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。

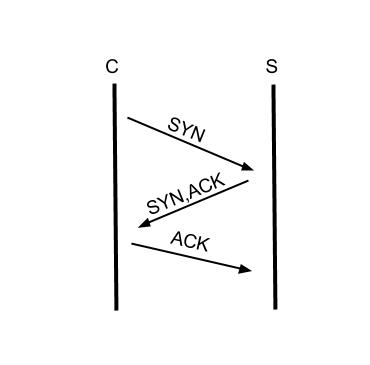
## TCP

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，

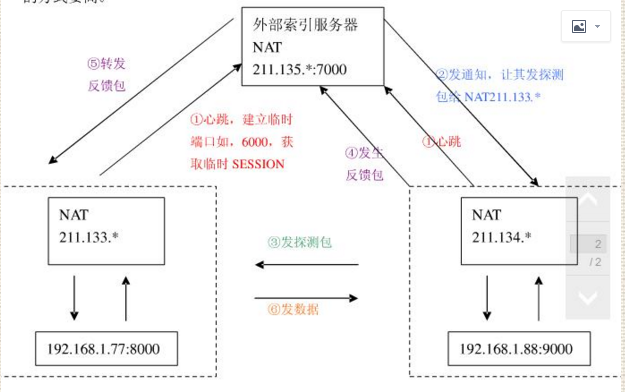
4、在这一次发送中，用户B的网关会拒收这条消息，因为它的映射中并没有这条规则。

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

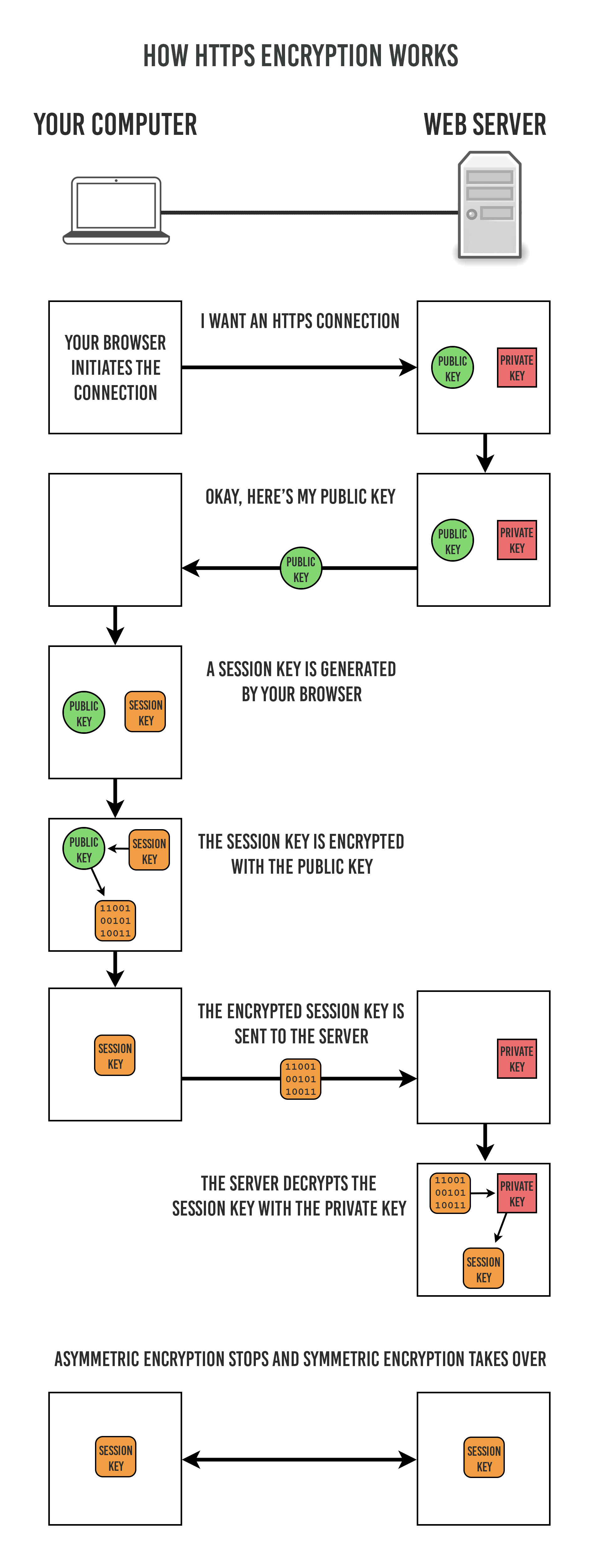
8、之后，由于网关A与网关B都增加了允许规则，所以A与B都可以向对方的外网IP和端口号发送消息。



## 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：

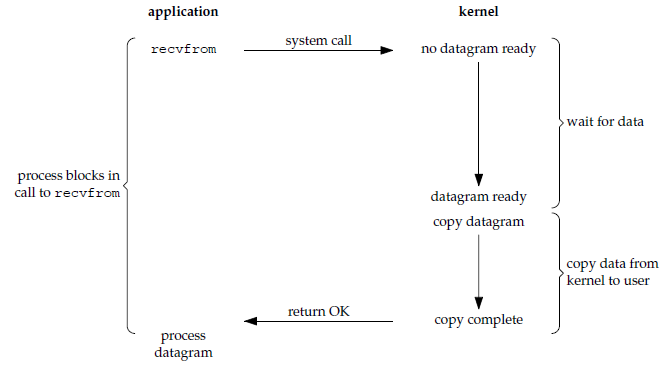
* 等待数据准备好
* 从内核向进程复制数据

Unix 有五种 I/O 模型：

阻塞式 I/O

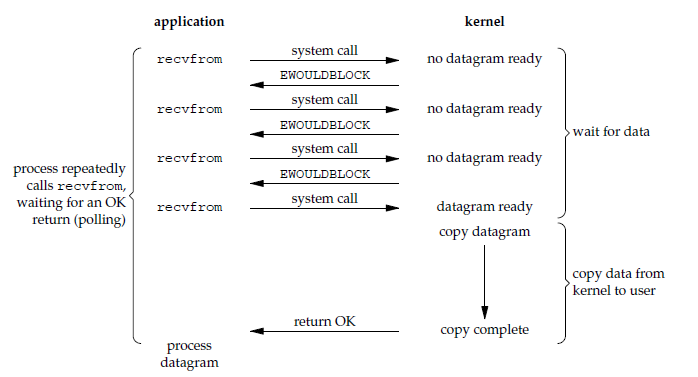
应用进程被阻塞，直到数据从内核缓冲区复制到应用进程缓冲区中才返回。

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。



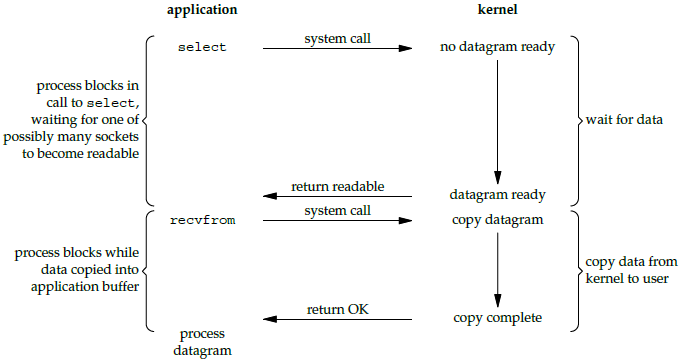
非阻塞式 I/O

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。



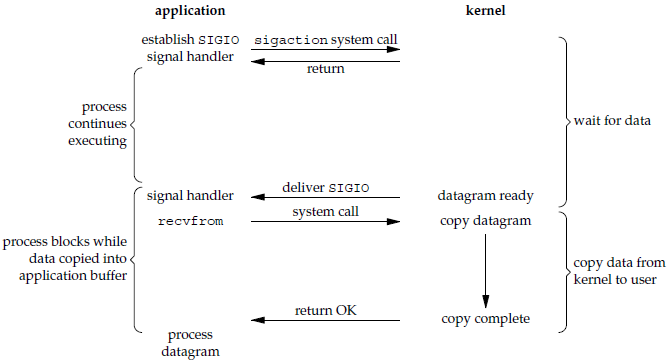
I/O 复用（select 和 poll）

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）

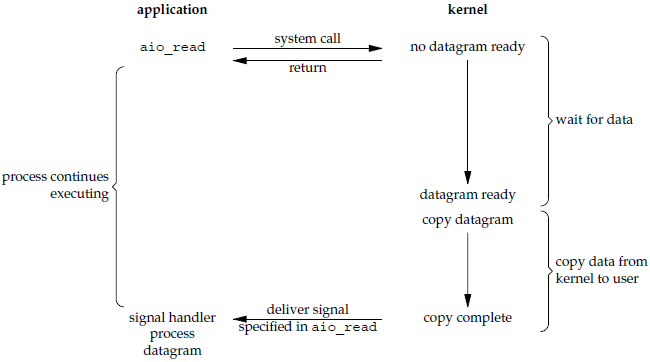
应用进程使用 sigaction 系统调用，内核立即返回，应用进程可以继续执行，也就是说等待数据阶段应用进程是非阻塞的。内核在数据到达时向应用进程发送 SIGIO 信号，应用进程收到之后在信号处理程序中调用 recvfrom 将数据从内核复制到应用进程中。



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。

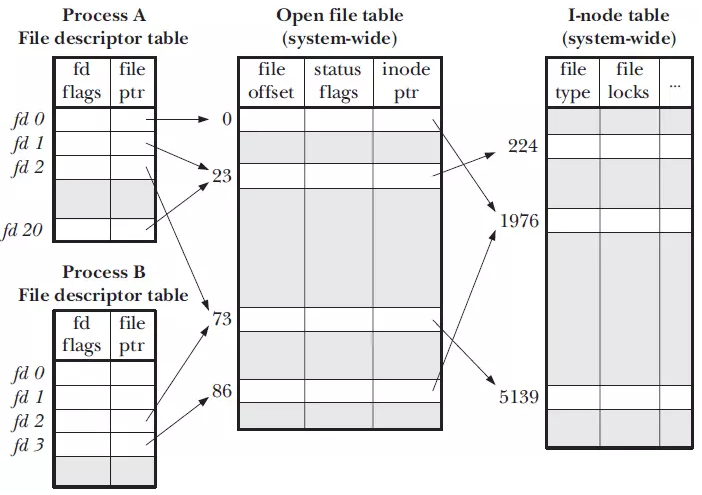
2.epoll的最大好处是不会随着FD的数目增长而降低效率，在selec中采用轮询处理，其中的数据结构类似一个数组的数据结构，而epoll是维护一个队列，直接看队列是不是空就可以了。epoll只会对"活跃"的socket进行操作---这是因为在内核实现中epoll是根据每个fd上面的callback函数实现的。那么，只有"活跃"的socket才会主动的去调用 callback函数（把这个句柄加入队列），其他idle状态句柄则不会，在这点上，epoll实现了一个"伪"AIO。但是如果绝大部分的I/O都是“活跃的”，每个I/O端口使用率很高的话，epoll效率不一定比select高（可能是要维护队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

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

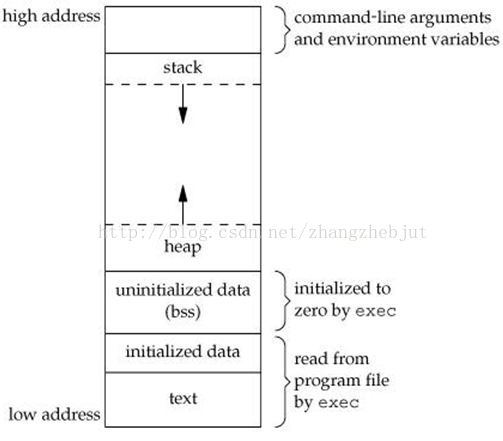
每个文件系统会为存储于其上的所有文件(包括目录)维护一个i-node表，单个i-node包含以下信息：

* **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

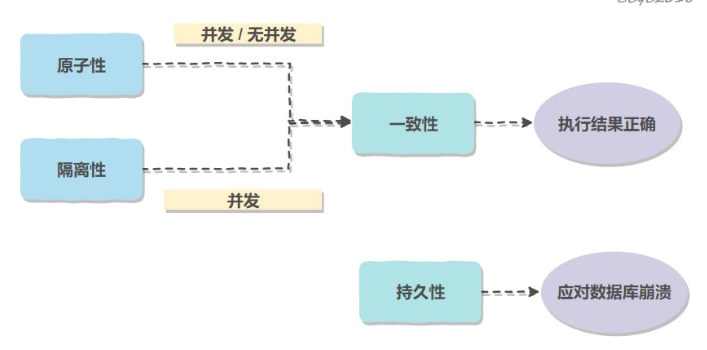
将 uniqueInstance 指向分配的内存地址

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。

## 事务隔离级别

**未提交读（READ UNCOMMITTED）**

事务中的修改，即使没有提交，对其它事务也是可见的。

**提交读（READ COMMITTED）**

一个事务只能读取已经提交的事务所做的修改。换句话说，一个事务所做的修改在提交之前对其它事务是不可见的。

**可重复读（REPEATABLE READ）**

保证在同一个事务中多次读取同样数据的结果是一样的。

**可串行化（SERIALIZABLE）**

强制事务串行执行。

## 多版本并发控制（Multi-Version Concurrency Control, MVCC）

MVCC是 MySQL 的 InnoDB 存储引擎实现隔离级别的一种具体方式，用于实现提交读和可重复读这两种隔离级别。而未提交读隔离级别总是读取最新的数据行，无需使用 MVCC。可串行化隔离级别需要对所有读取的行都加锁，单纯使用 MVCC 无法实现。

**版本号**

系统版本号：是一个递增的数字，每开始一个新的事务，系统版本号就会自动递增。

事务版本号：事务开始时的系统版本号。

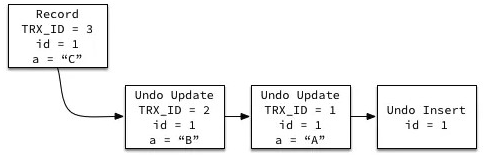
**隐藏的列**

MVCC 在每行记录后面都保存着两个隐藏的列，用来存储两个版本号：

创建版本号：指示创建一个数据行的快照时的系统版本号；

删除版本号：如果该快照的删除版本号大于当前事务版本号表示该快照有效，否则表示该快照已经被删除了。

MVCC 使用到的快照存储在 Undo 日志中，该日志通过回滚指针把一个数据行（Record）的所有快照连接起来。



以下实现过程针对可重复读隔离级别。

当开始新一个事务时，该事务的版本号肯定会大于当前所有数据行快照的创建版本号，理解这一点很关键。

1. SELECT

多个事务必须读取到同一个数据行的快照，并且这个快照是距离现在最近的一个有效快照。但是也有例外，如果有一个事务正在修改该数据行，那么它可以读取事务本身所做的修改，而不用和其它事务的读取结果一致。

把没有对一个数据行做修改的事务称为 T，T 所要读取的数据行快照的创建版本号必须小于 T 的版本号，因为如果大于或者等于 T 的版本号，那么表示该数据行快照是其它事务的最新修改，因此不能去读取它。除此之外，T 所要读取的数据行快照的删除版本号必须大于 T 的版本号，因为如果小于等于 T 的版本号，那么表示该数据行快照是已经被删除的，不应该去读取它。

2. INSERT

将当前系统版本号作为数据行快照的创建版本号。

3. DELETE

将当前系统版本号作为数据行快照的删除版本号。

4. UPDATE

将当前系统版本号作为更新前的数据行快照的删除版本号，并将当前系统版本号作为更新后的数据行快照的创建版本号。可以理解为先执行 DELETE 后执行 INSERT。

Next-Key Locks

Next-Key Locks 是 MySQL 的 InnoDB 存储引擎的一种锁实现。

MVCC 不能解决幻读的问题，Next-Key Locks 就是为了解决这个问题而存在的。在可重复读（REPEATABLE READ）隔离级别下，使用 MVCC + Next-Key Locks 可以解决幻读问题。

Record Locks

锁定一个记录上的索引，而不是记录本身。

如果表没有设置索引，InnoDB 会自动在主键上创建隐藏的聚簇索引，因此 Record Locks 依然可以使用。

Gap Locks

锁定索引之间的间隙，但是不包含索引本身。例如当一个事务执行以下语句，其它事务就不能在 t.c 中插入 15。

SELECT c FROM t WHERE c BETWEEN 10 and 20 FOR UPDATE;

Next-Key Locks

它是 Record Locks 和 Gap Locks 的结合，不仅锁定一个记录上的索引，也锁定索引之间的间隙。例如一个索引包含以下值：10, 11, 13, and 20，那么就需要锁定以下区间：

(negative infinity, 10]

(10, 11]

(11, 13]

(13, 20]

(20, positive infinity)

## Sql：

Groupby

SELECT col, COUNT(\*) AS num

FROM mytable

WHERE col > 2

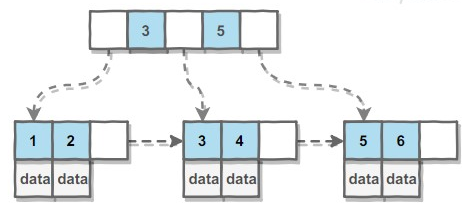
GROUP BY col

HAVING num >= 2;

## Mysql

B Tree 指的是 Balance Tree，也就是平衡树。平衡树是一颗查找树，并且所有叶子节点位于同一层。

在 B+ Tree 中，一个节点中的 key 从左到右非递减排列，如果某个指针的左右相邻 key 分别是 keyi 和 keyi+1，且不为 null，则该指针指向节点的所有 key 大于等于 keyi 且小于等于 keyi+1。



Mysql索引使用B+树

InnoDB 的 B+Tree 索引分为主索引和辅助索引。主索引的叶子节点 data 域记录着完整的数据记录，这种索引方式被称为聚簇索引

辅助索引的叶子节点的 data 域记录着主键的值，因此在使用辅助索引进行查找时，需要先查找到主键值，然后再到主索引中进行查找。

**哈希索引**

哈希索引能以 O(1) 时间进行查找，但是失去了有序性

**全文索引**

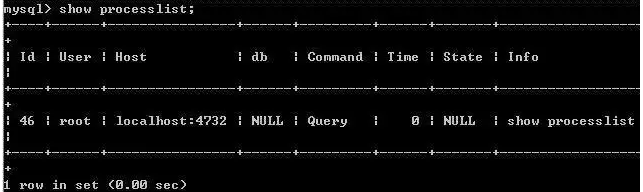
全文索引使用倒排索引实现，它记录着关键词到其所在文档的映射。MyISAM 存储引擎支持全文索引，InnoDB 存储引擎在 MySQL 5.6.4 版本中也开始支持全文索引

对于中到大型的表，索引就非常有效；

但是对于特大型的表，建立和维护索引的代价将会随之增长。这种情况下，需要用到一种技术可以直接区分出需要查询的一组数据，而不是一条记录一条记录地匹配，例如可以使用分区技术。

Explain 用来分析 SELECT 查询语句，开发人员可以通过分析 Explain 结果来优化查询语句。

**show processlist** 显示哪些线程正在运行。



**Time列：**此这个状态持续的时间，单位是秒。

**State列：**显示使用当前连接的sql语句的状态，很重要的列，后续会有所有的状态的描述，请注意，state只是语句执行中的某一个状态，一个 sql语句，以查询为例，可能需要经过copying to tmp table，Sorting result，Sending data等状态才可以完成

## 重构查询方式

### 1. 切分大查询

一个大查询如果一次性执行的话，可能一次锁住很多数据、占满整个事务日志、耗尽系统资源、阻塞很多小的但重要的查询。

DELETE FROM messages WHERE create < DATE\_SUB(NOW(), INTERVAL 3 MONTH);

rows\_affected = 0

do {

rows\_affected = do\_query(

"DELETE FROM messages WHERE create < DATE\_SUB(NOW(), INTERVAL 3 MONTH) LIMIT 10000")

} while rows\_affected > 0

### 2. 分解大连接查询

将一个大连接查询分解成对每一个表进行一次单表查询，然后在应用程序中进行关联，这样做的好处有：

* 让缓存更高效。对于连接查询，如果其中一个表发生变化，那么整个查询缓存就无法使用。而分解后的多个查询，即使其中一个表发生变化，对其它表的查询缓存依然可以使用。
* 分解成多个单表查询，这些单表查询的缓存结果更可能被其它查询使用到，从而减少冗余记录的查询。
* 减少锁竞争；
* 在应用层进行连接，可以更容易对数据库进行拆分，从而更容易做到高性能和可伸缩。
* 查询本身效率也可能会有所提升。例如下面的例子中，使用 IN() 代替连接查询，可以让 MySQL 按照 ID 顺序进行查询，这可能比随机的连接要更高效。

SELECT \* FROM tab

JOIN tag\_post ON tag\_post.tag\_id=tag.id

JOIN post ON tag\_post.post\_id=post.id

WHERE tag.tag='mysql';

SELECT \* FROM tag WHERE tag='mysql';

SELECT \* FROM tag\_post WHERE tag\_id=1234;

SELECT \* FROM post WHERE post.id IN (123,456,567,9098,8904);

## InnoDB

实现了四个标准的隔离级别，默认级别是可重复读（REPEATABLE READ）。在可重复读隔离级别下，通过多版本并发控制（MVCC）+ 间隙锁（Next-Key Locking）防止幻影读。

**水平切分**

水平切分又称为 Sharding，它是将同一个表中的记录拆分到多个结构相同的表中。

当一个表的数据不断增多时，Sharding 是必然的选择，它可以将数据分布到集群的不同节点上，从而缓存单个数据库的压力。

**Sharding 策略**

哈希取模：hash(key) % N；

范围：可以是 ID 范围也可以是时间范围；

映射表：使用单独的一个数据库来存储映射关系。

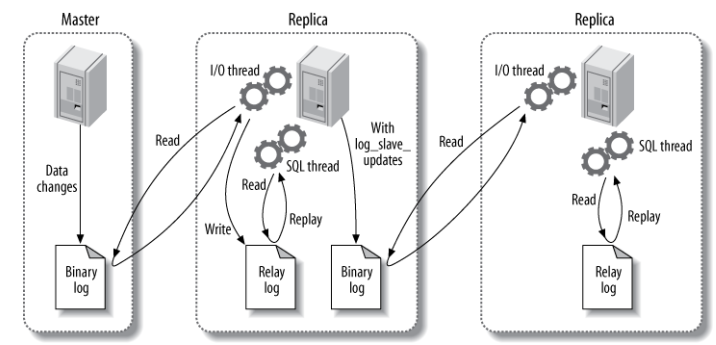
**垂直切分**

垂直切分是将一张表按列切分成多个表，通常是按照列的关系密集程度进行切分，也可以利用垂直切分将经常被使用的列和不经常被使用的列切分到不同的表中。

## 主从复制

主要涉及三个线程：binlog 线程、I/O 线程和 SQL 线程。

* **binlog 线程** ：负责将主服务器上的数据更改写入二进制日志（Binary log）中。
* **I/O 线程** ：负责从主服务器上读取二进制日志，并写入从服务器的中继日志（Relay log）。
* **SQL 线程** ：负责读取中继日志，解析出主服务器已经执行的数据更改并在从服务器中执行。



* 主从服务器负责各自的读和写，极大程度缓解了锁的争用；
* 从服务器可以使用 MyISAM，提升查询性能以及节约系统开销；

## Redis

键的类型只能为字符串，值支持的五种类型数据类型为：字符串、列表、集合、有序集合、散列表

Redis有哪些数据结构？

字符串String、字典Hash、列表List、集合Set、有序集合SortedSet。

如果你是Redis中高级用户，还需要加上下面几种数据结构HyperLogLog、Geo、Pub/Sub。

如果你说还玩过Redis Module，像BloomFilter，RedisSearch，Redis-ML，面试官得眼睛就开始发亮了。

使用过Redis分布式锁么，它是什么回事？

先拿setnx来争抢锁，抢到之后，再用expire给锁加一个过期时间防止锁忘记了释放。

这时候对方会告诉你说你回答得不错，然后接着问如果在setnx之后执行expire之前进程意外crash或者要重启维护了，那会怎么样？

这时候你要给予惊讶的反馈：唉，是喔，这个锁就永远得不到释放了。紧接着你需要抓一抓自己得脑袋，故作思考片刻，好像接下来的结果是你主动思考出来的，然后回答：我记得set指令有非常复杂的参数，这个应该是可以同时把setnx和expire合成一条指令来用的。对方这时会显露笑容，心里开始默念：摁，这小子还不错。

假如Redis里面有1亿个key，其中有10w个key是以某个固定的已知的前缀开头的，如果将它们全部找出来？

使用keys指令可以扫出指定模式的key列表。

对方接着追问：如果这个redis正在给线上的业务提供服务，那使用keys指令会有什么问题？

这个时候你要回答redis关键的一个特性：redis的单线程的。keys指令会导致线程阻塞一段时间，线上服务会停顿，直到指令执行完毕，服务才能恢复。这个时候可以使用scan指令，scan指令可以无阻塞的提取出指定模式的key列表，但是会有一定的重复概率，在客户端做一次去重就可以了，但是整体所花费的时间会比直接用keys指令长。

使用过Redis做异步队列么，你是怎么用的？

一般使用list结构作为队列，rpush生产消息，lpop消费消息。当lpop没有消息的时候，要适当sleep一会再重试。

如果对方追问可不可以不用sleep呢？list还有个指令叫blpop，在没有消息的时候，它会阻塞住直到消息到来。

如果对方追问能不能生产一次消费多次呢？使用pub/sub主题订阅者模式，可以实现1:N的消息队列。

如果对方追问pub/sub有什么缺点？

在消费者下线的情况下，生产的消息会丢失，得使用专业的消息队列如rabbitmq等。

如果对方追问redis如何实现延时队列？

使用sortedset，拿时间戳作为score，消息内容作为key调用zadd来生产消息，消费者用zrangebyscore指令获取N秒之前的数据轮询进行处理。

如果有大量的key需要设置同一时间过期，一般需要注意什么？

如果大量的key过期时间设置的过于集中，到过期的那个时间点，redis可能会出现短暂的卡顿现象。一般需要在时间上加一个随机值，使得过期时间分散一些。

Redis如何做持久化的？

bgsave做镜像全量持久化，aof做增量持久化。

因为bgsave会耗费较长时间，不够实时，在停机的时候会导致大量丢失数据，所以需要aof来配合使用。在redis实例重启时，会使用bgsave持久化文件重新构建内存，再使用aof重放近期的操作指令来实现完整恢复重启之前的状态。

对方追问那如果突然机器掉电会怎样？取决于aof日志sync属性的配置，如果不要求性能，在每条写指令时都sync一下磁盘，就不会丢失数据。但是在高性能的要求下每次都sync是不现实的，一般都使用定时sync，比如1s1次，这个时候最多就会丢失1s的数据。

对方追问bgsave的原理是什么？

你给出两个词汇就可以了，fork和cow。fork是指redis通过创建子进程来进行bgsave操作，cow指的是copy on write，子进程创建后，父子进程共享数据段，父进程继续提供读写服务，写脏的页面数据会逐渐和子进程分离开来。

Pipeline有什么好处，为什么要用pipeline？

可以将多次IO往返的时间缩减为一次，前提是pipeline执行的指令之间没有因果相关性。使用redis-benchmark进行压测的时候可以发现影响redis的QPS峰值的一个重要因素是pipeline批次指令的数目。

Redis的同步机制了解么？

Redis可以使用主从同步，从从同步。第一次同步时，主节点做一次bgsave，并同时将后续修改操作记录到内存buffer，待完成后将rdb文件全量同步到复制节点，复制节点接受完成后将rdb镜像加载到内存。加载完成后，再通知主节点将期间修改的操作记录同步到复制节点进行重放就完成了同步过程。

是否使用过Redis集群，集群的原理是什么？

Redis Sentinal着眼于高可用，在master宕机时会自动将slave提升为master，继续提供服务。

Redis Cluster着眼于扩展性，在单个redis内存不足时，使用Cluster进行分片存储。

gossip协议

分布式一致性

Paxos算法

LVS

Java：

基本类型的缓冲池：

valueOf() 方法的实现比较简单，就是先判断值是否在缓存池中，如果在的话就直接返回缓存池的内容。基本类型对应的缓冲池如下：

boolean values true and false

all byte values

short values between -128 and 127

int values between -128 and 127

char in the range \u0000 to \u007F

String 被声明为 final，因此它不可被继承。value 数组被声明为 final，这意味着 value 数组初始化之后就不能再引用其它数组。

**不可变的好处**

1. 可以缓存 hash 值

因为 String 的 hash 值经常被使用，例如 String 用做 HashMap 的 key。不可变的特性可以使得 hash 值也不可变，因此只需要进行一次计算。

2. String Pool 的需要

如果一个 String 对象已经被创建过了，那么就会从 String Pool 中取得引用。只有 String 是不可变的，才可能使用 String Pool。

3. 安全性

String 经常作为参数，String 不可变性可以保证参数不可变。例如在作为网络连接参数的情况下如果 String 是可变的，那么在网络连接过程中，String 被改变，改变 String 对象的那一方以为现在连接的是其它主机，而实际情况却不一定是。

4. 线程安全

String 不可变性天生具备线程安全，可以在多个线程中安全地使用。

在 Java 7 之前，String Pool 被放在运行时常量池中，它属于永久代。而在 Java 7，String Pool 被移到堆中。这是因为永久代的空间有限，在大量使用字符串的场景下会导致 OutOfMemoryError 错误。

从 Java 8 开始，接口也可以拥有默认的方法实现，这是因为不支持默认方法的接口的维护成本太高了。在 Java 8 之前，如果一个接口想要添加新的方法，那么要修改所有实现了该接口的类。

接口的字段默认都是 static 和 final 的。

JIT编译器、逃逸分析

最新网络协议http2、quic

G1、ZGC垃圾回收算法

## Jvm Monitor

1.Monitor是一种用来实现同步的工具

2.与每个java对象相关联，即每个java对象都有一个Monitor与之对应

3.Monitor是实现Sychronized(内置锁)的基础

Monitor具体是怎么实现的？

1.Monitor是在jvm底层实现的，底层代码是c++

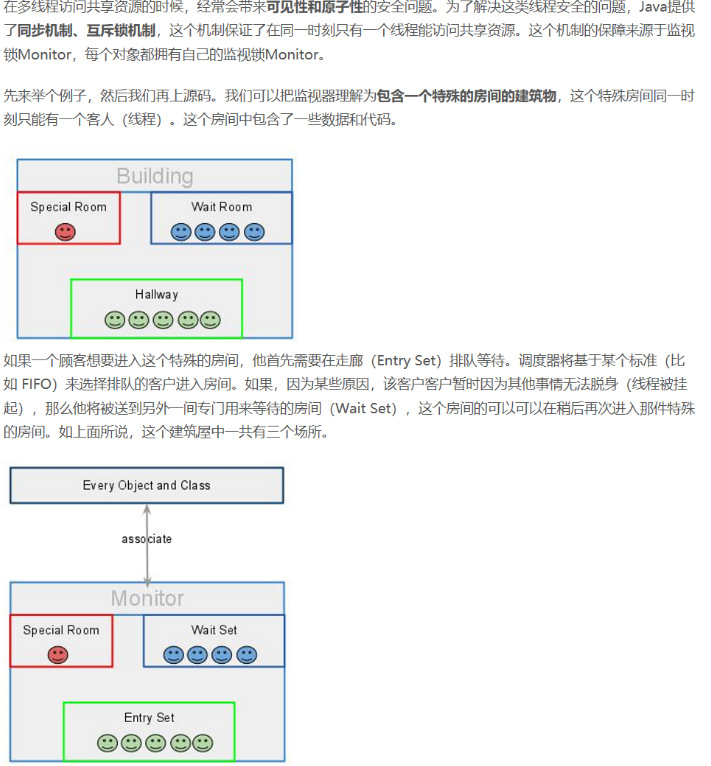
2.Monitor的enter方法：获取锁

3.Monitor的exit方法：释放锁

4.Monitor的wait方法：为java的Object的wait方法提供支持

5.Monitor的notify方法：为java的Object的notify方法提供支持

6.Monitor的notifyAll方法：为java的Object的notifyAll方法提供支持



**线程池提交任务时的执行顺序如下：**

向线程池提交任务时，会首先判断线程池中的线程数是否大于设置的核心线程数，如果不大于，就创建一个核心线程来执行任务。

如果大于核心线程数，就会判断缓冲队列是否满了，如果没有满，则放入队列，等待线程空闲时执行任务。

如果队列已经满了，则判断是否达到了线程池设置的最大线程数，如果没有达到，就创建新线程来执行任务。

如果已经达到了最大线程数，则执行指定的拒绝策略。这里需要注意队列的判断与最大线程数判断的顺序，不要搞反。

 反压机制（BackPressure）反压通常产生于这样的场景：短时负载高峰导致系统接收数据的速率远高于它处理数据的速率。许多日常问题都会导致反压，例如，垃圾回收停顿可能会导致流入的数据快速堆积，或者遇到大促或秒杀活动导致流量陡增。

类的加载分为加载、链接、初始化，其中链接又包括验证、准备、解析三步。看到图中上半部分深绿色，我们逐个分析：

加载是文件到内存的过程。通过类的完全限定名查找此类字节码文件，并利用字节码文件创建一个Class对象

验证是对类文件内容验证。目的在于确保Class文件符合当前虚拟机要求，不会危害虚拟机自身安全。主要包括四种：文件格式验证，元数据验证，字节码验证，符号引用验证。

准备阶段是进行内存分配。为类变量也就是类中由static修饰的变量分配内存，并且设置初始值，这里要注意，初始值是0或者null，而不是代码中设置的具体值，代码中设置的值是在初始化阶段完成的。另外这里也不包含用final修饰的静态变量，因为final在编译的时候就会分配了。

解析主要是解析字段、接口、方法。主要是将常量池中的符号引用替换为直接引用的过程。直接引用就是直接指向目标的指针、相对偏移量等。

最后是初始化：主要完成静态块执行与静态变量的赋值。这是类加载最后阶段，若被加载类的父类没有初始化，则先对父类进行初始化。

ZGC

Java 中的网络支持：

InetAddress：用于表示网络上的硬件资源，即 IP 地址；

URL：统一资源定位符；

Sockets：使用 TCP 协议实现网络通信；

Datagram：使用 UDP 协议实现网络通信。

Redis 的 RedLock 算法

使用了多个 Redis 实例来实现分布式锁，这是为了保证在发生单点故障时仍然可用。

尝试从 N 个相互独立 Redis 实例获取锁；

计算获取锁消耗的时间，只有当这个时间小于锁的过期时间，并且从大多数（N / 2 + 1）实例上获取了锁，那么就认为锁获取成功了；

如果锁获取失败，就到每个实例上释放锁。

Eureka Zookeeper

Mq

Nginx

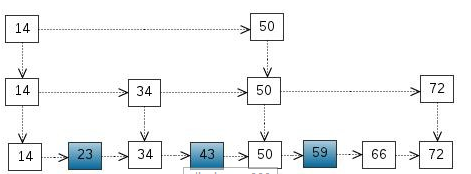
Ehcache, memcache,redis， ssdb高性能

## Kill -3

打印死机jvm的堆栈信息

## 跳表

跳表的核心思想，其实也是一种通过“空间来换取时间”的一个算法，通过在每个节点中增加了向前的指针，从而提升查找的效率。



## CAP

分布式系统不可能同时满足一致性（C：Consistency）、可用性（A：Availability）和分区容忍性（P：Partition Tolerance），最多只能同时满足其中两项。

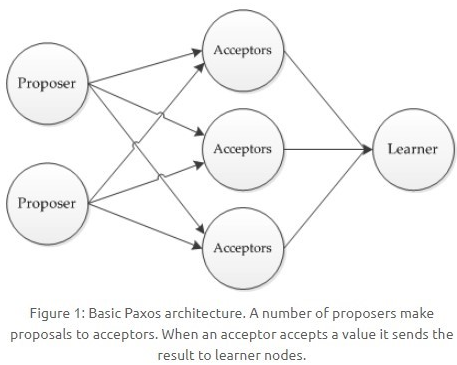
BASE 理论是对 CAP 中一致性和可用性权衡的结果，它的核心思想是：即使无法做到强一致性，但每个应用都可以根据自身业务特点，采用适当的方式来使系统达到最终一致性。

## Paxos

用于达成共识性问题，即对多个节点产生的值，该算法能保证只选出唯一一个值。

主要有三类节点：

* 提议者（Proposer）：提议一个值；
* 接受者（Acceptor）：对每个提议进行投票；
* 告知者（Learner）：被告知投票的结果，不参与投票过程。



一个proposer选择一个编号为n的议案，向所有的acceptor发送prepare请求

如果acceptor已经响应的prepare请求中议案编号都比n小，则它承诺不再响应prepare请求或者accept请求中议案编号小于n的，

并且找出已经accept的最大议案的value返回给该proposer

如果已响应的编号比n大，则直接忽略该prepare请求。

如果proposer收到了过半的acceptors响应，那么将提出一个议案（n，v）,v就是上述所有acceptor响应中最大accept议案的value，或者是proposer自己的value。然后将该议案发送给所有的acceptor。

这个请求叫做accept请求，这一步才是所谓发送议案请求，而前面的prepare请求更多的是一个构建出最终议案(n,v)的过程。

如果没有收到过半响应，则增大议案编号，重新回到Phase 1阶段。

acceptor接收到编号为n的议案，如果acceptor还没有对大于n的议案的prepare请求响应过，则acceptor就accept该议案，否则拒绝。

## Raft

Raft 也是分布式一致性协议，主要是用来竞选主节点。

有三种节点：Follower、Candidate 和 Leader。Leader 会周期性的发送心跳包给 Follower。每个 Follower 都设置了一个随机的竞选超时时间，一般为 150ms~300ms，如果在这个时间内没有收到 Leader 的心跳包，就会变成 Candidate，进入竞选阶段。

此时 Candidate A 发送投票请求给其它所有节点，其它节点会对请求进行回复，如果超过一半的节点回复了，那么该 Candidate 就会变成 Leader。由于每个节点设置的随机竞选超时时间不同，因此下一次再次出现多个 Candidate 并获得同样票数的概率很低。

## 分布式锁：

数据库锁：

优点：直接使用数据库，使用简单。

缺点：分布式系统大多数瓶颈都在数据库，使用数据库锁会增加数据库负担。

缓存锁：

优点：性能高，实现起来较为方便，在允许偶发的锁失效情况，不影响系统正常使用，建议采用缓存锁。

缺点：通过锁超时机制不是十分可靠，当线程获得锁后，处理时间过长导致锁超时，就失效了锁的作用。

zookeeper锁：

优点：不依靠超时时间释放锁；可靠性高；系统要求高可靠性时，建议采用zookeeper锁。

缺点：性能比不上缓存锁，因为要频繁的创建节点删除节点。

## 负载均衡

轮询（Round Robin）

轮询算法把每个请求轮流发送到每个服务器上。

加权轮询（Weighted Round Robbin）

最少连接（least Connections）最少连接算法就是将请求发送给当前最少连接数的服务器上。

加权最少连接（Weighted Least Connection）

随机算法（Random）

源地址哈希法 (IP Hash) 可以保证同一 IP 的客户端的请求会转发到同一台服务器上，用来实现会话粘滞（Sticky Session）

**实现手段：**

**HTTP 重定向，**

HTTP 重定向负载均衡服务器使用某种负载均衡算法计算得到服务器的 IP 地址之后，将该地址写入 HTTP 重定向报文中，状态码为 302。客户端收到重定向报文之后，需要重新向服务器发起请求。

**DNS 域名解析**

在 DNS 解析域名的同时使用负载均衡算法计算服务器 IP 地址。

优点：

DNS 能够根据地理位置进行域名解析，返回离用户最近的服务器 IP 地址。

缺点：

由于 DNS 具有多级结构，每一级的域名记录都可能被缓存，当下线一台服务器需要修改 DNS 记录时，需要过很长一段时间才能生效。

反向代理服务器

反向代理服务器位于源服务器前面，用户的请求需要先经过反向代理服务器才能到达源服务器。反向代理可以用来进行缓存、日志记录等，同时也可以用来做为负载均衡服务器。

在这种负载均衡转发方式下，客户端不直接请求源服务器，因此源服务器不需要外部 IP 地址，而反向代理需要配置内部和外部两套 IP 地址。

优点：

与其它功能集成在一起，部署简单。

缺点：

所有请求和响应都需要经过反向代理服务器，它可能会成为性能瓶颈。

4. 网络层

在操作系统内核进程获取网络数据包，根据负载均衡算法计算源服务器的 IP 地址，并修改请求数据包的目的 IP 地址，最后进行转发。

源服务器返回的响应也需要经过负载均衡服务器，通常是让负载均衡服务器同时作为集群的网关服务器来实现。

优点：

在内核进程中进行处理，性能比较高。

缺点：

和反向代理一样，所有的请求和响应都经过负载均衡服务器，会成为性能瓶颈。

5. 链路层

在链路层根据负载均衡算法计算源服务器的 MAC 地址，并修改请求数据包的目的 MAC 地址，并进行转发。

通过配置源服务器的虚拟 IP 地址和负载均衡服务器的 IP 地址一致，从而不需要修改 IP 地址就可以进行转发。也正因为 IP 地址一样，所以源服务器的响应不需要转发回负载均衡服务器，可以直接转发给客户端，避免了负载均衡服务器的成为瓶颈。

这是一种三角传输模式，被称为直接路由。对于提供下载和视频服务的网站来说，直接路由避免了大量的网络传输数据经过负载均衡服务器。

这是目前大型网站使用最广负载均衡转发方式，在 Linux 平台可以使用的负载均衡服务器为 LVS（Linux Virtual Server）。

## 缓存

缓存位置

浏览器

当 HTTP 响应允许进行缓存时，浏览器会将 HTML、CSS、JavaScript、图片等静态资源进行缓存。

ISP

网络服务提供商（ISP）是网络访问的第一跳，通过将数据缓存在 ISP 中能够大大提高用户的访问速度。

反向代理

反向代理位于服务器之前，请求与响应都需要经过反向代理。通过将数据缓存在反向代理，在用户请求反向代理时就可以直接使用缓存进行响应。

本地缓存

使用 Guava Cache 将数据缓存在服务器本地内存中，服务器代码可以直接读取本地内存中的缓存，速度非常快。

分布式缓存

使用 Redis、Memcache 等分布式缓存将数据缓存在分布式缓存系统中。

相对于本地缓存来说，分布式缓存单独部署，可以根据需求分配硬件资源。不仅如此，服务器集群都可以访问分布式缓存，而本地缓存需要在服务器集群之间进行同步，实现难度和性能开销上都非常大。

数据库缓存

MySQL 等数据库管理系统具有自己的查询缓存机制来提高查询效率。

缓存雪崩

指的是由于数据没有被加载到缓存中，或者缓存数据在同一时间大面积失效（过期），又或者缓存服务器宕机，导致大量的请求都到达数据库。

在有缓存的系统中，系统非常依赖于缓存，缓存分担了很大一部分的数据请求。当发生缓存雪崩时，数据库无法处理这么大的请求，导致数据库崩溃。

解决方案：

为了防止缓存在同一时间大面积过期导致的缓存雪崩，可以通过观察用户行为，合理设置缓存过期时间来实现；

为了防止缓存服务器宕机出现的缓存雪崩，可以使用分布式缓存，分布式缓存中每一个节点只缓存部分的数据，当某个节点宕机时可以保证其它节点的缓存仍然可用。

也可以进行缓存预热，避免在系统刚启动不久由于还未将大量数据进行缓存而导致缓存雪崩。

缓存一致性

缓存一致性要求数据更新的同时缓存数据也能够实时更新。

解决方案：

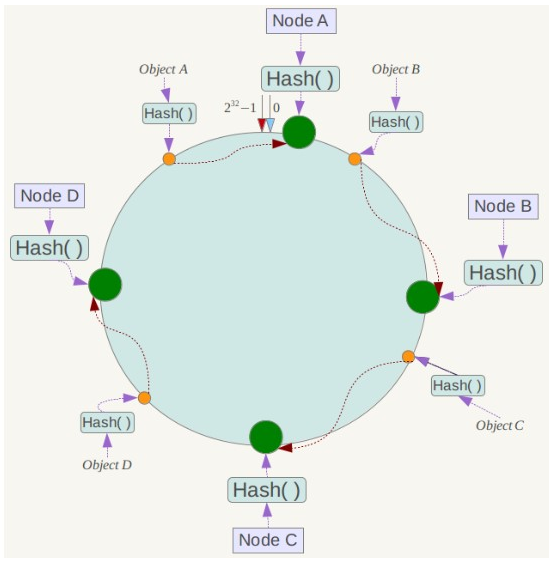
在数据更新的同时立即去更新缓存；

在读缓存之前先判断缓存是否是最新的，如果不是最新的先进行更新。

要保证缓存一致性需要付出很大的代价，缓存数据最好是那些对一致性要求不高的数据，允许缓存数据存在一些脏数据。

## 一致性哈希

将哈希空间 [0, 2n-1] 看成一个哈希环，每个服务器节点都配置到哈希环上。每个数据对象通过哈希取模得到哈希值之后，存放到哈希环中顺时针方向第一个大于等于该哈希值的节点上。



一致性哈希在增加或者删除节点时只会影响到哈希环中相邻的节点，例如下图中新增节点 X，只需要将它前一个节点 C 上的数据重新进行分布即可，对于节点 A、B、D 都没有影响。

上面描述的一致性哈希存在数据分布不均匀的问题，节点存储的数据量有可能会存在很大的不同。数据不均匀主要是因为节点在哈希环上分布的不均匀，这种情况在节点数量很少的情况下尤其明显。

解决方式是通过增加虚拟节点，然后将虚拟节点映射到真实节点上。虚拟节点的数量比真实节点来得多，那么虚拟节点在哈希环上分布的均匀性就会比原来的真实节点好，从而使得数据分布也更加均匀。

## 分布式最终一致性

最终一致性，人们首先想到的应该是2PC解决方案。

两阶段提交需要有一个协调者，来协调两个操作之间的操作流程。当参与方为更多时，其逻辑其实就比较复杂了。

而参与者需要实现两阶段提交协议。Pre commit阶段需要锁住相关资源，commit或rollback时分别进行实际提交或释放资源。

## 消息队列：

发送端的可靠性，发送端的问题是可能发不出去

发送端完成操作后一定能将消息成功发送到消息队列中。

实现方法：

在本地数据库或Redis中建一张消息表，将消息数据与业务数据保存在同一数据库实例里，这样就可以利用本地数据库的事务机制。事务提交成功后，将消息表中的消息转移到消息队列中，若转移消息成功则删除消息表中的数据，否则继续重传，重传两次失败后报警。

接收端的可靠性，收不到消息，同一消息多次消费。Redis加锁

接收端能够从消息队列成功消费一次消息。

两种实现方法：

保证接收端处理消息的业务逻辑具有幂等性：只要具有幂等性，那么消费多少次消息，最后处理的结果都是一样的。

保证消息具有唯一编号，并使用一张日志表来记录已经消费的消息编号。

## LVS KeepAlived

IPVS（IP virtual Server）虚拟服务器是负载均衡体系的基本架构，它分两层结构：转发器（Director）和真实服务器。负载均衡类型 （lb\_kind）一般分直接路由模式 DR 、网络地址转换模式 NAT 以及隧道模式 TUN 三种。

Keepalived 是运行在 lvs 之上，它的主要功能是实现真实机的故障隔离及负载均衡器间的失 败切换 FailOver.lvs 结合 keepalived，就实现了 3 层、4 层、5/7 层交换的功能，

1、 keepalived 是 lvs 的扩展项目，因此它们之间具备良好的兼容性。这点应该是 keepalived 部署比其他类似工具能更简洁的原因吧！ 2、 通过对服务器池对象的健康检查，实现对失效机器/服务的故障隔离。 3、 负载均衡器之间的失败切换 failover，是通过 VRRPv2（Virtual Router Redundancy Protocol） stack 实现的。

## TCP粘包拆包

## IP分片

[链路层](https://baike.sogou.com/lemma/ShowInnerLink.htm?lemmaId=16970&ss_c=ssc.citiao.link)具有[最大传输单元](https://baike.sogou.com/lemma/ShowInnerLink.htm?lemmaId=582088&ss_c=ssc.citiao.link)MTU这个特性，它限制了数据帧的最大长度，不同的网络类型都有一个上限值。以太网的MTU是1500，你可以用 netstat -i 命令查看这个值。如果IP层有数据包要传，而且数据包的长度超过了MTU，那么IP层就要对数据包进行分片（fragmentation）操作，使每一片的长度都小于或等于MTU。我们假设要传输一个UDP数据包，以太网的MTU为1500字节，一般IP首部为20字节，UDP首部为8字节，数据的净荷（payload）部分预留是1500-20-8=1472字节。如果数据部分大于1472字节，就会出现分片现象。

## UDP与TCP分段

MSS是TCP里的一个概念（首部的选项字段中）。MSS是TCP数据包每次能够传输的最大数据分段，TCP报文段的长度大于MSS时，要进行分段传输。

TCP分段的原因是MSS，IP分片的原因是MTU，由于一直有MSS<=MTU，很明显，分段后的每一段TCP报文段再加上IP首部后的长度不可能超过MTU，因此也就不需要在网络层进行IP分片了。因此TCP报文段很少会发生IP分片的情况。

UDP不会分段，就由IP来分。TCP会分段，当然就不用IP来分了！

## Netty

## MD5,SHA1，Base64

SHA-1基于MD5

Base64编码要求把3个8位字节（3\*8=24）转化为4个6位的字节（4\*6=24），之后在6位的前面补两个0，形成8位一个字节的形式。

## Thrift

## 分布式集群脑裂问题

集群的脑裂通常是发生在集群中部分节点之间不可达而引起的（或者因为节点请求压力较大，导致其他节点与该节点的心跳检测不可用）。当上述情况发生时，不同分裂的小集群会自主的选择出master节点，造成原本的集群会同时存在多个master节点。

1）添加冗余的心跳线，例如：双线条线（心跳线也HA），尽量减少“裂脑”发生几率；

　　　　2）启用磁盘锁。正在服务一方锁住共享磁盘，“裂脑”发生时，让对方完全“抢不走”共享磁盘资源。但使用锁磁盘也会有一个不小的问题，如果占用共享盘的一方不主动“解锁”，另一方就永远得不到共享磁盘。现实中假如服务节点突然死机或崩溃，就不可能执行解锁命令。后备节点也就接管不了共享资源和应用服务。于是有人在HA中设计了“智能”锁。即：正在服务的一方只在发现心跳线全部断开（察觉不到对端）时才启用磁盘锁。平时就不上锁了。

　　　　3）设置仲裁机制。例如设置参考IP（如网关IP），当心跳线完全断开时，2个节点都各自ping一下参考IP，不通则表明断点就出在本端。不仅“心跳”、还兼对外“服务”的本端网络链路断了，即使启动（或继续）应用服务也没有用了，那就主动放弃竞争，让能够ping通参考IP的一端去起服务。更保险一些，ping不通参考IP的一方干脆就自我重启，以彻底释放有可能还占用着的那些共享资源。

Zookeeper假死和脑裂问题：

是什么原因导致这样情况的出现呢？

主要原因是Zookeeper集群和Zookeeper client判断超时并不能做到完全同步，也就是说可能一前一后，如果是集群先于client发现那就会出现上面的情况。同时，在发现并切换后通知各个客户端也有先后快慢。一般出现这种情况的几率很小，需要master与Zookeeper集群网络断开但是与其他集群角色之间的网络没有问题，还要满足上面那些情况，但是一旦出现就会引起很严重的后果，数据不一致。

如何避免？

在slaver切换的时候不在检查到老的master出现问题后马上切换，而是在休眠一段足够的时间，确保老的master已经获知变更并且做了相关的shutdown清理工作了然后再注册成为master就能避免这类问题了，这个休眠时间一般定义为与Zookeeper定义的超时时间就够了，但是这段时间内系统不可用了。

## 网卡多队列技术与RSS功能

多队列网卡是一种技术，最初是用来解决网络IO QoS （quality of service）问题的，后来随着网络IO的带宽的不断提升，单核CPU不能完全处满足网卡的需求，通过多队列网卡驱动的支持，将各个队列通过中断绑定到不同的核上，以满足网卡的需求。

## 网卡中断

Linux内核在性能方面已经经历了很长一段时间的考验，尤其是2.6/3.x内核。然而，在高IO，尤其是网络方面的情况下，对中断的处理可能成为问题。我们已经 在拥有一个或多个饱和1Gbps网卡的高性能系统上发现过这个问题，近来在有许多小包并发（大约10000packets/second）超载的虚拟机上 也发现了这个问题。

原因很清楚：在最简单的模式中，内核通过硬件中断的方式来处理每个来自于网卡的包。但是随着 数据包速率的增长，带来的中断渐渐超过了单个cpu可处理的范围。单cpu概念很重要，系统管理员对此往往认识不足。在一个普通的4-16核的系统中，因 为整体cpu的使用率在6-25%左右并且系统看上去很正常，所以一个过载的内核很难被发现，。但是系统将运行很慢，并且会在没有告警，没有dmesg日 志，没有明显征兆的情况下严重丢包。

## Dubbo

如果你要开发分布式程序，你也可以直接基于 HTTP 接口进行通信，但是为什么要用 Dubbo呢？

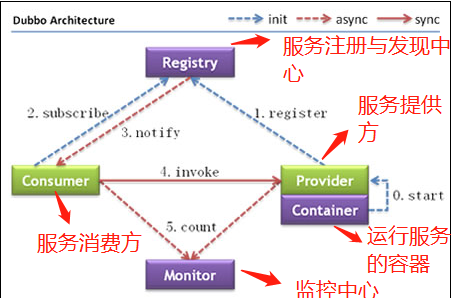
我觉得主要可以从 Dubbo 提供的下面四点特性来说为什么要用 Dubbo：

负载均衡——同一个服务部署在不同的机器时该调用那一台机器上的服务

服务调用链路生成——随着系统的发展，服务越来越多，服务间依赖关系变得错踪复杂，甚至分不清哪个应用要在哪个应用之前启动，架构师都不能完整的描述应用的架构关系。Dubbo 可以为我们解决服务之间互相是如何调用的。

服务访问压力以及时长统计、资源调度和治理——基于访问压力实时管理集群容量，提高集群利用率。

服务降级——某个服务挂掉之后调用备用服务



zookeeper宕机与dubbo直连的情况

zookeeper宕机与dubbo直连的情况在面试中可能会被经常问到，所以要引起重视。

在实际生产中，假如zookeeper注册中心宕掉，一段时间内服务消费方还是能够调用提供方的服务的，实际上它使用的本地缓存进行通讯，这只是dubbo健壮性的一种提现。

dubbo的健壮性表现：

监控中心宕掉不影响使用，只是丢失部分采样数据

数据库宕掉后，注册中心仍能通过缓存提供服务列表查询，但不能注册新服务

注册中心对等集群，任意一台宕掉后，将自动切换到另一台

注册中心全部宕掉后，服务提供者和服务消费者仍能通过本地缓存通讯

服务提供者无状态，任意一台宕掉后，不影响使用

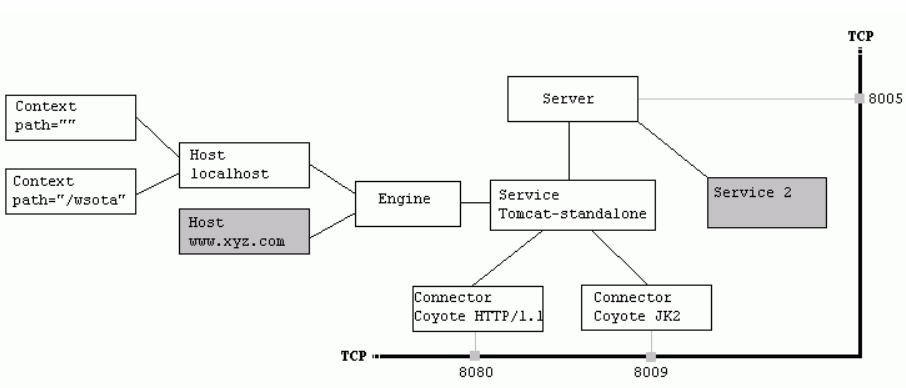
服务提供者全部宕掉后，服务消费者应用将无法使用，并无限次重连等待服务提供者恢复

我们前面提到过：注册中心负责服务地址的注册与查找，相当于目录服务，服务提供者和消费者只在启动时与注册中心交互，注册中心不转发请求，压力较小。所以，我们可以完全可以绕过注册中心——采用 dubbo 直连 ，即在服务消费方配置服务提供方的位置信息。

## Eureka

## Tomcat

Tomcat端口号：8080，http；8443，转发到https；shutdown端口:8005；ajp端口：8009



Tomcat处理Http请求的过程：

假设来自客户的请求为：   
<http://localhost:8080/wsota/wsota_index.jsp>   
1. 请求被发送到本机端口8080，被在那里侦听的Coyote HTTP/1.1 Connector获得   
2. Connector把该请求交给它所在的Service的Engine来处理，并等待来自Engine的回应   
3. Engine获得请求localhost/wsota/wsota\_index.jsp，匹配它所拥有的所有虚拟主机Host   
4. Engine匹配到名为localhost的Host（即使匹配不到也把请求交给该Host处理，因为该Host被定义为该Engine的默认主机）   
5. localhost Host获得请求/wsota/wsota\_index.jsp，匹配它所拥有的所有Context   
6. Host匹配到路径为/wsota的Context（如果匹配不到就把该请求交给路径名为”“的Context去处理）   
7. path=”/wsota”的Context获得请求/wsota\_index.jsp，在它的mapping table中寻找对应的servlet   
8. Context匹配到URL PATTERN为\*.jsp的servlet，对应于JspServlet类   
9. 构造HttpServletRequest对象和HttpServletResponse对象，作为参数调用JspServlet的doGet或doPost方法   
10. Context把执行完了之后的HttpServletResponse对象返回给Host   
11. Host把HttpServletResponse对象返回给Engine   
12. Engine把HttpServletResponse对象返回给Connector   
13. Connector把HttpServletResponse对象返回给客户browser

Ehcache，GuavaCache，Memcache，Redis

它具有上文 JVM 缓存不具有的功能，如自动清除数据、多种清除算法、清除回调等。

Guava 的 Cache 就非常适合，我利用了它的 N 个时间内不写入数据时缓存就清空的特点，在每次读取数据时判断异常信息是否大于 X 即可。

Redis高性能之谜

## Spring启动过程

1:找到原材料

2:把原材料装进容器里

2、bean的实例化和依赖注入

## Nginx

nginx在启动后，在unix系统中会以daemon的方式在后台运行，后台进程包含一个master进程和多个worker进程。

master进程主要用来管理worker进程，包含：接收来自外界的信号，向各worker进程发送信号，监控worker进程的运行状态，当worker进程退出后(异常情况下)，会自动重新启动新的worker进程。

　　worker进程则是处理基本的网络事件。多个worker进程之间是对等的，他们同等竞争来自客户端的请求，各进程互相之间是独立的。一个请求，只可能在一个worker进程中处理，一个worker进程，不可能处理其它进程的请求。

每个worker进程都是从master进程fork过来。在master进程里面，先建立好需要listen的socket之后，然后再fork出多个worker进程，这样每个worker进程都可以去accept这个socket(当然不是同一个socket，只是每个进程的这个socket会监控在同一个ip地址与端口，这个在网络协议里面是允许的)。一般来说，当一个连接进来后，所有在accept在这个socket上面的进程，都会收到通知，而只有一个进程可以accept这个连接，其它的则accept失败。

**相对于线程，采用进程的优点**

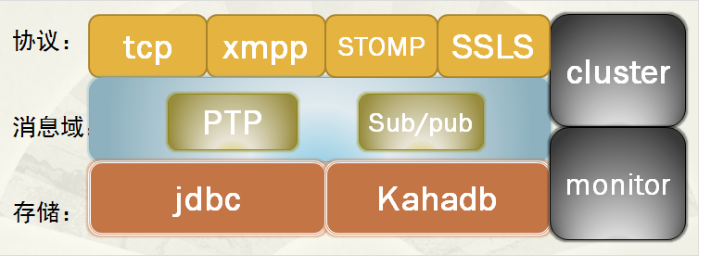
进程之间不共享资源，不需要加锁，所以省掉了锁带来的开销。

　　采用独立的进程，可以让互相之间不会影响，一个进程退出后，其它进程还在工作，服务不会中断，master进程则很快重新启动新的worker进程。

编程上更加容易。

目前Nginx面临的一个大问题是如何解决磁盘I/O的阻塞问题。当前版本中，如果没有足够的存储性能来满足worker的磁盘操作，worker会阻塞在磁盘的读取和写入上。

## ActiveMq



传输协议：消息之间的传递，无疑需要协议进行沟通，启动一个ActiveMQ打开了一个监听端口， ActiveMQ提供了广泛的连接模式，其中主要包括SSL、STOMP

、XMPP；ActiveMQ默认的使用的协议是openWire，端口号：61616;

消息域：ActiveMQ主要包含Point-to-Point (点对点),Publish/Subscribe Model (发布/订阅者)，其中在Publich/Subscribe 模式下又有Nondurable subscription和 durable subscription (持久化订阅)2种消息处理方式

消息存储：在消息传递过程中，部分重要的消息可能需要存储到数据库或文件系统中，当中介崩溃时，信息不回丢失

Cluster (集群): 最常见到 集群方式包括network of brokers和Master Slave；

Monitor (监控) :ActiveMQ一般由jmx来进行监控，这部分的内容我们以后讨论；