网络

## UDP

把数据打包

数据大小有限制（64k）

不建立连接

速度快，但可靠性低

为什么不直接使用IP协议而要额外增加一个UDP协议呢？ 一个重要的原因是IP协议中并没有端口(port)的概念。IP协议进行的是IP地址到IP地址的传输

UDP头：源端口，目的端口，包长度，校验码，共8字节

因为UDP套接口是无连接的，如果一方的数据报丢失，那另一方将无限等待，解决办法是设置一个超时。

UdpClient 如何结束Receive()方法的阻塞答案是自己给自己发一个消息，随便什么内容。

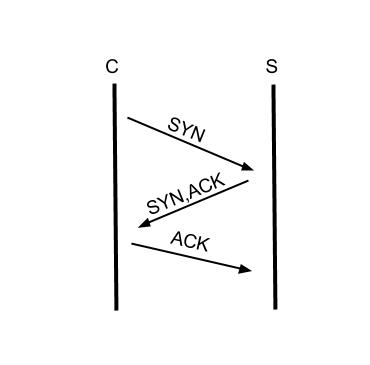
## TCP

以太网数据包（packet）的大小是固定的，最初是1518字节，后来增加到1522字节。其中， 1500 字节是负载（payload），22字节是头信息（head）。

IP 数据包在以太网数据包的负载里面，它也有自己的头信息，最少需要20字节，所以 IP 数据包的负载最多为1480字节。TCP 数据包在 IP 数据包的负载里面。它的头信息最少也需要20字节，因此 TCP 数据包的最大负载是 1480 - 20 = 1460 字节。由于 IP 和 TCP 协议往往有额外的头信息，所以 TCP 负载实际为1400字节左右。

建立连接，三次握手：

第一个报文SYN从发起方发出，第二个报文SYN,ACK是从被连接方发出，第三个报文ACK确认对方的SYN，ACK已经收到，



断开连接，四次握手

客户端发送FIN，服务端返回ACK，服务端发送FIN，客户端返回ACK，断开。

由于TCP连接时全双工的，因此，每个方向都必须要单独进行关闭，这一原则是当一方完成数据发送任务后，发送一个FIN来终止这一方向的连接，收到一个FIN只是意味着这一方向上没有数据流动了，即不会再收到数据了，但是在这个TCP连接上仍然能够发送数据，直到这一方向也发送了FIN。首先进行关闭的一方将执行主动关闭，而另一方则执行被动关闭。

为什么建立连接是三次握手，而关闭连接却是四次挥手呢？

这是因为服务端在LISTEN状态下，收到建立连接请求的SYN报文后，把ACK和SYN放在一个报文里发送给客户端。而关闭连接时，当收到对方的FIN报文时，仅仅表示对方不再发送数据了但是还能接收数据，己方也未必全部数据都发送给对方了，所以己方可以立即close，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，己方ACK和FIN一般都会分开发送。

为什么TIME\_WAIT状态需要经过2MSL(最大报文段生存时间)才能返回到CLOSE状态？

原因有二：

一、保证TCP协议的全双工连接能够可靠关闭

二、保证这次连接的重复数据段从网络中消失

先说第一点，如果Client直接CLOSED了，那么由于IP协议的不可靠性或者是其它网络原因，导致Server没有收到Client最后回复的ACK。那么Server就会在超时之后继续发送FIN，此时由于Client已经CLOSED了，就找不到与重发的FIN对应的连接，最后Server就会收到RST而不是ACK，Server就会以为是连接错误把问题报告给高层。这样的情况虽然不会造成数据丢失，但是却导致TCP协议不符合可靠连接的要求。所以，Client不是直接进入CLOSED，而是要保持TIME\_WAIT，当再次收到FIN的时候，能够保证对方收到ACK，最后正确的关闭连接。

再说第二点，如果Client直接CLOSED，然后又再向Server发起一个新连接，我们不能保证这个新连接与刚关闭的连接的端口号是不同的。也就是说有可能新连接和老连接的端口号是相同的。一般来说不会发生什么问题，但是还是有特殊情况出现：假设新连接和已经关闭的老连接端口号是一样的，如果前一次连接的某些数据仍然滞留在网络中，这些延迟数据在建立新连接之后才到达Server，由于新连接和老连接的端口号是一样的，又因为TCP协议判断不同连接的依据是socket pair，于是，TCP协议就认为那个延迟的数据是属于新连接的，这样就和真正的新连接的数据包发生混淆了。所以TCP连接还要在TIME\_WAIT状态等待2倍MSL，这样可以保证本次连接的所有数据都从网络中消失。

## HTTP

***Client头：***

Accept，作用： 浏览器端可以接受的媒体类型, 如：  Accept: text/html  代表浏览器可以接受服务器回发的类型为 text/html ，通配符 \* 代表任意类型， Accept: \*/\*  代表浏览器可以处理所有类型,(一般浏览器发给服务器都是发这个)

Accept-Encoding：浏览器申明自己接收的编码方法，通常指定压缩方法，是否支持压缩，支持什么压缩方法（gzip，deflate）

User-Agent：告诉HTTP服务器， 客户端使用的操作系统和浏览器的名称和版本.

Cookie: 最重要的header, 将cookie的值发送给HTTP 服务器

Content-Length：发送给HTTP服务器数据的长度。

Content-Type：如：Content-Type: application/x-www-form-urlencoded

Referer: 提供了Request的上下文信息的服务器，告诉服务器我是从哪个链接过来的

Connection：如：　Connection: keep-alive 当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的TCP连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的连接。如：  Connection: close  代表一个Request完成后，客户端和服务器之间用于传输HTTP数据的TCP连接会关闭， 当客户端再次发送Request，需要重新建立TCP连接。

Host（发送请求时，该报头域是必需的）请求报头域主要用于指定被请求资源的Internet主机和端口号。如: 我们在浏览器中输入：<http://www.guet.edu.cn/index.html>。浏览器发送的请求消息中，就会包含Host请求报头域，如下：Host：http://www.guet.edu.cn此处使用缺省端口号80，若指定了端口号，则变成：Host：指定端口号

***Response头：***

Expires：浏览器会在指定过期时间内使用本地缓存

P3P：用于跨域设置Cookie, 这样可以解决iframe跨域访问cookie的问题

Set-Cookie：非常重要的header, 用于把cookie 发送到客户端浏览器， 每一个写入cookie都会生成一个Set-Cookie.

ETag：和If-None-Match 配合使用。

Last-Modified: 用于指示资源的最后修改日期和时间。

Content-Type：WEB服务器告诉浏览器自己响应的对象的类型和字符集, Content-Type: text/html; charset=utf-8

Content-Length：指明实体正文的长度

Content-Encoding: WEB服务器表明自己使用了什么压缩方法（gzip，deflate）压缩响应中的对象。

Server: 指明HTTP服务器的软件信息

Location：用于重定向一个新的位置, 包含新的URL地址

***KeepAlive实现原理：***

TCP 的 SO\_KEEPALIVE 确切的应该叫做 is\_keep\_alive?，它是一个探测服务，由内核完成的。当一端 recv(socket, ...) 时，如果对端一直没有消息返回，就会一直阻塞，本端根本不知道对端处于什么状态。所以本端可以周期发一个探测包，对端如果一定时间没有返回确认包，则认为对端已经断开连接， recv 会返回，不再继续阻塞。HTTP 的 keepalive 是双方通过 "Connection: keep-alive" 数据头来建立长连接，每次发送完请求不关闭套接字，而是继续下一次循环的 recv()，等待对端继续发送数据。如果设置了超时时间，那么 recv() 返回，关闭连接。TCP 的 SO\_KEEPALIVE，当网络突然中断时，用来及时探测对端断开，避免无限制阻塞 recv。HTTP 的 keepalive，是双方约定长连接、还是收完一次数据后立刻关闭套接字。

**tcp的keepalive**是侧重在保持客户端和服务端的连接，一方会不定期发送心跳包给另一方，当一方断掉的时候，没有断掉的定时发送几次心跳包，如果间隔发送几次，对方都返回的是RST，而不是ACK，那么就释放当前链接。

建立TCP连接时，就有定时器与之绑定，其中的一些定时器就用于处理keepalive过程。当keepalive定时器到0的时候，便会给对端发送一个不包含数据部分的keepalive探测包（probe packet），如果收到了keepalive探测包的回复消息，那就可以断定连接依然是OK的。如果我们没有收到对端keepalive探测包的回复消息，我们便可以断定连接已经不可用，进而采取一些措施。

KeepAlive并不是默认开启的，在Linux系统上没有一个全局的选项去开启TCP的KeepAlive。需要开启KeepAlive的应用必须在TCP的socket中单独开启。Linux Kernel有三个选项影响到KeepAlive的行为：

1.net.ipv4.tcpkeepaliveintvl = 75

2.net.ipv4.tcpkeepaliveprobes = 9

3.net.ipv4.tcpkeepalivetime = 7200

tcpkeepalivetime的单位是秒，表示TCP链接在多少秒之后没有数据报文传输启动探测报文; tcpkeepaliveintvl单位是也秒,表示前一个探测报文和后一个探测报文之间的时间间隔，tcpkeepaliveprobes表示探测的次数。

tcp的keepalive就是为了检测链接的可用性。主要调节的参数有三个：

* tcp\_keepalive\_time // 距离上次传送数据多少时间未收到判断为开始检测
* tcp\_keepalive\_intvl // 检测开始每多少时间发送心跳包
* tcp\_keepalive\_probes // 发送几次心跳包对方未响应则close连接

在客户端和服务端进行完三次握手之后，客户端和服务端都处在ESTABLISH状态，这个时候进行正常的PSH和ACK交互，但是一旦一方服务中断了，另一方在距离上次PSH时间tcp\_keepalive\_time发现对方未发送数据，则开始心跳检测。心跳检测实际就是发送一个PSH的空心跳包，这里说的空心跳包就是包的数据为空，但是TCP包的头部的数据和标识和正常包一样。如果这个包获取到的是RST返回的话，下面就会继续每隔tcp\_keepalive\_intval的时长发送一个空心跳包，如果tcp\_keepalive\_probes次心跳包对方都是返回RST而不是ACK，则心跳发起方就判断这个连接已经失效，主动CLOST这个连接。

[tcp层的keepalive会在两个场景下比较有用](http://www.tldp.org/HOWTO/html_single/TCP-Keepalive-HOWTO/)：

## 检测连接的一方是否断了

这里说的连接的一方是否断了包含几种情况：

* 连接一方服务中止
* 网络不好导致的服务长时间无响应
* 连接一方服务重启中

结合这三种方式就很好理解为什么会有 tcp\_keepalive\_time, tcp\_keepalive\_intval, tcp\_keepalive\_probes三种的设置了。如果是对方服务器进行重启的时候，我们不能根据一次的tcp返回重置信号就判定这个连接失效。相反的，重启之后，这个心跳包一旦正常，这个连接仍然可以继续使用。

## 防止因为长时间不用链接导致连接失效

这个往往在代理或者内网状况下会使用到。一般NAT网络为了资源，会和外网保持一定的资源连接数，而且采用的是淘汰机制，淘汰掉旧的，不用的连接，创建和使用新的连接。如果我们没有心跳检测机制，那么我们的连接在一段时间没有使用的时候，NAT对外的机制会判断对应的对外网络是无用的，淘汰掉旧的，即使这个时候客户端和服务端都还正常服务着，只是长时间未联络了而已。keepalive的机制由于有定时心跳包，自然就能解决这个问题了。

**http的keep-alive**是一旦客户端发送connection:keep-alive头给服务端，且服务端也接受这个keep-alive的话，两边对上暗号，这个连接就可以复用了，一个http处理完之后，另外一个http数据直接从这个连接走了。http层有个keep-alive, 它主要是用于客户端告诉服务端，这个连接我还会继续使用，在使用完之后不要关闭。这个设置首先会在性能上对客户端和服务器端性能上有一定的提升。很好理解的是少了TCP的三次握手和四次挥手，第二次传递数据就可以通过前一个连接直接进行数据交互了。当然会提升服务性能了。由于HTTP服务的发起方一般都是浏览器，即客户端。但是先执行完逻辑，传输完数据的一定是服务端。那么一旦没有keep-alive机制，服务端在传送完数据之后会率先发起连接断开的操作。由于TCP的四次挥手机制，先发起连接断开的一方会在连接断开之后进入到TIME\_WAIT的状态达到2MSL之久。

什么是 MIME TYPE？资源的媒体类型。 HTTP 协议，通过 Content-Type 来表示的，例如:Content-Type: text/HTML

一个HTTP请求报文由请求行（request line）、请求头（header）、空行和请求数据4个部分组成。

HTTP1.0定义了三种请求方法： GET, POST 和 HEAD方法。

HTTP1.1新增了五种请求方法：OPTIONS, PUT, DELETE, TRACE 和 CONNECT 方法。

GET请求：由于不同的浏览器对地址的字符限制也有所不同，一般最多只能识别2083个字符。

POST方式对传送的数据大小没有限制（限制在web服务器端，如tomcat默认为2m）

为什么不能用post代替所有的get：

1、在使用 XMLHttpRequest 的 POST 方法时，浏览器会先发送 Header 再发送 Data。但并不是所有浏览器会这么做，例如火狐就不会。

多数浏览器对于POST采用两阶段发送数据的，先发送请求头，再发送请求体，即使参数再少再短，也会被分成两个步骤来发送（相对于GET），也就是第一步发送header数据，第二步再发送body部分。HTTP是应用层的协议，而在传输层有些情况TCP会出现两次连结的过程，HTTP协议本身不保存状态信息，一次请求一次响应。对于TCP而言，通信次数越多反而靠性越低，能在一次连结中传输完需要的消息是最可靠的，尽量使用GET请求来减少网络耗时。如果通信时间增加，这段时间客户端与服务器端一直保持连接状态，在服务器侧负载可能会增加，可靠性会下降。  
2、GET请求能够被cache，GET请求能够被保存在浏览器的浏览历史里面（密码等重要数据GET提交，别人查看历史记录，就可以直接看到这些私密数据）POST不进行缓存。  
3、GET参数是带在URL后面，传统IE中URL的最大可用长度为2048字符，其他浏览器对URL长度限制实现上有所不同。POST请求无长度限制（目前理论上是这样的）。  
4、GET提交的数据大小，不同浏览器的限制不同，一般在2k-8K之间，POST提交数据比较大，大小靠服务器的设定值限制。  
5、全部用POST不是十分合理，最好先把请求按功能和场景分下类，对数据请求频繁，数据不敏感且数据量在普通浏览器最小限定的2k范围内，这样的情况使用GET。其他地方使用POST。

## Yahoo高性能网站

使用CDN

为文件头指定Expires或Cache-Control：

对于静态内容：设置文件头过期时间Expires的值为“Never expire”（永不过期）

对于动态内容：使用恰当的Cache-Control文件头来帮助浏览器进行有条件的请求

Gzip压缩文件内容

从HTTP/1.1开始，web客户端都默认支持HTTP请求中有Accept-Encoding文件头的压缩格式：Accept-Encoding: gzip, deflate

Web服务器把压缩方式通过响应文件头中的Content-Encoding来返回给浏览器。

配置ETag

Etag是一个识别内容版本号的唯一字符串。增加ETag为实体的验证提供了一个比使用“last-modified date（上次编辑时间）”更加灵活的机制

尽早刷新输出缓冲

在PHP中，你可以使用flush()方法，它允许你把已经编译的好的部分HTML响应文件先发送给浏览器

使用GET来完成AJAX请求

当使用XMLHttpRequest时，浏览器中的POST方法是一个“两步走”的过程：首先发送文件头，然后才发送数据。因此使用GET最为恰当，因为它只需发送一个TCP包（除非你有很多cookie）。

## NAT（地址转换）

狭义上，NAT分为SNAT（原地址转换）和DNAT（目标地址转换）

NAP（p为port）T与NAT的区别在于，NAPT不仅转换IP包中的IP地址，还对IP包中TCP和UDP的Port进行转换。这使得多台私有网主机利用1个NAT公共IP就可以同时和公共网进行通信。（NAPT多了对TCP和UDP的端口号的转换）

DNAT，就是指数据包从网卡发送出去的时候，修改数据包中的目的IP，表现为如果你想访问A，可是因为网关做了DNAT，把所有访问A的数据包的目的IP全部修改为B，那么，你实际上访问的是B

ISP会把用户接入局域网，使得多个用户共享同一个公网IP，而每一个用户各分得一个局域网内网IP。而连接公网和局域网的这台路由器，称之为网关（gateway），NAT的过程就发生在这台网关路由器上。

在三层地址转换中，我们可以保证局域网内主机向公网发出的IP报文能顺利到达目的主机，但是从目的主机返回的IP报文却不能准确送至指定局域网主机（我们不能让网关把IP报文广播至全部局域网主机，因为这样必然会带来安全和性能问题）。为了解决这个问题，网关路由器需要借助传输层端口，通常情况下是TCP或UDP端口，由此来生成一张端口转换表。

让我们通过一个实例来说明端口转换表如何运作。  
假设局域网主机A192.168.1.100需要与公网上的目标主机B210.199.38.2:80进行一次TCP通信。其中A所在局域网的网关C的公网IP地址为210.177.63.2。步骤如下：

1. 局域网主机A192.168.1.100发出TCP连接请求，A上的TCP端口为系统分配的53600。该TCP握手包中，包含源地址和端口192.168.1.100:53600，目的地址和端口210.199.38.2:80。  
2. 网关C将该包的原地址和端口修改为210.177.63.2:63000，其中63000是网关分配的临时端口。  
3. 网关C在端口转换表中增加一条记录：

| **内网主机IP** | **内网主机端口** | **网关端口** | **目的主机IP** | **目的主机端口** |
| --- | --- | --- | --- | --- |
| 192.168.1.100 | 53600 | 63000 | 210.199.38.2 | 80 |

4. 网关C将修改后的TCP包发送至目的主机B。  
5. 目的主机B收到后，发送响应TCP包。该响应TCP包含有以下信息：源地址和端口210.199.38.2:80，目的地址和端口210.177.63.2:63000。  
6. 网关C收到这个来自B的响应包后，随即在端口转换表中查找记录。该记录须符合以下条件：目的主机IP==210.199.38.2，目的主机端口==80，网关端口==63000。  
7. 网关C搜索到这条记录，记录显示内网主机IP为192.168.1.100，内网主机端口为53600。  
8. 网关C将该包的目的地址和端口修改为192.168.1.100:53600。  
9. 网关C随即将该修改后的TCP包转发至192.168.1.100:53600，即局域网主机A。此时运输层数据的一次交换已完成。

***问题所在***

在网关C上，由于端口数量有限（0~65535），端口转换表的维护占用系统资源，因此不能无休止地向端口转换表中增加记录。对于过期的记录，网关需要将其删除。如何判断哪些是过期记录？网关认为，一段时间内无活动的连接是过期的，应定时检测转换表中的非活动连接，并将之丢弃。**而这个丢弃的过程，网关不会以任何的方式通告该连接的任何一端。**那么问题就来了：如果一个客户端应用程序由于业务需要，需要与服务端维持长连接（如TCP聊天程序），而如果在特别长的时间内（在博主的ISP环境下，该时间在3分钟左右），这个连接没有任何的数据交换，网关会认为这个连接过期并将这个连接从端口转换表中丢弃。该连接被丢弃时，客户端和服务端对此是完全无感知的。在连接被丢弃后，客户端将收不到服务端的数据推送，客户端发送的数据包也不能到达服务端。

**NAT打洞**

，NAT设备允许私网内主机主动向公网内主机发送数据,但却禁止反方向的主动传递，但在一些特殊的场合需要不同私网内的主机进行互联（例如P2P软件、网络会议、视频传输等），TCP穿越NAT的问题必须解决。

NAT设备的类型对于TCP穿越NAT,有着十分重要的影响,根据端口映射方式,NAT可分为如下4类,前3种NAT类型可统称为cone类型。

(1)全克隆( Full Clone) : NAT把所有来自相同内部IP地址和端口的请求映射到相同的外部IP地址和端口。任何一个外部主机均可通过该映射发送IP包到该内部主机。

(2)限制性克隆(Restricted Clone) : NAT把所有来自相同内部IP地址和端口的请求映射到相同的外部IP地址和端口。但是,只有当内部主机先给IP地址为X的外部主机发送IP包,该外部主机才能向该内部主机发送IP包。

(3)端口限制性克隆( Port Restricted Clone) :端口限制性克隆与限制性克隆类似,只是多了端口号的限制,即只有内部主机先向IP地址为X,端口号为P的外部主机发送1个IP包,该外部主机才能够把源端口号为P的IP包发送给该内部主机。

(4)对称式NAT ( Symmetric NAT) :这种类型的NAT与上述3种类型的不同,在于当同一内部主机使用相同的端口与不同地址的外部主机进行通信时, NAT对该内部主机的映射会有所不同。对称式NAT不保证所有会话中的私有地址和公开IP之间绑定的一致性。相反,它为每个新的会话分配一个新的端口号。

先假设：有一个服务器S在公网上有一个IP，两个私网分别由NAT-A和NAT-B连接到公网，NAT-A后面有一台客户端A，NAT-B后面有一台客户端B，现在，我们需要借助S将A和B建立直接的TCP连接，即由B向A打一个洞，让A可以沿这个洞直接连接到B主机，就好像NAT-B不存在一样。

实现过程如下：

1、 S启动两个网络侦听，一个叫【主连接】侦听，一个叫【协助打洞】的侦听。

2、 A和B分别与S的【主连接】保持联系。

3、 当A需要和B建立直接的TCP连接时，首先连接S的【协助打洞】端口，并发送协助连接申请。同时在该端口号上启动侦听。注意由于要在相同的网络终端上绑定到不同的套接字上，所以必须为这些套接字设置 SO\_REUSEADDR 属性（即允许重用），否则侦听会失败。

4、 S的【协助打洞】连接收到A的申请后通过【主连接】通知B，并将A经过NAT-A转换后的公网IP地址和端口等信息告诉B。

5、 B收到S的连接通知后首先与S的【协助打洞】端口连接，随便发送一些数据后立即断开，这样做的目的是让S能知道B经过NAT-B转换后的公网IP和端口号。

6、 B尝试与A的经过NAT-A转换后的公网IP地址和端口进行connect，根据不同的路由器会有不同的结果，有些路由器在这个操作就能建立连接，大多数路由器对于不请自到的SYN请求包直接丢弃而导致connect失败，但NAT-A会纪录此次连接的源地址和端口号，为接下来真正的连接做好了准备，这就是所谓的打洞，即B向A打了一个洞，下次A就能直接连接到B刚才使用的端口号了。

7、 客户端B打洞的同时在相同的端口上启动侦听。B在一切准备就绪以后通过与S的【主连接】回复消息“我已经准备好”，S在收到以后将B经过NAT-B转换后的公网IP和端口号告诉给A。

8、 A收到S回复的B的公网IP和端口号等信息以后，开始连接到B公网IP和端口号，由于在步骤6中B曾经尝试连接过A的公网IP地址和端口，NAT-A纪录了此次连接的信息，所以当A主动连接B时，NAT-B会认为是合法的SYN数据，并允许通过，从而直接的TCP连接建立起来了。

## UDP打洞

假设现在有内网客户端A和内网客户端B，有公网服务端S。

    如果A和B想要进行UDP通信，则必须穿透双方的NAT路由。假设为NAT-A和NAT-B。

    A发送数据包到公网S,B发送数据包到公网S,则S分别得到了A和B的公网IP，

S也和A B 分别建立了会话，由S发到NAT-A的数据包会被NAT-A直接转发给A，

由S发到NAT-B的数据包会被NAT-B直接转发给B，除了S发出的数据包之外的则会被丢弃。

所以：现在A B 都能分别和S进行全双工通讯了，但是A B之间还不能直接通讯。

    解决办法是：A向B的公网IP发送一个数据包，则NAT-A能接收来自NAT-B的数据包

并转发给A了（即B现在能访问A了）；再由S命令B向A的公网IP发送一个数据包，则

NAT-B能接收来自NAT-A的数据包并转发给B了（即A现在能访问B了）。

为了保证A的路由器有与B的session，A要定时与B做心跳包，同样，B也要定时与A做心跳，这样，双方的通信通道都是通的，就可以进行任意的通信了。

但是TCP和UDP在打洞上却有点不同。这是因为伯克利socket（标准socket规范）的

API造成的。

    UDP的socket允许多个socket绑定到同一个本地端口，而TCP的socket则不允许。

    这是这样一个意思：A B要连接到S，肯定首先A B双方都会在本地创建一个socket，

去连接S上的socket。创建一个socket必然会绑定一个本地端口（就算应用程序里面没写

端口，实际上也是绑定了的，至少java确实如此），假设为8888，这样A和B才分别建立了到

S的通信信道。接下来就需要打洞了，打洞则需要A和B分别发送数据包到对方的公网IP。但是

问题就在这里：因为NAT设备是根据端口号来确定session，如果是UDP的socket，A B可以

分别再创建socket，然后将socket绑定到8888，这样打洞就成功了。但是如果是TCP的

socket，则不能再创建socket并绑定到8888了，这样打洞就无法成功。

**UDP打洞**的过程大致如此：

1、双方都通过UDP与服务器通讯后，网关默认就是做了一个外网IP和端口号 与你内网IP与端口号的映射，这个无需设置的，服务器也不需要知道客户的真正内网IP

2、用户A先通过服务器知道用户B的外网地址与端口

3、用户A向用户B的外网地址与端口发送消息，

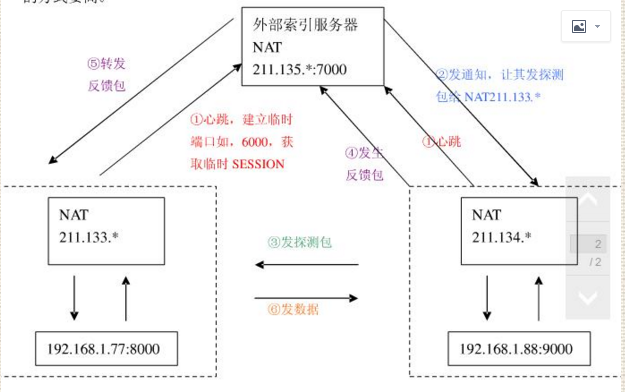
4、在这一次发送中，用户B的网关会拒收这条消息，因为它的映射中并没有这条规则。

5、但是用户A的网关就会增加了一条允许规则，允许接收从B发送过来的消息

6、服务器要求用户B发送一个消息到用户A的外网IP与端口号

7、用户B发送一条消息，这时用户A就可以接收到B的消息，而且网关B也增加了允许规则

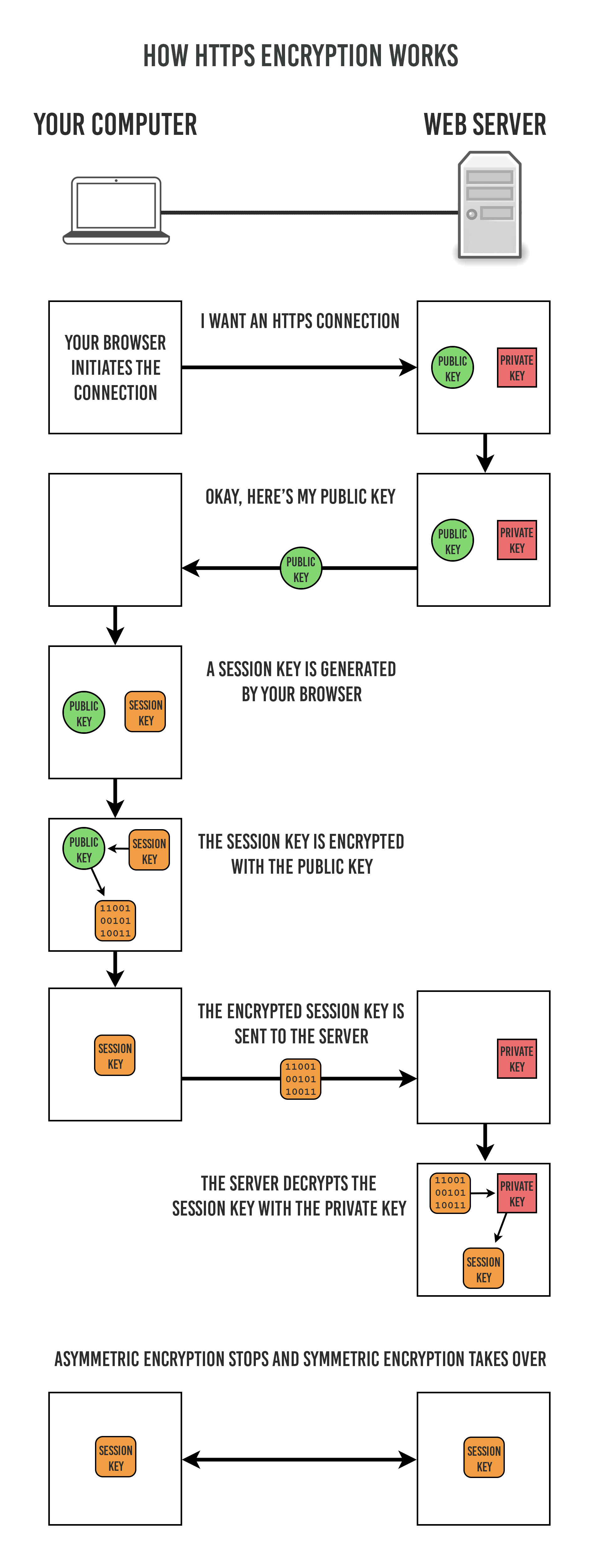
8、之后，由于网关A与网关B都增加了允许规则，所以A与B都可以向对方的外网IP和端口号发送消息。



## 路由器工作原理

路由表，NAT，TTL字段代表路由器跳数

HTTPS：

****

SSL(安全套接层，Secure Sockets Layer)  
TLS（Transport Layer Security，传输层安全）前身是SSL，

ICMP

Ping

telnet

DNS

CDN

反向代理

RPC

Socket编程：

io模型：

一个输入操作通常包括两个阶段：

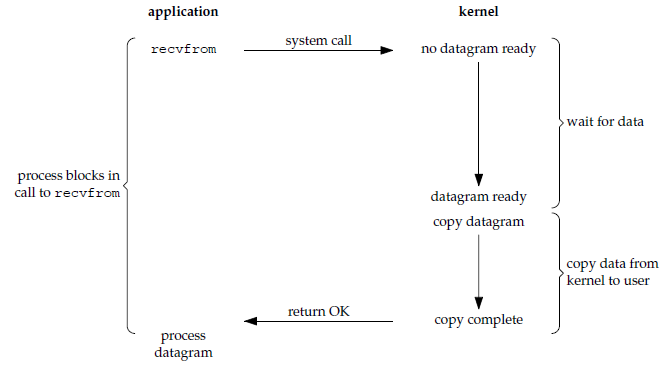
* 等待数据准备好
* 从内核向进程复制数据

Unix 有五种 I/O 模型：

阻塞式 I/O

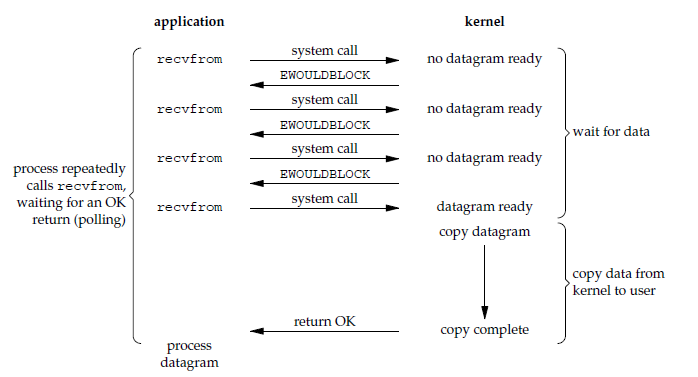
应用进程被阻塞，直到数据从内核缓冲区复制到应用进程缓冲区中才返回。

应该注意到，在阻塞的过程中，其它应用进程还可以执行，因此阻塞不意味着整个操作系统都被阻塞。因为其它应用进程还可以执行，所以不消耗 CPU 时间，这种模型的 CPU 利用率效率会比较高。



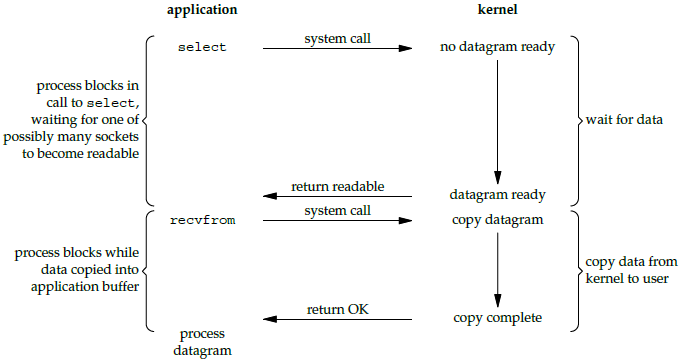
非阻塞式 I/O

应用进程执行系统调用之后，内核返回一个错误码。应用进程可以继续执行，但是需要不断的执行系统调用来获知 I/O 是否完成，这种方式称为轮询（polling）。由于 CPU 要处理更多的系统调用，因此这种模型的 CPU 利用率比较低。



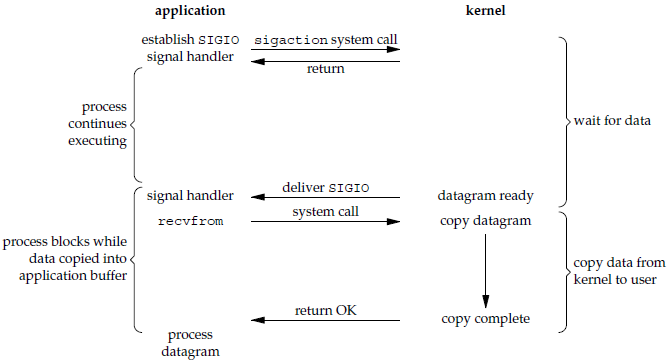
I/O 复用（select 和 poll）

使用 select 或者 poll 等待数据，并且可以等待多个套接字中的任何一个变为可读。这一过程会被阻塞，当某一个套接字可读时返回，之后再使用 recvfrom 把数据从内核复制到进程中。相比于多进程和多线程技术，I/O 复用不需要进程线程创建和切换的开销，系统开销更小。



信号驱动式 I/O（SIGIO）

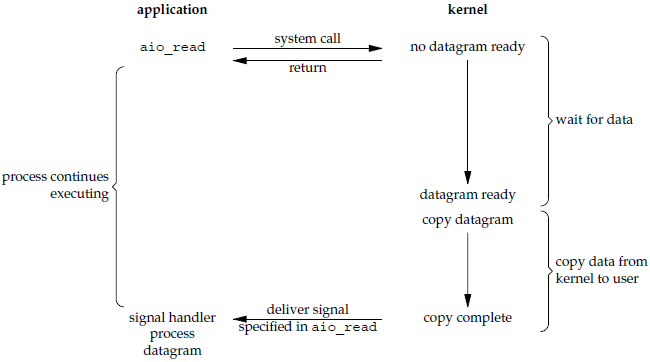
应用进程使用 sigaction 系统调用，内核立即返回，应用进程可以继续执行，也就是说等待数据阶段应用进程是非阻塞的。内核在数据到达时向应用进程发送 SIGIO 信号，应用进程收到之后在信号处理程序中调用 recvfrom 将数据从内核复制到应用进程中。



异步 I/O（AIO）

应用进程执行 aio\_read 系统调用会立即返回，应用进程可以继续执行，不会被阻塞，内核会在所有操作完成之后向应用进程发送信号。

异步 I/O 与信号驱动 I/O 的区别在于，异步 I/O 的信号是通知应用进程 I/O 完成，而信号驱动 I/O 的信号是通知应用进程可以开始 I/O。



## Select对比poll

select 会修改描述符，而 poll 不会；

select 的描述符类型使用数组实现，FD\_SETSIZE 大小默认为 1024，因此默认只能监听 1024 个描述符。如果要监听更多描述符的话，需要修改 FD\_SETSIZE 之后重新编译；而 poll 的描述符类型使用链表实现，没有描述符数量的限制；

poll 提供了更多的事件类型，并且对描述符的重复利用上比 select 高。

如果一个线程对某个描述符调用了 select 或者 poll，另一个线程关闭了该描述符，会导致调用结果不确定。

select 和 poll 速度都比较慢。

* select 和 poll 每次调用都需要将全部描述符从应用进程缓冲区复制到内核缓冲区。
* select 和 poll 的返回结果中没有声明哪些描述符已经准备好，所以如果返回值大于 0 时，应用进程都需要使用轮询的方式来找到 I/O 完成的描述符。

Epoll

int epoll\_create(int size);

int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event)；

int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout);

epoll\_ctl() 用于向内核注册新的描述符或者是改变某个文件描述符的状态。已注册的描述符在内核中会被维护在一棵红黑树上，通过回调函数内核会将 I/O 准备好的描述符加入到一个链表中管理，进程调用 epoll\_wait() 便可以得到事件完成的描述符。

从上面的描述可以看出，epoll 只需要将描述符从进程缓冲区向内核缓冲区拷贝一次，并且进程不需要通过轮询来获得事件完成的描述符。

epoll 仅适用于 Linux OS。

应用场景：

1. select 应用场景

select 的 timeout 参数精度为 1ns，而 poll 和 epoll 为 1ms，因此 select 更加适用于实时性要求比较高的场景，比如核反应堆的控制。

select 可移植性更好，几乎被所有主流平台所支持。

2. poll 应用场景

poll 没有最大描述符数量的限制，如果平台支持并且对实时性要求不高，应该使用 poll 而不是 select。

3. epoll 应用场景

只需要运行在 Linux 平台上，有大量的描述符需要同时轮询，并且这些连接最好是长连接。

需要同时监控小于 1000 个描述符，就没有必要使用 epoll，因为这个应用场景下并不能体现 epoll 的优势。

需要监控的描述符状态变化多，而且都是非常短暂的，也没有必要使用 epoll。因为 epoll 中的所有描述符都存储在内核中，造成每次需要对描述符的状态改变都需要通过 epoll\_ctl() 进行系统调用，频繁系统调用降低效率。并且 epoll 的描述符存储在内核，不容易调试。

## Select对比epoll

1.select的句柄数目受限，在linux/posix\_types.h头文件有这样的声明：#define \_\_FD\_SETSIZE    1024  表示select最多同时监听1024个fd。而epoll没有，它的限制是最大的打开文件句柄数目。

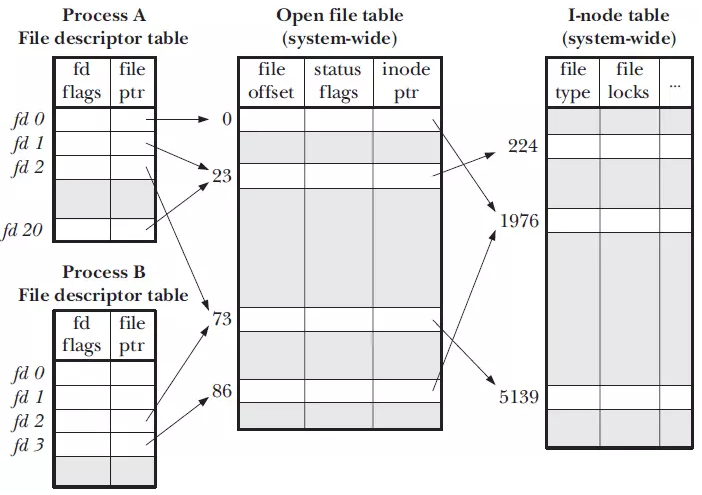
2.epoll的最大好处是不会随着FD的数目增长而降低效率，在selec中采用轮询处理，其中的数据结构类似一个数组的数据结构，而epoll是维护一个队列，直接看队列是不是空就可以了。epoll只会对"活跃"的socket进行操作---这是因为在内核实现中epoll是根据每个fd上面的callback函数实现的。那么，只有"活跃"的socket才会主动的去调用 callback函数（把这个句柄加入队列），其他idle状态句柄则不会，在这点上，epoll实现了一个"伪"AIO。但是如果绝大部分的I/O都是“活跃的”，每个I/O端口使用率很高的话，epoll效率不一定比select高（可能是要维护队列复杂）。

3.使用mmap加速内核与用户空间的消息传递。无论是select,poll还是epoll都需要内核把FD消息通知给用户空间，如何避免不必要的内存拷贝就很重要，在这点上，epoll是通过内核于用户空间mmap同一块内存实现的。

## Linux 文件描述符

由内核维护的3个数据结构：

* 进程级**文件描述符表**(file descriptor table)
* 系统级**打开文件表**(open file table)
* 文件系统**i-node表**(i-node table)



## 文件描述符表

内核为每个进程维护一个**文件描述符表**，该表每一条目都记录了单个文件描述符的相关信息，包括：

* **控制标志**(flags)，目前内核仅定义了一个，即close-on-exec
* **打开文件描述体指针**

## 打开文件表

内核对所有打开的文件维护一个系统级别的**打开文件描述表**(open file description table)，简称**打开文件表**。表中条目称为**打开文件描述体**(open file description)，存储了与一个打开文件相关的全部信息，包括：

* **文件偏移量**(file offset)，调用read()和write()更新，调用lseek()直接修改
* **访问模式**，由open()调用设置，例如：只读、只写或读写等
* **i-node对象指针**

## i-node表

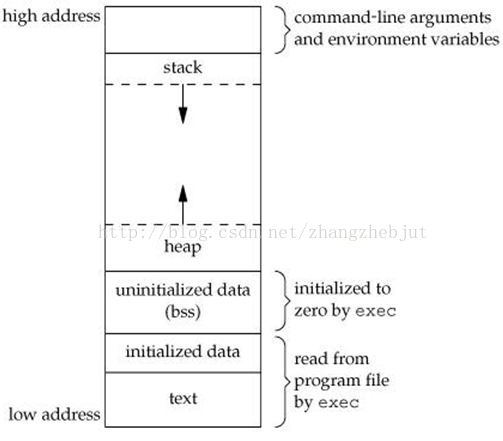
每个文件系统会为存储于其上的所有文件(包括目录)维护一个i-node表，单个i-node包含以下信息：

* **文件类型**(file type)，可以是常规文件、目录、套接字或FIFO
* **访问权限**
* **文件锁列表**(file locks)
* **文件大小**
* 等等

i-node存储在磁盘设备上，内核在内存中维护了一个副本，这里的**i-node表**为后者。副本除了原有信息，还包括：引用计数(从打开文件描述体)、所在设备号以及一些临时属性，例如文件锁。

Netty

## Linux进程内存结构



程序段(Text):程序代码在内存中的映射，存放函数体的二进制代码。

初始化过的数据(Data):在程序运行初已经对变量进行初始化的数据。

未初始化过的数据(BSS):在程序运行初未对变量进行初始化的数据。

栈 (Stack):存储局部、临时变量，函数调用时，存储函数的返回指针，用于控制函数的调用和返回。在程序块开始时自动分配内存,结束时自动释放内存，其操作方式类似于数据结构中的栈。

堆 (Heap):存储动态内存分配,需要程序员手工分配,手工释放.注意它与数据结构中的堆是两回事，分配方式类似于链表。

正常情况下，Linux进程不能对用来存放程序代码的内存区域执行写操作，即程序代码是以只读的方式加载到内存中，但它可以被多个进程安全的共享。

## Linux inode

inode 具体包含以下信息：

权限 (read/write/excute)；

拥有者与群组 (owner/group)；

容量；

建立或状态改变的时间 (ctime)；

最近一次的读取时间 (atime)；

最近修改的时间 (mtime)；

定义文件特性的旗标 (flag)，如 SetUID...；

该文件真正内容的指向 (pointer)。

inode 具有以下特点：

每个 inode 大小均固定为 128 bytes (新的 ext4 与 xfs 可设定到 256 bytes)；

每个文件都仅会占用一个 inode。

## Linux命令：

locate文件搜索。可以用关键字或者正则表达式进行搜索。它存储在内存中，并且每天更新一次，所以无法用 locate 搜索新建的文件。可以使用 updatedb 来立即更新数据库。

find文件搜索。可以使用文件的属性和权限进行搜索。

使用 tee 指令，一个输出会同时传送到文件和屏幕上。

awk可以根据字段的某些条件进行匹配，例如匹配字段小于某个值的那一行数据。

$ awk '条件类型 1 {动作 1} 条件类型 2 {动作 2} ...' filename

netstat查看占用端口的进程

## linux进程

wait()父进程调用 wait() 会一直阻塞，直到收到一个子进程退出的 SIGCHLD 信号，之后 wait() 函数会销毁子进程并返回。

waitpid()作用和 wait() 完全相同，但是多了两个可由用户控制的参数 pid 和 options。

孤儿进程

一个父进程退出，而它的一个或多个子进程还在运行，那么这些子进程将成为孤儿进程。

孤儿进程将被 init 进程（进程号为 1）所收养，并由 init 进程对它们完成状态收集工作。

由于孤儿进程会被 init 进程收养，所以孤儿进程不会对系统造成危害。

僵尸进程

一个子进程的进程描述符在子进程退出时不会释放，只有当父进程通过 wait() 或 waitpid() 获取了子进程信息后才会释放。如果子进程退出，而父进程并没有调用 wait() 或 waitpid()，那么子进程的进程描述符仍然保存在系统中，这种进程称之为僵尸进程。

僵尸进程通过 ps 命令显示出来的状态为 Z（zombie）。

系统所能使用的进程号是有限的，如果产生大量僵尸进程，将因为没有可用的进程号而导致系统不能产生新的进程。

要消灭系统中大量的僵尸进程，只需要将其父进程杀死，此时僵尸进程就会变成孤儿进程，从而被 init 所收养，这样 init 就会释放所有的僵尸进程所占有的资源，从而结束僵尸进程。

单例模式双重校验

双重校验锁-线程安全

uniqueInstance 只需要被实例化一次，之后就可以直接使用了。加锁操作只需要对实例化那部分的代码进行，只有当 uniqueInstance 没有被实例化时，才需要进行加锁。

双重校验锁先判断 uniqueInstance 是否已经被实例化，如果没有被实例化，那么才对实例化语句进行加锁。

public class Singleton {

private volatile static Singleton uniqueInstance;

private Singleton() {

}

public static Singleton getUniqueInstance() {

if (uniqueInstance == null) {

synchronized (Singleton.class) {

if (uniqueInstance == null) {

uniqueInstance = new Singleton();

}

}

}

return uniqueInstance;

}

}

考虑下面的实现，也就是只使用了一个 if 语句。在 uniqueInstance == null 的情况下，如果两个线程都执行了 if 语句，那么两个线程都会进入 if 语句块内。虽然在 if 语句块内有加锁操作，但是两个线程都会执行 uniqueInstance = new Singleton(); 这条语句，只是先后的问题，那么就会进行两次实例化。因此必须使用双重校验锁，也就是需要使用两个 if 语句。

if (uniqueInstance == null) {

synchronized (Singleton.class) {

uniqueInstance = new Singleton();

}

}

uniqueInstance 采用 volatile 关键字修饰也是很有必要的， uniqueInstance = new Singleton(); 这段代码其实是分为三步执行：

为 uniqueInstance 分配内存空间

初始化 uniqueInstance

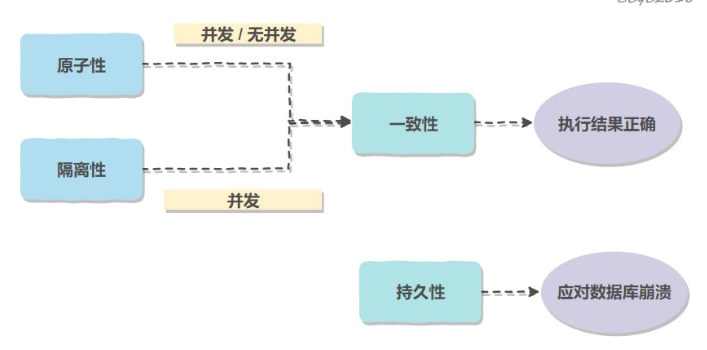
将 uniqueInstance 指向分配的内存地址

但是由于 JVM 具有指令重排的特性，执行顺序有可能变成 1>3>2。指令重排在单线程环境下不会出现问题，但是在多线程环境下会导致一个线程获得还没有初始化的实例。例如，线程 T1 执行了 1 和 3，此时 T2 调用 getUniqueInstance() 后发现 uniqueInstance 不为空，因此返回 uniqueInstance，但此时 uniqueInstance 还未被初始化。

使用 volatile 可以禁止 JVM 的指令重排，保证在多线程环境下也能正常运行

数据库：

ACID原子性（Atomicity）一致性（Consistency）隔离性（Isolation）持久性（Durability）。



并发一致性问题：

* 丢失修改，T1 先修改，T2 随后修改，T2 的修改覆盖了 T1 的修改。
* 读脏数据，T1 修改一个数据，T2 随后读取这个数据。如果 T1 撤销了这次修改，那么 T2 读取的数据是脏数据。
* 不可重复读，T2 读取一个数据，T1 对该数据做了修改。如果 T2 再次读取这个数据，此时读取的结果和第一次读取的结果不同。
* 幻影读，T1 读取某个范围的数据，T2 在这个范围内插入新的数据，T1 再次读取这个范围的数据，此时读取的结果和和第一次读取的结果不同。

MySQL 中提供了两种封锁粒度：行级锁以及表级锁。

应该尽量只锁定需要修改的那部分数据，而不是所有的资源。锁定的数据量越少，发生锁争用的可能就越小，系统的并发程度就越高。

但是加锁需要消耗资源，锁的各种操作（包括获取锁、释放锁、以及检查锁状态）都会增加系统开销。因此封锁粒度越小，系统开销就越大。

在选择封锁粒度时，需要在锁开销和并发程度之间做一个权衡。

三级封锁协议：

**一级封锁协议**

事务 T 要修改数据 A 时必须加 写 锁，直到 T 结束才释放锁。可以解决丢失修改问题

**二级封锁协议**

在一级的基础上，要求读取数据 A 时必须加 读锁，读取完马上释放 读 锁。可以解决读脏数据问题

**三级封锁协议**

在二级的基础上，要求读取数据 A 时必须加 读 锁，直到事务结束了才能释放 读 锁。可以解决不可重复读的问题

### 两段锁协议

加锁和解锁分为两个阶段进行。

可串行化调度是指，通过并发控制，使得并发执行的事务结果与某个串行执行的事务结果相同。

事务遵循两段锁协议是保证可串行化调度的充分条件。例如以下操作满足两段锁协议，它是可串行化调度。

lock-x(A)...lock-s(B)...lock-s(C)...unlock(A)...unlock(C)...unlock(B)

但不是必要条件，例如以下操作不满足两段锁协议，但是它还是可串行化调度。

lock-x(A)...unlock(A)...lock-s(B)...unlock(B)...lock-s(C)...unlock(C)

MySQL 的 InnoDB 存储引擎采用两段锁协议，会根据隔离级别在需要的时候自动加锁，并且所有的锁都是在同一时刻被释放，这被称为隐式锁定。