

# 写在前面的话

---

个人整理的一些求职面试资料，针对服务端岗位(JAVA)，内容比较基础，倾向小白突击复习，大神请绕路

个人水平有限，整理过程中应该会有错误，如果你发现了请联系我，有空的话这个仓库会更新下去的，希望学弟学妹们2021都有好的归宿

最后字节内推待入职后可联系我（抖音部门）

VX: sen873591769

email:[sen873591769@163.com](mailto:sen873591769@163.com)

## 0.场景题

---

### 十亿个 QQ 号判断一个是否在其中

---

由于待排序的数据记录较多，我们单纯地使用常见的排序方法时间效率较低，运行时间会很长。而且内存空间有限（限制为1MB左右），所以我们不能同时把所有整数读入内存（如果每个整数使用7个字节来存储，那么1MB内存空间只能存大约143000个数字）。当然我们可以多次读取输入文件，多次排序，但是更好的方案是使用位图排序，可以使用有限的1MB内存空间并只进行一趟排序。

- 1.根据待排序集合中最大的数，开辟一个位数组，用来表示待排序集合中的整数；
- 2.待排序集合中的数字在位数组中的对应位置置1，其他的置0；

例如，待排序集合{1,2,3,5,8,13}可以表示为：0-1-1-1-0-1-0-0-1-0-0-0-0-1

这样排序过程自然可以分为三步：

第一步：将所有的位都置为0；

第二步：通过读入文件中的每个整数，将每个对应的位都置为1；

第三步：检验每一位，如果该位为1，输出对应的整数。

注意：位图排序是使用一个二进制位而不是一个整数来表示0或1，这样可以大大地减少所需要的内存空间。使用位图排序的前提是要知道待排序序列中的最大数。位图排序的缺点是有些数没有出现过，仍要为其保留一个位。故位图排序比较适合关键字密集的序列，例如一个QQ号码。

```

/*Phase 1: initialize set to empty*/
for i = [0, n)
    bit[i] = 0

/*Phase 2: insert present elements into the set*/
for each i in the input file
    bit[i] = 1

/*Phase 3: write sorted output*/
for i = [0, n)
    if bit[i] == 1
        write i on the output file

```

### 三、使用位图排序的方法

位图排序时，我们需要考虑：给出一个数，如何找到其对应位图的位置，方法就是首先找到该数对应的字节，然后在找到该数对应的位。例如一个QQ号是：983262245，则将bit的98326625位进行标记。bitset是C++提供的一种位集合的数据结构，它让我们可以像使用数组一样使用位，可以访问指定下标的bit位。因此将通过bitset容器进行存储42个qq号码。由于一个字节可以存放8个QQ号码，则  $4200000000/8/1014/1024 = 500.679\text{Mb}$ ，内存合适，通过bit位下表来判断QQ号码是否存在。

```

#include<iostream>
#include<bitset>
#include<cstdio>
#include<vector>
#include<algorithm>
using namespace std;
const unsigned int MAX = 4200000010;
typedef unsigned int UT;
bitset<MAX> bit;
int main(){
    //开始存储QQ
    for(UT i=1;i<10;i++){
        UT qq;
        printf("请输入第%d个QQ号:",i);
        scanf("%d",&qq);
        bit.set(qq);
    }
    UT qq;
    printf("请输入:");
    while(scanf("%d",&qq)!=0){
        if(bit.test(qq)){
            printf("Yes\n");
        }
        printf("请输入:");
    }
    return 0;
}

```

存储：空间占用大约500Mb

查找：时间复杂度为O(1)

通过位排序的方法，在实现内存内，实现在O(1)时间复杂度内进行一个QQ号码的查找。

## 现在要完成一个微博评论的部分，想在用户进入新闻时优先看到自己好友对此新闻的评论，好友可能有多条评论，怎么设计结构。微博新闻评论设计

每一条新闻对应一个map<int,vector>，key是用户id，value中包括了这个用户所有评论在新闻中的位置。每一条新闻对应一个hashmap存放新闻中的评论位置和评论内容，当前用户访问时，对他的每一个好友id在map里面查找，并对应显示。感觉面试官还比较满意，不过忘记说应该有评论和评论id的对应了，可能关系不大。

## 1000w (L) 个整数排序，范围0到100w(S)，8g内存

计数排序 时间： $O(1000W+100W)$  空间： $O(100w)$

首先需要两个辅助数组，第一个数组A要记录最终排好序的输出数列(输出数组)，大小为n；第二个数组B要记录比某个数小的其他数字的个数，大小应当为K；

- 1、扫描整个集合S，对每一个 $S_i \in S$ ，找到在线性表L中小于等于 $S_i$ 的元素的个数 $T(S_i)$ ；
- 2、扫描整个线性表L，对L中的每一个元素 $L_i$ ，将 $L_i$ 放在输出数组的第 $T(L_i)$ 个位置上，并将 $T(L_i)$ 减1。

## 如果把访问次数过多的IP拉入黑名单，怎么实现，用什么数据结构，写个伪码

创建两个Map，一个(ipMap)用来存放用户Ip和访问次数，访问时间等主要信息，另一个(limitedIpMap)用来存放被限制的用户IP。Map的key为用户的IP，value为具体内容。当用户访问系统时，通过IPFilter检查limitedIpMap中是否存在当前IP，如果存在说明该IP之前存在过恶意刷新访问，已经被限制，跳转到异常提示页面；如果limitedIpMap中不存在则检查ipMap中是否存在当前IP，如果ipMap中不存在则说明用户初次访问，用户访问次数+1，初始访问时间为当前时间；如果存在则检查用户访问次数是否在规定的短时间内进行了大量的访问操作；如果是，则将当前IP添加到limitedIpMap中，并跳转到异常提示页面，否则不进行操作，直接放行本次请求。

## 两个大文件存url，找相同url，两个50亿url文件，找到相同url，内存4g

1. 分别扫描A，B两个文件，根据 $\text{hash}(\text{url})\%k$ (k为正整数，比如 $k=1000$ ，k的取值保证内存可以放下即可)将url划分到不同的k个文件中，比如 $a_0, a_1, \dots, a_{999}; b_0, b_1, \dots, b_{999}$ ；这样处理后相同的url肯定在对应的小文件中( $a_0$  vs  $b_0, a_1$  vs  $b_1, \dots, a_{999}$  vs  $b_{999}$ )因为相同的 $\text{url}\%1000$ 的值肯定相同，不对应的小文件不可能有相同的url；然后我们只要求出1000对小文件中相同的url即可。比如对于 $a_0$  vs  $b_0$ ，我们可以遍历 $a_0$ ，将其中的url存放到hash\_map中，然后遍历 $b_0$ ，如果 $b_0$ 中的某个url在hash\_map中，则说明此url在a和b中同时存在，保存下来即可。
2. 布隆过滤器

## 实现一个web服务器

以FTP服务器为例

首先创建一个服务器socket，然后bind地址，listen监听，然后把socket加入多路转接监听链表。当有连接到达的时候，我们对socket调用accept，返回一个已连接套接字描述符，然后根据用户传输过来的文件名去查找文件，读取文件内容并回送给用户。

accept之后创建一个线程，如果使用线程池的话就从池中取一个空闲线程，然后把已连接文件描述符传给这个线程，然后让线程去处理这个用户请求，一个线程处理一个用户请求。

## 设计实现一个HTTP代理服务器

基本原理：代理服务器作为真实服务器的一个代理端，客户端的请求信息不是直接发送到其真实请求的服务器而是发送到代理服务器，此时代理服务器是作为一个服务器，之后代理服务器通过解析客户端的请求信息，再向真实服务器发送请求报文，获得请求的信息，此时代理服务器是作为一个客户端。

使用代理服务器的好处是：1. 在请求客户端和真实服务器之间添加了一层，这样就可控的对于请求的响应报文做一些限制或者是改变，例如网站过滤、钓鱼网站等，使得响应到客户端的信息是代理服务器处理过的；2、还有就是请求报文先发送到代理服务器，这样代理服务器可以设立缓存，通过对请求报文解析后代理服务器可以通过查找本地缓存，如果有缓存好的，并且通过向服务器发送是否更新的信息后得到没有修改后就可以直接从代理服务器将响应报文返回给客户端，这样减少了服务端的负载，减少了流量。

<https://blog.csdn.net/rocketeerLi/article/details/83717613>

## 把一个文件快速下发到100w个服务器

**树状：** 1. 每个服务器既具有文件存储能力也应具有文件分发能力。

2. 每个服务器接收到文件之后向较近的服务器分发，具体类似多叉树，应该挺快的。

**索引状：** 1. 设置1000个缓存服务器，文件先下发到这些缓存上。（具体多少缓存、分几层缓存和具体业务有关。） 2. 每个缓存服务器接收1000个服务器取文件。

## 1-100顺序排列，丢失了一个怎么找到（二分法）

二分法，比如第一次的话，你取arr[50]，看看是不是50,如果是，说明缺失的数在后面，如果是49，则缺失的数在[0-49]位置，包括49，依次类推即可

## 先手必胜策略问题：

n本书，每次能够拿X-X本，怎么拿必胜，拿到最后剩下的算胜利。

N个糖果，每次只能取1个到6个，不能不取，你先取，请问是否有必胜策略，怎么取。

每次取到只剩7的倍数个糖果即可。因为剩下的为7的倍数时可以保证对方不能一次性取完。当取到最后剩下7个时，对方不能取完，然后自己可以取完。

## 有1亿条边，边是由两个它的顶点的二元组来描述的，如何找出有多少棵独立的树？分析算法的时间复杂度

先建邻接表然后BFS，其实可以用并查集做

## 以大型网站为例，你认为如果想设计一个稳定、高效的后台，都需要有哪些部分以及各部分的要点？

<https://blog.51cto.com/13527416/2085258?cid=700792>

# 从用户在浏览器中输入一个url并点击回车，到浏览器界面出现内容，都发生了什么？

<https://blog.csdn.net/alzzw/article/details/101902468>

首先输入域名，输入域名后，浏览器会通过DNS服务器获取对应的ip地址，这个获取过程就是，先在浏览器缓存里查找，如果浏览器里没有就去系统缓存里查找，还没有就去路由器缓存里查找，递归查找，直到找到，比如说：我一个[www.baidu.com](http://www.baidu.com)，首先根域名服务器会告诉本机域名服务器，让他去顶级域名服务器dns.com查找，查找后顶级域名服务器又让他去权限域名服务器dns.baidu.com去查找，然后查找完成后，会给他返回一个[www.baidu.com](http://www.baidu.com)的ip地址，本地域名服务器再告诉主机。查询到ip之后就建立连接，因为HTTP也是用TCP/IP实现的，所以也要进行三次握手，（讲一下三次握手）连接成功后开始通信，浏览器向服务器发送http或者https请求，（讲一下http与https的区别），浏览器会发送一个请求头，请求头里面有GET或者POST的方法，（讲一下GET和POST的区别），ACCEPT, USER, HOST, CONNECT（长连接和短连接），发送请求后，因为有些大网站会有很多的主机站点，为了负载均衡，（说一下负载均衡）所以会做重定向，而不是返回200，OK，是返回一个301，302的状态响应码，重定向到不同的或者最近的主机上，请求成功后，服务器会处理请求，然后响应200，OK。TCP四次挥手，显示浏览器页面，浏览器获取内容。

## 1. JAVA基础

### int和Integer有什么区别？

为了编程的方便还是引入了基本数据类型，但是为了能够将这些基本数据类型当成对象操作，Java为每一个基本数据类型都引入了对应的包装类型（wrapper class），int的包装类就是Integer，从Java 5开始引入了自动装箱/拆箱机制，使得二者可以相互转换。

Java 提供两种不同的类型：引用类型和原始类型（或内置类型）。Int是java的原始数据类型，Integer是java为int提供的封装类。

- 原始类型: boolean, char, byte, short, int, long, float, double

- 包装类型: Boolean, Character, Byte, Short, Integer, Long, Float, Double

### 请你谈谈大O符号(big-O notation)并给出不同数据结构的例子

大O符号描述了当数据结构里面的元素增加的时候，算法的规模或者是性能在最坏的场景下有多么好。大O符号也可用来描述其他的行为，比如：内存消耗。因为集合类实际上是数据结构，我们一般使用大O符号基于时间，内存和性能来选择最好的实现。大O符号表示一个程序运行时所需要的渐进时间复杂度上界。

### 请你解释什么是值传递和引用传递？

值传递是对**\*基本型变量而言\***的,传递的是该变量的一个副本,改变副本不影响原变量。

引用传递一般是对**\*对象型变量而言\***的,传递的是该对象地址的一个副本,并不是原对象本身。所以对引用对象进行操作会同时改变原对象.一般认为java内的传递都是值传递。

### 请你说说Lamda表达式的优缺点

优点：1. 简洁。2. 非常容易并行计算。3. 可能代表未来的编程趋势。

缺点：1. 若不用并行计算，很多时候计算速度没有比传统的 for 循环快。（并行计算有时需要预热才显示出效率优势）2. 不容易调试。3. 若其他程序员没有学过 lambda 表达式，代码不容易让其他语言的程序员看懂。

## 你知道java8的新特性吗，请简单介绍一下

---

Lambda 表达式 - Lambda允许把函数作为一个方法的参数

方法引用- 方法引用提供了非常有用的语法，可以直接引用已有Java类或对象（实例）的方法或构造器。与lambda联合使用，方法引用可以使语言的构造更紧凑简洁，减少冗余代码。

默认方法- 默认方法就是一个在接口里面有了一个实现的方法。

新工具- 新的编译工具，如：Nashorn引擎 jjs、类依赖分析器jdeps。

## ==与equals有什么区别？

---

**\*==\***

- 如果是基本类型，判断它们值是否相等；
- 如果是引用对象，判断两个对象指向的内存地址是否相同。

**\*equals\***

- 如果是字符串，表示判断字符串内容是否相同；
- 如果是object对象的方法，比较的也是引用的内存地址值；
- 如果自己的类重写equals方法，可以自定义两个对象是否相等。

## final关键字

---

当用final修饰一个类时，表明这个类不能被继承。“使用final方法的原因有两个。第一个原因是把方法锁定，以防任何继承类修改它的含义；第二个原因是效率。

对于一个final变量，如果是基本数据类型的变量，则其数值一旦在初始化之后便不能更改；如果是引用类型的变量，则在对其初始化之后便不能再让其指向另一个对象。

## 接口和抽象类的区别是什么？

---

Java提供和支持创建抽象类和接口。它们的实现有共同点，不同点在于：

接口中所有的方法隐含的都是抽象的。而抽象类则可以同时包含抽象和非抽象的**\*方法\***。

类可以实现很多个接口，但是只能继承一个抽象类

类可以不实现抽象类和接口声明的所有方法，当然，在这种情况下，类也必须得声明成是抽象的。

抽象类可以在不提供接口方法实现的情况下实现接口。

Java接口中声明的变量默认都是final的。抽象类可以包含非final的变量。

Java接口中的成员函数默认是public的。抽象类的成员函数可以是private，protected或者是public。

接口是绝对抽象的，不可以被实例化。抽象类也不可以被实例化，如果它包含main方法的话是可以被调用的。

## 请你说说Iterator和ListIterator的区别？

---

Iterator可用来遍历Set和List集合，但是ListIterator只能用来遍历List。  
Iterator对集合只能是前向遍历，ListIterator既可以前向也可以后向。  
ListIterator实现了Iterator接口，并包含其他的功能：增加元素，替换元素，获取前一个和后一个元素的索引

## 请问什么是java序列化？ 以及如何实现java序列化？

序列化就是一种用来处理对象流的机制，所谓对象流也就是将对象的内容进行流化。可以对流化后的对象进行读写操作，也可将流化后的对象传输于网络之间。序列化是为了解决在对对象流进行读写操作时所引发的问题。

## java内存模型(Java Memory Model)

java虚拟机规范定义的，用来屏蔽掉java程序在各种不同的硬件和操作系统对内存的访问的差异，这样就可以实现java程序在各种不同的平台上都能达到内存访问的一致性。可以避免像c++等直接使用物理硬件和操作系统的内存模型在不同操作系统和硬件平台下表现不同，比如有些c/c++程序可能在windows平台运行正常，而在linux平台却运行有问题。

## 四种IO模型

- 1) 同步阻塞IO (Blocking IO)
- 2) 同步非阻塞IO (Non-blocking IO)
- 3) IO多路复用 (IO Multiplexing)
- 4) 异步IO (Asynchronous IO)

### 阻塞与非阻塞

阻塞与非阻塞指的是单个线程内遇到同步等待时，是否在原地不做任何操作。

阻塞指的是遇到同步等待后，一直在原地等待同步方法处理完成。

非阻塞指的是遇到同步等待，不在原地等待，先去做其他的操作，隔断时间再来观察同步方法是否完成。

阻塞与非阻塞关注的是线程是否在原地等待。

笔者认为阻塞和非阻塞仅能与同步进行组合。而异步天然就是非阻塞的，而这个非阻塞是对主线程而言。（可能有人认为异步方法里面放入阻塞操作的话就是异步阻塞，但是思考一下，正是因为是阻塞操作所以才会将它放入异步方法中，不要阻塞主线程）

### 例子讲解

海底捞很好吃，但是经常要排队。我们就以生活中的这个例子进行讲解。

- A顾客去吃海底捞，就这样干坐着等了一小时，然后才开始吃火锅。(BIO)
- B顾客去吃海底捞，他一看要等挺久，于是去逛商场，每次逛一会就跑回来看有没有排到他。于是他最后既购了物，又吃上海底捞了。(NIO)
- C顾客去吃海底捞，由于他是高级会员，所以店长说，你去商场随便玩吧，等下有位置，我立马打电话给你。于是C顾客不用干坐着等，也不用每过一会儿就跑回来看有没有等到，最后也吃上了海底捞(AIO)

哪种方式更有效率呢？是不是一目了然呢？

### BIO

BIO全称是Blocking IO，是JDK1.4之前的传统IO模型，本身是同步阻塞模式。线程发起IO请求后，一直阻塞IO，直到缓冲区数据就绪后，再进入下一步操作。针对网络通信都是一请求一应答的方式，虽然简化了上层的应用开发，但在性能和可靠性方面存在着巨大瓶颈，试想一下如果每个请求都需要新建一个线程来专门处理，那么在高并发的场景下，机器资源很快就会被耗尽。

## NIO

NIO也叫Non-Blocking IO 是同步非阻塞的IO模型。线程发起io请求后，立即返回（非阻塞io）。同步指的是必须等待IO缓冲区内的数据就绪，而非阻塞指的是，用户线程不原地等待IO缓冲区，可以先做一些其他操作，但是要定时轮询检查IO缓冲区数据是否就绪。Java中的NIO 是new IO的意思。其实是NIO加上IO多路复用技术。普通的NIO是线程轮询查看一个IO缓冲区是否就绪，而Java中的new IO指的是线程轮询地去查看一堆IO缓冲区中哪些就绪，这是一种IO多路复用的思想。IO多路复用模型中，将检查IO数据是否就绪的任务，交给系统级别的select或epoll模型，由系统进行监控，减轻用户线程负担。

NIO主要有buffer、channel、selector三种技术的整合，通过零拷贝的buffer取得数据，每一个客户端通过channel在selector（多路复用器）上进行注册。服务端不断轮询channel来获取客户端的信息。channel上有connect,accept（阻塞）、read（可读）、write(可写)四种状态标识。根据标识来进行后续操作。所以一个服务端可接收无限多的channel。不需要新开一个线程。大大提升了性能。

## AIO

AIO是真正意义上的异步非阻塞IO模型。上述NIO实现中，需要用户线程定时轮询，去检查IO缓冲区数据是否就绪，占用应用程序线程资源，其实轮询相当于还是阻塞的，并非真正解放当前线程，因为它还是需要去查询哪些IO就绪。而真正的理想的异步非阻塞IO应该让内核系统完成，用户线程只需要告诉内核，当缓冲区就绪后，通知我或者执行我交给你的回调函数。

AIO可以做到真正的异步的操作，但实现起来比较复杂，支持纯异步IO的操作系统非常少，目前也就windows是IOCP技术实现了，而在Linux上，底层还是使用的epoll实现的。

## 反射

### 反射机制

对象的类型原本在程序编译时就已确定（类型写死），而反射机制可以在程序运行时动态地加载类（类型灵活），增加了程序的灵活性

反射机制应用场景：

- 1.反编译（.class—>.java）
- 2.Spring的配置文件
- 3.Idea等编译器的联想功能

getField（）返回public的变量

getDeclField（）返回public 和private变量

## String，Stringbuffer，StringBuilder的区别

### **\*String: \***

- String类是一个不可变的类，一旦创建就不可以修改。
- String是final类，不能被继承
- String实现了equals()方法和hashCode()方法

### **\*StringBuffer: \***

- 继承自AbstractStringBuilder，是可变类。
- StringBuffer是线程安全的



· 可以通过append方法动态构造数据。

#### **\*StringBuilder: \***

继承自AbstractStringBuilder, 是可变类。

StringBuilder是非线性安全的。

执行效率比StringBuffer高。

#### **\*String s 与new String的区别\***

[复制代码](#)

12	String str ="whx";String newStr =new String ("whx");

#### **\*String str ="whx"\***

先在常量池中查找有没有"whx" 这个对象,如果有, 就让str指向那个"whx".如果没有, 在常量池中新建一个"whx"对象, 并让str指向在常量池中新建的对象"whx"。

#### **\*String newStr =new String ("whx");\***

是在堆中建立的对象"whx",在栈中创建堆中"whx" 对象的内存地址。



## 反射的原理，反射创建类实例的三种方式是什么

#### **\*Java反射机制: \***

Java 的反射机制是指在运行状态中, 对于任意一个类都能够知道这个类所有的属性和方法; 并且对于任意一个对象, 都能够调用它的任意一个方法; 这种**\*动态获取信息以及动态调用对象\***方法的功能成为Java语言的反射机制

#### **\*获取Class 类对象三种方式: \***

- 使用 Class.forName 静态方法
- 使用类的.class 方法
- 使用实例对象的 getClass() 方法

## JDK动态代理与cglib实现的区别

- java动态代理是利用反射机制生成一个**\*实现代理接口的匿名类\***, 在调用具体方法前调用 InvokeHandler处理。
- cglib动态代理是利用asm开源包, 对代理对象类的class文件加载进来, 通过修改其字节码生成子类来处理。
- JDK动态代理只能对实现了接口的类生成代理, 而不能针对类
- cglib是针对类实现代理, 主要是对指定的类生成一个子类, 覆盖其中的方法。因为是继承, 所以该类或方法最好不要声明成final

## 谈谈序列化与反序列化

- 序列化是指将对象转换为字节序列的过程, 而反序列化则是将字节序列转换为对象的过程。

· Java对象序列化是将实现了Serializable接口的对象转换成一个字节序列，能够通过网络传输、文件存储等方式传输，传输过程中却不必担心数据在不同机器、不同环境下发生改变，也不必关心字节的顺序或其他任何细节，并能够在以后将这个字节序列完全恢复为原来的对象。

## 分布式结构,怎么保证数据一致性

---

规避分布式事务——业务整合  
经典方案 - eBay 模式

## 2. 数据库相关

---

### SQL注入

---

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。

1) SQL注入攻击的总体思路

- 寻找到SQL注入的位置
- 判断服务器类型和后台数据库类型
- 针对不同的服务器和数据库特点进行SQL注入攻击

2) 应对方法

- 使用正则表达式过滤传入的参数
- 参数绑定
- 使用预编译手段，绑定参数是最好的防SQL注入的方法。

### 请你说明一下 left join 和 right join 的区别？

---

left join(左联接)：返回包括左表中的所有记录和右表中\***联结字段**\*相等的记录

right join(右联接)：返回包括右表中的所有记录和左表中联结字段相等的记录

### 数据库ACID的特性

---

\***原子性**\*是指事务是一个不可分割的工作单位，事务中的操作要么都发生，要么都不发生。

\***一致性**\*指事务前后数据的完整性必须保持一致。

\***隔离性**\*指多个用户并发访问数据库时，一个用户的事务不能被其他用户的事务所干扰，多个并发事务之间数据要相互隔离。

\***持久性**\*是指一个事务一旦提交，它对数据库中数据的改变就是永久性的，即便数据库发生故障也不应该对其有任何影响。

### 请你介绍一下，数据库的三个范式？

---

第一范式（1NF）：强调的是列的原子性，即\***列不能够再分成其他几列**\*。

第二范式（2NF）：首先满足 1NF，另外包含两部分内容，一是\***表必须有一个主键**\*；二是没有包含在主键中的\***列必须完全依赖于主键**\*，而不能只依赖于主键的一部分。

第三范式（3NF）：首先是满足2NF，另外非主键列必须直接依赖于主键，不能存在传递依赖。即不能存在：非主键列 A 依赖于非主键列 B，非主键列 B 依赖于主键的情况

### 请你介绍一下数据库的隔离级别

---

<b>*隔离级别*</b>	<b>*脏读 (Dirty Read) *</b>	<b>*不可重复读 (NonRepeatable Read) *</b>	<b>*幻读 (Phantom Read) *</b>
未提交读	可能	可能	可能
已提交读	不可能	可能	可能
可重复读	不可能	不可能	可能
可串行化	不可能	不可能	不可能

未提交读(Read Uncommitted): 允许脏读, 也就是可能读取到其他会话中未提交事务修改的数据。

已提交读(Read Committed): 只能读取到已经提交的数据。Oracle等多数数据库默认都是该级别 (不重复读)。

可重复读(Repeated Read): 可重复读。在同一个事务内的查询都是事务开始时刻一致的, InnoDB默认级别。在SQL标准中, 该隔离级别消除了不可重复读, 但是还存在幻象读。

串行读(Serializable): 完全串行化的读, 每次读都需要获得表级共享锁, 读写相互都会阻塞。

## 数据库的脏读、幻读、不可重复读

### 1.脏读:

指一个事务A正在访问数据, 并且对该数据进行了修改, 但是这种修改还没有提交到数据库中 (也可能因为某些原因Rollback了)。这时候另外一个事务B也访问这个数据, 然后使用了这个被A修改的数据, 那么这个数据就是脏的, 并不是数据库中真实的数据。这就被称作脏读。**(\*事务A读到了事务B未提交的数据\*)**

解决办法: 把数据库事务隔离级别调整到READ\_COMMITTED

即让用户在更新时锁定数据库, 阻止其他用户读取, 直到更新全部完成才让你读取。

### 2.幻读:

指一个事务A对一个表中的数据进行了修改, 而且该修改涉及到表中所有的数据行; 同时另一个事务B也在修改表中的数据, 该修改是向表中插入一行新数据。那么经过这一番操作之后, 操作事务A的用户就会发现表中还有没修改的数据行, 就像发生了幻觉一样。这就被称作幻读。**(\*事务A修改数据, 事务B插入数据, A发现表中还没有修改的数据行\*)**

解决办法: 把数据库事务隔离级别调整到SERIALIZABLE\_READ

### 3.不可重复读:

指在一个事务A内, 多次读同一个数据, 但是事务A没有结束时, 另外一个事务B也访问该同一数据。那么在事务A的两次读数据之间, 由于事务B的修改导致事务A两次读到的数据可能是不一样的。这就发生了在一个事务内两次读到的数据不一样, 这就被称作不可重复读。**(\*事务A多次读数据, 事务B访问数据, A读到了B修改的数据, 导致两次读到的数据不一样\*)**

解决办法: 把数据库事务隔离级别调整到REPEATABLE\_READ

级别高低: 脏读 < 不可重复读 < 幻读

所以设置了最高级别的SERIALIZABLE\_READ就不需要设置其他的了, 即解决了幻读问题那么脏度和不可重复读自然就都解决了。

## MySQL 中是如何实现事务隔离的

首先说读未提交，它是性能最好，也可以说它是最野蛮的方式，因为它压根儿就不加锁，所以根本谈不上什么隔离效果，可以理解为没有隔离。

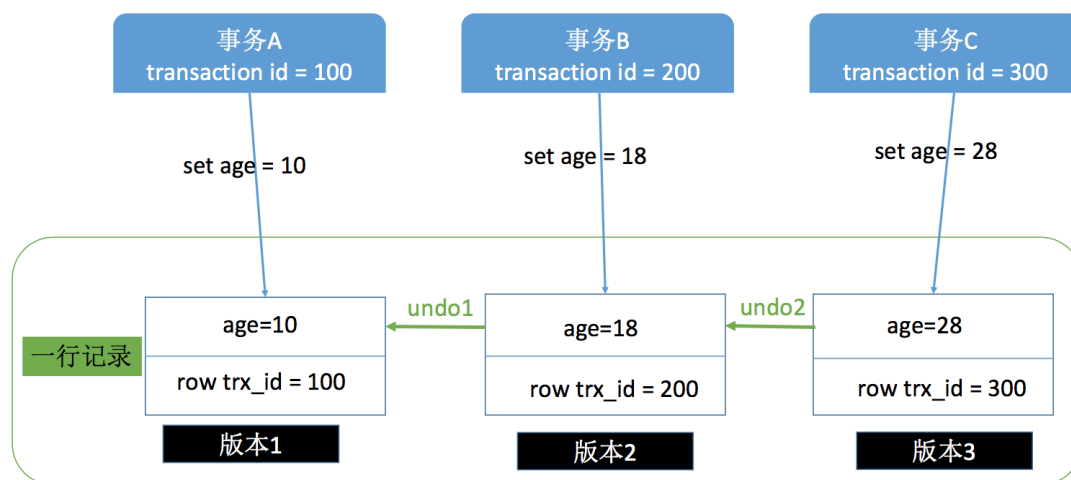
再来说串行化。读的时候加共享锁，也就是其他事务可以并发读，但是不能写。写的时候加排它锁，其他事务不能并发写也不能并发读。

最后说读提交和可重复读。这两种隔离级别是比较复杂的，既要允许一定的并发，又想要兼顾的解决问题。

### 实现可重复读

为了解决不可重复读，或者为了实现可重复读，MySQL 采用了 MVCC (多版本并发控制) 的方式。

我们在数据库表中看到的一行记录可能实际上有多个版本，每个版本的记录除了有数据本身外，还要有一个表示版本的字段，记为 row trx\_id，而这个字段就是使其产生的事务的 id，事务 ID 记为 transaction id，它在事务开始的时候向事务系统申请，按时间先后顺序递增。



按照上面这张图理解，一行记录现在有 3 个版本，每一个版本都记录这使其产生的事务 ID，比如事务 A 的 transaction id 是 100，那么版本 1 的 row trx\_id 就是 100，同理版本 2 和版本 3。

在上面介绍读提交和可重复读的时候都提到了一个词，叫做快照，学名叫做一致性视图，这也是可重复读和不可重复读的关键，可重复读是在事务开始的时候生成一个当前事务全局性的快照，而读提交则是每次执行语句的时候都重新生成一次快照。

对于一个快照来说，它能够读到那些版本数据，要遵循以下规则：

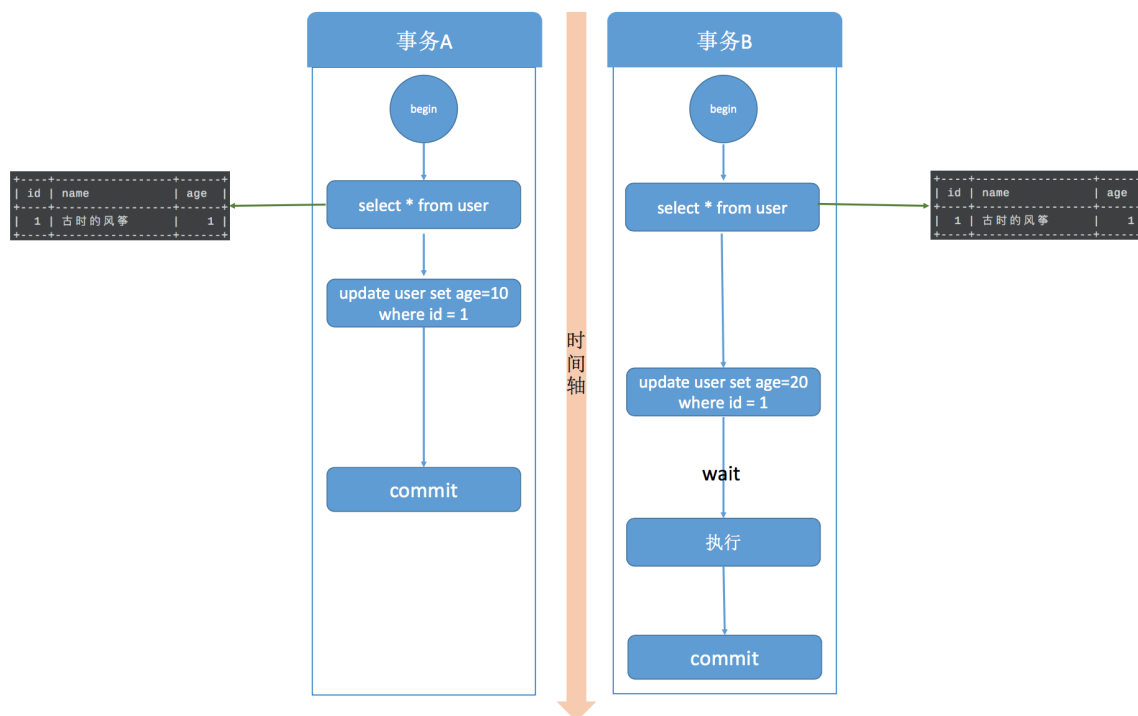
1. 当前事务内的更新，可以读到；
2. 版本未提交，不能读到；
3. 版本已提交，但是却在快照创建后提交的，不能读到；
4. 版本已提交，且是在快照创建前提交的，可以读到；

利用上面的规则，再返回去套用到读提交和可重复读的那两张图上就很清晰了。还是要强调，两者主要的区别就是在快照的创建上，可重复读仅在事务开始是创建一次，而读提交每次执行语句的时候都要重新创建一次。

### 并发写问题

存在这的情况，两个事务，对同一条数据做修改。最后结果应该是哪个事务的结果呢，肯定要是时间靠后的那个对不对。并且更新之前要先读数据，这里所说的读和上面说到的读不一样，更新之前的读叫做“当前读”，总是当前版本的数据，也就是多版本中最新一次提交的那版。

假设事务A执行 update 操作，update 的时候要对所修改的行加行锁，这个行锁会在提交之后才释放。而在事务A提交之前，事务B也想 update 这行数据，于是申请行锁，但是由于已经被事务A占有，事务B是申请不到的，此时，事务B就会一直处于等待状态，直到事务A提交，事务B才能继续执行，如果事务A的时间太长，那么事务B很有可能出现超时异常。如下图所示。



加锁的过程要分有索引和无索引两种情况，比如下面这条语句

```
update user set age=11 where id = 1
```

id 是这张表的主键，是有索引的情况，那么 MySQL 直接就在索引数中找到了这行数据，然后干净利落的加上行锁就可以了。

而下面这条语句

```
update user set age=11 where age=10
```

表中并没有为 age 字段设置索引，所以，MySQL 无法直接定位到这行数据。那怎么办呢，当然也不是加表锁了。MySQL 会为这张表中所有行加行锁，没错，是所有行。但是呢，在加上行锁后，MySQL 会进行一遍过滤，发现不满足的行就释放锁，最终只留下符合条件的行。虽然最终只为符合条件的行加了锁，但是这一锁一释放的过程对性能也是影响极大的。所以，如果是大表的话，建议合理设计索引，如果真的出现这种情况，那很难保证并发度。

### 解决幻读

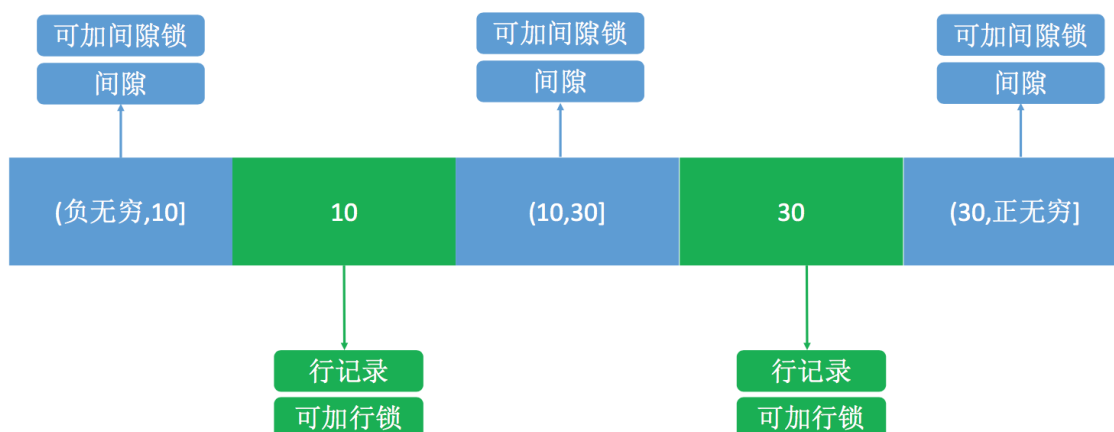
上面介绍可重复读的时候，那张图里标示着出现幻读的地方实际上在 MySQL 中并不会出现，MySQL 已经在可重复读隔离级别下解决了幻读的问题。

前面刚说了并发写问题的解决方式就是行锁，而解决幻读用的也是锁，叫做间隙锁，MySQL 把行锁和间隙锁合并在一起，解决了并发写和幻读的问题，这个锁叫做 Next-Key 锁。

假设现在表中有两条记录，并且 age 字段已经添加了索引，两条记录 age 的值分别为 10 和 30。

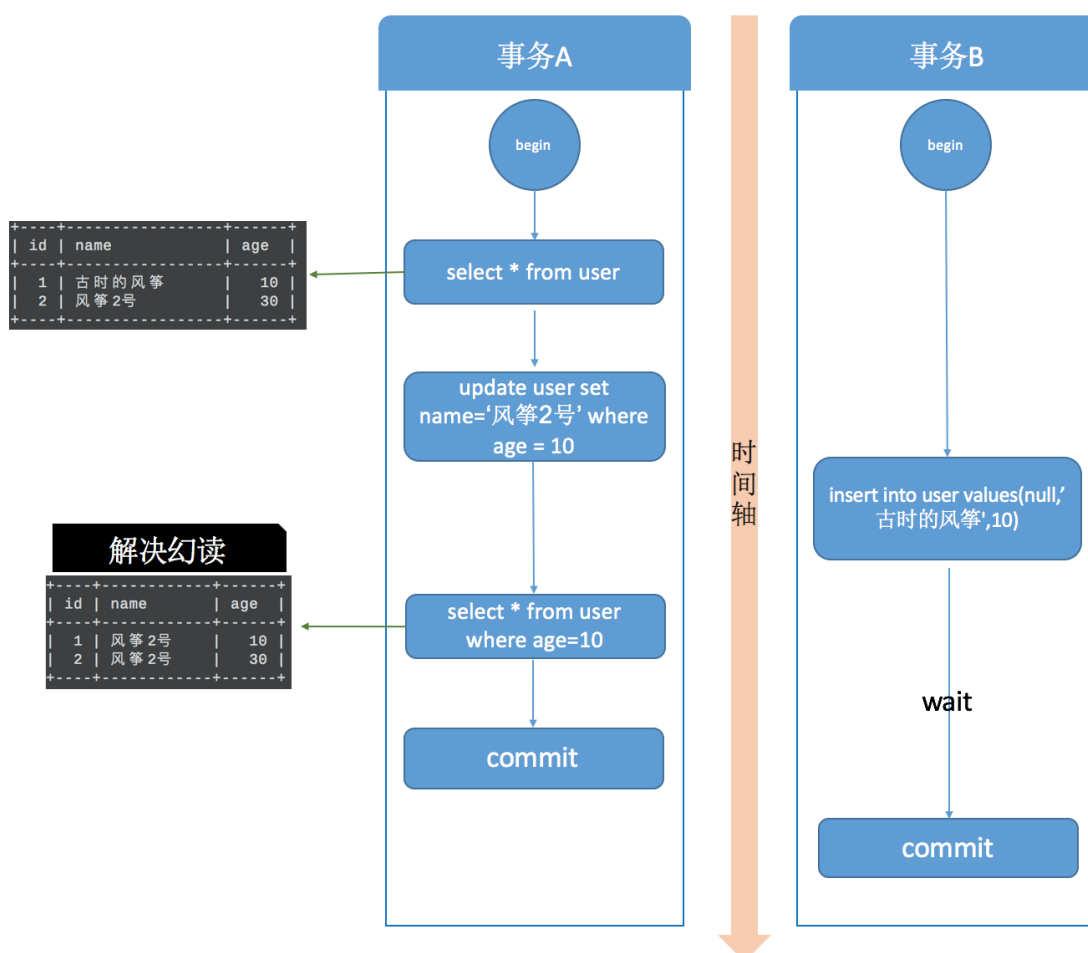
id	name	age
1	古时的风筝	10
2	风筝2号	30

此时，在数据库中会为索引维护一套B+树，用来快速定位行记录。B+索引树是有序的，所以会把这张表的索引分割成几个区间。



如图所示，分成了3个区间，(-无穷,10]、(10,30]、(30,正无穷]，在这3个区间是可以加间隙锁的。

之后，我用下面的两个事务演示一下加锁过程。



在事务A提交之前，事务B的插入操作只能等待，这就是间隙锁起得作用。当事务A执行 `update user set name='风筝2号' where age = 10;` 的时候，由于条件 `where age = 10`，数据库不仅在 `age = 10` 的行上添加了行锁，而且在这条记录的两边，也就是`(负无穷,10]`、`(10,30]`这两个区间加了间隙锁，从而导致事务B插入操作无法完成，只能等待事务A提交。不仅插入 `age = 10` 的记录需要等待事务A提交，`age < 10`、`10 < age < 30` 的记录页无法完成，而大于等于30的记录则不受影响，这足以解决幻读问题了。

这是有索引的情况，如果 `age` 不是索引列，那么数据库会为整个表加上间隙锁。所以，如果是没有索引的话，不管 `age` 是否大于等于30，都要等待事务A提交才可以成功插入。

## 排他锁和共享锁

排它锁 (Exclusive)，又称为X 锁，写锁。

共享锁 (Shared)，又称为S 锁，读锁。

读写锁之间有以下的关系：

- 一个事务对数据对象O加了 S 锁，可以对 O 进行读取操作，但是不能进行更新操作。加锁期间其它事务能对O 加 S 锁，但是不能加 X 锁。
- 一个事务对数据对象 O 加了 X 锁，就可以对 O 进行读取和更新。加锁期间其它事务不能对 O 加任何锁。

即读写锁之间的关系可以概括为：多读单写

## MVCC-隔离等级的实现-


 image-20200902134340451

 image-20200902134635410

在MySQL中，会在表中每一条数据后面添加两个字段：

创建版本号：创建一行数据时，将当前系统版本号作为创建版本号赋值

删除版本号：删除一行数据时，将当前系统版本号作为删除版本号赋值

### SELECT

select时读取数据的规则为：创建版本号 $\leq$ 当前事务版本号，删除版本号为空或 $>$ 当前事务版本号。

创建版本号 $\leq$ 当前事务版本号保证取出的数据不会有后启动的事务中创建的数据。这也是为什么在开始的示例中我们不会查出后来添加的数据的原因

删除版本号为空或 $>$ 当前事务版本号保证了至少在该事务开启之前数据没有被删除，是应该被查出来的数据。

### INSERT

insert时将当前的系统版本号赋值给创建版本号字段。

### UPDATE

插入一条新纪录，保存当前事务版本号为行创建版本号，同时保存当前事务版本号到原来删除的行，实际上这里的更新是通过delete和insert实现的。

### DELETE

删除时将当前的系统版本号赋值给删除版本号字段，标识该行数据在那一个事务中会被删除，即使实际上在位commit时该数据没有被删除。根据select的规则后开启懂数据也不会查询到该数据。

## bin Log



`binlog` 用于记录数据库执行的写入性操作(不包括查询)信息, 以二进制的形式保存在磁盘中。`binlog` 是 `mysql` 的逻辑日志, 并且由 `Server` 层进行记录, 使用任何存储引擎的 `mysql` 数据库都会记录 `binlog` 日志。

逻辑日志: 可以简单理解为记录的就是sql语句。

物理日志: 因为 `mysql` 数据最终是保存在数据页中的, 物理日志记录的就是数据页变更。

`binlog` 是通过追加的方式进行写入的, 可以通过 `max_binlog_size` 参数设置每个 `binlog` 文件的大小, 当文件大小达到给定值之后, 会生成新的文件来保存日志。

## binlog使用场景

在实际应用中, `binlog` 的主要使用场景有两个, 分别是**主从复制**和**数据恢复**。

1. **主从复制**: 在 `Master` 端开启 `binlog`, 然后将 `binlog` 发送到各个 `Slave` 端, `Slave` 端重放 `binlog` 从而达到主从数据一致。
2. **数据恢复**: 通过使用 `mysqlbinlog` 工具来恢复数据。

## binlog刷盘时机

对于 `InnoDB` 存储引擎而言, 只有在事务提交时才会记录 `biglog`, 此时记录还在内存中, 那么 `biglog` 是什么时候刷到磁盘中的呢? `mysql` 通过 `sync_binlog` 参数控制 `biglog` 的刷盘时机, 取值范围是 `0-N`:

- 0: 不去强制要求, 由系统自行判断何时写入磁盘;
- 1: 每次 `commit` 的时候都要将 `binlog` 写入磁盘;
- N: 每N个事务, 才会将 `binlog` 写入磁盘。

从上面可以看出, `sync_binlog` 最安全的是设置是 `1`, 这也是 `MySQL 5.7.7` 之后版本的默认值。但是设置一个大一些的值可以提升数据库性能, 因此实际情况下也可以将值适当调大, 牺牲一定的一致性来获取更好的性能。

为什么需要redo log

我们都知道, 事务的四大特性里面有一个是持久性, 具体来说就是只要事务提交成功, 那么对数据库做的修改就被永久保存下来了, 不可能因为任何原因再回到原来的状态。那么`mysql`是如何保证一致性的呢? 最简单的做法是在每次事务提交的时候, 将该事务涉及修改的数据页全部刷新到磁盘中。但是这么做会有严重的性能问题, 主要体现在两个方面:

因为`Innodb`是以页为单位进行磁盘交互的, 而一个事务很可能只修改一个数据页里面的几个字节, 这个时候将完整的数据页刷到磁盘的话, 太浪费资源了!

一个事务可能涉及修改多个数据页, 并且这些数据页在物理上并不连续, 使用随机IO写入性能太差!

因此`mysql`设计了redo log, 具体来说就是只记录事务对数据页做了哪些修改, 这样就能完美地解决性能问题了(相对而言文件更小并且是顺序IO)。

## redo log

redo log包括两部分: 一个是内存中的日志缓冲(redo log buffer), 另一个是磁盘上的日志文件(redo log file)。mysql每执行一条DML语句, 先将记录写入redo log buffer, 后续某个时间点再一次性将多个操作记录写到redo log file。这种先写日志, 再写磁盘的技术就是MySQL里经常说到的WAL(Write-Ahead Logging) 技术。

在计算机操作系统中, 用户空间(user space)下的缓冲区数据一般情况下是无法直接写入磁盘的, 中间必须经过操作系统内核空间(kernel space)缓冲区(OS Buffer)。因此, redo log buffer写入redo log file 实际上是先写入OS Buffer, 然后再通过系统调用fsync()将其刷到redo log file中





mysql支持三种将redo log buffer写入redo log file的时机，可以通过innodb\_flush\_log\_at\_trx\_commit参数配置，各参数值含义如下：

参数值含义0（延迟写）事务提交时不会将redo log buffer中日志写入到os buffer，而是每秒写入os buffer并调用fsync()写入到redo log file中。也就是说设置为0时是(大约)每秒刷新写入到磁盘中的，当系统崩溃，会丢失1秒钟的数据。1（实时写，实时刷）事务每次提交都会将redo log buffer中的日志写入os buffer并调用fsync()刷到redo log file中。这种方式即使系统崩溃也不会丢失任何数据，但是因为每次提交都写入磁盘，IO的性能较差。2（实时写，延迟刷）每次提交都仅写入到os buffer，然后是每秒调用fsync()将os buffer中的日志写入到redo log file。

## undo log

数据库事务四大特性中有一个是**原子性**，具体来说就是 **原子性是指对数据库的一系列操作，要么全部成功，要么全部失败，不可能出现部分成功的情况**。实际上，**原子性底层就是通过 undo log 实现的**。

undo log 主要记录了数据的逻辑变化，比如一条 INSERT 语句，对应一条 DELETE 的 undo log，对于每个 UPDATE 语句，对应一条相反的 UPDATE 的 undo log，这样在发生错误时，就能回滚到事务之前的数据状态。同时，undo log 也是 MVCC (多版本并发控制)实现的关键，这部分内容在[面试中的老大难-mysql事务和锁，一次性讲清楚！](#)中有介绍，不再赘述。

## MySQL的binlog有有几种录入格式？分别有什么区别？

statement模式下，每一条会修改数据的sql都会记录在binlog中。不需要记录每一行的变化，减少了binlog日志量，节约了IO，提高性能。由于sql的执行是有上下文的，因此在保存的时候需要保存相关的信息，同时还有一些使用了函数之类的语句无法被记录复制。

row级别下，不记录sql语句上下文相关信息，仅保存哪条记录被修改。记录单元为每一行的改动，基本是可以全部记下来但是由于很多操作，会导致大量行的改动(比如alter table)，因此这种模式的文件保存的信息太多，日志量太大。

mixed，一种折中的方案，普通操作使用statement记录，当无法使用statement的时候使用row。

## SQL 语句执行顺序

1. from**子句**组装来自不同数据源的数据；
2. where**子句**基于指定的条件对记录行进行筛选；
3. group by**子句**将数据划分为多个分组；
4. 使用聚集函数进行计算；
5. 使用having**子句**筛选分组；
6. 计算所有的表达式；
7. select 的字段；
8. 使用order by对结果集进行排序。

## 请你简单介绍一下，数据库水平切分与垂直切分

垂直拆分就是要将表按模块划分到不同数据库表中，单表大数据量依然存在性能瓶颈

水平切分就是要将一个表按照某种规则把数据划分到不同表或数据库里。

通俗理解：**\*水平拆分行，行数据拆分到不同表中，垂直拆分列，表数据拆分到不同表中\***。

## mysql数据库的索引类型

**主键索引:** 数据列不允许重复，不允许为NULL，一个表只能有一个主键。

**唯一索引:** 数据列不允许重复，允许为NULL值，一个表允许多个列创建唯一索引。

- 可以通过 `ALTER TABLE table_name ADD UNIQUE (column);` 创建唯一索引
- 可以通过 `ALTER TABLE table_name ADD UNIQUE (column1,column2);` 创建唯一组合索引

**普通索引:** 基本的索引类型，没有唯一性的限制，允许为NULL值。

- 可以通过 `ALTER TABLE table_name ADD INDEX index_name (column);` 创建普通索引
- 可以通过 `ALTER TABLE table_name ADD INDEX index_name(column1, column2, column3);` 创建组合索引

**全文索引:** 是目前搜索引擎使用的一种关键技术。

- 可以通过 `ALTER TABLE table_name ADD FULLTEXT (column);` 创建全文索引

## 创建索引的三种方式

第一种方式：在执行CREATE TABLE时创建索引

```
CREATE TABLE user_index2 ( id INT auto_increment PRIMARY KEY, first_name VARCHAR (16), last_name VARCHAR (16), id_card VARCHAR (18), information text, KEY name (first_name, last_name), FULLTEXT KEY (information), UNIQUE KEY (id_card) );
```

第二种方式：使用ALTER TABLE命令去增加索引

```
ALTER TABLE table_name ADD INDEX index_name (column_list);
```

- ALTER TABLE用来创建普通索引、UNIQUE索引或PRIMARY KEY索引。
- 其中table\_name是要增加索引的表名，column\_list指出对哪些列进行索引，多列时各列之间用逗号分隔。
- 索引名index\_name可自己命名，缺省时，MySQL将根据第一个索引列赋一个名称。另外，ALTER TABLE允许在单个语句中更改多个表，因此可以在同时创建多个索引。

第三种方式：使用CREATE INDEX命令创建

```
CREATE INDEX index_name ON table_name (column_list);
```

- CREATE INDEX可对表增加普通索引或UNIQUE索引。（但是，不能创建PRIMARY KEY索引）

## 删除索引

- 根据索引名删除普通索引、唯一索引、全文索引：`alter table 表名 drop KEY 索引名`

```
alter table user_index drop KEY name;
alter table user_index drop KEY id_card;
alter table user_index drop KEY information;
```

复制代码

- 删除主键索引：`alter table 表名 drop primary key`（因为主键只有一个）。这里值得注意的是，如果主键自增长，那么不能直接执行此操作（自增长依赖于主键索引）：

```
1 alter table user_index drop primary key
```

信息	概况	状态
----	----	----

[SQL]alter table user\_index drop primary key

[Err] 1075 - Incorrect table definition; there can be only one auto column and it must be defined as a key

- 需要取消自增长再行删除：

```
alter table user_index
-- 重新定义字段
MODIFY id int,
drop PRIMARY KEY
复制代码
```

- 但通常不会删除主键，因为设计主键一定与业务逻辑无关。

## 创建索引时需要注意什么？

非空字段：应该指定列为NOT NULL，除非你想存储NULL。在mysql中，含有空值的列很难进行查询优化，因为它们使得索引、索引的统计信息以及比较运算更加复杂。你应该用0、一个特殊的值或者一个空串代替空值；

取值离散大的字段：（变量各个取值之间的差异程度）的列放到联合索引的前面，可以通过count()函数查看字段的差异值，返回值越大说明字段的唯一值越多字段的离散程度高；

索引字段越小越好：数据库的数据存储以页为单位一页存储的数据越多一次IO操作获取的数据越大效率越高。

## 使用索引查询一定能提高查询的性能吗？为什么

通常，通过索引查询数据比全表扫描要快。但是我们也必须注意到它的代价。

- 索引需要空间来存储，也需要定期维护，每当有记录在表中增减或索引列被修改时，索引本身也会被修改。这意味着每条记录的INSERT，DELETE，UPDATE将为此多付出4，5次的磁盘I/O。因为索引需要额外的存储空间和处理，那些不必要的索引反而会使查询反应时间变慢。使用索引查询不一定能提高查询性能，索引范围查询(INDEX RANGE SCAN)适用于两种情况：
- 基于一个范围的检索，一般查询返回结果集小于表中记录数的30%
- 基于非唯一性索引的检索

## 主键索引与唯一索引的区别

主键是一种约束，唯一索引是一种索引，两者在本质上是不同的。

主键创建后一定包含一个唯一性索引，唯一性索引并不一定就是主键。

唯一性索引列允许空值，而主键列不允许为空值。

主键列在创建时，已经默认为空值 ++ 唯一索引了。

一个表最多只能创建一个主键，但可以创建多个唯一索引。

主键更适合那些不容易更改的唯一标识，如自动递增列、身份证号等。

主键可以被其他表引用为外键，而唯一索引不能。

## 聚集索引和非聚集索引的区别？

---

聚集索引：表中那行数据的索引和数据都合并在一起了。

非聚集索引：表中那行数据的索引和数据是分开存储的。

## 一个 SQL 执行的很慢，我们要分两种情况讨论：

---

1、偶尔很慢，则有如下原因

(1)、数据库在刷新脏页，例如 redo log 写满了需要同步到磁盘。

(2)、执行的时候，遇到锁，如表锁、行锁。

2、这条 SQL 语句一直执行的很慢，则有如下原因。

(1)、没有用上索引：例如该字段没有索引；由于对字段进行运算、函数操作导致无法用索引。

(2)、数据库选错了索引。

## SQL主从复制

---

**原理：**

(1) master服务器将数据的改变记录二进制binlog日志，当master上的数据发生改变时，则将其改变写入二进制日志中；

(2) slave服务器会在一定时间间隔内对master二进制日志进行探测其是否发生改变，如果发生改变，则开始一个I/OThread请求master二进制事件

(3) 同时主节点为每个I/O线程启动一个dump线程，用于向其发送二进制事件，并保存至从节点本地的中继日志中，从节点将启动SQL线程从中继日志中读取二进制日志，在本地重放，使得其数据和主节点的保持一致，最后I/OThread和SQLThread将进入睡眠状态，等待下一次被唤醒。

**也就是说：**

- 从库会生成两个线程,一个I/O线程,一个SQL线程;
- I/O线程会去请求主库的binlog,并将得到的binlog写到本地的relay-log(中继日志)文件中;
- 主库会生成一个log dump线程,用来给从库I/O线程传binlog;
- SQL线程,会读取relay log文件中的日志,并解析成sql语句逐一执行;

**注意：**

- 1--master将操作语句记录到binlog日志中，然后授予slave远程连接的权限（master一定要开启binlog二进制日志功能；通常为了数据安全考虑，slave也开启binlog功能）。
- 2--slave开启两个线程：IO线程和SQL线程。其中：IO线程负责读取master的binlog内容到中继日志relay log里；SQL线程负责从relay log日志里读出binlog内容，并更新到slave的数据库里，这样就能保证slave数据和master数据保持一致了。
- 3--Mysql复制至少需要两个Mysql的服务，当然Mysql服务可以分布在不同的服务器上，也可以在一台服务器上启动多个服务。
- 4--Mysql复制最好确保master和slave服务器上的Mysql版本相同（如果不能满足版本一致，那么要保证master主节点的版本低于slave从节点的版本）
- 5--master和slave两节点间时间需同步

#### 具体步骤：

- 1、从库通过手工执行change master to 语句连接主库，提供了连接的用户一切条件（user、password、port、ip），并且让从库知道，二进制日志的起点位置（file名 position 号）； start slave
- 2、从库的IO线程和主库的dump线程建立连接。
- 3、从库根据change master to 语句提供的file名和position号，IO线程向主库发起binlog的请求。
- 4、主库dump线程根据从库的请求，将本地binlog以events的方式发给从库IO线程。
- 5、从库IO线程接收binlog events，并存放本地relay-log中，传送过来的信息，会记录到master.info中
- 6、从库SQL线程应用relay-log，并且把应用过的记录到relay-log.info中，默认情况下，已经应用过的relay 会自动被清理purge

## SQL读写分离

MySQL读写分离基本原理是让master数据库处理写操作，slave数据库处理读操作。master将写操作的变更同步到各个slave节点。

MySQL读写分离能提高系统性能的原因在于：

- 1、物理服务器增加，机器处理能力提升。拿硬件换性能。
- 2、主从只负责各自的读和写，极大程度缓解X锁和S锁争用。
- 3、slave可以配置myiasm引擎，提升查询性能以及节约系统开销。
- 4、master直接写是并发的，slave通过主库发送来的binlog恢复数据是异步。
- 5、slave可以单独设置一些参数来提升其读的性能。
- 6、增加冗余，提高可用性。

#### 配置

##### 1、硬件配置

```
master 192.168.85.11
slave  192.168.85.12
proxy  192, 168.85.14
```

- 2、首先在master和slave上配置主从复制
- 3、进行proxy的相关配置

```
#1、下载mysql-proxy
https://downloads.mysql.com/archives/proxy/#downloads
#2、上传软件到proxy的机器
```

直接通过xftp进行上传

#3、解压安装包

```
tar -zxvf mysql-proxy-0.8.5-linux-glibc2.3-x86-64bit.tar.gz
```

#4、修改解压后的目录

```
mv mysql-proxy-0.8.5-linux-glibc2.3-x86-64bit mysql-proxy
```

#5、进入mysql-proxy的目录

```
cd mysql-proxy
```

#6、创建目录

```
mkdir conf
```

```
mkdir logs
```

#7、添加环境变量

#打开/etc/profile文件

```
vi /etc/profile
```

#在文件的最后面添加一下命令

```
export PATH=$PATH:/root/mysql-proxy/bin
```

#8、执行命令让环境变量生效

```
source /etc/profile
```

#9、进入conf目录，创建文件并添加一下内容

```
vi mysql-proxy.conf
```

添加内容

```
[mysql-proxy]
```

```
user=root
```

```
proxy-address=192.168.85.14:4040
```

```
proxy-backend-addresses=192.168.85.11:3306
```

```
proxy-read-only-backend-addresses=192.168.85.12:3306
```

```
proxy-lua-script=/root/mysql-proxy/share/doc/mysql-proxy/rw-splitting.lua
```

```
log-file=/root/mysql-proxy/logs/mysql-proxy.log
```

```
log-level=debug
```

```
daemon=true
```

#10、开启mysql-proxy

```
mysql-proxy --defaults-file=/root/mysql-proxy/conf/mysql-proxy.conf
```

#11、查看是否安装成功，打开日志文件

```
cd /root/mysql-proxy/logs
```

```
tail -100 mysql-proxy.log
```

#内容如下：表示安装成功

```
2019-10-11 21:49:41: (debug) max open file-descriptors = 1024
```

```
2019-10-11 21:49:41: (message) proxy listening on port 192.168.85.14:4040
```

```
2019-10-11 21:49:41: (message) added read/write backend: 192.168.85.11:3306
```

```
2019-10-11 21:49:41: (message) added read-only backend: 192.168.85.12:3306
```

```
2019-10-11 21:49:41: (debug) now running as user: root (0/0)
```

#### 4、进行连接

#mysql的命令行会出现无法连接的情况，所以建议使用客户端

```
mysql -uroot -p123 -h192.168.85.14 -P 4040
```

## 3. Redis

### redis常见的数据结构以及应用场景：

String：key-value缓存应用，最常规的set/get操作，value可以是String也可以是数字。一般做一些复杂的计数功能的缓存。

Hash: field-value映射表, 存储用户信息和商品信息

List: list分页查询

Set:实现差, 并, 交集操作, 比如共同喜好等

Sorted set: 用户列表, 礼物排行榜, 弹幕消息

## 缓存雪崩

缓存同一时间大面积失效, 所有请求都落到数据库造成短时间内承受大量请求而崩掉

## 如何解决缓存雪崩?

在缓存的时候给过期时间加上一个随机值, 这样就会大幅度的减少缓存在同一时间过期。

对于“Redis挂掉了, 请求全部走数据库”这种情况, 我们可以有以下的思路:

事发前: 实现Redis的高可用(主从架构+Sentinel 或者Redis Cluster), 尽量避免Redis挂掉这种情况发生。

事发中: **\*设置本地缓存\*(ehcache)+\*限流\*(hystrix)**, 尽量避免我们的数据库\*\*\*掉(起码能保证我们的服务还是能正常工作的)

事发后: **\*redis持久化\***, 重启后自动从磁盘上加载数据, 快速恢复缓存数据。

## 缓存穿透

恶意请求缓存中不存在的数据, 所有请求都落到数据库造成短时间内承受大量请求而崩掉

## 如何解决缓存穿透?

(一)**\*利用互斥锁\***, 缓存失效的时候, 先去获得锁, 得到锁了, 再去请求数据库。没得到锁, 则休眠一段时间重试, 互斥锁可以避免某一个热点数据失效导致数据库崩溃的问题, 而在实际业务中, 往往会存在一批热点数据同时失效的场景

(二)**\*采用异步更新策略, 无论key是否取到值, 都直接返回\***。value值中维护一个缓存失效时间, 缓存如果过期, 异步起一个线程去读数据库, 更新缓存。需要做缓存预热(项目启动前, 先加载缓存)操作。

(三)提供一个能迅速判断请求是否有效的拦截机制, 比如, **\*利用布隆过滤器, 内部维护一系列合法有效的key\***。迅速判断出, 请求所携带的Key是否合法有效。如果不合法, 则直接返回。

(四)使用空对象

## Redis和数据库双写一致性?

四种策略

**先删缓存, 再更新数据库**

该方案在线程A进行数据**更新**操作, 线程B进行**查询**操作时, 有可能出现下面的情况导致数据不一致:

1. 线程A删除缓存
2. 线程B查询数据, 发现缓存数据不存在
3. 线程B查询数据库, 得到旧值, 写入缓存
4. 线程A将新值更新到数据库

这样一来, 缓存中的数据仍然是旧值

如果线程B执行的是更新操作，线程B查询得到的是旧值，A更新到数据库新值，然后B基于旧值计算写入了计算后的值，A的更新操作被抹去了，这种情况下属于 **更新数据事务原子性**问题，需要用分布式锁来解决。

### 先更新数据库，再删缓存

当缓存失效时，线程B原子性被破坏时会出现不一致问题：

1. 缓存失效了
2. 线程B从数据库读取旧值
3. 线程A从数据库读取旧值
4. 线程B将新值更新到数据库
5. 线程B删除缓存
6. 线程A将旧值写入缓存

这种情况概率很低，实际上数据库的写操作会比读操作慢得多，**读操作必需在写操作前进入数据库操作，而又要晚于写操作更新缓存**，这种情况下只需要线程B延时删除缓存就好。另外在数据库主从同步的情况下，延时删除还能防止数据更新还未从主数据库同步到从数据库的情况。

### 缓存延时双删，更新前先删除缓存，然后更新数据，再延时删除缓存

延时双删即先删除缓存，然后更新数据，再延时n ms后删除缓存，这个我认为作用和**更新数据库再删除缓存的策略**几乎是等同的（欢迎讨论）

之所以设计为延时双删的目的在于当最后一次延时**删除缓存失败**的情况发生，至少一致性策略只会**退化**成**先删缓存再更新数据**的策略。

删除缓存失败这种事情个人认为在生产环境缓存高可用的情况下几乎不会出现，且这种情况如果发生了，不如考虑一下重试机制。

### 监听MySQL binlog进行缓存更新

首先，采取正确更新策略，**\*先更新数据库，再删缓存\***。其次，因为可能存在删除缓存失败的问题，提供一个补偿措施即可，例如利用消息队列。

#### **\*采用延时双删策略\***

- (1) 先淘汰缓存
- (2) 再写数据库（这两步和原来一样）
- (3) 休眠1秒，再次淘汰缓存

## Redis的过期策略

Redis是使用定期删除+惰性删除两者配合的过期策略。

### 定期删除

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，检测这些key是否过期，如果过期了就将其删掉。

因为key太多，如果全盘扫描所有的key会非常耗性能，所以是随机抽取一些key来删除。这样就有可能删除不完，需要惰性删除配合。

### 惰性删除

惰性删除不再是Redis去主动删除，而是在客户端要获取某个key的时候，Redis会先去检测一下这个key是否已经过期，如果没有过期则返回给客户端，如果已经过期了，那么Redis会删除这个key，不会返回给客户端。

所以惰性删除可以解决一些过期了，但没被定期删除随机抽取到的key。但有些过期的key既没有被随机抽取，也没有被客户端访问，就会一直保留在数据库，占用内存，长期下去可能会导致内存耗尽。所以Redis提供了内存淘汰机制来解决这个问题。



# Redis的内存淘汰机制：

---

Redis提供了8种内存淘汰策略

noeviction：当内存不足以容纳新写入数据时，新写入操作会报错。默认策略

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。

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。

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。

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。

如何选取合适的策略？比较推荐的是两种lru策略。根据自己的业务需求。如果你使用Redis只是作为缓存，不作为DB持久化，那推荐选择allkeys-lru；如果你使用Redis同时用于缓存和数据持久化，那推荐选择volatile-lru。

## 持久化如何处理过期

---

RDB

从内存数据库持久化数据到RDB文件：持久化key之前，会检查是否过期，过期的key不进入RDB文件

从RDB文件恢复数据到内存数据库：数据载入数据库之前，会对key先进行过期检查，如果过期，不导入数据库（主库情况）。

AOF

从内存数据库持久化数据到AOF文件：当key过期后，还没有被删除，此时进行执行持久化操作（该key是不会进入aof文件的，因为没有发生修改命令）

当key过期后，在发生删除操作时，程序会向aof文件追加一条del命令（在将来的以aof文件恢复数据的时候该过期的键就会被删掉）

AOF重写：重写时，会先判断key是否过期，已过期的key不会重写到aof文件

## 如何解决 Redis 的并发竞争 Key 问题

---

所谓 Redis 的并发竞争 Key 的问题也就是多个系统同时对一个 key 进行操作，最后执行的顺序和我们期望的顺序不同，导致了结果的不同！推荐一种方案：分布式锁（zookeeper 和 redis 都可以实现分布式锁）。

基于zookeeper临时有序节点可以实现的分布式锁。大致思想为：每个客户端对某个方法加锁时，在zookeeper上的与该方法对应的指定节点的目录下，生成一个唯一的瞬时有序节点。判断是否获取锁的方式很简单，只需要判断有序节点中序号最小的一个。当释放锁的时候，只需将这个瞬时节点删除即可。

## redis 持久化机制(怎么保证 redis 挂掉之后再重启数据可以进行恢复)

---

### \*1、快照 (snapshotting) 持久化 (RDB) \*

Redis可以通过创建快照来获得存储在内存里面的数据在某个时间点上的副本。Redis创建快照之后，可以对快照进行备份，可以将快照复制到其他服务器从而创建具有相同数据的服务器副本（Redis主从结构，主要用来提高Redis性能），还可以将快照留在原地以便重启服务器的时候使用。快照持久化是Redis默认采用的持久化方式，在redis.conf配置文件中默认有此下配置

### \*2、AOF (append-only file) 持久化\*

与快照持久化相比，AOF持久化 的实时性更好，因此已成为主流的持久化方案。默认情况下Redis没有开启AOF方式的持久化，可以通过appendonly参数开启.Redis 4.0 开始支持 RDB 和 AOF 的混合持久化（默认关闭，可以通过配置项 aof-use-rdb-preamble 开启）。

## 为什么 redis 读写速率快、性能好？

Redis是**\*纯内存数据库\***，相对于读写磁盘，读写内存的速度就不是几倍几十倍了，一般hash查找可以达到每秒百万次的数量级。

**\*多路复用IO\***，“多路”指的是多个网络连接，“复用”指的是复用同一个线程。采用多路 I/O 复用技术可以让单个线程高效的处理多个连接请求（尽量减少网络IO的时间消耗）。可以直接理解为：单线程的原子操作，避免上下文切换的时间和性能消耗；加上对内存中数据的处理速度，很自然的提高redis的吞吐量。

## 你了解最经典的KV、DB读写模式么？

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。

## 3.操作系统

### Fork

创建一个新进程：fork()函数

创建过程：

- (1) 给新进程分配一个标识符
- (2) 在内核中分配一个PCB,将其挂在PCB表上
- (3) 复制它的父进程的环境（PCB中大部分的内容）
- (4) 为其分配资源（程序、数据、栈等）
- (5) 复制父进程地址空间里的内容（代码共享，数据写时拷贝）
- (6) 将进程置成就绪状态，并将其放入就绪队列，等待CPU调度。

关于fork()函数：

调用一次，返回二次。一旦调用fork()函数，会生成一份与源文件一模一样的程序，这就是子进程，当然了，如果子

进程创建成功，父进程的fork()函数返回子进程的pid号码（大于0），这时，子进程的fork()函数返回0；但是如果子进程创建

失败，父进程的fork()函数返回-1，这时子进程都没有创建，当然也就没有返回了。

此外，当子进程创建成功之后，父、子进程是2个独立的进程，至于谁先接着下一步执行，这时不确定的。

## 多线程和多进程的区别？

线程是可以共享一些资源的，比如堆和方法区，进程是不共享资源的。

线程是分配资源的最小单位，进程是程序运行的基本单位。

 image-20200813173358322

## 死锁的四个条件

- (1) 互斥条件：一个资源每次只能被一个进程使用。
- (2) 占有且等待：一个进程因请求资源而阻塞时，对已获得的资源保持不放。
- (3) 不可强行占有：进程已获得的资源，在未使用完之前，不能强行剥夺。
- (4) 循环等待条件：若干进程之间形成一种头尾相接的循环等待资源关系。

## CAS算法以及问题（ABA）

算法涉及到三个操作数：

- 1.需要读写的内存位置V
- 2.需要进行比较的预期值A
- 3.需要写入的新值U

CAS具体执行时，当且仅当预期值A符合内存地址V中存储的值时，就用新值U替换掉旧值，并写入到内存地址V中。否则不做更新。

```
public boolean compareAndSwap(int value, int expect, int update) {  
    // 如果内存中的值value和期望值expect一样 则将值更新为新值update  
    if (value == expect) {  
        value = update;  
        return true;  
    } else {  
        return false;  
    }  
}
```

### ABA问题

例如有2个线程同时对同一个值(初始值为A)进行CAS操作，这三个线程如下

1. 线程1，期望值为A，欲更新的值为B
2. 线程2，期望值为A，欲更新的值为B

线程1抢先获得CPU时间片，而线程2因为其他原因阻塞了，线程1取值与期望的A值比较，发现相等然后将值更新为B，然后这个时候出现了线程3，期望值为B，欲更新的值为A，线程3取值与期望的值B比较，发现相等则将值更新为A，此时线程2从阻塞中恢复，并且获得了CPU时间片，这时候线程2取值与期望的值A比较，发现相等则将值更新为B，虽然线程2也完成了操作，但是线程2并不知道值已经经过了A->B->A的变化过程。

ABA问题带来的危害：

小明在提款机，提取了50元，因为提款机问题，有两个线程，同时把余额从100变为50

线程1（提款机）：获取当前值100，期望更新为50，

线程2（提款机）：获取当前值100，期望更新为50，

线程1成功执行，线程2某种原因block了，这时，某人给小明汇款50

线程3（默认）：获取当前值50，期望更新为100，

这时候线程3成功执行，余额变为100，

线程2从Block中恢复，获取到的也是100，compare之后，继续更新余额为50！！

此时可以看到，实际余额应该为100（100-50+50），但是实际上变为了50（100-50+50-50）这就是ABA问题带来的成功提交。

解决方法：在变量前面加上版本号，每次变量更新的时候变量的版本号都+1，即A->B->A就变成了1A->2B->3A。

循环时间长开销大

如果 CAS 操作失败，就需要循环进行 CAS 操作(循环同时将期望值更新为最新的)，如果长时间都不成功的话，那么会造成CPU极大的开销。

这种循环也称为自旋

解决方法： 限制自旋次数，防止进入死循环。

### 只能保证一个变量的原子操作

CAS 的原子操作只能针对一个共享变量。

解决方法： 如果需要对多个共享变量进行操作，可以使用加锁方式(悲观锁)保证原子性，或者可以把多个共享变量合并成一个共享变量进行 CAS 操作。

## 自旋锁

不放弃CPU资源，避免了用户态内核态的时间开销。

## 进程的调度算法

### 1、先来先服务调度算法

先来先服务(FCFS)调度算法是一种最简单的调度算法，该算法既可用于作业调度，也可用于进程调度。当在作业调度中采用该算法时，每次调度都是从后备作业队列中选择一个或多个最先进入该队列的作业，将它们调入内存，为它们分配资源、创建进程，然后放入就绪队列。在进程调度中采用FCFS算法时，则每次调度是从就绪队列中选择一个最先进入该队列的进程，为之分配处理机，使之投入运行。该进程一直运行到完成或发生某事件而阻塞后才放弃处理机。

### 2、短作业(进程)优先调度算法

短作业(进程)优先调度算法，是指对短作业或短进程优先调度的算法。它们可以分别用于作业调度和进程调度。短作业优先(SJF)的调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。而短进程优先(SPF)调度算法则是从就绪队列中选出一个估计运行时间最短的进程，将处理机分配给它，使它立即执行并一直执行到完成，或发生某事件而被阻塞放弃处理机时再重新调度。

### 3、时间片轮转法

在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把CPU分配给队首进程，并令其执行一个时间片。时间片的大小从几ms到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在一给定的时间内均能获得一时间片的处理机执行时间。换言之，系统能在给定的时间内响应所有用户的请求。

### 4、多级反馈队列调度算法

前面介绍的各种用作进程调度的算法都有一定的局限性。如短进程优先的调度算法，仅照顾了短进程而忽略了长进程，而且如果并未指明进程的长度，则短进程优先和基于进程长度的抢占式调度算法都将无法使用。而多级反馈队列调度算法则不必事先知道各种进程所需的执行时间，而且还可以满足各种类型进程的需要，因而它是目前被公认的一种较好的进程调度算法。在采用多级反馈队列调度算法的系统中，调度算法的实施过程如下所述：

1) 应设置多个就绪队列, 并为各个队列赋予不同的优先级。第一个队列的优先级最高, 第二个队列次之, 其余各队列的优先权逐个降低。该算法赋予各个队列中进程执行时间片的大小也各不相同, 在优先权愈高的队列中, 为每个进程所规定的执行时间片就愈小。例如, 第二个队列的时间片要比第一个队列的时间片长一倍, 第 $i+1$ 个队列的时间片要比第 $i$ 个队列的时间片长一倍。

2) 当一个新进程进入内存后, 首先将它放入第一队列的末尾, 按FCFS原则排队等待调度。当轮到该进程执行时, 如它能在该时间片内完成, 便可准备撤离系统; 如果它在一个时间片结束时尚未完成, 调度程序便将该进程转入第二队列的末尾, 再同样地按FCFS原则等待调度执行; 如果它在第二队列中运行一个时间片后仍未完成, 再依次将它放入第三队列, …… , 如此下去, 当一个长作业(进程)从第一队列依次降到第 $n$ 队列后, 在第 $n$ 队列便采取按时间片轮转的方式运行。

3) 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 $1 \sim (i-1)$ 队列均空时, 才会调度第 $i$ 队列中的进程运行。如果处理机正在第 $i$ 队列中为某进程服务时, 又有新进程进入优先权较高的队列(第 $1 \sim (i-1)$ 中的任何一个队列), 则此时新进程将抢占正在运行进程的处理机, 即第 $i$ 队列中某个正在运行的进程的时间片用完后, 由调度程序选择优先权较高的队列中的那一个进程, 把处理机分配给它。

## 5、优先权调度算法

为了照顾紧迫型作业, 使之在进入系统后便获得优先处理, 引入了最高优先权优先(FPF)调度算法。此算法常被用于批处理系统中, 作为作业调度算法, 也作为多种操作系统中的进程调度算法, 还可用于实时系统中。当把该算法用于作业调度时, 系统将从后备队列中选择若干个优先权最高的作业装入内存。当用于进程调度时, 该算法是把处理机分配给就绪队列中优先权最高的进程, 这时, 又可进一步把该算法分成如下两种。

### 1) 非抢占式优先权算法

在这种方式下, 系统一旦把处理机分配给就绪队列中优先权最高的进程后, 该进程便一直执行下去, 直至完成; 或因发生某事件使该进程放弃处理机时, 系统方可再将处理机重新分配给另一优先权最高的进程。这种调度算法主要用于批处理系统中; 也可用于某些对实时性要求不严的实时系统中。

### 2) 抢占式优先权调度算法

在这种方式下, 系统同样也是把处理机分配给优先权最高的进程, 使之执行。但在其执行期间, 只要又出现了另一个其优先权更高的进程, 进程调度程序就立即停止当前进程(原优先权最高的进程)的执行, 重新将处理机分配给新到的优先权最高的进程。因此, 在采用这种调度算法时, 是每当系统中出现一个新的就绪进程 $i$ 时, 就将其优先权 $P_i$ 与正在执行的进程 $j$ 的优先权 $P_j$ 进行比较。如果 $P_i \leq P_j$ , 原进程 $P_j$ 便继续执行; 但如果是 $P_i > P_j$ , 则立即停止 $P_j$ 的执行, 做进程切换, 使 $i$ 进程投入执行。显然, 这种抢占式的优先权调度算法能更好地满足紧迫作业的要求, 故而常用于要求比较严格的实时系统中, 以及对性能要求较高的批处理和分时系统中。

# 进程间通信和线程间通信的几种方式? 为什么要这么设计?

管道、有名管道、信号、信号量、消息队列、共享内存、套接字。

**管道( pipe ):** 管道是一种半双工的通信方式, 数据只能单向流动, 而且只能在具有亲缘关系的进程间使用。进程的亲缘关系通常是指父子进程关系。

**有名管道( named pipe ):** 有名管道也是半双工的通信方式, 但是它允许无亲缘关系进程间的通信。

**信号( sinal ):** 信号是一种比较复杂的通信方式, 用于通知接收进程某个事件已经发生。信号是进程间通信机制中唯一的异步通信机制, 可以看作是异步通知

**信号量(semaphore ):** 信号量是一个计数器, 可以用来控制多个进程对共享资源的访问。它常作为一种锁机制, 防止某进程正在访问共享资源时, 其他进程也访问该资源。因此, 主要作为进程间以及同一进程内不同线程之间的同步手段。

**消息队列( messagequeue ):** 消息队列是由消息的链表, 存放在内核中并由消息队列标识符标识。消息队列克服了信号传递信息少、管道只能承载无格式字节流以及缓冲区大小受限等缺点。

**共享内存(shared memory)**：共享内存就是映射一段能被其他进程所访问的内存，这段共享内存由一个进程创建，但多个进程都可以访问。共享内存是最快的 IPC 方式，它是针对其他进程间通信方式运行效率低而专门设计的。它往往与其他通信机制，如信号两，配合使用，来实现进程间的同步和通信。

**套接字(socket)**：套接口也是一种进程间通信机制，与其他通信机制不同的是，它实现了实现了网络通信。

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[1]：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。

## 如何实现共享内存？

要使用一块[共享内存](#)，进程必须首先分配它。随后需要访问这个共享内存块的每一个进程都必须将这个共享内存绑定到自己的[地址空间](#)中。当完成通信之后，所有进程都将脱离共享内存，并且由一个进程释放该共享内存块。

理解 Linux [系统内存](#)模型可以有助于解释这个绑定的过程。在 Linux 系统中，每个进程的[虚拟内存](#)是被分为许多页面的。这些内存页面中包含了实际的数据。每个进程都会维护一个从[内存地址](#)到虚拟内存页面之间的映射关系。尽管每个进程都有自己的内存地址，不同的进程可以同时将同一个内存页面映射到自己的地址空间中，从而达到共享内存的目的。

分配一个新的共享内存块会创建新的内存页面。因为所有进程都希望共享对同一块内存的访问，只应由一个进程创建一块新的[共享内存](#)。再次分配一块已经存在的内存块不会创建新的页面，而只是会返回一个标识该内存块的[标识符](#)。一个进程如需使用这个共享内存块，则首先需要将它绑定到自己的[地址空间](#)中。这样会创建一个从进程本身[虚拟地址](#)到共享页面的映射关系。当对共享内存的使用结束之后，这个映射关系将被删除。当再也没有进程需要使用这个共享内存块的时候，必须有一个（且只能是一个）进程负责释放这个被共享的内存页面。

所有共享内存块的大小都必须是系统页面大小的整数倍。系统页面大小指的是系统中单个内存页面包含的字节数。在 Linux 系统中，内存页面大小是4KB，不过您仍然应该通过调用 [getpagesize](#) 获取这个值。

(1) 进程通过调用[shmget](#) (Shared Memory GET，获取[共享内存](#)) 来分配一个共享内存块。

(2) 要让一个进程获取对一块[共享内存](#)的访问，这个进程必须先调用 shmat (SHared Memory Attach，绑定到共享内存)。将 shmget 返回的共享内存[标识符](#) SHMID 传递给这个函数作为第一个参数。该函数的第二个参数是一个[指针](#)，指向您希望用于映射该共享内存块的进程[内存地址](#)；如果您指定 NULL则Linux会自动选择一个合适的地址用于映射。

(3) 调用 shmctl ("Shared Memory Control", 控制[共享内存](#)) 函数会返回一个共享内存块的相关信息。要删除一个共享内存块，则应将 IPC\_RMID 作为第二个参数，而将 NULL 作为第三个参数。当最后一个绑定该共享内存块的进程与其脱离时，该共享内存块将被删除。

## 线程间的通信方式

1. 锁机制：包括互斥锁、条件变量、读写锁

互斥锁提供了以排他方式防止数据结构被并发修改的方法。

读写锁允许多个线程同时读共享数据，而对写操作是互斥的。

2. 条件变量可以以原子的方式阻塞进程，直到某个特定条件为真为止。对条件的测试是在互斥锁的保护下进行的。条件变量始终与互斥锁一起使用。



3. 信号量机制(Semaphore): 包括无名线程信号量和命名线程信号量  
信号机制(Signal): 类似进程间的信号处理

线程间的通信目的主要是用于线程同步, 所以线程没有像进程通信中的用于数据交换的通信机制。

## 用户态和内核态? 如何转换的?

内核态: 当一个任务(进程)执行系统调用而陷入内核代码中执行时, 我们就称进程处于内核运行态(或简称为内核态)。其他的属于用户态。用户程序运行在用户态, 操作系统运行在内核态。(操作系统内核运行在内核态, 而服务器运行在用户态)。用户态不能干扰内核态, 所以CPU指令就有两种, 特权指令和非特权指令。不同的状态对应不同的指令。特权指令: 只能由操作系统内核部分使用, 不允许用户直接使用的指令。**如, I/O指令、置终端屏蔽指令、清内存、建存储保护、设置时钟指令(这几种记好, 属于内核态)**。非特权指令: 所有程序均可直接使用。

所以:

系统态(核心态、特态、管态): 执行全部指令。

用户态(常态、目态): 执行非特权指令。

### 1) 用户态切换到内核态的3种方式

#### a. 系统调用

这是用户态进程主动要求切换到内核态的一种方式, 用户态进程通过系统调用申请使用操作系统提供的服务程序完成工作, 比如前例中fork()实际上就是执行了一个创建新进程的系统调用。而系统调用的机制其核心还是使用了操作系统为用户特别开放的一个中断来实现, 例如Linux的int 80h中断。

#### b. 异常

当CPU在执行运行在用户态下的程序时, 发生了某些事先不可知的异常, 这时会触发由当前运行进程切换到处理此异常的内核相关程序中, 也就转到了内核态, 比如缺页异常。

#### c. 外围设备的中断

当外围设备完成用户请求的操作后, 会向CPU发出相应的中断信号, 这时CPU会暂停执行下一条即将要执行的指令转而去执行与中断信号对应的处理程序, 如果先前执行的指令是用户态下的程序, 那么这个转换的过程自然也就发生了由用户态到内核态的切换。比如硬盘读写操作完成, 系统会切换到硬盘读写的中断处理程序中执行后续操作等。

## 用户态切换内核态耗时的原因



## 为什么指令重排序? 从CPU角度

首先可能充分利用寄存器, 减少内存器装填数值的时间, 比如

```
int a = 1;
int b = 1;
a = a + 1;
b = b + 1;
```

就可能没有

```
int a = 1;
a = a + 1;
int b = 1;
b = b + 1 ;
```

性能好，因为后者可以 a或b可能在寄存器中了。

其次，因为一个汇编指令也会涉及到很多步骤，每个步骤可能会用到不同的寄存器，CPU使用了流水线技术，也就是说，CPU有多个功能单元（如获取、解码、运算和结果），一条指令也分为多个单元，那么第一条指令执行还没完毕，就可以执行第二条指令，前提是这两条指令功能单元相同或类似，所以一般可以通过指令重排使得具有相似功能单元的指令接连执行来减少流水线中断的情况。

## 4.计算机网络

### 说一下UDP

UDP 的全称是 `User Datagram Protocol`，用户数据报协议。它不需要所谓的 `握手` 操作，从而加快了通信速度，允许网络上的其他主机在接收方同意通信之前进行数据传输。

数据报是与分组交换网络关联的传输单元。

UDP 的特点主要有

- UDP 能够支持容忍数据包丢失的带宽密集型应用程序
- UDP 具有低延迟的特点
- UDP 能够发送大量的数据包
- UDP 能够允许 DNS 查找，DNS 是建立在 UDP 之上的应用层协议。

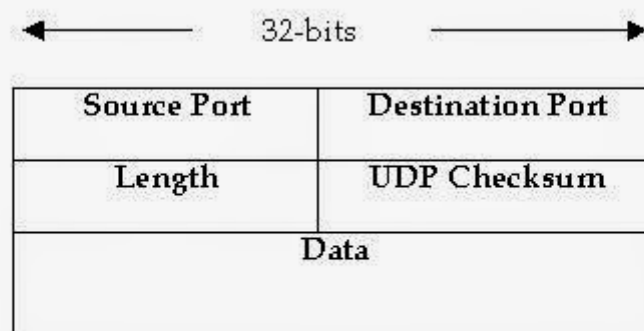


Figure 8: UDP segment structure

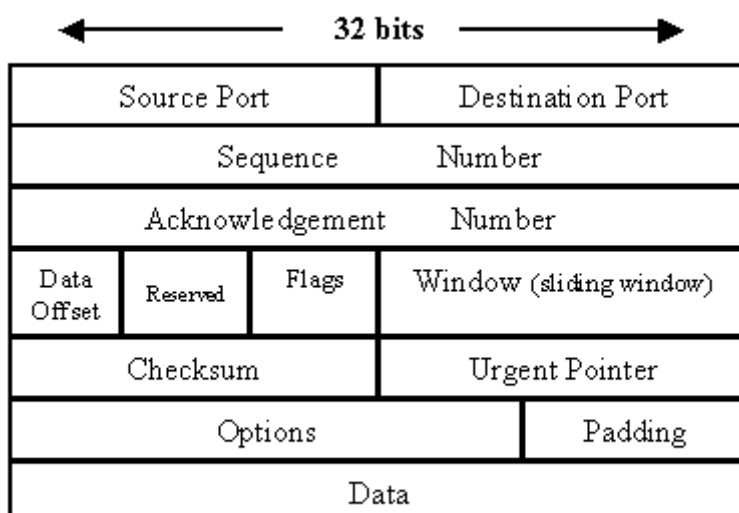
### 说一下TCP

TCP 的全称是 `Transmission Control Protocol`，传输控制协议。它能够帮助你确定计算机连接到 Internet 以及它们之间的数据传输。通过三次握手来建立 TCP 连接，三次握手就是用来启动和确认 TCP 连接的过程。一旦连接建立后，就可以发送数据了，当数据传输完成后，会通过关闭虚拟电路来断开连接。

- TCP 能够确保连接的建立和数据包的发送
- TCP 支持错误重传机制



- TCP 支持拥塞控制，能够在网络拥堵的情况下延迟发送
- TCP 能够提供错误校验和，甄别有害的数据包。



## TCP的保护边界和流

保护消息边界，就是指传输协议把数据当作一条独立的消息在网上传输，接收端只能接收独立的消息。也就是说存在保护消息边界，接收端一次只能接收发送端发出的一个数据包。而面向流则是指无保护消息边界的，如果发送端连续发送数据，接收端有可能在一次接收动作中，会接收两个或者更多的数据包。

例如，我们连续发送三个数据包，大小分别是2k，4k，8k,这三个数据包，都已经到达了接收端的网络堆栈中，如果使用**UDP**协议，不管我们使用多大的接收缓冲区去接收数据，我们必须有三次接收动作，才能够把所有的数据包接收完.而使用**TCP**协议，我们只要把接收的\*缓冲区大小设置在14k\*以上，我们就能够一次把所有的数据包接收下来，只需要有一次接收动作。

## TCP的粘包

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、数据量小的数据包，合并成一个大的数据包发送(把发送端的缓冲区填满一次性发送)。

2接收端底层会把tcp段整理排序交给缓冲区，这样接收端应用程序从缓冲区取数据就只能得到整体数据而不知道怎么拆分

这两个原因是tcp是面向流的协议

比如发送端发送了一个由2个100字节组成的200字节的数据包到接受端的缓冲区，接受端从缓冲去一次取80字节的数据，那么第一次取的就是一个不完整的数据包，第二次取就会带上第一个数据包的尾部和下一个数据包的头

**出现的原因：**

简单得说，在流传输中出现，UDP不会出现粘包，因为它有消息边界

1发送端需要等缓冲区满才发送出去，造成粘包

2接收方不及时接收缓冲区的包，造成多个包接收

具体点：

(1) 发送方引起的粘包是由TCP协议本身造成的，TCP为提高传输效率，发送方往往要收集到足够多的数据后才发送一包数据。若连续几次发送的数据都很少，通常TCP会根据优化算法把这些数据合成一包后一次发送出去，这样接收方就收到了粘包数据。

(2) 接收方引起的粘包是由于接收方用户进程不及时接收数据，从而导致粘包现象。这是因为接收方先把收到的数据放在系统接收缓冲区，用户进程从该缓冲区取数据，若下一包数据到达时前一包数据尚未被用户进程取走，则下一包数据放到系统接收缓冲区时就接到前一包数据之后，而用户进程根据预先设定的缓冲区大小从系统接收缓冲区取数据，这样就一次取到了多包数据。

粘包情况有两种，一种是粘在一起的包都是完整的数据包，另一种情况是粘在一起的包有不完整的包。

不是所有的粘包现象都需要处理，若传输的数据为不带结构的连续流数据（如文件传输），则不必把粘连的包分开（简称分包）。但在实际工程应用中，传输的数据一般为带结构的数据，这时就需要做分包处理。

在处理定长结构数据的粘包问题时，分包算法比较简单；在处理不定长结构数据的粘包问题时，分包算法就比较复杂。特别是粘在一起的包有不完整的包的粘包情况，由于一包数据内容被分在了两个连续的接收包中，处理起来难度较大。实际工程应用中应尽量避免出现粘包现象。

### 采取措施

(1) 对于发送方引起的粘包现象，用户可通过编程设置来避免，TCP提供了强制数据立即传送的操作指令push，TCP软件收到该操作指令后，就立即将本段数据发送出去，而不必等待发送缓冲区满；

(2) 对于接收方引起的粘包，则可通过优化程序设计、精简接收进程工作量、提高接收进程优先级等措施，使其及时接收数据，从而尽量避免出现粘包现象；

(3) 由接收方控制，将一包数据按结构字段，人为控制分多次接收，然后合并，通过这种手段来避免粘包。

## TCP和UDP的区别

- TCP 是面向连接的协议。UDP 是无连接的协议
- TCP 在发送数据前需要先建立连接，然后再发送数据。UDP 无需建立连接就可以直接发送大量数据
- TCP 会按照特定顺序重新排列数据包。UDP 数据包没有固定顺序，所有数据包都相互独立
- TCP 传输的速度比较慢。UDP 的传输会更快
- TCP 的头部字节有 20 字节。UDP 的头部字节只需要 8 个字节
- TCP 是重量级的，在发送任何用户数据之前，TCP需要三次握手建立连接。UDP 是轻量级的。没有跟踪连接，消息排序等。
- TCP 会进行错误校验，并能够进行错误恢复。UDP 也会错误检查，但会丢弃错误的数据包。
- TCP 有发送确认。UDP 没有发送确认
- TCP 会使用握手协议，例如 SYN, SYN-ACK, ACK。UDP无握手协议
- TCP 是可靠的，因为它可以确保将数据传送到路由器。UDP 中不能保证将数据传送到目标。

## 运行在TCP 或UDP的应用层协议分析

运行在TCP协议上的协议：

- HTTP (Hypertext Transfer Protocol, 超文本传输协议)，主要用于普通浏览。
- HTTPS (HTTP over SSL, 安全超文本传输协议),HTTP协议的安全版本。
- FTP (File Transfer Protocol, 文件传输协议)，用于文件传输。
- POP3 (Post Office Protocol, version 3, 邮局协议)，收邮件用。
- SMTP (Simple Mail Transfer Protocol, 简单邮件传输协议)，用来发送电子邮件。
- TELNET (Teletype over the Network, 网络电传)，通过一个终端 (terminal) 登陆到网络。
- SSH (Secure Shell, 用于替代安全性差的TELNET)，用于加密安全登陆用。

运行在UDP协议上的协议：

- BOOTP (Boot Protocol, 启动协议)，应用于无盘设备。
- NTP (Network Time Protocol, 网络时间协议)，用于网络同步。
- DHCP (Dynamic Host Configuration Protocol, 动态主机配置协议)，动态配置IP地址。

运行在TCP和UDP协议上：

- DNS (Domain Name Service, 域名服务)，用于完成地址查找，邮件转发等工作。
- ECHO (Echo Protocol, 回绕协议)，用于查错及测量应答时间（运行在TCP和UDP协议上）。
- SNMP (Simple Network Management Protocol, 简单网络管理协议)，用于网络信息的收集和  
网络管理。
- DHCP (Dynamic Host Configuration Protocol, 动态主机配置协议)，动态配置IP地址。
- ARP (Address Resolution Protocol, 地址解析协议)，用于动态解析以太网硬件的地址。

## ARP协议

**ARP协议完成了IP地址与物理地址的映射。**每一个主机都设有一个 ARP 高速缓存，里面有**所在的局域网**上的各主机和路由器的 IP 地址到硬件地址的映射表。当源主机要发送数据包到目的主机时，会先检查自己的ARP高速缓存中有没有目的主机的MAC地址，如果有，就直接将数据包发到这个MAC地址，如果没有，就向**所在的局域网**发起一个ARP请求的广播包（在发送自己的 ARP 请求时，同时会带上自己的 IP 地址到硬件地址的映射），收到请求的主机检查自己的IP地址和目的主机的IP地址是否一致，如果一致，则先保存源主机的映射到自己的ARP缓存，然后给源主机发送一个ARP响应数据包。源主机收到响应数据包之后，先添加目的主机的IP地址与MAC地址的映射，再进行数据传送。如果源主机一直没有收到响应，表示ARP查询失败。

如果所要找的主机和源主机不在同一个局域网，那么就要通过 ARP 找到一个位于本局域网上的某个路由器的硬件地址，然后把分组发送给这个路由器，让这个路由器把分组转发给下一个网络。剩下的工作就由下一个网络来做。

## 为什么基于TCP的HTTP是无状态呢？

因为HTTP是短连接，即每次“请求-响应”都是一次TCP连接。比如用户一次请求就是一次TCP连接，服务器响应结束后断开连接。而每次TCP连接是没有关联的，因此HTTP是无状态的。如果想要使得每次TCP连接之间有关联，服务器和浏览器就得存储相关的信息，这个就是Cookie和Session的作用。

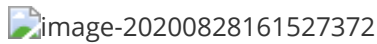
## 网络分层

image-20200817223058869

## HTTP请求体

- HTTP请求体是我们请求数据时先发送给服务器的数据，毕竟我向服务器那数据，先要表明我要什么吧
- HTTP请求体由：请求行、请求头、请求数据组成的，
- 注意：GIT请求是没有请求体的

post请求



GET请求（没有请求体的）



## http的响应报文？

- http的响应报是服务器返回给我们的数据，必须先有请求体再有响应报文
- 响应报文包含三部分 状态行、响应首部字段、响应内容实体实现



## 简述 HTTP1.0/1.1/2.0 的区别

### HTTP 1.0

HTTP 1.0 是在 1996 年引入的，从那时开始，它的普及率就达到了惊人的效果。

- HTTP 1.0 仅仅提供了最基本的认证，这时候用户名和密码还未经加密，因此很容易收到窥探。
- HTTP 1.0 被设计用来使用短链接，即每次发送数据都会经过 TCP 的三次握手和四次挥手，效率比较低。
- HTTP 1.0 只使用 header 中的 If-Modified-Since 和 Expires 作为缓存失效的标准。
- HTTP 1.0 不支持断点续传，也就是说，每次都会传送全部的页面和数据。
- HTTP 1.0 认为每台计算机只能绑定一个 IP，所以请求消息中的 URL 并没有传递主机名 (hostname) 。

### HTTP 1.1

HTTP 1.1 是 HTTP 1.0 开发三年后出现的，也就是 1999 年，它做出了以下方面的变化

- HTTP 1.1 使用了摘要算法来进行身份验证
- HTTP 1.1 默认使用长连接，长连接就是只需一次建立就可以传输多次数据，传输完成后，只需要一次切断连接即可。长连接的连接时长可以通过请求头中的 `keep-alive` 来设置
- HTTP 1.1 中新增加了 E-tag, If-Unmodified-Since, If-Match, If-None-Match 等缓存控制标头来控制缓存失效。
- HTTP 1.1 支持断点续传，通过使用请求头中的 `Range` 来实现。
- HTTP 1.1 使用了虚拟网络，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。

### HTTP 2.0

HTTP 2.0 是 2015 年开发出来的标准，它主要做的改变如下

- **头部压缩**，由于 HTTP 1.1 经常会出现 **User-Agent**、**Cookie**、**Accept**、**Server**、**Range** 等字段可能会占用几百甚至几千字节，而 Body 却经常只有几十字节，所以导致头部偏重。HTTP 2.0 使用 **HPACK** 算法进行压缩。
- **二进制格式**，HTTP 2.0 使用了更加靠近 TCP/IP 的二进制格式，而抛弃了 ASCII 码，提升了解析效率
- **强化安全**，由于安全已经成为重中之重，所以 HTTP2.0 一般都跑在 HTTPS 上。
- **多路复用**，即每一个请求都是是用作连接共享。一个请求对应一个id，这样一个连接上可以有多个请求。

## HTTP 常见的请求码

1XX Informational (请求正在处理)

2XX Success (请求成功)

3XX Redirection (重定向) 需要进行附加操作以完成请求

4XX Client Error (客户端错误)

5XX Server Error (服务器错误)

200 OK 请求正常处理

204 请求处理成功 但是没有任何资源返回给客户端(一般用于只需客户端向服务端发送消息)

206 对资源的某一部分请求 响应报文中包含由 Content-Range 指定范围的实体内容

301永久重定向 如果把资源对应的URI保存为书签, 则此时书签会根据Location首部字段提示的URI重新保存

302 临时重定向 临时地从旧地址A跳转到地址B

303 和301, 302类似 当使用post方法访问一个资源时, 把客户端以get的方式重定向到对应的URI, 返回303状态码

304 资源已经找到, 但是不满足条件, 所以不把资源返回给客户端。常用于协商缓存。

400 请求报文内有语法错误

401 该状态码表示发送的请求需要通过HTTP认证, 初次收到401响应浏览器弹出认证的对话框。若收到第二次401状态码, 则说明第一次验证失败。

403 请求资源的访问被服务器拒绝, 一般是未获得文件系统的访问权限, 访问权限出现问题。

404 服务器上找不到请求资源 或路径错误

405 请求方法被服务端识别, 但是服务端禁止使用该方法。可以用OPTIONS来查看服务器允许哪些访问方法

500 服务器端在执行请求时出错, 一般是因为web应用出现bug

502 代理服务器或网关从上游服务器中收到无效响应

503 服务器暂时处于超负载或停机维护, 目前无法处理请求

## 一次完整的HTTP请求经历几个步骤?

HTTP通信机制是在一次完整的HTTP通信过程中, web浏览器与web服务器之间将完成下列7个步骤:

### 1. 建立TCP连接

怎么建立连接的, 看上面的三次握手

### 2. Web浏览器向Web服务器发送请求行

一旦建立了TCP连接, **Web浏览器就会向Web服务器发送请求命令**。例如: GET /sample/hello.jsp HTTP/1.1。

### 3. Web浏览器发送请求头

浏览器发送其请求命令之后, 还要以头信息的形式向Web服务器发送一些别的信息, **之后浏览器发送了一空白行来通知服务器**, 它已经结束了该头信息的发送。

### 4. Web服务器应答

客户机向服务器发出请求后，服务器会客户机回送应答， **HTTP/1.1 200 OK**， **应答的第一部分是协议的版本号和应答状态码。**

#### 5. Web服务器发送应答头

正如客户端会随同请求发送关于自身的信息一样，服务器也会随同应答向用户发送关于它自己的数据及被请求的文档。

#### 6. Web服务器向浏览器发送数据

Web服务器向浏览器发送头信息后，它会发送一个空白行来表示头信息的发送到此为结束，接着， **它就以Content-Type应答头信息所描述的格式发送用户所请求的实际数据。**

#### 7. Web服务器关闭TCP连接

## 地址栏输入 URL 发生了什么

- 浏览器会根据你输入的 URL 地址，去查找域名是否被本地 DNS 缓存，不同浏览器对 DNS 的设置不同，如果浏览器缓存了你想访问的 URL 地址，那就直接返回 ip。如果没有缓存你的 URL 地址，浏览器就会发起系统调用来查询本机 `hosts` 文件是否有配置 ip 地址，如果找到，直接返回。如果找不到，就向网络中发起一个 DNS 查询。
- 浏览器需要和目标服务器建立 TCP 连接，需要经过三次握手的过程，具体的握手过程请参考上面的回答。
- 在建立连接后，浏览器会向目标服务器发起 `HTTP-GET` 请求，包括其中的 URL，HTTP 1.1 后默认使用长连接，只需要一次握手即可多次传输数据。
- 如果目标服务器只是一个简单的页面，就会直接返回。但是对于某些大型网站的站点，往往不会直接返回主机名所在的页面，而会直接重定向。返回的状态码就不是 200，而是 301,302 以 3 开头的重定向码，浏览器在获取了重定向响应后，在响应报文中 Location 项找到重定向地址，浏览器重新第一步访问即可。
- 然后浏览器重新发送请求，携带新的 URL，返回状态码 200 OK，表示服务器可以响应请求，返回报文。

## HTTP Get 和 Post 区别

HTTP 中包括许多方法，Get 和 Post 是 HTTP 中最常用的两个方法，基本上使用 HTTP 方法中有 99% 都是在使用 Get 方法和 Post 方法，所以有必要我们对这两个方法有更加深刻的认识。

get 方法一般用于请求，比如你在浏览器地址栏输入 [www.cxuanblog.com](http://www.cxuanblog.com) 其实就是发送了一个 get 请求，它的主要特征是请求服务器返回资源，而 post 方法一般用于

表单的提交，相当于是把信息提交给服务器，等待服务器作出响应，get 相当于一个是 pull/拉的操作，而 post 相当于是一个 push/推的操作。get 方法是不安全的，因为你在发送请求的过程中，你的请求参数会拼在 URL 后面，从而导致容易被攻击者窃取，对你的信息造成破坏和伪造；

/test/demo\_form.asp?name1=value1&name2=value2

而 post 方法是把参数放在请求体 body 中的，这对用户来说不可见。

```
POST /test/demo_form.asp HTTP/1.1
```

```
Host: w3schools.com
```

```
name1=value1&name2=value2
```

get 请求的 URL 有长度限制，而 post 请求会把参数和值放在消息体中，对数据长度没有要求。

get 请求会被浏览器主动 cache，而 post 不会，除非手动设置。

get 请求在浏览器反复的 回退/前进 操作是无害的，而 post 操作会再次提交表单请求。

get 请求在发送过程中会产生一个 TCP 数据包；post 在发送过程中会产生两个 TCP 数据包。对于 get



方式的请求，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）；而对于 post，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）。

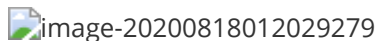
## 什么是无状态协议，HTTP 是无状态协议吗，怎么解决

无状态协议(Stateless Protocol) 就是指浏览器对于事务的处理没有记忆能力。举个例子来说就是比如客户请求获得网页之后关闭浏览器，然后再次启动浏览器，登录该网站，但是服务器并不知道客户关闭了一次浏览器。

HTTP 就是一种无状态的协议，他对用户的操作没有记忆能力。可能大多数用户不相信，他可能觉得每次输入用户名和密码登陆一个网站后，下次登陆就不再重新输入用户名和密码了。这其实不是 HTTP 做的事情，起作用的是一个叫做 小甜饼(Cookie) 的机制。它能够让浏览器具有记忆能力。

如果你的浏览器允许 cookie 的话，查看方式 chrome://settings/content/cookies

也就说明你的记忆芯片通电了..... 当你想服务端发送请求时，服务端会给你发送一个认证信息，服务器第一次接收到请求时，开辟了一块 Session 空间（创建了Session对象），同时生成一个 sessionId，并通过响应头的 **Set-Cookie: JSESSIONID=XXXXXXX** 命令，向客户端发送要求设置 Cookie 的响应；客户端收到响应后，在本机客户端设置了一个 **JSESSIONID=XXXXXXX** 的 Cookie 信息，该 Cookie 的过期时间为浏览器会话结束；



接下来客户端每次向同一个网站发送请求时，请求头都会带上该 Cookie信息（包含 sessionId），然后，服务器通过读取请求头中的 Cookie 信息，获取名称为 JSESSIONID 的值，得到此次请求的 sessionId。这样，你的浏览器才具有了记忆能力。

还有一种方式是使用 JWT 机制，它也是能够让你的浏览器具有记忆能力的一种机制。与 Cookie 不同，JWT 是保存在客户端的信息，它广泛的应用于单点登录的情况。JWT 具有两个特点

JWT 的 Cookie 信息存储在客户端，而不是服务端内存中。也就是说，JWT 直接本地进行验证就可以，验证完毕后，这个 Token 就会在 Session 中随请求一起发送到服务器，通过这种方式，可以节省服务器资源，并且 token 可以进行多次验证。

JWT 支持跨域认证，Cookies 只能用在单个节点的域或者它的子域中有效。如果它们尝试通过第三个节点访问，就会被禁止。使用 JWT 可以解决这个问题，使用 JWT 能够通过多个节点进行用户认证，也就是我们常说的跨域认证。

## Http和Https的区别

Http协议运行在TCP之上，明文传输，客户端和服务端都无法验证对方的身份；Https运行于SSL之上，SSL运行于TCP之上，是\***添加了加密和认证机制的Http**\*。

端口不同：http和https使用不同的连接方式，用的端口也不一样，前者是80端口，后者是443端口；

资源消耗不同：和http通信相比，https通信会由于加解密处理消耗更多的CPU和内存资源；

开销：https通信需要证书，而证书一般需要向认证机构购买。

https的加密机制是一种共享密钥加密和公开加密并用的混合加密机制。



## 对称加密与非对称加密

对称加密是指加密和解密使用同一个密钥的方式，这种方式存在的最大的问题就是密钥发送问题，即如何安全的将密钥发给对方；

而非对称加密是指使用一对非对称密钥，即公钥和私钥，公钥可以随意发布，但私钥只有自己知道。发送密文的一方使用对方的公钥进行加密处理，对方接收到加密信息，使用自己的私钥进行解密。

## cookie和session对于HTTP有什么用？

## 1. 什么是cookie

- cookie是由Web服务器保存在用户浏览器上的文件（key-value格式），可以包含用户相关的信息。客户端向服务器发起请求，就提取浏览器中的用户信息由http发送给服务器

## 2. 什么是session

- session 是浏览器和服务器会话过程中，服务器会分配的一块储存空间给session。
- 服务器默认为客户浏览器的cookie中设置 sessionid，这个sessionid就和cookie对应，浏览器在向服务器请求过程中传输的cookie 包含 sessionid，服务器根据传输cookie 中的 sessionid 获取出会话中存储的信息，然后确定会话的身份信息。

## 3. cookie与session区别

4. cookie数据存放在客户端上，安全性较差，session数据放在服务器上，安全性相对更高

5. 单个cookie保存的数据不能超过4K，session无此限制 信息后，使用自己的私钥进行解密。由于非对称加密的方式不需要发送用来解密的私钥，所以可以保证安全性；但是和对称加密比起来，非常的慢

6. cookie和session对于HTTP有什么用？

- HTTP协议本身是无法判断用户身份。所以需要cookie或者session

## 5. cookie与session区别

6. cookie数据存放在客户端上，安全性较差，session数据放在服务器上，安全性相对更高

7. 单个cookie保存的数据不能超过4K，session无此限制

8. session一定时间内保存在服务器上，当访问增多，占用服务器性能，考虑到服务器性能方面，应当使用cookie。

# TCP的三次握手



1. 第一次握手：Client将SYN置1，随机产生一个初始序列号seq发送给Server，进入SYN\_SENT状态；
2. 第二次握手：Server收到Client的SYN=1之后，知道客户端请求建立连接，将自己的SYN置1，ACK置1，产生一个acknowledge number=sequence number+1，并随机产生一个自己的初始序列号，发送给客户端；进入SYN\_RCVD状态；
3. 第三次握手：客户端检查acknowledge number是否为序列号+1，ACK是否为1，检查正确之后将自己的ACK置为1，产生一个acknowledge number=服务器发的序列号+1，发送给服务器；进入ESTABLISHED状态；服务器检查ACK为1和acknowledge number为序列号+1之后，也进入ESTABLISHED状态；完成三次握手，连接建立。

- 简单来说就是：

1. 客户端向服务端发送SYN
2. 服务端返回SYN,ACK
3. 客户端发送ACK

# 用现实理解三次握手的具体细节

- 三次握手的目的是建立可靠的通信信道，主要的目的就是双方确认自己与对方的发送与接收机能正常。

1. 第一次握手：客户什么都不能确认；服务器确认了对方发送正常

2. 第二次握手：客户确认了：自己发送、接收正常，对方发送、接收正常；服务器确认了：自己接收正常，对方发送正常

3. 第三次握手：客户确认了：自己发送、接收正常，对方发送、接收正常；服务器确认了：自己发送、接收正常，对方发送接收正常 所以三次握手就能确认双发收发功能都正常，缺一不可



## 建立连接可以两次握手吗？为什么？

不可以。

因为可能会出现已失效的连接请求报文段又传到了服务器端。 > client 发出的第一个连接请求报文段并没有丢失，而是在某个网络结点长时间的滞留了，以致延误到连接释放以后的某个时间才到达 server。本来这是一个早已失效的报文段。但 server 收到此失效的连接请求报文段后，就误认为是 client 再次发出的一个新的连接请求。于是就向 client 发出确认报文段，同意建立连接。假设不采用“三次握手”，那么只要 server 发出确认，新的连接就建立了。由于现在 client 并没有发出建立连接的请求，因此不会理睬 server 的确认，也不会向 server 发送数据。但 server 却以为新的运输连接已经建立，并一直等待 client 发来数据。这样，server 的很多资源就白白浪费掉了。采用“三次握手”的办法可以防止上述现象发生。例如刚才那种情况，client 不会向 server 的确认发出确认。server 由于收不到确认，就知道 client 并没有要求建立连接。

而且，两次握手无法保证 Client 正确接收第二次握手的报文（Server 无法确认 Client 是否收到），也无法保证 Client 和 Server 之间成功互换初始序列号。

## 第三次握手中，如果客户端的ACK没有送达服务器？

Server端：由于Server没有收到ACK确认，因此会每隔 3秒 重发之前的SYN+ACK（默认重发五次，之后自动关闭连接进入CLOSED状态），Client收到后会重新传ACK给Server。

Client端，会出现两种情况：

1. 在Server进行超时重发的过程中，如果Client向服务器发送数据，数据头部的ACK是为1的，所以服务器收到数据之后会读取 ACK number，进入 establish 状态
2. 在Server进入CLOSED状态之后，如果Client向服务器发送数据，服务器会以RST包应答

## 如果已经建立了连接，但客户端出现了故障怎么办？

服务器每收到一次客户端的请求后都会重新复位一个计时器，时间通常是设置为2小时，若两小时还没有收到客户端的任何数据，服务器就会发送一个探测报文段，以后每隔75秒钟发送一次。若一连发送10个探测报文仍然没反应，服务器就认为客户端出了故障，接着就关闭连接

## TCP的四次挥手



第一次挥手：Client将FIN置为1，发送一个序列号seq给Server；进入FIN\_WAIT\_1状态；

第二次挥手：Server收到FIN之后，发送一个ACK=1，acknowledge number=收到的序列号+1；进入CLOSE\_WAIT状态。此时客户端已经没有要发送的数据了，但仍可以接受服务器发来的数据。

第三次挥手：Server将FIN置1，发送一个序列号给Client；进入LAST\_ACK状态；

第四次挥手：Client收到服务器的FIN后，进入TIME\_WAIT状态；接着将ACK置1，发送一个acknowledge number=序列号+1给服务器；服务器收到后，确认acknowledge number后，变为CLOSED状态，不再向客户端发送数据。客户端等待2\*MSL（报文段最长寿命）时间后，也进入CLOSED状态。完成四次挥手。

## 用现实理解三次握手的具体细节TCP的四次挥手

- 四次挥手断开连接是因为要确定数据全部传书完了

1. 客户与服务器交谈结束之后，客户要结束此次会话，就会对服务器说：我要关闭连接了（第一次挥手）
2. 服务器收到客户的消息后说：好的，你要关闭连接了。（第二次挥手）
3. 然后服务器确定了没有话要和客户说了，服务器就会对客户说，我要关闭连接了。（第三次挥手）
4. 客户收到服务器要结束连接的消息后说：已收到你要关闭连接的消息。（第四次挥手），才关闭

## 为什么不能把服务器发送的ACK和FIN合并起来，变成三次挥手（CLOSE\_WAIT状态意义是什么）？

因为服务器收到客户端断开连接的请求时，可能还有一些数据没有发完，这时先回复ACK，表示接收到了断开连接的请求。等到数据发完之后再发FIN，断开服务器到客户端的数据传送。

## TIME\_WAIT状态的意义是什么？

第四次挥手时，客户端发送给服务器的ACK有可能丢失，TIME\_WAIT状态就是用来重发可能丢失的ACK报文。如果Server没有收到ACK，就会重发FIN，如果Client在 $2 * MSL$ 的时间内收到了FIN，就会重新发送ACK并再次等待 $2MSL$ ，防止Server没有收到ACK而不断重发FIN。MSL(Maximum Segment Lifetime)，指一个片段在网络中最大的存活时间， $2MSL$ 就是一个发送和一个回复所需的最大时间。如果直到 $2MSL$ ，Client都没有再次收到FIN，那么Client推断ACK已经被成功接收，则结束TCP连接。

## 什么是socket

网络上的两个程序通过一个双向的通讯连接实现数据的交换，这个双向链路的一端称为一个Socket。Socket通常用来实现客户端和服务方的连接。Socket是TCP/IP协议的一个十分流行的编程界面，一个Socket由一个IP地址和一个端口号唯一确定。

但是，Socket所支持的协议种类也不光TCP/IP、UDP，因此两者之间是没有必然联系的。在Java环境下，Socket编程主要是指基于TCP/IP协议的网络编程。

socket连接就是所谓的长连接，客户端和服务端需要互相连接，理论上客户端和服务端一旦建立起连接将不会主动断掉的，但是有时候网络波动还是有可能的

Socket偏向于底层。一般很少直接使用Socket来编程，框架底层使用Socket比较多

## Socket通讯的过程

基于TCP：服务器端先初始化Socket，然后与端口绑定(bind)，对端口进行监听(listen)，调用accept阻塞，等待客户端连接。在这时如果有个客户端初始化一个Socket，然后连接服务器(connect)，如果连接成功，这时客户端与服务器端的连接就建立了。客户端发送数据请求，服务器端接收请求并处理请求，然后把回应数据发送给客户端，客户端读取数据，最后关闭连接，一次交互结束。

基于UDP：UDP 协议是用户数据报协议的简称，也用于网络数据的传输。虽然 UDP 协议是一种不太可靠的协议，但有时在需要较快地接收数据并且可以忍受较小错误的情况下，UDP 就会表现出更大的优势。我客户端只需要发送，服务端能不能接收的到我不管

## TCP和UDP在Socket编程中的区别

TCP编程的服务器端一般步骤是：

- 1、创建一个socket，用函数socket();
- 2、设置socket属性，用函数setsockopt(); \* 可选
- 3、绑定IP地址、端口等信息到socket上，用函数bind();
- 4、开启监听，用函数listen();
- 5、接收客户端上来的连接，用函数accept();
- 6、收发数据，用函数send()和recv(), 或者read()和write();
- 7、关闭网络连接;
- 8、关闭监听;

#### TCP编程的客户端一般步骤是：

- 1、创建一个socket，用函数socket();
- 2、设置socket属性，用函数setsockopt();\* 可选
- 3、绑定IP地址、端口等信息到socket上，用函数bind();\* 可选
- 4、设置要连接的对方的IP地址和端口等属性;
- 5、连接服务器，用函数connect();
- 6、收发数据，用函数send()和recv(), 或者read()和write();
- 7、关闭网络连接;

与之对应的UDP编程步骤要简单许多，分别如下：

UDP编程的服务器端一般步骤是：

- 1、创建一个socket，用函数socket();
- 2、设置socket属性，用函数setsockopt();\* 可选
- 3、绑定IP地址、端口等信息到socket上，用函数bind();
- 4、循环接收数据，用函数recvfrom();
- 5、关闭网络连接;

#### UDP编程的客户端一般步骤是：

- 1、创建一个socket，用函数socket();
- 2、设置socket属性，用函数setsockopt();\* 可选
- 3、绑定IP地址、端口等信息到socket上，用函数bind();\* 可选
- 4、设置对方的IP地址和端口等属性;
- 5、发送数据，用函数sendto();
- 6、关闭网络连接;

## TCP协议Socket代码示例

服务端

```
package com.test.io;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;

//TCP协议Socket使用BIO进行通行：服务端
public class BIOServer {
    // 在main线程中执行下面这些代码
    public static void main(String[] args) {
        //1单线程服务
        ServerSocket server = null;
        Socket socket = null;
        InputStream in = null;
        OutputStream out = null;
```

```

        try {
            server = new ServerSocket(8000);
            System.out.println("服务端启动成功, 监听端口为8000, 等待客户端连接...");
            while (true){
                socket = server.accept(); //等待客户端连接
                System.out.println("客户连接成功, 客户信息为: " +
socket.getRemoteSocketAddress());
                in = socket.getInputStream();
                byte[] buffer = new byte[1024];
                int len = 0;
                //读取客户端的数据
                while ((len = in.read(buffer)) > 0) {
                    System.out.println(new String(buffer, 0, len));
                }
                //向客户端写数据
                out = socket.getOutputStream();
                out.write("hello!".getBytes());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

## 客户端

```

package com.test.io;

import java.io.IOException;
import java.io.OutputStream;
import java.net.Socket;
import java.util.Scanner;

//TCP协议Socket: 客户端
public class Client01 {
    public static void main(String[] args) throws IOException {
        //创建套接字对象socket并封装ip与port
        Socket socket = new Socket("127.0.0.1", 8000);
        //根据创建的socket对象获得一个输出流
        OutputStream outputStream = socket.getOutputStream();
        //控制台输入以IO的形式发送到服务器
        System.out.println("TCP连接成功 \n请输入: ");
        while(true){
            byte[] car = new Scanner(System.in).nextLine().getBytes();
            outputStream.write(car);
            System.out.println("TCP协议的Socket发送成功");
            //刷新缓冲区
            outputStream.flush();
        }
    }
}

```

## UDP协议Socket代码示例

## 服务端

```
//UDP协议Socket: 服务端
public class Server1 {
    public static void main(String[] args) {
        try {
            //DatagramSocket代表声明一个UDP协议的Socket
            DatagramSocket socket = new DatagramSocket(8888);
            //byte数组用于数据存储。
            byte[] car = new byte[1024];
            //DatagramPacket 类用来表示数据报包DatagramPacket
            DatagramPacket packet = new DatagramPacket(car, car.length);
            // //创建DatagramPacket的receive()方法来进行数据的接收,等待接收一个socket请求后才执行后续操作;
            System.out.println("等待UDP协议传输数据");
            socket.receive(packet);
            //packet.getLength返回将要发送或者接收的数据的长度。
            int length = packet.getLength();
            System.out.println("啥东西来了: " + new String(car, 0, length));
            socket.close();
            System.out.println("UDP协议Socket接受成功");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

## 客户端

```
//UDP协议Socket: 客户端
public class Client1 {
    public static void main(String[] args) {
        try {
            //DatagramSocket代表声明一个UDP协议的Socket
            DatagramSocket socket = new DatagramSocket(2468);
            //字符串存储人Byte数组
            byte[] car = "UDP协议的Socket请求, 有可能失败哟".getBytes();
            //InetSocketAddress类主要作用是封装端口
            InetSocketAddress address = new InetSocketAddress("127.0.0.1",
8888);

            //DatagramPacket 类用来表示数据报包DatagramPacket
            DatagramPacket packet = new DatagramPacket(car, car.length,
address);

            //send() 方法发送数据包。
            socket.send(packet);
            System.out.println("UDP协议的Socket发送成功");
            socket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

# 什么是跨域?

---

- 当一个请求url的**协议、域名、端口**三者之间任意一个与当前页面url不同即为跨域。
- 浏览器在解析执行一个网页时,如果页面中的js代码请求了另一个非同源的资源,则会产生跨域问题,而浏览器直接跳转另一个非同源的地址时不会有跨域问题

既然禁止跨域问题时浏览器的行为,那么只需要设置浏览器运行解析跨域请求的数据即可,但是这个设置必须放在服务器端,由服务器端来判断对方是否可信任,在响应头中添加一个字段,告诉浏览器,某个服务器是可信的。

## DNS域名系统（服务）协议

---

是一种分布式网络目录服务，主要用于域名与 IP 地址的相互转换，以及控制因特网的电子邮件的发送。

## 网关

---

网关又称网间连接器、协议转换器。网关在网络层以上实现网络互连，是复杂的网络互连设备，仅用于两个高层协议不同的网络互连。网关既可以用于广域网互连，也可以用于局域网互连。网关是一种充当转换重任的计算机系统或设备。

# 5.SSM

---

## Spring有几个模块?

---

## Spring框架的七大模块

---

### 1.Spring Core:

Core封装包是框架的最基础部分，提供IOC和依赖注入特性。这里的基础概念是BeanFactory，它提供对Factory模式的经典实现来消除对程序性单例模式的需要，并真正地允许你从程序逻辑中分离出依赖关系和配置。

### 2.Spring Context:

构建于Core封装包基础上的 Context封装包，提供了一种框架式的对象访问方法，有些象JNDI注册器。Context封装包的特性得自于Beans封装包，并添加了对国际化（I18N）的支持（例如资源绑定），事件传播，资源装载的方式和Context的透明创建，比如说通过Servlet容器。

### 3.Spring DAO:

DAO (Data Access Object)提供了JDBC的抽象层，它可消除冗长的JDBC编码和解析数据库厂商特有的错误代码。并且，JDBC封装包还提供了一种比编程性更好的声明性事务管理方法，不仅仅是实现了特定接口，而且对所有的POJOs（plain old Java objects）都适用。

### 4.Spring ORM:

ORM 封装包提供了常用的“对象/关系”映射APIs的集成层。其中包括JPA、JDO、Hibernate 和 iBatis。利用ORM封装包，可以混合使用所有Spring提供的特性进行“对象/关系”映射，如前边提到的简单声明性事务管理。

### 5.Spring AOP:

Spring的 AOP 封装包提供了符合AOP Alliance规范的面向方面的编程实现，让你可以定义，例如方法拦截器（method-interceptors）和切点（pointcuts），从逻辑上讲，从而减弱代码的功能耦合，清晰的被分离开。而且，利用source-level的元数据功能，还可以将各种行为信息合并到你的代码中。

## 6.Spring Web:

Spring中的 Web 包提供了基础的针对Web开发的集成特性，例如多方文件上传，利用Servlet listeners进行IOC容器初始化和针对Web的ApplicationContext。当与WebWork或Struts一起使用Spring时，这个包使Spring可与其他框架结合。

## 7.Spring Web MVC:

Spring中的MVC封装包提供了Web应用的Model-View-Controller（MVC）实现。Spring的MVC框架并不是仅仅提供一种传统的实现，它提供了一种清晰的分离模型，在领域模型代码和Web Form之间。并且，还可以借助Spring框架的其他特性。

# 请问什么是IoC和DI？并且简要说明一下DI是如何实现的？

IoC叫控制反转，DI叫依赖注入。控制反转是把传统上由程序代码直接操控的对象的调用权交给**\*容器\***，**\*通过容器来实现对象组件的装配和管理\***。"控制反转"就是**\*对组件对象控制权的转移\***，从程序代码本身转移到了外部容器，由容器来创建对象并管理对象之间的依赖关系。依赖注入的基本原则是应用组件不应该负责查找资源或者其他依赖的协作对象。配置对象的工作应该由容器负责，查找资源的逻辑应该从应用组件的代码中抽取出来，交给容器来完成。DI是对IoC更准确的描述，即组件之间的依赖关系由容器在运行期决定，即**\*由容器动态的将某种依赖关系注入到组件之中。\***

依赖注入可以通过setter方法注入（设值注入）、构造器注入和接口注入三种方式来实现，Spring支持setter注入和构造器注入，通常使用构造器注入来注入必须的依赖关系，对于可选的依赖关系，则setter注入是更好的选择，setter注入需要类提供无参构造器或者无参的静态工厂方法来创建对象。

依赖注入是从应用程序的角度在描述：**\*应用程序依赖容器\***创建并注入它所需要的外部资源；

控制反转是从容器的角度在描述：**\*容器控制应用程序\***，由容器反向的向应用程序注入应用程序所需要的外部资源。

# 请说明一下springIOC原理是什么？如果你要实现IOC需要怎么做？请简单描述一下实现步骤？

①IoC这是spring的核心，由spring来负责**\*控制对象的生命周期和对象间的关系\***。

IoC的一个在系统运行中，动态的向某个对象提供它所需要的其他对象。这一点是通过DI来实现的。比如对象A需要操作数据库，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，然后像\*\*\*一样，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。

②实现IOC的步骤

定义用来描述bean的配置的Java类、解析bean的配置、遍历存放HashMap对象

# 请谈一谈Spring中自动装配的方式有哪些

- no：不进行自动装配，手动设置Bean的依赖关系。
- byName：根据Bean的名字进行自动装配。
- byType：根据Bean的类型进行自动装配。
- constructor：类似于byType，不过是应用于构造器的参数，如果正好有一个Bean与构造器的参数类型相同则可以自动装配，否则会导致错误。



· autodetect: 如果有默认的构造器, 则通过constructor的方式进行自动装配, 否则使用byType的方式进行自动装配。

## bean的生命周期

### 请简要说明一下IOC和AOP是什么?

控制反转 (IoC) 与依赖注入 (DI) 是同一个概念, 引入IOC的目的: (1) 脱开、降低类之间的耦合; (2) 倡导面向接口编程、实施依赖倒换原则; (3) 提高系统可插入、可测试、可修改等特性。

具体做法: (1) 将bean之间的依赖关系尽可能地抓换为关联关系;

(2) 将对具体类的关联尽可能地转换为对Java interface的关联, 而不是与具体的服务对象相关联;

(3) Bean实例具体关联相关Java interface的哪个实现类的实例, 在配置信息的元数据中描述;

(4) 由IoC组件 (或称容器) 根据配置信息, 实例化具体bean类、将bean之间的依赖关系注入进来。

AOP, 即面向切面编程, 它利用一种称为"横切"的技术, 剖解封装的对象内部, 并将那些影响了多个类的公共行为封装到一个可重用模块, 并将其命名为"Aspect", 所谓"切面"是那些**\*与业务无关, 却为业务模块所共同调用的逻辑或责任封装起来\***, 便于减少系统的重复代码, 降低模块之间的耦合度, 并有利于未来的可操作性和可维护性。

使用"横切"技术, AOP把软件系统分为两个部分: 核心关注点和横切关注点。业务处理的主要流程是核心关注点, 与之关系不大的部分是横切关注点。横切关注点的一个特点是, 他们经常发生在核心关注点的多处, 而各处基本相似, 比如权限认证、日志、事物。AOP的作用在于分离系统中的各种关注点, 将核心关注点和横切关注点分离开。

### 请问Spring支持的事务管理类型有哪些? 以及你在项目中会使用哪种方式?

Spring支持**\*编程式事务管理\***和**\*声明式事务管理\***。许多Spring框架的用户选择声明式事务管理, 因为这种方式和应用程序的关联较少, 因此更加符合轻量级容器的概念。声明式事务管理要优于编程式事务管理, 尽管在灵活性方面它弱于编程式事务管理, 因为编程式事务允许你通过代码控制业务。

### 你如何理解AOP中的连接点 (Joinpoint)、切点 (Pointcut)、增强 (Advice)、引介 (Introduction)、织入 (Weaving)、切面 (Aspect) 这些概念?

a. 连接点 (Joinpoint): 程序执行的某个特定位置 (如: 某个方法调用前、调用后, 方法抛出异常后)。一个类或一段程序代码拥有一些具有边界性质的特定点, 这些代码中的特定点就是连接点。

b. 切点: 如果连接点相当于数据中的记录, 那么切点相当于查询条件, 一个切点可以匹配多个连接点。

c. 增强 (Advice): 增强是织入到目标类连接点上的一段程序代码。

d. 引介 (Introduction): 引介是一种特殊的增强, 它为类添加一些属性和方法。

e. 织入 (Weaving): 织入是将增强添加到目标类具体连接点上的过程, AOP有三种织入方式

f. 切面: 切面是由切点和增强 (引介) 组成的, 它包括了对横切关注功能的定义, 也包括了对连接点的定义。

### 请问AOP的原理是什么?

AOP指面向切面编程，用于*\*处理系统中分布于各个模块的横切关注点\**，比如事务管理、日志、缓存等等。AOP实现的关键在于AOP框架自动创建的AOP代理，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；而动态代理则以Spring AOP为代表。通常*\*使用AspectJ的编译时增强实现AOP\**，Aspectj是静态代理的增强，所谓的静态代理就是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。

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\**通过反射来接收被代理的类，并且要求被代理的类必须实现一个接口。核心是InvocationHandler接口和Proxy类。如果目标类没有实现接口，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某个类的子类

## 请问aop的应用场景有哪些？\*\*\*\*

Authentication 权限，Caching 缓存，Context passing 内容传递，Error handling 错误处理，Lazy loading 懒加载，Debugging 调试，logging, tracing, profiling and monitoring 记录跟踪 优化 校准，Performance optimization 性能优化，Persistence 持久化，Resource pooling 资源池，Synchronization 同步，Transactions 事务。

## Spring框架为企业级开发带来的好处有哪些？

- 非侵入式：支持基于POJO的编程模式，不强制性的要求实现Spring框架中的接口或继承Spring框架中的类。
- IoC容器：IoC容器帮助应用程序管理对象以及对象之间的依赖关系，对象之间的依赖关系如果发生了改变只需要修改配置文件而不是修改代码，因为代码的修改可能意味着项目的重新构建和完整的回归测试。
- AOP：将所有的*\*横切关注功能封装到切面中\**，*\*通过配置的方式将横切关注功能动态添加到目标代码上\**，进一步实现了业务逻辑和系统服务之间的分离。另一方面，有了AOP程序员可以省去很多自己写代理类的工作。
- MVC：Spring的MVC框架为Web表示层提供了更好的解决方案。
- 事务管理：Spring以宽广的胸怀接纳多种持久层技术，并且为其提供了声明式的事务管理，在不需要任何一行代码的情况下就能够完成事务管理。
- 其他：Spring为Java企业级开发提供了一站式选择，你可以在需要的时候使用它的部分和全部，更重要的是，甚至可以在感觉不到Spring存在的情况下，在你的项目中使用Spring提供的各种优秀的功能。

## Aop实现的几种方式：

第一种：静态织入，即在编译时，就将各种涉及AOP拦截的代码注入到符合一定规则的类中，编译后的代码与我们直接在RealA调用属性或方法前后增加代码是相同的，只是这个工作交由编译器来完成。

第二种：EMIT反射，即：通过Emit反射动态生成代理类

第三种：普通反射+利用Remoting的远程访问对象时的直实代理类来实现

## Spring如何选择用JDK还是CGLiB？

- 1) 当Bean实现接口时，Spring就会用JDK的动态代理。
- 2) 当Bean没有实现接口时，Spring使用CGLib是实现。

## 请问持久层设计要考虑的问题有哪些？请谈一下你用过的持久层框架都有哪些？\*\*\*\*

所谓"持久"就是将内存中的数据保存到关系型数据库、文件系统、消息队列等提供持久化支持的设备中。持久层就是系统中专注于实现数据持久化的相对独立的层面。

持久层设计的目标包括：

- 数据存储逻辑的分离，提供抽象化的数据访问接口。
- 数据访问底层实现的分离，可以在不修改代码的情况下切换底层实现。
- 资源管理和调度的分离，在数据访问层实现统一的资源调度（如缓存机制）。
- 数据抽象，提供更面向对象的数据操作。

## 请阐述一下实体对象的三种状态是什么？以及对应的转换关系是什么？\*\*\*\*

Hibernate文档中为Hibernate对象定义了四种状态，分别是：**\*瞬时态\***（new, or transient）、**\*持久态\***（managed, or persistent）、**\*游状态\***（detached）和**\*移除态\***

**瞬时态**：当new一个实体对象后，这个对象处于瞬时态，即这个对象只是一个保存临时数据的内存区域，如果没有变量引用这个对象，则会被JVM的垃圾回收机制回收。这个对象所保存的数据与数据库没有任何关系，除非通过Session的save()、saveOrUpdate()、persist()、merge()方法把瞬时态对象与数据库关联，并把数据插入或者更新到数据库，这个对象才转换为持久态对象。

**持久态**：持久态对象的实例在数据库中有对应的记录，并拥有一个持久化标识（ID）。对持久态对象进行delete操作后，数据库中对应的记录将被删除，那么持久态对象与数据库记录不再存在对应关系，持久态对象变成移除态（可以视为瞬时态）。持久态对象被修改变更后，不会马上同步到数据库，直到数据库事务提交。

**游离态**：当Session进行了close()、clear()、evict()或flush()后，实体对象从持久态变成游离态，对象虽然拥有持久和与数据库对应记录一致的标识值，但是因为对象已经从会话中清除掉，对象不在持久化管理之内，所以处于游离态（也叫脱管态）。游离态的对象与临时状态对象是十分相似的，只是它还含有持久化标识。

## 请说明一下锁机制的作用是什么？并且简述一下Hibernate的悲观锁和乐观锁机制是什么？

有些业务逻辑在执行过程中要求**\*对数据进行排他性的访问\***，于是需要通过一些机制保证在此过程中数据被锁住不会被外界修改，这就是所谓的锁机制。

Hibernate支持悲观锁和乐观锁两种锁机制。悲观锁，顾名思义**\*悲观的认为在数据处理过程中极有可能存在修改数据的并发事务\***（包括本系统的其他事务或来自外部系统的事务），于是将处理的数据设置为锁定状态。悲观锁必须依赖数据库本身的锁机制才能真正保证数据访问的排他性，乐观锁，顾名思义，对并发事务持乐观态度（认为对数据的并发操作不会经常性的发生），通过更加宽松的锁机制来解决由于悲观锁排他性的数据访问对系统性能造成的严重影响。

## 依赖注入的注入方式

- 1、使用构造函数提供
- 2、使用set方法提供
- 3、使用注解提供

## 创建bean的方式

- 1、使用默认构造函数
- 2、使用普通工厂中的方法创建对象
- 3、使用工厂中的静态方法创建对象

## Aspectj对AOP的实现：

1) 注册bean 2)配置aop 3) 定义切入点 4) 定义切面（哪种通知）

## Spring如何解决循环依赖

spring中循环依赖有三种情况：

**\*1、构造器注入形成的循环依赖。**\*也就是beanB需要在beanA的构造函数中完成初始化，beanA也需要在beanB的构造函数中完成初始化，这种情况的结果就是两个bean都不能完成初始化，循环依赖难以解决。

**\*2、setter注入构成的循环依赖。**\*beanA需要在beanB的setter方法中完成初始化，beanB也需要在beanA的setter方法中完成初始化，spring设计的机制主要就是解决这种循环依赖

**\*3、prototype作用域的bean的循环依赖。**\*这种循环依赖同样无法解决，因为spring不会缓存prototype作用域的bean，而spring中循环依赖的解决正是通过缓存来实现的。

spring只能解决setter注入构成的依赖，第二种情况中循环依赖的解决方案：

**\*步骤一：**\*beanA进行初始化，并且将自己进行初始化的状态记录下来，并提前向外暴露一个单例工程方法，从而使其他bean能引用到该bean

**\*步骤二：**\*beanA中有beanB的依赖，于是开始初始化beanB。

**\*步骤三：**\*初始化beanB的过程中又发现beanB依赖了beanA,于是又进行beanA的初始化，这时发现beanA已经在进行初始化了，程序发现了存在的循环依赖，然后通过步骤一中暴露的单例工程方法拿到beanA的引用（注意，此时的beanA只是完成了构造函数的注入但未完成其他步骤），从而beanB拿到beanA的引用，完成注入，完成了初始化，如此beanB的引用也就可以被beanA拿到，从而beanA也就完成了初始化。

## Spring的加载过程

初始化环境—>加载配置文件—>实例化Bean—>调用Bean显示信息

## SpringMVC的流程

- (1) 用户发送请求至前端控制DispatcherServlet
- (2) Dispatcher Servlet收到请求后，调用HandlerMapping处理器映射器，请求获取Handler
- (3) 处理器映射器根据请求url获取具体的处理器，返回给DispatcherServlet
- (4) DispatcherServlet调用HandlerAdapter处理器适配器
- (5) 处理器适配器经过适配调用具体的处理器
- (6) Handler执行完成返回ModelAndView
- (7) HandlerAdapter将Handler执行结果ModelAndView返回给DispatcherServlet
- (8) DispatcherServlet将ModelAndView传给ViewResolver视图解析器进行解析
- (9) ViewResolver经过解析后返回具体View
- (10) DispatcherServlet对View进行渲染视图
- (11) DispatcherServlet响应用户

## SpringMVC的主要组件？

- (1) 前端控制器DispatcherServlet（不需要程序员开发）：接收请求、响应结果，相当于转发器，有了DispatcherServlet就减少了其它组件之间的耦合度。
- (2) 处理器映射器HandlerMapping（不需要程序员开发）：根据请求的url来查找Handler
- (3) 处理器适配器HandlerAdapter：在编写Handler的时候要按照HandlerAdapter要求的规则去编

写，这样适配器HandlerAdapter才可以正确的去执行Handler。

(4) 处理器Handler (需要程序员开发)

(5) 视图解析器ViewResolver (不需要程序员开发)：进行视图的解析，根据视图逻辑名解析成真正的视图。

(6) 视图View (需要程序员开发jsp)：View是一个接口，它的实现类支持不同的视图类型

### \*3.SpringMVC和Struts2的区别? \*

(1) SpringMVC的入口是一个servlet即前端控制器，而Struts2入口是一个filter过滤器

(2) SpringMVC是基于方法开发的，传递参数到方法的形参上，Struts2是基于类开发的，传递参数是通过类的属性。

## SpringMVC怎么样设定重定向和转发的? \*\*\*\*

(1) 转发：在返回值前面加forward，如forward: user.do?name=method4

(2) 重定向：在返回值前面加redirect，如redirect:<http://www.baidu.com>

## SpringMVC怎么和ajax相互调用\*\*\*\*

通过jackson框架就可以把java里面的对象直接转化为js可以识别的json对象，具体步骤如下：

(1) 加入jackson.jar

(2) 在配置文件中配置json的映射

(3) 在接受Ajax方法里面可以直接返回Object，List等，但方法前面要加上@ResponseBody

## 如何解决post请求中乱码问题，get的又如何处理? \*\*\*\*

(1) 解决post请求乱码问题

在web.xml中配置一个CharacterEncodingFilter过滤器，设置成utf-8;

(2) 解决get请求乱码问题

1.修改tomcat配置文件添加编码与工程编码一致

2.对参数进行重新编码：

```
String userName= new String(request.getParameter("userName").getBytes("ISO8859-1"),"utf-8")
```

ISO8859-1是tomcat默认编码，需要将tomcat编码后的内容按utf-8编码。

## SpringMVC的控制器是不是单例模式，如果是，有什么问题，怎么解决？

单例模式：单例模式类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。这种**\*模式涉及到一个单一的类，该类负责创建自己的对象\***，同时确保只有单个对象被创建。这个**\*类提供了一种访问其唯一的对象的方式\***，可以直接访问，**\*不需要实例化该类的对象\***。

**\*意图：**\*保证一个类仅有一个实例，并提供一个访问它的全局访问点。

主要解决：一个全局使用的类频繁地创建与销毁。

何时使用：当您想控制实例数目，节省系统资源的时候。

如何解决：判断系统是否已经有这个单例，如果有则返回，如果没有则创建。

### **\*工厂模式\***

工厂模式提供了一种创建对象的最佳方式。在工厂模式中，我们在创建对象时不会对客户端暴露创建逻辑，并且是通过**\*使用一个共同的接口来指向新创建的对象\***。

**\*意图:** \*定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。

主要解决：主要解决接口选择的问题。

何时使用：我们明确地计划不同条件下创建不同实例时。

如何解决：让其子类实现工厂接口，返回的也是一个抽象的产品。

是单例模式，所以在多线程访问的时候有线程安全问题，不要用同步，会影响性能，解决方案是在控制器里面不能写字段。

## SpringMVC常用的注解有哪些？

@RequestMapping:用于处理请求url映射的注解，可用于类或方法上。用于类上，则表示类中的所有响应请求的方法都是以该地址作为父路径。

@RequestBody: 注解实现接受http请求的json数据，将json转换为java对象。

@ResponseBody:注解实现将Controller方法返回的对象转化为json对象响应给客户。

@RestController:注解相当于@Controller+@ResponseBody

@RequestParam:在controller的方法中参数名与表单中的参数名不一致，使用@RequestParam实现参数绑定

@RequestParam(name="username") String t\_username

@PathVariable: Controller除了可以接受表单提交的数据之外，还可以获取url中携带的变量，即路径变量

**\*9.如果在拦截请求中，我想拦截get方式提交的方法，怎么配置？\***

可以在@RequestMapping注解里面加上method=RequestMethod.GET

**\*10.如果想在拦截的方法里面得到从前台传入的参数，怎么得到？\***

直接在形参里面声明这个参数就可以，但必须名字和传过来的参数一样。

**\*11.SpringMVC中函数的返回值是什么？\***

返回值可以有多种类型，有String, void,和ModelAndView。

**\*12.SpringMVC用什么对象从后台向前台传递数据？\***

通过ModelMap对象，可以在这个对象里面调用put方法，把对象加到里面，前台就可以通过el表达式拿到。

**\*13.注解原理\***

注解本质上是一个继承了Annotation的特殊接口，其具体实现类是Java运行时生成的动态代理类。我们

通过反射获取注解时，返回的是Java运行时生成的动态代理对象。通过代理对象调用自定义注解的方法  
**最终调用AnnotationInvocationHandler的invoke方法。该方**从memberValues这个Map中索引出对应的值。而memberValues的来源是java常量池。

## 6.mybatis

### 什么是Mybatis?

1. Mybatis是一个半ORM（对象关系映射），它内部封装了JDBC，开发时只需要关注SQL语句本身，不

需要再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。

## Mybatis的优点

---

- 1.基于SQL语句编程，相当灵活，不会对应用程序或者数据库的现有设计造成任何影响，SQL写在XML里，解除sql与程序代码的耦合，便于统一管理。提供xml标签，支持编写动态SQL语句，并可重用。
- 2.与JDBC相比，减少了50%以上的代码量，消除了JDBC大量冗余的代码，不需要手动开关连接。
- 3.很好的与各种数据库兼容（因为Mybatis使用JDBC来连接数据库，所以只要JDBC支持的数据库MyBatis都支持）。
- 4.能够与Spring很好的集成。

## Mybatis框架的缺点

---

- 1.SQL语句的编写工作量较大，尤其当字段多、关联表多时，编写SQL语句的功底有一定要求。
- 2.SQL语句依赖于数据库，导致数据库移植性差，不能随意更换数据库。

## Mybatis与Hibernate有哪些不同？

---

- 1.Mybatis和Hibernate不同，它不完全是一个ORM框架，因为MyBatis需要程序员自己编写Sql语句。
- 2.Mybatis直接编写原生态sql，可以严格控制sql执行性能，灵活度高，非常适合关系数据模型要求不高的软件开发，因为这类软件需求变化频繁，一旦需求变化要求迅速输出成功。
- 3.Hibernate对象/关系映射能力强，数据库无关性好，对于关系模型要求高的软件，如果用hibernate开发可以节省很多代码，提高效率。

## #{}和\${}的区别是什么？

---

{ }是预编译处理,{ }是字符串替换

Mybatis在处理#{ }时会把sql中的#{ }替换为? 号，调用PreparedStatement的set方法来赋值。

Mybatis在处理\${ }时，就是把\${ }替换成变量的值。

使用#{ }可以有效的防止SQL注入，提高系统安全性。

## 当实体类中的属性名和表中的字段名不一样，怎么办？

---

- 1.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。
- 2.通过来映射字段名和属性名一一对应

## 模糊查询like语句该怎么写？

---

- 1.在Java代码中添加sql通配符。
- 2.在sql语句中拼接通配符，会引起sql注入

## 通常一个XML映射文件，都会写一个Dao接口与之对应，请问这个Dao接口

---

的工作原理是什么？ Dao接口里的方法，参数不同时，方法能重载吗？

- 1.Dao接口也即是Mapper接口，接口的全限名，就是映射文件中namespace的值；接口的方法名，就是映射文件中Mapper的Statement的id值；接口方法内的参数，就是传递给sql的参数。
- 2.Mapper接口是没有实现类的，当调用接口方法时，接口全限名+方法名拼接字符串作为key值，可唯一定位一个MapperStatement。



# Mybatis的XML映射文件中，不同的XML映射文件，id是否可以重复？

---

不同的XML映射文件，如果配置了namespace，那么id可以重复，如果没有配置namespace，那么id不能重复。

原因是namespace+id是作为map的key使用的，如果没有namespace，就剩下id，那么，id重复会导致数据互相覆盖。有了namespace，自然id就可以重复，namespace不同，namespace+id自然也就不同。

## 为什么说Mybatis是半自动ORM映射工具？它与全自动的区别在哪里？

---

Hibernate属于全自动ORM映射工具，使用Hibernate查询关联对象或者关联集合对象时，可以根据对象关系模型直接获取，所以它是全自动的。而Mybatis在查询关联对象或关联集合对象时，需要手动编写sql来完成，所以，称之为半自动ORM映射工具。

## MyBatis实现一对一有几种方式？具体怎么操作的？

---

有联合查询和嵌套查询，联合查询是几个表联合查询，只查询一次，通过resultMap里面配置association节点配置一对一的类就可以完成。

嵌套查询是先查一个表，根据这个表里面的结果的外键id，再去另一个表里查询数据，也是通过association配置，但另外一个表的查询通过select属性配置。

## 使用MyBatis的mapper接口调用时有哪些要求？

---

- 1.Mapper接口方法名和mapper.xml中定义每个sql的id相同。
- 2.Mapper接口方法的输入参数类型和mapper.xml中定义的每个sql的parameterType的类型相同。
- 3.Mapper接口方法的输出参数类型和mapper.xml中定义的每个sql的resultType的类型相同。
- 4.Mapper.xml文件中namespace即是mapper接口的类路径。

## mybatis的延迟加载

---

延迟加载：在真正使用数据时才发起查询，不用的时候不查询，按需加载

立即加载：不管用不用，只要一调用方法，马上发起查询

## Mybatis的一级缓存和二级缓存？

---

1) 一级缓存 Mybatis的一级缓存是指SQLSession，一级缓存的作用域是SQLSession, Mybatis默认开启一级缓存。在同一个SqlSession中，执行相同的SQL查询时；第一次会去查询数据库，并写在缓存中，第二次会直接从缓存中取。当执行SQL时候两次查询中间发生了增删改的操作，则SQLSession的缓存会被清空。每次查询会先去缓存中找，如果找不到，再去数据库查询，然后把结果写到缓存中。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) 二级缓存是mapper级别的，Mybatis默认是没有开启二级缓存的。第一次调用mapper下的SQL去查询用户的信息，查询到的信息会存放代该mapper对应的二级缓存区域。第二次调用namespace下的mapper映射文件中，相同的sql去查询用户信息，会去对应的二级缓存内取结果。如果调用相同namespace下的mapper映射文件中增删改sql，并执行了commit操作

一级缓存：也称为本地缓存，用于保存用户在一次会话过程中查询的结果，用户一次会话中只能使用一个sqlSession，一级缓存是自动开启的，不允许关闭。

二级缓存：也称为全局缓存，是mapper级别的缓存，是针对一个表的查询结果的存储，可以共享给所有针对这张表的查询的用户。也就是说对于mapper级别的缓存不同的sqlSession是可以共享的。

## JDBC编程有哪些不足之处，Mybatis是如何解决这些问题的？

1) 数据库连接的创建、释放频繁造成系统资源浪费从而影响了性能，如果使用数据库连接池就可以解决这个问题。当然JDBC同样能够使用数据源。

解决：在SQLMapConfig.xml中配置数据连接池，使用数据库连接池管理数据库连接。

2) SQL语句在写代码中不容易维护，事件需求中SQL变化的可能性很大，SQL变动需要改变JAVA代码。解决：将SQL语句配置在mapper.xml文件中与java代码分离。

3) 向SQL语句传递参数麻烦，因为SQL语句的where条件不一定，可能多，也可能少，占位符需要和参数一一对应。解决：Mybatis自动将java对象映射到sql语句。

4) 对结果集解析麻烦，sql变化导致解析代码变化，且解析前需要遍历，如果能将数据库记录封装成pojo对象解析比较方便。解决：Mybatis自动将SQL执行结果映射到java对象。

## Mybatis编程步骤？

Step1：创建SQLSessionFactory Step2：通过SQLSessionFactory创建SQLSession Step3：通过SQLSession执行数据库操作 Step4：调用session.commit()提交事物 Step5：调用session.close()关闭会话

## MyBatis与hibernate有哪些不同？

1) Mybatis MyBatis 是支持定制化 SQL、存储过程以及高级映射的一种持久层框架。MyBatis 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。Mybatis它不完全是一个ORM(对象关系映射)框架；它需要程序员自己编写部分SQL语句。mybatis可以通过xml或者注解的方式灵活的配置要运行的SQL语句，并将java对象和SQL语句映射生成最终的执行的SQL，最后将SQL执行的结果在映射生成java对象。Mybatis程序员可以直接的编写原生态的SQL语句，可以控制SQL执行性能，灵活度高，适合软件需求变换频繁的企业。缺点：Mybatis无法做到数据库无关性，如果实现支持多种数据库的软件，则需要自定义多套SQL映射文件，工作量大。2) Hibernate Hibernate是支持定制化 SQL、存储过程以及高级映射的一种持久层框架。Hibernate对象-关系映射能力强，数据库的无关性好，Hirberate可以自动生成SQL语句，对于关系模型要求高的软件，如果用Hirberate开发可以节省很多时间。

## SQLMapConfig.xml中配置有哪些内容？

properties (属性) settings (配置) typeAliases (类型别名) typeHandlers (类型处理器) objectFactory (对象工厂) plugins (插件) environments (环境集合属性对象) environment (环境子属性对象) transactionManager (事务管理) dataSource (数据源) mappers (映射器)

## Mybatis动态SQL？

传统的JDBC的方法，在组合SQL语句的时候需要去拼接，稍微不注意就会少了一个空格，标点符号，都会导致系统错误。Mybatis的动态SQL就是为了解决这种问题而产生的；Mybatis的动态SQL语句值基于OGNL表达式的，方便在SQL语句中实现某些逻辑；可以使用标签组合成灵活的sql语句，提供开发的效率。

## ORM:

---

对象关系映射 (Object Relational Mapping, 简称ORM) ,提供了概念性的、易于理解的模型化数据的方法,目的是想像操作对象一样操作数据库.因为数据库不是面向对象的,所以需要编程进行映射.

Mabatis三剑客分别是: mybatis-generator、mybatis-plugin、mybatis-pagehelper

## 配置文件总结

---

**\*web.xml\*\*\***

1.配置一个全局的参数: 指定spring容器的配置文件applicationContext.xml路径。

2.编码过滤器。

3.配置\*, 创建spring容器对象。

4.前端控制器, 指定配置文件spring-mvc.xml路径

**\*\*spring-mvc.xml\*\***

1.扫描包, 创建类对象。

2.视图解析器。

3.注解驱动。

4.自定义类型转换器。

5.文件上传。

6.拦截器。

7.静态资源放行。

**\*\*applicationContext.xml\*\*\*\***

持久层配置

1.引入数据库外部属性文件。

2.创建数据源对象。

3.创建sqlSessionFactory对象: 以前在测试类中使用构建者模式创建SessionFactory对象, 引入SqlMapConfig.xml文件。

4.扫描dao层接口的包, 创建动态代理对象, 存入spring容器中。

业务层配置

5.扫描包, 创建业务层所有类对象。

6.声明式事务

1) 事务管理类对象

2) 事务增强对象

3) aop配置: 切面配置

在使用springmvc框架的时候, 在处理json的时候需要用到spring框架特有的注解@ResponseBody或者@RestController注解, 这两个注解都会处理返回的数据格式, 使用了该类型注解后返回的不再是视图, 不会进行转跳, 而是返回json或xml数据格式, 输出在页面上。

那么, 这两个注解在使用上有什么区别呢?

@ResponseBody, 一般是使用在单独的方法上的, 需要哪个方法返回json数据格式, 就在哪个方法上使用, 具有针对性。

@RestController, 一般是使用在类上的, 它表示的意思其实就是结合了@Controller和@ResponseBody两个注解,

如果哪个类下的所有方法需要返回json数据格式的, 就在哪个类上使用该注解, 具有统一性; 需要注意的是, 使用了@RestController注解之后, 其本质相当于在该类的所有方法上都统一使用了

@ResponseBody注解, 所以该类下的所有方法都会返回json数据格式, 输出在页面上, 而不会再返回视图。

## 7.分布式

---

# 分布式锁的几种常用实现方式

分布式的CAP理论告诉我们“任何一个分布式系统都无法同时满足一致性（Consistency）、可用性（Availability）和分区容错性（Partition tolerance），最多只能同时满足两项。”所以，很多系统在设计之初就要对这三者做出取舍。在互联网领域的绝大多数的场景中，都需要**\*牺牲强一致性来换取系统的高可用性\***

针对分布式锁的实现，目前比较常用的有以下几种方案：

基于数据库实现分布式锁：核心思想是在数据库中创建一个表，表中包含方法名等字段，并在方法名字段上创建唯一索引，想要执行某个方法，就使用这个方法名向表中插入数据，成功插入则获取锁，执行完成后删除对应的行数据释放锁。

基于缓存实现分布式锁：实现思想：

（1）获取锁的时候，使用setnx加锁，并使用expire命令为锁添加一个超时时间，超过该时间则自动释放锁，锁的value值为一个随机生成的UUID，通过此在释放锁的时候进行判断。

（2）获取锁的时候还设置一个获取的超时时间，若超过这个时间则放弃获取锁。

（3）释放锁的时候，通过UUID判断是不是该锁，若是该锁，则执行delete进行锁释放。

基于Zookeeper实现分布式锁

ZooKeeper是一个为分布式应用提供一致\*\*\*的开源组件，它内部是一个分层的文件系统目录树结构，规定同一个目录下只能有一个唯一文件名。（1）创建一个目录mylock；

（2）线程A想获取锁就在mylock目录下创建临时顺序节点；

（3）获取mylock目录下所有的子节点，然后获取比自己小的兄弟节点，如果不存在，则说明当前线程顺序号最小，获得锁；

（4）线程B获取所有节点，判断自己不是最小节点，设置监听比自己次小的节点；

（5）线程A处理完，删除自己的节点，线程B监听到变更事件，判断自己是不是最小的节点，如果是则获得锁。

## 什么是分布式文件系统：

1、传统文件系统管理的文件就存储在本机。

2、分布式文件系统管理的文件存储在很多机器，这些机器通过网络连接，要被统一管理。无论是上传或是访问文件，都需要通过管理中心来访问。

功能丰富：

文件存储、文件同步、文件访问、存取负载均衡、在线扩容

适合有大容量存储需求的应用或系统。

FastDFS两个主要的角色：Tracker Server和Storage Server

Tracker Server：跟踪服务器，主要负责调度storage节点与client通信，在访问上起负载均衡的作用，和记录storage节点的允许状态，是连接client和storage节点的枢纽。

Storage Server：存储服务器，保存文件和文件的meta data（元数据），每个storage server会启动一个单独的线程主动向Tracker cluster中每个tracker server报告其状态信息，包括磁盘使用情况，文件同步情况及文件上传下载次数统计等信息。

Group：文件组，多台Storage Server的集群。上传一个文件到同组内的一台机器上后，FastDFS会将该文件即时同步到同组内的其它所有机器上，起到备份的作用。不同组的服务器，保存的数据不同，而且相互独立，不进行通信。

Tracker Cluster:跟踪服务器的集群，有一组Tracker Server(跟踪服务器)组成。

Storage Cluster：存储集群，有多个Group组成。

**\*FastDFS上传和下载流程\***

上传：

1.Client通过Tracker server查找可用的Storage server。

2.Tracker server向Client返回一台可用的Storage server的IP地址和端口号。

- 3.Client通过Tracker server返回的IP地址和端口与其中一台Storage server建立连接并进行文件上传。
- 4.上传完成，Storage server返回Client一个文件ID，文件上传结束。

下载：

- 1.Client通过Trackerserver查找要下载文件所在的Storage server。
- 2.Tracker server向Client返回包含指定文件的某个Storage server的IP地址和端口号。
- 3.Client直接通过Tracker server返回的IP地址和端口与其中一台Storage server建立连接并指定要下载文件。
- 4.下载文件成功。

## 为什么要使用RPC？ 组成部分？

---

在一个典型RPC的使用场景中，包含了服务发现、负载、容错、网络传输、序列化等组件，而RPC的主要目标是更容易地构建分布式应用。为实现该目标，RPC框架需提供一种透明调用机制，让使用者不必显式的区分本地调用和远程调用。

## 8.spring boot

---

### 热部署

---

热部署是指当我们修改代码后，服务能自动加载新修改的内容，这样大大提高了 我们开发的效率，否则每次都要手动重启，这样就比较耗时。

### 使用lombok

---

在之前编写的代码中，我们写了很多bean，这里面加了很多set、get方法，这些方法冗余，但却不可缺少。可以使用lombok包，通过该工具，就不用bean源码中手动添加set、get方法了，除此之外equals、hashCode、toString方法也无需手动在源码中添加了。

## 什么是SpringBoot？

---

SpringBoot是Spring开源组织下的子项目，是Spring组件一站式解决方案，主要是简化了使用Spring的难度，简省了繁重的配置，提供了各种启动器，开发者能快速上手。

## SpringBoot有哪些有哪些优点？

---

- 1.减少开发，测试时间和努力
- 2.使用JavaConfig有助于避免使用XML
- 3.避免大量的Maven导入和各种版本冲突。

## SpringBoot的核心配置文件有哪几个？ 它们的区别是什么？

---

SpringBoot的核心配置文件是application和bootstrap配置文件  
application：主要用于Spring Boot项目的自动化配置。

@Controller

@ResponseBody

@RestController

## SpringBoot的配置文件有哪几种格式？ 它们有什么区别？

---

.properties和.yml，它们的区别主要是书写格式不同。

## Spring Boot的核心注解是哪个？它主要由哪几个注解组成的？

---

@SpringBootApplication是SpringBoot的核心注解，主要组合包含了以下3个注解：

- 1.@SpringBootConfiguration:组合了@Configuration注解，实现配置文件的功能。
- 2.@EnableAutoConfiguration：打开自动配置的功能，也可以关闭某个自动配置的选项，如关闭数据源自动配置功能：@SpringBootApplication(exclude={DataSourceAutoConfiguration.class})
- 3.@ComponentScan:Spring组件扫描

## 开启Spring Boot特性有哪几种方式？

---

- 1.继承spring-boot-start-parent项目
- 2.导入spring-boot-dependencies项目依赖

## Spring Boot需要独立的容器运行吗？

---

可以不需要，内置了Tomcat/Jetty等容器。

## 运行SpringBoot有哪几种方式？

---

- 1、打包用命令或者放到容器中运行。
- 2、用Maven/Gradle插件运行。
- 3.直接执行main方法。

## SpringBoot自动配置原理是什么？

---

注解@EnableAutoConfiguration,@Configuration,@ConditionalOnClass就是自动配置的核心，首先它得是一个配置文件，其次根据类路径下是否有这个类去自动配置。

## SpringBoot实现分页和排序？

---

使用Spring Data-JPA可以实现将可分页的org.springframework.data.domain.Pageable传递给存储库方法。

## 如何实现Spring Boot应用程序的安全性？

---

使用spring-boot-starter-security依赖项，并且必须添加安全配置。

## 如何集成Spring Boot和ActiveMQ？

---

使用spring-boot-start-activemq依赖关系。它只需要很少的配置，并且不需要样板代码。

## SpringBoot中的监视器是什么？

---

Spring boot actuator是spring启动框架中的重要功能之一。spring boot监视器可帮助您访问生产环境中正在运行的应用程序的当前状态。

## 什么是Swagger？你用Spring Boot实现了它吗？

---

Swagger是用于生成Restful Web服务的可视化表示的工具，规范和完整框架实现。它使文档能够以与服务器相同的速度更新。

## 如何使用Spring Boot实现异常处理？

---

Spring提供了一种使用ControllerAdvice处理异常的非常有用的方法。我们通过实现一个ControllerAdvice类，来处理控制器类抛出的所有异常。

## RequestMapping和GetMapping的不同之处在哪里？

---

RequestMapping具有类属性的，可以进行GET\POST\PUT或者其它的注释中具有的请求方法  
GetMapping是GET请求方法中的一个特例。它只是RequestMapping的一个延伸，目的是为了提  
清晰度。

## 9.web

---

### 请谈一谈JSP有哪些内置对象？以及这些对象的作用分别是什么？

---

JSP有9个内置对象：

- request：封装客户端的请求，其中包含来自GET或POST请求的参数；
- response：封装服务器对客户端的响应；
- pageContext：通过该对象可以获取其他对象；
- session：封装用户会话的对象；
- application：封装服务器运行环境的对象；
- out：输出服务器响应的输出流对象；
- config：Web应用的配置对象；
- page：JSP页面本身（相当于Java程序中的this）；
- exception：封装页面抛出异常的对象。

### 请简要说明一下JSP和Servlet有哪些相同点和不同点？另外他们之间的联系又是什么呢？

---

JSP 是Servlet技术的扩展，本质上是Servlet的简易方式，更强调应用的外表表达。JSP编译后是“类servlet”。Servlet和JSP最主要的不同点在于，Servlet的应用逻辑是在Java文件中，并且完全从表示层中的HTML里分离开来。而JSP的情况是Java和HTML可以组合成一个扩展名为.jsp的文件。JSP侧重于视图，Servlet主要用于控制逻辑

## Netty和Tomcat

---

最大的区别就在于通信协议，Tomcat是基于Http协议的，他的实质是一个**\*基于http协议的web容器\***，但是Netty不一样，他能通过编程**\*自定义各种协议\***，因为netty能够通过codec自己来编码/解码字节流，完成类似redis访问的功能，这就是netty和tomcat最大的不同。



## 请问过滤器有哪些作用？以及过滤器的用法又是什么呢？

---

过滤器（filter）是从Servlet 2.3规范开始增加的功能，对Web应用来说，过滤器是一个驻留在服务器端的Web组件，它可以**\*截取客户端和服务器之间的请求与响应信息，并对这些信息进行过滤\***。当Web容器接受到一个对资源的请求时，它将判断是否有过滤器与这个资源相关联。如果有，那么容器将把请求交给过滤器进行处理。在过滤器中，你可以改变请求的内容，或者重新设置请求的报头信息，然后再将请求发送给目标资源。

## servlet的生命周期是什么。servlet是否为单例以及原因是什么？

---

Servlet 通过调用 init () 方法进行初始化。

Servlet 调用 service() 方法来处理客户端的请求。

Servlet 通过调用 destroy() 方法终止（结束）。

最后，Servlet 是由 JVM 的垃圾回收器进行垃圾回收的。

Servlet单实例，减少了产生servlet的开销；

## cookie 和 session 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，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，考虑到减轻服务器性能方面，应当使用COOKIE。
- 4、单个cookie保存的数据不能超过4K，很多浏览器都限制一个站点最多保存20个cookie。

## get和post请求，并且说说它们之间的区别？

---

- ①get请求用来从服务器上**\*获得资源\***，而post是用来向服务器**\*提交数据\***；
- ②get将表单中数据按照name=value的形式，添加到action 所指向的URL 后面，并且两者使用"?"连接，而各个变量之间使用"&"连接；post是将表单中的数据放在HTTP协议的请求头或消息体中，传递到action所指向URL；
- ③get传输的数据要受到URL长度限制（1024字节）；post可以传输大量的数据，上传文件通常要使用post方式；
- ④使用get时参数会显示在地址栏上，如果这些数据不是敏感数据可以使用get；对于敏感数据还是应用使用post；

## 转发和重定向之间的区别？

---

forward是容器中控制权的转向，是服务器请求资源，服务器直接访问目标地址的URL，把那个URL 的响应内容读取过来，然后**\*把这些内容再发给浏览器\***，浏览器根本不知道服务器发送的内容是从哪儿来的，所以它的地址栏中还是原来的地址。

redirect就是服务器端根据逻辑，发送一个状态码，告诉浏览器重新去请求那个地址，因此从浏览器的地址栏中可以看到**\*跳转后的链接地址\***，很明显redirect无法访问到服务器保护起来资源，但是可以从一个网站redirect到其他网站。forward更加高效，在有些情况下，比如需要访问一个其它服务器上的资源，则必须使用重定向

## 解决session共享问题

---

方法一、使用Nginx让它绑定ip,配置Nginx。

方法二、使用spring session+redis的方法解决session共享问题

## 9.电商项目

---

### 跨域？

---

当一个资源去访问另一个不同域名或者同域名不同端口的资源时，就会发出跨域请求。如果此时另一个资源不允许其进行跨域资源访问，那么访问的那个资源就会遇到跨域问题。

因为跨域问题是浏览器对于ajax请求的一种安全限制：一个页面发起的ajax请求，只能是于当前页同域名的路径，这能有效的阻止跨站攻击。

### 异步查询工具axios

---

异步查询数据，自然是通过ajax查询，大家首先想起的肯定是jQuery。但jQuery与MVVM的思想不吻合

### 解决跨域问题的方案

---

Jsonp：最早的解决方案，利用script标签可以跨域的原理实现。缺点：需要服务的支持、只能发起GET请求

nginx反向代理：利用nginx反向代理把跨域为不跨域，支持各种请求方式。缺点：需要在nginx进行额外配置

CORS（跨域资源共享）：规范化的跨域请求解决方案，安全可靠。

优势：在服务端进行控制是否允许跨域，可自定义规则、支持各种请求方式

缺点：会产生额外的请求

### 同源策略

---

是浏览器的安全策略。是一种约定，是浏览器最核心最基本的安全功能。如果没有同源策略，浏览器很容易收到XSS，CSRF攻击。保证用户信息安全，防止恶意网站窃取数据

同源指“协议+域名（主机）+端口”三者相同。任一不同，都属于非同源。即使不同域名对应同一IP地址也非同源。

### 服务治理（SOA）

---

当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。此时，用于提高机器利用率的资源调度和治理中心(SOA)是关键

服务治理要做什么？

- 服务注册中心，实现服务自动注册和发现，无需人为记录服务地址
- 服务自动订阅，服务列表自动推送，服务调用透明化，无需关心依赖关系
- 动态监控服务状态监控报告，人为控制服务状态

## 远程调用方式

---

无论是微服务还是SOA，都面临着服务间的远程调用。常见的远程调用方式有以下几种：

RPC：Remote Produce Call远程过程调用，类似的还有RMI。自定义数据格式，基于原生TCP通信，速度快，效率高。早期的webservice，现在热门的dubbo，都是RPC的典型

Http：http其实是一种网络传输协议，基于TCP，规定了数据传输的格式。现在客户端浏览器与服务端通信基本都是采用Http协议。也可以用来进行远程服务调用。缺点是消息封装臃肿。

现在热门的Rest风格，就可以通过http协议来实现。

微服务，更加强调的是独立、自治、灵活。而RPC方式的限制较多，因此微服务框架中，一般都会采用基于Http的Rest风格服务。

## Nginx

---

nginx可以作为web服务器，但更多的时候，我们把它作为网关，因为它具备网关必备的功能：反向代理、负载均衡、动态路由、请求过滤。

## nginx作为反向代理

---

### 什么是反向代理？

---

- 代理：通过客户机的配置，实现让一台代理服务器代理客户机，客户的所有请求都交给代理服务器
- 反向代理：用一台服务器，代理真实服务器，用户访问时，不再是访问真实服务器，而是代理服务器。

### nginx可以当做反向代理服务器来使用：

---

- 我们需要提前在nginx中配置好反向代理的规则，不同的请求，交给不同的真实服务器处理
- 当请求到达nginx，nginx会根据已经定义的规则进行请求的转发，从而实现路由功能
- 利用反向代理，就可以解决我们前面所说的端口问题

## 10.Linux

---

### 部署项目用到的linux命令

---

**\*1.进入tomcat的bin目录\*** cd /data/apache-tomcat-6.0.39/bin

**\*2.查看tomcat的进程\*** ps -ef | grep tomcat

**\*3.杀死进程\*** kill -9 + 进程数

查看进程 2.1、ps -ef | grep xx    2.2、ps -aux | grep xxx (-aux显示所有状态)

查看端口：1、netstat -anp | grep 端口号（状态为LISTEN表示被占用）

**\*4.启动项目\*** sh startup.sh

**\*5.永久删除文件\*** rm -rf 文件

**\*6.复制文件\*** cp -Rf 原路径/ 目的路径/

### \*7.压缩文件夹\*

解压: tar zxvf FileName.tar.gz

压缩: tar zcvf FileName.tar.gz DirName

### \*8.解压(安装zip命令)\*\* unzip 压缩包

### \*9.移动\* mv +路径/文件 +要移到的路径

### \*9.从本机复制文件到远程\* scp -r ROOT [root@192.168.1.1:/data/apache-tomcat-6.0.39/webapps](#)

scp -r 目录名 远程计算机用户名@远程计算机的ip: 远程计算机存放该目录的路径

## 11.并发编程

### 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、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，而线程是共享进程中的数据、地址空间
- 3、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。

### 如何保证线程安全？

通过合理的时间调度，避开共享资源的存取冲突。另外，在并行任务设计上可以通过适当的策略，保证任务与任务之间不存在共享资源

### 线程同步和线程调度的相关方法。

- wait(): 使一个线程处于等待状态，并且释放所持有的对象的锁；
- sleep(): 使一个正在运行的线程处于睡眠状态，是一个静态方法，调用此方法要处理异常；
- notify(): 唤醒一个处于等待状态的线程，当然在调用此方法的时候，并不能确切的唤醒某一个等待状态的线程，而是由JVM确定唤醒哪个线程，而且与优先级无关；
- notifyAll(): 唤醒所有处于等待状态的线程，该方法并不是将对象的锁给所有线程，而是让它们竞争，只有获得锁的线程才能进入就绪状态；

### Java中有几种方法可以实现一个线程？用什么关键字修饰同步方法？

有四种实现方法，分别是\*继承Thread类与实现Runnable接口\*，使用Callable和Future创建线程、使用线程池例如用Executor框架。

### 启动一个线程是用run()还是start()？

启动一个线程是调用start()方法，使线程所代表的虚拟处理机处于可运行状态，这意味着它可以由JVM调度并执行。这并不意味着线程就会立即运行。run()方法可以产生必须退出的标志来停止一个线程。

### 请说明一下sleep() 和 wait() 有什么区别？

sleep是线程类（Thread）的方法，导致此线程暂停执行指定时间，把\***执行机会给其他线程**\*，但是监控状态依然保持，到时会自动恢复。调用sleep不会释放对象锁。

wait是Object类的方法，对此对象调用wait方法导致本\***线程放弃对象锁**\*，进入等待此对象的等待锁定池，只有针对此对象发出notify方法（或notifyAll）后本线程才进入对象锁定池准备获得对象锁进入运行状态。

## 请分析一下同步方法和同步代码块的区别是什么？

区别：同步方法默认用this或者当前类class对象作为锁；

同步代码块可以选择以什么来加锁，比同步方法要更细颗粒度，可以选择只同步会发生同步问题的部分代码而不是整个方法。

## 请详细描述一下线程从创建到死亡的几种状态都有哪些？

新建( new )：新创建了一个线程对象。

可运行( runnable )：线程对象创建后，其他线程(比如 main 线程)调用了该对象的 start ()方法。该状态的线程位于可运行线程池中，等待被线程调度选中，获取 cpu 的使用权。

运行( running )：可运行状态( runnable )的线程获得了 cpu 时间片（ timeslice ），执行程序代码。

阻塞( block )：阻塞状态是指线程因为某种原因放弃了 cpu 使用权，阻塞的情况分三种：等待阻塞、同步阻塞、其他阻塞：

死亡( dead )：线程 run ()、 main () 方法执行结束，或者因异常退出了 run ()方法，则该线程结束生命周期。死亡的线程不可再次复生。

## 请问什么是死锁(deadlock)?

两个线程或两个以上线程都在等待对方执行完毕才能继续往下执行的时候就发生了死锁。结果就是这些线程都陷入了无限的等待中。如何避免线程死锁？

只要破坏产生死锁的四个条件中的其中一个就可以了。

破坏互斥条件：这个条件我们没有办法破坏，因为我们用锁本来就是想让他们互斥的（临界资源需要互斥访问）。

破坏请求与保持条件：一次性申请所有的资源。

破坏不剥夺条件：占用部分资源的线程进一步申请其他资源时，如果申请不到，可以主动释放它占有的资源。

破坏循环等待条件：靠按序申请资源来预防。按某一顺序申请资源，释放资源则反序释放

## JAVA中如何确保N个线程可以访问N个资源，但同时又不导致死锁？

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。

## 常用线程池，线程池有哪几个参数

Java通过Executors提供四种线程池，分别为：

1) newCachedThreadPool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。

2) newFixedThreadPool 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。

3) newScheduledThreadPool 创建一个定长线程池，支持定时及周期性任务执行。

4) newSingleThreadExecutor 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。

参数

corePoolSize：核心线程数量

maximumPoolSize：线程最大线程数

workQueue：阻塞队列，存储等待执行的任务 很重要 会对线程池运行产生重大影响

keepAliveTime：线程没有任务时最多保持多久时间终止

unit：keepAliveTime的时间单位

threadFactory：线程工厂，用来创建线程

rejectHandler：当拒绝处理任务时的策略

## 线程池怎么用

```
ExecutorService cachePool = Executors.newCachedThreadPool();
```

```
cachePool.execute(getThread(i));
```

## Synchronized和lock

synchronized当它用来修饰一个方法或者一个代码块的时候，能够保证在同一时刻最多只有一个线程执行该段代码。Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现；synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而Lock在发生异常时，如果没有主动通过unlock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；Lock可以让等待锁的线程响应中断，而synchronized却不行，\*\*\*\*使用synchronized时，等待的线程会一直等待下去，不能够响应中断；通过Lock可以知道有没有成功获取锁，而synchronized却无法办到。

## Synchronized锁，如果用这个关键字修饰一个静态方法，锁住了什么？如果修饰成员方法，锁住了什么？

synchronized修饰静态方法以及同步代码块的synchronized (类.class)用法**\*锁的是类\***，线程想要执行对应同步代码，需要获得类锁。

synchronized修饰成员方法，线程获取的是当前调用该方法的对象实例的**\*对象锁\***。

## synchronized 和 CAS 的区别

synchronized 采用的是 CPU 悲观锁机制，即线程获得的是独占锁，**\*其他线程只能依靠阻塞来等待线程释放锁\***。而在 CPU 转换线程阻塞时会引起线程上下文切换，当有很多线程竞争锁的时候，会引起 CPU 频繁的上下文切换导致效率很低。尽管 Java1.6 为 synchronized 做了优化，增加了从偏向锁到轻量级锁再到重量级锁的过度，但是在最终转变为重量级锁之后，性能仍然较低。

CAS 是比较并替换。它当中使用了3个基本操作数：内存地址 V，旧的预期值 A，要修改的新值 B。采用的是一种乐观锁的机制，它不会阻塞任何线程，所以在效率上，它会比 synchronized 要高。所谓乐观锁就是：**\*每次不加锁而是假设没有冲突而去完成某项操作，如果因为冲突失败就重试，直到成功为止。\***

所以，在并发量非常高的情况下，我们尽量的用同步锁，而在其他情况下，我们可以灵活的采用 CAS 机制。

## synchronized关键字和volatile关键字比较：

- volatile关键字是线程同步的轻量级实现，所以volatile性能肯定比synchronized关键字要好。
- 多线程访问volatile关键字不会发生阻塞，而synchronized关键字可能会发生阻塞
- volatile关键字能保证数据的可见性，但不能保证数据的原子性。synchronized关键字两者都能保证。
- volatile关键字主要用于解决变量在多个线程之间的可见性，而synchronized关键字解决的是多个线程之间访问资源的同步性。

## 线程池运行原理分析\*\*\*\*

---

- 1、创建一个线程池，在还没有任务提交的时候，默认线程池里面是没有线程的。也可以调用

```
startCoreThread
```

方法，来预先创建一个核心线程。
- 2、线程池里还没有线程或者线程池里存活的线程数小于核心线程数`corePoolSize`时，这时对于一个新提交的任务，线程池会创建一个线程去处理提交的任务。此时线程池里面的线程会一直存活，就算空闲时间超过了`keepAliveTime`，线程也不会被销毁，而是一直阻塞在那里一直等待任务队列的任务来执行。
- 3、当线程池里面存活的线程数已经等于`corePoolSize`了，对于一个新提交的任务，会被放进任务队列`workQueue`排队等待执行。而之前创建的线程并不会被销毁，而是不断的去拿阻塞队列里面的任务，当任务队列为空时，线程会阻塞，直到有任务被放进任务队列，线程拿到任务后继续执行，执行完了过后会继续去拿任务。这也是为什么线程池队列要是用阻塞队列。
- 4、当线程池里面存活的线程数已经等于`corePoolSize`了,并且任务队列也满了，这里假设`maximumPoolSize > corePoolSize`(如果等于的话，就直接拒绝了),这时如果再来新的任务，线程池就会继续创建新的线程来处理新的任务，知道线程数达到`maximumPoolSize`，就不会再创建了。这些新创建的线程执行完了当前任务过后，在任务队列里面还有任务的时候也不会销毁，而是去任务队列拿任务出来执行。在当前线程数大于`corePoolSize`过后，线程执行完当前任务，会有一个判断当前线程是否需要销毁的逻辑：如果能从任务队列中拿到任务，那么继续执行，如果拿任务时阻塞（说明队列中没有任务），那超过`keepAliveTime`时间就直接返回`null`并且销毁当前线程，直到线程池里面的线程数等于`corePoolSize`之后才不会进行线程销毁。
- 5、如果当前的线程数达到了`maximumPoolSize`，并且任务队列也满了，这种情况下还有新的任务过来，那就直接采用拒绝的处理器进行处理。默认的处理器逻辑是抛出一个`RejectedExecutionException`异常。

## 请说明一下线程池有什么优势？

---

第一：降低资源消耗。第二：提高响应速度。第三：提高线程的可管理性

## 线程池的运行流程

---

首先判断核心线程池里的线程是否都在执行任务，如果不是则直接从核心线程池中创建一个线程来执行，如果都在忙则判断任务队列是否也满了，没满的话将任务放进去等待执行，满了就判断线程池的全部线程是否都在忙，如果都在忙就交给饱和策略来处理，否则就创建一个线程来帮助核心线程处理任务。

## 线程阻塞

---

- BIO，同步阻塞式IO，简单理解：一个连接一个线程
- NIO，同步非阻塞IO，简单理解：一个请求一个线程
- AIO，异步非阻塞IO，简单理解：一个有效请求一个线程

## AQS原理

---



抽象的队列式同步器，是除了java自带的synchronized关键字之外的锁机制，**\*AQS的核心思想\***是，如果被请求的共享资源空闲，则将当前请求资源的线程设置为有效的工作线程，并将共享资源设置为锁定状态，如果被请求的共享资源被占用，那么就需要一套线程阻塞等待以及被唤醒时锁分配的机制，这个机制AQS是用CLH队列锁实现的，即将暂时获取不到锁的线程加入到队列中。

CLH队列是一个虚拟的双向队列，虚拟的双向队列即不存在队列实例，仅存在节点之间的关联关系。

**\*AQS是将每一条请求共享资源的线程封装成一个CLH锁队列的一个结点(Node)，来实现锁的分配\***

## 12.JVM

---

### 主内存：

---

多个线程共享的内存，方法区和堆属于主内存区域。

### 线程工作内存

---

每个线程独享的内存。虚拟机栈、本地方法栈、程序计数器属于线程独享的工作内存

### JVM加载class文件的原理是什么？

---

JVM中类的装载是由ClassLoader和它的子类来实现的,Java ClassLoader 是一个Java运行时系统组件。它负责在运行时查找和装入类文件的类。

Java中的所有类都需要由类加载器装载到JVM中才能运行。类加载器的工作就是把class文件从硬盘读取到内存中。

### 双亲委派机制

---

如果一个类加载器收到了类加载请求，他并不会自己先去加载，而是把这个请求委托给父类的加载器去执行；

如果父类加载器还存在其父类加载器，则进一步向上委托，依次递归，请求最终达到顶层的启动类加载器

如果父类加载器可以完成类加载任务就成功返回，如果父类加载器不能完成加载任务，子加载器互尝试自己去加载

### 堆内存的分配策略

---

1对象优先在eden区分配，Eden区没有足够的空间，将触发一次Minor GC。

2大对象直接进入老年代（为了避免为大对象分配内存时由于分配担保机制带来的复制而降低效率），

3长期存活的对象进入老年代

4动态对象年龄判断：如果Survivor区中相同年龄的所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象可以直接进入到老年代，无需等到MaxTenuringThreshold中要求的年龄

5空间分配担保：-XX: HandlePromotionFailure

### Minor GC与Full GC分别在什么时候发生？

---

**\*触发MinorGC(Young GC)\***

虚拟机在进行minorGC之前会判断**\*老年代最大的可用连续空间\***是否大于**\*新生代的所有对象总空间\***

- 1、如果大于的话，直接执行minorGC
- 2、如果小于，判断是否开启HandlerPromotionFailure，没有开启直接FullGC
- 3、如果开启了HanlerPromotionFailure, JVM会判断老年代的最大连续内存空间是否大于历次晋升的大小，如果小于直接执行FullGC

#### **\*触发FullGC\***

老年代空间不足

如果创建一个对象，Eden区域当中放不下这个大对象，会直接保存在老年代当中，如果老年代空间也不足，就会触发Full GC。为了避免这种情况，最好就是不要创建太大的对象。

持久代空间不足

如果有持久代空间的话，系统当中需要加载的类，调用的方法很多，同时持久代当中没有足够的空间，就出触发一次Full GC

YGC出现promotion failure

## Minor GC 和 Full GC 有什么不同呢？

---

新生代 GC (Minor GC) :指发生新生代的垃圾收集动作，Minor GC 非常频繁，回收速度一般也比较快。

老年代 GC (Major GC/Full GC) :指发生在老年代的 GC，出现了 Major GC 经常会伴随至少一次的 Minor

GC (并非绝对)，Major GC 的速度一般会比 Minor GC 的慢 10 倍以上

## 分代收集器

---

老生代和新生代两个区域，而新生代又会分为：Eden 区和两个 Survivor区 (From Survivor、To Survivor)

## 为什么 Survivor 分区不能是 0 个

---

如果 Survivor 是 0 的话，也就是说新生代只有一个 Eden 分区，每次垃圾回收之后，存活的对象都会进入老生代，这样老生代的内存空间很快就被占满了，从而触发最耗时的 Full GC，显然这样的收集器的效率是我们完全不能接受的。

## 为什么 Survivor 分区不能是 1 个？

---

如果 Survivor 分区是 1 个的话，假设我们把两个区域分为 1:1，那么任何时候都有一半的内存空间是闲置的，显然空间利用率太低不是最佳的方案。但如果设置内存空间的比例是 8:2，只是看起来似乎“很好”，假设新生代的内存为 100 MB (Survivor 大小为 20 MB)，现在有 70 MB 对象进行垃圾回收之后，剩余活跃的对象为 15 MB 进入 Survivor 区，这个时候新生代可用的内存空间只剩了 5 MB，这样很快又要进行垃圾回收操作，显然这种垃圾回收器最大的问题就在于，需要频繁进行垃圾回收。

## 为什么 Survivor 分区是 2 个？

---

如果 Survivor 分区有 2 个分区，我们就可以把 Eden、From Survivor、To Survivor 分区内存比例设置为 8:1:1，那么任何时候新生代内存的利用率都 90%，这样空间利用率基本是符合预期的。再者就是虚拟机的大部分对象都符合“朝生夕死”的特性，所以每次新对象的产生都在空间占比较大的 Eden 区，垃圾回收之后再把存活的对象方法存入 Survivor 区，如果是 Survivor 区存活的对象，那么“年龄”就 +1，当年龄增长到 15 (可通过设定) 对象就升级到老生代。

## 请说明一下垃圾回收的优点以及原理

---

使得Java程序员在编写程序的时候不再需要考虑内存管理。由于有个垃圾回收机制，Java中的对象不再有"作用域"的概念，只有对象的引用才有"作用域"。垃圾回收可以**\*有效的防止内存泄露\***，有效的使用可以使用的内存。垃圾回收器通常是作为一个单独的低级别的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。不再会被使用的对象的内存不能被回收，就是内存泄露

## 判断对象是否可回收的方法

---

引用计数法:给对象中添加一个引用计数器，每当有一个地方引用它，计数器就加 1；当引用失效，计数器就减 1；任何时候计数器为 0 的对象就是不可能再被使用的

可达性分析法:通过一系列的称为“GC Roots”的对象作为起点，从这些节点开始向下搜索，节点所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链相连的话，则证明此对象是不可用的

## 强引用、软引用、弱引用、虚引用（虚引用与软引用和弱引用的区别、软引用能带来的好处）

---

### 1. 强引用

以前我们使用的大部分引用实际上都是强引用，这是使用最普遍的引用。如果一个对象具有强引用，那就类似于必

不可少的生活用品，垃圾回收器绝不会回收它。当内存空间不足，Java 虚拟机宁愿抛出 OutOfMemoryError 错

误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足问题

### 2. 软引用

如果一个对象只具有软引用，那就类似于**\*可有可无的生活用品\***。如果内存空间足够，垃圾回收器就不会回收它，如

果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可

用来实现内存敏感的高速缓存。

软引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收，JAVA 虚拟机就会把这个软引用加入到与之关联的引用队列中

### 3. 弱引用

如果一个对象只具有弱引用，那就类似于**\*可有可无的生活用品\***。弱引用与软引用的区别在：只具有弱引用的对象

拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。

弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java 虚拟机就

会把这个弱引用加入到与之关联的引用队列中。

### 4. 虚引用

虚引用主要用来跟踪对象被垃圾回收的活动，在程序设计中一般很少使用弱引用与虚引用，使用软引用的情况较多，因为**\*软引用可以加速\*** JVM对垃圾内存的回收速度，可以维护系统的运行安全，防止内存溢出等问题的产生\*\*

## 如何判断一个常量是废弃常量

---

运行时常量池主要回收的是废弃的常量。

假如在常量池中存在字符串"abc"，如果当前没有任何 String 对象引用该字符串常量的话，就说明常量"abc" 就是

废弃常量，如果这时发生内存回收的话而且有必要的话，"abc" 就会被系统清理出常量池。

# 如何判断一个类是无用的类

方法区主要回收的是无用的类,类需要同时满足下面 3 个条件才能算是无用的类

- 1、该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。
- 2、加载该类的 ClassLoader 已经被回收。
- 3、该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

## 垃圾回收算法：

### 1、标记-清除算法

算法分为“标记”和“清除”阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。它是

最基础的收集算法，效率也很高，但是会带来两个明显的问题：\1. 效率问题\2. 空间问题

### 2、复制算法

为了解决效率问题，“复制”收集算法出现了。它可以将内存分为大小相同的两块，每次使用其中的一块。当这一块

的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收

都是对内存区间的一半进行回收。

### 3、标记-整理算法

根据老年代的特点特出的一种标记算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象回收，而是让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存。

### 4、分代收集算法

当前虚拟机的垃圾收集都采用分代收集算法，根据对象存活周期的不同将内存分为几块。一般将 java 堆分为新生代和老年代，这样我们就可以根据各个年代的特点选择合适的垃圾收集算法。

比如在新生代中，每次收集都会有大量对象死去，所以可以选择复制算法，只需要付出少量对象的复制成本就可以

完成每次垃圾收集。而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须

选择“\*标记\*-\*清除\*”或“\*标记\*-\*整理\*”算法进行垃圾收集。

## 垃圾收集器

1、Serial 收集器 它的“\*单线程\*”的意义不仅仅意味着它只会使用一条垃圾收集线程去完成垃圾收集工作，更重要的是它在进行垃圾收集工作的时候必须暂停其他所有的工作线程，直到它收集结束。新生代采用复制算法，老年代采用标记-整理算法

2、ParNew 收集器其实就是 Serial 收集器的多线程版本，除了使用多线程进行垃圾收集外，其余行为（控制参数、收集算法、回收策略等等）和 Serial 收集器完全一样

3、Parallel Scavenge 收集器它关注点是否吞吐量（高效率的利用 CPU）。CMS 等垃圾收集器的关注点更多的是用户线程的停顿时间（提高用户体验）。新生代采用复制算法，老年代采用标记-整理算法。

4、Serial Old 收集器，它是一个单线程收集器。它主要有两大用途：一种用途是在 JDK1.5 以及以前的版本中与 Parallel Scavenge 收集器搭配使用，另一种用途是作为 CMS 收集器的后备方案。

5、Parallel Old 收集器

6、CMS 收集器，是一种以获取最短回收停顿时间为目标的收集器。它而非常符合在注重用户体验的应用上使用。CMS（Concurrent Mark Sweep）收集器是 HotSpot 虚拟机第一款真正意义上的并发收集器，它第一次实现了让垃圾收集线程与用户线程（基本上）同时工作

7、G1 收集器是一款面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器.以极高概率满

足 GC 停顿时间要求的同时,还具备高吞吐量性能特征.

## 13.项目（此部分仅做参考）

---

### 认证授权是如何实现的？

---

Spring security + Oauth2完成用户认证及用户授权。认证授权流程如下：

- 1、用户请求认证服务完成身份认证。
- 2、认证服务下发用户身份令牌和JWT令牌，拥有身份令牌表示身份合法，Jwt令牌用于完成授权。
- 3、用户携带jwt令牌请求资源服务。
- 4、网关校验用户身份令牌的合法，不合法表示用户没有登录，如果合法则放行继续访问。
- 5、资源服务获取jwt令牌，根据jwt令牌完成授权

### 认证与授权实现思路

---

如果系统的模块多，每个模块都需要就行授权与认证，所以选择基于token的形式进行授权与认证，用户根据用户名密码认证成功，然后获取当前用户角色的一系列权限值，并以用户名为key，权限列表为value的形式存入redis缓存中，根据用户名相关信息生成token返回，浏览器将token记录到cookie中，每次调用api接口都默认将token携带到header请求头中，Spring-security解析header头获取token信息，解析token获取当前用户名，根据用户名就可以从redis中获取权限列表，这样Spring-security就能够判断当前请求是否有权限访问

### 使用rabbitMQ

---

- 1、平台包括多个站点，页面归属不同的站点，需求是发布一个页面应将该页面发布到所属站点的服务器上。
- 2、每个站点服务部署CMS Client程序，并与交换机绑定，绑定时指定站点Id为routingKey。指定站点id为routingKey就可以实现cms client只能接收到所属站点的页面发布消息。
- 3、页面发布程序向MQ发布消息时指定页面所属站点Id为routingKey，根据routingKey将消息发给指定的CMS Client。

### 分布式事务

---

- 1、在微服务中使用Spring 声明式事务控制方式进行控制，在Service方法上添加@Transactional注解即可实现事务控制，它控制的是MySQL的本地事务。
- 2、项目中存在分布式事务控制，比如下单支付、课程发布等地址都用到了分布式事务。本项目实现分布式事务控制实现最终数据一致性，做法是：
  - a、将分布式事务拆分为多个本地事务。
  - b、提交事务前每个参与者要通过数据校验，和资源预留。
  - c、由消息队列去通知多个事务参与者完成本地事务的提交。
  - d、提交失败的本地事务会重试。

### 项目中课程搜索采用ElasticSearch来完成。实现方法是：

---

- 1、使用 Logstash（logstash是ES下的一款开源软件，它能够同时 从多个来源采集数据、转换数据）将MySQL中的课程信息读取到ES中创建索引，使用IK分词器进行分词。
- 2、使用 Java High Level REST Client完成搜索。
- 3、生产环境使用ES部署为集群。

# 系统对异常的处理使用统一的异常处理流程

- 1、自定义异常类型。
- 2、自定义错误代码及错误信息。
- 3、对于可预知的异常由程序员在代码中主动抛出自定义异常类型的异常，抛出异常时需要指定错误代码。
- 4、对于不可预知的异常（运行时异常）由SpringMVC统一捕获Exception类型的异常，由统一的异常捕获类来解析处理，并转换为与自定义异常类型一致的信息格式（错误代码+错误信息）。
- 5、可预知的异常及不可预知的运行时异常最终会采用统一的信息格式（错误代码+错误信息）来表示，最终也会随请求响应给客户端。

## http协议处理视频流

http采用DASH传输视频流，原理是DASH服务器将视频进行了切片，MPD是一个XML，为接收端播放器提供表示不同分片的URL、时序、比特率、清晰度等信息。客户端通过接受、解析MPD文件作为自适应流的依据，客户端基于MPD信息为分片发送HTTP请求，然后客户端的媒体播放器为收到的分片进行解码和播放。

## 你在开发中遇到什么问题？是怎么解决的？

例子：

在处理订单时要用到定时任务，当时采用的是Spring Task来完成，由于一个订单服务会部署多个，多个订单服务

同时去处理任务会造成任务被重复处理的情况，如何解决任务的重复处理。

解决：

采用乐观锁解决，在任务表中设置一个version字段记录版本号，取出任务记录同时拿到任务的版本号，执行前对

任务进行锁定，具体的做法是执行update根据当前版本号将版本号加1，update成功表示锁定任务成功，即可开始执行任务。

例子：

Redis服务器 can not get resource from pool. 1000个线程并发还能跑，5000个线程的时候出现这种问题，查后台debug日志，发现redis 线程池不够。刚开始设置的是：

解决：顺便也改了一下jdbc的连接池参数，最大空闲和最大连接数都改成1000.在测一下。可以

例子：

注册中心和服务提供者集群注册失败，启动报错

解决：修改yaml文件，修改启动类上的注解，yaml文件缩进出了问题，服务端启动类上应该添加注解@EnableEurekaServer，客户端@EnableDiscoveryClient

例子：

xml文件配置错误，页面访问一直无数据

解决：根据浏览器的开发工具，定位错误，检查xml文件的SQL，namespace，parameterType等是否正确，是否假如应有的注解

例子：

git合并冲突，甲乙都是根据point.java文件进行开发，甲开发出来版本2，并且提交了代码，乙开发出来版本3，也需要提交代码，此时将会报错存在冲突。因为甲提交版本之后，此时远端的代码已经是版本2了，而乙是在版本1的基础上进行开发出了版本3，所以乙想要提交代码，势必要将自己的代码更新为版本2的代码，然后在进行提交

解决：拉去远端代码，存在冲突，会报错。此时我们需要将本地代码暂存起来stash;更新本地代码，将本地代码版本更新和远端的代码一致，将暂存的代码合并到更新后的代码后，有冲突的要手动解决冲突，然后提交解决冲突后的代码。Git<resolve conflict

例子：

支付接口采用微信的扫码支付拿到需求后，首先去阅读微信的接口文档，重点阅读统一下单、支付结果通知、支付结果查询三个接口。下载官方提供的sdk编写单元测试用例测试每个接口。测试时没有使用微信的沙箱测试，直接使用正式接口，将金额改的小一些进行测试。单元测试通过后开发整个支付功能，最终集成测试通过。

解决：订单支付接口参数的签名问题，当时是因为自己没有仔细看接口文档导致少写一个必填参数一直报签名失败，随后将所有必填参数填写完成，最终解决问题。

例子：

一个接口出现Bug你是怎么调试的？

1、接口的开发需要前端和服务端共同调试，要仔细阅读测试人员反映的bug信息，判断这个bug是服务端的bug还是前端的bug。通常服务接口开发完成会使用postman工具进行测试，测试没有问题再提交到Git或SVN。

2、找到bug的出错点就可以根据bug信息进行修改。修改完成需要前后端再次连调测试，按照测试人员提交的测试流程重新进行测试，测试通过将此bug置为已解决。