# Go并发面试题

1. **Go的并发模型**
   1. 共享内存并发模型：

通过直接共享内存 + 锁的方式同步信息，传统多线程并发

* 1. CSP并发模型（communicating sequential process）：

通过goroutine和channel实现用发消息的方式来同步信息。

1. **GO的并发同步原语**
2. **WaitGroup的实现原理**
3. **Go Cond实现原理**
4. **安全读写共享变量的方式**
5. **排查数据竞争问题**

**3.2.1-5**

用户进程不能直接访问硬件资源，内核可以与硬件交互

虚拟内存的最大容量与物理内存无关

内核会为每个用户进程分配的是虚拟内存而不是物理内存，分配在用户空间，每个用户进程都以为分配给它的是整个用户空间

当两个用户空间的一部分指向物理内存的同一个页框时是允许的，这正是共享内存的基础

**3.2.2**

所有的系统调用都属于原子操作

**3.2.3 管道**

1. 管道是一种半双工的通信方式（可以接收或发送但不能同时进行）

（单工通信：传输方向只有一个方向，单工通信只有一根数据线，它也只在一个方向上进行，如打印机、电视机等。

半双工通信：可以双向通信，但只能轮流传输，也只有一根数据线，不同于单工通信的是这根数据线即可作为发送又可作为接收，虽数据可在两个方向上传送，但通信双方不能同时收发数据。

全双工通信：可以同时双向传输数据，数据的发送和接收用两根不同的数据线，通信双方在同一时刻都能进行发送和接收，发送和接收同时进行，没有延迟。）

1. 使用缓冲读取器的好处就是可以非常方便和灵活地读取需要的内容，而不是只能先把所有内容先读取出来再做处理。
2. 匿名管道会在管道缓冲区被写满之后使写数据的进程阻塞。
3. Linux管道可以把一个命令的输出作为另一个命令的输入，Go代码也可以做到这一点，而且实现起来很简洁。（示例）
4. 匿名管道：父子进程通信；命名管道：任何两个进程。（示例）
5. 命名管道默认是阻塞式的。只有在对这个命名管道的读操作和写操作都已准备就绪之后，数据才开始流转。
6. 命名管道默认会在其中一端还未就绪的时候阻塞另一端的进程。
7. 在使用close（）方法关闭管道的某一端后，另一端在写数据或读数据的时候会得到一个预定义的error类型值。

**3.2.4 信号**

（1）信号是IPC中唯一一种异步通讯的方法。

（2）信号的来源有：键盘输入，硬件故障，系统函数调用和软件中的非法运算

（3）进程响应信号的方式有3种：忽略、捕捉和执行默认操作

（4）对于绝大多数标准信号而言，我们可以自定义程序对它的响应方式（函数表示）

（5）类Unix操作系统的SIDKILL和SIGSTOP两种信号既不能自行处理也不能被忽视。

（6）函数signal.Stop（）会取消掉在之前调用signal.Notif（）函数时告知signal处理程序需要自行处理若干信号的行为。只有把当初传递给signal.Notify函数的哪个signal接收通道作为调用signal.Stop（）函数时的参数值，才能取消掉之前的行为。

（7）在调用signal.Stop（）函数后，应使用内嵌函数close关闭该signal接收通道。

proc,err:=os.FindProcess(ID) //通过进程ID找到该进程

err=proc.Signal(syscall.SIGINT) //向该进程发送信号syscall.SIGINT

**3.2.5 socket（套接字）【通过网络连接通信、传数据】**

（1）在面向有链接的socket之间传输数据之前必须建立逻辑链接。面向无连接的socket则不需要。链接暗含了双方的地址。

（2）在面向无连接的socket上，数据流只能是单向的，不能即发送又接收

（3）使用socket时，一般把第三参数设为0，意思是让其根据第一第二参数自行选择协议。

（4）系统调用socket会返回一个int类型的值，该值是socket实例唯一标识符的文件描述符。

**Socket编程：**

1. listener,err := net.Listen(“协议”,”host:port”) //得到一个监听器

//返回值net.Listener和error

1. conn,err := listener.Accept() //阻塞，待连接后得到一个连接

//返回值是net.Conn和error

1. func Dial(network,address string) (Conn,error) //Dial函数用于向参数里 的网络地址发送连接建立申请
2. func DialTimeout(“tcp”,”127.0.0.1:8085”,2\*time.Second)

//设定Dial函数的连接超时时间

**net.Conn接口类型的8个方法：**

1. **conn.Read**（b []byte)(n int,err error) //读到的数据写入字节切片b

//n是本次操作实际读到的字节数

实际应用中可以包装变量conn

reader:=bufio.NewReader(conn) //给conn附加缓冲

line,err:=reader.ReadBytes(‘\n’) //第一次读到‘\n’时停止

1. **conn.Write**(b []byte) (n int, err error) //将切片b中数据写入连接，同样可以包装

writer:=bufio.NewWriter(conn)

1. **conn.Close()**

当Read或Write函数还在执行时，也会立即结束并返回一个非空的err值

1. **conn.LocalAddr()** //得到一个net.Addr接口类型值，其有两个方法

//NetWork和String

conn.LocalAdrr().Network() 可以的到“tcp”

conn.LocalAdrr().String（） 可以的到“127.0.0.1:8085”

1. **conn.RemoteAdrr()** //和前一个类似
2. **conn.SetDeadLine()**

b:=make([]byte,10)

for{

conn.SetReadline(time.Now().Add(2\*time.Second))

n,err:=conn.Read(b) //设置每次读操作时间不超过两秒

}

1. **conn.SetReadDeadLine()**
2. **conn.SetWriteDeadLine()**

**注意：**

1. 在服务端，我们应该尽量为每一个链接开启一个goroutine，进行独立的处理，并且使这些处理过程并发执行
2. 不论怎样，把当前连接及时关闭都是一件很重要的事conn.Close（）
3. 重点记忆往连接conn里读和写的细节P89

**3.3 多线程编程**

多线程编程是一种比多进程编程更加灵活、高效的并发编程方式，绝大多数现代操作系统都支持它。

3.3.1

1. 每个进程都有一个主线程，它随进程而创建
2. 线程不可独立于进程
3. 每个线程都有属于自己的线程栈，都包含在其所属进程的虚拟内存地址中
4. 每个线程也有属于自己的ID（TID），只在所属进程范围内唯一
5. 同一进程的所有线程都是平等的，可以对其他线程进行有效的管理，如下：

线程的管理：

1. 、创建线程 系统调用：pthread\_create
2. 、终止线程 系统调用：pthread\_cancel
3. 、连接已终止的线程 系统调用：pthread\_join
4. 、分离线程 系统调用：pthread\_detach
5. 、终止自身 系统调用：pthread\_exit
6. 终止：
7. 在线程start函数中执行return语句，会使该线程随着start函数的结束而终止。
8. 但是在主线程中执行return，所有线程都会终止；在任意线程调用exit（）也会达到同样的效果
9. 只终止自身线程：调用pthread\_exit,（主线程也只会终止自身）

**3.3.1-4线程调度**

1. I/O受限的线程具有更高的动态优先级优先使用CPU
2. 所有等待使用CPU的线程会按照动态优先级从高到低的顺序排列
3. 调度器会尽量使一个线程在一个特定的CPU上运行

**3.3.1-5线程实现模型**

**用户级线程模型：**

1. 多对一（多用户线程对一内核线程）
2. 此模型的线程是由用户线程库全权管理的
3. 如果一线程被阻塞，其所属进程也会被阻塞
4. 一个进程的多个线程无法被分配给不同的CPU运行

**内核级线程模型：**

1. 一对一
2. 进程中一个线程阻塞，进程也不会受到影响
3. 现代很多操作系统都是以内核级线程模型实现的包括Linux

**两级线程模型：**

1. 多对多
2. 实现复杂，不会被采用
3. Go的并发模型与两级线程模型在理念上非常类似
4. goroutine并不是线程

**3.3.2线程的同步**

互斥量的注意事项：

1. 必须保证互斥量只会被初始化一次
2. 互斥变量只能被锁定一次，不可以二次加锁；同样不可以重复解锁
3. 对一个互斥变量的锁定和解锁应该成对出现
4. 离开临界区时必须解锁互斥量
5. 应该尽量少地使用互斥量。每个互斥量保护的临界区应该在合理范围内并尽量大。可以一定程度提高程序的整体性能。
6. 不同的互斥量保护的临界区不应该包含对同一个共享资源的访问
7. 尽量不要让不同互斥量所保护的临界区重叠
8. 当第无法做到临界区不重叠时，有两种解决办法：“试锁定-回退”（先锁定一个，然后再试着锁定另一个，锁定失败就回退）和“固定顺序锁定”（只适用于对多个临界区顺序访问的情况）；这两种通常会混合使用
9. “试锁定-回退”和“固定顺序锁定”是下策之选，保持共享数据的独立性是预防因使用互斥量而导致死锁的最佳方法。

**条件变量**：

在对应的共享数据的状态发生变化时，通知其他因此而被阻塞的线程。其总是与互斥变量组合使用。

条件变量注意事项：

1. 使用前必须创建和初始化，且初始化必须唯一。
2. 在使用之前必须与某个互斥量进行绑定。
3. 条件变量提供的操作有三种：等待通知、单发通知（至少一个）、广播通知（所有）
4. 在执行等待通知操作时必须先解锁与该条件变量绑在一起的互斥变量然后再使当前进程阻塞。
5. 互斥量可以实现对了临界区的保护，并会阻止静态条件的发生。条件变量作为补充手段，可以让多方协作更有效率。

**3.3.3线程安全性**

1. 如果有一个代码块，它可以被多个程序并发执行，并总能产生预期的结果，那么该代码块就是线程安全的。如果某代码块包含了对共享数据的更新操作，那么这个代码块就可能是非线程安全的。
2. 让函数具有线程安全性的最有效方式是使其可重入，如果某个进程中的所有线程都可以并发的对一个函数进行调用，并且无论他们调用该函数的实际执行情况怎么样，该函数都可以产生预期的结果，那么就可以说这个函数是可重入的
3. 更通俗的讲，如果多个线程并发的调用该函数，与它们任意依次调用它所产生的效果总是相同的，那么该函数就是一个可重入函数。
4. 如果一个函数把共享数据作为他返回的结果，或者包含在它返回的结果中，那么该函数就肯定不是一个可重入函数。因此，为了使函数可重入，我们必须只能杜绝在该函数的返回结果中掺杂任何共享数据。
5. 是函数成为可重入函数，是实现其线程安全性最直接的方式，并且也是最简单和最高效的方式。如果做不到，我们应该使用互斥变量把相关的代码保护起来。
6. 如果某个局部变量的值中包含了对共享数据的引用，就不能说该局部变量所属的函数是可重入的。

**3.4多线程与多进程**

在某些情况下，比如对信号的处理、同时运行多套不同的程序以及包含多个需要超大内存支持的任务等，传统的多进程编程方式可能会更加适合。

**3.5多核时代的并发编程**

1. 让程序真正在多核CPU上并行运行起来的前提是采用某种并发编程方式来编写程序。
2. 让程序更加高效的并发运行。高效指的是在保证程序正确性和可伸缩性的前提下，提升程序的响应时间和吞吐量。
3. 保证并发程序正确性和可伸缩性的建议：
4. 控制临界区的纯度。临界区中仅应包含操作共享数据的代码。
5. 控制临界区的粒度。由于粒度过细的临界区会增加底层协调工作的发生次数，所以有时候你需要粗化临界区。如果存在相邻的多个临界区，并且它们的内部都是操作同一个共享数据的代码，那么就合并它们。
6. 减少临界区中代码的执行耗时。提高临界区的纯度，可以减少临界区中代码的执行耗时。
7. 避免长时间持有互斥量。
8. 优先使用原子操作，而不是互斥量。使用互斥量一般会比使用原子操作造成大得多的程序性能消耗。

**第四章**

Go语言的标语：不要使用共享内存的方式来通信。作为代替，应该使用通信作为手段来共享内存

Go推荐使用channel在多个goroutine之间传递数据。并且还会保证线程的安全性

**4.1.1线程实现模型：**

M：（machine）内核线程

P：（processor）执行一个Go代码片段必须的资源【“上下文”】

G：（goroutine）

（1）一个goroutine的执行需要上下文环境和内核线程的支持，一个内核线程在与一个上下文环境关联之后就形成了一个有效的goroutine的运行环境。

（2）M与P是一对一的，P与G是一对多的

（3）虽然go函数是可以有结果声明的，但是他们返回的结果值会在其执行完成时被丢弃。也就是说，即使他们返回了结果值，也不会产生任何意义。

（4）我们不应该对并发执行的先后顺序有任何的假设。需要通过额外的手段去实现。

（5）runtime.Gosched()函数的作用是暂停当前的goroutine，好让其他的goroutine有机会运行。

**4.2.2主goroutine的运作**

主goroutine的任务：

1. 设定每一个goroutine所能申请的栈空间的最大尺寸。
2. 检查当前内核线程是否是runtime.m0。不是则抛出异常。
3. 创建一个特殊的defer语句，用作退出时的善后处理
4. 启用专用于在后台清扫内存垃圾的goroutine，并设置可GC的标识
5. 执行mian包中的init函数

**4.2.3runtime包与goroutine**

1. runtime.GOMAXPROCS（），设置常规运行时系统中的P的最大数量。
2. runtime.Goexit（），会立即使当前goroutine的运行终止，而其他goroutine并不会受此影响。
3. runtime.Gosched（），暂停当前goroutine的运行。
4. runtime.NumGoroutine（），返回当前Go运行时系统处于非Gdead状态的用户的数量。这些被视为“可被调度运行的“。该函数的返回值总是会大于等于1。
5. runtime.LockOSThread（）和runtime.UnlockOSThread（），对前者的调用会使当前goroutine与当前M锁定在一起，而对后者的调用则会解除这样的锁定。
6. runtime/debug.SetMaxStack（），约束单个goroutine所能申请栈空间的最大尺寸。如果运行时系统在为某个goroutinr增加栈空间的时候，发现他的实际尺寸已经超过了设定值，就会发起一运行时恐慌并终止程序的运行。
7. runtime/debug.SetThreads（），对Go运行时系统所使用的内核线程（M）的数量进行设置。
8. runtime/debug.SetGCPercent（）【设定触发自身GC的条件】、runtime.GC（）和runtime/debug.FreeOSMemory（）【后两者用于发起手动GC】。自动GC在默认情况下是并发运行的，而手动GC则总是串行运行的，这也意味着在后两个函数的执行期间，调度是停止的。另外，最后一个比runtime.GC（）多做了一件事，那就是在GC之后还要清扫一次堆内存。

**4.3 channel通道【引用类型】**

channel提供了一种机制。它既可以同步两个并发执行的函数，又可以让这两个函数通过相互传递特定类型的值来通信。

**4.3.1 channel的基本概念**

通道类型的变量是用来传递值的，而不是存储值的。

Channel注意事项

1. 在同一时刻仅有一个goroutine可以向一个通道发送元素值，同时也仅有一个goroutine可以从通道接收值。
2. 通道在使用之前必须用内建函数make来初始化。
3. 判断接收：elem,ok:=<-channel1 ok==false表示通道已经关闭且为空，并且elem被赋予该通道类型的零值
4. 试图向一个未被初始化的通道（即值为nil的通道）发送或者取数据都会造成当前goroutine永久阻塞。向一个关闭的通道发送则会造成一个恐慌。
5. 发送入通道时，通道会复制值；如果通道满，则发送的goroutine阻塞；如果通道空，且已有别的goroutine等待接收，则直接复制给等待的goroutine。所以经过通道传递的值至少会被复制一次至多会被复制两次
6. 接收时得到通道中的副本，若通道为空则当前goroutine阻塞
7. 对于同一个元素值来说，把它发送给某个通道的操作，一定会在从通道中接收它的操作完成之前完成。
8. 通道的缓冲队列是环形队列。
9. 通道仅允许关闭一次，重复关闭会引起恐慌（关闭nil通道会引起恐慌）。
10. 利用函数可以把双向通道转换为单向通道。单向通道不应该出现在变量的声明中。
11. range迭代通道会将值全部取出。range不能迭代输入通道。
12. 非缓冲通道：发送时会阻塞，直至被接受。接收时会阻塞，直至有发送。接收方或发送方有多个时也需排队“握手”。
13. struct{}代表的是不包含任何字段的结构体类型，也可称为空结构体类型。在Go语言中，空结构体类型的变量是不占用内存空间的，并且所有该类型的变量都拥有相同的内存地址。建议用于传递信号的通道都以struct{}作为元素类型。

（赋的值为struct{}{}）

1. 当通道里面的值是引用类型时，对元素值的副本的修改会影响到原值。

**4.3.3 for语句与channel**

（1）range子句可以持续地从一个通道接收元素。其迭代的目标不能是一个发送通道。也不能是未初始化的通道。

**4.3.4 select**

Select语句是一个仅能用于通道发送和接受操作的语句。只会选择一个分支执行。

**Select注意事项：**

1. 跟在case后面只能是针对某个通道的发送和接收语句。
2. 无论某个case会不会被执行，其后面的表达式都会被求值。
3. 如果一个case后面的语句可以被立即执行那么这个case就会被选中。如果几个case同时满足，系统用伪随机算法选择一个。
4. 如果所有case都不满足上述条件，那么就执行default case。如果也没有default case那么当前goroutine就会被阻塞于此，直到有一个case满足条件。如果是主goroutine那么就会发生死锁。
5. break语句可以结束当前select语句。搭配使用标签可以跳出select语句之外的for循环语句。
6. 在实际应用场景中我们常常需要把select语句放到一个单独的goroutine中执行，这样即使select语句阻塞了也不会造成死锁。

**4.3.5 非缓冲的channel**

（1）非缓冲通道只能同步地传递元素值，而缓冲通道以异步的方式传递元素值

（2）发送时阻塞直到有接收（接收goroutine先得到副本，然后再唤醒发送goroutine，接收操作会先完成），接收时阻塞直到有发送

（3）调用内建函数cap就可以知道一个通道是否带有缓冲

（4）如果通道是非缓冲的却想要异步地执行发送操作，就需要另行异步化，比如启用额外的goroutine执行此操作。

**4.3.6 time包和channel**

定时器 time.Timer是一个结构体

1. time.Timer无法在外进行初始化。想得到此类型的值可以用time.NewTimer（）函数和time.AfterFunc（）函数。
2. Timer：=time.NewTimer（3\*time.Hour+36\*time.Minute）
3. 这里的timer是\*time.Timer类型而不是time.Timer类型。它有两个方法Reset（）和Stop（），两个方法都返回一个bool值，如果是false表明当前方法调用之前定时器已经停止。
4. 在time.Timer类型中，对外通知定时器到期的途径就是通道，由字段c表示（time.C）
5. 在需要频繁地使用相对到期时间相同的定时器，应该尽量复用，而不是重新创建。复用时应该确保旧的通知已被接收。
6. 如果你在定时器到期之前停止了它，那么该定时器的字段c也就没有机会缓冲任何元素值了。此时，在重置之前一定不要对它的c字段执行接收操作。因为会造成当前goroutine永久阻塞。
7. 到期时间应该为正整数。否则该定时器会立即到期，无定时效果。
8. time.AfterFunc函数同样会返回新建的定时器。不过这样的定时器在到期时，并不会向它的通知通道发送元素值，取而代之的是新启用一个goroutine执行调用方传入的函数。无论它是否被重置以及被重置多少次都会是这样。

**断续器 time.Ticker**

定时到期之后立即进入下一个周期，周而复始，直到被停止。

1. 初始化：

var ticker \*time.Ticker=time.NewTicker(time.Second).

1. 每隔一个相对到期时间，断续器就会向它的字段C通道发送一个代表了档次的绝对到期时间的元素值。
2. 断续器只有一个方法Stop（）。

**第五章 同步**

**5.1互斥锁**

var mutex sync.Mutex

mutex.Lock()

defer mutext.Unlock()

注意事项：

1. 如果锁定一个已经锁定了的互斥锁，那么进行重复锁定的这个goroutine将会被阻塞，直到该互斥锁回到解锁状态。
2. 对一个未锁定的互斥锁进行解锁会造成恐慌。并且是不可以在defer中修复的。
3. 建议对一个互斥锁的锁定和解锁放到同一层次的代码中；如在一个函数中锁定和解锁；又如把互斥锁作为某一个结构体类型中的字段，以便在该类型的多个方法中使用它。

**5.1.2 读写锁**

读写锁可以针对读操作和写操作进行锁定和解锁

写写互斥，写读互斥，读读不互斥

var rwm sync.RwMutex

写：rwm.Lock()

rwm.Unlock()

读：rwm.RLock()

rwm.RUnlock()

注意事项：

1. 写解锁会试图唤醒所有欲进行读锁定而被阻塞的goroutine，而读解锁只会在已无任何读锁定的情况下唤醒一个因欲进行写锁定而被阻塞的goroutine。
2. 未进行读写锁定就解锁依然会造成恐慌。
3. 在锁定之后应该立即用defer解锁。

**5.2 条件变量**

（1）Go标准库中的sync.Cond代表了条件变量，条件变量总要与互斥量组合使用。

（2）结构体指针类型\*sync.Cond的方法集合中有三个方法：Wait、Signal、Broadcat

（3）Go提供的互斥锁、读写锁和条件变量都基本遵循了POSIX标准中描述的对应同步工具的行为规范。他们简单且高效，可以用来为复杂的类型提供并发安全的保证。在一些情况下，他们比通道更加适用。在只需对一个或多个临界区区进行保护的时候，使用锁往往会使程序的性能损耗更小。

**5.3 原子操作**

在针对某个值的原子操作执行过程当中，CPU绝不会再去执行其他针对该值的操作。

## Go语言提供的原子操作都是非侵入式的。它们由标准库库代码包sync/atomic中的众多函数代表，我们可以通过调用这些函数对几种简单类型的值进行原子操作。这些类型包括六种：int32、int64、uint32、uint64、uintptr、和unsafe.Pointer。这些函数提供的原子操作共有五种：

## 增或减

## 比较并交换

## 载入

## 存储

## 交换

**5.3.1增或减**

newi32 := atomic.AddInt32(&i32,3) //增大3（如果减小，则第二个参数是负数，如-3），第一个参数必须是指针类型，i32同样已经被修改了，对无法被取址的数值是无法进行原子操作的。

atomic.Uint64等其他4种也可以，但没有atomic.AddPointer（）

**5.3.2 比较并交换CAS**

如果想并发安全的更新前面这六类型，我们应该总是优先选择CAS操作：

func CopareAndSwapInt32(addr \*int32, old, new int32)(swapped bool)

//先比较\*addr的值和old值是否相等，是的话再和new交换

**5.3.3 载入（原子地读取某个值）**

v:=atomic.LoadInt32(&value)

原子的取value的值并赋给v

**5.3.4 存储（写入）**

atomic.StoreInt32(\*int32，int32)//第一个参数是存储的地址，第 二个是需要存入的新值

**5.3.5 交换**

atomic.WapInt32(\*int32，int32)//把第二个参数的值换入第一个参 数地址内，并返回第一个参数原来的值

**5.3.6 原子值**

sync/atomic是一个结构体类型，用于存储需要原子读写的值，该类型有两个指针方法Load和Store

var countVal atomic.Value

countVal.Store([]int{1,3,5,7})

countVal.Load( )

**5.5 WaitGroup**

var wg sync.WaitGroup //结构体类型

wg.Add(3)或wg.Add(-3)

wg.Done( )

wg.Wait( )

**5.6 临时对象池**

结构体sync.Pool 有两个指针方法：Get和Put

Get：从池中获取一个interface{}类型的值，返回的不一定是在池中的值

Put：把一个interface{}类型的值放置于池中

特性：

1. 临时对象池可以把由其中的对象值产生的存储压力进行分摊。
2. 对垃圾回收友好
3. 临时对象值的实例不应该被复制。

# 代码知识：

P14.从键盘获得输入

inputReader:=bufio.NewReader(os.Stdin)//得到一个带缓存的读取器

Input,err:=inputReader.ReadString(‘\n’)//开始获取输入，以回车结尾，输 入结果是字符串

Input=input[:len(input)-1]//去掉输入的回车符‘\n’

P41.将字符串中的字符全部转换成小写字母

Input=strings.Tolower(input)

Input=strings.ToUpper(input)

P49.获取进程的PID和PPID

Pid:=os.Getpid()

Ppid:=os.Getppid()

p73发送信号

proc,err:=os.FindProcess(ID) //通过进程ID找到该进程

err=proc.Signal(syscall.SIGINT) //向该进程发送信号syscall.SIGINT

p77

listener,err := net.Listen(“协议”,”host:port”) //得到一个监听器

conn,err := listener.Accept() //调用监听器的方法，流程会阻塞于此

//监听成功时，conn就是得到的连接net.Conn接口类型

net.Dial()用于向指定的网络地址发送建立连接申请

conn,err := net.Dial(“协议”,”host:port”)

可以为Dial函数设定超时时间：

conn,err := net.DialTimeout(“协议”,”host:port”,2\*time.Second)

**P84**

reader := bufio.NewReader(conn) //可以用buf方法包装链接

line,err := reader.ReadBytes(‘\n’)//包装后可使用bufi中的方法缓冲读取

writer := bufio.NewWrite(conn) // 类似

P86

conn.LocalAdrr().Network() / conn.LocalAdd().String()

conn.Remote()类似

P87设定io操作超时时间：

b := make([]byte,10)

for {

conn.SetDeadline(time.Now().Add(2\*time.Second))

n,err := conn.Read(b)

...

}