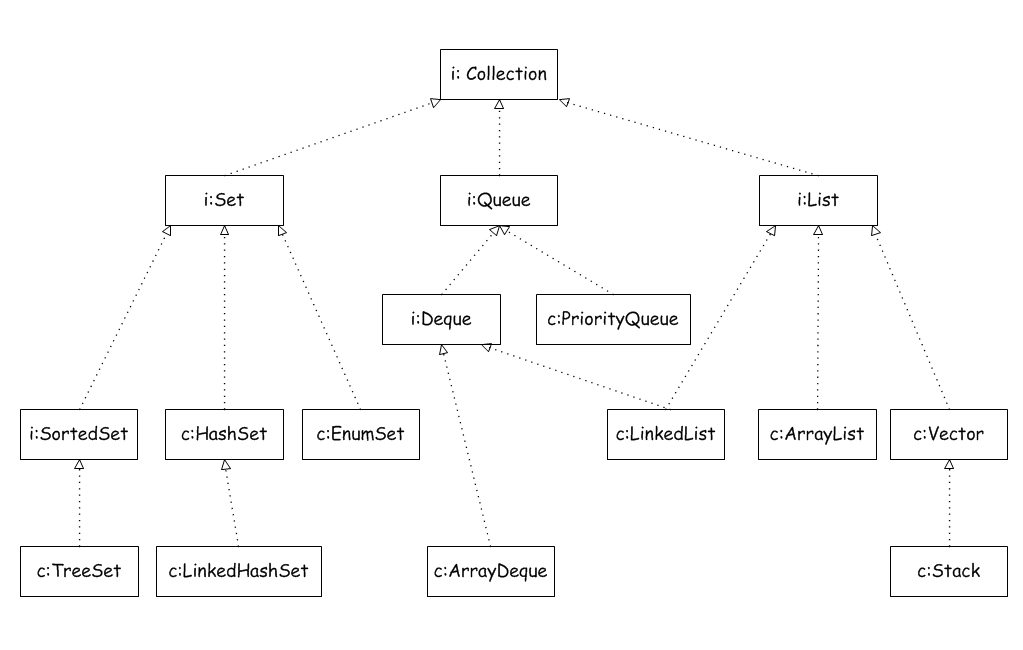
# Java基础

## 1 集合框架



### 1.1 Set接口

Set类继承了Conllection类：不允许出现重复数据；根据equals方法判断是否相同。

**(1) HashSet类**

**特征：**实现set接口：①不允许出现重复数据；②可以出现null；③数据是无序的。

**原理：**HashSet的底层是是一个hashmap；元素都存到HashMap键值对的Key上面，而Value时有一个统一的值private static final Object PRESENT = new Object(); 添加时：当新放入的key与集合中原有的key相同（hashCode()返回值相等，通过equals比较也返回true），则不添加；

**(2 ) LinkedHashSet**

**特征：**继承hashset，唯一不同是有序（插入的顺序）。

**原理：**构造方法调用LinkedHashMap，但它同时使用链表维护元素的次序。对集合迭代时，按增加顺序返回元素。添加性能略低于HashSet，但迭代访问元素时会有好性能。

(3) **SortedSet接口及TreeSet实现类**

**特征：**①不能写入null；②数据是有序的（不是插入的顺序，是按关键字大小排序的）。

**原理：**底层是使用TreeMap来包含Set集合中的所有元素。默认对元素进行默认自然升序排序（String）。自己定义的类必须实现Comparable接口，并且覆写相应的compareTo()函数，才可以正常使用。如果在比较的时候两个对象返回值为0，那么元素重复。

### 2 List接口

List 是一个元素有序的、可以重复、可以为 null 的集合，增加了与索引位置相关的操作。

**(1) ArrayList**

**特征：**①基于数组实现，长度是可变的；②查询速度快，增删较慢；③不同步

**原理：**ArrayList默认会生成一个大小为10的Object类型的数组。也可以指定数组的大小。如果加入元素后数组大小不够会先进行扩容，每次扩容都将数组大小增大一半。

**结论：**建议在单线程中才使用ArrayList，而在多线程中选择者CopyOnWriteArrayList。

**(2) LinkedList**

**特征**：①基于双端链表实现。②增删速度快，查询较慢；③不同步

区别：①ArrayList基于动态数组实现，LinkedList基于双端链表实现；②查询，ArrayList更优，因为LinkedList要移动指针；③增删，LinedList更优，因为ArrayList要移动数据。

**(3) Vector**

**特征：**①基于数组实现的；②线程安全的。③性能会比ArrayList低。

**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。

### 3 Map接口

**(1) HashMap**

**(2) LinkedHashMap**

**特征：**①继承HashMap；②允许key为null，value为null；③有序。④插入删除慢，

**原理：**LinkedHashMap可以认为是HashMap+LinkedList。通过维护一个双向链表，维护元素迭代的顺序。该迭代顺序可以是插入顺序或者是访问顺序（构造函数可以传入一个参数accessOrder：false按照插入的顺序排列；true按照访问的顺序LRU）。

构造方法中，实际调用了父类HashMap的相关构造方法来构造一个底层存放的table数组。LinkedHashMap重新定义了**Entry**数据结构，在hashmap的Entry基础上，新增了before，和after。(next是用于维护HashMap指定table位置上连接的Entry的顺序的) before、After是用于维护Entry插入的先后顺序的。

**(3) TreeMap**

TreeMap是SortedMap接口基于红黑树的实现，默认按照升序排列关键字。HashMap是根据键的HashCode 值存储数据，取得数据的顺序是完全随机的，HashMap取值的速度更快。

**(4) Hashtable**

**特征：**①数据结构为哈希表，②同步的，③不允许null作为键和值。

**原理：**初始大小为11，负载因子为0.75的Entry数组。而Entry实际上就是一个单向链表。当添加元素的时候，首先计算key的hash值，然后通过hash值确定在table数组中的索引位置，最后将value值替换或者插入新的元素，如果容器的数量达到阈值，就会进行扩容。

## 2 hashcode、equals

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。

## 3 final、finally 、finalize

**(1) final：**①被final修饰的类，就意味着不能再派生出新的子类，不能作为父类被继承。因此一个类不能被声明为abstract，又被声明为final。②将变量或方法声明为final。可以保证他们在使用的时候不被改变。被声明为final的变量必须在声明时给出变量的初始值，而在以后的引用中只能读取。③被声明为final的方法也只能使用，不能重写。

**(2) finally：**在异常处理时提供finally块来执行任何清除操作。无论异常是否发生，finally块都会被执行。

**(3) finalize：**finalize是方法名。它是在Object类中定义的，因此，所有的类都继承了它。finalize()方法是在垃圾收集器删除对象之前对这个对象调用的。

## 4 异常分类以及处理机制

Throwable：分为Error和Exception。

(1) Error：当程序发生不可控的错误时，通常做法是通知用户并中止程序的执行。

(2) Exception:一般分为Checked异常和Runtime异常。

**编译异常：**没有处理Checked异常，该程序在编译时就会发生错误无法编译。处理方法：

①当前方法知道如何处理该异常，则用try...catch块来处理该异常。

②当前方法不知道如何处理，则在定义该方法声明时使用throws抛出该异常。

**常见异常：**EOFException、FileNotFoundException、IOException

**运行时异常：**ArrayIndexOutOfBoundsException、ArithmeticException、NullPointerException、ClassNotFoundException、IllegalArgumentException。

## 5 重载和重写的区别

**(1) 重载 overloading**

①方法重载是让类以统一的方式处理不同类型数据的一种手段。多个同名函数同时存在，具有不同的参数个数/类型。重载是一个类中多态性的一种表现。

②重载的时候，方法名一样，但是参数类型和个数不一样，返回值类型可以相同也可以不相同。无法以返回型别作为重载函数的区分标准。

**(2) 重写Override**

①参数列表相同。②返回的类型相同。

③子类方法大于等于重写方法的访问权限。

④子类方法不能抛出比父类方法更多的异常。

## 6 抽象类和接口有什么区别

抽象类和接口都不能被直接实例化。接口和抽象类的概念不一样。抽象类表示的是，这个对象是什么。接口表示的是，这个对象能做什么。

①接口中所有的方法都是抽象的。而抽象类可以进行声明也可以对方法进行实现。

②子类使用extends关键字来继承，如果子类不是抽象类，需要重写所有抽象方法；接口使用implements来实现，需要重写所有方法。

③接口可以多继承，抽象类只能继承一个类。

④抽象方法不能是静态的static，接口方法可以是静态的；

## 7 static关键字什么情况下使用

①用来修饰成员变量，将其变为类的成员，从而实现所有对象对于该成员的共享；

②用来修饰成员方法，将其变为类方法，可以直接使用“类名.方法名”的方式调用，常用于工具类；

③静态块用法，将多个类成员放在一起初始化，使得程序更加规整，其中理解对象的初始化过程非常关键；

④静态导包用法，将类的方法直接导入到当前类中，从而直接使用“方法名”即可调用类方法，更加方便。

## 8 GET 与 POST的区别

①GET参数通过URL传递，POST放在request body中。

②GET请求在URL中传递的参数是有长度限制的，最大是2k，而POST理论上没有限制。

③GET产生一个TCP数据包；POST产生两个TCP数据包。GET方式的请求，浏览器会把http header和data一并发送出去，服务器响应200(返回数据); 对于POST，浏览器先发送header，服务器响应100 continue，浏览器再发送data，服务器响应200 ok（返回数据）。

④GET比POST更不安全，因为参数直接暴露在URL中，所以不能用来传递敏感信息。

⑤GET请求只能进行URL编码，而POST支持多种编码方式。

## 9 session 与 cookie 区别

**(1) 区别**

①Cookie和Session都是会话技术，Cookie是运行在客户端，Session是运行在服务器端。

②Cookie有大小限制以及浏览器在存cookie的个数也有限制，Session是没有大小限制和服务器的内存大小有关。

③Cookie有安全隐患，通过拦截或本地文件找得到你的cookie后可以进行攻击。

④Session会在一定时间内保存在服务器上。当访问增多，会比较占用你服务器的性能。

(2) 注意：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。

## 10 session 分布式处理

(1) 分布式Session的几种实现方式

**第一种：粘性session**

**原理：**粘性Session是指将用户锁定到某一个服务器上，比说说，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，那么用户以后的每次请求都会转发到A服务器上，相当于把用户和A服务器粘到了一块，这就是粘性Session机制。

**优点：**简单，不需要对session做任何处理。

**缺点：**缺乏容错性，如果当前服务器发生故障，用户被转移到其他服务器时，session信息都将失效。

**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。

**第二种：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，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：网络负荷造成压力，如果session量大可能会造成网络堵塞，拖慢服务器性能。

实现方式：设置tomcat，server.xml 开启tomcat集群功能；在web.xml中添加选项

<distributable/>支持分布式。

**第三种：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memcached。多个Memcached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复制放到它对应的memcahed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memcached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：memcached做主从复制，写入session都往从memcached服务上写，读取都从主memcached读取，tomcat本身不存储session。

优点：可容错，session实时响应。

**第四种：session持久化到数据库**

原理：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，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。

第五种：利用cookie记录session

## 11 equals 与 == 的区别

① == 是运算符，equals()是Object类中方法；

②对于基本数据类型：== 可以用来对比值是否相等，equals不能。

③对于引用数据类型：​== 和equals 都比较的是对象的地址。如果不重写equals方法，底层调用的就是==；也可以像String类一样重写equals方法，用来对字符串的内容进行比较。

## 12 Int和integer的区别

注意：给一个Integer对象赋一个int值的时候，会调用Integer类的静态方法valueOf：会缓存-128到127，即整型字面量的值在-128到127之间，将不会new新的Integer对象，而是直接引用常量池中的Integer对象。

## 13 &和&&的区别？

&运算符有两种用法：(1)按位与；(2)逻辑与。&&运算符是短路与运算。如果&&左边的表达式的值是false，右边的表达式不会进行运算。

## 14 switch 类型

在Java 5以前，switch(expr)中，expr只能是byte、short、char、int。从Java 5开始，引入了枚举类型；从Java 7开始，引入字符串（String），但是长整型（long）在目前所有的版本中都是不可以的。

## 15 请用至少四种写法写单例模式

|  |
| --- |
| //1、饿汉模式  public class Singleton {  private static Singleton instance = new Singleton();  private Singleton (){}  public static Singleton getInstance(){  return instance;  }  } |
| // 2、**双重校验锁**  public class Singleton {  private volatile static Singleton instance;  private Singleton (){}  public static Singleton getInstance (){  if (instance == null) {  synchronized (Singleton.class) {  if (instance == null) {  instance = new Singleton ();  }  }  }  return singletonDemo7;  }  } |
| //3、静态内部类  public class Singleton {  private static class SingletonHolder{  private static final Singleton INSTANCE = new Singleton ();  }  private Singleton (){}  public static final Singleton getInsatance(){  return SingletonHolder.INSTANCE;  }  } |
| //4、**枚举**  public enum Singleton {  INSTANCE;  } |

**双重原因:**如果线程一执行到第二个if判断时,切换到线程二,这时instance仍是null,则会出现两个instance.

**volatile原因：**instance = new Singleton();分解为三步：(1)分配对象的内存空间;(2)初始化对象;(3)设置instance指向刚分配的内存地址；2、3会重排，导致获取到未初始化的非null对象。volatile屏蔽指令重排序的语义在JDK1.5才被完全恢复。

## 16 类的实例化顺序

有父类的情况

(1) 加载父类：①静态属性；②静态初始化块（从上至下）

(2) 加载子类：①静态属性；②静态初始化块（从上至下）

(3) 加载父类构造器：①成员变量；②实例初始化块；③ 构造器内容

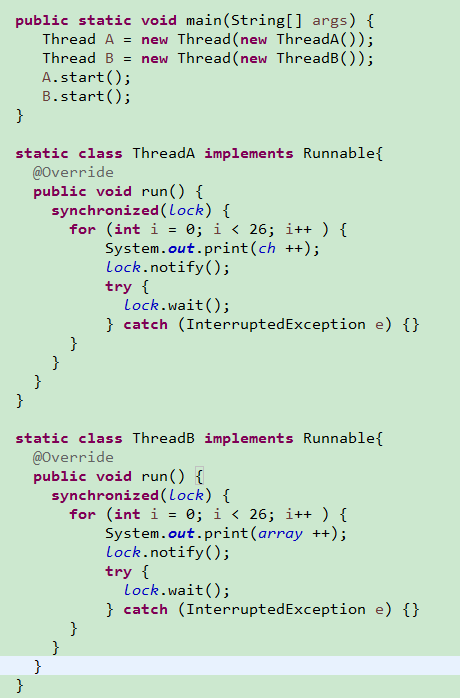
(4) 加载子类构造器：①成员变量；②实例初始化块；③ 构造器内容

## 17 生产者消费者

|  |
| --- |
| //1、方式一：synchronized、wait和notify  public class ProducerConsumerWithWaitNofity {  public static void main(String[] args) {  Resource resource = new Resource();  ProducerThread p1 = new ProducerThread(resource); //生产者线程  ProducerThread p2 = new ProducerThread(resource); //生产者线程  ConsumerThread c1 = new ConsumerThread(resource); //消费者线程  p1.start();  p2.start();  c1.start();  }  }  class Resource { //公共资源类  private int num = 0; //当前资源数量  private int size = 10; //资源池中允许存放的资源数目  public synchronized void remove() { //从资源池中取走资源  if (num > 0) {  num--;  System.out.println("消费者" + Thread.currentThread().getName() + "消耗一件资源，当前线程池  有" + num + "个");  notifyAll(); //通知生产者生产资源  } else {  try {  wait(); //如果没有资源，则消费者进入等待状态  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  public synchronized void add() { //向资源池中添加资源  if (num < size) {  num++;  System.out.println("生产者" + Thread.currentThread().getName() + "生产一件资源，  当前资源池有" + num + "个");  notifyAll(); //通知等待的消费者  } else {  try {  wait();//如果当前资源池中有10件资源，生产者进入等待状态  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  }  class ConsumerThread extends Thread { //消费者线程  private Resource resource;  public ConsumerThread(Resource resource) {  this.resource = resource;  }  @Override  public void run() {  while (true) {  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  resource.remove();  }  }  }  class ProducerThread extends Thread { //生产者线程  private Resource resource;  public ProducerThread(Resource resource) {  this.resource = resource;  }  @Override  public void run() {  while (true) { //不断地生产资源  try {  Thread.sleep(1000);  } catch (InterruptedException e) {  e.printStackTrace();  }  resource.add();  }  }  } |
| // 2、使用阻塞队列BlockingQueue解决生产者消费者  public class BlockingQueueConsumerProducer {  public static void main(String[] args) {  Resource3 resource = new Resource3();  ProducerThread3 p = new ProducerThread3(resource); //生产者线程  ConsumerThread3 c1 = new ConsumerThread3(resource); //多个消费者  ConsumerThread3 c2 = new ConsumerThread3(resource);  ConsumerThread3 c3 = new ConsumerThread3(resource);  p.start();  c1.start();  c2.start();  c3.start();  }  }  class ConsumerThread3 extends Thread { //消费者线程  private Resource3 resource3;  public ConsumerThread3(Resource3 resource) {  this.resource3 = resource;  }  public void run() {  while (true) {  try {  Thread.sleep((long) (1000 \* Math.random()));  } catch (InterruptedException e) {  e.printStackTrace();  }  resource3.remove();  }  }  }  //生产者线程  class ProducerThread3 extends Thread {  private Resource3 resource3;  public ProducerThread3(Resource3 resource) {  this.resource3 = resource;  }  public void run() {  while (true) {  try {  Thread.sleep((long) (1000 \* Math.random()));  } catch (InterruptedException e) {  e.printStackTrace();  }  resource3.add();  }  }  }  class Resource3 {  private BlockingQueue resourceQueue = new LinkedBlockingQueue(10);  public void add() { // 向资源池中添加资源  try {  resourceQueue.put(1);  System.out.println("生产者" + Thread.currentThread().getName()  + "生产一件资源," + "当前资源池有" + resourceQueue.size() + "个资源");  } catch (InterruptedException e) {  e.printStackTrace();  }  }  public void remove() { //向资源池中移除资源  try {  resourceQueue.take();  System.out.println("消费者" + Thread.currentThread().getName() +  "消耗一件资源," + "当前资源池有" + resourceQueue.size() + "个资源");  } catch (InterruptedException e) {  e.printStackTrace();  }  }  } |
| //3 使用Condition  public class Mutil\_Producer\_ConsumerByCondition {  public static void main(String[] args) {  Resource r = new Resource();  Mutil\_Producer pro = new Mutil\_Producer(r);  Mutil\_Consumer con = new Mutil\_Consumer(r);  Thread t0 = new Thread(pro); //生产者线程  Thread t1 = new Thread(pro);  Thread t2 = new Thread(con); //消费者线程  Thread t3 = new Thread(con);  t0.start(); //启动线程  t1.start();  t2.start();  t3.start();  }  }  class Resource {  private String name;  private int count = 1;  private boolean flag = false;  **Lock lock = new ReentrantLock();//创建一个锁对象**  //通过已有的锁获取两组监视器，一组监视生产者，一组监视消费者。  **Condition producer\_con = lock.newCondition();**  **Condition consumer\_con = lock.newCondition();**  public void product(String name) { //生产  lock.lock();  try {  while (flag) {  try {  producer\_con.await();  } catch (InterruptedException e) {  }  }  this.name = name + count;  count++;  System.out.println(Thread.currentThread().getName() + "...生产者..." + this.name);  flag = true;  consumer\_con.signal();//直接唤醒消费线程  } finally {  lock.unlock();  }  }  public void consume() { //消费  lock.lock();  try {  while (!flag) {  try {  consumer\_con.await();  } catch (InterruptedException e) {  }  }  System.out.println(Thread.currentThread().getName() + "...消费者......." + this.name);  flag = false;  producer\_con.signal();//直接唤醒生产线程  } finally {  lock.unlock();  }  }  }  class Mutil\_Producer implements Runnable { //生产者线程  private Resource r;  Mutil\_Producer(Resource r) {  this.r = r;  }  public void run() {  for (int i = 0; i < 10; i++) {  r.product("北京烤鸭");  }  }  }  //消费者线程  class Mutil\_Consumer implements Runnable {  private Resource r;  Mutil\_Consumer(Resource r) {  this.r = r;  }  public void run() {  for (int i = 0; i < 10; i++) {  r.consume();  }  }  } |

|  |
| --- |
| public void product(String name) {  lock.lock();  try{  while(flag){  try{  producer\_con.await();  }catch(InterruptedException e){}  }  this.name = name + count;  count++;  System.out.println(Thread.currentThread().getName()+"...生产者5.0..."+this.name);  flag = true;  consumer\_con.signal();//直接唤醒消费线程  }finally{  lock.unlock();  }  } |
| public synchronized void product(String name){  while(flag){  //此时有烤鸭，等待  try {  this.wait();  } catch (InterruptedException e) {}  }  this.name=name+count;//设置烤鸭的名称  count++;  System.out.println(Thread.currentThread().getName()+"...生产者..."+this.name);  flag=true;//有烤鸭后改变标志  notifyAll();//通知消费线程可以消费了  } |

## 18 线程交替打印例子



## 18 设计模式

单例模式，**适配器模式**，工厂模式，**装饰模式**，模板方法模式，**代理模式**，责任链模式

## 19 拦截器怎么实现，拦截器和过滤器有什么区别

## 20 spring新功能知道哪些

## 21 JAVA1.8版本增了哪些新功能

## 22 你们的应用用的服务器是哪种？

Tomcat是Apache 软件基金会的一个核心项目，由Apache、Sun 和其他一些公司及个人共同开发而成。因为Tomcat 技术先进、性能稳定，而且免费，成为目前比较流行的Web 应用服务器。目前最新版本是9.0。

Tomcat运行时**占用的系统资源小，扩展性好，支持负载平衡与邮件服务**等开发应用系统常用的功能；支持Servlet和JSP等特性。

Resin是CAUCHO公司的产品，是一个非常流行的application server，对servlet和JSP提供了良好的支持，性能也比较优良，resin自身采用JAVA语言开发。支持负载均衡。

1.先从是否收费上进行比较：

Resin作为个人开发是免费的，如果需要将开发产品作为商业产品发布是需要收费的。

Tomcat是开源的，免费。

2.Resin 在一台机器上配置多个运行实例时，稍显麻烦，不像Tomcat复制多份，修改个端口即可，完全独立。

二者在稳定性上都没有任何问题、性能在访问量不大的话，都没有多大的差别。目前选择Tomcat的较多。

## 23 QPS计算

问题：每天80%的访问集中在20%的时间，每天有300万的pv，而单台机器的qps为50，需要几台机器？

原理：( 总PV数 \* 80% ) / ( 每天秒数 \* 20% ) = 峰值时间每秒请求数(QPS)

峰值时间每秒QPS / 单台机器的QPS = 需要的机器

解答：这台机器需要多少QPS？ ( 3000000 \* 0.8 ) / (86400 \* 0.2 ) = 139 (QPS)

需要几台机器来支持？139 / 50 = 3

## 24 linux 常用命令

find /path -name '\*string\*' 查找文件名包含某个字符串的的文件

grep "被查找的字符串" 文件名 查询文件内容

-n 显示行数

-i 忽略大小写

-c 查找匹配的行数

find . -name '\*.sql' -exec grep –i

rm –rf 递归强制删除

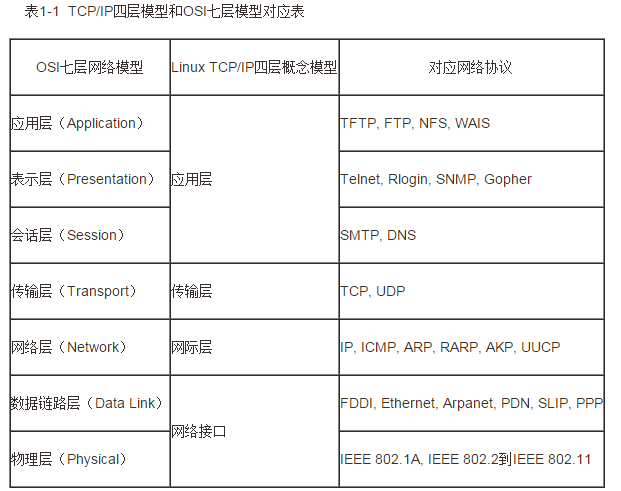
sudo chmod u+x g+w o+r filename 修改权限

tar –zxvf文件名 解压 (-z：gzip压缩文件；-v：显示压缩或解压缩过程；-f：使用档名；-x：解压)

netstat显示网络状态信息

# 通信协议

## 1 说一下TCP /IP四层?



## 2 常见的状态码

**2开头** （请求成功）表示成功处理了请求的状态代码。

200 （成功） 服务器已成功处理了请求。 通常，这表示服务器提供了请求的网页。

203 （非授权信息） 服务器已成功处理了请求，但返回的信息可能来自另一来源。

204 （无内容） 服务器成功处理了请求，但没有返回任何内容。

205 （重置内容） 服务器成功处理了请求，但没有返回任何内容。

206 （部分内容） 服务器成功处理了部分 GET 请求。

**3开头** （请求被重定向）表示要完成请求，需要进一步操作。

301 （永久移动） 请求的网页已永久移动到新位置。

302 （临时移动）

**4开头** （请求错误）这些状态代码表示请求可能出错，妨碍了服务器的处理。

400 （错误请求） 服务器不理解请求的语法。

401 （未授权） 请求要求身份验证。 对于需要登录的网页，服务器可能返回此响应。

403 （禁止） 服务器拒绝请求。

404 （未找到） 服务器找不到请求的网页。

**5开头**（服务器错误）这些状态代码表示服务器在尝试处理请求时发生内部错误。

500 （服务器内部错误） 服务器遇到错误，无法完成请求。

502 （错误网关） 服务器作为网关或代理，从上游服务器收到无效响应。

503 （服务不可用） 服务器目前无法使用（由于超载或停机维护）。

# 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?

## 2 抢红包

3 给你一个3t的文件，里面有很多ipv4的地址，内存只有1g，怎么算出所有ip各自出现的次数

4 给你一个百万级的list，然后给你十个线程算他们的和

5 遍历arraylist 删除元素

# 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 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.**

## 3 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。

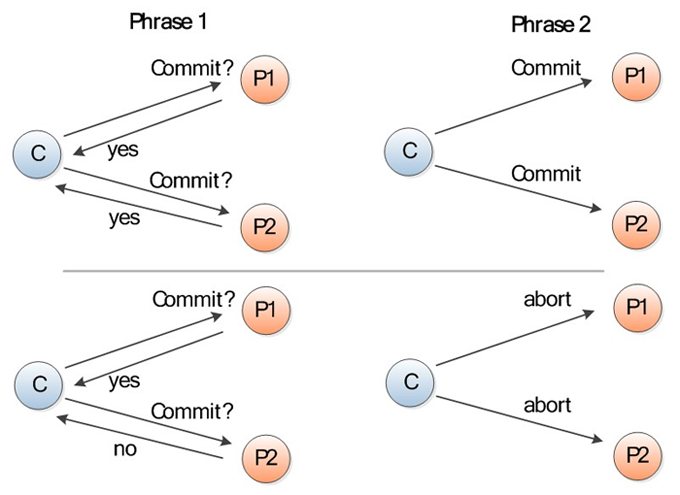
③ **BASE理论**：在分布式系统中，我们往往追求的是可用性，它的重要程序比一致性要高，那么如何实现高可用性呢？提出来了另外一个理论，就是BASE理论，它是用来对CAP定理进行进一步扩充的。BASE理论指的是：Basically Available（基本可用）、Soft state（软状态）、Eventually consistent（最终一致性）。BASE理论是对CAP中的一致性和可用性进行一个权衡的结果，理论的核心思想就是：我们无法做到强一致，但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性（Eventual consistency）。

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。

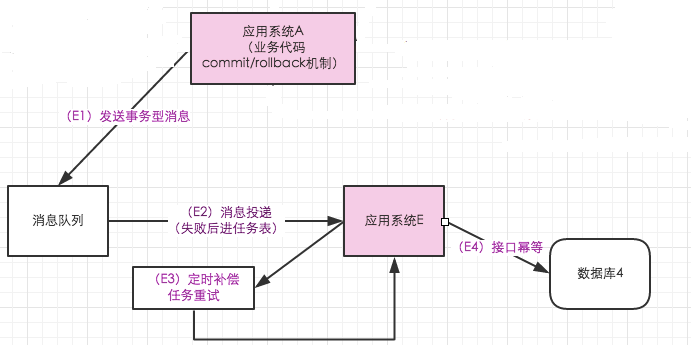
## 9 数据最终一致性方案

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。

## 11 消息队列丢消息怎么处理

## 12 zookeeper选举算法

# 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 的initial bean具体怎么实现的

(1)容器启动时：首先调用BeanFactoryPostProcessor ->postProcessBeanFactory()

(2)getBean时：实例化之后调用： InstantiationAwareBeanPostProcessor

->postProcessPropertyValues()

(3)初始化时：

①设置属性值；

②调用Bean中的BeanNameAware.setBeanName()方法，如果(实现了BeanNameAware接口)；

③调用Bean中的BeanFactoryAware.setBeanFactory()方法，如果(实现BeanFactoryAware接口)；

④调用BeanPostProcessors.postProcessBeforeInitialization()方法；@PostConstruct注解后的方法就是在这里被执行的；

⑤调用Bean中的afterPropertiesSet方法，如果该Bean实现了InitializingBean接口；

⑥调用Bean中的init-method，通常是在配置bean的时候指定了init-method，例如：<bean class="beanClass" init-method="init"></bean>

⑦调用BeanPostProcessors.postProcessAfterInitialization()方法；

⑧如果该Bean是单例的，则当容器销毁并且该Bean实现了DisposableBean接口的时候，调用destory方法；如果该Bean是prototype，则将准备好的Bean提交给调用者，后续不再管理该Bean的生命周期。

## 4 容器初始化



## 5 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。

## 6 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。

## 7 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。

## 12 spring注入static变量

(1) spring 不允许/不支持把值注入到静态变量中，@Value("${mongodb.db}")会得到null；解决方法有：

|  |  |
| --- | --- |
| ①可以利用非静态setter 方法注入静态变量。  @Value("${mongodb.db}")  public void setDatabase(String db) {  DATABASE = db;  } | ②使用配置文件的方式注入：  <bean class="TestStatic">  <property name="from" value="abc"/>  </bean> |

(2) 注入的是static类：

①配置文件

|  |
| --- |
| <bean id="mongoFileOperationUtil" class="com.\*.\*.MongoFileOperationUtil" init-method="init">  <property name="dsForRW" ref="dsForRW"/>  </bean> |

②@PostConstruct方式实现：

|  |
| --- |
| @Component  public class MongoFileOperationUtil {  @Autowired  private static AdvancedDatastore dsForRW;  private static MongoFileOperationUtil mongoFileOperationUtil;  **@PostConstruct**  **public void init() {**  mongoFileOperationUtil = this;  mongoFileOperationUtil.dsForRW = this.dsForRW;  }  } |

③3.set方法上添加@Autowired注解，类定义上添加@Component注解；

|  |
| --- |
| private static AdvancedDatastore dsForRW;  @Autowired  public void setDatastore(AdvancedDatastore dsForRW) {  MongoFileOperationUtil.dsForRW = dsForRW;  } |

# 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 队列同步器AbstractQueuedSynchronizer

队列同步器AQS 是用来构建锁或者其他同步组件的基础框架。它使用了一个volatile int state成员变量来控制同步状态。当state=0时，则说明没有任何线程占有共享资源的锁；当state=1时，则说明有线程目前正在使用共享变量，AQS则会将当前线程以及等待状态等信息封装为Node结点调用CAS方法compareAndSetTail()加入同步队列的尾部等待；在释放同步状态时，将会唤醒后继节点，后继节点将会在获取同步状态成功时将自己设置为首节点。

AQS内部通过内部类Node构成FIFO的同步队列【双向链表】来完成线程获取锁的排队工作，同时利用内部类ConditionObject构建等待队列，当Condition调用await()方法后，线程将会加入等待队列中，而当Condition调用signal()方法后，线程将从等待队列转移动同步队列中进行锁竞争。注意这里涉及到两种队列，一种的同步队列，当线程请求锁而等待的后将加入同步队列等待，而另一种则是等待队列(可有多个)，通过Condition调用await()方法释放锁后，将加入等待队列。

(1)**独占式同步状态获取？**

①通过调用aqs的acquire(int arg)方法获取同步状态，完成同步状态获取、节点构造、加入同步队列以及在同步队列中自旋等待的工作。

主要是：①先调用实现者的tryAcquire(int arg)方法，该方法保证线程安全的获取同步状态。②如果获取失败，则构造同步节点，并通过addWaiter(Node node)方法加入到同步队列的尾部；③调用acquireQueued(node, int)方法，以死循环的方式获取同步状态，直到前驱节点的出列或阻塞线程被中断实现。

②addWaiter()：先通过快速尝试设置尾节点，如果失败，则调用enq(Node node)方法设置尾节点。两个方法都是通过一个CAS方法compareAndSetTail(Node expect, Node update)来设置尾节点，该方法可以确保节点是线程安全添加的。在enq(Node node)方法中，AQS通过“死循环”的方式来保证节点可以正确添加，只有成功添加后，当前线程才会从该方法返回，否则会一直执行下去。

③acquireQueued()：节点进入同步队列之后，就进入了一个自旋过程，当条件满足，获得同步状态。当前线程在“死循环”中尝试获取同步状态，而只有前驱节点是头结点才能够尝试获取同步状态。

**(2) 共享式同步状态获取**

①通过调用同步器的acquireShared(int arg)方法可以共享式地获取同步状态。首先调用实现类的 tryAcquireShared(int arg)方法尝试获取同步状态。当返回值大于等于0时，表示能够获取到同步状态。因此，在共享式获取的自旋过程中，成功获取到同步状态并退出自旋的条件就是tryAcquireShared(int arg)方法返回值大于等于0。即当前节点的前驱为头结点时，尝试获取同步状态，如果返回值大于等于0，表示该次获取同步状态成功并从自旋过程中退出。获取失败，自旋获取同步状态。

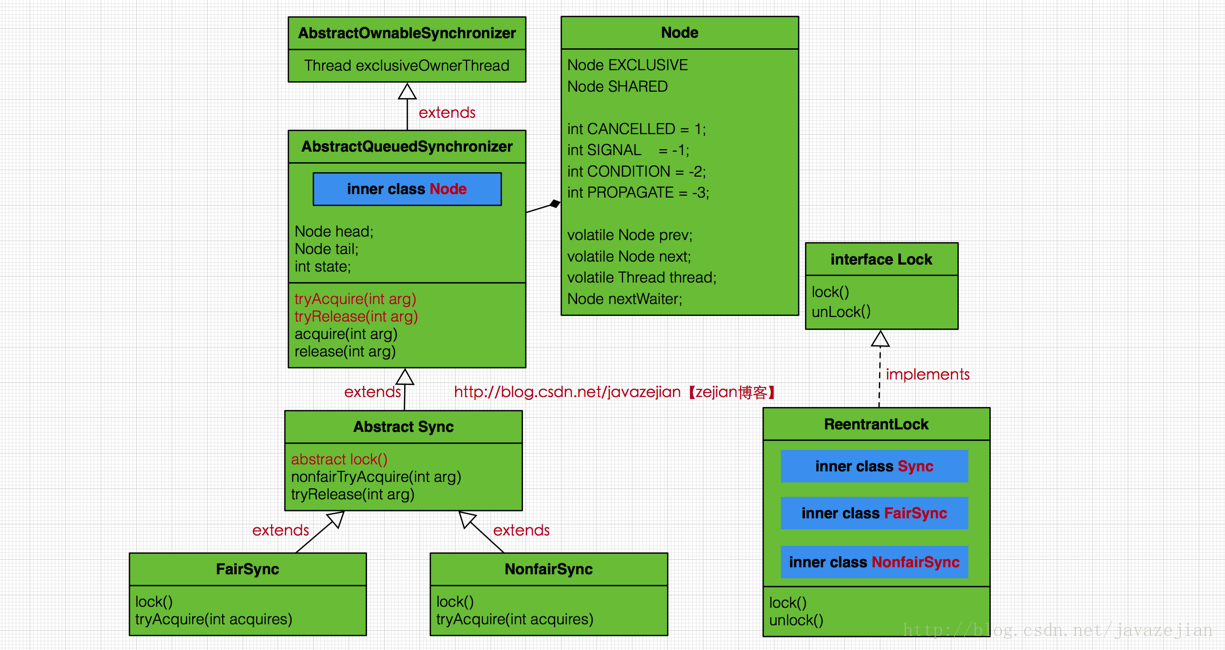
## 5 Lock

在java中有两种方法实现锁机制，一种是synchronized，而另一种是Lock。Lock确保当一个线程位于代码的临界区时，另一个线程不进入临界区， Lock接口及其实现类提供了更加强大、灵活的锁机制。

|  |  |
| --- | --- |
| public void test(){  synchronized(this){  //do something  }  } | Lock lock = new ReentrantLock();  lock.lock();  try{  //do something  } **finally {**  **lock.unlock(); //**注意：**unlock()操作必须在finally中**，这样可确保即使  } //执行抛出异常，线程最终也能正常释放锁。 |

在Java 1.5中，官方在concurrent并发包中加入了Lock接口，该接口中提供了lock()方法和unLock()方法对显式加锁和显式释放锁操作进行支持。Lock对象锁还提供了synchronized所不具备的其他同步特性，如尝试非阻塞地获取锁，能被中断锁的获取(synchronized在等待获取锁时是不可中的)，超时锁的获取，等待唤醒机制的多条件变量Condition等。

## 6 ReetrantLock

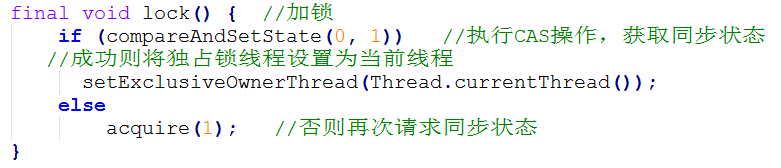


ReentrantLock，可重入锁，是一种独占锁。ReentrantLock提供了公平锁也非公平锁的选择，构造方法接受一个可选的公平参数（默认非公平锁），**当设置为true时，表示公平锁，否则为非公平锁**。公平锁与非公平锁的区别在于公平锁的锁获取是有顺序的。

Sync为ReentrantLock里面的一个内部类，它继承AQS（AbstractQueuedSynchronizer），它有两个子类：公平锁FairSync和非公平锁NonfairSync。

**(1) 非公平锁**

**①获取锁**



* 首先对同步状态执行CAS操作，尝试把state的状态从0设置为1。如果返回true则代表获取同步状态成功，如果返回false，获取锁失败，执行 acquire(1)方法，该方法是AQS中的方法；
* acquire(int arg)：首先会执行tryAcquire(arg)方法，该方法在AQS中并没有具体实现，而是交由子类实现，因此该方法是由ReetrantLock类内部实现的；
* 非公平锁中实现tryAcquire：①尝试再次获取同步状态，如果获取成功则将当前线程设置为OwnerThread，返回true；②判断当前线程current是否为OwnerThread，如果是则属于重入锁，state自增1，并获取锁成功，返回true；③反之，失败，返回false；
* 回到AQS中acquire方法：如果tryAcquire(arg)返回true，获取锁成功；如果tryAcquire(arg)返回false，则会执行addWaiter(Node.EXCLUSIVE)【ReentrantLock属于独占锁，因此结点类型为Node.EXCLUSIVE】加入同步队列队尾。如果第一次加入或者CAS入队失败，则执行enq入队操作：如果还没有初始同步队列则创建新结点并使用compareAndSetHead设置头结点，tail也指向head；如果队列已存在，则将新结点node添加到队尾。
* acquireQueued(addWaiter(Node.EXCLUSIVE), arg))：节点加入同步队列之后，就进入了一个自旋过程，当条件满足，获得同步状态。当前线程在“死循环”中尝试获取同步状态，而只有前驱节点是头结点才能够尝试获取同步状态。

**②释放锁**

|  |
| --- |
| public void unlock() {  sync.release(1);  } |

* unlock内部使用Sync的release(int arg)释放锁；该方法是AQS中的方法；
* release()：首先调用tryRelease尝试释放锁，此该方法是由ReetrantLock类内部实现的。如果释放锁成功，唤醒后继节点的线程；
* 非公平锁中实现tryRelease：首先用一个临时变量c保存释放锁之后的状态值；然后判断状态是否为0，如果是，则说明已释放同步状态，设置Owner为null；这是更新同步状态，如果状态值为0，返回true；

**(2) 公平锁**

**公平锁与非公平锁的区别在于获取锁的时候是否按照FIFO的顺序来。释放锁不存在公平性和非公平性。**

**①获取锁**

比较非公平锁和公平锁获取同步状态的过程，会发现两者唯一的区别就在于公平锁在获取同步状态时多了一个限制条件：hasQueuedPredecessors()：即判断当前线程是否位于CLH同步队列中的第一个。如果是则返回true，否则返回false。如果该方法返回true，则表示有线程比当前线程更早地请求获取锁，因此需要等待前驱线程获取并释放锁之后才能继续获取锁。

## 7 Condition

Condition是一个接口类，通过Condition能够精细的控制多线程的休眠与唤醒。对于一个锁，我们可以为多个线程间建立不同的Condition。Condition的实现类是AQS的内部类ConditionObject。

**Condition的实现原理**：使用Condition前必须获得锁，同时在Condition的等待队列上的结点与前面同步队列的结点是同一个类即Node，其结点的waitStatus的值为CONDITION。

每个Condition都对应着一个等待队列，也就是说如果一个锁上创建了多个Condition对象，那么也就存在多个等待队列。等待队列是一个FIFO的队列【单链表】，在队列中每一个节点都包含了一个线程的引用，而该线程就是Condition对象上等待的线程。当一个线程调用了await()相关的方法，那么该线程将会释放锁，并构建一个Node节点封装当前线程的相关信息加入到等待队列中进行等待，直到被唤醒、中断、超时才从队列中移出。

**等待**：await()方法主要做了3件事，①调用addConditionWaiter()方法将当前线程封装成node结点加入等待队列，②调用fullyRelease(node)方法释放同步状态并唤醒后继结点的线程。三是调用isOnSyncQueue(node)方法判断结点是否在同步队列中，注意是个while循环，如果同步队列中没有该结点就直接挂起该线程，需要明白的是如果线程被唤醒后就调用acquireQueued(node, savedState)执行自旋操作争取锁，即当前线程结点从等待队列转移到同步队列并开始努力获取锁。

## Semaphore

## 7

5 阻塞队列和非阻塞队列有什么不同，他们是怎么实现的

6 怎么实现一个线程池，线程池原理

7 cyclicbarrier和coutdownlatch有什么不同

8 hashmap和concurrenthashmap

# JVM

## 1 详细JVM内存分布

答： Java虚拟机所管理的内存将会包括以下几个运行是数据区域：

**①程序计数器：**当前线程所执行的字节码行号指示器。每一条线程都有一个独立的程序计数器，即为线程私有的。注意，此内存区域是唯一一个没有规定任何OutOfMemoryError情况的区域。

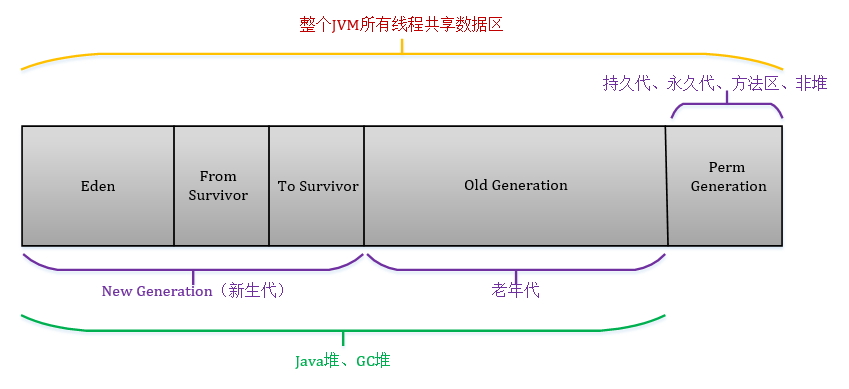
**②Java虚拟机栈(Stack)：**生命周期与线程相同，虚拟机栈用于存放局部变量表，操作数栈，动态链接，方法出口等信息。两种异常状况：①如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常；②如果虚拟机栈在动态扩展时无法申请到足够的内存，将抛出OutOfMemoryError异常。

**③本地方法栈：**本地方法栈则为虚拟机使用到的Native方法服务，两种异常。

**④Java堆(Heap)：**Java 堆（Java Heap）是被所有线程共享的内存区域，在虚拟机启动时创建。这个区域是用来存放对象实例的，几乎所有对象实例都会在这里分配内存。堆是Java垃圾收集器管理的主要区域（GC堆）。Java堆可以细分为：新生代和老年代；再细致一点的有Eden空间，From Survivor空间，To Survivor空间等。空间比例为Eden:S0:S1==8:1:1。Java堆可以处于物理上不连续的内存空间中，只要逻辑上是连续的即可，就像我们的磁盘空间一样。可以通过-Xmx和-Xms控制。

**⑤方法区 ：**方法区（Method Area）也叫永久代，永久代也是被所有线程共享的内存区域，它用于存储已被虚拟机加载的类信息、常量、静态变量（JDK7中被移到Java堆）、即使编译器编译后的代码等数据。

**运行时常量池，**它是方法区的一部分，用于存放编译期生成的各种字面量和符号引用（其实就是八大基本类型的包装类型和String类型数据**（JDK7中被移到Java堆）**）。从JDK7开始永久代的移除工作，贮存在永久代的一部分数据已经转移到了Java Heap或者是Native Heap。但**永久代仍然存在于JDK7，并没有完全的移除**，随着**JDK8的到来，JVM不再有PermGen。**



* **JVM中共享数据空间可以分成三个大区**，新生代（Young Generation）、老年代（Old Generation）、永久代（Permanent Generation），其中JVM堆分为新生代和老年代
* 新生代可以划分为三个区，Eden区（存放新生对象），两个幸存区（From Survivor和To Survivor）（存放每次垃圾回收后存活的对象）。

## 2 JVM内存模型

Java内存模型是JVM的抽象模型（主内存，本地内存）；

(1)**主内存与工作内存**

Java内存模型的主要目标是**定义程序中各个变量的访问规则**，即在虚拟机中将变量存储到内存和从内存中取出变量这样的底层细节。此处的变量包括：实例字段、静态字段和构成数组对象的元素，但不包括局部变量与方法参数。

Java内存模型规定了所有的变量都存储在主内存中。每条线程还有自己的工作内存，线程的工作内存中保存了该线程使用到的变量的主内存副本拷贝，线程对变量的所有操作（读取、赋值）都必须在工作内存中进行，而不能直接读写主内存中的变量。不同的线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成。

**(2) 内存间的交互操作**

Java内存模型定义了以下八种操作来完成，每一种操作都是原子的、不可再分的：

①**lock（锁定）**：作用于主内存的变量，把一个变量标识为一条线程独占状态；

②**unlock（解锁）**：作用于主内存变量，把锁定状态的变量释放出来，释放后才可以被其他线程锁定；

③**read（读取）**：作用于主内存变量，把一个变量值从主内存传输到线程的工作内存中；

④**load（载入）**：作用于工作内存的变量，把read操作从主内存中得到的变量值放入工作内存的变量副本中。

⑤**use（使用）**：作用于工作内存的变量；

⑥**assign（赋值）**：作用于工作内存的变量；

⑦**store（存储）**：作用于工作内存的变量，把工作内存中的一个变量的值传送到主内存中；

⑧**write（写入）**：作用于主内存的变量，它把store操作从工作内存中一个变量的值传送到主内存的变量中。

如果要把一个变量从主内存中复制到工作内存，就需要按顺寻地执行**read和load**操作，如果把变量从工作内存中同步回主内存中，就要按顺序地执行**store和write**操作。Java内存模型只要求上述操作**必须按顺序执行，而没有保证必须是连续执行**。

**(3) Java内存模型还规定了在执行上述八种基本操作时，必须满足如下规则：**

①不允许read和load、store和write操作之一单独出现，既不允许一个变量从主内存读取了但工作内存不接受，或者从工作内存发起回写了但主内存不接受的情况出现。

②不允许一个线程丢弃它的最近assign的操作，即变量在工作内存中改变了之后必须同步到主内存中。

③不允许一个线程无原因地（没有发生过任何assign操作）把数据从工作内存同步回主内存中。

④一个新的变量只能在主内存中诞生，不允许在工作内存中直接使用一个未被初始化的变量。即就是对一个变量实施use和store操作之前，必须先执行过了assign和load操作。

⑤一个变量在同一时刻只允许一条线程对其进行lock操作，但lock操作可以被同一条线程重复执行多次，多次执行lock后，只有执行相同次数的unlock操作，变量才会被解锁。lock和unlock必须成对出现;

⑥如果对一个变量执行lock，将会清空工作内存中此变量的值，使用这个变量前需要重新执行load或assign操作初始化变量的值;

⑦如果一个变量事先没有被lock操作锁定，则不允许对它执行unlock操作；也不允许去unlock一个被其他线程锁定的变量;

⑧对一个变量执行unlock操作之前，必须先把此变量同步到主内存中（执行store和write操作）。

**(4)对于volatile型变量的特殊规则**

当一个变量定义为volatile之后，它将具备两种特性：第一：保证此变量对所有线程的可见性。第二：禁止指令重排序 (在JDK1.5中才被完全修复) 。

**(5)对于long和double型变量的特殊规则**

Java模型对于64为的数据类型（long和double），在模型中特别定义了一条相对宽松的规定：允许虚拟机将没有被volatile修饰的64位数据的读写操作分为两次32为的操作来进行，即允许虚拟机实现选择可以不保证64位数据类型的load、store、read和write这4个操作的原子性。目前虚拟机都把64位数据的读写操作作为原子操作来对待。

**(6)原子性、可见性和有序性**

JAVA内存模型是围绕着并发过程中如何处理原子性、可见性和有序性来建立的：

**①原子性：**即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。基本数据类型的访问读写是具备原子性的（例外是long和double），在synchronized块之间的操作也具备原子性。

**②可见性：**可见性是指当一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。除了volatile之外，Java还有两个关键字能实现可见性：synchronized和final。

**③有序性：**即程序执行的顺序按照代码的先后顺序执行。Java提供了volatile和synchronized两个关键字来保证线程之间操作的有序性。

**(7)先行发生原则happen-before**

这些先行发生关系无须任何同步就已经存在，如果不再此列就不能保障顺序性，虚拟机就可以对它们任意地进行重排序：

1.程序次序规则：在一个线程内，按照程序代码顺序，书写在前面的操作先行发生于书写在后面的操作。

2.管程锁定规则：同一个锁，一个unlock操作先发生于后面对同一个锁的lock操作。

3.Volatile变量规则：对一个volatile变量的写操作先行发生于后面对这个变量的读操作。

4.线程启动规则：Thread对象的start()方法先行发生于此线程的每一个动作

5.线程终止规则：线程中的所有操作都先发生于对此线程的终止；

6.线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断时间的发生；

7.对象终结规则：一个对象的初始化完成(构造函数执行结束)先行发生于它的finalize()方法的开始

8.传递性：如果操作A先发生于操作B，操作B先发生于操作C，那就可以得出操作A先发生于操作C。

## 3 讲讲什么情况下会出现内存溢出，内存泄漏？

**内存溢出(out of memory)**：指程序在申请内存时，没有足够的内存空间供其使用，出现out of memory。

**内存泄露(memory leak)**：指程序在申请内存后，无法释放已申请的内存空间。

**内存溢出的原因：**

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；

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；

4.使用的第三方软件中的BUG；

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。

**StackOverflowError：**线程请求的栈深度大于虚拟机所允许的深度时抛出该异常。要么是方法调用层次过多（比如存在无限递归调用），要么是线程栈太小。减少方法调用层次；调整-Xss参数增加线程栈大小。

**OutOfMemoryError: PermGen space**：在JDK8之前的HotSpot实现中，类的元数据如方法数据、方法信息（字节码，栈和变量大小）、运行时常量池、已确定的符号引用和虚方法表等被保存在永久代中，32位默认永久代的大小为64M，64位默认为85M，可以通过参数-XX:MaxPermSize进行设置。JDK8的HotSpot中，把永久代从Java堆中移除了，并把类的元数据直接保存在本地内存区域。

## 4 说说Java线程栈

jstack pid可以查看线程栈：JVM线程堆栈是一个给定时间的快照，提供所有被创建出来的Java线程的完整清单。每一个被发现的Java线程都会给你如下信息：– 线程的名称，- 线程类型 & 优先级， - Java线程ID， - 状态 以及– Java线程栈跟踪。

(1) found one java-level deadlock ，线程1 waiting to lock A，locked B；线程2 waiting to lock B，locked A；

## 5 JVM垃圾回收机制--新生代到年老代

堆大小=新生代+老年代；（新生代占堆空间的1/3、老年代占堆空间2/3）。新生代又分为 eden、from survivor、to survivor(8:1:1)。

当新创建对象时一般在新生代中分配内存空间，当新生代垃圾收集器回收几次之后仍然存活的对象会被移动到年老代内存中，当大对象在新生代中无法找到足够的连续内存时也直接在年老代中创。当Eden区内存不够的时候就会触发MinorGC，对新生代区进行一次垃圾回收。

**MinorGC的过程：**MinorGC采用**复制算法**。首先，把Eden和Servivor From中还存活的对象一次性复制到Servicor To区域（如果有对象的年龄以及达到了老年的标准，则赋值到老年代区），同时把这些对象的年龄+1（如果Servicor To不够位置了就放到老年区，当年龄达到某个值时（默认15，通过设置参数-xx:maxtenuringThreshold来设置），这些对象就会成为老年代）；然后，清空Eden和ServicorFrom中的对象；最后，ServicorTo和Servicor From互换，原Servicor To成为下一次GC时的Servicor From区。

注意：**(1) 对象进入老年代的4种情况：**

①假如进行Minor GC时发现，存活的对象在ToSpace区中存不下，那么把存活的对象存入老年代;②大对象直接进入老年代；③长期存活的对象将进入老年代；④动态对象年龄判定：如果在From空间中，相同年龄所有对象的大小总和大于From和To空间总和的一半，那么年龄大于等于该年龄的对象就会被移动到老年代，而不用等到15岁(默认)：。

**(2) 触发Full GC：**①调用System.gc时，系统建议执行Full GC，但是不必然执行；②老年代空间不足；③perm gen 空间不足；④通过Minor GC后进入老年代的平均大小大于老年代的可用内存；⑤由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小。

**(3) 内存分配合回收策略：**

* 对象优先在Eden 分配；
* 大对象直接进入老年代；
* 长期存活的对象将进入老年代；
* 动态对象年龄判定；
* 空间分配担保：

①在发生Minor GC之前，虚拟机会先检查老年代最大可用的连续空间是否大于新生代所有对象总空间，如果这个条件成立，那么Minor GC可以确保是安全的。②如果不成立，则虚拟机会查看HandlerPromotionFailure这个参数设置的值(true或flase)是否允许担保失败，③如果允许，那么会继续检查老年代最大可用的连续空间是否大于历次晋升到老年代对象的平均大小，如果大于，将尝试进行一次Minor GC；④如果小于，或者HandlerPromotionFailure为false，那么这次Minor GC将升级为Full GC。

## 6 JVM垃圾回收机制-- 频繁full GC

开启了 -XX:+HeapDumpBeforeFullGC。用jvisualvm查看，查看占用内存多的数据：

①程序里有内存缓存，缓存的是字符串，内存缓存逐渐增多，逐步移步老年代，最终导致爆满。

②有大量拼接字符串的地方，

③static的变量，存储大量的字符串，排名第六的是hashMap，猜想可能是有static的 hashMap？？

解决方案：①Xms值与Xmx相等，这样就不会因为所使用的Java堆不够用而进行调节；

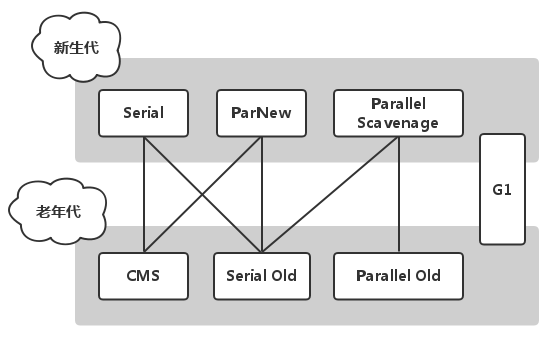
②-XX:**CMSInitiatingOccupancyFraction**=80 -XX:+UseCMSInitiatingOccupancyOnly：在old区达到80%时FGC；

③-XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintHeapAtGC -**Xloggc**:gc.log:打印gc信息。

④开启XX:+CMSScavengeBeforeRemark：在FGC前执行YGC，防止老年代对象的GC ROOT在新生代的话；

④程序优化：尽早释放无用对象的引用，特别注意一些复杂对象，如数组，队列等，将不用的引用对象赋为null；尽量少用finalize函数

## 7 各种回收器，各自优缺点，重点CMS、G1



**(1) Serial收集器（串行GC）**

Serial 是一个采用**单个线程**并基于**复制算法**工作在**新生代**的收集器，进行垃圾收集时，必须暂停其他所有的工作线程。对于单CPU环境来说，Serial由于没有线程交互的开销，可以很高效的进行垃圾收集动作，是Client模式下新生代默认的收集器。

**(2) ParNew收集器（并行GC）**

ParNew是serial的多线程版本，许多运行在Server模式下的虚拟机中首选的新生代收集器，除Serial外，只有它能与CMS收集器配合工作。除了使用多条线程进行垃圾收集之外，其余与Serial一样。

**(3) Parallel Scavenge收集器（并行回收GC）**

Parallel Scavenge是一个采用**多线程**基于**复制算法**并工作在新生代的收集器，其关注点在于达到一个可控的吞吐量，经常被称为“吞吐量优先”的收集器。(吞吐量 = 用户代码运行时间 /(用户代码运行时间 + 垃圾收集时间))。

**(4) Serial Old收集器（串行GC）**

Serial Old是Serial收集器的老年代版本，同样是单线程收集器，使用“标记-整理算法”。

**(5) Parallel Old收集器（并行GC）**

Parallel Old 是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理算法”算法。

**(6) CMS收集器（并发GC）**

CMS(Concurrent Mark Sweep) 是一种以**获得最短回收停顿时间为目标**的收集器，基于“标记-整理算法”算法。整个过程分为以下4步：

①**初始标记**：这个过程只是标记一下GC Roots能够直接关联的对象，但是仍然会Stop The World；  
②**并发标记**：进行GC Roots Tracing的过程，可以和用户线程一起工作。  
③**重新标记**：用于修正并发标记期间由于用户程序继续运行而导致标记产生变动的那部分记录，这个过程会暂停所有线程，但其停顿时间远比并发标记的时间短；  
④**并发清理**：可以和用户线程一起工作。

优点： **并发收集、低停顿**；

缺点：① **CMS对CUP资源非常敏感** 。在并发阶段，占用部分线程而导致应用程序变慢， **总吞吐量降低**。

②**CMS无法处理浮动垃圾**。可能出现Concurrent Mode Failure 失败而导致另一次 fullGC的产生。CMS不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。在 JDK1.6中，启动阈值为92%。要是 CMS运行期间预留的内存无法满足程序需要就会出现 Concurrent Mode Failure失败，这是就会临时启用 Serial old收集器来重新进行老年代的垃圾收集，这样停顿时间就更长了。

③CMS是基于“标记-清除”算法实现的收集器，**在收集结束时会有大量空间碎片产生**，可能导致出现老年代剩余空间很大，却无法找到足够大的连续空间分配当前对象，不得不提前触发一次Full GC。

**(7) G1收集器**

G1（Garbage First）是JDK1.7提供的一个工作**在新生代和老年代**的收集器，基于“**标记-整理”**算法实现，在收集结束后可以避免内存碎片问题。

**G1优点：**①并行与并发：充分利用多CPU来缩短Stop The World的停顿时间；  
②分代收集：采用不同的方式处理新建的对象、已经存活一段时间和经历过多次GC的对象；  
③空间整合：整体来看是基于“标记 -整理”算法实现的，从局部看来是基于“复制”算法实现的;  
④停顿预测：G1中可以建立可预测的停顿时间模型，能让使用者明确指定在M毫秒的时间片段内，消耗在垃圾收集上的时间不得超过N毫秒。

使用G1收集器时，整个Java堆会被划分为多个大小相等的独立区域Region，新生代和老年代不再是物理隔离了，都是一部分Region（不需要连续）的集合。G1会跟踪各个Region的垃圾收集情况（回收空间大小和回收消耗的时间），维护一个优先列表，根据允许的收集时间，优先回收价值最大的Region。为了可达性判定确定对象是否存活的时候保证准确性，使用 Remembered Set来避免全堆扫描。G1的每个 region都有一个与之对应的Remembered Set，在 GC根节点的枚举范围中加入 Remembered Set及可保证不对全堆扫描也不会有遗漏。

## 8 各种回收算法

答：垃圾收集算法主要有：标记-清除、复制和标记-整理。

(1)**标记-清除算法**：

算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。

缺点：①效率问题：标记和清除两个过程的效率都不高；②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。

**(2)复制算法**

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。

**(3) 标记-整理算法**

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。

**(4) 分代收集算法**

新生代中，只有少量对象存活，所以选择复制算法。Java虚拟机对新生代的垃圾回收称为Minor GC，次数比较频繁，每次回收时间也比较短。使用java虚拟机**-Xmn**参数可以指定新生代内存大小。

老年代中因为对象存活率高，使用“标记-清理”或者“标记-整理”算法。Java虚拟机对年老代的垃圾回收称为MajorGC/Full GC。

永久代也使用标记-整理算法进行垃圾回收，java虚拟机参数-XX:PermSize和-XX:MaxPermSize可以设置永久代的初始大小和最大容量。

## 9 Java中的引用

在Java层面，一共有四种引用：强引用、软引用、弱引用、虚引用，从名字也可以发现，这几种引用的生命周期由强到弱。

**(1)强引用（Strong Reference）**：使用最普遍的引用，99%的代码可能都是强引用，是指创建一个对象并把这个对象赋给一个引用变量。类似“Object obj = new Object()”这类的应用。如果一个对象，和GC Root有强引用的关系，当内存不足发生GC时，宁可抛出OOM异常，终止程序，也不会回收这些对象。相反，当一个对象，和GC Root没有强引用关系时，可能会被回收（因为可能还有其它引用），如果没有任何引用关系，GC之后，该对象就被回收了。

**(2)软引用（Soft Reference）**：主要用来描述一些不那么重要的对象，内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存，这种特别适合用来实现缓存。比如：

|  |
| --- |
| Object reference = new MyObject();  Reference root = new **SoftReference**(reference); //在JDK1.2之后， SoftReference实现软应用。  reference = null; // MyObject对象只有软引用 |

**(3)** **弱引用（Weak Reference）**：相对于软引用，它的生命周期更短，当发生GC时，如果扫描到一个对象只有弱引用，不管当前内存是否足够，都会对它进行回收，被弱引用关联的对象只能生存到下一次垃圾收集发生之前。在java中，用java.lang.ref.WeakReference类来表示。

**(4)** **虚引用（Phantom Reference）：**和之前两种引用的最大不同是：它的get方法一直返回null。为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。虚引用的使用场景很窄，在JDK中，目前只知道在申请堆外内存时有它的身影。提供了PhantomReference类来实现弱应用。

## 10 类加载为什么要使用双亲委派模式，有没有什么场景是打破了这个模式？

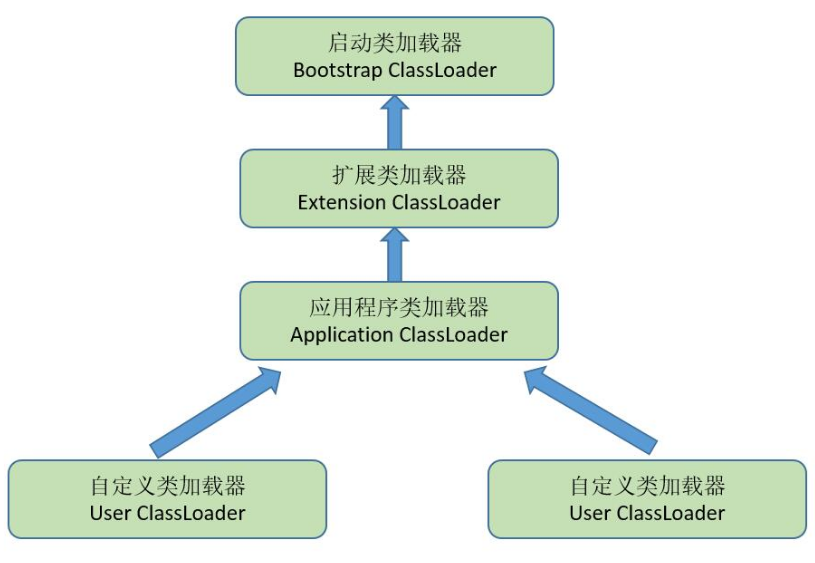
JVM提供了3种类加载器：

**(1)启动类加载器（Bootstrap ClassLoader）：**是用本地代码实现的类装入器，负责加载 JAVA\_HOME\lib 目录中的，或通过-Xbootclasspath参数指定路径中的，且被虚拟机认可（按文件名识别，如rt.jar）的类。

**(2)扩展类加载器（Extension ClassLoader）：**负责加载 JAVA\_HOME\lib\ext 目录中的，或通过java.ext.dirs系统变量指定路径中的类库。

**(3)应用程序类加载器（Application ClassLoader）：**负责加载用户路径（classpath）上的类库,开发者可以直接使用这个类加载器，如果没有自定义过自己的类加载器，一般情况下这个就是程序中默认的类加载器。

JVM基于上述类加载器，通过双亲委派模型进行类的加载，当然我们也可以通过继承java.lang.ClassLoader实现自定义的类加载器。这张图表示**类加载器的双亲委派模型**。



**双亲委派模型的工作过程是**：如果一个类加载器收到了类加载的请求，它首先不会自己去尝试加载这个类，而是把这个请求委派给父类加载器去完成，每一个层次的类加载器都是如此，因此所有的加载请求最终都是应该传送到顶层的启动类加载器中，只有当父类加载器反馈自己无法完成这个加载请求（它的搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。

**使用双亲委派模型的好处就是：**保证某个范围的类一定是被某个类加载器所加载的，这就保证在程序中同一个类不会被不同的类加载器加载。例如类java.lang.Object,它存放在rt.jar中，无论哪一个类加载器要加载这个类，最终都是委派给处于模型最顶端的启动类加载器进行加载，因此Object类在程序的各种类加载器环境中都是同一个类。相反，如果没有使用双亲委派模型，由各个类加载器自行去加载的话，如果用户自己编写了一个称为java.lang.object的类，并放在程序的ClassPath中，那系统中将会出现多个不同的Object类，Java类型体系中最基础的行为也就无法保证，应用程序也将会变得一片混乱。

**双亲委派模型的实现：**实现双亲委派模型代码都集中在java.lang.ClassLoader的loadClass()方法之中，首先检查是否已经被加载过，若没有加载则调用父加载器的loadClass()方法，若父加载器为空则默认使用启动类加载器作为父加载器。如果父类加载失败，抛出ClassNotFoundException异常后，再调用自己的findClass()方法进行加载。

打破双亲委派机制则不仅要继承ClassLoader类，还要重写loadClass和findClass方法。

## 11 JAVA命令

Jps

Jstat

Jinfo

Jmap

Jstack

## 12 JVM常见启动参数

-Xms / -Xmx — 堆的初始大小 / 堆的最大大小

-Xmn — 堆中年轻代的大小

-XX:-DisableExplicitGC — 让System.gc()不产生任何作用

-XX:+PrintGCDetails — 打印GC的细节

-XX:+PrintGCDateStamps — 打印GC操作的时间戳

-XX:NewSize / XX:MaxNewSize — 设置新生代大小/新生代最大大小

-XX:NewRatio — 可以设置老生代和新生代的比例

-XX:PrintTenuringDistribution — 设置每次新生代GC后输出幸存者乐园中对象年龄的分布

-XX:InitialTenuringThreshold / -XX:MaxTenuringThreshold：设置老年代阀值的初始值和最大值

-XX:TargetSurvivorRatio：设置幸存区的目标使用率

## 12 怎么定位虚拟机应用出现的问题

# 缓存

## 1 Redis用过哪些数据数据，以及Redis底层怎么实现

Redis是一种基于键值对（key-value）的NoSQL数据库，redis中的值可以是由**String**（字符串）、**hash**（哈希）、**list**（列表）、**set**（集合）、**zset（**有序集合）、Bitmaps（位图）等多种数据结构和算法组成。Redis还提供了键过期、发布订阅、事务、Lua脚本等附加功能。Redis使用了**单线程架构**和I/O多路复用模型来实现高性能的内存数据库服务。

|  |  |
| --- | --- |
| 查看所有键 keys \* | 键总数 dbsize |
| 键是否存在 exists key | 删除键 del key |
| 键过期 expire key seconds | 键的剩余过期时间 ttl key |

**(1) 字符串 String**

字符串对象的编码可以是int、raw或者embstr。embstr应该是Redis 3.0新增的数据结构。如果字符串对象的长度小于39字节，就用embstr对象。否则用传统的raw对象。

①设置值 set key value [ex seconds] [px milliseconds] [nx|xx]

ex seconds：为键设置秒级过期时间 px seconds：为键设置毫秒级过期时间

nx：键必须不存在，才能设置成功 xx：键必须存在，才能设置成功，用于更新。

|  |  |
| --- | --- |
| 获取值 get key | 计数 incr key |
| 批量设置/获取值 mset / mget key | 设置并返回原值： getset key value |
| setex key seconds value | setnx key value 失败返回0，成功返回OK |

**(2) 哈希hash**

key= {{field1,value1},…{ fieldN,valueN}}，底层实现类似Java里面的Map<String,Object>

|  |  |
| --- | --- |
| hset key field value | hget key field |
| hmget key field [field …] | Hmset key field value [field value …] |
| hkeys key 获取所有field | hvals key 获取所有value |
| hexists key field field是否存在 | hgetall key 获取所有的field-value |

(3) **列表list**

列表用来存储多个有序的字符串。列表中的元素 **有序 可重复**。list的实现为一个双向链表，即可以支持反向查找和遍历。

|  |  |
| --- | --- |
| 增加 rpush lpush linsert | 查 lrange lindex llen |
| 删除 lpop rpop lrem ltrim | 修改 lset |
| 阻塞操作 blpop/brpop key [key …] timeout 列表为空，阻塞timeout 后返回 | |

**(4) 集合 set**

不允许重复 无序， set 的内部实现是一个 value永远为null的HashMap，即为hashtable。

|  |  |
| --- | --- |
| 添加 sad key element [element …] | 个数 scard key |
| 是否存在 sismenmber key element | 获取所有元素 smembers key |
| 删除 srem key element [element …] | 随机弹出元素 spop key |
| 交集/并集/差集: sinter / suinon / sdiff key [key …] | |

**(5) 有序集合 zset**

sorted set可以通过用户额外提供一个优先级(score)的参数来为成员排序，并且是插入有序的，即自动排序。**zset的成员是唯一的,但分数(score)却可以重复**。

sorted set的内部使用HashMap和跳跃表(SkipList)来保证数据的存储和有序。skiplist是一种跳跃表，它实现了有序集合中的快速查找，在大多数情况下它的速度都可以和平衡树差不多。但它的实现比较简单，可以作为平衡树的替代品。

|  |  |
| --- | --- |
| 添加 zadd key score member [score member …] | 范围的成员 zrange key start end |
| 删除 zrem key member [member …] | 成员个数 zcard key |

**(6) 注意**

persist命令可以删除任意类型键的过期时间，但是set命令也会删除字符串类型键的过期时间，这在开发时容易被忽视。

## 2 Redis缓存穿透，缓存雪崩

**(1)** **缓存穿透**

**什么是缓存穿透？**指查询一个根本不存在的数据，缓存层和存储层都不会命中。缓存穿透将导致不存在的数据每次请求都要到存储层去查询，失去了缓存保护后端存储的意义。

**解决办法：**

**①缓存空对象，将 null 变成一个值**

如果一个查询返回的数据为空（不管是数 据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。

缓存空对象会有两个问题：

第一，空值做了缓存，意味着缓存层中存了更多的键，需要更多的内存空间，针对这类数据设置一个较短的过期时间，让其自动剔除。

第二，缓存层和存储层的数据会有一段时间窗口的不一致，可能会对业务有一定影响。例如过期时间设置为 5分钟，如果此时存储层添加了这个数据，那此段时间就会出现缓存层和存储层数据的不一致，此时可以利用消息系统或者其他方式清除掉缓存层中的空对象。

**②布隆过滤器拦截**

将所有可能存在的数据哈希到一个足够大的bitmap中，一个一定不存在的数据会被这个bitmap拦截掉，从而避免了对底层存储系统的查询压力。这种方式适合数据命中不高，数据性对固定、实时性低（数据集较大）的应用场景。

在访问缓存层和存储层之前，将存在的key用布隆过滤器提前保存起来，做第一层拦截。如果布隆过滤器认为该key不存在，那么就不会访问存储层，在一定程度保护了存储层。

**2) 缓存雪崩**

**什么是缓存雪崩：**当缓存服务器重启或者大量缓存集中在某一个时间段失效，这样在失效的时候，也会给后端系统(比如DB)带来很大压力。

**如何避免？** ①在缓存失效后，通过加锁或者队列来控制读数据库写缓存的线程数量。比如对某个key只允许一个线程查询数据和写缓存，其他线程等待。②不同的key，设置不同的过期时间，让缓存失效的时间点尽量均匀。③保证缓存层服务高可用性。

**3) 缓存击穿/热点key重建优化**

**什么是缓存击穿？**对于一些设置了过期时间的key，如果这些key可能会在某些时间点被超高并发地访问，是一种非常“热点”的数据。这个时候，需要考虑一个问题：缓存被“击穿”的问题，这个和缓存雪崩的区别在于这里**针对某一key缓存**，前者则是很多key。

**如何解决？**①使用互斥锁(mutex key)。简单地来说，就是在缓存失效的时候（判断拿出来的值为空），不是立即去load db，而是先使用缓存工具的某些带成功操作返回值的操作（比如Redis的SETNX或者Memcache的ADD）去set一个mutex key，当操作返回成功时，再进行load db的操作并回设缓存；否则，就重试整个get缓存的方法。此方法只允许一个线程重建缓存，其他线程待重建缓存的线程执行完，重新从缓存获取数据即可。②"提前"使用互斥锁(mutex key)；③永远不过期，这种方法对于性能非常友好，唯一不足的就是构建缓存时候，其余线程(非构建缓存的线程)可能访问的是老数据。

|  |
| --- |
| public String get(key) {  String value = redis.get(key);  if (value == null) { //代表缓存值过期  //设置3min的超时，防止del操作失败的时候，下次缓存过期一直不能load db  if (redis.setnx(key\_mutex, 1, 3 \* 60) == 1) { //代表设置成功  value = db.get(key);  redis.set(key, value, expire\_secs);  redis.del(key\_mutex);  } else { //这个时候代表同时候的其他线程已经load db并回设到缓存了  sleep(50);  get(key); //重试  }  } else {  return value;  }  } |

## 3 如何使用Redis来实现分布式锁

当前没有锁（key不存在），那么就进行加锁操作，并对锁设置个有效期，同时value表示加锁的客户端；已有锁存在，不做任何操作。

|  |
| --- |
| public class RedisTool {  private static final Long RELEASE\_SUCCESS = 1L;  private static final String LOCK\_SUCCESS = "OK";  private static final String SET\_IF\_NOT\_EXIST = "NX";  private static final String SET\_WITH\_EXPIRE\_TIME = "PX";  /\*\*  \* 尝试获取分布式锁  \* @param jedis Redis客户端  \* @param lockKey 锁  \* @param requestId 请求标识  \* @param expireTime 超期时间  \* @return 是否获取成功  \*/  public static boolean tryGetDistributedLock(Jedis jedis, String lockKey, String requestId, int expireTime) {  String result = jedis.set(lockKey, requestId, SET\_IF\_NOT\_EXIST, SET\_WITH\_EXPIRE\_TIME, expireTime);  if (LOCK\_SUCCESS.equals(result)) {  return true;  }  return false;  }  /\*\*  \* 释放分布式锁  \* @param jedis Redis客户端  \* @param lockKey 锁  \* @param requestId 请求标识  \* @return 是否释放成功  \*/  public static boolean releaseDistributedLock(Jedis jedis, String lockKey, String requestId) {  String script = "if redis.call('get', KEYS[1]) == ARGV[1] then return redis.call('del', KEYS[1]) else return 0 end";  Object result = jedis.eval(script, Collections.singletonList(lockKey), Collections.singletonList(requestId));  if (RELEASE\_SUCCESS.equals(result)) {  return true;  }  return false;  }  } |

**加锁：**第一个为key，我们使用key来当锁，因为key是唯一的。第二个为value，我们传的是requestId，通过给value赋值为requestId，就知道这把锁是哪个请求加的了。requestId可以使用UUID.randomUUID().toString()方法生成。第三个为nxxx，这个参数我们填的是NX，意思是SET IF NOT EXIST，即当key不存在时，我们进行set操作；若key已经存在，则不做任何操作；第四个为expx，这个参数我们传的是PX，意思是我们要给这个key加一个过期的设置，第五个为time，代表key的过期时间。

首先，set()加入了NX参数，可以保证如果已有key存在，则函数不会调用成功，也就是只有一个客户端能持有锁，满足互斥性。其次，由于我们对锁设置了过期时间，即使锁的持有者后续发生崩溃而没有解锁，锁也会因为到了过期时间而自动解锁（即key被删除），不会发生死锁。最后，因为我们将value赋值为requestId，代表加锁的客户端请求标识，那么在客户端在解锁的时候就可以进行校验是否是同一个客户端。

**错误示范：**setnx() + expire()方法：由于这是两条Redis命令，不具有原子性，如果程序在执行完setnx()之后突然崩溃，导致锁没有设置过期时间。那么将会发生死锁。

**解锁：**第一行代码，是一个简单的Lua脚本代码。第二行代码，将Lua代码传到jedis.eval()方法里，并使参数KEYS[1]赋值为lockKey，ARGV[1]赋值为requestId。eval()方法是将Lua代码交给Redis服务端执行，eval()方法可以确保原子性。

## 4 Redis的并发竞争问题

方案一：可以使用独占锁的方式，类似操作系统的mutex机制。（网上有例子，http://blog.csdn.net/black\_ox/article/details/48972085 不过实现相对复杂，成本较高）。

方案二：使用乐观锁的方式进行解决（成本较低，非阻塞，性能较高）。本质上是假设不会进行冲突，使用redis的命令watch进行构造条件。本质上是假设不会进行冲突，使用redis的命令watch进行构造条件。伪代码如下：

|  |
| --- |
| watch price  get price $price  $price = $price + 10  multi  set price $price  exec |

在WATCH执行之后，EXEC执行之前，有其他客户端修改了key 的值，那么当前客户端的事务就会失败。

## 5 Redis持久化

**几种实现方式？**Redis支持两种方式的持久化：①RDB：RDB持久化是把当前进程数据生成快照保存到硬盘的过程。可以**在指定的时间间隔内生成内存中整个数据集的持久化快照**。快照文件默认被存储在当前文件夹中，名称为dump.rdb，可以通过dir和dbfilename参数来修改默认值。②AOF，AOP（append only file）将Reids的操作日志以追加的方式写入文件，解决了数据持久化的实时性。开启AOF功能需要设置配置：appendonly yes，默认是不开启。AOF文件名通过appendfilename配置设置，默认文件名是：appendonly.aof。

**RDB存在哪些优势呢？**①RDB是一个紧凑压缩的二进制文件（使用LZF算法压缩），代表Redis在某个时间点上的数据快照，非常适用于备份；②Redis加载RDB恢复数据远远快于AOF的方式。

**RDB缺点？**①RDB是间隔一段时间进行持久化，没法做到实时持久化，如果持久化之间redis发生故障，会发生数据丢失。所以这种方式更适合数据要求不严谨的时候。②如果当数据集较大时，可能会导致整个服务器停止服务几百毫秒，甚至是1秒钟。

**AOF存在哪些优势呢？**①可以带来更高的数据安全性，可以设置不同的 fsync 策略，Redis中提供了3中同步策略，即每秒同步、每修改同步和不同步。②由于该机制对日志文件的写入操作采用的是append模式，因此在写入过程中即使出现宕机现象，也不会破坏日志文件中已经存在的内容。③如果日志过大，Redis可以自动启用重写： 重写后的新 AOF 文件包含了恢复当前数据集所需的最小命令集合。

AOF缺点？①对于相同数量的数据集而言，AOF文件通常要大于RDB文件。RDB 在恢复大数据集时的速度比 AOF 的恢复速度要快。②根据同步策略的不同，AOF在运行效率上往往会慢于RDB。

**怎么选择？**做备份：当数据量大，且对恢复速度有要求，并且数据的一致性要求不高的话，可以只使用RDB；②只做缓存：不用开启任何的持久化方式；③两者都开启的建议。

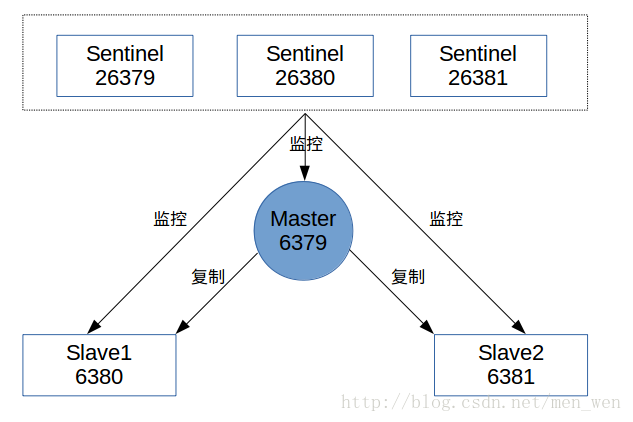
**怎么实现的？**在配置文件中修改：

|  |
| --- |
| **save 900 1**  #900秒内如果有超过1个key被修改则发起保存快照  **save 300 10**  #300秒内如果有超过10个key被修改则发起保存快照  **save 60 10000** #60秒内如果有超过1000个key被修改则发起保存快照 |
| appendonly yes #启用AOF方式  appendfsync always #每次有新命令追加到 AOF 文件时就执行一次 fsync  appendfsync everysec #每秒 fsync 一次：足够快，并且在故障时只会丢失 1 秒钟的数据  appendfsync no #从不 fsync ：将数据交给操作系统来处理。更快，也更不安全 |

## 6 Redis集群，高可用，缓存分片

**(1) Redis Sentinel([ˈsentɪnl],哨兵)的高可用性**：Redis Sentinel是一个分布式架构，包含若干个Sentinel节点和Redis数据节点，每个sentinel节点会对数据节点和其余Sentinel节点进行监控，，当发现节点不可达时，会对节点做下线标识。当大多数sentinel节点都认为主节点不可达时，会选举出一个Sentinel节点来完成自动故障转移的工作。

Sentinel用于管理多个redis服务器，主要执行三个任务：监控、提醒、自动故障转移。



**(2) Redis Cluster：**Redis 集群是一个提供在多个Redis间节点间共享数据的程序集，(节点数量为6个才能保证组成完整高可用的集群)集群使用了主从复制模型。

**创建集群：** redis-trib.rb create –replicas 1 127.0.0.1:7031 127.0.0.1:7032 127.0.0.1:7033 127.0.0.1:7034 127.0.0.1:7035 127.0.0.1:7036

其中：replicas参数指定集群中每个主节点配备几个从节点，这里配置为1.

**连接集群：redis-cli -c -h 127.0.0.1 -p 7031**

**新增节点：添加节点；**分配槽；指定从节点；

**删除节点：**将这个节点的数据重新分片到其他主节点上；删除

(3) Redis cluster采用虚拟槽分区，所有的键根据哈希函数映射到0~16383整数槽内，

计算公式：**CRC16(key)&16383**，每个节点负责维护一部分槽以及槽所映射的键值数据。

集群功能限制：①key批量操作支持有限，如mset、mget，目前只支持具有相同slot值的key执行批量操作；②key事务操作支持有限；③key作为数据分区的最小粒度，因此不能将一个大的键值对象如hash、list等映射到不同的节点；④不支持多数据库空间；⑤复制结构只支持一层，从节点只能复制主节点。

## 7 Redis的缓存失效策略

(1) 影响生存时间的一些操作

①DEL 命令来删除整个 key 来移除，或者被 SET 和 GETSET 命令覆盖原来的数据；

②执行EXPIRE命令，新指定的生存时间会取代旧的生存时间。

(2) redis 内存数据集大小上升到一定大小的时候，就会实行数据淘汰策略。

①volatile-lru：从已设置过期时间的数据集中挑选最近最少使用的数据淘汰；

②volatile-ttl：从已设置过期时间的数据集中挑选将要过期的数据淘汰；

③volatile-random：从已设置过期时间的数据集中任意选择数据淘汰

④allkeys-lru：从数据集中挑选最近最少使用的数据淘汰

⑤allkeys-random：从数据集中任意选择数据淘汰

⑥no-enviction（驱逐）：禁止驱逐数据

## 8 redis怎么扩容

## 9 redis主从怎么同步

## 19 redis高可用，不丢失消息

# 数据库

## 1分页优化

第一种简单粗暴，就是不允许查看这么靠后的数据；

第二种方法，在查询下一页时把上一页的行id作为参数传递给客户端程序，然后sql就改成了：select \* from table where id>3000000 limit 10;

如果主键id是自增的，并且中间没有删除和断点，那么还有一种方式，比如100页的10条数据：select \* from table where id>100\*10 limit 10;

第三种：select table.\* from table inner join ( select id from table limit 3000000,10 ) as tmp on tmp.id=table.id;

## 2悲观锁&乐观锁

**悲观锁：**排它锁，当事务在操作数据时把这部分数据进行锁定，直到操作完毕后再解锁，其他事务操作才可操作该部分数据。这将防止其他进程读取或修改表中的数据。

一般使用 select ...for update操作来实现悲观锁。当数据库执行select for update时会获取被select中的数据行的行锁，获取的行锁会在当前事务结束时自动释放。

如果加锁的时间过长，其他用户长时间无法访问，影响了程序的并发访问性，同时这样对数据库性能开销影响也很大

**乐观锁：**先进行业务操作，不到万不得已不去拿锁。在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，更新被拒绝的，可以让用户重新操作。

大多数基于**数据版本（Version）**记录机制实现(或者时间戳)：一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，version值加一。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据。

update t\_goods set status=2,version=version+1 where id=#{id} and version=#{version};

## **3 表锁&行锁**

|  |  |  |
| --- | --- | --- |
|  | 行锁 | 表锁 |
| MyISAM |  | √ |
| InnoDB | √ | √ |

(1) MyISAM表锁

**表锁：**MyISAM存储引擎只支持表锁。MySQL表级锁有两种模式：表共享锁（Table Read Lock）和表独占写锁（Table Write Lock）。**对MyISAM的读操作，不阻塞读，阻塞写；对MyISAM的写操作，则会阻塞读和写。**

**如何加表锁：**MyISAM在执行查询语句（SELECT）前，会自动加读锁，在执行更新操作（UPDATE、DELETE、INSERT等）前，会自动加写锁。

**(2) InnoDB的行锁模式及加锁方法**

InnoDB实现了两种行锁：共享锁(S)和排它锁(X)；另外，为了允许行锁和表锁共存，实现多粒度锁机制，还有两种内部使用的意向锁（Intention Locks），是InnoDB自动加的，不需用户干预。意向锁都是表锁：意向共享锁(IS)和意向排它锁(IX)。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 当前/请求锁模式 | X | IX | S | IS |
| X | 冲突 | 冲突 | 冲突 | 冲突 |
| IX | 冲突 | 兼容 | 冲突 | 兼容 |
| S | 冲突 | 冲突 | 兼容 | 兼容 |
| IS | 冲突 | 兼容 | 兼容 | 兼容 |

**(3)间隙锁（Next-Key锁）**

当用范围条件而不是相等条件检索数据，并请求共享或排他锁时，InnoDB会给符合条件的已有数据的索引项加锁；对于键值在条件范围内但并不存在的记录，叫做“间隙(GAP)”，InnoDB也会对这个“间隙”加锁，这种锁机制是所谓的间隙锁（Next-Key锁）。间隙锁的主要作用是为了防止出现幻读，但是它会把锁定范围扩大。

间隙锁的出现主要集中在同一个事务中先delete 后 insert的情况下，当我们通过一个参数去删除一条记录的时候，如果参数在数据库中存在， 那么这个时候产生的是普通行锁， 锁住这个记录，然后删除，然后释放锁。如果这条记录不存在，数据库会扫描索引，发现这个记录不存在，这个时候的delete语句获取到的就是一个间隙锁，然后数据库会向左扫描扫到第一个比给定参数小的值，向右扫描到第一个比给定参数大的值，然后以此为界，构建一个区间，锁住整个区间内的数据。

## **4** 组合索引，最左原则

**需要创建索引场景？**

①主键自动建立唯一索引

②频繁作为查询条件的字段，查询中排序或者分组的字段，应该创建索引

③查询中与其他表关联的字段，外键关系建立索引

④频繁更新的字段不适合建立索引

⑤单键/组合索引的选择问题，who?(在高并发下倾向创建组合索引)

**最左原则？**如果想使用索引，必须保证按索引的最左边前缀来进行查询。如果where条件中只用到索引项，则加的是行锁；否则加的是表锁。

①匹配全值：对索引中的所有列进行精确匹配，索引可以被用到。

②匹配最左前缀：查询条件精确匹配索引的左边连续一个或几个列，索引可以被用到。

**失效情况：**

①如果条件中有or，索引失效；

②like查询是以%开头，索引失效；

③如果列类型是字符串，查询条件中没有使用引号引起来，索引失效；

④一直向右匹配直到遇到范围查询(>、<、between、like)就停止匹配

⑤查询条件中使用函数，索引失效；

⑥对于多列索引，不是使用的第一部分，索引失效；

## 5 事务的特性和隔离级别

**事务具有4属性，通常称为事务的ACID属性**：

①原性性（Actomicity）：事务是一个原子操作单元，其对数据的修改，要么全都执行，要么全都不执行。

②一致性（Consistent）：在事务开始和完成时，数据都必须保持一致状态。这意味着所有相关的数据规则都必须应用于事务的修改，以操持完整性；事务结束时，所有的内部数据结构（如B树索引或双向链表）也都必须是正确的。

③隔离性（Isolation）：数据库系统提供一定的隔离机制，保证事务在不受外部并发操作影响的“独立”环境执行。这意味着事务处理过程中的中间状态对外部是不可见的，反之亦然。

④持久性（Durable）：事务完成之后，它对于数据的修改是永久性的，即使出现系统故障也能够保持。

**事务隔离级别**

**①Read uncommitted (未提交读)：**一个事务可以读取另一个未提交事务的数据。这也被称为脏读(dirty read)。

**②Read Committed(提交读)：**一个事务只能读取数据库中已经提交过的数据，解决了脏读问题，但不能重复读，即一个事务内的两次查询返回的数据是不一样的。

**③Repeatable Read(可重复读)：**Mysql默认的隔离级别。保证同一事务中多次读取同样记录的结果是一致的。但别的事务新增数据也能读取到，即但不能避免幻读的问题

**④Serializable(可串行化)：**读取的每一行数据上都加上共享锁和排他锁。

|  |  |  |  |
| --- | --- | --- | --- |
| 隔离级别 | 脏读 | 不可重读的 | 幻读 |
| Read uncommitted | √ | √ | √ |
| Read Committed | × | √ | √ |
| Repeatable Read | × | × | √ |
| Serializable | × | × | × |

## 6 myisam & innodb区别

①InnoDB支持事物，而MyISAM不支持事物

②InnoDB支持行级锁，而MyISAM支持表级锁

③InnoDB支持外键，而MyISAM不支持

④InnoDB不支持全文索引，而MyISAM支持

⑤InnoDB 中不保存表的具体行数，要扫描一遍整个表来计算；MyISAM保存行数。

⑥MyISAM的索引和数据是分开的，并且索引是有压缩的，而Innodb是索引和数据是紧密捆绑的，没有使用压缩，从而会造成Innodb比MyISAM体积庞大不小。

## 7 分表分库

## 8 性能优化

## 9 索引分类

## 10 b树，b+树，红黑树

有什么不同，MySQL为什么用b+树索引

：B+，hash；什么情况用什么索引