契数列，斐波那契数列的特点是随着要求的数逐渐增大，递归调用的次数逐渐增多，耗时也会增多。为了增加用户体验，我们时刻打印一个小图标让用户知道我们在帮他计算，显然打印的动作与数学计算的动作是要并发执行的。

```
package main

import (
  "fmt"
  "time"
)

func main() {
  go spinner(time.Millisecond * 100) //启动一个打印goroutine
  fmt.Printf("\n%d\n", fib(45))
}

//计算斐波那契数列
func fib(x int) int {
  if x < 2 {
    return x
  }
  return fib(x-2) + fib(x-1)
}

//此函数目的只是为了用户体验
func spinner(delay time.Duration) {
  for {
    for _, r := range `-\|/` {
      fmt.Printf("\r%c", r)
      time.Sleep(delay)
    }
  }
}
```

执行上述代码，感受一下吧，注意在本代码中增加的Goroutine并非是为了提高斐波那契数列的计算速度。

1.5 Go语言运行时

所谓运行时，就是运行的时刻，Go语言的运行时就是描述与进程运行相关的信息，我们可以使用runtime包来显示一些运行时的信息。

1.5.1 GOMAXPROCS

```
func GOMAXPROCS(n int) int
//当n<=1时，查看当前进程可以并行的goroutine最大数量（CPU核心数）
//当n>1时，代表设置可并行的最大goroutine数量
```

这个函数可以帮我们查看或设置当前进程的最大CPU核心数。

```
package main

import (
  "fmt"
  "runtime"
)

func main() {
  fmt.Println("Go Max proc =", runtime.GOMAXPROCS(0))
  runtime.GOMAXPROCS(2)
  fmt.Println("Go Max proc =", runtime.GOMAXPROCS(0))
}
```

执行结果如下:

```
localhost: teacher$ go run 03-runtime.go
Go Max proc = 4
Go Max proc = 2
```

其余与运行时相关的函数有:

- Goexit 退出调用的goroutine，退出前仍然会触发defer设定的延迟退出函数
- Gosched 让出本次goroutine的调度，很谦让的一种方式

2. 同步

同步在不同的语境代表不同的含义，在数据库中是指数据的同步，在分布式系统中是指系统内的数据一致，而在语言层面的同步是指运行时步调一致，避免竞争，有先有后。

2.1 如何做到同步

为了提高CPU的使用效率，我们需要启动多个Goroutine，而多个Goroutine比线程的颗粒度还小，他们之间必然存在争抢同一资源的现象，就像我们在线程中要控制同步一样，多个Goroutine在访问同一共享资源时，我们仍然要控制同步。

如何做到最直接的同步，可以借鉴我们生活中的例子，在火车上我们去卫生间的时候，都会把门锁上，这样别人就没法进来了。在这个例子中，我们和其他人就是Goroutine，而卫生间就是那个共享资源，我们不允许发生大家一起进入使用的情况，而解决这个问题的关键就是锁！

那么Go语言给我们提供了哪些同步机制呢？主要有如下方式：

- WaitGroup 计数等待法
- Once 执行一次
- Mutex 互斥锁
- RWMutex 读写锁
- Cond 条件变量
- channel 通道，Go语言的最大特性

在上述方法中，除了channel，其余的同步方式都在Go语言的sync包当中。

2.2 WaitGroup

Go语言的同步方式很多，WaitGroup的实现方式很巧妙，主要只有3个API。

```
//增加计数
func (wg *WaitGroup) Add(delta int)
//减少计数
func (wg *WaitGroup) Done()
//阻塞等待计数变为0
func (wg *WaitGroup) Wait()
```

它的核心思想是当启动一个Goroutine时，使用Add添加一个计数器，而Wait的功能是阻塞等待计数器归0，Done的作用则是Goroutine运行结束后执行此句话清掉计数器的一个计数。

利用WaitGroup的特性，我们可以优雅的实现一个例子：启动10个Goroutine，让他们顺序退出，main-goroutine等待所有Goroutine退出后才可退出。

```
package main

import (
  "fmt"
  "sync"
  "time"
)

var w sync.WaitGroup

func main() {
  for i := 0; i < 10; i++ {
    w.Add(1) //添加一个要监控的Goroutine数量
    go func(num int) {
```

```
        time.Sleep(time.Second * time.Duration(num))
        fmt.Printf("I am %d Goroutine\n", num)
        w.Done() //释放一个
    }(i)
  }

  w.Wait() //阻塞等待
}
```

2.3 Mutex互斥锁

提到互斥锁，我们通常会提到一个临界区的概念，Goroutine在准备访问共享数据时，我们就认为它进入了临界区。

使用互斥锁的核心思想就是进入临界区之前先要申请锁，申请到锁的Goroutine继续执行，而没有申请到的Goroutine则阻塞等待别人释放这个mutex，这样就可以有效的控制Goroutine之间竞争的问题。下述代码就是一个存在数据修改竞争的例子，循环1000次对一个数据自增，目标的输出结果是1000，但执行下面的代码很难获得1000。

```
package main
import (
    "fmt"
    "sync"
    )
var x  = 0
func increment(wg *sync.WaitGroup) {
    x = x + 1
    wg.Done()
}
func main() {
    var w sync.WaitGroup
    for i := 0; i < 1000; i++ {
        w.Add(1)
        go increment(&w)
    }
    w.Wait()
    fmt.Println("final value of x", x)
}
```

上述代码如果在进入临界区前使用mutex，就可以很好的解决该问题。代码主要修改increment函数即可：

```
func increment(wg *sync.WaitGroup) {
  mutex.Lock()   //上锁
  x = x + 1      //临界区
  mutex.Unlock() //释放锁
  wg.Done()
}
```

修改后再执行代码，就可以很好的看到效果。

```
root:teacher teacher$ go run 03-mutex.go
final value of x 1000
```

2.4 RWMutex

RWMutex我们可以称其为读写锁，它算是mutex的改进版，因为mutex的特点是排他性，只要有一个上锁成功了，其余人都不可以使用。但在实际开发过程中，经常会出现多个Goroutine去访问相同的共享资源，只不过这些Goroutine中有些是读数据，有些是写数据。开发者肯定明白，读数据不会对数据造成影响，这样理论上来说，一个读的Goroutine上锁了，其余的读Goroutine理应也可以访问，这样就出现了读写锁。对于读写锁来说，关键是掌握它的原则：

- 读共享
- 写独占
- 写优先级高

```
package main

import (
  "fmt"
  "sync"
  "time"
)

var rwlock sync.RWMutex
var wg sync.WaitGroup

var x = 0

func go_reader(num int) {
  for {
    rwlock.RLock()
    fmt.Printf("I am %d reader goroutine x = %d\n", num, x)
    time.Sleep(time.Millisecond * 2)
    rwlock.RUnlock()
  }

  wg.Done()
}

func go_writer(num int) {
  for {
    rwlock.Lock()
    x += 1
```

```
    fmt.Printf("I am %d writer goroutine x = %d\n", num, x)
    time.Sleep(time.Millisecond * 2)
    rwlock.Unlock()
  }

  wg.Done()
}

func main() {
  for i := 0; i < 7; i++ {
    wg.Add(1)
    go go_reader(i)
  }
  for i := 0; i < 3; i++ {
    wg.Add(1)
    go go_writer(i)
  }
  wg.Wait()
}
```

2.5 Once

在很多时候，我们会有一个需求，那就是虽然多个Goroutine都要运行一段代码，但我们却希望这段代码只能被一个Goroutine运行，也就是说只被允许运行一次。在Go语言当中，就给我们提供了这样的机制 -- Once。

```
type Once struct {
  done uint32
  m    Mutex
}
```

Once的实现机制其实就是首先执行要抢锁，抢到锁之后再判断这件事是否已经被做过了，如果做过了，就不需要再做了！Once内部也只有一个对外的函数Do，它的参数是一个函数。

```
func (o *Once) Do(f func())
```

来看一段代码：

```
package main

import (
  "fmt"
  "sync"
)
```

```
func main() {
  var count int

  var once sync.Once

  var wg sync.WaitGroup

  for i := 0; i < 100; i++ {
    wg.Add(1)
    go func() {
      defer wg.Done()
      once.Do(func() {
        count += 1 //确保只执行一次
      })
    }()
  }

  wg.Wait()

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

最后的执行结果是count只会被累加一次。

2.6 Cond条件变量

条件变量也是在提到同步时不得不说的话题，而且条件变量用起来远远比互斥锁要复杂的多。那么条件变量究竟要解决什么问题?

使用锁的方式可以确保吃货们协调好吃饼的顺序，但是却会出现一些无谓的竞争，因为当筐里无饼的时候，他们再竞争也无意义，这个时候他们应带等待一件事情的发生后才应该再去抢饼，这个要等待的事情就是一个条件。他们都在等待一个条件的满足，然后才继续竞争，这个案例就是大名鼎鼎的生产者和消费者模型。而条件变量本身不是锁，它与锁相互配合实现了避免竞争的一些机制。

至于条件的使用方式，需要了解它的机制和API。我们先来说结构与API：

```
type Cond struct {
  noCopy noCopy

  // L is held while observing or changing the condition
  L Locker

  notify  notifyList
  checker copyChecker
}
type Locker interface {
  Lock()
  Unlock()
}
```

Go语言当中的条件变量要借助mutex，因为它使用了Locker接口，mutex显然支持了Locker接口。

```
func NewCond(l Locker) *Cond
func (c *Cond) Wait()
func (c *Cond) Signal()
func (c *Cond) Broadcast()
```

- Wait函数的作用很复杂，首先在本函数调用前，该Groutine一定要获得了该Cond的锁，在调用本函数时，该锁会被立即释放，同时阻塞等待被人唤醒（条件产生）；当被人唤醒时，再去抢锁，注意由于Wait函数在阻塞时已经释放了锁，所以多个Goroutine都可能被唤醒且同时抢锁；谁抢到锁，谁继续往下执行，没抢到的继续阻塞。
- Signal 唤醒服务，唤醒等待该条件的一个Goroutine
- Broadcast 唤醒服务，唤醒等待该条件的所有Goroutine

来参考一下示例代码，该代码使用了两个Cond，一个用来控制容器的数量，防止生产者生产过剩，另外一个用来控制消费者避免无必要的竞争。

```
package main

import (
  "fmt"
  "sync"
  "time"
)

var cond *sync.Cond
var mutex sync.Mutex

var cond_five *sync.Cond
var mutex_five sync.Mutex
var wg sync.WaitGroup

var five [5]int
var five_count int
```

```
var customer_index int = 0

func productor() {
  index := 0
  prodnum := 1000
  for {
    mutex.Lock()
    time.Sleep(time.Millisecond * 100)
    five[index] = prodnum
    fmt.Println("Productor num=======", prodnum)
    prodnum++
    five_count++
    index = (index + 1) % 5
    cond.Signal()
    mutex.Unlock()
    if five_count == 5 { //限制生产者不能随意生产
      mutex_five.Lock()
      cond_five.Wait()
      mutex_five.Unlock()
    }
    //time.Sleep(time.Millisecond)

  }

  wg.Done()
}

func customer(num int) {

  for {
    mutex.Lock()
    cond.Wait()
    if five_count > 0 {
      time.Sleep(time.Millisecond * 10)
      fmt.Println(num, "customer num=======", five[customer_index])

      customer_index = (customer_index + 1) % 5
      five_count--
      cond_five.Signal()
    }
    mutex.Unlock()
  }
  wg.Done()
}

func main() {
  cond = sync.NewCond(&mutex)
  cond_five = sync.NewCond(&mutex_five)
  wg.Add(2)
```

```
  go productor()
  go customer(1)
  go customer(2)
  wg.Wait()
}
```

2.7 channel同步

2.7.1 channel通信与CSP并发模型

在linux系统编程中，有一种进程间通信的方式叫管道，两个进程可以借助内核开辟的缓冲区进行数据交换，形象上就像是一个水管（内核的缓冲区）把数据从一个进程流向另外一个进程。在Go语言当中，也设计了一款类似的通信方式 -- channel，利用channel读写的特性，不光可以实现Goroutine之间精准通信，也可以控制Goroutine之间的同步协调。

这个并发模型就是著名的CSP（Communicating Sequential Process），这个模型最早是上世纪70年代提出的。

在Go语言之中，我们借助内置make函数创建channel，channel的创建可以有缓冲区，也可以无缓冲区。

```
make(chan chantype)
make(chan chantype, 5)
```

对于通道，我们关键是掌握他们的读写行为。

- 写行为

通道缓冲区已满（无缓冲区），写阻塞直到缓冲区有空间（或读端有读行为） 通道缓冲区未满，顺利写入，结束

- 读行为

缓冲区无数据（无缓冲区时写端未写数据），读阻塞直到写端有数据写入 缓冲区有数据，顺利读数据，结束

```
package main

import (
  "fmt"
  "time"
)

var c chan string

func reader() {
  msg := <-c //读通道
  fmt.Println("I am reader,", msg)
```

```
}

func main() {
  c = make(chan string)
  go reader()
  fmt.Println("begin sleep")
  time.Sleep(time.Second * 3) //睡眠3s为了看执行效果
  c <- "hello" //写通道
  time.Sleep(time.Second * 1) //睡眠1s为了看执行效果
}
```

通过上述例子，我们看到Go语言设计的真是简洁，读channel就是<-chan， 而写通道自然就是chan<-，一目了然，印象深刻。 我们来是实现一个通过goroutine实现数字传递的例子，goroutine1循环将1，2，3，4，5传递给goroutine2，goroutine2负责将数字平方后传递给goroutine3，goroutine3负责打印接收到的数字。

分析该应用，我们需要至少2个channel，3个goroutine，其中main函数可以直接是第三个goroutine，所以再创建2个就够了。

```
package main

import (
  "fmt"
  "time"
)

var c1 chan int
var c2 chan int

func main() {
  c1 = make(chan int)
  c2 = make(chan int)
  //counter
  go func() {
    for i := 0; i < 10; i++ {
      c1 <- i //向通道c1写入数据
      time.Sleep(time.Second * 1)
    }
  }()
  //squarer
  go func() {
    for {
      num := <-c1      //读c1数据
      c2 <- num * num //将平方写入c2
    }

  }()
  //printer
  for {
```

```
    num := <-c2
    fmt.Println(num)
  }
}
```

这样执行完效果不太好，因为当第一个goroutine执行输出10个后，后面没有goroutine向通道写数据，这样就会出现Go语言不允许的情况，这种错误当然也是可预见的，就是代表进程被锁死了，所以Go语言给定义的错误是Deadlock（死锁）。

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

这是由于channel的知识点我们还需要知道，通道可以创建，也可以关闭，在读取的时候，也可以使用指示器变量来判断有没有问题，顺便提一下range在这里仍然可以读取channel，此时不需要“<-”。我们来尝试结束后关闭channel，然后优雅的结束整个进程。

```
package main

import (
  "fmt"
  "time"
)

var c1 chan int
var c2 chan int

func main() {
  c1 = make(chan int)
  c2 = make(chan int)
  //counter
  go func() {
    for i := 0; i < 10; i++ {
      c1 <- i //向通道c1写入数据
      time.Sleep(time.Second * 1)
    }
    close(c1) //关闭c1
  }()
  //squarer
  go func() {
    for {
      num, ok := <-c1 //读c1数据
      if !ok {
        break
      }
      c2 <- num * num //将平方写入c2
    }
    close(c2) //关闭c2

  }()
```

```
  //printer
  for {
    num, ok := <-c2
    if !ok {
      break
    }
    fmt.Println(num)
  }
}
```

特别注意，对通道的读写操作都会使goroutine阻塞，通道的关闭应该由写端来操作。此外，channel也可以作为函数参数，默认情况下一个channel是读写都可以的，为了防止不该写的goroutine发生写行为，Go语言设计了channel传递给函数的时候可以指定为单方向，读或者写！而这个单方向表述非常明确：

```
chan_name chan<- chan_type //只写通道
chan_name <-chan chan_type //只读通道
```

我们将上述的例子改造，因为三个goroutine对channel的操作就是读或者写。

```
 //counter, 对c1只写
  go func(out chan<- int) {
    for i := 0; i < 10; i++ {
      out <- i //向通道c1写入数据
      time.Sleep(time.Second * 1)
    }
    close(out)
  }(c1)
  //squarer, 对c1只读, 对c2只写
  go func(in <-chan int, out chan<- int) {
    for {
      num, ok := <-in //读c1数据
      if !ok {
        break
      }
      out <- num * num //将平方写入c2
    }
    close(out)

  }(c1, c2)
```

2.7.2 定时器

接下来我们来实现一个火箭发射的例子，准备一个倒数计时5秒，然后打印一个发射。当然这个例子可以用Sleep来控制每隔1s计数一次，不过在这里我们使用Go语言为我们提供的定时器来做这件事，定时器的关键也是channel。

```
//创建定时器
func NewTimer(d Duration) *Timer
//定时器结构体
type Timer struct {
  C <-chan Time
  r runtimeTimer
}
//定时器停止
func (t *Timer) Stop() bool
```

在time包中存在一个NewTimer，传入一个时间间隔n，获得一个Timer，Timer结构体中包含了一个C，这是一个通道类型，于是在时间n之后，C中会被写入时间戳。

```
timer := time.NewTimer(time.Second * 1)
  data := <-timer.C
  fmt.Println(data)
  timer.Stop() //停止定时器
```

timer只是一次性事件，不足以满足我们的需求，不过Go语言也提供了周期性定时器Ticker。

```
func NewTicker(d Duration) *Ticker
type Ticker struct {
  C <-chan Time // The channel on which the ticks are delivered.
  r runtimeTimer
}
```

周期性ticker，示例如下：

```
//创建周期性定时器
  ticker := time.NewTicker(time.Second)
  num := 5
  for {
    data := <-ticker.C
    fmt.Println(data)
    num--
    if num == 0 {
      break
    }
  }
  ticker.Stop()
```

接下来我们可以将考虑实现火箭发射的需求，定时器已经有了，可以来尝试。

```
package main

import (
  "fmt"
```

```
  "time"
)

func launch() {
  fmt.Println("发射!")
}

func main() {
  ticker := time.NewTicker(time.Second)
  num := 5
  for {
    <-ticker.C //读取无人接收
    fmt.Println(num)
    num--
    if num == 0 {
      break
    }
  }
  ticker.Stop()
  launch() //发射火箭
}
```

这样可以实现火箭发射的功能，不过如果临时想取消发射，该如何做呢？按ctrl+c的方式太简单粗暴了一下，比如想要按下任意键取消发射呢？

2.7.3 多路channel监控

我们很自然想到读标准输入就可以了，甚至也会立刻想到启动一个goroutine去监听标准输入，如果有输入，立即退出进程。

```
func cancel() {
  data := make([]byte, 10)
  os.Stdin.Read(data) //读标准输入
  os.Exit(1)          //退出进程
}
```

我们可以实现这样一个函数，读取标准输入，但是这样退出整个进程也不太优雅，我们还是想比较稳妥的退出。于是很多人想到，我们可以在建立一个channel，当标准输入有数据的时候，将数据写入该channel，在main函数中监控该channel，如果读到数据，则不执行后面的发射，直接return。但是问题来了，通道读都是阻塞的，我们的火箭发射还怎么做呢？在linux下我们知道多路IO监控可以使用select、poll、epoll等，在Go语言里，同样提供了一个机制对多路channel监控，这个机制的关键就是select-case语句。

```
for {
    select {
    case <-ticker.C:
      fmt.Println(num)
      num--
    case <-chan_stdin:
```

```
      fmt.Println("取消发射! ")
      return
    }
    if num == 0 {
      ticker.Stop()
      break
    }
  }
```

select 可以这也监控多个通道，当任一通道有数据写入时，select都会立即返回解除阻塞。完整代码如下：

```
package main

import (
  "fmt"
  "os"
  "time"
)

func launch() {
  fmt.Println("发射!")
}

func main() {
  ticker := time.NewTicker(time.Second)
  fmt.Println("开始倒计时准备发射，按回键可以取消发射! ")
  num := 5
  chan_stdin := make(chan string)
  go func(out chan<- string) {
    data := make([]byte, 10)
    os.Stdin.Read(data) //该goroutine读也会阻塞
    out <- "cancel"
  }(chan_stdin)
  for {
    select {
    case <-ticker.C:
      fmt.Println(num)
      num--
    case <-chan_stdin:
      fmt.Println("取消发射! ")
      return
    }
    if num == 0 {
      ticker.Stop()
      break
    }
  }
```

```
  launch() //发射火箭
}
```

3. 代码测试

作为一个优秀的开发者，任何代码可能的执行分支都应测试。在Go语言当中，官方包testing给我们提供了很好的测试服务。官方包为我们提供了两种测试：单元测试与压力测试。

至于单元测试，除了项目本身的代码外，还要提供专门针对某package或功能的测试文件。该文件有3个要求：

- 文件必须以"_test.go"作为结尾
- 文件内的测试函数必须以TestXxx开头，Xxx要求首字母必须大写
- 函数参数是 *testing.T 类型

比如我们测试，新建一个目录animal，在目录下创建两个文件，一个是源码文件，一个是测试代码文件。

```
//源码包
package animal

import (
  "fmt"
)

type Cat struct {
  Name  string
  Color string
  Age   uint
}

func NewCat(name, color string, age uint) *Cat {
  return &Cat{name, color, age}
}

func (c *Cat) Sleeping() {
  fmt.Println(c.Color, "Cat", c.Name, "is sleeping")
}

func (c *Cat) Eating() {
  fmt.Println(c.Color, "Cat", c.Name, "is Eating")
}

func (c *Cat) Print() {
  fmt.Printf("+%v\n", c)
}
```

```
//测试代码
package animal
```

```
import (
  "testing"
)

func Test_sleeping(t *testing.T) {
  c := NewCat("Sinmigo", "white", 20)
  c.Sleeping()
}

func Test_eating(t *testing.T) {
  c := NewCat("Sinmigo", "black", 20)
  c.Eating()
  if c.Color == "white" {
    t.Log("Eating测试通过")
  } else {
    t.Error("Eating测试不通过")
  }
}

func BenchmarkBigLen(b *testing.B) {
  //c := NewCat("Sinmigo", "white", 20)

  for i := 0; i < b.N; i++ {
    //c.Sleeping()
  }
}
```

接下来我们来说如何测试，可以借助Go语言的命令行来查看测试的帮助：

```
go help testflag
-v
-cover
```

对我们来说，主要关注两个参数

- v 显示测试详细信息
- cover 显示测试的覆盖率

上述的代码我们可以跑一下测试，看一下效果

```
localhost:animal teacher$ go test -v -cover
=== RUN   Test_sleeping
white Cat Sinmigo is sleeping
--- PASS: Test_sleeping (0.00s)
=== RUN   Test_eating
black Cat Sinmigo is Eating
--- FAIL: Test_eating (0.00s)
    ani_test.go:18: Eating测试不通过
FAIL
coverage: 75.0% of statements
exit status 1
FAIL  github.com/teacher/animal 0.005s
```

经过运行，我们看到代码测试覆盖率为75%，Eating测试不通过（我们人为造成的）。

在上面的测试代码中，我们可以看到一个BenchmarkBigLen函数，这个函数是Go语言当中性能测试的函数。这类性能测试的要求是：

- BenchmarkXxx 这样的函数声明，Xxx首字母要大写
- 函数参数为*testing.B

可以在运行的时候添加 -bench参数，来测试函数执行的性能，它的内部可以是执行数量巨大的循环。

```
localhost:animal teacher$ go test -bench=.
white Cat Sinmigo is sleeping
black Cat Sinmigo is Eating
goos: darwin
goarch: amd64
pkg: github.com/teacher/animal
BenchmarkBigLen-4      1000000000               0.266 ns/op
PASS
ok    github.com/teacher/animal 0.299s
```

bench后面是一个正则表达式，如果只针对某一个函数测试，写具体的函数名称也可以。

4. 包及包管理

4.1 包与init函数

Go语言是面向工程的一门开发语言，在一个工程内部，需要使用各种package，可以是自定义的，也可以是官方或来源于第三方的。对于任意一个包来说，都可以定义init函数，该函数原型为：

```
func init()
```

该函数会在该包被引用时自动运行，作用也很明显，就是为了初始化。当有多个包多层次引用时，init函数的执行的顺序是从里层引用到最外层，但是要注意的是每个init函数只会被执行一次。

上图中，包的引用顺序是B引用C，A引用B，那么最后init函数的执行顺序就是C-B-A。

我们可以用一些代码来测试一下关于包和init的理论，先看一下目录代码结构：

```
localhost:tutorial teacher$ tree
.
├── mathdemo
│   └── mathdemo.go
└── pkgdemo
    └── pkgdemo.go

2 directories, 2 files
```

看一下具体的代码

mathdemo.go

```
package mathdemo

import (
  "fmt"
)

func init() {
  fmt.Println("mathdemo'init is called")
}
```

pkgdemo.go

```
package pkgdemo

import (
  "fmt"

  _ "github.com/teacher/tutorial/mathdemo"
)

//外部可导出结构体
type ExternalPerson struct {
  Name string //大写，可导出
  Age  int
  sex  string //小写，不可导出
}

//内部不可导出结构体
type internalPerson struct {
  Name string
  Age  int
  sex  string
}
```

```
func init() {
  fmt.Println("pkgdemo'init is called")
}

//外部可导出函数
func NewPerson(name string, age int, sex string) *ExternalPerson {
  return &ExternalPerson{name, age, sex}
}

func NewInternalPerson(name string, age int, sex string) *internalPerson {
  return &internalPerson{name, age, sex}
}
```

通过阅读代码，我们发现pkgdemo.go包含mathdemo.go，如果我们在main函数中在引用pkgdemo，就形成了三级的引用关系。可以来观察一下，引用包的调用先后次序。 特别注意：当包不希望显示引用时，可以在包名前加_，此时仍然会调用对应的init函数。

4.2 包的导出

通过官方包的调用我们都知道，通过包名可以调用包内的变量、结构体、函数，我们把这种方式叫导出。Go语言当中包有一个基本的原则可以总结为5个字：首字母大写。只有首字母大写的变量、常量、结构体、结构体字段、方法、函数才可以被外部访问。

4.3 go module

Go语言在开发过程中经常会引用第三方包，早期在设计时，Go语言提供的go get命令可以帮助我们下载工程所依赖的源码包，虽然go get可以实现自动安装，但是在下载和处理上却经常存在一些问题，比如我们国内进行go get的时候，经常会遇到无法直接在google服务器下载的情况，Go语言在1.12版本推出了go module的处理方式，这给我们包管理带来了大大的便捷！

我们可以使用命令`go mod help`来查看mod操作的相关帮助，可以看到下面信息：

```
localhost: teacher$ go mod help
Go mod provides access to operations on modules.

Note that support for modules is built into all the go commands,
not just 'go mod'. For example, day-to-day adding, removing, upgrading,
and downgrading of dependencies should be done using 'go get'.
See 'go help modules' for an overview of module functionality.

Usage:

  go mod <command> [arguments]

The commands are:

  download    download modules to local cache
  edit        edit go.mod from tools or scripts
```

```
  graph       print module requirement graph
  init        initialize new module in current directory
  tidy        add missing and remove unused modules
  vendor      make vendored copy of dependencies
  verify      verify dependencies have expected content
  why         explain why packages or modules are needed

Use "go help mod <command>" for more information about a command.
```

其中对我们来说用的最多的是`init`和`tidy`，一般情况下，一个项目我们初始化后，就可以`go build`命令去编译了，此时工程缺少的第三方包，也会自动被安装到本地。而需要补充的时候，使用`tidy`就可以了。

我们来通过一个例子，来具体实践一下。

比如我们打算用web开发框架echo，来进行项目开发，我们准备跑一下echo的测试代码：

```
package main

import (
  "net/http"
  "github.com/labstack/echo/v4"
  "github.com/labstack/echo/v4/middleware"
)

func main() {
  // Echo instance
  e := echo.New()

  // Middleware
  e.Use(middleware.Logger())
  e.Use(middleware.Recover())

  // Routes
  e.GET("/", hello)

  // Start server
  e.Logger.Fatal(e.Start(":1323"))
}

// Handler
func hello(c echo.Context) error {
  return c.String(http.StatusOK, "Hello, World!")
}
```

将此段代码保存在非`$GOPATH`目录下，比如起一个工程名字就叫`echoserver`

```
mkdir ~/echoserver
cd ~/echoserver
```

将代码保存为`main.go`，接下来两部就可以将代码运行起来，首先初始化mod

```
localhost:echoserver teacher$ go mod init echoserver
go: creating new go.mod: module echoserver
```

之后我们就可以build代码了，不过很多时候会报错。

```
go: finding github.com/labstack/echo/v4/middleware latest
go: golang.org/x/crypto@v0.0.0-20190701094942-4def268fd1a4: unrecognized
import path "golang.org/x/crypto" (https fetch: Get
https://golang.org/x/crypto?go-get=1: dial tcp 216.239.37.1:443: i/o timeout)
go: error loading module requirements
```

我们需要设置一下代理，此后下载就方便很多。

```
# Enable the go modules feature
export GO111MODULE=on
# Set the GOPROXY environment variable
export GOPROXY=https://goproxy.io
```

此时，我们再来build就没有问题了

```
localhost:echoserver teacher$ go mod init echoserver
go: creating new go.mod: module echoserver
localhost:echoserver yekai$ go build
go: finding github.com/labstack/echo/v4 v4.1.11
go: downloading github.com/labstack/echo/v4 v4.1.11
go: extracting github.com/labstack/echo/v4 v4.1.11
go: finding github.com/stretchr/testify v1.4.0
go: finding github.com/labstack/gommon v0.3.0
go: finding golang.org/x/crypto v0.0.0-20190701094942-4def268fd1a4
go: finding github.com/valyala/fasttemplate v1.0.1
go: finding github.com/dgrijalva/jwt-go v3.2.0+incompatible
go: finding github.com/stretchr/objx v0.1.0
go: finding github.com/pmezard/go-difflib v1.0.0
go: finding gopkg.in/yaml.v2 v2.2.2
go: finding github.com/davecgh/go-spew v1.1.0
go: finding github.com/valyala/bytebufferpool v1.0.0
go: finding golang.org/x/net v0.0.0-20190404232315-eb5bcb51f2a3
go: finding golang.org/x/sys v0.0.0-20190412213103-97732733099d
go: finding github.com/mattn/go-colorable v0.1.2
go: finding github.com/mattn/go-isatty v0.0.9
go: finding gopkg.in/check.v1 v0.0.0-20161208181325-20d25e280405
go: finding golang.org/x/text v0.3.0
```

```
go: finding golang.org/x/crypto v0.0.0-20190308221718-c2843e01d9a2
go: finding golang.org/x/sys v0.0.0-20190813064441-fde4db37ae7a
go: finding github.com/mattn/go-isatty v0.0.8
go: finding golang.org/x/sys v0.0.0-20190215142949-d0b11bdaac8a
go: finding golang.org/x/sys v0.0.0-20190222072716-a9d3bda3a223
go: downloading github.com/labstack/gommon v0.3.0
go: downloading github.com/valyala/fasttemplate v1.0.1
go: downloading golang.org/x/crypto v0.0.0-20190701094942-4def268fd1a4
go: downloading github.com/dgrijalva/jwt-go v3.2.0+incompatible
go: extracting github.com/labstack/gommon v0.3.0
go: downloading github.com/mattn/go-isatty v0.0.9
go: downloading github.com/mattn/go-colorable v0.1.2
go: extracting github.com/valyala/fasttemplate v1.0.1
go: downloading github.com/valyala/bytebufferpool v1.0.0
go: extracting github.com/dgrijalva/jwt-go v3.2.0+incompatible
go: extracting github.com/mattn/go-isatty v0.0.9
go: extracting github.com/mattn/go-colorable v0.1.2
go: extracting github.com/valyala/bytebufferpool v1.0.0
go: extracting golang.org/x/crypto v0.0.0-20190701094942-4def268fd1a4
go: downloading golang.org/x/net v0.0.0-20190404232315-eb5bcb51f2a3
go: extracting golang.org/x/net v0.0.0-20190404232315-eb5bcb51f2a3
go: downloading golang.org/x/text v0.3.0
go: extracting golang.org/x/text v0.3.0
```

此时就可以将服务运行起来了：

```
localhost:echoserver teacher$ ./echoserver


   ____    __
  / __/___/ /  ___
 / _// __/ _ \/ _ \
/___/\__/_//_/\___/ v4.1.11
High performance, minimalist Go web framework
https://echo.labstack.com
_____________________________________O/_______
                                    O\
⇨ http server started on [::]:1323
```

需要注意的问题：

- 版本要在1.12.x以上
- 如果Go语言安装多个版本可能会有问题

五、拓展点

> 说明：
>
> 1. 典型面试题、笔试题；
> 2. 新技术 or 经验分享；
> 3. 未来计划、行业趋势分享；

六、总结

> 说明：
>
> 1. 回顾本堂课所有知识点；
> 2. 提示`注意点`和`重点`；
> 3. 提示学习方法；
> 4. 提示哪些需要记、哪些需要背、哪些代码需要敲；

- Go语言天然并发性；
- Go语言同步是继承和发扬光大的；
- 同步的案例应给予更多思考；
- 应建立并发的运行思想；
- Go语言测试应练习；
- Go语言包相关知识点要练习；

七、大作业

> 说明：
>
> 1. 给出明确的作业要求，形成文字、图片或测试题等形式；
> 2. 给出明确的解答方式；
> 3. 频次低，可设计为：1次/周 或 1次/2周；

可从下面的检测题题库任意选择。

八、集中答疑

> 说明：完成本堂课所有知识点的讲解后，由学员集中提问，讲师逐一解答；

九、检测题

> 说明：
>
> 1. 针对本堂课，设计5-10个题，由学员课下完成（课下刷题）；
> 2. 出题范围可从检查点里挑选，可以是面试题或笔试题；

> 3. 题型要求是 单选、多选、判断、填空中的一种或多种；
> 4. 频次高，原则上每次课都要有检测题；

十、下节课预告

- Go语言数据库编程