# 基本功

## 1 面向对象的特征

## 2 final, finally, finalize 的区别

## 3 int 和 Integer 有什么区别

## 4 重载和重写的区别

## 5 抽象类和接口有什么区别

## 6 说说反射的用途及实现

## 7 说说自定义注解的场景及实现

## 8 HTTP 请求的 GET 与 POST 方式的区别

## 9 session 与 cookie 区别

<https://blog.csdn.net/duan1078774504/article/details/51912868>

## 10 session 分布式处理

目前网上能找到的方案有：  
1.基于数据库的Session共享  
2.基于NFS共享文件系统  
3.基于memcached 的session，如何保证 memcached 本身的高可用性？  
4. 基于resin/tomcat web容器本身的session复制机制  
5. 基于TT/Redis 或 jbosscache 进行 session 共享。  
6. 基于cookie 进行session共享

<https://blog.csdn.net/u014352080/article/details/51764311>

<https://blog.csdn.net/collonn/article/details/52701963>

<https://www.cnblogs.com/study-everyday/p/7853145.html>

<https://www.cnblogs.com/newP/p/6518918.html>

<https://blog.csdn.net/u010028869/article/details/50773174>

<https://blog.52itstyle.com/archives/759/>

## 11 JDBC 流程

12 MVC 设计思想

<https://blog.csdn.net/u013215018/article/details/52814805>

## 13 equals 与 == 的区别

<https://www.cnblogs.com/lingyujuan/p/6491883.html>

<https://www.jianshu.com/p/e93cf60246de>

<https://blog.csdn.net/haobaworenle/article/details/53819838>

<http://www.cnblogs.com/skywang12345/p/3324958.html>

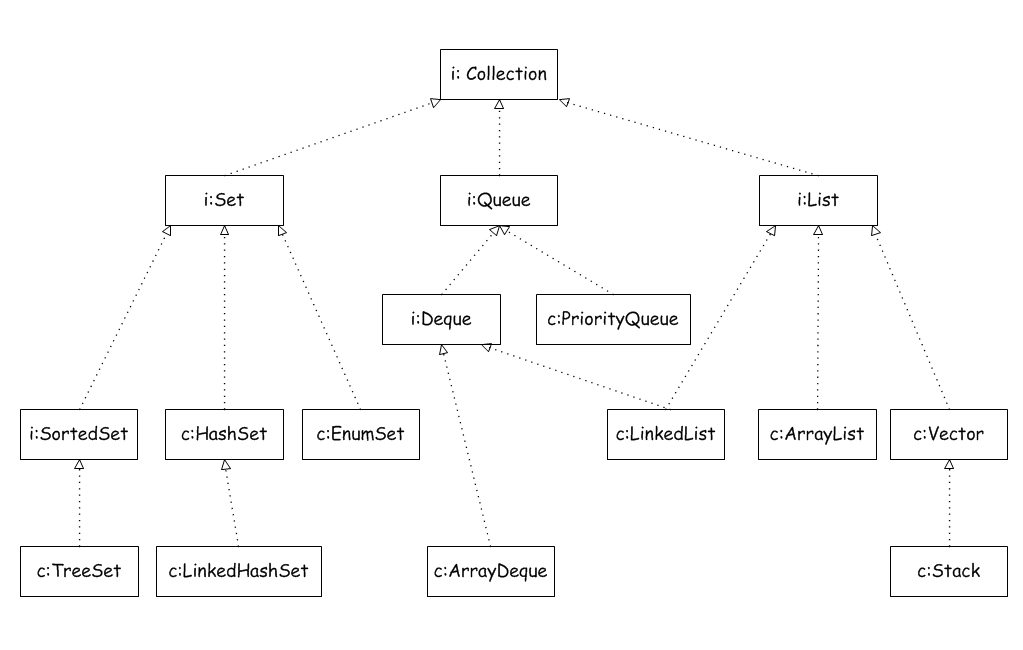
<https://www.cnblogs.com/jesonjason/p/5492208.html>

## 1、hashcode相等两个类一定相等吗?equals呢?相反呢?

答：**两个对象equals相等那么hashcode 是一定相等的**；**如果两个对象的hashCode相同，它们equals并不一定相同。所以二者必须同时重写。**

因为例如HasmMap中通过获取key的哈希值，然后通过hash方法计算出一个值，这个值作为比对的依据。也就是，**如果两个对象相等，那么他们的hashCode就一定要相等，不然这里这两个对象就会得到不同hash值从而存在不同的地方。**HashCode说白了是地址值经过一系列的复杂运算得到的结果，而Object中的equals方法底层比较的就是地址值，所以equals()相等，hashCode必定相等，反equals()不等，在java底层进行哈希运算的时候有一定的几率出现相等的hashCode,所以hashCode（）可等可不等。

## 2、介绍一下集合框架?



**1. Set子接口**：**不允许包含相同**的元素，而判断两个对象是否相同则是根据equals方法。

**1.1 HashSet类**

不能保证元素的排列顺序，加入的元素要特别注意hashCode()方法的实现。

HashSet不是同步的，多线程访问同一步HashSet对象时，需要手工同步。

集合元素值可以是null。

**1.2 LinkedHashSet类**

LinkedHashSet类也是根据元素的hashCode值来决定元素的存储位置，但它同时使用链表维护元素的次序。与HashSet相比，特点：

对集合迭代时，按增加顺序返回元素。性能略低于HashSet，因为需要维护元素的插入顺序。但迭代访问元素时会有好性能，因为它采用链表维护内部顺序。

**1.3 SortedSet接口及TreeSet实现类**

TreeSet类是SortedSet接口的实现类。**因为需要排序**，所以性能肯定差于HashSet。

可以定义比较器（Comparator）来实现自定义的排序。**默认自然升序排序。**

**2. List子接口：**：允许**重复**的元素，**有序**集合，所以与Set相比，增加了与索引位置相关的操作。

**2.1 ArrayList**

是数组结构，长度是可变的，原理是（创建新数组+复制数组），查询速度快，增删较慢，不同步。ArrayList不是线程安全的，只能用在单线程环境下，多线程环境下可以考虑使用Collections.synchronizedList（List i）函数返回一个线程安全的ArrayList类，或者使用Concurrent并发包下对应的集合类。

**2.2 Vector实现类**

基于数组实现的List类。Vector是线程安全的。但Vector的性能会比ArrayList低。

Vector提供一个子类Stack，可以挺方便的模拟“栈”这种数据结构（LIFO，后进先出）。

结论：不推荐使用Vector类，即使需要考虑同步，即也可以通过其它方法实现。

**2.3 LinkedList类**

是基于双向循环链表实现的，是链表结构，不同步的，增删速度快，查询较慢。由于实现了Queue接口，因此也可以用于实现堆栈、队列。

**3 Map**

**3.1 HashMap**

数据结构为哈希表，不同步，允许null作为键和值，无序的

**3.1.1 LinkedHashMap :**

类似于HashMap，但是迭代遍历它时，取得“键值对”的顺序是其插入次序，或者是最近最少使用(LRU)的次序。只比HashMap慢一点。而在迭代访问时发而更快，因为它使用链表维护内部次序。

**3.2 TreeMap**

基于红黑树数据结构的实现。查看“键”或“键值对”时，它们会被排序(次序由Comparabel或Comparator决定)。TreeMap的特点在于，你得到的结果是经过排序的。

**3.3 Hashtable**

数据结构为哈希表，同步的，不允许null作为键和值。

**3.3.1 Properties**：属性集，键和值都是字符串，可结合流进行键值的操作

## 3、hashmap hastable 底层实现什么区别?hashtable和concurrenthashtable呢?

Hashtable 也是一个散列表，它存储的内容是键值对(key-value)映射。Hashtable 的方法都是同步的(每个方法前面都加上了synchronized关键字)，这意味着它是线程安全的。**它的key、value都不可以为null**。此外，Hashtable中的映射不是有序的。

在JDK1.6，JDK1.7中，HashMap采用位桶+链表实现，即使用链表处理冲突，同一hash值的链表都存储在一个链表里。但是当位于一个桶中的元素较多，即hash值相等的元素较多时，通过key值依次查找的效率较低。而JDK1.8中，HashMap采用位桶+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。

## 4、hashmap和treemap什么区别?低层数据结构是什么?

Treemap：TreeMap的实现是红黑树算法的实现。因为TreeMap是有序的，通过comparator接口我们可以对TreeMap的内部排序进行精密的控制。

## 10、请用至少四种写法写一个单例模式?

**JVM**

# 1、请介绍一下JVM内存模型??用过什么垃圾回收器都说说呗

2、线上发送频繁full gc如何处理? CPU 使用率过高怎么办?

如何定位问题?如何解决说一下解决思路和处理方法

3、知道字节码吗?字节码都有哪些?Integer x =5,int y =5，比较x =y 都经过哪些步骤?

4、讲讲类加载机制呗都有哪些类加载器，这些类加载器都加载哪些文件?

手写一下类加载Demo

5、知道osgi吗? 他是如何实现的???

6、请问你做过哪些JVM优化?使用什么方法达到什么效果???

7、classforName("java.lang.String")和String classgetClassLoader() LoadClass("java.lang.String") 什么区别啊??



JVM

**数据库**

1、使用mysq1索引都有哪些原则? ?索引什么数据结构? 3+tree 和B tree 什么区别?

2、mysq1有哪些存储引擎啊?都有啥区别? 要详细!

3、设计高并发系统数据库层面该怎么设计??数据库锁有哪些类型?如何实现呀?

4、数据库事务有哪些?

**分库分表**

1、如何设计可以动态扩容缩容的分库分表方案?

2、用过哪些分库分表中间件，有啥优点和缺点?讲一下你了解的分库分表中间件的底层实现原理?

3、我现在有一个未分库分表的系统，以后系统需分库分表，如何设计，让未分库分表的系统动态切换到分库分表的系统上???TCC? 那若出现网络原因，网络连不通怎么办啊???

4、分布式事务知道吗? 你们怎么解决的?

5、为什么要分库分表啊???

6、分布式寻址方式都有哪些算法知道一致性hash吗?手写一下java实现代码??你若userId取摸分片，那我要查一段连续时间里的数据怎么办???

7、如何解决分库分表主键问题有什么实现方案??

**分布式缓存**

1、redis和memcheched 什么区别为什么单线程的redis比多线程的memched效率要高啊?

2、redis有什么数据类型都在哪些场景下使用啊?

3、reids的主从复制是怎么实现的redis的集群模式是如何实现的呢redis的key是如何寻址的啊?

4、使用redis如何设计分布式锁?使用zk可以吗?如何实现啊这两种哪个效率更高啊??

5、知道redis的持久化吗都有什么缺点优点啊? ?具体底层实现呢?

6、redis过期策略都有哪些LRU 写一下java版本的代码吧??

**分布式服务框架**

1、说一下dubbo的实现过程注册中心挂了可以继续通信吗??

2、zk原理知道吗zk都可以干什么Paxos算法知道吗?说一下原理和实现??

3、dubbo支持哪些序列化协议?hessian 说一下hessian的数据结构PB知道吗为啥PB效率是最高的啊??

4、知道netty吗'netty可以干嘛呀NIO,BIO,AIO 都是什么啊有什么区别啊?

5、dubbo复制均衡策略和高可用策略都有哪些啊动态代理策略呢?

6、为什么要进行系统拆分啊拆分不用dubbo可以吗'dubbo和thrift什么区别啊?

**分布式消息队列**

1、为什么使用消息队列啊消息队列有什么优点和缺点啊?

2、如何保证消息队列的高可用啊如何保证消息不被重复消费啊

3、kafka ，activemq,rabbitmq ，rocketmq都有什么优点，缺点啊???

4、如果让你写一个消息队列，该如何进行架构设计啊?说一下你的思路

**高并发高可用架构设计**

1、如何设计一个高并发高可用系统

2、如何限流?工程中怎么做的，说一下具体实现

3、缓存如何使用的缓存使用不当会造成什么后果?

4、如何熔断啊?熔断框架都有哪些?具体实现原理知道吗?

5、如何降级如何进行系统拆分，如何数据库拆分????



分布式专题架构

**通信协议**

1、说一下TCP 'IP四层?

2、http的工作流程?? ?http1.0 http1.1http2.0 具体哪些区别啊?

3、TCP三次握手，四层分手的工作流程画一下流程图为什么不是四次五次或者二次啊?

4、画一下https的工作流程?具体如何实现啊?如何防止被抓包啊??

**算法**

1、比较简单，我一个文件，有45亿个阿拉伯数字，如何进行去重啊如何找出最大的那个数啊?

**数据结构**

1、二叉树和红黑树等。

**源码中所用到的经典设计思想及常用设计模式**



**福利**

这里给大家提供一个学习交流的平台，Java架构师群：726610841

1. 具有1-5工作经验的，面对目前流行的技术不知从何下手，需要突破技术瓶颈的可以加群。
2. 在公司待久了，过得很安逸，但跳槽时面试碰壁。需要在短时间内进修、跳槽拿高薪的可以加群。
3. 如果没有工作经验，但基础非常扎实，对java工作机制，常用设计思想，常用java开发框架掌握熟练的可以加群。

# Mybatis

## 1 代码自动生成器：generatorConfiguration

① .xml：包含数据库驱动器；数据库连接；生成实体类名，指定包名以及生成的地址；生成Dao文件；对应数据库表；

② 新建文件夹

③java -jar C:\Users\Laura\Desktop\mybatis-generator\mybatis-generator\mybatis-generator-core-1.3.1.jar -configfile C:\Users\Laura\Desktop\mybatis-generator\generatorBp.xml –overwrite

## 2 MyBatis关联查询、嵌套查询

Association

Collection

## 3 缓存使用场景及选择策略

Mybatis的一级缓存是指SqlSession。一级缓存的作用域是一个SqlSession。Mybatis默认开启一级缓存。在同一个SqlSession中，执行相同的查询SQL，第一次会去查询数据库，并写到缓存中；第二次直接从缓存中取。当执行SQL时两次查询中间发生了增删改操作，则SqlSession的缓存清空。

Mybatis的二级缓存是指mapper映射文件。二级缓存的作用域是同一个namespace下的mapper映射文件内容，多个SqlSession共享。Mybatis需要手动设置启动二级缓存。在同一个namespace下的mapper文件中，执行相同的查询SQL，第一次会去查询数据库，并写到缓存中；第二次直接从缓存中取。当执行SQL时两次查询中间发生了增删改操作，则二级缓存清空。

开启：mybatis文件：<setting name="cacheEnabled" value="true" />

在映射文件中，开启二级缓存：<cache/>

**多表操作一定不能使用缓存.**

## 4 Mybatis事务

## 5 MyBatis的动态代理

当定义好一个Mapper接口(UserDao)里，我们并不需要去实现这个类，但sqlSession.getMapper()最终会返回一个实现该接口的对象。这个对象是Mybatis利用jdk的动态代理实现的。这里将介绍这个代理对象的生成过程及其方法的实现过程。

①Mapper代码对象的生成过程：获取每个Mapper接口对应一个MapperProxyFactory对象实例，然后调用MapperRegistry.getMapper()方法，生成一个MapperProxy对象。

②MapperProxyFactory的newInstance方法：创建一个MapperProxy对象，这个方法实现了JDK动态代理中的InvocationHandler接口，说明Mapper接口被代理了，这样子返回的对象就是Mapper接口的子类，方法被调用时会被mapperProxy拦截,也就是执行mapperProxy.invoke()方法 。invoke方法会拦截Mapper接口(UserDao)的所有方法， MapperProxy会根据方法找到对应的MapperMethod对象来实现这次调用。

③MapperMethod对应会读取方法中的注解，从Configuration中找到相对应的MappedStatement对象，再执行。

# 分布式

## 1 谈谈业务中使用分布式的场景

(1)**首先，需要了解系统为什么使用分布式。**

随着互联网的发展，传统单工程项目的很多性能瓶颈越发凸显，性能瓶颈可以有几个方面。**一、应用服务层**：随着用户量的增加，并发量增加，单项目难以承受如此大的并发请求导致的性能瓶颈。**二、底层数据库层**：随着业务的发展，数据库压力越来越大，导致的性能瓶颈。

(2)**针对上面两点，我觉得可以从两方面解决**。

应用系统集群最简单的就是服务器集群。应用系统集群的时候，比较凸显的问题是**session共享**，session共享我们一是可以通过服务器插件来解决。另外一种也可以通过redis等中间件实现。

服务化拆分，是目前非常火热的一种方式。现在都在提微服务话。通过对传统项目进行服务化拆分，达到服务独立解耦，单服务又可以横向扩容。服务化拆分遇到的经典问题就是**分布式事务问题**。目前，比较常用的分布式事务解决方案有几种：消息最终一致性、TCC补偿型事务、尽最大能里通知。

底层数据库层：如果系统的性能压力出现在数据库，那我们就可以读写分离、分库分表等方案进行解决。

## 2 Session 分布式方案

目前网上能找到的方案有：

**1. 基于数据库的Session共享：**

原理：就不用多说了吧，拿出一个数据库，专门用来存储session信息。保证session的持久化。

优点：服务器出现问题，session不会丢失

缺点：如果网站的访问量很大，把session存储到数据库中，会对数据库造成很大压力，还需要增加额外的开销维护数据库。

**2.客户端存储**：

思路：将session存储到浏览器cookie中，每个端只要存储一个用户的数据了；

缺点：每次http请求都携带session，占外网带宽；数据存储在端上，并在网络传输，存在泄漏、篡改、窃取等安全隐患；session存储的数据大小受cookie限制。

**3. 粘性session**

原理：**粘性Session是指将用户锁定到某一个服务器上**。用户第一次请求时，负载均衡器将用户的请求转发到了A服务器上，如果负载均衡器设置了粘性Session的话，那么用户以后的每次请求都会转发到A服务器上。

优点：简单，不需要对session做任何处理。

缺点：缺乏容错性，如果当前访问的服务器发生故障，用户被转移到第二个服务器上时， session将失效。

适用场景：发生故障对客户产生的影响较小；服务器发生故障是低概率事件。

实现方式：以Nginx为例，在upstream模块配置ip\_hash属性即可实现粘性Session。

**4.** **服务器session复制**

原理：任何一个服务器上的session发生改变，该节点会把这个 session的所有内容序列化，然后广播给所有其它节点，不管其他服务器需不需要session，以此来保证Session同步。

优点：可容错，各个服务器间session能够实时响应。

缺点：会对网络负荷造成一定压力，如果session量大的话可能会造成网络堵塞，拖慢服务器性能。

实现方式：① 设置tomcat ，server.xml 开启tomcat集群功能；② 在应用里增加信息：通知应用当前处于集群环境中，支持分布式，在web.xml中添加选项 <distributable/>。

**5. session共享机制**

使用分布式缓存方案比如memcached、redis，但是要求Memcached或Redis必须是集群。

**① 粘性session处理方式**

原理：不同的 tomcat指定访问不同的主redis。多个redis之间信息是同步的，能主从备份和高可用。用户访问时首先在tomcat中创建session，然后将session复制一份放到它对应的redis上。memcache只起备份作用，读写都在tomcat上。当某一个tomcat挂掉后，集群将用户的访问定位到备tomcat上，然后根据cookie中存储的SessionId找session，找不到时，再去相应的redis上去session，找到之后将其复制到备tomcat上。

**② 非粘性session处理方式**

原理：redis做主从复制，写入session都往从redis服务上写，读取都从主redis读取，tomcat本身不存储session。

优点：可容错，session实时响应。

**Spring Session + Redis实现分布式Session共享：Redis Server版本不低于2.8**

[**https://blog.csdn.net/zouxucong/article/details/53286748**](https://blog.csdn.net/zouxucong/article/details/53286748)

添加依赖：spring-session-data-redis/jedis

Spring配置：RedisHttpSessionConfiguration/JedisConnectionFactory

配置web.xml过滤器：配置过滤器DelegatingFilterProxy

## 3 分布式锁的场景

比较敏感的数据比如金额修改，同一时间只能有一个人操作，想象下2个人同时修改金额，一个加金额一个减金额，为了防止同时操作造成数据不一致，需要锁。如果是数据库需要的就是行锁或表锁，如果是在集群里，多个客户端同时修改一个共享的数据就需要分布式锁。场景：例如秒杀。

## 4 分布式锁的实现方案

谈到分布式锁，有很多实现方式，如数据库、redis、ZooKeeper等。

**1. 数据库实现分布式锁-行锁**

select \* from lock where lock\_name=xxx for update;

数据库的lock表，lock\_name是主键,通过for update操作，数据库就会对该行记录加上record lock，从而阻塞其他人对该记录的操作。一旦获取到了锁，就可以开始执行业务逻辑，最后通过connection.commit()操作来释放锁。

问题：首先性能不是特别高。通过数据库的锁来实现多进程之间的互斥，但是这貌似也有一个问题：就是sql超时异常的问题。jdbc超时具体有3种超时：框架层的事务超时（不涉及）；jdbc的查询超时（mysql的jdbc驱动会向服务器发送kill query命令来取消查询）；Socket的读超时（如果一旦出现Socket的读超时，对于如果是同步通信的Socket连接来说，该连接基本上不能使用了，需要关闭该连接，从新换用新的连接，因为会出现请求和响应错乱的情况，比如jedis出现的类型转换异常）。

**2. redis实现分布式锁**

redis通常可以使用setnx来实现分布式锁。setnx来创建一个key，如果key不存在则创建成功返回1，如果key已经存在则返回0。依照上述来判定是否获取到了锁，获取到锁的执行业务逻辑，完毕后删除lock\_key，来实现释放锁。

改进：一旦获取到锁的客户端挂了，没有执行上述释放锁的操作，则其他客户端就无法获取到锁了，所以在这种情况下有2种方式来解决：①为lock\_key设置一个过期时间；②对lock\_key的value进行判断是否过期：一旦发现lock\_key的值已经小于当前时间了，说明该key过期了，然后对该key进行getset设置，一旦getset返回值是原来的过期值，说明当前客户端是第一个来操作的，代表获取到了锁，一旦getset返回值不是原来过期时间则说明前面已经有人修改了，则代表没有获取到锁。

问题：①lock timeout的存在也使得失去了锁的意义，即存在并发的现象。一旦出现锁的租约时间，就意味着获取到锁的客户端必须在租约之内执行完毕业务逻辑，一旦业务逻辑执行时间过长，租约到期，就会引发并发问题。所以有lock timeout的可靠性并不是那么的高。②上述方式仅仅是redis单机情况下，还存在redis单点故障的问题。如果为了解决单点故障而使用redis的sentinel或者cluster方案，则问题更多。

**3. ZooKeeper实现分布式锁**

<https://www.jianshu.com/p/2d22df6eccf8>

首先zookeeper创建个PERSISTENT持久节点，然后每个要获得锁的线程都会在这个节点下创建个临时顺序节点，然后规定节点最小的那个获得锁，所以每个线程首先都会判断自己是不是节点序号最小的那个，如果是则获取锁，如果不是则监听比自己小的上一个节点，如果上一个节点不存在了，然后会再一次判断自己是不是序号最小的那个节点，是则获得锁，不是重复上述动作。

1 获取锁

public void lock(){

path = 在父节点下创建临时顺序节点

while(true){

children = 获取父节点的所有节点

if(path是children中的最小的){

//代表获取了节点

return;

}else{

//添加监控前一个节点是否存在的watcher

wait();

}

}

}

watcher中的内容{

notifyAll();

}

2 释放锁

public void release(){

删除上述创建的节点

}

## 5 分布式事务

① 在说分布式事务之前，我们先从数据库事务说起。数据库事务的几个特性：原子性(Atomicity )、一致性( Consistency )、隔离性或独立性( Isolation)和持久性(Durabilily)，简称就是**ACID**。

② **分布式理论- CAP定理**：在分布式系统中，一致性（Consistency）、可用性（Availability）和分区容忍性（Partition Tolerance）3 个要素最多只能同时满足两个，不可兼得。其中，分区容忍性又是不可或缺的。

一致性：分布式环境下多个节点的数据是否强一致。

可用性：分布式服务能一直保证可用状态。当用户发出一个请求后，服务能在有限时间内返回结果。

分区容忍性：特指对网络分区的容忍性。

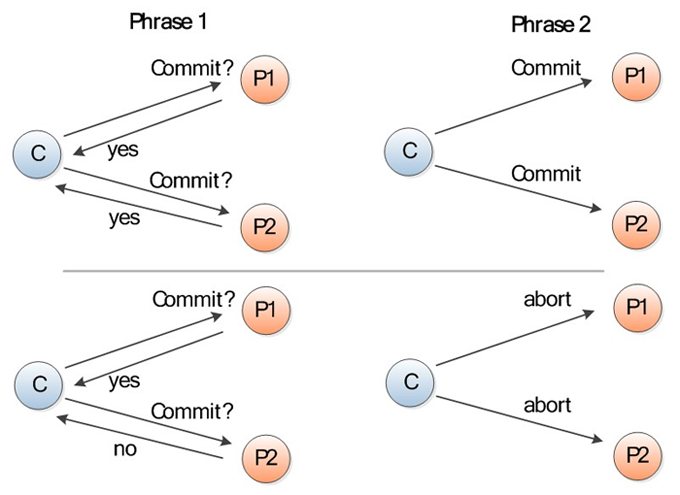
③ **BASE理论**：在分布式系统中，我们往往追求的是可用性，它的重要程序比一致性要高，那么如何实现高可用性呢？提出来了另外一个理论，就是BASE理论，它是用来对CAP定理进行进一步扩充的。BASE理论指的是：Basically Available（基本可用）、Soft state（软状态）、Eventually consistent（最终一致性）。BASE理论是对CAP中的一致性和可用性进行一个权衡的结果，理论的核心思想就是：我们无法做到强一致，但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性（Eventual consistency）。

基本可用（BasicallyAvailable）：指分布式系统在出现故障时，允许损失部分的可用性来保证核心可用。软状态（SoftState）：指允许分布式系统存在中间状态，该中间状态不会影响到系统的整体可用性。

最终一致性（EventualConsistency）：指分布式系统中的所有副本数据经过一定时间后，最终能够达到一致的状态。

④ **分布式事务：**简单的说，就是一次大的操作由不同的小操作组成，这些小的操作分布在不同的服务器上，且属于不同的应用，分布式事务需要保证这些小操作要么全部成功，要么全部失败。本质上来说，分布式事务就是为了保证不同数据库的数据一致性。

**一、两阶段提交（2PC）**：两阶段提交这种解决方案属于牺牲了一部分可用性来换取的一致性。



优点： 尽量保证了数据的强一致，适合对数据强一致要求很高的关键领域。（其实也不能100%保证强一致）

缺点： 实现复杂，牺牲了可用性，对性能影响较大，不适合高并发高性能场景。

**二、补偿事务（TCC）：**T其核心思想是：针对每个操作，都要注册一个与其对应的确认和补偿（撤销）操作。它分为三个阶段：① Try 阶段主要是对业务系统做检测及资源预留；② Confirm 阶段主要是对业务系统做确认提交，Try阶段执行成功并开始执行 Confirm阶段时，默认 Confirm阶段是不会出错的。即：只要Try成功，Confirm一定成功。③ Cancel 阶段主要是在业务执行错误，需要回滚的状态下执行的业务取消，预留资源释放。举个例子，假入 Bob 要向 Smith 转账，思路大概是：

1、首先在 Try 阶段，要先调用远程接口把 Smith 和 Bob 的钱给冻结起来。

2、在 Confirm 阶段，执行远程调用的转账的操作，转账成功进行解冻。

3、如果第2步执行成功，转账成功，如果第二步执行失败，则调用远程冻结接口对应的解冻方法 (Cancel)。

优点：跟2PC比起来，实现以及流程相对简单了一些，但数据的一致性比2PC也要差一些

缺点：在2,3步中都有可能失败。所以需要程序员在实现的时候多写很多补偿的代码。

**三、本地消息表（异步确保）：**本地消息表这种实现方式应该是业界使用最多的，其核心思想是将分布式事务拆分成本地事务进行处理。基本思路就是：

消息生产方，需要额外建一个消息表，并记录消息发送状态。消息表和业务数据要在一个事务里提交，也就是说他们要在一个数据库里面。然后消息会经过MQ发送到消息的消费方。如果消息发送失败，会进行重试发送。

消息消费方，需要处理这个消息，并完成自己的业务逻辑。此时如果本地事务处理成功，表明已经处理成功了，如果处理失败，那么就会重试执行。如果是业务上面的失败，可以给生产方发送一个业务补偿消息，通知生产方进行回滚等操作。

生产方和消费方定时扫描本地消息表，把还没处理完成的消息或者失败的消息再发送一遍。如果有靠谱的自动对账补账逻辑，这种方案还是非常实用的。

**四、MQ 事务消息**：有一些第三方的MQ是支持事务消息的，比如RocketMQ，他们支持事务消息的方式也是类似于采用的二阶段提交。

以阿里的 RocketMQ 中间件为例，其思路大致为：

第一阶段Prepared消息，会拿到消息的地址。

第二阶段执行本地事务，第三阶段通过第一阶段拿到的地址去访问消息，并修改状态。

也就是说在业务方法内要想消息队列提交两次请求，一次发送消息和一次确认消息。如果确认消息发送失败了RocketMQ会定期扫描消息集群中的事务消息，这时候发现了Prepared消息，它会向消息发送者确认，所以生产方需要实现一个check接口，RocketMQ会根据发送端设置的策略来决定是回滚还是继续发送确认消息。这样就保证了消息发送与本地事务同时成功或同时失败。

## 6 集群与负载均衡的算法与实现

**主要负载均衡方案**：

**一、HTTP 重定向负载均衡：**这种负载均衡方式仅适合WEB 服务器。用户发出请求时，负载均衡服务器会根据HTTP请求，重新计算出实际的WEB服务器地址，通过302重定向相应发送给用户浏览器。用户浏览器再根据302响应信息，对实际的WEB服务器发出请求。HTTP重定向方案有点是比较简单，缺点是性能比较差，需要2次请求才能返回实际结果,还有就是仅适合HTTP服务器使用。

**二、DNS 域名解析负载均衡：**在DNS中存储了一个域名的多个主机地址，每次域名解析请求，都可以根据负载均衡算法返回一个不同的IP地址。这样多个WEB服务器就构成了一个集群，并由DNS服务器提供了负载均衡服务。DNS域名解析负载均衡的优点是由DNS来完成负载均衡工作，服务本身不用维护负载均衡服务器的工作。缺点也是，由于负载均衡服务器不是自己维护，没法做精细控制，而且DNS在客户端往往带有缓存，服务器的变更很难及时反映到客户端上。

**三、反向代理负载均衡：**反向代理服务器位于实际的服务器之前，他能够缓存服务器响应，加速访问，同时也启到了负载均衡服务器的效果。反向代理服务器解析客户端请求，根据负载均衡算法转发到不同的后台服务器上。用户和后台服务器之间不再有直接的链接。请求，响应都由反向代理服务器进行转发。优点是和负载均衡服务集成在一起，部署简单。缺点是所有的请求回应都需要经过反向代理服务器。其本身可能会成为性能的瓶颈。著名的 Nginx服务器就可以部署为反向代理服务器，实现WEB 应用的负载均衡。

**负载均衡算法**：

**1、轮询（默认）**：每个请求按时间顺序逐一分配到不同的后端服务，如果后端某台服务器死机，自动剔除故障系统，使用户访问不受影响。

**2、weight（轮询权值）：**weight的值越大分配到的访问概率越高，主要用于后端每台服务器性能不均衡的情况下。或者仅仅为在主从的情况下设置不同的权值，达到合理有效的地利用主机资源。

**3、源地址哈希法：**每个请求按访问IP的哈希结果分配，使来自同一个IP的访客固定访问一台后端服务器，并且可以有效解决动态网页存在的session共享问题。

**4、随机法**

**5、最小连接数法**：通过“最少连接"调度算法动态地将网络请求调度到已建立的链接数最少的服务器上如果集群系统的真实服务器具有相近的系统性能,采用“最小连接"调度算法可以较好地均衡负载

**4、fair（第三方）【Nginx】：**比 weight、ip\_hash更加智能的负载均衡算法，fair算法可以根据页面大小和加载时间长短智能地进行负载均衡，也就是根据后端服务器的响应时间来分配请求，响应时间短的优先分配。Nginx本身不支持fair，需要安装upstream\_fair模块。

**5、url\_hash（第三方）【Nginx】**：按访问的URL的哈希结果来分配请求，使每个URL定向到一台后端服务器，可以进一步提高后端缓存服务器的效率。Nginx本身不支持url\_hash，则安装Nginx的hash软件包。

## 7 说说分库与分表设计

垂直(纵向)拆分：是指按功能模块拆分，以解决表与表之间的io竞争。比如分为订单库、商品库、用户库...这种方式多个数据库之间的表结构不同。水平(横向)拆分：将同一个表的数据进行分块保存到不同的数据库中，来解决单表中数据量增长出现的压力。这些数据库中的表结构完全相同。

分库分表策略：1. 按照时间区间；2. 按照主键ID区间；3. 按照指定字段hash后再取模；4. 按照用户ID取模。

分库分表的策略相对于前边两种复杂一些，一种常见的路由策略如下：

１、中间变量　＝ user\_id%（库数量\*每个库的表数量）;

２、库序号　＝　取整（中间变量／每个库的表数量）;

３、表序号　＝　中间变量％每个库的表数量;

例如：数据库有256 个，每一个库中有1024个数据表，用户的user\_id＝262145，按照上述的路由策略，可得：

１、中间变量　＝ 262145%（256\*1024）= 1;

２、库序号　＝　取整（1／1024）= 0;

３、表序号　＝　1％1024 = 1;

这样的话，对于user\_id＝262145，将被路由到第０个数据库的第１个表中。

## 8 分库与分表带来的分布式困境与应对之策

**分库分表需要解决的问题：**

**1、事务问题：**

解决事务问题目前有两种可行的方案：分布式事务

方案一：使用分布式事务

优点：交由数据库管理，简单有效

缺点：性能代价高，特别是shard越来越多时

方案二：由应用程序和数据库共同控制

原理：将一个跨多个数据库的分布式事务分拆成多个仅处 于单个数据库上面的小事务，并通过应用程序来总控 各个小事务。

优点：性能上有优势

缺点：需要应用程序在事务控制上做灵活设计。

**2、跨节点Join、count、order by、group by以及聚合函数问题**

Join：分两次查询实现。在第一次查询的结果集中找出关联数据的id,根据这些id发起第二次请求。

其他：分别在各个节点上得到结果后在应用程序端进行合并。可以并行执行。

**3、数据迁移，容量规划，扩容等问题**

利用对2的倍数取余具有向前兼容的特性（如对4取余得1的数对2取余也是1）来分配数据，避免了行级别的数据迁移，但是依然需要进行表级别的迁移，同时对扩容规模和分表数量都有限制。

**4、ID问题**

(1)UUID：UUID是一个128bit长的数字，一般用16进制表示。全球唯一; UUID往往是使用字符串存储，存储空间比较大，查询的效率比较低；不可读。

(2) Redis生成ID：利用redis的lua脚本执行功能，在每个节点上通过lua脚本生成唯一ID。

(3)Twitter的分布式自增ID算法Snowflake：使用41bit作为毫秒数，10bit作为机器的ID，（5个bit是数据中心，5个bit的机器ID），12bit作为毫秒内的流水号，最前面还有一个符号位，永远是0。

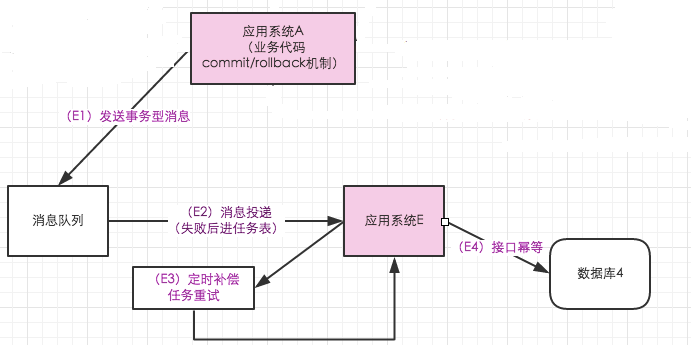
## 9 数据最终一致性方案

一、基于事务型消息队列的最终一致性

借助消息队列，在处理业务逻辑的地方，发送消息，业务逻辑处理成功后，提交消息，确保消息是发送成功的，之后消息队列投递来进行处理，如果成功，则结束，如果没有成功，则重试，直到成功，不过仅仅适用业务逻辑中，第一阶段成功，第二阶段必须成功的场景。

二、基于消息队列+定时补偿机制的最终一致性

前面部分和上面基于事务型消息的队列，不同的是，第二阶段重试的地方，不再是消息中间件自身的重试逻辑了，而是单独的补偿任务机制。其实在大多数的逻辑中，第二阶段失败的概率比较小，所以单独独立补偿任务表出来，可以更加清晰，能够比较明确的直到当前多少任务是失败的。



发送消息给消息中间件🡪消息中间件入库消息🡪消息中间件返回结果🡪业务操作🡪发送业务操作结果给消息中间件🡪更改存储中消息状态。

## 10 幂等性

幂等性：就是用户对于同一操作发起的一次请求或者多次请求的结果是一致的，不会因为多次点击而产生了副作用。更复杂的操作幂等保证是利用唯一交易号(流水号)实现。

**1.查询**：可以说是天然的幂等性，因为你查询一次和查询两次，对于系统来讲，没有任何数据的变更，所以，查询一次和查询多次一样的。

**2.MVCC方案**：合理的选择乐观锁，通过version或者其他条件，来做乐观锁。乐观锁的实现方式多种多样可以通过version或者其他状态条件：①通过版本号实现；②2. 通过条件限制。

update table\_xxx set name=#name#,version=version+1 where id=#id# and version=#version#

update table\_xxx set amount= amount-#subAmount# where id=#id# and amount-#subAmount# >= 0

**3.单独的去重表：**如果涉及到的去重的地方特别多，可以单独搞一张去重表，在插入数据的时候，插入去重表，利用数据库的唯一索引特性，保证唯一的逻辑。

**4. 分布式锁：**在业务系统插入数据或者更新数据，获取分布式锁，然后做操作，之后释放锁

**5. 删除操作**：删除数据，仅仅第一次删除是真正的操作数据，第二次甚至第三次删除，直接返回成功。

**6.插入数据的唯一索引：**插入数据的唯一性，可以通过业务主键来进行约束，例如一个特定的业务场景，三个字段肯定确定唯一性，那么，可以在数据库表添加唯一索引来进行标示。

**7. token机制，防止页面重复提交**：①数据提交前要向服务的申请token，token放到redis或jvm内存，token有效时间；②提交后后台校验token，同时删除token，生成新的token返回

**8. 对外提供接口的api：**对外提供接口为了支持幂等调用，接口有两个字段必须传，一个是来源source，一个是来源方序列号seq，这个两个字段在提供方系统里面做联合唯一索引，先在本方系统里面查询一下，是否已经处理过，返回相应处理结果；没有处理过，进行相应处理，返回结果。

**9.状态机幂等**：如果状态机已经处于下一个状态，这时候来了一个上一个状态的变更，理论上是不能够变更的，这样的话，保证了有限状态机的幂等。

# Spring

## 1 BeanFactory 和 ApplicationContext 有什么区别

**BeanFactory：**是IOC容器的核心接口，它定义了IOC的基本功能，我们看到它主要定义了getBean方法。getBean方法是IOC容器获取bean对象和引发依赖注入的起点。方法的功能是返回特定的名称的Bean。注意，BeanFactory 只能管理单例（Singleton）Bean 的生命周期。它不能管理原型(prototype,非单例)Bean 的生命周期。这是因为原型 Bean 实例被创建之后便被传给了客户端,容器失去了对它们的引用。

**ApplicationContext：**ApplicationContext由BeanFactory派生而来，提供了更多面向实际应用的功能。在BeanFactory中，很多功能需要以编程的方式实现，而在ApplicationContext中则可以通过配置实现。ApplicationContext还在功能上做了扩展，相较于BeanFactorty，ApplicationContext还提供了以下的功能：1.支持信息源，可以实现国际化。（实现MessageSource接口）； 2.访问资源。(实现ResourcePatternResolver接口，这个后面要讲)； 3.支持应用事件。(实现ApplicationEventPublisher接口)。

**FactoryBean**：在使用容器时，可以使用转义符“&”来得到FactoryBean本身，用来区分通过容器来获取FactoryBean产生的对象和获取FactoryBean本身。FactoryBean不是简单的Bean，是一个能产生或者修饰对象生成的工厂Bean，**它的实现与设计模式中的工厂模式和修饰器模式类似。**

**区别：**1.BeanFactroy采用的是延迟加载形式来注入Bean的，即只有在使用到某个Bean时(调用getBean())，才对该Bean进行加载实例化。而ApplicationContext则相反，它是在容器启动时，一次性创建了所有的Bean。占用内存空间。当应用程序配置Bean较多时，程序启动较慢。

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

3.作用：BeanFactory负责读取bean配置文档，管理bean的加载，实例化，维护bean之间的依赖关系，负责bean的声明周期。ApplicationContext除了提供上述BeanFactory所能提供的功能之外，还提供了更完整的框架功能。

## 2 Spring Bean 的生命周期

1. spring对bean进行实例化，由BeanFactory读取Bean定义文件，并生成各个实例，默认bean是单例；

2. 按照Spring上下文对实例化的Bean进行配置，也就是IOC注入；

3. 如果Bean实现了BeanNameAware接口，会调用它实现的**setBeanName**(String beanId)方法，此处传递的是Spring配置文件中Bean的ID；

4. 如果Bean实现了BeanFactoryAware接口，会调用它实现的**setBeanFactory**()，将BeanFactory实例传进来，传递的是Spring工厂本身（可以用这个方法获取到其他Bean）；

5. 如果Bean实现了ApplicationContextAware接口，会调用setApplicationContext(ApplicationContext)方法，传入Spring上下文；

6. 如果Bean实现了BeanPostProcessor接口，将会调用**postProcessBeforeInitialization**(Object obj, String s)方法，BeanPostProcessor经常被用作是Bean内容的更改，并且由于这个是在Bean初始化结束时调用After方法，也可用于内存或缓存技术；

7. 如果Bean在Spring配置文件中配置了**init-method**属性会自动调用其配置的初始化方法；

8. 如果Bean实现了BeanPostProcessor接口，将会调用**postAfterInitialization**(Object obj, String s)方法；

注意：以上工作完成以后就可以用这个Bean了，那这个Bean是一个single的，所以一般情况下我们调用同一个ID的Bean会是在内容地址相同的实例

9. 当Bean不再需要时，会经过清理阶段，如果Bean实现DisposableBean接口，调用其实现的destroy方法；

10. 最后，如果这个Bean的Spring配置中配置了destroy-method属性，会自动调用其配置的销毁方法。

以上10步骤可以作为面试或者笔试的模板，另外我们这里描述的是应用Spring上下文Bean的生命周期，如果应用Spring的工厂也就是BeanFactory的话去掉第5步就Ok了。一旦把一个Bean纳入Spring IOC容器之中，这个Bean的生命周期就会交由容器进行管理，一般担当管理角色的是BeanFactory或者ApplicationContext。

## 3 Spring IOC

**(1)什么是IOC/DI**：**所谓IoC，对于spring框架来说，就是由spring来负责控制对象的生命周期和对象间的关系。**在传统的java应用开发中，我们要实现某一个功能至少需要两个或以上的对象来协作完成，在没有使用Spring的时候，每个对象在需要使用他的合作对象时，自己均要使用像**new object()** 这样的语法来将合作对象创建出来，这个合作对象是由自己主动创建出来的，创建合作对象的主动权在自己手上，自己需要哪个合作对象，就主动去创建，创建合作对象的主动权和创建时机是由自己把控的，而这样就会使得对象间的耦合度高了。而使用了Spring之后就不一样了，创建合作对象B的工作是由Spring来做的，Spring创建好B对象，然后存储到一个容器里面，当A对象需要使用B对象时，Spring就从存放对象的那个容器里面取出A要使用的那个B对象，然后交给A对象使用，至于Spring是如何创建那个对象，以及什么时候创建好对象的，A对象不需要关心这些细节问题(你是什么时候生的，怎么生出来的我可不关心，能帮我干活就行)。

DI是由Martin Fowler 在2004年初的一篇论文中首次提出的。他总结：控制的什么被反转了？就是：**获得依赖对象的方式反转**了。IoC的一个重点是在系统运行中，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（Dependency Injection，依赖注入）来实现的。

(2) **IoC容器的初始化：**IoC容器的初始化是由refresh()方法来启动的，这个方法标志着IoC容器的正式启动，具体包括①BeanDefinition的Resource定位；②载入；③注册这三个基本的过程。第一个过程是Resource定位过程，这个Resource定位指的是BeanDefinition的资源定位，它由ResourceLoader通过统一的Resource接口来完成；第二个过程是BeanDefinition的载入，这个载入过程是把用户定义好的Bean表示成IoC容器内部的数据结构；第三个过程是想IoC容器注册这些BeanDefinition的过程，这个过程是通过调用BeanDefinitionRegistry接口的实现来完成（将BeanDefinition注入到一个HashMap中）。

下面以FileSystemXmlApplicationContext为例：

**Resource定位过程：①**FileSystemXmlApplicationContext构造器：首先，调用父类容器的构造方法为容器设置好Bean资源加载器；然后，再调用父类AbstractRefreshableConfigApplicationContext的方法设置Bean定义资源文件的定位路径。然后调用父类AbstractApplicationContext. refresh()方法实现BeanFactory的更新；**②refresh()是一个模板方法**，refresh()方法的作用是：在创建IoC容器前，如果已经有容器存在，则需要把已有的容器销毁和关闭，以保证在refresh之后使用的是新建立起来的IoC容器。调用**obtainFreshBeanFactory函数载入；③**调用**子类refreshBeanFactory**函数；**④**refreshBeanFactory首先判断BeanFactory是否存在，如果存在则先销毁beans并关闭beanFactory，接着调用createBeanFactory()创建IoC容器，使用的是DefaultListableBeanFactory；然后调用**子类AbstractXmlApplicationContext 的**loadBeanDefinitions(beanFactory)装载bean；**⑤loadBeanDefinitions**：创建**XmlBeanDefinitionReader**，即创建Bean读取器，容器使用该读取器读取Bean定义资源；调用其父类AbstractBeanDefinitionReader的 reader.loadBeanDefinitions方法读取Bean定义资源；**⑥**loadBeanDefinitions：首先调用DefaultResourceLoader的getResource完成具体的Resource定位；然后调用其子类XmlBeanDefinitionReader的loadBeanDefinitions方法真正执行加载功能。

DefaultResourceLoader中的getSource()方法，因为FileSystemXmlApplicationContext本身就是DefaultResourceLoader的实现类，所以此时又回到了FileSystemXmlApplicationContext中来。

**BeanDefinition的载入和解析：①**XmlBeanDefinitionReader加载Bean定义资源。首先从特定XML文件中实际载入Bean定义资源，将XML文件转换为DOM对象，解析过程由documentLoader实现；然后按照Spring的Bean规则对Document对象解析的过程是在接口实现类**DefaultBeanDefinitionDocumentReader**中实现的。经过对Spring Bean定义资源文件转换的Document对象中的元素层层解析，将XML形式定义的Bean定义资源文件转换为Spring IoC所识别的数据结构——BeanDefinition。

**注册：**真正完成注册功能的是DefaultListableBeanFactory。使用一个HashMap的集合对象存放。注册的过程中需要线程同步，以保证数据的一致性。

**总结：**①初始化的入口在容器实现中的 refresh()调用来完成；②对 bean 定义载入 IOC 容器使用的方法是 loadBeanDefinition,其中的大致过程如下：通过 ResourceLoader 来完成资源文件位置的定位，DefaultResourceLoader 是默认的实现，同时上下文本身就给出了 ResourceLoader 的实现，可以从类路径，文件系统, URL 等方式来定为资源位置。如果是 XmlBeanFactory作为 IOC 容器，那么需要为它指定 bean 定义的资源，也就是说 bean 定义文件时通过抽象成 Resource 来被 IOC 容器处理的，容器通过 BeanDefinitionReader来完成定义信息的解析和 Bean 信息的注册,往往使用的是XmlBeanDefinitionReader 来解析 bean 的 xml 定义文件 - 实际的处理过程是委托给 BeanDefinitionParserDelegate 来完成的，从而得到 bean 的定义信息，这些信息在 Spring 中使用 BeanDefinition 对象来表示，容器解析得到 BeanDefinition以后，需要把它在 IOC 容器中注册，这由 IOC 实现 BeanDefinitionRegistry 接口来实现。注册过程就是在 IOC 容器内部维护的一个HashMap 来保存得到的 BeanDefinition 的过程。这个 HashMap 是 IoC 容器持有 bean 信息的场所，以后对 bean 的操作都是围绕这个HashMap 来实现的。

**(3)依赖注入：**依赖注入在以下两种情况发生：①用户第一次通过getBean方法向IoC容索要Bean时，IoC容器触发依赖注入。②当用户在Bean定义资源中为<Bean>元素配置了lazy-init属性，即让容器在解析注册Bean定义时进行预实例化，触发依赖注入。

①在BeanFactory中我们看到getBean（String…）函数，它的具体实现在AbstractBeanFactory中，AbstractBeanFactory通过getBean向IoC容器获取被管理的Bean：如果Bean定义的单态模式(Singleton)，则容器在创建之前先从缓存中查找，确保整个容器中只存在一个实例对象。如果Bean定义的是原型模式，则容器每次都会创建一个新的实例对象。除此之外，Bean定义还可以扩展为指定其生命周期范围。

②具体的Bean实例对象的创建过程由实现了ObejctFactory接口的匿名内部类的createBean方法完成。ObejctFactory使用委派模式，具体的Bean实例创建过程交由其实现类**AbstractAutowireCapableBeanFactory**完成，调用createBeanInstance生成Bean所包含的java对象实例，**调用populateBean对Bean属性的依赖注入进行处理。**

③在createBeanInstance方法中，根据指定的初始化策略，使用静态工厂、工厂方法或者容器的自动装配特性生成java实例对象。调用相应的工厂方法或者参数匹配的构造方法即可完成实例化对象的工作，但是对于我们最常使用的默认无参构造方法就需要使用相应的初始化策略(JDK的反射机制或者CGLIB)来进行初始化了。

④populateBean：调用applyPropertyValues对属性进行注入：当属性值类型不需要转换时，直接准备进行依赖注入；当属性值需要进行类型转换时，如对其他对象的引用等，首先需要解析属性值，由**resolveValueIfNecessary**方法实现；然后对解析后的属性值通过BeanWrapper.setPropertyValues进行依赖注入，具体实现在BeanWrapperImpl。

Spring IoC容器是如何将属性的值注入到Bean实例对象中去的：

(1).对于集合类型的属性，将其属性值解析为目标类型的集合后直接赋值给属性。

(2).对于非集合类型的属性，大量使用了JDK的反射和内省机制，通过属性的getter方法(reader method)获取指定属性注入以前的值，同时调用属性的setter方法(writer method)为属性设置注入后的值。看到这里相信很多人都明白了Spring的**setter注入**原理。

## 4 Autowired

容器对Bean实例对象的属性注入的处理发生在AbstractAutoWireCapableBeanFactory类中的populateBean方法中，我们通过程序流程分析autowiring的实现原理：

(1). AbstractAutoWireCapableBeanFactory对Bean实例进行属性依赖注入：应用第一次通过getBean方法(配置了lazy-init预实例化属性的除外)向IoC容器索取Bean时，容器创建Bean实例对象，并且对Bean实例对象进行属性依赖注入，populateBean方法就是实现Bean属性依赖注入的功能.

(2).Spring IoC容器根据Bean名称或者类型进行autowiring自动依赖注入：

通过属性名进行自动依赖注入的相对比通过属性类型进行自动依赖注入要稍微简单一些，但是真正实现属性注入的是DefaultSingletonBeanRegistry类的registerDependentBean方法。

(3).DefaultSingletonBeanRegistry的registerDependentBean方法对属性注入：

通过对autowiring的源码分析，我们可以看出，autowiring的实现过程：

a.对Bean的属性迭代调用getBean方法，完成依赖Bean的初始化和依赖注入。

b.将依赖Bean的属性引用设置到被依赖的Bean属性上。

c.将依赖Bean的名称和被依赖Bean的名称存储在IoC容器的集合中。

## 5 Spring AOP

Spring提供了两种方式来生成代理对象: **JDKProxy和Cglib**，具体使用哪种方式生成由AopProxyFactory根据AdvisedSupport对象的配置来决定。默认的策略是如果目标类是接口，则使用JDK动态代理技术，否则使用Cglib来生成代理。

①AbstractAutowireCapableBeanFactory类的initializeBean方法会对创建完成的Bean进行初始化，然后调用postProcessAfterInitialization()为容器产生的Bean实例对象添加BeanPostProcessor后置处理器，具体由BeanPostProcessor的子类实现。②AbstractAutoProxyCreator.java类的postProcessAfterInitialization方法中，调用wrapIfNecessary；③wrapIfNecessary中Spring在这里先获取配置好的Advisor信息，然后调用createProxy方法为目标对象创建了代理，接着将创建的代理对象返回。

(1) **初始化AopProxy：**①ProxyCreatorSupport.java类的createAopProxy方法。这就是生成代理的入口，你会发现传入的参数是是this，其实传入的就是this的父类AdvisedSupport.java中的advisor等生成代理的核心参数。②调用AopProxyFactory的createAopProxy，实际调用子类DefaultAopProxyFactory；③根据代理的目标对象是否实现了接口，来返回JdkDynamicAopProxy的动态代理或者CGLIB的代理，并且传入advisor核心参数（JdkDynamicAopProxy这个实现了InvocationHandler，要实现invoke的关键就是传入的advisor）。

**(2)获取代理对象：**getProxy方法，使用java reflect包提供的原生创建代理类方法Proxy.newProxyInstance方法创建并返回目标对象的代理对象。

## 6 动态代理（cglib 与 JDK动态代理）

|  |  |
| --- | --- |
| public interface Service {  public void add();  } | public class UserServiceImpl implements Service {  @Override  public void add() {  System.out.println("This is add service");  }  } |

|  |  |
| --- | --- |
| public static void main(String[] args) {  Service service = new UserServiceImpl();  MyInvocatioHandler handler = new  MyInvocatioHandler(service);  Service serviceProxy = (Service)handler.getProxy();  serviceProxy.add();  } | public static void main(String[] args) {  UserServiceImpl service = new UserServiceImpl();  CGLIBProxy cglib = new CGLIBProxy();// 创建CglibProxy对象  UserServiceImpl proxy = (UserServiceImpl)  cglib.getProxy(service); //17 生成代理类  proxy.add();  } |

**6.1 动态代理**

|  |
| --- |
| class MyInvocatioHandler implements InvocationHandler {  private Object target;  public MyInvocatioHandler(Object target) {  this.target = target;  }  @Override  public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  System.out.println("-----before-----");  Object result = method.invoke(target, args);  System.out.println("-----end-----");  return result;  }  // 生成代理对象  public Object getProxy() {  ClassLoader loader = Thread.currentThread().getContextClassLoader();  Class<?>[] interfaces = target.getClass().getInterfaces();  return Proxy.newProxyInstance(loader, interfaces, this);  }  } |

它的好处理时可以为我们生成任何一个接口的代理类，并将需要增强的方法织入到任意目标函数。但它仍然具有一个局限性，就是**只有实现了接口的类，才能为其实现代理**。

**6.2 CGLIB**

|  |
| --- |
| public class CGLIBProxy implements MethodInterceptor {  private Object target;  public Object getProxy(Object target) {  this.target = target; //给业务对象赋值  Enhancer enhancer = new Enhancer(); //创建加强器，用来创建动态代理类  enhancer.setSuperclass(this.target.getClass()); //为加强器指定要代理的业务类（即：为下面生成的代理类指定父类）  enhancer.setCallback(this); //设置回调：对于代理类上所有方法的调用，都会调用CallBack  return enhancer.create(); // 创建动态代理类对象并返回  }  @Override  public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {  System.out.println("预处理——————");  Object result = proxy.invokeSuper(obj, args);  System.out.println("调用后操作——————");  return result;  }  } |

CGLIB解决了动态代理的难题，它通过生成目标类子类的方式来实现来实现代理，而不是接口，规避了接口的局限性。当然CGLIB也具有局限性，对于无法生成子类的类（final类），肯定是没有办法生成代理子类的。

## 7 Spring 事务实现方式

①引入一系列的约束头文件以及标签；②配置C3P0数据源以及DAO、Service；③配置事务管理器以及事务代理工厂Bean。事务管理器：DataSourceTransactionManager

方式一：通过事务代理工厂bean进行配置[XML方式]：事务代理工厂：注入事务管理器，隔离级别等

方式二：注解：@Transactional

方式三：Aspectj AOP配置事务：<tx:advice>事务管理传播配置

## 8 Spring 事务

**(1) 数据库事务的四大特性：**

**ACID，**指数据库事务正确执行的四个基本要素的缩写。包含：

* 原子性（Atomicity）：原子性是指事务包含的所有操作要么全部成功，要么全部失败回滚；
* 一致性（Consistency）：一致性是指事务必须使数据库从一个一致性状态变换到另一个一致性状态；
* 隔离性（Isolation）：隔离性是当多个用户并发访问数据库时，数据库为每一个用户开启的事务，不能被其他事务的操作所干扰，多个并发事务之间要相互隔离。
* 持久性（Durability）。持久性是指一个事务一旦被提交了，那么对数据库中的数据的改变就是永久性的，即便是在数据库系统遇到故障的情况下也不会丢失提交事务的操作。

一个支持事务（Transaction）的数据库，必须要具有这四种特性。

**(2) 数据库隔离级别：**

* READ UNCIMMITTED（未提交读）：事务还没提交，而别的事务可以看到修改的数据，也就是脏读；
* READ COMMITTED（提交读）：只能看到已经完成的事务的结果，正在执行的，是无法被其他事务看到的。可防止脏读，但幻读和不可重复读仍可发生；
* REPEATABLE READ（可重复读）：对相同字段的多次读取是一致的，除非数据被事务本身改变。可防止脏、不可重复读，但幻读仍可能发生
* SERIALIZABLE（可串行化）：完全服从ACID的隔离级别，确保不发生脏、幻、不可重复读。这在所有的隔离级别中是最慢的，它是典型的通过完全锁定在事务中涉及的数据表来完成的。

**MySQL默认采用REPEATABLE\_READ隔离级别；Oracle默认采用READ\_COMMITTED隔离级别**

**(3) 事务传播行为：（七种）**

REQUIRED--支持当前事务，如果当前没有事务，就新建一个事务。这是最常见的选择。

SUPPORTS--支持当前事务，如果当前没有事务，就以非事务方式执行。

MANDATORY--支持当前事务，如果当前没有事务，就抛出异常。

REQUIRES\_NEW--新建事务，如果当前存在事务，把当前事务挂起。

NOT\_SUPPORTED--以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

NEVER--以非事务方式执行，如果当前存在事务，则抛出异常。

NESTED--如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则进行与REQUIRED类似的操作。拥有多个可以回滚的保存点，内部回滚不会对外部事务产生影响。

## 9 Spring MVC 运行流程

①客户端发出一个http请求给web服务器，web服务器对http请求进行解析，如果匹配DispatcherServlet的请求映射路径（在web.xml中指定），web容器将请求转交给**DispatcherServlet**。

②DispatcherServlet收到请求调用HandlerMapping处理器映射器，找到具体的处理器(可以根据xml配置、注解进行查找)，生成处理器对象及处理器拦截器(如果有则生成)一并返回给DispatcherServlet。

③DispatcherServlet调用HandlerAdapter处理器适配器。

④HandlerAdapter经过适配调用具体的处理器(Controller，也叫后端控制器)。

⑤Controller执行完成返回ModelAndView。

⑥HandlerAdapter将controller执行结果ModelAndView返回给DispatcherServlet。

⑦DispatcherServlet将ModelAndView传给ViewReslover视图解析器。

⑧ViewReslover解析后返回具体View。

⑨DispatcherServlet根据View进行渲染视图（即将模型数据填充至视图中）。

⑩DispatcherServlet响应用户。

## 10 Spring 的单例实现原理

Spring的依赖注入（包括lazy-init方式）都是发生在 **AbstractBeanFactory** 的 getBean 里。 getBean 的 doGetBean 方法调用 getSingleton 进行bean的创建。lazy-init方式，在容器初始化时候进行调用，非lazy-init方式，在用户向容器第一次索要bean时进行调用。

1. **doGetBean**：(1)根据指定的名称获取被管理Bean的名称，如果指定的是别名，将别名转换为规范的Bean名称；(2)调用**getSingleton**()先从缓存中取是否已经有被创建过的单态类型的Bean(IoC容器中只创建一次)；(3)如果指定名称的Bean在容器中已有单态模式的Bean被创建，直接返回；(4)如果没有，委托当前容器的父级容器去查找，如果找到则返回；(5)如果一直找不到，就创建bean：①先获取其父级的Bean定义，主要解决B继承时子类合并父类公共属性；②获取当前Bean所有依赖Bean的名称；③createBean()创建bean。

2. **getSingleton**：使用了**双重判断加锁**的单例模式。(1)首先从缓存singletonObjects（实际上是一个map，存储的是完全实例化的BeanName->Bean Instance）中获取bean实例；(2)如果为null，但该bean正在创建过程中，则对缓存singletonObjects**加锁**；(3) 然后再从尝试从earlySingletonObjects中获取。如果没有，则从singletonFactories中获取。这是因为spring创建单例bean的时候，存在循环依赖的问题，为了解决循环依赖的问题，spring采取了一种将创建的bean实例提早暴露加入到缓存中，一旦下一个bean创建的时候需要依赖上个bean，则直接使用ObjectFactory来获取bean。提前暴露bean实例到缓存的时机是在bean实例创建（调用构造方法）之后，初始化bean实例（属性注入）之前。在从singletonFactories获取bean后，会将其存储到earlySingletonObjects中，然后从singletonFactories移除该bean，之后在要获取该bean就直接从earlySingletonObjects获取。(4)如果找到就返回bean实例，否则返回null。

3. **doCreateBean**：如果Bean定义的单态模式(Singleton)，则容器在创建之前先从缓存中查找，以确保整个容器中只存在一个实例对象。如果Bean定义的是原型模式(Prototype)，则容器每次都会创建一个新的实例对象。除此之外，Bean定义还可以扩展为指定其生命周期范围。生成Bean所包含的java对象实例；对Bean属性的依赖注入进行处理。

## 11 Spring 框架中用到了哪些设计模式

(1) 单例模式：AbstractBeanFactory通过getBean方法向IoC容器索要Bean时默认使用单例模式；

(2) 代理模式：Spring Aop 中 Jdk 动态代理就是利用代理模式技术实现的；

(3) 工厂模式：Spring Bean的创建是典型的工厂模式。

(4) 装饰器：FactoryBean是一个能产生或者修饰对象生成的工厂Bean，它的实现与设计模式中的工厂模式和装饰器模式类似。

(5) 解释器:Spring主要以Spring Expression Language（SpEL）为例

(6) 模板模式：JdbcTemplate；

(7) 策略模式：FactoryBean接口为Spring容器提供了一个很好的封装机制，具体的getObject有不同的实现类根据不同的实现策略来具体提供。

# 多线程

## 1 创建线程的方式及实现

Java中创建线程主要有三种方式：

(1)继承Thread类创建线程类，并重写该类的run方法；

|  |
| --- |
| public class FirstThread extends Thread {  private int i;  public void run() {  for (i = 0; i < 10; i++) {  System.out.println(**getName()** + " " + i);  }  }  public static void main(String[] args) {  **new FirstThread ().start();**  **new FirstThread ().start();**  for (int i = 0; i < 10; i++) {  System.out.println(Thread.currentThread().getName() + " : " + i);  }  }  } |

(2)实现Runnable接口创建线程类，并重写该接口的run()方法；

|  |
| --- |
| public class RunnableThreadTest implements Runnable {  private int i;  public void run() {  for (i = 0; i < 10; i++) {  System.out.println(**Thread.currentThread().getName()** + " " + i);  }  }  public static void main(String[] args) {  for (int i = 0; i < 10; i++) {  System.out.println(Thread.currentThread().getName() + " " + i);  }  **RunnableThreadTest rtt = new RunnableThreadTest();**  **new Thread(rtt, "新线程1").start();**  **new Thread(rtt, "新线程2").start();**  }  } |

(3) 通过Callable和Future创建线程；并实现call()方法。

|  |
| --- |
| public class CallableThreadTest implements Callable<Integer> {  @Override  public Integer call() throws Exception {  int i = 0;  for (; i < 10; i++) {  System.out.println(Thread.currentThread().getName() + " " + i);  }  return i;  }  public static void main(String[] args) {  for (int i = 0; i < 10; i++) {  System.out.println(Thread.currentThread().getName() + " 的循环变量i的值 " + i);  }  **CallableThreadTest ctt = new CallableThreadTest();**  **FutureTask<Integer> ft = new FutureTask<Integer>(ctt);**  **new Thread(ft, "有返回值的线程").start();**  try {  System.out.println("子线程的返回值" + ft.get()); // get()方法来获得子线程执行结束后的返回值  } catch (Exception e) {  e.printStackTrace();  }  }  } |

对比：

(1) 采用接口的方式，还可以继承其他类。在这种方式下，多个线程可以共享同一个target对象，所以非常适合多个相同线程来处理同一份资源的情况。如果要访问当前线程，则必须使用Thread.currentThread()方法。

(2) 使用继承Thread类的方式，如果需要访问当前线程，则无需使用Thread.currentThread()方法，直接使用this即可获得当前线程。

## 2 sleep 、join、yield有什么区别

**(1) sleep**

Thread类的方法，必须带一个时间参数，它可以让当前正在执行的线程在指定的时间内暂停执行，进入阻塞状态并释放CPU，提供其他线程运行的机会且不考虑优先级，但不会释放“锁标志”，也就是说如果有synchronized同步块，其他线程不能获得同步锁。

**(2) yield**

谦让：Thread类的方法，类似sleep但无法指定时间并且只会提供相同或更高优先级的线程运行的机会，也不会释放“锁标志”，不推荐使用。

**(3) wait**

Object类的方法，必须放在循环体和同步代码块中，执行该方法的线程会释放锁，进入线程等待池中等待被再次唤醒(notify随机唤醒，notifyAll全部唤醒，线程结束自动唤醒)即放入锁池中竞争同步锁；

线程A调用 wait()，A就转为等待状态；线程A 调用 notify，其他线程被唤醒争抢资源；

**(4) join**

一种特殊的wait，当前运行线程调用另一个线程的join方法，当前线程进入阻塞状态直到另一个线程运行结束；在线程A中调用B.join() 表示阻塞A线程，直到线程B 运行结束或超时。

## 3 并发工具类- CountDownLatch

CountDownLatch允许一个或多个线程一直等待其他线程完成操作。CountDownLatch通过AQS里面的共享锁来实现的。在构造方法里，会创建一个Sync[sɪŋk]对象。Sync 是CountDownLatch的内部私有类，继承AQS。

在创建CountDownLatch实例时，需要传递一个int型的参数count，该参数为计数器的初始值，表示该共享锁可以获取的总次数，会赋值给了AQS中state(private volatile long)。当某个线程调用await()方法时，程序首先判断state的值是否为0，如果不会0的话则会一直等待直到为0为止。当其他线程调用countDown()方法时，则执行释放共享锁状态，使count值 - 1。当在创建CountDownLatch时初始化的count参数，必须要有count线程调用countDown方法才会使计数器count等于0，锁才会释放，前面等待的线程才会继续运行。注意CountDownLatch不能回滚重置。

## 4并发工具类-CyclicBarrier

## 5并发工具类-Semaphore

## 5、线程池用过吗都有什么参数?底层如何实现的?

## 6、sychnized和Lock什么区别?sychnize 什么情况情况是对象锁? 什么时候是全局锁为什么?

synchronized(this)以及非static的synchronized方法，只能防止多个线程同时执行同一个对象的同步代码段。**synchronized锁住的是括号里的对象，而不是代码。对于非static的synchronized方法，锁的就是对象本身也就是this。**

用synchronized(Sync.class)实现了全局锁的效果。

static synchronized方法，static方法可以直接类名加方法名调用，方法中无法使用this，所以它锁的不是this，而是类的Class对象，所以，static synchronized方法也相当于全局锁，相当于锁住了代码段。

## 7、ThreadLocal 是什么底层如何实现?写一个例子呗?

ThreadLocal类用来提供线程内部的局部变量。这些变量在多线程环境下访问(通过get或set方法访问)时能保证各个线程里的变量相对独立于其他线程内的变量，ThreadLocal实例通常来说都是private static类型。

ThreadLocal不是为了解决多线程访问共享变量，而是**为每个线程创建一个单独的变量副本**，每个线程都可以修改自己所拥有的变量副本, 而不会影响其他线程的副本. 其实这也是解决线程安全的问题的一种方法.

ThreadLocal是如何做到为每一个线程维护变量的副本的呢？其实实现的思路很简单：在ThreadLocal类中有一个静态内部类ThreadLocalMap(其类似于Map)，用键值对的形式存储每一个线程的变量副本，ThreadLocalMap中元素的key为当前ThreadLocal对象，而value对应线程的变量副本，每个线程可能存在多个ThreadLocal。

ThreadLocalMap中的每个节点Entry,其键key是ThreadLocal并且还是弱引用，这也导致了后续会产生内存泄漏问题的原因。每个thread中都存在一个map, map的类型是ThreadLocal.ThreadLocalMap. Map中的key为一个threadlocal实例. 这个Map的确使用了弱引用,**不过弱引用只是针对key.** 每个key都弱引用指向threadlocal. **当把threadlocal实例置为null以后,没有任何强引用指向threadlocal实例,所以threadlocal将会被gc回收.** 但是,我们的value却不能回收,因为存在一条从current thread连接过来的强引用. 只有当前thread结束以后, current thread就不会存在栈中,强引用断开, Current Thread, Map, value将全部被GC回收.所以得出一个结论就是只要这个线程对象被gc回收，就不会出现内存泄露，但在threadLocal设为null和线程结束这段时间不会被回收的，就发生了我们认为的内存泄露。比如使用线程池的时候，线程结束是不会销毁的，会再次使用的。就可能出现内存泄露。

public class TreadLocalTest {

static class Student {

private int age;

public int getAge() {

return age;

}

public void setAge(int age) {

this.age = age;

}

}

public static void main(String[] args) {

TestThreadLocal t = new TestThreadLocal();

new Thread(t, "Thread A").start();

new Thread(t, "Thread B").start();

}

static class TestThreadLocal implements Runnable {

ThreadLocal<Student> studentLocal = new ThreadLocal<Student>();

@Override

public void run() {

String currentThreadName = Thread.currentThread().getName();

System.out.println(currentThreadName + " is running...");

Random random = new Random();

int age = random.nextInt(100);

System.out.println(currentThreadName + " is set age: " + age);

Student s = getStudent(); //通过这个方法，为每个线程都独立的new一个student对象，

s.setAge(age);

System.out.println("current thread first get age " + currentThreadName + ":" + s.getAge());

}

public Student getStudent() {

Student s = (Student) studentLocal.get();

if (s == null) {

s = new Student();

studentLocal.set(s);

}

return s;

}

}

}

## 1 无锁操作-CAS

**(1) Compare and Swap, 即比较并交换。**

java.util.concurrent包中借助CAS实现了区别于**synchronized**同步锁的一种乐观锁。整个AQS同步组件、Atomic原子类操作等等都是以CAS实现的。CAS有3个操作数，内存值V，旧的预期值A，要修改的新值B。当且仅当预期值A和内存值V相同时，将内存值V修改为B，否则说明已经有其他线程做了更新，什么都不做。

CAS通过调用JNI的代码实现的。JNI:Java Native Interface为JAVA本地调用，允许java调用其他语言。而compareAndSwapInt(有四个参数，分别代表：对象、对象的地址、预期值、修改值)就是借助C来调用CPU底层指令实现的。因为CAS是一种系统原语，原语属于操作系统用语范畴，是由若干条指令组成的，用于完成某个功能的一个过程，并且原语的执行必须是连续的，在执行过程中不允许被中断，也就是说CAS是一条CPU的原子指令，不会造成所谓的数据不一致问题。

**(2) CAS实现原子操作的三大问题**

**①ABA问题**：

CAS需要检查值有没有发生变化，如果没有变化则更新，但是如果一个值**原来是A，变成B，又变成了A**，那么使用CAS进行检查时会发现它的值没有发生变化，实际上变化了。

ABA问题的解决思路就是：使用版本号。在变量前面追加上版本号，每次改变时加1，即A —> B —> A，变成1A —> 2B —> 3A。

从JDK1.5开始，JDK的Atomic包里提供了一个类AtomicStampedReference来解决ABA问题。通过Pair(AtomicStampedReference的内部类)，主要用于记录引用和版本戳信息包装，从而避免ABA问题。这个类的compareAndSet(有四个参数，分别表示：预期引用、更新后的引用、预期标志、更新后的标志)方法的作用是首先检查当前引用是否等于预期引用，并且检查当前标志是否等于预期标志，如果全部相等，则以原子方式设置更新值。

注：atomicStampedReference.compareAndSet(100, 110,

**atomicStampedReference.getStamp()**, **atomicStampedReference.getStamp() + 1**);

**②循环时间长开销大**

自旋CAS如果长时间不成功，会给CPU带来非常大的执行开销。

**③只能保证一个共享变量的原子操作**

对多个共享变量操作时，循环CAS无法保证操作的原子性，这个时候可以用锁。从JDK1.5开始，JDK提供了**AtomicReference类来保证引用对象之间的原子性**，可以把多个变量放在一个对象里来进行CAS操作。

## 2 轻级的锁-volatile

(1) Volatile是轻量级的synchronized，它在多处理器开发中保证了共享变量的“可见性”。即当一个线程修改了一个被volatile修饰共享变量的值，新值可以被其他线程立即得知。确保所有线程看到这个变量的值是一致的。Volatile不会引起线程上下文的切换和调度。

(2) **volatile的特性：**①可见性：当读一个volatile变量时，JMM会把该线程对应的本地内存置为无效。线程从主内存中读取共享变量。**②原子性**：对任意单个volatile变量的读/写具有原子性，但类似于volatile++这种复合操作不具有原子性。**③禁止指令重排序优化**【1.5才完全恢复】。

**(3) Volatile底层实现：**在JVM底层volatile是采用“内存屏障”来实现的。volatile关键字时，会多出一个lock前缀指令。lock前缀指令其实就相当于一个内存屏障。内存屏障会提供3个功能：①如果第一个操作为volatile读，则不管第二个操作是啥，都不能重排序。当第二个操作为volatile写是，则不管第一个操作是啥，都不能重排序。当第一个操作volatile写，第二操作为volatile读时，不能重排序；②它会强制将对缓存的修改操作立即写入主存；③如果是写操作，它会导致其他CPU中对应的缓存行无效。

## 3 重量级的锁：synchronized

JavaSE1.6对synchronized进行了各种优化，有些情况下他就并不那么重了。

synchronized可以保证方法或者代码块在运行时，同一时刻只有一个方法可以进入到临界区，同时它还可以保证共享变量的内存可见性。

(1) **Java中的每一个对象都可以作为锁**

对于普通同步方法，锁是当前实例对象

对于静态同步方法，锁是当前类的class对象

对于同步方法块，锁是synchronized括号里配置的对象

**(2) synchronized底层原理**

Java 虚拟机中的同步(Synchronization)基于进入和退出管程(Monitor)对象实现。同步代码块是使用monitorenter和monitorexit指令实现的，而同步方法并非如此，是由方法调用指令读取运行时常量池中方法的 ACC\_SYNCHRONIZED 标志来隐式实现的。

①同步代码块：同步语句块的实现使用的是monitorenter 和 monitorexit 指令，其中monitorenter指令指向同步代码块的开始位置，monitorexit指令则指明同步代码块的结束位置，当执行monitorenter指令时，当前线程将试图获取 objectref(即对象锁) 所对应的 monitor 的持有权，当计数器为 0，那线程可以成功取得 monitor，并将计数器值设置为 1，取锁成功。如果当前线程已经拥持有权，那它可以重入这个 monitor，重入时计数器的值也会加 1。倘若其他线程已经拥有所有权，那当前线程将被阻塞，直到正在执行线程执行完毕，即monitorexit指令被执行，执行线程将释放 monitor(锁)并设置计数器值为0 ，其他线程将有机会持有 monitor 。

②同步方法：JVM可以从方法常量池中的方法表结构中的 ACC\_SYNCHRONIZED 访问标志区分一个方法是否同步方法。当方法调用时，调用指令将会 检查方法的 ACC\_SYNCHRONIZED 访问标志是否被设置，如果设置了，执行线程将先持有monitor。

**(3) Java对象头**

Synchronized用的锁对象是存储在Java对象头里的，jvm中采用2个字来存储对象头(如果对象是数组则会分配3个字，多出来的1个字记录的是数组长度)，其主要结构是由Mark Word 和 Class Metadata Address 组成。

Java对象头里的mark word里默认储存对象的hashcode、分代年龄和锁标记位。mark word 里存储的数据会随着锁标志位的变化而变化。锁标志位分为：轻量级锁00、重量级锁10、GC标记11、偏向锁01。

**(4) Java虚拟机对synchronized的优化**

在JavaSE1.6中，**引入了轻量级锁和偏向锁**。锁一共有4中状态，级别从低到高依次是：无锁状态、偏向锁状态、轻量级锁、重量级锁。锁可以升级但不能降级。

**①偏向锁：**偏向锁就是在**无竞争的情况下把整个同步都消除掉**，连CAS都不做了。偏向锁的核心思想是，如果一个线程获得了锁，那么锁就进入偏向模式，此时Mark Word 的结构也变为偏向锁结构，当这个线程再次请求锁时，无需再做任何同步操作。所以，对于没有锁竞争的场合，偏向锁有很好的优化效果，但是对于锁竞争比较激烈的场合，偏向锁就失效了，升级为轻量级锁。

**②轻量级锁：**轻量级锁能够提升程序性能的依据是“对绝大部分的锁，在整个同步周期内都不存在竞争”，注意这是经验数据。需要了解的是，轻量级锁所适应的场景是线程交替执行同步块的场合，如果存在同一时间访问同一锁的场合，就会导致轻量级锁膨胀为重量级锁。

**③自旋锁：**轻量级锁失败后，虚拟机为了避免线程真实地在操作系统层面挂起，还会进行一项称为自旋锁的优化手段。因此自旋锁会假设在不久将来，当前的线程可以获得锁，因此虚拟机会让当前想要获取锁的线程做几个空循环(这也是称为自旋的原因)，一般不会太久，可能是50个循环或100循环，在经过若干次循环后，如果得到锁，就顺利进入临界区。如果还不能获得锁，那就会将线程在操作系统层面挂起，这就是自旋锁的优化方式，这种方式确实也是可以提升效率的。最后没办法也就只能升级为重量级锁了。

**(5) 其他**

①Synchronized是重入锁；②线程的中断操作对于正在等待获取的锁对象的synchronized方法或者代码块并不起作用；如果一个线程在等待锁，要么它获得这把锁继续执行，要么它就保存等待，即使调用中断线程的方法，也不会生效；③ 等待唤醒notify/notifyAll和wait方法，必须处于synchronized代码块或者synchronized方法中；

## 4 Lock& ReentrantLock

## 5 AQS