# go语言相比于其他语言的优势：

1、编译速度非常快：因为编译器只会关注那些直接被引用的库

2、采用goroutine对并发进行支持，不使用共享内存、全局变量，因为这需要复杂的锁规则来防止对同一个变量的不同步修改，而是使用channel来保证同一时刻只会有一个goroutine来修改数据，goroutine之间通过channel保证数据的同步。

3、支持面向对象，但却不支持继承，采用的是组合设计模式；并且具有独特的接口实现机制，允许用户对行为进行建模，而不是对类型进行建模。

4、自动gc

# GO语言的类型系统

**用户自定义的类型：**

创建结构类型变量时可以使用结构字面量来创建，可以用key ： value的格式，也可以用value的格式，但是后者的话，需要与声明一一对应。

1. 内置类型：数值、字符串、布尔。

引用类型：切片、映射、通道、接口、函数

**方法：**

在关键字func和和方法名之间增加一个参数，能给用户定义的类型添加新的行为，这个参数叫做接收者

如果该函数有接收者，那么就称为方法，否则称为函数。

接收者分为：值接收者和指针接收者。

当该方法的接收者是值时，你用指针去调用该方法，指针会被编译器调整为值，如：

lisa := &user("lisa", "lisa@email.com")

lisa.notify() （因为notify方法的接收者为值接接收者，所以编译器会将其调整为值，即（\*lisa).notify()) // &为取地址， \*为取值

反之，当该方法的接收者是指针时，你用值去调用该方法，值会被编译器调整为指针。

总结一下：值接收者使用值的副本来调用方法，而指针接收者使用实际值来调用方法。

bill := user("bill", "bill@email.com")

bill.changeEmail("bill@newdomain.com") (因为changeEmail方法的接收者是指针，所以会被调整为指针，即（&bil.changEmail())。

在声明一个新类型后，声明该类型的方法之前，需要先明白，如果你是想增加或删除一个值，那就应该用值接收者，如果只是想修改当前值，那么就应该使用指针接收者。

# 数组

//可以使用...自动计算声明数组的长度:

array := [...]int{10, 20, 30, 40, 50}

//可以声明数组并指定特定元素的值

array := [5]int{ 2 : 10}

//指针数组里面存的都是执行该类型的指针

array := [5]\*int{ 0: new(int)} --> 存的是指向整形的指针

复制指针数组，只会复制指针的值，而不会复制只想所指向的值，即复制之后，两个指针指向同一组字符串。

对于多维数组

//声明并初始化外层数组中索引为1和3的元素

array := [4][2]int{1 : {20, 21}, 3: {40, 41}}

//声明并初始化外层数组和内层数组的单个元素

array := [4][2]int{1 : {0 : 20}, 3: {1 : 41}}

因为在函数键传递变量时总是以值的方式传递的，所以在函数间传递数组是一个开销很大的操作，不管这个数组有多长，都会完整复制，并传递给函数。

所以最好的方法是传递指向数组的指针，这样一个8MB的数组就只需要8B的内存。

# 切片：

讲底层数组切出一部分，切片里存的都是指向底层数组的指针、长度和容量，各占8字节。所以函数传递时最好传递的是切片，而不是数组。

切片是围绕动态数组的概念构建的，通过内置函数append来实现。

**创建切片**

1、使用make函数创建切片可以指定切片的长度length和容量capacity，其中容量是底层数组的长度，但是能访问的只有前length个元素，可以只指定长度，默认长度和容量相等，推荐使用这种方式，尽量不要用var slice []type创建空切片，因为你下文可能直接用到索引，不指定长度会报空错误。

2、也可以直接使用切片字面量来创建切片，初始的长度和容量会基于初始化时提供的元素的个数确定。

3、使用索引声明切片：

slice ：= []string{99:""} // 创建了长度和容量都是100个元素的切片

**使用切片：**

使用切片创建切片时，共享的是同一个底层数组，只是看到的范围不同。如果一个切片修改了该底层数组的共享部分，另一个切片也能感知到。

计算切片的长度和容量：

对于底层数组容量为k的切片slice[i : j]:

长度 = j - i

容量 = k - i

**增长切片：**

当添加的元素超过底层数组的容量时，底层数组将进行扩容，当容量小于1000时，将成倍增加容量，如果超过1000，每次将增加25%。

使用切片创建切片时，可以使用三个索引创建切片，第三个索引的意思是，讲容量扩充到该位置：

slice[i : j : k]:

长度 = j - i

容量 = k - i

append可以用来增加切片的容量，具体看你追加的元素是否超过原来切片的容量，增长后的切片依旧指向底层数组，添加的元素对底层数组依旧会有影响。

为了不会影响底层数组的元素，在对切片进行切片时，可以强制指定新切片的长度等于其容量，这样当你append增长切片时，就会创建一个新的底层数组，实现了剥离。如：

source := []string{"1", "2", "3", "4", "5"}

slice := source[2:3:3]

source[3] = "KiWi"

slice = append(slice, "KiWi") // 创建新的底层数组["3", "KiWi"]， slice和source将不再指向同一个底层数组。

将一个切片s2追加到切片s1的后面:

append(s1, s2...)

**迭代切片：**

range创建了每个元素的副本，而不是直接返回对该元素的引用（slice[i]），因为迭代返回的变量是一个迭代过程中根据切片依次赋值的新变量，所以value的地址总是相同的。

# 映射

映射无序，因为使用了散列表

映射使用两个数据结构来存储数据。

第一个数据结构是一个数组，内部存储的是选择桶的散列键的高八位值，这个数组用于区分每个键值对存在那个桶里

第二个数据结构是一个字节数组，用于存储键值对，该字节数组先依次存储了这个桶里所有的键，之后依次存储了这个桶里所有的值。

映射的键可以是任何值，只要这个值可以用==运算符做比较，但不能是切片、函数、结构类型，因为这些类型具有引用语义，编译会报错。

映射的值可以使切片类型：如 dict := map[int][]string{}

映射中有一个独特的方法，叫delete，可以用来删除映射中的某一项，

如: delete（colors， “Coral”）

在函数间传递映射时，并不会制造出该映射的一个副本，如果函数中对这个映射做了修改，那么所有对该映射的引用都会察觉到这个修改。所以这个特性和切片类似，都不会传递数据，而只是传递指针。

# 接口

使用值接收者实现接口的方法，那么你可以用值和指针调用它，

但是使用指针接收者实现接口的方法，那么你只能用指针去调用它

方法集的规则如下：

Values Methods receivers

T （t T）

\*T (t T) and (t \* T)

Methods receivers Values

(t T) T and \*T

(t \*T) \*T

go中似乎只有通过接口实现编译时多态，通过传入不同的类型（这些不同的类型都实现了接口类型，所以形参是接口类型，实参是实现类型），实现方法的重载；

因为没有继承机制，没法实现方法的重写，所以不存在运行时多态。

# 嵌入类型

要嵌入一个类型，只需要声明这个类型的名字就可以了，如我们讲user类型嵌入admin类型当中

type admin struct{

user // 嵌入类型

level strintg

}

内部类型user实现了一个notify方法，那么外部类型admin也可以调用该notify方法。

那么假如user实现了notifier接口，就相当于admin也实现了notifier接口。

这就是内部类型的提升。

但假如外部类型admin也实现了notifier接口，那么内部类型将不会提升，将会各自调用各自的notify方法。

# 公开和未公开的表示符

当一个标识符的名字以小写字母开头时，这个标识符就是没公开的，即包外的代码不可见，就算你import这个包，还是不可见。

如果想可见，那么可以设置一个工厂函数New，返回该标志符，注意，将工厂函数命名为New是Go语言的一个习惯，

但是要使用短变量声明操作符进行操作，因为永远不能创建一个未公开的类型的变量。

所以如果你声明了一个公开的自定义的类型，那么其属性也最好实现为公开的，即都用大写字母，不然的话，就算在别的包中应用了这个类型，却还是不能调用该属性，这就很尴尬了。

# 并发

Go语言中的并发值得是能让某个函数独立于其他函数运行的能力。

Go语言的并发同步模型来自一个叫做通信顺序进程（csp）的范型，是一种消息传递模型，通过在goroutine之间传递数据来传递消息，而不是对数据进行加锁来实现同步访问。

进程：包含了所有可能分配的常用资源的进程，这些资源但不限于内存地址空间、文件和设备的句柄以及线程。

一个线程是一个执行空间，这个空间会被操作系统调度来运行函数中所写的代码。每个进程的初始线程称为主线程。操作系统讲线程调度到某个处理器上运行，这个处理器不一定是进程所在的处理器，不同操作系统使用的线程调度算法一般都不一样但是这种不同会被操作系统屏蔽，并不会展示给程序员。

操作系统会在物理处理器上调度线程来运行，而Go的运行会在逻辑处理器上调度goroutine来运行。

在1.5版上，默认为每个物理处理器分配一个逻辑处理器，这些逻辑处理器会用于执行所有被创建的goroutine，即便只有一个逻辑处理器，Go也可以以神奇的效率和性能，并发调度无数个goroutine。

如果创建一个goroutine并准备运行，会被放到调度器的全局运行队列中，之后，调度器将这些队列的goroutine分配给一个逻辑处理器，并放到这个逻辑处理器对应的本地运行队列中，本地运行队列中的goroutine会一直等待知道自己被分配的逻辑处理器执行。

当正在运行的goroutine需要执行一个阻塞的系统调用，如打开一个文件。当这类调用发生是，线程和goroutine会从逻辑处理器上分离，该线程会继续阻塞，等待系统调用的返回。调度器会创建一个新线程，并将其绑定到该逻辑处理器上。

要实现并发，可能会让goroutine在不同的线程上运行。

使用NumCPU来返回可以使用的物理处理器的数量。

可以通过runtime.GOMAXPROCS(runtime.NumCPU())来修改每个可用的物理处理器上逻辑处理器的数量。

要想让goroutine并行运行，并不是要让逻辑处理器的数量等于goroutine的数量，而是让物理处理器的数量等于goroutine的数量，显然不显示。

竞争状态时共享数据出错的原因是，每个goroutine都会拷贝共享数据到本地，对副本进行操作，然后再将副本写入共享数据，这样的话容易覆盖之前另一个goroutine写的数据。

可以使用go build -race来检测产生竞争状态的代码行。

可以通过atomic包的几个函数和sync包里的mutex类型对共享资源进行加锁。

例如atomic包中的AddInt64可以直接对共享整形进行操作，保持同步，强制同一时刻只能有一个goroutine运行并完成这个加法操作。类似的还有读（LoadInt64）和写（StoreInt64）。

用的最多的还是互斥锁mutex，用于在代码上创建一个临界区，保证同一时间只有一个goroutine可以执行这个临界区代码。

因为不论开启了多少个goroutine，调用的还是同一段代码，这要该代码被锁住了，那么也不可能对共享资源进行操作的代码也就被锁住了。

原子函数和互斥锁都能工作，但是依靠它们都不会让编写并发程序变得更简单，更不容易出错，或者更有趣，这个时候就引入了通道，通过发送和接收共享的资源，在goroutine之间做同步。

原子函数是直接对共享变量进行操作，互斥锁是只让一个goroutine对资源进行操作。

而chan是将共享资源存在通道中，**谁要用，就去拿，用完之后记得放回来**，那么每个goroutine之间就不会出现创建副本，在由副本赋值给共享变量，引起覆盖写的情况。因为你任何时候操作的都是共享资源。

通道里有值，goroutine才会运行。

如：

buffered := make(chan string, 10) // 缓冲区大小为10的字符串通道

buffered <- "GoPher"

value := <- buffered

无缓存的通道：指在接收前没有能力保存任何值的通道，要求发送方和接收方同时准备好，才能发送。也就是说发送和接收两个是绑定在一起的。可以模拟网球比赛和接力赛，因为球和接力棒必须在其中一方手上，不能悬空。

有缓冲的通带：指在接收前能存储一个或者多个值的通道。

前者用于两个goroutine之间，一个往通道里写，一个从通道里取，后者适合多个goroutine，因为资源很多，那么就不存在竞争问题。

值得注意的是：goroutine依旧和从关闭的通道中接收数据，但是却不能再发送数据。

# 深入理解并发安全的sync.Map

## Map和sync.Map的比较

Golang中内置map关键字，但是是非线程安全的，故加入sync.Map，提供并发安全的map。







但是当map的数据量非常大的时候，会引发大量goroutine争夺同一把锁，这种现象将直接导致性能的急剧下降。有点类似于java中的hashMap和concurrentHashMap。







## 2、load



对dirty加锁，对read不加锁，减少锁的争用。原理就是只将数据锁一段时间，也就是只锁dirty，当dirty满了，那么就拷贝read，read是不加锁的。而满没满是看misses值是否小于les(m.dirty)。而且因为不管有没有从dirty中拿到数据，misses都会自加1，所以保证了数据不会长时间加锁。对数据加锁的方式就是，锁掉读数据的代码，这样就不会存在多个goroutine同时读数据了。而且会对read，进行两次读，防止在对读dirty数据的代码加锁的时候，dirty中的数据已经拷贝到read了。



## 3、Store



Read中是否有key，

有且未标记删除，直接修改指针的val值。

有但标记删除，插入dirty，修改read的key的指针指向即可。

Read中没有key

dirty中有该键，更新val

dirty中也没有，将read中赋值未删除的数据到dirty（保持dirty中数据是最新的），然后再在dirty中用newEntry新增entry。



## 4、Range



## **5、Sync.Map的使用例子**



## **适用场景**

Sync.Map并不是为了代替锁+map的组合。

两种情况应该选择sync.Map

1. key值一次写入，多次读取，因为总是会double check read
2. 多个goroutine的读取、写入和覆盖在不相交的key集。

# 原子操作和互斥锁的区别

原子操作即是进行过程中不能被终端的操作，针对某个值的原子操作在被进行的过程中，cpu绝不会再去进行其他的针对该值的操作。为了实现这样的严谨行，原子操作仅会由一个独立的cpu指令代表和完成。原子操作是无锁的，常常通过cpu指令直接实现，事实上，其他同步及时的实现常常依赖与原子操作。

## 1、Go对原子操作的支持：

Go语言的sync/atomic包提供了对原子操作的支持，用于同步访问整数和指针。

五种：增减、比较并交换、载入、存储、交换。

原子操作支持的类型包括int32，int64、uint32、uint64、uintptr、unsafe.Pointer

下面演示两种原子操作的实例：

AddInt32



CompareAndSwarp



## **2、原子操作和互斥锁区别：**

首先二者均不可被其他线程打断，

Atomic操作的优势是更轻量，比如CAS可以在不行成临界区和创建互斥量的情况下完成并发安全的值替换操作。这可以大大的减少同步对程序性能的损耗。

当然也有劣势，趋于乐观，总是假设被操作值未被改变，并一段确认这个假设的真实性就立即进行值替换，那么在被操作值被频繁变更的情况下，CAS操作并不容易成功。相反，互斥锁的做法趋于悲观，总假设会有并发的操作要修改被操作的值，并用锁将相关操作放入临界区进行保护。

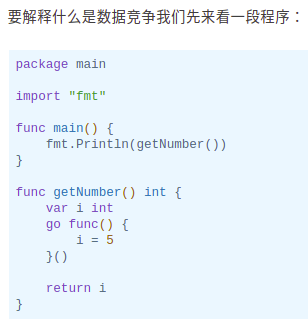
总结：

1. 互斥锁是一种数据结构，用来让一个线程执行程序的关键不分，完成互斥的多个操作。
2. 原子操作是针对某个值的单个互斥操作。
3. 互斥锁可以理解为悲观锁，共享资源每次只给一个线程使用，其他线程阻塞，用完后再把资源转让给其他线程。

# Go并发变成里的数据竞争以及解决之道

Go语言以容易进行并发编程而闻名。

## 数据竞争



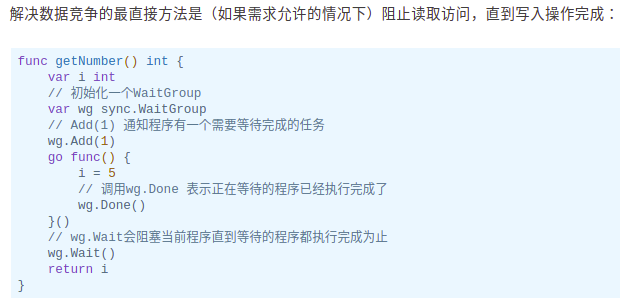
这段代码有两个携程：

1. 主携程：返回i
2. 自定义携程：设置i

那么可能返回的i值是0或5，那么这就产生了数据竞争。

## 2、解决数据竞争

### 1、使用WaitGroup



类似于信号量机制

Var wg waitGroup

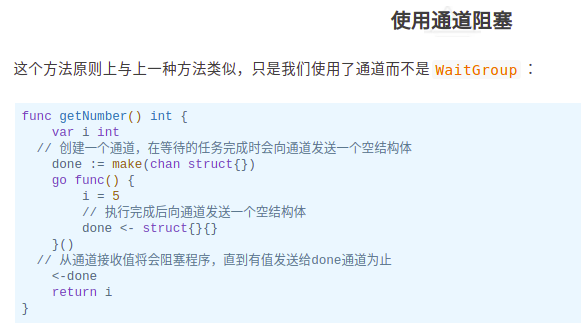
Wg.add(1) // 开几个协程，就add多少

Wg.done

Wg.wait

### 2、使用通道阻塞

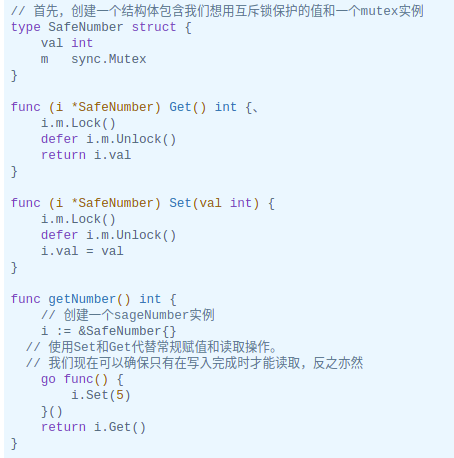
通道中如果没有值，但如果你要从通道里取值，那么会阻塞程序，直到有值取出，我们可以在我们期待的优先执行的goroutine中给通道存值，后运行的goroutine中取值来实现避免数据竞争。



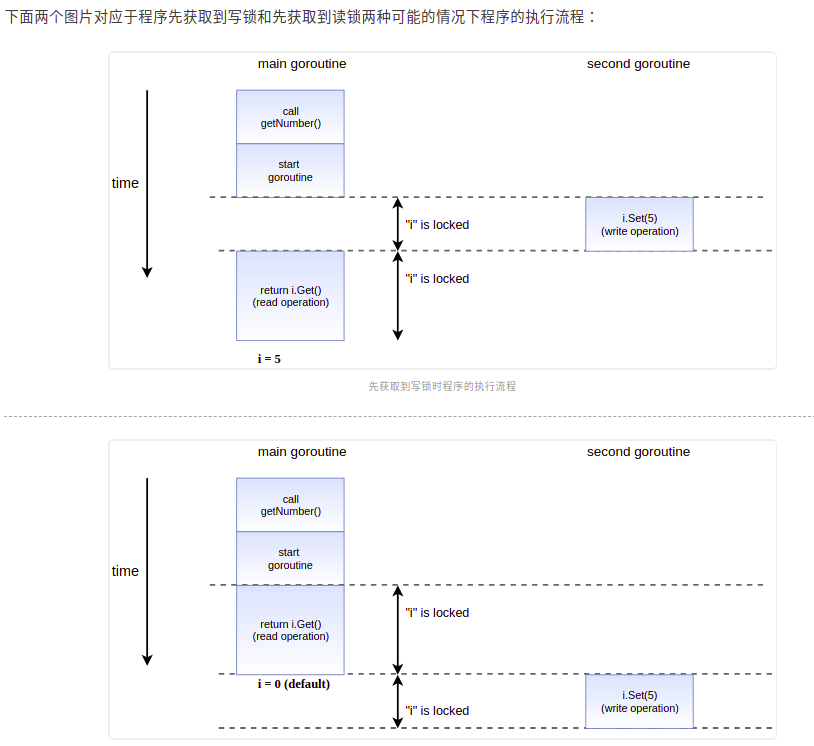
通道的好处是，既能实现协程之间的同步，又能够进行通讯和资源的共享。

### 使用Mutex

上诉两种方法，保证的是先写入后读取，但假如不需要保证顺序，只需要保证二者是互斥的，不能同时发生即可，那么应该考虑使用Mutex互斥锁。



I是互斥资源，只能有一个协程访问，所以也就是用互斥实现了同步



### mutex Vs channel

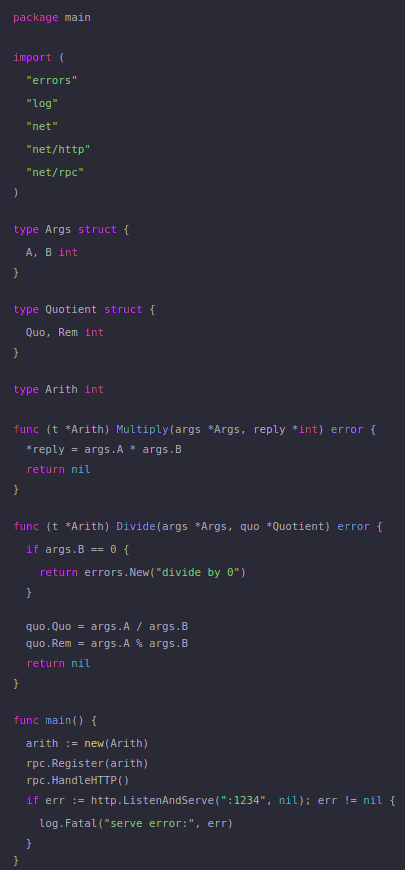
大多数新手都视图用通道解决所有并发问题，这是go语言的一个很酷的特性，这是不对的，二者都各有好处。

通常，当goroutine需要相互通信时使用通道，当确保只有一个goroutine能访问代码的关键部分时使用互斥锁。比如这个问题，互斥锁更为适合，你在通道里面传入的空结构体并没有任何通信的作用。

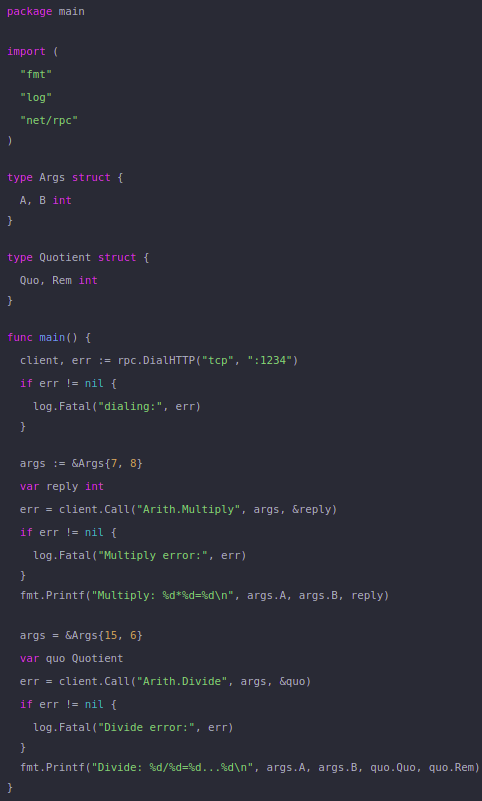
# GO标准库net/rpc

## 基于HTTP RPC同步调用：

服务端：

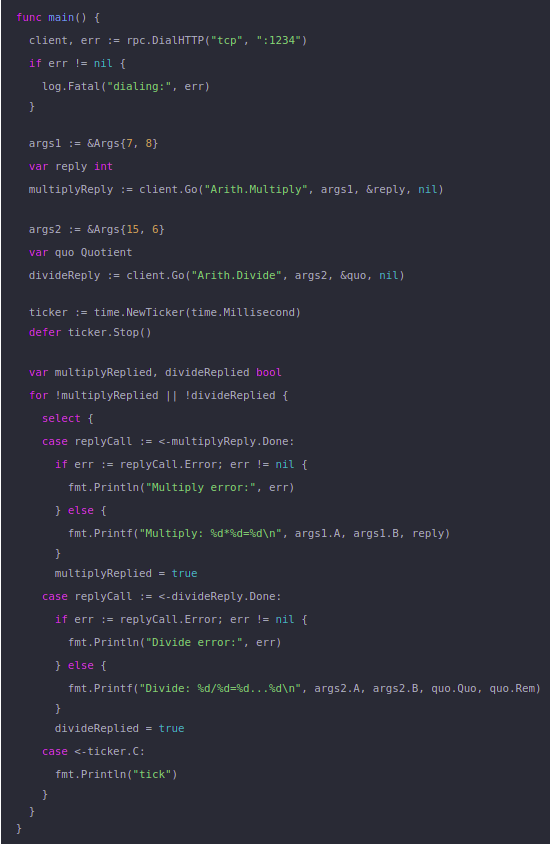


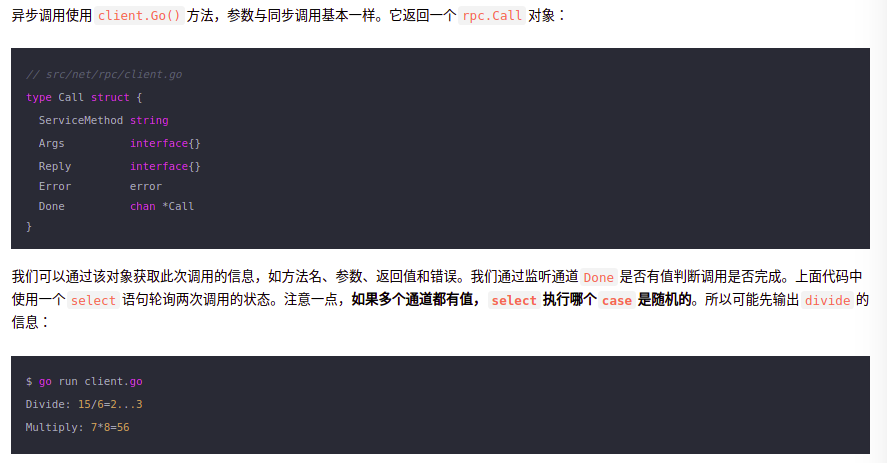
客户端：



## 基于HTTP RPC异步调用：

客户端：





## TCP



## 定制方法名

