# 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。

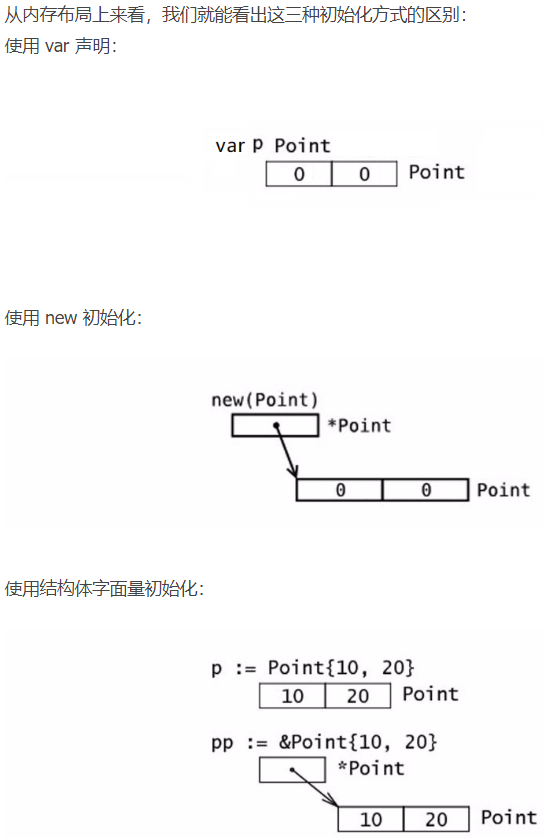
内存布局（不断补充）

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。

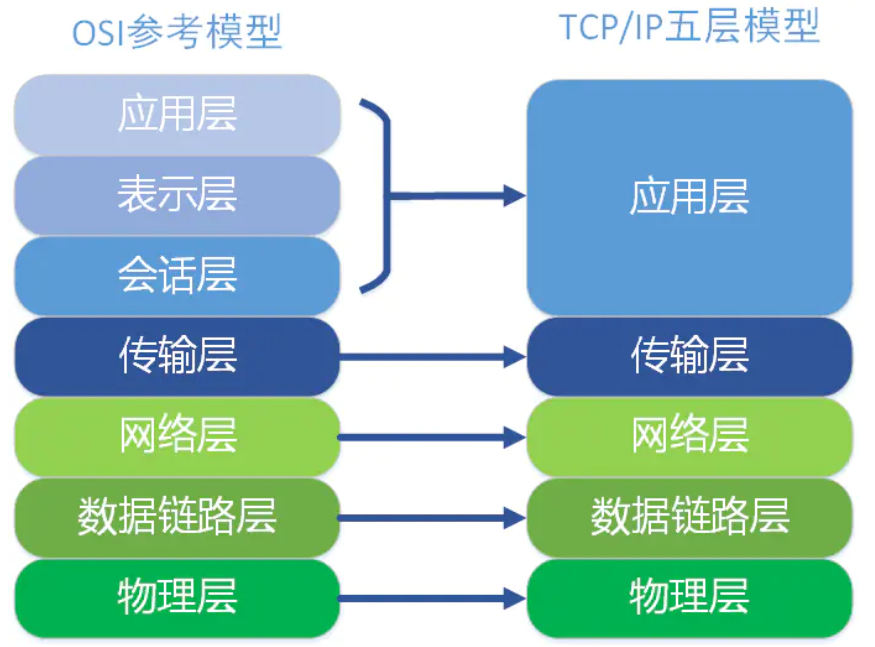
     数据链路层又分为2个子层：逻辑链路控制子层（LLC）和媒体访问控制子层（MAC）。

    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。

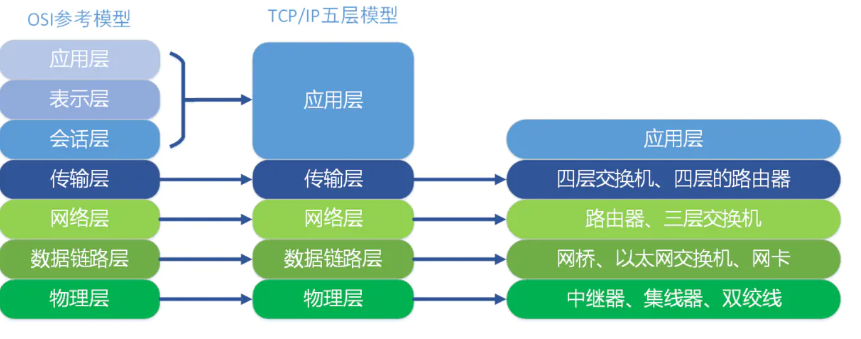
<7>  物理层

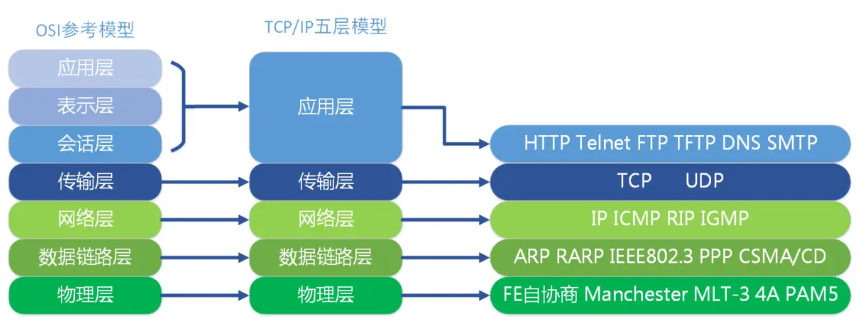
  实际最终信号的传输是通过物理层实现的。通过物理介质传输比特流。规定了电平、速度和电缆针脚。常用设备有（各种物理设备）集线器、中继器、调制解调器、网线、双绞线、同轴电缆。这些都是物理层的传输介质。

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

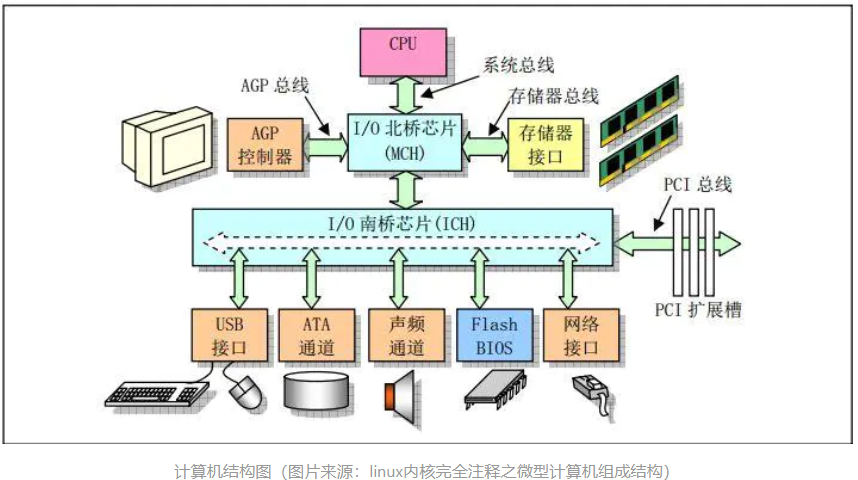
A stream socket is like a phone call -- one side places the call, the other answers, you say hello to each other (SYN/ACK in TCP), and then you exchange information. Once you are done, you say goodbye (FIN/ACK in TCP). If one side doesn't hear a goodbye, they will usually call the other back since this 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.

# 17.nature of epoll

**一、从网卡接收数据说起**

下图是一个典型的计算机结构图，计算机由CPU、存储器（内存）、网络接口等部件组成。了解epoll本质的**第一步**，要从**硬件**的角度看计算机怎样接收网络数据。



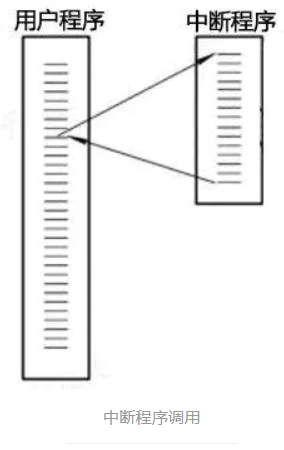
下图展示了网卡接收数据的过程。在①阶段，网卡收到网线传来的数据；经过②阶段的硬件电路的传输；最终将数据写入到内存中的某个地址上（③阶段）。这个过程涉及到DMA传输、IO通路选择等硬件有关的知识，但我们只需知道：**网卡会把接收到的数据写入内存(由内核从内核缓冲区写入内存)。**

**二、如何知道接收了数据？**

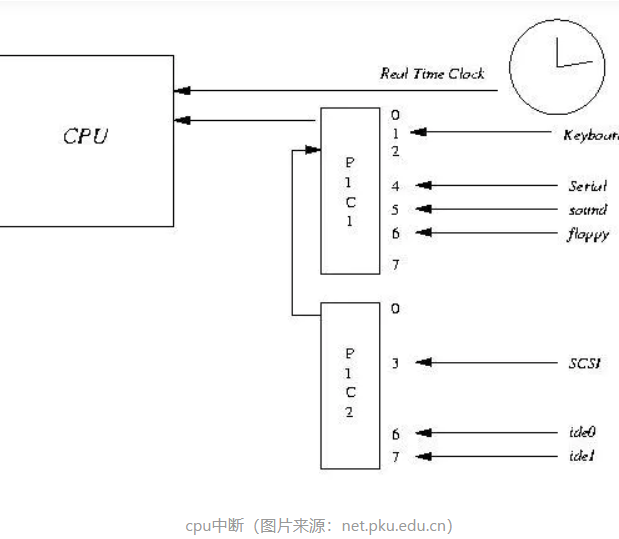
了解epoll本质的**第二步**，要从**CPU**的角度来看数据接收。要理解这个问题，要先了解一个概念——中断。

计算机执行程序时，会有优先级的需求。比如，当计算机收到断电信号时（电容可以保存少许电量，供CPU运行很短的一小段时间），它应立即去保存数据，保存数据的程序具有较高的优先级。

一般而言，由硬件产生的信号需要cpu立马做出回应（不然数据可能就丢失），所以它的优先级很高。cpu理应中断掉正在执行的程序，去做出响应；当cpu完成对硬件的响应后，再重新执行用户程序。中断的过程如下图，和函数调用差不多。只不过函数调用是事先定好位置，而中断的位置由“信号”决定。



以键盘为例，当用户按下键盘某个按键时，键盘会给cpu的中断引脚发出一个高电平。cpu能够捕获这个信号，然后执行键盘中断程序。下图展示了各种硬件通过中断与cpu交互。



现在可以回答本节提出的问题了：当网卡把数据写入到内存后，**网卡向cpu发出一个中断信号，操作系统便能得知有新数据到来**，再通过网卡**中断程序**去处理数据。

**三、进程阻塞为什么不占用cpu资源？**

了解epoll本质的**第三步**，要从**操作系统进程调度**的角度来看数据接收。阻塞是进程调度的关键一环，指的是进程在等待某事件（如接收到网络数据）发生之前的等待状态，recv、select和epoll都是阻塞方法。**了解“进程阻塞为什么不占用cpu资源？”，也就能够了解这一步**。

为简单起见，我们从普通的recv接收开始分析，先看看下面代码：

//create socket

Int s=socket(AF\_INET,SOCK\_STREAM,0);

//bind

Bind(s,…0)

//listen

Listen(s,…)

//recv client conn

Int c=accept(s,…)

//recv client data

Recv(c,…)

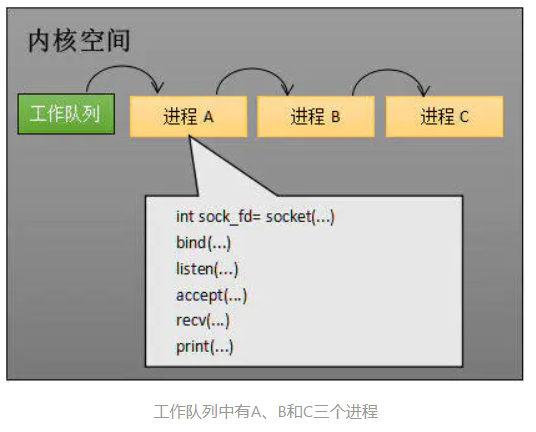
这是一段最基础的网络编程代码，先新建socket对象，依次调用bind、listen、accept，最后调用recv接收数据。recv是个阻塞方法，当程序运行到recv时，它会一直等待，直到接收到数据才往下执行。

那么阻塞的原理是什么？

**工作队列**

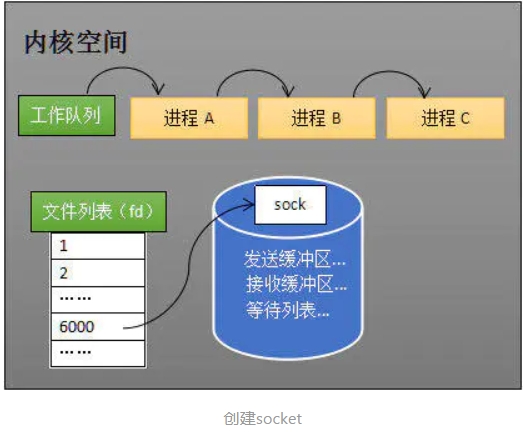
操作系统为了支持多任务，实现了进程调度的功能，会把进程分为“运行”和“等待”等几种状态。运行状态是进程获得cpu使用权，正在执行代码的状态；等待状态是阻塞状态，比如上述程序运行到recv时，程序会从运行状态变为等待状态，接收到数据后又变回运行状态。操作系统会分时执行各个运行状态的进程，由于速度很快，看上去就像是同时执行多个任务。

下图中的计算机中运行着A、B、C三个进程，其中进程A执行着上述基础网络程序，一开始，这3个进程都被操作系统的工作队列所引用，处于运行状态，会分时执行。

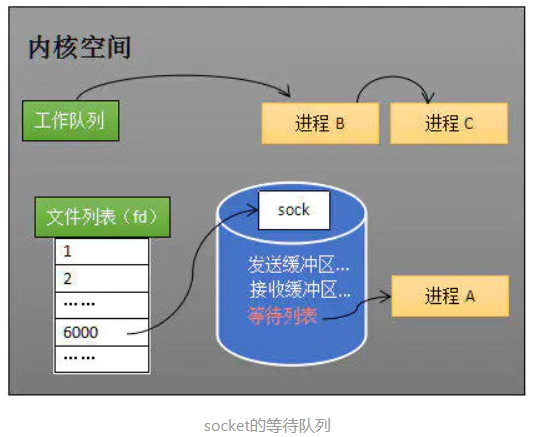


**等待队列**

当进程A执行到创建socket的语句时，操作系统会创建一个由文件系统管理的socket对象（如下图）。这个socket对象包含了发送缓冲区、接收缓冲区、等待队列等成员。等待队列是个非常重要的结构，它指向所有需要等待该socket事件的进程。



当程序执行到recv时，操作系统会将进程A从工作队列移动到该socket的等待队列中（如下图）。由于工作队列只剩下了进程B和C，依据进程调度，cpu会轮流执行这两个进程的程序，不会执行进程A的程序。**所以进程A被阻塞，不会往下执行代码，也不会占用cpu资源**。



ps：操作系统添加等待队列只是添加了对这个“等待中”进程的引用，以便在接收到数据时获取进程对象、将其唤醒，而非直接将进程管理纳入自己之下。上图为了方便说明，直接将进程挂到等待队列之下。

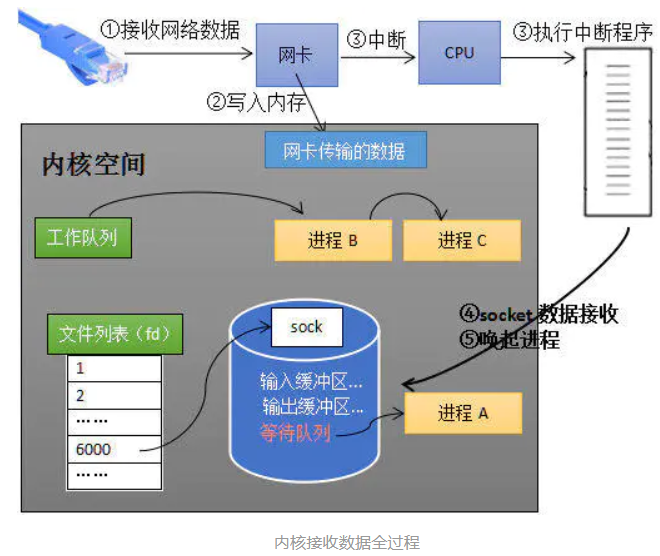
**唤醒进程**

当socket接收到数据后，操作系统将该socket等待队列上的进程重新放回到工作队列，该进程变成运行状态，继续执行代码。也由于socket的接收缓冲区已经有了数据，recv可以返回接收到的数据。

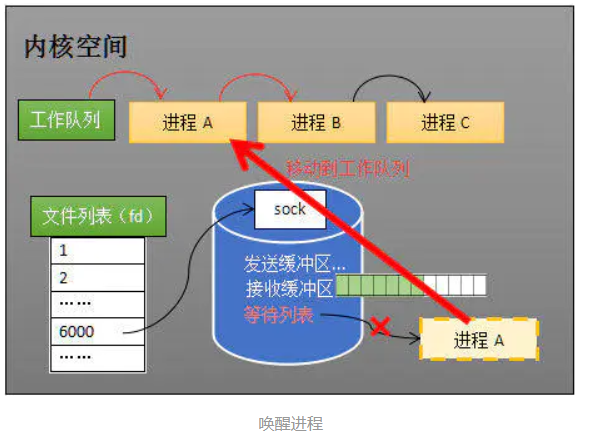
**四、内核接收网络数据全过程**

**这一步，贯穿网卡、中断、进程调度的知识，叙述阻塞recv下，内核接收数据全过程。**

如下图所示，进程在recv阻塞期间，计算机收到了对端传送的数据（步骤①）。数据经由网卡传送到内存（步骤②），然后网卡通过中断信号通知cpu有数据到达，cpu执行中断程序（步骤③）。此处的中断程序主要有两项功能，先将网络数据写入到对应socket的接收缓冲区里面（步骤④），再唤醒进程A（步骤⑤），重新将进程A放入工作队列中。



唤醒进程的过程如下图所示。



**以上是内核接收数据全过程**

这里留有两个思考题，大家先想一想。

其一，操作系统如何知道网络数据对应于哪个socket？

其二，如何同时监视多个socket的数据？

（——我是分割线，想好了才能往下看哦~）

公布答案的时刻到了。

第一个问题：因为一个socket对应着一个端口号，而网络数据包中包含了ip和端口的信息，内核可以通过端口号找到对应的socket。当然，为了提高处理速度，操作系统会维护端口号到socket的索引结构，以快速读取。

第二个问题是**多路复用的重中之重，**是本文后半部分的重点！

**五、同时监视多个socket的简单方法**

服务端需要管理多个客户端连接，而recv只能监视单个socket，这种矛盾下，人们开始寻找监视多个socket的方法。epoll的要义是**高效**的监视多个socket。从历史发展角度看，必然先出现一种不太高效的方法，人们再加以改进。只有先理解了不太高效的方法，才能够理解epoll的本质。

假如能够预先传入一个socket列表，**如果列表中的socket都没有数据，挂起进程，直到有一个socket收到数据，唤醒进程**。这种方法很直接，也是select的设计思想。

为方便理解，我们先复习select的用法。在如下的代码中，先准备一个数组（下面代码中的fds），让fds存放着所有需要监视的socket。然后调用select，如果fds中的所有socket都没有数据，select会阻塞，直到有一个socket接收到数据，select返回，唤醒进程。用户可以遍历fds，通过FD\_ISSET判断具体哪个socket收到数据，然后做出处理。

Int s=socket(AF\_INET,SOCK\_STREAM,0);

Bind(s,…)

Listen(s,…)

Int fds[]=put in sockets listening

While(1){

Int n=select(…,fds,…)

For(int i=0;i<fds.count;i++){

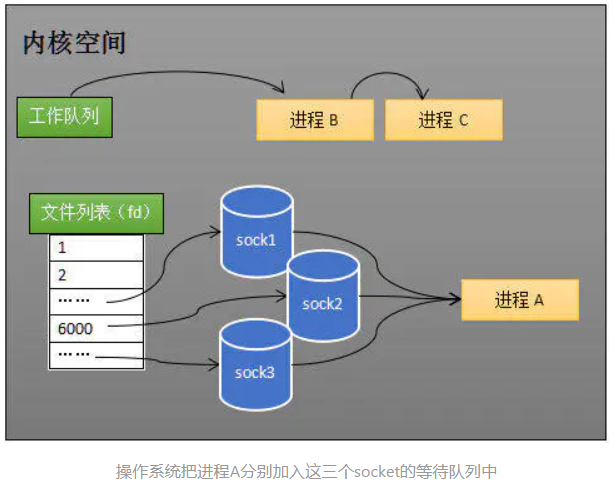
If(FD\_ISSET(fds[i],…)){//fds[i] recv data process}

}

}

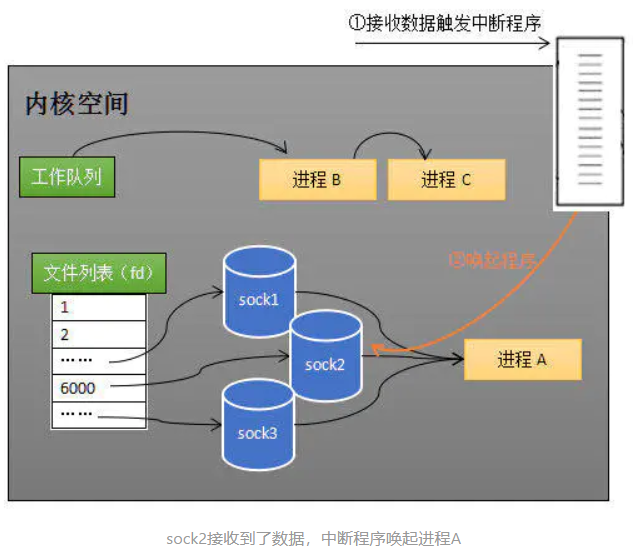
**select的流程**

select的实现思路很直接。假如程序同时监视如下图的sock1、sock2和sock3三个socket，那么在调用select之后，操作系统把进程A分别加入这三个socket的等待队列中。

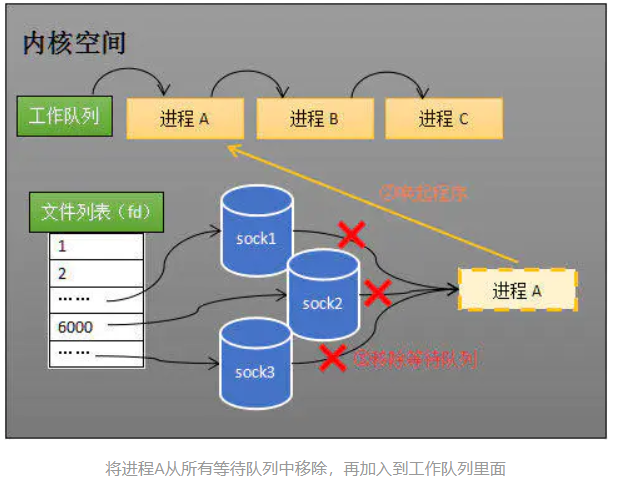


当任何一个socket收到数据后，中断程序将唤起进程。下图展示了sock2接收到了数据的处理流程。

ps：recv和select的中断回调可以设置成不同的内容。



所谓唤起进程，就是将进程从所有的等待队列中移除，加入到工作队列里面。如下图所示。



经由这些步骤，当进程A被唤醒后，它知道至少有一个socket接收了数据(唤醒后的进程不知道对应哪个socket来了数据，故需要挨个遍历查找)。程序只需遍历一遍socket列表，就可以得到就绪的socket。

这种简单方式**行之有效**，在几乎所有操作系统都有对应的实现。

**但是简单的方法往往有缺点，主要是：**

其一，每次调用select都需要将进程加入到所有监视socket的等待队列，每次唤醒都需要从每个队列中移除。这里涉及了两次遍历，而且每次都要将整个fds列表传递给内核，有一定的开销。正是因为遍历操作开销大，出于效率的考量，才会规定select的最大监视数量，默认只能监视1024个socket。

其二，进程被唤醒后，程序并不知道哪些socket收到数据，还需要遍历一次。

那么，有没有减少遍历的方法？有没有保存就绪socket的方法？这两个问题便是epoll技术要解决的。

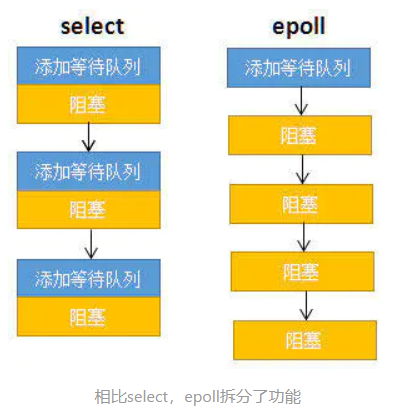
补充说明： 本节只解释了select的一种情形。当程序调用select时，内核会先遍历一遍socket，如果有一个以上的socket接收缓冲区有数据，那么select直接返回，不会阻塞。这也是为什么select的返回值有可能大于1的原因之一。如果没有socket有数据，进程才会阻塞。

**六、epoll的设计思路**

epoll是在select出现N多年后才被发明的，是select和poll的增强版本。epoll通过以下一些措施来改进效率。

**措施一：功能分离**

select低效的原因之一是将“维护等待队列”和“阻塞进程”两个步骤合二为一。如下图所示，每次调用select都需要这两步操作，然而大多数应用场景中，需要监视的socket相对固定，并不需要每次都修改。epoll将这两个操作分开，先用epoll\_ctl维护等待队列，再调用epoll\_wait阻塞进程。显而易见的，效率就能得到提升。



为方便理解后续的内容，我们先复习下epoll的用法。如下的代码中，先用epoll\_create创建一个epoll对象epfd，再通过epoll\_ctl将需要监视的socket添加到epfd中，最后调用epoll\_wait等待数据。

int s = socket(AF\_INET, SOCK\_STREAM, 0);

bind(s, ...)

listen(s, ...)

int epfd = epoll\_create(...);

epoll\_ctl(epfd, ...); //将所有需要监听的socket添加到epfd中

while(1){

    int n = epoll\_wait(...)

    for(接收到数据的socket){

        //处理

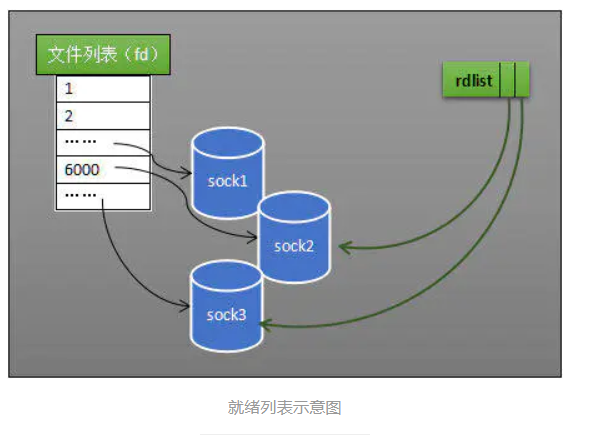
    }

}

功能分离，使得epoll有了优化的可能。

**措施二：就绪列表**

select低效的另一个原因在于程序不知道哪些socket收到数据，只能一个个遍历。如果内核维护一个“就绪列表”，引用收到数据的socket，就能避免遍历。如下图所示，计算机共有三个socket，收到数据的sock2和sock3被rdlist（就绪列表）所引用。当进程被唤醒后，只要获取rdlist的内容，就能够知道哪些socket收到数据。

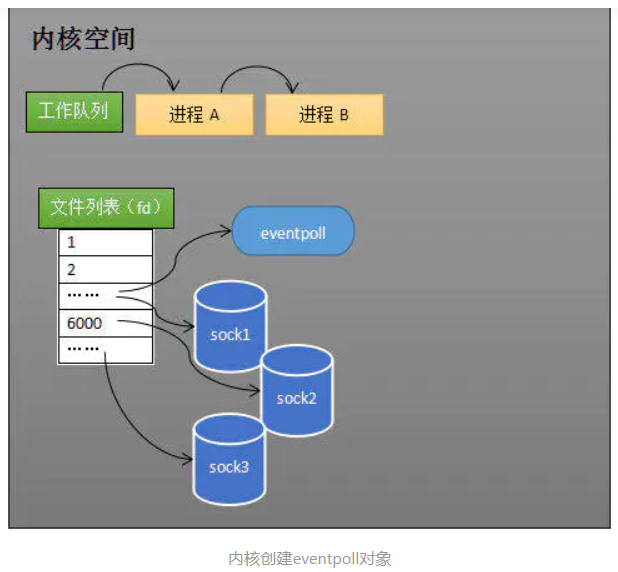


**七、epoll的原理和流程**

本节会以示例和图表来讲解epoll的原理和流程。

**创建epoll对象**

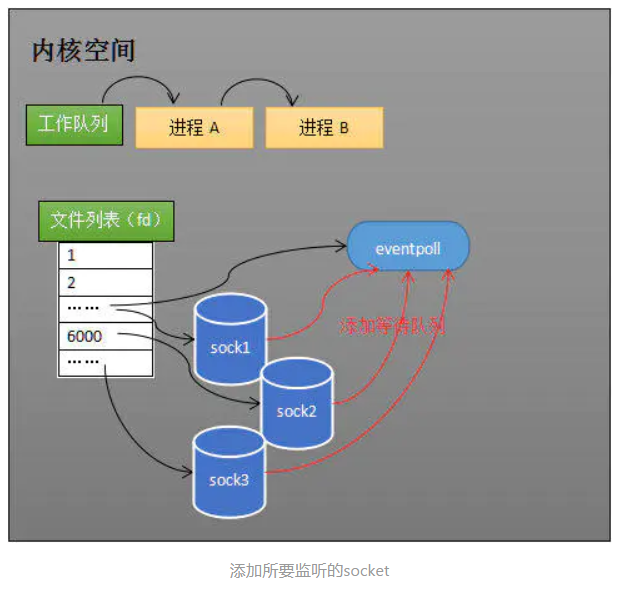
如下图所示，当某个进程调用epoll\_create方法时，内核会创建一个eventpoll对象（也就是程序中epfd所代表的对象）。eventpoll对象也是文件系统中的一员，和socket一样，它也会有等待队列。



创建一个代表该epoll的eventpoll对象是必须的，因为内核要维护“就绪列表”等数据，“就绪列表”可以作为eventpoll的成员。

**维护监视列表**

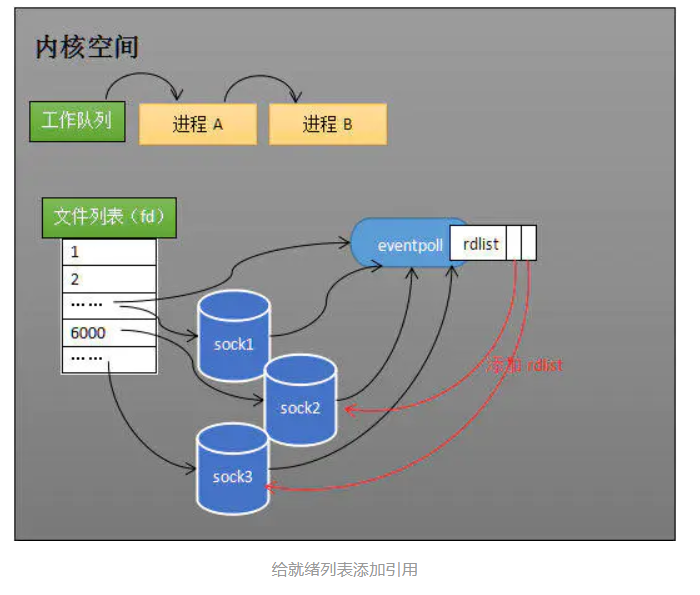
创建epoll对象后，可以用epoll\_ctl添加或删除所要监听的socket。以添加socket为例，如下图，如果通过epoll\_ctl添加sock1、sock2和sock3的监视，内核会将eventpoll添加到这三个socket的等待队列中。



当socket收到数据后，中断程序会操作eventpoll对象，而不是直接操作进程。

**接收数据**

当socket收到数据后，中断程序会给eventpoll的“就绪列表”添加socket引用。如下图展示的是sock2和sock3收到数据后，中断程序让rdlist引用这两个socket。



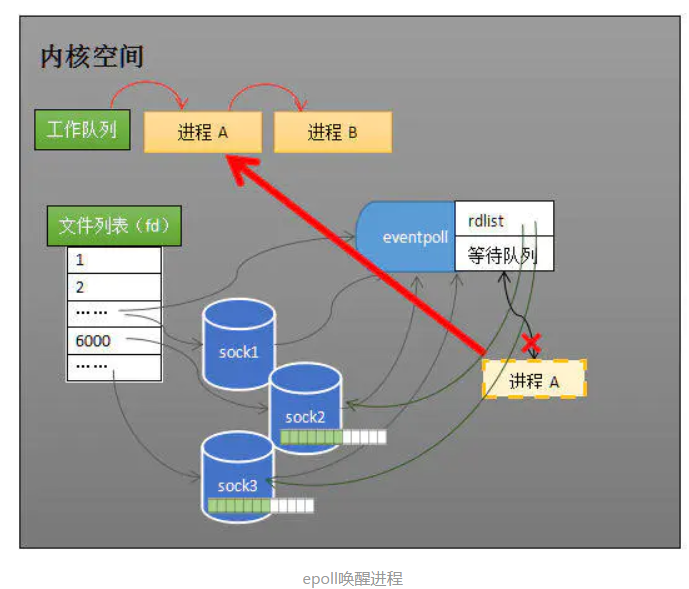
eventpoll对象相当于是socket和进程之间的中介，socket的数据接收并不直接影响进程，而是通过改变eventpoll的就绪列表来改变进程状态。

当程序执行到epoll\_wait时，如果rdlist已经引用了socket，那么epoll\_wait直接返回，如果rdlist为空，阻塞进程。

**阻塞和唤醒进程**

假设计算机中正在运行进程A和进程B，在某时刻进程A运行到了epoll\_wait语句。如下图所示，内核会将进程A放入eventpoll的等待队列中，阻塞进程。

当socket接收到数据，中断程序一方面修改rdlist，另一方面唤醒eventpoll等待队列中的进程，进程A再次进入运行状态（如下图）。也因为rdlist的存在，进程A可以知道哪些socket发生了变化。



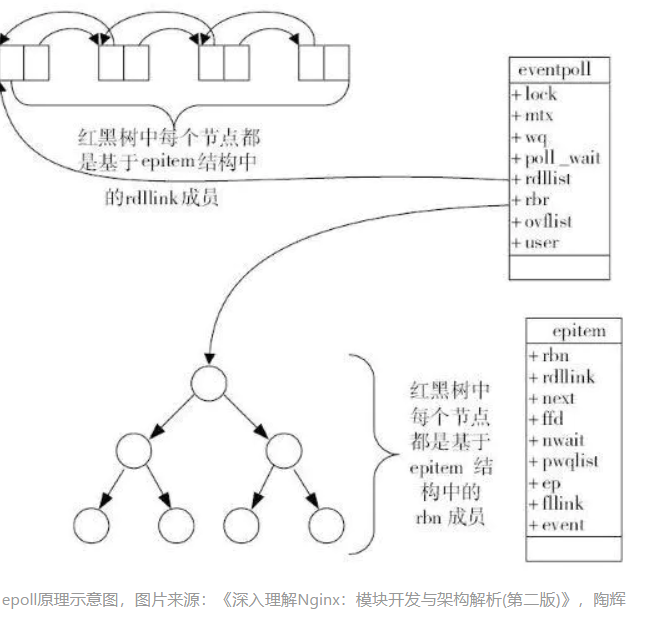
**八、epoll的实现细节**

至此，相信读者对epoll的本质已经有一定的了解。但我们还留有一个问题，**eventpoll的数据结构**是什么样子？

再留两个问题，**就绪队列**应该应使用什么数据结构？eventpoll应使用什么数据结构来管理通过epoll\_ctl添加或删除的socket？

（——我是分割线，想好了才能往下看哦~）

如下图所示，eventpoll包含了lock、mtx、wq（等待队列）、rdlist等成员。rdlist和rbr是我们所关心的。



**就绪列表的数据结构**

就绪列表引用着就绪的socket，所以它应能够快速的插入数据。

程序可能随时调用epoll\_ctl添加监视socket，也可能随时删除。当删除时，若该socket已经存放在就绪列表中，它也应该被移除。

所以就绪列表应是一种能够快速插入和删除的数据结构。双向链表就是这样一种数据结构，epoll使用双向链表来实现就绪队列（对应上图的rdllist）。

**索引结构**

既然epoll将“维护监视队列”和“进程阻塞”分离，也意味着需要有个数据结构来保存监视的socket。至少要方便的添加和移除，还要便于搜索，以避免重复添加。红黑树是一种自平衡二叉查找树，搜索、插入和删除时间复杂度都是O(log(N))，效率较好。epoll使用了红黑树作为索引结构（对应上图的rbr）。

ps：因为操作系统要兼顾多种功能，以及由更多需要保存的数据，rdlist并非直接引用socket，而是通过epitem间接引用，红黑树的节点也是epitem对象。同样，文件系统也并非直接引用着socket。为方便理解，本文中省略了一些间接结构。

# 18.a word about goroutine, stacks and splits

Stacks

Since the number of goroutines in a Go program is non-deterministic, and can go up to several millions in practice, the runtime must take the conservative route when allocating stack space for goroutines to avoid eating up all of the available memory. As such, every new goroutine is given an initial tiny **2kB stack** by the runtime (said stack is **actually allocated on the heap** behind the scenes).

As a goroutine runs along doing its job, it might end up outgrowing its contrived, initial stack-space (i.e. stack-overflow). **To prevent this from happening, the runtime makes sure that when a goroutine is running out of stack, a new, bigger stack with two times the size of the old one gets allocated, and that the content of the original stack gets copied over to the new one.** This process is known as a **stack-split** and effectively makes **goroutine stacks dynamically-sized.**