Taotao商城

## 1.自我介绍

我叫胡桃，目前在西安电子科技大学读研，我的专业是计算机软件与理论，我的研究方向是多线性程序的检测。本科就读于湘潭大学, 主修专业是数学，由于对计算机兴趣浓厚，辅修了计你好，我叫胡桃，本科毕业于湖南省的湘潭大学，专业是信息与计算科学，他一是国家重点学科。虽然是一个以数学为主的专业，但是也学了一些计算机相关的课程如C语言,数据结构，离散数据。在大一期间我通过了计算机二级，这让我看到了自己的编程天赋，所以本科选择辅修计算机科学与技术的双学位，学习了java、计算机网络、数据库等学科。通过自己努力我获得了学院的保研资格，由于喜爱计算机，所以我选择了西安电子科技大学的计算机院；我目前的专业是计算机软件与理论，研究方向是多线程安全验证。在做好自己的研究项目的同时，我通过视屏、书籍和博客深入的了解了java后台开发的相关知识。

由于当前电商行业非常火爆（影响到了各行各业），也是当前互联网发展趋势，故此选择了一个商城后台的开发项目，一来是为了方便找工作，二来时为了学以致用。

## 2.项目介绍

这个商城项目是模拟京东商城的架构，项目分为两部分，前台和后台，首先讲一下后台，后台是给商城管理员使用，其主要功能是用于进行商品的添加修改，删除，后台采用Maven聚合工程和ssm框架，我们将商品的图片放在特殊的图片服务器中，将大文本信息放在文件服务器中，使用nginx反向代理管理图片服务器，文件服务器和数据库，由于后台访问量比较少因此不考虑缓存和分布式。

接下来就是前台项目，采用Maven管理jar包和ssm框架，前台项目和后台公用同一个数据库，前台主要是供客户访问，包括商城页面的展示，搜索功能，商品详情展示，单点登陆系统，购物车和订单系统等，由于前台访问量比较大，故此我们采用的是分布式架构，使用redis作用缓存，redis时基于内存的缓存服务器，拥有诸多优良的特性，并搭建redis集群，我们经常访问的数据放在redis中，并设置过期时间，以减缓大量访问时数据库的压力；对于商城项目而言，商品搜索时比较常见的，对于搜索的实现我们使用solr搜索引擎，只需要简单配置就可以实现全文搜索，由于solr中默认是没有中文分析器，因此需要我们配置，接下来将数据库中的数据导入，solr自带缓存，所以不需要再配置redis；单点登陆系统主要是session和cookie，使用redis 模拟session，将用户信息传给前台，当再某些特殊业务时需要强制登陆。

## 3.redis相关

## （1）为什么使用redis？

a.性能极高 – Redis是基于内存的，故此具有较高的读写频率。（能支持超过 100K+ 每秒的）

b.丰富的数据类型 – Redis支持二进制案例的 Strings, Lists, Hash, Set 及 Zset数据类型操作。

c.原子 – Redis的所有操作都是原子性的，同时Redis还支持对几个操作全并后的原子性执行。

d.丰富的特性 – Redis还支持 publish/subscribe, 通知, key 过期等等特性。

e.支持数据的持久化。redis会周期性的把更新的数据写入磁盘或者把修改操作写入追加的记录文件，并且在此基础上实现了master-slave(主从)同步.

## （2）redis 是底层是如何实现的

Redis内部维护一个db数组，每个db都是一个数据库，默认情况下Redis会创建16个数据库。我们可以通过select命令来切换数据库，如select 1切换到数据库号为1的数据库。select实现是通过修改客户端的db指针，通过指针指向不同的数据库来实现数据库的切换操作的。

需要注意的是，为了不造成操作数据库号错误，最好执行命令之前，手动select一下数据库。

二.数据库键空间

Redis是一个键值对数据库服务器，Redis通过字典保存了数据库中的所有键值对，我们将这个字典称为键空间。键空间的每个键都是一个字符串对象，键空间的值也就是数据库的值，可以是字符串对象，列表对象，哈希表对象，集合对象，有序集合对象中的任何一种。

1.添加新键

每次添加一个新键就是将一个新键值对添加到键空间里面，其中键为字符串对象，值为任意一种类型的Redis对象。

2.删除键

删除键就是在键空间里删除键所对应的键值对对象。

3.更新键

更新键就是对键空间里面键所对应的值对象进行更新。

4.查找键

查找键就是在键空间中取出键所对应的值对象。

## （3）Redis实现数据的同步于更新

每次在键空间读取一个键之后，服务器会更新键的LRU时间，用于计算键的闲置时间。如果服务器在读取一个键时发现该键已经过期，那么服务器会先删除这个过期键，然后才执行后续操作。如果有客户端使用watch命令监视了某个键，那么服务器在对被监视的键进行修改之后，会将这个键标记为dirty，从而让事务注意到这个键被修改过。服务器每次修改一个键之后，都会对键计数器的值+1，这个计数器用来触发服务器的持久化操作。如果服务器开启了数据库通知功能，那么在对键进行修改之后，服务器将按配置发送相应的数据库通知。

## （4）设置redis数据库里面的生存时间和过期时间以及删除过期

我们知道expire命令或者pexpire命令可以对一个键设置生存时间，经过指定的时间之后，服务器会自动删除生存时间为0的键。那么Redis是如何实现删除过期键的操作的呢？Redis有四个命令可以设置键的过期时间，包括expire,pexpire,expireat,pexpireat，不过这四个命令最后都会转化成pexpireat命令来实现。Redis使用一个过期字典记录所有带过期时间的键，字典的键指向键空间中的某个键对象，字典的值是一个long long类型的整数，这个整数保存了键空间所指向的数据库键的过期时间。通过过期字典，程序可以检查一个给定键是否过期，检查给定键是否存在于过期字典，如果存在，取得键的过期时间，检查当前时间戳是否大于键的过期时间，如果是的话，键已经过期，否则键未过期。

四.过期键的删除策略

如果一个键过期了，那么它什么时候被删除呢？通常我们可以用三种删除策略删除过期键

1.定时删除：在设置键过期时间的同时，创建一个定时器，让定时器在键的过期时间来临时，删除键

2.惰性删除：放任键过期不管，但是每次动键空间获取键时，都会检查键是否过期，如果过期，则删除。

3.定期删除：每隔一段时间，程序就对数据库进行一次检查，删除里面的过期键。

这几种方式各有利有弊，首先定时删除对内存最友好，当一个键过期时，一定会删除这个键，释放内存。不过定时删除对CPU最不友好，在过期键比较多的情况下，删除过期键这一行为可能会占用相当一部分CPU时间。此外，创建定时器需要用到Redis服务器中的时间时间，而当前时间时间的实现方式-无序链表查找一个事件的时间复杂度为O(N)，不能高效地处理大量时间事件。

惰性删除策略对CPU是最友好的，但是对内存最不友好。如果一个键已经过期，这个键又保留在数据库中，那么内存就会一直占用不释放。

定期删除算是前两种策略的一种整合和折中，定期策略每隔一段时间执行一次删除过期键操作，并通过限制删除操作执行的时长和频率减少删除操作对CPU时间的影响。定期删除过期键可以有效地减少因为过期键带来的内存浪费。

Redis过期键的删除使用惰性删除和定期删除两种策略配合使用。惰性策略比较好理解，所有读写数据库的命令执行之前都会对输入键进行检查，如果键过期，那么从数据库中删除键。定期删除策略的实现由Redis的serverCron函数来执行，这个函数每100ms执行一次，它在规定的时间内，分多次遍历服务器中的各个数据库，从数据库的expires字典中随机检查一部分键的过期时间，删除其中的过期键。

## （5）复制功能对过期键的处理

Redis复制主要包括RDB复制和AOF复制，在RDB复制中，每次执行SAVE或BGSAVE命令创建一个新的RDB文件时，程序会对数据库中的键进行检查，已过期的键不会被保存到新创建的RDB文件中。载入RDB文件时，服务器也会对保存的键进行检查，如果键已过期，则不会载入。当使用AOF持久化模式运行时，当过期键被惰性删除或者定期删除之后，程序会向AOF文件追加一条删除命令，记录键已被删除。

## （6）当访问量特别大的时候，redis内存有限，如何实现有效的访问。

a.设置内存最大值

b.浏览器缓存

c.集群

## （7）redis集群

在redis的每一个节点上，一个是插槽（slot）可以理解为是一个可以存储两个数值的一个变量这个变量的取值范围是：0-16383。还有一个就是cluster，理解为是一个集群管理的插件。当我们的存取的key到达的时候，redis会根据crc16的算法得出一个结果，然后把结果对 16384 求余数，这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，通过这个值，去找到对应的插槽所对应的节点，然后直接自动跳转到这个对应的节点上进行存取操作。

还有就是因为如果集群的话，是有好多个redis一起工作的，那么，就需要这个集群不是那么容易挂掉，所以呢，理论上就应该给集群中的每个节点至少一个备用的redis服务。这个备用的redis称为从节点（slave）。每一个节点都存有这个集群所有主节点以及从节点的信息。

它们之间通过互相的ping-pong判断是否节点可以连接上。如果有一半以上的节点去ping一个节点的时候没有回应，集群就认为这个节点宕机了，然后去连接它的备用节点。如果某个节点和所有从节点全部挂掉，我们集群就进入faill状态。还有就是如果有一半以上的主节点宕机，那么我们集群同样进入faill状态。

## （8）redis实现商品秒杀

a.在秒杀前，为了避免大量人员访问，使页面处于静态化，将商品库存放入redis，下单量初始化为0；

b.商品秒杀中，仅仅放入略大于商品库存数量的用户进入，当用户下单量等于库存量时，代表商品已经抢完，如发生订单取消时，再恢复库存供他人继续抢购。

## （9）redis和memcahsed区别

1、 Redis和Memcache都是将数据存放在内存中，都是内存数据库。不过memcache还可用于缓存其他东西，例如图片、视频等等。  
2、Redis不仅仅支持简单的k/v类型的数据，同时还提供list，set，hash等数据结构的存储。  
3、[虚拟内存](https://www.baidu.com/s?wd=%E8%99%9A%E6%8B%9F%E5%86%85%E5%AD%98&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "_blank)--Redis当物理内存用完时，可以将一些很久没用到的value 交换到磁盘  
4、过期策略--memcache在set时就指定，例如set key1 0 0 8,即永不过期。Redis可以通过例如expire 设定，例如expire name 10  
5、分布式--设定memcache集群，利用magent做一主多从;redis可以做一主多从。都可以一主一从  
6、存储数据安全--memcache挂掉后，数据没了；redis可以定期保存到磁盘（持久化）  
7、[灾难恢复](https://www.baidu.com/s?wd=%E7%81%BE%E9%9A%BE%E6%81%A2%E5%A4%8D&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao" \t "_blank)--memcache挂掉后，数据不可恢复; redis数据丢失后可以通过aof恢复  
8、Redis支持数据的备份，即master-slave模式的数据备份。

## (10) Solr搜索

Solr 是一个**开源的全文搜索**服务器，已经在众多大型的网站中使用，较为成熟和稳定。 Solr 使用 Java 语言开发，基于 Apache Lucene 实现，对Lucene进行了扩展.

1. 提供了丰富的查询语言，同时提高了可配置、可扩展性，
2. 并对查询性能进行了优化，而且提供了一个完善的功能管理界面。
3. Solr 可运行在Java的Servlet容器上，如Tomcat 、Jetty等。文档通过Http利用XML 加到一个搜索集合中。Solr查询该集合也是通过http收到一个XML/JSON响应来实现。
4. 它的主要特性包括：高效、灵活的缓存功能，垂直搜索功能，高亮显示搜索结果，支持多种输出格式（包括 XML/XSLT 和 JSON 格式），通过索引复制来提高可用性，提供一套强大Data Schema来定义字段，类型和设置文本分析，提供基于Web的管理界面等。

## (11) solr实现搜素的本质

Solr中使用全文搜索实现全文扫描的，先建立索引在对索引进行搜索的过程（全文检索）

 1.索引创建：将现实世界中的所有结构化和非结构化数据提取信息，创建索引的过程

将原文档传给分词组件，通过分词得到词元，

 2搜索索引:就是得到用户查询的请求，搜索创建的索引，然后返回结果的过程

**1. 索引过程：**

**1) 有一系列被索引文件**

**2) 被索引文件经过语法分析和语言处理形成一系列词(Term)。**

**3) 经过索引创建形成词典和反向索引表。**

**4) 通过索引存储将索引写入硬盘。**

**2. 搜索过程：**

**a) 用户输入查询语句。**

**b) 对查询语句经过语法分析和语言分析得到一系列词(Term)。**

**c) 通过语法分析得到一个查询树。**

**d) 通过索引存储将索引读入到内存。**

**e) 利用查询树搜索索引，从而得到每个词(Term)的文档链表，对文档链表进行交，差，并得到结果文档。**

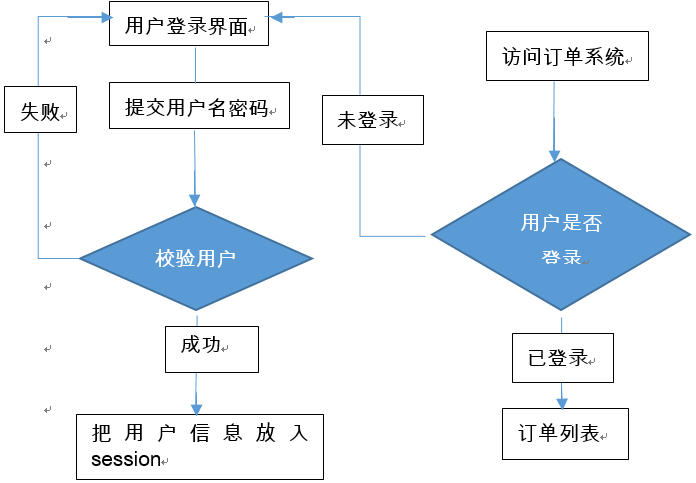
**f) 将搜索到的结果文档对查询的相关性进行排序。**

**g) 返回查询结果给用户。**

## (12)单点登陆系统

SSO英文全称Single Sign On，单点登录。SSO是在多个应用系统中，用户只需要登录一次就可以访问所有相互信任的应用系统。它包括可以将这次主要的登录映射到其他应用中用于同一个用户的登录的机制。它是目前比较流行的企业业务整合的解决方案之一。

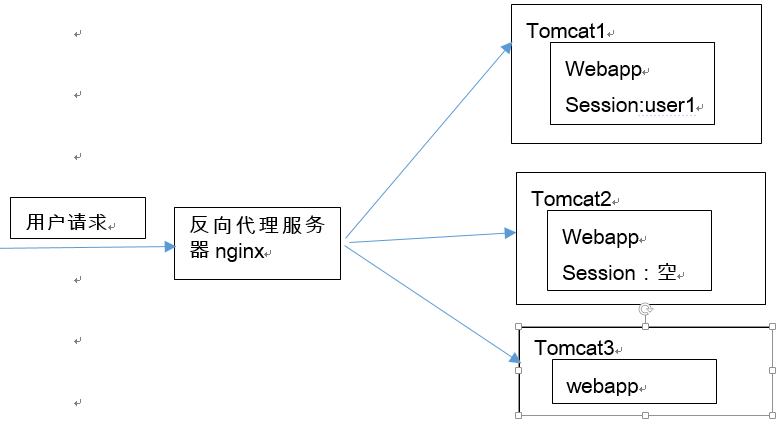
传统的登入系统流程：



**并发量高的情况下**

2000~3000并发

一个tomcat 不能满足业务需求，需要做集群。见如下：



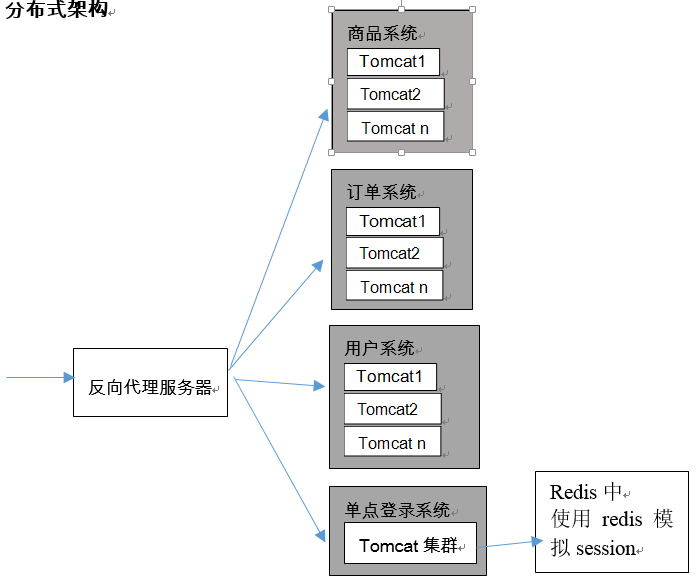
**该登入系统的问题**：当用户第一次访问服务时，tomcat1提供服务，要求用户登入并将用户信息放在他的session中。当客户再次访问时，tomcat2提供服务，此时tomcat2的session中没有用户信息，要求客户再次登入；同理当访问tomcat3时还需登入…。

这时因为每个tomcat都有一个独立的Session,每个tomcat的信息都放在自己的session中。集群环境中session不能共享

Session共享的问题：

1、tomcat做集群时，配置session复制。如果集群中节点很多，**会形成网络风暴**（每个tomcat都会把自己的ssession向集群中的所有tomcat广播）。**推荐节点数量不要超过5个**。

2、使用分布式架构。按照功能模块拆分成多个子系统。在分布式中，不同的子系统不能进行session复制，所以我们作如下操作，将登入抽离出来，使用redies模拟session.将用户信息放在redis中，然后每个用户信息都生成一个token最为redis中key，并将token保存在cookie中，这样当不同的系统切换时，我们都可以从cookie中取出token，然后调用sso服务来获取用户信息。



如图：我们将登入功能抽取出来，作为单独的子系统——sso

将用户信息放在在哪？我们使用Redis模拟session，将用户信息放在redis中并设置过期时间。（因为redis和session都可以设置过期时间，都存储key-value形式的数据）

其他系统都查询SSO发布的服务，看用户是否登入。

我们生成一个token，当做SessionId，redis中key是token，value是用户信息。

我们把token写到cookie中，在不同的系统中，我们可以到cookie中去token，然后使用该token来调用sso服务，来查询用户身份。

下面我们来创建SSO工程8084（它包含登入和注册页面）

使用的技术：Mybatis、Spring、Springmvc、Jedis

他的打包方式是war，参展rest来整合ssm

## 单点登录系统的实现

注册功能(将注册的用户保存在数据库中的tb\_user表中)、登录功能、查询功能。

这些功能都是根据开发文档实现的。

**注册功能实现：**

**数据校验接口（是用来检测注册的信息的有效性和正确性，要与**tb\_user**表中的设置一样）**

分析：要查询的表tb\_user。根据查询条件到数据库中进行查询，如果查询到结果返回false，查询结果为空返回true。使用taotaoResult包装一下返回。需要支持jsonp，根据callback判断。

**Dao层：**可以使用逆向工程。

**Service层：**接收两个参数，一个是要校验的数据，一个是数据类型。根据不同的数据类型生成不同的查询条件，到tb\_user中进行查询如果查询到结果返回false，查询结果为空返回true；使用taotaoResult包装一下返回

**Controller层：**只是发布服务。接收三个参数，一个是要校验的数据，一个数据类型，一个是callback。调用Service校验。返回json数据。若callback不为空，需要支持jsonp 。

正常直接返回调用service层方法的返回结果，即TaotaoResult.ok(boolean)

异常返回TaotaoResult.build(500, ExceptionUtil.getStackTrace(e))

**用户注册接口**

分析：使用的表tb\_user，向tb\_user表中插入数据。插入之前还需要进行数据校验，如果校验失败返回state为400，如果校验成功执行插入操作，完成后返回TaotaoResult.ok（）

**Dao层：**可以使用逆向工程。

**Service层：**接收参数，对数据进行校验，校验成功，插入数据，返回结果。

**注意：将数据插入到tb\_user时，需要对密码加密：下面使用Spirng自带的工具进行MD5加密：DigestUtils.md5DigestAsHex(usr.getPassword().g**etBytes())

参数：TbUser

返回值：正常返回TaotaoResult.ok()。

异常返回： TaotaoResult.build(400, "xxxx");

**Controller层：**接收一个表单，请求的方法为post。使用TbUser接收表单的内容，调用Service插入数据，返回。

正常直接返回调用service层方法的返回结果，即TaotaoResult.ok()

异常返回TaotaoResult.build(500, ExceptionUtil.getStackTrace(e))

**用户登录接口**

、接收用户名和密码

、校验用户名密码

、生成token，可以使用UUID

、把用户信息写入redis，key就是token

、把token写入cookie。

、返回登录成功需要把token返回给客户端。

工程中频繁用到将数据写入cookie和从cookie中取数据，我们在common工程中添加CookieUtils工具类

**Dao层:**可以使用逆向工程生成的代码。

**Service层:**接收参数：用户名、密码。校验密码是否正确，生成token，向redis中写入用户信息，把token写入cookie，返回TaotaoResult包含token。

参数：用户名、密码、HttpServletResponse、HttpServletRequest

返回值：TaotaoResult

|  |
| --- |
| @Service  **public** **class** LoginServiceImpl **implements** LoginService {  @Autowired  **private** TbUserMapper userMapper;  @Autowired  **private** JedisClient jedisClient;  @Value("${REDIS\_SESSION\_KEY}")  **private** String REDIS\_SESSION\_KEY;  @Value("${SESSION\_EXPIRE}")  **private** Integer SESSION\_EXPIRE;  @Override  **public** TaotaoResult login(String username, String password, HttpServletRequest request,  HttpServletResponse response) {  //1校验用户名密码是否正确  TbUserExample example = **new** TbUserExample();  Criteria criteria = example.createCriteria();  criteria.andUsernameEqualTo(username);  List<TbUser> list = userMapper.selectByExample(example);  //取用户信息  **if** (list == **null** || list.isEmpty()) {  **return** TaotaoResult.*build*(400, "用户名或密码错误");  }  TbUser user = list.get(0);  //校验密码  **if**(!user.getPassword().equals(DigestUtils.*md5DigestAsHex*(password.getBytes()))) {  **return** TaotaoResult.*build*(400, "用户名或密码错误");  }  //登录成功  String token = UUID.*randomUUID*().toString();//生成token  //把用户信息写入**redis**，**key:**REDIS\_SESSION:{TOKEN},**value:**user转json  //所以我们需要将user序列化，未了安全，序列化之前，应将密码清空  user.setPassword(**null**);  jedisClient.set(REDIS\_SESSION\_KEY + ":" + token, JsonUtils.*objectToJson*(user));  jedisClient.expire(REDIS\_SESSION\_KEY + ":" + token, SESSION\_EXPIRE); //设置session的过期时间  CookieUtils.*setCookie*(request, response, "TT\_TOKEN", token); //把token写入cookie  **return** TaotaoResult.*ok*(token);  }  } |

**Controller层：**请求的url：/user/login，请求方式是post

接收参数：username、password

调用Service，返回taotaoResult对象。

响应json数据。

**通过token查询用户信息（**当我们从sso系统跳转到其他系统时，只能根据cookie中的token来判断用户是否登入）

**Session有一个特点：当我们设置他的过期时间是30min,若我们一直不动他，30min后，他会后期，若在期间我们刷新了页面或访问了页面的一些功能…，他将重新开始计时，所以redis也应该这样。**

**分析：**根据token到redis查询用户信息，如果用户信息不存在说明session已经过期，返回400并提示用户session已经过期。如果查询到用户，返回用户信息，并且更新一下用户的过期时间。

请求url：/user/token/{token}

需要支持jsonp

返回：TaotaoResult

**Dao层**：使用我们写好的JedisClient

**Service层：**根据token查询redis，查询到结果返回用户对象，更新过期时间。如果查询不到结果，返回Session已经过期，状态码400.

参数：String token

返回值：异常TaotaoResult.build(400, "用户session已经过期");

正常 TaotaoResult.ok(user);

**Controller层:**从url中取token的内容,调用Service取用户信息,返回TaotaoResult(json数据）

参数：String token, String callback

请求的url：/user/token/{token}

**安全退出**

接收token，删除resdis中token对应的数据。返回ok。

　 超文本传输协议HTTP协议被用于在Web浏览器和网站服务器之间传递信息，HTTP协议传输的数据都是未加密的，也就是明文的，因此使用HTTP协议传输隐私信息非常不安全，为了保证这些隐私数据能加密传输，于是设计了SSL（Secure Sockets Layer）协议用于对HTTP协议传输的数据进行加密，从而就诞生了HTTPS。简单来说，HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，要比http协议安全。SSL（安全套接字层）依靠证书来验证服务器的身份，并为浏览器和服务器之间的通信加密。（）（12）HTTPS和HTTP的区别主要如下：

　1、https协议需要到ca申请证书，一般免费证书较少，因而需要一定费用。

　2、http是超文本传输协议，信息是明文传输，https则是具有安全性的ssl加密传输协议。

　3、http和https使用的是完全不同的连接方式，用的端口也不一样，前者是80，后者是443。

　4、http的连接很简单，是无状态的；HTTPS协议是由SSL+HTTP协议构建的可进行加密传输、身份认证的网络协议，比http协议安全。

http状态码：

200 ：请求成功

400：http相关错误

404 ：找不到指定路径下的服务器资源

500：服务器内部错误

## （13）内存泄漏与内存溢出，及其原因

**内存泄漏**：分配出去的内存无法回收（不再使用的对象或者变量仍占内存空间），在Java中可以看出内存泄漏是内存溢出的一种诱因，但不是唯一因素。内存泄漏就是存在一些被分配的对象（可达的，却是无用的）无法被gc回收。

**内存溢出**（outofMomeryError）：程序要求的内存超出了系统所能分配的范围（比如：栈满还入栈 出现上溢，栈空还出栈 出现下溢）

（3）内存泄露的原因：

1、长生命周期的对象持有短生命周期对象的引用，即静态集合类。例如：在static HashMap中缓存局部变量，且没清空，随时间的推移，这个map会越来越大，造成内存泄露。

2、变量不合理的作用域。

3、没有及时的将对象设置为null

3、各种连接没显示关闭。数据库连接、网络连接、IO连接，没显示的close，会造成很多对象无法回收。

4、监听器。释放对象时没有删除监听器。

（4）避免内存泄漏

1、尽早释放无用对象的引用

2、使用字符串处理，避免使用String，应大量使用StringBuffer，每一个String对象都得独立占用内存一块区域

3、尽量少用静态变量，因为静态变量存放在永久代（方法区）

4、避免在循环中创建对象

5、开启大型文件或从数据库一次拿了太多的数据很容易造成内存溢出，所以在这些地方要大概计算一下数据量的最大值是多少，并且设定所需最小及最大的内存空间值。

内存溢出的原因：

方法区内存溢出（outOfMemoryError：permgem space）方法区主要存放的是类信息、常量、静态变量等。所以如果程序加载的类过多，或者使用反射、gclib等这种动态代理生成类的技术，就可能导致该区发生内存溢出

线程栈溢出（java.lang.StackOverflowError）线程栈时线程独有的一块内存结构，所以线程栈发生问题必定是某个线程运行时产生的错误。 一般线程栈溢出是由于递归太深或方法调用层级过多导致的。

引起内存溢出的**原因**有很多种，常见的有以下几种：  
　　1.内存中加载的数据量过于庞大，如一次从数据库取出过多数据

.堆内存溢出（OutOfMemoryError）的例子：申请了很多内存，没释放；  
　　2.集合类中有对对象的引用，使用完后未清空，使得JVM不能回收；  
　　3.代码中存在死循环或循环产生过多重复的对象实体；  
　　5.启动参数内存值设定的过小；

内存溢出的**解决方案**：  
**第一步**，修改JVM启动参数，直接增加内存。(-Xms，-Xmx参数一定不要忘记加。)

**第二步**，检查错误日志，查看“OutOfMemory”错误前是否有其它异常或错误。

**第三步**，对代码进行走查和分析，找出可能发生内存溢出的位置。

十二，GC原理及优化策略

gc即垃圾收集机制,是指jvm用于释放那些不再使用的对象所占用的内存。也就是用jvm提供的垃圾收集器，对方法区和堆中的对象进行回收。可自动把不再被使用的存储空间释放掉，也就是说，GC机制可以有效地防止内存泄露以及内存溢出。

**垃圾回收器的主要作用**：分配内存；确保被引用对象的内存不被错误地回收；回收不再被引用的对象的内存空间

**优化策略：**

1.减少new对象。

2.多使用局部变量，减少使用静态变量。

3.避免使用finalize，该方法会给GC增添很大的负担；

4.如果是单线程，尽量使用非线程安全的，因为线程安全来自于同步机制，同步机制会降低性能。

5.单线程程序，能使用HashMap，就不要用HashTable。

6.尽量减少使用synchronized

7.用移位符号替代乘除号。eg：a\*8应该写作a<<3

8.对于经常反复使用的对象使用缓存；

9.尽量使用基本类型而不是包装类型，尽量使用一维数组而不是二维数组；10.尽量使用final修饰符，final表示不可修改，访问效率高

十三，Java的Exception类型

Exception类型分为Runnable异常和Check异常；对于Check异常javac强制要求程序员为这样的异常做预处理工作（使用try…catch…finally或者throws）。在方法中要么用try-catch语句捕获它并处理，要么用throws子句声明抛出它，否则编译不会通过。这样的异常一般是由程序的运行环境导致的。因为程序可能被运行在各种未知的环境下，而程序员无法干预用户如何使用他编写的程序，于是程序员就应该为这样的异常时刻准备着。如SQLException , IOException,ClassNotFoundException 等。

运行时异常包括：

NullPointerException - 空指针引用异常  
ClassCastException - 类型强制转换异常。  
IllegalArgumentException - 传递非法参数异常。  
ArithmeticException - 算术运算异常

IndexOutOfBoundsException - 下标越界异常

NumberFormatException - 数字格式异常

十四 ，数据库四种隔离级别

## 十五 Spring框架介绍

Spring框架优势：可扩展、易验证、易维护、易分工、易理解、代码复用。“低耦合高内聚”的原则。而“框架”就是用户遵守它的开发规则，就可以实现良好的模块化，避免软件开发中潜在的问题。开发人员只要在Spring框架中填充自己的业务逻辑就能完成一个模块划分清晰的系统。Spring框架主要可以分为2个核心内容：

    1、控制反转（IoC ，Inversion of Control）

    2、面向切面编程（AOP ，Aspect-Oriented Programming）

**IoC**：控制反转，将对象的创建的控制权由程序员的手动创建转变为spring容器来帮助我们实现对象的创建，只需在spring配置文件中编写对象的bean并且注入对象的属性。

AOP：是面向切面编程，是对目标类的实现增强，可理解为一个拦截器框架，通常使用JDK代理，CGLIB代理；

AOP  好处：使用aop编程，可以实现业务分工，提高代码的复用性，简化代码，使程序员的精力集中在业务的实现上，而不是重复的写相同代码。

应用场景：

AOP：如性能监测，访问控制，事务管理、缓存、对象池管理以及日志记录

IOC： 对象的注入

CAS

## 十六 OOP（面向对象）设计原则

OOP中主要采用了三大技术:封装，继承，多态。

   封装是将对象的属性与实现隐藏起来，只给使用者信息，将数据与数据的操作捆绑在一起成为类，对象只是类的一个实例化，就像人是一类群体，那么我们实实在在的自然人就是人类这个群体的实例化，有生命周期，诞生与消亡。我们程序员面对的只有一种类型：类，父类派生子类，将父类的基本数据类型或方法写在子类中，子类具有了父类的功能与属性这样就实现了累的继承，简单理解就是“你怎么跟你爹长的这么像”，这就是继承。多态是指在一个类型或多个类中让方法可以使用相同的名字，这样该方法不同数据类型的参数是传递的类型不一样，这是多态的实现。多态的一个重要特性就是根据不同的数据类型，但是相同方法名称自动匹配相应的处理逻辑。

## 十七，抽象类和接口的区别

（单纯的语法区别只能给一半分，要说出来适用场景）；

第一点． 接口是抽象类的变体，接口比抽象类更加抽象，接口中所有的方法都是抽象的。第二点． 每个类只能继承一个抽象类，但是可以实现多个接口  
第三点．抽象类中不一定都是抽象方法，抽象的而抽象类可以实现部分方法。但是接口中 方法必须为public修饰的、抽象的不能实现具体的法。  
第四点． 接口中基本数据类型为static 而抽象类不是的。

使用场景：由于接口更加注重动作，抽象类更加注重本质，当用来描述某些事物，其中有很多属性是相同的，但是只有少数方法是大家都有但是根据个体差异发生变化的时候，此类方法定一为抽象方法； 例如：以人为例，大家都有吃饭，睡觉，学习的共性，但是在性别，个性上时就有差异，此种定一为抽象方法。所以这种情况下把人定义为一个抽象类，其子类去单继承就可以体现出个性的差异。

十八，hashcode()和equals()的关系； 在Object类中有hashcode()和equals()方法，equals()时基于==实现的，用于判断传入的对象与调用者对象是否相同，但在Math,String等类的中都重写equals()方法，用于比较前后对象的值是否相等；

Hashcode（）主要用于hashtablehe hashMap中，用与返回一个hash code值，使用hash算法，计算的时存储的物理地址，使用hash算法执行的效率很快，因此在hashtablehe hashMap中添加元素时先判断hashcode值，如若当前位置无对象存储，则直接存放该物理地址上，如若有则判断equals（）方法是否相等，不相等则添加，相等则说明对象重复。故此使用hashcode能够减小比较次数，加快查找和判断，以及储存。

HashMAP底层实现

位桶+链表+红黑树

对于HashMap及其子类而言，它们采用Hash算法来决定集合中元素的存储位置。当系统开 始初始化HashMap时，系统会创建一个长度为capacity的Entry数组，这个数组里可以存 储元素的位置被称为“桶（bucket）”，每个bucket都有其指定索引，系统可以根据其索引快速访问该bucket里存储的元素。当每个bucket只存储一个元素时，HashMap性能最好。当解决冲突而产生的链越长，性能越差。



二十，Object的常用的一些方法；

toString（）方法：将对象信息用字符串返回，默认输出对象地址；

equals()方法，判断两个对象地址是否相等，一般都被重写；

hashcode():返回该对象物理物理存储地址（哈希码值），通常和eauals()重写，确保相等的两个对象拥有相同的hash code值；

## 二十一，讲一下反射以及用到的场景；

Java反射机制：在程序的运行过程中对于其任何一个类可以知道其属性和任意方法，对于任意对象调用任意方法，这种动态获取以及动态调用的机制 机制被称为反射机制。

反射机制实现：

class对象的获取；一 ：调用getClass()方法；二：类的class属性 person.class 三：Class.forName（）；

Class<?> class=Person.getClass();

获取class对象的属性/方法和构造器 :

class对象的 属性：getDeclaredFields()

class中所有方法： getDeclaredMethods(),返回Method[]；

class的父类： getSuperclass();

…

class对象的动态生成



**反射机制的应用场景：**

逆向代码 ，例如反编译

与注解相结合的框架 例如Retrofit

单纯的反射机制应用框架 例如EventBus 2.x

动态生成类框架 例如Gson

**反射机制的优缺点：**

**优点：**

    运行期类型的判断，动态类加载，动态代理使用反射。

**缺点：**

性能是一个问题，反射相当于一系列解释操作，通知jvm要做的事情，性能比直接的java代码要慢很多。

## 排序算法

各种排序算法的思想：



冒泡排序**：O(**n^2)

冒泡排序跟选择排序一样的简单，好理解，整个过程就想气泡一样往上升，假设从小到大排序，对于给定的n个记录，从第一个记录开始依次对相邻的两个记录进行比较，当前面的记录大于后面的记录时，交换位置，进行一轮比较后，第n位上就是整个记录中最大的数，然后在对前n-1个记录进行第二轮比较，重复该过程直到进行比较的记录只剩下一个为止。

**选择排序：**

先将第一个位值上的数跟之后所有位置上的数依次进行比较，如果第一个位置上的数比第二个位置上的数大，则进行互换，然后继续将第一个位置上的数与第三个位置上的数进行比较，经过一轮的比较后，第一个位值上的数就是所有数中最小的一个；

接着将第二个位置上的数与之后所有位置上的数进行比较，同样的规则，第二轮比较结束后，第二位放的就是所有数中第二小的数；

依次往下比，直到最后一个位置结束。

插入排序**：**将一个无序序列插入到一个有序序列中。

首先将待插入数组中的第一个数作为有序数组的第一个元素，从第二个元素开始插入。

在插入过程中，反向遍历有序素组，将所有比待插入数据大的元素向移动一位，然后将待插入数据放在合适的位置。

**归并排序：**归并排序是利用归并的思想实现的排序方法，采用经典的分治算法实现，这里的分就是把问题无序的数分解成一些小的组并使其有序，而治就是将这些有序的组进行有序的组合实现整个数列有序，其是通过归并实现的**。**

**快速排序**：通过选取一个基准（理论上是可以任意取序列中的一个元素），通过基准将一个序列分割成独立的两部分，将小于基准的所有数据都和大于基准的另外一部分的所有数据分开，然后再按此方法对这两部分数据分别进行类似的排序，最终整个序列就可以实现有序，整个排序过程可以递归进行。

## 18)线程池

基本思想还是一种对象池的思想，开辟一块内存空间，里面存放了众多(未死亡)的线程，池中线程执行调度由池管理器来处理。当有线程任务时，从池中取一个，执行完成后线程对象归池，这样可以避免反复创建线程对象所带来的性能开销，节省了系统的资源。

**Executors 提供四种线程池：**

 1）newCachedThreadPool 是一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。对于执行很多短期异步任务的程序而言，这些线程池通常可提高程序性能。调用 execute() 将重用以前构造的线程（如果线程可用）。如果现有线程没有可用的，则创建一个新线程并添加到池中。终止并从缓存中移除那些已有 60 秒钟未被使用的线程。因此，长时间保持空闲的线程池不会使用任何资源。注意，可以使用 ThreadPoolExecutor 构造方法创建具有类似属性但细节不同（例如超时参数）的线程池。

 2）newSingleThreadExecutor 创建是一个单线程池，也就是该线程池只有一个线程在工作，所有的任务是串行执行的，如果这个唯一的线程因为异常结束，那么会有一个新的线程来替代它，此线程池保证所有任务的执行顺序按照任务的提交顺序执行。

 3）newFixedThreadPool 创建固定大小的线程池，每次提交一个任务就创建一个线程，直到线程达到线程池的最大大小，线程池的大小一旦达到最大值就会保持不变，如果某个线程因为执行异常而结束，那么线程池会补充一个新线程。

 4）newScheduledThreadPool 创建一个大小无限的线程池，此线程池支持定时以及周期性执行任务的需求。

JVM  
JVM是指java虚拟机，jvm的存在可以使java语言变的是与平台无关的语言，同时jvm负责堆和方法区的内存回收，无需程序员显示指定，同时java虚拟机负责类和对象的加载等功能。  
2）讲一下JVM的分代回收以及具体算法  
新生代由于其对象存活时间短，且需要经常gc，因此采用效率较高的复制算法，其将内存区分为一个eden区和两个suvivor区，eden区和survivor区的比例是8:1，分配内存时先分配eden区，当eden区满时，使用复制算法进行gc，将存活对象复制到一个survivor区，当一个survivor区满时，将其存活对象复制到另一个区中，当对象存活时间大于某一阈值时，将其放入老年代。 老年代和永久代因为其存活对象时间长，因此使用标记清除或标记整理算法

**标记清除**  
也就是每次GC，会先扫描内存区标记存活对象，而后释放未被标记的对象空间。这种算法在工作时需要停止工作线程，再进行标记清除，同时会产生内存碎片，而且jvm需要维护一个内存空间表，用于分配内存空间。   
**复制清除**   
将内存区分为两个相同大小的区域，先在A上分配对象，当其分配满后，进行标记操作，而后将标记对象按照内存顺序复制到B区域中，最后将A区域全部清除。这种方式可以有效避免内存碎片的问题，但是复制操作会产生额外的耗时，当存活对象多时并不适合使用该算法，因为复制所产生的耗时会很长。因此这种算法适合对象存活时间短的情况。同时会造成一定的内存浪费。

**标记整理**  
这种算法就是在标记清除的基础上为了避免内存碎片的产生而优化产生的算法。总的过程分为两步，分别是标记和整理。  
3）讲一下JVM的垃圾收集器，G1和CMS有啥区别？  
目前为止，JVM的垃圾收集器一共有7种，其中有三种是针对新生代，主要用的是针对新生代的复制算法，分别为serial 收集器，parNew ，pareller 收集器 ，serial收集器主要是单线程的，parnew为多线程的，parallel 为多线程，且主要为了改变吞吐量；其余四种为serial old ，paraller old cms G1 四种，主要针对老年代，serial old为单线程，paraller old 多线程；

CMS 收集器：是为了获取最短停顿时间为目标，使用标记-清除算法，是一种并发低停顿的收集器，缺点是对CPU资源敏感，会产生浮动垃圾，由于使用标记-清楚算法，会产生大量碎片。

G1收集器：是一款面向服务端性能优秀的收集器，能够充分使用CPU和多核环境下的硬件优势，可以并发并行，分带收集，空间整合，可预测的停顿等特性；  
4）讲一下一个变量从产生到结束所经历的过程，讲一下字符串常量的过程？  
  
5）讲线程安全问题产生的原因？  
对于共享资源，当进行多线程访问对共享资源进行修改更新时，就会出现线程安全的问题，比如操作数据库，一个人在读取数据的时候，另外一个人在窜改你正在读的数据  
6）讲一下乐观锁和悲观锁  
乐观锁：它假设多用户并发的事务在处理时不会彼此互相影响，各事务能够在不产生锁的情况下处理各自影响的 那部分数据。在提交数据更新之前，每个事务会先检查在该事务读取数据后，有没有其他事务又修改了该数据。如果其他事务有更新的话，正在提交的事务会进行回 滚。  
 悲观锁： 在关系数据库管理系统里，悲观锁指的是悲观并发控制。它可以阻止一个事务以影响其他用户的方式来修改数据。如果一个事务执行的操作读某行数据应用了 锁，那只有当这个事务把锁释放，其他事务才能够执行与该锁冲突的操作。  
7）乐观锁是怎么保证一致性的  
  
8）Integer和int有啥区别，integer中有哪些特殊的函数？  
int为基本数据类型，Interger 为int的引用类型，是int的包装类；例如在几何中只能用引用类型，  
bitCount（）：为了求出整数中二进制1的位数；

**highestOneBit**（）：  
9）讲一下数据库的隔离等级  
 ① Serializable (串行化)：可避免脏读、不可重复读、幻读的发生。

　② Repeatable read (可重复读)：可避免脏读、不可重复读的发生。

　③ Read committed (读已提交)：可避免脏读的发生。

　④ Read uncommitted (读未提交)：最低级别，任何情况都无法保证。  
10）说一下MVC  
 全称Model（模型）-View（视图）-Controller（控制器），这是一种开发模式，他的好处是可以将界面和业务逻辑分离。**MVC的处理过程**：首先控制器接受用户的请求，调用相应的模型来进行业务处理，并返回数据给控制器。控制器调用相应的视图来显示处理的结果。并通过视图呈现给用户。实现人机数据之间的交互  
11）说一聚簇索引和非聚簇索引的有什么不同   
索引：对 数据表中一列或多列的值进行排序的一种结构，使用索引可以快速访问数据库表中的特定信息；

 聚簇索引：实际存储的循序结构与数据存储的物理机构是一致的，所以通常来说物理顺序结构只有一种，那么一个表的聚簇索引也只能有一个，通常默认都是主键，设置了主键，系统默认就为你加上了聚簇索引，当然有人说我不想拿主键作为聚簇索引，我需要用其他字段作为索引，当然这也是可以的，这就需要你在设置主键之前自己手动的先添加上唯一的聚簇索引，然后再设置主键，总而言之，聚簇索引是顺序结构与数据存储物理结构一致的一种索引，并且一个表的聚簇索引只能有唯一的一条；

非聚簇索引：非聚簇索引记录的物理顺序与逻辑顺序没有必然的联系，与数据的存储物理结构没有关系；一个表对应的非聚簇索引可以有多条，根据不同列的约束可以建立不同要求的非聚簇索引；

13）.servlet是线程安全的吗？项目中怎么处理多个session？

Servlet 默认是单例模式，在web 容器中只创建一个实例，所以多个线程同时访问servlet的时候，Servlet是线程不安全的。

Session ：服务端执行session机制时候会生成不同的session id值，因此直径可以通过不同的id进行区分。（没搜到）

## 14）.Session 和cookie 区别 ：

  cookie和session的方案虽然分别属于客户端和服务端，但是服务端的session的实现对客户端的cookie有依赖关系的，服务端执行session机制时候会生成session的id值，  
       这个id值会发送给客户端，客户端每次请求都会把这个id值放到http请求的头部发送给服务端，而这个id值在客户端会保存下来，保存的容器就是cookie，因此当我们完全禁掉浏览器的cookie的时候，服务端的session也会不能正常使用  
15).mysql中having和where有哪些区别？

作用的对象不同。WHERE 子句作用于表和视图，HAVING 子句作用于组 。WHERE 子句不能包含聚集函数；相反，HAVING 子句总是包含聚集函数。having一般跟在group by之后，执行记录组选择的一部分来工作的。   
16). threadLocal

ThreadLocal为解决多线程程序的并发问题提供了一种新的思路。使用这个工具类可以很简洁地编写出优美的多线程程序，ThreadLocal并不是一个Thread，而是Thread的[局部变量](https://baike.baidu.com/item/%E5%B1%80%E9%83%A8%E5%8F%98%E9%87%8F/9844788" \t "_blank)。在Java的多线程编程中，为保证多个线程对共享变量的安全访问，通常会使用synchronized来保证同一时刻只有一个线程对共享变量进行操作。这种情况下可以将[类变量](https://baike.baidu.com/item/%E7%B1%BB%E5%8F%98%E9%87%8F" \t "_blank)放到ThreadLocal类型的对象中，使变量在每个线程中都有独立拷贝，不会出现一个线程读取变量时而被另一个线程修改的现象。

17) 设计模式

设计模式是软件开发人员在软件开发过程中面临的一般问题的解决方案。设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了重用代码、让代码更容易被他人理解、保证代码可靠性。设计模式使代码编制真正工程化;

18)线程池

线程池是一种管理线程的办法，由于线程的创建比较消耗性能，使用线程池就是可以方便的管理线程，也可以减少内存的消耗。

19）treeSet，红黑树

红黑树（Red Black Tree） 是一种自平衡二叉查找树，是在[计算机](https://baike.baidu.com/item/%E8%AE%A1%E7%AE%97%E6%9C%BA" \t "_blank)科学中用到的一种[数据结构](https://baike.baidu.com/item/%E6%95%B0%E6%8D%AE%E7%BB%93%E6%9E%84/1450" \t "_blank)，典型的用途是实现[关联数组](https://baike.baidu.com/item/%E5%85%B3%E8%81%94%E6%95%B0%E7%BB%84/3317025" \t "_blank)。

每个节点是红色或者黑色，根节点是黑色，每个叶子节点NIL是黑色。它的时间复杂度是O(lgn)

2.为什么要用Maven？

Maven是一个被广泛使用的聚合框架，使用Maven构建项目，便于项目的惯例，他能自动关联和下载依赖的jar包，便于对依赖jar包的升级和管理，同时maven在大型项目部署时简单快速。

3.为什么要做这个项目？

因为我比较想去网商行业，虽然这个行业竞争压力比较大，技术更新换代也比较快，但是在这个行业的人都非常优秀，在这里可以学到很多知识和技术，也有利于自己的发展，之所以选择这个项目是因为是一个比较完善的项目体系，可以将学到的框架，数据库等知识进行运用，同时也能了解到我们在购物时轻轻地一点，底层到底经历了那些操作，这都是比较有意思和有趣的。

4.这个项目让你有什么比较深的体会？

这个项目虽然有教程，但是自己一路做下来也出现很多问题，明明和别人做的一样，但是为什么会出现bug，在各种寻找出错的地方反复查找资料，看博文，最终一步步的解决，这个过程也收货了 很多，也提高了自己解决问题的能力，项目作为一个训练自己的过程，能从中得到收获才是最重要的。

5.你对什么网络技术比较感兴趣？

我对大数据比较感兴趣，我对大数据的理解就是海量和多样化的数据，尤其在现在社会互联网和手机的大范围使用，大数据一次出现的频率也越来越多，大数据数据量庞大，一般采用分布式架构，尤其对海量数据进行处理和挖掘难度较大，同样获取的有效数据也有较高价值，我觉得马云说的很对，未来的时代是数据科技时代。所以我对大数据比较感兴趣，但是由于科研任务比较重和课余时间用来学java，所以对这方面没有什么深入的了解和学习，如果有机会的话希望有机会多了解这方便的 ，更上时代的步伐。

6.SS0单点登录系统

用户只需要登录一次就可以访问所有相互信任的应用系统，当用户第一次访问应用系统的时候，因为还没有登录，会被引导到认证系统中进行登录;根据用户提供的登录信息，认证系统进行身份校验，如果通过校验，应该返回给用户一个认证的凭据--ticket;用户再访问别的应用的时候，就会将这个ticket带上，作为自己认证的凭据，应用系统接受到请求之后会把ticket送到认证系统进行校验，检查ticket的合法性。如果通过校验，用户就可以在不用再次登录的情况下访问应用系统2和应用系统3了。  
8. 你购物车存cookie里边 可以实现不登录就可以使用购物车 那么我现在没有登录把商品存购物车了 然后登录了然后我换台电脑并且登录了还能不能看见我购物车的信息？

淘淘商城现阶段使用的仅仅是把购物车的商品写入cookie中，这样服务端基本上么有存储的压力。但是弊端就是用户更换电脑后购物车不能同步。打算下一步这么实现：当用户没有登录时向购物车添加商品是添加到cookie中，当用户登录后购物车的信息是存储在redis中的并且是跟用户id相互关联的，此时你更换电脑后使用同一账号登录购物车的信息就会展示出来。

9.出现商品 秒杀，你怎么办？

# Redis，solr，tomcat等项目相关

Mvc设计模式

抽象类和接口区别

常见的排序算法，200w数据大小再【0，200】之间，如何快速排序。

操作系统有了解吗？

多线程常的关键字，synchronized 底层是什么实现的

从1到200w，找出其中的质数。

如何快速算出一个数的二进制中有多少个1？

五条跑道，25中🐏，如何找到跑的最快的3只？

2000pb数据（1pb=1024TB）,北京的数据中心的数据损坏，上海中心有备份，如何最快的时间将数据进行恢复。

7个球，其中 有一个质量较重或者较轻，有一个天平，如何快速的判断最终或者最轻的那个球。

有100个数，其中99个是从1到99，有一个重复，求出这个重复数。

Mysql如何使用索引，使用索引有什么原则

内存溢出和内存泄漏区别，如何处理

内存泄漏是指你向内存进行了系统

垃圾回收机制

Final finally finalized 区别

Final修饰类时，表明该类不能被其他类所继承，当final修饰一个方法时，把方法性锁定防止其他继承类对方法进行更改，当父类方法使用final和private修饰时，此时子类中方法名相同的方法不是继承而是重写了一个方法。Final修饰成员变量时，只能赋值时进行初始化，复制后不能改变。

Finally作为异常处理的一部分，只能和try或者try catch同用，一般情况下，finally修饰代码块一定会在return前执行，如果finally中有return值 则会覆盖方法中其他return值；但在强制退出（调用exit（）方法时）和未进入到try之前就退出时可不执行finally块中的代码。

Finalize（）方法是java.lang.object的方法，这个方法在gc时，该对象在回收的时候被调用，　一个对象的finalize()方法只会被调用一次，而且finalize()被调用不意味着gc会立即回收该对象，所以有可能调用finalize()后，该对象又不需要被回收了，然后到了真正要被回收的时候，因为前面调用过一次，所以不会调用finalize()。

**1，数据库有2000w条记录，复制到另一个服务器内，保证无重复，速度快**

**2，问了关于tomcat的东西(好像是说tomcat除了io连接还有什么连接**)

**3， Redis事务实现 集群实现**

redis的常用数据结构，用过哪些，怎么用的

redis和memorycache的区别以及各自的应用场景 , memcached默认过期时

.redis部署方式。

redis怎么加入节点

. redis持久化

秒杀场景？怎么实现。redis怎么限流，限流算法。

redis实现队列、实现优先级队列

redis说一下sorted set底层原理

**4，分布式锁:**Redis和Zookeeper两种实现 

**5.** **手写个单例。**

6,亿数字找最大100个数，不考虑内存问题，问的很细

**7，万一Redis集群某个node的weight改变了怎么办**

8，，redis数据类型 zset跳表

**,9，消息队列**

**rabbitmq队列可以连多少个消费者。**

mq如何判断消费者链接哪个队列

**10** 关注了什么技术热点，说了SpringBoot和SpringCloud，

 SpringBoot是干嘛的，说了一个引擎，内置了Tomcat

**11，一致性hash算法**

**12， 讲一下生产者消费者模式**（

13 Session和Cookie的区别和联系（说了都在在哪用的，SessionID和Cookie的关系，Session和Cookie都存放什么）

**14，**全量复制和增量复制

15，解决session不一致问题的流程图。发布锁实现的流程图。  
**16，**画出代理模式与适配器模式

17，Slice与数组区别，Slice底层结构

18，讲讲分布式或者集群

19，可以自己实现一个与java.lang.String完全相同的类吗？（答了自己的理解，扯到类加载和类相同的问题上），双亲委派类加载模型，可以加载自定义的String类吗？如何实现？   
20、分布式锁，使用场景，知道哪些实现方式。

21用什么算法从大量的IP中找出频率最高的IP的？文件非常大，容量不够？HashMap 引导---用分而治之的思想算法复杂度 哪种排序算法适合数据量比较大的排序？ 堆排序的时间复杂度

22 RPC了解么，我说了主要是协议栈+数据格式+序列化方式，然后需要有服务注册中心管理生产者和消费者，他问我注册中心宕机怎么办，我说可以做高可用，他说要问的不是这个，是想问我注册中心宕机时消费者是否能访问生产者。   
我说消费者本地有缓存，可以访问缓存中的生产者。

讲一下项目的主要架构，你在里面做了什么   
3 有什么比较复杂的业务逻辑讲一下。   
4 最大的难点是什么，收获是什么。   
9 分布式了解哪些东西，消息队列了解么，用在什么场景，说了削峰，限流和异步。说了kafka，问我怎么保证数据不丢失，以及确保消息不会被重复消费。还问了消息送达确认是怎么做的。   
10 cap了解么，分别指什么，base呢，强一致性和弱一致性有什么方法来做，2pc了解么，说一下大概过程。   
11 负载均衡怎么做的呢，为什么这么做，了解过集群雪崩么。   
12 这样一个题目，一个节点要和客户连接建立心跳检测，大概有百万数量的连接，并且会定期发送心跳包，要写一个update方法和check方法，update方法更新心跳状态，check删除超时节点，怎么做，刚开始做了个hash发现check要轮询太慢了，然后用计时器和开线程检测也不行，最后说了个LRU，他说OK的。   
关于字符串的问题，给出输出并解释原因。intern()的作用（当调用 intern 方法时，如果池已经包含一个等于此 String 对象的字符串（该对象由 equals(Object) 方法确定），则返回池中的字符串。否则，将此 String 对象添加到池中，并且返回此 String 对象的引用。）  
String str1 = "abc"; String str2 = new String("abc"); String str3 = "ab"; String str4 = "c"; String str5 = str3 + str4; System.out.println(str1 == str2); System.out.println(str1 == str5); 讲一下JvMvolatile如何保证可见性，以及指令重排是什么？synchronized与volatile的区别？

# 虚拟机：

Java堆

1）Java堆(Java Heap)是JVM所管理的最大内存区域，也是所有线程共享的一块区域，在JVM启动时创建。

2）它存放的都是对象的实例和数组。

3) Java堆是垃圾回收的主要区域，并且Java堆可以处于物理上不连续的内存空间中。

4）当前JVM对于堆的垃圾回收，采用分代收集的策略。根据堆中对象的存活周期将堆内存分为新生代和老年代。在新生代中，每次垃圾回收都有大批对象死去，只有少量存活。而老年代中存放的对象存活率高。

1. **JVM的运行是数据区域**
2. **双亲委派以及如何破坏双亲委派，还有使用双亲委派**的好处? 为什么要使用双亲委派模型？
3. **内存泄露和内存溢出有啥区别，继而引出JavaGC，Java内存溢出用什么工具调**

如何查到代码那里出现问题

jvm默认的垃圾回收器**，**垃圾回收算法

什么时候会进行Full GC

CMS回收停顿了几次，为什么要停顿两次。

Q；Java栈什么时候会发生内存溢出，Java堆呢，说一种场景，我说集合类持有对象。

M:，那集合类如何解决这个问题呢，我说用软引用和弱引用，那你讲一下这两个引用的区别吧。

1. **对象和引用之间一般用什么连接？句柄池的作用是什么。**
2. java是怎样分配对象的，新生代，老年代
3. java的内存模型
4. **虚拟机client和server的区别**
5. **Tomcat类加载器结构**
6. **平时了解哪些技术是牺牲了一致性来保证可用性的，**我说消息队列，他问我MySQL是不是，我说MySQL主从复制应该是。

**10，CAP定理说一下，为什么三者只能选二，为什么分区容忍性必须保证。**

工作中CAP的A和P是怎么选择的

1. LVS
2. 如何优化页面请求速度，
3. 设计一个高并发高可用的方案
4. 讲一下JvMvolatile如何保证可见性，以及指令重排是什么？synchronized与volatile的区别？

# 数据库

## SQL注入讲一下*（*sql注入就是本来只有我能操作数据库，只是让你输入内容就走，而你却输入命令，从而在我不知情下操作数据库*）*

**答：**所谓SQL注入，就是通过把SQL命令插入到Web[表单](https://baike.baidu.com/item/%E8%A1%A8%E5%8D%95/5380322" \t "_blank)或输入域名或页面请求的查询字符串中，最终达到欺骗服务器执行恶意的SQL命令。具体来说，它是利用现有应用程序，将（恶意的）SQL命令注入到后台数据库引擎执行的能力，它可以通过在Web表单中输入（恶意）SQL语句得到一个存在安全漏洞的网站上的数据库，而不是按照设计者意图去执行SQL语句。——通过SQL语句，实现无帐号登录，甚至篡改数据库。

根据相关技术原理，SQL注入可以分为平台层注入和代码层注入。前者由不安全的数据库配置或数据库平台的漏洞所致；后者主要是由于程序员对输入未进行细致地过滤，从而执行了非法的数据查询。

SQL注入的实例：比如在一个登录界面，如下输入可实现免帐号和密码登录：

用户名： ‘or 1 = 1--

密 码：

点登陆,如若没有做特殊处理,那么这个非法用户就很得意的登陆进去了，why?

从理论上说，后台认证程序中会有如下的SQL语句：

"SELECT \* FROM user WHERE username=' "+username+" ' AND password=' "+password+" ' ";

当输入了上面的用户名和密码，上面的SQL语句变成：

SELECT \* FROM user WHERE username=' ' or 1=1 -- AND password=' ';

**分析：**Where语句后面的username=' ' or 1=1这个条件一定会成功；然后后面的--是注释符，它将后面的语句注释，所以语句永远都能正确执行，用户轻易骗过系统，获取合法身份。

如果是执行：SELECT \* FROM user\_table WHERE username=' ' ;DROP DATABASE (DB Name) --' and password=' '将导致数据库被删除。

**解决方法：**

下面我针对JSP，说一下应对方法：

**1.（简单又有效的方法）PreparedStatement**

采用预编译语句集，它内置了处理SQL注入的能力，只要使用它的setXXX方法传值即可。原理：

sql注入只对sql语句的准备(编译)过程有破坏作用

而PreparedStatement已经准备好了,执行阶段只是把输入串作为数据处理,

而不再对sql语句进行解析,准备,因此也就避免了sql注入问题.

**2.使用正则表达式过滤传入的参数**

**3.字符串过滤**

**4.jsp中调用该函数检查是否包函非法字符**

防止SQL从URL注入：

**5.JSP页面判断代码：**

使用javascript在客户端进行不安全字符屏蔽

## 数据库中的索引

数据库索引：索引是对数据库表中**一列或多列**的值进行排序的一种结构，使用索引可快速访问数据库表中的特定信息。在mysql中索引是在存储引擎层实现的。

A)索引的分类：

按照存储方式分为：聚集索引与非聚集索引  
按照维护与管理索引角度分为：普通索引、唯一索引（主键所以）

B）建立索引的原则

为表设置索引要付出代价的：一是增加了数据库的存储空间，索引表占用的空间是数据表的1.5倍；二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。

**哪些列应该创建索引**

1）定义有主键的数据列一定要建立索引。因为主键可以加速定位到表中的某一行

2）定义有外间的数据列一定要建立索引。外键列通常用于表与表之间的连接，在其上创建索引可以加快表间的连接。

**3）**在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；

4）在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；

5）在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。

6）在经常需要搜索的列上，可以加快搜索的速度；

**哪些列不应该创建索引。**

1)对于那些在查询中很少使用或者参考的列不应该创建索引。

2)对于那些只有很少数据值的列也不应该增加索引。例如人事表的性别列

3)对于那些定义为text, image和bit数据类型的列不应该增加索引。

4)当修改性能远远大于检索性能时，不应该创建索引。

*C*）普通索引

这是最基本的索引，它没有任何限制。它有以下几种创建方式：

◆创建索引

**CREATE** **INDEX** indexName **ON** 表名(字段名(length));

如果是CHAR，VARCHAR类型，length可以小于字段实际长度；如果是BLOB和TEXT类型，必须指定 length，下同。

◆修改表结构

**ALTER** mytable **ADD** **INDEX** [indexName] **ON** (username(length)) ;

◆创建表的时候直接指定

1. **CREATE** **TABLE** mytable(
2. ID **INT** NOT NULL,
3. username **VARCHAR**(16) NOT NULL,
4. **INDEX** [indexName] (username(length))
5. );

◆删除索引的语法：

**DROP** **INDEX** [indexName] **ON** mytable;

C）主键、唯一约束、主键索引、唯一索引。

主键就是主键索引，主键索引是唯一索引的一种特殊情况.

创建唯一约束时，会自动的创建唯一索引；注意主键也有唯一约束。

1 一个表的主键索引只能有一个，而唯一索引可以建多个。  
2主键索引不可为null，唯一索引可以为null。

创建主键时，数据库引擎会自动创建唯一的索引来强制实施 PRIMARY KEY 约束的唯一性要求。如果表中不存在聚集索引或未显式指定非聚集索引，则将创建唯一的聚集索引以强制实施 **PRIMARY KEY。**

◆创建索引

**A，CREATE** **UNIQUE** **INDEX** indexName **ON** mytable(username(length))

**B,创表时指定**

1. **CREATE** **TABLE** mytable(
2. ID **INT** NOT NULL,
3. username **VARCHAR**(16) NOT NULL,
4. **UNIQUE** [indexName] (username(length))
5. );

◆添加索引

**ALTER** mytable **ADD** **UNIQUE** [indexName] **ON** (username(length))

D）全文索引：创建全文索引(FullText index)

   旧版的MySQL的全文索引只能用在MyISAM表格的char、varchar和text的字段上。

   不过新版的MySQL5.6.24上InnoDB引擎也加入了全文索引，

◆创建索引

a创建表的同时创建全文索引

             CREATE TABLE article (   
                  id INT AUTO\_INCREMENT NOT NULL PRIMARY KEY,   
                  title VARCHAR(200),   
                  body TEXT,   
                  FULLTEXT(title, body)   
              ) TYPE=MYISAM;   
  b 直接通过create index的方式

      CREATE FULLTEXT INDEX ft\_email\_name ON `student` (`name`)

◆添加索引的语法：

      ALTER TABLE `student` ADD FULLTEXT INDEX ft \_name  (`name`)    或者：

ALTER TABLE `student` ADD FULLTEXT ft\_stu\_name  (`name`)

◆删除索引的语法：

A 直接使用 drop index(注意：没有 drop fulltext index 这种用法)

      DROP INDEX full\_idx\_name ON tommy.girl ; --（能删除所有类型的引用）

B 使用 alter table的方式

     ALTER TABLE tommy.girl DROP INDEX ft\_email\_abcd;

◆使用全文索引

     跟普通索引稍有不同,使用全文索引的格式：MATCH (columnName) AGAINST ('string')

     eg:

           SELECT \* FROM `student` WHERE MATCH(`name`) AGAINST('聪')

           当查询多列数据时：

                建议在此多列数据上创建一个联合的全文索引，否则使用不了索引的。

          SELECT \* FROM `student` WHERE MATCH(`name`,`address`) AGAINST('聪 广东')

D）组合索引：在多列上创建索引。

组合索引中，列的顺序非常重要，MySQL仅能对索引最左边的前缀进行有效的查找；

查询条件中出现组合索引的第一列或者全部,则能利用组合索引；否则不能

注意：包含多个列的主键始终会自动以组合索引的形式创建索引，其列的顺序是它们在表定义中出现的顺序，而不是在主键定义中指定的顺序。在考虑将来通过主键索引执行的搜索，确定哪一列应该排在最前面。

E）聚簇索引（InnoDB）与非聚簇索引（MyISAM）

**定义：**对于一张表来说，**聚集索引**只能有一个，因为数据真实的物理存储顺序就是按照聚集索引存储的；**非聚簇索引**记录的物理顺序与逻辑顺序没有必然的联系；一个表对应的非聚簇索引可以有多条，根据不同列的约束可以建立不同要求的非聚簇索引；

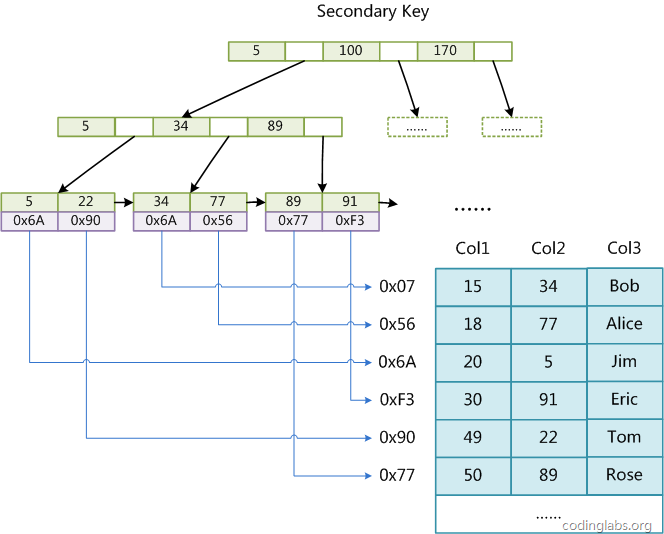
聚簇索引：索引和数据存储在一块

非聚簇索引：索引数据和存储数据是分离的。

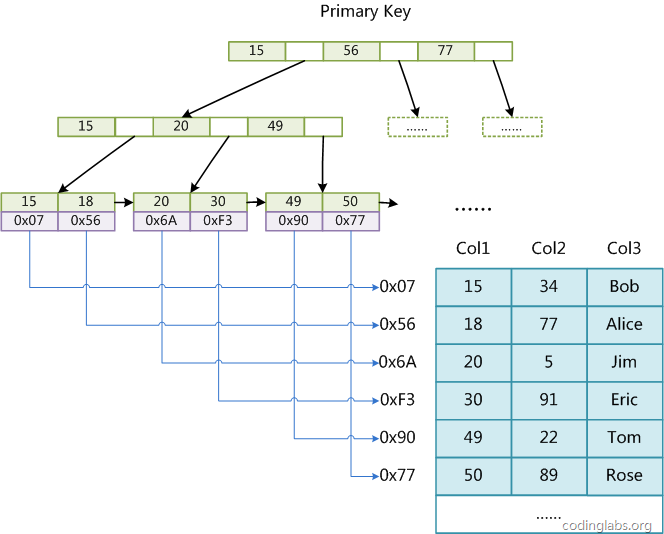
**区别与联系：**聚簇索引和非聚簇索引都是用B+树实现的。但是聚簇索引中叶子节点存储的是**一条完整的记录数据**，非聚簇索引的叶子节点中存储的是指向**主键的引用**。

F)MySQL中的MyISAM 和 InnoDB 的索引实现

* MyISAM的实现原理，利用B+树



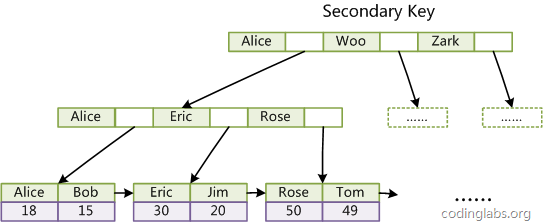
以非主键为索引(辅助索引)



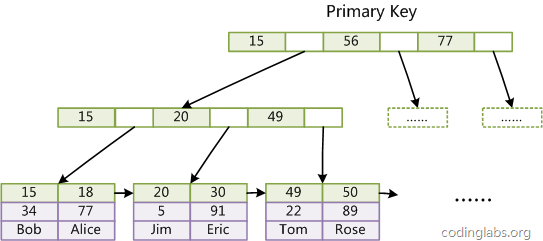
以主键为索引

在MyIsam中只有非聚餐索引，所以以主键为索引和以非主键为索引查询都是先查找到物理位置，然后再到物理位置上去寻找数据。

* **下面是InnoDB 用B+树实现索引**



以非主键为索引(辅助索引)



以主键为索引

在InnoDB中，以主键为索引的聚簇索引，他的叶子节点保存的是一条完整的记录；以非主键为索引的聚簇索引，他的叶子结点保存的是主键，检索时，先通过非主键索引树找到主键，再以主键为索引取出数据。

\*使用主索引的时候，更适合使用聚簇索引，因为聚簇索引只需要查找一次，而非聚簇索引在查到数据的地址后，还要进行一次I/O查找数据。

\*因为聚簇辅助索引存储的是主键的键值，因此可以在数据行移动或者页分裂的时候降低委会成本，因为这时不用维护辅助索引。但是辅助索引会占用更多的空间。

注意: innodb来说,   
1: 主键索引 既存储索引值,又在叶子中存储行的数据  
2: 如果没有主键, 则会Unique key做主键   
3: 如果没有unique,则系统生成一个内部的rowid做主键.  
4: 像innodb中,主键的索引结构中,既存储了主键值,又存储了行数据,这种结构称为”聚簇索引”

在建立主键的时候可以声明为CLUETERED(聚集)或NONCLUETERED(非聚集)！

 从MySQL5.6版本开始支持InnoDB引擎的全文索引，旧版的MySQL的全文索引只能用在MyISAM表格的char、varchar和text的字段上

下面是一个简单的比较表

|  |  |  |
| --- | --- | --- |
|  | **主键** | **聚集索引** |
| 用途 | 强制表的实体完整性 | 对数据行的排序，方便查询用 |
| 一个表多少个 | 一个表最多一个主键 | 一个表最多一个聚集索引 |
| 是否允许多个字段来定义 | 一个主键可以多个字段来定义 | 一个索引可以多个字段来定义 |
| 是否允许 null 数据行出现 | 如果要创建的数据列中数据存在null，无法建立主键。 创建表时指定的 PRIMARY KEY 约束列隐式转换为 NOT NULL。 | 没有限制建立聚集索引的列一定必须 not null . 也就是可以列的数据是 null 参看最后一项比较 |
| 是否要求数据必须唯一 | 要求数据必须唯一 | 数据即可以唯一，也可以不唯一。看你定义这个索引的 UNIQUE 设置。 （这一点需要看后面的一个比较，虽然你的数据列可能不唯一，但是系统会替你产生一个你看不到的唯一列） |
| 创建的逻辑 | 数据库在创建主键同时，会自动建立一个唯一索引。 **如果这个表之前没有聚集索引**，同时建立主键时候没有强制指定使用非聚集索引，则建立主键时候，同时建立一个唯一的聚集索引 | 如果未使用 UNIQUE 属性创建聚集索引，数据库引擎 将向表自动添加一个四字节 uniqueifier 列。  必要时，数据库引擎 将向行自动添加一个 uniqueifier 值，使每个键唯一。此列和列值供内部使用，用户不能查看或访问。 |

下表给出了何时使用聚簇索引与非聚簇索引:

|  |  |  |
| --- | --- | --- |
| **动作** | **使用聚簇索引** | **使用非聚簇索引** |
| 列经常被分组排序 | 应 | 应 |
| 返回某范围内的数据 | 应 | 不应 |
| 一个或极少不同值 | 不应 | 不应 |
| 小数目的不同值 | 应 | 不应 |
| 大数目的不同值 | 不应 | 应 |
| 频繁更新的列 | 不应 | 应 |
| 外键列 | 应 | 应 |
| 主键列 | 应 | 应 |
|  |  |  |

聚集索引,表中存储的数据按照索引的顺序存储,检索效率比普通索引高,但对数据新增/修改/删除的影响比较大 非聚集索引,不影响表中的数据存储顺序,检索效率比聚集索引低,对数据新增/修改/删除的影响很小

在没有聚集索引的表中，最大可以设置249个非聚集索引

## 3.2.索引为什么能查的快？

数据在磁盘上是以块的形式存储的。为确保对磁盘操作的原子性，访问数据的时候会一并访问所有数据块。磁盘上的这些数据块与链表类似，即它们都包含一个数据段和一个指针，指针指向下一个节点（数据块）的内存地址，而且它们都不需要连续存储

当表中有大量记录时，若要对表进行查询，第一种搜索信息方式是全表搜索，是将所有记录一一取出，和查询条件进行一一对比，然后返回满足条件的记录，这样做会消耗大量数据库系统时间，并造成大量磁盘I/O操作；第二种就是在表中建立索引，然后在索引中找到符合查询条件的索引值，最后通过保存在索引中的ROWID（相当于页码）快速找到表中对应的记录。

## B+树和B-树的区别

**B树：又称为多路平衡（BF=0）查找树。B树中所有节点的最大孩子数目称为B的阶。通常用m表示。由于它是查找树，中序遍历B树，关键字是一个从小到大的排序序列。**

**m叉树的特性：**1：若根节点不是叶子节点，则至少有两颗子树。树中每个节点至多有m颗子树；

2,：每一个非根的非叶子结点都有k-1个元素（关键字），和k个孩子（ceil(m/2)<=k<=m）

**3：所有的叶子结点都位于同一层次**

4：所有的非叶结点的结构如上图：（n,A0,K1,A1,K2,A2,….,Kn,An）;n为节点中关键字的个数；Ki为结点中的关键字，Ai-1为节点中子树的引用.Ki<Ki+1,

Ai所指的子树中的所有结点的关键字都大于Ki但是小于Ki+1.

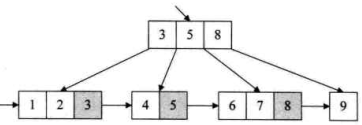
**B树的高度**：m阶，n个关键字，h=logm(n+2)(m是底数)

**B树的查找：**与二叉搜索树相似，但不是两路分支。他的查找分为两个基本的操作。1：在B树中找结点；2：在结点内找关键字。由于B树常存储在磁盘中，则前一个查找是在磁盘中进行的，而后一个查找是在内存中进行的，即找到目标结点后，将其读入内存，然后采用顺序或折半查找法查找等于k的关键字。**这是一个内存与外存的交叉过程。**

**为了减少查找过程中内存与外存的交互次数，可将B树的阶数（或结点的元素）与硬盘存储的页面大小相匹配。**

在B数中我们不经要遍历节点，还要对每个节点中的关键字进行遍历，这样非常的糟糕，所以引入了B+树。

**B+树是一种B树的变形树，**它更适合实际应用中操作系统的文件索引和数据库索引



1.其定义基本与B-树同，除了：

2.非叶子结点的子树数目与关键字个数相同；

5.所有的叶子节点中包含了全部的关键字信息，及

指向含这些关键字记录的指针，且叶子节点本身按

关键字大小从小到大顺序链接。

（非叶结点仅具有索引作用，跟记录有关的信息均存放在叶结点中。

树的所有叶结点构成一个有序链表）

由于B+树的叶子结点存储了所有数据，分支结点均为索引，方便扫库，只需要扫一遍叶子结点即可。B树因为其分支结点同样存储着数据，我们要找到具体的数据，需要进行一次中序遍历按序来扫，所以B+树更加适合在**区间查询**的情况，所以通常B+树用于数据库索引，而B树则常用于文件索引。

但是B树也有优点，其优点在于，由于**B树的每一个节点都包含key和value，因此经常访问的元素可能离根节点更近**，因此访问也更迅速。

如图：B+树有两个头指针，一个指向根节点，一个指向关键字最小的叶子节点，所以有两种查询方式。

## 数据库哈希索引 B+树索引 原理 优缺点实现

在mysql中，只有memory引擎显式支持哈希索引，这也是memory引擎表的默认索引类型.

Hash索引是基于哈希表实现，只对精确匹配所有索引列的查询才有效。存储引擎将所有的索引列根据hash函数映射到索引表中，同时保存指向对应的一条记录的指针，对于hash值相同时，采用链表解决冲突。检索时不需要类似B+树那样从根节点到叶子节点逐级查找，只需一次哈希算法即可立刻定位到相应的位置，速度非常快。

从上面的图来看，B+树索引和哈希索引的明显区别是：

* 如果是等值查询，hash索引只需要经过一次算法即可找到相应的键值；但若有hash碰撞，就需要先找到该键所在位置，然后再根据链表往后扫描，直到找到相应的数据，所以当出现大连hash碰撞，他的效率反而低；B+树索引的关键字检索效率比较平均。
* Hash索引**不支持范围查询，B+可以。**
* **哈希索引没办法利用索引完成排序，**以及like ‘xxx%’ 这样的部分模糊查询（这种部分模糊查询，其实本质上也是范围查询）；**B+可以**
* 哈希索引**不支持组合索引的最左前缀匹配规则**；**B+可以**

**注意：**

1. like %keyword    索引失效，使用全表扫描。但可以通过翻转函数+like前模糊查询+建立翻转函数索引=走翻转函数索引，不走全表扫描。   
**2. like keyword%    索引有效。**   
3. like %keyword% 索引失效，也无法使用反向索引。

## 数据库索引失效的场景

* 由于索引的最左前缀匹配原则，单独引用组合索引里非第一位置的索引列时；
* **对单字段建了索引，where条件多字段。**
* **类型错误，如**字符型字段为数字时在where条件里不添加引号.
* **对索引列运算，运算包括（+、-、\*、/、！、<>、%、like'%\_'（%放在前面）、**自动/手动类型转换，**in、exist，等），导致索引失效。**
* **索引字段使用 or 时，会导致索引失效而转向全表扫描。**注意：要想使用or，又想让索引生效，只能将or条件中的每个列都加上索引
* 如果某个数据列里包含着许多重复的值，就算为它建立了索引也不会有很好的效果。
* 1、查询谓词没有使用索引的主要边界,换句话说就是select \*，可能会导致不走索引。

## 索引的底层使用什么实现的，为什么不用红黑树

**B+树只有叶节点存放数据，其余节点用来索引，而B-树是每个索引节点都会有Data域。**  
**那么Mysql如何衡量查询效率呢？– 磁盘IO次数。** B-树/B+树 的特点就是每层节点数目非常多，层数很少，目的就是为了就少磁盘IO次数，但是B-树的每个节点都有data域（指针），这无疑增大了节点大小，说白了增加了磁盘IO次数（磁盘IO一次读出的数据量大小是固定的，单个数据变大，每次读出的就少，IO次数增多，一次IO多耗时），而B+树除了叶子节点其它节点并不存储数据，节点小，磁盘IO次数就少。**这是优点之一。**   
**另一个优点是：** B+树所有的Data域在叶子节点，一般来说都会进行一个优化，就是**将所有的叶子节点用指针串起来**。这样**遍历**叶子节点就能获得全部数据，这样就能进行区间访问啦。在数据库中基于范围的查询是非常频繁的，而B树不支持这样的遍历操作。

B树相对于红黑树的区别

R-B Tree，全称是Red-Black Tree，又称为“红黑树”，它一种特殊的二叉查找树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。

**红黑树的特性**:  
**（1）每个节点或者是黑色，或者是红色。**  
**（2）根节点是黑色。**  
**（3）每个叶子节点（NIL）是黑色。 [注意：这里叶子节点，是指为空(NIL或NULL)的叶子节点！]**  
**（4）如果一个节点是红色的，则它的子节点必须是黑色的。**  
**（5）从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点。**

**注意**：  
(01) 特性(3)中的叶子节点，是只为空(NIL或null)的节点。  
(02) 特性(5)，确保没有一条路径会比其他路径长出俩倍。因而，红黑树是相对是接近平衡的二叉树。

红黑树示意图如下：

[](https://images0.cnblogs.com/i/497634/201403/251730074203156.jpg)

红黑树的应用比较广泛，主要是用它来存储有序的数据，它的时间复杂度是O(lgn)，效率非常之高。  
例如，Java集合中的[TreeSet](http://www.cnblogs.com/skywang12345/p/3311268.html)和[TreeMap](http://www.cnblogs.com/skywang12345/p/3310928.html)，C++ STL中的set、map，以及Linux虚拟内存的管理，都是通过红黑树去实现的。

**AVL 数和红黑树基本都是存储在内存中才会使用的数据结构**。在大规模数据存储的时候，红黑树往往出现由于**树的深度过大**而造成磁盘IO读写过于频繁，进而导致效率低下的情况。为什么会出现这样的情况，我们知道要获取磁盘上数据，必须先通过磁盘移动臂移动到数据所在的柱面，然后找到指定盘面，接着旋转盘面找到数据所在的磁道，最后对数据进行读写。磁盘IO代价主要花费在查找所需的柱面上，树的深度过大会造成磁盘IO频繁读写。根据**磁盘查找存取的次数往往由树的高度所决定**，所以，只要我们通过某种较好的树结构减少树的结构尽量减少树的高度，B树可以有多个子女，从几十到上千，可以降低树的高度。

**数据库系统的设计者巧妙利用了磁盘预读原理**，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入。为了达到这个目的，在实际实现B-Tree还需要使用如下技巧：每次新建节点时，直接申请一个页的空间，这样就保证**一个节点物理上也存储在一个页里**，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。

最后让我根据ip设计数据库字段，并设计索引.要求可以很快查到一个ip和某个范围的ip,

我当时想到的是把ip转化为整数，然后用这个整数建索引，但面试官说这个是解决了第一个问题，第二个问题没解决，让我再想想，最后也没想出来

**10, ，MySQL集群的主从复制怎么做的，具体有哪些线程做哪些事情，使用了哪些日志。**

1). 主：binlog线程——记录下所有改变了数据库数据的语句，放进master上的二进制日志binlog中；

2). 从：io线程——在使用start slave 之后，负责从master上拉取 binlog 内容，放进 自己的中继日志relay log中；

3.) 从：sql执行线程——执行relay log中的语句；

## mysql写数据的时候，需要先将数据写到buffer里，再写到磁盘里，万一MySQL这时候突然挂了，怎么办？

## 如果mysql直接将数据写到磁盘里会怎样

## 问了数据库优化的东西

1. 减少查询字段数  
2. 表关联尽量用主键  
3. 查询条件尽量避免模糊查询  
4. 避免使用排序字段，排序字段尽量使用主键  
5. 尽量使用限制查询条件

6. 查询条件使用有效索引

MySQL中InnoDB引擎的行锁是通过加在什么上完成(或称实现)的？为什么是这样子的？  
答：InnoDB是基于索引来完成行锁  
例: select \* from tab\_with\_index where id = 1 for update;  
for update 可以根据条件来完成行锁锁定,并且 id 是有索引键的列,  
如果 id 不是索引键那么InnoDB将完成表锁,,并发将无从谈起

1） 表级锁：开销小，加锁快；不会出现死锁；锁定粒度大，发生锁冲突的概率最高，并发度最低。  
2） 行级锁：开销大，加锁慢；会出现死锁；锁定粒度最小，发生锁冲突的概率最低，并发度也最高。

**3，数据库**

“Where” 是一个约束声明，使用Where来约束来之[数据库](http://lib.csdn.net/base/mysql" \t "_blank" \o "MySQL知识库)的数据，Where是在结果返回之前起作用的，且Where中不能使用聚合函数。

“Having”是一个过滤声明，是在查询返回结果集以后对查询结果进行的过滤操作，在Having中可以使用聚合函数。

**4，数据库优化、以及优化慢查询的过程、sql优化等**

语句优化，索引优化和失效，和优化锁对象，优化磁盘IO等方面

MySQL的sql优化有索引优化。 他问我如何做一条sql的优化，说了慢日志和explain。

怎么查看是哪一条SQL执行慢（这个之前在面拼多多的时候也被问到过，但楼主没去看，被自己不善于总结坑了），slow\_query\_log，long\_query\_time，slow\_query\_log\_file，log\_queries\_not\_using\_indexes这个命令大家可以去看下

**5，数据库事务隔离级别,** 脏读，不可重复读，幻读

**6，写一个 Java 事务？**

8，数据库分页怎么实现

9、用Mysql查询慢的时候怎么办呢？设计表结构、建索引、分表、分库啥的...

10、看你做过登录系统，那给你存10W个username，你用一个username去查，怎么查？ 最开始我以为是存数据库，后来他说要存数据库不好， 读写太慢了，要直接存到内存，问用什么数据结构存，一脸懵逼，然后他提示说想一想集合， 比如用hashset......,不知道怎么回，说回去看看hashset吧.... 其实问的挺基础的，整个过程关于hashset和数据库优化的没怎么答出来，其他的都多多少

12、一个表 A B C字段 index（A,B,C） select \* from table where A= xx and B>0 order by C 走索引嘛?

事实上走不走索引，关键取决于where后面包含还是不包含

创建聚集索引的时候的**第一个字段**

创建非聚集索引的时候的**第一个字段**

跟select \*没有关系的，select \* 最大的影响就是**额外的IO开销**

13 mysql集群原理

14，count(\*)和count(1)的区别？（讲道理，这个我真不知道）

count(\*)对行的数目进行计算，包含NULL  
count(column)对特定的列的值具有的行数进行计算，不包含NULL值。

count(\*)和count(1)其实并没有区别，这两者都会将表中所有行都算进来，也就是该表的总行数.

A ：从server返回给client的数据是多个包组成的，客户端必须完全接收整个结果。不能接收一部分包就让server停止发送。所以尽量少发数据，减少通信数据包。

多路IO？为何epoll效率高？epoll优点？ET方式需要注意的问题？

# 多线程

1.Synchronized和volatitle的区别

a. Volititle是线程同步的轻量级实现，性能优于synchronized，volatitle只能修饰变量，synchronized只能修饰方法以及代码块。

b.多线程在执行volatile时不会发生阻塞，但在访问synchronized时会阻塞。

c.volatile会保证数据的可见性，但是不能保证操作的原子性，synchronized会阻塞。

d.volatile会组织重排序。

2. lock锁和synchronized锁区别

 a.lock是一个接口，而synchronized是java的一个关键字，synchronized是内置的语言实现；   
 b.synchronized在发生异常时候会自动释放占有的锁，因此不会出现死锁；而lock发生异常时候，不会主动释放占有的锁，必须手动unlock来释放锁，可能引起死锁的发生。（所以最好将同步代码块用try catch包起来，finally中写入unlock，避免死锁的发生。）   
 c.lock等待锁过程中可以用interrupt来终端等待，而synchronized只能等待锁的释放，不能响应中,synchronized释放锁有两种可能一种是执行完synchronized修饰里面的代码块，另外一种就是发生异常，如若synchronzied方法执行过程中线程sleep或者阻塞，其他线程需要一直等待，会非常影响效率；而使用lock可以通过设置等待时间返回避免此问题；   
 d.lock可以通过trylock来知道有没有获取锁，而synchronized不能；   
 e.Lock可以提高多个线程进行读操作的效率。（可以通过readwritelock实现读写分离）

3..ThreadLocal和同步机制的使用场景分析

ThreadLocal会为每一个线程提供一个独立的变量副本，从而隔离了多个线程对数据的访问冲突。因为每一个线程都拥有自己的变量副本，从而也就没有必要对该变量进行同步了。

当然ThreadLocal并不能替代同步机制，两者面向的问题领域不同。同步机制是为了同步多个线程对相同资源的并发访问，是为了多个线程之间进行通信的有效方式；而ThreadLocal是隔离多个线程的数据共享，从根本上就不在多个线程之间共享资源(变量)，这样当然不需要对多个线程进行同步了。**所以，如果你需要进行多个线程之间进行通信，则使用同步机制；如果需要隔离多个线程之间的共享冲突，可以使用ThreadLocal，这将极大地简化你的程序，使程序更加易读、简洁。**

　 首先，在每个线程Thread内部有一个ThreadLocal.ThreadLocalMap类型的成员变量threadLocals，这个threadLocals就是用来存储实际的变量副本的，键值为当前ThreadLocal变量，value为变量副本（即T类型的变量）。  
 初始时，在Thread里面，threadLocals为空，当通过ThreadLocal变量调用get()方法或者set()方法，就会对Thread类中的threadLocals进行初始化，并且以当前ThreadLocal变量为键值，以ThreadLocal要保存的副本变量为value，存到threadLocals。  
  
  
通过源代码可以看到每个线程都可以独立修改属于自己的副本而不会互相影响，从而隔离了线程和线程.避免了线程访问实例变量发生安全问题. 同时我们也能得出下面的结论：   
  
（1）ThreadLocal只是操作Thread中的ThreadLocalMap对象的集合；  
  
（2）ThreadLocalMap变量属于线程的内部属性，不同的线程拥有完全不同的ThreadLocalMap变量；  
  
（3）线程中的ThreadLocalMap变量的值是在ThreadLocal对象进行set或者get操作时创建的；  
  
（4）使用当前线程的ThreadLocalMap的关键在于使用当前的ThreadLocal的实例作为key来存储value值；  
  
（5） ThreadLocal模式至少从两个方面完成了数据访问隔离，即纵向隔离(线程与线程之间的ThreadLocalMap不同)和横向隔离(不同的ThreadLocal实例之间的互相隔离)；  
  
（6）一个线程中的所有的局部变量其实存储在该线程自己的同一个map属性中；  
  
（7）线程死亡时，线程局部变量会自动回收内存；  
  
（8）线程局部变量时通过一个 Entry 保存在map中，该Entry 的key是一个 WeakReference包装的ThreadLocal, value为线程局部变量，key 到 value 的映射是通过：ThreadLocal.threadLocalHashCode & (INITIAL\_CAPACITY - 1) 来完成的；  
  
（9）当线程拥有的局部变量超过了容量的2/3(没有扩大容量时是10个)，会涉及到ThreadLocalMap中Entry的回收；  
  
对于多线程资源共享的问题，同步机制采用了“以时间换空间”的方式，而ThreadLocal采用了“以空间换时间”的方式。前者仅提供一份变量，让不同的线程排队访问，而后者为每一个线程都提供了一份变量，因此可以同时访问而互不影响

1. **Java中的锁，如何实现可重入锁。**

可重入性在我看来实际上表明了锁的分配机制：基于线程的分配，而不是基于方法调用的分配

自旋锁原理

**sychonized关键字的内存含义和Lock的区别**

synchronize用于方法和代码块，可以锁对象和类以及方法，Lock一般锁一块代码。并且Lock可以搭配condition使用。

实现原理的话，synchronized使用底层的mutex锁，需要系统调用，而Lock则使用AQS实现。

synchronized锁升级的过程，说了偏向锁到轻量级锁再到重量级锁，然后问我它们分别是怎么实现的，解决的是哪些问题，什么时候会发生锁升级。

**Synchronized 加载静态方法 和普通方法的区别？**

什么是线程安全?

.重入锁有哪些 底层实现方式

1. **CAS 并发**

CAS (compare and swap) 比较并交换，就是将内存值与预期值进行比较，如果相等才将新值替换到内存中，并返回true表示操作成功；如果不相等，则直接返回false表示操作失败。

CAS

当出现非线程安全时，使用sychronized会影响性能，所以我们可以使用java并发包原子操作类（Atomic开头），他是利用CAS原理的。CAS是比较并替换。CAS需要三个操作数：内存地址V，旧的预期值A，即将要更新的目标值B。CAS指令执行时，仅当内存地址V的值与预期值A相等时，将内存地址V的值修改位B，否则就什么也不做，整个比较并替换的操作是一个原子操作。CAS虽然保证了原子操作，但是循环时间开销很大，只能证一个共享变量的原子性并且存在ABA的问题。——他的应用是在堆内存中给对象分配内存。

什么是ABA问题？

如果内存地址V初次读取的值是A，并且在准备赋值的时候还是A，但并不能保证期间他的值一直是A，可能曾被改为B然后又变成了A，但是CAS操作会误潘伟他从来没有改变过，这就是CAS的ABA问题。Java并发包为了解决这个问题，提供了一个带有标记的原子引用类，他通过控制变量值的版本来保证CAS的正确性。因此在使用CAS之前要考虑ABA问题是否会影响程序并发的正确性，若影响，改用传统的互斥方法可能比原子类更高效。

AQS

AQS依赖同步队列（一个FIFO双向队列）来完成同步状态的管理。当前线程获取同步状态失败时，AQS会将当前线程以及等待状态等信息构造成一个节点(Node)并且将其加入到同步队列中，同时会阻塞当前线程，当同步状态释放时，会把首节点中的线程唤醒，使其再次尝试获取同步状态。  
AQS的核心是通过一个共享变量来同步状态，变量的状态由子类去维护。AQS所做的工作主要有两部分：   
 （1）线程阻塞队列的维护   
 （2）线程的阻塞与唤醒   
 共享变量的修改都是通过Unsafe类提供的CAS操作完成的。AbstractQueuedSynchronizer类的主要方法是acquire和release，它们是典型的模板方法。   
acquire方法用来获取锁，返回true说明线程获取成功继续执行，一旦返回false则线程加入到等待队列中，等待被唤醒，release方法用来释放锁。 下面这4个方法由子类去实现

java 中的锁--自旋锁

(https://blog.csdn.net/zqz\_zqz/article/details/70233767)

自旋锁的目的是为了占着CPU的资源不释放，等到获取到锁立即进行处理。如果在获取自旋锁时，没有任何执行单元保持该锁，那么将立即得到锁；如果在获取自旋锁时锁已经有保持者，那么获取锁操作将自旋在那里，直到该自旋锁的保持者释放了锁。自旋锁是一种比较低级的保护数据结构或代码片段的原始方式，这种锁可能存在两个问题：[死锁](https://baike.baidu.com/item/%E6%AD%BB%E9%94%81" \t "_blank)和过多占用[cpu](https://baike.baidu.com/item/cpu" \t "_blank)资源。

线程和进程的区别

1）、首先是定义

进程：是执行中的一段程序，即一旦程序被载入到内存中并准备执行，它就是一个进程。进程是表示资源分配的的基本概念，又是调度运行的基本单位，是系统中的并发执行的单位。

线程：单个进程中执行的每个任务就是一个线程。线程是进程中执行运算的最小单位。

2）、一个线程只能属于一个进程，但是一个进程可以拥有多个线程。多线程处理就是允许一个进程中在同一时刻执行多个任务。

3）、线程是一种轻量级的进程，与进程相比，线程给操作系统带来的创建、维护、和管理的负担要轻，意味着线程的代价或开销比较小。

4）、线程没有地址空间，线程包含在进程的地址空间中。线程上下文只包含一个堆栈、一个寄存器、一个优先权，线程文本包含在他的进程 的文本片段中，进程拥有的所有资源都属于线程。所有的线程共享进程的内存和资源。但是每个线程拥有自己的栈段， 寄存器的内容，栈段又叫运行时段，用来存放所有局部变量和临时变量。

5）、父和子进程使用进程间通信机制，同一进程的线程通过读取和写入数据到进程变量来通信。

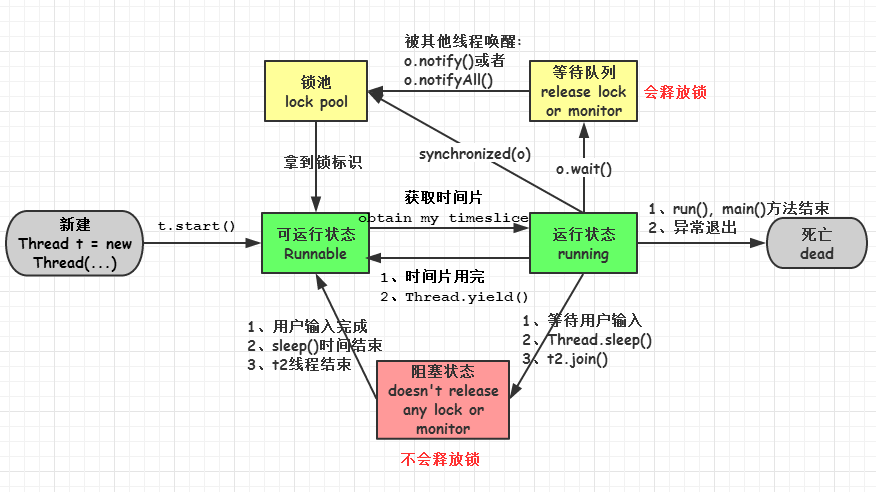
6）、进程内的任何线程都被看做是同位体，且处于相同的级别。不管是哪个线程创建了哪一个线程，进程内的任何线程都可以销毁、挂起、恢复和更改其它线程的优先权。线程也要对进程施加控制，**进程中任何线程都可以通过销毁主线程来销毁进程**，销毁主线程将导致该进程的销毁，对主线程的修改可能影响所有的线程。

7）、子进程不对任何其他子进程施加控制，进程的线程可以对同一进程的其它线程施加控制。子进程不能对父进程施加控制，进程中所有线程都可以对主线程施加控制。

**相同点：**

进程和线程都有ID/寄存器组、状态和优先权、信息块，创建后都可更改自己的属性，都可与父进程共享资源、都不鞥直接访问其他无关进程或线程的资源。

五.线程的几种状态



**新建状态**（New）

用new语句创建的线程处于新建状态，此时它和其他Java对象一样，仅仅在堆区中被分配了内存。

**就绪状态**（Runnable）——也叫作可运行状态

当一个线程对象创建后，其他线程调用它的start()方法，该线程就进入就绪状态，Java虚拟机会为它创建方法调用栈和程序计数器。处于这个状态的线程位于可运行池中，等待获得CPU的使用权。

**运行状态**（Running）

处于这个状态的线程占用CPU，执行程序代码

**阻塞状态**（Blocked）

阻塞状态是指线程因为某些原因放弃CPU，暂时停止运行。当线程处于阻塞状态时，Java虚拟机不会给线程分配CPU。直到线程重新进入就绪状态，它才有机会转到运行状态。

阻塞状态可分为以下3种：

（1等待阻塞）：当运行的线程执行了某个对象的wait()方法，JVM会把该线程放入等待队列中。

（2同步阻塞）：当运行的线程获取某个对象的同步锁时，如果该同步锁已经被其他线程占用，JVM会把这个线程放到锁池中.

(3其他阻塞)：当前线程执行了sleep()方法，或者调用了其他线程的join()方法，或者发出了I/O请求时，就会进入这个状态。当 sleep ()状态超时、 join ()等待线程终止或者超时、或者 I / O 处理完毕时，线程重新转入就绪状态

**死亡状态**（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

六．线程池**：**（https://blog.csdn.net/mine\_song/article/details/70948223）

**6.1引入线程池的原因：**

线程池的作用就是限制系统中执行线程的数量。由于创建和销毁线程都需要很大的开销

当要大量的任务时，运用线程池就可以大大的缓解这些内存开销的问题。

**6.2创建线程池需要使用 ThreadPoolExecutor 类，它的构造函数参数如下：**

public ThreadPoolExecutor**(**

int corePoolSize, //核心线程的数量

int maximumPoolSize, //线程池最大线程数量

long keepAliveTime, //超出核心线程数量后，空闲线程的存活时间

TimeUnit unit, //存活时间的单位

BlockingQueue<Runnable> workQueue, //保存待执行任务的队列

ThreadFactory threadFactory, //创建新线程使用的工厂

RejectedExecutionHandler handler // 当任务无法执行时的处理器

**) {...}**

corePoolSize：核心池的大小，这个参数跟后面讲述的线程池的实现原理有非常大的关系。在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法，从这2个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建corePoolSize个线程或者一个线程。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中；

maximumPoolSize：线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程；

keepAliveTime：表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize，即当线程池中的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0；

unit：参数keepAliveTime的时间单位，有7种取值，在TimeUnit类中有7种静态属性：

workQueue：一个阻塞队列，用来存储等待执行的任务，这个参数的选择也很重要，会对线程池的运行过程产生重大影响，。

threadFactory：线程工厂，主要用来创建线程；

handler：表示当拒绝处理任务时的策略，

**6.3任务提交给线程池之后的处理策略，这里总结一下主要有4点：**

1）如果当前线程池中的线程数目小于corePoolSize，则每来一个任务，就会创建一个线程去执行这个任务；

2） 如果当前线程池中的线程数目>=corePoolSize，则每来一个任务，会尝试将其添加到任务缓存队列当中，若添加成功，则该任务会等待空闲线程将其取出去执行；若添加失败（一般来说是任务缓存队列已满），则会尝试创建新的线程去执行这个任务；

3）如果当前线程池中的线程数目达到maximumPoolSize，则会采取任务拒绝策略进行理；

4）如果线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止，直至线程池中的线程数目不大于corePoolSize；如果允许为核心池中的线程设置存活时间，那么核心池中的线程空闲时间超过keepAliveTime，线程也会被终止。

6.4如何合理地选择或者配置线程池：

了解 JDK 提供的几种线程池实现，在实际开发中如何选择呢？

根据任务类型决定。

前面已经介绍了，这里再小节一下：

CachedThreadPool 用于并发执行大量短期的小任务，或者是负载较轻的服务器。

FixedThreadPool 用于负载比较重的服务器，为了资源的合理利用，需要限制当前线程数量。

SingleThreadExecutor 用于串行执行任务的场景，每个任务必须按顺序执行，不需要并发执行。

ScheduledThreadPoolExecutor 用于需要多个后台线程执行周期任务，同时需要限制线程数量的场景。

自定义线程池时，如果任务是 CPU 密集型（需要进行大量计算、处理），则应该配置尽量少的线程，比如 CPU 个数 + 1，这样可以避免出现每个线程都需要使用很长时间但是有太多线程争抢资源的情况；

如果任务是 IO密集型（主要时间都在 I/O，CPU 空闲时间比较多），则应该配置多一些线程，比如 CPU 数的两倍，这样可以更高地压榨 CPU。

为了错误避免创建过多线程导致系统奔溃，建议使用有界队列。因为它在无法添加更多任务时会拒绝任务，这样可以提前预警，避免影响整个系统。

执行时间、顺序有要求的话可以选择优先级队列，同时也要保证低优先级的任务有机会被执行。

七．多线程间协作与通信方式

　线程与线程之间不是相互独立的个体，它们彼此之间需要相互通信和协作，最典型的例子就是生产者-消费者问题。当队列满时，生产者需要等待队列有空间才能继续往里面放入商品，而在等待的期间内，生产者必须释放对临界资源（即队列）的占用权。因此，一般情况下，当队列满时，会让生产者交出对临界资源的占用权，并进入挂起状态。同样地，当队列为空时，消费者也必须等待，等待生产者通知它队列中有商品了。这种互相通信的过程就是线程间的协作。（**wait/notify 机制**）

**synchronized+notify+wait+flag**

这种方案是通过对两个线程中分别要执行的方法加锁synchronized，保证每次执行执行时不被双方打断，同时采用了对象object的notify和wait来实现线程之间的通信。当一个方法执行让另一个方法等待；

缺点：当存在多个线程同时等待时，只能通过notifyAll来通知所有的线程，不够灵活。

**lock+condition+flag**

使用 lock锁的方式，同时condition可以非常灵活的去控制线程与线程之间的通信。因为在一个类中可以创建多个condition的实例，我们可以通过condition不同的实例的signal和await方法来标识不同的两个线程之间相互通信的标识，而不是统一使用object的notify和wait方法了。

Synchronized vs Lock



ReentrantLock优点

可以添加多个检控条件, 如果使用synchronized,则只能使用一个. 使用 reentrant locks 可以有多个wait()/notify() 队列. [译注:直接多new 几个ReentrantLock就可以了,不同的场景/条件用不同的ReentrantLock ]

可以控制线程得到锁的顺序,也就是有公平锁(按照进入顺序得到资源),也可以不按照顺就像.synchronized 一样.

可以查看锁的状态, 锁是否被锁上了.

可以查看当前有多少线程再等待锁.

reentrantlock的缺点

需要使用import 引入相关的Class

不能忘记在finally 模块释放锁,这个看起来比synchronized 丑陋

synchronized可以放在方法的定义里面, 而reentrantlock只能放在块里面. 比较起来, synchronized可以减少嵌套

从用法上可以看出，与synchronized相比， ReentrantLock就稍微复杂一点。因为必须在finally中进行解锁操作，如果不在 finally解锁，有可能代码出现异常锁没被释放

八．Http和https的区别（https://www.cnblogs.com/wqhwe/p/5407468.html）

**4，讲了所有进程间通信的方法和优缺点，适用范围**

**5，AQS中线程等待怎么做的（线程挂起状态怎么做的）**

6，线程interrupt和stop方法区别

7、**问了一些juc并发库有哪些类**（futrue，futruetask，excutors，executorthreadpool，countdownlatch，cyclicbarrier，symphone等）

cyclicbarrier和countdownlatch有什么区别，以及应用场景。

8，volatile了解吗，写个volatile的应用场景（我就说了单列的双重检验）,  
9、什么情况下会有并发问题、怎么解决  
10,原子性和可见性

11.手写消费者生产者伪代码

12自增（两个加号）的线程安全性

1. 使用过concurrent包下的哪些类，使用场景等等。
2. 解释synchronized和valatile关键字，有什么区别，如何理解valatile保证内存有序性（答了自己的理解）

# Java基础

栈与队列区别：

栈是先进后出，队列是先进先出

1.都是线性结构。

3.都可以通过顺序结构和链式结构实现。、

4.插入与删除都只能在一端进行，且时间复杂度都是O（1）

栈与队列的不同点：

1.删除数据元素的位置不同，栈的删除操作在表尾进行，队列的删除操作在表头进行。

2.应用场景不同；栈用于递归实现，深度优先遍历等；

队列用于广度优先搜索遍历等。

HashMAP底层实现

在JDK1.6，JDK1.7中，HashMap采用位桶+链表实现，当出现hash冲突的时候，使用链表解决冲突，也就是说同一hash值的元素储存在同一链表中。当处于同一桶中元素较多时，查找效率会变慢，在JDK1.8中HashMap采用了位桶+链表+红黑树实现，当链表长度超过阈值（8）时，将链表转换为红黑树，这样大大减少了查找时间。

**2.1为什么会出现hash冲突**

在hash code计算时，hash列表的长度有限，但是具体存放的对象个数一般远远大于hash表的长度，因此对于两个不相等的KEY值，其hashcode值也会相同。

**2.4如何线程安全的hashMap**

有三种方式hashtable，Conllections.SynchronizedMap,ConcurrentHashMap

Hashtable（同步方法）， SynchronizedMap（同步方法块）都是通过对修改集合的操纵使用Synchronized锁来保证安全性的，性能比较低。

ConcurrentHashMap

1）jdk1.7中它的底层是：**hashEntry+segment.**它是使用分段锁来保证线程安全的。将分数据分割成后存储，然后给没一段数据配置一把锁。默认是16段。

扩容：只对某个segment进行扩容，不会对整个容器进行扩容

ConcurrentHashMap中的get方法没有加锁，因为他将get方法总要使用的共享变量都声明为**volatile。使得这些变量能够在线程之间保持可见性，能够被多个线程同时读，但是只能被单线程写**（除非写入的值不依赖原始的值）

2）1.8中的底层实现是：**和hashmap的底层一样，但是 使用了CAS+Synchronized（同步方法块）**，来保证线程安全的

**1，Java中的concurrenthashmap的实现原理以及内部细节**

**Java的HashMap在jdk1.7和jdk1.8的实现**

介绍线程安全的list（copyonwritelist）

线程安全的map，具体是怎么实现的，与Hashtable什么区别。

hashMap的get和put

在并发的时候，hashmap会出现死锁，有没有考虑过为什么？ 为什么ConcurrentHashMap会比hashtable的效率要高？

怎么获取key，怎么判断是否包含key

**2，同步 异步 非阻塞 阻塞 bio nio aio**

.AIO在Java中应用到了吗？

异步是使用的内核级线程还是用户态线程。

Java Io nio 不close会怎样

Nio具体怎么读写文件？具体的java语句

**3，java异常机制类图手写**

**4，一个try finally 块返回值是哪个**

**5，值传递 引用传递**

**6，重写 静态方法能被重写吗？**

**7，一个Map，要让value相同的视为同一个元素,咋搞？**

**8,反射能不能反射私有成员？**

**9，** jdk1.8的特性。

10、用HashMap去实现HashSet（手撕代码）

11，java如何做到平台无关  
12, 2.讲了String的底层实现，自己写一个不可变对象的策略（4个点）

13.谈一谈arraylist与linkedlist，如果一直往最后一个插数据选谁。。en，选linked。为啥不选array，因为他要扩容，那扩容会造成什么影响？？？蛤。。我说应该空间会不够。。还有呢？？还有我也不清楚。。面试官说因为根本原因arraylist是静态内存。如果一直插，需要耗费时间，并且可能会造成没内存泄露，，好吧，我回答了不会造成。。。说完这个我好像记起了堆内存，好像是这样。。  
14, 栈上的空间什么时候分配的

15静态内部类的优点是什么

# 数据结构与算法

1. **手撕字符串转int**
2. **索引 B+树和二叉树**
3. 排序算法，背包算法
4. 1**0亿个数去重，**我说用hash分片做，他说可能不均匀，然后我说了bitmap，他说那数字量更多怎么办，我说那就两个bitmap把。他说下一题吧。
5. **十亿个数找前10个最大的**，小顶堆过一遍即可。他说如果要提高效率呢，我说可以分片做堆排序再进行归并即可。他说OK。
6. 红黑树走一遍
7. 场景题：一个做题系统，十万题数的题库，有一个功能，用户做题时，可以点击下一个按钮，切换题目，做过的题目不会再出现，在保证业务和性能的前提下怎么设计实现
8. 两个数字不用第三方变量交换，我说了异或，他问还有别的吗，我说想不起来了
9. 判断一棵树是否是对称的  非递归+递归
10. 判断一个链表是否有环。
11. 限流算法 漏桶、令牌桶和计数   
    算法题：将字符串转换成人民币大写
12. .TB级别的日志文件中存储词汇，找出出现频率最高的十个 .一致性哈希算法 多模匹配算法
13. 13手写djikstra算法
14. B+ 树和AVL树在单点查询的情况下的效率区别。

# 计算机网络

**1，讲讲tcp**

https三次握手四次挥手的原理和意义，TCP/IP 三次握手具体交换哪些信息？状态，报文方面）

timewait.socket如何在Windows实现tcp udp，keepalive

TCP流量控制

TCP拥塞控制

TCP慢启动、拥塞控制、快重传快恢复；  
TCP和UDP的区别

http和tcp的区别

**2，长连接(我说只听过http1.1的长连接)，他问mq的连接是长连接吗**？(我猜是)

**3，.HTTP无状态？session和cookie怎么实现的，怎么去更新session？**

http长连接和短连接

http 请求头有哪些

HTTP完整的请求过程

.http报文请求行 请求头、请求正文

.HTTP一次请求响应时间过长，怎么分析解决？

http下载文件有什么弊端

4,输入url发生什么

**5、osi模型，每一层做了什么事情，有什么协议**

**6，** DNS服务器解析原理说一下

7访问一个网站的过程

tcp可靠传输有哪些机制来保证（应答确认，超时重传，滑动窗口  
，网络拥塞处理过程全讲了）

(11)收到IP数据报之后，它如何确定给TCP还是UDP，TCP再往上给应用层  
(12)如何知道它对应的是那种服务,http,dns？那现在在nginx（反向代理使）  
如何确定给内网的哪一个主机发送？  
(13)https和tls机制，非对称加密，为什么tls要使用非对称加密？  
这里面有对称加密吗？  
(14)说一下epoll过程？

2、ping使用了什么协议，用了哪几层网络结构？(icmp,控制报文协议)   
3、icmp为什么可以实现ping的   
4、手机连上wifi的时候是怎么获取ip的？（这个没答道，查了下好像是dhcp） 

# 操作系统

1. **孤儿进程与僵尸进程。**
2. **创建一个进程要用哪些资源**
3. 8.操作系统：说一下进程和线程,进程通信方式
4. 7 项目用到了多线程，如果线程数很多会怎么样，我说会占内存，还有就是切换线程比较频繁，他问切换线程会发生什么，应该就是CPU切换上下文，具体就是寄存器和内存地址的刷新。
5. (8)如果一个进程CPU占用太大，如何定位和调试这个进程？  
   (9)如果一个需要一个进程对外提供尽可能多的服务，但是它的cpu  
   上不去，有什么办法定位到这个进程，并解决这个问题让它的  
   cup占用上去？  
   (11)cpu上指令的地址是什么地址？逻辑的还是物理的？那它怎么映射  
   到物理内存的（地址映射过程）  
   (12)如何查看一个进程多少个线程？现在调试如果要转到某个线程  
   上去，如何查看线程信息，转到某一个线程上，查看它的堆栈？

# 框架和MVC

1. **spring事务传播机制**

spring优点、缺点

Spring bean的作用域和生命周期？

说了Spring，问我Spring中如何让A和B两个bean按顺序加载，我说依赖关系，他问怎么实现，然后说要用dependon注解，我说不了解。

**2,beanfactory和applicationcontext是什么关系，使用有什么区别。**

3，SpringAOP用过吗，答了异常处理器，怎么实现的？

4，Spring 过滤器 、拦截器、监听器的访问顺序

动态代理的实现。  
12）cglib无法增强的类。  
13）spring容器的启动过程。

Mvc设计模式

# linux

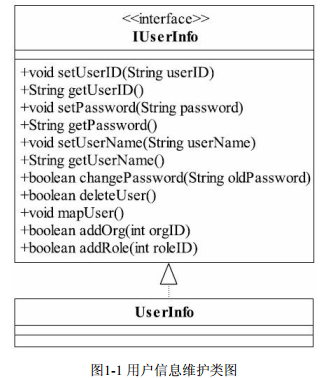
linux有哪几种方法把一个可执行文件变为任意路径下可执行  
linux显示一个路径下所有txt后缀文件

# 设计模式

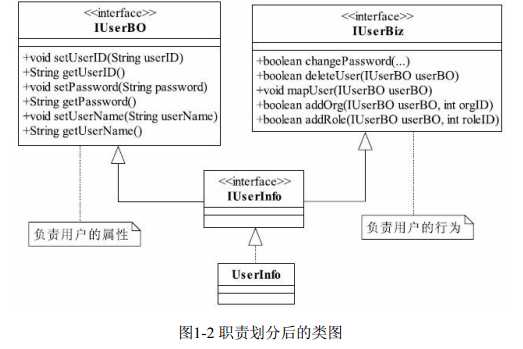
## 第一部分：6大原则

**1单一职责原则：**一个接口或类只有一个原因引起变化，也就一个接口或类只有一个职责，他就负责一件事情。

* 例一：



职责划分后的类图(SRP）

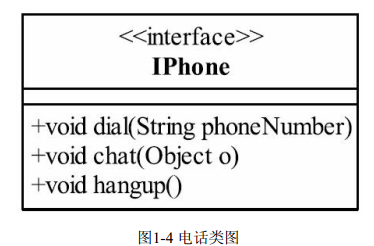


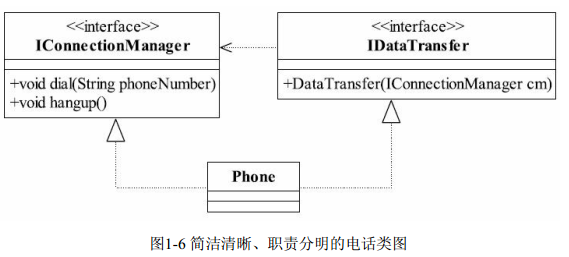
UserBiz

UserBo

如上图，左边就是未使用SRP写的接口，复杂、不易维护，写法错误。应写为右边的形式。按职责拆分为两份接口后，由于这两个接口可以独立完成他们的方法，所以要是用两个实现类。

* 例二：





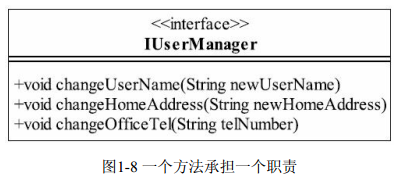
IPhone这个接口包含了两个职责：一个是数据传输（chat()）,一个是协议管理（dial()和hangup()）.所以应该分成两个接口，但是这三个方法结合在一起才完成一件事——通话，所以应使用一个类来实现这两个接口。

**注意：**对于接口，我们一定要做到单一，但是对于实现类就要多方面考虑了。

1.1单一职责原则同样也适用于方法，即一个方法尽可能做一件事。

* 例三





左边的接口严重错误，应写为右边的接口

**2里氏替换原则：**所有引用基类的地方都能透明地使用其子类对象。（反过来不行）

Java中的继承提高代码的复用性，但是增强了子类与父类的耦合性。虽然利大于弊，但是为了减少弊，可使用里氏替换原则。

**注意：**在类中调用其他类时必须使用父类（包含抽象类）或接口，否则说明类的设计已经违背里氏替换原则。（只适用于类中要用父类中已定义的方法）

但若子类不能完全实现父类的方法，或父类的某些方法在子类中已经发生“畸变”，则建议断开父子继承关系，采用依赖、聚集、组合等关系代替继承。

何为畸变？例如我们定义一个抢的抽象类AbstractGun,他有一个杀人方法shoot()。然后定义他

的实现类HandGun和MachineGun，他们都可以用来杀人。最后我们想有一把玩具枪的类ToyGun,

很明显他不能用来杀人（即不能实现shoot（））,这时ToyGun就不能继承AbstractGun，应将他

定义为一个独一的类。

当子类重载父类的方法，为了使用里氏替换原则，应该使子类中重载的方法的参数范围比父类大

子类中重载或重写父类中的方法时即，返回值的类型应小于或等于父类中的。

**3依赖倒置原则：**面向接口编程——减少耦合、提高稳定、

模块间的依赖通过抽象发生，实现类间不直接发生依赖关系，其依赖关系是通过抽象类或接口产生的。

接口或抽象类不依赖实现类。

实现类依赖接口或抽象类。

依赖倒置对并行开发的影响：两个类之间有依赖关系，只要制定出两者之间的接口（或抽象类）就可以独立开发、**独立测试，**进而保证并行开发的效率和质量。（目前见的少）

**3.1依赖的三种传递方式。**依赖是可以传递的。

interface **ICar**{

public void run();

}

**1）构造函数传递依赖对象（**构造函数注入）

interface **IDriver**{

public void driver();

}

public class **Driver** implements IDriver{

private Icar car;

public Driver(Icar icar){

this.car=car;

}

Public void driver(){

car.run();

}

}

**2)setter方法传递依赖对象(**set依赖注入)

interface **IDriver**{

public void driver();

public setCar(Icar car);

}

public class **Driver** implements IDriver{

private Icar car;

public void setCar(Icar icar){

this.car=car;

}

Public void driver(){

car.run();

}

}

**3）接口声明依赖对象（接口注入）**

在接口的方法中声明依赖对象

interface **IDriver**{

public void driver(Icar car);

}

**3.2最佳实现：尽量遵循以下原则**

1. 每个类尽量都有接口或抽象类，或者他们两者都有
2. 变量的编译类型尽量是接口或抽象类
3. 任何类都不应该从具体类中派生（非绝对）
4. 尽量不要重写基类的方法
5. 接口负责定义pubic属性和方法，并申明与其他对象的依赖关系；抽象类负责公共构造部分的实现。

4**接口隔离原则：**类间的依赖关系应该建立在最小的接口上。一个接口只服务于一个子模块或业务逻辑

接口要尽量小：这是接口隔离的核心定义，不出现臃肿的接口，但是小是有限度的——**拆分接口时，首先必须满足单一职责原则。**

接口要高内聚：接口中尽量少公布public方法

定制服务：单独为每类访问者（客户端）提供他们需要的服务。（不要讲所有客户端的需求写在一个接口中）

**5迪米特法则（最少知识原则）---解耦合**

一个类应对自己需要耦合或调用的类知道的最少。它包含如下几个含义；

朋友类的定义：出现在成员变量、方法的输入输出参数中的类称为成员朋友类。

1. 只和朋友交流。

一个类只和朋友类交流，不和陌生类（指的是自己定义的类，排除JDK API中的）交流。所以一个方法中尽量不引用一个类中没有的对象，当然，JDK API提供的类除非。

1. 朋友间也是由距离的。

一个类公开的public类越多，修改时涉及的面也就越大。因此，在设计时需反复权衡：是否还可以减少public方法和属性，是否可以修改位private,protect,default等访问权限，是否可以加上final关键字。

1. 是自己的就是自己的

如果一个方法放在本地类中，既不增加类间关系，也不对奔雷产生负面影响，那就放置在本类中。

1. 谨慎使用serializeble

**6开闭原则：**

一个软件的实体如类、模块和方法，应该对扩展开放，对修改关闭。即：一个软件的实体应该通过扩展来实现变化，而不是通过修改已有的代码来实现变化。

开闭原则并不以为着不做任何修改，一个模块的变化会对其他模块产生影响，特别是一个底层次的模块变化必然会引起高层模块的变化，因此通过扩展完成变化时，高层次的模块修改时必然的。

6.1如何使用开闭原则

开闭原则是前面的5大原则的基础。

1. 抽象约束。其包含三个含义
2. 通过接口或抽象类约束扩展，对扩展进行边界限定，不允许出现在接口或抽象类中不存在的public方法
3. 参数类型、引用对象尽量使用接口或抽象类，而不是实现类
4. 抽象层尽量保持稳定，一旦确定即不许修改。
5. 元数据控制模块行为

元数据指的是用来描述环境和数据的数据，通俗的说就是配置参数。参数可以从文件中获取，也可以从数据库中获取。

## 单例模式**：**

1. 确保某一个类只有一个实例，而且自行实例化并向这个那个系统提供提供这个实例

**基本实现思路：**

类的构造函数定义为私有方法，这样其他类就无法调用该类的构造函数来创建实例。

在该内中提供一个静态共有方法，共外部调用来得到类对象（常命名为getInstance）

注意：单例模式在多线程下需谨慎使用。如果当唯一实例尚未创建时，有两个线程同时调用创建方法，且同时没检测到同意实例的存在，导致同时各自创建一个实例。这样就违背了单例模式中实例的唯一性。解决办法是用互斥锁（虽然会降低效率）。

**1.1常见的几种单例模式**

单例模式可分为懒汉式和饿汉式。

懒汉式单例模式：在类加载时不初始化。

饿汉式单例模式：在类加载是初始化，所以类加载的比较慢，但是获取对象的速度快。

第一种：懒汉式。线程不安全的（不可用）

**public** **class** Singleton{

**private** **static** Singleton *instance*;

**private** Singleton() {};

**public** **static** Singleton getInstance() {

**if**(*instance*==**null**)

*instance*=**new** Singleton();

**return** *instance*;

}

}

在多线程环静下不能使用

第二种：懒汉式。使用同步方法，线程安全的（不推荐用）

**class** Singleton2{

**private** **static** Singleton2 *instance*;

**private** Singleton2() {}

**public** **static** **synchronized** Singleton2 getInstance() {

**if**(*instance*==**null**)

*instance*=**new** Singleton2();

**return** *instance*;

}

}

缺点：效率太低了，每个线程在想获得类的实例时候，执行getInstance()方法都要进行同步。而其实这个方法只执行一次实例化代码就够了，后面的想获得该类实例，直接return就行了。方法进行同步效率太低要改进。

第三种：懒汉式。使用同步代码块-线程安全（不可用）

**class** Singleton3 {

**private** **static** Singleton3 *instance*;

**private** Singleton3() {}

**public** **static** Singleton3 getInstance() {

**if**(*instance*==**null**) {

**synchronized**(Singleton3.**class**){

*instance*=**new** Singleton3();

}

}

**return** *instance*;

}

}

这种同步并不能起到线程同步的作用。跟第1种实现方式遇到的情形一致，假如一个线程进入了if (singleton == null)判断语句块，还未来得及往下执行，另一个线程也通过了这个判断语句，这时便会产生多个实例。

第四种：饿汉(可用)

**class** Singleton4{

**private** **static** Singleton4 *instance*=**new** Singleton4();

**private** Singleton4(){}

**public** **static** Singleton4 getInstance() {

**return** *instance*;

}

}

优点：这种写法比较简单，就是在类装载的时候就完成实例化。避免了线程同步问题。

缺点：在类装载的时候就完成实例化，没有达到Lazy Loading的效果。如果从始至终从未使用过这个实例，则会造成内存的浪费。

第五种：饿汉式(静态块)（可用）

**class** Singleton5{

**private** **static** Singleton5 *instance*;

**static** {

*instance*=**new** Singleton5();

}

**private** Singleton5() {}

**public** **static** Singleton5 getInstance() {

**return** *instance*;

}

}

这种方法与上面类似，优缺点也一样。

第六种：静态内部类（推荐用）

**class** Singleton6{

**private** Singleton6() {}

**private** **static** **class** Singleton6Instance{

**private** **static** **final** Singleton6 ***instance***=**new** Singleton6();

}

**public** **static** Singleton6 getInstance() {

**return** Singleton6Instance.***instance***;

}

}

这种方式跟饿汉式方式采用的机制类似，但又有不同。两者都是采用了类装载的机制来保证线程安全性。不同的地方在饿汉式方式是只要Singleton类被装载就会实例化，没有Lazy-Loading的作用，而静态内部类方式在Singleton类被装载时并不会立即实例化，而是在需要实例化时，调用getInstance方法，才会装载SingletonInstance类，从而完成Singleton的实例化。

优点：避免了线程不安全，延迟加载，效率高。

第七种：枚举(推荐用)

**enum** Singleton7{

***instance***;

**public** **void** whateverMethod() {

}

}

借助JDK1.5中添加的枚举来实现单例模式。不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象。

第八种：双冲检查(推荐用)

**class** Singleton8{

**private** **static** Singleton8 *instance*;

**private** Singleton8() {}

**public** **static** Singleton8 getInstance() {

**if**(*instance*==**null**) {

**synchronized**(Singleton8.**class**) {

**if**(*instance*==**null**)

*instance*=**new** Singleton8();

}

}

**return** *instance*;

}

}

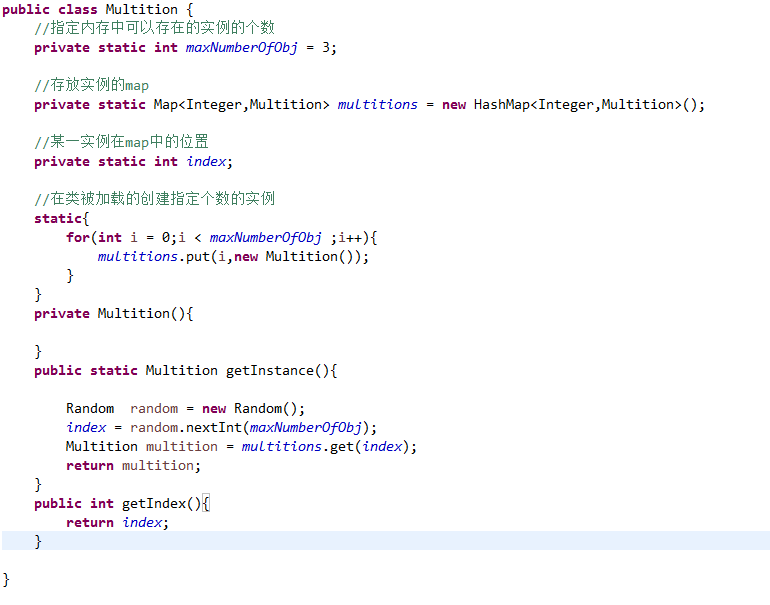
两次使用if (singleton == null)检查，可以保证线程安全。且实例化代码只用执行一次，后面再次访问时，判断if (singleton == null)，直接return实例化对象。优点：线程安全；延迟加载；效率较高。

单例模式一般无接口和抽象类：因为单列模式要“自行实例化”，且提供单一实例，接口和抽象类不能实例化。所以单例模式扩展困难。并且单例模式对单元测试不利，如果单例模式没有完成，是不能进行测试的，没有借口也不能虚拟一个对象。

对象的赋值。Java中，对象默认是不可被复制的。若实现了Cloneable接口和clone方法，则可直接通过对象赋值创建一个新的对像，且复制不需要构造方法，这使得即使使用private的构造器，还是可以使用其他类创建该类对象。解决办法是：让单例类不要实现Cloneable接口。

**1.2单例模式的扩展**

一个类产生多个对象直接用new.一个类只产生一个对象用单例模式。一个类产生固定数目该怎么办？——这种需要产生固定数目对象的模式就叫做有上限的多利模式，他是单例模式的一种扩展。



在Spring 中，每个Bean默认是单例的，这样的优点是Spring容器可以管理这些Bean的生命周期。如果采用非单例模式，则Bean初始化后的管理交由J2EE容器，Spring 不在跟踪Bean的生命周期了。

## 第8章工厂方法模式



**定义一个用于创建对象的接口（也包含抽象类），让子类决定实例化哪一个类。该方法使一个类的实例化延迟到其子类。**

**1、简单工厂模式**

* **产品类**

**abstract** **class** Human{

**public** **abstract** **void** color();

}

**class** WhiteHuman **extends** Human{

**public** **void** color() {

System.***out***.println("皮肤是白色的");

}}

**class** YelloHuman **extends** Human{

**public** **void** color() {

System.***out***.println("皮肤是黄色的");

}}

* **工厂类**

**class** HumanFactory {

**public** Human static creatHuman(String type) {

Human human=**null**;

**switch**(type){

**case** “WhiteHuman”:

human =n**ew WhiteHuman()**;**break;**

**case** “YelloHuman”**:**

human =**new YelloHuman();break;**

}

**return** human;

**}}**

* **应用场景**

**public** **class** NvWa {

**public** **static** **void** main(String[]args) {

**Human human=** **HumanFactory.creatHuman(“YelloHuman”);**

human.color();

**}}**

**2工厂方法模式**

* **产品类一样**
* **工厂类**

**//第一种形式工厂类**

**abstract** **class** AbstractHumanFactorty{

**public** **abstract** <T **extends** Human> T creatHuman(Class<T> c);

}

**class** HumanFactory **extends** AbstractHumanFactorty{

**public** <T **extends** Human> T creatHuman(Class<T> c) {

Human human=**null**;

**try** {

**human=(T)Class.forName(c.getName()).newInstance();**

} **catch** (Exception e) {

System.***out***.println("造人错误");}

**return** (T)human;

**}}**

**//第二种形式工厂类（多工厂模式）**

**abstract** **class** AbstractHumanFactorty{

**public** **abstract** Human();

}

**class** WhiteHumanFactory **extends** AbstractHumanFactorty{

**public** Human creatHuman() {

**return** new WhiteHuman();

**}}**

**class** YelloHumanFactory **extends** AbstractHumanFactorty{

**public** Human creatHuman() {

**return** new YelloHuman();

**}}**

* **对应两种应用场景**

**public** **class** NvWa {

**public** **static** **void** main(String[]args) {

AbstractHumanFactorty factorty=**new** HumanFactory();

**YelloHuman human=factorty.creatHuman(YelloHuman.class);**

human.color();

**}}**

形参类型是class<T>,则传入的实参形式为T.class——其中T要给具体的类型。

human.color();

}}

形参类型是class<T>,则传入的实参形式为T.class——其中T要给具体的类型。

**public** **class** NvWa {

**public** **static** **void** main(String[]args) {

AbstractHumanFactorty factorty=**new** YelloHumanFactory();

**YelloHuman human=factorty.creatHuman();**

human.color();

}}

human.color();

}}

形参类型是class<T>,则传入的实参形式为T.class——其中T要给具体的类型。

工厂方法模式类是一个比较实用、可扩展的框架。

例如我们若想创建一个黑种人：

第一种工厂类：只需要添加一个继承Human的实现类BlackHuman,其他地方不变。

第二种工厂类：添加一个生产黑人的工厂实现类BlackHumanFactory,然后在应用场景中稍作改变就行。

两种方法都有非常优秀的扩展性，满足开闭原则——不改变已有的工厂类接口（抽象类），就能实现。

* **简单工厂和工厂方法模式的比较**

1简单工厂最大的优点是，工厂类中包含了必要的逻辑判断，根据客户端的选择条件动态的实例化相关类，对客户端来说去除了与具体产品的依赖，遵循依赖倒置原则. create()方法通常是静态的，所以也称之为静态工厂。

但是：每添加一个产品子类，都必须在工厂类中添加一个判断分支，这违背了开放-封闭原则。——工厂模式就是为了解决这个问题而产生的。

2工厂方法模式以简单工厂模式一致，多的一点优点就是遵循了开放-封闭原则，使得模式的灵活性更强。工厂方法模式是典型的解耦合框架

**第9章：抽象工厂模式——工厂方法模式的升级版**

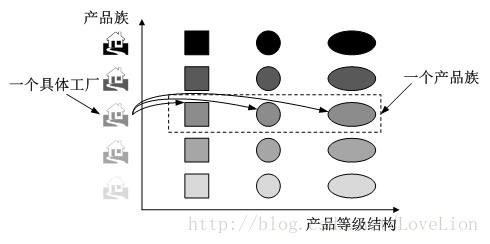
为创建一组相关或相互依赖的对象提供一个接口，而且无需指定他们的具体类。

使用抽象类不使用接口的好处。当各层类（接口、类、抽象类）的子类中包含相同

的方法时，使用抽象类可直接在当前类中实现，这样减少代码的冗余。

抽象工厂模式与工厂方法模式最大的区别在于，工厂方法模式针对的是一个产品等级结构（都是电视），而抽象工厂模式需要面对多个产品等级结构（可以是电视、冰箱、空调）。

矩形表示电视机，圆形表示冰箱、椭圆表示空调。不同颜色表示不同品牌如海尔、奥克斯、格力等



产品族和产品等级结构的关系

1. **产品等级结构**：**产品等级结构即产品的继承结构**，如一个抽象类是电视机，其子类有海尔电视机、海信电视机、TCL电视机，则抽象电视机与具体品牌的电视机之间构成了一个产品等级结构，抽象电视机是父类，而具体品牌的电视机是其子类。——产品等级个数对应工厂实现类个数。
2. **产品族**：在抽象工厂模式中，**产品族是指由同一个工厂生产的，位于不同产品等级结构中功能相关联的产品组成的家族**，如海尔电器工厂生产的海尔电视机、海尔电冰箱，海尔电视机位于电视机产品等级结构中，海尔电冰箱位于电冰箱产品等级结构中，海尔电视机、海尔电冰箱构成了一个产品族。产品族个数——抽象工厂类中的创建方法个数。

* //两个产品族：空调和电视

**abstract** **class** AbstractTelevision{

**public** **abstract** **void** doWatching();

}

**abstract** **class** AbstarctAirConditioner{

**public** **abstract** **void** doTemprature();

}

**class** AUXTelevision **extends** AbstractTelevision{

**public** **void** doWatching() {

System.***out***.println("我是奥克斯产的电视");

}}

**class** HairTelevision **extends** AbstractTelevision{

**public** **void** doWatching() {

System.***out***.println("我是海尔产的电视");

}}

**class** AUXAirConditioner **extends** AbstarctAirConditioner{

**public** **void** doTemprature() {

System.***out***.println("我是奥克斯产的空调");

}}

**class** HairAirConditioner **extends** AbstarctAirConditioner{

**public** **void** doTemprature() {

System.***out***.println("我是海尔产的空调");

}}

* //工厂抽象类，包含两个方法，用来生成不同产品等级的产品

**abstract** **class** Factory{

**public** **abstract** AbstractTelevision getInstanceT();

**public** **abstract** AbstarctAirConditioner getInstanceA();

}

//两个工厂实现类，每一个类创建一个族中的所有产品等级

**class** AUXFactory **extends** Factory{

**public** AbstractTelevision getInstanceT() {

**return** **new** AUXTelevision();

}

**public** AbstarctAirConditioner getInstanceA() {

**return** **new** AUXAirConditioner();

}}

**class** HairFactory **extends** Factory{

**public** AbstractTelevision getInstanceT() {

**return** **new** HairTelevision();

}

**public** AbstarctAirConditioner getInstanceA() {

**return** **new** HairAirConditioner();

}}

* **//应用场景**

**public** **class** DriverTest{

**public** **static** **void** main(String[]args) {

Factory factory=**new** AUXFactory();

AbstractTelevision television=factory.getInstanceT();

television.doWatching();

Factory factory2=**new** HairFactory();

AbstarctAirConditioner airConditioner=factory2.getInstanceA();

airConditioner.doTemprature();

}

}

优点：封装性好。场景中（高层模块）没有一个方法与实现类有关，对于一个产品，我们只需知道他的工厂方法就可以创建一个产品，无需关注他的实现类。

缺点：产品族扩展非常困难，需要在抽象工厂中加一个方法，然后他的两个实习那类都要修改，违背开闭原则。但是产品等级是非常好扩展的。

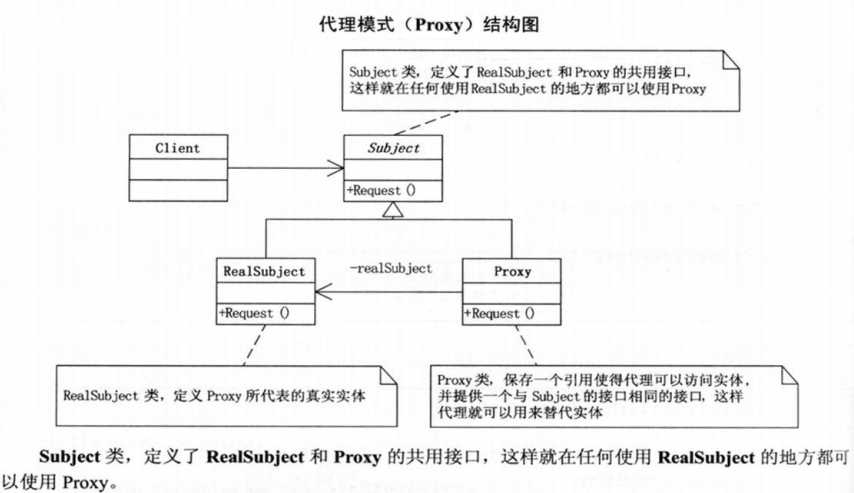
## 代理模式-委托模式

**定义**：为一个对象提供一种代理以控制对这个对象的访问—通过代理对象访问目标对象.

**好处：**可以在目标对象实现的基础上,增强额外的功能操作,即扩展目标对象的功能.

**使用的编程思想:**不要随意去修改别人已经写好的代码或者方法,如果需改修改,可以通过代理的方式来扩展该方法（开闭原则）

**关键点**:代理对象与目标对象.代理对象是对目标对象的扩展,代理对象中要包含目标对象的引用，和目标对象中的方法。



**3种代理模式**

1. **静态代理：静**态代理在使用时,需要定义接口或者父类,被代理对象与代理对象一起实现相同的接口或者是继承相同父类.

**静态代理**

* **接口**

**public** **interface** IUserDao {

**void** save();}

* **目标类**

**public** **class** UserDao **implements** IUserDao {

**public** **void** save() {

System.***out***.println("----已经保存数据!----");

}}

* **代理类**

**public** **class** UserDaoProxy **implements** IUserDao{

**private** IUserDao target; //接收保存目标对象

**public** UserDaoProxy(IUserDao target){

**this**.target=target;

}

**public** **void** save() {

System.***out***.println("开始事务...");

target.save();//执行目标对象的方法

System.***out***.println("提交事务...");

}}

* **应用场景**

**public** **class** StaticProxy{

**public** **static** **void** main(String[] args) {

IUserDao dao=**new** UserDao();

ProxyUserDao proxyUserDao=**new** ProxyUserDao(dao);

proxyUserDao.save();

}}

//静态代理的扩展

**普通代理**

**public** **interface** IUserDao {

**void** save();}

**public** **class** UserDao **implements** IUserDao {

**public** **void** save() {

System.***out***.println("----已经保存数据!----");

}}

**public** **class** UserDaoProxy **implements** IUserDao{

**private** IUserDao target; //接收保存目标对象

**public** UserDaoProxy(){

target=new UserDao();

}

**public** **void** save() {

System.***out***.println("开始事务...");

target.save();//执行目标对象的方法

System.***out***.println("提交事务...");

}}

**public** **class** StaticProxy{

**public** **static** **void** main(String[] args) {

ProxyUserDao proxyUserDao=**new** ProxyUserDao(dao);

proxyUserDao.save();

}}

* **强制代理**

**interface** IUserDao{

**void** save();

IUserDao getProxy();//新增一个方法

}

//目标类

**class** UserDao **implements** IUserDao{

**private** IUserDao proxy=**null**;

**public** **void** save() {

**if**(isProxy())

System.***out***.println("数据已经保存");

**else** System.***out***.println("请去找代理");

}

**public** IUserDao getProxy() {

**this**.proxy=**new** ProxyUserDao(**this**);

**return** proxy;

}

**private** **boolean** isProxy() {

**if**(proxy==**null**)**return** **false**;

**return** **true**;

}}

//代理类

**class** ProxyUserDao **implements** IUserDao{

IUserDao userDao;

**public** ProxyUserDao(IUserDao userDao ) {

**this**.userDao=userDao;

}

**public** **void** save() {

System.***out***.println("开启事务");

userDao.save();

System.***out***.println("关闭事务");

}

**public** IUserDao getProxy() {

**return** **this**;

}}

**public** **class** StaticProxy{

**public** **static** **void** main(String[] args) {

IUserDao dao=**new** UserDao();

IUserDao proxyUserDao=dao.getProxy();

proxyUserDao.save();

}

}

**普通代理下：**调用者只知道代理对像，而不知道目标

对象是谁，屏蔽了目标类变更对高层模块的影响。

**强制代理类：**要从目标对象查找代理对象，不允许

直接调用目标对象的方法。

**静态代理总结:**  
1.可以做到在不修改目标对象的功能前提下,对目标功能扩展.  
2.缺点:因为代理对象需要与目标对象实现一样的接口,所以一旦接口增加方法,目标对象与代理对象都要维护——可用动态代理来解决

**代理类是有个性的，一个代理类除了可以实现目标接口，还可以实现其他接口**

## 动态代理（被代理类必须要实现一个接口）

AOP的拦截功能是由java中的动态代理来实现的。就是在目标类的基础上增加切面逻辑，生成增强的目标类（该切面逻辑或者在目标类函数执行之前，或者目标类函数执行之后，或者在目标类函数抛出异常时候执行。不同的切入时机对应不同的Interceptor的种类，如BeforeAdviseInterceptor，AfterAdviseInterceptor以及ThrowsAdviseInterceptor等）。  
 AOP的源码中用到了两种动态代理来实现拦截切入功能：jdk动态代理和cglib动态代理。两种方法同时存在，各有优劣。jdk动态代理是由java内部的反射机制来实现的，cglib动态代理底层则是借助asm来实现的。总的来说，反射机制在生成类的过程中比较高效，而asm在生成类之后的相关执行过程中比较高效（可以通过将asm生成的类进行缓存，这样解决asm生成类过程低效问题）。还有一点必须注意：jdk动态代理的应用前提，必须是目标类基于统一的接口。如果没有上述前提，jdk动态代理不能应用。

**特点:**  
1.代理对象不需要实现接口  
2.代理对象的生成,是利用JDK的API,动态的在内存中构建代理对象(需要我们指定创建代理对象/目标对象实现的接口的类型)  
3.动态代理也叫做:JDK代理,接口代理

* **接口和目标实现类与静态代理一样**
* **动态代理类工厂**

**class** ProxyFactory{

//维护一个目标对象

**private** Object target;//增加通用性，可作为任何一个目标类的代理

**public** ProxyFactory(Object target) {

**this**.target=target;

}

//给目标对象生成代理对象

**public** Object getProxyInstance() {

**return** Proxy.*newProxyInstance*(

target.getClass().getClassLoader(),

target.getClass().getInterfaces(),

**new** InvocationHandler() {

**public** Object invoke(Object proxy, Method method, Object[] args)

**throws** Throwable {

System.***out***.println("开启事务");

//执行目标类中的所有方法

Object returnValue=method.invoke(target,args);

System.***out***.println("关闭事务");

**return** returnValue;

}

});

}}

**public** **class** DynamicProxy{

**public** **static** **void** main(String[] args) {

IUserDao dao=**new** UserDao();//创建目标对象

ProxyFactory proxy=**new** ProxyFactory(dao);//根据目标对象创建一个代理类工厂

IUserDao userp=(IUserDao) proxy.getProxyInstance();//从工厂中获取代理类对象

userp.save();

}}

**JDK中生成代理对象的API**：代理类所在包:java.lang.reflect.Proxy  
JDK实现代理只需要使用newProxyInstance方法,但是该方法需要接收三个参数：

**static** Object **newProxyInstance**(**ClassLoader** loader,

**Class<?>**[] interfaces,

**InvocationHandler** h )；其中三个参数的含义如下：

* ClassLoader loader,:指定当前目标对象使用类加载器,获取加载器的方法是**固定的**。
* Class<?>[] interfaces,:目标对象实现的接口的类型,使用泛型方式确认类型
* InvocationHandler h:事件处理,执行目标对象的方法时,会触发事件处理器的方法,会把当前执行目标对象的方法作为参数传入。

**总结:**  
代理对象不需要实现接口,**但是目标对象一定要实现接口**,否则不能用动态代理

**三、Cglib代理——子类代理**

静态代理和动态代理模式都是要求目标对象是实现一个接口的目标对象——基于接口。

Cglib代理是通过继承得到个子类对象从而实现对目标对象功能的扩展.——基于继承，所以当目标对象没有实现接口,就可以使用Cglib代理实现.

Cglib是一个强大的高性能的代码生成包,它可以在运行期扩展java类与实现java接口.它广泛的被许多AOP的框架使用,例如Spring AOP和synaop,为他们提供方法的interception(拦截)

Cglib包的底层是通过使用一个小块的字节码处理框架ASM来转换字节码并生成新的类.不鼓励直接使用ASM,因为它要求你必须对JVM内部结构包括class文件的格式和指令集都很熟悉.

Cglib子类代理实现方法:

1.需要引入cglib的jar文件,但是Spring的核心包中已经包括了Cglib功能,所以直接引入spring-core-3.2.5.jar即可.

2.引入功能包后,就可以在内存中动态构建子类

3.代理的类不能为final,否则报错

4.目标对象的方法如果为final/static,那么就不会被拦截,即不会执行目标对象额外的业务方法.

**class** UserDao{

**public** **void** save() {

System.***out***.println("数据已存");

}}

* 代理工厂类

**class** ProxyFactory **implements** MethodInterceptor{

**private** Object target;

**public** ProxyFactory(Object target) {

**this**.target=target;

}

**public** Object getProxyInstance() {

Enhancer en=**new** Enhancer();//工具类

en.setSuperclass(target.getClass());//设置父类

en.setCallback(**this**);//设置回调函数

**return** en.create();//创建子类——代理对象

}

**public** Object intercept(Object obj, Method method,

Object[] args, MethodProxy proxy) **throws** Throwable {

System.***out***.println("开启事务");

//执行目标对象的方法

Object returnValue=method.invoke(target, args);

System.***out***.println("提交事务");

**return** returnValue;

}}

**public** **class** CglibProxy{

**public** **static** **void** main(String[] args) {

UserDao dao=**new** UserDao();

ProxyFactory proxy=**new** ProxyFactory(dao);

UserDao userp=(UserDao) proxy.getProxyInstance();

userp.save();

}}

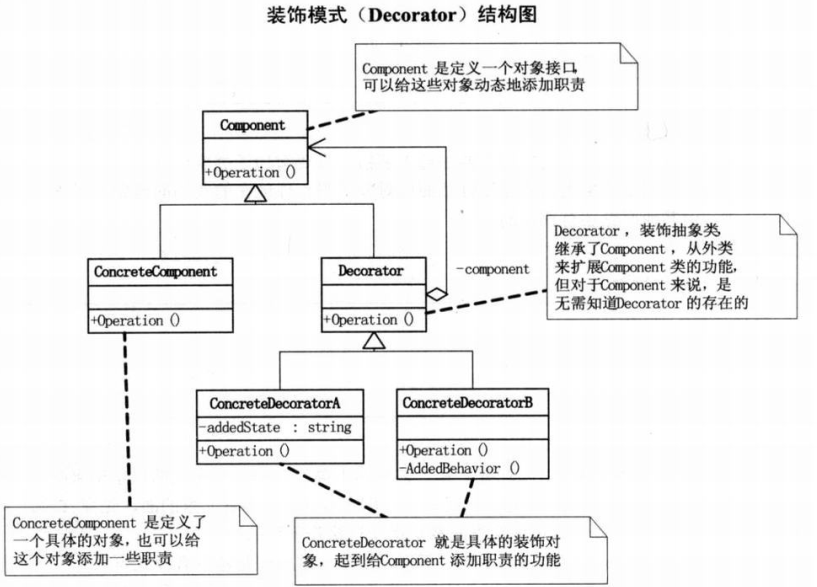
在Spring的AOP编程中:

如果加入容器的目标对象有实现接口,用JDK代理

如果目标对象没有实现接口,用Cglib代理

**第17章 装饰模式**

动态的给一个对象添加一些额外的职责。就添加功能来说，装饰模式比生成子类更灵活。



装饰着模式：为已有功能动态的添加更多功能的一种方式。将类的核心职责和装饰功能隔离，可以很方便对装饰功能进行添加和去除。

* **装饰模式和代理模式的比较：**

**相同点：**

装饰模式：装饰者（decorator）和被装饰者（decoratee）都实现同一个接口。

代理模式：代理类（proxy class）和真实处理的类（real class）都实现同一个接口。

此外，不论我们使用哪一个模式，都可以很容易地在真实对象的方法前面或者后面加上自定义的方法。

**不同点：**

装饰器模式关注于在一个对象上动态的添加方法，然而代理模式关注于控制对对象的访问。

装饰模式以对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案；它是在运行的时候才知道目标对象（也就是被装饰的对象）

代理模式给一个对象提供一个代理对象，并有代理对象来控制对原有对象的引用；它在编译的时候就清楚知道所需要代理的对象

* **装饰模式的代码实现：**
* **//抽象组件**

**abstract** **class** People{

**public** **abstract** **void** showCloth();

}

* **具体组件（被装饰类）**

**class** PeopleImpl **extends** People{

**private** String name;

**public** PeopleImpl(String name) {

**this**.name=name;

}

**public** **void** showCloth() {

System.***out***.println("she dressing in a pink shirt!");

}}

* **//抽象装饰类**

**abstract** **class** Decorator **extends** People{

**private** People people;

**public** Decorator(People people) {

**this**.people=people;

}

**public** **void** showCloth() {

people.showCloth();

}}

* **//具体装饰类1**

**class** HatDecorator **extends** Decorator{

**public** HatDecorator(People people) {

**super**(people);

}

**private** **void** showHat() {

System.***out***.println("she have a beautiful hat");

}

**public** **void** showCloth() {**//重写父类的方法**

**this**.showHat();

**super**.showCloth();

}}

* **//具体装饰类2**

**class** ShoesDecorator **extends** Decorator{

**public** ShoesDecorator(People people) {

**super**(people);

}

**private** **void** showShoes() {

System.***out***.println("she dress a beautiful shoes");

}

**public** **void** showCloth() {**//重写父类的方法**

**super**.showCloth();

**this**.showShoes();

}}

**public** **class** CommandTest{

**public** **static** **void** main(String[] args) {

People taotao=**new** PeopleImpl("peach");

People t1=**new** HatDecorator(taotao);

People t2=**new** ShoesDecorator(t1);

t2.showCloth();

}}

原始方法和装饰方式的执行顺序在具体的装饰类是固定的，可以通过方法重载实现多种顺序

首先用HatDecorator类装饰PeopleImpl，然后用ShoesDecorator类装饰t1,这样层层装饰

**第15章 命令模式**

任何模式的出现，都是为了解决一些特定的场景的耦合问题，以达到对修改封闭，对扩展开放的效果。命令模式也不例外：

命令模式是为了解决命令的请求者（Inover）和命令的执行者（Receiver）之间的耦合关系。他的封装性好，通用代码简单。

定义：将一个请求封装成一个对象，从而让你使用不同的请求把客户端参数化，对请求排队或者记录请求日志，可以提供命令的撤销和恢复功能。

* **Reciver类是抽象类时；**

//接受者

**abstract** **class** Reciver{

**public** **abstract** **void** barbecue();

}

**class** MuttonReciver **extends** Reciver{

**public** **void** barbecue() {

System.***out***.println("考羊肉");

}}

**class** ChichenReciver **extends** Reciver{

**public** **void** barbecue() {

System.***out***.println("烤鸡肉");

}}

//命令对象

**abstract** **class** Command{

**public** **abstract** **void** excute();

}

**class** CommandImpl **extends** Command{

**private** Reciver reciver;

**public** CommandImpl(Reciver reciver) {

**this**.reciver=reciver;

}

**public** **void** excute() {

reciver.barbecue();

}}

//请求者

**class** Ivoker{

**private** Command command;

**public** **void** setCommand(Command command) {

**this**.command=command;

}

**public** **void** action() {

**this**.command.excute();

}}

**public** **class** CommandTest {

**public** **static** **void** main(String[] agrs) {

Reciver reciver=**new** MuttonReciver();

Command command=**new** CommandImpl(reciver);

Ivoker invoker=**new** Ivoker();

invoker.setCommand(command);

invoker.action();

}}

我的理解是:命令模式中，当reciver类是抽象的时，command只需要一个实现类，这时可以使用多态来定位每个reciver的实现类。当reciver是具体类时，command才需要创建多个实现类，每一个实现类执行reciver中的特定方法

* **Reciver类是具体类；**

//接受者

**class** Reciver{

**public** **void** barbecueMotton() {

System.***out***.println("考羊肉");

}

**public** **void** barbecueChichen() {

System.***out***.println("烤鸡肉");

}}

//命令对象

**abstract** **class** Command{

**public** **abstract** **void** excute();

}

**class** CommandImpl1 **extends** Command{

**private** Reciver reciver;

**public** CommandImpl(Reciver reciver) {

**this**.reciver=reciver;

}

**public** **void** excute() {

reciver.barbecueMutton();

}}

**class** CommandImpl2 **extends** Command{

**private** Reciver reciver;

**public** CommandImpl(Reciver reciver) {

**this**.reciver=reciver;

}

**public** **void** excute() {

reciver.barbecueChichen();

}}

//请求者

**class** Ivoker{

**private** Command command;

**public** **void** setCommand(Command command) {

**this**.command=command;

}

**public** **void** action() {

**this**.command.excute();

}}

**public** **class** CommandTest {

**public** **static** **void** main(String[] agrs) {

Reciver reciver=**new** Reciver();

Command command=**new** CommandImpl1(reciver);

Ivoker invoker=**new** Ivoker();

invoker.setCommand(command);

invoker.action();

}}

**第16章 责任链模式**

使多个对象都有机会处理请求，从而避免了请求的发送者和接受者之间的耦合关系。**将这些对象连成一条链，并沿着这条链传递该请求，直到有对象处理他为止。——由一条链去处理相似的请求，在链中决定谁来处理这个请求，并返回相应的结果。**

**abstract** **class** AbstractWoman{

**protected** **int** type;//1表示未出嫁，2出嫁有夫。3夫死有子

**protected** String request;

**public** AbstractWoman(**int** type,String request) {

**this**.type=type;

**this**.request=request;}

**public** **abstract** **int** getType();

**public** **abstract** **void** getRequest();

}

**class** Wife **extends** AbstractWoman{

**public** Wife(String request) {

**super**(1, request);}

**public** **int** getType() {

**return** **this**.type;}

**public** **void** getRequest() {

System.***out***.println( "妻子的请求是"+**this**.request);

}}

**class** Mother **extends** AbstractWoman{

**public** Mother(String request) {

**super**(2, request); }

**public** **int** getType() {

**return** **this**.type;}

**public** **void** getRequest() {

System.***out***.println( "母亲的请求是"+**this**.request);

}}

**abstract** **class** Handler{

**private** **int** level;

**private** Handler nextHandler;

**public** Handler(**int** level) {

**this**.level=level;

}

**public** **void** handleRequset(AbstractWoman woman) {

**if**(woman.getType()==**this**.level) {

**this**.response(woman);

}**else** {

**if**(**this**.nextHandler!=**null**)

**this**.nextHandler.handleRequset(woman);

**else** {

System.***out***.print("妇女的请求是");

woman.getRequest();

System.***out***.println("不同意");

}}}

**public** **void** setNextHandler(Handler nextHandler){

**this**.nextHandler=nextHandler;

}

**public** **abstract** **void** response(AbstractWoman woman);

}

**class** Husband **extends** Handler{

**public** Husband() {

**super**(1);

}

**public** **void** response(AbstractWoman woman) {

woman.getRequest();

System.***out***.println("丈夫同意请求");

}

}

**class** Son **extends** Handler{

**public** Son() {

**super**(2);

}

**public** **void** response(AbstractWoman woman) {

woman.getRequest();

System.***out***.println("儿子同意请求");

}

}

**public** **class** CommandTest{

**public** **static** **void** main(String[]args) {

AbstractWoman woman=**new** Mother("化妆") ;

Handler husband=**new** Husband();

Handler son=**new** Son();

husband.setNextHandler(son);

husband.handleRequset(woman);

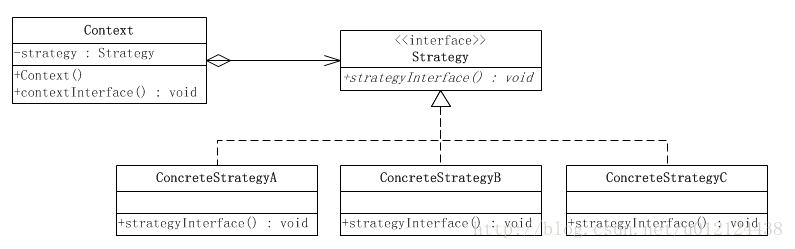
}

}

该模式屏蔽了责任请求者和责任处理者的耦合，且满足开闭原则。无论是添加责任请求者还是责任处理者，我们都不必修改已有的代码，只需在原有的代码上进行扩展。

缺点：每个请求都是从连头遍历到连尾，特别是链比较长时，性能是一个非常大的问题。由于采用递归，调试不方便

**第18章 策略模式**

定义一组算法，将每个算发都封装起来，使他们之间可以互换（实现相同的接口）。使用继承和多态的机制

这个模式涉及到三个角色：

环境(Context)角色：持有一个Strategy的引用。

抽象策略(Strategy)角色：这是一个抽象角色，通常由一个接口或抽象类实现。此角色给出所有的具体策略类所需的接口。

具体策略(ConcreteStrategy)角色：包装了相关的算法或行为。（但一个策略的具体策略超过4个时，则应使用混合模式）

**//策略角色**

**abstract** **class** Stratery{

**public** **abstract** **void** OffLevel();

}

**class** GoldStratery **extends** Stratery{

**public** **void** OffLevel() {

System.***out***.println("黄金会员开通plus会员需149元");

}}

**class** DiamondStratery **extends** Stratery{

**public** **void** OffLevel() {

System.***out***.println("钻石会员开通plus会员需129元");

}}

**class** PlatinumStratery **extends** Stratery{

**public** **void** OffLevel() {

System.***out***.println("珀金会员开通plus会员需119元");

}}

//封装角色

**class** Context{

**private** Stratery stratery;

**public** Context(Stratery stratery) {

**this**.stratery=stratery;

}

**public** **void** Operate() {

stratery.OffLevel();

}}

//高层模块

**public** **class** CommandTest{

**public** **static** **void** main(String[] args){

Stratery stratery=**new** DiamondStratery();

Context context=**new** Context(stratery);

context.Operate();

}

}

写代码时总会出很多的if…else，或者case。如果在一个条件语句中又包含了多个条件语句就会使得代码变得臃肿，维护的成本也会加大，而策略模式就能较好的解决这个问题

优点：扩展性好、避免多重条件判断、扩展性好

缺点：每一个策略都是一个类，可复用性小，类的数量多。违背迪米特法则。

**19章 适配器模式**

将一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口不匹配而无法在一起工作的两个类能够在一起工作

适配器就是一种适配中间件，它存在于不匹配的二者之间，用于连接二者，将不匹配变得匹配，简单点理解就是平常所见的转接头，转换器之类的存在。

　　适配器模式有两种：类适配器、对象适配器、接口适配器

前二者在实现上有些许区别，作用一样，第三个接口适配器差别较大。

1. **类适配器**——通过继承来实现适配器功能。

当我们要访问的接口A中没有我们想要的方法，却在另一个接口B中发现了合适的方法，在不能改变访问的接口A的情况下，可以定义一个适配器P来进行中转，这个适配器p要实现接口A，并继承接口B的实现类BB。这样我们可以在适配器P中访问接口A和B的方法了，这时我们在适配器P中的接口A方法中直接引用BB中的合适方法，这样就完成了一个简单的类适配器。

例子：我需要给MP4充电，他的充电接口是PS2接口，但是电脑上的主机结构是USB，如何通过USB得到ps2接口？

**类适配器**

**interface** USB {

**void** showPort();

}

**interface** PS2{

**void** thisIs();

}

**class** PS2Impl **implements** PS2{

**public** **void** thisIs() {

System.***out***.println("我是PS2接口");

}

}

**class** Adapter **extends** PS2Impl **implements** USB{

**public** **void** showPort() {

**super**.thisIs();

}

}

**public** **class** CommandTest{

**public** **static** **void** main(String[] args) {

USB usb=**new** Adapter();

usb.showPort();

}

}

2**.对象适配器**——通过组合来实现适配器功能

为了用接口A来访问接口B中的方法，我们需要适配器p来中转。他实现我们访问的接口A，然后拥有B类型的私有变量C，再定义一个带参数的构造器用来为对象C赋值，再在A接口的方法实现中使用对象C调用其来源于B接口的方法。

**对象适配器**

**interface** USB {

**void** showPort();

}

**interface** PS2{

**void** thisIs();

}

**class** PS2Impl **implements** PS2{

**public** **void** thisIs() {

System.***out***.println("我是PS2接口");

}

}

**class** Adapter **implements** USB{

**private** PS2 ps2;

**public** Adapter(PS2 ps2) {

**this**.ps2=ps2;

}

**public** **void** showPort() {

ps2.thisIs();

}

}

**public** **class** CommandTest{

**public** **static** **void** main(String[]args) {

PS2 ps2=**new** PS2Impl();

USB usb=**new** Adapter(ps2);

usb.showPort();

}

}

3**接口适配器—**—通过抽象类来实现适配

**应用场景：**不想实现接口中的所有方法

当不需要**全部实现**接口提供的方法时，可先设计一个**抽象类**实现接口——即为适配器，并为该接口中每个方法提供一个**默认实现**（空方法），那么该抽象类的子类可**有选择**地覆盖父类的某些方法来实现需求，它适用于**一个接口不想使用其所有的方法**的情况。

**interface** A{

**void** showA();

**void** showB();

**void** showC();

}

**abstract** **class** AbstractA **implements** A{

**public** **void** showA() {}

**public** **void** showB() {}

**public** **void** showC() {}

}

**class** AImple **extends** AbstractA{

**public** **void** showA() {

System.***out***.println("我今天去游泳");

}

}

**public** **class** CommandTest{

**public** **static** **void** main(String[] args) {

A a=**new** AImple();

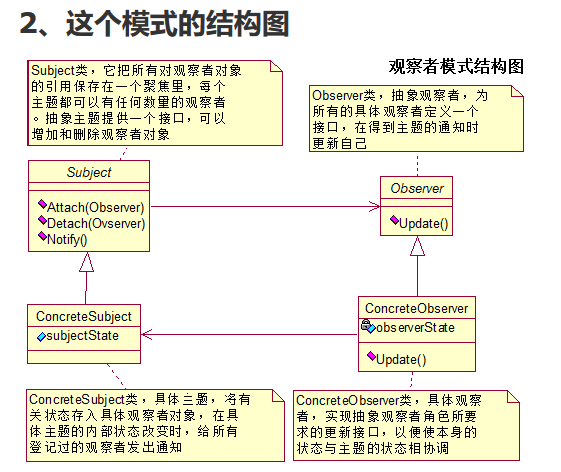
a.showA();

}

}

**22章 观察者模式**—发布订阅模式

定义一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于他的对象都会得到通知并自动更新。



使用场景：有一个微信公众号服务，不定时发布一些消息，关注公众号就可以收到推送消息，取消关注就收不到推送消息。

**interface** Observer{

**void** update(String str);

}

**class** UserA **implements** Observer{

**public** **void** update(String str) {

System.***out***.println("A接收到消息："+str);

}}

**class** UserB **implements** Observer{

**public** **void** update(String str) {

System.***out***.println("B接受到消息："+str);

}}

**interface** IObservable{

**void** registerUser(Observer user);

**void** deletUser(Observer user);

**void** notifyUser(String str);

}

**interface** IWXServer{

**void** doSomething();

}

**class** WXServer **implements** IObservable,IWXServer{

**private** List<Observer>userList;

**public** WXServer() {

userList=**new** ArrayList<>();

}

**public** **void** registerUser(Observer user) {

userList.add(user);

}

**public** **void** deletUser(Observer user) {

userList.remove(user);

}

**public** **void** notifyUser(String str) {

**for**(Observer user:userList)

user.update(str);

}

**public** **void** doSomething() {

System.***out***.println("微信服务号更新了：");

**this**.notifyUser("穿衣搭配");

}}

**public** **class** CommandTest{

**public** **static** **void** main(String[]args) {

Observer userA=**new** UserA();

Observer userB=**new** UserB();

WXServer server=**new** WXServer();

server.registerUser(userB);

server.registerUser(userA);

server.doSomething();

}}

Java的JDK中提供了java.Util.Obserable类(被观察者)和java.util.Observer接口（观察者）

**//使用JDK提供的Observer接口和Obserable类的效果**

**class** UserA **implements** Observer{

**public** **void** update(**Observable wxServer**, Object str) {

System.***out***.println("A接收到消息："+str);

}}

**class** UserB **implements** Observer{

**public** **void** update(Observable wxServer, Object str) {

System.***out***.println("B接受到消息："+str);

}}

**interface** IWXServer{

**void** doSomething();}

**class** WXServer **extends** Observable **implements** IWXServer{

**public** **void** doSomething() {

System.***out***.println("微信服务号更新了：");

**super**.setChanged();

**super**.notifyObservers("穿衣搭配");

}}

**//测试类不变**