1.不是所有的值都有地址，但是所有的变量都有。

2.计算机底层全是位，而实际操作则是基于大小固定的单元中的数值，称为字（word），这些值可以解释为整数、浮点数、位集（bitset）或内存地址等。

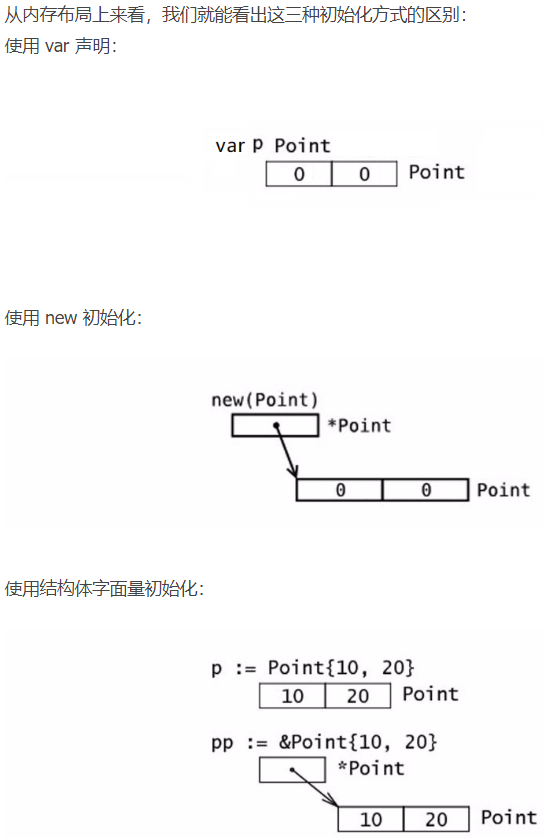
内存布局（不断补充）

Binary of 0x80000000 is 1000 0000 0000 0000 0000 0000 0000 0000

3.Go语言的宕机机制让延迟执行的函数在栈清理之前调用。

4.Go 三种初始化方式：通过var声明结构体，使用new和使用字面量的区别。

注意：表达式new(Type)和&Type{}是等价的



5.Go Pointer使用方式

Pointer表示指向任意类型的一个指针。

//任意类型的指针值可以转换成一个Pointer

//一个Pointer可以转换成一个任意类型的指针值

//一个uintptr可以转换成一个Pointer

//一个Pointer可以转换成一个uintptr

Pointer因此可以让程序击败类型系统并且读写任意内存（小心使用）

(1) 将\*T1转换成一个指向\*T2的Pointer

假设T2不比T1大，并且它们内存布局相同，这种转换允许将一种类型的数据重新解释为另一种类型的数据，例子：math.Float64bits

Func Float64bits(f float64) uint64{

Return \*(\*uint64)(unsafe.Pointer(&f))

}

(2)将一个Pointer转换成一个uintptr

将一个Pointer转换成一个uintptr将生成指针所指向值的内存地址（以整数形式表示），这种uintptr的通常使用便是打印地址。

Pointer转换成uintptr创造了一个没有指针语义的整数值，反之通常不会将uintptr转换回Pointer。接下来的模式将枚举仅仅合理的从uintptr到Pointer的转换。

(3)将Pointer转换成一个uintptr并且算术运算后再返回成一个Pointer

P=unsafe.Pointer(uintptr(p)+offset)

通常用法是使用这种模式获取struct的fields或者获取array的elements，

f:=unsafe.Pointer(uintptr(unsafe.Pointer(&s))+unsafe.Offsetof(s.f))

e:=unsafe.Pointer(uintptr(unsafe.Pointer(&x[0]))+i\*unsafe.Sizeof(x[0]))//等价于e:=unsafe.Pointer(&x[i])

不像C语言，让指针超出初始分配边界的做法是不可取的:

//INVALID

Var s thing

End=unsafe.Pointer(uintptr(unsafe.Pointer(&s))+unsafe.Sizeof(s))

//INVALID

B:=make([]byte,n)

End=unsafe.Pointer(uintptr(unsafe.Pointer(&b[0]))+uintptr(n))

注意：所有转换表达式必须在一个表达式内完成

(4) conversion of a Pointer to a uintptr when calling syscall.Syscall

Syscall.Syscall(SYS\_READ, uintptr(fd), uintptr(unsafe.Pointer(p)), uintptr(n))

(5) conversion of the result of reflect.Value.Pointer or reflect.Value.UnsafeAddr from uintptr to Pointer.

P:=(\*int)(unsafe.Pointer(reflect.ValueOf(new(int)).Pointer()))

(6) conversion of a reflect.SliceHeader or reflect.StringHeader Data field to or from Pointer

//VALID

Var s string

Hdr:=(\*reflect.StringHeader)(unsafe.Pointer(&s))

Hdr.Data=uintptr(unsafe.Pointer(p))

Hdr.Len=n

//INVALID

Var hdr reflect.StringHeader

Hdr.Data=uintptr(unsafe.Pointer(p))

Hdr.Len=n

S:=\*(\*string)(unsafe.Pointer(&hdr))

6. Semantic Versioning 2.0.2

<https://semver.org/>

Given a version number MAJOR.MINOR.PATCH, increment the:

(1). MAJOR version when you make increment incompatible API changes,

(2). MINOR version when you add functionality in a backwards compatible manner, and

(3). PATCH version when you make backwards compatible bug fixes.

Additional lables for pre-release and build metadata are available as extension to the MAJOR.MINOR.PATCH format.

7. Goroutine

8. 数组指针和指针数组

Var arr =[…]int{5:2}

X,y:=1,2

数组指针：var pf \*[4]int=&arr

指针数组：pfarr:=[…]\*int{&x,&y}

9.

If you're looking for a practical solution, the answer is simple and annoying. reflect.TypeOf takes an empty interface type into which you put whatever data you want to pass. The problem with this is that an interface type can't hold another interface type, which means that you effectively can't pass an interface to reflect.TypeOf. There is a workaround, but it's a bit painful. What you have to do is make a composite type (like a struct or slice or map) where one of the element types is an interface type, and extract it. For example:

var sliceOfEmptyInterface []interface{}

var emptyInterfaceType = reflect.TypeOf(sliceOfEmptyInterface).Elem()

10.

var v interface{}

v = (\*string)(nil)

// Reflect says it is nil

val := reflect.ValueOf(v)

if val.IsNil() {

fmt.Println("val is nil")

} else {

fmt.Println("val is not nil")

}

// This says it is not nil

if v == nil {

fmt.Println("v is nil")

} else {

fmt.Println("v is not nil")

}

11.Mutext (mutual exclusion lock)互斥锁

Mutext fairness.

2种mode：normal and starving

Normal：waiters (等待线程)在FIFO队列中等待，队列头部waiter不能得到锁，需要和新到来的goroutine（非常多）竞争锁的拥有权，如果头部waiter失败则保持在队列头部，如果超过1ms（1e6）没有得到锁，则转成starving mode。

Starving mode: 只有队列头部waiter可以得到释放的锁，其他新到来的goroutine排到队列末尾，如果一个waiter得到了锁并且发现（1）队列中只有它（2）它等待小于1ms，mutext转成normal mode.

12.OSI七层模型

OSI定义了网络互连的七层框架（物理层、数据链路层、网络层、传输层、会话层、表示层、应用层），即ISO开放互连系统参考模型。

每一层实现各自的功能和协议，并完成与相邻层的接口通信。OSI的服务定义详细说明了各层所提供的服务。某一层的服务就是该层及其下各层的一种能力，它通过接口提供给更高一层。各层所提供的服务与这些服务是怎么实现的无关。



<1>    应用层

 OSI参考模型中最靠近用户的一层，是为计算机用户提供应用接口，也为用户直接提供各种网络服务。我们常见应用层的网络服务协议有：HTTP，HTTPS，FTP，POP3、SMTP等。

<2>    表示层

表示层提供各种用于应用层数据的编码和转换功能,确保一个系统的应用层发送的数据能被另一个系统的应用层识别。如果必要，该层可提供一种标准表示形式，用于将计算机内部的多种数据[格式转换成通信](https://links.jianshu.com/go?to=http%3A%2F%2Fbaike.baidu.com%2Fview%2F1026250.htm)[中采用的标准表示形式。数据压缩和加密也是表示层可提供的转换功能之一。](https://links.jianshu.com/go?to=http%3A%2F%2Fbaike.baidu.com%2Fview%2F538641.htm)

<3>    会话层

会话层就是负责建立、管理和终止表示层实体之间的通信会话。该层的通信由不同设备中的应用程序之间的服务请求和响应组成。

<4>   传输层

传输层建立了主机端到端的链接，传输层的作用是为上层协议提供端到端的可靠和透明的数据传输服务，包括处理差错控制和流量控制等问题。该层向高层屏蔽了下层数据通信的细节，使高层用户看到的只是在两个传输实体间的一条主机到主机的、可由用户控制和设定的、可靠的数据通路[。我们通常说的，TCP UDP就是在这一层。端口号既是这里的“端”。](https://links.jianshu.com/go?to=http%3A%2F%2Fbaike.baidu.com%2Fview%2F1866929.htm)

<5>   网络层

本层通过IP寻址来建立两个节点之间的连接，为源端的运输层送来的分组，选择合适的路由和交换节点，正确无误地按照地址传送给目的端的运输层。就是通常说的IP层。这一层就是我们经常说的IP协议层。IP协议是Internet的基础。

<6>   数据链路层

 将比特组合成字节,再将字节组合成帧,使用链路层地址 (以太网使用MAC地址)来访问介质,并进行差错检测。

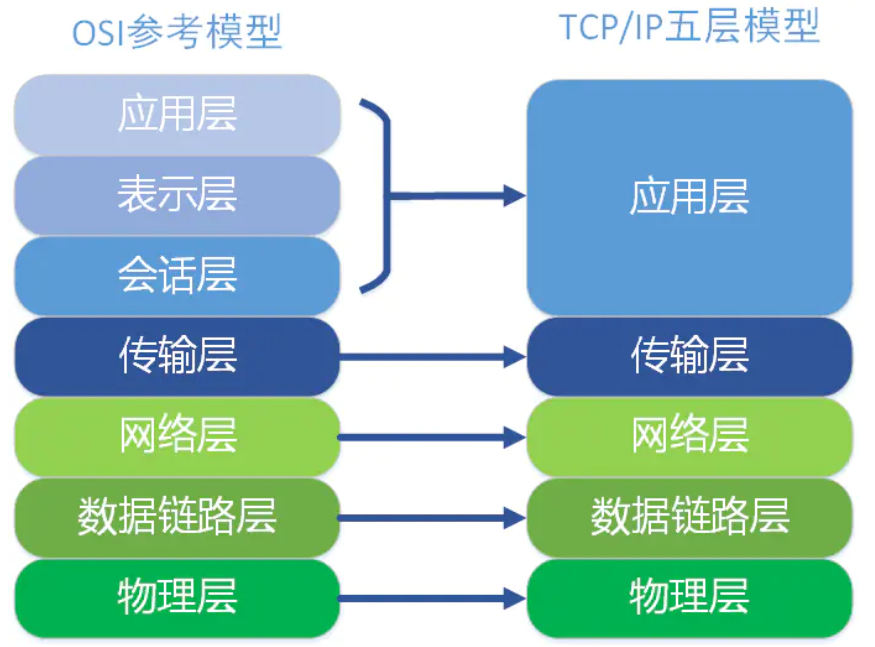
     数据链路层又分为2个子层：逻辑链路控制子层（LLC）和媒体访问控制子层（MAC）。

    MAC子层处理CSMA/CD算法、数据出错校验、成帧等；LLC子层定义了一些字段使上次协议能共享数据链路层。 在实际使用中，LLC子层并非必需的。

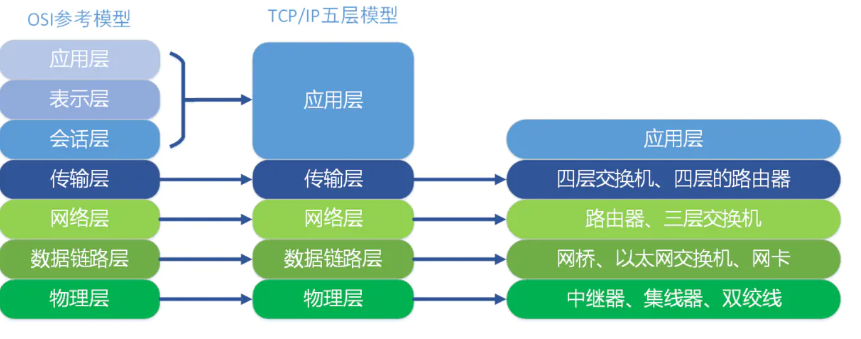
<7>  物理层

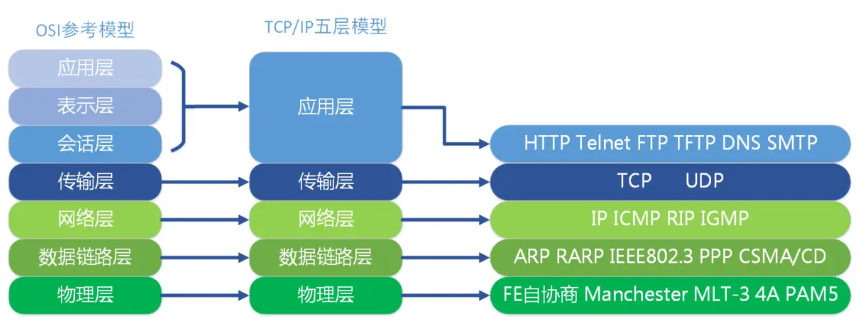
  实际最终信号的传输是通过物理层实现的。通过物理介质传输比特流。规定了电平、速度和电缆针脚。常用设备有（各种物理设备）集线器、中继器、调制解调器、网线、双绞线、同轴电缆。这些都是物理层的传输介质。

TCP/IP五层模型



 在每一层都工作着不同的设备，比如我们常用的交换机就工作在数据链路层的，一般的路由器是工作在网络层的。





13.spurious wakenups-consumer producer model

死锁：两个或以上线程执行过程中，由于竞争资源或彼此通信而造成的一种阻塞现象，若无外力作用，它们都无法推进下去。

Notify/nitify all：将随机一个或全部等待池中的线程加入entry set 中抢锁（如果抢到锁，则从wait之后的代码执行！）

两种池（队列）：entry set (锁池)，waiting set (等待池)

虚假唤醒(if（condition）{wait})：如果某个消费者消费完后notify，但notify到一个消费者线程，那么消费buffer为空，但用if（condition）{wait}线程不会再次判断而进入waiting set中，所有对空buffer取值报错，故而需要用while(condition){wait}进行反复判断。

14.IPC (interProcess Communication)# todo

指在不同进程之间传播或交换信息。

15.Message Queue VS Pipes

Message Queues are:

* UNIDIRECTIONAL
* Fixed number of entries
* Each entry has a maximum size
* All the queue memory (# entries \* entry size) allocated at creation
* Datagram-like behavior: reading an entry removes it from the queue. If you don't read the entire data, the rest is lost. For example: send a 20 byte message, but the receiver reads 10 bytes. The remaining 10 bytes are lost.
* Task can only pend on a single queue using msqQReceive (there are ways to change that with alternative API)
* When sending, you will pend if the queue is full (and you don't do NO\_WAIT)
* When receiving, you will pend if the queue is empty (and you don't do NO\_WAIT)
* Timeouts are supported on receive and send

Pipes

* Are a layer over message Queues <--- Unidirectional!
* Have a maximum number of elements and each element has maximum size
* is NOT A STREAMING INTERFACE. Datagram semantics, just list message Queues
* On read, WILL PEND until there is data to read
* On write, WILL PEND until there is space in the underlying message queue
* Can use select facility to wait on multiple pipes

That's what I can think of right now.

16.Streams vs Datagrams

A stream socket is like a phone call -- one side places the call, the other answers, you say hello to each other (SYN/ACK in TCP), and then you exchange information. Once you are done, you say goodbye (FIN/ACK in TCP). If one side doesn't hear a goodbye, they will usually call the other back since this is an unexpected event; usually the client will reconnect to the server. There is a guarantee that data will not arrive in a different order than you sent it, and there is a reasonable guarantee that data will not be damaged.

A datagram socket is like passing a note in class. Consider the case where you are not directly next to the person you are passing the note to; the note will travel from person to person. It may not reach its destination, and it may be modified by the time it gets there. If you pass two notes to the same person, they may arrive in an order you didn't intend, since the route the notes take through the classroom may not be the same, one person might not pass a note as fast as another, etc.