# 一、 java基础

## 1.如何在后台发送一个请求?

1. 通过java.net.URLConnection
2. 通过sun.net.www.http.HttpClient
3. 通过com.mashape.unirest.http.Unirest，开源的，设置超时时间有坑，每次setTimeOuts都会建立新的线程.

## 2. Arraylist和linkedlist的区别

1. ArrayList与LinkedList都是List接口的实现类
2. ArrayList是实现了基于动态数组的数据结构，LinkedList基于双链表的数据结构。
3. 对于随机访问get和set，ArrayList优于LinkedList，ArrayList使用索引的方式来快速定位对象的位置,而LinkedList要移动指针。
4. 对于新增和删除操作add和remove，LinekedList比较占优势，因为ArrayList要移动后段的数组元素，从而会重新调整索引顺序,调整索引顺序会消耗一定的时间，所以速度上就会比LinkedList要慢许多. 相反,LinkedList是使用链表实现的,若要从链表中删除或插入某一个对象,只需要改变前后对象的引用即可。LinekedList适合用来实现Stack(堆栈)与Queue(队列),前者先进后出，后者是先进先出.

## 3.hashmap源码，什么版本的，为什么快

1. 1.7的版本
2. Java 8中，如果hash相同的key的数量大于8，会使用平衡树代替链表；
3. 线程不安全的，key/value都可以使用null；
4. HashMap的数据结构是基于数组和单链表的。（以数组存储元素，如有hash相同的元素，在数组结构中，创建链表结构，再把hash相同的元素放到链表的下一个节点）
5. 每个table是entry[]，相同hash的元素放入同一个位置的链表中，先放的元素在尾部，后放入的元素在头部；
6. 初始容量是16，负载因子默认是0.75，当超过16\*0.75时，要扩容，默认扩容至原来的两倍，即32；
7. 如何get和put？第一步，看key是否为null，为null，直接取value，不为null，则计算key的hashCode；第二步，根据hashCode计算hash值；第三步，根据hash和length，计算所在数组的位置索引；第四步，逐个遍历链表上的元素，用equals来比较内容，put--内容相同，value直接覆盖，不同则放入链表，get--内容相同，取value；
8. key的类型最好是采用String或者Integer这样的引用类型，因为String是不可变的，也是final的，而且已经重写了equals()和hashCode()方法了，需要保证在放入和取出时，元素的hashCode不变，不同的对象有着不同的hashCode，碰撞的几率也会降低，提高性能；
9. 线程不安全的原因--在多个线程同时发现HashMap的大小过小时，都会尝试调整大小，会造成条件竞争。在调整大小的过程中，存储在链表中的元素的次序会反过来，因为移动到新的bucket位置的时候，HashMap并不会将元素放在链表的尾部，而是放在头部，这是为了避免尾部遍历(tail traversing)。如果条件竞争发生了，那么就死循环了。

## 4.Arraylist和vector的区别

1. ArrayList是最常用的List实现类，内部是通过数组实现的，它允许对元素进行快速随机访问。数组的缺点是每个元素之间不能有间隔，当数组大小不满足时需要增加存储能力，就要讲已经有数组的数据复制到新的存储空间中。当从ArrayList的中间位置插入或者删除元素时，需要对数组进行复制、移动、代价比较高。因此，它适合随机查找和遍历，不适合插入和删除。
2. Vector与ArrayList一样，也是通过数组实现的，不同的是它通过synchronized关键字支持线程的同步，即某一时刻只有一个线程能够写Vector，避免多线程同时写而引起的不一致性，但实现同步需要很高的花费，因此，访问它比访问ArrayList慢

## 5.hashmap和hashtable的区别

1. 最主要的区别在于Hashtable是线程安全，而HashMap则非线程安全。Hashtable的实现方法里面都添加了synchronized关键字来确保线程同步，因此相对而言HashMap性能会高一些，我们平时使用时若无特殊需求建议使用HashMap，在多线程环境下若使用HashMap需要使用Collections.synchronizedMap()方法来获取一个线程安全的集合（Collections.synchronizedMap()实现原理是Collections定义了一个SynchronizedMap的内部类，这个类实现了Map接口，在调用方法时使用synchronized来保证线程同步,当然了实际上操作的还是我们传入的HashMap实例，简单的说就是Collections.synchronizedMap()方法帮我们在操作HashMap时自动添加了synchronized来实现线程同步，类似的其它Collections.synchronizedXX方法也是类似原理。JDK5以上多线程环境下建议使用concurrenthashMap;
2. HashMap可以使用null作为key，不过建议还是尽量避免这样使用。HashMap以null作为key时，总是存储在table数组的第一个节点上。而Hashtable则不允许null作为key。
3. HashMap的初始容量为16，Hashtable初始容量为11，两者的填充因子默认都是0.75。
4. HashMap扩容时是当前容量翻倍即:capacity\*2，Hashtable扩容时是容量翻倍+1即:capacity\*2+1。
5. HashMap和Hashtable的底层实现都是数组+链表结构实现。
6. 两者计算hash的方式不同：hashTable是一次hash，直接用key的hashCode和length计算数组索引位置，hashMap则是二次hash，在hashCode的基础上二次hash，再计算索引值，散列程度更高
7. 在HashMap 中不能用get()方法来判断HashMap 中是否存在某个键，而应该用containsKey()方法来判断。Hashtable 的键值都不能 为null，所以可以用get()方法来判断是否含有某个键
8. HashMap的迭代器(Iterator)是fail-fast迭代器，而Hashtable的enumerator迭代器不是fail-fast的。所以当有其它线程改变了HashMap的结构（增加或者移除元素），将会抛出ConcurrentModificationException，但迭代器本身的remove()方法移除元素则不会抛出ConcurrentModificationException异常。但这并不是一个一定发生的行为，要看JVM。这条同样也是Enumeration和Iterator的区别。
9. HashMap不能保证随着时间的推移Map中的元素次序是不变的

## Io nio的区别

在此之前，先弄清几个概念：

1. 同步：使用同步IO时，用户进程自己处理IO读写。
2. 异步：使用异步IO时，用户进程将IO读写委托给内核处理，需要将数据缓冲区地址和大小传给内核，完成后内核通知Java处理（回调）。
3. 阻塞：使用阻塞IO时，Java调用会一直阻塞到读写完成才返回。
4. 非阻塞：使用非阻塞IO时，如果不能立马读写，Java调用会马上返回，当IO事件分发器通知可读写时在进行读写，不断循环直到读写完成。
5. BIO：同步并阻塞，服务器的实现模式是一个连接一个线程，这样的模式很明显的一个缺陷是：由于客户端连接数与服务器线程数成正比关系，可能造成不必要的线程开销，严重的还将导致服务器内存溢出。当然，这种情况可以通过线程池机制改善，但并不能从本质上消除这个弊端。
6. NIO：在JDK1.4以前，Java的IO模型一直是BIO，但从JDK1.4开始，JDK引入的新的IO模型NIO，它是同步非阻塞的。而服务器的实现模式是多个请求一个线程，即请求会注册到多路复用器Selector上，多路复用器轮询到连接有IO请求时才启动一个线程处理。
7. AIO：JDK1.7发布了NIO2.0，这就是真正意义上的异步非阻塞，服务器的实现模式为多个有效请求一个线程，客户端的IO请求都是由OS先完成再通知服务器应用去启动线程处理（回调）

## 7.concurrenthashmap的实现原理

1. ConcurrentHashMap所使用的锁分段技术，首先将数据分成一段一段的存储，然后给每一段数据配一把锁，当一个线程占用锁访问其中一个段数据的时候，其他段的数据也能被其他线程访问；
2. ConcurrentHashMap是由Segment数组结构和HashEntry数组结构组成。Segment是一种可重入锁ReentrantLock，在ConcurrentHashMap里扮演锁的角色，HashEntry则用于存储键值对数据。一个ConcurrentHashMap里包含一个Segment数组，Segment的结构和HashMap类似，是一种数组和链表结构， 一个Segment里包含一个HashEntry数组，每个HashEntry是一个链表结构的元素， 每个Segment守护者一个HashEntry数组里的元素,当对HashEntry数组的数据进行修改时，必须首先获得它对应的Segment锁。
3. ConcurrentHashMap使用多个子Hash表，也就是段(Segment)。每个Segment相当于一个子Hash表
4. get时，不加锁，先定位到segment然后在找到头结点进行读取操作。而value是volatile变量，所以可以保证在竞争条件时保证读取最新的值，如果读到的value是null，则可能正在修改，那么就调用ReadValueUnderLock函数，加锁保证读到的数据是正确的。
5. Put时会加锁，一律添加到hash链的头部。
6. Remove时也会加锁，由于next是final类型不可改变，所以必须把删除的节点之前的节点都复制一遍。
7. 8.hashMap、linkedHashMap、TreeMap排序方面的区别？
8. 三者均非不同步，线程不安全；
9. hashmap无序；
10. linkedHashMap按放入的顺序排序；
11. treeMap底层用tree的结构，默认按key的升序排序；

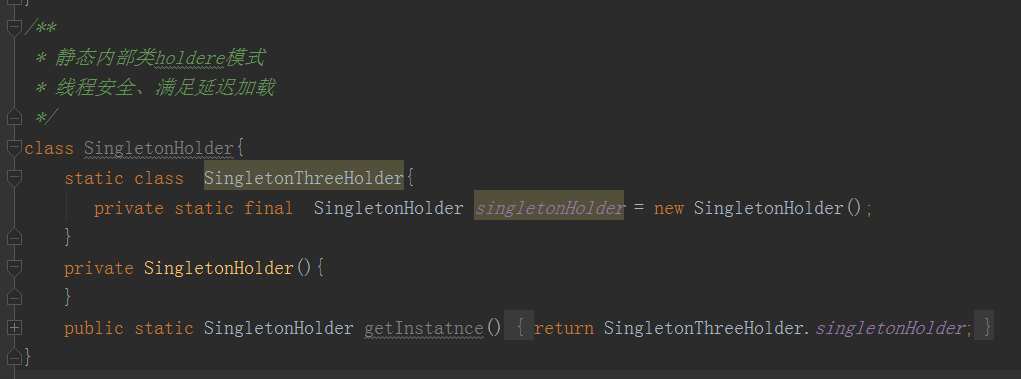
## 9.递归说一下，手写一个递归

1. 递归思想：1）递归就是方法里调用自身；2）在使用递归策略时，必须有一个明确的递归结束条件，称为递归出口；3）递归算法代码显得很简洁，但递归算法解题的运行效率较低。在递归调用的过程中系统为每一层的返回点、局部量等开辟了栈来存储。递归次数过多容易造成栈溢出等，所以一般不提倡用递归算法设计程序；
2. 比如10！
3. int fact(int n){
4. if(n==1){
5. return 1;
6. }else{
7. return fact(n-1)\*n;
8. }
9. }

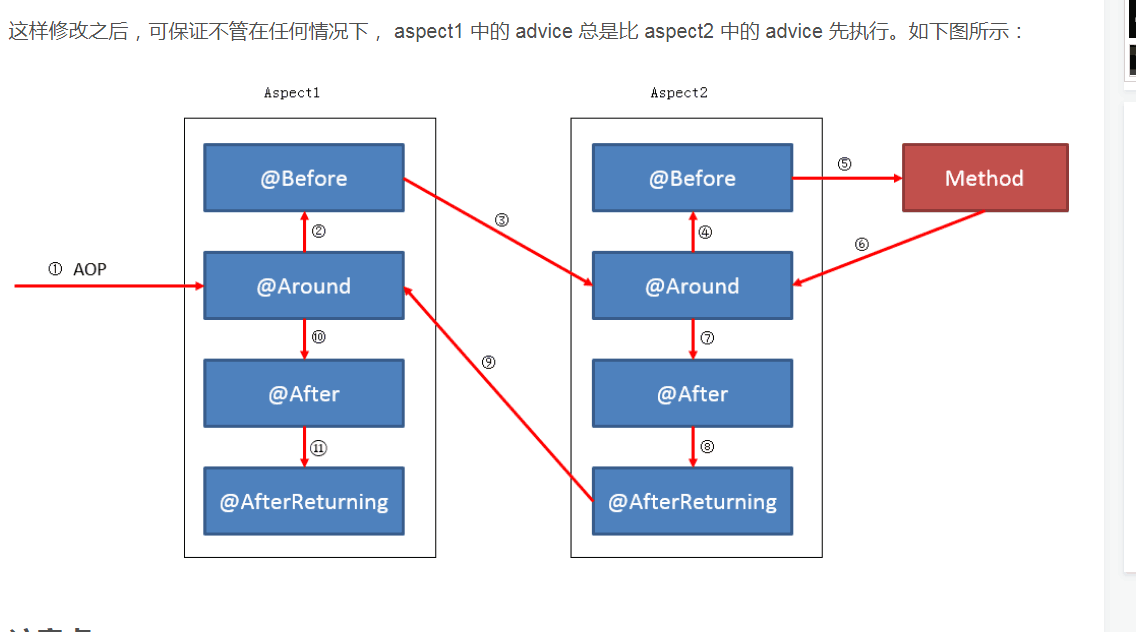
## 10.设计模式知道哪些，单例模式有哪些，区别，手写一个饿汉式，双检索说一下

1. Singleton（单例模式）--一个类在Java虚拟机中只有一个对象，并提供一个全局访问点，结构有懒汉模式和饿汉模式，常用的是配置文件、数据库连接对象等
2. /\*
3. \* 懒汉模式：在加载类的时候并不会生成唯一实例，只有在
4. \* 调用时候才会判断生成和返回实例，后续再调用实例的时候
5. \* 由于声明的对象不为空，直接返回原来创建的实例
6. \* 线程不安全
7. \*/
8. public class Singleton {
9. //私有化构造器，不能外部构造，只能通过get方法获取
10. private Singleton(){
11. }
12. //创建累的唯一实例
13. private static Singleton s;
14. //提供get方法获取实例
15. public static Singleton getS() {
16. //判断实例是否为空，
17. if(s==null){
18. s=new Singleton();
19. }
20. return s;
21. }
22. }
23. /\*
24. \*饿汉模式：对于只需要一个对象采用单例模式
25. \* 作用：保证实例有且只有一个
26. \*常用，容易造成垃圾对象
27. \*/
28. public class Singleton {
29. private Singleton(){
31. }
32. //这里创建一个类的唯一实例
33. private static final Singleton singleton=new Singleton();
34. //提供get方法来获取实例,外部调用几次都只会创建一个实例
35. public static Singleton getSingleton() {
36. return singleton;
37. }
39. }

* 饿汉的特点是加载类比较慢（由于加载类的时候需要加载累的对象，比较慢），但是获取对象的速度比较快，线程安全；
* 懒汉的特点是加载类比较快，但是获取对象的速度比较慢，线程不安全；
* 提供两种线程安全的模式，均是在懒汉模式上改造

1. public class SingleModuleSafe {
2. // 定义一个全局实例对象
3. private static SingleModuleSafe instance = null;
4. //私有构造器
5. private SingleModuleSafe() {
6. }
7. //方法
8. public static synchronized SingleModuleSafe getInstance() {
9. if (instance == null) {
10. instance = new SingleModuleSafe();
11. }
12. return instance;
13. }
14. }
15. /\*\*
16. \* 加载方式:懒加载
17. \* 线程安全:线程安全
18. \* 这种方式采用双锁机制，安全且在多线程情况下能保持高性能。
19. \*/
20. public class SingleModuleDoubleLock {
21. private volatile static SingleModuleDoubleLock singleton;
22. private SingleModuleDoubleLock() {
23. }
24. public static SingleModuleDoubleLock getSingleton() {
25. if (singleton == null) {
26. synchronized (SingleModuleDoubleLock.class) {
27. if (singleton == null) {
28. singleton = new SingleModuleDoubleLock();
29. }
30. }
31. }
32. return singleton;
33. }
34. }
35. 
36. Factory（简单的工厂模式）--用一个方法来代替new关键字，spring核心模式
37. Proxy（代理模式）--为其他对象提供一个代理，以控制对当前对象的访问，常用AOP
38. Strategy（策略模式）--做一件事情有很多种方法，结构是声明一个顶级接口，定义一个策略方法，具体的实例都要实现这个接口，比如付款接口
39. Template（模板模式）--父类定义流程，子类实现流程，定义一个抽象父类定义流程，或者常用方法和常量，子类继承父类，实现具体的细节方法

## 11.spring Aop

1. Spring AOP中的动态代理主要有两种方式，JDK动态代理和CGLIB动态代理。JDK动态代理通过反射来接收被代理的类，并且要求被代理的类必须实现一个接口。JDK动态代理的核心是InvocationHandler接口和Proxy类。如果目标类没有实现接口，那么Spring AOP会选择使用CGLIB来动态代理目标类。CGLIB（Code Generation Library），是一个代码生成的类库，可以在运行时动态的生成某个类的子类，注意，CGLIB是通过继承的方式做的动态代理，因此如果某个类被标记为final，那么它是无法使用CGLIB做动态代理的。
2. 组成部分切面类(@Aspect)、切入点(@Pointcut)、通知(@Before、@Around、@After、@AfterThrowing、@AfterReturning)
3. 多个切面的执行顺序
4. 

## 12.CAS与ABA问题

**描述**:变量num=1,线程1查询num=1,线程2查询num=1,线程2cas更新num=2,线程2查询num=2,线程2cas更新num=1,线程1cas更新num=3

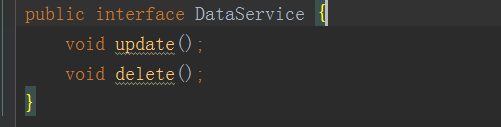
**问题**：对于线程1来说,其实num已经发生了变化,但是线程1并不知道

**解决办法**:增加版本号/时间戳

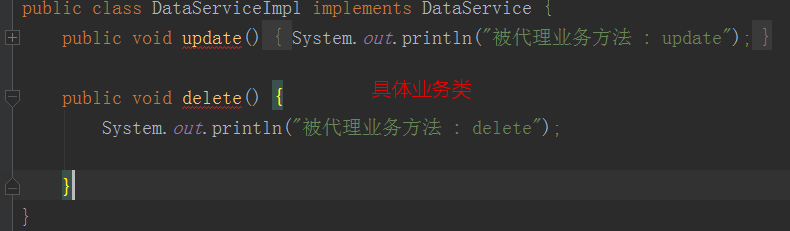
## 13动态代理

* Jdk动态代理

1.定义接口



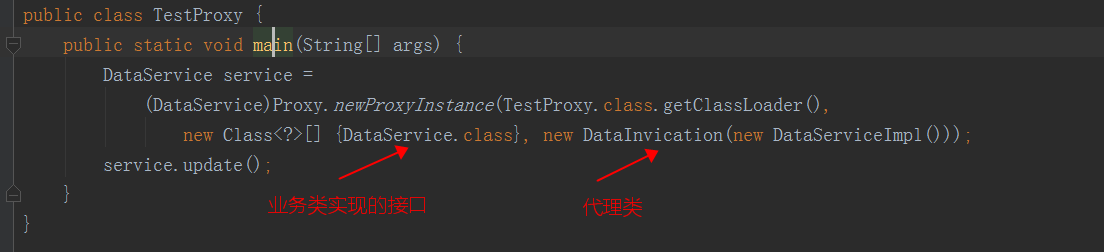
2.接口实现

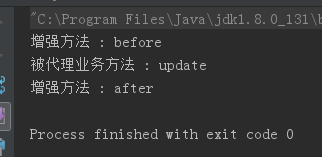


3.实现增强类



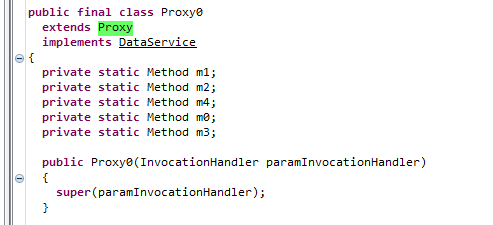
4.测试





原理：

反编译后的代理类

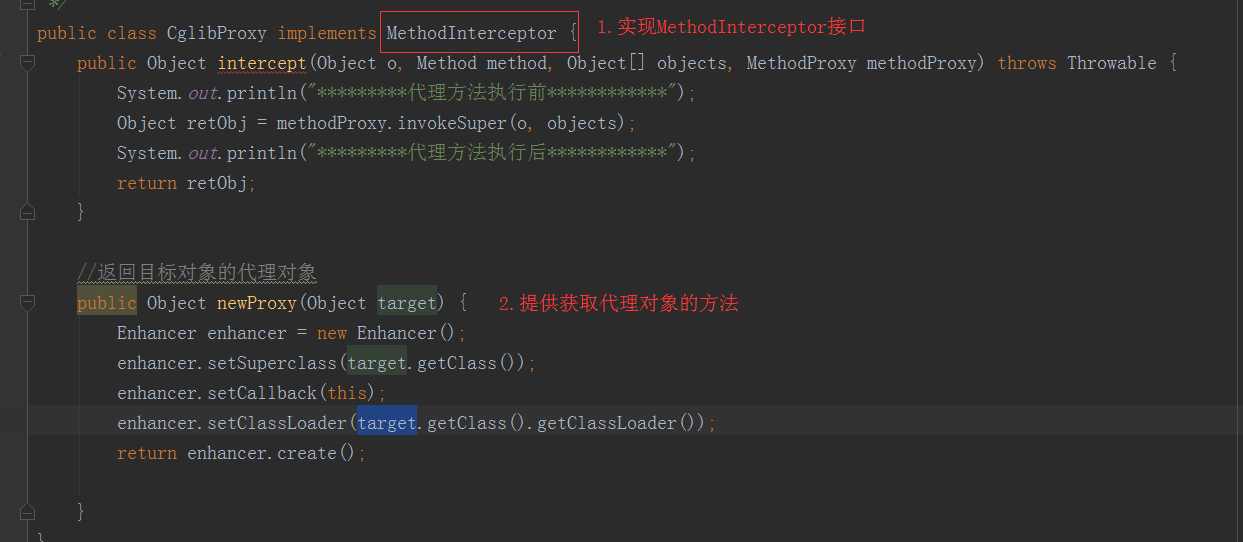


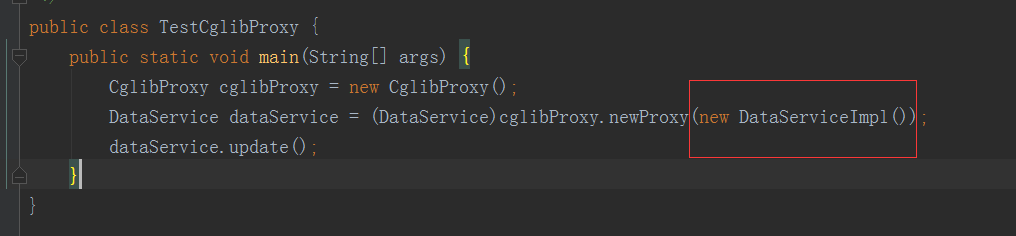


Proxy根据接口使用反射原理生成代理类,代理类继承Proxy对象,实现参数传入的接口数组。其中从Proxy继承到InvocationHandler属性引用的就是用户自己实现的增强类,当调用代理类实现的接口方法时，其实就是调用InvocationHandler的invoke方法。

* Cglib动态代理

原理：继承被代理类，生成代理类，通过重写被代理类方法，实现代理；





## 14.事务先关

* 事务的ACID特性

事务--就是平常所说的数据库事务，数据库执行的逻辑单位

ACID特性--原子性、一致性、隔离性、持久性

* 事务的隔离级别

如果不考虑事务的隔离性，会发生的几种问题

|  |  |
| --- | --- |
| 影响 | 描述 |
| 脏读 | 同一个事物内两次查询结果不一致,是由于能查询到其他事物的未提交数据导致的 |
| 不可重复读 | 同一个事物内两次查询结果不一致,是由于其他事物提交数据导致的(一条记录更新)； |
| 幻读 | 同一个事物内两次查询结果不一致,是由于其他事物插入了新的数据(记录数量); |

脏读、不可重复读区别:脏读是读取到未提交数据,不可重复读是读取到了又提交的数据;

幻读、不可重复读区别: 都是读取已提交数据,不可重复读针有点类似针对一条记录,幻读针对一批数据整体,比如个数；

隔离级别：

|  |  |
| --- | --- |
| Serializable (串行化) | 可避免脏读、不可重复读、幻读的发生 |
| Repeatable read (可重复读) | 可避免脏读、不可重复读的发生 |
| Read committed (读已提交) | 可避免脏读的发生 |
| Read uncommitted (读未提交) | 最低级别，任何情况都无法保证 |

* Spring事务的传播级别(即Transactional注解的propagation属性)

@Transactional失效的情况

|  |
| --- |
| 1.加在非public方法上; |
| 2.spring会对unchecked异常进行事务回滚；如果是checked异常则不回滚。 辣么什么是checked异常java里面将派生于Error或者RuntimeException（比如空指针，1/0）的异常称为unchecked异常，其他继承自java.lang.Exception得异常统称为Checked Exception，如IOException、TimeoutException等 |

传播级别：

|  |  |
| --- | --- |
| REQUIRED(默认的) | 业务方法需要在一个容器中运行,如果运行时方法已处在一个事务中，则加入进去，否则自己新建一个事务; |
| NOT\_SUPPORTED | 方法不需要事务,如果方法没有关联到一个事务,容器不会为他开启事务,如果方法在事务中被调用,则该事务会先被挂起,调用结束后,该事务恢复执行 |
| REQUIRESNEW | 不管是否已存在事务,总是为该方法创建一个新事物,如果方法已经运行在事务中，则原事务被挂起，新事物被创建; |
| MANDATORY | 该方法不需要事务执行，如果运行在事务中则报错; |
| SUPPORTS | 如果在事务内被调用则成为事务的一部分,如果在事务外被调用则该方法就在没有事务的环境下执行; |
| NEVER | 只有没有关联到事务才能正常执行 |
| NESTED | 如果一个活动的事务存在，则运行在一个嵌套的事务中。如果没有活动事务，则按REQUIRED属性执行。它使用了一个单独的事务，这个事务拥有多个可以回滚的保存点。内部事务的回滚不会对外部事务造成影响。它只对DataSourceTransactionManager事务管理器起效 |

# 二、 mysql数据库

## 数据库的引擎有哪些，优劣点及区别

## 现在用的引擎是什么，特点是什么，和哪些引擎存在竞争

## 现在的版本号，底层基于什么数据结构

## 索引有哪些，索引失效的情况，建立索引的注意事项

## 分库分表按照何种来拆分

## SQL语句如何优化

# 三、 框架

## ajax\jsp\javascript

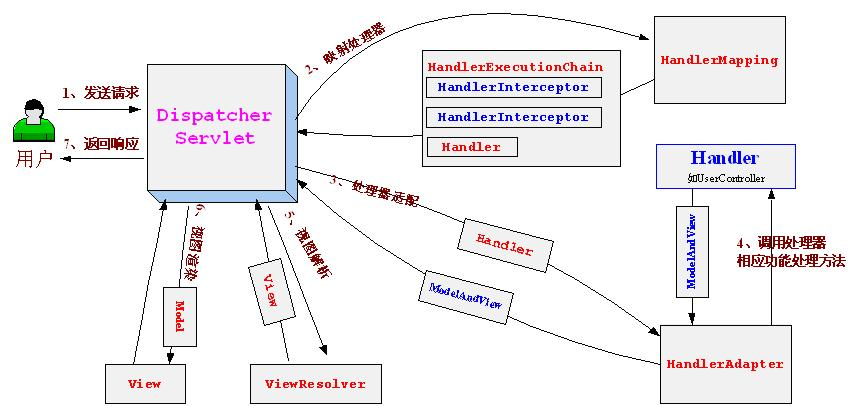
## tomcat配置

## 过滤器、拦截器怎么配置实现的，有什么区别？

1. 拦截器--实现HandlerInterceptor接口，springmvc-servlet.xml里面增加拦截器的配置
2. 拦截器--基于java反射机制，是AOP的一种应用，只能拦截controller
3. 过滤器--基于函数回调，可以对几乎所有请求进行过滤，只在容器初始化时调用一次
4. 过滤器--在web.xml注册过滤器

## Quartz的原理

## mvc框架说下？

1. 
2. 强制性的使应用程序输入、处理和输出分开。使用MVC应用程序被分成三个核心部件：模型、视图、控制器
3. 客户端请求提交到DispatcherServlet
4. 由DispatcherServlet控制器查询HandlerMapping，找到并分发到指定的Controller中。
5. Controller调用业务逻辑处理后，返回ModelAndView
6. DispatcherServlet查询一个或多个ViewResoler视图解析器，找到ModelAndView指定的视图
7. 视图负责将结果显示到客户端

## 与struts2对比，为什么mvc越来越受追捧？

1. SpringMVC--方法之间基本上独立的，独享request response数据，struts2--所有Action变量是共享的，一个Action对象对应一个request上下文
2. springMVC--分发器是servlet，集成了ajax，只要一个@ResponseBody即可；
3. struts2--分发器是filter，异步比较麻烦；
4. 校验方面--springMVC利用注解就可以了，struts2校验比较麻烦；

## get post的差异？

1. get把请求的数据放在url上，其格式为： 以?分割URL和传输数据，参数之间以&相连
2. post把数据放在HTTP的包体内（requrest body）
3. get提交的数据最大是2k,post理论上没有限制
4. GET产生一个TCP数据包，浏览器会把http header和data一并发送出去，服务器响应200(返回数据); POST产生两个TCP数据包，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok(返回数据)。
5. GET在浏览器回退时是无害的，POST会再次提交请求。
6. GET请求会被浏览器主动cache，而POST不会，除非手动设置。
7. GET请求只能进行url编码，而POST支持多种编码方式。
8. GET请求参数会被完整保留在浏览器历史记录里，而POST中的参数不会被保留。

## 后端要是我访问一个url要做哪些操作

## 你们项目选mybatis，它适用于什么场景，与hibernate的差异？

1. 项目若基本是增删改查等语句，没有复杂的查询，hibernate性能要比mybatis高
2. mysql需要自己手动写sql，hibernate有良好的映射机制；
3. 数据库移植性比较好，因已封装好sql，不像mysql手动的sql局限性较大
4. sql是直接对数据库表的操作，hql是对与表有映射关系的实体类的操作，底层会再自动生成相应的sql。

## 你对spring的理解

1. 一个开源的Java EE开发框架
2. 一个轻量级的控制反转（IoC）和面向切面（AOP）的容器框架。

## Aop的实现方式，优点及使用场景

1. 把公共的逻辑抽取出来，使得业务和系统服务拆分开；
2. 如果目标类实现接口，则使用JDK动态代理技术，如果目标对象没有实现接口，则默认会采用CGLIB代理
3. 日志记录、事务控制、权限控制、性能统计、异常处理及事务处理

## IOC的思想，优点

1. 负责实例化、定位、配置应用程序中的对象及建立这些对象间的依赖

## 什么是反射？

1. 在运行时根据指定的类名获得类的信息；
2. 包中的类总是需要变动，不能总是修改xxx=new xxx()，运用反射
3. Class.forName("className").newInstance();
4. new对象，无法调用该类里面私有的东西，反射反之;
5. new属于静态编译，而反射属于动态编译，意思就说只有到运行时他才会去获得该对象的实例；

## 你对soa和微服务的理解？

1. 微服务是SOA发展出来的产物，它是一种比较现代化的细粒度的SOA实现方式
2. 微服务是一个「富终端、瘦通信」，使用轻量级的通信机制，每个服务有自己的处理逻辑，它知道它要找的服务在哪里，不需要在通信的链路上做什么事情
3. 微服务相当于是SOA去掉了ESB

## Springcloud组件有哪些，组件的作用？

1. eureka、zuul、feign、ribbon、config、Hystrix

## Springcloud与dubbo的区别？

## Springboot的优势及弊端

1. 优势--继承了spring的特性，简化了spring的使用过程；
2. 快速启动容器，搭建spring框架；
3. 简化配置，几乎可以不用XML配置
4. 对主流开发框架的无配置集成，starter
5. 无需WAR包，直接可以用命令启动，java -jar

# 四、 中间件

## 1.redis

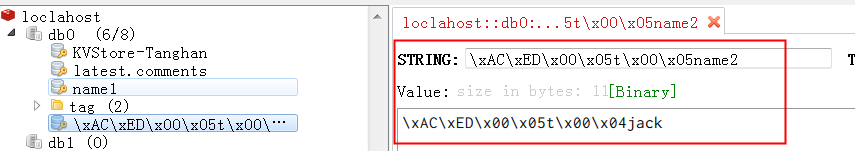
springBoot+redisTemplate只需在application.properties中配置spring.redis为前置的参数即可;RedisAutoConfiguration会自动实例化redisTemplate

### StringRedisTemplate/RedisTemplate的区别

两者使用的序列化类不同

StringRedisTemplate--->StringRedisSerializer(直接存入)

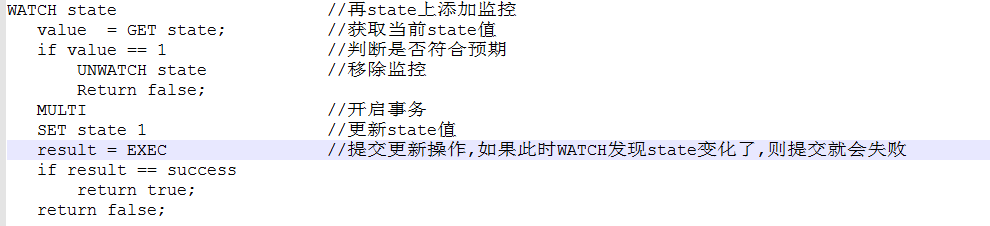
RedisTemplate--->JdkSerializationRedisSerializer(转化成字节数组存入)



### redis的数据类型？

|  |  |
| --- | --- |
| 类型 |  |
| String | 常规的get/set操作,key可以是 |
| Hash | 这里 Value 存放的是结构化的对象，比较方便的就是操作其中的某个字段。 |
| List | 使用 List 的数据结构，可以做简单的消息队列的功能。另外还有一个就是，可以利用 lrange 命令，做基于 Redis 的分页功能，性能极佳，用户体验好。 |
| Set | Set 堆放的是一堆不重复值的集合 |
| SortSet | Sorted Set多了一个权重参数 Score，集合中的元素能够按 Score 进行排列 |

### Redis的事务



### Redis跟memcache的区别在哪，优势在哪

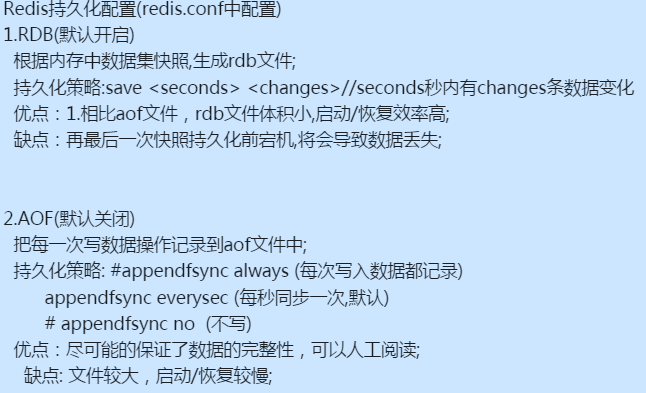
a.性能方面：Redis只使用单核，而Memcached可以使用多核，所以平均每一个核上Redis在存储小数据时比Memcached性能更高。而在100k以上的数据中，Memcached性能要高于Redis；

b.内存使用效率对比：使用简单的key-value存储的话，Memcached的内存利用率更高，而如果Redis采用hash结构来做key-value存储，由于其组合式的压缩，其内存利用率会高于Memcached

1. redis提供的数据类型丰富，支持数据的备份和数据的持久化

### 如何解决redis与数据库的一致性问题

### 如何实现redis的持久化？



### 7）如何做榜单

### 8)Redis的过期策略以及内存淘汰机制

* 3种过期策略

Redis采用的过期策略:惰性删除+定期删除

Memached采用惰性删除策略

|  |  |  |  |
| --- | --- | --- | --- |
| 过期策略 | 定义 | 优点 | 缺点 |
| 定时删除 | 设置key过期时间同时,为key创建一个定时器,定时器在过期时间来临时,删除key； | 及时释放了内存; | 同一时间有大量数据过期，去删除key会占用大量cpu资源,为每一个key创建一个定时器也非常影响性能,因此，这种策略基本没人用 |
| 惰性删除 | key过期不删除,每次访问key时检查是否过期,是则删除,返回null; | 删除只发生在访问key时，对cpu资源占用较少; | 很明显，不再访问这个key就不会删除,有大量类似数据有可能形成堆积; |
| 定期删除 | 每隔一段时间执行一次对过期key的删除操作; | 通过限制删除操作的时长、频率，减少对cpu的占用时间;  完善了惰性删除遗留的 堆积key的缺点; | 对内存释放不如定时删除,对cpu占用时间，不如惰性删除;  难点:不好把握时长和频率的设置 |

* 过期key对持久化的影响

|  |  |
| --- | --- |
| 持久化方式 | 过期key的影响(都无影响) |
| RDB | 1.持久化前，判断key是否过期,过期进入rdb文件;  2.rdb文件恢复到redis，会对rdb文件内的key进行过期检查,过期则不导入redis; |
| AOF | 1.key过期后,还没有删除,不会进入aof;key过期进行删除操作，会向aof文件追加一条del命令; |

* 内存淘汰机制

当内存不足以容纳新写入数据时怎么办？

Redis.conf中# maxmemory-policy volatile-lru

|  |  |
| --- | --- |
| 策略 | 定义 |
| **noeviction** | 新写入报错 |
| **allkeys-lru** | 移除最近最少使用的，推荐使用 |
| **allkeys-random** | 随机删除 |
| **volatile-lru** | 在设置了过期时间的key上，移除最近最少使用的 |
| **volatile-random** | 在设置了过期时间的key上，随机删除 |
| **volatile-ttl** | 在设置了过期时间的key上，有更早过期时间的 Key 优先移除 |

## 2.rabbitMQ

## 3.kafka简要说下，能解决哪些问题

## 4.zookeeper的主要用途

zookeeper是一个分布式的应用程序协调服务。

其实zookeeper只是提供功能

1. 分布式文件文件系统;
2. 节点的监听/通知
3. 再此基础上，用来实现注册服务（dubbo）、实现分布式锁

# 五、JVM

## jvm内存模型？

1. Java堆：所有线程共享的一块内存区域，对象实例几乎都在这分配内存
2. Java虚拟机栈：线程私有的，与线程生命周期相同，用于存储局部变量表，操作栈，方法返回值。局部变量表放着基本数据类型，还有对象的引用
3. 方法区：各个线程共享的区域，储存虚拟机加载的类信息，常量，静态变量，编译后的代码
4. 程序计数器：是一个数据结构，用于保存当前正常执行的程序的内存地址。Java虚拟机的多线程就是通过线程轮流切换并分配处理器时间来实现的，为了线程切换后能恢复到正确的位置，每条线程都需要一个独立的程序计数器，互不影响，该区域为“线程私有”。
5. 运行时常量池：代表运行时每个class文件中的常量表。包括几种常量：编译时的数字常量、方法或者域的引用。

## 为什么会有gc，gc有哪几种，简要的说下

1. 堆的内存结构



1. 那些对象会被gc

a.引用计数法,认为引用计数为0的对象都已死,可以清理;

实现简单、效率高、无法解决互引用问题;

b.可达性分析算法,通过一系列”gc roots"对象作为起点向下搜索,当一个对象不在搜索路径上时,就认为此对象不可达; gc roots包括:虚拟机栈中引用的对象、方法区中类静态属性引用的对象、方法区中常量引用的对象、本地方法栈中引用的对象;

1. GC所做的事情流程：
2. 在什么时候：
3. 1.新生代有一个Eden区和两个survivor区，首先将对象放入Eden区，如果空间不足就向其中的一个survivor区上放，如果仍然放不下就会引发一次发生在新生代的minor GC，将存活的对象放入另一个survivor区中，然后清空Eden和之前的那个survivor区的内存。在某次GC过程中，如果发现仍然又放不下的对象，就将这些对象放入老年代内存里去。
4. 2.大对象以及长期存活的对象直接进入老年区。
5. 3.当每次执行minor GC的时候应该对要晋升到老年代的对象进行分析，如果这些马上要到老年区的老年对象的大小超过了老年区的剩余大小，那么执行一次Full GC以尽可能地获得老年区的空间。
6. 对什么东西：从GC Roots搜索不到，而且经过一次标记清理之后仍没有复活的对象。
7. 做什么： 新生代：复制清理； 老年代：标记-清除和标记-压缩算法； 永久代：存放Java中的类和加载类的类加载器本身。
8. GC Roots都有哪些： 1. 虚拟机栈中的引用的对象 2. 方法区中静态属性引用的对象，常量引用的对象 3. 本地方法栈中JNI（即一般说的Native方法）引用的对象
9. 新生代--1）由 Eden 与 Survivor Space（S0，S1）构成，大小通过-Xmn参数指定，Eden 与 Survivor Space 的内存大小比例默认为8:1，可以通过-XX:SurvivorRatio 参数指定，比如新生代为10M 时，Eden分配8M，S0和S1各分配1M。2）对象在Eden中分配，当Eden没有足够空间时，会触发一次Minor GC；3）会将存活的对象移动到S0内存区域，并清空Eden区域，当再次发生Minor GC时，将Eden和S0中存活的对象移动到S1内存区域；4）当对象从Eden移动到Survivor或者在Survivor之间移动时，对象的GC年龄自动累加，当GC年龄超过默认阈值15时，会将该对象移动到老年代，可以通过参数-XX:MaxTenuringThreshold 对GC年龄的阈值进行设置；
10. 老年代--1）空间大小即-Xmx 与-Xmn 两个参数之差，2）用于存放经过几次Minor GC之后依旧存活的对象。3）当老年代的空间不足时，会触发Major GC/Full GC；
11. 永久代--JDK8中，类的元数据保存在本地内存中，元空间的最大可分配空间就是系统可用内存空间；

## 堆栈如何调整，分别是做什么的？

|  |  |
| --- | --- |
| -Xms | 初始堆大小。如：-Xms256m |
| -Xmx | 最大堆大小。如：-Xmx512m |
| -Xmn | 新生代大小。通常为 Xmx 的 1/3 或 1/4。新生代 = Eden + 2 个 Survivor 空间。实际可用空间为 = Eden + 1 个 Survivor，即 90% |
| -Xss | JDK1.5+ 每个线程堆栈大小为 1M，一般来说如果栈不是很深的话， 1M 是绝对够用了的。 |
| -XX:NewRatio | 新生代与老年代的比例，如 –XX:NewRatio=2，则新生代占整个堆空间的1/3，老年代占2/3 |
| -XX:SurvivorRatio | 新生代中 Eden 与 Survivor 的比值。默认值为 8。即 Eden 占新生代空间的 8/10，另外两个 Survivor 各占 1/10 |
| -XX:PermSize | 永久代(方法区)的初始大小 |
| -XX:MaxPermSize | 永久代(方法区)的最大值 |
| -XX:+PrintGCDetails | 打印 GC 信息 |
| -XX:+HeapDumpOnOutOfMemoryError | 让虚拟机在发生内存溢出时 Dump 出当前的内存堆转储快照，以便分析用 |

## 4.gc算法有哪些，说下各自的优劣点

1. 垃圾收集算法主要有：标记-清除、复制和标记-整理。
2. 1、标记-清除算法
3. 对待回收的对象进行标记。
4. 算法缺点：效率问题，标记和清除过程效率都很低；空间问题，收集之后会产生大量的内存碎片，不利于大对象的分配。
5. 2、复制算法
6. 复制算法将可用内存划分成大小相等的两块A和B，每次只使用其中一块，当A的内存用完了，就把存活的对象复制到B，并清空A的内存，不仅提高了标记的效率，因为只需要标记存活的对象，同时也避免了内存碎片的问题，代价是可用内存缩小为原来的一半。
7. 3、标记-整理算法
8. 在老年代中，对象存活率较高，复制算法的效率很低。在标记-整理算法中，标记出所有存活的对象，并移动到一端，然后直接清理边界以外的内存。

## 5.jvm编译类有哪几步，每步具体是干嘛的

## 6.jvm调优命令有哪些？

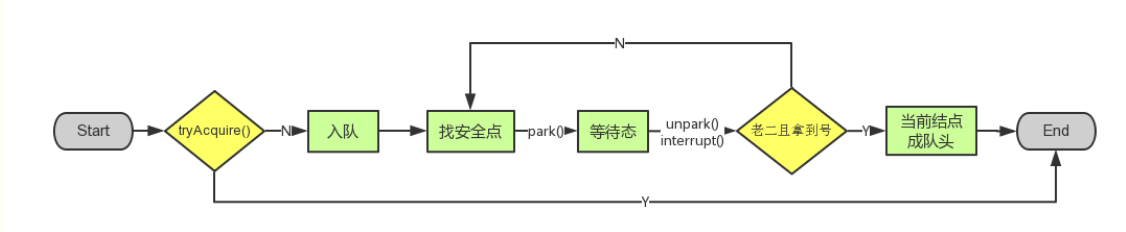
1. jps--显示指定系统内所有的HotSpot虚拟机进程
2. jstat--监视虚拟机运行时状态信息的命令，它可以显示出虚拟机进程中的类装载、内存、垃圾收集、JIT编译等运行数据。
3. jmap--用于生成heap dump文件,查询finalize执行队列、Java堆和永久代的详细信息，如当前使用率、当前使用的是哪种收集器等。
4. jhat--与jmap搭配使用，用来分析jmap生成的dump，jhat内置了一个微型的HTTP/HTML服务器，生成dump的分析结果后，可以在浏览器中查看。在此要注意，一般不会直接在服务器上进行分析，因为jhat是一个耗时并且耗费硬件资源的过程，一般把服务器生成的dump文件复制到本地或其他机器上进行分析
5. jstack--生成java虚拟机当前时刻的线程快照。线程快照是当前java虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等。 线程出现停顿的时候通过jstack来查看各个线程的调用堆栈，就可以知道没有响应的线程到底在后台做什么事情，或者等待什么资源。
6. jinfo--实时查看和调整虚拟机运行参数

# 六、多线程

## 1.conncurrent包下的常用类用过哪些，特点

1. 一些支持原子操作的数据类型,如:AtomicBoolean/AtomicInteger/AtomicLong
2. 一些锁,如ReentrantLock(可重入锁)/ReentrantReadWriteLock(读写锁)
3. ReentrantReadWriteLock--允许多个线程在同一时间对某特定资源进行读取，但同一时间内只能有一个线程对其进行写入
4. 控制并发的工具类,如：ThreadPoolExecutor(线程池)、CountDownLatch(阻塞，直到线程都结束) ？？？其他

## 2.多线程AQS框架原理



1. 有A、B、C三个线程，使用ReentrantLock加锁
2. 分别调用java.util.concurrent.locks.ReentrantLock.Sync#lock, ABC竞争锁,通过cas操作,只会有 一个线程加锁成功,假设为A;
3. 失败的BC线程调用java.util.concurrent.locks.AbstractQueuedSynchronizer#acquire方法
4. acquire方法干了啥
5. 再次尝试加锁;
6. 尝试加锁失败,构造当前线程对应node,放入等待队列;
7. 检查当前线程处于head节点后第一位置,再次尝试加锁,假设失败;
8. 检查前置线程状态是否是Node.SIGNAL,假设是;
9. 当前线程可以挂起休息，知道unpark()/interrupt();

## 3.线程是不是越多越好，为什么

1. 线程多了可以提高程序并行执行的速度，但是并不是越多越好
2. 每个线程都要占用内存，多线程就意味着更多的内存资源被占用
3. 从微观上讲，一个cpu不是同时执行两个线程的，他是轮流执行的，所以线程太多，cpu必须不断的在各个线程间快回更换执行，线程间的切换无意间消耗了许多时间，所以cpu有效利用率反而是下降的
4. 线程的数量 根据实际独立任务的数量作参考

## 4.volatile的实现思想

1. 适用于类的成员变量、类的静态成员变量
2. 保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。--会强制将对缓存的修改操作立即写入主存，写操作，它会导致其他CPU中对应的缓存行无效
3. 禁止进行指令重排序--1）当程序执行到volatile变量的读操作或者写操作时，在其前面的操作的更改肯定全部已经进行，且结果已经对后面的操作可见；在其后面的操作肯定还没有进行；2）在进行指令优化时，不能将在对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行。
4. 使用时需要保证原子性，这样才能发挥该关键字的作用，常用在布尔变量的状态标记作用和单例模式的双锁中；
5. 和Synchronized的区别：前者--实现可见性、有序性，不阻塞，针对变量；后者--实现可见性、有序性和原子性，线程阻塞；

## 5.lock与syn对比区别

1. Lock是一个接口，而synchronized是Java中的关键字，synchronized是内置的语言实现；
2. synchronized在发生异常时，会自动释放线程占有的锁，因此不会导致死锁现象发生；而Lock在发生异常时，如果没有主动通过unLock()去释放锁，则很可能造成死锁现象，因此使用Lock时需要在finally块中释放锁；
3. Lock可以让等待锁的线程响应中断，而synchronized却不行，使用synchronized时，等待的线程会一直等待下去，不能够响应中断；
4. 通过Lock可以知道有没有成功获取锁，而synchronized却无法办到。
5. Lock可以提高多个线程进行读操作的效率。
6. 在性能上来说，如果竞争资源不激烈，两者的性能是差不多的，而当竞争资源非常激烈时（即有大量线程同时竞争），此时Lock的性能要远远优于synchronized。所以说，在具体使用时要根据适当情况选择。

## 6.多线程异步接收值

见9

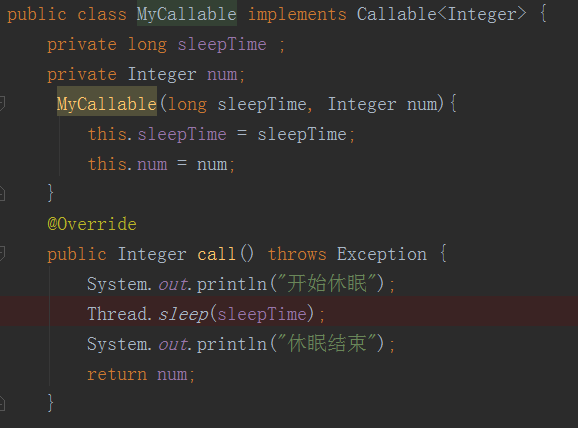
## 7.可重入锁

1. 可重入锁--如果锁具备可重入性，则称作为可重入锁。像synchronized和ReentrantLock都是可重入锁，可重入性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。
2. 可中断锁：可以相应中断的锁。在Java中，synchronized就不是可中断锁，而Lock是可中断锁。
3. 公平锁即尽量以请求锁的顺序来获取锁。比如同是有多个线程在等待一个锁，当这个锁被释放时，等待时间最久的线程（最先请求的线程）会获得该所，这种就是公平锁。
4. 非公平锁即无法保证锁的获取是按照请求锁的顺序进行的。这样就可能导致某个或者一些线程永远获取不到锁。在Java中，synchronized就是非公平锁，它无法保证等待的线程获取锁的顺序。而对于ReentrantLock和ReentrantReadWriteLock，它默认情况下是非公平锁，但是可以设置为公平锁。
5. ReadWriteLock就是读写锁，它是一个接口，ReentrantReadWriteLock实现了这个接口。可以通过readLock()获取读锁，通过writeLock()获取写锁。
6. Synchronized，它就是一个：非公平，悲观，独享，互斥，可重入的重量级锁
7. ReentrantLock，它是一个：默认非公平但可实现公平的，悲观，独享，互斥，可重入，重量级锁。
8. ReentrantReadWriteLocK，它是一个，默认非公平但可实现公平的，悲观，写独享，读共享，读写，可重入，重量级锁。

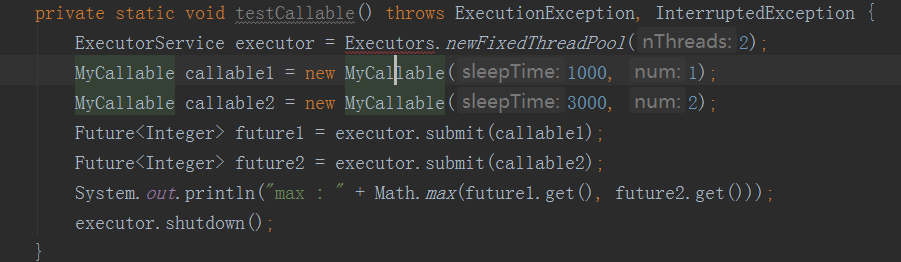
## 8.线程安全的队列

1. java.util.concurrent.ConcurrentLinkedQueue
2. 线程安全、无边界、链表形式、使用cas保证线程安全
3. Add()和offer()都是加入元素的方法（在ConcurrentLinkedQueue中，这两个方法投有任何区别）
4. Poll()和peek()都是取头元素节点，区别在于前者会删除元素，后者不会。

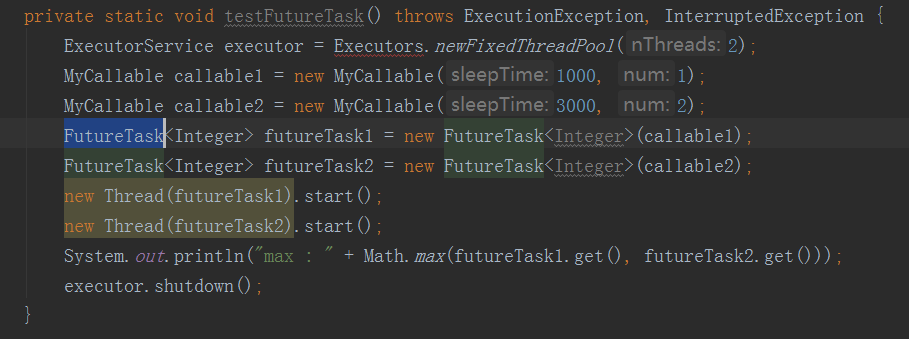
## 9.主线程如何在子线程执行完再执行下步操作



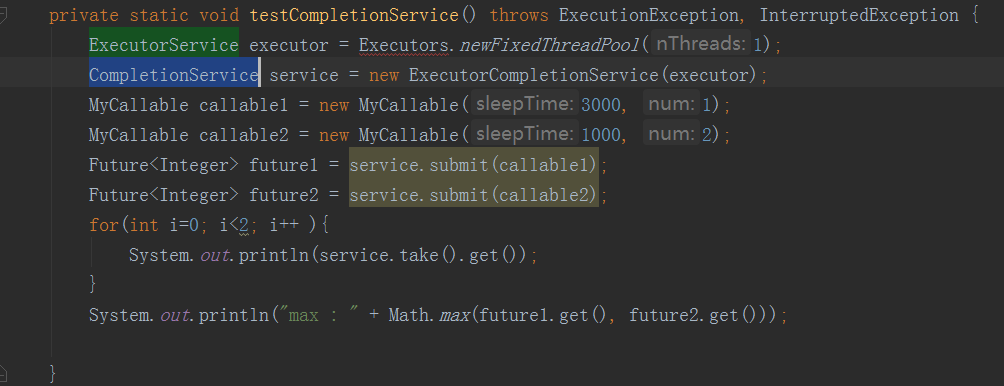
* + - 1. 使用ExecutorService + callable接口实现



* + - 1. 使用ExecutorService+FutureTask



* + - 1. 使用ExecutorService+CompletionService



1. 使用java.util.concurrent.CountDownLatch

主线程中声明CountDownLatch countDownLatch = new CountDownLatch(num);

子线程中调用一次countDownLatch.countDown(), 声明中的num-1

子线程启动后调用countDownLatch.await(); 主线程阻塞，直到num变成0

CountDownLatch也是基于AQS（抽象队列同步器）框架

1. 用sleep方法，让主线程睡眠一段时间
2. 使用Thread的join()等待所有的子线程执行完毕，主线程在执行，thread.join()把指定的线程加入到当前线程，可以将两个交替执行的线程合并为顺序执行的线程

# 七、分布式

## 1.为什么要提出base理论

1. CAP--一致性（C：Consistency）、可用性（A：Availability）和分区容错性（P：Partition tolerance），一致性是指数据在多个副本之间能否保持一致的特性，可用性是指系统提供的服务必须一直处于可用的状态，对于用户的每一个操作请求总是能够在有限的时间内返回结果，分区容错性是指遇到任何网络分区故障的时候，仍然需要能够保证对外提供满足一致性和可用性的服务；
2. 任何一个分布式系统都不可能同时满足CAP三者，但P必须是要满足的，所以要在C和A之间选择，目前大部分是选择A和P，最终一致即可。
3. BASE是Basically Available（基本可用）、Soft state（软状态）和Eventually consistent（最终一致性），在遇到网络故障的时候，可以损失部分可用性，但系统还是能用的，数据同步可以存在延迟，最终一致即可；

## 2.分布式锁有哪些实现，如何优化锁？

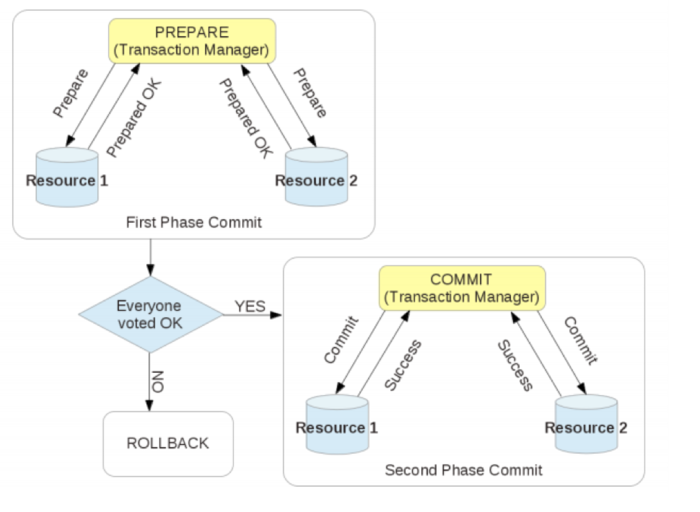
1. 目的--跨JVM的互斥机制来控制共享资源的访问
2. 分布式锁的满足条件：

* 在分布式系统环境下，一个方法在同一时间只能被一个机器的一个线程执行；
* 高可用的获取锁与释放锁；
* 高性能的获取锁与释放锁；
* 具备可重入特性；
* 具备锁失效机制，防止死锁；
* 具备非阻塞锁特性，即没有获取到锁将直接返回获取锁失败。

1. 三种实现方式：基于数据库实现、基于缓存（redis）、基于zookeeper实现
2. 基于数据库的方式优化思路：1）数据库自身的性能问题，最基本双机部署、数据同步、主备切换；2）需要记录获取锁的机器或者线程，不然再次获取和释放都会有问题；3）要有锁失效机制，以及定时去清理失效数据；
3. 基于redis的方式：目前比较好的方式，但redis集群不支持lua脚本，在释放锁的时候会有危险
4. 基于zookeeper的方式：需要频繁创建节点，删除节点，性能没有redis好

## 4.2pc说下，还有哪些方法可以解决分布式事务存在的问题

1. 基于二阶段提交的XA协议
2. 第一阶段：协调者询问所有参与者是否可以执行提交操作，参与者执行准备工作，例如为资源上锁，预留资源，写undo/redo log。
3. 第二阶段：若所有参与者回应“可提交”，则向所有参与者发送正式提交命令；若某个参与者回应“拒绝提交”，则向所有参与者发送回滚命令。
4. 缺点：XA协议保障了事务的强一致性，然而由于其采用的阻塞协议带来的巨大性能开销，难以达到较高的系统吞吐量



rocketmq+本地事物

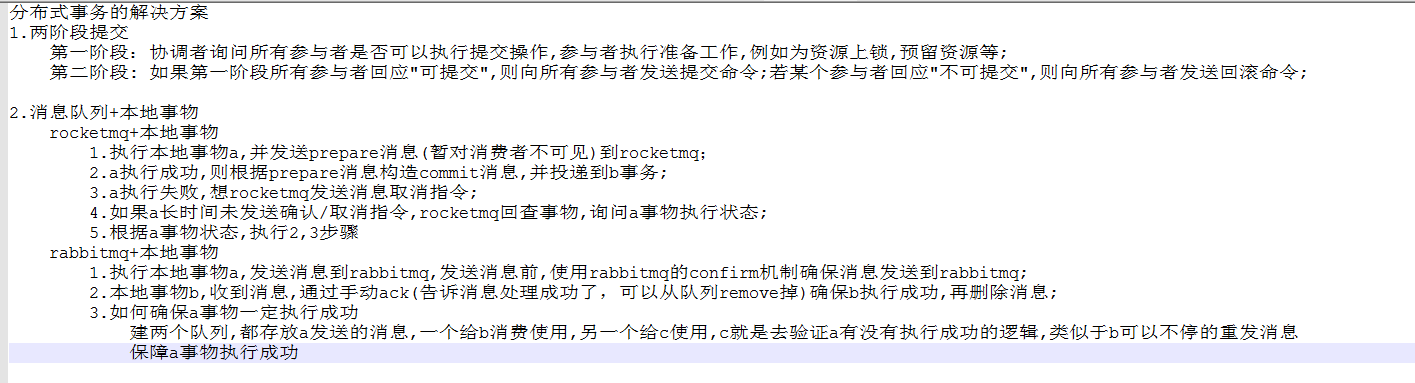
详情待补充

## 5.你对分布式的理解，与集群的差异

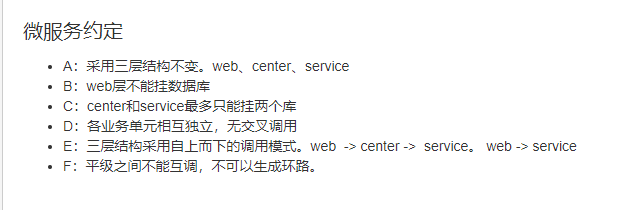
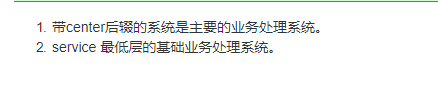
1. 集群是物理形态上的说法，同一个业务部署在多个机器上；
2. 分布式是一个业务分拆多个子业务，部署在不同的服务器上，节点互通，但又相互独立
3. 分布式系统，耦合性降低，单个系统独立开发、独立测试和独立部署，排错容易，开发效率高，扩展较容易，并且单个系统的可复用性较高；

## 分布式事务的解决方案

在分布式中运行的事务，数据库分布在不同的机器上，事务的参与者有多个，分布式事务是为了保证不同数据库中的一致性，事务在任何地方所做的所有动作，提交或回滚事务的决定必须产生统一的结果（全部提交或全部回滚）。



# 八.spring-cloud

1. 
2. 
3. center和service都是有业务处理的，center也是与数据库交互的，center不一定还要访问下层service,我们这边支付相关的所有信息都是在center处理，因为支付算是比较重要的功能，这个功能的代码，尽量不要拆分到多个模块，我目前解除到的代码里，service是用来处理一些比较基础的业务，就是web要处理一个界面查询啊，center去redis或者cache拿，没有，调用service,service用来提供一些基础的功能，一些简单的处理

## 如何拆分的设计思想？

1. 横向拆分。按照不同的业务域进行拆分，例如订单、营销、风控、积分资源等。形成独立的业务领域微服务集群。
2. 纵向拆分。把一个业务功能里的不同模块或者组件进行拆分。例如把公共组件拆分成独立的原子服务，下沉到底层，形成相对独立的原子服务层。这样一纵一横，就可以实现业务的服务化拆分。
3. 要做好微服务的分层：梳理和抽取核心应用、公共应用，作为独立的服务下沉到核心和公共能力层，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求
4. 服务拆分是越小越好吗？微服务的大与小是相对的。比如在初期，我们把交易拆分为一个微服务，但是随着业务量的增大，可能一个交易系统已经慢慢变得很大，并且并发流量也不小，为了支撑更多的交易量，我会把交易系统，拆分为订单服务、投标服务、转让服务等。因此微服务的拆分力度需与具体业务相结合，总的原则是服务内部高内聚，服务之间低耦合。

## 5.架构演化的步骤：

1. 先梳理平台的服务，对不同的服务进行分类，以确认演化的节奏。
2. 先让团队熟悉Spring Boot技术，并且优先在基础服务上进行技术改造，推动改动后的项目投产上线
3. 当团队熟悉Spring Boot之后，再推进使用Spring Cloud对原有的项目进行改造。
4. 在进行微服务改造过程中，优先应用于新业务系统，前期可以只是少量的项目进行了微服务化改造，随着大家对技术的熟悉度增加，可以加快加大微服务改造的范围
5. 传统项目和微服务项目共存是一个很常见的情况，除非公司业务有大的变化，不建议直接迁移核心项目

## 6.总结下项目的拆分思路

1. 项目的初衷：pass平台意在整合整个数据治理和展现中的各个环节，可以让不同分工的人员在同一套平台操作，统一管理；
2. 梳理各个环节（即现有的项目），先进行横向拆分，形成一个个的业务领域，比如调度、元数据管理、数据报表展现等，然后再进行纵向拆分，把不同项目之间共用的一些模块，拆分出来作为原子服务模块，比如用户模块、元数据模块，另外再抽取比较核心的模块，作为核心层，比如平台中的调度工具，最终把服务分为基础服务（与具体业务无关，如短信系统和邮件系统）、业务服务（单一的业务领域）和组合服务等，整个系统分为三层，web、center和service

## 7.常见面试题还需看下？

# 九、大数据

## Hadoop简要说一下

## Es现在用的地方很多包括阿里都在重点使用，他的优劣点在哪

# 十、linux基础

## 常用基础命令

1. cd，cp（复制），mv（重命名或者移动文件），rm，ps（进程），tar（解压缩），cat(查看内容)，chmod（权限操作），vim（编辑文件），find，ls

## 2.查看系统性能

## 3.配置环境

# 十一、其他小知识点

## [transient](http://1.transient)关键字

实现了序列化接口的对象的属性被transient关键字修饰,则该属性不会被序列化;