# 1 计算机网络

## TCP/IP协议栈



* **应用层**：为特定应用程序提供数据传输服务，数据单位为报文。主要功能是：为应用软件提供了很多服务，比如文件服务器、数据库服务、电子邮件与其他网络软件服务。
* **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。表示层（presentation），帮我们解决不同系统之间的通信语法问题。
* **会话层**：现在我们已经保证给正确的计算机，发送正确的封装过后的信息了。但是用户级别的体验好不好？难道我每次都要调用TCP去打包，然后调用IP协议去找路由，自己去发？当然不行，所以我们要建立一个自动收发包，自动寻址的功能。

于是，发明了会话层。会话层的作用就是建立和管理应用程序之间的通信。

* **传输层**：传输层的由来：网络层的ip帮我们区分子网，以太网层的mac帮我们找到主机，然后大家使用的都是应用程序，你的电脑上可能同时开启qq，暴风影音，等多个应用程序，

那么我们通过ip和mac找到了一台特定的主机，如何标识这台主机上的应用程序，答案就是端口，端口即应用程序与网卡关联的编号。传输层功能：建立端口到端口的通信。

* **网络层**：传输层只是解决了打包的问题。但是如果我有多台计算机，怎么找到我要发的那台？或者，A要给F发信息，中间要经过B，C，D，E，但是中间还有好多节点如K.J.Z.Y。我怎么选择最佳路径？这就是路由要做的事。

于是，发明了网络层。即路由器，交换价那些具有寻址功能的设备所实现的功能。这一层定义的是IP地址，通过IP地址寻址。所以产生了IP协议。

* **数据链路层**：数据链路层就是来对电信号来做分组的。一组电信号称之为一个数据包，或者叫做一个“帧”，每一数据帧分成：报头head和数据data两部分，报头包括发送者、接受者、数据类型，数据data包含数据的具体内容。保存的是Mac地址。
* **物理层**：主要定义物理设备标准，如网线的接口类型、光纤的接口类型、各种传输介质的传输速率等。它的主要作用是传输比特流(就是由1、0转化为电流强弱来进行传输，到达目的地后在转化为1、0，也就是我们常说的数模转换与模数转换)。这一层的数据叫做比特。

<https://blog.csdn.net/taotongning/article/details/81352985>

<https://www.cnblogs.com/evan51/p/7994109.html>

## Ip地址、子网掩码和ARP地址协议

（1）IP地址 = 网络地址 + 主机地址（前半段和后半段）

（2）网络地址 = IP地址 & 子网掩码（按位与）

（3）主机地址 = IP地址-网络地址

子网掩码用来表示IP地址中的那几个位置是网络地址。

ARP地址协议

1. 首先，每个主机都会在自己的ARP缓冲区中建立一个ARP列表，以表示IP地址和MAC地址（MAC地址在每个网卡出场的时候就有一个全球唯一的MAC地址）之间的对应关系
2. 当源主机要发送数据时，首先检查ARP列表中是否有对应IP地址的目的主机的MAC地址，如果有，则直接发送数据，如果没有，就向本网段的所有主机发送ARP请求分组，该数据包包括的内容有：**源主机** **IP地址，源主机MAC地址，目的主机的IP 地址**。
3. 当本网络的所有主机收到该ARP请求分组时，首先检查数据包中的IP地址是否是自己的IP地址，如果不是，则忽略该数据包，如果是，则首先从请求分组中取出源主机的IP和MAC地址写入到ARP列表中，如果已经存在，则覆盖，然后将自己的MAC地址写入ARP响应分组中，告诉源主机它想要找的MAC地址。
4. 源主机收到ARP响应分组后。将目的主机的IP和MAC地址写入ARP列表，并利用此信息发送数据。如果源主机一直没有收到ARP响应分组，表示ARP查询失败。

https://blog.csdn.net/sinat\_21591675/article/details/82844611

## TCP和UDP区别

* 1、传输控制协议（Transmission Control Protocol）TCP面向连接（如打电话要先拨号建立连接），可靠的数据传输服务，数据单位为报文段; 用户数据报协议UDP（User Datagram Protocol）是无连接的，即发送数据之前不需要建立连接，数据单位为用户数据报。
* 2、TCP提供可靠的服务。也就是说，通过TCP连接传送的数据，无差错，不丢失，不重复，且按序到达;UDP尽最大努力交付，即不保证可靠交付
* 3、UDP具有较好的实时性，工作效率比TCP高，更适用于对高速传输和实时性有较高要求的通信或广播通信。
* 4、每一条TCP连接只能是点到点的，即TCP不支持组播或者广播传输模式。UDP支持一对一，一对多，多对一和多对多的交互通信。
* 5、TCP对系统资源要求较多，UDP对系统资源要求较少。

## TCP与UDP数据格式

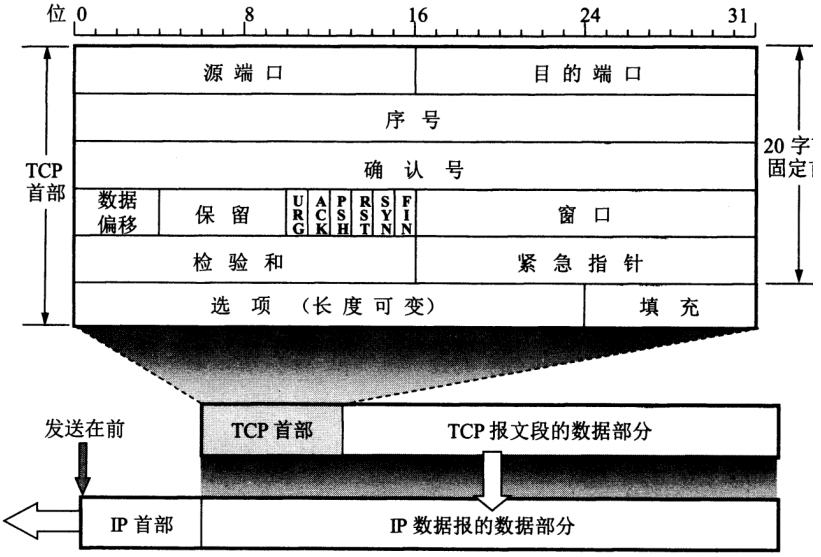


图1 TCP报文段的首部格式

* 序号：用于对字节流进行编号，例如序号为 301，表示第一个字节的编号为 301，如果携带的数据长度为 100字节，那么下一个报文段的序号应为 401。
* 确认号 ：期望收到的下一个报文段的序号。例如 B 正确收到 A 发送来的一个报文段，序号为501，携带的数据长度为 200 字节，因此 B 期望下一个报文段的序号为 701，B 发送给 A 的确认报文段中确认号就为 701。
* 数据偏移 ：指的是数据部分距离报文段起始处的偏移量，实际上指的是首部的长度。
* 确认ACK ：当 ACK=1 时确认号字段有效，否则无效。TCP 规定，在连接建立后所有传送的报文段都必须把ACK 置 1。
* 同步SYN ：在连接建立时用来同步序号。当 SYN=1，ACK=0 时表示这是一个连接请求报文段。若对方同意建立连接，则响应报文中 SYN=1，ACK=1。
* 终止FIN ：用来释放一个连接，当 FIN=1 时，表示此报文段的发送方的数据已发送完毕，并要求释放连接。
* 窗口 ：窗口值作为接收方让发送方设置其发送窗口的依据。之所以要有这个限制，是因为接收方的数据缓存空间是有限的。

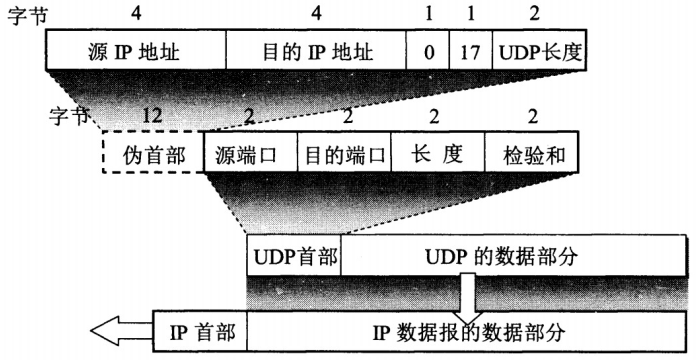


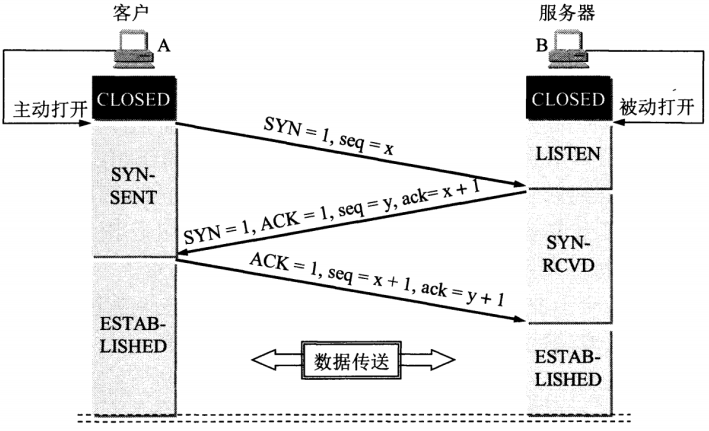
图2 UDP用户数据报的首部和伪首部

首部字段只有 8 个字节，包括源端口、目的端口、长度、检验和。12 字节的伪首部是为了计算检验和临时添加的。

## TCP三次握手与四次挥手

（1）三次握手

三次握手流程



假设 A 为客户端，B 为服务器端。

* 首先 B 处于 LISTEN（监听）状态，等待客户的连接请求。
* A 向 B 发送连接请求报文，SYN=1，ACK=0，选择一个初始的序号 x。
* B 收到连接请求报文，如果同意建立连接，则向 A 发送连接确认报文，SYN=1，ACK=1，确认号为 x+1，同时也选择一个初始的序号 y。
* A 收到 B 的连接确认报文后，还要向 B 发出确认，确认号为 y+1，序号为 x+1。
* B 收到 A 的确认后，连接建立。

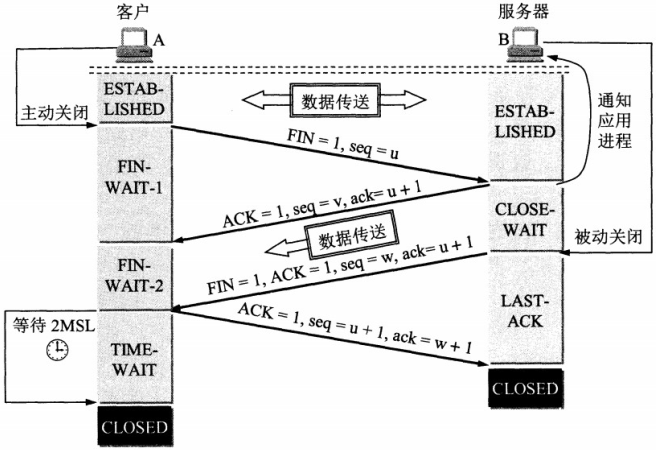
三次握手的原因

第三次握手是为了防止失效的连接请求到达服务器，让服务器错误打开连接。

客户端发送的连接请求如果在网络中滞留，那么就会隔很长一段时间才能收到服务器端发回的连接确认。客户端等待一个超时重传时间之后，就会重新请求连接。但是这个滞留的连接请求最后还是会到达服务器，如果不进行三次握手，那么服务器就会打开两个连接。如果有第三次握手，客户端会忽略服务器之后发送的对滞留连接请求的连接确认，不进行第三次握手，因此就不会再次打开连接。

（2）TCP 的四次挥手

四次挥手过程



以下描述不讨论序号和确认号并且不讨论 ACK，因为 ACK 在连接建立之后都为 1。

* A 发送连接释放报文，FIN=1。
* B 收到之后发出确认，此时 TCP 属于半关闭状态，B 能向 A 发送数据但是 A 不能向 B 发送数据。
* 当 B 不再需要连接时，发送连接释放报文，FIN=1。
* A 收到后发出确认，进入 TIME-WAIT 状态，等待 2 MSL（最大报文存活时间）后释放连接。
* B 收到 A 的确认后释放连接。

四次挥手的原因

客户端发送了FIN连接释放报文之后，服务器收到了这个报文，就进入了CLOSE-WAIT 状态。这个状态是为了让服务器端发送还未传送完毕的数据，传送完毕之后，服务器会发送 FIN 连接释放报文。

TIME\_WAIT客户端接收到服务器端的 FIN 报文后进入此状态，此时并不是直接进入 CLOSED 状态，还需要等待一个时间计时器设置的时间 2MSL（最大生存时间）。这么做有两个理由：

* 确保最后一个确认报文能够到达。如果 B 没收到 A 发送来的确认报文，那么就会重新发送连接释放请求报文，A 等待一段时间就是为了处理这种情况的发生。
* 等待一段时间是为了让本连接持续时间内所产生的所有报文都从网络中消失，使得下一个新的连接不会出现旧的连接请求报文。

## TCP 粘包

在socket网络程序中，TCP和UDP分别是面向连接和非面向连接的。因此TCP的socket编程，收发两端（客户端和服务器端）都要有成对的socket，因此，发送端为了将多个发往接收端的包，更有效的发到对方，使用了优化方法（Nagle算法），将多次间隔较小、数据量小的数据，合并成一个大的数据块，然后进行封包。这样，接收端，就难于分辨出来了，必须提供科学的拆包机制。

对于UDP，不会使用块的合并优化算法，这样，实际上目前认为，是由于UDP支持的是一对多的模式，所以接收端的skbuff(套接字缓冲区）采用了链式结构来记录每一个到达的UDP包，在每个UDP包中就有了消息头（消息来源地址，端口等信息），这样，对于接收端来说，就容易进行区分处理了。所以UDP不会出现粘包问题。

TCP粘包是指发送方发送的若干包数据到接收方接收时粘成一包，从接收缓冲区看，后一包数据的头紧接着前一包数据的尾；

1. 发送方原因

当连续发送数据时，由于tcp协议使用了nagle算法，会将较小的内容拼接成大的内容，一次性发送到服务器端，因此造成粘包。

1. 接收方原因

当发送内容较大时，由于服务器端的recv（buffer\_size）方法中的buffer\_size较小，不能一次性完全接收全部内容，因此在下一次请求到达时，接收的内容依然是上一次没有完全接收完的内容，因此造成粘包现象。

也就是说：接收方不知道该接收多大的数据才算接收完毕，造成粘包。

1. 解决方法
2. 发送方：对于第一种粘包产生方式可以在两次send（）直接使用recv（）来阻止连续发送的情况发生。代码就不用展示了。
3. 接收方：由于产生粘包的原因是接收方的无边界接收，因此发送端可以在发送数据之前向接收端告知发送内容的大小即可。

* 什么时候需要处理粘包现象？

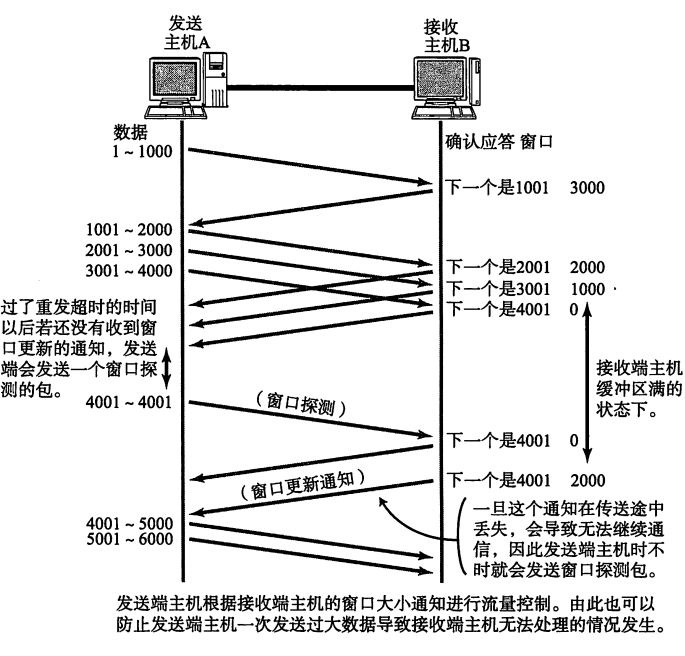
1. 如果发送方发送的多组数据本来就是同一块数据的不同部分，比如说一个文件被分成多个部分发送，这时当然不需要处理粘包现象。

2. 如果多个分组毫不相干，甚至是并列关系，那么这个时候就一定要处理粘包现象了。

## TCP流量控制与拥塞控制

1流量控制

1. 概念：接收端处理数据的速度是有限的，如果发送方的速度太快，就会把缓冲区打满。这时如果继续发送数据，就会导致丢包等一系列连锁反应。所以TCP支持根据接收端能力来决定发送端的发送速度。这个机制叫做流控制。
2. 机理（滑动窗口）
   * + 接收端将自己可以接收的缓冲区大小放入 TCP 首部中16位的“窗口大小”字段，通过ACK端通知发送端；窗口大小字段越大，说明网络的吞吐量越高；（实际窗口大小是窗口字段的值左移M（窗口扩大因子）位）
     + 接收端一旦发现自己的缓冲区快满了，就会将窗口大小设置成一个更小的值通知给发送端；发送端接受到这个窗口之后，就会减慢自己的发送速度；
     + 如果接收端缓冲区满了，就会将窗口置为0，这时发送方不再发送数据，但是需要定期发送一个窗口探测数据段, 使接收端把窗口大小告诉发送端。



* + - 当某个ACK报文丢失了，就会出现A等待B确认，并且B等待A发送数据的死锁状态。为了解决这种问题，TCP引入了持续计时器（Persistence timer），当A收到rwnd=0时，就启用该计时器，时间到了则发送一个1字节的探测报文，询问B是很忙还是上个ACK丢失了，然后B回应自身的接收窗口大小，返回仍为0（A重设持续计时器继续等待）或者会重发rwnd=x。

2拥塞控制

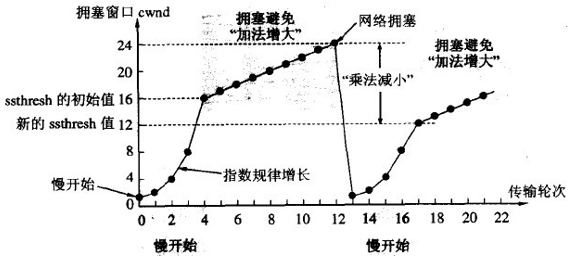
1. 概念

对资源的需求超过了可用的资源。若网络中许多资源同时供应不足，网络的性能就要明显变坏，整个网络的吞吐量随之负荷的增大而下降。

1. 方法
2. 慢开始
   * + 发送方维持一个叫做拥塞窗口cwnd（congestion window）的状态变量。拥塞窗口的大小取决于网络的拥塞程度，并且动态地在变化。发送方让自己的发送窗口等于拥塞窗口，另外考虑到接受方的接收能力，发送窗口可能小于拥塞窗口。慢开始算的思路就是：不要一开始就发送大量的数据，先探测一下网络的拥塞程度，也就是说由小到大逐渐增加拥塞窗口的大小。初始值为1，每次收到一个ACK应答，拥塞窗口加倍。
     + 为了防止cwnd增长过大引起网络拥塞，还需设置一个慢开始门限ssthresh状态变量。ssthresh的用法如下：

|  |
| --- |
| 当cwnd<ssthresh时，使用慢开始算法。  当cwnd>ssthresh时，改用拥塞避免算法。  当cwnd=ssthresh时，慢开始与拥塞避免算法任意。 |

* + - 拥塞避免算法让拥塞窗口缓慢增长，即每经过一个往返时间RTT就把发送方的拥塞窗口cwnd加1，而不是加倍。这样拥塞窗口按线性规律缓慢增长。
    - 无论是在慢开始阶段还是在拥塞避免阶段，只要发送方判断网络出现拥塞（其根据就是没有收到确认，虽然没有收到确认可能是其他原因的分组丢失，但是因为无法判定，所以都当做拥塞来处理），就把慢开始门限设置为出现拥塞时的发送窗口大小的一半。然后把拥塞窗口设置为1，执行慢开始算法。如下图：



A . 超时重传机制

TCP中设计了超时重传机制。该机制规定当发送方A向B发送数据包P1时，开启时长为RTO（Retransmission Timeout）的重传定时器，如果A在RTO内未收到B对P1的确认报文，则认为P1在网络中丢失，此时重新发送P1。由此，引出RTO大小的设定问题。

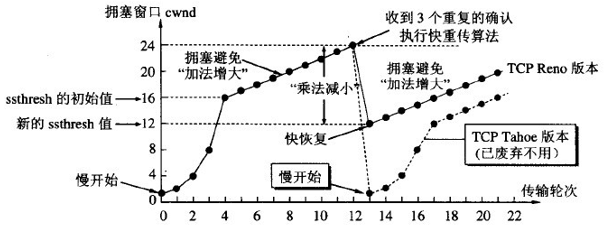
决定报文是否有必要重传的主要机制是重传计时器（retransmission timer），它的主要功能是维护重传超时（RTO）值。当报文使用TCP传输时，重传计时器启动，收到ACK时计时器停止。报文发送至接收到ACK的时间称为往返时间（RTT）。对若干次时间取平均值，该值用于确定最终RTO值。在最终RTO值确定之前，确定每一次报文传输是否有丢包发生使用重传计时器，下图说明了TCP重传过程。

　当报文发送之后，但接收方尚未发送TCP ACK报文，发送方假设源报文丢失并将其重传。重传之后，RTO值加倍；如果在2倍RTO值到达之前还是没有收到ACK报文，就再次重传。如果仍然没有收到ACK，那么RTO值再次加倍。如此持续下去，每次重传RTO都翻倍，直到收到ACK报文或发送方达到配置的最大重传次数。

　　最大重传次数取决于发送操作系统的配置值。默认情况下，Windows主机默认重传5次。大多数Linux系统默认最大15次。两种操作系统都可配置。

1. 快重传和快恢复
   * + 快重传要求接收方在收到一个失序的报文段后就立即发出重复确认（为的是使发送方及早知道有报文段没有到达对方）而不要等到自己发送数据时捎带确认。快重传算法规定，发送方只要一连收到三个重复确认（为什么是3个ACK？）就应当立即重传对方尚未收到的报文段，而不必继续等待设置的重传计时器时间到期。
     + 快重传配合使用的还有快恢复算法，有以下两个要点：

|  |
| --- |
| * 当发送方连续收到三个重复确认时，就执行“乘法减小”算法，把ssthresh门限减半。但是接下去并不执行慢开始算法。 * 考虑到如果网络出现拥塞的话就不会收到好几个重复的确认，所以发送方现在认为网络可能没有出现拥塞。所以此时不执行慢开始算法，而是将cwnd设置为ssthresh的大小，然后执行拥塞避免算法。 |



3流量控制与拥塞控制的区别

1. 拥塞控制就是防止过多的数据注入网络中，这样可以使网络中的路由器或链路不致过载。拥塞控制是一个全局性的过程，和流量控制不同，流量控制指点对点通信量的控制。
2. 所谓流量控制就是让发送发送速率不要过快，让接收方来得及接收。利用滑动窗口机制就可以实施流量控制。原理这就是运用TCP报文段中的窗口大小字段来控制，发送方的发送窗口不可以大于接收方发回的窗口大小。

## 为何快速重传是选择3次ACK？

主要的考虑还是要区分包的丢失是由于链路故障还是乱序等其他因素引发。两次duplicated ACK时很可能是乱序造成的！三次duplicated ACK时很可能是丢包造成的！四次duplicated ACK更更更可能是丢包造成的！但是这样的响应策略太慢。丢包肯定会造成三次duplicated ACK!综上是选择收到三个重复确认时窗口减半效果最好，这是实践经验

## TCP可靠传输

1. 序号：TCP首部的序号字段用来保证数据能有序提交给应用层，TCP把数据看成无结构的有序的字节流。数据流中的每一个字节都编上一个序号字段的值是指本报文段所发送的数据的第一个字节序号。
2. 确认：TCP首部的确认号是期望收到对方的下一个报文段的数据的第一个字节的序号；
3. 重传：超时重传和冗余ACK重传
4. 流量控制：TCP采用大小可变的滑动窗口进行流量控制，窗口大小的单位是字节。发送窗口在连接建立时由双方商定。但在通信的过程中，接收端可根据自己的资源情况，随时动态地调整对方的发送窗口上限值(可增大或减小)。
   * 接受窗口rwnd，接收端缓冲区大小。接收端将此窗口值放在 TCP 报文的首部中的窗口字段，传送给发送端。
   * 拥塞窗口cwnd，发送缓冲区大小。
   * 发送窗口swnd，发送窗口的上限值= Min [rwnd, cwnd]。
5. 拥塞控制；

## TCP的重传机制？

* 超时重传

说白了就是在请求包发出去的时候，开启一个计时器，当计时器达到时间之后，没有收到ACK，则就进行重发请求的操作，一直重发直到达到重发上限次数或者收到ACK。

* 快速重传

还有一种机制就是快速重传，当接收方收到的数据包是不正常的序列号，那么接收方会重复把应该收到的那一条ACK重复发送，这个时候，如果发送方收到连续3条的同一个序列号的ACK，那么就会启动快速重传机制，把这个ACK对应的发送包重新发送一次。

## 常见的web攻击手段及其防御方式

* XSS （跨站脚本攻击) 【攻击者在 Web 页面中插入恶意脚本，当用户浏览页面时，促使脚本执行，从而达到攻击目的。

防御

转移和过滤用户提交的信息，将输入的数据进行转义处理，比如说讲 < 转义成&lt

* DDos 分布式拒绝服务，【发送大量请求，使服务器瘫痪】

防御

1. 最直接的方法增加带宽。但是攻击者用各地的电脑进行攻击，他的带宽不会耗费很多钱，但对于服务器来说，带宽非常昂贵。
2. 云服务提供商有自己的一套完整DDoS解决方案，并且能提供丰富的带宽资源。

* CSRF 跨站请求伪造 【用户本地存储cookie，攻击者利用用户的cookie进行认证，然后伪造用户发出请求】

防御

之所以被攻击是因为攻击者利用了存储在浏览器用于用户认证的cookie，那么如果我们不用cookie来验证不就可以预防了。所以我们可以采用token（不存储于浏览器）认证，为每一个提交的表单生成一个随机token，存储在session中，每次验证表单token，检查token是否正确。

* SQL注入 通过sql命令伪装成正常的http请求参数，传递到服务器端，服务器执行sql命令造成对数据库进行攻击【通过用户输入，拼接成恶意sql, 并执行】

防御

在java中，我们可以使用预编译语句(PreparedStatement)，这样的话即使我们使用sql语句伪造成参数，到了服务端的时候，这个伪造sql语句的参数也只是简单的字符，并不能起到攻击的作用。

很多orm框架已经可以对参数进行转义

做最坏的打算，即使被’拖库‘('脱裤，数据库泄露')。数据库中密码不应明文存储的，可以对密码使用md5进行加密，为了加大破解成本，所以可以采用加盐的（数据库存储用户名，盐（随机字符长），md5后的密文）方式。

## 一次完整的HTTP请求

* 1.对www.baidu.com这个网址进行DNS域名解析，得到对应的IP地址
* 2.根据这个IP，找到对应的服务器，发起TCP的三次握手
* 3.建立TCP连接后发起HTTP请求
* 4.服务器响应HTTP请求，浏览器得到html代码
* 5.浏览器解析html代码，并请求（从服务器上请求）html代码中的资源（如js、css图片等）（先得到html代码，才能去找这些资源）
* 6.浏览器对页面进行渲染呈现给用户

## HTTP请求格式与响应格式

（1）请求格式。主要有四部分组成，分别是：请求行、请求头、空行、消息体，每部分内容占一行

GET lindex_html HTTP/I _ 1 
Date: Thu, 20 May 2004 21:12:55 GMT 
connection: close 
Host: wwn.v_myfavoriteamazjngsite.com 
From: joebloe@someweositesomewnere.com 
Accept: text/html, text/plain 
User-Agent: Mozjlla/4.O (compatible; MSIE 6.0; Windows NT 5.1) 
Request Line 
General Headers 
Request Headers 
HTTP 
Request 
Entity Headers 
Message Body 

* 请求行：由三部分组成，GET/POST请求方法、请求资源URL、HTTP版本号
* 请求头：和缓存相关的头（Cache-Control，If-Modified-Since），客户端身份信息（User-Agent）等等
* 消息体：客户端发给服务端的请求数据，这部分数据并不是每个请求必须的。

（2）响应格式。包括：状态行、响应头、空行、消息体。每部分内容占一行。

HTTP/I.1 200 OK 
Date: Tnu, 20 May 2004 21:12:58 GMT 
Connection: close 
server: Apacne,'l .3.27 
Accept-Ranges: bytes 
Content-Type: text/html 
content-Length: 170 
Last-Modified: Tue, 18 May 2004 10:14:49 GMT 
<head> 
to the Amazjng 
<bodp 
site is under construction. Please come 
back later 
ontml> 
Status Line 
General Headers 
Response Headers 
Entity Headers 
HTTP 
Response 
Message Body 

* 状态行：HTTP协议版本号，状态码和状态说明三部分构成
* 响应头：响应头是服务器传递给客户端用于说明服务器的一些信息（Content-Type，charset等），以及将来继续访问该资源时的策略。
* 响应体：返回给客户端的HTML文本内容，或者其他格式的数据，比如：视频流、图片或者音频数据。

## HTTP状态码

服务器返回的响应报文中第一行为状态行，包含了状态码以及原因短语，用来告知客户端请求的结果。



**1XX 信息**

* 100 Continue ：表明到目前为止都很正常，客户端可以继续发送请求或者忽略这个响应。

2XX 成功

* **200 OK**
* 204 No Content ：请求已经成功处理，但是返回的响应报文不包含实体的主体部分。一般在只需要从客户端往服务器发送信息，而不需要返回数据时使用。
* 206 Partial Content：表示客户端进行了范围请求，响应报文包含由 Content-Range 指定范围的实体内容。

**3XX 重定向**

* 301 Moved Permanently ：永久性重定向
* 302 Found ：临时性重定向
* 303 See Other ：和 302 有着相同的功能，但是 303 明确要求客户端应该采用 GET 方法获取资源。

注：虽然 HTTP 协议规定 301、302 状态下重定向时不允许把 POST 方法改成 GET 方法，但是大多数浏览器都会在 301、302 和 303 状态下的重定向把 POST 方法改成 GET 方法。

* 304 Not Modified ：如果请求报文首部包含一些条件，例如：If-Match，If-Modified-Since，If-NoneMatch，If-Range，If-Unmodified-Since，如果不满足条件，则服务器会返回 304 状态码。
* 307 Temporary Redirect ：临时重定向，与 302 的含义类似，但是 307 要求浏览器不会把重定向请求的POST 方法改成 GET 方法。

**4XX 客户端错误**

* 400 Bad Request ：请求报文中存在语法错误。
* 401 Unauthorized ：该状态码表示发送的请求需要有认证信息（BASIC 认证、DIGEST 认证）。如果之前已进行过一次请求，则表示用户认证失败。
* 403 Forbidden ：请求被拒绝。
* 404 Not Found

**5XX 服务器错误**

* 500 Internal Server Error ：服务器正在执行请求时发生错误。
* 503 Service Unavailable ：服务器暂时处于超负载或正在进行停机维护，现在无法处理请求。
* 504 - Gateway Timeout:网关超时，由作为代理或网关的服务器使用，表示不能及时地从远程服务器获得应答

## HTTP1.0与HTTP1.1区别

1 缓存

在HTTP/1.0中，使用Expire头域来判断资源的fresh或stale，并使用条件请求（conditional request）来判断资源是否仍有效。例如，cache服务器通过If-Modified-Since头域向服务器验证资源的Last-Modefied头域是否有更新，源服务器可能返回304（Not Modified），则表明该对象仍有效；也可能返回200（OK）替换请求的Cache对象。

此外，HTTP/1.0中还定义了Pragma:no-cache头域，客户端使用该头域说明请求资源不能从cache中获取，而必须回源获取。HTTP/1.1在1.0的基础上加入了一些cache的新特性，当缓存对象的Age超过Expire时变为stale对象，cache不需要直接抛弃stale对象，而是与源服务器进行重新激活（revalidation）。

1. 带宽优化

HTTP/1.0中，存在一些浪费带宽的现象，例如客户端只是需要某个对象的一部分，而服务器却将整个对象送过来了。又比如下载大文件时不支持断点续传功能，在发生断连后不得不重新下载完整的包。

HTTP/1.1中在请求消息中引入了range头域，它支持只请求资源的某个部分。在响应消息中Content-Range头域声明了返回的这部分对象的偏移值和长度。如果服务器相应地返回了对象所请求范围的内容，则响应码为206（Partial Content），它可以防止Cache将响应误以为是完整的一个对象。

3 长连接

HTTP1.0每次HTTP请求都会创建一个新的TCP连接，请求完成之后这个TCP连接就会关闭。这种通信模式的效率不高，所以在HTTP/1.1中，引用了HTTP长连接的概念，使用长连接的HTTP协议。会在响应头加入Connection:keep-alive。这样当浏览器完成一次请求后，浏览器和服务器之间的TCP连接不会关闭，再次访问这个服务器上的网页时，浏览器会继续使用这一条已经建立的连接，也就是说两个请求可能共用一个TCP连接。

4 消息传递

HTTP消息中可以包含任意长度的实体，通常它们使用Content-Length来给出消息结束标志。但是，对于很多动态产生的响应，只能通过缓冲完整的消息来判断消息的大小，但这样做会加大延迟。如果不使用长连接，还可以通过连接关闭的信号来判定一个消息的结束。

HTTP/1.1中引入了Chunkedtransfer-coding来解决上面这个问题，发送方将消息分割成若干个任意大小的数据块，每个数据块在发送时都会附上块的长度，最后用一个零长度的块作为消息结束的标志。这种方法允许发送方只缓冲消息的一个片段，避免缓冲整个消息带来的过载。

5 Host头域

HTTP1.1在Request消息头里头多了一个Host域，而且是必传的，HTTP1.0则没有这个域。

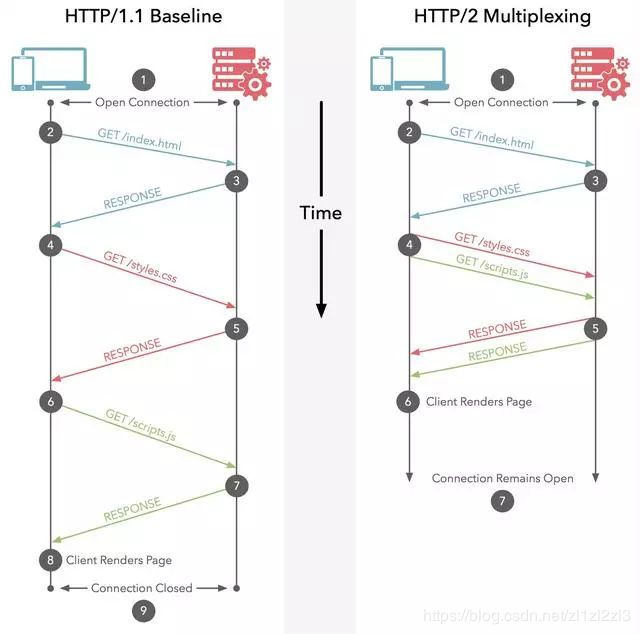
在HTTP1.0中认为每台服务器都绑定一个唯一的IP地址，因此，请求消息中的URL并没有传递主机名（hostname）。但随着虚拟主机技术的发 展，在一台物理服务器上可以存在多个虚拟主机（Multi-homed Web Servers），并且它们共享一个IP地址。

HTTP1.1的请求消息和响应消息都应支持Host头域，且请求消息中如果没有Host头域会报告一个错误（400 Bad Request）。此外，服务器应该接受以绝对路径标记的资源请求。

## HTTP2.0与HTTP1.1区别

* 多路复用

多路复用允许单一的HTTP2.0连接同时发起多重的请求-响应消息。



整个访问流程第一次请求index.html页面,之后浏览器会去请求style.css和scripts.js的文件。左边的图是顺序加载两个个文件的，右边则是并行加载两个文件。

* 服务端推送
* HTTP/2.0 在客户端请求一个资源时，会把相关的资源一起发送给客户端，客户端就不需要再次发起请求了。例如客户端请求page.html 页面，服务端就把script.js和style.css等与之相关的资源一起发给客户端。

服务端推送是一种在客户端请求之前发送数据的机制。当代网页使用了许多资源:HTML、样式表、脚本、图片等等。在HTTP/1.x中这些资源每一个都必须明确地请求。这可能是一个很慢的过程。浏览器从获取HTML开始，然后在它解析和评估页面的时候，增量地获取更多的资源。因为服务器必须等待浏览器做每一个请求，网络经常是空闲的和未充分使用的。

为了改善延迟，HTTP/2引入了server push，它允许服务端推送资源给浏览器，在浏览器明确地请求之前。一个服务器经常知道一个页面需要很多附加资源，在它响应浏览器第一个请求的时候，可以开始推送这些资源。这允许服务端去完全充分地利用一个可能空闲的网络，改善页面加载时间。

* 首部压缩

为什么要压缩？在 HTTP/1 中，HTTP 请求和响应都是由「状态行、请求 / 响应头部、消息主体」三部分组成。一般而言，消息主体都会经过 gzip 压缩，或者本身传输的就是压缩过后的二进制文件（例如图片、音频），但状态行和头部却没有经过任何压缩，直接以纯文本传输。

随着 Web 功能越来越复杂，每个页面产生的请求数也越来越多，导致消耗在头部的流量越来越多，尤其是每次都要传输 UserAgent、Cookie 这类不会频繁变动的内容，完全是一种浪费。

HTTP/2.0 也使用 Huffman 编码对首部字段进行压缩。

## HTTP与HTTPS区别

1 HTTP和HTTPS的基本概念

HTTP：是互联网上应用最为广泛的一种网络协议，是一个客户端和服务器端请求和应答的标准（TCP），用于从WWW服务器传输超文本到本地浏览器的传输协议，它可以使浏览器更加高效，使网络传输减少。

HTTPS：是以安全为目标的HTTP通道，简单讲是HTTP的安全版，即HTTP下加入SSL层，HTTPS的安全基础是SSL，因此加密的详细内容就需要SSL。

HTTPS协议的主要作用可以分为两种：一种是建立一个信息安全通道，来保证数据传输的安全；另一种就是确认网站的真实性。

2 HTTPS和HTTP的区别

* 1、https协议需要到ca申请证书，一般免费证书很少，需要交费。
* 2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。
* 3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443.
* 4、http的连接很简单，是无状态的，https协议是由SSL（安全套接字层）+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。
* 5、HTTPS比HTTP页面加载慢一些，不如HTTP高效，但更安全。
* 在OSI模型中，HTTP工作于应用层，而HTTPS工作于传输层；

3 HTTPS加密原理

* **对称加密**：数据使用 A 密钥加密，变成加密后的数据，加密后的数据使用 A 密钥解密，获取数据。
* **非对称加密**：使用一对非对称的密钥，一把叫私钥，一把叫公钥。**公钥加密的密文只能用私钥解密，私钥加密的密文只能用公钥解密。**公钥我们可以随便公开，因为别人知道了公钥毫无用处，经过公钥加密后的密文只能通过私钥来解密。

但由于公开密钥比共享密钥要慢，所以就需要综合一下两者的优缺点，使他们共同使用，而这也是HTTPS采用的加密方式。**在交换密钥阶段使用公钥加密，之后建立通信交换报文阶段则使用私钥加密。**

**如何确保公钥是真的呢？**为了解决公钥在传输过程中被攻击者替换，可以使用**由数字证书认证机构（CA）颁发的公钥证书。**接收到证书的客户端可以使用数字证书认证机构的公开密钥，对那张证书上的数字签名进行验证，一旦验证通过，客户端便可以明确两件事：

* 1 认证服务器的公开密钥的是真实有效的数字证书认证机构。
* 2 服务器的公开密钥是值得信赖的。
* **HTTPS通信过程**：
* 1. 客户端发起HTTPS请求

这个没什么好说的，就是用户在浏览器里输入一个https网址，然后连接到server的443端口。

* 2. 服务端的配置

采用HTTPS协议的服务器必须要有一套数字证书，可以自己制作，也可以向组织申请。区别就是自己颁发的证书需要客户端验证通过，才可以继续访问，而使用受信任的公司申请的证书则不会弹出提示页面(startssl就是个不错的选择，有1年的免费服务)。这套证书其实就是一对公钥和私钥。如果对公钥和私钥不太理解，可以想象成一把钥匙和一个锁头，只是全世界只有你一个人有这把钥匙，你可以把锁头给别人，别人可以用这个锁把重要的东西锁起来，然后发给你，因为只有你一个人有这把钥匙，所以只有你才能看到被这把锁锁起来的东西。

* 3. 传送证书

这个证书其实就是公钥，只是包含了很多信息，如证书的颁发机构，过期时间等等。

* 4. 客户端解析证书

这部分工作是有客户端的TLS来完成的，首先会验证公钥是否有效，比如颁发机构，过期时间等等，如果发现异常，则会弹出一个警告框，提示证书存在问题。如果证书没有问题，那么就生成一个随即值。然后用证书对该随机值进行加密。就好像上面说的，把随机值用锁头锁起来，这样除非有钥匙，不然看不到被锁住的内容。

* 5. 传送加密信息

这部分传送的是用证书加密后的随机值，目的就是让服务端得到这个随机值，以后客户端和服务端的通信就可以通过这个随机值来进行加密解密了。

* 6. 服务段解密信息

服务端用私钥解密后，得到了客户端传过来的随机值(私钥)，然后把内容通过该值进行对称加密。所谓对称加密就是，将信息和私钥通过某种算法混合在一起，这样除非知道私钥，不然无法获取内容，而正好客户端和服务端都知道这个私钥，所以只要加密算法够彪悍，私钥够复杂，数据就够安全。

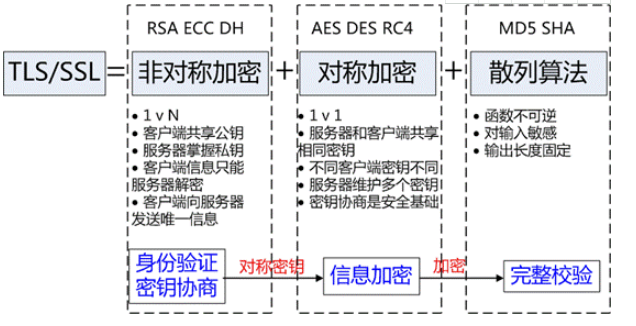
* 7. 传输加密后的信息

这部分信息是服务段用私钥加密后的信息，可以在客户端被还原

* 8. 客户端解密信息

客户端用之前生成的私钥解密服务段传过来的信息，于是获取了解密后的内容。整个过程第三方即使监听到了数据，也束手无策。

TLS/SSL的功能实现主要依赖于三类基本算法：散列函数 Hash、对称加密和非对称加密，其利用非对称加密实现身份认证和密钥协商，对称加密算法采用协商的密钥对数据加密，基于散列函数验证信息的完整性。



TLS的基本工作方式是，客户端使用非对称加密与服务器进行通信，实现身份验证并协商对称加密使用的密钥，然后对称加密算法采用协商密钥对信息以及信息摘要进行加密通信，不同的节点之间采用的对称密钥不同，从而可以保证信息只能通信双方获取。

一般情况下将二者写在一起TLS/SSL，我们可以将二者看做同一类协议，只不过TLS是SSL的升级版。

## GET与POST的区别

Http定义了与服务器交互的不同方法，最基本的方法有4种，分别是GET，POST，PUT，DELETE。

1. url可见性：

get，参数url可见

post，url参数不可见

\*\*get把请求的数据放在url上，即HTTP协议头上，其格式为：以?分割URL和传输数据，参数之间以&相连；post把数据放在HTTP的包体内（requrest body）

1. 传输数据的大小：

get一般传输数据大小不超过2k-4k

post请求传输数据的大小根据php.ini 配置文件设定，也可以无限大

HTTP规范中，没有对URL的长度和传输的数据大小进行限制。但是在实际开发过程中，对于GET，特定的浏览器和服务器对URL的长度有限制。因此，在使用GET请求时，传输数据会受到URL长度的限制。

\*\*get提交的数据最大是2k（原则上url长度无限制，那么get提交的数据也没有限制咯？限制实际上取决于浏览器，浏览器通常都会限制url长度在2K个字节，即使(大多数)服务器最多处理64K大小的url，也没有卵用）；

post理论上没有限制。实际上IIS4中最大量为80KB，IIS5中为100KB。

1. 数据传输上：

get，通过拼接url进行传递参数

post，通过body体传输参数

\*\*GET产生一个TCP数据包，浏览器会把http header和data一并发送出去，服务器响应200(返回数据);

POST产生两个TCP数据包，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok(返回数据)。

1. 后退页面的反应：

get请求页面后退时，不产生影响

post请求页面后退时，会重新提交请求

\*\*GET在浏览器回退时是无影响的，POST会再次提交请求

1. 缓存性：

get请求是可以缓存的

post请求不可以缓存

\*\*GET请求会被浏览器主动cache，而POST不会，除非手动设置。

1. 安全性：

都不安全，原则上post肯定要比get安全，毕竟传输参数时url不可见，但也挡不住部分人闲的没事在那抓包玩，浏览器还会缓存get请求的数据。安全性个人觉得是没多大区别的，防君子不防小人就是这个道理。对传递的参数进行加密，其实都一样。

## Cookie与Session（解决HTTP无状态）？

http无状态协议，是指协议对于交互性场景没有记忆能力。

Cookie的工作原理：（Cookie信息都是放在请求头和响应头里）

（1）浏览器端第一次发送请求到服务器端

（2）服务器端创建Cookie，该Cookie中包含用户的信息，然后将该Cookie发送到浏览器端

（3）浏览器端再次访问服务器端时会携带服务器端创建的Cookie

（4）服务器端通过Cookie中携带的数据区分不同的用户。

Session的工作原理：

（1）浏览器端第一次发送请求到服务器端，服务器端创建一个Session，同时会创建一个特殊的Cookie（name为JSESSIONID的固定值，value为session对象的ID），然后将该Cookie发送至浏览器端

（2）浏览器端发送第N（N>1）次请求到服务器端,浏览器端访问服务器端时就会携带该name为JSESSIONID的Cookie对象

（3）服务器端根据name为JSESSIONID的Cookie的value(sessionId),去查询Session对象，从而区分不同用户。

可是session有一个缺陷：如果web服务器做了负载均衡，那么下一个操作请求到了另一台服务器的时候session会丢失。

name为JSESSIONID的Cookie不存在（关闭或更换浏览器），返回1中重新去创建Session与特殊的Cookie

name为JSESSIONID的Cookie存在，根据value中的SessionId去寻找session对象

value为SessionId不存在\*\*（Session对象默认存活30分钟）\*\*，返回1中重新去创建Session与特殊的Cookie

value为SessionId存在，返回session对象。

Session和Cookie的区别：

(1)cookie数据存放在客户的浏览器上，session数据放在服务器上

(2)cookie不是很安全，别人可以分析存放在本地的COOKIE并进行COOKIE欺骗,如果主要考虑到安全应当使用session

(3)session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能，如果主要考虑到减轻服务器性能方面，应当使用COOKIE

(4)单个cookie在客户端的限制是4K，就是说一个站点在客户端存放的COOKIE不能4K。

可以考虑将登陆信息等重要信息存放为session，不重要的信息可以放在cookie中。

## 单点登录Cookie被禁用？

单点登录的原理是后端生成一个 session ID，然后设置到 cookie，后面的所有请求浏览器都会带上 cookie，然后服务端从 cookie 里获取 session ID，再查询到用户信息。所以，保持登录的关键不是 cookie，而是通过cookie 保存和传输的 session ID，其本质是能获取用户信息的数据。

系统中的确要考虑cookie被禁用的情况，可以把当前session的ID存入服务器中，请求页面的时候要带上这个ID去本地服务器中去查询当前用户是否已经登录或者存在。

## Token的介绍和与session的区别？

Token的引入：Token是在客户端频繁向服务端请求数据，服务端频繁的去数据库查询用户名和密码并进行对比，判断用户名和密码正确与否，并作出相应提示，在这样的背景下，Token便应运而生。

Token的定义：Token是服务端生成的一串字符串，以作客户端进行请求的一个令牌，当第一次登录后，服务器生成一个Token便将此Token返回给客户端，以后客户端只需带上这个Token前来请求数据即可，无需再次带上用户名和密码。最简单的token组成:uid(用户唯一的身份标识)、time(当前时间的时间戳)、sign(签名，由token的前几位+盐以哈希算法压缩成一定长的十六进制字符串，可以防止恶意第三方拼接token请求服务器)。

使用Token的目的：Token的目的是为了减轻服务器的压力，减少频繁的查询数据库，使服务器更加健壮。

传统身份验证

HTTP 是一种没有状态的协议，也就是它并不知道是谁是访问应用。这里我们把用户看成是客户端，客户端使用用户名还有密码通过了身份验证，不过下回这个客户端再发送请求时候，还得再验证一下。

解决的方法就是，当用户请求登录的时候，如果没有问题，我们在服务端生成一条记录，这个记录里可以说明一下登录的用户是谁，然后把这条记录的 ID 号发送给客户端，客户端收到以后把这个 ID 号存储在 Cookie 里，下次这个用户再向服务端发送请求的时候，可以带着这个 Cookie ，这样服务端会验证一个这个 Cookie 里的信息，看看能不能在服务端这里找到对应的记录，如果可以，说明用户已经通过了身份验证，就把用户请求的数据返回给客户端。

上面说的就是 Session，我们需要在服务端存储为登录的用户生成的 Session ，这些 Session 可能会存储在内存，磁盘，或者数据库里。我们可能需要在服务端定期的去清理过期的 Sessio

基于 Token 的身份验证

使用基于 Token 的身份验证方法，在服务端不需要存储用户的登录记录。大概的流程是这样的：

客户端使用用户名跟密码请求登录

服务端收到请求，去验证用户名与密码

验证成功后，服务端会签发一个 Token，再把这个 Token 发送给客户端

客户端收到 Token 以后可以把它存储起来，比如放在 Cookie 里或者 Local Storage 里

客户端每次向服务端请求资源的时候需要带着服务端签发的 Token

服务端收到请求，然后去验证客户端请求里面带着的 Token，如果验证成功，就向客户端返回请求的数据。

# 操作系统

## 2.1进程与线程的区别？

* 什么是进程

进程是系统中正在运行的一个程序，程序一旦运行就是进程。进程可以看成程序执行的一个实例。进程是系统资源分配的独立实体，每个进程都拥有独立的地址空间。一个进程无法访问另一个进程的变量和数据结构，如果想让一个进程访问另一个进程的资源，需要使用进程间通信，比如管道，文件，套接字等。

* 什么是线程

线程是进程的一个实体，是进程的一条执行路径。线程是进程的一个特定执行路径。当一个线程修改了进程的资源，它的兄弟线程可以立即看到这种变化。

* 进程与线程的区别

1. 进程是资源分配的最小单位，线程是程序执行的最小单位（资源调度的最小单位）
2. 开销方面：进程有自己的独立地址空间，每启动一个进程，系统就会为它分配地址空间，建立数据表来维护代码段、堆栈段和数据段，这种操作非常昂贵。而线程是共享进程中的数据的，使用相同的地址空间，因此CPU切换一个线程的花费远比进程要小很多，同时创建一个线程的开销也比进程要小很多。
3. 共享资源：线程之间的通信更方便，同一进程下的线程共享全局变量、静态变量等数据，而进程之间的通信需要以通信的方式（IPC)进行。
4. 多进程程序更健壮，多线程程序只要有一个线程死掉，整个进程也死掉了，而一个进程死掉并不会对另外一个进程造成影响，因为进程有自己独立的地址空间。

## 2.2什么情况下会发生死锁？解决策略？

所谓死锁，是指多个进程循环等待它方占有的资源而无限期地僵持下去的局面。很显然，如果没有外力的作用，那麽死锁涉及到的各个进程都将永远处于封锁状态。当两个或两个以上的进程同时对多个互斥资源提出使用要求时，有可能导致死锁。

1. 互斥条件。资源互斥使用；即某个资源在一段时间内只能由一个进程占有，不能同时被两个或两个以上的进程占有。
2. 不可抢占条件。资源不可被剥夺；进程所获得的资源在未使用完毕之前，资源申请者不能强行地从资源占有者手中夺取资源，而只能由该资源的占有者进程自行释放。
3. 占有且申请条件。多个进程保持一定的资源，但又请求新的资源；进程至少已经占有一个资源，但又申请新的资源；由于该资源已被另外进程占有，此时该进程阻塞；但是，它在等待新资源之时，仍继续占用已占有的资源。
4. 循环等待条件。存在一个进程等待序列{P1，P2，...，Pn}，其中P1等待P2所占有的某一资源，P2等待P3所占有的某一源，......，而Pn等待P1所占有的的某一资源，形成一个进程循环等待环。

死锁的预防是保证系统不进入死锁状态的一种策略。它的基本思想是要求进程申请资源时遵循某种协议，从而打破产生死锁的四个必要条件中的一个或几个，保证系统不会进入死锁状态。

1. 打破互斥条件。即允许进程同时访问某些资源。但是，有的资源是不允许被同时访问的，像打印机等等，这是由资源本身的属性所决定的。所以，这种办法并无实用价值。
2. 打破不可抢占条件。即允许进程强行从占有者那里夺取某些资源。就是说，当一个进程已占有了某些资源，它又申请新的资源，但不能立即被满足时，它必须释放所占有的全部资源，以后再重新申请。它所释放的资源可以分配给其它进程。这就相当于该进程占有的资源被隐蔽地强占了。这种预防死锁的方法实现起来困难，会降低系统性能。
3. 打破占有且申请条件。可以实行资源预先分配策略。即进程在运行前一次性地向系统申请它所需要的全部资源。如果某个进程所需的全部资源得不到满足，则不分配任何资源，此进程暂不运行。只有当系统能够满足当前进程的全部资源需求时，才一次性地将所申请的资源全部分配给该进程。由于运行的进程已占有了它所需的全部资源，所以不会发生占有资源又申请资源的现象，因此不会发生死锁。
4. 打破循环等待条件，实行资源有序分配策略。采用这种策略，即把资源事先分类编号，按号分配，使进程在申请，占用资源时不会形成环路。所有进程对资源的请求必须严格按资源序号递增的顺序提出。进程占用了小号资源，才能申请大号资源，就不会产生环路，从而预防了死锁

解决策略：

鸵鸟策略：

鸵鸟策略的实质：出现死锁的概率很小，并且出现之后处理死锁会花费很大的代价，还不如不做处理，OS中这种置之不理的策略称之为鸵鸟策略（也叫鸵鸟算法）。

银行家算法：

系统给当前进程分配资源时，先检查是否安全：在满足当前的进程X资源申请后，是否还能有足够的资源去满足下一个距最大资源需求最近的进程（如某进程最大需要5个单位资源，已拥有1个，还尚需4个），若可以满足，则继续检查下一个距最大资源需求最近的进程，若均能满足所有进程，则表示为安全，可以允许给当前进程X分配其所需的资源申请，否则让该进程X进入等待。

## 2.3 Linux进程的状态

（1）R（task\_running），可执行状态

只有在该状态的进程才可能在CPU上运行。同一时刻有可能有多个进程处于可执行状态，这些进程的task\_struct结构（进程控制块）被放入对应CPU的可执行队列中（一个进程最多只能出现在一个CPU的可执行队列中）。

（2）S (task\_interruptible)，可中断的睡眠状态

处于这个状态的进程因为等待某某事件的发生（比如等待socket连接、等待信号量），而被挂起。这些进程的task\_struct结构被放入对应事件的等待队列中。当这些事件发生时（由外部中断触发、或由其他进程触发），对应的等待队列中的一个或多个进程将被唤醒。

（3）D(task\_uninterruptible)，不可中断的睡眠状态

在这个状态下的进程处于睡眠状态，但是此刻进程是不可中断的。不可中断，指的并不是CPU不响应外部硬件的中断，而是指进程不响应异步信号（不然的话kill -9就不能杀死进程）。而task\_uninterruptible状态存在的意义就在于：内核的某些处理流程是不能被打断的。

（4）T(task\_stopped or task\_traced)，暂停状态或跟踪状态

表示进程暂停下来了；向进程发送一个SIGSTOP信号，它就会因响应该信号而进入TASK\_STOPPED状态，向进程发送一个SIGCONT信号，可以让其从TASK\_STOPPED状态恢复到TASK\_RUNNING状态。

（5）Z(TASK\_DEAD - EXIT\_ZOMBIE)，退出状态，进程成为僵尸进程。

进程在退出的过程中，处于task\_read状态，此时进程占有的所有资源将被回收，除了task\_struct结构以及少数资源以外。于是进程就只剩下task\_struct这么个空壳，故称为僵尸。之所以保留task\_struct，是因为task\_struct里面保存了进程的退出码、以及一些统计信息。而其父进程很可能会关心这些信息。

（6）X (TASK\_DEAD - EXIT\_DEAD)，退出状态，进程即将被销毁。

## 2.4进程的五种状态

运行态：该进程正在执行。

就绪态：进程已经做好了准备，只要有机会就开始执行。

阻塞态（等待态）：进程在某些事情发生前不能执行，等待阻塞进程的事件完成。

新建态：刚刚创建的进程，操作系统还没有把它加入到可执行进程组中，通常是进程控制块已经创建但是还没有加载到内存中的进程。

退出态：操作系统从可执行进程组中释放出的进程，或由于自身或某种原因停止运行。

## 2.4操作系统的进程调度算法

* 先来先服务调度算法FCFS

先到的进程先调度，执行过程不会被中断直到进程结束。

优点：易于实现，且相当公平。

缺点：比较有利于长进程，而不利于短进程。

* 短作业优先调度算法SJF

优先分配给短进程执行。

优点：平均周转时间最短，进程等待时间缩短，可以增大系统吞吐量。

缺点：难以准确预估进程执行时间，开销较大；不利于长进程，有可能“饥饿”现象。

* **高响应比调度算法HRRN**

**一种关于先来先服务和短作业优先的折中算法，当一个长进程等待时间过长，就会获得较高的优先权，因此不会出现“饥饿”现象。**

**优先级D=（执行时间+等待时间）/执行时间**

**优点：不会出现“饥饿”现象，长作业也有机会被调度。**

**缺点：每次都需要计算优先级，系统开销大。**

* **时间片轮转调度算法RR**

进程设定时间片，即每个进程运行的时间，在一个时间片结束时，发生时钟中断，调度程序暂停执行并加入队尾，通过上下文切换执行当前队首进程

优点：算法简单，响应时间短。

缺点：不利于处理紧急作业；时间片过小会导致频繁进程上下文切换，增大系统开销；时间片过长则会退化为FCFS。

* **多级反馈队列调度算法**

不必事先知道各种进程所需要执行的时间，它是目前被公认的一种较好的进程调度算法。 其实施过程如下：

a.设置多个就绪队列，并为各个队列赋予不同的优先级。在优先权越高的队列中，为每个进程所规定的执行时间片就越小。

b.当一个新进程进入内存后，首先放入第一队列的末尾，按FCFS原则排队等候调度。 如果他能在一个时间片中完成，便可撤离；如果未完成，就转入第二队列的末尾，在同样等待调度…… 如此下去，当一个长作业（进程）从第一队列依次将到第n队列（最后队列）后，便按第n队列时间片轮转运行。

c.仅当第一队列空闲时，调度程序才调度第二队列中的进程运行；仅当第1到第（i-1）队列空时， 才会调度第i队列中的进程运行，并执行相应的时间片轮转。

d.如果处理机正在处理第i队列中某进程，又有新进程进入优先权较高的队列， 则此新队列抢占正在运行的处理机，并把正在运行的进程放在第i队列的队尾。

## 2.5父进程创建子进程，fork详解

1. 函数原型：pid\_t fork(void); //void代表没有任何形式参数
2. 除了0号进程（系统创建的）之外，linux系统中都是由其他进程创建的。创建新进程的进程，即调用fork函数的进程为父进程，新建的进程为子进程。
3. fork函数不需要任何参数，对于返回值有三种情况：
4. 对于父进程，fork函数返回新建子进程的pid；
5. 对于子进程，fork函数返回 0；
6. 如果出错， fork 函数返回 -1。

|  |
| --- |
| int pid=fork();  if(pid < 0){  //失败，一般是该用户的进程数达到限制或者内存被用光了  ........  } else if(pid == 0){  //子进程执行的代码  ......  } else{  //父进程执行的代码  .........  } |

1. vfork 也可以创建新线程，但是有一点点不一样。fork 后的进程完全复制了父进程的资源，所以它的资源是独立的，因此开销比较大。fork出来的子进程一般会调用 exec ，然后自己就结束了，所以比较浪费。而 vfork 是共享父进程的资源，它改了什么，父进程会跟着受影响，但它减少了开销,因为它不需要将父进程的资源重新拷贝一份。并且，**vfork 后子进程先运行，父进程阻塞，直到子进程结束**。

## 2.5僵尸进程，孤儿进程，父进程退出子进程？

**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，那么那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。**

**危害：因此孤儿进程并不会有什么危害。**

**僵尸进程：一个进程使用fork创建子进程，如果子进程退出，而父进程并没有调用wait或waitpid获取子进程的状态信息，那么子进程的进程描述符仍然保存在系统中。这种进程称之为僵死进程**

**危害：如果进程不调用wait / waitpid的话， 那么保留的那段信息就不会释放，其进程号就会一直被占用，但是系统所能使用的进程号是有限的，如果大量的产生僵死进程，将因为没有可用的进程号而导致系统不能产生新的进程. 此即为僵尸进程的危害，应当避免。**

父进程退出后，子进程会发生什么？

在linux中，所有进程都有一个共同的父进程systemd，如果父进程退出了，子进程还没运行结束，子进程会被init进程收养，成为它的子进程；

## 2.6如何让进程(或正在运行的程序)到后台运行?

在Linux中，如果要让进程在后台运行，一般情况下，我们在命令后面加上&即可，实际上，这样是将命令放入到一个作业队列中了；

对于已经在前台执行的命令，也可以重新放到后台执行，首先按ctrl+z暂停已经运行的进程，然后使用bg命令将停止的作业放到后台运行；

但是如上方到后台执行的进程，其父进程还是当前终端shell的进程，而一旦父进程退出，则会发送hangup信号给所有子进程，子进程收到hangup以后也会退出。如果我们要在退出shell的时候继续运行进程，则需要使用nohup忽略hangup信号，或者使用setsid将父进程设为init进程(进程号为1)；

对于已经在后台运行的进程，可以将其放在subshell中执行，从而变成后台进程；方法很简单，将命令用括号() 括起来即可；

## 2.7进程间通信的方式有几种，区别是什么？

1) 管道（管道，通常指无名管道）

1. 半双工的，具有固定的读端和写端；
2. 只能用于具有亲属关系（父子进程、兄弟进程）的进程之间的通信；

linux中用管道实现父子进程通信

1. 用户要实现父进程到子进程的数据通道，可以在父进程关闭管道读出一端，然后相应的子进程关闭管道的输入端。
2. 先用pipe()建立管道 然后fork函数创建子进程。父进程向子进程发消息，子进程读消息。
3. 实现

2) FiFO（有名管道）

FIFO可以在两个独立的进程之间交换数据，与无名管道不同，会在本地文件系统创建一个有名字的文件，用于通信。

3) 消息队列（消息队列适合短小的数据通信）

1）消息队列可以实现消息的随机查询。消息不一定要以先进先出的次序读取，编程时可以按消息的类型读取。

2）消息队列允许一个或多个进程向它写入或者读取消息。

3）与无名管道、命名管道一样，从消息队列中读出消息，消息队列中对应的数据都会被删除

4）每个消息队列都有消息队列标识符，消息队列的标识符在整个系统中是唯一的。

5）消息队列是消息的链表，存放在内存中，由内核维护。只有内核重启或人工删除消息队列时，该消息队

消息队列可以实现消息的随机查询。

4) 信号量

1. 信号量是一个计数器，信号量用于实现进程间的互斥与同步，而不是用于存储进程间通信数据；
2. 信号量用于进程间同步，若要在进程间传递数据需要结合共享内存；
3. 信号量基于操作系统的 PV 操作，程序对信号量的操作都是原子操作。
4. 每次对信号量的 PV 操作不仅限于对信号量值加 1 或减 1，而且可以加减任意正整数。

5) 共享内存

1. 共享内存，指两个或多个进程共享一个给定的存储区；
2. 共享内存是最快的一种进程通信方式，因为进程是直接对内存进行存取；
3. 因为多个进程可以同时操作，所以需要进行同步；
4. 信号量+共享内存通常结合在一起使用。
5. 共享内存的用处：比较适合在进程间处理容量较大的数据。在 Linux 终端运行 ipcs -m命令，你就可以查看目前系统现存的共享内存。
6. Socket通信 在机器上开一个端口作为服务器让客户连接，走TCP/UDP协议。这就是不同机器之间进程通信了。Socket对TCp / udp协议进行封装

效率的原因

共享内存是通过把同一块内存分别映射到不同的进程空间中实现进程间通信。而共享内存本身不带任何互斥与同步机制，但当多个进程同时对同一内存进行读写操作时会破坏该内存的内容，所以，在实际中，同步与互斥机制需要用户来完成。

效率：

采用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次数据： 一次从输入文件到共享内存区，另一次从共享内存区到输出文件。

实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建立共享内存区域。而是保持共享区域，直到通信完毕为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。

## 2.8常见的5种IO模型？

1) 阻塞I/O

首先，要从你常用的IO操作谈起，比如read和write，通常IO操作都是阻塞I/O的，也就是说当你调用read时，如果没有数据收到，那么线程或者进程就会被挂起，直到收到数据。阻塞的意思，就是一直等着。阻塞I/O就是等着数据过来，进行读写操作。应用的函数进行调用，但是内核一直没有返回，就一直等着。应用的函数长时间处于等待结果的状态，我们就称为阻塞I/O。每个应用都得等着，每个应用都在等着，浪费啊！很像现实中的情况。大家都不干活，等着数据过来，过来工作一下，没有的话继续等着。

（*系统调用（一般是 IO 接口）不返回调用结果并让当前线程一直阻塞，只有当该系统调用获得结果或者超时出错时才返回。*）

2) 非阻塞I/O

非阻塞IO很简单，通过fcntl（POSIX）或ioctl（Unix）设为非阻塞模式，这时，当你调用read时，如果有数据收到，就返回数据，如果没有数据收到，就立刻返回一个错误，如EWOULDBLOCK。这样是不会阻塞线程了，但是你还是要不断的轮询来读取或写入。相当于你去查看有没有数据，告诉你没有，过一会再来吧！应用过一会再来问，有没有数据？没有数据，会有一个返回。但是依旧很不好。应用必须得过一会来一下，问问内核有木有数据啊。这和现实很像啊！好多情况都得去某些地方问问好了没有？木有，明天再过来。明天，好了木有？木有，后天再过来。。。。。忙碌的应用。。。。

3) I/O复用

多路复用是指使用一个线程来检查多个文件描述符（Socket）的就绪状态，比如调用select和poll函数，传入多个文件描述符（FileDescription，简称FD），如果有一个文件描述符（FileDescription）就绪，则返回，否则阻塞直到超时。得到就绪状态后进行真正的操作可以在同一个线程里执行，也可以启动线程执行（比如使用线程池）。虾米意思？就是派一个代表，同时监听多个文件描述符是否有数据到来。等着等着，如有有数据，就告诉某某你的数据来啦！赶紧来处理吧。有没有很感动，一个人待着，帮了很多人。医院的黄牛，一个人排队，大家只要把钱给它，它就会把号给需要的人，开个玩笑。。。。

4) 信号驱动I/O

首先我们允许套接口进行信号驱动I/O,并安装一个信号处理函数，进程继续运行并不阻塞。当数据准备好时，进程会收到一个SIGIO信号，可以在信号处理函数中调用I/O操作函数处理数据。

5）异步I/O

用户进程发起read操作之后，立刻就可以开始去做其它的事。而另一方面，从kernel的角度，当它受到一个asynchronous read之后，首先它会立刻返回，所以不会对用户进程产生任何block。然后，kernel会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，kernel会给用户进程发送一个signal，告诉它read操作完成了。

当一个异步过程调用发出后，调用者不能立刻得到结果。实际处理这个调用的部件在完成后，通过状态、通知和回调来通知调用者的输入输出操作。

## 2.9BIO NIO AIO的区别 同步 异步 阻塞 非阻塞？

1. 同步阻塞I/O(BIO)：

同步并阻塞，服务器实现模式为一个连接一个线程，即客户端有连接请求时服务器端就需要启动一个线程进行处理，如果这个连接不做任何事情会造成不必要的线程开销，当然可以通过线程池机制改善。

1. 同步非阻塞I/O(NIO)：

同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。

（3）异步非阻塞I/O(AIO)：

异步 IO 是基于事件和回调机制实现的，也就是应用操作之后会直接返回，不会堵塞在那里，当后台处理完成，操作系统会通知相应的线程进行后续的操作。

同步和异步：

同步就是一个任务的完成需要依赖另外一个任务时，只有等待被依赖的任务完成后，依赖的任务才能算完成，这是一种可靠的任务序列。要么成功都成功，失败都失败，两个任务的状态可以保持一致。而异步是不需要等待被依赖的任务完成，只是通知被依赖的任务要完成什么工作，依赖的任务也立即执行，只要自己完成了整个任务就算完成了。至于被依赖的任务最终是否真正完成，依赖它的任务无法确定，所以它是不可靠的任务序列。我们可以用打电话和发短信来很好的比喻同步与异步操作。

阻塞和非阻塞：

阻塞与非阻塞主要是从 CPU 的消耗上来说的，阻塞就是 CPU 停下来等待一个慢的操作完成 CPU 才接着完成其它的事。非阻塞就是在这个慢的操作在执行时 CPU 去干其它别的事，等这个慢的操作完成时，CPU 再接着完成后续的操作。虽然表面上看非阻塞的方式可以明显的提高 CPU 的利用率，但是也带了另外一种后果就是系统的线程切换增加。增加的 CPU 使用时间能不能补偿系统的切换成本需要好好评估。

## 2.10 select，poll，epoll的区别？

* select==>时间复杂度O(n)

它仅仅知道了，有I/O事件发生了，却并不知道是哪那几个流（可能有一个，多个，甚至全部），我们只能无差别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。所以**select具有O(n)的无差别轮询复杂度**，同时处理的流越多，无差别轮询时间就越长。

* poll==>时间复杂度O(n)

poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态， **但是它没有最大连接数的限制**，原因是它是基于链表来存储的。

* epoll==>时间复杂度O(1)

**epoll可以理解为event poll**，不同于忙轮询和无差别轮询，epoll会把哪个流发生了怎样的I/O事件通知我们。所以我们说epoll实际上是**事件驱动（每个事件关联上fd）**的，此时我们对这些流的操作都是有意义的。**（复杂度降低到了O(1)）。**

select和epoll最大的区别就是：select只是告诉你一定数目的流有事件了，至于哪个流有事件，还得你一个一个地去轮询，而epoll会把发生的事件告诉你，通过发生的事件，就自然而然定位到哪个流了。不能不说epoll跟select相比，是质的飞跃，我觉得这也是一种**牺牲空间，换取时间的思想**，毕竟现在硬件越来越便宜了。

Select，poll，epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。**但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的**，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。

## 2.11线程同步和互斥

（1）线程同步

* 同步就是协同步调，按预定的先后次序进行运行。线程同步是指多线程通过特定的设置（如互斥量，事件对象，临界区）来控制线程之间的执行顺序（即所谓的同步），也可以说是在线程之间通过同步建立起执行顺序的关系；最基本的场景就是：两个或两个以上的进程或线程在运行过程中协同步调，按预定的先后次序运行。比如 A 任务的运行依赖于 B 任务产生的数据。

（2）线程互斥

* 指对于共享的进程系统资源，在每一个线程访问时的排它性。当有多个线程都要使用某一共享资源时，任何时刻最多只允许一个线程去使用，其它要使用该资源的线程必须等待，直到占用资源者释放该资源。线程互斥可以看成是一种特殊的线程同步；最基本的场景就是：一个公共资源同一时刻只能被一个进程或线程使用，多个进程或线程不能同时使用公共资源。

（3）线程同步的方式

* 1、临界区：通过对多线程的串行化来访问公共资源或一段代码，速度快，适合控制数据访问。在任意时刻只允许一个线程对共享资源进行访问，如果有多个线程试图访问公共资源，那么在有一个线程进入后，其他试图访问公共资源的线程将被挂起，并一直等到进入临界区的线程离开，临界区在被释放后，其他线程才可以抢占。它并不是核心对象，不是属于操作系统维护的，而是属于进程维护的。
* 2、互斥对象：互斥对象和临界区很像，采用互斥对象机制，只有拥有互斥对象的线程才有访问公共资源的权限。因为互斥对象只有一个，所以能保证公共资源不会同时被多个线程同时访问。当前拥有互斥对象的线程处理完任务后必须将线程交出，以便其他线程访问该资源。
* 3、信号量：信号量也是内核对象。它允许多个线程在同一时刻访问同一资源，但是需要限制在同一时刻访问此资源的最大线程数目；

当需要一个计数器来限制可以使用某共享资源的线程数目时，可以使用“信号量”对象。Semaphore类对象保存了对当前访问某一个指定资源的线程的计数值，该计数值是当前还可以使用该资源的线程数目。如果这个计数达到了零，则所有对这个Semaphore类对象所控制的资源的访问尝试都被放入到一个队列中等待，直到超时或计数值不为零为止。

* 4、事件对象： 通过通知操作的方式来保持线程的同步，还可以方便实现对多个线程的优先级比较的操作。事件是内核对象，事件分为手动置位事件和自动置位事件。事件Event内部它包含一个使用计数，一个布尔值表示是手动置位事件还是自动置位事件，另一个布尔值用来表示事件有无触发。事件可以解决线程间同步问题，因此也能解决互斥问题。事件机制，则允许一个线程在处理完一个任务后，主动唤醒另外一个线程执行任务。比如在某些网络应用程序中，一个线程如A负责侦听通信端口，另外一个线程B负责更新用户数据，利用事件机制，则线程A可以通知线程B何时更新用户数据。

## 2.12不带缓存、带缓存、内存映射处理大文件

写数据的时候

不带缓存的IO操作数据流向路径：数据——内核缓存区——磁盘

好处：减少磁盘IO

带缓存IO操作数据流向路径：数据——流缓存区——内核缓存区——磁盘

好处：一次写入到内核缓存区的时间变多了，减少调用read和write的系统的调用次数

内存映射文件和之前说的不带缓存IO操作最大的不同之处就在于它虽然最终也是要从磁盘读取数据，但是它并不需要将数据读取到OS内核缓冲区，而是直接将进程的用户私有地址空间中的一部分区域与文件对象建立起映射关系，就好像直接从内存中读、写文件一样，速度当然快了。

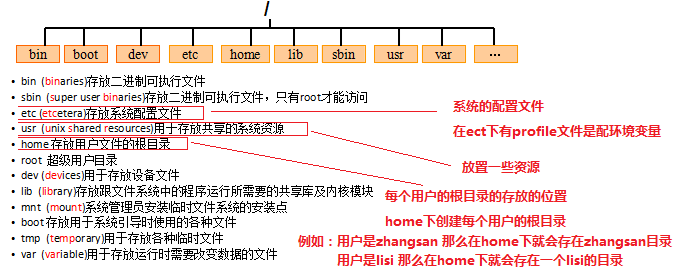
具体过程：

此时并没有拷贝数据到内存中去，而是当进程代码第一次引用这段代码内的虚拟地址时，触发了缺页异常，这时候OS根据映射关系直接将文件的相关部分数据拷贝到进程的用户私有空间中去，当有操作第N页数据的时候重复这样的OS页面调度程序操作。注意啦，原来内存映射文件的效率比不带缓存IO高的重要原因就是因为少了把数据拷贝到OS内核缓冲区这一步。

<https://blog.csdn.net/WANGYONGZIXUE/article/details/46421931>

https://blog.csdn.net/FcBayernMunchen/article/details/8635427

## Linux目录结构



## 2.14 linux 内核链表和普通的链表有什么区别？

一般的链表结构体中都会包含两部分：一个是指针（前向指针和后向指针）部分，还有就是有效数据部分。将我们链表需要存放的数据放在结构体中的有效数据；其中，通过两个节点分别指向上一个节点和下一个节点。这是一般的链表的使用方法。

* 但是内核链表的结构体中只有两个指针，并没有有效数据区，所以内核链表的用法和我们的普通链表的用法是不一样的；
* 我们使用内核链表的方法就是：自己建立一个结构体去包含这个内核链表，将整个结构体作为一个链表的节点，然后使用内置的链表的前向指针指向下一个结构体中内置的链表，使用内置的链表的后向指针指向上一个结构体中内置的链表。

## 2.15 Linux常用命令

|  |  |
| --- | --- |
| ls -al | 显示当前目录下所有文件详细信息，包括隐藏文件 |
| cd [~] [-] | 切换目录 [~目前用户身份的主目录] [-前一个工作目录] |
| chmod 770 文件名 | 修改文件的权限 |
| cp A B | 复制文件A为B |
| rm A  rm -r A  rm –f | 删除A  删除非空目录或文件A  强制删除 |
| mkdir/rmdir | 新建目录、删除一个空目录 |
| man 命令 | 查询命令详细解释 |
| pwd | 显示当前目录 |
| mv 文件名 目标位置  mv A B | 移动文件  重命名A为B（目录或文件都可以） |
| vim 文件名 | 进入后按i可以输入；退出按ESC然后:wq；不保存退出：按ESC 然后:q! |
| reboot | 重启服务器 |
| shift+PgUp\PgDn | 命令行界面上下滚动 |
| tcpdump [host 192.168.1.120] | 截获所有192.168.1.120 的主机收到的和发出的所有的数据包 |
| kill -9 PID | 关闭服务，比如某程序PID=6817 kill -9 6817 就关闭了这个服务 |
| top | 查看当前系统负载情况 |
| netstat -ntlp  netstat -ntulp | 查看当前所有TCP端口  查看当前所有TCP和UDP端口 |
| netstat -nap| grep 5672 | 查看端口号5672是否被监听 |
| lsof –i:端口号 | 查看端口被哪个进程占用 |
| free -m | 查看内存使用情况 |
| ps -A  ps –ef | grep java | 显示所有运行中的进程  搜索java进程 |
| God status  god stop 服务名 | 查看当前部署的服务  停止服务 |

* Tab：命令与文件名补全。
* Ctrl+C：中断正在运行的程序；
* Ctrl+D：结束键盘输入（End Of File，EOF）
* Linux如何对文件内容中的关键字进行查找?
  + 如果是用vi打开文件后，在命令行下输入“/关键字”
  + 如果是在没有打开文件的前提就用"cat 文件名 | grep "关键字""
* nohup命令：（让某个程序在后台运行）如果你正在运行一个进程，而且你觉得在退出帐户时该进程还不会结束，那么可以使用nohup命令。该命令可以在你退出帐户/关闭终端之后继续运行相应的进程。

## 2.16请简述Windows内存管理的方式

https://www.cnblogs.com/Allen-rg/p/7173077.html

# java基础

## 3.1基础知识

### 3.1.1面向对象的三大特性？

1 封装

* 定义：把数据和操作数据的方法绑定起来，对数据的访问只通过已定义的接口，面向对象的本质就是把现实世界的事物描绘成一系列自治、封闭的对象；我们在类中编写的方法就是对实现细节的一种封装，编写的类就是对**属性和行为的封装**。在java中能使用private、protected、public三种修饰符或不用（即默认defalut）对外部对象访问该对象的属性和行为进行限制。

2 继承

* 定义：继承是**从已有的类得到继承信息创建新类的过程**；提供继承信息的类称为父类，得到继承信息的类称为子类；
* 优缺点：提高代码复用性；提高代码的维护性；但是类之间的耦合性增强了，我们开发的时候的原则是：高内聚、低耦合（内聚：是指自己完成事情的能力，自己能完成就不要麻烦别人；耦合：类与类的关系）；

3 多态

* 定义：**指父对象中的同一个行为能在其多个子对象中有不同的表现**。简单的说，就是一句话：**允许父类引用指向子类对象**。多态则是体现在overriding（重写）上，而overload（重载）则不属于面向对象中多态的范畴，因为重载在非面向对象中也存在。重写是面向对象中的多态，因为重写是与继承紧密联系，是面向对象所特有的。实现多态的技术称为：动态绑定（dynamic binding），是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。
  + 重写：子类重新定义父类中的方法；
  + 重载：一个类中允许有多个同名的函数，而这些函数的参数列表不同（参数的个数不同、参数的类型不同、参数的顺序不同）；
* 优点：消除类型之间的耦合关系；可以实现接口的重用；

### 3.1.2C语言实现面向对象

面向对象编程（OOP）并不是一种特定的语言或者工具，它只是一种设计方法、设计思想。它表现出来的三个最基本的特性就是封装、继承与多态。C语言也同样可以实现OOP，里面具体用到的有C语言中的宏，结构体，函数指针, 聚合组合等。

C语言中的定义：

* 结构体：在C语言中，常把一个对象用结构体进行封装，这样便于对对象进行操作；结构体可以嵌套。因而可以把一个结构体当成另一个结构体的成员；
* 函数指针：函数指针是指针的一种，它指向函数的首地址（函数的函数名即为函数的首地址），可以通过函数指针来调用函数。
* extern与static：extern和static是C语言中的两个修饰符，extern可用于修饰函数或者变量，表示该变量或者函数在其他文件中进行了定义；static也可用于修饰函数或者变量，表示该函数或者变量只能在该文件中使用。可利用它们对数据或者函数进行隐藏或者限制访问权限。
* 封装：在C语言中，可以用结构+函数指针来模拟类的实现，而用这种结构定义的变量就是对象。
* 继承：在C语言中，可以利用“结构在内存中的布局与结构的声明具有一致的顺序”这一事实实现继承。比如我们要设计一个作图工具，其中可能涉及到的对象有Point(点)，Circle(圆)，由于圆是由点组成的，所有可以看成Circle继承自Point。另外，Point和Circle都需要空间申请，空间释放等操作，所有他们有共同的基类Base。
* 多态：可以是用C语言中的万能指针void\* 实现多态。

### 3.1.3Java中泛型的优点？

1.安全：在编译的时候检查类安全  
2.省心：所有强制转换都是自动和隐式的，提高代码的重用率

* 泛型：是对 Java 语言的类型系统的一种扩展，以支持创建可以按类型进行参数化的类。可以把类型参数看作是使用参数化类型时指定的类型的一个占位符，就像方法的形式参数是运行时传递的值的占位符一样。泛型的本质是参数化类型，也就是说所操作的数据类型被指定为一个参数。这种参数类型可以用在类、接口和方法的创建中，分别称为泛型类、泛型接口、泛型方法。
* 使用泛型的优点：
  + 类型安全。可以在编译的时候检查类型安全。
  + 消除强制类型转换。消除源代码中的许多强制类型转换。这使得代码更加可读，并且减少了出错机会。
  + 潜在的性能收益。在泛型的初始实现中，编译器将强制类型转换（没有泛型的话，程序员会指定这些强制类型转换）插入生成的字节码中。由于泛型的实现方式，支持泛型（几乎）不需要 JVM 或类文件更改。所有工作都在编译器中完成，编译器生成类似于没有泛型（和强制类型转换）时所写的代码，只是更能确保类型安全而已。
* 泛型的使用规则：
  + 泛型的类型参数只能是类类型，不能使简单的类型；
  + 泛型的类型参数可以有多个；
  + 泛型的参数类型可以使用extends语句，比如<T extends superclass>；
  + 泛型的参数类型还可以是通配符型，比如Class<?> classType = Class.forName(java.lang.String)；

### 3.1.4抽象类和接口的区别？



### 3.1.5Object类中的方法有哪些？

|  |  |
| --- | --- |
| clone() | 创建并返回此对象的一个副本 |
| hashCode() | 返回该对象的hash值 |
| equals() | 判断一个对象是否与此对象相等 |
| toString() | 返回该对象的字符串表示 |
| finalize() | 当垃圾回收器确定不存在对该对象的更多引用时，由对象的垃圾回收器调用此方法 |
| getClass() | 是一个native方法，返回的是此Object对象的类对象/运行时类对象Class<?> |
| notify() | 唤醒在此对象监视器上等待的单个线程/所有线程。 |
| notifyAll() | 唤醒在此对象监视器上等待的所有线程。 |
| wait() | 暂停当前线程，直到其他线程调用这个对象的notify()方法 |

### 3.1.6Collection与Collections的区别？

（1）java.util.Collection 是一个集合接口。它提供了对集合对象进行基本操作的通用接口方法。Collection接口在Java 类库中有很多具体的实现。Colleciton是集合接口，继承它的接口有List，Set，注意map集合虽然也属于集合体系，但是map并不继承collection，map和collection是平级关系。

（2）java.util.Collections 是一个包装类。它包含有各种有关集合操作的静态多态方法。此类不能实例化，就像一个工具类，服务于Java的Collection框架 。Collections是针对集合类的一个帮助类，提供一系列静态方法实现对各种集合的搜索、排序、线程安全等操作 。比如 排序Collections.sort()，混排Collections.shuffling()，反转Collections.reverse()，替换所有的元素Collections.fill()，拷贝Collections.copy()。

### 3.1.7Collections.sort()或Arrays.sort()底层实现？

（1）不论是Collections.sort方法（是对list集合进行排序）或者是Arrays.sort方法（是对数组进行排序），底层实现都是TimSort实现的，这是jdk1.7新增的，以前是归并排序。

* Colletions.sort()实际会将list转为数组，然后调用Arrays.sort()，排完了再转回List。
* Arrays.sort()，对原始类型(int[],double[],char[],byte[])，JDK6里用的是快速排序，对于对象类型(Object[])，JDK6则使用归并排序。到了JDK7，快速排序升级为双基准快排(双基准快排vs三路快排)；归并排序升级为归并排序的改进版TimSort；

（2）Timsort是一种结合了归并排序和插入排序的混合算法，由Tim Peters在2002年提出，并且已经成为Python 2.3版本以后内置排序算法。针对现实中需要排序的数据分析看，大多数据通常是有部分已经排好序的数据块，在Timsort中称这些已经排好序的数据块为“run”（一个一个的“分区”）。在排序时，Timsort迭代数据元素，将其放到不同的 run 里，同时针对这些 run ，按规则进行合并至只剩一个，则这个仅剩的 run 即为排好序的结果。简单来说，这个算法可以概括为两步：

* 1）判断数组的大小，小于32使用二分插入排序：使用二分查找位置，进行插入排序。
* 2）数组大于32时，先算出一个合适的大小，在将输入按其升序和降序特点进行了分区，排序的输入的单位不是一个个单独的数字，而是一个个的run。将run入栈，当栈顶的run的长度满足：runLen[n-2] <= runLen[n-1] + runLen[n]或者 runLen[n-1] <= runLen[n]，则对两个短run归并为一个新run，则到只剩栈顶元素时排序也完成了。

（3）本质上 Timsort 是一个经过大量优化的归并排序，在最坏的情况下，Timsort 时间复杂度为 O(nlogn)。在最佳情况下，即输入已经排好序，它则以线性时间运行O(n)。

时间复杂度什么时候出现Logn————————当数据量减半时，它的复杂度就是就是logn了

### 3.1.8List接口下的类

实现Collection接口的接口主要有 List Set Queue 接口

List接口下的类都是有序，重复的

可以存储null,由于它具有重复性，添加几个就存储几个

1. ArrayList

底层是用数组实现，查询快（数组存储连续，有index角标），增删比较慢，线程不安全，初始容量是10 ，加载因子是1(即当容量大于10之后，才会触发扩容)，扩容的容量后的容量等于原容量 + 原容量的0.5倍 +1

ArrayList中的快速失败机制（fail-fast）

快速失败机制产生的条件:当多个线程对Collection进行操作时，若其中某一个线程通过Iterator遍历集合时，该集合的内容被其他线程所改变，则会抛出ConcurrentModificationException异常。

1. LinkedList

底层是基于双向链表实现的，查询慢，增删比较快，线程不安全，由于它底层是链表实现的，没有初始化的大小，也没有扩容机制。

1. Vector

Vector是线程安全的ArrayList,线程安全（是因为在每个方法上加了synchronized关键字），初始容量是10 加载因子是1(即当容量大于10之后，才会触发扩容)，扩容的容量后的容量等于原容量 + 原容量

### 3.1.9Set接口下的类

Set接口下的类都是无限，不能重复的,由于其具有不可重复性，添加多个null

只能存储一个

1. hashSet

HashSet实际上是一个HashMap实例，\*\*都是一个存放链表的数组\*\*。它不保证存储元素的迭代顺序；此类允许使用null元素。HashSet中不允许有重复元素，这是因为HashSet是基于HashMap实现的，HashSet中的元素都存放在HashMap的key上面，而value中的值都是统一的一个固定对象private static final Object PRESENT = new Object();

HashSet中add方法调用的是底层HashMap中的put()方法，而如果是在HashMap中调用put，首先会判断key是否存在，如果key存在则修改value值，如果key不存在这插入这个key-value。而在set中，因为value值没有用，也就不存在修改value值的说法，因此往HashSet中添加元素，首先判断元素（也就是key）是否存在，如果不存在这插入，如果存在着不插入，这样HashSet中就不存在重复值。

所以判断key是否存在就要重写元素的类的equals()和hashCode()方法，当向Set中添加对象时，首先调用此对象所在类的hashCode()方法，计算次对象的哈希值，此哈希值决定了此对象在Set中存放的位置；若此位置没有被存储对象则直接存储，若已有对象则通过对象所在类的equals()比较两个对象是否相同，不相同，则不能添加，相等，则覆盖了。

其是线性不安全的，扩容机制和HashMap相同

1. LinkedHashSet

hashSet加双向链表

LinkedHashSet的增删改查都是调用HashSet的方法。

HashSet底层实现是调用HashMap的方法。

此时HashSet的HashMap已经被转化成LinkedHashMap实例，

所以，LinkedHashSet的增删改查都是调用LinkedHashMap的方法。

LinkedHashSet也定义了accessOrder变量，值为true表示按照访问顺序排序，值为false表示按照插入顺序排序。accessOrder的默认值为false，也就是按照插入顺序排序。

1. TreeSet

，它是一个有序的集合类，TreeSet的底层是通过TreeMap实现的。TreeSet并不是根据插入的顺序来排序，而是根据实际的值的大小来排序。TreeSet也支持两种排序方式：

自然排序

自定义排序

### 3.1.11Queue接口下的类

1. ArrayQueue 是一个循环队列，内部是通过数组实现的，同时增加 和 删除 元素不会引起内部数组的拷贝，效率较高。

由于数组大小是提前确定的，也无法自动扩容(可以调用 resize() 进行手动扩容)，因此不太适合当前队列个数未知的情况。

2. LinkedList

3. PriorityQueue为Queue接口的实现类，具有队列的性质。但是该队列会将内部元素进行排序，即有序的队列。PriorityQueue实现有序的方法和TreeSet相同。方式如下：

队列内部元素实现Comparable接口，并重写其中compareTo(Object obj)方法，定制排序规则

创建PriorityQueue对象时，传入Comparator对象，实现 compare(Object obj1,Object obj2)方法负责排序规则。

### 3.1.12Map接口下的集合类

1. HashTable

Hashtable 中的方法是Synchronize的，而HashMap中的方法在缺省情况下是非Synchronize的。

不支持null键和值，很少被推荐

哈希值的使用不同，HashTable直接使用对象的hashCode。而HashMap重新计算hash值。

HashTable中hash数组默认大小是11，增加的方式是 old\*2+1。

### HashMap

1. HashMap是基于数组（数组中存元素的每个位置又称为一个bucket,即桶）和链表实现的数据结构，JDK1.8之后变成了数组 + 链表 + 红黑树的实现方式，HashMap的数组和链表中是一个Entry对象，每个Entry对象包含三部分key（键）、value（值），next(指向下一个Entry。线程不安全，初始容量是16，扩容因子是0.75，扩容容量后的容量是原来容量的2倍。数组小于64则进行扩容，等于64了，并且链表长度大于8时，就将链表转换为红黑树了。

2. HashMap的put操作，put操作的时候，首先调用key的hashCode函数得到了hash值，再计算hash &（n -1），其中n是hashMap的容量，即是数组的长度，这个操作在n为2的幂次时相当于 hash % n ,由于&操作比较快，所以就用了上面的表达式，得到的值即为要插入桶的位置，如果该位置没有元素则直接插入，如果有元素则，调用key.equal(),判断key是否相等，如果相等，就覆盖当前元素，如果不相等，就将该元素插入到已存元素的下面。另外当链表中的元素大于8的时候，就会将链表转换成了红黑树，当链表的元素小于6的时候，就会退化成链表。

HashMap的get操作，get操作的时候，和put操作的时候一样，调用HashCode函数，计算相应的桶的位置，从该桶中的key.equal（）进行比较，不相等就在链表或者红黑树中接着寻找，相等就就找到了相应的value值，返回即可。

3. HashMap多线程扩容的时候，容易形成环

参考这个链接https://blog.csdn.net/zhuqiuhui/article/details/51849692

避免尾部遍历。在扩容的时候，尾部遍历是为了避免在新列表插入数据时，遍历队尾的位置。因为，直接插入的效率更高。

JDK1.7中rehash的时候，旧链表迁移新链表的时候，如果在新表的数组索引位置相同，则链表元素会倒置，JDK1.8不会倒置，通过增加tail指针，既避免了死循环问题（让数据直接插入到队尾），又避免了尾部遍历。

4. hashMap解决碰撞是链地址法，就是将hash值相等的值，key不相等的，放在当前桶的链表后面

另外两种，一个是开放地址法：当发生地址冲突时，按照某种方法继续探测哈希表中的其他存储单元，直到找到空位置为止。这个过程可用下式描述： H i ( key ) = ( H ( key )+ d i ) mod m ( i = 1,2,…… ， k ( k ≤ m – 1))

其中： H ( key ) 为关键字 key 的直接哈希地址， m 为哈希表的长度， di 为每次再探测时的地址增量。

采用这种方法时，首先计算出元素的直接哈希地址 H ( key ) ，如果该存储单元已被其他元素占用，则继续查看地址为 H ( key ) + d 2 的存储单元，如此重复直至找到某个存储单元为空时，将关键字为 key 的数据元素存放到该单元。

增量 d 可以有不同的取法，并根据其取法有不同的称呼：

d i ＝ 1 ， 2 ， 3 ， …… 线性探测再散列；

d i ＝ 1^2 ，－ 1^2 ， 2^2 ，－ 2^2 ， k^2， -k^2…… 二次探测再散列；

d i ＝ 伪随机序列 伪随机再散列；

最后一种是再hash法，有多个Hash函数，通过hash函数得到一个Hash值，如果冲突，就再hash一次。

5. hashMap的扩容

JDk1.7中才是判断当前大小>=阈值 且 准备插入桶发生hash冲突才进行扩容，重新计算插入值的数组下标。1.8是先插入之后全部统一计算，因为已经插入了，所以(++size > 阈值)的话，直接进行扩容，并不会去管是不是这次新插入的值有没有发生过哈希冲突。

扩容后数据存储位置的计算方式也不一样：

在JDK1.7的时候是直接用hash值和需要扩容的二进制数进行&（这里就是为什么扩容的时候为啥一定必须是2的多少次幂的原因所在，因为如果只有2的n次幂的情况时最后一位二进制数才一定是1，这样能最大程度减少hash碰撞）（hash值 & length-1）

而在JDK1.8的时候直接用了JDK1.7的时候计算的规律，也就是扩容前的原始位置+扩容的大小值=JDK1.8的计算方式，而不再是JDK1.7的那种异或的方法。但是这种方式就相当于只需要判断Hash值的新增参与运算的位是0还是1就直接迅速计算出了扩容后的储存方式

6. 容量为2的幂的原因？

当我们根据key的hash确定其在数组的位置时，如果n为2的幂次方，可以保证数据的均匀插入，如果n不是2的幂次方，可能数组的一些位置永远不会插入数据，浪费数组的空间，加大hash冲突。

7. HashMap中并没有直接使用KV中K原有的hash值; 在HashMap的put、get操作时也未直接使用K中原有的hash值，而使用了一个hash()方法。让我们一起看一下这个方法。

```

static final int hash(Object key) {

int h;

return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);

}

```

8. HashMap的遍历

keySet（）函数得到的包含key的一个Set集合

Map.entrySet() 这个方法返回的是一个Set<Map.Entry<K,V>>，Map.Entry 是一个接口，他的用途是表示一个映射项（里面有Key和Value），而Set<Map.Entry<K,V>>表示一个映射项的Set。Map.Entry里有相应的getKey和getValue方法，让我们能够从一个项中取出Key和Value。

1. CurrentHashMap

1. java7

ConcurrentHashMap主要有三大结构：整个Hash表，segment（段），HashEntry（节点）。每个segment就相当于一个HashTable。

当ConcurrentHashMap上有一个线程在做写入操作,如果另外一个线程是去其他的segment里读数据的话,此时是允许操作的,但是如果此时这个线程是去相同的segment中做读操作的话,就不允许此次读取,只有当那个写入操作的线程完成后,才能执行读操作

ConcurrentHashMap有些方法需要跨段，比如size()和containsValue()，它们可能需要锁定整个表而而不仅仅是某个段，这需要按顺序锁定所有段，操作完毕后，又按顺序释放所有段的锁。这里“按顺序”是很重要的，否则极有可能出现死锁，在ConcurrentHashMap内部，段数组是final的，并且其成员变量实际上也是final的，但是，仅仅是将数组声明为final的并不保证数组成员也是final的，这需要实现上的保证。这可以确保不会出现死锁，因为获得锁的顺序是固定的。

2. Java8 ConcurrentHashMap结构基本上和Java8的HashMap一样，不过保证线程安全性。

JDK1.8的实现已经摒弃了Segment的概念，而是直接用Node数组+链表+红黑树的数据结构来实现，并发控制使用Synchronized和CAS来操作

CAS插入空的节点，当节点为空的时候，就用CAS进行赋值。

Synchronized插入的时候，锁定Node节点，不是Node节点的链表

get方法没有同步锁，是用voliate修饰的

1. LinkedHashMap

LinkedHashMap底层维护了一个hashMap和双向链表，LinkedHashMap定义了accessOrder变量，值为true表示按照访问顺序排序，值为false表示按照插入顺序排序。accessOrder的默认值为false，也就是按照插入顺序排序。

1. TreeMap

它是有序的集合。而且是一个红黑树结构，每个key-value都作为一个红黑树的节点，存取的时间复杂度都是O(log(n))。

TreeMap 默认排序规则：按照key的字典顺序来排序（升序）

队列内部元素实现Comparable接口，并重写其中compareTo(Object obj)方法，定制排序规则

创建TreeMap对象时，传入Comparator对象，实现 compare(Object obj1,Object obj2)方法负责排序规则

### 3.1.13final、finally、finalize的区别

1. final：用于声明属性，方法和类，分别表示属性不可变，方法不可覆盖，被其修饰的类不可继承。
2. finally：异常处理语句结构的一部分，表示总是执行。
3. finalize： Object 类的一个方法，在垃圾回收器执行的时候会调用被回收对象的此方法，可以覆盖此方法提供垃圾收集时的其他资源回收，例如关闭文件等。该方法更像是一个对象生命周期的临终方法，当该方法被系统调用则代表该对象即将“死亡”，但是需要注意的是，我们主动行为上去调用该方法并不会导致该对象“死亡”，这是一个被动的方法（其实就是回调方法），不需要我们调用。

### 3.1.14String、StringBuilder、StringBuffer的区别

== 和 equals( ) 方法的区别

首先，==号在比较基本数据类型时比较的是值，而用==号比较两个对象时比较的是两个对象的地址值。，那equals()方法呢？我们可以通过查看源码知道，equals()方法存在于Object类中，因为Object类是所有类的直接或间接父类，也就是说所有的类中的equals()方法都继承自Object类，而通过源码我们发现，Object类中equals()方法底层依赖的是==号，那么，在所有没有重写equals()方法的类中，调用equals()方法其实和使用==号的效果一样，也是比较的地址值，然而，Java提供的所有类中，绝大多数类都重写了equals()方法，重写后的equals()方法一般都是比较两个对象的值。

String 字符串的一些比较

<https://blog.csdn.net/zhuzbYR/article/details/89736919>

* String 被声明为 final，因此它不可被继承。

1. 可变性

+ String 不可变

+ StringBuffer 和 StringBuilder 可变

2. 线程安全

+ String 不可变，因此是线程安全的

+ StringBuilder 不是线程安全的

+ StringBuffer 是线程安全的，内部使用 synchronized 进行同步

### 3.1.15栈和队列的区别

* 相同点：

1. 都是线性结构。
2. 插入操作都是限定在表尾进行。
3. 都可以通过顺序结构和链式结构实现。
4. 插入与删除的时间复杂度都是O(1)，在空间复杂度上两者也一样。
5. 多链栈和多链队列的管理模式可以相同。

* 不同点：

1. 队列先进先出，栈先进后出
2. 删除数据元素的位置不同，栈的删除操作在表尾进行，队列的删除操作在表头进行。
3. 应用场景不同；常见栈的应用场景包括括号问题的求解，表达式的转换和求值，函数调用和递归实现，深度优先搜索遍历等；常见的队列的应用场景包括计算机系统中各种资源的管理，消息缓冲器的管理和广度优先搜索遍历等。
4. 顺序栈能够实现多栈空间共享，而顺序队列不能。

### 3.1.16Java跨平台和反射机制

Java编写的程序，一次编译，到处运行。这也是Java程序为什么是无关平台的所在，原因在于，java的源代码会被编译成.class文件字节码，只要装有Java虚拟机JVM的地方（Java提供了各种不同平台上的虚拟机制，第一步由Java IDE进行源代码编译，得到相应类的字节码.class文件，第二步，Java字节码由JVM执行解释给目标计算机，第三步，目标计算机将结果呈现给我们计算机用户；因此，Java并不是编译机制，而是解释机制），.class文件畅通无阻。

反射的定义

反射就是动态加载对象，并对对象进行剖析。

Java的反射机制，操作的就是这个.class文件，首先加载相应类的字节码（运行idea的时候，.class文件的字节码会加载到内存中），随后取出（反射 reflect）字节码中的构造函数、方法以及变量（字段）。

### 3.1.17Class.forName()和ClassLoader的区别？

1. 相同点：在java中Class.forName()和ClassLoader都可以对类进行加载。
2. Class.forName除了将类的.class文件加载到jvm中之外，还会对类进行解释，执行类中的static块。

而classloader只干一件事情，就是将.class文件加载到jvm中，不会执行static中的内容，只有在newInstance才会去执行static块。

所以这就是为什么在jdbc进行数据库操作的时候要用Class.forName进行驱动的加载。

（3）应用场景：

* Spring 框架中的 IOC 的实现就是使用的ClassLoader；
* JDBC通常是使用 Class.forName() 方法来加载数据库连接驱动，因为在 JDBC 规范中明确要求 Driver(数据库驱动)类必须向 DriverManager 注册自己。

### 3.1.18数组复制的几种方式？高效率？

（1）System.arraycopy（浅拷贝）【推荐使用】

|  |
| --- |
| /\*\*参数：src - 源数组。srcPos - 源数组中的起始位置。dest - 目标数组。destPos - 目标数据中的起始位置。length - 要复制的数组元素的数量\*/  public static native void arraycopy(Object src,int srcPos,Object dest,int destPos,int length) |

（2）使用clone方法（对数组是浅拷贝，对于对象而言是深拷贝）

|  |
| --- |
| /\*\*java.lang.Object类的clone()方法为protected类型，不可直接调用，需要先对要克隆的类进行下列操作：  首先被克隆的类实现Cloneable接口；然后在该类中覆盖clone()方法，并且在该clone()方法中调用super.clone()；这样，super.clone()便可以调用java.lang.Object类的clone()方法。\*/ |

（3）Arrays.copyOf（浅拷贝）

|  |
| --- |
| //实际上它调用的就是System.arraycopy，所以肯定也是浅拷贝  public static byte[] copyOfRange(byte[] original, int from, int to) {  int newLength = to - from;  if (newLength < 0)  throw new IllegalArgumentException(from + " > " + to);  byte[] copy = new byte[newLength];  System.arraycopy(original, from, copy, 0,  Math.min(original.length - from, newLength));  return copy;  } |

1. for循环遍历赋值

### 3.1.19Java中NIO？



* + IO是面向流的处理，NIO是面向块(缓冲区)的处理
  + 面向流的I/O：系统一次一个字节地处理数据。
  + 面向块(缓冲区)的I/O：系统以块的形式处理数据。

NIO主要有三个核心部分组成：buffer缓冲区；Channel管道；Selector选择器（**多路复用器**）；

### 3.1.20解决hash冲突的三个方法

1. 开放地址法

当关键字key的哈希地址p=H（key）出现冲突时，以p为基础，产生另一个哈希地址p1，如果p1仍然冲突，再以p为基础，产生另一个哈希地址p2，…，直到找出一个不冲突的哈希地址pi ，将相应元素存入其中。再散列函数形式

Hi=（H（key）+di）% m   i=1，2，…，n

1. 再哈希法

这种方法是同时构造多个不同的哈希函数：

Hi=RH1（key） i=1，2，…，k

当哈希地址Hi=RH1（key）发生冲突时，再计算Hi=RH2（key）……，直到冲突不再产生。这种方法不易产生聚集，但增加了计算时间。

1. 链地址法

这种方法的基本思想是将所有哈希地址为i的元素构成一个称为同义词链的单链表，并将单链表的头指针存在哈希表的第i个单元中，因而查找、插入和删除主要在同义词链中进行。链地址法适用于经常进行插入和删除的情况。

### 3.1.21 java的异常处理机制



所有异常的根类为 java.lang.Throwable，Throwable 下面又派生了两个子类：Error 和 Exception，Error 表示应用程序本身无法克服和恢复的一种严重问题。

Java的异常处理机制原理

Java通过面向对象的方式对异常进行处理，Java把异常按照不同的类型进行分类，并提供了良好的接口。在Java中，每个异常都是一个对象，它都是Throwable

或其子类的实例。当一个方法出现异常后就会抛出一个异常对象，该对象中包含有异常信息，调用这个对象的方法可以捕获到这个异常并对异常进行处理。Java的异常处理是通过5个关键词来实现的：try catch throw throws finally。

一般情况下是用try来执行一段程序，如果出现异常，系统会抛出（throws），我们可以通过它的类型来捕捉它，或最后由缺省处理器来处理它（finally）。

try：用来指定一块预防所有异常的程

catch：紧跟在try后面，用来捕获异常

throw：用来明确的抛出一个异常

throws：用来标明一个成员函数可能抛出的各种异常

finally：确保一段代码无论发生什么异常都会被执行的一段代码

### 3.1.22运行时异常和非运行时异常的区别

运行时异常，都是RuntimeException类及其子类异常，如NullPointerException(空指针异常)、IndexOutOfBoundsException(下标越界异常)等，这些异常是*不检查异常*，程序中可以选择捕获处理，也可以不处理。这些异常一般是*由程序逻辑错误引起*的，程序应该从逻辑角度尽可能避免这类异常的发生。

非运行时异常（编译时异常）是RuntimeException以外的异常，类型上都属于Exception类及其子类。从程序语法角度讲是必须进行处理的异常，如果*不处理，程序就不能编译通过*。如IOException、SQLException等以及用户自定义的Exception异常，一般情况下不自定义检查异常。

### 3.1.23curentHashMap中的size怎么求取？

size操作就是遍历了两次所有的Segments，每次记录Segment的modCount值，然后将两次的modCount进行比较，如果相同，则表示期间没有发生过写入操作，就将原先遍历的结果返回，如果不相同，则把这个过程再重复做一次，如果再不相同，则就需要将所有的Segment都锁住，然后一个一个遍历了。

原文链接：<https://blog.csdn.net/chenyiminnanjing/article/details/82716180>

### 3.1.24 avl（平衡二叉树）和红黑树的区别

红黑树的优点

红黑是用非严格的平衡来换取增删节点时候旋转次数的降低，任何不平衡都会在三次旋转之内解决，而AVL是严格平衡树，因此在增加或者删除节点的时候，根据不同情况，旋转的次数比红黑树要多。所以红黑树的插入效率更高！！！

与AVL树的比较

红黑树要求从根节点到叶子节点的最长路径不大于最短路径的两倍

AVL 树要求每一个子树的左右孩子节点高度差不超过1

保持平衡的要求上面，AVL树要求大于RBtree, 也就带来了search, insert 和delete操作的性能差异

相同点

插入、删除操作为了保持原来的性质，有可能要进行必要的调整，主要看调整的代价，时间复杂度；其中AVL树的调整频繁，代价大(要保持高度平衡)，而RBtree 删除和插入的时间复杂度要好于AVL，

而对于search 操作而言，不涉及旋转调整操作，由于AVL树保持高度平衡，树的最大高度不超过1.44log(n), 而红黑树最大深度不超过 2log(n+2),

## 3.2多线程编程

### 3.2.1创建线程的3种方式

（1）继承Thread类创建线程

1. 自定义线程类继承Thread类；
2. 重写run()方法，编写线程执行体；
3. 创建线程对象，调用start()方法启动线程；

（2）实现Runnable接口创建线程

1. 定义Runnable接口实现类
2. 重写run()方法，编写线程执行
3. 创建线程对象，调用start()方法启动线程

（3）实现Callable接口创建线程

1. 实现Callable接口，需要返回值类型
2. 重写call方法，需要抛出异常
3. 创建目标对象
4. 创建执行服务：ExecutorService ser = Executors.newFixedThreadPool(1);
5. 提交执行：Future result1 = ser.submit(callable);
6. 获取结果：boolean r1 = result1.get()
7. 关闭服务：ser.shutdownNow()

|  |
| --- |
| public class ThreadCurrent implements Callable<Boolean> {  @Override  public Boolean call() throws Exception {  for (int i = 0; i < 100; i++) {  System.out.println(Thread.currentThread().getName() + "..." + i);  }  return true;  }  public static void main(String[] args) throws Exception {  ThreadCurrent callable = new ThreadCurrent();  //创建执行任务  ExecutorService service = Executors.newFixedThreadPool(2);  //提交执行  Future<Boolean> submit1 = service.submit(callable);  Future<Boolean> submit2 = service.submit(callable);  //获取结果  Boolean aBoolean1 = submit1.get();  Boolean aBoolean2 = submit1.get();  //判断线程是否顺利结束或有异常  System.out.println(aBoolean1);  System.out.println(aBoolean2);  //关闭服务  service.shutdown();  }  } |

1. 创建线程的3种方式比较

采用实现Runnable、Callable接口的方式创见多线程时：

优势是：线程类只是实现了Runnable接口或Callable接口，还可以继承其他类。在这种方式下，多个线程可以共享同一个target对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将CPU、代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想。

同时处理同一份资源https://blog.csdn.net/edmond999/article/details/45073329

劣势是：编程稍微复杂，如果要访问当前线程，则必须使用Thread.currentThread()方法。

使用继承Thread类的方式创建多线程时

优势是:编写简单，如果需要访问当前线程，则无需使用Thread.currentThread()方法，直接使用this即可获得当前线程。

劣势是：线程类已经继承了Thread类，所以不能再继承其他父类。

### 3.2.2守护线程和非守护线程

在Java中有两类线程：User Thread(用户线程)、Daemon Thread(守护线程)

只要当前JVM实例中存在任何一个非守护线程没有结束，守护线程就全部工作；只有当最后一个非守护线程结束时，守护线程随着JVM一同结束工作。守护线程的作用是为其他线程的运行提供便利服务，守护线程最典型的应用就是 GC (垃圾回收器)，它就是一个很称职的守护者。

(1) thread.setDaemon(true)必须在thread.start()之前设置，否则会跑出一个IllegalThreadStateException异常。你不能把正在运行的常规线程设置为守护线程。

(2) 在Daemon线程中产生的新线程也是Daemon的。

### 3.2.3线程的生命周期

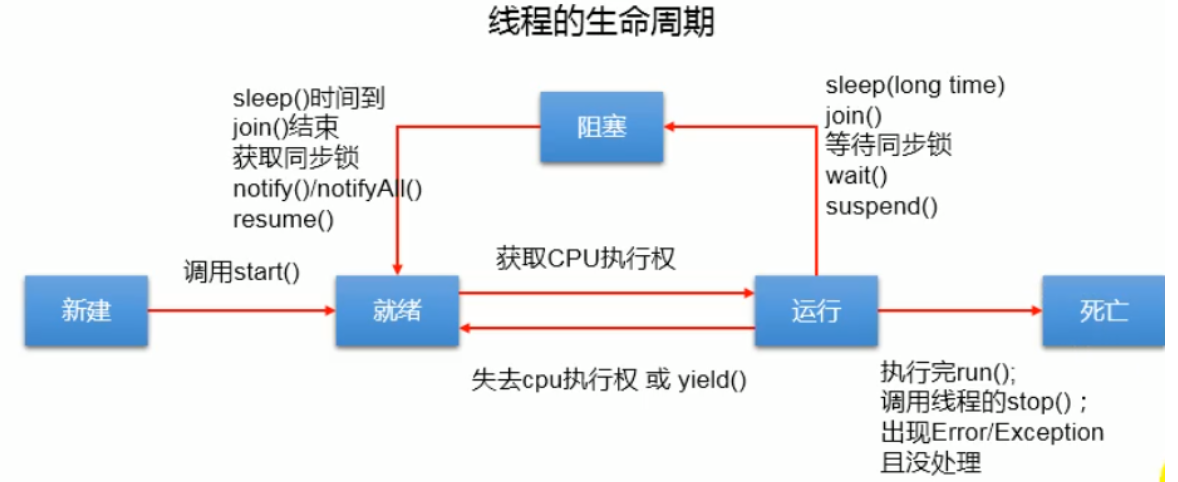
（1）新建：当一个Thread类或其子类的对象被声明并创建时。新生的线程对象属于新建状态。

（2）就绪：处于新建状态的线程执行start()方法后，进入线程队列等待CPU时间片，该状态具备了运行的状态，只是没有分配到CPU资源。

（3）运行：当就绪的线程分配到CPU资源便进入运行状态，run()方法定义了线程的操作。

（4）阻塞：在某种特殊情况下，被人为挂起或执行输入输出操作时，让出CPU并临时终止自己的的执行，进入阻塞状态。

1. 死亡：当线程执行完自己的操作或提前被强制性的终止或出现异常导致结束，会进入死亡状态。



### 3.2.4 sleep()、wait()、yield()、join()

（1）sleep()方法：不会释放对象锁

1. sleep()是Thread类中的方法；
2. sleep()方法导致了程序暂停，但是他的监控状态依然保持着，当指定的时间到了又会自动恢复运行状态。在调用sleep()方法的过程中，线程不会释放对象锁。

（2）wait()方法：释放对象锁

1. wait()是Object类中的方法；
2. wait()方法会使当前线程阻塞，直到调用notify()，则被唤醒。
3. wait()和notify()会对对象的“锁标志”进行操作，所以它们必须在synchronized函数或synchronized代码块中进行调用，否则在运行时会发生IllegalMonitorStateException的异常。

（3）yield，()方法:不会释放对象锁

1. yield()是Thread类中的方法，作用：**暂停当前正在执行的线程对象，并执行其他线程**。其实yield()方法是一种启发式的方法，会提醒调度器我愿意放弃当前的CPU资源，如果CPU资源不紧张，则会忽略这种提醒。
2. yield()不会导致线程的等待/睡眠/阻塞状态。

（4）join()方法：释放对象锁

1. join()方法是指等待调用join()方法的线程执行结束，程序才会继续执行下去，在A线程中调用了B线程的join()方法时，表示只有当B线程执行完毕时，A线程才能继续执行。
2. join方法中如果传入参数，则表示：如果A线程中掉用B线程的join(10)，则表示A线程会等待B线程执行10毫秒，10毫秒过后，A、B线程并行执行。需要注意的是，jdk规定，join(0)的意思不是A线程等待B线程0秒，而是A线程等待B线程无限时间，直到B线程执行完毕，即join(0)等价于join()。(其实join()中调用的是join(0))
3. 方法join(long)的功能在内部是使用wait(long)来实现的，所以join(long)方法具有释放锁的特点。
4. sleep()方法和yield()方法区别：

sleep 方法使当前运行中的线程睡眼一段时间，进入不可运行状态，这段时间的长短是由程序设定的，yield 方法使当前线程让出 CPU 占有权，但让出的时间是不可设定的。实际上，yield()方法对应了如下操作：先检测当前是否有相同优先级的线程处于同可运行状态，如有，则把 CPU 的占有权交给此线程，否则，继续运行原来的线程。所以yield()方法称为“退让”，它把运行机会让给了同等优先级的其他线程。

另外，sleep 方法允许较低优先级的线程获得运行机会，但 yield() 方法执行时，当前线程仍处在可运行状态，所以，不可能让出较低优先级的线程些时获得 CPU 占有权。在一个运行系统中，如果较高优先级的线程没有调用 sleep 方法，又没有受到 I\O 阻塞，那么，较低优先级线程只能等待所有较高优先级的线程运行结束，才有机会运行。

### 3.2.5 JAVA中的锁有哪些？

|  |  |
| --- | --- |
| 锁 | 解释 |
| 公平锁/非公平锁 | 公平锁是指多个线程按照申请锁的顺序来获取锁 |
| 可重入锁 | 同一线程在外层方法已获取锁时，进入内层方法会自动获取锁 |
| 独享锁/共享锁 | 一次可被单个/多个线程所持有，例如ReadWriteLock的写锁/读锁 |
| 互斥锁/读写锁 | 一种互斥锁：ReentrantLock；一种读写锁：ReadWriteLock |
| 乐观锁/悲观锁 | 悲观锁认为对于同一个数据的并发操作一定会发生修改 CAS是乐观锁 |
| 分段锁 | 一种锁的设计，具体应用有ConcurrentHashMap |
| 自旋锁 | 尝试获取锁的线程不会立即阻塞，而是采用循环的方式去尝试获取 |
| 偏向锁/轻量级锁/重量级锁 | 指锁的状态，并且是针对synchronized   * 偏向锁是指一段同步代码一直被一个线程所访问，那么该线程会自动获取锁。 * 轻量级锁是指当锁是偏向锁的时候，被另一个线程所访问，偏向锁就会升级为轻量级锁，其他线程会通过自旋的形式尝试获取锁，不会阻塞，提高性能。 |

公平锁：表示线程获取锁的顺序是按照加锁的顺序来分配的，及先来先得，先进先出的顺序。

非公平锁：表示获取锁的抢占机制，是随机获取锁的，和公平锁不一样的就是先来的不一定能拿到锁，有可能一直拿不到锁，所以结果不公平。

可重入锁：如果一个线程已经获得了锁，当执行到需要获取锁的代码段时可以直接进行，无需再次获取锁。不然他会把自己等死。重入锁的这种机制也说明了它是以”线程“为粒度获取锁，而不是以”调用“为粒度。

其实，自旋锁与互斥锁比较类似，它们都是为了解决对某项资源的互斥使用。无论是互斥锁，还是自旋锁，在任何时刻，最多只能有一个保持者，也就说，在任何时刻最多只能有一个执行单元获得锁。但是两者在调度机制上略有不同。对于互斥锁，如果资源已经被占用，资源申请者只能进入睡眠状态。但是自旋锁不会引起调用者睡眠，如果自旋锁已经被别的执行单元保持，调用者就一直循环在那里看是否该自旋锁的保持者已经释放了锁，”自旋”一词就是因此而得名。

ABA问题的根本在于cas在修改变量的时候，无法记录变量的状态，比如修改的次数，否修改过这个变量。这样就很容易在一个线程将A修改成B时，另一个线程又会把B修改成A,造成casd多次执行的问题

（3）适用场景

悲观锁：比较适合写入操作比较频繁的场景，如果出现大量的读取操作，每次读取的时候都会进行加锁，这样会增加大量的锁的开销，降低了系统的吞吐量。

乐观锁：比较适合读取操作比较频繁的场景，如果出现大量的写入操作，数据发生冲突的可能性就会增大，为了保证数据的一致性，应用层需要不断的重新获取数据，这样会增加大量的查询操作，降低了系统的吞吐量。

总结：两种所各有优缺点，读取频繁使用乐观锁，写入频繁使用悲观锁。

### 3.2.6 CAS原理及产生的问题？

（1）CAS原理

Conmpare And Swap是用于实现多线程同步的原子指令。CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当旧的预期值A和内存值V相同时，将内存值V修改为B，否则什么都不做。整个比较并替换的操作是一个原子操作。

（2）CAS算法的缺点

* **ABA问题**：如果一个变量V初次读取的时候是A值，并且在准备赋值的时候检查到它仍然是A值，那我们就能说明它的值没有被其他线程修改过了吗？很明显是不能的，因为在这段时间它的值可能被改为其他值，然后又改回A，那CAS操作就会误认为它从来没有被修改过。这个问题被称为CAS操作的 "ABA"问题。
* ABA问题的解决思路就是使用版本号。在变量前面追加上版本号，每次变量更新的时候把版本号加1，那么A－B－A 就会变成1A-2B－3A。
* **循环时间长开销大**：自旋CAS（也就是不成功就一直循环执行直到成功）如果长时间不成功，会给CPU带来非常大的执行开销。
* **只能保证一个共享变量的原子操作**：CAS只对单个共享变量有效，当操作涉及跨多个共享变量时CAS无效。但是从JDK 1.5开始，提供了AtomicReference类来保证引用对象之间的原子性，可以把多个变量放在一个对象里来进行CAS操作。

### 3.2.7 synchronized底层实现原理？

（1）对象（或代码块）的Synchronized同步

Synchronized进过编译，会在同步块的前后分别形成Monitorenter和Monitorexit这个两个字节码指令。。

* Monitorenter：

每个对象都是一个监视器锁（monitor）。当monitor被占用时就会处于锁定状态，线程执行monitorenter指令时尝试获取monitor的所有权，过程如下：

1. 如果monitor的进入数为0，则该线程进入monitor，然后将进入数设置为1，该线程即为monitor的所有者。
2. 如果线程已经占有该monitor，只是重新进入，则进入monitor的进入数加1。
3. 如果其他线程已经占用了monitor，则该线程进入阻塞状态，直到monitor的进入数为0，再重新尝试获取monitor的所有权。

* Monitorexit：

释放锁则是通过monitorexit指令，执行monitorexit的线程必须是monitor的所有者，指令执行时，monitor的进入数减1，如果减1后进入数为0，那线程退出monitor，不再是这个monitor的所有者。其他被这个monitor阻塞的线程可以尝试去获取这个 monitor 的所有权。

（2）方法的synchronized同步

加了synchronized关键字的方法，相对于普通方法，其常量池中多了ACC\_SYNCHRONIZED标示符。JVM就是据该标示符来实现方法的同步的：

1. 当方法调用时，调用指令将会检查方法的 ACC\_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先获取monitor，获取成功之后才能执行方法体，方法执行完后再释放monitor。
2. 在方法执行期间，其他任何线程都无法再获得同一个monitor对象。

### 3.2.8线程安全的本质？

“线程安全”也不是指线程的安全，而是指内存的安全。所以线程安全指的是，在堆内存中的数据由于可以被任何线程访问到，在没有限制的情况下存在被意外修改的风险。即堆内存空间在没有保护机制的情况下，对多线程来说是不安全的地方，因为你放进去的数据，可能被别的线程“破坏”。

解决办法就是加锁：https://blog.csdn.net/weixin\_45637066/article/details/101402353

1. 同步代码块
2. 同步方法
3. Lock

### 3.2.9volatile底层实现原理？

“**volatile的数组只针对数组的引用具有volatile的语义，而不是它的元素**”

* + Volatile不能保证原子性可以保证变量的可见性，“可见性”的意思是当一个线程修改一个变量时，另外一个线程能读到这个修改的值。
  + volatile修饰的变量不会被编译器优化，即不会造成指令重排；

底层原理：

通过对OpenJDK下的unsafe.cpp的源码查看，发现被volatile修饰的变量有一个“lock”的前缀，这个前缀实际上是一个内存屏障，可以提供下面几个保障：

1. 可以确保指令重排序的时候，不会将其后面的代码排到内存屏障之前；不会将其前面的代码排到内存屏障之后。
2. 确保在执行到内存屏障修饰的指令时，前面的代码全部执行完成。
3. 如果线程工作内存中值的修改了，就强制将其刷新到主内存中。
4. 如果是写操作，则会导致其他线程工作内存(CPU Cache)中的缓存数据失效。

### 3.2.10（lock接口）ReentrantLock底层实现原理？

ReentrantLock是基于AQS的（CAS+CHL），有公平和非公平两种区别，AQS使用一个int成员变量来表示同步状态，通过内置的FIFO队列来完成获取资源线程的排队工作。

ReentrantLock 构造函数中提供了两种锁：公平锁（FairSync）和非公平锁（NonfairSync、默认）。

|  |
| --- |
| Lock unfairLock = new ReentrantLock();  Lock fairLock = new ReentrantLock(true);//调用构造函数时，传入参数true得到的就是公平锁 |

* 在公平的锁上，线程按照他们发出请求的顺序获取锁，FIFO先进先出顺序，每个线程获取锁的过程是公平的，等待时间最长的会最先被唤醒获取锁。但在非公平锁上，则允许插队。
* 在公平的锁上，如果有另一个线程持有锁或者有其他线程在等待队列中等待这个锁，那么新发出的请求的线程将被放入到队列中。

而非公平锁上，只有当锁被某个线程持有时，新发出请求的线程才会被放入队列中。

ReentrantLock在采用非公平锁构造时，首先检查锁状态，如果锁可用，直接通过CAS设置成持有状态，且把当前线程设置为锁的拥有者。如果当前锁已经被持有，那么接下来进行可重入检查，如果可重入，需要为锁状态加上请求数。如果不属于上面两种情况，那么说明锁是被其他线程持有，当前线程应该放入等待队列。在放入等待队列的过程中，首先要检查队列是否为空队列，如果为空队列，需要创建虚拟的头节点，然后把对当前线程封装的节点加入到队列尾部。由于设置尾部节点采用了CAS，为了保证尾节点能够设置成功，这里采用了无限循环的方式，直到设置成功为止。

### 3.2.11synchronized、volatile、ReentrantLock区别？

Synchronized和Volatile的区别？

volatile本质是在告诉jvm当前变量在寄存器（工作内存）中的值是不确定的，需要从主存中读取； synchronized则是锁定当前变量，只有当前线程可以访问该变量，其他线程被阻塞住。

* + 使用上：volatile仅能修饰实例变量或者类变量；synchronized可以修饰方法、类、代码块，不能修饰变量。
  + 原子性可见性：volatile可以保证变量的修改可见性，不能保证原子性；而synchronized可以保证变量的修改可见性和原子性。
  + 指令重排：volatile禁止编译器对其进行指令重排，可以保证有序性；虽然synchronized关键字所修饰的同步方法也可以保证顺序性，但是这种顺序性是以程序的串行化执行换来的，在synchronized关键字所修饰的代码块中代码指令也会发生指令重排序的情况。
  + volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞。

Sychronized和ReentrantLock区别：

相似点：

两种同步方式有很多相似之处，它们都是加锁方式同步，而且都是阻塞式的同步，也就是说当如果一个线程获得了对象锁，进入了同步块，其他访问该同步块的线程都必须阻塞在同步块外面等待。

ReenTrantLock的字面意思就是再进入的锁，其实synchronized关键字所使用的锁也是可重入的。两者都是同一个线程每进入一次，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。

区别：

1. Synchronized是依赖于JVM实现的，而ReenTrantLock是JDK实现的，就类似于操作系统来控制实现和用户自己敲代码实现的区别。
2. Synchronized的使用比较方便简洁，并且由编译器去保证锁的加锁和释放，而ReenTrantLock需要手工声明来加锁和释放锁，为了避免忘记手工释放锁造成死锁，所以要在finally中声明释放锁（unlock()）。
3. ReenTrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。

### 3.2.12Synchronized加静态和非静态方法上？

我们可以知道锁的最小粒度是线程，这点很重要。

Synchronized是可重入锁。

（1）Synchronzied 修饰非静态方法实际上是对调用该方法的对象加锁，俗称“对象锁”。

（2）Synchronzied 修饰静态方法，实际上是对该类对象加锁，称为类锁；

### 3.2.13TreadLocal*分析*

ThreadLocal 是 JDK java.lang 包中的一个用来实现相同线程数据共享不同的线程数据隔离的一个工具。主要用于将私有线程和该线程存放的副本对象做一个映射，各个线程之间的变量互不干扰，在高并发场景下，可以实现无状态的调用，适用于各个线程不共享变量值的操作。

工作原理：每个Thread线程内部都有一个Map。Map里面存储线程本地对象（key）和线程的变量副本（value）

Thread内部的Map是由ThreadLocal维护的，由ThreadLocal负责向map获取和设置线程的变量值。对于不同的线程，每次获取副本值时，别的线程并不能获取到当前线程的副本值，形成了副本的隔离，彼此之间互不干扰。

ThreadLocal 如何解决 Hash 冲突？

与 HashMap 不同，ThreadLocalMap 结构非常简单，没有 next 引用，也就是说 ThreadLocalMap 中解决 Hash 冲突的方式并非链表的方式，而是采用线性探测的方式。所谓线性探测，就是根据初始 key 的 hashcode 值确定元素在 table 数组中的位置，如果发现这个位置上已经被其他的 key 值占用，则利用固定的算法寻找一定步长的下个位置，依次判断，直至找到能够存放的位置。

ThreadLocal 的内存泄露是怎么回事？

ThreadLocal 在 ThreadLocalMap 中是以一个弱引用身份被 Entry 中的 Key 引用的，因此如果 ThreadLocal 没有外部强引用来引用它，那么 ThreadLocal 会在下次 JVM 垃圾收集时被回收。这个时候 Entry 中的 key 已经被回收，但是 value 又是一强引用不会被垃圾收集器回收，这样 ThreadLocal 的线程如果一直持续运行，value 就一直得不到回收，这样就会发生内存泄露。

为什么 ThreadLocalMap 的 key 是弱引用？

我们知道 ThreadLocalMap 中的 key 是弱引用，而 value 是强引用才会导致内存泄露的问题，至于为什么要这样设计，这样分为两种情况来讨论：

key 使用强引用：这样会导致一个问题，引用的 ThreadLocal 的对象被回收了，但是 ThreadLocalMap 还持有 ThreadLocal 的强引用，如果没有手动删除，ThreadLocal 不会被回收，则会导致内存泄漏。

key 使用弱引用：这样的话，引用的 ThreadLocal 的对象被回收了，由于 ThreadLocalMap 持有 ThreadLocal 的弱引用，即使没有手动删除，ThreadLocal 也会被回收。value 在下一次 ThreadLocalMap 调用 set、get、remove 的时候会被清除。

比较以上两种情况，我们可以发现：由于 ThreadLocalMap 的生命周期跟 Thread 一样长，如果都没有手动删除对应 key，都会导致内存泄漏，但是使用弱引用可以多一层保障，弱引用 ThreadLocal 不会内存泄漏，对应的 value 在下一次 ThreadLocalMap 调用 set、get、remove 的时候被清除，算是最优的解决方案。

ThreadLocal 的应用场景有哪些？

ThreadLocal 适用于独立变量副本的情况，比如 Hibernate 的 session 获取场景。

### 3.2.14 CountDownLatch的使用

Java的concurrent包里面的CountDownLatch其实可以把它看作一个计数器，只不过这个计数器的操作是原子操作，同时只能有一个线程去操作这个计数器，也就是同时只能有一个线程去减这个计数器里面的值。

你可以向CountDownLatch对象设置一个初始的数字作为计数值，任何调用这个对象上的await()方法都会阻塞，直到这个计数器的计数值被其他的线程减为0为止。

CountDownLatch的一个非常典型的应用场景是：有一个任务想要往下执行，但必须要等到其他的任务执行完毕后才可以继续往下执行。假如我们这个想要继续往下执行的任务调用一个CountDownLatch对象的await()方法，其他的任务执行完自己的任务后调用同一个CountDownLatch对象上的countDown()方法，当子线程执行完成后，在 finnaly 块调用 countDown() 方法，表示一个等待已经完成，把计数器减一，直到减为0。

### 3.2.15使用线程池的好处、四种常见的线程池？

1. 线程的复用：线程的创建和销毁对系统来说是巨大的开销，而用线程池管理线程能大大减少了这种不必要的开销。
2. 控制线程的并发数：控制线程池中线程的并发数，可以防止大量线程争夺CPU资源而造成的堵塞。比如用

FixedThreadPool 可以控制线程的最大并发数，超出部分在队列中等待。

1. 可以对线程进行管理：线程池可以提供定时、定期，使用

ScheduledThreadPool线程池来执行S秒后，每隔N秒执行一次的任务。

使用CachedThreadPool不需要制定线程数量，可以灵活创建线程。空闲线程由JVM回收。

使用SingleThreadExecutor 来实现单线程化的线程。

四种线程池：

1. newFixedThreadPool 定长线程池：创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。
2. newCachedThreadPool 可缓冲线程池：创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。这边有一个超时机制，当空闲的线程超过60s内没有用到的话，就会被回收，它可以一定程序减少频繁创建/销毁线程,减少系统开销，适用于执行时间短并且数量多的任务场景。
3. ScheduledThreadPool 周期线程池：创建一个定长线程池，支持定时及周期性任务执行，通过过schedule方法可以设置任务的周期执行。核心线程池固定，大小无限的线程池。
4. newSingleThreadExecutor 单任务线程池：创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行，每次任务到来后都会进入阻塞队列，然后按指定顺序执行。

ExecutorService singleThreadExecutor = Executors.newSingleThreadExecutor();

### 线程池的核心参数、缓存队列BlockingQueue？

corePoolSize 线程池中核心线程的数量

maximumPoolSize线程池中最大线程数量

keepAliveTime非核心线程的超时时长，当系统中非核心线程闲置时间超过keepAliveTime之后，则会被回收。如果ThreadPoolExecutor的allowCoreThreadTimeOut属性设置为true，则该参数也表示核心线程的超时时长。

unit第三个参数的单位，有纳秒、微秒、毫秒、秒、分、时、天等。

workQueue线程池中的任务队列，该队列主要用来存储已经被提交但是尚未执行的任务。存储在这里的任务是由ThreadPoolExecutor的execute方法提交来的。

threadFactory为线程池提供创建新线程的功能，这个我们一般使用默认即可

handler拒绝策略，当线程无法执行新任务时（一般是由于线程池中的线程数量已经达到最大数或者线程池关闭导致的），默认情况下，当线程池无法处理新线程时，会抛出一个RejectedExecutionException。

其中workQueue ：它决定了缓存任务的排队策略。对于不同的应用场景我们可能会采取不同的排队策略，这就需要不同类型的队列。这个队列需要一个实现了BlockingQueue接口的任务等待队列。

ArrayBlockingQueue：基于数组的阻塞队列实现，在ArrayBlockingQueue内部，维护了一个定长的数组，以便缓存队列中的数据对象，其内部没实现读写分离，也就意味着生产和消费者不能完全并行。长度是需要定义的，可以指定先进先出或者先进后出，因为长度是需要定义的，所以也叫有界队列，在很多场合非常适合使用。

LinkedBlockingQueue：基于链表的阻塞队列，同ArrayBlockingQueue类似，其内部也维持着一个数据缓冲队列（该队列由一个链表构成），LinkedBlockingQueue之所以能够高效地处理并发数据，是因为其内部实现采用分离锁（读写分离两个锁），从而实现生产者和消费者操作完全并行运行。需要注意一下，它是一个无界队列。

SynchronousQueue：一种没有缓冲的队列，生产者产生的数据直接会被消费者获取并且立刻消费。

PriorityBlockingQueue：基于优先级别的阻塞队列（优先级的判断通过构造函数传入的Compator对象来决定，也就是说传入队列的对象必须实现Comparable接口），在实现PriorityBlockingQueue时，内部控制线程同步的锁采用的是公平锁，需要注意的是它也是一个无界的队列。

DelayQueue：带有延迟时间的Queue，其中的元素只有当其指定的延迟时间到了，才能够从队列中获取到该元素。

### 3.2.17线程池的拒绝策略

当提交任务数大于 corePoolSize 的时候，会优先将任务放到 workQueue 阻塞队列中。当阻塞队列饱和后，会扩充线程池中线程数，直到达到 maximumPoolSize 最大线程数配置。此时，再多余的任务，则会触发线程池的拒绝策略了。

总结起来，也就是一句话，当提交的任务数大于（workQueue.size() + maximumPoolSize ），就会触发线程池的拒绝策略。

四种拒绝策略：

1. CallerRunsPolice：当触发拒绝策略，只要线程池没有关闭的话，则使用调用线程直接运行任务。，别的任务只能在被拒绝的任务执行完之后才会继续被提交到线程池执行。一般并发比较小，性能要求不高，不允许失败。但是，由于调用者自己运行任务，如果任务提交速度过快，可能导致程序阻塞，性能效率上必然的损失较大
2. AbortPolicy：丢弃任务，并抛出拒绝执行 RejectedExecutionException 异常信息。线程池默认的拒绝策略。必须处理好抛出的异常，否则会打断当前的执行流程，影响后续的任务执行。
3. DiscardPolicy：直接丢弃，其他啥都没有。
4. DiscardOldestPolicy：当触发拒绝策略，只要线程池没有关闭的话，丢弃阻塞队列 workQueue 中最老的一个任务，并将新任务加入。

RejectedExecutionHandler rejected = null;

rejected = new ThreadPoolExecutor.AbortPolicy();//默认，队列满了丢任务抛出异常

rejected = new ThreadPoolExecutor.DiscardPolicy();//队列满了丢任务不异常

rejected = new ThreadPoolExecutor.DiscardOldestPolicy();//将最早进入队列的任务删，之后再尝试加入队列

rejected = new ThreadPoolExecutor.CallerRunsPolicy();//如果添加到线程池失败，只用调用者所在线程来运行任务；

### 线程池大小应该怎么确定？

有种简单的估算方式，设N为CPU个数

对于CPU密集型的应用，线程池的大小设置为N+1

对于I/O密集型的应用，线程池的大小设置为2N+1

这种设置方式适合于一台机器上的应用的类型是单一的，并且只有一个线程池，实际情况还需要根据实际的应用进行验证。

在I/O优化中，以下的估算公式可能更合理：

最佳线程数量 = （（线程等待时间+线程CPU时间）／ 线程CPU时间）\* CPU个数

由公式可得，线程等待时间所占比例越高，需要越多的线程。

线程CPU时间所占比例越高，所需的线程数越少。

### 3.2.18线程池都有哪些状态?

1. RUNNING：这是最正常的状态，接受新的任务，处理等待队列中的任务。
2. SHUTDOWN：不接受新的任务提交，但是会继续处理等待队列中的任务。
3. STOP：不接受新的任务提交，不再处理等待队列中的任务，中断正在执行任务的线程。
4. TIDYING（tidying）：所有的任务都已终止，任务数量为 0，线程池的状态在转换为 TIDYING 状态时，会执行函数terminated()。
5. TERMINATED(terminated)：terminated()方法结束后，线程池的状态就是这个状态。

### 3.2.19确保N个线程可以访问N个资源同时又不导致死锁？

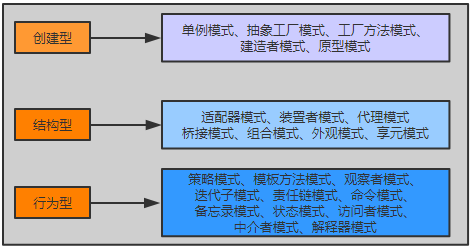
使用多线程的时候， 一种非常简单的避免死锁的方式就是：指定获取锁的顺序，并强制线程按照指定的顺序获取锁。因此，如果所有的线程都是以同样的顺序加锁和释放锁，就不会出现死锁了。

* 1.加锁顺序（线程按照一定的顺序加锁）
* 2.加锁时限（线程尝试获取锁的时候加上一定的时限，超过时限则放弃对该锁的请求，并释放自己占有的锁）
* 3.死锁检测（主要是针对那些不可能实现按序加锁并且锁超时也不可行的场景）

参考网址<https://blog.csdn.net/ls5718/article/details/51896159>

## 3.3设计模式

### 3.3.1三种设计模式



### 3.3.2设计模式遵循6大原则

* 开闭原则：对扩展开放，对修改关闭。
* 里氏代换原则：只有当衍生类可以替换掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加新的行为。
* 依赖倒转原则：这个是开闭原则的基础，对接口编程，依赖于抽象而不依赖于具体。
* 接口隔离原则：使用多个隔离的借口来降低耦合度。
* 迪米特法则（最少知道原则）：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。
* 合成复用原则：原则是尽量使用合成/聚合的方式，而不是使用继承。继承实际上破坏了类的封装性，超类的方法可能会被子类修改。

### 3.3.2八种设计模式

// 单例设计模式

懒汉模式

饿汉模式

双重锁机制

// 工厂设计模式

interface

实现interface的具体类，具有自己的方法

定义一个工厂类，里面进行创建类，一个工厂方法 多个工厂方法，还有静态方法

// 抽象工厂设计模式

具体类的的接口

工厂的接口

工厂类的接口，定义了多个工厂类来创建对象

// 建造者模式

单车对象bean

定义了单车的接口

具体的摩拜单车，实现了接口，将对象的属性进行赋值

工程部，注入了单车的接口，也可以理解为生产线，然后定义了单车的生产顺序，对单车进行设计

// 适配器模式

1 、 类适配器模式 定义一个类的方法 然后定义一个接口包含两个方法，然后实现接口继承类

对类中的方法进行重写。

2 、 对象适配器 定义一个类的方法 然后定义一个接口包含两个方法，然后继承接口，在类中

新建了一个类的对象，然后类的对象应用类中有的方法中。

3 、 接口适配器 定义一个接口，然后再定义一个抽象类来实现它，如果你要什么功能，自己定义

类来对相应的方法进行重写。

// 装饰器模式

定义一个接口，然后定义一个类实现这个接口，

定义一个装饰器，将定义的接口写进来

再定义一个具体对这个方法的装饰器，继承上面的装饰器。

// 策略模式

定义一个接口，用不同的类来实现接口，并利用不同的算法来重写里面的方法

然后再定义一个上下文，进行调用不同类中的方法。（这里的多态用的真的不错）

// 代理模式

静态代理

定义一个接口，然后定义了一个类，在代理类来继承这个类，并对其中的方法进行改进

动态代理

定义一个接口，然后定义了一个类A，在定义一个代理类实现了InvocationHandler方法，

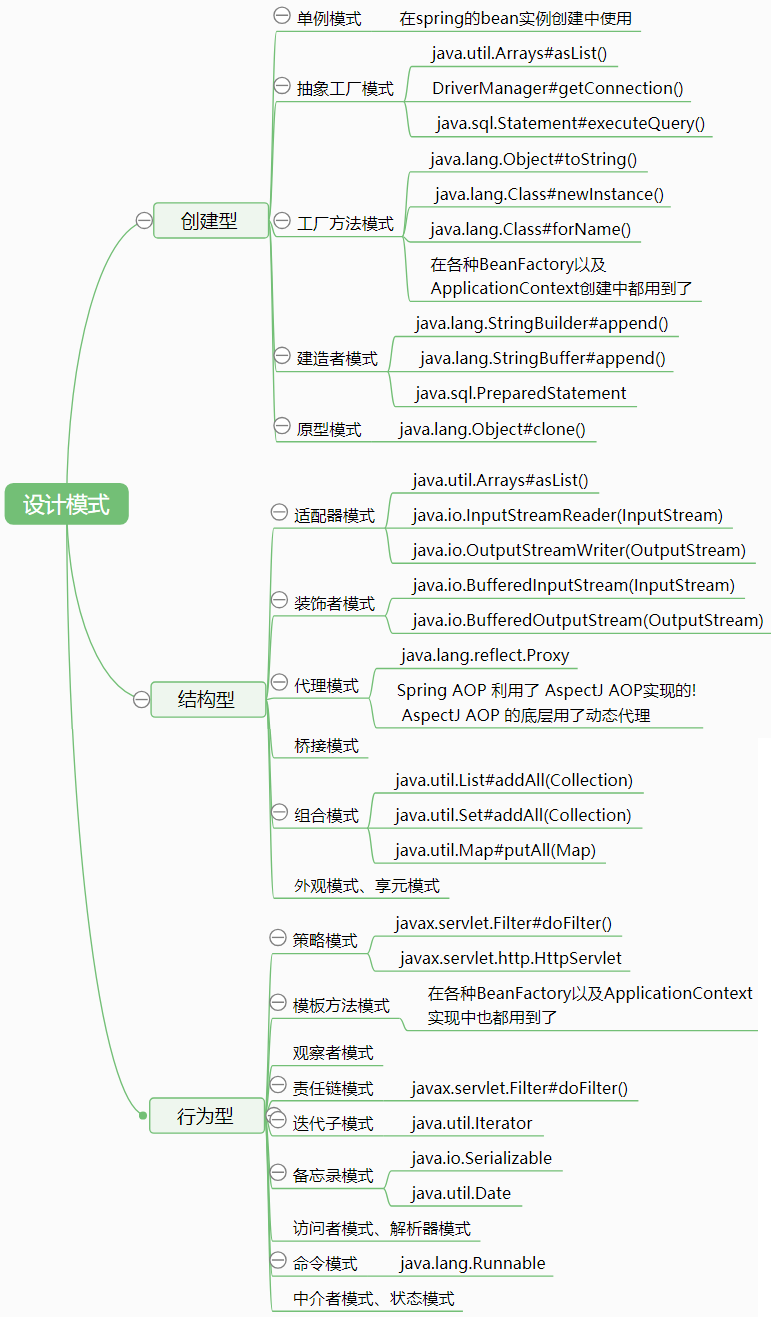
其中的要调用math.invoke()方法

调用

先创建类A，再创建代理类的对象，将类放在参数里面，然后proxy代理，有三个参数，

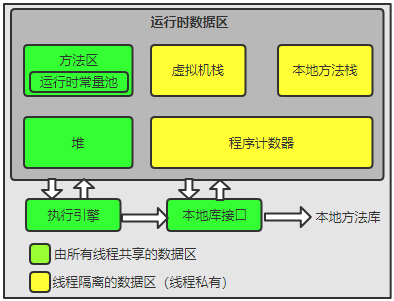
第一个类的加载，handler.getClass().getClassLoder(); A.getClass.getInterfance();

handler;



## 3.4java虚拟机

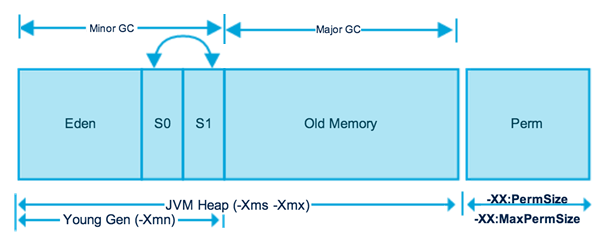
### 3.4.1 JVM内存模型（区域）



* 程序计数器：占用内存空间小，通过改变这个计数器的值来选取下一条需要执行的字节码指令，唯一一个在java虚拟机规范中没有规定任何OutOfMemoryError情况的区域；
* 虚拟机栈：每个方法在执行的同时都会创建一个栈桢(Stack Frame)用于存储局部变量，操作数栈，动态链接，方法出口等信息；局部变量表存放了编译期间可知的各种基本数据类，对象引用类型，它不等同于对象本身和returnAddress类型(一条指向字节码指令的地址)。
* 本地方法栈：为虚拟机使用到的Native方法服务。
* 堆：在虚拟机启动时创建；所有的对象实例和数组都要在堆上分配。（大小由-Xmx和-Xms控制）
* 方法区：存储已被虚拟机加载的类信息，常量，静态变量，即时编译器便后的代码等数据。
* 运行时常量池：Class文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池(Constant Pool Table)，用于存放编译期间生成的各种字面量和符号引用。这部分内容将在类加载后进入方法区的运行时常量池中存放。具备动态性，运行期间也可能将新常量放入池中比如String类的intern()方法。
* 直接内存：不是虚拟机运行时数据区的一部分，也不是Java虚拟机规范中定义的内存区域；NIO(New Input/Output)类，引入了一种基于通道(Channel)与缓冲区(Buffer)de I/O方式，它可以使用Native函数库直接分配堆外内存，然后通过一个存储在Java堆中的DirectByteBuffer对象作为这块内存的引用进行操作，来避免在Java堆和Native堆中来回复制数据。直接内存的分配不会受到Java堆大小的限制，但是会受到本机总内存大小以及处理器寻址空间的限制，因此也可能导致OutOfMemoryError。

### 3.4.2 JAVA堆内存参数

* Java堆（Java Heap）是java虚拟机所管理的内存中最大的一块；
* java堆被所有线程共享的一块内存区域；
* 虚拟机启动时创建java堆；
* java堆的唯一目的就是存放对象实例；
* java堆是垃圾收集器管理的主要区域；
* 从内存回收的角度来看，由于现在收集器基本都采用分代收集算法，所以Java堆可以细分为：新生代（Young）和老年代（Old）。新生代又被划分为三个区域Eden、From Survivor，To Survivor等。无论怎么划分，最终存储的都是实例对象，进一步划分的目的是为了更好的回收内存，或者更快的分配内存；
* java堆的大小是可扩展的，通过-Xmx和-Xms控制；
* 如果堆内存不够分配实例对象， 并且对也无法在扩展时，将会抛出outOfMemoryError异常。



### 3.4.3 MinorGC和FullGC的触发情况

FullGC的触发情况

1. System.gc()方法的调用
2. 老年代代空间不足
3. 永生区空间不足

JVM规范中运行时数据区域中的方法区，在HotSpot虚拟机中又被习惯称为永生代或者永生区，Permanet Generation中存放的为一些class的信息、常量、静态变量等数据，当系统中要加载的类、反射的类和调用的方法较多时，Permanet Generation可能会被占满。

1. 通过Minor GC后进入老年代的平均大小大于老年代的可用内存

在进行Minor GC时，做了一个判断，如果之前统计所得到的Minor GC晋升到旧生代的平均大小大于旧生代的剩余空间，那么就直接触发Full GC。例如程序第一次触发Minor GC后，有6MB的对象晋升到旧生代，那么当下一次Minor GC发生时，首先检查旧生代的剩余空间是否大于6MB，如果小于6MB，则执行Full GC。

1. 堆中分配很大的对象

由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小。

MinorGC的触发情况：  
 当年轻代满时就会触发Minor GC，这里的年轻代满指的是Eden代满，Survivor满不会引发GC。

### 3.4.4 新生代对象转移到老年代的触发条件？

1. Eden区满时，进行Minor GC，当Eden和一个Survivor区中依然存活的对象无法放入到Survivor中，则通过分配担保机制提前转移到老年代中。
2. 若对象体积太大，新生代无法容纳这个对象，就会绕过新生代，直接在老年代分配。
3. 长期存活的对象将进入老年代。

虚拟机对每个对象定义了一个对象年龄（Age）计数器。当年龄增加到一定的临界值时，就会晋升到老年代中。如果对象在Eden出生并在第一次发生MinorGC时仍然存活，并且能够被Survivor中所容纳的话，则该对象会被移动到Survivor中，并且设Age=1；以后每经历一次Minor GC，该对象还存活的话Age=Age+1。

1. 动态对象年龄判定。

虚拟机并不总是要求对象的年龄必须达到设定的年龄（MaxTenuringThreshold）才能晋升到老年代，如果在Survivor区中相同年龄（设年龄为age）的对象的所有大小之和超过Survivor空间的一半，年龄大于或等于该年龄（age）的对象就可以直接进入老年代，无需等到MaxTenuringThreshold中要求的年龄。

### 3.4.5 怎样判断对象是否会被回收？

1. 引用计数算法

给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器为0的对象就是不可能再被使用的。JVM里面并没有选用引用计数算法来管理内存，主要原因是它很难解决对象之间相互循环引用的问题。

1. 可达性分析算法（JVM使用的）

通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链(Reference Chain)，当一个对象到GC Roots没有任何引用链相连时（也就是从GC Roots到这个对象不可达时），则证明此对象是不可用的。

在java语言中，可以作为GC Roots的对象包括以下几种：

1. 虚拟机栈（栈帧中的本地变量表）中引用的对象；
2. 方法区中类静态属性引用的对象；
3. 方法区中常量引用的对象；
4. 本地方法栈中JNI（即一般说的Native方法）引用的对象；

### 3.4.6 垃圾回收算法

（1）标记-清除算法

最基础的垃圾收集器算法，分为“标记”和“清除”两个阶段，先标记处所需要回收的对象，标记完成后，统一回收掉所有被标记的对象。

缺点：

* 效率问题，标记和清除的效率不高。
* 清除后会产生大量的不连续的内存碎片，可能会导致在程序需要为较大对象分配内存时无法找到足够连续的内存，不得不提前触发垃圾收集动作。

（2）复制算法（Serial收集器）

将内存容量分成大小相等的两块，每次只使用其中一块，当一块用完时，将还存活的对象复制到另一块去，然后把之前使用满的那块空间一次性清理掉，如此反复。

* 优点：内存分配的时候不用考虑内存碎片问题，只移动堆顶指针，按顺序分配即可，简单高效。
* 缺点：内存空间浪费大，每次只能使用当前内存空间的一半；当对象存活率较高时，需要有大量的复制操作，效率低。

（3）标记-整理（Serial Old收集器）

标记-整理 是在 标记-清除 基础上改进得来。有标记阶段，标记出所有需要回收的对象，但是不会直接清理，而是将存活的对象向一端移动，在移动过程中清理掉可回收对象。这样就解决了内存碎片的问题。

* 优点：解决了之前内存碎片的问题，特别是在存活率高的时候，效率远高于复制算法。

1. 分代收集算法

根据内存对象的存活周期不同，将内存划分成几块，java虚拟机中一般将内存划分成新生代和老年代，当新建对象时一般在新生代中分配内存，在新生代垃圾收集器回收几次后仍然存活的对象，将被移动到老年代，或者当大的对象在新生代中无法分配到足够连续的内存空间时也会直接分配到老年代。

### 3.4.7 垃圾回收器

1. CMS收集器

CMS (Concurrent Mark Swep)收集器是一个比较重要的回收器，现在应用非常广泛，CMS是一种获取最短回收停顿时间为目标的收集器，这使得它很适合用于和用户交互的业务。从名字(Mark Swep)就可以看出，CMS收集器是基于标记清除算法实现的（用于老年代）。它的收集过程分为四个步骤：

* 初始标记(initial mark)：独占PUC，仅标记GCroots能直接关联的对象；
* 并发标记(concurrent mark)：可以和用户线程并行执行，标记所有可达对象；
* 重新标记(remark)：独占CPU(STW)，对并发标记阶段用户线程运行产生的垃圾对象进行标记修正；
* 并发清除(concurrent sweep)：可以和用户线程并行执行，清理垃圾；

注意初始标记和重新标记还是会stop the world，但是在耗费时间更长的并发标记和并发清除两个阶段都可以和用户进程同时工作。

优点：并发，低停顿

缺点：

* + 对CPU非常敏感：在并发阶段虽然不会导致用户线程停顿，但是会因为占用了一部分线程使应用程序变慢；
  + 无法处理浮动垃圾：在最后一步并发清理过程中，用户线程执行也会产生垃圾，但是这部分垃圾是在标记之后，所以只有等到下一次gc的时候清理掉，这部分垃圾叫浮动垃圾；
  + CMS使用“标记-清除”法会产生大量的空间碎片，当碎片过多，将会给大对象空间的分配带来很大的麻烦，往往会出现老年代还有很大的空间但无法找到足够大的连续空间来分配当前对象，不得不提前触发一次FullGC，为了解决这个问题CMS提供了一个开关参数，用于在CMS顶不住，要进行FullGC时开启内存碎片的合并整理过程，但是内存整理的过程是无法并发的，空间碎片没有了但是停顿时间变长了；

1. Parallel Scavenge + Parallel Old（JDK1.8使用的垃圾收集器）

* Parallel Scavenge：是新生代收集器，采用复制算法的收集器，又是并行的多线程收集器。它的目标是达到一个可控制的吞吐量。虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量，Parallel Scavenge就可以实现这种自适应调整策略。
* Parallel Old：是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。

1. G1收集器（JDK9）

G1收集器是一款面向服务端应用的垃圾收集器。HotSpot团队赋予它的使命是在未来替换掉 JDK1.5中发布的CMS收集器。与其他GC收集器相比，G1具备如下特点：

* 并行与并发：G1能更充分的利用CPU、多核环境下的硬件优势来缩短stop the world的停顿时间。
* 分代收集：和其他收集器一样，分代的概念在G1中依然存在，不过G1不需要其他的垃圾回收器的配合就可以独自管理整个GC堆，可以自己独自管理新生代和老年代。
* 空间整合：由于G1使用了独立区域（Region）概念，G1从整体来看是基于“标记-整理”算法实现收集，从局部（两个Region）上来看是基于“复制”算法实现的，但无论如何，这两种算法都意味着G1运作期间不会产生内存空间碎片。
* 可预测的非停顿：这是G1相对于CMS的另一大优势，降低停顿时间是G1和CMS共同的关注点，但G1除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过 N毫秒。

与其它收集器相比，G1变化较大的是它将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留了新生代和来年代的概念，但新生代和老年代不再是物理隔离的了它们都是一部分Region（不需要连续）的集合。同时，为了避免全堆扫描，G1使用了Remembered Set来管理相关的对象引用信息。当进行内存回收时，在GC根节点的枚举范围中加入Remembered Set即可保证不对全堆扫描也不会有遗漏了。

### 3.4.8 java四种引用类型？

强引用：当我们使用new创建对象时，被创建的对象就是强引用，如Object object = new Object()，其中的object就是一个强引用了。如果一个对象具有强引用，JVM就不会去GC它，JVM宁可会报OOM来终止程序，也不回收该对象。

软引用: 如果一个对象只具备软引用，如果内存空间足够，那么JVM就不会GC它，如果内存空间不足了，就会GC该对象。

弱引用: 如果一个对象只具有弱引用，只要JVM的GC线程检测到了，就会立即回收。弱引用的生命周期要比软引用短很多。不过，如果垃圾回收器是一个优先级很低的线程，也不一定会很快就会释放掉软引用的内存。

虚引用：如果一个对象只具有虚引用，那么它就和没有任何引用一样，随时会被JVM当作垃圾进行GC。

上面的四种引用对应的是new关键字以及java.lang.ref包中的SoftReference，WeakReference, PhantomReference。我们注意到在java.lang.ref包中，还存在一个类叫做ReferenceQueue。

### 3.4.9 类加载器双亲委派机制

* 什么是双亲委派模型(Parent-Delegation Model)？

JVM中加载类机制采用的是双亲委派模型：当一个类加载器收到一个类加载的请求，它首先会将该请求委派给父类加载器去加载，每一个层次的类加载器都是如此，因此所有的类加载请求最终都应该被传入到顶层的启动类加载器(Bootstrap ClassLoader)中，只有当父类加载器反馈无法完成这个类的加载请求时（它的搜索范围内不存在这个类），子类加载器才尝试加载。



* 启动类加载器：负责加载虚拟机的核心类库。
* 扩展类加载器：负责加载 JAVA\_HOME\lib\ext 目录中的类库。
* 应用程序类加载器：负责加载用户路径（classpath）上的类库。
* 用户自定义的类加载器：1. java.lang.ClassLoader的子类。2. 用户可以定制类的加载方式。
* 为什么使用双亲委派模型？
* 1可以避免重复加载，父类已经加载了，子类就不需要再次加载；
* 2更加安全，很好的解决了各个类加载器的基础类的统一问题，如果不使用该种方式，那么用户可以随意定义类加载器来加载核心类，会带来相关隐患。

例如类java.lang.Object，它存放在rt.jar中，无论哪个类加载器要加载这个类，最终都会委派给启动类加载器进行加载，因此Object类在程序的各种类加载器环境中都是同一个类。相反，如果用户自己写了一个名为java.lang.Object的类，并放在程序的Classpath中，那系统中将会出现多个不同的Object类，java类型体系中最基础的行为也无法保证，应用程序也会变得一片混乱。；

* 缓存机制

缓存机制将会保证所有加载过的Class都会被缓存，当程序中需要使用某个Class时，类加载器先从缓存区寻找该Class，只有缓存区不存在，系统才会读取该类对应的二进制数据，并将其转换成Class对象，存入缓存区。这就是为什么修改了Class后，必须重启JVM，程序的修改才会生效。

### 3.4.10 JDBC、Tomcat打破双亲委派模型

Tomcat打破双亲委派模型的原因

Tomcat是一个servlet容器，他也是Java编写的，需要运行在JVM之上。我们编写的服务端程序即servlet部署在tomcat中才是完整的服务端程序（tomcat完成了与客户端通信、管理servlet生命周期等功能）。一般生产环境下，一个Tomcat中会运行着多个servlet，这个时候我们就不应该使用双亲模型了。原因是多个servlet应该是完全隔离的，而使用双亲模型，多个服务器程序就是共用一套类库了，这样各种错误都来了。

正确的做法是，每个部署在tomcat中的servlet都有自己的WebAppClassLoader，加载类时直接使用它加载，这样各个servlet加载的类互不干涉。

一个Tomcat启动会伴随一个JVM进程的启动（因为Tomcat也是Java编写的），多个部署在Tomcat的应用可以互不干涉的运行在这一个JVM中，正式因为破坏了双亲委托机制。

JDBC打破双亲委派模型的原因

JDBC的Driver接口定义在JDK中，但是其具体实现由各个数据库的服务商来提供，比如MySQL驱动包。

DriverManager 类位于 $JAVA\_HOME中jre/lib/rt.jar 包，由BootStrap类加载器加载，而其具体Driver实现类是位于服务商提供的 Jar 包。根据类加载机制，当被装载的类引用了另外一个类的加载时候，虚拟机就会使用装载第一个类的类装载器装载被引用的类，也就是说BootStrap类加载器还要去加载jar包中的Driver接口的实现类，但BootStrap类加载器默认只负责加载 $JAVA\_HOME中jre/lib里的class，所以又需要由子类加载器去加载Driver实现，这就破坏了双亲委派模型。

### 3.4.11 JVM类的加载过程

类的加载机制分为三个部分：加载、连接（验证、准备、解析）、初始化；

（1）加载：查找并加载类的二进制数据

* 通过类的全限定名来获取定义此类的二进制字节流；
* 将这个类字节流代表的静态存储结构转为方法区的运行时数据结构，在堆中生成一个代表此类的java.lang.Class对象，作为访问方法区这些数据结构的入口。
* 这个过程主要就是类加载器完成。

（2）连接

* + 验证：确保被加载的类的正确性；
  + 准备：为类的静态变量分配内存，并将其初始化为默认值；
  + 解析：把常量池内的符号引用替换为直接引用

（3）初始化：为类的静态变量赋予正确的初始值

### 3.4.12 内存屏障

为什么会有内存屏障？

首先是可见性问题：假设有两个线程A、B分别在两个不同的CPU上运行，它们共享同一个变量X。如果线程A对X进行修改后，并没有将X更新后的结果同步到主内存，则变量X的修改对B线程是不可见的。这样就会造成可见性问题

然后是重排序问题：假设A、B两个线程共享两个变量X、Y，A和B分别在不同的CPU上运行。在A中先更改变量X的值放到高速缓存区，然后再更改变量Y的值放到高速缓存区。这时有可能发生Y的值被同步回主内存，而X的值没有同步回主内存的情况，此时对于B线程来说是无法感知到X变量被修改的，或者可以认为对于B线程来说，Y变量的修改被重排序到了X变量修改的前面

就是为了解决上面的多线程里面的可见性和重序性问题，所以有了下面的内存屏障技术。

内存屏障是什么？

* 硬件层的内存屏障分为两种：Load Barrier 和 Store Barrier即读屏障和写屏障。
* 内存屏障有两个作用：
* 阻止屏障两侧的指令重排序；
* 强制把写缓冲区/高速缓存中的脏数据等写回主内存，让缓存中相应的数据失效。
* 对于Load Barrier来说，在指令前插入Load Barrier，可以让高速缓存中的数据失效，强制从新从主内存加载数据；
* 对于Store Barrier来说，在指令后插入Store Barrier，能让写入缓存中的最新数据更新写入主内存，让其他线程可见。

### 3.4.13 内存溢出（oom）、内存泄漏

（1）内存溢出和内存泄漏的区别？

内存泄漏是导致内存溢出的原因之一；

内存泄漏积累起来将导致内存溢出；

内存泄漏可以通过完善代码来避免；

内存溢出可以通过调整配置来减少发生的频率，但是无法彻底避免；

（2）可造成内存溢出情况：

* 堆上没有足够的内存可以完成对象的分配，并且堆没有办法扩展的时候-->OutOfMemoryError
* 方法区无法满足内存分配需求时-->OutOfMemoryError
* 虚拟机栈（本地方法栈）扩展时无法申请足够的内存-->OutOfMemoryError

（3）可造成内存泄漏情况：

* + 程序动态的分配了内存，但在程序结束时没有释放这部分内存，导致那一部分内存不可用。
  + 当被分配的对象可达，但是已经没有作用时，比如Student s1 = new Student1();Student s2 = new Student2();然后将s1,s2放入ArrayList中后，将s1,s2设成null，但是这两个对象所占的内存并没有释放，因为ArrayList里还存放着对象的引用；

（4）如何避免内存泄漏、溢出？

* 尽早释放无用对象的引用。
* 使用临时变量的时候，让引用变量在退出活动域后自动设置为null，暗示垃圾收集器来收集该对象，防止发生内存泄露。
* 程序进行字符串处理时,尽量避免使用String，而应使用StringBuffer，因为每一个String对象都会独立占用内存一块区域。

### 3.4.14 java程序性能分析之thread dump和heap dump

在故障定位(尤其是out of memory)和性能分析的时候，经常会用到一些文件来帮助我们排除代码问题，heap dump记录内存信息的，thread dump是记录CPU信息的。

heap dump：

heap dump文件是一个二进制文件，它保存了某一时刻JVM堆中对象使用情况。HeapDump文件是指定时刻的Java堆栈的快照，是一种镜像文件。Heap Analyzer工具通过分析HeapDump文件，哪些对象占用了太多的堆栈空间，来发现导致内存泄露或者可能引起内存泄露的对象。

thread dump：

thread dump文件主要保存的是java应用中各线程在某一时刻的运行的位置，即执行到哪一个类的哪一个方法哪一个行上。thread dump是一个文本文件，打开后可以看到每一个线程的执行栈，以stacktrace的方式显示。通过对thread dump的分析可以得到应用是否“卡”在某一点上，即在某一点运行的时间太长，如数据库查询，长期得不到响应，最终导致系统崩溃。单个的thread dump文件一般来说是没有什么用处的，因为它只是记录了某一个绝对时间点的情况。比较有用的是，线程在一个时间段内的执行情况。

两个thread dump文件在分析时特别有效，困为它可以看出在先后两个时间点上，线程执行的位置，如果发现先后两组数据中同一线程都执行在同一位置，则说明此处可能有问题，因为程序运行是极快的，如果两次均在某一点上，说明这一点的耗时是很大的。通过对这两个文件进行分析，查出原因，进而解决问题。

### 3.4.15 CPU使用很高，频繁FullGC排查小总结

1. 首先我们通过top命令查看当前CPU消耗过高的进程是哪个，从而得到进程id；

然后通过top -Hp <pid>来查看该进程中有哪些线程CPU过高，一般超过80%就是比较高的，80%左右是合理情况。这样我们就能得到CPU消耗比较高的线程id。接着通过该线程id的十六进制表示在jstack日志中查看当前线程具体的堆栈信息。jstack <进程号>

这里又分为两种情况：

1. （1）如果是正常的用户线程，则通过该线程的堆栈信息查看其具体是在哪处用户代码处运行比较消耗 CPU。

（2）如果该线程是 VM Thread（VM Thread 指的就是垃圾回收的线程，一般前面会有 nid=0x.......，这里 nid 的意思就是操作系统线程 id），则通过 jstat -gcutil 命令监控当前系统的 GC状况。使用jmap生成堆内存信息，可以执行下面的命令生成Heap Dump：jmap -dump:live,format=b,file=heap-dump.bin <pid>导出系统当前的内存数据。导出系统当前的内存数据。jhat 是JDK自带的用于分析JVM Heap Dump文件的工具，使用下面的命令可以将堆文件的分析结果以HTML网页的形式进行展示：jhat <heap-dump-file>；执行成功之后：访问 http://localhost:7000/ 就可以看到结果了。

（导出之后将内存情况放到 Eclipse 的 Mat 工具中进行分析即可得出内存中主要是什么对象比较消耗内存，进而可以处理相关代码。）

1. 如果通过 top 命令看到 CPU 并不高，并且系统内存占用率也比较低。此时就可以考虑是否是由于另外三种情况导致的问题。

具体的可以根据具体情况分析：

1: 如果是接口调用比较耗时，并且是不定时出现，则可以通过压测的方式加大阻塞点出现的频率，从而通过 jstack 查看堆栈信息，找到阻塞点。

2: 如果是某个功能突然出现停滞的状况，这种情况也无法复现，此时可以通过多次导出 jstack 日志的方式对比哪些用户线程是一直都处于等待状态，这些线程就是可能存在问题的线程。

3: 如果通过 jstack 可以查看到死锁状态，则可以检查产生死锁的两个线程的具体阻塞点，从而处理相应的问题。

获取ThreadDump文件：

有很多方式可用于获取ThreadDump, 下面列出一部分获取方式：

操作系统命令获取ThreadDump:

Windows:

1.转向服务器的标准输出窗口并按下Control + Break组合键, 之后需要将线程堆栈复制到文件中；

UNIX/ Linux：

首先查找到服务器的进程号(process id), 然后获取线程堆栈.

1. ps –ef | grep java 查看包含“java”的所有进程

2. kill -3 <pid>

注意：一定要谨慎, 一步不慎就可能让服务器进程被杀死。kill -9 命令会杀死进程。

JVM 自带的工具获取线程堆栈:

JDK自带命令行工具获取PID，再获取ThreadDump:

1. jps (也是显示所用java进程的命令)或 ps –ef|grepjava (获取PID) 查看包含“java”的所有进程

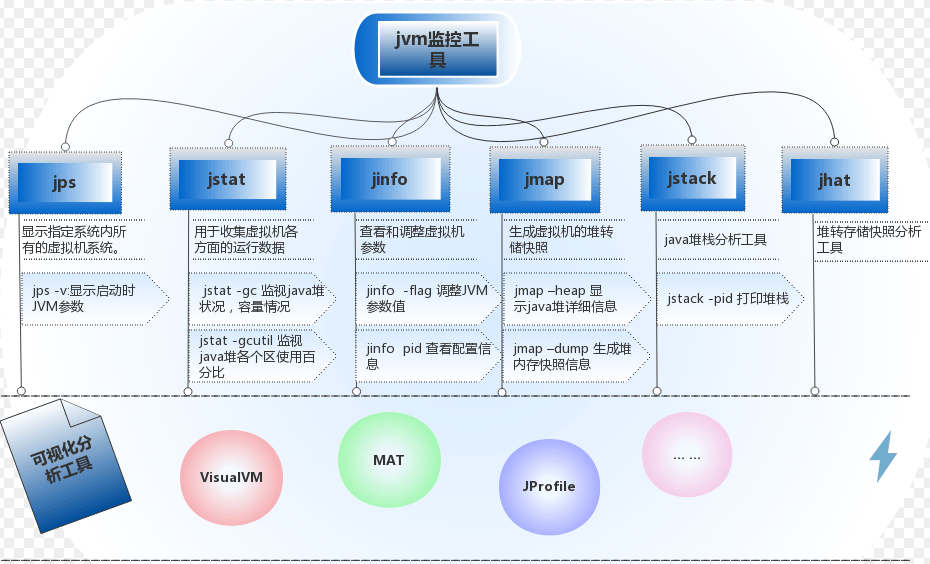
2. jstack [-l ]<pid> | tee -a jstack.log (获取ThreadDump)

得到dump可以用jhat去分析

jstat是用于监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。

例如：jstat -gc -h3 31736 1000 10

表示分析进程id为31736 的gc情况，每隔1000ms打印一次记录，打印10次停止，每3行后打印指标头部



<http://cmsblogs.com/?p=5155>

https://www.cnblogs.com/klvchen/p/11089632.html

### 3.4.16 jvm内存调优

对JVM内存的系统级的调优主要的目的是减少GC的频率和Full GC的次数，过多的GC和Full GC是会占用很多的系统资源（主要是CPU），影响系统的吞吐量。

调优手段主要是通过控制堆内存的各个部分的比例和GC策略来实现，下面来看看各部分比例不良设置会导致什么后果

1). 新生代设置过小

一是新生代GC次数非常频繁，增大系统消耗；二是导致大对象直接进入旧生代，占据了旧生代剩余空间，诱发Full GC

2). 新生代设置过大

一是新生代设置过大会导致旧生代过小（堆总量一定），从而诱发Full GC；二是新生代GC耗时大幅度增加。一般说来新生代占整个堆1/3比较合适

3). Survivor设置过小

导致对象从eden直接到达旧生代，降低了在新生代的存活时间

4). Survivor设置过大

导致eden过小，增加了GC频率

另外，通过-XX:MaxTenuringThreshold=n来控制新生代存活时间，尽量让对象在新生代被回收。

Jvm参数的设置：

-Xms:初始堆大小

-Xmx:最大堆大小

-XX:NewSize=n:设置年轻代大小

-XX:NewRatio=n:设置年轻代和年老代的比值。如:为3，表示年轻代与年老代比值为1：3，年轻代占整个年轻代年老代和的1/4

-XX:SurvivorRatio=n:年轻代中Eden区与两个Survivor区的比值。注意Survivor区有两个。如：3，表示Eden：Survivor=3：2，一个Survivor区占整个年轻代的1/5

-XX:MaxPermSize=n:设置持久代大小

# 4数据库与缓存

## 4.1 MySQL

### 4.1.1关系型数据库与非关系型数据库

|  |  |  |  |
| --- | --- | --- | --- |
| 数据库类型 | 特性 | 优点 | 缺点 |
| 关系型：  SQLite、  Oracle、  mysql | 1、关系型数据库，是指采用了关系模型来组织数据的数据库；简单来说，关系模型指的就是二维表格模型，而一个关系型数据库就是由二维表及其之间的联系所组成的一个数据组织。  2、最大特点就是事务的一致性； | 1、容易理解：二维表结构是非常贴近逻辑世界一个概念，关系模型比网状、层次等模型来说更容易理解；  2、使用方便：通用的SQL语言使得操作关系型数据库非常方便；  3、易于维护：实体完整性、用户定义的完整性减低了数据冗余和数据不一致的概率；  4、支持SQL，可用于复杂的查询。 | 1、读写性能比较差，尤其是海量数据的高效率读写；  2、固定的表结构，灵活性欠缺；  3、高并发读写需求，磁盘IO是一个瓶颈； |
| 非关系型：  MongoDB、  redis、  HBase | 1、使用键值对存储数据；  2、分布式；  3、一般不支持ACID特性；  4、非关系型数据库严格上不是一种数据库，应该是一种数据结构化存储方法的集合。 | 1、无需经过sql层解析，读写性能很高；  2、基于键值对，数据没有耦合性，容易扩展；  3、存储数据的格式：nosql的存储格式是key,value形式、文档形式、图片形式等，而关系型数据库则只支持基础类型。 | 1、不提供sql支持，学习和使用成本较高；  2、无事务处理；  3、数据结构相对复杂，复杂查询方面稍欠； |

### 4.1.2Mysql储存引擎Innodb引擎和MyISAM引擎

Innodb引擎和MyISAM引擎的区别：

1. MyISAM不支持事务，InnoDB是事务类型的存储引擎，当我们的表需要用到事务支持的时候，那肯定是不能选择MyISAM了。
2. MyISAM只支持表级锁，而InnoDB支持行级锁和表级锁默认为行级锁。
3. MyISAM引擎不支持外键，InnoDB支持外键。
4. MyISAM支持全文类型索引，而InnoDB不支持全文索引。
5. MyISAM引擎的表的查询、更新、插入的效率要比InnoDB高。

应用场景：

1、MyISAM管理非事务表，提供高速存储和检索以及全文搜索能力，如果再应用中执行大量select操作，应该选择MyISAM

2、InnoDB用于事务处理，具有ACID事务支持等特性，如果在应用中执行大量insert和update操作，应该选择InnoDB

### 4.1.3事务的特性（ACID）

数据库事务transanction正确执行的四个基本要素。ACID：原子性(Atomicity)、一致性(Correspondence)、隔离性(Isolation)、持久性(Durability)。

1. 原子性：整个事务中的所有操作，要么全部完成，要么全部不完成，不可能停滞在中间某个环节。事务在执行过程中发生错误，会被回滚（Rollback）到事务开始前的状态，就像这个事务从来没有执行过一样。
2. 一致性：在事务开始之前和事务结束以后，数据库的完整性约束没有被破坏。

一个事务在执行之前和执行之后，数据库数据必须保持一致性。数据库的一致性状态应该满足模式锁指定的约束条件，那么在完整执行该事务后，数据库仍然处于一致性状态。

例如：银行转账，转账前后两个账户金额之和应保持不变。

1. 隔离性：由并发事务所作的修改必须与任何其它并发事务所作的修改隔离。事务查看数据库时数据所处的状态，要么是另一并发事务修改它之前的状态，要么是另一事务修改它之后的状态，事务不会查看中间状态的数据。

例如：对任何一对事务T1和T2，对T1而言，T2要么在T1开始之前已经结束，要么在T1完成之后再开始执行。

1. 持久性：在事务完成以后，该事务所对数据库所作的更改便持久的保存在数据库之中，并不会被回滚。

### 4.1.4数据库事务的隔离级别？

丢失更新：一个事务的更新操作会被另一个事务的更新操作覆盖，从而导致数据的不一致。

* 脏读：一个事务可以读到另一个事务未提交的数据；
* 不可重复读：（**读取数据本身的对比**）一个事务对同一行数据重复读取两次，但是却得到了不同的结果。
* 幻读：（**读取结果集条数的对比**）一个事务按相同的查询条件查询之前检索过的数据，确发现检索出来的结果集条数变多或者减少，类似产生幻觉。这是因为在两次查询过程中有另外一个事务插入数据造成的。

1. 读未提交，Read Uncommited。

指的是一个事务读取到另外一个事务还没有提交的内容。如果读取到了别的事务未提交的数据，这种情况称为脏读。要想解决脏读的问题，可以提高数据库的事务隔离级别，把事务隔离级别设置为读已提交。

1. 读已提交，Read Committed。

这个隔离级别可以解决脏读的问题。

一个事务对同一行数据重复读取两次，但是却得到了不同的结果。值改变的原因是，穿插执行的事务2对该行数据进行了update操作。在同一个事务中，两次select出来的值不相同的问题称为不可重复读问题。要想解决不可重复读问题，需要把数据的隔离级别设置为可重复读。

1. 可重复读。Repeatable Read。

在这个隔离级别下，可以解决不可重复读的问题，但会出现幻读。

1. 串行化，Serialization。

串行化可以解决幻读的问题。它要求事务的执行完全串行执行。所以失去了并发的效率。

Mysql的默认隔离级别为可重复读。

### 4.1.5MySQL索引以及索引建立的原则？？

索引提供指向存储在表的指定列中的数据值的指针，然后根据您指定的排序顺序对这些指针排序。数据库使用索引以找到特定值，然后顺指针找到包含该值的行。这样可以使对应于表的SQL语句执行得更快，可快速访问数据库表中的特定信息。索引就像是书的目录一样。

索引的优点：可以大大加快 数据的检索速度，这也是创建索引的最主要的原因。

索引建立的原则：

1. 为经常需要排序、分组和联合操作的字段建立索引：经常需要ORDER BY、GROUP BY、DISTINCT和UNION等操作的字段，排序操作会浪费很多时间。如果为其建立索引，可以有效地避免排序操作。
2. 为常作为查询条件的字段建立索引：如果某个字段经常用来做查询条件，那么该字段的查询速度会影响整个表的查询速度。因此，为这样的字段建立索引，可以提高整个表的查询速度。
3. 尽量使用前缀来索引：如果索引字段的值很长，最好使用值的前缀来索引。例如，TEXT和BLOG类型的字段，进行全文检索会很浪费时间。如果只检索字段的前面的若干个字符，这样可以提高检索速度。
4. 最左匹配原则，非常重要的原则：mysql会一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配，比如a = 1 and b= 2 c> 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，如果建立(a,b,d,c)的索引则都可以用到，a,b,d的顺序可以任意调整。

最左匹配原则：

最左优先，以最左边的为起点任何连续的索引都能匹配上。同时遇到范围查询(>、<、between、like)就会停止匹配。

例如：b = 2 如果建立(a,b)顺序的索引，是匹配不到(a,b)索引的；但是如果查询条件是a = 1 and b = 2或者a=1(又或者是b = 2 and b = 1)就可以，因为优化器会自动调整a,b的顺序。再比如a = 1 and b = 2 and c > 3 and d = 4 如果建立(a,b,c,d)顺序的索引，d是用不到索引的，因为c字段是一个范围查询，它之后的字段会停止匹配。

索引的缺点：

第一，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

第三，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

### 4.1.6数据库有几种索引

1. **主键索引**

主键索引是唯一索引的特殊类型。

数据库表通常有一列或列组合，其值用来唯一标识表中的每一行。该列称为表的主键。

1. **唯一索引**

唯一索引不允许两行具有相同的索引值。

如果现有数据中存在重复的键值，则大多数数据库都不允许将新创建的唯一索引与表一起保存。当新数据将使表中的键值重复时，数据库也拒绝接受此数据。例如，如果在 employee 表中的职员姓氏(lname) 列上创建了唯一索引，则所有职员不能同姓。

主键索引和唯一索引的不同是：

1. 主健不可为空，唯一索引可；
2. 每个表只能有一个。
3. **普通索引**

这是最基本的索引类型，而且它没有唯一性之类的限制。

创建索引，例如CREATE INDEX <索引的名字> ON tablename (列的列表);

1. **全文索引**

但是当在MySQL中存储了文本,比如某个字段的值为坚决贯彻党的十八大精神,我们想用贯彻和十八大作为关键字时都可以搜索到这条记录.那么只能使用like关键字.而对于like我们知道,当不是用左边前缀搜索的时候,无法命中索引,因此对于这条语句select \* from articles where content like '%贯彻%',MySQL只能进行全表扫描,逐一进行匹配.这样的效率极其低下.

而全文索引呢,通过建立倒排索引,可以极大的提升检索效率.

倒排索引（英语：Inverted index），也常被称为反向索引、置入档案或反向档案，是一种索引方法，被用来存储在全文搜索下某个单词在一个文档或者一组文档中的存储位置的映射。它是文档检索系统中最常用的数据结构

1. **联合索引**

利用索引中的附加列，您可以缩小搜索的范围，但使用一个具有两列的索引 不同于使用两个单独的索引。复合索引的结构与电话簿类似，人名由姓和名构成，电话簿首先按姓氏对进行排序，然后按名字对有相同姓氏的人进行排序。如果您知道姓，电话簿将非常有用；如果您知道姓和名，电话簿则更为有用，但如果您只知道名不姓，电话簿将没有用处。

多个单列索引在多条件查询时只会生效第一个索引！所以多条件联合查询时最好建联合索引！

联合索引本质：

当创建(a,b,c)联合索引时，相当于创建了(a)单列索引，(a,b)联合索引以及(a,b,c)联合索引

想要索引生效的话,只能使用 a和a,b和a,b,c三种组合；当然，我们上面测试过，a,c组合也可以，但实际上只用到了a的索引，c并没有用到！

### 4.1.7SQL语句的执行顺序？

|  |
| --- |
| (8)SELECT (9)DISTINCT<select\_list>  (1)FROM<left\_table>  (3)<join\_type>JOIN<right\_table>  (2) ON<join\_condition>  (4)WHERE<where\_condition>  (5)GROUP BY<group\_by\_list>  (6)WITH{CUBE|ROLLUP}  (7)HAVING<having\_condition>  (10)ORDER BY<order\_by\_list>  (11)LIMIT<limit\_number> |

以上的sql语句前的数字代表该sql语句的执行顺序，每一个操作都会产生一个虚拟表，这个虚拟表作为下个操作的输入，依次类推，最后生成的虚拟表作为结果输出。如果没有在查询中指定某一字句，则将跳过相应的步骤:

1. FROM:对FROM子句中的左表<left\_table>和右表<right\_table>执行笛卡尔积，产生虚拟表VT1.
2. ON：对虚拟表VT1应用ON筛选，只有那些符合条件<join\_condition>的行才会被插入虚拟表VT2中。
3. JOIN:如果是外连接，还要根据外连接的类型执行join操作，例如左外连接将左表中的行并且不在VT2中的行加入到VT2中，产生VT3.
4. WHERE：对虚拟表VT3应用where过滤条件，只有符合<where\_condition>的记录才被插入虚拟表VT4中。
5. GROUP BY:根据GROUP BY字句中的列，对VT4中的记录进行分组操作，产生VT5.
6. CUBE|ROLLUP:对表VT5执行cube或者rollup操作，产生虚拟表VT6.
7. HAVING:对虚拟表VT6应用HAVING过滤器，只有符合<having\_condition>的记录才会被插入虚拟表VT7中
8. SELECT:对VT7执行select操作，将指定的数据插入到虚拟表VT8中.
9. DISTINCT:去除重复数据，产生虚拟表VT9.
10. ORDER BY:按照<order\_by\_list>对VT9进行排序操作，产生虚拟表VT10
11. LIMIT:选出指定的行，产生虚拟表VT11,返回给用户。

### 4.1.8DML、DDL、DCL

* DML（Data Manipulation Language）：数据操纵语言，对数据库中的数据进行操作
* SELECT、UPDATE、INSERT、DELETE。
* DDL（Data Definition Language）：数据库定义语言，用于定义和管理SQL数据库中所有对象的语言
* CREATE、ALTER、DROP、TRUNCATE。
* DML（Data Control Language）：数据控制语言，用来授予或回收访问数据库的某种特权，并控制数据库事务
* COMMIT（提交）、ROLLBACK（回滚）、SAVEPOINT（保存点）、SET TRANSACTION（设置当前事务的属性）。

### 4.1.9为什么mysql innodb索引是B+树数据结构而不是B树？

1.文件很大，不可能全部存储在内存中，故要存储到磁盘上

2.索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数（为什么使用B-/+Tree，还跟磁盘存取原理有关。）

3、B+树所有的Data域在叶子节点，一般来说都会进行一个优化，就是将所有的叶子节点用指针串起来，这样遍历叶子节点就能获得全部数据。

那为什么是B+树而不是B树呢，因为它内节点不存储data，这样一个节点就可以存储更多的key。这样就可以减少磁盘IO的次数。

一般来说，索引很大，往往以索引文件的形式存储的磁盘上，索引查找时产生磁盘I/O消耗，相对于内存存取，I/O存取的消耗要高几个数量级，所以评价一个数据结构作为索引的优劣最重要的指标就是在查找过程中磁盘I/O操作次数的时间复杂度。树高度越小，I/O次数越少。

Mysql索引的一些知识https://www.cnblogs.com/williamjie/p/11187470.html

### 4.1.10平衡二叉树和红黑树的区别

红黑树的性质：

1.节点是红色或黑色。

2.根节点是黑色。

3.每个叶子节点都是黑色的空节点（NIL节点）。

4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)

5.从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

这些约束强制了红黑树的关键性质: 从根到叶子的最长的可能路径不多于最短的可能路径的两倍长。结果是这个树大致上是平衡的。因为操作比如插入、删除和查找某个值的最坏情况时间都要求与树的高度成比例，这个在高度上的理论上限允许红黑树在最坏情况下都是高效的，而不同于普通的二叉查找树。

平衡二叉树的性质：

它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。这个方案很好的解决了二叉查找树退化成链表的问题，把插入，查找，删除的时间复杂度最好情况和最坏情况都维持在O(logN)。但是频繁旋转会使插入和删除牺牲掉O(logN)左右的时间，不过相对二叉查找树来说，时间上稳定了很多。

两者区别：

1、红黑树放弃了追求完全平衡，追求大致平衡，在与平衡二叉树的时间复杂度相差不大的情况下，保证每次插入最多只需要三次旋转就能达到平衡，实现起来也更为简单。

2、平衡二叉树追求绝对平衡，条件比较苛刻，实现起来比较麻烦，每次插入新节点之后需要旋转的次数不能预知。

### 4.1.11 聚簇索引与非聚簇索引？

聚簇索引：将数据存储与索引放到了一块，索引结构的叶子节点保存了行数据。聚集索引就是按照每张表的主键构造一颗B+树，并且叶节点中存放着整张表的行记录数据，因此也让聚集索引的叶节点成为数据页。由于实际的数据页只能按照一颗B+树进行排序，因此每张表只能拥有一个聚集索引。

非聚簇索引：将数据与索引分开存储，索引结构的叶子节点指向了数据对应的位置。

数据库表的索引从数据存储方式上可以分为聚簇索引和非聚簇索引（又叫二级索引）两种。Innodb的聚簇索引在同一个B-Tree中保存了索引列和具体的数据，在聚簇索引中，实际的数据保存在叶子页中，中间的节点页保存指向下一层页面的指针。“聚簇”的意思是数据行被按照一定顺序一个个紧密地排列在一起存储。一个表只能有一个聚簇索引，因为在一个表中数据的存放方式只有一种。

非聚簇索引，又叫二级索引。二级索引的叶子节点中保存的不是指向行的物理指针，而是行的主键值。当通过二级索引查找行，存储引擎需要在二级索引中找到相应的叶子节点，获得行的主键值，然后使用主键去聚簇索引中查找数据行，这需要两次B-Tree查找。

两种索引的不同：

1. 聚簇索引将索引和数据行保存在同一个B-Tree中，查询通过聚簇索引可以直接获取数据，相比非聚簇索引需要第二次查询（非覆盖索引的情况下）效率要高。
2. 聚簇索引对于范围查询的效率很高，因为其数据是按照大小排列的。
3. 聚簇索引的更新代价比较高，如果更新了行的聚簇索引列，就需要将数据移动到相应的位置。

https://www.cnblogs.com/s-b-b/p/8334593.html

<https://www.cnblogs.com/wade-luffy/p/6292991.html>

https://www.jianshu.com/p/6991eefbde13

### 4.1.12 B+树索引和哈希索引的区别

b树（balance tree）和b+树应用在数据库索引，可以认为是m叉的多路平衡查找树，但是从理论上讲，二叉树查找速度和比较次数都是最小的，为什么不用二叉树呢？

因为我们要考虑磁盘IO的影响，它相对于内存来说是很慢的。数据库索引是存储在磁盘上的，当数据量大时，就不能把整个索引全部加载到内存了，只能逐一加载每一个磁盘页（对应索引树的节点）。所以我们要减少IO次数，对于树来说，IO次数就是树的高度，而“矮胖”就是b树的特征之一，它的每个节点最多包含m个孩子，m称为b树的阶，m的大小取决于磁盘页的大小。

**B+树的特点**

B+树的非叶子节点只是存储key，占用空间非常小，因此每一层的节点能索引到的数据范围更加的广。换句话说，每次IO操作可以搜索更多的数据。

支持范围查询，而且部分范围查询非常高效，每个节点能索引的范围更大更精确,也意味着 B+树单次磁盘IO的信息量大于B-树,I/O效率更高。

原因是数据都是存储在叶子节点这一层，并且有指针指向其他叶子节点，这样范围查询只需要遍历叶子节点这一层，无需整棵树遍历。

**区别**

1. 如果是等值查询，那么哈希索引明显有绝对优势，因为只需要经过一次算法即可找到相应的键值；当然了，这个前提是，键值都是唯一的。如果键值不是唯一的，就需要先找到该键所在位置，然后再根据链表往后扫描，直到找到相应的数据。

如果是范围查询检索，这时候哈希索引就毫无用武之地了，因为原先是有序的键值，经过哈希算法后，有可能变成不连续的了，就没办法再利用索引完成范围查询检索。

同理，哈希索引也没办法利用索引完成排序，以及like ‘xxx%’ 这样的部分模糊查询（这种部分模糊查询，其实本质上也是范围查询）。

1. 哈希索引也不支持多列联合索引的最左匹配规则。
2. B+树索引的关键字检索效率比较平均，在有大量重复键值情况下，哈希索引的效率也是极低的，因为存在所谓的哈希碰撞问题。

### 4.1.13数据库设计三范式？

第一范式：属性不可分割；

所谓第一范式（1NF）是指数据库表的每一列都是不可分割的基本数据项，同一列中不能有多个值，即实体中的某个属性不能有多个值或者不能有重复的属性。如果出现重复的属性，就可能需要定义一个新的实体，新的实体由重复的属性构成，新实体与原实体之间为一对多关系。在第一范式（1NF）中表的每一行只包含一个实例的信息。在任何一个关系数据库中，第一范式（1NF）是对关系模式的基本要求，不满足第一范式（1NF）的数据库就不是关系数据库。

第二范式：要有主键，要求其他字段都依赖于主键；

第二范式是在第一范式的基础上建立起来的，即满足第二范式必须先满足第一范式。第二范式要求数据库表中的每个实例或行必须可以被惟一的区分。为实现区分通常需要为表加上一个列，以存储各个实例的惟一标识。要求实体的属性完全依赖于主关键字。

第三范式：消除传递依赖；

满足第三范式（3NF）必须先满足第二范式（2NF）。简而言之，第三范式（3NF）要求一个数据库表中不包含已在其它表中已包含的非主关键字信息。在第二范式的基础上，数据表中如果不存在非关键字段对任一候选关键字段的传递函数依赖则符合第三范式。

不能存在传递依赖。即：除主键外，其他字段必须依赖主键。

### 4.1.14如何优化数据库？

1. 优化sql语句；

原则： 1.1 尽量根据主键查询；

1.2 尽量使用单表查询，不要使用关联查询；

1.3 查询时可以使用in，但是绝对不要使用not in；

1. 创建索引；CREATE INDEX 索引名字 ON 表名称 （列名称）；[列名称：规定你需要索引的列]。
2. 添加缓存；

例如：mybatis的一、二级缓存；该操作效率低；

redis缓存 / memercache缓存（String）；有效的缓解数据库的压力；

1. 使用数据库的读写分离；
2. 定期将历史数据进行转储；
3. 进行分库分表操作（最后的操作）；数据库服务器数量和运维都需要花费很多时间和精力。

### sql语句优化

* 在表中建立索引，优先考虑where、group by使用到的字段。
* 尽量避免使用select \*，返回无用的字段会降低查询效率。如下：SELECT \* FROM t

优化方式：使用具体的字段代替\*，只返回使用到的字段。

* 尽量避免使用in 和not in，会导致数据库引擎放弃索引进行全表扫描。如下：

SELECT \* FROM t WHERE id IN (2,3)

SELECT \* FROM t1 WHERE username IN (SELECT username FROM t2)

优化方式：如果是连续数值，可以用between代替。如下：

SELECT \* FROM t WHERE id BETWEEN 2 AND 3

如果是子查询，可以用exists代替。如下：

SELECT \* FROM t1 WHERE EXISTS (SELECT \* FROM t2 WHERE t1.username = t2.username)

* 尽量避免使用or，会导致数据库引擎放弃索引进行全表扫描。如下：

SELECT \* FROM t WHERE id = 1 OR id = 3

优化方式：可以用union代替or。如下：

SELECT \* FROM t WHERE id = 1

UNION

SELECT \* FROM t WHERE id = 3

（PS：如果or两边的字段是同一个，如例子中这样。貌似两种方式效率差不多，即使union扫描的是索引，or扫描的是全表）

* 尽量避免在字段开头模糊查询，会导致数据库引擎放弃索引进行全表扫描。如下

SELECT \* FROM t WHERE username LIKE '%li%

优化方式：尽量在字段后面使用模糊查询。如下：

SELECT \* FROM t WHERE username LIKE 'li%'

<https://www.cnblogs.com/wangzhengyu/p/10412499.html>

### 4.1.16数据库读写分离（mysql主从复制）

为什么要做主从复制：

1. 在业务复杂的系统中，有这么一个情景，有一句sql语句需要锁表，导致暂时不能使用读的服务，那么就很影响运行中的业务，使用主从复制，让主库负责写，从库负责读，这样，即使主库出现了锁表的情景，通过读从库也可以保证业务的正常运行。
2. 做数据的热备，主库宕机后能够及时替换主库，保证业务可用性。
3. 架构的扩展。业务量越来越大，I/O访问频率过高，单机无法满足，此时做多库的存储，降低磁盘I/O访问的频率，提高单个机器的I/O性能。

这种集群方式的本质就是**把访问的压力从主库转移到从库**，也就是在单机数据库无法支撑并发读写的时候，并且读的请求很多的情况下适合这种读写分离的数

据库集群。如果**写的操作很多的话不适合这种集群方式**，因为你的数据库压力还是在写操作上，即使主从了之后压力还是在主库上和单机区别就不大了。

MySQL 主从复制原理的是啥？

MySQL的复制（replication）是一个异步的复制，从一个Master复制到Slave。整个复制操作主要由三个进程完成的，其中两个进程在Slave（Sql进程和IO进程），另外一个进程在Master（IO进程）上。要实施复制，首先必须打开Master端的binarylog（bin-log）功能，否则无法实现。

过程：主库将变更写入binlog日志，然后从库连接到主库之后，从库有一个IO线程，将主库的binlog日志拷贝到自己本地，写入一个relay中继日志中。接着从库中有一个SQL线程（会从中继日志读取binlog，然后执行binlog日志中的内容，也就是在自己本地再次执行一遍 SQL，这样就可以保证自己跟主库的数据是一样的。

主从同步延迟和注会数据丢失：

* 问题1：由于从库从主库拷贝日志以及串行执行SQL的特点，在高并发场景下，从库的数据一定会比主库慢一些，是有延时的。所以经常出现，刚写入主库的数据可能是读不到的，要过几十毫秒，甚至几百毫秒才能读取到。
* 问题2：如果主库突然宕机，然后恰好数据还没同步到从库，那么有些数据可能在从库上是没有的，有些数据可能就丢失了。

MySQL有两个机制解决上面的问题：一个是半同步复制，用来解决主库数据丢失问题；一个是并行复制，用来解决主从同步延时问题。

* + 半同步复制，也叫semi-sync复制，指的就是主库写入binlog日志之后，就会强制立即将数据同步到从库，从库将日志写入自己本地的relay log之后，接着会返回一个ack给主库，主库接收到至少一个从库的ack之后才会认为写操作完成了。
  + 并行复制，指的是从库开启多个线程，并行读取relay log中不同库的日志，然后并行重放不同库的日志，这是库级别的并行。

### 数据库的分库分表

<https://www.jianshu.com/p/05da0fc0950e>

https://blog.csdn.net/yingshuanglu2564/article/details/99564585

### 4.1.17sql语句调优过程？

（1）第一步：开启慢查询日志，捕获慢SQL

* 查看慢查询日志是否开启，如果没有开启，就开启慢查询日志；
* 查看慢查询日志的阈值（这个值一般表示超过多少时间的SQL语句会被记录到慢查询日志中），当然也可以根据业务需求自己设置慢查询日志的阈值；
* 查看有多少SQL语句的查询时间超过了阈值；
* 进入MQSQL安装目录，使用mysqldumpslow命令查看使用次数最多的SQL语句、查询时间排序的sql；

（2）第二步：explain+慢SQL分析

* 使用EXPLAIN关键字可以模拟优化器执行SQL查询语句，从而知道MySQL是如何处理SQL语句的。可以据此分析查询语句或是表结构的性能瓶颈。
* 使用方式：Explain+SQL语句，执行这个命令后，显示如下：

|  |
| --- |
| +---+----+-----+------+------+------+-----+------+-----+-----+  | id | select\_type | table | type | possible\_keys | key | key\_len | ref | rows | Extra |  +---+----+-----+------+------+------+-----+------+-----+-----+ |

* type：显示访问类型，结果值从好到坏依次是：

system>const>eq\_ref>ref……… >unique\_subquery>index\_subquery>range>index>ALL

一般来说，得保证查询至少达到range级别，最好能达到ref。

* ref：显示索引的哪一列被使用了，哪些列或常量被用于查找索引列上的值。

（3）第三步：show profile查询SQL语句在服务器中的执行细节和生命周期

* Show Profile是MySQL提供可以用来分析当前会话中语句执行的资源消耗情况，可以用于SQL的调优测量；默认关闭，并保存最近15次的运行结果；比如遇到下面的情况要进行优化：创建临时表；把内存中的临时表复制到磁盘中；

（4）第四步：SQL数据库服务器参数调优

* 当order by 和 group by无法使用索引时，增大max\_length\_for\_sort\_data参数设置和增大sort\_buffer\_size参数的设置；

### 4.1.18防止SQL注入的方式？

所谓SQL注入，就是通过把SQL命令插入到Web表单提交或输入域名或页面请求的查询字符串，最终达到欺骗服务器执行恶意的SQL命令。它不是利用操作系统的BUG来实现攻击，而是针对程序员编程时的疏忽，通过SQL语句，实现无帐号登录，甚至篡改数据库。

注入攻击实例：https://www.cnblogs.com/pressur/p/11226392.html

（1）PreparedStatement

* 采用预编译语句集，它内置了处理SQL注入的能力，只要使用它的setXXX方法传值即可。sql注入只对sql语句的编译过程有破坏作用，而PreparedStatement已经准备好了，执行阶段只是把输入串作为数据处理，而不再对sql语句进行解析,准备,因此也就避免了sql注入问题.

（2）使用正则表达式过滤传入的参数

* 要引入的包：import Java.util.regex.\*；然后判断输入的sql语句是不是与正则表达式匹配；

（3）字符串过滤

* 这是一个比较通用的方法，将输入的sql字符串使用 | 隔开，最后进行分割；

（4）JSP页面判断代码

* 使用JavaScript在客户端进行不安全字符屏蔽；

（5）使用一些持久层的框架，比如mybatis等；

### 4.1.20MySQL如何优化limit？

（1）分页查询的语句：\*\*\*\* limit [offset] rows;

一般是用于select语句中用以从结果集中拿出特定的一部分数据。offset是偏移量，表示我们现在需要的数据是跳过多少行数据之后的，可以忽略；rows表示我们现在要拿多少行数据。例子：

|  |
| --- |
| 1. select \* from mytbl limit 10000,100 表示从表mytbl中拿数据，跳过10000行之后，拿100行 2. select \* from mytbl limit 0,100 表示从表mytbl拿数据，跳过0行之后，拿取100行 3. select \* from mytbl limit 100 这条SQL跟②的效果是完全一样的，表示拿前100条数据 |

（2）用处：

目前用到的地方是数据库查询分页，比如前台要展示数据库中数据，需要后台实现分页，传入数据要有“页码page”跟“每页数据条数nums”。对应SQL大概是：select \* from mytbl order by id limit (page-1)\*nums,nums

（3）问题

在数据量不大或者是大数据量的前几页的时候，性能还算不坏，但是大数据量页码稍微大一点性能便下降比较严重。原因出在Limit的偏移量offset上，比如limit 100000,10虽然最后只返回10条数据，但是偏移量却高达100000，数据库的操作其实是拿到100010数据，然后返回最后10条。

（4）解决方式

* 使用记录主键的方式进行优化

limit最大的问题在于要扫描前面不必要的数据，所以可以先对主键的条件做设定，然后记录住主键的位置再取行。

select \* from p2p\_20131230 where main\_id > 1000000 order by main\_id limit 10;

* 使用子查询进行查询：

select \* from p2p\_20131230 where main\_id >= (select main\_id from p2p\_20131230 limit 1000000,1) limit 10;

https://blog.csdn.net/mr253727942/article/details/46352085

### 4.1.20Mysql批量插入数据

在MySQL数据库中插入大量的数据，怎么才能快速插入到MySQL里呢？

数据量非常大的时候，不能使用INSERT INTO一条一条地插入，MySQL提高数据插入效率的基本原则如下：

* 批量插入数据的效率比单数据行插入的效率高
* 插入无索引的数据表比插入有索引的数据表快一些
* 较短的SQL语句的数据插入比较长的语句快

使用load data 批量插入记录到数据库表中41

几个结论：

* 使用LOAD DATA语句要比INSERT语句效率高，因为它批量插入数据行。服务器只需要对一个语句（而不是多个语句）进行语法分析和解释。索引只有在所有数据行处理完之后才需要刷新，而不是每处理一行都刷新。
* 如果你只能使用INSERT语句，那就要使用将多个数据行在一个语句中给出的格式：INSERT INTO table\_name VALUES(...),(...),...，这将会减少你需要的语句总数，最大程度地减少了索引刷新的次数。
* 在默认情况下，LOAD DATA语句将假设各数据列的值以制表符（t）分隔，各数据行以换行符（n）分隔，数据值的排列顺序与各数据列在数据表里的先后顺序一致。

### 4.1.21数据库索引什么时候失效？

1. 对于创建的多列索引（复合索引），不是使用的第一部分就不会使用索引

alter table student add index my\_index(name, age) // name左边的列， age 右边的列

select \* from student where name = 'aaa' // 会用到索引

select \* from student where age = 18 // 不会使用索引

1. 对于使用 like 查询， 查询如果是 ‘%aaa’ 不会使用索引，而 ‘aaa%’ 会使用到索引。

select \* from student where name like 'aaa%' // 会用到索引

select \* from student where name like '%aaa' 或者 '\_aaa' // 不会使用索引

1. 如果条件中有 or， 有条件没有使用索引，即使其中有条件带索引也不会使用，换言之， 就是要求使用的所有字段，都必须单独使用时能使用索引。
2. 如果列类型是字符串，那么一定要在条件中使用引号引用起来，否则不使用索引。
3. 如果mysql认为全表扫面要比使用索引快，则不使用索引。

如：表里只有一条数据。

### 4.1.22Mysql的MVCC

多版本并发控制（MVCC）是一种用来解决读-写冲突的无锁并发控制，也就是为事务分配单向增长的时间戳，为每个修改保存一个版本，版本与事务时间戳关联，读操作只读该事务开始前的数据库的快照。 所以MVCC可以为数据库解决以下问题

在并发读写数据库时，可以做到在读操作时不用阻塞写操作，写操作也不用阻塞读操作，提高了数据库并发读写的性能

同时还可以解决脏读，幻读，不可重复读等事务隔离问题，但不能解决更新丢失问题。

### 4.1.23 OCC,PCC,MVCC三者的关系

悲观并发控制（PCC）是一种用来解决读-写冲突和写-写冲突的的加锁并发控制, 为每个操作都加锁，同一时间下，只有获得该锁的事务才能有权利对该数据进行操作，没有获得锁的事务只能等待其他事务释放锁；所以可以解决脏读，幻读，不可重复读，第一类更新丢失，第二类更新丢失的问题

乐观并发控制（OCC）是一种用来解决写-写冲突的无锁并发控制，认为事务间争用没有那么多，所以先进行修改，在提交事务前，检查一下事务开始后，有没有新提交改变，如果没有就提交，如果有就放弃并重试。乐观并发控制类似自旋锁。乐观并发控制适用于低数据争用，写冲突比较少的环境；无法解决脏读，幻读，不可重复读，但是可以解决更新丢失问题

多版本并发控制（MVCC）是一种用来解决读-写冲突的无锁并发控制，也就是为事务分配单向增长的时间戳，为每个修改保存一个版本，版本与事务时间戳关联，读操作只读该事务开始前的数据库的快照。 这样在读操作时就不用阻塞写操作，写操作也不用阻塞读操作；不仅可以提高并发性能，还可以解决脏读，幻读，不可重复读等事务问题。更新丢失问题除外。

https://blog.csdn.net/SnailMann/article/details/88388829

## 4.2Redis

### 4.2.1Redis五种数据类型

（1）String

* 格式：set key value
* 说明：二进制安全的，String类型可包含任何数据；String在redis内部存储默认是一个动态字符串，被redisObject所引用，当遇到incr，decr等操作时会转成数值型进行计算，此时redisObject的encoding字段为int。

（2）Hash

* 格式：hmset name key1 value1 key2 value2
* hash 是一个键值对集合，string类型的field和value的映射表，特别适合用于存储对象。
* Hash实际是内部存储的Value为一个HashMap，并提供了直接存取这个Map成员的接口，当我们需要更改数据时，只需要通过key获取value，通过value可以得到具体的字段和属性值。

1. List：list列表是简单的字符串列表，按照插入顺序排序(内部实现为LinkedList)，可以选择将一个链表插入到头部或尾部。

常用命令 :lpush（添加左边元素）,rpush,lpop（移除左边第一个元素）,rpop,lrange（获取列表片段，LRANGE key start stop）等。

（4）Set：内部的实现是一个value永远为null的HashMap，实际上是通过计算hash的方式快速去重；

* 格式：sadd name value
* 说明：Redis的Set是string类型的无序集合。集合是通过哈希表实现的，所以添加，删除，查找的复杂度都是O(1)。利用Redis提供的Sets数据结构，可以存储一些集合性的数据，比如在微博应用中，可以将一个用户所有的关注人存在一个集合中，将其所有粉丝存在一个集合。Redis还为集合提供了求交集、并集、差集等操作，可以非常方便的实现如共同关注、共同喜好等功能。

（5）Sorted Set有序集合：sorted set的内部使用HashMap和跳跃表（SkipList）来保证数据的存储和有序；

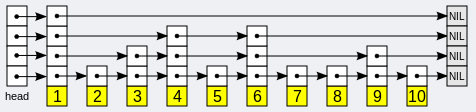
* 格式：zadd name score value
* zset 和 set 一样也是string类型元素的集合，且不允许重复的成员。不同的是每个元素都会关联一个double类型的分数。redis正是通过分数来为集合中的成员进行从小到大的排序。zset的成员是唯一的，但分数(score)却可以重复。

### 4.2.2Redis中的跳跃表

跳跃表是有序的数据结构，它通过在每个节点中维持多个指向其他节点的指针，从而达到快速访问节点的目的。其效率可以比拟平衡二叉树，查找、删除、插入等操作都可以在对数期望时间内完成。跳跃表主要有以下几个部分构成：

* + 1、表头head：负责维护跳跃表的节点指针；
  + 2、节点node：实际保存元素值，每个节点有一层或多层；
  + 3、层level：保存着指向该层下一个节点的指针；
  + 4、表尾tail：全部由null组成；

跳跃表的遍历总是从高层开始，然后随着元素值范围的缩小，慢慢降低到低层。



对于具有 n 个元素的链表，我们可以采取 \*\* (logn + 1) 层指针路径的形式**，**就可以实现在 O(logn) 的时间复杂度内，查找到某个目标元素了，这种数据结构，我们也称之为跳跃表**，**跳跃表也可以算是链表的一种变形，只是它具有二分查找的功能。9个结点，一共4层。

Redis中跳跃表的数据结构对原来的跳跃表进行了一下修改：

* + 1、允许重复的score值：多个不同的元素(member)的score值可以相同；
  + 2、进行元素对比的时候，不仅要检查score值，还需要检查member：当score值相等时，需要比较member域进行比较；
  + 3、结构保存一个tail指针：跳跃表的表尾指针；
  + 4、每个节点都有一个高度为1层的前驱指针，用于从底层表尾向表头方向遍历；

### 4.2.3Redis为什么这么快？

* 1：Redis是纯内存数据库，一般都是简单的存取操作，线程占用的时间很多，时间的花费主要集中在IO上，所以读取速度快。
* 2：Redis使用的是非阻塞IO，IO多路复用，使用了单线程来轮询描述符，将数据库的开、关、读、写都转换成了事件，减少了线程切换时上下文的切换和竞争。
* 3：Redis采用了单线程的模型，保证了每个操作的原子性，也减少了线程的上下文切换和竞争。
* 4：数据结构也帮了不少忙，Redis全程使用hash结构，读取速度快，还有一些特殊的数据结构，对数据存储进行了优化，如压缩表，对短数据进行压缩存储，再如，跳表，使用有序的数据结构加快读取的速度。

### 4.2.4Redis缓存穿透？缓存雪崩？缓存击穿？如何避免？

1. 缓存穿透是指缓存和数据库中都没有的数据，而用户不断发起请求，如发起为id为“-1”的数据或id为特别大不存在的数据。这时的用户很可能是攻击者，攻击会导致数据库压力过大。

解决方案：

1. 接口层增加校验，如用户鉴权校验，id做基础校验，id<=0的直接拦截；
2. 从缓存取不到的数据，在数据库中也没有取到，这时也可以将key-value对写为key-null，缓存有效时间可以设置短点，如30秒（设置太长会导致正常情况也没法使用）。这样可以防止攻击用户反复用同一个id暴力攻击。
3. 缓存击穿

缓存击穿是指缓存中没有但数据库中有的数据（一般是缓存时间到期），这时由于并发用户特别多，同时读缓存没读到数据，又同时去数据库去取数据，引起数据库压力瞬间增大，造成过大压力。

解决方案：

1. 设置热点数据永远不过期
2. 加互斥锁Syn
3. 缓存雪崩是指缓存中数据大批量到过期时间，而查询数据量巨大，引起数据库压力过大甚至down机。和缓存击穿不同的是，缓存击穿指并发查同一条数据，缓存雪崩是不同数据都过期了，很多数据都查不到从而查数据库。

解决方案：

1. 缓存数据的过期时间设置随机，防止同一时间大量数据过期现象发生。
2. 如果缓存数据库是分布式部署，将热点数据均匀分布在不同的缓存数据库中。
3. 设置热点数据永远不过期。

### 4.2.5Redis持久化方式？

1. RDB：快照形式，定期把内存中当前时刻的数据保存到磁盘。Redis默认支持的持久化方案。速度快但是服务器断电的时候会丢失部分数据。
2. AOF形式：append only file。把所有对redis数据库操作的命令，增删改操作的命令，保存到文件中。数据库恢复时把所有的命令执行一遍即可。两种持久化方案同时开启使用AOF文件来恢复数据库.能保证数据的完整性,但是速度慢。
3. 两者如何选择?

* 不要仅仅使用 RDB，因为那样会导致你丢失很多数据；
* 也不要仅仅使用 AOF，因为那样有两个问题：第一，你通过 AOF 做冷备，没有 RDB 做冷备来的恢复速度更快；第二，RDB 每次简单粗暴生成数据快照，更加健壮，可以避免 AOF 这种复杂的备份和恢复机制的 bug；
* redis 支持同时开启开启两种持久化方式，我们可以综合使用 AOF 和 RDB 两种持久化机制，用 AOF 来保证数据不丢失，作为数据恢复的第一选择; 用 RDB 来做不同程度的冷备，在 AOF 文件都丢失或损坏不可用的时候，还可以使用 RDB 来进行快速的数据恢复。

### 4.2.6Redis过期策略与内存淘汰机制

（1）Redis的过期策略：定期删除+惰性删除

定期删除指的是redis默认每隔100ms就**随机抽取**一些设置了过期时间的key，检查是否过期，如果过期就删除。假设 redis 里放了 10w 个 key，都设置了过期时间，你每隔几百毫秒，就检查 10w 个 key，那 redis 基本上就死了，cpu 负载会很高的，消耗在你的检查过期 key 上了。注意，这里可不是每隔 100ms 就遍历所有的设置过期时间的 key，那样就是一场性能上的灾难。实际上 redis 是每隔 100ms随机抽取一些 key 来检查和删除的。

定期删除可能会导致很多过期 key 到了时间并没有被删除掉，可以使用惰性删除：在你获取某个 key 的时候，redis 会检查这个key，如果key过期了就会删除，不会返回任何东西。

但是实际上还是有问题的，如果定期删除漏掉了很多过期的key，没有及时的去查，也没有惰性删除，redis内存依然会耗尽。解决方式：内存淘汰机制。

（2）内存淘汰机制

redis内存淘汰机制有以下几个：

|  |  |
| --- | --- |
| noeviction | 当内存不足以容纳新写入数据时，新写入操作会报错（一般不会用）。 |
| allkeys-lru | 当内存不足以容纳新写入数据时，在**键空间**中，移除最近最少使用的 key（最常用）。 |
| allkeys-random | 当内存不足以容纳新写入数据时，在**键空间**中，随机移除某个 key（一般很少用）。 |
| volatile-lru | 当内存不足以容纳新写入数据时，在**设置了过期时间的键空间**中，移除最近最少使用的key（这个一般不太合适）。 |
| volatile-random | 当内存不足以容纳新写入数据时，在**设置了过期时间的键空间**中，随机移除某个 key。 |
| volatile-ttl | 当内存不足以容纳新写入数据时，在**设置了过期时间的键空间**中，有更早过期时间的 key 优先移除。 |

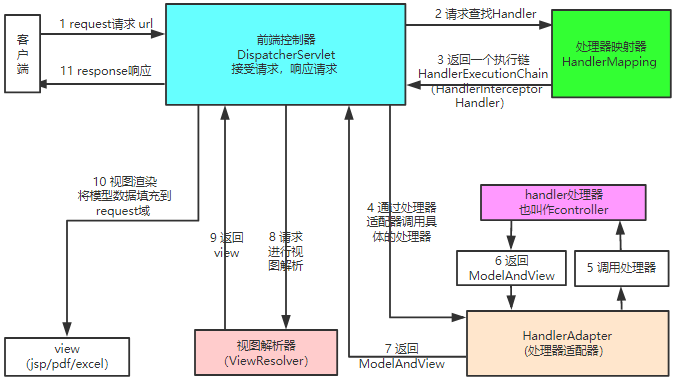
### 4.2.7 Redis的主从复制？

slave启动成功连接到master后会发送一个sync命令，master接到命令启动后台存盘进程，同时收集所有接收到修改数据的命令在后台进程执行完毕后，master将传送整个数据文件到slave以完成一次完全同步。slave接收到数据文件后将其存盘并加载内存。

master继续将新的收集修改数据的命令依次发送给slave已完成数据的同步增量。

# 5框架

## 5.1SpringMVC框架流程



第一步：发起请求到前端控制器(DispatcherServlet)；

第二步：前端控制器根据请求信息调用HandlerMapping查找 Handler（可以根据xml配置、注解进行查找）；

第三步：处理器映射器HandlerMapping向前端控制器返回Handler；

第四步：前端控制器调用处理器适配器去执行Handler；

第五步：处理器适配器去执行Handler；

第六步：Handler执行完成给处理器适配器返回ModelAndView；

第七步：处理器适配器向前端控制器返回ModelAndView；

ModelAndView是springmvc框架的一个底层对象，包括 Model和view

第八步：前端控制器请求视图解析器去进行视图解析；

根据逻辑视图名解析成真正的视图(jsp)

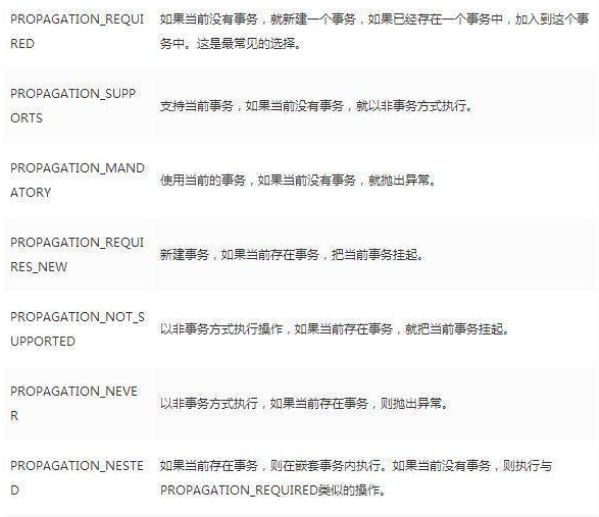
第九步：视图解析器向前端控制器返回View；

第十步：前端控制器进行视图渲染；

视图渲染将模型数据(在ModelAndView对象中)填充到request域

第十一步：前端控制器向用户响应结果；

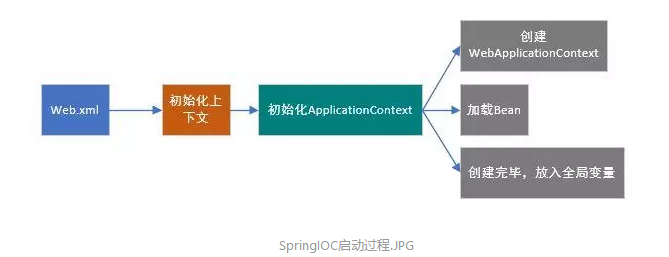
## 5.1spring事务的传播行为



required

## 5.2Spring的启动过程

spring的启动是建筑在servlet容器之上的，所有web工程的初始位置就是web.xml,它配置了servlet的上下文（context）和监听器（Listener）



## 5.3Spring、SpringMVC和SpringBoot的区别

spring boot就是一个大框架里面包含了许许多多的东西，其中spring就是最核心的内容之一，当然也就包含spring mvc。

spring mvc 只是spring 处理web层请求的一个模块。因此他们的关系大概就是这样：spring mvc  < spring < springboot。

说得更简便一些：Spring 最初利用“工厂模式”（DI）和“代理模式”（AOP）解耦应用组件。

大家觉得挺好用，于是按照这种模式搞了一个 MVC框架（一些用Spring 解耦的组件），用于开发 web 应用（ SpringMVC ）。

然后又发现每次开发都写很多样板代码，为了简化工作流程，于是开发出了一些“懒人整合包”（starter），这套就是 Spring Boot。

## 5.4SpringMVC的过滤器和拦截器的区别？

过滤器是基于函数回调的，需要依赖servlet容器，在请求进入容器后，还未进入Servlet之前进行预处理，并且请求结束返回给前端进行后期处理，只能在容器初始化时调用一次，可以自定义过滤器实现并在web.xml中注册

拦截器不依赖于servlet容器，依赖于web框架，在springmvc中就是依赖于SpringMVC框架，实现上基于Java反射机制，用处：⑴请求还没有到controller层时进行拦截。⑵请求走出controller层次，还没有到渲染时图层时进行拦截。⑶结束视图渲染，但是还没有到servlet的结束时进行拦截。对其他的一些比如直接访问静态资源的请求则没办法进行拦截处理,拦截器功在对请求权限鉴定方面确实很有用

区别：

1、拦截器是基于Java的反射机制的，而过滤器是基于函数回调

2、过滤器依赖与servlet容器，而拦截器不依赖与servlet容器

3、拦截器只能对action请求起作用，而过滤器则可以对几乎所有的请求起作用

4、拦截器可以访问action上下文、值栈里的对象，而过滤器不能。

1. 在action的生命周期中，拦截器可以多次被调用，而过滤器只能在容器初始化时被调用一次

## 5.5Spring IOC

IOC就是控制反转，是指创建对象的控制权的转移，以前创建对象的主动权和时机是由自己把控的，而现在这种权力转移到Spring容器中，降低了对象与对象的耦合。

DI就是依赖注入， 在“控制反转”中，我们是这么说的：当java程序需要使用某个对象的时候，直接去spring容器里取就好了。这句话是站在java程序的角度说的。那么反过来，站在spring容器的角度来又该怎么说呢？spring容器直接给出java程序需要的对象，那么java程序就可以直接使用这个对象了。那么这个“给出”的动作，就叫做“依赖注入”。即专业的说法是：当java程序需要使用某个对象的时候，spring容器就把这个对象及其依赖关系注入到java程序里供其使用。

https://blog.csdn.net/u012556994/article/details/81149312

比如对象 A 需要操作数据库，以前我们总是要在A中自己编写代码来获得一个Connection对象，有了spring 我们就只需要告诉spring，A中需要一个 Connection，至于这个 Connection 怎么构造，何时构造，A不需要知道。在系统运行时，spring 会在适当的时候制造一个 Connection， 然后注射到A当中，这样就完成了对各个对象之间关系的控制。A需要依赖 Connection才能正常运行，而这个 Connection 是由 spring 注入到A中的，依赖注入的名字就这么来的。那么 DI 是如何实现的呢？Java 1.3之后一个重要特征是反射（reflection），它允许程序在运行的时候动态的生成对象、执行对象的方法、改变对象的属性，spring 就是通过反射来实现注入的。

## 5.6Spring AOP

面向切面编程（AOP），将程序中那些重复的、非核心的业务提取出来，独立进行管理。最常见的就是与数据持久层关系紧密的事务、日志、安全、权限认证等方面的处理。

作为面向对象的一种补充，用于将那些与业务无关，但却对多个对象产生影响的公共行为和逻辑，抽取并封装为一个可重用的模块，这个模块被命名为“切面”（Aspect），减少系统中的重复代码，降低了模块间的耦合度，同时提高了系统的可维护性。可用于权限认证、日志、事务处理。

AOP实现的关键在于代理模式，AOP代理主要分为静态代理和动态代理。静态代理的代表为AspectJ；动态代理则以Spring AOP为代表。

1. AspectJ是静态代理的增强，所谓静态代理，就是AOP框架会在编译阶段生成AOP代理类，因此也称为编译时增强，他会在编译阶段将AspectJ(切面)织入到Java字节码中，运行的时候就是增强之后的AOP对象。
2. Spring AOP使用的动态代理，所谓的动态代理就是说AOP框架不会去修改字节码，而是每次运行时在内存中临时为方法生成一个AOP对象，这个AOP对象包含了目标对象的全部方法，并且在特定的切点做了增强处理，并回调原对象的方法。Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理：

* JDK动态代理只提供接口的代理，不支持类的代理。核心InvocationHandler接口和Proxy类，InvocationHandler通过invoke()方法反射来调用目标类中的代码，动态地将横切逻辑和业务编织在一起；接着，Proxy利用 InvocationHandler动态创建一个符合某一接口的的实例，生成目标类的代理对象。
* 如果代理类没有实现 InvocationHandler 接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成指定类的一个子类对象，并覆盖其中特定方法并添加增强代码，从而实现AOP。CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。

1. 静态代理与动态代理区别在于生成AOP代理对象的时机不同，相对来说AspectJ的静态代理方式具有更好的性能，但是AspectJ需要特定的编译器进行处理，而Spring AOP则无需特定的编译器处理。

* InvocationHandler 的 invoke(Object proxy,Method method,Object[] args)：proxy是最终生成的代理实例；method 是被代理目标实例的某个具体方法；args 是被代理目标实例某个方法的具体入参，在方法反射调用时使用。

## 5.7Spring依赖注入的三种方法

1.通过构造器来注入；

2.通过setter方法来注入；

3.通过注解来注入；

1 注入普通类型属性

（1）在要注入值的属性上面使用注解 @Value，不需要set方法

@Value(value="lucymary")

**private**Stringbname;

2 注入对象类型属性（重点）

（0）首先把对象使用注解创建

第一个注解： @Autowired（按类型by type）

@Autowired

**private** UserDaouserDao;

//注解方式不需要生成set方法

（2）第二个注解： @Qualifier

\* 这个注解使用，需要和@Autowired一起使用(by name)

\* 根据dao里面value值进行对象注入

//第二个注解使用对象类型属性注入

@Autowired

@Qualifier(value="userDao")

**private** UserDaouserDao;

第三个注解：@Resource(默认by name)

\* 这个注解实现上面两个注解功能

//第三种注解注入对象类型属性

@Resource(name="userDao")

**private** UserDaouserDao;  
后续更新，注入复杂类型，数组集合等

https://www.cnblogs.com/annofyf/articles/9772896.html

## 5.6Spring创建bean的三种方式

https://www.jianshu.com/p/646c1f657144

1调用构造器创建Bean（常用）

|  |
| --- |
| <bean id="user" class="com.beans.user">  </bean> |

2调用静态工厂方法创建Bean对象

**静态工厂方法创建bean对象的核心是：class + factory-method**

首先是封装类对象，如果需要对属性注入值，则需要定义有参构造方法或者生产setter方法；

其次是定义静态工厂方法

|  |
| --- |
| public class UserFactory {  // 在这里我们定义创建方法是调用构造方法来创建对象  public static User createPerson(){  return new User();  }  public static User createPerson(Integer id,String name){  return new User(id,name);  }  } |

然后在XML文件定义bean对象

|  |
| --- |
| <!-- 调用静态工厂方法创建bean对象 -->  <bean id="user" class="com.beans.factory.UserFactory" factory-method="createPerson">  <constructor-arg name="id" value="666"></constructor-arg>  <constructor-arg name="name" value="Leorizon"></constructor-arg>  </bean> |

使用静态工厂方法创建Bean实例需要为<bean />元素指定出id如下属性：

* class：指定静态工厂的实现类(即包名.类名)【相当于指定工厂的地址】
* factory-method：指定由静态工厂的哪个方法创建该Bean实例（方法名）【指定由工厂的哪个车间创建Bean】
* 如果静态工厂方法需要参数，则使用<constructor-arg />元素传入

3调用实例工厂方法创建Bean对象

调用实例工厂方法创建bean对象相比较静态工厂方法创建bean对象，就是将静态方法中的factory实现类先实例化一个factorBean对象，然后调用factorBean对象来创建bean对象；

|  |
| --- |
| <!-- 先配置工厂Bean，class指定该工厂的实现类，该Bean负责产生其他Bean实例 -->  <bean id="userFactory" class="com.beans.factory.UserFactory"/>  <!-- 再引用工厂Bean来配置 其他Bean -->  <bean id="user" factory-bean="userFactory" factory-method="createPerson">  <constructor-arg name="id" value="666"></constructor-arg>  <constructor-arg name="name" value="Leorizon"></constructor-arg>  </bean> |

调用实例化工厂需要为<bean />指以下两个属性

* factory-bean ：该属性指定工厂Bean的id
* factory-method：该属性指定实例工厂的工厂方法。
* 如果方法需要参数，则使用<constructor-arg />元素传入

## 5.7Spring中bean的作用域？

|  |  |
| --- | --- |
| singleton | 唯一 bean 实例，Spring 中的 bean 默认都是单例的 |
| prototype | 每次请求都会创建一个新的 bean 实例 |
| request | 每一次HTTP请求都会产生一个新的bean，该bean仅在当前HTTP request内有效。 |
| session | 每一次HTTP请求都会产生一个新的 bean，该bean仅在当前 HTTP session 内有效。 |
| global-session | 全局session作用域，仅仅在基于portlet的web应用中才有意义，Spring5已经没有了。 |

## 5.8将一个类声明为Spring的 bean 的注解有哪些?

我们一般使用 @Autowired 注解（装配属性）自动装配 bean，要想把类标识成可用于 @Autowired注解自动装配的 bean 的类,采用以下注解可实现：

* @Component：通用的注解，可标注任意类为 Spring 组件。如果一个Bean不知道属于哪个层，可以使用@Component 注解标注。
* @Repository：对应持久层即 Dao 层，主要用于数据库相关操作。
* @Service：对应服务层，主要涉及一些复杂的逻辑，需要用到 Dao层。
* @Controller：对应 Spring MVC 控制层，主要用户接受用户请求并调用 Service 层返回数据给前端页面。

## 5.9Spring中bean的生命周期

Spring Bean的生命周期只有这四个阶段

* 实例化 Instantiation
* 属性赋值 Populate
* 初始化 Initialization
* 销毁 Destruction

## 5.10Spring BeanFactory与FactoryBean的区别

1.BeanFactory

BeanFactory，以Factory结尾，表示它是一个工厂类(接口)，用于管理Bean的一个工厂。在Spring中，BeanFactory是IOC容器的核心接口，它的职责包括：实例化、定位、配置应用程序中的对象及建立这些对象间的依赖。

Spring为我们提供了许多易用的BeanFactory实现，XmlBeanFactory就是常用的一个，该实现将以XML方式描述组成应用的对象及对象间的依赖关系。XmlBeanFactory类将持有此XML配置元数据，并用它来构建一个完全可配置的系统或应用。

实例化容器

|  |
| --- |
| Resource resource = new FileSystemResource("beans.xml");  BeanFactory factory = new XmlBeanFactory(resource);  ------------------------------------------------------------------------------------------------  ClassPathResource resource = new ClassPathResource("beans.xml");  BeanFactory factory = new XmlBeanFactory(resource);  ------------------------------------------------------------------------------------------------  ApplicationContext context = new ClassPathXmlApplicationContext(new String[] {"applicationContext.xml", "applicationContext-part2.xml"});  BeanFactory factory = (BeanFactory) context; |

基本就是这些了，接着使用getBean(String beanName)方法就可以取得bean的实例；BeanFactory提供的方法及其简单，仅提供了六种方法供客户调用：

* boolean containsBean(String beanName)：判断工厂中是否包含给定名称的bean定义，若有则返回true
* Object getBean(String)：返回给定名称注册的bean实例。根据bean的配置情况，如果是singleton模式将返回一个共享实例，否则将返回一个新建的实例，如果没有找到指定bean,该方法可能会抛出异常
* Object getBean(String, Class) ：返回以给定名称注册的bean实例，并转换为给定class类型
* Class getType(String name) ：返回给定名称的bean的Class,如果没有找到指定的bean实例，则排除NoSuchBeanDefinitionException异常
* boolean isSingleton(String) ：判断给定名称的bean定义是否为单例模式
* String[] getAliases(String name) ：返回给定bean名称的所有别名

2.FactoryBean

一般情况下，Spring通过反射机制利用<bean>的class属性指定实现类实例化Bean，在某些情况下，实例化Bean过程比较复杂，如果按照传统的方式，则需要在<bean>中提供大量的配置信息。配置方式的灵活性是受限的，这时采用编码的方式可能会得到一个简单的方案。

Spring为此提供了一个org.springframework.bean.factory.FactoryBean的工厂类接口，用户可以通过实现该接口定制实例化Bean的逻辑。FactoryBean接口对于Spring框架来说占用重要的地位，Spring自身就提供了70多个FactoryBean的实现。它们隐藏了实例化一些复杂Bean的细节，给上层应用带来了便利。从Spring3.0开始，FactoryBean开始支持泛型，即接口声明改为FactoryBean<T>的形式。

以Bean结尾，表示它是一个Bean，不同于普通Bean的是：它是实现了FactoryBean<T>接口的Bean，根据该Bean的ID从BeanFactory中获取的实际上是FactoryBean的getObject()返回的对象，而不是FactoryBean本身，如果要获取FactoryBean对象，请在id前面加一个&符号来获取。

3.BeanFactory和FactoryBean的区别

* BeanFactory是接口，提供了OC容器最基本的形式，给具体的IOC容器的实现提供了规范
* FactoryBean也是接口，为IOC容器中Bean的实现提供了更加灵活的方式，FactoryBean在IOC容器的基础上给Bean的实现加上了一个简单工厂模式和装饰模式，我们可以在getObject()方法中灵活配置。其实在Spring源码中有很多FactoryBean的实现类.
* BeanFactory是个Factory，也就是IOC容器或对象工厂，FactoryBean是个Bean。在Spring中，所有的Bean都是由BeanFactory(也就是IOC容器)来进行管理的。但对FactoryBean而言，这个Bean不是简单的Bean，而是一个能生产或者修饰对象生成的工厂Bean，它的实现与设计模式中的工厂模式和修饰器模式类似。

## 5.11BeanFactory和ApplicationContext区别

1. BeanFactory负责读取bean配置文档，管理bean的加载，实例化，维护bean之间的依赖关系，负责bean的声明周期。

          2. ApplicationContext除了提供上述BeanFactory所能提供的功能之外，还提供了更完整的框架功能：

BeanFactory和ApplicationContext是Spring的两大核心接口，都可以当做Spring的容器。其中ApplicationContext是BeanFactory的子接口。

（1）BeanFactory：是Spring里面最底层的接口，包含了各种Bean的定义，读取bean配置文档，管理bean的加载、实例化，控制bean的生命周期，维护bean之间的依赖关系。ApplicationContext接口作为BeanFactory的派生，除了提供BeanFactory所具有的功能外，还提供了更完整的框架功能：

> 继承MessageSource，因此支持国际化。

> 统一的资源文件访问方式。

> 提供在监听器中注册bean的事件。

> 同时加载多个配置文件。

> 载入多个（有继承关系）上下文 ，使得每一个上下文都专注于一个特定的层次，比如应用的web层。

（2）BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。这样，我们就不能发现一些存在的Spring的配置问题。如果Bean的某一个属性没有注入，BeanFacotry加载后，直至第一次使用调用getBean方法才会抛出异常。

ApplicationContext，它是在容器启动时，一次性创建了所有的Bean。这样，在容器启动时，我们就可以发现Spring中存在的配置错误，这样有利于检查所依赖属性是否注入。ApplicationContext启动后预载入所有的单实例Bean，通过预载入单实例bean，确保当你需要的时候，你就不用等待，因为它们已经创建好了。

相对于基本的BeanFactory，ApplicationContext 唯一的不足是占用内存空间。当应用程序配置Bean较多时，程序启动较慢。

（3）BeanFactory通常以编程的方式被创建，ApplicationContext还能以声明的方式创建，如使用ContextLoader

（4）BeanFactory和ApplicationContext都支持BeanPostProcessor、BeanFactoryPostProcessor的使用，但两者之间的区别是：BeanFactory需要手动注册，而ApplicationContext则是自动注册。

## 5.12SpringBoot的核心配置文件及它的格式？

（1）application

基本可以将所有支持spring boot相关技术的配置文件都统一配置到application里面。即如果你想添加或修改一些配置的话，直接操作application.properites即可。（修改端口号、项目的url、开发环境的配置）

（2）bootstrap 配置文件

bootstrap 配置文件有以下几个应用场景。

使用 Spring Cloud Config 配置中心时，这时需要在 bootstrap 配置文件中添加连接到配置中心的配置属性来加载外部配置中心的配置信息；

一些固定的不能被覆盖的属性；

一些加密/解密的场景；

.properties 和 .yml，它们的区别主要是书写格式不同

## 5.12SpringBoot的核心注解？

**@SpringBootApplication 相当于三个注解** @SpringBootConfigryation、@EnableAutoConfiguration、@ComponentScan三个注解的组合。

1. @Configuration：提到@Configuration就要提到他的搭档@Bean。使用这两个注解就可以创建一个简单的spring配置类，可以用来替代相应的xml配置文件。
2. @EnableAutoConfiguration：能够自动配置spring的上下文，试图猜测和配置你想要的bean类，通常会自动根据你的类路径和你的bean定义自动配置，自动注册bean。
3. @ComponentScan：会自动扫描指定包下的全部标有@Component的类，并注册成bean，当然包括@Component下的子注解@Service,@Repository,@Controller。

https://www.cnblogs.com/MaxElephant/p/8108140.html

## SpringBoot的启动过程？

https://blog.csdn.net/u010811939/article/details/80592461

SpringBoot启动的时候，不论调用什么方法，都会构造一个SpringApplication的实例，然后调用这个实例的run方法，这样就表示启动SpringBoot。

（1）在run方法调用之前，也就是构造SpringApplication的时候会进行初始化的工作，初始化的时候会做以下几件事：

* + 1.把参数sources设置到SpringApplication属性中，这个sources可以是任何类型的参数。
  + 2.判断是否是web程序，并设置到webEnvironment这个boolean属性中。
  + 3.找出所有的初始化器，默认有5个，设置到initializers属性中。
  + 4.找出所有的应用程序监听器，默认有9个，设置到listeners属性中。
  + 5.找出运行的主类(main class)。

（2）SpringApplication构造完成之后调用run方法，启动SpringApplication，run方法执行的时候会做以下几件事：

* + 1.构造一个StopWatch，观察SpringApplication的执行。
  + 2.找出所有的SpringApplicationRunListener并封装到SpringApplicationRunListeners中，用于监听run方法的执行。监听的过程中会封装成事件并广播出去让初始化过程中产生的应用程序监听器进行监听。
  + 3.构造Spring容器(ApplicationContext)，并返回。
* 3.1判断是否是web环境，是的话构造AnnotationConfigEmbeddedWebApplicationContext，否则构造AnnotationConfigApplicationContext。
* 3.2 初始化过程中产生的初始化器在这个时候开始工作。
* 3.3 Spring容器的刷新(完成bean的解析、各种processor接口的执行、条件注解的解析等等)。
* 4.从Spring容器中找出ApplicationRunner和CommandLineRunner接口的实现类并排序后依次执行。

## 5.14什么是Mybatis

1. Mybatis是一个半ORM（对象关系映射）框架，它内部封装了JDBC，开发时只需要关注SQL语句本身，不需要花费精力去处理加载驱动、创建连接、创建statement等繁杂的过程。程序员直接编写原生态sql，可以严格控制sql执行性能，灵活度高。
2. MyBatis 可以使用 XML 或注解来配置和映射原生信息，将 POJO映射成数据库中的记录，避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。
3. 通过xml 文件或注解的方式将要执行的各种 statement 配置起来，并通过java对象和 statement中sql的动态参数进行映射生成最终执行的sql语句，最后由mybatis框架执行sql并将结果映射为java对象并返回。（从执行sql到返回result的过程）。

## 5.15Mybatis中#{}和${}的区别

1. #{}是预编译处理，${}是字符串替换。
2. Mybatis在处理#{}时，会将sql中的#{}替换为?号，调用PreparedStatement的set方法来赋值；使用#{}可以有效的防止SQL注入，提高系统安全性。
3. Mybatis在处理${}时，就是把${}替换成变量的值。

#{}在mybatis中的底层是运用了PreparedStatement 预编译，传入的参数会以 ? 形式显示，因为sql的输入只有在sql编译的时候起作用，当sql预编译完后，传入的参数就仅仅是参数，不会参与sql语句的生成，而${}则没有使用预编译，传入的参数直接和sql进行拼接，由此会产生sql注入的漏洞。

https://blog.csdn.net/weixin\_45179130/article/details/90761966

## 5.16Mybatis中实体类中的属性名和表中字段名不一样？

（1） 通过在查询的sql语句中定义字段名的别名，让字段名的别名和实体类的属性名一致。

|  |
| --- |
| <select id=”selectorder” parametertype=”int” resultetype=”me.gacl.domain.order”>  select order\_id id, order\_no orderno ,order\_price price form orders where order\_id=#{id};  </select> |

（2） 通过<resultMap>来映射字段名和实体类属性名的一一对应的关系。

|  |
| --- |
| <select id="getOrder" parameterType="int" resultMap="orderresultmap">  select \* from orders where order\_id=#{id}  </select>  <resultMap type=”me.gacl.domain.order” id=”orderresultmap”>  <!–用id属性来映射主键字段–>  <id property=”id” column=”order\_id”>  <!–用result属性来映射非主键字段，property为实体类属性名，column为数据表中的属性–>  <result property = “orderno” column =”order\_no”/>  <result property=”price” column=”order\_price” />  </reslutMap> |

## 5.17Mybatis的一级、二级缓存

1）一级缓存 Mybatis的指SqlSession。Mybatis默认开启一级缓存。在同一个SqlSession中，执行相同的查询SQL，第一次会去查询数据库，并写到缓存中；第二次直接从缓存中取。当执行SQL时两次查询中间发生了增删改操作，则SqlSession的缓存清空。 Mybatis的内部缓存使用一个HashMap，key为hashcode+statementId+sql语句。Value为查询出来的结果集映射成的java对象。 SqlSession执行insert、update、delete等操作commit后会清空该SQLSession缓存。

2）二级缓存 指mapper映射文件。二级缓存的作用域是同一个namespace下的mapper映射文件内容，多个SqlSession共享。Mybatis需要手动设置启动二级缓存。在同一个namespace下的mapper文件中，执行相同的查询SQL，第一次会去查询数据库，并写到缓存中；第二次直接从缓存中取。当执行SQL时两次查询中间发生了增删改操作，则二级缓存清空。

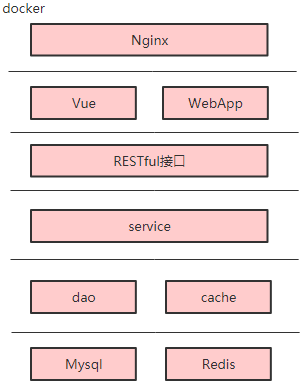
## 5.18mybatis返回结果转换为JSON格式？

直接用JSONArray无法实现转换，最后使用google的Gson配合JSONArray成功将结果集转换为json：

|  |
| --- |
| Gson gson = new Gson();  String jsonResult = gson.toJson(dataList, new TypeToken<List>(){}.getType());  JSONArray object = JSONArray.parseArray(jsonResult); |

# 6项目总结

## 6.1项目架构



底层mysql是存储数据，redis是缓存，dao层采用mybatis操作数据库，cache层操作redis，service层处理业务逻辑，rest api层为前端提供rest接口。前端这边用Vue+WebApp进行模块化和打包部署。网关nginx进行负载均衡。mysql、redis、nginx和springboot应用都放在docker里部署。

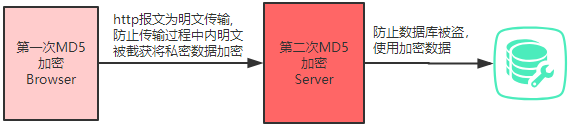
## 6.2Shiro密码加密（权限管理）

对于同一密码，同一加密算法会产生相同的hash值。这样，当用户进行身份验证时，对用户输入的明文密码应用相同的hash加密算法，得出一个hash值，然后使用该hash值和之前存储好的密文值进行对照，如果两个值相同，则密码认证成功，否则密码认证失败。

在shiro中也提供了编码，解码，加密，加密算法实现等等一系列的内容。加salt加密，salt是唯一值，比如身份证等，主要是为了解决多用户密码相同时加密后密码相同的问题，加salt后的密码是唯一值；

在Spring配置文件中可以配置自定义realm，指定加密的算法（MD5）以及加密的次数（两次）；

## 6.3为什么使用两次MD5加密？



第一次MD5+固定Salt加密:

第一次MD5是在浏览器端，通过js文件中进行MD5加密，由于http报文的明文传输，如果传输的数据被截获(例如使用fiddler等抓包工具)，那用户密码将暴露无疑，所以我们使用MD5和固定Salt将表单的关键数据(例如密码)进行加密。

第二次MD5+随机Salt加密:

第二次MD5是在服务器端完成的，它将用户的关键信息进行加密后存入到数据库中，这样如果数据库的被盗后也得不到用户原始的密码信息。注意：

* MD5加密是不可逆的，但是你可能会发现有一些md5在线解密的网站能够通过密文获得你原始的数据，但它并不是反推出来的，而是通过反向查询，穷举字符组合的方式，创建了明文密文对应查询数据库，这样获得的你的原始数据。

## 6.4MD5加密原理？

MD5算法的过程分为四步：处理原文，设置初始值，循环加工，拼接结果。

* 第一步:处理原文：
  + 首先，我们计算出原文长度(bit)对512求余的结果，如果不等于448，就需要填充原文使得原文对512求余的结果等于448。填充的方法是第一位填充1，其余位填充0。填充完后，信息的长度就是512\*N+448。
  + 之后，用剩余的位置（512-448=64位）记录原文的真正长度，把长度的二进制值补在最后。这样处理后的信息长度就是512\*(N+1)。
* 第二步:设置初始值：
  + MD5的哈希结果长度为128位，按每32位分成一组共4组。这4组结果是由4个初始值A、B、C、D经过不断演变得到。MD5的官方实现中，A、B、C、D的初始值由官方给定。
* 第三步:循环加工：
  + 每一次循环都会让旧的ABCD产生新的ABCD。循环的次数由处理后的原文长度决定。
* 第四步:拼接结果：
  + 把循环加工最终产生的A，B，C，D四个值拼接在一起，转换成（32位的16进制）字符串即可。

## 6.5如何防止通过URL地址栏直接访问页面？

（1）将所有页面放在WEB-INF目录下

* WEB-INF是Java的web应用安全目录，只对服务端开放，对客户端是不可见的。所以可以把除首页（index.jsp）以外的页面都放在WEB-INF目录下，这样就无法通过URL直接访问页面了。

（2）http请求头字段中的 referer值

* 根据HTTP协议，在HTTP头中有一个字段叫Referer，它记录了该HTTP请求的来源地址。在通常情况下，访问一个安全受限页面的请求必须来自于同一个网站。比如，要防御CSRF（跨站点请求伪造）攻击，网站只需要对于每一个请求验证其Referer值，如果是以bank.test开头的域名，则说明该请求是来自自己网站的请求，是合法的。如果Referer是其他网站的话，就有可能是CSRF攻击，则拒绝该请求。在Java 中，拦截器是由Filter来实现的。

（3）在请求地址中添加token并验证

* CSRF攻击之所以能够成功，是因为攻击者可以伪造用户的请求，该请求中所有的用户验证信息都存在于Cookie中，因此攻击者可以在不知道这些验证信息的情况下直接利用用户自己的Cookie来通过安全验证。由此可知，抵御CSRF攻击的关键在于：在请求中放入攻击者所不能伪造的信息，并且该信息不存在于Cookie之中。因此可以在HTTP请求中以参数的形式加入一个随机产生的token，并在服务器端建立一个拦截器来验证这个token，如果请求中没有token或者token内容不正确，则认为可能是CSRF攻击而拒绝该请求。

（4）在HTTP头中自定义属性并验证

* 自定义属性的方法也是使用token并进行验证，和前一种方法不同的是，这里并不是把token以参数的形式置于HTTP请求之中，而是把它放到HTTP头中自定义的属性里。通过XMLHttpRequest这个类，可以一次性给所有该类请求加上csrftoken这个HTTP头属性，并把token值放入其中。这样解决了前一种方法在请求中加入token的不便，同时，通过这个类请求的地址不会被记录到浏览器的地址栏，也不用担心token会通过Referer泄露到其他网站。

（5）其他防御方法

* CSRF攻击是有条件的，当用户访问恶意链接时，认证的cookie仍然有效，所以当用户关闭页面时要及时清除认证cookie，对支持TAB模式（新标签打开网页）的浏览器尤为重要。
* 尽量少用或不要用request()类变量，获取参数时指定是request.form()还是request.querystring ()，这样有利于阻止CSRF漏洞攻击，此方法只不能完全防御CSRF攻击，只是一定程度上增加了攻击的难度。

## 6.6如何防止中间人攻击？

* 对Https数据先自己进行加密（Ex：AES、RSA），再Https传输，这样即使截取到Https数据，也看不到明文。
* Https双向验证 ：服务器端对请求它的客户端要进行身份验证，客户端对自己所请求的服务器也会做身份验证。服务端一旦验证到请求自己的客户端为不可信任的，服务端就拒绝继续通信。客户端如果发现服务端为不可信任的，那么也中止通信。
* 强制使用SSL pinning：这是对抗MiTM（中间人）攻击的另一种方式。使用有效证书颁发机构提供的证书是第一步，它是通过返回的受信任的根证书以及是否与主机名匹配来验证该服务器提供的证书的有效性。通过SSL pinning可以验证客户端检查服务器证书的有效性。
* SSL/TLS 系列中有五种协议：SSL v2，SSL v3，TLS v1.0，TLS v1.1和TLS v1.2：
* SSL v2 是不安全的，不能使用。当与 HTTP一起使用时，SSL v3 是不安全的，也是过时的，不应该被使用。
* TLS v1.0 也是不应该使用的传统协议，但在实践中通常仍然是必需的。其主要弱点（BEAST）在现代浏览器中得到缓解，但其他问题仍然存在。
* TLS v1.1 和 v1.2 都没有已知的安全问题，只有 TLS v1.2 提供了现代的加密算法，因为它是唯一提供现代认证加密（也称为 AEAD）的版本。

## 6.7怎样实现敏感词过滤？

在文字过滤算法中，DFA（有穷自动机）算法是一个比较高效的算法，它是通过event和当前的state得到下一个state，不断地进行状态的转换。

在实现的时候，可以通过查询实现状态的转换，通过S query U、V，通过U query V、P，通过V query U P。通过这样的转变我们可以将状态的转换转变为使用Java集合的查找。这样就可以将敏感词库构建成一颗一颗的树（类似于字典树），再判断一个词是否是敏感词就大大的减少了匹配的范围，比如我们要判断日本人，根据第一个字我们就可以确认需要检索的是哪棵树，然后再在这棵树中进行检索。

但是如何来判断一个敏感词已经结束了呢？利用标识位来判断。

可以使用hashmap来实现DFA算法，具体过程如下：加入查找的敏感词是“日本人”：

* + 1、在hashMap中查询“日”看其是否在hashMap中存在，如果不存在，则证明已“日”开头的敏感词还不存在，则我们直接构建这样的一棵树。判断该字是否为该词中的最后一个字。若是表示敏感词结束，设置标志位isEnd = 1，否则设置标志位isEnd = 0。
  + 2、如果在hashMap中查找到了，表明存在以“日”开头的敏感词，设置hashMap = hashMap.get(“日”)，跳至1，依次匹配“本”、“人”。

## 6.8秒杀系统如何优化

（1）秒杀系统特点：

* 秒杀业务简单，卖家查询，买家下订单减库存。
* 秒杀时网站访问流量激增，出现峰值；
* 访问请求数量远大于实际需求量。

（2）整体优化思路：

* 限流：屏蔽掉无用的流量，允许少部分流量流向后端。
  + 令牌桶算法
    - 令牌桶算法(Token Bucket)和 Leaky Bucket 效果一样但方向相反的算法,更加容易理解.随着时间流逝,系统会按恒定1/QPS时间间隔(如果QPS=100,则间隔是10ms)往桶里加入Token(想象和漏洞漏水相反,有个水龙头在不断的加水),如果桶已经满了就不再加了.新请求来临时,会各自拿走一个Token,如果没有Token可拿了就阻塞或者拒绝服务.
  + 漏桶算法
  + 漏桶(Leaky Bucket)算法思路很简单,水(请求)先进入到漏桶里,漏桶以一定的速度出水(接口有响应速率),当水流入速度过大会直接溢出(访问频率超过接口响应速率),然后就拒绝请求,可以看出漏桶算法能强行限制数据的传输速率
* 削峰：瞬时大流量峰值容易压垮系统，解决这个问题是重中之重。常用的消峰方法有异步处理、缓存和消息中间件等技术。
* 异步处理：秒杀系统是一个高并发系统，采用异步处理模式可以极大地提高系统并发量，其实异步处理就是削峰的一种实现方式。
* 内存缓存：秒杀系统最大的瓶颈一般都是数据库读写，由于数据库读写属于磁盘IO，性能很低，如果能够把部分数据或业务逻辑转移到内存缓存，效率会有极大地提升。
* 可拓展：当然如果我们想支持更多用户，更大的并发，最好就将系统设计成弹性可拓展的，如果流量来了，拓展机器就好了。像淘宝、京东等双十一活动时会增加大量机器应对交易高峰。
* 消息队列：消息队列可以削峰，将拦截大量并发请求，这也是一个异步处理过程，后台业务根据自己的处理能力，从消息队列中主动的拉取请求消息进行业务处理。

（3）具体优化

* 中间代理层：可利用负载均衡（例如反响代理Nginx等）使用多个服务器并发处理请求，减小服务器压力。
* 控制层(网关层)：限制同一UserID访问频率：尽量拦截浏览器请求，但针对某些恶意攻击或其它插件，在服务端控制层需要针对同一个访问uid，限制访问频率。
* 服务层：当用户量非常大的时候，拦截流量后的请求访问量还是非常大，此时仍需进一步优化。
  + 业务分离：将秒杀业务系统和其他业务分离，单独放在高配服务器上，可以集中资源对访问请求抗压。
  + 采用消息队列缓存请求：将大流量请求写到消息队列缓存，利用服务器根据自己的处理能力主动到消息缓存队列中抓取任务处理请求，数据库层订阅消息减库存，减库存成功的请求返回秒杀成功，失败的返回秒杀结束。
  + 利用缓存应对读请求：对于读多写少业务，大部分请求是查询请求，所以可以读写分离，利用缓存分担数据库压力。
  + 利用缓存应对写请求：缓存也是可以应对写请求的，可把数据库中的库存数据转移到Redis缓存中，所有减库存操作都在Redis中进行，然后再通过后台进程把Redis中的用户秒杀请求同步到数据库中。
* 数据库层：数据库层是最脆弱的一层，一般在应用设计时在上游就需要把请求拦截掉，数据库层只承担“能力范围内”的访问请求。所以，上面通过在服务层引入队列和缓存，让最底层的数据库高枕无忧。如果不使用缓存来作为中间缓冲而是直接访问数据库的话，可以对数据库进行优化，减少数据库压力。

（4）案例：利用消息中间件和Redis缓存实现

Redis是一个分布式缓存系统，支持多种数据结构，可利用Redis轻松实现一个强大的秒杀系统。

我们可以采用Redis 最简单的key-value数据结构，用一个原子类型的变量值(AtomicInteger)作为key，把用户id作为value，库存数量便是原子变量的最大值。对于每个用户的秒杀，我们使用RPUSH key value插入秒杀请求， 当插入的秒杀请求数达到上限时，停止所有后续插入。

然后我们可以再启动多个工作线程，使用LPOP key读取秒杀成功者的用户id，然后再操作数据库做最终的下订单减库存操作。

当然，上面Redis也可以替换成消息中间件RocketMQ，也可以将缓存和消息中间件组合起来，缓存系统负责接收记录用户请求，消息中间件负责将缓存中的请求同步到数据库。

## 6.9如何解决高并发问题？（分布式锁）

（1）业务场景

所谓秒杀，从业务角度看，是短时间内多个用户“争抢”资源，这里的资源在大部分秒杀场景里是商品；将业务抽象，技术角度看，秒杀就是多个线程对资源进行操作，所以实现秒杀，就必须控制线程对资源的争抢，既要保证高效并发，也要保证操作的正确。

（2）可能的解决方式

* 秒杀在技术层面的抽象应该就是一个方法，在这个方法里可能的操作是将商品库存-1，将商品加入用户的购物车等等，在不考虑缓存的情况下应该是要操作数据库的。那么最简单直接的实现就是在这个方法上加上synchronized关键字，通俗的讲就是锁住整个方法；
* 锁住整个方法这个策略简单方便，但是似乎有点粗暴。可以稍微优化一下，只锁住秒杀的代码块，比如写数据库的部分；
* 既然有并发问题，那我就让他“不并发”，将所有的线程用一个队列管理起来，使之变成串行操作，自然不会有并发问题。

上面解决方式带来的问题：**锁的粒度比较大**，如果两个线程同时执行秒杀方法，这两个线程操作的是不同的商品,从业务上讲应该是可以同时进行的，但是如果采用第一二种方法，这两个线程也会去争抢同一个锁，这其实是不必要的。那么如何将锁控制在更细的粒度上呢？可以考虑为每个商品设置一个互斥锁，以和商品ID相关的字符串为唯一标识，这样就可以做到只有争抢同一件商品的线程互斥，不会导致所有的线程互斥。分布式锁恰好可以帮助我们解决这个问题。

（3）redis 分布式锁

分布式锁是控制分布式系统之间同步访问共享资源的一种方式。分布式锁有 3 个重要的考量点：互斥（只能有一个客户端获取锁）；不能死锁；容错（只要大部分 redis 节点创建了这把锁就可以）；

第一个最普通的实现方式，就是在redis里使用 **setnx** 命令创建一个 key，这样就算加锁。执行这个命令就可以实现：使用SETNX key value，（通常value使用的是系统的时间戳）如果成功返回1，失败返回0；

当然上面可能出现的问题就是：如果一个持有锁的客户端失败或崩溃了不能释放锁，就会发生阻塞；可以通过锁的键对应的时间戳来判断这种情况是否发生了，如果当前的时间已经大于key对应的value的值，说明该锁已失效，可以被重新使用。不能通过简单的DEL删除锁，然后重新SETNX；因为会出现下面的问题：

|  |
| --- |
| C0操作超时了，但它还持有着锁，C1和C2读取key检查时间戳，先后发现超时了。  C1 发送DEL lock.foo  C1 发送SETNX key 并且成功了。  C2 发送DEL key  C2 发送SETNX key 并且成功了。  这样一来，C1，C2都拿到了锁！就出现问题了。 |

这种问题是可以避免的：

|  |
| --- |
| * C3发送SETNX key 想要获得锁，由于C0还持有锁，所以Redis返回给C3一个0 * C3发送GET key 以检查锁是否超时了，如果没超时，则等待或重试。反之，如果已超时，C3通过下面的操作来尝试获得锁：GETSET key <current Unix time + lock timeout + 1> * 通过GETSET（将给定的key的值设为value，并返回key的旧值），C3拿到的时间戳如果仍然是超时的，那就说明，C3如愿以偿拿到锁了。 * 如果在C3之前，有个叫C4的客户端比C3快一步执行了上面的操作，那么C3拿到的时间戳是个未超时的值，这时，C3没有获得锁，需要再次等待或重试。 |

## 6.10订单处理流程

* 1首先，扣减服务作为下单流程的入口，会先对商品的库存做扣减。同样它会检查商品是否还有库存，如果库存足够，创建订单，否则返回失败；
* 2由于订单对应的操作步骤比较多，为了让流量变得平滑，这里使用消息队列MQ（为什么使用MQ？主要原因是由于在高并发的环境下，由于服务器来不及同步处理，请求往往会发生堵塞的现象，比如说大量的增加、修改（insert、updata）之类的请求同时到达mysql，直接导致无数的行锁，表锁、甚至最后请求堆积过多，从而触发too many connections错误。通过使用消息队列MQ，我们就可以异步处理请求，从而缓解系统的压力。）存放每个订单请求，等待订单处理服务完成具体业务。订单处理服务实现多线程，它们不断监听队列中的消息。一旦发现有新订单请求，就取出订单进行后续处理。
* 3订单处理服务处理完订单以后会把结果写到数据库。写数据库是 IO 操作，耗时长。所以，在写数据库的同时，会把结果先写入缓存中，这样用户是可以第一时间查询自己是否下单成功了。结果写入数据库，这个操作有可能成功也有可能失败，所以为了保证数据的最终一致性，我们用订单结果同步服务不断的对比缓存和数据库中的订单结果信息，一旦发现不一致，会去做重试操作。如果重试依旧不成功，会重写信息到缓存，让用户知道失败原因。

## 6.11防止超卖

数据库加唯一索引：防止用户重复该买  
SQL加库存数量判断，防止库存编程负数

为了防止超卖现象，我会将这个库存信息放到缓存中。比如使用Redis存放库存信息，高并发的时候会出现资源争夺的情况，为了解决这个问题我们需要实现分布式锁：每一次用户去访问订单请求接口的时候，同时会去访问 Redis 中存放的库存信息，每接受用户一次请求，都会从 Redis 的库存中减去 1 个商品库存量。当任何一个进程访问 Redis 中的库存资源时，其他进程是不能访问的。

还需要考虑锁的释放问题：如果锁长期没有释放，需要考虑锁的过期时间，需要设置两个超时时间：

* 1资源本身的超时时间，一旦资源被使用一段时间还没有被释放，Redis 会自动释放掉该资源给其他服务使用。
* 2服务获取资源的超时时间，一旦一个服务获取资源一段时间后，不管该服务是否处理完这个资源，都需要释放该资源给其他服务使用。

用户大量提交。根据用户提交的产品信息，获取到redis中需要的key值，查询缓存（为了保证缓存有效，如果第一次没有查询到，可以到数据库查询，然后在缓存一下，不过一般不会出现），得到库存量，判断当前库存是否大于零，如果大于零，判断当前的set集合中是否用该用户ID，如果没有，减库存并且将用户的ID放入集合中，并对库存减一，如果库存为0，提示用户，商品已售完等文案信息，如果集合中已经存在该用户id，则不做任何处理，直接处理下一个请求。直到最后库存售完，上面的过程可以利用redis事务和watch功能完成对数据一致性的控制即**超卖问题。**

https://www.jianshu.com/p/c4a743bbe3a4

## 6.12项目中缓存是如何使用的？

（1）为什么要用缓存？

* 第一：Redis缓存实现session共享

首先我的项目中需要用户进行登录，如果只有一台服务器的话，直接使用session来存储用户的登录信息完全没问题，但是假如我的项目部署到两台或多台服务器上（serverA和serverB，两台都是主服务器，或者一台是主服务器，一台是备用服务器），在A服务器上登录后，并且采用的session存储是进程内存储的方式，在访问其他页面的情况下，有可能将我的请求转发到服务器B，但服务器B进程内并没有保存你的用户信息，在校验权限的时候，会将你重定向到登录页面。

解决方式：利用Redis统一管理session，服务器每次读写session的时候，都去访问Redis服务器。

* 第二：使用Redis做缓存主要有两个用途：**高性能、高并发**。
* **高性能**：假设这么个场景，你有个操作，一个请求过来，吭哧吭哧你各种乱七八糟操作 mysql，半天查出来一个结果，耗时 600ms。但是这个结果可能接下来几个小时都不会变了，或者变了也可以不用立即反馈给用户。那么此时咋办？缓存啊，折腾 600ms 查出来的结果，扔缓存里，一个 key 对应一个 value，下次再有人查，别走 mysql了，直接从缓存里，通过一个 key 查出来一个 value，2ms 搞定。性能提升 300 倍。就是说对于一些需要复杂操作耗时查出来的结果，且确定后面不怎么变化，但是有很多读请求，那么直接将查询出来的结果放在缓存中，后面直接读缓存就好。
* **高并发**：mysql 这么重的数据库，压根儿设计不是让你玩儿高并发的，虽然也可以玩儿，但是天然支持不好。mysql 单机支撑到 `2000QPS` 也开始容易报警了。所以要是你有个系统，高峰期一秒钟过来的请求有 1万，那一个 mysql 单机绝对会死掉。你这个时候就只能上缓存，把很多数据放缓存，别放 mysql。缓存功能简单，说白了就是 `key-value` 式操作，单机支撑的并发量轻松一秒几万十几万，支撑高并发 so easy。单机承载并发量是 mysql 单机的几十倍。（缓存是走内存的，内存天然就支撑高并发）。

（2）用了缓存之后会有什么不良后果？

常见的缓存问题有以下几个：

1. 缓存与数据库双写不一致；
2. 缓存雪崩、缓存穿透；
3. 缓存并发竞争；

* **缓存和数据库双写不一致**：
  + 最经典的缓存+数据库读写的模式，就是 Cache Aside Pattern。
    - **读的时候，先读缓存，缓存没有的话，就读数据库，然后取出数据后放入缓存，同时返回响应。**
    - **更新的时候，先更新数据库，然后再删除缓存**。

为什么是删除缓存，而不是更新缓存？

|  |
| --- |
| 原因很简单，很多时候，**在复杂点的缓存场景，缓存不单单是数据库中直接取出来的值**。比如可能更新了某个表的一个字段，然后其对应的缓存，是需要查询另外两个表的数据并进行运算，才能计算出缓存最新的值的。  另外**更新缓存的代价有时候是很高的**。是不是说，每次修改数据库的时候，都一定要将其对应的缓存更新一份？也许有的场景是这样，但是对于比较复杂的缓存数据计算的场景，就不是这样了。如果你频繁修改一个缓存涉及的多个表，缓存也频繁更新。但是问题在于，这个缓存到底会不会被频繁访问到？  例如，一个缓存涉及的表的字段，在 1 分钟内就修改了 20 次，或者是 100 次，那么缓存更新 20 次、100 次；但是这个缓存在 1 分钟内只被读取了 1 次，有**大量的冷数据**。实际上，如果你只是删除缓存的话，那么在 1 分钟内，这个缓存不过就重新计算一次而已，开销大幅度降低。用到缓存才去算缓存。  其实删除缓存，而不是更新缓存，就是一个 **lazy 计算**的思想，不要每次都重新做复杂的计算，不管它会不会用到，而是让它到需要被使用的时候再重新计算。像 mybatis，hibernate，都有懒加载思想。查询一个部门，部门带了一个员工的 list，没有必要每次查询部门，把里面的 1000 个员工的数据也同时查出来。80% 的情况，查这个部门，就只是要访问这个部门的信息就可以了。先查部门，同时要访问里面的员工，那么这个时候只有在你要访问里面的员工的时候，才会去数据库里面查询 1000 个员工。 |

* + 最初级的缓存不一致问题及解决方案

问题：先更新数据库，再删除缓存。如果删除缓存失败了，那么会导致数据库中是新数据，缓存中是旧数据，数据就出现了不一致。



解决思路：先删除缓存，再更新数据库。如果数据库更新失败了，那么数据库中是旧数据，缓存中是空的，那么数据不会不一致。因为读的时候缓存没有，所以去读了数据库中的旧数据，然后更新到缓存中。

* + 比较复杂的数据不一致问题分析

数据发生了变更，先删除了缓存，然后要去修改数据库，此时还没修改。一个请求过来，去读缓存，发现缓存空了，去查询数据库，**查到了修改前的旧数据**，放到了缓存中。随后数据变更的程序完成了数据库的修改。完了，数据库和缓存中的数据不一样…

上面的情况只有在上亿流量高并发场景下才会发生，解决方式：

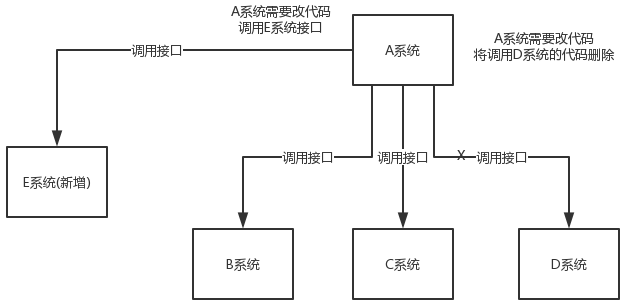
|  |
| --- |
| 更新数据的时候，根据**数据的唯一标识**，将操作路由之后，发送到一个jvm 内部队列中。读取数据的时候，如果发现数据不在缓存中，那么将重新读取数据+更新缓存的操作，根据唯一标识路由之后，也发送同一个 jvm 内部队列中。  一个队列对应一个工作线程，每个工作线程**串行**拿到对应的操作，然后一条一条的执行。这样的话，一个数据变更的操作，先删除缓存，然后再去更新数据库，但是还没完成更新。此时如果一个读请求过来，没有读到缓存，那么可以先将缓存更新的请求发送到队列中，此时会在队列中积压，然后同步等待缓存更新完成。  这里有一个**优化点**，一个队列中，其实**多个更新缓存请求串在一起是没意义的**，因此可以做过滤，如果发现队列中已经有一个更新缓存的请求了，那么就不用再放个更新请求操作进去了，直接等待前面的更新操作请求完成即可。  待那个队列对应的工作线程完成了上一个操作的数据库的修改之后，才会去执行下一个操作，也就是缓存更新的操作，此时会从数据库中读取最新的值，然后写入缓存中。  如果请求还在等待时间范围内，不断轮询发现可以取到值了，那么就直接返回；如果请求等待的时间超过一定时长，那么这一次直接从数据库中读取当前的旧值。 |

## 6.13为什么使用消息队列？消息队列优点和缺点？

消息队列常见的使用场景有很多，但是比较核心的有 3 个：**解耦、异步、削峰**。

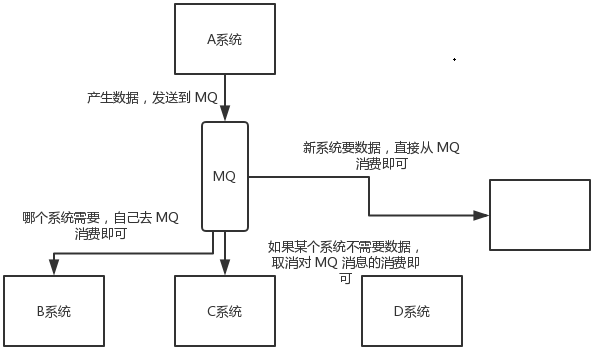
* 解耦

A 系统发送数据到 BCD 三个系统，通过接口调用发送。如果 E 系统也要这个数据呢？那如果 C 系统现在不需要了呢？A 系统负责人几乎崩溃......



在这个场景中，A 系统跟其它各种乱七八糟的系统严重耦合，A 系统产生一条比较关键的数据，很多系统都需要 A 系统将这个数据发送过来。A 系统要时时刻刻考虑 BCDE 四个系统如果挂了该咋办？要不要重发，要不要把消息存起来？

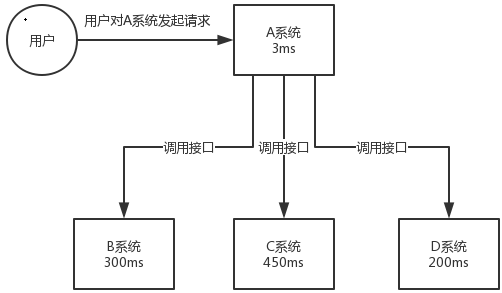
如果使用 MQ，A 系统产生一条数据，发送到 MQ 里面去，哪个系统需要数据自己去 MQ 里面消费。如果新系统需要数据，直接从 MQ 里消费即可；如果某个系统不需要这条数据了，就取消对 MQ 消息的消费即可。这样下来，A 系统不需要去考虑要给谁发送数据，不需要维护这个代码，也不需要考虑人家是否调用成功、失败超时等情况。



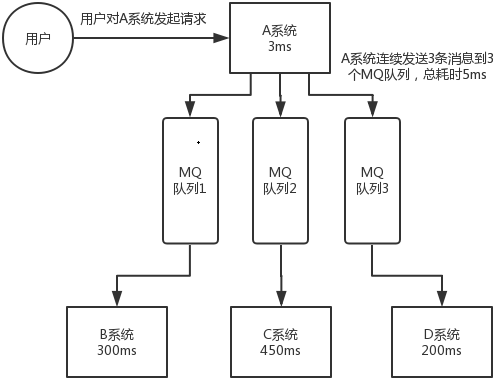
总结：通过一个 MQ，Pub/Sub 发布订阅消息这么一个模型，A 系统就跟其它系统彻底解耦了。

* 异步

A 系统接收一个请求，需要在自己本地写库，还需要在 BCD 三个系统写库，自己本地写库要 3ms，BCD 三个系统分别写库要 300ms、450ms、200ms。最终请求总延时是 3 + 300 + 450 + 200 = 953ms，接近 1s，用户感觉搞个什么东西，慢死了慢死了。用户通过浏览器发起请求，等待个 1s，这几乎是不可接受的。



对于用户直接的操作，一般要求是每个请求都必须在 200 ms 以内完成，对用户几乎是无感知的。如果**使用 MQ**，那么 A 系统连续发送 3 条消息到 MQ 队列中，假如耗时 5ms，A 系统从接受一个请求到返回响应给用户，总时长是 3 + 5 = 8ms，对于用户而言，其实感觉上就是点个按钮，8ms 以后就直接返回了。

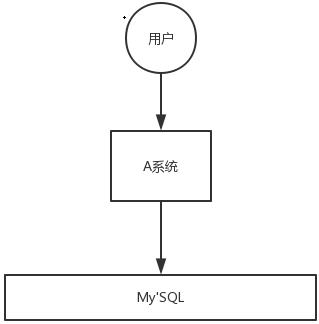


* 削峰

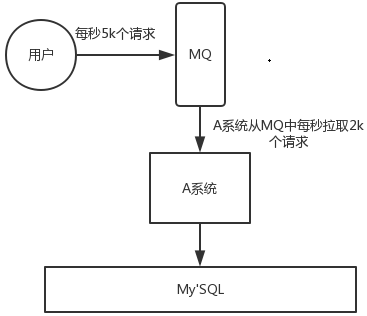
每天 0:00 到 12:00，A 系统风平浪静，每秒并发请求数量就 50 个。结果每次一到 12:00 ~ 13:00 ，每秒并发请求数量突然会暴增到 5k+ 条。但是系统是直接基于 MySQL 的，大量的请求涌入 MySQL，每秒钟对 MySQL 执行约 5k 条 SQL。

一般的 MySQL，扛到每秒 2k 个请求就差不多了，如果每秒请求到 5k 的话，可能就直接把 MySQL 给打死了，导致系统崩溃，用户也就没法再使用系统了。

但是高峰期一过，到了下午的时候，就成了低峰期，可能也就 1w 的用户同时在网站上操作，每秒中的请求数量可能也就 50 个请求，对整个系统几乎没有任何的压力。



如果使用 MQ，每秒 5k 个请求写入 MQ，A 系统每秒钟最多处理 2k 个请求，因为 MySQL 每秒钟最多处理 2k 个。A 系统从 MQ 中慢慢拉取请求，每秒钟就拉取 2k 个请求，不要超过自己每秒能处理的最大请求数量就 ok，这样下来，哪怕是高峰期的时候，A 系统也绝对不会挂掉。而 MQ 每秒钟 5k 个请求进来，就 2k 个请求出去，结果就导致在中午高峰期（1 个小时），可能有几十万甚至几百万的请求积压在 MQ 中。



这个短暂的高峰期积压是 ok 的，因为高峰期过了之后，每秒钟就 50 个请求进 MQ，但是 A 系统依然会按照每秒 2k 个请求的速度在处理。所以说，只要高峰期一过，A 系统就会快速将积压的消息给解决掉。

消息队列有什么缺点？

* 系统可用性降低：系统引入的外部依赖越多，越容易挂掉。本来你就是 A 系统调用 BCD 三个系统的接口就好了，人 ABCD 四个系统好好的，没啥问题，你偏加个 MQ 进来，万一 MQ 挂了咋整，MQ 一挂，整套系统崩溃，如何保证消息队列的高可用？
* 系统复杂度提高：加个MQ进来，怎么保证消息没有重复消费？怎么处理消息丢失的情况？怎么保证消息传递的顺序性？
* 一致性问题：A 系统处理完了直接返回成功了，人都以为你这个请求就成功了；但是问题是，要是 BCD 三个系统那里，BD 两个系统写库成功了，结果 C 系统写库失败了，数据就不一致了。

## 6.13RocketMQ消息的可靠性？

所有消息队列都可能发生的问题：

（1）生产者发送消息后，队列未成功接收（网络原因或其他）而生产者不知情，消息丢失；

（2）生产者发送消息后，队列接收成功->生产者确认，但消息并未持久化，队列崩溃，消息丢失；

1.生产者可靠性保证

* rocketMQ提供了生产者消息发送确认的模式，比如在生产者发送消息后返回SendResult，如果isSuccess返回为true,则表示消息已经确认发送到服务器，并且被服务器接收存储。整个发送过程是一个同步的过程。保证消息送达服务器并返回结果。

2.broker可靠性保证

* meta服务器收到生产者发送的消息后，做必要的校验和检查之后，然后就将其写入磁盘，写入成功之后返回应答给生产者。因此，可以确认每条发送结果为成功的消息服务器都是写入磁盘的。 写入磁盘，不意味着数据落到磁盘设备上，毕竟我们还隔着一层os，os对写有缓冲。
* Meta有以下刷盘策略：异步刷盘（每1000条（可配置）或者每隔10秒、同步刷盘（需要设置存储配置上的groupCommitEnable选项为true，会在写入消息后，立即强制刷盘）。

2.消费者可靠性保证

* 消费者顺序消费，只有在成功消费一条消息后才会接着消费下一条。如果在消费某条消息失败（如异常），则会尝试重试消费这条消息（默认最大5次），超过最大次数后仍然无法消费，则将消息存储在消费者的本地磁盘，由后台线程继续做重试。而主线程继续往后走，消费后续的消息。因此， 只有在MessageListener确认成功消费一条消息后，meta的消费者才会继续消费另一条消息。由此来保证消息的可靠消费。
* 消费者的另一个可靠性的关键点是offset的存储，也就是拉取数据的偏移量。默认存储在zoopkeeper上，zookeeper通过集群来保证数据的安全性。Offset会定期保存，并且在每次重新负载均衡前都会强制保存一次，因此可能会存在极端情况下的消息的重复消费。

## 6.14JWT身份验证

+传统的身份验证：

（1）HTTP Basic Auth：就是每次请求API时都提供用户的username和password，简言之，Basic Auth是配合RESTful API 使用的最简单的认证方式，只需提供用户名密码即可，但是可能把用户名密码暴露给第三方客户端，使用的越来越少。因此，在开发对外开放的RESTful API时，尽量避免采用HTTP Basic Auth。

（2）OAuth（开放授权）：是一个开放的授权标准，允许用户让第三方应用访问该用户在某一web服务上存储的私密的资源（如照片，视频，联系人列表），而无需将用户名和密码提供给第三方应用。OAuth允许用户提供一个令牌，而不是用户名和密码来访问他们存放在特定服务提供者的数据。每一个令牌授权一个特定的第三方系统（例如，视频编辑网站)在特定的时段（例如，接下来的2小时内）内访问特定的资源（例如仅仅是某一相册中的视频）。

（3）Cookie Auth：为一次请求认证在服务端创建一个Session对象，同时在客户端的浏览器端创建了一个Cookie对象；通过客户端带上来Cookie对象来与服务器端的session对象匹配来实现状态管理的。默认的，当我们关闭浏览器的时候，cookie会被删除。但可以通过修改cookie 的expire time使cookie在一定时间内有效。

+基于 Token 的身份验证方法

使用基于 Token 的身份验证方法，大概的流程是这样的：

1）客户端使用用户名跟密码请求登录；

2）服务端收到请求，去验证用户名与密码；

3）验证成功后，服务端会签发一个 Token然后保存（缓存或者数据库），再把这个 Token 发送给客户端；

4）客户端收到 Token 以后可以把它存储起来，比如放在 Cookie 里或者 Local Storage 里；

5）客户端每次向服务端请求资源的时候需要带着服务端签发的 Token；

6）服务端收到请求，然后去验证客户端请求里面带着的 Token，如果验证成功，就向客户端返回请求的数据；

Token机制相对于Cookie机制又有什么好处呢？

1）支持跨域访问：Cookie是不允许垮域访问的，这一点对Token机制是不存在的，前提是传输的用户认证信息通过HTTP头传输.

2）无状态：Token机制在服务端不需要存储session信息，因为Token 自身包含了所有登录用户的信息，只需要在客户端的cookie存储状态信息。

3）更适用CDN： 可以通过内容分发网络请求你服务端的所有资料（如：javascript，HTML,图片等），而你的服务端只要提供API即可.

4）去耦：不需要绑定到一个特定的身份验证方案。Token可以在任何地方生成，只要在你的API被调用的时候，你可以进行Token生成调用即可.

5）基于标准化：API可以采用标准化的 JSON Web Token (JWT)，这个标准已经存在多个后端库（.NET, Ruby, Java,Python, PHP）和多家公司的支持（如：Firebase,Google, Microsoft）.

+基于JWT的Token认证机制实现

JWT 标准的 Token 有三个部分，中间用点分隔开，并且都会使用 Base64 编码：

header（头部）：包含了使用的算法，这个 JWT 是不是带签名的或者加密的。主要就是说明一下怎么处理这个 JWT token 。比如一个加密的 JWT 里面要包含使用的加密的算法。唯一在头部里面要包含的是 alg 这个属性，如果是加密的 JWT，这个属性的值就是使用的签名或者解密用的算法。

payload（数据）：Payload 里面是 Token 的具体内容，这些内容里面有一些是标准字段，你也可以添加其它需要的内容。下面是标准字段：[iss：Issuer，发行者]，[sub：Subject，主题]，[aud：Audience，观众]，[exp：Expiration time，过期时间]，[signature（签名）]，

Signature：包含三个部分，先是用 Base64 编码的 header.payload ，再用加密算法加密一下，加密的时候要放进去一个 Secret ，这个相当于是一个密码，这个密码秘密地存储在服务端。

## 6.15分布式系统

一些常问的点：

* + Dubbo 工作原理：服务注册、注册中心、消费者、代理通信、负载均衡；
  + 网络通信、序列化：dubbo 协议、长连接、NIO、hessian 序列化协议；
  + 负载均衡策略、集群容错策略、动态代理策略：dubbo 跑起来的时候一些功能是如何运转的？怎么做负载均衡？怎么做集群容错？怎么生成动态代理？
  + Dubbo SPI 机制：你了解不了解 dubbo 的 SPI 机制？如何基于 SPI 机制对 dubbo 进行扩展？

### 6.15.1为什么将系统拆分，如何拆分？

（1）系统不拆分的缺点

一个大系统几十万行代码，维护起来非常困难，会造成代码冲突、系统异常、技术不敢随意升级、修改代码后全部重新测试；

（2）系统拆分的优点

将整个系统拆分成不同的服务，每个服务部署到单独的机器上，每个人都维护自己独立的代码，每次就发布自己负责的一个小服务就可以了；不同的服务可以进行技术升级，只要保证接口不变就行；没有代码冲突，开发效率提升。

（3）拆分后不使用Dubbo可以吗？

当然可以了，拆分后的各个系统之间直接基于SpringMVC，利用HTTP接口互相通信。但是这个肯定是有问题的，因为http接口通信维护起来成本很高，要考虑**超时重试**、**负载均衡**等各种问题，比如说订单系统调用商品系统，商品系统部署了5台机器，怎么把请求均匀地分给那5台机器？这不就是负载均衡？

Dubbo是一种RPC框架，就是说本地就是进行接口调用，但是Dubbo会代理这个调用请求，跟远程机器网络通信，处理掉负载均衡、服务实例上下线自动感知、超时重试等等问题。

### 6.15.2分布式CAP理论

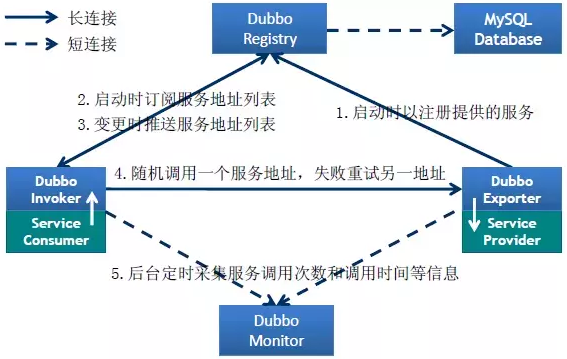
一个分布式系统最多只能同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance）这三项中的两项。CAP理论中的CA和数据库事务中ACID的CA并完全是同一回事儿。两者之中的C都是一致性(Consistency)。CAP中的A指的是可用性（Availability），而ACID中的A指的是原子性（Atomicity)，切勿混为一谈。

* 一致性指更新操作成功并返回客户端完成后，所有节点在同一时间的数据完全一致。
* 可用性指服务一直可用，而且是正常响应时间。对于一个可用性的分布式系统，每一个非故障的节点必须对每一个请求作出响应。所以，一般我们在衡量一个系统的可用性的时候，都是通过停机时间来计算的。
* 分区容错性指分布式系统在遇到某节点或网络分区故障的时候，仍然能够对外提供满足一致性和可用性的服务。分区容错性和扩展性紧密相关。在分布式应用中，可能因为一些分布式的原因导致系统无法正常运转。好的分区容错性要求能够使应用虽然是一个分布式系统，而看上去却好像是在一个可以运转正常的整体。

### 6.15.3Dubbo的工作原理？

推荐使用 Zookeeper 作为注册中心，还有 Redis、Multicast、Simple 注册中心，但不推荐。

（1）Dubbo角色和设计是怎么样的，原理是怎么样的?

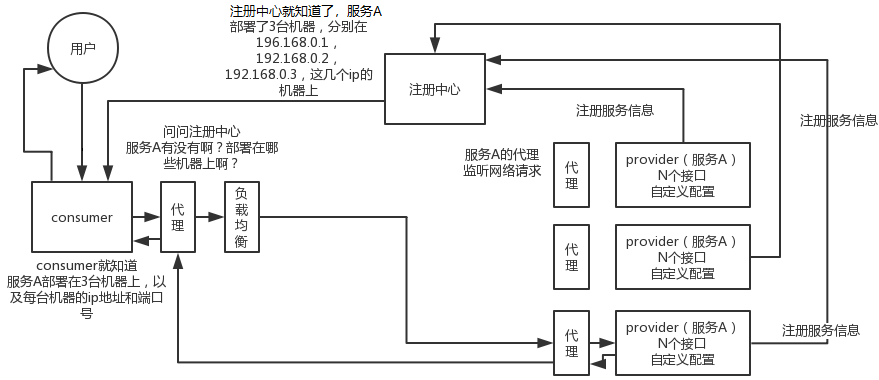


（2）dubbo 工作原理

* 第一层：service 层，接口层，给服务提供者和消费者来实现的
* 第二层：config 层，配置层，主要是对 dubbo 进行各种配置的
* 第三层：proxy 层，服务代理层，无论是 consumer 还是 provider，dubbo 都会给你生成代理，代理之间进行网络通信
* 第四层：registry 层，服务注册层，负责服务的注册与发现
* 第五层：cluster 层，集群层，封装多个服务提供者的路由以及负载均衡，将多个实例组合成一个服务
* 第六层：monitor 层，监控层，对 rpc 接口的调用次数和调用时间进行监控
* 第七层：protocal 层，远程调用层，封装 rpc 调用
* 第八层：exchange 层，信息交换层，封装请求响应模式，同步转异步
* 第九层：transport 层，网络传输层，抽象 mina 和 netty 为统一接口
* 第十层：serialize 层，数据序列化层

（3）工作流程

* 第一步：provider 向注册中心去注册；
* 第二步：consumer 从注册中心订阅服务，注册中心会通知 consumer 已经注册的服务；
* 第三步：consumer 调用 provider；
* 第四步：consumer 和 provider 都异步通知监控中心；



（3）注册中心挂了可以继续通信吗？

答案是可以，因为刚开始初始化的时候，消费者会将提供者的地址等信息拉取到本地缓存，所以注册中心挂了可以继续通信。注册中心对等集群，任意一台宕机后，将会切换到另一台；注册中心全部宕机后，服务的提供者和消费者仍能通过本地缓存通讯。服务提供者无状态，任一台宕机后，不影响使用；服务提供者全部宕机，服务消费者会无法使用，并无限次重连等待服务者恢复；

挂掉是不要紧的，但前提是你没有增加新的服务，如果你要调用新的服务，则是不能办到的。

### 6.15.4Dubbo支持哪些协议？PB？

在一个典型的RPC的使用场景中，包含了服务发现、负载、容错、序列化和网络传输等组件，其中RPC协议指明了程序如何进行序列化和网络传输，也就是说一个RPC协议的实现等于一个非透明的RPC调用。



简单来说，**分布式框架的核心是RPC框架，RPC框架的核心是RPC协议**。

序列化就是把数据结构或者是一些对象，转换为二进制串的过程；

反序列化是将在序列化过程中所生成的二进制串转换成数据结构或者对象的过程。

（1）dubbo协议

默认是dubbo协议，单个长连接，进行的是NIO异步通信，基于hessian作为序列化协议。底层是TCP协议。

使用的场景是：消费者个数多；传输数据量小（每次请求在 100kb 以内），但是并发量很高。

为了要支持高并发场景，一般是服务提供者就几台机器，但是服务消费者有上百台，可能每天调用量达到上亿次！此时用长连接是最合适的，就是跟每个服务消费者维持一个长连接就可以，可能总共就 100 个连接。然后后面直接基于长连接 NIO 异步通信，可以支撑高并发请求。

dubbo 支持的RPC协议列表



为什么 PB 的效率是最高的？

可能有一些同学比较习惯于JSON or XML数据存储格式，对于Protocol Buffer还比较陌生。Protocol Buffer其实是Google 出品的一种轻量并且高效的结构化数据存储格式，性能比JSON、XML要高很多。

其实PB之所以性能如此好，主要得益于两个：

1. 它使用proto编译器，自动进行序列化和反序列化，速度非常快，比XML和JSON快上了 20~100倍；
2. 它的数据压缩效果好，序列化后的数据量体积小。因为体积小，传输起来带宽和速度上会有优化。

### 6.15.5Dubbo在安全机制方面是如何解决的？

Dubbo通过Token令牌防止用户绕过注册中心直连，然后在注册中心上管理授权。Dubbo还提供服务黑白名单，来控制服务所允许的调用方。

### 6.15.6dubbo协议为什么不能传大包？

因dubbo协议采用单一长连接，如果每次请求的数据包大小为500KByte，假设网络为千兆网卡(1024Mbit=128MByte)，每条连接最大7MByte(不同的环境可能不一样，供参考)，单个服务提供者的TPS(每秒处理事务数)最大为：128MByte / 500KByte = 262。单个消费者调用单个服务提供者的TPS(每秒处理事务数)最大为：7MByte / 500KByte = 14。如果能接受，可以考虑使用，否则网络将成为瓶颈。

### 6.15.7dubbo协议为什么采用异步单一长连接？

因为服务的现状大都是服务提供者少，通常只有几台机器，而服务的消费者多，可能整个网站都在访问该服务，比如Morgan的提供者只有6台提供者，却有上百台消费者，每天有1.5亿次调用，如果采用常规的hessian服务，服务提供者很容易就被压跨，通过单一连接，保证单一消费者不会压死提供者，长连接，减少连接握手验证等，并使用异步IO，复用线程池，防止C10K问题。

【C10K问题：

C10K问题的本质上是操作系统的问题。对于Web 1.0/2.0时代的操作系统，传统的同步阻塞I/O模型处理方式都是requests per second。当创建的进程或线程多了，数据拷贝频繁（缓存I/O、内核将数据拷贝到用户进程空间、阻塞），进程/线程上下文切换消耗大， 导致操作系统崩溃，这就是C10K问题的本质。可见，解决C10K问题的关键就是尽可能减少这些CPU资源消耗。

从网络编程技术的角度来说：

* 每个连接分配一个独立的线程/进程；
* 同一个线程/进程同时处理多个连接（Select poll epoll）；

】

### 6.15.8Dubbo的服务发布作了几件事？

* 暴露本地服务
* 暴露远程服务
* 启动netty
* 连接zookeeper
* 到zookeeper注册
* 监听zookeeper

### 6.15.9服务提供者能实现失效踢出是什么原理(高频题)

在分布式系统中，我们常常需要知道某个机器是否可用，传统的开发中，可以通过Ping某个主机来实现，Ping得通说明对方是可用的，相反是不可用的；ZK中我们让所有的机器都注册一个临时节点，我们判断一个机器是否可用，只需要判断这个节点在ZK中是否存在就可以了，不需要直接去连接需要检查的机器，降低系统的复杂度。

### 6.15.10Dubbo负载均衡策略和集群容错策略都有哪些？动态代理策略呢？

（1）dubbo 负载均衡策略

* random loadbalance：默认情况下，dubbo是 random loadbalance ，即随机调用实现负载均衡，可以对 provider 不同实例设置不同的权重，会按照权重来负载均衡，权重越大分配流量越高，一般就用这个默认的就可以了。
* roundrobin loadbalance：就是均匀地将流量打到各个机器上去，但是如果各个机器的性能不一样，容易导致性能差的机器负载过高。所以此时需要调整权重，让性能差的机器承载权重小一些，流量少一些。

举个栗子：跟运维同学申请机器，有的时候，我们运气好，正好公司资源比较充足，刚刚有一批热气腾腾、刚刚做好的虚拟机新鲜出炉，配置都比较高：8 核 + 16G 机器，申请到 2 台。过了一段时间，我们感觉 2 台机器有点不太够，我就去找运维同学说，“哥儿们，你能不能再给我一台机器”，但是这时只剩下一台 4 核 + 8G 的机器。我要还是得要。这个时候，可以给两台 8 核 16G 的机器设置权重 4，给剩余 1 台 4 核 8G 的机器设置权重 2。<dubbo:service interface= “…” loadbalance=”roundrobin” />

* leastactive loadbalance：自动感知，如果某个机器性能越差，那么接收的请求越少，越不活跃，此时就会给不活跃的性能差的机器更少的请求。
* consistanthash loadbalance：一致性 Hash 算法，相同参数的请求一定分发到一个 provider 上去，provider 挂掉的时候，会基于虚拟节点均匀分配剩余的流量，抖动不会太大。如果你需要的不是随机负载均衡，是要一类请求都到一个节点，那就走这个一致性 Hash 策略。

（2）dubbo 集群容错策略

* failover cluster 模式：失败自动切换，自动重试其他机器，默认就是这个，常见于读操作。（失败重试其它机器），可以通过以下几种方式配置重试次数：<dubbo:service retries="2" />
* failfast cluster 模式：一次调用失败就立即失败，常见于非幂等性的写操作，比如新增一条记录（调用失败就立即失败）。
* failsafe cluster 模式：出现异常时忽略掉，常用于不重要的接口调用，比如记录日志。配置示例如下：<dubbo:service cluster="failsafe" />
* failback cluster 模式：失败了后台自动记录请求，然后定时重发，比较适合于写消息队列这种。
* forking cluster 模式：并行调用多个 provider，只要一个成功就立即返回。常用于实时性要求比较高的读操作，但是会浪费更多的服务资源，可通过 forks="2" 来设置最大并行数。
* broadcacst cluster：逐个调用所有的 provider。任何一个 provider 出错则报错（从2.1.0版本开始支持）。通常用于通知所有提供者更新缓存或日志等本地资源信息。

（3）dubbo动态代理策略

默认使用 javassist 动态字节码生成，创建代理类。但是可以通过 spi 扩展机制配置自己的动态代理策略。

### 6.15.10怎么通过dubbo实现服务降级的，降级的方式有哪些，又有什么区别？

当网站处于高峰期时，并发量大，服务能力有限，那么我们只能暂时屏蔽边缘业务，这里面就要采用服务降级策略了。首先dubbo中的服务降级分成两个：屏蔽(mock=force)、容错(mock=fail)。

* mock=force:return+null 表示消费方对该服务的方法调用都直接返回 null 值，不发起远程调用。用来屏蔽不重要服务不可用时对调用方的影响。
* mock=fail:return+null 表示消费方对该服务的方法调用在失败后，再返回 null 值，不抛异常。用来容忍不重要服务不稳定时对调用方的影响。

要生效需要在dubbo后台进行配置的修改。

说下dubbo怎么区分不同的服务？

分组分区！

## 6.16Zookeeper

### 6.16.1zookeeper是什么？为什么使用？优点？

<https://www.zhihu.com/question/65852003> 关于zookeeper的四个功能

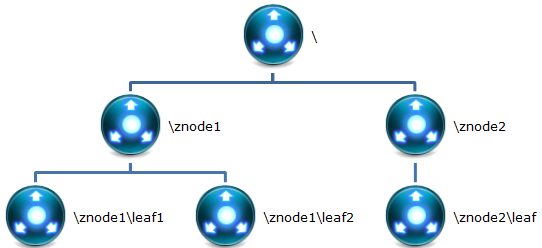
* 定义：Zookeeper 是 Google 的 Chubby一个开源的实现，是 Hadoop 的分布式协调服务；它包含一个简单的原语集，分布式应用程序可以基于它实现同步服务，配置维护和命名服务等；
* 为什么使用？
  + 大部分分布式应用需要一个主控、协调器或控制器来管理物理分布的子进程（如资源、任务分配等）；
  + 目前，大部分应用需要开发私有的协调程序，缺乏一个通用的机制；
  + 协调程序的反复编写浪费，且难以形成通用、伸缩性好的协调器；
  + Keepalived可以提供通用的分布式锁服务，用以协调分布式应用，但是：Keepalived监控节点不好管理；Keepalive 采用优先级监控；没有协同工作；功能单一；Keepalive可扩展性差。
* 优点？

|  |  |
| --- | --- |
| 最终一致性 | 为客户端展示同一个视图，这是非常重要的功能 |
| 可靠性 | 1如果消息被一台服务器接受，那么它将被所有的服务器接受；  2集群中只要有半数以上节点存活，Zookeeper集群就能正常服务 |
| 实时性 | Zookeeper不能保证两个客户端能同时得到刚刚更新的数据，如果需要最新的数据，应该在读数据之前调用sync()接口。 |
| 独立性 | 各个client之间互不干预 |
| 原子性 | 更新只能成功或失败，没有中间状态 |
| 顺序性 | 所有server,，同一消息发布顺序一致 |

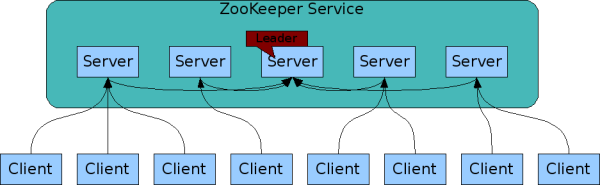
* 数据结构？

ZooKeeper数据模型的结构与Unix文件系统很类似，整体上可看作是**一棵树**，每个节点称作一个**ZNode**。

* 每个 ZNode默认能存储**1MB**数据，每个ZNode都可以**通过其路径唯一标识**。
* 可以使用zkCli命令，登录到zookeeper上，并通过ls、create、delete、sync等命令操作这些znode节点；
* znode除了名称、数据以外，还有一套属性：zxid。这套zid与时间戳对应，记录zid不同的状态；



* 工作原理？
  + 1.每个Server在内存中存储了一份数据；
  + 2.Zookeeper启动时，将从实例中选举一个leader（Paxos协议）；
  + 3.Leader负责处理数据更新等操作；
  + 4.一个更新操作成功，当且仅当大多数Server在内存中成功修改数据。



### 6.16.2判断Zookeeper节点是持久节点还是临时节点？

* zookeeper持久节点：该数据节点被创建后，就会一直存在于zookeeper服务器上，直到有删除操作来主动删除这个节点。
* zookeeper临时节点：临时节点的生命周期和客户端会话绑定在一起，客户端会话失效，则这个节点就会被自动清除。

创建节点的完成之后，使用get命令查看节点的State数据结构信息，其中有一个字段是ephemeralOwner意思是这个节点的临时拥有者：

* 当ephemeralOwner 值不为0时，表明这个节点是临时节点，值为会话id。
* 当ephemeralOwner 值为0时，表明这个节点是持久节点。

可以给节点加一个watcher（监控）监控节点，如果挂掉了可以重新建立会话；也可以使用高级开发包curator，可以创建一个client不主动close，session就能一直保持。而且内部还有重连机制等等。

### 6.16.3zookeeper的选举机制？

所谓的zookeeper容错是指，当宕掉几个zookeeper服务器之后，剩下的个数必须大于宕掉的个数，也就是剩下的服务数必须大于n/2，zookeeper才可以继续使用，无论奇偶数都可以选举leader。5台机器最多宕掉2台，还可以继续使用，因为剩下3台大于5/2。

说为什么最好为奇数个，是在以最大容错服务器个数的条件下，会节省资源，比如，最大容错为2的情况下，对应的zookeeper服务数，奇数为5，而偶数为6，也就是6个zookeeper服务的情况下最多能宕掉2个服务，所以从节约资源的角度看，没必要部署6（偶数）个zookeeper服务。

A 选举条件

Leader选举是保证分布式数据一致性的关键所在。当Zookeeper集群中的一台服务器出现以下两种情况之一时，需要进入Leader选举。

1、服务器初始化启动。

若进行Leader选举，则至少需要两台机器，这里选取3台机器组成的服务器集群为例。在集群初始化阶段，当有一台服务器Server1启动时，其单独无法进行和完成Leader选举，当第二台服务器Server2启动时，此时两台机器可以相互通信，每台机器都试图找到Leader，于是进入Leader选举过程。选举过程如下：

* **1）每个Server发出一个投票**。由于是初始情况，Server1和Server2都会将自己作为Leader服务器来进行投票，每次投票会包含所推举的服务器的myid和ZXID，使用(myid, ZXID)来表示，此时Server1的投票为(1, 0)，Server2的投票为(2, 0)，然后各自将这个投票发给集群中其他机器。
* **2）接受来自各个服务器的投票。**集群的每个服务器收到投票后，首先判断该投票的有效性，如检查是否是本轮投票、是否来自LOOKING状态的服务器。
* **3）处理投票。**针对每一个投票，服务器都需要将别人的投票和自己的投票进行PK，PK规则如下：
* 优先检查ZXID。ZXID比较大的服务器优先作为Leader。
* 如果ZXID相同，那么就比较myid。myid较大的服务器作为Leader服务器。

对于Server1而言，它的投票是(1, 0)，接收Server2的投票为(2, 0)，首先会比较两者的ZXID，均为0，再比较myid，此时Server2的myid最大，于是更新自己的投票为(2, 0)，然后重新投票，对于Server2而言，其无须更新自己的投票，只是再次向集群中所有机器发出上一次投票信息即可。

* **4）统计投票。**每次投票后，服务器都会统计投票信息，判断是否已经有过半机器接受到相同的投票信息，对于Server1、Server2而言，都统计出集群中已经有两台机器接受了(2, 0)的投票信息，此时便认为已经选出了Leader。
* **5）改变服务器状态。**一旦确定了Leader，每个服务器就会更新自己的状态，如果是Follower，那么就变更为FOLLOWING，如果是Leader，就变更为LEADING。

2、服务器运行期间无法和Leader保持连接。

在Zookeeper运行期间，Leader与非Leader服务器各司其职，即便当有非Leader服务器宕机或新加入，此时也不会影响Leader，但是一旦Leader服务器挂了，那么整个集群将暂停对外服务，进入新一轮Leader选举，其过程和启动时期的Leader选举过程基本一致。假设正在运行的有Server1、Server2、Server3三台服务器，当前Leader是Server2，若某一时刻Leader挂了，此时便开始Leader选举。选举过程如下：

* 1）变更状态。Leader挂后，余下的非Observer服务器都会讲自己的服务器状态变更为LOOKING，然后开始进入Leader选举过程。
* 2）每个Server会发出一个投票。在运行期间，每个服务器上的ZXID可能不同，此时假定Server1的ZXID为123，Server3的ZXID为122；在第一轮投票中，Server1和Server3都会投自己，产生投票(1, 123)，(3, 122)，然后各自将投票发送给集群中所有机器。
* 3）接收来自各个服务器的投票。与启动时过程相同。
* 4）处理投票。与启动时过程相同，此时，Server1将会成为Leader。
* 5）统计投票。与启动时过程相同。
* 6）改变服务器的状态。与启动时过程相同。

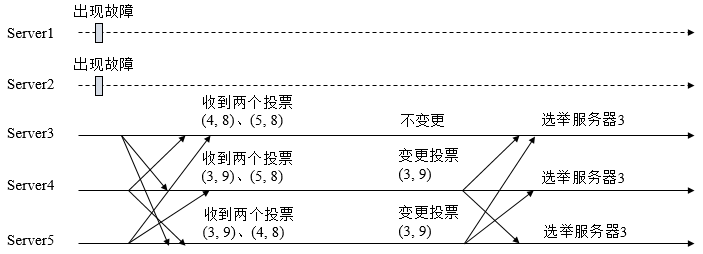
B Leader选举算法分析（FastLeaderElection选举算法）

当一台机器进入Leader选举时，当前集群可能会处于以下两种状态：

* 集群中已经存在Leader。
* 此种情况一般都是某台机器启动得较晚，在其启动之前，集群已经在正常工作，对这种情况，该机器试图去选举Leader时，会被告知当前服务器的Leader信息，对于该机器而言，仅仅需要和Leader机器建立起连接，并进行状态同步即可。
* 集群中不存在Leader。
* **（1）第一次投票**。无论哪种导致进行Leader选举，集群的所有机器都处于试图选举出一个Leader的状态，即LOOKING状态，LOOKING机器会向所有其他机器发送消息，该消息称为投票。投票中包含了SID（服务器的唯一标识）和ZXID（事务ID），(SID, ZXID)形式来标识一次投票信息。假定Zookeeper由5台机器组成，SID分别为1、2、3、4、5，ZXID分别为9、9、9、8、8，并且此时SID为2的机器是Leader机器，某一时刻，1、2所在机器出现故障，因此集群开始进行Leader选举。在第一次投票时，每台机器都会将自己作为投票对象，于是SID为3、4、5的机器投票情况分别为(3, 9)，(4, 8)， (5, 8)。
* **（2）变更投票**。每台机器发出投票后，也会收到其他机器的投票，每台机器会根据一定规则来处理收到的其他机器的投票，并以此来决定是否需要变更自己的投票，这个规则也是整个Leader选举算法的核心所在，其中术语描述如下：
* vote\_sid：接收到的投票中所推举Leader服务器的SID。
* vote\_zxid：接收到的投票中所推举Leader服务器的ZXID。
* self\_sid：当前服务器自己的SID。
* self\_zxid：当前服务器自己的ZXID。

每次对收到的投票的处理，都是对(vote\_sid, vote\_zxid)和(self\_sid, self\_zxid)对比的过程：

* 规则一：如果vote\_zxid大于self\_zxid，就认可当前收到的投票，并再次将该投票发送出去。
* 规则二：如果vote\_zxid小于self\_zxid，那么坚持自己的投票，不做任何变更。
* 规则三：如果vote\_zxid等于self\_zxid，那么就对比两者的SID，如果vote\_sid大于self\_sid，那么就认可当前收到的投票，并再次将该投票发送出去。
* 规则四：如果vote\_zxid等于self\_zxid，并且vote\_sid小于self\_sid，那么坚持自己的投票，不做任何变更。



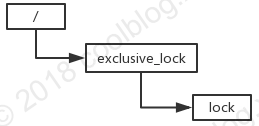
* **（3）确定Leader**。经过第二轮投票后，集群中的每台机器都会再次接收到其他机器的投票，然后开始统计投票，如果一台机器收到了超过半数的相同投票，那么这个投票对应的SID机器即为Leader。此时Server3将成为Leader。

由上面规则可知，通常那台服务器上的数据越新（ZXID会越大），其成为Leader的可能性越大，也就越能够保证数据的恢复。如果ZXID相同，则SID越大机会越大。

### 6.16.4Zookeeper实现分布式锁？

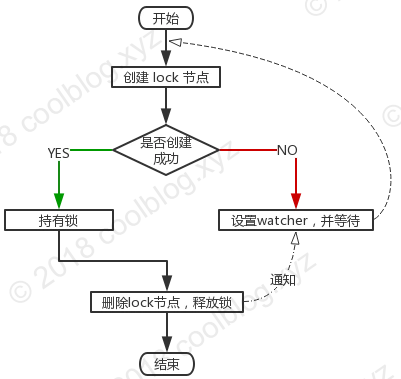
（1）独占锁

独占锁又称排它锁，从字面意思上很容易理解他们的用途。即如果某个操作 O1 对访问资源 R1 的过程加锁，在操作 O1 结束对资源 R1 访问前，其他操作不允许访问资源 R1。



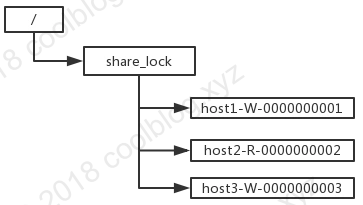
如上图，对于独占锁，可以将资源 R1 看做是 lock 节点，操作 O1 访问资源 R1 看做创建 lock 节点，释放资源 R1 看做删除 lock 节点。这样我们就将独占锁的定义对应于具体的 Zookeeper 节点结构，通过创建 lock 节点获取锁，删除节点释放锁。详细的过程如下：

* 多个客户端竞争创建 lock 临时节点
* 其中某个客户端成功创建 lock 节点，其他客户端对 lock 节点设置 watcher
* 持有锁的客户端删除 lock 节点或该客户端崩溃，由 Zookeeper 删除 lock 节点
* 其他客户端获得 lock 节点被删除的通知
* 重复上述4个步骤，直至无客户端在等待获取锁了



（2）读写锁

读写锁包含一个读锁和写锁，操作 O1 对资源 R1 加读锁，且获得了锁，其他操作可同时对资源 R1 设置读锁，进行共享读操作。如果操作 O1 对资源 R1 加写锁，且获得了锁，其他操作再对资源 R1 设置不同类型的锁都会被阻塞。总结来说，读锁具有共享性，而写锁具有排他性。

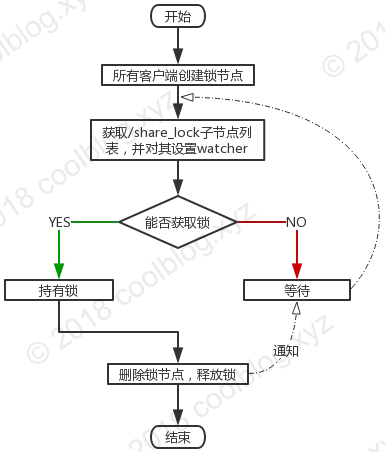


在 Zookeeper 中，由于读写锁和独占锁的节点结构不同，读写锁的客户端不用再去竞争创建 lock 节点。所以在一开始，所有的客户端都会创建自己的锁节点。如果不出意外，所有的锁节点都能被创建成功，此时锁节点结构如上图所示。之后，客户端从 Zookeeper 端获取 /share\_lock 下所有的子节点，并判断自己能否获取锁。

* 如果客户端创建的是读锁节点，获取锁的条件（满足其中一个即可）如下：
  + 自己创建的节点序号排在所有其他子节点前面
  + 自己创建的节点前面无写锁节点
* 如果客户端创建的是写锁节点，由于写锁具有排他性。所以获取锁的条件要简单一些，只需确定自己创建的锁节点是否排在其他子节点前面即可。
* 读写锁第一种实现

第一种实现是对 /share\_lock 节点设置 watcher，当 /share\_lock 下的子节点被删除时，未获取锁的客户端收到 /share\_lock 子节点变动的通知。在收到通知后，客户端重新判断自己创建的子节点是否可以获取锁，如果失败，再次等待通知。详细流程如下：

* 所有客户端创建自己的锁节点
* 从 Zookeeper 端获取 /share\_lock 下所有的子节点，并对 /share\_lock 节点设置 watcher
* 判断自己创建的锁节点是否可以获取锁，如果可以，持有锁。否则继续等待
* 持有锁的客户端删除自己的锁节点，其他客户端收到 /share\_lock 子节点变动的通知
* 重复步骤2、3、4，直至无客户端在等待获取锁了

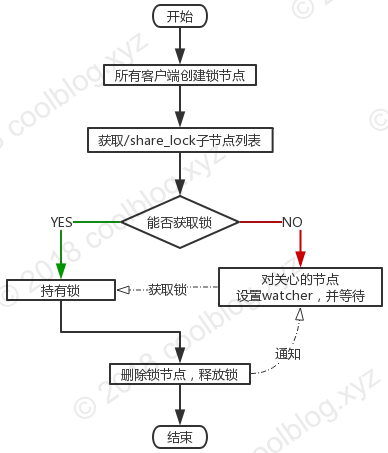


上面获取读写锁流程并不复杂，但却存在性能问题。以图3所示锁节点结构为例，第一个锁节点 host1-W-0000000001 被移除后，Zookeeper 会将 /share\_lock 子节点变动的通知分发给所有的客户端。但实际上，该子节点变动通知除了能影响 host2-R-0000000002 节点对应的客户端外，分发给其他客户端则是在做无用功，因为其他客户端即使获取了通知也无法获取锁。所以这里需要做一些优化，优化措施是让客户端只在自己关心的节点被删除时，再去获取锁。

* 读写锁的第二种实现

在了解读写锁第一种实现的弊端后，我们针对这一实现进行优化。这里客户端不再对 /share\_lock 节点进行监视，而只对自己关心的节点进行监视。还是以图3的锁节点结构进行举例说明，host2-R-0000000002 对应的客户端 C2 只需监视 host1-W-0000000001 节点是否被删除即可。而 host3-W-0000000003 对应的客户端 C3 只需监视 host2-R-0000000002 节点是否被删除即可，只有 host2-R-0000000002 节点被删除，客户端 C3 才能获取锁。而 host1-W-0000000001 节点被删除时，产生的通知对于客户端 C3 来说是无用的，即使客户端 C3 响应了通知也没法获取锁。这里总结一下，不同客户端关心的锁节点是不同的。如果客户端创建的是读锁节点，那么客户端只需找出比读锁节点序号小的最后一个的写锁节点，并设置 watcher 即可。而如果是写锁节点，则更简单，客户端仅需对该节点的上一个节点设置 watcher 即可。详细的流程如下：

* 所有客户端创建自己的锁节点
* 从 Zookeeper 端获取 /share\_lock 下所有的子节点
* 判断自己创建的锁节点是否可以获取锁，如果可以，持有锁。否则对自己关心的锁节点设置 watcher
* 持有锁的客户端删除自己的锁节点，某个客户端收到该节点被删除的通知，并获取锁
* 重复步骤4，直至无客户端在等待获取锁了



（4）zookeeper分布式锁

第一种方式

zk 分布式锁，其实可以做的比较简单，就是某个节点尝试创建临时 znode，此时创建成功了就获取了这个锁；这个时候别的客户端来创建锁会失败，只能**注册个监听器**监听这个锁。释放锁就是删除这个 znode，一旦释放掉就会通知客户端，然后有一个等待着的客户端就可以再次重新加锁。

第二种方式：创建临时顺序节点：

如果有一把锁，被多个人给竞争，此时多个人会排队，第一个拿到锁的人会执行，然后释放锁；后面的每个人都会去监听排在自己前面的那个人创建的 node 上，一旦某个人释放了锁，排在自己后面的人就会被 zookeeper 给通知，一旦被通知了之后，就 ok 了，自己就获取到了锁，就可以执行代码了。

（5）redis 分布式锁和zk分布式锁的对比

* redis 分布式锁，其实需要自己不断去尝试获取锁，比较消耗性能。
* zk 分布式锁，获取不到锁，注册个监听器即可，不需要不断主动尝试获取锁，性能开销较小。
* 如果是 redis 获取锁的那个客户端 出现 bug 挂了，那么只能等待超时时间之后才能释放锁；而 zk 的话，因为创建的是临时 znode，只要客户端挂了，znode 就没了，此时就自动释放锁。

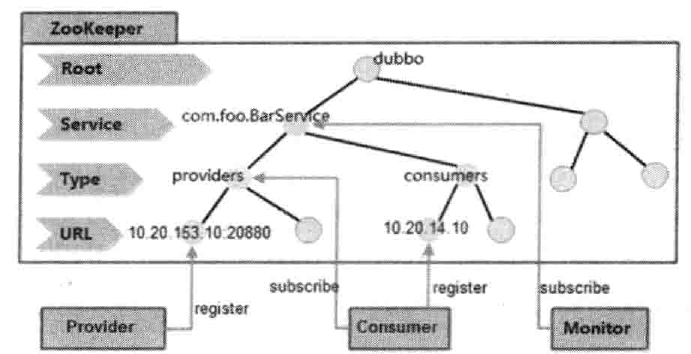
### 6.16.5Zookeeper实现服务注册中心？

1 ZooKeeper中的节点

ZooKeeper是一个树形结构的目录服务，支持变更推送，因此非常适合作为Dubbo服务的注册中心。

|  |
| --- |
| * 在ZooKeeper中，节点分为两类，第一类是指构成集群的机器，称为机器节点；第二类是指数据模型中的数据单元，称为数据节点ZNode。ZooKeeper将所有数据存储在内存中，数据模型是一棵树(ZNode Tree)，由斜杠（/）进行分割的路径，就是一个ZNode。每个ZNode上都会保存自己的数据内容，同时还会保存一系列属性信息。 * 在ZooKeeper中，Znode可分为持久节点和临时节点两类，所谓持久节点是指一旦这个ZNode被创建了，除非主动进行ZNode的移除操作，否则这个ZNode将一直保存在ZooKeeper上。而临时节点就不一样了，它的生命周期和客户端会话绑定，一旦客户端会话失效，那么这个客户端创建的所有临时节点都会被移除。 |

2基于ZooKeeper实现的注册中心节点结构示意图：



* /dubbo：这是dubbo在ZooKeeper上创建的根节点；
* /dubbo/com.foo.BarService：这是服务节点，代表了Dubbo的一个服务；
* /dubbo/com.foo.BarService/providers：服务提供者的根节点，其子节点代表了每一个服务真正的提供者；
* /dubbo/com.foo.BarService/consumers：服务消费者的根节点，其子节点代表每一个服务真正的消费者；

3 注册中心的工作流程

* 服务提供方启动
* 服务提供者在启动的时候，会在ZooKeeper上注册服务。所谓注册服务，其实就是在ZooKeeper的/dubbo/com.foo.BarService/providers节点下创建一个子节点，并写入自己的URL地址，这就代表了com.foo.BarService这个服务的一个提供者。
* 服务消费者启动
* 服务消费者在启动的时候，会向ZooKeeper注册中心订阅自己的服务。其实，就是读取并订阅ZooKeeper上/dubbo/com.foo.BarService/providers节点下的所有子节点，并解析出所有提供者的URL地址来作为该服务地址列表。
* 同时，服务消费者还会在ZooKeeper的/dubbo/com.foo.BarService/consumers节点下创建一个临时节点，并写入自己的URL地址，这就代表了com.foo.BarService这个服务的一个消费者。
* 消费者远程调用提供者
* 服务消费者，从提供者地址列表中，基于软负载均衡算法，选一个提供者进行调用，如果调用失败，再选另一个提供者调用。
* 增加服务提供者
* 增加提供者，也就是在providers下面新建子节点。一旦服务提供方有变动，zookeeper就会把最新的服务列表推送给消费者。
* 减少服务提供者
* **所有提供者在ZooKeeper上创建的节点都是临时节点**，利用的是临时节点的生命周期和客户端会话相关的特性，因此一旦提供者所在的机器出现故障导致该提供者无法对外提供服务时，该临时节点就会自动从ZooKeeper上删除，同样，zookeeper会把最新的服务列表推送给消费者。
* ZooKeeper宕机之后
* 消费者每次调用服务提供方是不经过ZooKeeper的，消费者只是从zookeeper那里获取服务提供方地址列表。所以当zookeeper宕机之后，不会影响消费者调用服务提供者，影响的是zookeeper宕机之后如果提供者有变动，增加或者减少，无法把最新的服务提供者地址列表推送给消费者，所以消费者感知不到。

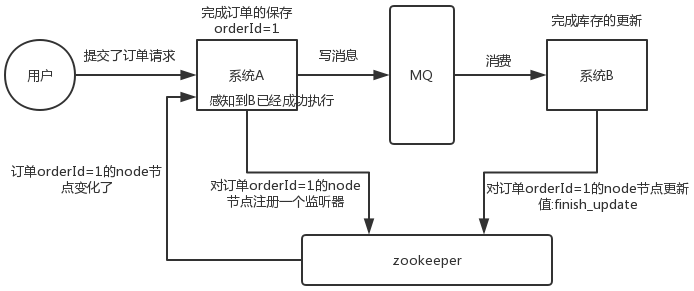
### 6.16.6zookeeper都有哪些使用场景？

zookeeper 是分布式系统中很常见的一个基础系统，zookeeper 的使用场景如下：

* 分布式协调；
* 分布式锁；
* 元数据/配置信息管理；
* HA高可用性；

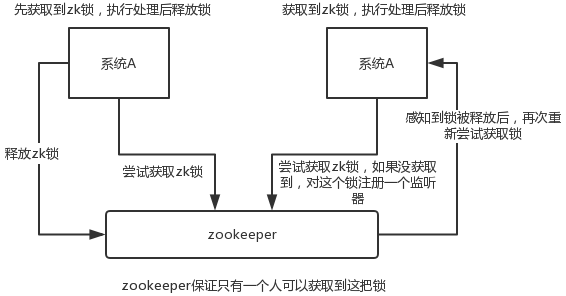
（1）分布式协调

这个其实是 zookeeper很经典的一个用法，简单来说，就好比，你 A 系统发送个请求到 mq，然后 B 系统消息消费之后处理了。那A系统如何知道 B 系统的处理结果？用 zookeeper 就可以实现分布式系统之间的协调工作。A系统发送请求之后可以在zookeeper上**对某个节点的值注册个监听器**，一旦B系统处理完了就修改zookeeper那个节点的值，A 系统立马就可以收到通知，完美解决。



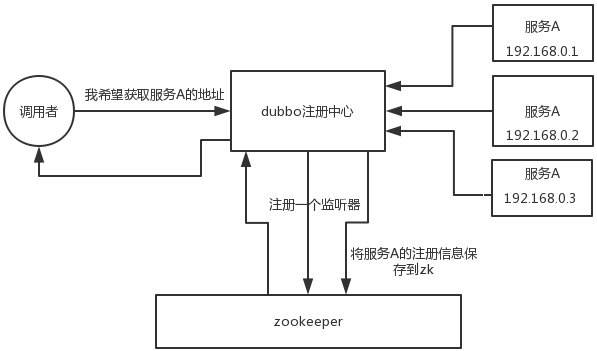
（2）分布式锁（见7.9 的分布式锁实现方式）【项目中使用的】

对某一个数据连续发出两个修改操作，两台机器同时收到了请求，但是只能一台机器先执行完另外一个机器再执行。那么此时就可以使用 zookeeper 分布式锁，一个机器接收到了请求之后先获取 zookeeper 上的一把分布式锁，就是可以去创建一个 znode，接着执行操作；然后另外一个机器也**尝试去创建**那个 znode，结果发现自己创建不了，因为被别人创建了，那只能等着，等第一个机器执行完了自己再执行。



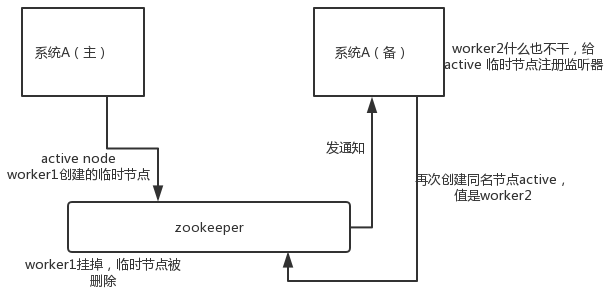
（3）元数据/配置信息管理

zookeeper 可以用作很多系统的配置信息的管理，比如 kafka、storm 等等很多分布式系统都会选用 zookeeper 来做一些元数据、配置信息的管理，包括 **dubbo注册中心不也支持zookeeper** 么？



（4）高可用性（HA high availability）

这个应该是很常见的，比如 hadoop、hdfs、yarn 等很多大数据系统，都选择基于 zookeeper 来开发 HA 高可用机制，就是一个重要进程一般会做主备两个，主进程挂了立马通过 zookeeper 感知到切换到备用进程。



## 6.17电商网站50W-100W高并发，秒杀功能是怎么实现的？

https://www.cnblogs.com/song-wentao/p/11505449.html

## 6.18Nginx

### 6.18.1 Nginx的简介

<https://www.cnblogs.com/gucb/p/11237776.html>

### Nginx和Dubbo的区别

dubbo的负载均衡是服务层面，nginx的负载均衡还在http请求层面。

dubbo具备了server注册，发现、路由、负载均衡的功能。

dubbo在服务发现这个地方做的更像一个dns（个人感觉），一个消费者需要知道哪里有这么一个服务，dubbo告诉他，然后他自己去调用。

而nginx在具备了以上功能，还有两个最主要的功能是：

1，维持尽可能多的连接。

2，把每个连接的具体服务需求pass到真正的worker上。

nginx是横在用户的浏览器和自家的服务器之间，通常用于去管理一些资源，即把不同种类的资源分别放到不同的服务器上面，用nginx去加以管理。

dubbo是横在自家的服务器和自家的服务器之间，是业务的桥梁，是把不同的服务分别的放到不同的服务器上，即分布式开发，然后对其注册过的服务进行统一整合管理。

原文链接：https://blog.csdn.net/qq\_35001776/article/details/78498430

# 7编程及算法

## 7.1HashMap排序题

已知一个 HashMap<Integer，User>集合，User有name（String）和age（int）属性。请写一个方法实现对HashMap 的排序功能，该方法接收HashMap<Integer，User>为形参，返回类型为HashMap<Integer，User>，要求对 HashMap 中的User的age倒序进行排序。排序时key=value键值对不得拆散。

|  |
| --- |
| **public class** FunClass {  **public static void** main(String[] args) {  HashMap<Integer, User> users = **new** HashMap<>();  users.put(1, **new** User(**"张三"**, 25));  users.put(3, **new** User(**"李四"**, 22));  users.put(2, **new** User(**"王五"**, 28));  System.***out***.println(users);  HashMap<Integer,User> sortHashMap = *sortHashMap*(users);  System.***out***.println(sortHashMap);  **/\*\*  控制台输出内容  {1=User [name=张三, age=25], 2=User [name=王五, age=28], 3=User [name=李四, age=22]}  {2=User [name=王五, age=28], 1=User [name=张三, age=25], 3=User [name=李四, age=22]}  \*/** }  **public static** HashMap<Integer, User> sortHashMap(HashMap<Integer, User> map) {  **// 首先拿到 map 的键值对集合** Set<Map.Entry<Integer, User>> entrySet = map.entrySet();  **// 将 set 集合转为 List 集合，为什么，为了使用工具类的排序方法** List<Map.Entry<Integer, User>> list = **new** ArrayList<>(entrySet);  **// 使用 Collections 集合工具类对 list 进行排序，排序规则使用匿名内部类来实现** Collections.*sort*(list, **new** Comparator<Map.Entry<Integer, User>>(){  @Override  **public int** compare(Map.Entry<Integer, User> o1, Map.Entry<Integer, User> o2) {  **return** o2.getValue().getAge() - o1.getValue().getAge();  }  });  **//创建一个新的有序的 HashMap 子类的集合** LinkedHashMap <Integer, User> linkedHashMap = **new** LinkedHashMap<>();  **//将 List 中的数据存储在 LinkedHashMap 中  for** (Map.Entry<Integer, User> entry : list) {  linkedHashMap.put(entry.getKey(), entry.getValue());  }  **return** linkedHashMap;  } } |

## 7.2递归输出文件目录中以.txt结尾的文件

|  |
| --- |
| **public class** FindFiles {  **public static void** main(String[] args) {  List<File> list = **new** ArrayList<>();  String strPath = **"E:/aaa"**;  List<File> fileList = *getFileList*(strPath, list);  System.***out***.println(fileList.toString());  **//结果就是：[E:\aaa\1\2\a.txt, E:\aaa\1\b.txt, E:\aaa\1\c.txt, E:\aaa\d.txt]** }  **/\*\*  \* getFileList  \* @param *strPath* :文件夹路径  \* @param *list* : 结果集合  \* @return java.util.List**<**java.io.File**>  **\*/  public static** List<File> getFileList(String strPath,List<File> list) {  File dir = **new** File(strPath);**//由strPath创建File对象** File[] files = dir.listFiles();**//获取所有的目录数组files  if** (files != **null**) {  **for** (**int** i = 0; i < files.**length**; i++) {  **if** (files[i].isDirectory()) {**//判断file是不是文件夹** *getFileList*(files[i].getAbsolutePath(),list);  }**else**{  **if** (files[i].getName().endsWith(**".txt"**)) {**//判断文件是不是以.txt结尾** list.add(files[i]);  }  }  }  }  **return** list;  } } |

## 7.3缓存算法实现（手写LRU）？

LRU算法的设计原则是：如果一个数据在最近一段时间没有被访问到，那么在将来它被访问的可能性也很小。也就是说，当限定的空间已存满数据时，应当把最久没有被访问到的数据淘汰。

它支持两个操作：

* get(key)：如果key在cache中，则返回对应的value值，否则返回-1；
* set(key,value)：如果key不在cache中，则将该(key,value)插入cache中（注意，如果cache已满，则必须把最近最久未使用的元素从cache中删除）；如果key在cache中，则重置value的值。

|  |
| --- |
| 实例LeetCode上的146题（<https://leetcode.com/problems/lru-cache/>）：  LRUCache cache = new LRUCache( 2);//缓存容量为2  cache.put(1, 1);  cache.put(2, 2);  cache.get(1); // 返回 1  cache.put(3, 3); // 该操作会使得密钥 2 作废  cache.get(2); // 返回 -1 (未找到)  cache.put(4, 4); // 该操作会使得密钥 1 作废  cache.get(1); // 返回 -1 (未找到)  cache.get(3); // 返回 3  cache.get(4); // 返回 4 |

实现方式：利用链表和hashmap。

当需要插入新的数据项的时候，如果新数据项在链表中存在（一般称为命中），则把该节点移到链表头部，如果不存在，则新建一个节点，放到链表头部，若缓存满了，则把链表最后一个节点删除即可。在访问数据的时候，如果数据项在链表中存在，则把该节点移到链表头部，否则返回-1。这样一来在链表尾部的节点就是最近最久未访问的数据项。



|  |
| --- |
| /\*\*首先定义一个Node节点\*/  public class Node {  int key;  int val;  Node next;  Node prev;  public Node(int key, int value) {  this.key = key;  this.val = value;  next = null;  prev = null;  }  }  /\*\*具体的实现\*/  public class LRUCache {  final Node head = new Node(0, 0);  final Node tail = new Node(0, 0);  final Map<Integer, Node> map;  final int capacity;  public LRUCache(int capacity) {//初始化cache  this.capacity = capacity;  map = new HashMap<>(capacity);  head.next = tail;  tail.prev = head;  }  public int get(int key) {  int res = -1;  if(map.containsKey(key)){//map中包含查找的key  Node n = map.get(key);//取出key对应的Node  remove(n);//移除Node  insertToHead(n);//将查到的Node加入到链表的首部  res = n.val;//返回查到的key对应的Node中的值  }  return res;  }  public void put(int key, int value) {//插入元素  if(map.containsKey(key)){  Node n = map.get(key);  remove(n);  n.val = value;  insertToHead(n);  } else {  if(map.size() == capacity){//当前cache容量达到定义的值  map.remove(tail.prev.key);//移除链表尾部的map值  remove(tail.prev);//移除移除链表尾部的Node  }  Node n = new Node(key, value);  insertToHead(n);  map.put(key, n);  }  }  private void remove(Node n){//移除 Node n 的方法  n.prev.next = n.next;  n.next.prev = n.prev;  }  private void insertToHead(Node n){//将节点插入链表的首部  Node headNext = head.next;  head.next = n;  headNext.prev = n;  n.prev = head;  n.next = headNext;  }  } |

也可以使用JDK自带的数据结构实现：

|  |
| --- |
| public class LRUCache\_JDK<K,V> extends LinkedHashMap<K,V> {  private int CACHE\_SIZE;  public LRUCache\_JDK(int cacheSize){  super((int) Math.ceil(cacheSize / 0.75) + 1, 0.75f, true);  CACHE\_SIZE = cacheSize;  }  @Override  protected boolean removeEldestEntry(Map.Entry<K, V> eldest) {  return size() > CACHE\_SIZE;  }  } |

## 7.4在100G文件中找出出现次数最多的100个IP?

将100G的大文件对100取模，得到了100个小文件，如何仍有文件比较大，内存装不下，则将该小文件再取模，争取能装下，对每个小文件，统计每个文件中出现的词以及相应的频率（可以采用trie树/hash\_map等），并取出出现频率最大的100个IP（可以用含100个结点的最小堆），并把100词及相应的频率存入文件，这样又得到了100\*(分单位文件数的文件)个文件。下一步就是把这5000个文件进行归并（类似与归并排序）的过程了。

https://blog.csdn.net/v\_july\_v/article/details/6685962

## 7.5 LRU页面置换算法

地址映射过程中，若在页面中发现所要访问的页面不在内存中，则产生缺页中断。当发生缺页中断时，如果操作系统内存中没有空闲页面，则操作系统必须在内存选择一个页面将其移出内存，以便为即将调入的页面让出空间。而用来选择淘汰哪一页的规则叫做页面置换算法。

https://blog.csdn.net/huyang0304/article/details/82694526

## 7.6八大排序算法

### 7.6.1排序算法术语及复杂度

* 稳定：如果a原本在b前面，而a=b，排序之后a仍然在b的前面；
* 不稳定：如果a原本在b的前面，而a=b，排序之后a可能会出现在b的后面；
* 内排序：所有排序操作都在内存中完成；
* 外排序：由于数据太大，因此把数据放在磁盘中，而排序通过磁盘和内存的数据传输才能进行；
* 时间复杂度： 一个算法执行所耗费的时间。
* 空间复杂度：运行完一个程序所需内存的大小。



n：数据规模

k：桶的个数

In-place：占用常数内存，不占用额外内存

Out-place：占用额外内存

### 7.6.2比较和非比较的区别

* 比较排序：在排序的最终结果里，元素之间的次序依赖于它们之间的比较。每个数都必须和其他数进行比较，才能确定自己的位置。
* 比较排序的优势是，适用于各种规模的数据，也不在乎数据的分布，都能进行排序。可以说，比较排序适用于一切需要排序的情况。
* 非比较排序是通过确定每个元素之前，应该有多少个元素来排序。针对数组arr，计算arr[i]之前有多少个元素，则唯一确定了arr[i]在排序后数组中的位置。计数排序、基数排序、桶排序则属于非比较排序。
* 非比较排序只要确定每个元素之前的已有的元素个数即可，所以一次遍历即可解决。算法时间复杂度O(n)。
* 非比较排序时间复杂度低，但由于非比较排序需要占用空间来确定唯一位置。所以对数据规模和数据分布有一定的要求。

先知道算法的思想，然后再写代码，比较容易理解基记忆。

### 7.6.2冒泡算法

冒泡排序的思想 稳定 时间复杂度为 o(n2)

依次比较相邻两元素，若前一元素大于后一元素则交换之，直至最后一个元素即为最大；然后重新从首元素开始重复同样的操作，

直至倒数第二个元素即为次大元素；依次类推。如同水中的气泡，依次将最大或最小元素气泡浮出水面

private void sort (int [] a) {

for (int i = 0; i < a.length; i++) {

for (int j = 0 ; j < a.length - i ; j++) {

if (a[j] > a[j + 1 ]) {

// 交换元素

int temp = a[j + 1];

a[j + 1] = a[j];

a[j] = temp;

}

}

}

}

### 7.6.3选择排序

选择排序的思想 不稳定 时间复杂度为 o(n2)

首先选择第一个元素跟之后的多个元素进行比较得到最小或者最大的元素，然后依次类推，将数组中二，到第N - 1个元素与后面的元素进行比较，然后得到了相应的结果

public void select (int [] a) {

for (int i = 0; i < a.length; i++) {

for (int j = i; j < a.length; j++) {

if (a[j] > a[j + 1]) {

// 交换，把最小的放到前面

}

}

}

}

不稳定性的体现 ：举个例子，序列5 8 5 2 9，我们知道第一遍选择第1个元素5会和2交换，那么原序列中2个5的相对前后顺序就被破坏了，所以选择排序不是一个稳定的排序算法。

### 7.6.4插入排序

// 插入排序的思想 稳定 时间复杂度为 o(n2)

// 首先从第二个元素开始，第一个元素假定是有序的，然后将得到的元素与第一个元素进行比较，若大于前面的值就将

// 前面的值后移到一位，接着循环，来比较已排序好前面的值和要插入的值的大小，一直找到不符合 a[j] > temp跳出了循环

// 然后，就找到了temp的正确插入位置。

### 7.6.5希尔排序

// 希尔排序的思想

// 希尔排序是对插入排序的改进，插入排序的时候首先是将要插入的数据一个一个进行比较，然后进行交换，得到要插入的位置

// 而希尔排序首先设置一个gap,通常是长度的一半，然后按照长度除以2的情况进行递减，一直到gap = 1 ,就是普通的插入排序方法

// 这种能够减少插入排序过程中，数据的比较次数。

### 7.6.6快速排序

// 快速排序的思想

// 选择数组的第一个元素作为基准元素，然后分别从前后进行遍历，

// 将元素进行比较，找到大于基准的元素和小于基准的元素，将其进行交换，到达最后的时候，将

//然后将第一个基准的元素，与a[i] 进行交换，此时，将数据以基准元素分成了两部分，然后再对

// 这两部分，进行了进行排序。进行递归，将所有排好序即可

### 7.6.7堆排序

// 堆排序的思想

// 首先给你一个初始的数组，对这个数组构造大根堆

，然后得到的大根堆，则数组的第一个元素跟最后

// 一个元素进行比较，然后对n - 1 个元素构造大根堆，依次类推，直到剩余一个元素，则得到的就是

// 升序排列的数组。

### 7.6.8归并排序

// 归并排序思想

// 首先归并的前提是要先把数组进行二分，一直分到每个单元只有一个元素，但是写程序的时候，不需要

// 一直分，只需要求出middle,就相当于得到了二分的数组了，这种利用二分查找的思想，然后将两个子数组，

// 进行排序，则以此逆着向上就得到了进行排序就得到了要排序的数组了

### 7.6.9基数排序

// 基数排序的思想

// 首先找到最大的数，然后得到这个最大的数有多少位，再创建十个桶，用来存储从0 - 9 的数字

// 其是用ArrayList实现的，然后，根据遍历最大数位数的次数，计算每个数相应为的数，找到第几个桶，然后将该位的数放入到桶中

// 下面将重新排列得到的数组，注意写新的数组的时候，要将老的ArrayList中的数据删除掉。