2美团 Java 面试 154 道题分享！

题目摘抄自-公众号：Java基基

我是程序员的

源码精品专栏 来源：程序员面试

[一、Java集合22题](#header-n6)   
 [1. ArrayList 和 Vector 的区别。。](#header-n7)  
 [2. 快速失败 (fail-fast) 和安全失败 (fail-safe) 的区别是什么？](#header-n29)   
 [3. hashmap 的数据结构。](#header-n82)   
 [4. HashMap 的工作原理是什么?](#header-n188)   
 [4. HashMap常用方法](#header-n281)  
 [5. Hashmap 什么时候进行扩容呢？](#header-n346)   
 [6. List、Map、Set 三个接口，存取元素时，各有什么特点？](#header-n439)   
 [7.Set 里的元素是不能重复的，那么用什么方法来区分重复与否呢? 是用 == 还是 equals()? 它们有何区别?](#header-n464)  
 [8.两个对象值相同 (x.equals(y) == true)，但却可有不同的 hash code，这句话对不对?](#header-n535)   
 [9.heap 和 stack 有什么区别。](#header-n593)   
 [10.Java 集合类框架的基本接口有哪些？](#header-n613)   
 [11.HashSet 和 TreeSet 有什么区别？](#header-n633)   
 [12.HashSet 的底层实现是什么?](#header-n649)   
 [13.LinkedHashMap 的实现原理?](#header-n687)   
 [14.为什么集合类没有实现 Cloneable 和 Serializable 接口？](#header-n735)   
 [15.什么是迭代器 (Iterator)？](#header-n764)   
 [16.Iterator 和 ListIterator 的区别是什么？](#header-n828)   
 [17.数组 (Array) 和列表 (ArrayList) 有什么区别？什么时候应该使用 Array 而不是 ArrayList？](#header-n853)   
 [18.Java 集合类框架的最佳实践有哪些？](#header-n876)  
 [19.Set 里的元素是不能重复的，那么用什么方法来区分重复与否呢？是用 == 还是 equals()？它们有何区别？](#header-n884)   
 [20.Comparable 和 Comparator 接口是干什么的？列出它们的区别](#header-n886)   
 [21.Collection 和 Collections 的区别。](#header-n925)  
 [22.说说 ArrayList,Vector, LinkedList 的存储性能和特性](#header-n938)  
 [二、JVM与调优21题](#header-n940)  
 [1、Java 类加载过程？](#header-n941)  
 [2.描述一下 JVM 加载 Class 文件的原理机制?](#header-n984)   
 [3.Java 内存分配。](#header-n1005)   
 [4.GC 是什么? 为什么要有 GC？](#header-n1383)   
 [5.简述 Java 垃圾回收机制](#header-n1403)   
 [6.如何判断一个对象是否存活？（或者 GC 对象的判定方法）](#header-n1458)   
 [7.垃圾回收的优点和原理。并考虑 2 种回收机制](#header-n1497)   
 [8.垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收？](#header-n1525)   
 [9.Java 中会存在内存泄漏吗，请简单描述](#header-n1533)   
 [10.深拷贝和浅拷贝。](#header-n1543)  
 [11.System.gc() 和 Runtime.gc() 会做什么事情？](#header-n1579)   
 [12.finalize() 方法什么时候被调用？析构函数 (finalization) 的目的是什么？](#header-n1591)   
 [13.如果对象的引用被置为 null，垃圾收集器是否会立即释放对象占用的内存？](#header-n1616)   
 [14.什么是分布式垃圾回收（DGC）？它是如何工作的？](#header-n1632)   
 [15.串行（serial）收集器和吞吐量（throughput）收集器的区别是什么？](#header-n1654)   
 [16.在 Java 中，对象什么时候可以被垃圾回收？](#header-n1674)   
 [17.简述 Java 内存分配与回收策率以及 Minor GC 和 Major GC。](#header-n1790)  
 [18.JVM 的永久代中会发生垃圾回收么？](#header-n1803)   
 [19.Java 中垃圾收集的方法有哪些？](#header-n1814)   
 [20.什么是类加载器，类加载器有哪些？](#header-n1849)   
 [21.类加载器双亲委派模型机制？](#header-n1983)  
 [三、并发编程28题](#header-n2035)  
 [1.Synchronized 用过吗，其原理是什么？](#header-n2036)   
 [2.你刚才提到获取对象的锁，这个“锁”到底是什么？如何确定对象的锁？](#header-n2085)   
 [3.什么是可重入性，为什么说 Synchronized 是可重入锁？](#header-n2159)  
 [4.JVM 对 Java 的原生锁做了哪些优化？](#header-n2175)   
 [5.为什么说 Synchronized 是非公平锁？](#header-n2193)   
 [6.什么是锁消除和锁粗化？](#header-n2221)   
 [7.为什么说 Synchronized 是一个悲观锁？乐观锁的实现原理又是什么？什么是 CAS，它有什么特性？](#header-n2329)   
 [8.乐观锁一定就是好的吗？](#header-n2354)   
 [9.跟 Synchronized 相比，可重入锁 ReentrantLock 其实现原理有什么不同？](#header-n2375)   
 [10.那么请谈谈 AQS 框架是怎么回事儿？](#header-n2529)   
 [11.请尽可能详尽地对比下 Synchronized 和 ReentrantLock 的异同。](#header-n2569)   
 [12.ReentrantLock 是如何实现可重入性的？](#header-n2606)   
 [13.除了 ReetrantLock，你还接触过 JUC 中的哪些并发工具？](#header-n2639)   
 [14.请谈谈 ReadWriteLock 和 StampedLock。](#header-n2640)   
 [15.如何让 Java 的线程彼此同步？你了解过哪些同步器？请分别介绍下。](#header-n2641)   
 [16.CyclicBarrier 和 CountDownLatch 看起来很相似，请对比下呢？](#header-n2642)   
 [17.Java 线程池相关问题](#header-n2643)   
 [18.Java 中的线程池是如何实现的？](#header-n2644)   
 [19.创建线程池的几个核心构造参数？](#header-n2645)   
 [20.线程池中的线程是怎么创建的？是一开始就随着线程池的启动创建好的吗？](#header-n2646)   
 [21.既然提到可以通过配置不同参数创建出不同的线程池，那么 Java 中默认实现好的线程池又有哪些呢？请比较它们的异同](#header-n2647)   
 [22.如何在 Java 线程池中提交线程？](#header-n2648)   
 [23.什么是 Java 的内存模型，Java 中各个线程是怎么彼此看到对方的变量的？](#header-n2649)   
 [24.请谈谈 volatile 有什么特点，为什么它能保证变量对所有线程的可见性？](#header-n2650)   
 [25.既然 volatile 能够保证线程间的变量可见性，是不是就意味着基于 volatile 变量的运算就是并发安全的？](#header-n2651)   
 [26.请对比下 volatile 对比 Synchronized 的异同。](#header-n2652)   
 [27.请谈谈 ThreadLocal 是怎么解决并发安全的？](#header-n2653)   
 [28.很多人都说要慎用 ThreadLocal，谈谈你的理解，使用 ThreadLocal 需要注意些什么？](#header-n2654)   
 [附加：29synchronized与lock 对象锁、互斥锁、共享锁以及公平锁和非公平锁](#header-n2655)  
 [四、spring 25题](#header-n2675)  
 [1.什么是 Spring 框架？Spring 框架有哪些主要模块？](#header-n2676)   
 [2.使用 Spring 框架能带来哪些好处？](#header-n2677)   
 [3.什么是控制反转(IOC)？什么是依赖注入？](#header-n2678)   
 [4.请解释下 Spring 框架中的 IoC？](#header-n2679)   
 [5.BeanFactory 和 ApplicationContext 有什么区别？](#header-n2680)   
 [6.Spring 有几种配置方式？](#header-n2681)   
 [7.如何用基于 XML 配置的方式配置 Spring？](#header-n2682)   
 [8.如何用基于 Java 配置的方式配置 Spring？](#header-n2683)   
 [9.怎样用注解的方式配置 Spring？](#header-n2684)   
 [10.请解释 Spring Bean 的生命周期？](#header-n2685)   
 [11.Spring Bean 的作用域之间有什么区别？](#header-n2686)   
 [12.什么是 Spring inner beans？](#header-n2687)   
 [13.Spring 框架中的单例 Beans 是线程安全的么？](#header-n2688)   
 [14.请举例说明如何在 Spring 中注入一个 Java Collection？](#header-n2689)   
 [15.如何向 Spring Bean 中注入一个 Java.util.Properties？](#header-n2690)   
 [16.请解释 Spring Bean 的自动装配？](#header-n2691)   
 [17.请解释自动装配模式的区别？](#header-n2692)   
 [18.如何开启基于注解的自动装配？](#header-n2693)   
 [19.请举例解释@Required 注解？](#header-n2694)   
 [20.请举例解释@Autowired 注解？](#header-n2695)   
 [21.请举例说明@Qualifier 注解？](#header-n2696)   
 [22.构造方法注入和设值注入有什么区别？](#header-n2697)   
 [23.Spring 框架中有哪些不同类型的事件？](#header-n2698)   
 [24.FileSystemResource 和 ClassPathResource 有何区别？](#header-n2699)   
 [25.Spring 框架中都用到了哪些设计模式？](#header-n2700)   
 [五、设计模式 10题](#header-n2701)   
 [1.请列举出在 JDK 中几个常用的设计模式？](#header-n2702)   
 [2.什么是设计模式？你是否在你的代码里面使用过任何设计模式？](#header-n2703)   
 [3.Java 中什么叫单例设计模式？请用 Java 写出线程安全的单例模式](#header-n2704)   
 [4.在 Java 中，什么叫观察者设计模式（observer design pattern）？](#header-n2705)   
 [5.使用工厂模式最主要的好处是什么？在哪里使用？](#header-n2706)   
 [6.举一个用 Java 实现的装饰模式(decorator design pattern)？它是作用于对象层次还是类层次？](#header-n2707)   
 [7.在 Java 中，为什么不允许从静态方法中访问非静态变量？](#header-n2708)   
 [8.设计一个 ATM 机，请说出你的设计思路？](#header-n2709)   
 [9.在 Java 中，什么时候用重载，什么时候用重写？](#header-n2710)   
 [10.举例说明什么情况下会更倾向于使用抽象类而不是接口](#header-n2711)   
 [六、springboot 22题](#header-n2712)  
 [1.什么是 Spring Boot？](#header-n2713)   
 [2.Spring Boot 有哪些优点？](#header-n2714)   
 [3.什么是 JavaConfig？](#header-n2715)   
 [4.如何重新加载 Spring Boot 上的更改，而无需重新启动服务器？](#header-n2716)   
 [5.Spring Boot 中的监视器是什么？](#header-n2717)   
 [6.如何在 Spring Boot 中禁用 Actuator 端点安全性？](#header-n2718)   
 [7.如何在自定义端口上运行 Spring Boot 应用程序？](#header-n2719)   
 [8.什么是 YAML？](#header-n2720)   
 [9.如何实现 Spring Boot 应用程序的安全性？](#header-n2721)   
 [10.如何集成 Spring Boot 和 ActiveMQ？](#header-n2722)   
 [11.如何使用 Spring Boot 实现分页和排序？](#header-n2723)   
 [12.什么是 Swagger？你用 Spring Boot 实现了它吗？](#header-n2724)   
 [13.什么是 Spring Profiles？](#header-n2725)   
 [14.什么是 Spring Batch？](#header-n2726)   
 [15.什么是 FreeMarker 模板？](#header-n2727)   
 [16.如何使用 Spring Boot 实现异常处理？](#header-n2728)   
 [17.您使用了哪些 starter maven 依赖项？](#header-n2729)   
 [18.什么是 CSRF 攻击？](#header-n2730)   
 [19.什么是 WebSockets？](#header-n2731)   
 [20.什么是 AOP？](#header-n2732)   
 [21.什么是 Apache Kafka？](#header-n2733)   
 [22.我们如何监视所有 Spring Boot 微服务？](#header-n2734)   
 [七、Redis 16题](#header-n2736)   
 [1.什么是redis?](#header-n2737)   
 [2.Reids的特点](#header-n2738)   
 [3.Redis支持的数据类型](#header-n2739)   
 [4.Redis是单进程单线程的](#header-n2740)   
 [5.redis的虚拟内存](#header-n2741)   
 [6.Redis锁](#header-n2742)   
 [7.读写分离模型](#header-n2743)   
 [8.数据分片模型](#header-n2744)   
 [9.Redis的回收策略](#header-n2745)   
 [10.使用Redis有哪些好处？](#header-n2746)   
 [11.redis相比memcached有哪些优势？](#header-n2747)   
 [12.redis常见性能问题和解决方案](#header-n2748)   
 [13.MySQL里有2000w数据，redis中只存20w的数据，如何保证redis中的数据都是热点数据](#header-n2749)   
 [14.Memcache与Redis的区别都有哪些？](#header-n2750)   
 [15.Redis 常见的性能问题都有哪些？如何解决？](#header-n2751)   
 [16.Redis 最适合的场景](#header-n2752)   
 [八、Netty10题](#header-n2753)  
 [1.BIO、NIO和AIO的区别？](#header-n2754)   
 [2.NIO的组成？](#header-n2755)   
 [3.Netty的特点？](#header-n2756)   
 [4.Netty的线程模型？](#header-n2757)   
 [5.TCP 粘包/拆包的原因及解决方法？](#header-n2758)   
 [6.了解哪几种序列化协议？](#header-n2759)   
 [7.如何选择序列化协议？](#header-n2760)   
 [8.Netty的零拷贝实现？](#header-n2761)   
 [9.Netty的高性能表现在哪些方面？](#header-n2762)   
 [10.NIOEventLoopGroup源码？](#header-n2763)

## 一、Java集合22题

### 1. ArrayList 和 Vector 的区别。。

**同题目-22.说说 ArrayList,Vector, LinkedList 的存储性能和特性**

答：[博客地址](https://www.cnblogs.com/williamjie/p/11158523.html)   
 首先看三个类都实现List接口，而List接口一共有三个实现类，分别是ArrayList、Vector和LinkedList。List用于存放多个元素，能够维护元素的次序，并且允许元素的重复。   
 3个具体实现类的相关区别如下：

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，就要将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。
3. LinkedList是用链表结构存储数据的，很适合数据的动态插入和删除，随机访问和遍历速度比较慢。另外，他还提供了List接口中没有定义的方法，专门用于操作表头和表尾元素，可以当作堆栈、队列和双向队列使用。
4. vector是线程（Thread）同步（Synchronized）的，所以它也是线程安全的，而Arraylist是线程异步（ASynchronized）的，是不安全的。如果不考虑到线程的安全因素，一般用Arraylist效率比较高。

**场景考虑：**

* 如果集合中的元素的数目大于目前集合数组的长度时，vector增长率为目前数组长度的100%,而arraylist增长率为目前数组长度的50%.如过在集合中使用数据量比较大的数据，用vector有一定的优势。
* 如果查找一个指定位置的数据，vector和arraylist使用的时间是相同的，都是0(1),这个时候使用vector和arraylist都可以。而如果移动一个指定位置的数据花费的时间为0(n-i),n为总长度，这个时候就应该考虑到使用Linkedlist,因为它移动一个指定位置的数据所花费的时间为0(1),而查询一个指定位置的数据时花费的时间为0(i)。
* ArrayList 和Vector是采用数组方式存储数据，此数组元素数大于实际存储的数据以便增加和插入元素，都允许直接序号索引元素，但是插入数据要设计到数组元素移动 等内存操作，所以索引数据快插入数据慢，
* Vector由于使用了synchronized方法（线程安全）所以性能上比ArrayList要差，LinkedList使用双向链表实现存储，按序号索引数据需要进行向前或向后遍历，但是插入数据时只需要记录本项的前后项即可，所以插入数度较快！  
  笼统来说：LinkedList：增删改快； ArrayList：查询快（有索引的存在）

### 2. 快速失败 (fail-fast) 和安全失败 (fail-safe) 的区别是什么？

[论坛地址 ](https://www.cnblogs.com/shamo89/p/6685216.html)   
欢迎访问个人独立博客，http://791202.com/，分享java开发干货   
快速失败(fail-fast)和安全失败(fail-safe)的区别

主要分析一下几点

1. fail-fast和fail-safe比较
2. 如何解决fail-fast
3. fail-fast原理
4. 解决fail-fast的原理
5. fail-fast和fail-safe比较  
   **Iterator 的安全失败是基于对底层集合做拷贝**，因此，它不受源集合上修改的影响。

* java.util 包下面的所有的集合类都是**快速失败**的，而 java.util.concurrent 包下面的所有的类都是**安全失败**的。**快速失败的迭代器**会抛出 ConcurrentModificationException 异常，而**安全失败的迭代器**永远不会抛出这样的异常。
* 快速失败示例

import java.util.\*;  
 import java.util.concurrent.\*;  
   
 /\*  
 \* @desc java集合中Fast-Fail的测试程序。  
 \*  
 \* fast-fail事件产生的条件：当多个线程对Collection进行操作时，  
 若其中某一个线程通过iterator去遍历集合时，该集合的内容被其他线程所改变；  
 则会抛出ConcurrentModificationException异常。  
 \* fast-fail解决办法：通过util.concurren t集合包下的相应类去处理，则不会产生fast-fail事件。  
 \*  
 \* 本例中，分别测试ArrayList和CopyOnWriteArrayList这两种情况。  
 ArrayList会产生fast-fail事件，而CopyOnWriteArrayList不会产生fast-fail事件。  
 \* (01) 使用ArrayList时，会产生fast-fail事件，抛出ConcurrentModificationException异常；定义如下：  
 \* private static List<String> list = new ArrayList<String>();  
 \* (02) 使用时CopyOnWriteArrayList，不会产生fast-fail事件；定义如下：  
 \* private static List<String> list = new CopyOnWriteArrayList<String>();  
 \*  
 \* @author skywang  
 \*/  
 public class FastFailTest {  
   
 private static List<String> list = new ArrayList<String>();  
 //private static List<String> list = new CopyOnWriteArrayList<String>();  
 public static void main(String[] args) {  
   
 // 同时启动两个线程对list进行操作！  
 new ThreadOne().start();  
 new ThreadTwo().start();  
 }  
   
 private static void printAll() {  
 System.out.println("");  
   
 String value = null;  
 Iterator iter = list.iterator();  
 while(iter.hasNext()) {  
 value = (String)iter.next();  
 System.out.print(value+", ");  
 }  
 }  
   
 /\*\*  
 \* 向list中依次添加0,1,2,3,4,5，每添加一个数之后，就通过printAll()遍历整个list  
 \*/  
 private static class ThreadOne extends Thread {  
 public void run() {  
 int i = 0;  
 while (i<6) {  
 list.add(String.valueOf(i));  
 printAll();  
 i++;  
 }  
 }  
 }  
   
 /\*\*  
 \* 向list中依次添加10,11,12,13,14,15，每添加一个数之后，就通过printAll()遍历整个list  
 \*/  
 private static class ThreadTwo extends Thread {  
 public void run() {  
 int i = 10;  
 while (i<16) {  
 list.add(String.valueOf(i));  
 printAll();  
 i++;  
 }  
 }  
 }  
 }

1. 如何解决

**fail-fast机制**，是一种**错误检测机制**。   
 它只能被用来检测错误，因为JDK并不保证fail-fast机制一定会发生。若在多线程环境下使用fail-fast机制的集合，建议**使用“java.util.concurrent包下的类”去取代“java.util包下的类”**。  
 所以，本例中只需要将ArrayList替换成java.util.concurrent包下对应的类即可。  
 即，将代码

private static List<String> list = new ArrayList<String>();

替换为

private static List<String> list = new CopyOnWriteArrayList<String>();

则可以解决该办法。

1. fail-fast原理  
   产生fail-fast事件，是通过抛出ConcurrentModificationException异常来触发的。  
   那么，**ArrayList是如何抛出ConcurrentModificationException异常的呢**?  
   我们知道，ConcurrentModificationException是在操作Iterator时抛出的异常。我们先看看Iterator的源码。ArrayList的Iterator是在父类AbstractList.java中实现的。   
   代码如下：

package java.util;  
   
 public abstract class AbstractList<E> extends AbstractCollection<E> implements List<E> {  
   
 ...  
   
 // AbstractList中唯一的属性  
 // 用来记录List修改的次数：每修改一次(添加/删除等操作)，将modCount+1  
 protected transient int modCount = 0;  
   
 // 返回List对应迭代器。实际上，是返回Itr对象。  
 public Iterator<E> iterator() {  
 return new Itr();  
 }  
   
 // Itr是Iterator(迭代器)的实现类  
 private class Itr implements Iterator<E> {  
 int cursor = 0;  
   
 int lastRet = -1;  
   
 // 修改数的记录值。  
 // 每次新建Itr()对象时，都会保存新建该对象时对应的modCount；  
 // 以后每次遍历List中的元素的时候，都会比较expectedModCount和modCount是否相等；  
 // 若不相等，则抛出ConcurrentModificationException异常，产生fail-fast事件。  
 int expectedModCount = modCount;  
   
 public boolean hasNext() {  
 return cursor != size();  
 }  
   
 public E next() {  
 // 获取下一个元素之前，都会判断“新建Itr对象时保存的modCount”和“当前的modCount”是否相等；  
 // 若不相等，则抛出ConcurrentModificationException异常，产生fail-fast事件。  
 checkForComodification();  
 try {  
 E next = get(cursor);  
 lastRet = cursor++;  
 return next;  
 } catch (IndexOutOfBoundsException e) {  
 checkForComodification();  
 throw new NoSuchElementException();  
 }  
 }  
   
 public void remove() {  
 if (lastRet == -1)  
 throw new IllegalStateException();  
 checkForComodification();  
   
 try {  
 AbstractList.this.remove(lastRet);  
 if (lastRet < cursor)  
 cursor--;  
 lastRet = -1;  
 expectedModCount = modCount;  
 } catch (IndexOutOfBoundsException e) {  
 throw new ConcurrentModificationException();  
 }  
 }  
   
 final void checkForComodification() {  
 if (modCount != expectedModCount)  
 throw new ConcurrentModificationException();  
 }  
 }  
   
 ...  
 }

从中，我们可以发现在**调用 next() 和 remove()时**，都会执行 checkForComodification()。若 “modCount 不等于 expectedModCount”，则抛出ConcurrentModificationException异常，产生fail-fast事件。

要搞明白 fail-fast机制，我们就要需要理解什么时候“modCount 不等于 expectedModCount”！  
从Itr类中，我们知道 expectedModCount 在创建Itr对象时，被赋值为 modCount。通过Itr，我们知道：expectedModCount不可能被修改为不等于 modCount。所以，需要考证的就是**modCount何时会被修改**。

接下来，我们查看ArrayList的源码，来看看modCount是如何被修改的。

package java.util;  
   
 public class ArrayList<E> extends AbstractList<E>  
 implements List<E>, RandomAccess, Cloneable, java.io.Serializable  
 {  
   
 ...  
   
 // list中容量变化时，对应的同步函数  
 public void ensureCapacity(int minCapacity) {  
 // 修改modCount  
 modCount++;  
 int oldCapacity = elementData.length;  
 if (minCapacity > oldCapacity) {  
 Object oldData[] = elementData;  
 int newCapacity = (oldCapacity \* 3)/2 + 1;  
 if (newCapacity < minCapacity)  
 newCapacity = minCapacity;  
 // minCapacity is usually close to size, so this is a win:  
 elementData = Arrays.copyOf(elementData, newCapacity);  
 }  
 }  
   
   
 // 添加元素到队列最后  
 public boolean add(E e) {  
 // 修改modCount  
 ensureCapacity(size + 1); // Increments modCount!!  
 elementData[size++] = e;  
 return true;  
 }  
   
   
 // 添加元素到指定的位置  
 public void add(int index, E element) {  
 if (index > size || index < 0)  
 throw new IndexOutOfBoundsException(  
 "Index: "+index+", Size: "+size);  
   
 // 修改modCount  
 ensureCapacity(size+1); // Increments modCount!!  
 1 System.arraycopy(elementData, index, elementData, index + 1,  
 size - index);  
 elementData[index] = element;  
 size++;  
 }  
   
 // 添加集合  
 public boolean addAll(Collection<? extends E> c) {  
 Object[] a = c.toArray();  
 int numNew = a.length;  
 // 修改modCount  
 ensureCapacity(size + numNew); // Increments modCount  
 System.arraycopy(a, 0, elementData, size, numNew);  
 size += numNew;  
 return numNew != 0;  
 }  
   
   
 // 删除指定位置的元素   
 public E remove(int index) {  
 RangeCheck(index);  
   
 // 修改modCount  
 modCount++;  
 E oldValue = (E) elementData[index];  
   
 int numMoved = size - index - 1;  
 if (numMoved > 0)  
 System.arraycopy(elementData, index+1, elementData, index, numMoved);  
 elementData[--size] = null; // Let gc do its work  
   
 return oldValue;  
 }  
   
   
 // 快速删除指定位置的元素   
 private void fastRemove(int index) {  
   
 // 修改modCount  
 modCount++;  
 int numMoved = size - index - 1;  
 if (numMoved > 0)  
 System.arraycopy(elementData, index+1, elementData, index,  
 numMoved);  
 elementData[--size] = null; // Let gc do its work  
 }  
   
 // 清空集合  
 public void clear() {  
 // 修改modCount  
 modCount++;  
   
 // Let gc do its work  
 for (int i = 0; i < size; i++)  
 elementData[i] = null;  
   
 size = 0;  
 }  
   
 ...  
 }

从中，我们发现：<u>无论是add()、remove()，还是clear()，只要涉及到修改集合中的元素个数时，都会改变modCount的值</u>。

接下来，我们再系统的梳理一下fail-fast是怎么产生的。步骤如下：  
(01) 新建了一个ArrayList，名称为arrayList。  
(02) 向arrayList中添加内容。  
(03) 新建一个“线程a”，并在“线程a”中通过Iterator反复的读取arrayList的值。  
(04) 新建一个“线程b”，在“线程b”中删除arrayList中的一个“节点A”。  
(05) 这时，就会产生有趣的事件了。

在某一时刻，“线程a”创建了arrayList的Iterator。此时“节点A”仍然存在于arrayList中，创建arrayList时，expectedModCount = modCount(假设它们此时的值为1)。

在“线程a”在遍历arrayList过程中的某一时刻，“线程b”执行了，并且“线程b”删除了arrayList中的“节点A”。“线程b”执行remove()进行删除操作时，在remove()中执行了“modCount++”，此时modCount变成了2！

“线程a”接着遍历，当它执行到next()函数时，调用checkForComodification()比较“expectedModCount”和“modCount”的大小；而“expectedModCount=1”，“modCount=2”,这样，便抛出ConcurrentModificationException异常，产生fail-fast事件。

至此，我们就完全了解了fail-fast是如何产生的！

即，当多个线程对同一个集合进行操作的时候，某线程访问集合的过程中，该集合的内容被其他线程所改变(即其它线程通过add、remove、clear等方法，改变了modCount的值)；这时，就会抛出ConcurrentModificationException异常，产生fail-fast事件。

1. 解决fail-fast的原理  
   上面，说明了“解决fail-fast机制的办法”，也知道了“fail-fast产生的根本原因”。接下来，我们再进一步谈谈java.util.concurrent包中是如何解决fail-fast事件的。  
   还是以和ArrayList对应的CopyOnWriteArrayList进行说明。   
   我们先看看CopyOnWriteArrayList的源码：

package java.util.concurrent;  
 import java.util.\*;  
 import java.util.concurrent.locks.\*;  
 import sun.misc.Unsafe;  
   
 public class CopyOnWriteArrayList<E>  
 implements List<E>, RandomAccess, Cloneable, java.io.Serializable {  
   
 ...  
   
 // 返回集合对应的迭代器  
 public Iterator<E> iterator() {  
 return new COWIterator<E>(getArray(), 0);  
 }  
   
 ...  
   
 private static class COWIterator<E> implements ListIterator<E> {  
 private final Object[] snapshot;  
   
 private int cursor;  
   
 private COWIterator(Object[] elements, int initialCursor) {  
 cursor = initialCursor;  
 // 新建COWIterator时，将集合中的元素保存到一个新的拷贝数组中。  
 // 这样，当原始集合的数据改变，拷贝数据中的值也不会变化。  
 snapshot = elements;  
 }  
   
 public boolean hasNext() {  
 return cursor < snapshot.length;  
 }  
   
 public boolean hasPrevious() {  
 return cursor > 0;  
 }  
   
 public E next() {  
 if (! hasNext())  
 throw new NoSuchElementException();  
 return (E) snapshot[cursor++];  
 }  
   
 public E previous() {  
 if (! hasPrevious())  
 throw new NoSuchElementException();  
 return (E) snapshot[--cursor];  
 }  
   
 public int nextIndex() {  
 return cursor;  
 }  
   
 public int previousIndex() {  
 return cursor-1;  
 }  
   
 public void remove() {  
 throw new UnsupportedOperationException();  
 }  
   
 public void set(E e) {  
 throw new UnsupportedOperationException();  
 }  
   
 public void add(E e) {  
 throw new UnsupportedOperationException();  
 }  
 }  
   
 ...  
   
 }

从中，我们可以看出:

ArrayList和CopyOnWriteArrayList的区别  
 (01). 和ArrayList继承于AbstractList不同，CopyOnWriteArrayList没有继承于AbstractList，它仅仅只是实现了List接口。  
 (02). ArrayList的iterator()函数返回的Iterator是在AbstractList中实现的；而**CopyOnWriteArrayList是自己实现Iterator。**  
 (03) ArrayList的Iterator实现类中调用next()时，会“调用checkForComodification()比较‘expectedModCount’和‘modCount’的大小”；   
 但是，CopyOnWriteArrayList的Iterator实现类中，**没有所谓的checkForComodification()，更不会抛出ConcurrentModificationException异常**！

### 3. hashmap 的数据结构。

[网上引用地址](https://www.jianshu.com/p/003256ce41ce) -答案来源-https://www.jianshu.com/p/003256ce41ce

**数据结构解析-HashMap**   
AntCoding小脑斧  
字数 1,245 阅读 7,211  
概要

HashMap在JDK1.8之前的实现方式 数组+链表,但是在JDK1.8后对HashMap进行了底层优化,改为了由 数组+链表+红黑树实现,主要的目的是提高查找效率。

如图所示：

|  |  |  |  |
| --- | --- | --- | --- |
| JDK版本 | 实现方式 | 节点数>=8 | 节点数<=6 |
| 1.8以前 | 数组+单向链表 | 数组+单向链表 | 数组+单向链表 |
| 1.8以后 | 数组+单向链表+红黑树 | 数组+红黑树 | 数组+单向链表 |

**HashMap**

1. 继承关系

public class HashMap<K,V> extends AbstractMap<K,V>  
 implements Map<K,V>, Cloneable, Serializable

1. 常量&构造方法

* 1个无参构造函数和3个有参构函数

//这两个是限定值 当节点数大于8时会转为红黑树存储  
 static final int TREEIFY\_THRESHOLD = 8;  
 //当节点数小于6时会转为单向链表存储  
 static final int UNTREEIFY\_THRESHOLD = 6;  
 //红黑树最小长度为 64  
 static final int MIN\_TREEIFY\_CAPACITY = 64;  
 //HashMap容量初始大小  
 static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4; // aka 16  
 //HashMap容量极限  
 static final int MAXIMUM\_CAPACITY = 1 << 30;  
 //负载因子默认大小  
 static final float DEFAULT\_LOAD\_FACTOR = 0.75f;  
 //Node是Map.Entry接口的实现类  
 //在此存储数据的Node数组容量是2次幂  
 //每一个Node本质都是一个单向链表  
 transient Node<K,V>[] table;  
 //HashMap大小,它代表HashMap保存的键值对的多少  
 transient int size;  
 //HashMap被改变的次数  
 transient int modCount;  
 //下一次HashMap扩容的大小  
 int threshold;  
 //存储负载因子的常量  
 final float loadFactor;  
  
 //默认的构造函数  
 public HashMap() {  
 this.loadFactor = DEFAULT\_LOAD\_FACTOR; // all other fields defaulted  
 }  
 //指定容量大小  
 public HashMap(int initialCapacity) {  
 this(initialCapacity, DEFAULT\_LOAD\_FACTOR);  
 }  
 //指定容量大小和负载因子大小  
 public HashMap(int initialCapacity, float loadFactor) {  
 //指定的容量大小不可以小于0,否则将抛出IllegalArgumentException异常  
 if (initialCapacity < 0)  
 throw new IllegalArgumentException("Illegal initial capacity: " +  
 initialCapacity);  
 //判定指定的容量大小是否大于HashMap的容量极限  
 if (initialCapacity > MAXIMUM\_CAPACITY)  
 initialCapacity = MAXIMUM\_CAPACITY;  
 //指定的负载因子不可以小于0或为Null，若判定成立则抛出IllegalArgumentException异常  
 if (loadFactor <= 0 || Float.isNaN(loadFactor))  
 throw new IllegalArgumentException("Illegal load factor: " +  
 loadFactor);  
   
 this.loadFactor = loadFactor;  
 // 设置“HashMap阈值”，当HashMap中存储数据的数量达到threshold时，就需要将HashMap的容量加倍。  
 this.threshold = tableSizeFor(initialCapacity);  
 }  
 //传入一个Map集合,将Map集合中元素Map.Entry全部添加进HashMap实例中  
 public HashMap(Map<? extends K, ? extends V> m) {  
 this.loadFactor = DEFAULT\_LOAD\_FACTOR;  
 //此构造方法主要实现了Map.putAll()  
 putMapEntries(m, false);  
 }

1. Node单向链表的实现

* Node类实现了了Map.entry<k,V>

//实现了Map.Entry接口  
 static class Node<K,V> implements Map.Entry<K,V> {  
 final int hash;  
 final K key;  
 V value;  
 Node<K,V> next;  
 //构造函数  
 Node(int hash, K key, V value, Node<K,V> next) {  
 this.hash = hash;  
 this.key = key;  
 this.value = value;  
 this.next = next;  
 }  
  
 public final K getKey() { return key; }  
 public final V getValue() { return value; }  
 public final String toString() { return key + "=" + value; }  
  
 public final int hashCode() {  
 return Objects.hashCode(key) ^ Objects.hashCode(value);  
 }  
  
 public final V setValue(V newValue) {  
 V oldValue = value;  
 value = newValue;  
 return oldValue;  
 }  
 //equals属性对比  
 public final boolean equals(Object o) {  
 if (o == this)  
 return true;  
 if (o instanceof Map.Entry) {  
 Map.Entry<?,?> e = (Map.Entry<?,?>)o;  
 if (Objects.equals(key, e.getKey()) &&  
 Objects.equals(value, e.getValue()))  
 return true;  
 }  
 return false;  
 }  
 }

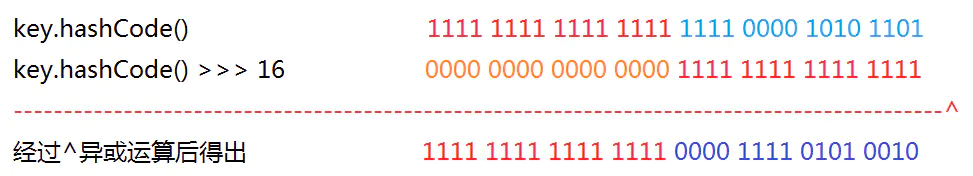
1. TreeNode红黑树实现

* TreeNode继承了LinkedHashMap.LinkedHashMapEntry<K,V>

static final class TreeNode<K,V> extends LinkedHashMap.LinkedHashMapEntry<K,V> {  
 TreeNode<K,V> parent; // 红黑树的根节点  
 TreeNode<K,V> left; //左树  
 TreeNode<K,V> right; //右树  
 TreeNode<K,V> prev; // 上一个几点  
 boolean red; //是否是红树  
 TreeNode(int hash, K key, V val, Node<K,V> next) {  
 super(hash, key, val, next);  
 }  
  
 /\*\*  
 \* 根节点的实现  
 \*/  
 final TreeNode<K,V> root() {  
 for (TreeNode<K,V> r = this, p;;) {  
 if ((p = r.parent) == null)  
 return r;  
 r = p;  
 }  
 }  
 ...

1. Hash的计算实现

//主要是将传入的参数key本身的hashCode与h无符号右移16位进行二进制异或运算得出一个新的hash值  
 static final int hash(Object key) {  
 int h;  
 return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
 }

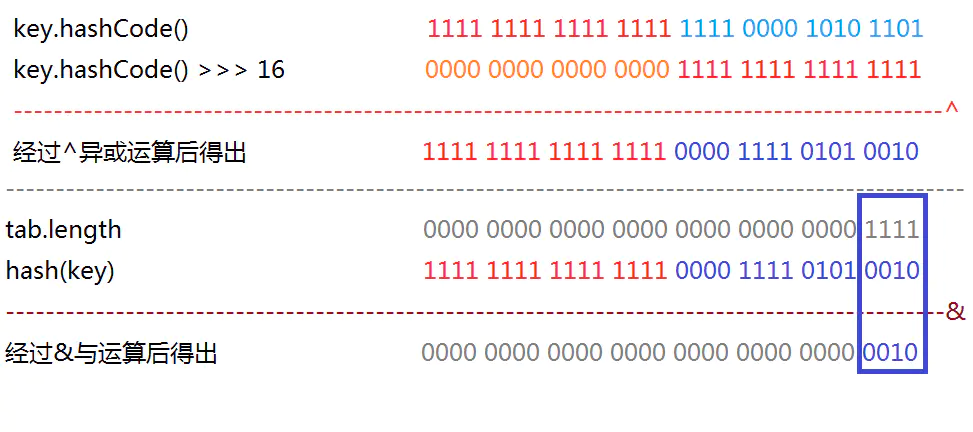
延伸讲解   
5.1. 下面的做了一个例子讲解,经过hash函数计算后得到的key的hash值  


5.2. 那为什么要这么做呢？   
 直接通过key.hashCode()获取hash不得了吗?为什么在右移16位后进行异或运算？

答案 : 与HashMap的table数组下计算标有关系

我们在下面讲解的put/get函数代码块中都出现了这样一段代码

//put函数代码块中  
tab[i = (n - 1) & hash])   
//get函数代码块中  
tab[(n - 1) & hash])

我们知道这段代码是根据索引得到tab中节点数据,它是如何与hash进行与运算后得到索引位置呢! 假设tab.length()=1<<4  
   
这样做的根本原因是当发生较大碰撞时也用树形存储降低了冲突。既减少了系统的开销

1. HashMap.put的源码实现

public V put(K key, V value) {  
 return putVal(hash(key), key, value, false, true);  
 }  
 //HashMap.put的具体实现  
 final V putVal(int hash, K key, V value, boolean onlyIfAbsent,  
 boolean evict) {  
 Node<K,V>[] tab; Node<K,V> p; int n, i;  
 //判定table不为空并且table长度不可为0,否则将从resize函数中获取  
 if ((tab = table) == null || (n = tab.length) == 0)  
 n = (tab = resize()).length;  
 //这样写法有点绕,其实这里就是通过索引获取table数组中的一个元素看是否为Nul  
 if ((p = tab[i = (n - 1) & hash]) == null)  
 //若判断成立,则New一个Node出来赋给table中指定索引下的这个元素  
 tab[i] = newNode(hash, key, value, null);  
 else { //若判断不成立  
 Node<K,V> e; K k;  
 //对这个元素进行Hash和key值匹配  
 if (p.hash == hash &&  
 ((k = p.key) == key || (key != null && key.equals(k))))  
 e = p;  
 else if (p instanceof TreeNode) //如果数组中德这个元素P是TreeNode类型  
 //判定成功则在红黑树中查找符合的条件的节点并返回此节点  
 e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);  
 else { //若以上条件均判断失败，则执行以下代码  
 //向Node单向链表中添加数据  
 for (int binCount = 0; ; ++binCount) {  
 if ((e = p.next) == null) {  
 p.next = newNode(hash, key, value, null);  
 //若节点数大于等于8  
 if (binCount >= TREEIFY\_THRESHOLD - 1) // -1 for 1st  
 //转换为红黑树  
 treeifyBin(tab, hash);  
 break;  
 }  
 if (e.hash == hash &&  
 ((k = e.key) == key || (key != null && key.equals(k))))  
 break;  
 p = e; //p记录下一个节点  
 }  
 }  
 if (e != null) { // existing mapping for key  
 V oldValue = e.value;  
 if (!onlyIfAbsent || oldValue == null)  
 e.value = value;  
 afterNodeAccess(e);  
 return oldValue;  
 }  
 }  
 ++modCount;  
 if (++size > threshold) //判断是否需要扩容  
 resize();  
 afterNodeInsertion(evict);  
 return null;  
 }

梳理以下HashMap.put函数的执行过程

1. 首先获取Node数组table对象和长度，若table为null或长度为0，则调用resize()扩容方法获取table最新对象，并通过此对象获取长度大小
2. 判定数组中指定索引下的节点是否为Null，若为Null 则new出一个单向链表赋给table中索引下的这个节点
3. 若判定不为Null,我们的判断再做分支   
   3.1 首先对hash和key进行匹配,若判定成功直接赋予e   
   3.2 若匹配判定失败,则进行类型匹配是否为TreeNode 若判定成功则在红黑树中查找符合条件的节点并将其回传赋给e   
   3.3 若以上判定全部失败则进行最后操作,向单向链表中添加数据若单向链表的长度大于等于8,则将其转为红黑树保存，记录下一个节点,对e进行判定若成功则返回旧值
4. 最后判定数组大小需不需要扩容
5. HashMap.get的源码实现

//这里直接调用getNode函数实现方法  
 public V get(Object key) {  
 Node<K,V> e;  
 //经过hash函数运算 获取key的hash值  
 return (e = getNode(hash(key), key)) == null ? null : e.value;  
 }  
   
 final Node<K,V> getNode(int hash, Object key) {  
 Node<K,V>[] tab; Node<K,V> first, e; int n; K k;  
 //判定三个条件 table不为Null & table的长度大于0 & table指定的索引值不为Null  
 if ((tab = table) != null && (n = tab.length) > 0 &&  
 (first = tab[(n - 1) & hash]) != null) {  
 //判定 匹配hash值 & 匹配key值 成功则返回 该值  
 if (first.hash == hash &&   
 ((k = first.key) == key || (key != null && key.equals(k))))  
 return first;  
 //若 first节点的下一个节点不为Null  
 if ((e = first.next) != null) {  
 if (first instanceof TreeNode) //若first的类型为TreeNode 红黑树  
 //通过红黑树查找匹配值 并返回  
 return ((TreeNode<K,V>)first).getTreeNode(hash, key);   
 //若上面判定不成功 则认为下一个节点为单向链表,通过循环匹配值  
 do {  
 if (e.hash == hash &&  
 ((k = e.key) == key || (key != null && key.equals(k))))  
 //匹配成功后返回该值  
 return e;  
 } while ((e = e.next) != null);  
 }  
 }  
 return null;  
 }

梳理以下HashMap.get函数的执行过程

1. 判定三个条件 table不为Null & table的长度大于0 & table指定的索引值不为Null
2. 判定 匹配hash值 & 匹配key值 成功则返回 该值
3. 若 first节点的下一个节点不为Null   
   3.1 若first的类型为TreeNode 红黑树 通过红黑树查找匹配值 并返回查询值   
   3.2若上面判定不成功   
   则认为下一个节点为单向链表,通过循环匹配值
4. HashMap扩容原理分析

//重新设置table大小/扩容 并返回扩容的Node数组即HashMap的最新数据  
final Node<K,V>[] resize() {  
 Node<K,V>[] oldTab = table; //table赋予oldTab作为扩充前的table数据  
 int oldCap = (oldTab == null) ? 0 : oldTab.length;   
 int oldThr = threshold;  
 int newCap, newThr = 0;  
 if (oldCap > 0) {  
 //判定数组是否已达到极限大小，若判定成功将不再扩容，直接将老表返回  
 if (oldCap >= MAXIMUM\_CAPACITY) {  
 threshold = Integer.MAX\_VALUE;  
 return oldTab;  
 }  
 //若新表大小(oldCap\*2)小于数组极限大小 并且 老表大于等于数组初始化大小  
 else if ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY &&  
 oldCap >= DEFAULT\_INITIAL\_CAPACITY)  
 //旧数组大小oldThr 经二进制运算向左位移1个位置 即 oldThr\*2当作新数组的大小  
 newThr = oldThr << 1; // double threshold  
 }  
 //若老表中下次扩容大小oldThr大于0  
 else if (oldThr > 0)  
 newCap = oldThr; //将oldThr赋予控制新表大小的newCap  
 else { //若其他情况则将获取初始默认大小  
 newCap = DEFAULT\_INITIAL\_CAPACITY;  
 newThr = (int)(DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY);  
 }  
 //若新表的下表下一次扩容大小为0  
 if (newThr == 0) {   
 float ft = (float)newCap \* loadFactor; //通过新表大小\*负载因子获取  
 newThr = (newCap < MAXIMUM\_CAPACITY && ft < (float)MAXIMUM\_CAPACITY ?  
 (int)ft : Integer.MAX\_VALUE);  
 }  
 threshold = newThr; //下次扩容的大小  
 @SuppressWarnings({"rawtypes","unchecked"})  
 Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];  
 table = newTab; //将当前表赋予table  
 if (oldTab != null) { //若oldTab中有值需要通过循环将oldTab中的值保存到新表中  
 for (int j = 0; j < oldCap; ++j) {  
 Node<K,V> e;  
 if ((e = oldTab[j]) != null) {//获取老表中第j个元素 赋予e  
 oldTab[j] = null; //并将老表中的元素数据置Null  
 if (e.next == null) //若此判定成立 则代表e的下面没有节点了  
 newTab[e.hash & (newCap - 1)] = e; //将e直接存于新表的指定位置  
 else if (e instanceof TreeNode) //若e是TreeNode类型  
 //分割树，将新表和旧表分割成两个树，并判断索引处节点的长度是否需要转换成红黑树放入新表存储  
 ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);  
 else { // preserve order  
 Node<K,V> loHead = null, loTail = null; //存储与旧索引的相同的节点  
 Node<K,V> hiHead = null, hiTail = null; //存储与新索引相同的节点  
 Node<K,V> next;  
 //通过Do循环 获取新旧索引的节点  
 do {  
 next = e.next;  
 if ((e.hash & oldCap) == 0) {  
 if (loTail == null)  
 loHead = e;  
 else  
 loTail.next = e;  
 loTail = e;  
 }  
 else {  
 if (hiTail == null)  
 hiHead = e;  
 else  
 hiTail.next = e;  
 hiTail = e;  
 }  
 } while ((e = next) != null);  
 //通过判定将旧数据和新数据存储到新表指定的位置  
 if (loTail != null) {  
 loTail.next = null;  
 newTab[j] = loHead;  
 }  
 if (hiTail != null) {  
 hiTail.next = null;  
 newTab[j + oldCap] = hiHead;  
 }  
 }  
 }  
 }  
 }  
 //返回新表  
 return newTab;  
 }

梳理以下HashMap.resize函数的执行过程

1. 判定数组是否已达到极限大小，若判定成功将不再扩容，直接将老表返回
2. 若新表大小(oldCap2)小于数组极限大小&老表大于等于数组初始化大小 判定成功则 旧数组大小oldThr 经二进制运算向左位移1个位置 即 oldThr2当作新数组的大小   
   2.1. 若[2]的判定不成功，则继续判定 oldThr （代表 老表的下一次扩容量）大于0，若判定成功 则将oldThr赋给newCap作为新表的容量   
   2.2 若 [2] 和[2.1]判定都失败,则走默认赋值 代表 表为初次创建
3. 确定下一次表的扩容量, 将新表赋予当前表
4. 通过for循环将老表中德值存入扩容后的新表中   
   4.1. 获取旧表中指定索引下的Node对象 赋予e 并将旧表中的索引位置数据置空   
   4.2. 若e的下面没有其他节点则将e直接赋到新表中的索引位置   
   4.3. 若e的类型为TreeNode红黑树类型

4.3.1. 分割树，将新表和旧表分割成两个树，并判断索引处节点的长度是否需要转换成红黑树放入新表存储   
4.3.2 通过Do循环 不断获取新旧索引的节点   
4.3.3 通过判定将旧数据和新数据存储到新表指定的位置   
最后返回值为 扩容后的新表 。

1. HashMap 的treeifyBin讲解

final void treeifyBin(Node<K,V>[] tab, int hash) {  
 int n, index; Node<K,V> e;  
 //做判定 tab 为Null 或 tab的长度小于 红黑树最小容量  
 if (tab == null || (n = tab.length) < MIN\_TREEIFY\_CAPACITY)  
 //则通过扩容,扩容table数组大小  
 resize();  
 //做判定 若tab索引位置下数据不为空  
 else if ((e = tab[index = (n - 1) & hash]) != null) {  
 //定义两个红黑树；分别表示头部节点、尾部节点  
 TreeNode<K,V> hd = null, tl = null;   
 //通过循环将单向链表转换为红黑树存储  
 do {  
 //将单向链表转换为红黑树  
 TreeNode<K,V> p = replacementTreeNode(e, null);  
 if (tl == null) //若头部节点为Null,则说明该树没有根节点  
 hd = p;  
 else {  
 p.prev = tl; //指向父节点  
 tl.next = p; //指向下一个节点  
 }  
 tl = p; //将当前节点设尾节点  
 } while ((e = e.next) != null); //若下一个不为Null,则继续遍历  
 //红黑树转换后,替代原位置上的单项链表  
 if ((tab[index] = hd) != null)  
 hd.treeify(tab); // 构建红黑树，以头部节点定为根节点  
 }  
 }  
   
 TreeNode<K,V> replacementTreeNode(Node<K,V> p, Node<K,V> next) {  
 return new TreeNode<>(p.hash, p.key, p.value, next);  
 }

梳理以下HashMap.treeifyBin函数的执行过程   
1.做判定 tab 为Null 或 tab的长度小于红黑树最小容量， 判定成功则通过扩容,扩容table数组大小   
2.做判定 若tab索引位置下数据不为空，判定成功则通过循环将单向链表转换为红黑树存储   
2.1 通过Do循环将当前节点下的单向链表转换为红黑树，若下一个不为Null,则继续遍历   
2.2 构建红黑树，以头部节点定为根节点

This ALL! Thanks EveryBody!

### 4. HashMap 的工作原理是什么?

**你知道HashMap的工作原理吗？**

胡子发芽 2019.08.28 20:39:40 字数 1,294 阅读 388

目录

1、HashMap简介

2、底层数据结构分析

jdk1.8之前和jdk1.8之后对比

类的属性

3、hashmap源码

(1).Node节点类源码

(2).TreeNode树节点类源码

(3).构造方法

(4).putMapEntries方法

(5).put方法

(6).get方法

(7).resize方法

#### HashMap简介

HashMap 主要用来存放键值对，它基于哈希表的Map接口实现，是常用的Java集合之一。

JDK1.8 之前 HashMap 由 数组+链表 组成的，数组是 HashMap 的主体，链表则是主要为了解决哈希冲突而存在的（**“拉链法”解决冲突**）.JDK1.8 以后在解决哈希冲突时有了较大的变化，当链表长度大于阈值（默认为 8）时，**将链表转化为红黑树**，*以减少搜索时间*。

#### 底层数据结构分析

(1).JDK1.8之前和jdk1.8对比

* JDK1.8 之前

HashMap 底层是数组和链表结合在一起使用,也就是**链表散列**。

HashMap 通过 key 的 hashCode 经过**扰动函数**处理过后得到 hash 值，然后通过(n - 1) & hash判断当前元素存放的位置（这里的 n 指的是数组的长度），如果当前位置存在元素的话，就判断该元素与要存入的元素的 hash 值以及 key 是否相同，如果相同的话，直接覆盖，不相同就通过拉链法解决冲突。

**所谓扰动函数**指的就是<u>HashMap 的 hash 方法</u>。使用 hash 方法也就是扰动函数是*为了防止一些实现比较差的 hashCode() 方法* ,换句话说使用扰动函数之后可以减少碰撞。

JDK 1.8 HashMap 的 hash 方法源码:

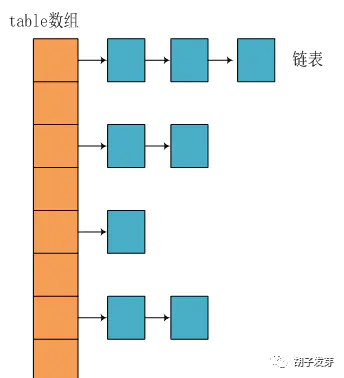
/\*\*  
 \* Computes key.hashCode() and spreads (XORs) higher bits of hash  
 \* to lower. Because the table uses power-of-two masking, sets of  
 \* hashes that vary only in bits above the current mask will  
 \* always collide. (Among known examples are sets of Float keys  
 \* holding consecutive whole numbers in small tables.) So we  
 \* apply a transform that spreads the impact of higher bits  
 \* downward. There is a tradeoff between speed, utility, and  
 \* quality of bit-spreading. Because many common sets of hashes  
 \* are already reasonably distributed (so don't benefit from  
 \* spreading), and because we use trees to handle large sets of  
 \* collisions in bins, we just XOR some shifted bits in the  
 \* cheapest possible way to reduce systematic lossage, as well as  
 \* to incorporate impact of the highest bits that would otherwise  
 \* never be used in index calculations because of table bounds.  
 \*/  
 static final int hash(Object key) {  
 int h;  
 return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);  
 }

JDK 1.8 的 hash方法 相比于 JDK 1.7 hash 方法更加简化，但是原理不变。

/\*\*  
 \* Retrieve object hash code and applies a supplemental hash function to the  
 \* result hash, which defends against poor quality hash functions. This is  
 \* critical because HashMap uses power-of-two length hash tables, that  
 \* otherwise encounter collisions for hashCodes that do not differ  
 \* in lower bits. Note: Null keys always map to hash 0, thus index 0.  
 \*/  
 final int hash(Object k) {  
 int h = hashSeed;  
 if (0 != h && k instanceof String) {  
 return sun.misc.Hashing.stringHash32((String) k);  
 }  
  
 h ^= k.hashCode();  
  
 // This function ensures that hashCodes that differ only by  
 // constant multiples at each bit position have a bounded  
 // number of collisions (approximately 8 at default load factor).  
 h ^= (h >>> 20) ^ (h >>> 12);  
 return h ^ (h >>> 7) ^ (h >>> 4);  
 }

对比一下 JDK1.7的 HashMap 的 hash 方法源码.

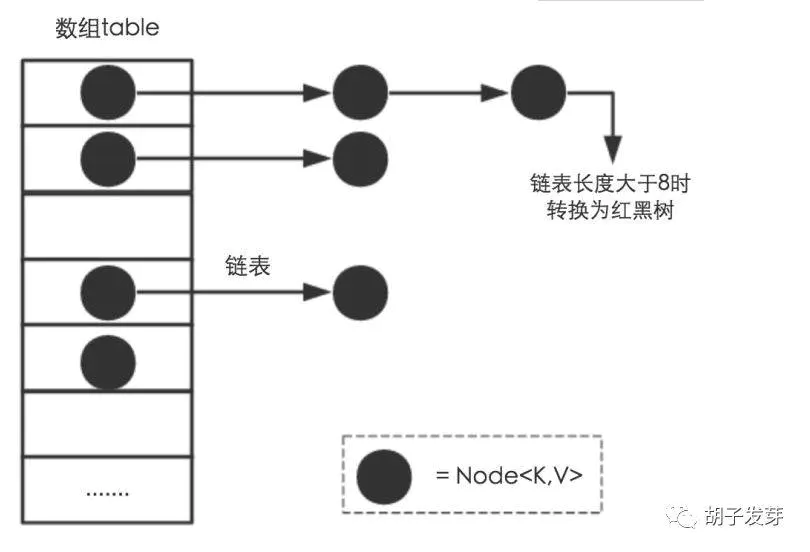
（相比于 JDK1.8 的 hash 方法 ，JDK 1.7 的 hash 方法的性能会稍差一点点，因为毕竟扰动了 4 次）。

**所谓 “拉链法”就是**：  
<u>将链表和数组相结合。也就是说创建一个链表数组，数组中每一格就是一个链表。若遇到哈希冲突，则将冲突的值加到链表中即可</u>。   
 

* JDK1.8之后

HashMap是数组、链表和红黑树结合在一起使用。

相比于之前的版本，jdk1.8在解决哈希冲突时有了较大的变化。当链表长度大于阈值（默认为8）时，将链表转化为红黑树，以减少搜索时间。

  
(2).类的属性：

public class HashMap<K,V> extends AbstractMap<K,V>  
 implements Map<K,V>, Cloneable, Serializable {  
  
 private static final long serialVersionUID = 362498820763181265L;  
  
 /\*\*默认初始容量  
 \* The default initial capacity - MUST be a power of two.  
 \*/  
 static final int DEFAULT\_INITIAL\_CAPACITY = 1 << 4; // aka 16  
  
 /\*\*最大容量   
 \* The maximum capacity, used if a higher value is implicitly specified  
 \* by either of the constructors with arguments.  
 \* MUST be a power of two <= 1<<30.  
 \*/  
 static final int MAXIMUM\_CAPACITY = 1 << 30;  
  
 /\*\*默认的填充因子  
 \* The load factor used when none specified in constructor.  
 \*/  
 static final float DEFAULT\_LOAD\_FACTOR = 0.75f;  
  
 /\*\* 当桶（bucket）上的节点数大于这个数时会转成红黑树  
 \* The bin count threshold for using a tree rather than list for a  
 \* bin. Bins are converted to trees when adding an element to a  
 \* bin with at least this many nodes. The value must be greater  
 \* than 2 and should be at least 8 to mesh with assumptions in  
 \* tree removal about conversion back to plain bins upon  
 \* shrinkage.  
 \*/  
 static final int TREEIFY\_THRESHOLD = 8;  
  
 /\*\*当桶（bucket）上的节点数小于这个值时树转链表  
 \* The bin count threshold for untreeifying a (split) bin during a  
 \* resize operation. Should be less than TREEIFY\_THRESHOLD, and at  
 \* most 6 to mesh with shrinkage detection under removal.  
 \*/  
 static final int UNTREEIFY\_THRESHOLD = 6;  
  
 /\*\*  
 \* The smallest table capacity for which bins may be treeified.  
 \* (Otherwise the table is resized if too many nodes in a bin.)  
 \* Should be at least 4 \* TREEIFY\_THRESHOLD to avoid conflicts  
 \* between resizing and treeification thresholds.  
 \*/  
 static final int MIN\_TREEIFY\_CAPACITY = 64;  
  
   
  
 /\* ---------------- Fields -------------- \*/  
  
 /\*\*  
 \* The table, initialized on first use, and resized as  
 \* necessary. When allocated, length is always a power of two.  
 \* (We also tolerate length zero in some operations to allow  
 \* bootstrapping mechanics that are currently not needed.)  
 \*/  
 transient Node<K,V>[] table;  
  
 /\*\*  
 \* Holds cached entrySet(). Note that AbstractMap fields are used  
 \* for keySet() and values().  
 \*/  
 transient Set<Map.Entry<K,V>> entrySet;  
  
 /\*\*  
 \* The number of key-value mappings contained in this map.  
 \*/  
 transient int size;  
  
 /\*\*  
 \* The number of times this HashMap has been structurally modified  
 \* Structural modifications are those that change the number of mappings in  
 \* the HashMap or otherwise modify its internal structure (e.g.,  
 \* rehash). This field is used to make iterators on Collection-views of  
 \* the HashMap fail-fast. (See ConcurrentModificationException).  
 \*/  
 transient int modCount;  
  
 /\*\*  
 \* The next size value at which to resize (capacity \* load factor).  
 \*  
 \* @serial  
 \*/  
 // (The javadoc description is true upon serialization.  
 // Additionally, if the table array has not been allocated, this  
 // field holds the initial array capacity, or zero signifying  
 // DEFAULT\_INITIAL\_CAPACITY.)  
 int threshold;  
  
 /\*\*  
 \* The load factor for the hash table.  
 \*  
 \* @serial  
 \*/  
 final float loadFactor;  
}

**loadFactor加载因子**

loadFactor加载因子:**控制数组存放数据的疏密程度**，loadFactor越趋近于1，那么 数组中存放的数据(entry)也就越多，也就越密，也就是会让链表的长度增加，loadFactor越小，也就是趋近于0，数组中存放的数据(entry)也就越少，也就越稀疏。

loadFactor太大导致查找元素效率低，太小导致数组的利用率低，存放的数据会很分散。loadFactor的默认值为0.75f是官方给出的一个比较好的临界值。

给定的默认容量为 16，负载因子为 0.75。Map 在使用过程中不断的往里面存放数据，当数量达到了 16 \* 0.75 = 12 就需要将当前 16 的容量进行扩容，而**扩容这个过程涉及到 rehash、复制数据等操作，所以非常消耗性能**。

**threshold**

threshold = capacity \* loadFactor，当Size>=threshold的时候，那么就要考虑对数组的扩增了，也就是说，这个的意思就是衡量数组是否需要扩增的一个标准。

#### HashMap源码分析

(1).Node节点类源码:

\*\*  
 \* Basic hash bin node, used for most entries. (See below for  
 \* TreeNode subclass, and in LinkedHashMap for its Entry subclass.)  
 \*/  
 static class Node<K,V> implements Map.Entry<K,V> {  
 final int hash;  
 final K key;  
 V value;  
 Node<K,V> next;  
  
 Node(int hash, K key, V value, Node<K,V> next) {  
 this.hash = hash;  
 this.key = key;  
 this.value = value;  
 this.next = next;  
 }  
  
 public final K getKey() { return key; }  
 public final V getValue() { return value; }  
 public final String toString() { return key + "=" + value; }  
  
 public final int hashCode() {  
 return Objects.hashCode(key) ^ Objects.hashCode(value);  
 }  
  
 public final V setValue(V newValue) {  
 V oldValue = value;  
 value = newValue;  
 return oldValue;  
 }  
  
 public final boolean equals(Object o) {  
 if (o == this)  
 return true;  
 if (o instanceof Map.Entry) {  
 Map.Entry<?,?> e = (Map.Entry<?,?>)o;  
 if (Objects.equals(key, e.getKey()) &&  
 Objects.equals(value, e.getValue()))  
 return true;  
 }  
 return false;  
 }  
 }

(2).TreeNode树节点类源码:

/\*\*  
 \* Entry for Tree bins. Extends LinkedHashMap.Entry (which in turn  
 \* extends Node) so can be used as extension of either regular or  
 \* linked node.  
 \*/  
 static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {  
 TreeNode<K,V> parent; // red-black tree links  
 TreeNode<K,V> left;  
 TreeNode<K,V> right;  
 TreeNode<K,V> prev; // needed to unlink next upon deletion  
 boolean red;  
 TreeNode(int hash, K key, V val, Node<K,V> next) {  
 super(hash, key, val, next);  
 }  
  
 /\*\*  
 \* Returns root of tree containing this node.  
 \*/  
 final TreeNode<K,V> root() {  
 for (TreeNode<K,V> r = this, p;;) {  
 if ((p = r.parent) == null)  
 return r;  
 r = p;  
 }  
 }  
 }

(3).构造方法

HashMap 中有四个构造方法，它们分别如下：

/\*\*  
 \* Constructs an empty <tt>HashMap</tt> with the specified initial  
 \* capacity and load factor.  
 \*  
 \* @param initialCapacity the initial capacity  
 \* @param loadFactor the load factor  
 \* @throws IllegalArgumentException if the initial capacity is negative  
 \* or the load factor is nonpositive  
 \*/  
 public HashMap(int initialCapacity, float loadFactor) {  
 if (initialCapacity < 0)  
 throw new IllegalArgumentException("Illegal initial capacity: " +  
 initialCapacity);  
 if (initialCapacity > MAXIMUM\_CAPACITY)  
 initialCapacity = MAXIMUM\_CAPACITY;  
 if (loadFactor <= 0 || Float.isNaN(loadFactor))  
 throw new IllegalArgumentException("Illegal load factor: " +  
 loadFactor);  
 this.loadFactor = loadFactor;  
 this.threshold = tableSizeFor(initialCapacity);  
 }  
  
 /\*\*  
 \* Constructs an empty <tt>HashMap</tt> with the specified initial  
 \* capacity and the default load factor (0.75).  
 \*  
 \* @param initialCapacity the initial capacity.  
 \* @throws IllegalArgumentException if the initial capacity is negative.  
 \*/  
 public HashMap(int initialCapacity) {  
 this(initialCapacity, DEFAULT\_LOAD\_FACTOR);  
 }  
  
 /\*\*  
 \* Constructs an empty <tt>HashMap</tt> with the default initial capacity  
 \* (16) and the default load factor (0.75).  
 \*/  
 public HashMap() {  
 this.loadFactor = DEFAULT\_LOAD\_FACTOR; // all other fields defaulted  
 }  
  
 /\*\*  
 \* Constructs a new <tt>HashMap</tt> with the same mappings as the  
 \* specified <tt>Map</tt>. The <tt>HashMap</tt> is created with  
 \* default load factor (0.75) and an initial capacity sufficient to  
 \* hold the mappings in the specified <tt>Map</tt>.  
 \*  
 \* @param m the map whose mappings are to be placed in this map  
 \* @throws NullPointerException if the specified map is null  
 \*/  
 public HashMap(Map<? extends K, ? extends V> m) {  
 this.loadFactor = DEFAULT\_LOAD\_FACTOR;  
 putMapEntries(m, false);  
 }

(4).putMapEntries方法：

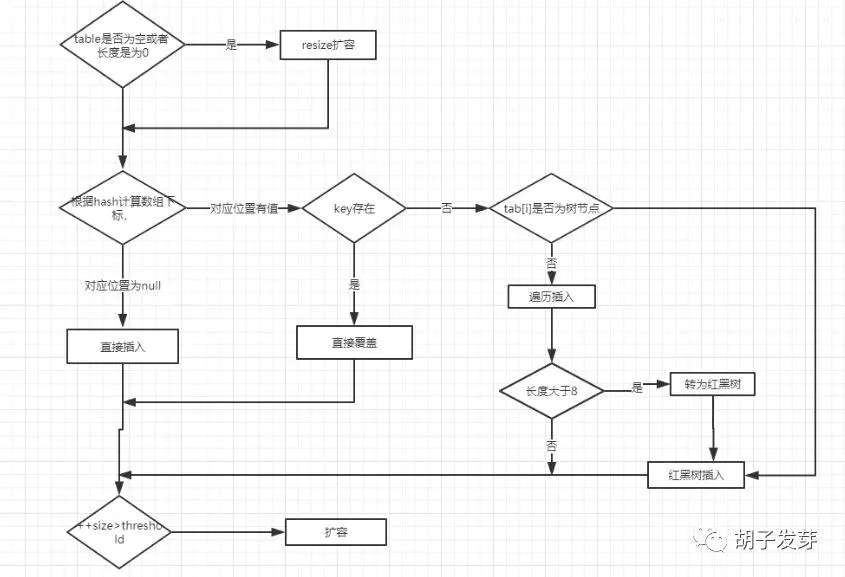
/\*\*  
 \* Implements Map.putAll and Map constructor  
 \*  
 \* @param m the map  
 \* @param evict false when initially constructing this map, else  
 \* true (relayed to method afterNodeInsertion).  
 \*/  
 final void putMapEntries(Map<? extends K, ? extends V> m, boolean evict) {  
 int s = m.size();  
 if (s > 0) {  
 //判断table是否初始化了  
 if (table == null) { // pre-size  
 //未初始化，s为m的实际元素个数  
 float ft = ((float)s / loadFactor) + 1.0F;  
 int t = ((ft < (float)MAXIMUM\_CAPACITY) ?  
 (int)ft : MAXIMUM\_CAPACITY);  
 //计算得到的t大于阈值，则初始化阈值  
 if (t > threshold)  
 threshold = tableSizeFor(t);  
 }  
 //已初始化，且m元素个数大于阈值，进行扩容处理  
 else if (s > threshold)  
 resize();  
 //将m中所有元素添加至hashmap中  
 for (Map.Entry<? extends K, ? extends V> e : m.entrySet()) {  
 K key = e.getKey();  
 V value = e.getValue();  
 putVal(hash(key), key, value, false, evict);  
 }  
 }  
 }

(5).put方法

HashMap只提供了put用于添加元素，putVal方法只是给put方法调用的一个方法，并没有提供给用户使用。

对putVal方法添加元素的分析如下：

①如果定位到的数组位置没有元素 就直接插入。

②如果定位到的数组位置有元素就和要插入的key比较，如果key相同就直接覆盖，如果key不相同，就判断p是否是一个树节点，如果是就调用e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value)将元素添加进入。如果不是就遍历链表插入(插入的是链表尾部)。  


//jdk1.8  
/\*\*  
 \* Associates the specified value with the specified key in this map.  
 \* If the map previously contained a mapping for the key, the old  
 \* value is replaced.  
 \*  
 \* @param key key with which the specified value is to be associated  
 \* @param value value to be associated with the specified key  
 \* @return the previous value associated with <tt>key</tt>, or  
 \* <tt>null</tt> if there was no mapping for <tt>key</tt>.  
 \* (A <tt>null</tt> return can also indicate that the map  
 \* previously associated <tt>null</tt> with <tt>key</tt>.)  
 \*/  
 public V put(K key, V value) {  
 return putVal(hash(key), key, value, false, true);  
 }  
  
 /\*\*  
 \* Implements Map.put and related methods  
 \*  
 \* @param hash hash for key  
 \* @param key the key  
 \* @param value the value to put  
 \* @param onlyIfAbsent if true, don't change existing value  
 \* @param evict if false, the table is in creation mode.  
 \* @return previous value, or null if none  
 \*/  
 final V putVal(int hash, K key, V value, boolean onlyIfAbsent,  
 boolean evict) {  
 Node<K,V>[] tab; Node<K,V> p; int n, i;  
 if ((tab = table) == null || (n = tab.length) == 0)  
 n = (tab = resize()).length;  
 if ((p = tab[i = (n - 1) & hash]) == null)  
 tab[i] = newNode(hash, key, value, null);  
 else {  
 Node<K,V> e; K k;  
 if (p.hash == hash &&  
 ((k = p.key) == key || (key != null && key.equals(k))))  
 e = p;  
 else if (p instanceof TreeNode)  
 e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);  
 else {  
 for (int binCount = 0; ; ++binCount) {  
 if ((e = p.next) == null) {  
 p.next = newNode(hash, key, value, null);  
 if (binCount >= TREEIFY\_THRESHOLD - 1) // -1 for 1st  
 treeifyBin(tab, hash);  
 break;  
 }  
 if (e.hash == hash &&  
 ((k = e.key) == key || (key != null && key.equals(k))))  
 break;  
 p = e;  
 }  
 }  
 if (e != null) { // existing mapping for key  
 V oldValue = e.value;  
 if (!onlyIfAbsent || oldValue == null)  
 e.value = value;  
 afterNodeAccess(e);  
 return oldValue;  
 }  
 }  
 ++modCount;  
 if (++size > threshold)  
 resize();  
 afterNodeInsertion(evict);  
 return null;  
 }

我们再来对比一下 JDK1.7 put方法的代码

对于put方法的分析如下：

①如果定位到的数组位置没有元素 就直接插入。

②如果定位到的数组位置有元素，遍历以这个元素为头结点的链表，依次和插入的key比较，如果key相同就直接覆盖，不同就采用头插法插入元素。

/\*\*  
 \* Associates the specified value with the specified key in this map.  
 \* If the map previously contained a mapping for the key, the old  
 \* value is replaced.  
 \*  
 \* @param key key with which the specified value is to be associated  
 \* @param value value to be associated with the specified key  
 \* @return the previous value associated with <tt>key</tt>, or  
 \* <tt>null</tt> if there was no mapping for <tt>key</tt>.  
 \* (A <tt>null</tt> return can also indicate that the map  
 \* previously associated <tt>null</tt> with <tt>key</tt>.)  
 \*/  
 public V put(K key, V value) {  
 if (table == EMPTY\_TABLE) {  
 inflateTable(threshold);  
 }  
 if (key == null)  
 return putForNullKey(value);  
 int hash = hash(key);  
 int i = indexFor(hash, table.length);  
 for (Entry<K,V> e = table[i]; e != null; e = e.next) {  
 Object k;  
 if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {  
 V oldValue = e.value;  
 e.value = value;  
 e.recordAccess(this);  
 return oldValue;  
 }  
 }  
  
 modCount++;  
 addEntry(hash, key, value, i);  
 return null;  
 }  
  
 /\*\*  
 \* Offloaded version of put for null keys  
 \*/  
 private V putForNullKey(V value) {  
 for (Entry<K,V> e = table[0]; e != null; e = e.next) {  
 if (e.key == null) {  
 V oldValue = e.value;  
 e.value = value;  
 e.recordAccess(this);  
 return oldValue;  
 }  
 }  
 modCount++;  
 addEntry(0, null, value, 0);  
 return null;  
 }

(6).get方法

//jdk1.8  
/\*\*  
 \* Returns the value to which the specified key is mapped,  
 \* or {@code null} if this map contains no mapping for the key.  
 \*  
 \* <p>More formally, if this map contains a mapping from a key  
 \* {@code k} to a value {@code v} such that {@code (key==null ? k==null :  
 \* key.equals(k))}, then this method returns {@code v}; otherwise  
 \* it returns {@code null}. (There can be at most one such mapping.)  
 \*  
 \* <p>A return value of {@code null} does not <i>necessarily</i>  
 \* indicate that the map contains no mapping for the key; it's also  
 \* possible that the map explicitly maps the key to {@code null}.  
 \* The {@link #containsKey containsKey} operation may be used to  
 \* distinguish these two cases.  
 \*  
 \* @see #put(Object, Object)  
 \*/  
 public V get(Object key) {  
 Node<K,V> e;  
 return (e = getNode(hash(key), key)) == null ? null : e.value;  
 }  
  
 /\*\*  
 \* Implements Map.get and related methods  
 \*  
 \* @param hash hash for key  
 \* @param key the key  
 \* @return the node, or null if none  
 \*/  
 final Node<K,V> getNode(int hash, Object key) {  
 Node<K,V>[] tab; Node<K,V> first, e; int n; K k;  
 if ((tab = table) != null && (n = tab.length) > 0 &&  
 (first = tab[(n - 1) & hash]) != null) {  
 if (first.hash == hash && // always check first node  
 ((k = first.key) == key || (key != null && key.equals(k))))  
 return first;  
 if ((e = first.next) != null) {  
 if (first instanceof TreeNode)  
 return ((TreeNode<K,V>)first).getTreeNode(hash, key);  
 do {  
 if (e.hash == hash &&  
 ((k = e.key) == key || (key != null && key.equals(k))))  
 return e;  
 } while ((e = e.next) != null);  
 }  
 }  
 return null;  
 }

(7).resize方法

进行扩容，会伴随着一次重新hash分配，并且会遍历hash表中所有的元素，是非常耗时的。在编写程序中，要尽量避免resize。

/\*\*  
 \* Initializes or doubles table size. If null, allocates in  
 \* accord with initial capacity target held in field threshold.  
 \* Otherwise, because we are using power-of-two expansion, the  
 \* elements from each bin must either stay at same index, or move  
 \* with a power of two offset in the new table.  
 \*  
 \* @return the table  
 \*/  
 final Node<K,V>[] resize() {  
 Node<K,V>[] oldTab = table;  
 int oldCap = (oldTab == null) ? 0 : oldTab.length;  
 int oldThr = threshold;  
 int newCap, newThr = 0;  
 if (oldCap > 0) {  
 if (oldCap >= MAXIMUM\_CAPACITY) {  
 threshold = Integer.MAX\_VALUE;  
 return oldTab;  
 }  
 else if ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY &&  
 oldCap >= DEFAULT\_INITIAL\_CAPACITY)  
 newThr = oldThr << 1; // double threshold  
 }  
 else if (oldThr > 0) // initial capacity was placed in threshold  
 newCap = oldThr;  
 else { // zero initial threshold signifies using defaults  
 newCap = DEFAULT\_INITIAL\_CAPACITY;  
 newThr = (int)(DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY);  
 }  
 if (newThr == 0) {  
 float ft = (float)newCap \* loadFactor;  
 newThr = (newCap < MAXIMUM\_CAPACITY && ft < (float)MAXIMUM\_CAPACITY ?  
 (int)ft : Integer.MAX\_VALUE);  
 }  
 threshold = newThr;  
 @SuppressWarnings({"rawtypes","unchecked"})  
 Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];  
 table = newTab;  
 if (oldTab != null) {  
 for (int j = 0; j < oldCap; ++j) {  
 Node<K,V> e;  
 if ((e = oldTab[j]) != null) {  
 oldTab[j] = null;  
 if (e.next == null)  
 newTab[e.hash & (newCap - 1)] = e;  
 else if (e instanceof TreeNode)  
 ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);  
 else { // preserve order  
 Node<K,V> loHead = null, loTail = null;  
 Node<K,V> hiHead = null, hiTail = null;  
 Node<K,V> next;  
 do {  
 next = e.next;  
 if ((e.hash & oldCap) == 0) {  
 if (loTail == null)  
 loHead = e;  
 else  
 loTail.next = e;  
 loTail = e;  
 }  
 else {  
 if (hiTail == null)  
 hiHead = e;  
 else  
 hiTail.next = e;  
 hiTail = e;  
 }  
 } while ((e = next) != null);  
 if (loTail != null) {  
 loTail.next = null;  
 newTab[j] = loHead;  
 }  
 if (hiTail != null) {  
 hiTail.next = null;  
 newTab[j + oldCap] = hiHead;  
 }  
 }  
 }  
 }  
 }  
 return newTab;  
 }

### 4. HashMap常用方法

(1). Hashmap的存值：

public static void main(String[] args) {  
 ///\*Integer\*/map.put("1", 1);  
 //向map中添加值(返回这个key以前的值,如果没有返回null)  
 HashMap<String, Integer> map=new HashMap<>();  
 System.out.println(map.put("1", 1));//null  
 System.out.println(map.put("1", 2));//1  
}

(2). Hashmap的取值：

public static void main(String[] args) {  
 HashMap<String, Integer> map=new HashMap<>();  
 map.put("DEMO", 1);  
 /\*Value的类型\*///得到map中key相对应的value的值  
 System.out.println(map.get("1"));//null  
 System.out.println(map.get("DEMO"));//1  
}

(3). Hashmap的判断为空：

public static void main(String[] args) {  
 HashMap<String, Integer> map=new HashMap<>();  
 /\*boolean\*///判断map是否为空  
 System.out.println(map.isEmpty());//true  
 map.put("DEMO", 1);  
 System.out.println(map.isEmpty());//false  
}

(4). Hashmap判断是否含有key：

public static void main(String[] args) {  
 HashMap<String, Integer> map=new HashMap<>();  
 /\*boolean\*///判断map中是否存在这个key  
 System.out.println(map.containsKey("DEMO"));//false  
 map.put("DEMO", 1);  
 System.out.println(map.containsKey("DEMO"));//true  
}

(5). Hashmap判断是否含有value：

public static void main(String[] args) {  
 HashMap<String, Integer> map=new HashMap<>();  
 /\*boolean\*///判断map中是否存在这个value  
 System.out.println(map.containsValue(1));//false  
 map.put("DEMO", 1);  
 System.out.println(map.containsValue(1));//true  
}

(6).Hashmap删除这个key值下的value：

public static void main(String[] args) {  
 HashMap<String, Integer> map=new HashMap<>();  
 /\*Integer\*///删除key值下的value  
 System.out.println(map.remove("1"));//null  
 map.put("DEMO", 2);  
 System.out.println(map.remove("DEMO"));//2(删除的值)  
}

(7). Hashmap显示所有的value值：

public static void main(String[] args) {  
 HashMap<String, Integer> map=new HashMap<>();  
 /\*Collection<Integer>\*///显示所有的value值  
 System.out.println(map.values());//[]  
 map.put("DEMO1", 1);  
 map.put("DEMO2", 2);  
 System.out.println(map.values());//[1, 2]  
}

(8). Hashmap的元素个数：

public static void main(String[] args) {  
 HashMap<String, Integer> map=new HashMap<>();  
 /\*int\*///显示map里的值得数量  
 System.out.println(map.size());//0  
 map.put("DEMO1", 1);  
 System.out.println(map.size());//1  
 map.put("DEMO2", 2);  
 System.out.println(map.size());//2  
}

(9). Hashmap显示所有的key：

public static void main(String[] args) {  
 HashMap<String, Integer> map=new HashMap<>();  
 /\*SET<String>\*///显示map所有的key  
 System.out.println(map.keySet());//[]  
 map.put("DEMO1", 1);  
 System.out.println(map.keySet());//[DEMO1]  
 map.put("DEMO2", 2);  
 System.out.println(map.keySet());//[DEMO1, DEMO2]  
}

(10). Hashmap显示所有的key和value：

public static void main(String[] args) {  
 HashMap<String, Integer> map=new HashMap<>();  
 /\*SET<map<String,Integer>>\*///显示所有的key和value  
 System.out.println(map.entrySet());//[]  
 map.put("DEMO1", 1);  
 System.out.println(map.entrySet());//[DEMO1=1]  
 map.put("DEMO2", 2);  
 System.out.println(map.entrySet());//[DEMO1=1, DEMO2=2]  
}

(11). Hashmap添加另一个同一类型的map下的所有制：

public static void main(String[] args) {  
 HashMap<String, Integer> map=new HashMap<>();  
 HashMap<String, Integer> map1=new HashMap<>();  
 /\*void\*///将同一类型的map添加到另一个map中  
 map1.put("DEMO1", 1);  
 map.put("DEMO2", 2);  
 System.out.println(map);//{DEMO2=2}  
 map.putAll(map1);  
 System.out.println(map);//{DEMO1=1, DEMO2=2}  
}

(12). Hashmap删除这个key和value：

map.remove(Object key, Object value)：(java8新增方法)

Hashmap删除这个key，value对应的数据，并返回boolean：

public static void main(String[] args) {  
 HashMap<String, Integer> map=new HashMap<>();  
 /\*boolean\*///删除这个键值对  
 map.put("DEMO1", 1);  
 map.put("DEMO2", 2);  
 System.out.println(map);//{DEMO1=1, DEMO2=2}  
 System.out.println(map.remove("DEMO2", 1));//false  
 System.out.println(map.remove("DEMO2", 2));//true  
 System.out.println(map);//{DEMO1=1}  
}

源码如下：

@Override  
 public boolean remove(Object key, Object value) {  
 return removeNode(hash(key), key, value, true, true) != null;  
 }

(13). Hashmap替换这个key的value：(java8)

public static void main(String[] args) {  
 HashMap<String, Integer> map=new HashMap<>();  
 /\*value\*///判断map中是否存在这个key  
 map.put("DEMO1", 1);  
 map.put("DEMO2", 2);  
 System.out.println(map);//{DEMO1=1, DEMO2=2}  
 System.out.println(map.replace("DEMO2", 1));//2  
 System.out.println(map);//{DEMO1=1, DEMO2=1}  
}

(14). 清空这个hashmap：

public static void main(String[] args) {  
 HashMap<String, Integer> map=new HashMap<>();  
 /\*void\*///清空map  
 map.put("DEMO1", 1);  
 map.put("DEMO2", 2);  
 System.out.println(map);//{DEMO1=1, DEMO2=2}  
 map.clear();//2  
 System.out.println(map);//{}  
}

(15). Hashmap的克隆：

public static void main(String[] args) {  
 HashMap<String, Integer> map=new HashMap<>();  
 /\*object\*///克隆这个map  
 map.put("DEMO1", 1);  
 map.put("DEMO2", 2);  
 System.out.println(map.clone());//{DEMO1=1, DEMO2=2}  
 Object clone = map.clone();  
 System.out.println(clone);//{DEMO1=1, DEMO2=2}  
}

(15). map.putIfAbsent  
如果当前Map不存在键key或者该key关联的值为null，那么就执行put(key, value)；否则，便不执行put操作：（java8新增方法）

使用场景：如果我们要变更某个key的值，我们又不知道key是否存在的情况下，而又不希望增加key的情况使用。

public static void main(String[] args) {  
 HashMap<String, Integer> map=new HashMap<>();  
 /\*boolean\*///判断map中是否存在这个key  
 map.put("DEMO1", 1);  
 map.put("DEMO2", 2);  
 System.out.println(map);//{DEMO1=1, DEMO2=2}  
 System.out.println(map.putIfAbsent("DEMO1", 12222));//1  
 System.out.println(map.putIfAbsent("DEMO3", 12222));//null  
 System.out.println(map);//{DEMO1=12222, DEMO2=2}  
}

(16). map.compute   
如果当前Map的value为v1时,则值为v2,否则为v3：（java8新增方法）

compute方法更适用于更新key关联的value时，新值依赖于旧值的情况

public static void main(String[] args) {  
 HashMap<String, Integer> map=new HashMap<>();  
 /\*boolean\*///当这个value为null时为1，否则为3  
 map.put("DEMO1", 1);  
 map.put("DEMO2", 2);  
 System.out.println(map);//{DEMO1=1, DEMO2=2}  
 map.compute("DEMO2", (k,v)->v==null?1:3);  
 System.out.println(map);//{DEMO1=1, DEMO2=3}  
}

(17). map.computeIfAbsent：(java8新增方法)   
和putIfAbsent类似但是，在返回值上不一样，value值不存在的时候，返回的是新的value值，同时可以通过自定义一些条件进行过滤。

重新认识HashMap（jdk1.8新增特性）  
https://blog.csdn.net/hqy1719239337/article/details/83044599  
Java8（3）：Java8 中 Map 接口的新方法  
https://segmentfault.com/a/1190000007838166

/\*\*/map.computeIfAbsent(key, mappingFunction);  
 @Override  
 public V computeIfAbsent(K key,  
 Function<? super K, ? extends V> mappingFunction) {  
 if (mappingFunction == null)  
 throw new NullPointerException();  
 int hash = hash(key);  
 Node<K,V>[] tab; Node<K,V> first; int n, i;  
 int binCount = 0;  
 TreeNode<K,V> t = null;  
 Node<K,V> old = null;  
 if (size > threshold || (tab = table) == null ||  
 (n = tab.length) == 0)  
 n = (tab = resize()).length;  
 if ((first = tab[i = (n - 1) & hash]) != null) {  
 if (first instanceof TreeNode)  
 old = (t = (TreeNode<K,V>)first).getTreeNode(hash, key);  
 else {  
 Node<K,V> e = first; K k;  
 do {  
 if (e.hash == hash &&  
 ((k = e.key) == key || (key != null && key.equals(k)))) {  
 old = e;  
 break;  
 }  
 ++binCount;  
 } while ((e = e.next) != null);  
 }  
 V oldValue;  
 if (old != null && (oldValue = old.value) != null) {  
 afterNodeAccess(old);  
 return oldValue;  
 }  
 }  
 V v = mappingFunction.apply(key);  
 if (v == null) {  
 return null;  
 } else if (old != null) {  
 old.value = v;  
 afterNodeAccess(old);  
 return v;  
 }  
 else if (t != null)  
 t.putTreeVal(this, tab, hash, key, v);  
 else {  
 tab[i] = newNode(hash, key, v, first);  
 if (binCount >= TREEIFY\_THRESHOLD - 1)  
 treeifyBin(tab, hash);  
 }  
 ++modCount;  
 ++size;  
 afterNodeInsertion(true);  
 return v;  
 }

(18). map.computeIfPresent：(java8新增方法)

和computeIfAbsent方法正好相反，只有当key存在的时候才执行操作，

/\*\*/map.computeIfPresent(key, remappingFunction);   
  
public V computeIfPresent(K key,  
 BiFunction<? super K, ? super V, ? extends V> remappingFunction) {  
 if (remappingFunction == null)  
 throw new NullPointerException();  
 Node<K,V> e; V oldValue;  
 int hash = hash(key);  
 if ((e = getNode(hash, key)) != null &&  
 (oldValue = e.value) != null) {  
 V v = remappingFunction.apply(key, oldValue);  
 if (v != null) {  
 e.value = v;  
 afterNodeAccess(e);  
 return v;  
 }  
 else  
 removeNode(hash, key, null, false, true);  
 }  
 return null;  
 }

(19). map.forEach：(java8新增方法)

/\*\*/map.forEach());   
 @Override  
 public void forEach(BiConsumer<? super K, ? super V> action) {  
 Node<K,V>[] tab;  
 if (action == null)  
 throw new NullPointerException();  
 if (size > 0 && (tab = table) != null) {  
 int mc = modCount;  
 for (int i = 0; i < tab.length; ++i) {  
 for (Node<K,V> e = tab[i]; e != null; e = e.next)  
 action.accept(e.key, e.value);  
 }  
 if (modCount != mc)  
 throw new ConcurrentModificationException();  
 }  
 }

(20). map.merge：(java8新增方法)

/\*\*/map.merge(key, value, remappingFunction);  
 @Override  
 public V merge(K key, V value,  
 BiFunction<? super V, ? super V, ? extends V> remappingFunction) {  
 if (value == null)  
 throw new NullPointerException();  
 if (remappingFunction == null)  
 throw new NullPointerException();  
 int hash = hash(key);  
 Node<K,V>[] tab; Node<K,V> first; int n, i;  
 int binCount = 0;  
 TreeNode<K,V> t = null;  
 Node<K,V> old = null;  
 if (size > threshold || (tab = table) == null ||  
 (n = tab.length) == 0)  
 n = (tab = resize()).length;  
 if ((first = tab[i = (n - 1) & hash]) != null) {  
 if (first instanceof TreeNode)  
 old = (t = (TreeNode<K,V>)first).getTreeNode(hash, key);  
 else {  
 Node<K,V> e = first; K k;  
 do {  
 if (e.hash == hash &&  
 ((k = e.key) == key || (key != null && key.equals(k)))) {  
 old = e;  
 break;  
 }  
 ++binCount;  
 } while ((e = e.next) != null);  
 }  
 }  
 if (old != null) {  
 V v;  
 if (old.value != null)  
 v = remappingFunction.apply(old.value, value);  
 else  
 v = value;  
 if (v != null) {  
 old.value = v;  
 afterNodeAccess(old);  
 }  
 else  
 removeNode(hash, key, null, false, true);  
 return v;  
 }  
 if (value != null) {  
 if (t != null)  
 t.putTreeVal(this, tab, hash, key, value);  
 else {  
 tab[i] = newNode(hash, key, value, first);  
 if (binCount >= TREEIFY\_THRESHOLD - 1)  
 treeifyBin(tab, hash);  
 }  
 ++modCount;  
 ++size;  
 afterNodeInsertion(true);  
 }  
 return value;  
 }

(21). map.getOrDefault：(java8新增方法)

/\*\*/map.getOrDefault(key, defaultValue);  
 // Overrides of JDK8 Map extension methods  
  
 @Override  
 public V getOrDefault(Object key, V defaultValue) {  
 Node<K,V> e;  
 return (e = getNode(hash(key), key)) == null ? defaultValue : e.value;  
 }

(22).map.remove(Object key, Object value)：(java8新增方法)

Hashmap删除这个key，value对应的数据，并返回boolean：

————————————————  
版权声明：本文为CSDN博主「AlbenXie」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/AlbenXie/article/details/87950482

### 5. Hashmap 什么时候进行扩容呢？

[答案选自网络博客](https://blog.csdn.net/pange1991/article/details/82347284)https://blog.csdn.net/pange1991/article/details/82347284

5.0概述

1. 什么时候扩容
2. 源码分析
3. 补充一下jdk 1.8中，HashMap扩容时红黑树的表现
4. 总结

HashMap的扩容机制---resize()

**1、概述**

首先要了解HashMap的扩容过程，我们就得了解一些HashMap中的变量：

**Node：**链表节点，包含了key、value、hash、next指针四个元素  
**table：**Node<K,V>类型的数组，里面的元素是链表，用于存放HashMap元素的实体  
**size：**记录了放入HashMap的元素个数  
**loadFactor：**负载因子  
**threshold：**扩容的阈值，决定了HashMap何时扩容，以及扩容后的大小，一般等于 table \* loadFactor

[概述引用位置](作者：Jacknolfskin 链接：https://www.jianshu.com/p/937292c5d196 来源：简书 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处)https://www.jianshu.com/p/937292c5d196

**2、什么时候扩容：**

当向容器添加元素的时候，会判断当前容器的元素个数，如果大于等于阈值---即当前数组的长度乘以加载因子的值的时候，就要自动扩容啦。

扩容(resize)就是重新计算容量，向HashMap对象里不停的添加元素，而HashMap对象内部的数组无法装载更多的元素时，对象就需要扩大数组的长度，以便能装入更多的元素。当然Java里的数组是无法自动扩容的，方法是***使用一个新的数组代替已有的容量小的数组***，就像我们用一个小桶装水，如果想装更多的水，就得换大水桶。

网上总结的会有很多，大多都总结的不够完整或者不够准确。大多数可能说了满足我下面条件一的情况。

扩容必须满足两个条件：

**1、 存放新值的时候当前已有元素的个数必须大于等于阈值**

**2、 存放新值的时候当前存放数据发生hash碰撞（当前key计算的hash值换算出来的数组下标位置已经存在值）**

[内容引用](https://www.cnblogs.com/yanzige/p/8392142.html)https://www.cnblogs.com/yanzige/p/8392142.html

5.2. 源码分析

先看一下什么时候，resize();

首先是put()方法,在put()方法中有调用addEntry()方法，这个方法里面是具体的存值，在存值之前还有判断是否需要扩容,如果需要扩容，调用扩容的方法resize().

public V put(K key, V value) {  
　　　　//判断当前Hashmap(底层是Entry数组)是否存值（是否为空数组）  
　　　　if (table == EMPTY\_TABLE) {  
　　　　　　inflateTable(threshold);//如果为空，则初始化  
　　　　}  
　　　　  
　　　　//判断key是否为空  
　　　　if (key == null)  
　　　　　　return putForNullKey(value);//hashmap允许key为空  
　　　　  
　　　　//计算当前key的哈希值　　　　  
　　　　int hash = hash(key);  
　　　　//通过哈希值和当前数据长度，算出当前key值对应在数组中的存放位置  
　　　　int i = indexFor(hash, table.length);  
　　　　for (Entry<K,V> e = table[i]; e != null; e = e.next) {  
　　　　　　Object k;  
　　　　　　//如果计算的哈希位置有值（及hash冲突），且key值一样，则覆盖原值value，并返回原值value  
　　　　　　if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {  
　　　　　　　　V oldValue = e.value;  
　　　　　　　　e.value = value;  
　　　　　　　　e.recordAccess(this);  
　　　　　　　　return oldValue;  
　　　　　　}  
　　　　}  
   
　　　　modCount++;  
　　　　//存放值的具体方法  
　　　　addEntry(hash, key, value, i);  
　　　　return null;  
　　}  
/\*\*   
 \* HashMap 添加节点   
 \*   
 \* @param hash 当前key生成的hashcode   
 \* @param key 要添加到 HashMap 的key   
 \* @param value 要添加到 HashMap 的value   
 \* @param bucketIndex 桶，也就是这个要添加 HashMap 里的这个数据对应到数组的位置下标   
 \*/   
void addEntry(int hash, K key, V value, int bucketIndex) {   
 //1、判断当前个数是否大于等于阈值  
　　　　//2、当前存放是否发生哈希碰撞  
　　　　//如果上面两个条件都发生，那么就扩容  
 //size：The number of key-value mappings contained in this map.   
 //threshold：The next size value at which to resize (capacity \* load factor)   
 //数组扩容条件：1.已经存在的key-value mappings的个数大于等于阈值   
 // 2.底层数组的bucketIndex坐标处不等于null   
 if ((size >= threshold) && (null != table[bucketIndex])) {   
 resize(2 \* table.length);//扩容之后，数组长度变了   
 hash = (null != key) ? hash(key) : 0;//为什么要再次计算一下hash值呢？   
 bucketIndex = indexFor(hash, table.length);//扩容之后，数组长度变了，在数组的下标跟数组长度有关，得重算。   
 }   
 createEntry(hash, key, value, bucketIndex);   
}   
   
/\*\*   
 \* 这地方就是链表出现的地方，有2种情况   
 \* 1，原来的桶bucketIndex处是没值的，那么就不会有链表出来啦   
 \* 2，原来这地方有值，那么根据Entry的构造函数，把新传进来的key-value mapping放在数组上，原来的就挂在这个新来的next属性上了   
 \*/   
void createEntry(int hash, K key, V value, int bucketIndex) {   
 HashMap.Entry<K, V> e = table[bucketIndex];   
 table[bucketIndex] = new HashMap.Entry<>(hash, key, value, e);   
 size++;   
}

我们分析下resize的源码，鉴于JDK1.8融入了红黑树，较复杂，为了便于理解我们仍然使用JDK1.7的代码，好理解一些，本质上区别不大，具体区别后文再说。

void resize(int newCapacity) { //传入新的容量  
 Entry[] oldTable = table; //引用扩容前的Entry数组  
 int oldCapacity = oldTable.length;  
 if (oldCapacity == MAXIMUM\_CAPACITY) { //扩容前的数组大小如果已经达到最大(2^30)了  
 threshold = Integer.MAX\_VALUE; //修改阈值为int的最大值(2^31-1),这样就不会扩容了  
 return;  
 }  
   
 Entry[] newTable = new Entry[newCapacity]; //初始化一个新的Entry数组  
 transfer(newTable); //！！将数据转移到新的Entry数组里  
 table = newTable; //HashMap的table属性引用新的Entry数组  
 threshold = (int) (newCapacity \* loadFactor);//修改阈值  
 }

这里就是使用一个容量更大的数组来代替已有的容量小的数组，transfer()方法将***原有Entry数组的元素拷贝到新的Entry数组里***。(transfer()在实际扩容时候把原来数组中的元素放入新的数组中)

void transfer(Entry[] newTable) {  
 Entry[] src = table; //src引用了旧的Entry数组  
 int newCapacity = newTable.length;  
 for (int j = 0; j < src.length; j++) { //遍历旧的Entry数组  
 Entry<K, V> e = src[j]; //取得旧Entry数组的每个元素  
 if (e != null) {  
 src[j] = null;//释放旧Entry数组的对象引用(for循环后,旧的Entry数组不再引用任何对象）  
 do {  
 Entry<K, V> next = e.next;  
 　　　　　　　　//通过key值的hash值和新数组的大小算出在当前数组中的存放位置  
 int i = indexFor(e.hash, newCapacity);//！！重新计算每个元素在数组中的位置  
 e.next = newTable[i]; //标记[1]  
 newTable[i] = e; //将元素放在数组上  
 e = next; //访问下一个Entry链上的元素  
 } while (e != null);  
 }  
 }  
 }  
   
 static int indexFor(int h, int length) {  
 return h & (length - 1);  
 }

newTable[i]的引用赋给了e.next，也就是使用了**单链表的头插入方式**，同一位置上新元素总会被放在链表的头部位置；这样先放在一个索引上的元素终会被放到Entry链的尾部(**如果发生了hash冲突的**话），这一点和Jdk1.8有区别，下文详解。在旧数组中同一条Entry链上的元素，通过重新计算索引位置后，有可能被放到了新数组的不同位置上。

**从上面的for循环内部开始说起吧:**

详细解释下，这个转存的过程和头插入法.  
Entry<K, V> e = src[j];  
这句话，就把原来数组上的那个链表的引用就给接手了，所以下面src[j] = null;可以放心大胆的置空，释放空间。告诉gc这个地方可以回收啦。  
继续到do while 循环里面，  
Entry<K, V> next = e.next;  
int i = indexFor(e.hash, newCapacity);

计算出元素在新数组中的位置  
下面就是单链表的头插入方式转存元素啦

**关于这个 单链表的头插入方式 的理解，我多说两句。**  
**这地方我再看的时候，就有点蒙了，他到底怎么在插到新的数组里面的？**  
**要是在插入新数组的时候，也出现了一个数组下标的位置处，出现了多个节点的话，那又是怎么插入的呢？**  
**1，假设现在刚刚插入到新数组上，因为是对象数组，数组都是要默认有初始值的，那么这个数组的初始值都是null。不信的可以新建个Javabean数组测试下。**  
**那么e.next = newTable[i],也就是e.next = null啦。然后再newTable[i] = e;也就是 说这个时候，这个数组的这个下标位置的值设置成这个e啦。**  
**2，假设这个时候，继续上面的循环，又取第二个数据e2的时候，恰好他的下标和刚刚上面的那个下标相同啦，那么这个时候，是又要有链表产生啦、**  
**e.next = newTable[i];，假设上面第一次存的叫e1吧，那么现在e.next = newTable[i];也就是e.next = e1；**  
**然后再，newTable[i] = e;，把这个后来的赋值在数组下标为i的位置，当然他们两个的位置是相同的啦。然后注意现在的e，我们叫e2吧。e2.next指向的是刚刚的e1，e1的next是null。**  
**这就解释啦：先放在一个索引上的元素终会被放到Entry链的尾部。这句话。**

关于什么时候resize（）的说明：

看1.7的源码上说的条件是：

if ((size >= threshold) && (null != table[bucketIndex])) {。。。}   
其中   
size表示当前hashmap里面已经包含的元素的个数。   
threshold：threshold = (int) Math.min(capacity \* loadFactor, MAXIMUM\_CAPACITY + 1);   
一般是容量值X加载因子。

而1.8的是：

if (++size > threshold){}   
其中   
size：The number of key-value mappings contained in this map.和上面的是一样的   
threshold：newThr = (int)(DEFAULT*LOAD*FACTOR \* DEFAULT*INITIAL*CAPACITY);  
也是一样的，   
最后总结一下：就是这个map里面包含的元素，也就是size的值，大于等于这个阈值的时候，才会resize（）；   
具体到实际情况就是：假设现在阈值是4；在添加下一个假设是第5个元素的时候，这个时候的size还是原来的，还没加1，size=4，那么阈值也是4的时候,当执行put方法，添加第5个的时候，这个时候，4 >= 4。元素个数等于阈值。就要resize（）啦。添加第4的时候，还是3 >= 4不成立，不需要resize（）。   
经过这番解释，可以发现下面的这个例子，不应该是在添加第二个的时候resize（），而是在添加第三个的时候，才resize（）的。   
**这个也是我后来再细看的时候，发现的。当然，这个咱可以先忽略，重点看如何resize（），以及如何将旧数据移动到新数组的**

下面举个例子说明下扩容过程。

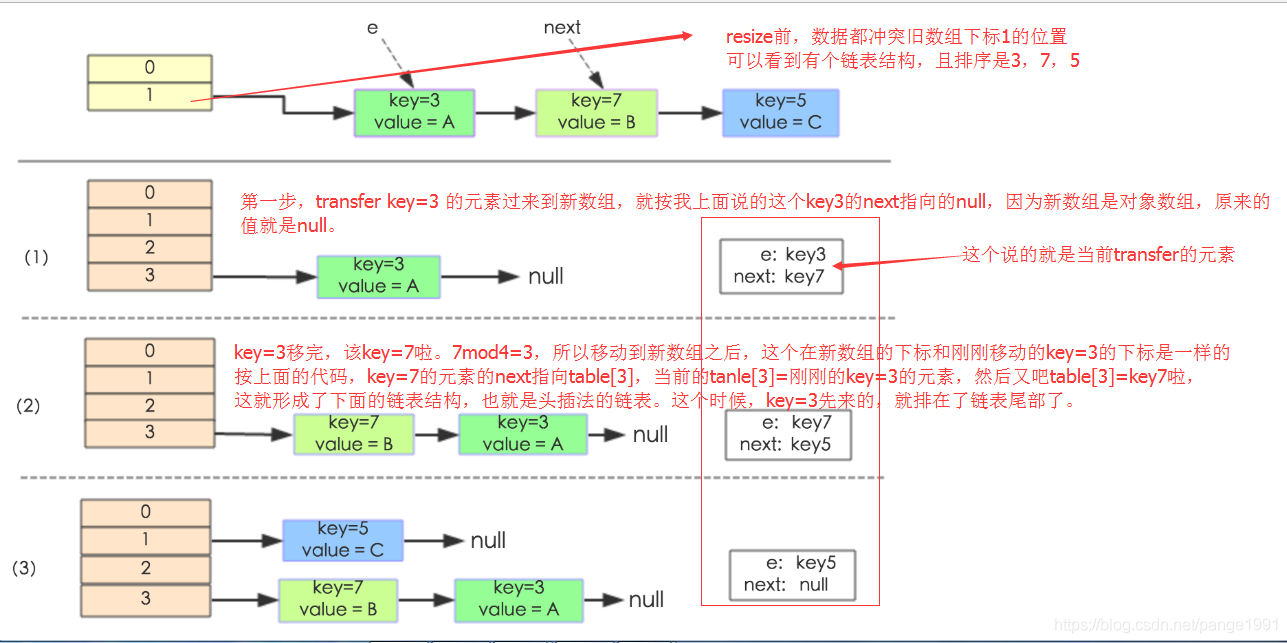
这句话是重点----

hash(){  
  
 return key % table.length;  
}

方法,就是翻译下面的一行解释：

假设了我们的hash算法就是简单的用key mod 一下表的大小（也就是数组的长度）。

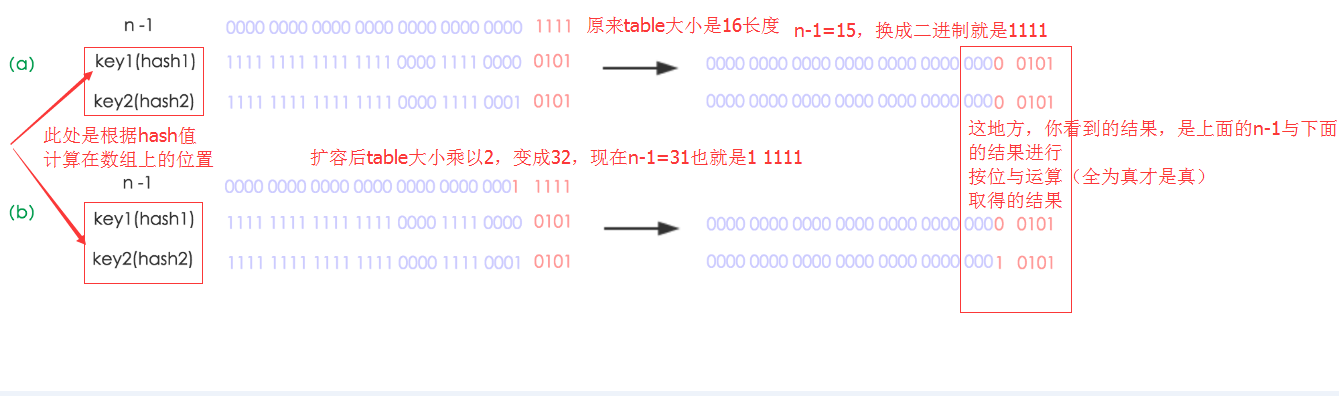
其中的哈希桶数组table的size=2， 所以key = 3、7、5，put顺序依次为 5、7、3。在mod 2以后都冲突在table[1]这里了。这里假设负载因子 loadFactor=1，即当键值对的实际大小size 大于 table的实际大小时进行扩容。接下来的三个步骤是哈希桶数组 resize成4，然后所有的Node重新rehash的过程。

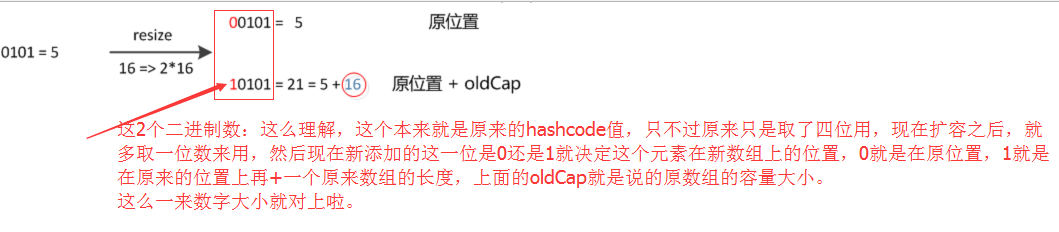


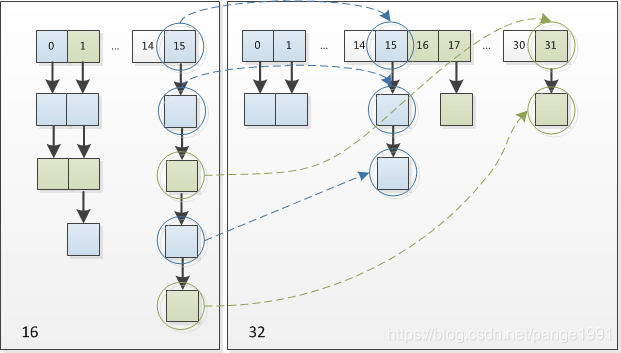
下面我们讲解下JDK1.8做了哪些优化。经过观测可以发现，我们使用的是2次幂的扩展(指长度扩为原来2倍)，所以，

经过rehash之后，元素的位置要么是在原位置，要么是在原位置再移动2次幂的位置。对应的就是下方的resize的注释。

/\*\*   
 \* Initializes or doubles table size. If null, allocates in   
 \* accord with initial capacity target held in field threshold.   
 \* Otherwise, because we are using power-of-two expansion, the   
 \* elements from each bin must either stay at same index, or move   
 \* with a power of two offset in the new table.   
 \*   
 \* @return the table   
 \*/   
final Node<K,V>[] resize() { }

看下图可以明白这句话的意思，n为table的长度，图（a）表示扩容前的key1和key2两种key确定索引位置的示例，图（b）表示扩容后key1和key2两种key确定索引位置的示例，其中hash1是key1对应的哈希值(也就是根据key1算出来的hashcode值)与高位与运算的结果。  
  
元素在重新计算hash之后，因为n变为2倍，那么n-1的mask范围在高位多1bit(红色)，因此新的index就会发生这样的变化：



因此，我们在扩充HashMap的时候，不需要像JDK1.7的实现那样重新计算hash，只需要看看原来的hash值新增的那个bit是1还是0就好了，是0的话索引没变，是1的话索引变成“原索引+oldCap”，可以看看下图为16扩充为32的resize示意图：  


这个设计确实非常的巧妙，既省去了重新计算hash值的时间，而且同时，由于新增的1bit是0还是1可以认为是随机的，因此resize的过程，均匀的把之前的冲突的节点分散到新的bucket了。这一块就是JDK1.8新增的优化点。有一点注意区别，JDK1.7中rehash的时候，旧链表迁移新链表的时候，如果在新表的数组索引位置相同，则链表元素会倒置，但是从上图可以看出，JDK1.8不会倒置。有兴趣的同学可以研究下JDK1.8的resize源码，写的很赞，如下:

final Node<K,V>[] resize() {  
 Node<K,V>[] oldTab = table;  
 int oldCap = (oldTab == null) ? 0 : oldTab.length;  
 int oldThr = threshold;  
 int newCap, newThr = 0;  
 if (oldCap > 0) {  
 // 超过最大值就不再扩充了，就只好随你碰撞去吧  
 if (oldCap >= MAXIMUM\_CAPACITY) {  
 threshold = Integer.MAX\_VALUE;  
 return oldTab;  
 }  
 // 没超过最大值，就扩充为原来的2倍  
 else if ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY &&  
 oldCap >= DEFAULT\_INITIAL\_CAPACITY)  
 newThr = oldThr << 1; // double threshold  
 }  
 else if (oldThr > 0) // initial capacity was placed in threshold  
 newCap = oldThr;  
 else { // zero initial threshold signifies using defaults  
 newCap = DEFAULT\_INITIAL\_CAPACITY;  
 newThr = (int)(DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY);  
 }  
 // 计算新的resize上限  
 if (newThr == 0) {  
   
 float ft = (float)newCap \* loadFactor;  
 newThr = (newCap < MAXIMUM\_CAPACITY && ft < (float)MAXIMUM\_CAPACITY ?  
 (int)ft : Integer.MAX\_VALUE);  
 }  
 threshold = newThr;  
 @SuppressWarnings({"rawtypes"，"unchecked"})  
 Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];  
 table = newTab;  
 if (oldTab != null) {  
 // 把每个bucket都移动到新的buckets中  
 for (int j = 0; j < oldCap; ++j) {  
 Node<K,V> e;  
 if ((e = oldTab[j]) != null) {  
 oldTab[j] = null;  
 if (e.next == null)  
 newTab[e.hash & (newCap - 1)] = e;  
 else if (e instanceof TreeNode) //如果是红黑树节点  
 ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);  
 else { // 链表优化重hash的代码块  
 Node<K,V> loHead = null, loTail = null;  
 Node<K,V> hiHead = null, hiTail = null;  
 Node<K,V> next;  
 do {  
 next = e.next;  
 // 原索引  
 if ((e.hash & oldCap) == 0) {  
 if (loTail == null)  
 loHead = e;  
 else  
 loTail.next = e;  
 loTail = e;  
 }  
 // 原索引+oldCap  
 else {  
 if (hiTail == null)  
 hiHead = e;  
 else  
 hiTail.next = e;  
 hiTail = e;  
 }  
 } while ((e = next) != null);  
 // 原索引放到bucket里  
 if (loTail != null) {  
 loTail.next = null;  
 newTab[j] = loHead;  
 }  
 // 原索引+oldCap放到bucket里  
 if (hiTail != null) {  
 hiTail.next = null;  
 newTab[j + oldCap] = hiHead;  
 }  
 }  
 }  
 }  
 }  
 return newTab;  
}

先略过红黑树的情况，描述下简单流程，在JDK1.8中发生hashmap扩容时，遍历hashmap每个bucket里的链表，每个链表可能会被拆分成两个链表，不需要移动的元素置入loHead为首的链表，需要移动的元素置入hiHead为首的链表，然后分别分配给老的buket和新的buket。

5.3.**补充一下jdk 1.8中，HashMap扩容时红黑树的表现**

扩容时，如果节点是红黑树节点，就会调用TreeNode的split方法对**当前节点作为根节点的红黑树进行修剪**

...  
else if (e instanceof TreeNode) //如果是红黑树节点  
 ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);  
...  
  
 //参数介绍  
 //tab 表示保存桶头结点的哈希表  
 //index 表示从哪个位置开始修剪  
 //bit 要修剪的位数（哈希值）  
 final void split(HashMap<K,V> map, Node<K,V>[] tab, int index, int bit) {  
 TreeNode&lt;K,V&gt; b = this;  
 // Relink into lo and hi lists, preserving order  
 TreeNode<K,V> loHead = null, loTail = null;  
 TreeNode<K,V> hiHead = null, hiTail = null;  
 int lc = 0, hc = 0;  
 for (TreeNode<K,V> e = b, next; e != null; e = next) {  
 next = (TreeNode<K,V>)e.next;  
 e.next = null;  
 //如果当前节点哈希值的最后一位等于要修剪的 bit 值  
 if ((e.hash &amp; bit) == 0) {  
 //就把当前节点放到 lXXX 树中  
 if ((e.prev = loTail) == null)  
 loHead = e;  
 else  
 loTail.next = e;  
 //然后 loTail 记录 e  
 loTail = e;  
 //记录 lXXX 树的节点数量  
 ++lc;  
 }  
 else { //如果当前节点哈希值最后一位不是要修剪的  
 //就把当前节点放到 hXXX 树中  
 if ((e.prev = hiTail) == null)  
 hiHead = e;  
 else  
 hiTail.next = e;  
 hiTail = e;  
 //记录 hXXX 树的节点数量  
 ++hc;  
 }  
 }  
   
   
 if (loHead != null) {  
 //如果 lXXX 树的数量小于 6，就把 lXXX 树的枝枝叶叶都置为空，变成一个单节点  
 //然后让这个桶中，要还原索引位置开始往后的结点都变成还原成链表的 lXXX 节点  
 //这一段元素以后就是一个链表结构  
 if (lc <= UNTREEIFY\_THRESHOLD)  
 tab[index] = loHead.untreeify(map);  
 else {  
 //否则让索引位置的结点指向 lXXX 树，这个树被修剪过，元素少了  
 tab[index] = loHead;  
 if (hiHead != null) // (else is already treeified)  
 loHead.treeify(tab);  
 }  
 }  
 if (hiHead != null) {  
 //同理，让 指定位置 index + bit 之后的元素  
 //指向 hXXX 还原成链表或者修剪过的树  
 if (hc <= UNTREEIFY\_THRESHOLD)  
 tab[index + bit] = hiHead.untreeify(map);  
 else {  
 tab[index + bit] = hiHead;  
 if (loHead != null)  
 hiHead.treeify(tab);  
 }  
 }  
 }

从上述代码可以看到，HashMap 扩容时对红黑树节点的修剪主要分两部分，先分类、再根据元素个数决定是还原成链表还是精简一下元素仍保留红黑树结构。

1.分类

指定位置、指定范围，让指定位置中的元素 （hash & bit) == 0 的，放到 lXXX 树中，不相等的放到 hXXX 树中。

2.根据元素个数决定处理情况

符合要求的元素（即 lXXX 树），在元素个数小于等于 6 时还原成链表，最后让哈希表中修剪的桶 tab[index] 指向 lXXX 树；在元素个数大于 6 时，还是用红黑树，只不过是修剪了下枝叶；

不符合要求的元素（即 hXXX 树）也是一样的操作，只不过最后它是放在了修剪范围外 tab[index + bit]。

5.4. **小结**

[引用](https://www.cnblogs.com/yanzige/p/8392142.html)https://www.cnblogs.com/yanzige/p/8392142.html

Hashmap的扩容需要满足两个条件：**当前数据存储的数量（即size()）大小必须大于等于阈值；当前加入的数据是否发生了hash冲突。**

因为上面这两个条件，所以存在下面这些情况

（1）、就是hashmap在存值的时候（默认大小为16，负载因子0.75，阈值12），可能达到最后存满16个值的时候，再存入第17个值才会发生扩容现象，因为前16个值，每个值在底层数组中分别占据一个位置，并没有发生hash碰撞。

（2）、当然也有可能存储更多值（超多16个值，最多可以存26个值）都还没有扩容。原理：前11个值全部hash碰撞，存到数组的同一个位置（这时元素个数小于阈值12，不会扩容），后面所有存入的15个值全部分散到数组剩下的15个位置（这时元素个数大于等于阈值，但是每次存入的元素并没有发生hash碰撞，所以不会扩容），前面11+15=26，所以在存入第27个值的时候才同时满足上面两个条件，这时候才会发生扩容现象。

(1) 扩容是一个特别耗性能的操作，所以当程序员在使用HashMap的时候，估算map的大小，初始化的时候给一个大致的数值，避免map进行频繁的扩容。

(2) 负载因子是可以修改的，也可以大于1，但是建议不要轻易修改，除非情况非常特殊。

(3) HashMap是线程不安全的，不要在并发的环境中同时操作HashMap，建议使用ConcurrentHashMap。

(4) JDK1.8引入红黑树大程度优化了HashMap的性能。

(5) 还没升级JDK1.8的，现在开始升级吧。HashMap的性能提升仅仅是JDK1.8的冰山一角。

### 6. List、Map、Set 三个接口，存取元素时，各有什么特点？

[博客地址](https://www.cnblogs.com/guweiwei/p/6635530.html) -答案来源https://www.cnblogs.com/guweiwei/p/6635530.html

概述：

1、Set里面不允许有重复的元素

2、List表示有先后顺序的集合

3、Map是双列的集合

答：List与Set都是单列元素的集合，它们有一个共同的父接口Collection。

1、Set里面不允许有重复的元素，

存元素：add方法有一个boolean的返回值，当集合中没有某个元素，此时add方法可成功加入该元素时，则返回true；当集合含有与某个元素equals相等的元素时，此时add方法无法加入该元素，返回结果为false。

取元素：没法说取第几个，只能以Iterator接口取得所有的元素，再逐一遍历各个元素。  
2、List表示有先后顺序的集合，

存元素：多次调用add(Object)方法时，每次加入的对象按先来后到的顺序排序，也可以插队，即调用add(int index,Object)方法，就可以指定当前对象在集合中的存放位置。

取元素：方法1：Iterator接口取得所有，逐一遍历各个元素

方法2：调用get(index i)来明确说明取第几个。

3、Map是双列的集合，存放用put方法:put(obj key,obj value)，每次存储时，要存储一对key/value，不能存储重复的key，这个重复的规则也是按equals比较相等。

取元素：用get(Object key)方法根据key获得相应的value。

也可以获得所有的key的集合，还可以获得所有的value的集合，

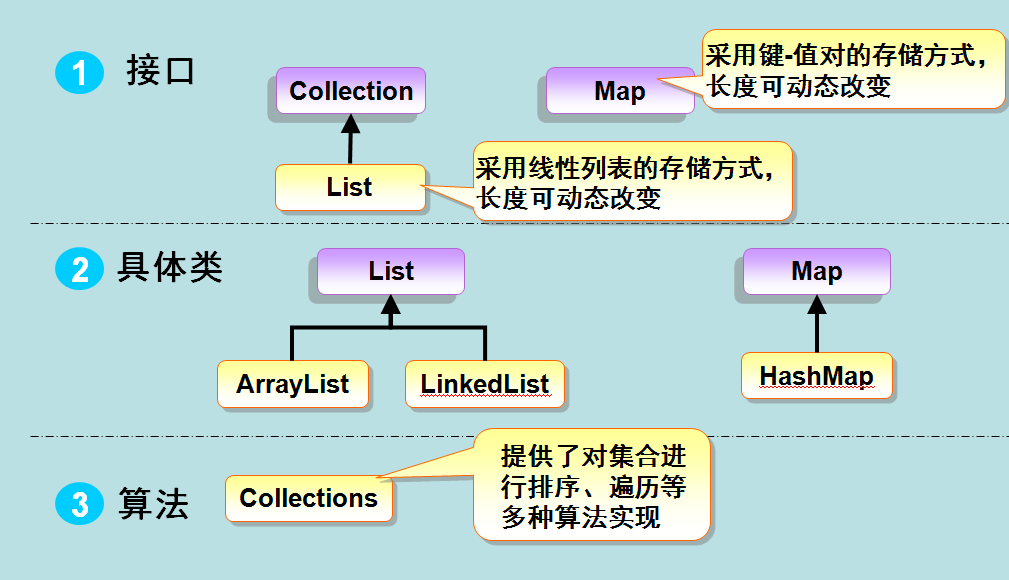
还可以获得key和value组合成的Map.Entry对象的集合。

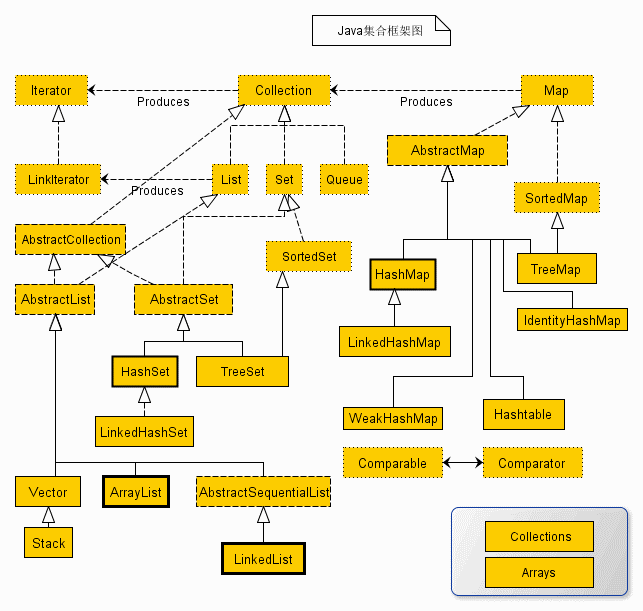
总结：

List以特定次序来持有元素，可有重复元素。Set 无法拥有重复元素,内部排序。Map 保存key-value值，value可多值。

另附了三个集合相关的图片

图片来源：https://www.runoob.com/java/java-collections.html





### 7.Set 里的元素是不能重复的，那么用什么方法来区分重复与否呢? 是用 == 还是 equals()? 它们有何区别?

[答案来自博客地址](https://www.cnblogs.com/ncl-960301-success/p/7631707.html)-小菜鸟的Java心得-https://www.cnblogs.com/ncl-960301-success/)

同题目：19.Set 里的元素是不能重复的，那么用什么方法来区分重复与否呢？是用 == 还是 equals()？它们有何区别？

* 1、概述
* 2、== 和 equals()区别
* 3、总结
* 1、概述

Set 里的元素是不能重复的，元素重复与否是使用 equals()方法进行判断的

==用于比较引用和比较基本数据类型时具有不同的功能：  
比较基本数据类型，如果两个值相同，则结果为true  
而在比较引用时，如果引用指向内存中的同一对象，结果为true;

equals()作为方法，实现对象的比较。由于==运算符不允许我们进行覆盖，也就是说它限制了我们的表达。  
因此我们复写equals()方法，达到比较对象内容是否相同的目的。而这些通过==运算符是做不到的。

set元素不重复的关键就是equals方法。这么说还是不恰当，准确的说应该是equals和hashcode方法。

**2、equals（）和==的区别**

==操作符专门用来比较**两个变量的值**是否相等，也就是用于比较变量**所对应的内存中所存**  
**储的数值**是否相同， 要比较两个基本类型的数据或两个引用变量是否相等，只能用==操作  
符。  
如果一个变量指向的数据是对象类型的，那么，这时候涉及了两块内存， **对象本身占用一块**  
**内存（堆内存），变量引用也占用一块内存**，例如 Objet obj = new Object();变量 obj 是一个内存，  
new Object()是另一个内存，此时，变量 obj 所对应的内存中存储的数值就是**对象占用的那**  
**块内存的首地址**。对于指向对象类型的变量，如果要比较两个变量是否指向同一个对象，即  
要看这两个变量所对应的内存中的数值是否相等，这时候就需要用==操作符进行比较。  
equals 方法是用于比较两个独立对象的内容是否相同，就好比去比较两个人的长相是否相  
同，它比较的两个对象是独立的。例如，对于下面的代码：

String a=new String("foo");  
String b=new String("foo");

两条 new 语句创建了**两个对象**，然后用 a和b 这两个变量分别指向了各自的对象，

这是两个不同的对象，它们的首地址是不同的，即 **a 和 b 中存储的数值是不相同的。**

所以，表达式 a==b 将返回**false**，而这两个对象中的内容是相同的(String重写equals方法)，所以，表达式 a.equals(b)将返回**true**。

String重写equals方法

/\*\*  
 \* Compares this string to the specified object. The result is {@code  
 \* true} if and only if the argument is not {@code null} and is a {@code  
 \* String} object that represents the same sequence of characters as this  
 \* object.   
 \*  
 \* @param anObject  
 \* The object to compare this {@code String} against  
 \*  
 \* @return {@code true} if the given object represents a {@code String}  
 \* equivalent to this string, {@code false} otherwise  
 \*  
 \* @see #compareTo(String)  
 \* @see #equalsIgnoreCase(String)  
 \*/  
 public boolean equals(Object anObject) {  
 //地址相等，一定相等  
 if (this == anObject) {  
 return true;  
 }  
 //传入的对象是String  
 if (anObject instanceof String) {  
 String anotherString = (String)anObject;  
 int n = value.length;  
 //有限比较两者的长度（数组的长度）  
 if (n == anotherString.value.length) {  
 char v1[] = value;  
 char v2[] = anotherString.value;  
 int i = 0;  
 //长度相等，在循环比较，数组中的每个值都相等  
 while (n-- != 0) {  
 if (v1[i] != v2[i])  
 return false;  
 i++;  
 }  
 return true;  
 }  
 }  
 return false;  
 }

在实际开发中，我们经常要比较传递进行来的字符串内容是否等，例如，

String input= “quit”;  
  
input.equals(“quit”)，

许多人稍不注意就使用==进行比较了，这是错误的，随便从网上  
找几个项目实战的教学视频看看，里面就有大量这样的错误。

**记住，字符串的比较基本上都是使用 equals 方法。**  
如果一个类没有自己定义 equals 方法，那么它将继承 Object 类的 equals 方法， Object 类  
的 equals 方法的实现代码如下：

boolean equals(Object o){  
 return this==o;  
}

这说明，如果一个类没有自己定义 equals 方法，它默认的 equals 方法（从 Object 类继承  
的）就是使用==操作符，也是在比较两个变量指向的对象是否是同一对象，这时候使用  
equals 和使用==会得到同样的结果，如果比较的是两个独立的对象则总返回 false。如果你  
编写的类希望能够比较该类创建的两个实例对象的内容是否相同，那么你必须覆盖 equals  
方法，由你自己写代码来决定在什么情况即可认为两个对象的内容是相同的

由上述可见：

3、总结如下：

==：

基本类型：比较的是值是否相同

引用类型：比较的是地址值是否相同

equals（）：

引用类型：默认情况下，比较的是地址值，可进行重写，比较的是对象的成员变量值是否相同。

附录：

**set的源码**

https://blog.csdn.net/qq\_41610623/article/details/82257233

Set 不重复实现原理

转载[qq\_41610623](https://me.csdn.net/qq_41610623) 最后发布于2018-08-31 15:35:59 阅读数 1130 收藏

Java中的set是一个不包含重复元素的集合，确切地说，是不包含e1.equals(e2)的元素对。Set中允许添加null。Set不能保证集合里元素的顺序。

在往set中添加元素时，如果指定元素不存在，则添加成功。也就是说，如果set中不存在(e==null ? e1==null : e.queals(e1))的元素e1,则e1能添加到set中。

下面以set的一个实现类HashSet为例，简单介绍一下set不重复实现的原理：

[java] view plain copy  
import java.util.HashSet;  
import java.util.Iterator;  
import java.util.Set;  
  
/\*\*  
\\* @version 1.0  
\\* @author Oliver  
\\* @since 1.0  
\*/  
publicclass HashSetTest  
{  
 //自定义MyString类  
 staticclass MyString{  
 String value;  
 public MyString(String value)   
 {   
 this.value = value;   
 }   
 }   
 publicstaticvoid main(String[] args)   
 {   
 //创建一个HashSet对象   
 Set<Object> set = new HashSet<Object>();   
 //创建连个String对象   
 String s1 = new String("a");   
 String s2 = new String("a");   
 //创建连个MyString对象   
 MyString s3 = new MyString("a");   
 MyString s4 = new MyString("a");   
 //添加元素   
 set.add(s1);   
 set.add(s2);   
 set.add(s3);   
 set.add(s4);   
 //看看对象的equals   
 System.out.println("s1.equals(s2):"+s1.equals(s2));   
 System.out.println("s3.equals(s4):"+s3.equals(s4));   
 //打印几个大小及里面的元素   
 System.out.println("set size:"+set.size());   
 for(Iterator<Object> it=set.iterator();it.hasNext();){   
 System.out.println(it.next());   
 }   
 }   
   
 }

运行程序，输出结果：  
s1.equals(s2):true

s3.equals(s4):false

set size:3

oliver.examination.part1.HashSetTest$MyString@4f1d0d

a

oliver.examination.part1.HashSetTest$MyString@1fc4bec

也许你已经看出关键来了，没错就是equals方法。这么说还是不恰当，准确的说应该是equals和hashcode方法。

java.lnag.Object中对hashCode的约定：

1. 在一个应用程序执行期间，如果一个对象的equals方法做比较所用到的信息没有被修改的话，则对该对象调用hashCode方法多次，它必须始终如一地返回同一个整数。
2. 如果两个对象根据equals(Object o)方法是相等的，则调用这两个对象中任一对象的hashCode方法必须产生相同的整数结果。
3. 如果两个对象根据equals(Object o)方法是不相等的，则调用这两个对象中任一个对象的hashCode方法，不要求产生不同的整数结果。但如果能不同，则可能提高散列表的性能。

根据第一条，s1和s2返回的hashcode值是一样的。

在HashSet中，基本的操作都是有HashMap底层实现的，因为HashSet底层是用HashMap存储数据的。当向HashSet中添加元素的时候，首先计算元素的hashcode值，然后用这个（元素的hashcode）%（HashMap集合的大小）+1计算出这个元素的存储位置，如果这个位置位空，就将元素添加进去；如果不为空，则用equals方法比较元素是否相等，相等就不添加，否则找一个空位添加。

会后，附赠HashSet源码中文注释版,摘自javaeye：http://xifangyuhui.javaeye.com/blog/798796

[java] view plain copy

public class HashSet  
extends AbstractSet  
implements Set, Cloneable, java.io.Serializable  
{  
 static final long serialVersionUID = -5024744406713321676L;  
 // 底层使用HashMap来保存HashSet中所有元素。   
 private transient HashMap<E,Object> map;   
  
 // 定义一个虚拟的Object对象作为HashMap的value，将此对象定义为static final。   
 private static final Object PRESENT = new Object();   
  
 /\*\*   
 \* 默认的无参构造器，构造一个空的HashSet。   
 \*   
 \* 实际底层会初始化一个空的HashMap，并使用默认初始容量为16和加载因子0.75。   
 \*/   
 public HashSet() {   
 map = new HashMap<E,Object>();   
 }   
  
 /\*\*   
 \* 构造一个包含指定collection中的元素的新set。   
 \*   
 \* 实际底层使用默认的加载因子0.75和足以包含指定   
 \* collection中所有元素的初始容量来创建一个HashMap。   
 \* @param c 其中的元素将存放在此set中的collection。   
 \*/   
 public HashSet(Collection<? extends E> c) {   
 map = new HashMap<E,Object>(Math.max((int) (c.size()/.75f) + 1, 16));   
 addAll(c);   
 }   
  
 /\*\*   
 \* 以指定的initialCapacity和loadFactor构造一个空的HashSet。   
 \*   
 \* 实际底层以相应的参数构造一个空的HashMap。   
 \* @param initialCapacity 初始容量。   
 \* @param loadFactor 加载因子。   
 \*/   
 public HashSet(int initialCapacity, float loadFactor) {   
 map = new HashMap<E,Object>(initialCapacity, loadFactor);   
 }   
  
 /\*\*   
 \* 以指定的initialCapacity构造一个空的HashSet。   
 \*   
 \* 实际底层以相应的参数及加载因子loadFactor为0.75构造一个空的HashMap。   
 \* @param initialCapacity 初始容量。   
 \*/   
 public HashSet(int initialCapacity) {   
 map = new HashMap<E,Object>(initialCapacity);   
 }   
  
 /\*\*   
 \* 以指定的initialCapacity和loadFactor构造一个新的空链接哈希集合。   
 \* 此构造函数为包访问权限，不对外公开，实际只是是对LinkedHashSet的支持。   
 \*   
 \* 实际底层会以指定的参数构造一个空LinkedHashMap实例来实现。   
 \* @param initialCapacity 初始容量。   
 \* @param loadFactor 加载因子。   
 \* @param dummy 标记。   
 \*/   
 HashSet(int initialCapacity, float loadFactor, boolean dummy) {   
 map = new LinkedHashMap<E,Object>(initialCapacity, loadFactor);   
 }   
  
 /\*\*   
 \* 返回对此set中元素进行迭代的迭代器。返回元素的顺序并不是特定的。   
 \*   
 \* 底层实际调用底层HashMap的keySet来返回所有的key。   
 \* 可见HashSet中的元素，只是存放在了底层HashMap的key上，   
 \* value使用一个static final的Object对象标识。   
 \* @return 对此set中元素进行迭代的Iterator。   
 \*/   
 public Iterator<E> iterator() {   
 return map.keySet().iterator();   
 }   
  
 /\*\*   
 \* 返回此set中的元素的数量（set的容量）。   
 \*   
 \* 底层实际调用HashMap的size()方法返回Entry的数量，就得到该Set中元素的个数。   
 \* @return 此set中的元素的数量（set的容量）。   
 \*/   
 public int size() {   
 return map.size();   
 }   
  
 /\*\*   
 \* 如果此set不包含任何元素，则返回true。   
 \*   
 \* 底层实际调用HashMap的isEmpty()判断该HashSet是否为空。   
 \* @return 如果此set不包含任何元素，则返回true。   
 \*/   
 public boolean isEmpty() {   
 return map.isEmpty();   
 }   
  
 /\*\*   
 \* 如果此set包含指定元素，则返回true。   
 \* 更确切地讲，当且仅当此set包含一个满足(o==null ? e==null : o.equals(e))   
 \* 的e元素时，返回true。   
 \*   
 \* 底层实际调用HashMap的containsKey判断是否包含指定key。   
 \* @param o 在此set中的存在已得到测试的元素。   
 \* @return 如果此set包含指定元素，则返回true。   
 \*/   
 public boolean contains(Object o) {   
 return map.containsKey(o);   
 }   
  
 /\*\*   
 \* 如果此set中尚未包含指定元素，则添加指定元素。   
 \* 更确切地讲，如果此 set 没有包含满足(e==null ? e2==null : e.equals(e2))   
 \* 的元素e2，则向此set 添加指定的元素e。   
 \* 如果此set已包含该元素，则该调用不更改set并返回false。   
 \*   
 \* 底层实际将将该元素作为key放入HashMap。   
 \* 由于HashMap的put()方法添加key-value对时，当新放入HashMap的Entry中key   
 \* 与集合中原有Entry的key相同（hashCode()返回值相等，通过equals比较也返回true），   
 \* 新添加的Entry的value会将覆盖原来Entry的value，但key不会有任何改变，   
 \* 因此如果向HashSet中添加一个已经存在的元素时，新添加的集合元素将不会被放入HashMap中，   
 \* 原来的元素也不会有任何改变，这也就满足了Set中元素不重复的特性。   
 \* @param e 将添加到此set中的元素。   
 \* @return 如果此set尚未包含指定元素，则返回true。   
 \*/   
 public boolean add(E e) {   
 return map.put(e, PRESENT)==null;   
 }   
  
 /\*\*   
 \* 如果指定元素存在于此set中，则将其移除。   
 \* 更确切地讲，如果此set包含一个满足(o==null ? e==null : o.equals(e))的元素e，   
 \* 则将其移除。如果此set已包含该元素，则返回true   
 \* （或者：如果此set因调用而发生更改，则返回true）。（一旦调用返回，则此set不再包含该元素）。   
 \*   
 \* 底层实际调用HashMap的remove方法删除指定Entry。   
 \* @param o 如果存在于此set中则需要将其移除的对象。   
 \* @return 如果set包含指定元素，则返回true。   
 \*/   
 public boolean remove(Object o) {   
 return map.remove(o)==PRESENT;   
 }   
  
 /\*\*   
 \* 从此set中移除所有元素。此调用返回后，该set将为空。   
 \*   
 \* 底层实际调用HashMap的clear方法清空Entry中所有元素。   
 \*/   
 public void clear() {   
 map.clear();   
 }   
  
 /\*\*   
 \* 返回此HashSet实例的浅表副本：并没有复制这些元素本身。   
 \* /  
 public Object clone() {  
 try {  
 HashSet<E> newSet = (HashSet<E>) super.clone();  
 newSet.map = (HashMap<E, Object>) map.clone();  
 return newSet;  
 } catch (CloneNotSupportedException e) {  
 throw new InternalError(e);  
 }  
 }  
 private void writeObject(java.io.ObjectOutputStream s)  
 throws java.io.IOException {  
 // Write out any hidden serialization magic  
 s.defaultWriteObject();  
  
 // Write out HashMap capacity and load factor  
 s.writeInt(map.capacity());  
 s.writeFloat(map.loadFactor());  
  
 // Write out size  
 s.writeInt(map.size());  
  
 // Write out all elements in the proper order.  
 for (E e : map.keySet())  
 s.writeObject(e);  
 }  
  
 /\*\*  
 \* Reconstitute the <tt>HashSet</tt> instance from a stream (that is,  
 \* deserialize it).  
 \*/  
 private void readObject(java.io.ObjectInputStream s)  
 throws java.io.IOException, ClassNotFoundException {  
 // Read in any hidden serialization magic  
 s.defaultReadObject();  
  
 // Read capacity and verify non-negative.  
 int capacity = s.readInt();  
 if (capacity < 0) {  
 throw new InvalidObjectException("Illegal capacity: " +  
 capacity);  
 }  
  
 // Read load factor and verify positive and non NaN.  
 float loadFactor = s.readFloat();  
 if (loadFactor <= 0 || Float.isNaN(loadFactor)) {  
 throw new InvalidObjectException("Illegal load factor: " +  
 loadFactor);  
 }  
  
 // Read size and verify non-negative.  
 int size = s.readInt();  
 if (size < 0) {  
 throw new InvalidObjectException("Illegal size: " +  
 size);  
 }  
  
 // Set the capacity according to the size and load factor ensuring that  
 // the HashMap is at least 25% full but clamping to maximum capacity.  
 capacity = (int) Math.min(size \* Math.min(1 / loadFactor, 4.0f),  
 HashMap.MAXIMUM\_CAPACITY);  
  
 // Create backing HashMap  
 map = (((HashSet<?>)this) instanceof LinkedHashSet ?  
 new LinkedHashMap<E,Object>(capacity, loadFactor) :  
 new HashMap<E,Object>(capacity, loadFactor));  
  
 // Read in all elements in the proper order.  
 for (int i=0; i<size; i++) {  
 @SuppressWarnings("unchecked")  
 E e = (E) s.readObject();  
 map.put(e, PRESENT);  
 }  
 }  
  
 /\*\*  
 \* Creates a <em><a href="Spliterator.html#binding">late-binding</a></em>  
 \* and <em>fail-fast</em> {@link Spliterator} over the elements in this  
 \* set.  
 \*  
 \* <p>The {@code Spliterator} reports {@link Spliterator#SIZED} and  
 \* {@link Spliterator#DISTINCT}. Overriding implementations should document  
 \* the reporting of additional characteristic values.  
 \*  
 \* @return a {@code Spliterator} over the elements in this set  
 \* @since 1.8  
 \*/  
 public Spliterator<E> spliterator() {  
 return new HashMap.KeySpliterator<E,Object>(map, 0, -1, 0, 0);  
 }   
 }

### 8.两个对象值相同 (x.equals(y) == true)，但却可有不同的 hash code，这句话对不对?

答：不对，有相同的 hash code  
这是java语言的定义：   
(1)、对象相等则hashCode一定相等；  
(2)、hashCode相等对象未必相等  
基本规则：

1. 基本变量

* 如果是基本变量，没有hashcode和equals方法，基本变量的比较方式就只有==;

**2. 变量**

* 如果是变量，由于在java中**所有变量定义都是一个指向实际存储的一个句柄**（你可以理解为c++中的指针），在这里**==是比较句柄的地址（你可以理解为指针的存储地址）**，而不是句柄指向的实际内存中的内容，如果要比较实际内存中的内容，那就要用equals方法，但是！！！
* 如果是你**自己定义的一个类，比较自定义类用equals和= =是一样的，都是比较句柄地址**，因为自定义的类是继承于object，而object中的equals就是用==来实现的，你可以看源码。
* 那为什么我们用的String等等类型equals是比较实际内容呢，是因为**String等常用类已经重写了object中的equals方法，让equals来比较实际内容**，你也可以看源码。

**3. hashcode**

* 在一般的应用中你不需要了解hashcode的用法，但当你用到hashmap，hashset等集合类时要注意下hashcode。
* 你想通过一个object的key来拿hashmap的value，hashmap的工作方法是，通过你传入的object的hashcode在内存中找地址，当找到这个地址后再通过equals方法来比较这个地址中的内容是否和你原来放进去的一样，一样就取出value。
* 所以这里要匹配2部分，hashcode和equals。
* 但假如说你new一个object作为key去拿value是永远得不到结果的，因为每次new一个object，这个object的hashcode是永远不同的，所以我们要重写hashcode，你可以令你的hashcode是object中的一个恒量，这样永远可以通过你的object的hashcode来找到key的地址，然后你要重写你的equals方法，使内存中的内容也相等。。。
* 一般来讲，equals这个方法是给用户调用的，如果你想判断2个对象是否相等，你可以重写equals方法，然后在代码中调用，就可以判断他们是否相等了。简单来讲，equals方法主要是用来判断从表面上看或者从内容上看，2个对象是不是相等。举个例子，有个学生类，属性只有姓名和性别，那么我们可以认为只要姓名和性别相等，那么就说这2个对象是相等的。
* hashcode方法一般用户不会去调用，比如在hashmap中，由于key是不可以重复的，他在判断key是不是重复的时候就判断了hashcode这个方法，而且也用到了equals方法。这里不可以重复是说equals和hashcode只要有一个不等就可以了！所以简单来讲，hashcode相当于是一个对象的编码，就好像文件中的md5，他和equals不同就在于他返回的是int型的，比较起来不直观。我们一般在覆盖equals的同时也要覆盖hashcode，让他们的逻辑一致。举个例子，还是刚刚的例子，如果姓名和性别相等就算2个对象相等的话，那么hashcode的方法也要返回姓名的hashcode值加上性别的hashcode值，这样从逻辑上，他们就一致了。
* 要从物理上判断2个对象是否相等，用==就可以了

补充：关于equals和hashCode方法，很多Java程序员都知道，但很多人也就是仅仅知道而已，在Joshua Bloch的大作《Effective Java》（很多软件公司，《Effective Java》、《Java编程思想》以及《重构：改善既有代码质量》是Java程序员必看书籍，如果你还没看过，那就赶紧去亚马逊买一本吧）中是这样介绍equals方法的：

首先equals方法必须满足:

1. 反性（x.equals(x)必须返回true）;
2. 对称性（x.equals(y)返回true时，y.equals(x)也必须返回true);
3. 传递性（x.equals(y)和y.equals(z)都返回true时，x.equals(z)也必须返回true);
4. 一致性（当x和y引用的对象信息没有被修改时，多次调用x.equals(y)应该得到同样的返回值），而且对于任何非null值的引用x，x.equals(null)必须返回false。

实现高质量的equals方法的诀窍包括：

1. 使用==操作符检查”参数是否为这个对象的引用”；
2. 使用instanceof操作符检查”参数是否为正确的类型”；
3. 对于类中的关键属性，检查参数传入对象的属性是否与之相匹配；
4. 编写完equals方法后，问自己它是否满足对称性、传递性、一致性；
5. 重写equals时总是要重写hashCode；
6. 不要将equals方法参数中的Object对象替换为其他的类型，在重写时不要忘掉@Override注解。

### 9.heap 和 stack 有什么区别。

答案：https://blog.csdn.net/u014306011/article/details/51044091

1、堆栈的概念

2、堆和栈的区别

2.1堆栈空间分配区别

2.2堆栈缓存方式区别

2.3堆栈数据结构区别

3、Java中栈和堆的区别

4、Java中变量在内存中的分配

**1、堆栈的概念：**

堆栈是两种数据结构。堆栈都是一种数据项按序排列的数据结构，只能在一端(称为栈顶(top))对数据项进行插入和删除。在单片机应用中，堆栈是个**特殊的存储区**，主要功能是暂时存放数据和地址，通常用来保护断点和现场。

要点：堆，队列优先,**先进先出**（FIFO—first in first out）。栈，**先进后出**(FILO—First-In/Last-Out)。

**2、堆和栈的区别：**

**2.1、堆栈空间分配区别：**

1、栈（操作系统）：由操作系统自动分配释放 ，存放函数的参数值，局部变量的值等。其操作方式类似于数据结构中的栈；  
 2、堆（操作系统）： 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收，分配方式倒是类似于链表。

**2.2、堆栈缓存方式区别：**  
 1、栈使用的是一级缓存， 他们通常都是被调用时处于存储空间中，调用完毕立即释放；  
 2、堆是存放在二级缓存中，生命周期由虚拟机的垃圾回收算法来决定（并不是一旦成为孤儿对象就能被回收）。所以调用这些对象的速度要相对来得低一些。

**2.3、堆栈数据结构区别：**  
 堆（数据结构）：堆可以被看成是一棵树，如：堆排序；  
 栈（数据结构）：一种先进后出的数据结构。

**3、Java中栈和堆的区别：**  
 栈(stack)与堆(heap)都是Java用来在Ram中存放数据的地方。与C++不同，Java自动管理栈和堆，程序员不能直接地设置栈或堆。  
 在函数中定义的一些**基本类型的变量**和**对象的引用变量**都在函数的栈内存中分配。当在一段代码块定义一个变量时，Java就在栈中为这个变量分配内存空间，当超过变量的作用域后，Java会自动释放掉为该变量所分配的内存空间，该内存空间可以立即被另作他用。  
 堆内存用来存放由new创建的对象和数组，在堆中分配的内存，由Java虚拟机的自动垃圾回收器来管理。在堆中产生了一个数组或对象后，还可以在栈中定义一个特殊的变量，让栈中这个变量的取值等于数组或对象在堆内存中的首地址，栈中的这个变量就成了数组或对象的引用变量。引用变量就相当于是为数组或对象起的一个名称，以后就可以在程序中使用栈中的引用变量来访问堆中的数组或对象。  
**4、Java中变量在内存中的分配：**  
 1、类变量（static修饰的变量）：在**程序加载时系统就为它在堆中开辟了内存**，堆中的内存地址存放于栈以便于高速访问。静态变量的生命周期–一直持续到整个”系统”关闭。  
 2、实例变量：当你使用java关键字new的时候，系统在堆中开辟并不一定是连续的空间分配给变量（比如说类实例），然后根据零散的堆内存地址，通过哈希算法换算为一长串数字以表征这个变量在堆中的”物理位置”。 实例变量的生命周期–当实例变量的引用丢失后，将被GC（垃圾回收器）列入可回收“名单”中，但并不是马上就释放堆中内存。  
 3、局部变量：局部变量，由声明在某方法，或某代码段里（比如for循环），执行到它的时候在栈中开辟内存，当局部变量一但脱离作用域，内存立即释放。  
这里要涉及到Java内存问题，可以参考：Java的内存机制

顺便说一句：在2016年腾讯校招笔试中出现过这道论述题。(一般人我还不告诉他，哈哈哈，见笑了）  
————————————————  
版权声明：本文为CSDN博主「天命王子」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/u014306011/article/details/51044091

### 10.Java 集合类框架的基本接口有哪些？

答案：https://blog.csdn.net/qq\_18433441/article/details/78221810

Collection：代表一组对象，每一个对象都是它的子元素

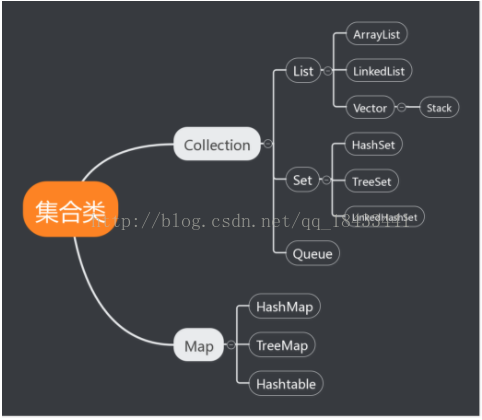
Set：不包括重复元素的Collection

List：有顺序的Collection，并且可以包含重复元素

Map：可以把键（key）映射到值（value）的对象，键不能重复

下面是详细解释：

转自：[牛客网](https://www.nowcoder.com/questionTerminal/709828a56a4443d899d89ae95b7c2940)



（一）总共有两大接口：Collection和Map，一个是元素集合，一个是键值对集合。

（二）其中List接口和Set接口继承了Collection接口，一个是有序元素集合，一个是无序元素集合

（三）ArrayList类和LinkList类实现了List接口

（3.1）ArrayList底层采用数组存储，因此适合查询，不适合增删

（3.2）LinkList底层采用双向链表，适合增删，不适合查询

（四）HashSet（哈希表、散列表）实现了Set接口，底层是哈希表，由HashMap实现。

（五）TreeSet实现了SortedSet接口（图上没画出来）。无序，不可重复，但可按照元素大小自动排序，或者自定义排序方法，底层实现，继承了HashSet,实现了SortedSet接口.

（六）HashMap和HashTable实现了Map接口，其中HashTable是线程安全的，但是HashMap性能更好

（七）TreeMap实现了SortedMap接口（图上没画出来）。无序，不可重复，但可按照元素大小自动排序或者自定义排序方法

### 11.HashSet 和 TreeSet 有什么区别？

答案https://blog.csdn.net/xiangyuenacha/article/details/84255629

**1、HashSet：**  
不能保证元素的排列顺序，顺序有可能发生变化  
集合元素可以是null,但只能放入一个null  
HashSet底层是采用HashMap实现的，HashSet底层是哈希表实现的

**2、TreeSet：**  
Treeset中的数据是排好序的，不允许放入null值。  
TreeSet是通过TreeMap实现的,只不过Set用的只是Map的key。  
TreeSet的底层实现是采用二叉树（红-黑树）的数据结构。

**3、测试类**

参考代码： HashSetTest TreeSetTest TreeSetStudentTest

// Set的使用(不允许有重复的对象),可以是null,但只能放入一个null  
public class HashSetTest {  
 public static void main(String[] args) {  
 HashSet<String> hashSet = new HashSet<String>();  
 String a = new String("A");  
 String b = new String("B");  
 String E = new String("E");  
 String F = new String("F");  
 String c = new String("B");  
 String h = null;  
 String i = null;  
 hashSet.add(a);  
 hashSet.add(b);  
 hashSet.add(E);  
 hashSet.add(F);  
 hashSet.add(h);  
 hashSet.add(i);  
 System.out.println("hashSet得长度：" + hashSet.size());  
   
  
 // 元素B已经存在，插入时不在插入  
 String cz = hashSet.add(c) ? "此对象不存在" : "已经存在";  
 System.out.println("测试是否可以添加对象: " + cz);  
   
 // 测试其中是否已经包含某个对象  
 System.out.println("hashSet中包含A：" + hashSet.contains("A"));  
 Iterator<String> iter = hashSet.iterator();  
 while (iter.hasNext()) {  
 System.out.println("hashSet中的元素：" + iter.next());  
 }  
 // 测试某个对象是否可以删除  
 System.out.println("删除不存在的元素e:" + hashSet.remove("e"));  
 System.out.println("删除存在的元素E:" + hashSet.remove("E"));  
 System.out.println();  
 // 如果你想再次使用iter变量,必须重新更新,重新获取  
 iter = hashSet.iterator();  
 while (iter.hasNext()) {  
 System.out.println("删除存在的元素E后:" + iter.next());  
 }  
 }  
  
}

// 有序的  
public class TreeSetTest {  
   
  
 public static void main(String[] args) {  
 TreeSet<String> tree = new TreeSet<String> ();  
 tree.add("China");  
 tree.add("America");  
 tree.add("Japan");  
 tree.add("Chinese");  
 tree.add("Diio");  
 Iterator<String> iter = tree.iterator();  
 while (iter.hasNext()) {  
 System.out.println(iter.next());  
 }  
 }  
  
}

/\*\*  
  
 \* 测试TreeSet存储自定义对象，并对对象排序的两种方式  
 \*/  
 public class TreeSetStudentTest {  
  
 public static void main(String[] args) {  
 System.out.println("下面是元素实现Comparable接口=====");  
 TreeSet<Student> ts = new TreeSet<Student>();  
 ts.add(new Student("zhangsan01", 25));  
 ts.add(new Student("zhangsan02", 21));  
 ts.add(new Student("zhangsan03", 18));  
 ts.add(new Student("zhangsan04", 26));  
 ts.add(new Student("zhangsan04", 27));  
 ts.add(new Student("zhangsan04", 27));  
 ts.add(new Student("zhangsan05", 50));  
  
 Iterator<Student> it = ts.iterator();  
 while (it.hasNext()) {  
 Student stu = (Student) it.next();  
 System.out.println("姓名：" + stu.getName() + " 年龄：" + stu.getAge());  
 }  
  
   
  
 System.out.println("下面是用自定义比较器类来实现比较的=====");  
 TreeSet<Student> tsc = new TreeSet<Student>(new MyCompare());  
 tsc.add(new Student("zhangsan01", 25));  
 tsc.add(new Student("zhangsan02", 21));  
 tsc.add(new Student("zhangsan03", 18));  
 tsc.add(new Student("zhangsan04", 26));  
 tsc.add(new Student("zhangsan04", 27));  
 tsc.add(new Student("zhangsan04", 27));  
 tsc.add(new Student("zhangsan05", 50));  
   
 Iterator<Student> itc = tsc.iterator();  
 while (itc.hasNext()) {  
 Student stu = (Student) itc.next();  
 System.out.println("姓名：" + stu.getName() + " 年龄：" + stu.getAge());  
 }  
  
 }  
  
}  
  
class Student implements Comparable<Object> {  
 private String name;  
 private int age;  
  
 public Student(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
   
 /\*\*  
 \* 年龄和名称都相等的时候  
 \*/  
 @Override   
 public boolean equals(Object obj) {   
 if (this == obj)   
 return true;   
 if (obj == null)   
 return false;   
 if (getClass() != obj.getClass())   
 return false;   
 Student other = (Student) obj;   
 if (age != other.age)   
 return false;   
 if (name == null) {   
 if (other.name != null)   
 return false;   
 } else if (!name.equals(other.name))   
 return false;   
 return true;   
 }   
   
 @Override   
 public int hashCode() {  
 final int prime = 31;   
 int result = 1;   
 result = prime \* result + age;   
 result = prime \* result + ((name == null) ? 0 : name.hashCode());   
 System.out.println("hashCode : "+ result);   
 return result;   
 }   
   
 public String getName() {  
 return name;  
 }  
   
 public void setName(String name) {  
 this.name = name;  
 }  
   
 public int getAge() {  
 return age;  
 }  
   
 public void setAge(int age) {  
 this.age = age;  
 }  
   
 public int compareTo(Object obj) {  
 if (!(obj instanceof Student))  
 throw new RuntimeException("不是学生对象");  
   
 // 先按照年龄，之后看名称  
 Student stu = (Student) obj;  
 if (this.age > stu.age)  
 return 1;  
 if (this.age == stu.age)  
 return this.name.compareTo(stu.name);  
 return -1;  
 }  
  
}  
  
// 自定义比较类  
class MyCompare implements Comparator<Object> {   
 public int compare(Object o1,Object o2) {   
 Student s1 = (Student) o1;   
 Student s2 = (Student) o2;   
 // 先按名称，之后看年龄  
 int num = s1.getName().compareTo(s2.getName());   
 if( num == 0 )   
 return new Integer(s1.getAge()).compareTo(new Integer(s2.getAge()));   
 return num;   
 }   
}

说明：  
**HashSet**：  
当向HashSet集合中存入一个元素时，HashSet会调用该对象的hashCode()方法来得到该对象的**hashCode值**，然后根据 hashCode值来决定该对象在HashSet中存储位置，

如果相等，那就不能插入，

如果不等，才会调用equals()方法，如果equals结果为true，说明已经存在，就不能再插入，如果为false，可以插入。  
简单的说，HashSet集合判断两个元素相等的标准是***两个对象通过equals方法比较相等，并且两个对象的hashCode()方法返回值相等***。  
注意，如果要把一个对象放入HashSet中，重写该对象对应类的equals方法，也应该重写其hashCode()方法。其规则是如果两个对象通过equals方法比较返回true时，其hashCode也应该相同。另外，对象中用作equals比较标准的属性，都应该用来计算 hashCode的值。

**TreeSet：**  
TreeSet是怎么实现有序的，它是按什么规则排序的？  
TreeSet的底层实现是采用**红-黑树的数据结构**，采用这种结构可以从Set中获取有序的序列，但是前提条件是：**元素必须实现Comparable接口，该接口中只用一个方法，就是compareTo()方法**。当往Set中插入一个新的元素的时候，首先会遍历Set中已经存在的元素，并调用compareTo()方法，根据返回的结果，决定插入位置。进而也就保证了元素的顺序。  
TreeSet是SortedSet接口的唯一实现类，TreeSet可以确保集合元素处于排序状态。

TreeSet支持两种排序方式，自**然排序 和定制排序**，其中自然排序为默认的排序方式。向TreeSet中加入的应该是同一个类的对象。  
TreeSet判断两个对象不相等的方式是**两个对象通过equals方法返回false，或者通过CompareTo方法比较没有返回0**  
**自然排序**  
自然排序使用要排序元素的CompareTo（Object obj）方法来比较元素之间大小关系，然后将元素按照升序排列。  
Java提供了一个Comparable接口，该接口里定义了一个compareTo(Object obj)方法，该方法返回一个整数值，实现了该接口的对象就可以比较大小。  
obj1.compareTo(obj2)方法如果返回0，则说明被比较的两个对象相等，如果返回一个正数，则表明obj1大于obj2，如果是 负数，则表明obj1小于obj2。  
如果我们将两个对象的equals方法总是返回true，则这两个对象的compareTo方法返回应该返回0  
**定制排序**  
自然排序是根据集合元素的大小，以升序排列，如果要定制排序，应该使用Comparator接口，实现 int compare(T o1,T o2)方法。  
————————————————  
版权声明：本文为CSDN博主「xiangyuenacha」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/xiangyuenacha/article/details/84255629

### 12.HashSet 的底层实现是什么?

答案：https://blog.csdn.net/Sugar\_Rainbow/article/details/68257208

<u>深入理解HashSet</u>  
<u>原创Nuub 最后发布于2017-03-30 00:01:56 阅读数 23381 收藏</u>  
<u>展开</u>  
<u>首先是有一个悲伤的故事</u>  
<u>讲道理，这是面试时遇到的第一个卡壳以至于转移面试官注意力的地方（……），还好之前有被人指点一下加确实已经仔细研究过HashMap，才不至于无法补救</u>

<u>其次我TM惊呆了</u>  
<u>本想着回来以后好好看看HashSet的底层实现，结果打开源码一看的我惊呆了</u>

<u>wocao怎么这么刺眼呢？你是set啊，你是Collection的子类啊，你叔叔才是Map啊，</u>

<u>你这样我心好痛啊</u>

~~冷静下来我仔细一想，Set不能有重复的元素，HashMap不允许有重复的键，又是一口老血，当时也没想到也没敢去这么想~~

转一下dalao的博客  
于是接着去看网上的dalao的博客，发现了这一篇私自转载dalao博文侵删

1、HashSet概述和实现

2、源码分析

3、LinkedHashSet分析

4、TreeSet分析

5、总结

**1、HashSet概述和实现**

HashSet实现Set接口，由哈希表（实际上是一个HashMap实例）支持。它不保证set 的迭代顺序；特别是它不保证该顺序恒久不变，此类允许使用null元素。  
在HashSet中，元素都存到HashMap键值对的Key上面，而Value是有一个统一的值

private static final Object PRESENT = new Object();，

**1.1、HashSet插入**  
当有新值加入时，底层的HashMap会判断Key值是否存在（HashMap细节请移步深入理解HashMap），如果不存在，则插入新值，同时这个插入的细节会依照HashMap插入细节；如果存在就不插入

**1.2、HashSet删除**  
同HashMap删除原理

**2、源码分析**  
盗（xue）用（xi）一下dalao 的分析代码，侵权请告之，立马删除

public class HashSet<E>   
 extends AbstractSet<E>   
 implements Set<E>, Cloneable, java.io.Serializable   
{   
 static final long serialVersionUID = -5024744406713321676L;   
  
 // 底层使用HashMap来保存HashSet中所有元素。   
 private transient HashMap<E,Object> map;   
   
 // 定义一个虚拟的Object对象作为HashMap的value，将此对象定义为static final。   
 private static final Object PRESENT = new Object();   
   
 /\*\*   
 \* 默认的无参构造器，构造一个空的HashSet。   
 \*   
 \* 实际底层会初始化一个空的HashMap，并使用默认初始容量为16和加载因子0.75。   
 \*/   
 public HashSet() {   
 map = new HashMap<E,Object>();   
 }   
   
 /\*\*   
 \* 构造一个包含指定collection中的元素的新set。   
 \*   
 \* 实际底层使用默认的加载因子0.75和足以包含指定   
 \* collection中所有元素的初始容量来创建一个HashMap。   
 \* @param c 其中的元素将存放在此set中的collection。   
 \*/   
 public HashSet(Collection<? extends E> c) {   
 map = new HashMap<E,Object>(Math.max((int) (c.size()/.75f) + 1, 16));   
 addAll(c);   
 }   
   
 /\*\*   
 \* 以指定的initialCapacity和loadFactor构造一个空的HashSet。   
 \*   
 \* 实际底层以相应的参数构造一个空的HashMap。   
 \* @param initialCapacity 初始容量。   
 \* @param loadFactor 加载因子。   
 \*/   
 public HashSet(int initialCapacity, float loadFactor) {   
 map = new HashMap<E,Object>(initialCapacity, loadFactor);   
 }   
   
 /\*\*   
 \* 以指定的initialCapacity构造一个空的HashSet。   
 \*   
 \* 实际底层以相应的参数及加载因子loadFactor为0.75构造一个空的HashMap。   
 \* @param initialCapacity 初始容量。   
 \*/   
 public HashSet(int initialCapacity) {   
 map = new HashMap<E,Object>(initialCapacity);   
 }   
   
 /\*\*   
 \* 以指定的initialCapacity和loadFactor构造一个新的空链接哈希集合。   
 \* 此构造函数为包访问权限，不对外公开，实际只是是对LinkedHashSet的支持。   
 \*   
 \* 实际底层会以指定的参数构造一个空LinkedHashMap实例来实现。   
 \* @param initialCapacity 初始容量。   
 \* @param loadFactor 加载因子。   
 \* @param dummy 标记。   
 \*/   
 HashSet(int initialCapacity, float loadFactor, boolean dummy) {   
 map = new LinkedHashMap<E,Object>(initialCapacity, loadFactor);   
 }   
   
 /\*\*   
 \* 返回对此set中元素进行迭代的迭代器。返回元素的顺序并不是特定的。   
 \*   
 \* 底层实际调用底层HashMap的keySet来返回所有的key。   
 \* 可见HashSet中的元素，只是存放在了底层HashMap的key上，   
 \* value使用一个static final的Object对象标识。   
 \* @return 对此set中元素进行迭代的Iterator。   
 \*/   
 public Iterator<E> iterator() {   
 return map.keySet().iterator();   
 }   
   
 /\*\*   
 \* 返回此set中的元素的数量（set的容量）。   
 \*   
 \* 底层实际调用HashMap的size()方法返回Entry的数量，就得到该Set中元素的个数。   
 \* @return 此set中的元素的数量（set的容量）。   
 \*/   
 public int size() {   
 return map.size();   
 }   
   
 /\*\*   
 \* 如果此set不包含任何元素，则返回true。   
 \*   
 \* 底层实际调用HashMap的isEmpty()判断该HashSet是否为空。   
 \* @return 如果此set不包含任何元素，则返回true。   
 \*/   
 public boolean isEmpty() {   
 return map.isEmpty();   
 }   
   
 /\*\*   
 \* 如果此set包含指定元素，则返回true。   
 \* 更确切地讲，当且仅当此set包含一个满足(o==null ? e==null : o.equals(e))   
 \* 的e元素时，返回true。   
 \*   
 \* 底层实际调用HashMap的containsKey判断是否包含指定key。   
 \* @param o 在此set中的存在已得到测试的元素。   
 \* @return 如果此set包含指定元素，则返回true。   
 \*/   
 public boolean contains(Object o) {   
 return map.containsKey(o);   
 }   
   
 /\*\*   
 \* 如果此set中尚未包含指定元素，则添加指定元素。   
 \* 更确切地讲，如果此 set 没有包含满足(e==null ? e2==null : e.equals(e2))   
 \* 的元素e2，则向此set 添加指定的元素e。   
 \* 如果此set已包含该元素，则该调用不更改set并返回false。   
 \*   
 \* 底层实际将将该元素作为key放入HashMap。   
 \* 由于HashMap的put()方法添加key-value对时，当新放入HashMap的Entry中key   
 \* 与集合中原有Entry的key相同（hashCode()返回值相等，通过equals比较也返回true），   
 \* 新添加的Entry的value会将覆盖原来Entry的value，但key不会有任何改变，   
 \* 因此如果向HashSet中添加一个已经存在的元素时，新添加的集合元素将不会被放入HashMap中，   
 \* 原来的元素也不会有任何改变，这也就满足了Set中元素不重复的特性。   
 \* @param e 将添加到此set中的元素。   
 \* @return 如果此set尚未包含指定元素，则返回true。   
 \*/   
 public boolean add(E e) {   
 return map.put(e, PRESENT)==null;   
 }   
   
 /\*\*   
 \* 如果指定元素存在于此set中，则将其移除。   
 \* 更确切地讲，如果此set包含一个满足(o==null ? e==null : o.equals(e))的元素e，   
 \* 则将其移除。如果此set已包含该元素，则返回true   
 \* （或者：如果此set因调用而发生更改，则返回true）。（一旦调用返回，则此set不再包含该元素）。   
 \*   
 \* 底层实际调用HashMap的remove方法删除指定Entry。   
 \* @param o 如果存在于此set中则需要将其移除的对象。   
 \* @return 如果set包含指定元素，则返回true。   
 \*/   
 public boolean remove(Object o) {   
 return map.remove(o)==PRESENT;   
 }   
   
 /\*\*   
 \* 从此set中移除所有元素。此调用返回后，该set将为空。   
 \*   
 \* 底层实际调用HashMap的clear方法清空Entry中所有元素。   
 \*/   
 public void clear() {   
 map.clear();   
 }   
   
 /\*\*   
 \* 返回此HashSet实例的浅表副本：并没有复制这些元素本身。   
 \*   
 \* 底层实际调用HashMap的clone()方法，获取HashMap的浅表副本，并设置到HashSet中。   
 \*/   
 public Object clone() {   
 try {   
 HashSet<E> newSet = (HashSet<E>) super.clone();   
 newSet.map = (HashMap<E, Object>) map.clone();   
 return newSet;   
 } catch (CloneNotSupportedException e) {   
 throw new InternalError();   
 }   
 }   
  
}

注意  
说白了，HashSet就是限制了功能的HashMap，所以了解HashMap的实现原理，这个HashSet自然就通了。  
对于HashSet中保存的对象，主要是要正确重写equals方法和hashCode方法，以保证放入Set对象的唯一性  
虽说Set是对于重复的元素不放入，倒不如直接说是底层的Map直接把原值替代了（这个Set的put方法的返回值真有意思）  
HashSet没有提供get()方法，原因是同HashMap一样，Set内部是无序的，只能通过迭代的方式获得

说起来你可能不信  
本来是打算分开写集合框架的底层分析的，直到我发现，LinkedHashSet是继承自HashSet，底层实现是LinkedHashMap。并且其初始化时直接super(......)，瞬间我就觉得，Set写在一起得了

**3、LinkedHashSet分析**

**3.1概述**

同HashSet相比并没有实现新的功能（新的方法），只不过把HashSet中预留的构造方法启用了，因而可以实现有序插入，而这个具体的实现要去看LinkedHashMap了，我们使用时是不需要再去设置参数的，直接拿来用即可。

/\*\*  
 \* The iteration ordering method for this linked hash map: <tt>true</tt>  
 \* for access-order, <tt>false</tt> for insertion-order.  
 \*  
 \* @serial  
 \*/  
final boolean accessOrder;

查看了LinkedHashMap的构造方法后，发现其因为继承自HashMap，所以其底层实现也是HashMap!!!（呵呵，我已经发现了……怪不得还是得主要研究HashMap啊），然后发现了LinkedHashMap调用父类构造方法初始化时，还顺便设置了变量accessOrder = false，看上面得源码可以知道，这是给了迭代器一个参数，false代表迭代时使用插入得顺序（追根溯源了，真爽）

**3.2可分割迭代器**

偶然发现,查看源码时，我发现了一个奇怪的重写的方法：public Spliterator spliterator()，查了查资料发现叫做可分割迭代器，这个接口是为了并行遍历数据源中的元素而设计的迭代器，为了更好的发挥多核CPU的能力。  
其实这样我想起了要去关注一下集合框架中的并发安全了。

**4、TreeSet分析**

**4.1TreeSet概述**

根据Set的这个尿性，我先猜测一波，TreeSet的底层实现是TreeMap（而且我在猜TreeMap的底层实现借助了HashMap）。一看源码，哎呦我去，还真是（呵呵，到底谁才是你爹…..心疼一波Collection,Map又不继承Collection接口）

public TreeSet() {  
 this(new TreeMap<E,Object>());  
}

**4.2 TreeSet特点与实现机制**  
TreeSet中存放的元素是有序的（不是插入时的顺序，是有按关键字大小排序的），且元素不能重复。  
而如何实现有序存储，就需要有一个比较器，其实说起来，TreeSet更受关注的是不重复且有序，这个有序就需要有一个compare的过程，因此会需要参数实现Comparable接口。

/\*\*  
 \* Constructs a new, empty tree set, sorted according to the specified  
 \* comparator. All elements inserted into the set must be <i>mutually  
 \* comparable</i> by the specified comparator: {@code comparator.compare(e1,  
 \* e2)} must not throw a {@code ClassCastException} for any elements  
 \* {@code e1} and {@code e2} in the set. If the user attempts to add  
 \* an element to the set that violates this constraint, the  
 \* {@code add} call will throw a {@code ClassCastException}.  
 \*  
 \* @param comparator the comparator that will be used to order this set.  
 \* If {@code null}, the {@linkplain Comparable natural  
 \* ordering} of the elements will be used.  
 \*/  
public TreeSet(Comparator<? super E> comparator) {  
 this(new TreeMap<>(comparator));  
}

所以说使用Set需要注意的还是根据自己的需求选取正确的存储结构即可，而因为并没有get()方法给你使用，所以还是要用迭代器来获取想要的元素，然后本次Set深入分析到此结束，我要去再开一坑研究TreeMap了（滑稽）

**5、小总结**  
经历这么一次滑稽的经历，看来真的有必要把几个常用的集合框架的底层实现都看一遍，以免再次搞出这样的尴尬（手动滑稽）  
其实深入到这个程度我觉得常用的集合除了List的家族还有Queue，其实都可以规结为深入理解HashMap，来，就是这个节奏。走起。  
————————————————  
版权声明：本文为CSDN博主「Nuub」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/Sugar\_Rainbow/article/details/68257208

### 13.LinkedHashMap 的实现原理?

答案： [LinkedHashMap的实现原理](https://www.cnblogs.com/ganchuanpu/p/8908093.html)(https://www.cnblogs.com/ganchuanpu/p/8908093.html)

1. LinkedHashMap概述：

LinkedHashMap是Map接口的哈希表和链接列表实现，具有可预知的迭代顺序。此实现提供所有可选的映射操作，并允许使用null值和null键。此类不保证映射的顺序，特别是它不保证该顺序恒久不变。  
 LinkedHashMap实现与HashMap的不同之处在于，后者维护着一个运行于所有条目的双重链接列表。此链接列表定义了迭代顺序，该迭代顺序可以是插入顺序或者是访问顺序。  
 注意，此实现不是同步的。如果多个线程同时访问链接的哈希映射，而其中至少一个线程从结构上修改了该映射，则它必须保持外部同步。

1. LinkedHashMap的实现：

对于LinkedHashMap而言，它继承与HashMap、底层使用哈希表与双向链表来保存所有元素。其基本操作与父类HashMap相似，它通过重写父类相关的方法，来实现自己的链接列表特性。下面我们来分析LinkedHashMap的源代码：

1) Entry元素：

LinkedHashMap采用的hash算法和HashMap相同，但是它重新定义了数组中保存的元素Entry，该Entry除了保存当前对象的引用外，还保存了其上一个元素before和下一个元素after的引用，从而在哈希表的基础上又构成了双向链接列表。看源代码：

1 /\*\*   
 2 \* 双向链表的表头元素。   
 3 \*/   
 4 private transient Entry<K,V> header;   
 5   
 6 /\*\*   
 7 \* LinkedHashMap的Entry元素。   
 8 \* 继承HashMap的Entry元素，又保存了其上一个元素before和下一个元素after的引用。   
 9 \*/   
10 private static class Entry<K,V> extends HashMap.Entry<K,V> {   
11 Entry<K,V> before, after;   
12 ……   
13 }

2) 初始化：

通过源代码可以看出，在LinkedHashMap的构造方法中，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。如：

public LinkedHashMap(int initialCapacity, float loadFactor) {   
 super(initialCapacity, loadFactor);   
 accessOrder = false;   
}

HashMap中的相关构造方法：

1 public HashMap(int initialCapacity, float loadFactor) {   
 2 if (initialCapacity < 0)   
 3 throw new IllegalArgumentException("Illegal initial capacity: " +   
 4 initialCapacity);   
 5 if (initialCapacity > MAXIMUM\_CAPACITY)   
 6 initialCapacity = MAXIMUM\_CAPACITY;   
 7 if (loadFactor <= 0 || Float.isNaN(loadFactor))   
 8 throw new IllegalArgumentException("Illegal load factor: " +   
 9 loadFactor);   
10   
11 // Find a power of 2 >= initialCapacity   
12 int capacity = 1;   
13 while (capacity < initialCapacity)   
14 capacity <<= 1;   
15   
16 this.loadFactor = loadFactor;   
17 threshold = (int)(capacity \* loadFactor);   
18 table = new Entry[capacity];   
19 init();   
20 }

我们已经知道LinkedHashMap的Entry元素继承HashMap的Entry，提供了双向链表的功能。在上述HashMap的构造器中，最后会调用init()方法，进行相关的初始化，这个方法在HashMap的实现中并无意义，只是提供给子类实现相关的初始化调用。  
 LinkedHashMap重写了init()方法，在调用父类的构造方法完成构造后，进一步实现了对其元素Entry的初始化操作。

void init() {   
 header = new Entry<K,V>(-1, null, null, null);   
 header.before = header.after = header;   
}

3) 存储：

LinkedHashMap并未重写父类HashMap的put方法，而是重写了父类HashMap的put方法调用的子方法void addEntry(int hash, K key, V value, int bucketIndex) 和void createEntry(int hash, K key, V value, int bucketIndex)，提供了自己特有的双向链接列表的实现。

1 void addEntry(int hash, K key, V value, int bucketIndex) {   
 2 // 调用create方法，将新元素以双向链表的的形式加入到映射中。   
 3 createEntry(hash, key, value, bucketIndex);   
 4   
 5 // 删除最近最少使用元素的策略定义   
 6 Entry<K,V> eldest = header.after;   
 7 if (removeEldestEntry(eldest)) {   
 8 removeEntryForKey(eldest.key);   
 9 } else {   
10 if (size >= threshold)   
11 resize(2 \* table.length);   
12 }   
13 }   
14   
15 void createEntry(int hash, K key, V value, int bucketIndex) {   
16 HashMap.Entry<K,V> old = table[bucketIndex];   
17 Entry<K,V> e = new Entry<K,V>(hash, key, value, old);   
18 table[bucketIndex] = e;   
19 // 调用元素的addBrefore方法，将元素加入到哈希、双向链接列表。   
20 e.addBefore(header);   
21 size++;   
22 }   
23   
24 private void addBefore(Entry<K,V> existingEntry) {   
25 after = existingEntry;   
26 before = existingEntry.before;   
27 before.after = this;   
28 after.before = this;   
29 }

4) 读取：

LinkedHashMap重写了父类HashMap的get方法，实际在调用父类getEntry()方法取得查找的元素后，再判断当排序模式accessOrder为true时，记录访问顺序，将最新访问的元素添加到双向链表的表头，并从原来的位置删除。由于的链表的增加、删除操作是常量级的，故并不会带来性能的损失。

1 public V get(Object key) {   
 2 // 调用父类HashMap的getEntry()方法，取得要查找的元素。   
 3 Entry<K,V> e = (Entry<K,V>)getEntry(key);   
 4 if (e == null)   
 5 return null;   
 6 // 记录访问顺序。   
 7 e.recordAccess(this);   
 8 return e.value;   
 9 }   
10   
11 void recordAccess(HashMap<K,V> m) {   
12 LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;   
13 // 如果定义了LinkedHashMap的迭代顺序为访问顺序，   
14 // 则删除以前位置上的元素，并将最新访问的元素添加到链表表头。   
15 if (lm.accessOrder) {   
16 lm.modCount++;   
17 remove();   
18 addBefore(lm.header);   
19 }   
20 }

5) 排序模式：

LinkedHashMap定义了排序模式accessOrder，该属性为boolean型变量，对于访问顺序，为true；对于插入顺序，则为false。

private final boolean accessOrder;

一般情况下，不必指定排序模式，其迭代顺序即为默认为插入顺序。看LinkedHashMap的构造方法，如：

public LinkedHashMap(int initialCapacity, float loadFactor) {   
 super(initialCapacity, loadFactor);   
 accessOrder = false;   
}

这些构造方法都会默认指定排序模式为插入顺序。如果你想构造一个LinkedHashMap，并打算按从近期访问最少到近期访问最多的顺序（即访问顺序）来保存元素，那么请使用下面的构造方法构造LinkedHashMap：

public LinkedHashMap(int initialCapacity,   
 float loadFactor,   
 boolean accessOrder) {   
 super(initialCapacity, loadFactor);   
 this.accessOrder = accessOrder;   
}

该哈希映射的迭代顺序就是最后访问其条目的顺序，这种映射很适合构建LRU缓存。LinkedHashMap提供了removeEldestEntry(Map.Entry<K,V> eldest)方法，在将新条目插入到映射后，put和 putAll将调用此方法。该方法可以提供在每次添加新条目时移除最旧条目的实现程序，默认返回false，这样，此映射的行为将类似于正常映射，即永远不能移除最旧的元素。

protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {   
 return false;   
}

此方法通常不以任何方式修改映射，相反允许映射在其返回值的指引下进行自我修改。如果用此映射构建LRU缓存，则非常方便，它允许映射通过删除旧条目来减少内存损耗。  
 例如：重写此方法，维持此映射只保存100个条目的稳定状态，在每次添加新条目时删除最旧的条目。

private static final int MAX\_ENTRIES = 100;   
protected boolean removeEldestEntry(Map.Entry eldest) {   
 return size() > MAX\_ENTRIES;   
}

补充下

LRU是一种缓存替换算法，根据字面意思，就是将最近最少使用的页面或者元素进行替换，将最近最多使用的页面或者元素保持在缓存里。有关缓存的知识后面再仔细研究下。由于缓存的容量大小有限，这才有了LRU之类的缓存算法。

---引用自[LRU 缓存机制及 3 种简单实现](https://www.cnblogs.com/dogeLife/p/11370811.html)https://www.cnblogs.com/dogeLife/p/11370811.html

### 14.为什么集合类没有实现 Cloneable 和 Serializable 接口？

答案连接：[CodeSariel](https://me.csdn.net/wangzibai) 请解释为什么集合类没有实现Cloneable和Serializable接口？https://blog.csdn.net/wangzibai/article/details/102632174

1、接口概念

2、分析

3、框架中使用

**1、接口概念**

Cloneable接口是一个标识接口，不包含任何方法的！这个标识仅仅是针对Object类中clone()方法的，如果clone类没有实现Cloneable接口，并调用了Object的 clone()方法（也就是调用了super.Clone()方法），那么Object的clone()方法就会抛出 CloneNotSupportedException异常

Serializable接口，实现这个接口的类的对象，可以写入到输出流，称为**可序列化对象**  
特点：也是一个标记接口，因为它没有抽象方法

**2、分析**

这个问题说的不清楚，集合类框架中的接口没有实现Cloneable和Serializable接口，但是具体的实现类是实现了这些接口的，比如Arraylist。

接口，比如collection list iterable。

克隆（cloning）或者序列化（serialization）的予以和含义是根具体的实现相关的。因此，应该由集合类的具体实现来决定如何被克隆或者是序列化。

实现Serializable序列化的作用：将对象状态保存在存储媒体中以便可以在以后重写创建出完全相同的副本；将对象从一个应用程序域发送到另一个应用程序域。

实现Serializable接口的作用就是可以把对象存到字节流，然后可以恢复。所以你想如果你的对象没有序列化，怎么才能进行网络传输呢？要网络传输就得转为字节流，所以在分布式应用中，你就得实现序列化。如果不需要分布式应用就没必要实现序列化。

**3、框架中使用**

java如果用json传输数据javabean时，需要实现Serializable。

**个人观点：我写的应用中不需要实现Serializable因为我使用了json数据格式。**

**结论：**java如果使用json+restfulAPI进行数据交互，那么javabean不需要implements Serializable接口。因为对象数据并没有真正的进行网络传输或者持久化。而是转换为了json字符串。进行网络传输的是对象的具体信息而不是对象本身。所以在这种场景下不需要implements Serializable。

如果使用dubbo这种RPC（Remote Procedure Call）框架，那么使用面向接口编程的时候，接口中的参数如果是类，那么该类就必须implements Serializable才可以让对象在远程调用的时候，在网络上传输。

javabean为什么不implements Serializable也可以保存到数据库中

不是说数据持久化就必须implements Serializable接口么

**原因：**

Java的基本数据类型（primitive：short int long float double byte char boolean）在数据库中都有对应的字段。一些Java的集合具体实现类也都实现了Serializable接口。那么JavaBean此时不实现该接口也可以存储到数据库中

序列化绝对不是实现Serializable接口一种方式。还存在其他序列化方式可能效率更高。但是稳定性肯定是Serializable最好。因为该接口从Java1.1开始就存在并且一直保持使用

https://github.com/eishay/jvm-serializers/wiki

序列化存储到本地，若从存储介质中读取数据字节流并反序列化的时候并不会调用类的构造函数

具体序列化参考：

https://juejin.im/post/5ce3cdc8e51d45777b1a3cdf#heading-11  
————————————————  
版权声明：本文为CSDN博主「CodeSariel」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/wangzibai/article/details/102632174

### 15.什么是迭代器 (Iterator)？

转自：[牛客网](https://www.nowcoder.com/questionTerminal/8863f297b1fc4bbca6de95528b6051e1)https://www.nowcoder.com/questionTerminal/8863f297b1fc4bbca6de95528b6051e1

一、什么是迭代器（Iterator）？

二、使用Iterator的简单例子

三、关于Iterator的简单的解释

四、关于Iterator的详细的解释

五、Java中Iterator接口

**一、什么是迭代器（Iterator）？**

答：Iterator接口提供了很多对集合元素进行迭代的方法。每一个集合类都包括了可以返回迭代器实例的迭代方法。迭代器可以在迭代过程中删除底层集合的元素，但是不可以直接调用集合的remove(Object obj)删除，可以通过迭代器的remove()方法删除

**二、使用Iterator的简单例子**

以ArrayList和HashMap为例，使用Iterator的简单例子代码如下：

public class TestIterator {  
  
public static void main(String[] args) {  
  
List list = new ArrayList();  
Map map = new HashMap();  
for(int i= 0 ;i< 10 ;i++){  
 list.add(new String( "list" +i) );  
 map.put(i,new String( "map" +i));  
}   
Iterator iterList= list.iterator();//List接口实现了Iterable接口  
while (iterList.hasNext()){  
 String strList=(String)iterList.next();  
 System.out.println(strList.toString());  
}  
Iterator iterMap = map.entrySet().iterator();  
while (iterMap.hasNext()){  
 Map.Entry strMap=(Map.Entry)iterMap.next();  
 System.out.println(strMap.getValue());  
}}

通过迭代器，我们可以遍历，得到集合中的每一个值

**三、关于Iterator的简单的解释**

Iterator提供了遍历操作集合元素的统一接口，Collection接口实现了Iterable接口，每个集合都通过实现Iterable接口中的iterator()方法返回Iterator接口的实例，然后对集合的元素进行迭代操作。

1. Iterable接口  
 Iterator iterator();  
2. Iterator接口  
 boolean hasNext();  
 E next();  
 void remove();

**四、关于Iterator的详细的解释**

转自：[java提高篇（三十）—— Iterator](http://blog.csdn.net/chenssy/article/details/37521461)(http://blog.csdn.net/chenssy/article/details/37521461)

迭代对于我们搞Java的来说绝对不陌生。我们常常使用JDK提供的迭代接口，进行Java集合的迭代。示例如下：

Iterator iterator = list.iterator();  
 while(iterator.hasNext()){  
 String string = iterator.next();  
 //do something  
 }

迭代其实我们可以简单地理解为遍历，是一个标准化遍历各类容器里面的所有对象的方法类，它是一个很典型的设计模式。Iterator模式是用于遍历集合类的标准访问方法。它可以把访问逻辑从不同类型的集合类中抽象出来，从而避免向客户端暴露集合的内部结构。 在没有迭代器时我们都是这么进行处理的。代码如下：

对于数组我们是使用下标来进行处理的:  
 int[] arrays = new int[10];  
 for(int i = 0 ; i < arrays.length ; i++){  
 int a = arrays[i];  
 //do something  
 }  
 对于ArrayList是这么处理的:  
  
 List<String> list = new ArrayList<String>();  
 for(int i = 0 ; i < list.size() ; i++){  
 String string = list.get(i);  
 //do something  
 }

对于这两种方式，我们总是都事先知道集合的内部结构，访问代码和集合本身是紧密耦合的，无法将访问逻辑从集合类和客户端代码中分离出来。同时每一种集合对应一种遍历方法，客户端代码无法复用。 在实际应用中如何将上面两个集合的访问方法进行整合是相当麻烦的。

所以为了解决以上问题，Iterator模式腾空出世，它总是用同一种逻辑来遍历集合。使得客户端自身不需要来维护集合的内部结构，所有的内部状态都由Iterator来维护。客户端从不直接和集合类打交道，它总是控制Iterator，向它发送"向前"，"向后"，"取当前元素"的命令，就可以间接遍历整个集合。

**五、Java中Iterator接口**

上面只是对Iterator模式进行简单的说明，下面我们看看Java中Iterator接口，看他是如何来进行实现的。

5.1、java.util.Iterator  
 在Java中Iterator为一个接口，它只提供了迭代了基本规则，在JDK中他是这样定义的：对 collection 进行迭代的迭代器。迭代器取代了 Java Collections Framework 中的 Enumeration。迭代器与枚举有两点不同：

1、迭代器允许调用者利用定义良好的语义，在迭代期间从迭代器所指向的 collection 移除元素。

2、方法名称得到了改进。

其接口定义如下：  
 public interface Iterator {  
 boolean hasNext();  
 Object next();  
 void remove();  
 }

其中： Object next()：返回迭代器刚越过的元素的引用，返回值是Object，需要强制转换成自己需要的类型

boolean hasNext()：判断容器内是否还有可供访问的元素

void remove()：删除迭代器刚越过的元素；

对于我们而言，我们只一般只需使用next()、hasNext()两个方法即可完成迭代。如下：

for(Iterator it = c.iterator(); it.hasNext(); ) {  
　　 Object o = it.next();  
　　 //do something  
 }

5.2、ArrayList的Iterator的实现

前面阐述了Iterator有一个很大的优点,就是我们不必知道集合的内部结构,集合的内部结构、状态由Iterator来维持，通过统一的方法hasNext()、next()来判断、获取下一个元素，至于具体的内部实现我们就不用关心了。但是作为一个合格的程序员我们非常有必要来弄清楚Iterator的实现。下面就以ArrayList的源码进行分析。

其实如果我们理解了ArrayList、Hashset、TreeSet的数据结构，内部实现，对于他们是如何实现Iterator也会胸有成竹的。因为ArrayList的内部实现采用数组，所以我们只需要记录相应位置的索引即可，其方法的实现比较简单。

5.2.1、ArrayList的Iterator实现  
 在ArrayList内部首先是定义一个内部类Itr，该内部类实现Iterator接口，代码如下：

private class Itr implements Iterator<E> {  
 //do something  
}

而ArrayList的iterator()方法实现,代码如下：

public Iterator<E> iterator() {  
 return new Itr();  
 }

所以通过使用ArrayList.iterator()方法返回的是Itr()内部类，现在我们需要关心的就是Itr()内部类的实现：

在Itr内部定义了三个int型的变量：cursor、lastRet、expectedModCount。

其中cursor表示下一个元素的索引位置，lastRet表示上一个元素的索引位置。代码如下：

int cursor;   
int lastRet = -1;   
int expectedModCount = modCount;

从cursor、lastRet定义可以看出，lastRet一直比cursor少一，所以hasNext()实现方法异常简单，只需要判断cursor和lastRet是否相等即可。

public boolean hasNext() {  
 return cursor != size;  
}

对于next()实现其实也是比较简单的，只要返回cursor索引位置处的元素即可，然后修改cursor、lastRet即可，next()方法源码实现如下：

public E next() {  
 checkForComodification();  
 int i = cursor; //记录索引位置  
 if (i >= size) //如果获取元素大于集合元素个数，则抛出异常  
 throw new NoSuchElementException();  
 Object[] elementData = ArrayList.this.elementData;  
 if (i >= elementData.length)  
 throw new ConcurrentModificationException();  
 cursor = i + 1; //cursor + 1  
 return (E) elementData[lastRet = i]; //lastRet + 1 且返回cursor处元素  
}

checkForComodification()主要用来判断集合的修改次数是否合法，即用来判断遍历过程中集合是否被修改过。在上面的第二讲中已经阐述了。

modCount用于记录ArrayList集合的修改次数，初始化为0，，每当集合被修改一次（结构上面的修改，内部update不算），如add、remove等方法，modCount + 1，所以如果modCount不变，则表示集合内容没有被修改。该机制主要是用于实现ArrayList集合的快速失败机制，在Java的集合中，较大一部分集合是存在快速失败机制的，这里就不多说，前面已经讲到。

所以要保证在遍历过程中不出错误，我们就应该保证在遍历过程中不会对集合产生结构上的修改（当然remove方法除外），出现了异常错误，我们就应该认真检查程序是否出错而不是catch后不做处理。

final void checkForComodification() {  
 if (modCount != expectedModCount)  
 throw new ConcurrentModificationException();  
 }

对于remove()方法的是实现，它是调用ArrayList本身的remove()方法删除lastRet位置元素，然后修改modCount即可。

public void remove() {  
 if (lastRet < 0)  
 throw new IllegalStateException();  
 checkForComodification();   
 try {  
 ArrayList.this.remove(lastRet);  
 cursor = lastRet;  
 lastRet = -1;  
 expectedModCount = modCount;  
 } catch (IndexOutOfBoundsException ex) {  
 throw new ConcurrentModificationException();  
 }  
 }

这里就对ArrayList的Iterator实现讲解到这里，对于Hashset、TreeSet等集合的Iterator实现，各位如果感兴趣可以继续研究，个人认为在研究这些集合的源码之前，有必要对该集合的数据结构有清晰的认识，这样会达到事半功倍的效果！！！！

——————————————————————————————————————————————  
-- 原文出自：http://cmsblogs.com/?p=1180。尊重作者的成果，转载请注明出处！

-- 个人站点：http://cmsblogs.com

————————————————  
版权声明：本文为CSDN博主「chenssy」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/chenssy/article/details/37521461

### 16.Iterator 和 ListIterator 的区别是什么？

答案：Iterator和ListIterator之间有什么区别？https://blog.csdn.net/xiangyuenacha/article/details/84253630

一、源码展示

二、方法总结

三、分析相同点和不同点

**一、源码展示**

看源码Iterator ：public interface Iterator

/\*\*  
 \* An iterator over a collection. {@code Iterator} takes the place of  
 \* {@link Enumeration} in the Java Collections Framework. Iterators  
 \* differ from enumerations in two ways:  
 \*  
 \* <ul>  
 \* <li> Iterators allow the caller to remove elements from the  
 \* underlying collection during the iteration with well-defined  
 \* semantics.  
 \* <li> Method names have been improved.  
 \* </ul>  
 \*  
 \* <p>This interface is a member of the  
 \* <a href="{@docRoot}/../technotes/guides/collections/index.html">  
 \* Java Collections Framework</a>.  
 \*  
 \* @param <E> the type of elements returned by this iterator  
 \*  
 \* @author Josh Bloch  
 \* @see Collection  
 \* @see ListIterator  
 \* @see Iterable  
 \* @since 1.2  
 \*/  
public interface Iterator<E> {  
 /\*\*  
 \* Returns {@code true} if the iteration has more elements.  
 \* (In other words, returns {@code true} if {@link #next} would  
 \* return an element rather than throwing an exception.)  
 \*  
 \* @return {@code true} if the iteration has more elements  
 \*/  
 boolean hasNext();  
  
 /\*\*  
 \* Returns the next element in the iteration.  
 \*  
 \* @return the next element in the iteration  
 \* @throws NoSuchElementException if the iteration has no more elements  
 \*/  
 E next();  
  
 /\*\*  
 \* Removes from the underlying collection the last element returned  
 \* by this iterator (optional operation). This method can be called  
 \* only once per call to {@link #next}. The behavior of an iterator  
 \* is unspecified if the underlying collection is modified while the  
 \* iteration is in progress in any way other than by calling this  
 \* method.  
 \*  
 \* @implSpec  
 \* The default implementation throws an instance of  
 \* {@link UnsupportedOperationException} and performs no other action.  
 \*  
 \* @throws UnsupportedOperationException if the {@code remove}  
 \* operation is not supported by this iterator  
 \*  
 \* @throws IllegalStateException if the {@code next} method has not  
 \* yet been called, or the {@code remove} method has already  
 \* been called after the last call to the {@code next}  
 \* method  
 \*/  
 default void remove() {  
 throw new UnsupportedOperationException("remove");  
 }  
  
 /\*\*  
 \* Performs the given action for each remaining element until all elements  
 \* have been processed or the action throws an exception. Actions are  
 \* performed in the order of iteration, if that order is specified.  
 \* Exceptions thrown by the action are relayed to the caller.  
 \*  
 \* @implSpec  
 \* <p>The default implementation behaves as if:  
 \* <pre>{@code  
 \* while (hasNext())  
 \* action.accept(next());  
 \* }</pre>  
 \*  
 \* @param action The action to be performed for each element  
 \* @throws NullPointerException if the specified action is null  
 \* @since 1.8  
 \*/  
 default void forEachRemaining(Consumer<? super E> action) {  
 Objects.requireNonNull(action);  
 while (hasNext())  
 action.accept(next());  
 }  
}

看源码ListIterator继承Iterator：public interface ListIterator extends Iterator

摘自jdk源码1.8  
/\*\*  
 \* An iterator for lists that allows the programmer  
 \* to traverse the list in either direction, modify  
 \* the list during iteration, and obtain the iterator's  
 \* current position in the list. A {@code ListIterator}  
 \* has no current element; its <I>cursor position</I> always  
 \* lies between the element that would be returned by a call  
 \* to {@code previous()} and the element that would be  
 \* returned by a call to {@code next()}.  
 \* An iterator for a list of length {@code n} has {@code n+1} possible  
 \* cursor positions, as illustrated by the carets ({@code ^}) below:  
 \* <PRE>  
 \* Element(0) Element(1) Element(2) ... Element(n-1)  
 \* cursor positions: ^ ^ ^ ^ ^  
 \* </PRE>  
 \* Note that the {@link #remove} and {@link #set(Object)} methods are  
 \* <i>not</i> defined in terms of the cursor position; they are defined to  
 \* operate on the last element returned by a call to {@link #next} or  
 \* {@link #previous()}.  
 \*  
 \* <p>This interface is a member of the  
 \* <a href="{@docRoot}/../technotes/guides/collections/index.html">  
 \* Java Collections Framework</a>.  
 \*  
 \* @author Josh Bloch  
 \* @see Collection  
 \* @see List  
 \* @see Iterator  
 \* @see Enumeration  
 \* @see List#listIterator()  
 \* @since 1.2  
 \*/  
public interface ListIterator<E> extends Iterator<E> {  
 // Query Operations  
  
 /\*\*  
 \* Returns {@code true} if this list iterator has more elements when  
 \* traversing the list in the forward direction. (In other words,  
 \* returns {@code true} if {@link #next} would return an element rather  
 \* than throwing an exception.)  
 \*  
 \* @return {@code true} if the list iterator has more elements when  
 \* traversing the list in the forward direction  
 \*/  
 boolean hasNext();  
  
 /\*\*  
 \* Returns the next element in the list and advances the cursor position.  
 \* This method may be called repeatedly to iterate through the list,  
 \* or intermixed with calls to {@link #previous} to go back and forth.  
 \* (Note that alternating calls to {@code next} and {@code previous}  
 \* will return the same element repeatedly.)  
 \*  
 \* @return the next element in the list  
 \* @throws NoSuchElementException if the iteration has no next element  
 \*/  
 E next();  
  
 /\*\*  
 \* Returns {@code true} if this list iterator has more elements when  
 \* traversing the list in the reverse direction. (In other words,  
 \* returns {@code true} if {@link #previous} would return an element  
 \* rather than throwing an exception.)  
 \*  
 \* @return {@code true} if the list iterator has more elements when  
 \* traversing the list in the reverse direction  
 \*/  
 boolean hasPrevious();  
  
 /\*\*  
 \* Returns the previous element in the list and moves the cursor  
 \* position backwards. This method may be called repeatedly to  
 \* iterate through the list backwards, or intermixed with calls to  
 \* {@link #next} to go back and forth. (Note that alternating calls  
 \* to {@code next} and {@code previous} will return the same  
 \* element repeatedly.)  
 \*  
 \* @return the previous element in the list  
 \* @throws NoSuchElementException if the iteration has no previous  
 \* element  
 \*/  
 E previous();  
  
 /\*\*  
 \* Returns the index of the element that would be returned by a  
 \* subsequent call to {@link #next}. (Returns list size if the list  
 \* iterator is at the end of the list.)  
 \*  
 \* @return the index of the element that would be returned by a  
 \* subsequent call to {@code next}, or list size if the list  
 \* iterator is at the end of the list  
 \*/  
 int nextIndex();  
  
 /\*\*  
 \* Returns the index of the element that would be returned by a  
 \* subsequent call to {@link #previous}. (Returns -1 if the list  
 \* iterator is at the beginning of the list.)  
 \*  
 \* @return the index of the element that would be returned by a  
 \* subsequent call to {@code previous}, or -1 if the list  
 \* iterator is at the beginning of the list  
 \*/  
 int previousIndex();  
  
  
 // Modification Operations  
  
 /\*\*  
 \* Removes from the list the last element that was returned by {@link  
 \* #next} or {@link #previous} (optional operation). This call can  
 \* only be made once per call to {@code next} or {@code previous}.  
 \* It can be made only if {@link #add} has not been  
 \* called after the last call to {@code next} or {@code previous}.  
 \*  
 \* @throws UnsupportedOperationException if the {@code remove}  
 \* operation is not supported by this list iterator  
 \* @throws IllegalStateException if neither {@code next} nor  
 \* {@code previous} have been called, or {@code remove} or  
 \* {@code add} have been called after the last call to  
 \* {@code next} or {@code previous}  
 \*/  
 void remove();  
  
 /\*\*  
 \* Replaces the last element returned by {@link #next} or  
 \* {@link #previous} with the specified element (optional operation).  
 \* This call can be made only if neither {@link #remove} nor {@link  
 \* #add} have been called after the last call to {@code next} or  
 \* {@code previous}.  
 \*  
 \* @param e the element with which to replace the last element returned by  
 \* {@code next} or {@code previous}  
 \* @throws UnsupportedOperationException if the {@code set} operation  
 \* is not supported by this list iterator  
 \* @throws ClassCastException if the class of the specified element  
 \* prevents it from being added to this list  
 \* @throws IllegalArgumentException if some aspect of the specified  
 \* element prevents it from being added to this list  
 \* @throws IllegalStateException if neither {@code next} nor  
 \* {@code previous} have been called, or {@code remove} or  
 \* {@code add} have been called after the last call to  
 \* {@code next} or {@code previous}  
 \*/  
 void set(E e);  
  
 /\*\*  
 \* Inserts the specified element into the list (optional operation).  
 \* The element is inserted immediately before the element that  
 \* would be returned by {@link #next}, if any, and after the element  
 \* that would be returned by {@link #previous}, if any. (If the  
 \* list contains no elements, the new element becomes the sole element  
 \* on the list.) The new element is inserted before the implicit  
 \* cursor: a subsequent call to {@code next} would be unaffected, and a  
 \* subsequent call to {@code previous} would return the new element.  
 \* (This call increases by one the value that would be returned by a  
 \* call to {@code nextIndex} or {@code previousIndex}.)  
 \*  
 \* @param e the element to insert  
 \* @throws UnsupportedOperationException if the {@code add} method is  
 \* not supported by this list iterator  
 \* @throws ClassCastException if the class of the specified element  
 \* prevents it from being added to this list  
 \* @throws IllegalArgumentException if some aspect of this element  
 \* prevents it from being added to this list  
 \*/  
 void add(E e);  
}

**二、方法分析**

首先看一下Iterator和ListIterator迭代器的方法有哪些。  
Iterator迭代器包含的方法有：

hasNext()：如果迭代器指向位置后面还有元素，则返回 true，否则返回false  
next()：返回集合中Iterator指向位置后面的元素  
remove()：删除集合中Iterator指向位置后面的元素

ListIterator迭代器包含的方法有：

add(E e): 将指定的元素插入列表，插入位置为迭代器当前位置之前  
hasNext()：以正向遍历列表时，如果列表迭代器后面还有元素，则返回 true，否则返回false  
hasPrevious():如果以逆向遍历列表，列表迭代器前面还有元素，则返回 true，否则返回false  
next()：返回列表中ListIterator指向位置后面的元素  
nextIndex():返回列表中ListIterator所需位置后面元素的索引  
previous():返回列表中ListIterator指向位置前面的元素  
previousIndex()：返回列表中ListIterator所需位置前面元素的索引  
remove():从列表中删除next()或previous()返回的最后一个元素（有点拗口，意思就是对迭代器使用hasNext()方法时，删除ListIterator指向位置后面的元素；当对迭代器使用hasPrevious()方法时，删除ListIterator指向位置前面的元素）  
set(E e)：从列表中将next()或previous()返回的最后一个元素返回的最后一个元素更改为指定元素e

三、分析

**3.1 相同点**  
都是迭代器，当需要对集合中元素进行遍历不需要干涉其遍历过程时，这两种迭代器都可以使用。

**3.2 不同点**  
1.使用范围不同，Iterator可以应用于所有的集合，Set、List和Map和这些集合的子类型。而ListIterator只能用于List及其子类型。  
2.ListIterator有add方法，可以向List中添加对象，而Iterator不能。  
3.ListIterator和Iterator都有hasNext()和next()方法，可以实现顺序向后遍历，但是ListIterator有hasPrevious()和previous()方法，可以实现逆向（顺序向前）遍历。Iterator不可以。  
4.ListIterator可以定位当前索引的位置，nextIndex()和previousIndex()可以实现。Iterator没有此功能。  
5.都可实现删除操作，但是ListIterator可以实现对象的修改，set()方法可以实现。Iterator仅能遍历，不能修改。

参考代码：ListIteratorTest

public class ListIteratorTest {  
  
 public static void main(String[] args) {  
 List<String> list = new ArrayList<String>();  
 list.add("beijing");  
 list.add("shanghai");  
 list.add("guangzhou");  
 list.add("tangshan");  
   
 Iterator<String> it = list.iterator();  
 while (it.hasNext()) {  
 String item = it.next();  
 if (item.equals("shanghai")) {  
 it.remove();  
 }  
 }  
   
 Iterator<String> it2 = list.iterator();  
 while (it2.hasNext()) {  
 String item = it2.next();  
 System.out.println("Iterator:" + item);  
 }  
  
   
 List<String> list2 = new ArrayList<String>();  
 list2.add("beijing");  
 list2.add("shanghai");  
 list2.add("guangzhou");  
 list2.add("tangshan");  
 ListIterator<String> iter = list2.listIterator();  
 String first = iter.next();  
   
 System.out.println("first:" + first);  
 iter.add("tianjing");  
   
 // 遍历List元素  
 System.out.println("遍历List中元素，方法一：");  
 for (String str : list2)  
 System.out.println(str + " ");  
   
 iter = list2.listIterator();  
 System.out.println("遍历List中元素，方法二：");  
 while (iter.hasNext()) {  
 System.out.println("下一个元素索引：" + iter.nextIndex());  
 System.out.println("下一个元素：" +iter.next());  
 }  
 }  
  
}

————————————————  
版权声明：本文为CSDN博主「xiangyuenacha」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/xiangyuenacha/article/details/84253630

### 17.数组 (Array) 和列表 (ArrayList) 有什么区别？什么时候应该使用 Array 而不是 ArrayList？

答案：https://www.cnblogs.com/yonyong/p/9323546.html

下面列出了Array和ArrayList的不同点：  
Array可以包含基本类型和对象类型，ArrayList只能包含对象类型。  
Array大小是固定的，ArrayList的大小是动态变化的。  
ArrayList提供了更多的方法和特性，比如：addAll()，removeAll()，iterator()等等。  
对于基本类型数据，集合使用自动装箱来减少编码工作量。但是，当处理固定大小的基本数据类型的时候，这种方式相对比较慢

ArrayList可以算是Array的加强版，（对array有所取舍的加强）。

另附分类比较：

**存储内容比较：**

* Array数组可以包含基本类型和对象类型，
* ArrayList却只能包含对象类型。

但是需要注意的是：Array数组在存放的时候一定是同种类型的元素。ArrayList就不一定了，因为ArrayList可以存储Object。

**空间大小比较：**

* Array数组它的空间大小是固定的，空间不够时也不能再次申请，所以需要事前确定合适的空间大小。
* ArrayList的空间是动态增长的，如果空间不够，它会创建一个空间比原空间大约0.5倍的新数组，然后将所有元素复制到新数组中，接着抛弃旧数组。而且，每次添加新的元素的时候都会检查内部数组的空间是否足够。（比较麻烦的地方）。

**附上arraylist扩充机制：**

**newCapacity=oldCapacity+(oldCapacity>>1)（注： >>1:右移1位，相当于除以2，例如10>>1 得到的就是5）但由于源码里（不再分析，这里简要略过）传过来的minCapcatiy的值是size+1，能够实现grow方法调用就肯定是(size+1)>elementData.length的情况，所以size就是初始最大容量或上一次扩容后达到的最大容量，所以才会进行扩容。因此，扩容后的大小应该是原来的1.5倍+1**

**方法上的比较：**  
ArrayList作为Array的增强版，当然是在方法上比Array更多样化，比如添加全部addAll()、删除全部removeAll()、返回迭代器iterator()等。

**适用场景：**  
如果想要保存一些在整个程序运行期间都会存在而且大小不变的数据，我们可以将它们放进一个全局数组里，但是如果我们单纯只是想要以数组的形式保存数据，而不对数据进行增加等操作，只是方便我们进行查找的话，那么，我们就选择ArrayList。而且还有一个地方是必须知道的，就是如果我们需要对元素进行频繁的移动或删除，或者是处理的是超大量的数据，那么，使用ArrayList就真的不是一个好的选择，因为它的效率很低，使用数组进行这样的动作就很麻烦，那么，我们可以考虑选择LinkedList。arraylist和linkedlist的区别在哪里，什么时候使用，可以参考我的另一篇博客：[《ArrayList和LinkedList有什么区别？》](https://www.cnblogs.com/yonyong/p/9323588.html)

**作 者**：[**yonyong**](https://www.cnblogs.com/yonyong)  
**出 处**：https://www.cnblogs.com/yonyong/p/9323546.html  
**关于博主**：编程路上的苦行僧，热爱技术，喜欢专研。评论和私信会在第一时间回复。或者直接私信我。本科毕业于2019年六月，现居南京。猎头可直接加微，备注来源蟹蟹  
**版权声明**：署名 - 非商业性使用 - 禁止演绎，[协议普通文本](https://creativecommons.org/licenses/by-nc-nd/4.0/) | [协议法律文本](https://creativecommons.org/licenses/by-nc-nd/4.0/legalcode)。  
**声援博主**：如果您觉得文章对您有帮助，可以点击文章右下角**【**[**推荐**](javascript:void(0);)**】**一下。您的鼓励是博主的最大动力！

### 18.Java 集合类框架的最佳实践有哪些？

答案https://www.cnblogs.com/yonyong/p/9327284.html

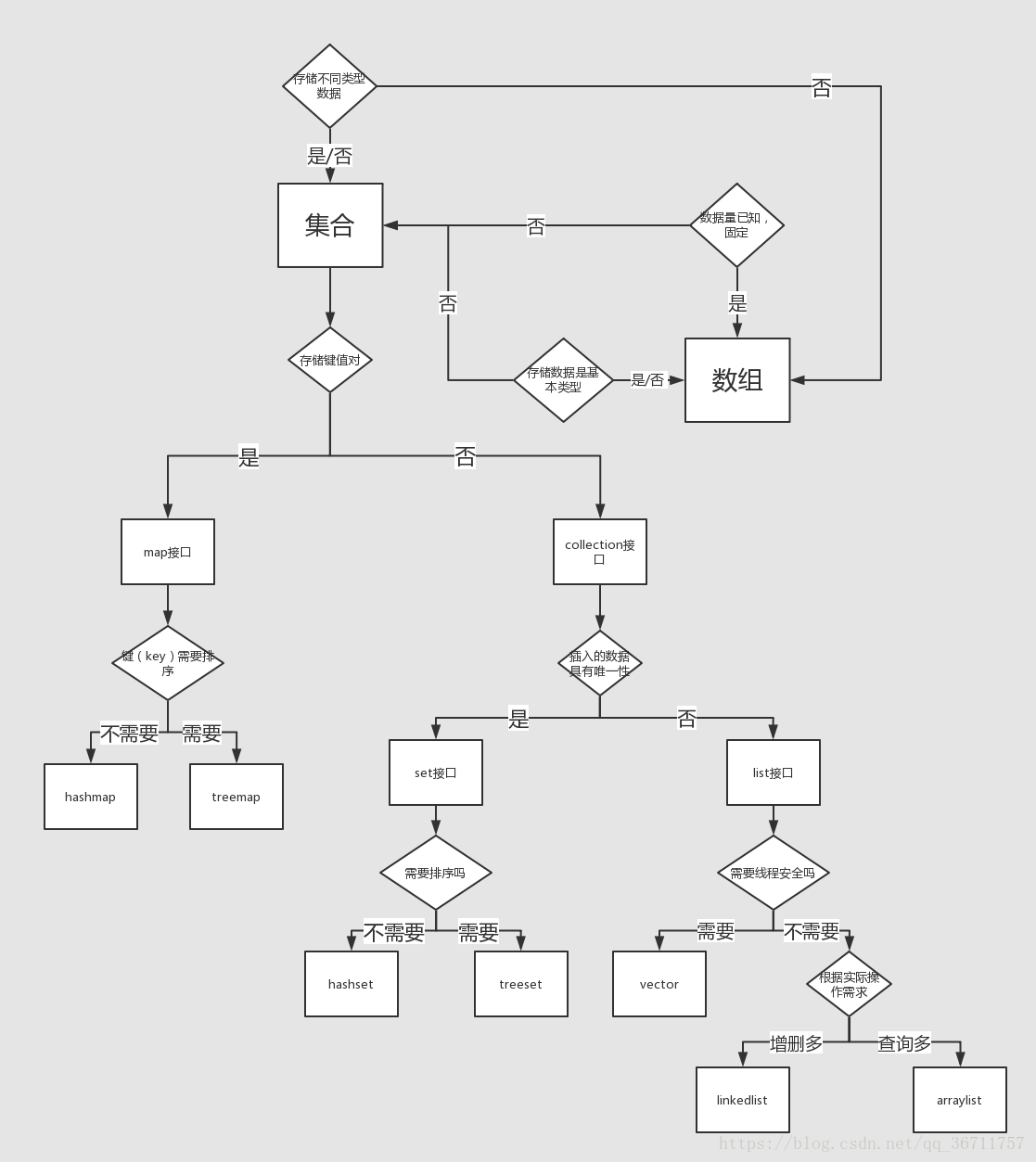
**根据应用的需要选择合适的集合对性能是非常重要的。**

**如果一个集合的元素数量是固定的，而且我们能够提前知道固定的数量，那么就可以使用数组，而不是ArrayList。**

**每个集合都可以设置初始容量，如果我们提前能够估算出它的初始容量，那么就可以避免重新计算它的hash值与扩容。**

**为了保证程序的类型安全、健壮性与可读性，一般我们都需要使用泛型。泛型可以避免运行时的ClassCastException。同时如果使用不可变类作为map的键，那么可以避免为我们自己实现的类实现equals()与hashcode()方法。**

**编程的时候接口优于实现。 如果底层的集合实际上是空的情况下，返回的集合要为长度为0的集合或者数组，不能为空。 总而言之一句话，如果频繁用于查找，则用数组类型的集合，如果频繁用于删除，则使用链表类型的集合，下面是一张总的集合适用场合的图片**

[](https://images2018.cnblogs.com/blog/1302153/201807/1302153-20180718090736798-274691787.png)

### 19.Set 里的元素是不能重复的，那么用什么方法来区分重复与否呢？是用 == 还是 equals()？它们有何区别？

同题目：7

### 20.Comparable 和 Comparator 接口是干什么的？列出它们的区别

[答案链接](https://www.cnblogs.com/williamjie/p/11164313.html)https://www.cnblogs.com/williamjie/p/11164313.html

java提供了只包含一个compareTo()方法的Comparable接口。这个方法可以个给两个对象排序。具体来说，它返回负数，0，正数来表明已经存在的对象小于，等于，大于输入对象。

Java提供了包含compare()和equals()两个方法的Comparator接口。compare()方法用来给两个输入参数排序，返回负数，0，正数表明第一个参数是小于，等于，大于第二个参数。equals()方法需要一个对象作为参数，它用来决定输入参数是否和comparator相等。只有当输入参数也是一个comparator并且输入参数和当前comparator的排序结果是相同的时候，这个方法才返回true。

Comparable & Comparator 都是用来实现集合中元素的比较、排序的，只是

Comparable 是在集合内部定义的方法实现的排序，

Comparator 是在集合外部实现的排序，

所以，如想实现排序，就需要在集合外定义 Comparator 接口的方法或在集合内实现 Comparable 接口的方法。 Comparator位于包java.util下，而Comparable位于包 java.lang下 Comparable 是一个对象本身就已经支持自比较所需要实现的接口（如 String、Integer 自己就可以完成比较大小操作，已经实现了Comparable接口） 自定义的类要在加入list容器中后能够排序，可以实现Comparable接口，在用Collections类的sort方法排序时，如果不指定Comparator，那么就以自然顺序排序， 这里的自然顺序就是实现Comparable接口设定的排序方式。 而 Comparator 是一个专用的比较器，当这个对象不支持自比较或者自比较函数不能满足你的要求时，你可以写一个比较器来完成两个对象之间大小的比较。 可以说一个是自已完成比较，一个是外部程序实现比较的差别而已。

用 Comparator 是策略模式（strategy design pattern），就是不改变对象自身，而用一个策略对象（strategy object）来改变它的行为。 比如：你想对整数采用绝对值大小来排序，Integer 是不符合要求的，你不需要去修改 Integer 类（实际上你也不能这么做）去改变它的排序行为，只要使用一个实现了 Comparator 接口的对象来实现控制它的排序就行了。

两种方式，各有各的特点：使用Comparable方式比较时，我们将比较的规则写入了比较的类型中，其特点是高内聚。但如果哪天这个规则需要修改，那么我们必须修改这个类型的源代码。如果使用Comparator方式比较，那么我们不需要修改比较的类，其特点是易维护，但需要自定义一个比较器，后续比较规则的修改，仅仅是改这个比较器中的代码即可。

答案二链接https://blog.csdn.net/lsqingfeng/article/details/80342620

Comparable和Comparator都是接口，都是用来比较和排序的，那么他们两个之间到底有这什么样的区别呢？

一、Comparable用法

在给集合排序的时候，我们需要用到一个工具类叫做Collections，这个工具类可以用来给集合排序,详见如下代码：

List<Integer> list = new ArrayList<>();  
list.add(14);  
list.add(30);  
list.add(3);  
list.add(12);  
Collections.sort(list);  
System.out.println(list);

这个打印的结果是：[3,12，14,30];

很显然，Collections对于Integer类型的数组默认的排序结果是升序的  
  
那么如果我们创建一个自定义类型的Person数组能否进行排序呢，大家可以用代码试一下，结果是不可以的，为什么会有这样的问题呢，我们去看一下Collections中的sort方法，就可以发现问题：

public static <T extends Comparable<? super T>> void sort(List<T> list) {  
 list.sort(null);  
 }

在泛型的规则中，有一个T extends Comparable的泛型通配符 ，对于要排序的list中的T进行了一个限制，要求集合中的T必须要实现Comparable接口，我们可以按照这个思路，写一个Person类，实现Comparable接口，而这个接口中，有一个抽象方法需要我们实现，这个方法就是CompareTo

public class Person implements Comparable<Person>{  
 String name;  
 Integer age;  
 public Person(String name, int age) {  
 super();  
 this.name = name;  
 this.age = age;  
 }  
 public Person() {  
 super();  
 }  
 @Override  
 public String toString() {  
 return "Person [name=" + name + ", age=" + age + "]";  
 }  
 public String getName() {  
 return name;  
}  
public void setName(String name) {  
 this.name = name;  
}  
public Integer getAge() {  
 return age;  
}  
public void setAge(Integer age) {  
 this.age = age;  
}  
//排序的规则  
public int compareTo(Person o) {  
 //引用类型（可以排序的类型）可以直接调用CompareTo方法  
 //基本类型--使用 减   
 //return this.age - o.age;//用this对象 - 参数中的对象，是按照该属性的升序进行的排列  
 //return o.age - this.age;  
   
 //return this.name.compareTo(o.name);  
 //return o.name.compareTo(this.name);  
   
 return this.age.compareTo(o.age);  
}

而compareTo方法，实际上就是我们需要设置的排序的规则，到底按照什么样的方式进行排序。简单的记，使用this对象和参数比较，就是升序，反之就是降序。所以我们如果想要让Person集合中的对象按照年龄进行降序排列，就可以使用o.age -this.age；（基本类型可以使用减法替代compareTo）；

这样，你再次使用Collections.sort就可以对Person的List进行排序了，排序的结果是按照年龄的降序。

总结一下，如果我们想要让一个List可以使用Collections.sort(list) 的方法进行排序，则必须要求集合中的元素类型，实现Comparable接口，也就是让他具备比较能力，这也是为什么Integer类型的数组可以排序，就是因为Integer已经实现了该接口，并且他是按照升序的规则实现的，这也就解释了为什么上边的第一个程序得到的结果是升序。

好了那么既然Integer是按照升序的方式实现的排序，那么如果我想要得到一个降序的Integer集合该怎么办呢？难道就实现不了了么？我们接着来看下一个接口。

二、Comparator

正如上文所说，对于已经实现了Comparable接口的集合，或者是我压根就不想实现Comparable接口的集合难道就排不了序了么，或者就无法更改排序的规则了么，实际上不是的，我们可以通过另一种方式来排序，就是利用Comparator接口。

在集合的工具类中种还有这样的一个方法：

public static <T> void sort(List<T> list, Comparator<? super T> c)

我们可以通过这个方法实现上面的需求：

Collections.sort(list,new Comparator<Integer>(){  
 @Override  
 public int compare(Integer o1, Integer o2) {  
 return o2 - o1;  
 }  
 });

比如这段代码，就实现了一个Integer集合的降序排列。这个接口中有一个方法叫做compare，里边包含两个参数：如果用第一个和第二个做比较得到的就是升序，反之得到的就是降序。同样的你也可以使用这种方式对我们自己定义的类进行排序。

好了，这就是Comparable接口和Comparator接口的用法，另外要注意：

Comparable接口位于 java.lang包下，Comparator接口位于java.util包下。  
  
Comparable: 内部比较器，一个类如果想要使用 Collections.sort(list) 方法进行排序，则需要实现该接口  
  
Comparator: 外部比较器用于对那些没有实现Comparable接口或者对已经实现的Comparable中的排序规则不满意进行排序.无需改变类的结构，更加灵活。（策略模式）

————————————————  
版权声明：本文为CSDN博主「一缕82年的清风」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/lsqingfeng/article/details/80342620

### 21.Collection 和 Collections 的区别。

引用地址：https://www.iteye.com/blog/pengcqu-492196

**1、java.util.Collection**

是一个集合接口（集合类的一个顶级接口）。它提供了对集合对象进行基本操作的通用接口方法。Collection接口在Java 类库中有很多具体的实现。Collection接口的意义是为各种具体的集合提供了最大化的统一操作方式，其直接继承接口有List与Set。

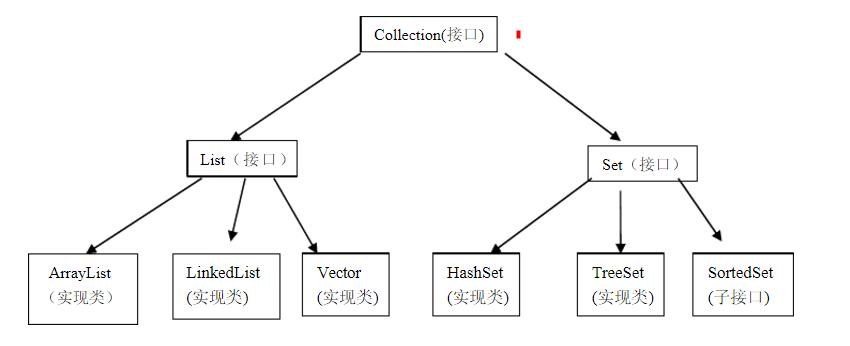
Collection  
├List  
│├LinkedList  
│├ArrayList  
│└Vector  
│ └Stack  
└Set

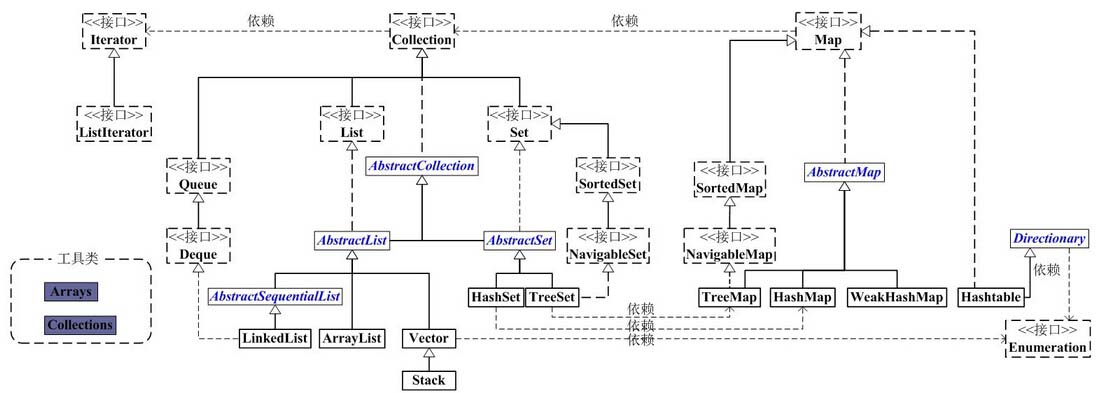
**2、java.util.Collections**

是一个包装类（工具类/帮助类）。它包含有各种有关集合操作的静态多态方法。此类不能实例化，就像一个工具类，用于对集合中元素进行排序、搜索以及线程安全等各种操作，服务于Java的Collection框架。

import java.util.ArrayList;  
import java.util.Collections;  
import java.util.List;  
   
public class TestCollections {  
   
 public static void main(String args[]) {  
 //注意List是实现Collection接口的  
 List list = new ArrayList();  
 double array[] = { 112, 111, 23, 456, 231 };  
 for (int i = 0; i < array.length; i++) {  
 list.add(new Double(array[i]));  
 }  
 Collections.sort(list);  
 for (int i = 0; i < array.length; i++) {  
 System.out.println(list.get(i));  
 }  
 // 结果：23.0 111.0 112.0 231.0 456.0  
 }  
}

以下为集合的依赖关系图：





建议再详细读取Collections工具类详解 https://blog.csdn.net/weixin\_37730482/article/details/70677979

集合（一）：Collectionhttps://blog.csdn.net/weixin\_34378767/article/details/91775594

### 22.说说 ArrayList,Vector, LinkedList 的存储性能和特性

同1

## 二、JVM与调优21题

### 1、Java 类加载过程？

答案：https://www.cnblogs.com/luohanguo/p/9469851.html

[深入浅出Java类加载过程](https://www.cnblogs.com/luohanguo/p/9469851.html)

学习笔记二之Java虚拟机中类加载的过程

1、加载

2、链接

3、初始化

4、小结

5、附录

当程序要使用某个类时，如果该类还未被加载到内存中，则系统会通过**加载，连接，初始化**三步来实现这个类进行初始化。

**1.** **加载**

加载，是指<u>Java虚拟机查找字节流（查找.class文件），并且根据字节流创建java.lang.Class对象的过程</u>。这个过程，将类的.class文件中的二进制数据读入内存，放在运行时区域的**方法区内**。然后在堆中创建java.lang.Class对象，用来封装类在方法区的**数据结构**。

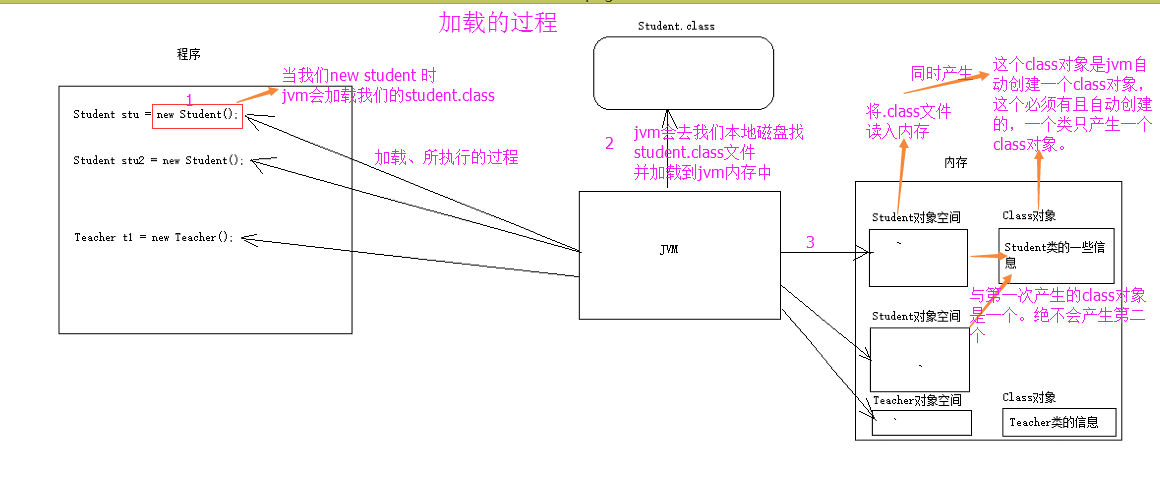
类加载阶段：

（1）Java虚拟机将.class文件读入内存，并为之创建一个Class对象。

（2）任何类被使用时系统都会为其**创建一个且仅有一个**Class对象。

（3）这个Class对象描述了这个类创建出来的对象的所有信息，比如有哪些构造方法，都有哪些成员方法，都有哪些成员变量等。

Student类加载过程图示：



**2.** **链接**

链接包括**验证、准备以及解析**三个阶段。

（1）验证阶段。主要的目的是确保被加载的类（.class文件的字节流）满足Java虚拟机规范，不会造成安全错误。

（2）准备阶段。负责为类的静态成员分配内存，并设置默认初始值。

（3）解析阶段。将类的二进制数据中的***符号引用***替换为**直接引用**。

**说明：**

**符号引用**。即一个字符串，但是这个字符串给出了一些能够唯一性识别一个方法，一个变量，一个类的相关信息。

**直接引用**。可以理解为一个**内存地址**，或者**一个偏移量**。比如类方法，类变量的直接引用是指向方法区的指针；而实例方法，实例变量的直接引用则是从实例的头指针开始算起到这个实例变量位置的偏移量。

举个例子来说，现在调用方法hello()，这个方法的地址是0xaabbccdd，那么hello就是符号引用，0xaabbccdd就是直接引用。

在解析阶段，虚拟机会把所有的类名，方法名，字段名这些符号引用替换为具体的内存地址或偏移量，也就是直接引用。

**3.** **初始化**

初始化，则是**为标记为常量值的字段赋值的过程**。换句话说，只对static修饰的变量或语句块进行初始化。

如果初始化一个类的时候，其父类尚未初始化，则优先初始化其父类。

如果同时包含多个静态变量和静态代码块，则按照自上而下的顺序依次执行。

**4.** **小结**

类加载过程只是一个类生命周期的一部分，在其前，有编译的过程，只有对源代码编译之后，才能获得能够被虚拟机加载的字节码文件；在其后还有具体的类使用过程，当使用完成之后，还会在方法区垃圾回收的过程中进行卸载（垃圾回收）。

**5.** **附录**

常见问题：在自己的项目里新建一个java.lang包，里面新建了一个String类，能代替系统String吗？

不能，因为根据类加载的双亲委派机制，会将请求转发给父类加载器，**父类加载器发现冲突了String就不会加载了**。

**6. 参考**

【1】 周志明. 深入理解Java虚拟机：JVM高级特性与最佳实践（第二版）【M】.北京：机械工业出版社，2013

【2】爱飞翔，周志明（译）.Java虚拟机规范（Java SE 8版）【M】.北京：机械工业出版社，2015.

【3】https://blog.csdn.net/ln152315/article/details/79223441

【4】https://blog.csdn.net/sinat\_38259539/article/details/71794617

### 2.描述一下 JVM 加载 Class 文件的原理机制?

答案地址：https://blog.csdn.net/weisg81/article/details/77415937

【JVM加载class文件的原理机制简单总结】

1、java中的类

2、类装载的方式

3、类加载的动态性体现

4、java类装载器

5、类加载器之间是如何协调工作的

6、类的加载过程

7、类的初始化

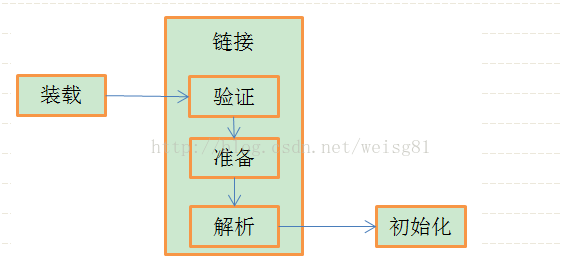
Java中的所有类，必须被装载到jvm中才能运行，这个装载工作是由jvm中的**类装载器**完成的,类装载器所做的工作实质是<u>\*把类文件从硬盘读取到内存中\*</u>   
**1、java中的类大致分为三种：**  
 1).系统类   
 2).扩展类   
 3).由程序员自定义的类  
**2、类装载方式，有两种**  
 1).隐式装载， 程序在运行过程中当碰到通过new 等方式生成对象时，隐式调用类装载器加载对应的类到jvm中  
 2).显式装载， 通过class.forName()等方法，显式加载需要的类   
**3、类加载的动态性体现**  
 一个应用程序总是由n多个类组成，Java程序启动时，并不是一次把所有的类全部加载后再   
运行，它总是先把保证程序运行的基础类一次性加载到jvm中，其它类等到**jvm用到的时候再加载**，这样的好处是节省了内存的开销，因为java最早就是为嵌入式系统而设计的，内存宝贵，这是一种可以理解的机制，而用到时再加载这也是java动态性的一种体现   
**4、java类装载器**  
 Java中的类装载器实质上也是类，功能是把类载入jvm中，值得注意的是jvm的类装载器并不是一个，而是三个，层次结构如下：   
 Bootstrap Loader - 负责加载系统类   
 | ExtClassLoader - 负责加载扩展类   
 | AppClassLoader - 负责加载应用类   
 为什么要有三个类加载器，一方面是**分工**，各自负责各自的区块，另一方面为了**实现委托模型**，下面会谈到该模型  
1) Bootstrap类加载器 – JRE/lib/rt.jar  
2) Extension类加载器 – JRE/lib/ext或者java.ext.dirs指向的目录  
3) Application类加载器 – CLASSPATH环境变量, 由-classpath或-cp选项定义,或者是JAR中的Manifest的classpath属性定义.  
**5、类加载器之间是如何协调工作的**  
类加载器的工作原理基于三个机制：**委托、可见性和单一性**  
**（1）、委托机制**  
当一个类加载和初始化的时候，类仅在有需要加载的时候被加载。假设你有一个应用需要的类叫作Abc.class，

首先加载这个类的请求由 Application类加载器委托给它的父类加载器Extension类加载器，然后再委托给Bootstrap类加载器。Bootstrap类加载器 会先看看rt.jar中有没有这个类，因为并没有这个类，所以这个请求由回到Extension类加载器，它会查看jre/lib/ext目录下有没有这 个类，如果这个类被Extension类加载器找到了，那么它将被加载，而Application类加载器不会加载这个类；而如果这个类没有被 Extension类加载器找到，那么再由Application类加载器从classpath中寻找。记住**classpath定义的是类文件的加载目 录**，而PATH是定义的是**可执行程序如javac，java等的执行路径**。  
**（2）、可见性机制**  
可见性的原理是子类的加载器可以看见所有的父类加载器加载的类，而父类加载器看不到子类加载器加载的类。  
**（3）、单一性机制**  
根据这个机制，父加载器加载过的类不能被子加载器加载第二次。虽然重写违反委托和单一性机制的类加载器是可能的，但这样做并不可取。

**线程上下文类加载器**（context class loader）是从 JDK 1.2 开始引入的。类 Java.lang.Thread中的方法 getContextClassLoader()和 setContextClassLoader(ClassLoader cl)用来获取和设置线程的上下文类加载器。如果没有通过 setContextClassLoader(ClassLoader cl)方法进行设置的话，线程将***继承其父线程的上下文类加载器***。Java 应用运行的初始线程的上下文类加载器是**系统类加载器**。在线程中运行的代码可以通过此类加载器来加载类和资源。  
 前面提到的类加载器的代理模式并不能解决 Java 应用开发中会遇到的类加载器的全部问题。Java 提供了很多服务提供者接口（Service Provider Interface，SPI），允许第三方为这些接口提供实现。常见的 SPI 有 JDBC、JCE、JNDI、JAXP 和 JBI 等。这些 SPI 的接口由 Java 核心库来提供，如 JAXP 的 SPI 接口定义包含在 javax.xml.parsers包中。这些 SPI 的实现代码很可能是作为 Java 应用所依赖的 jar 包被包含进来，可以通过类路径（CLASSPATH）来找到，如实现了 JAXP SPI 的 Apache Xerces所包含的 jar 包。SPI 接口中的代码经常需要加载具体的实现类。如 JAXP 中的 javax.xml.parsers.DocumentBuilderFactory类中的 newInstance()方法用来生成一个新的 DocumentBuilderFactory的实例。这里的实例的真正的类是继承自 javax.xml.parsers.DocumentBuilderFactory，由 SPI 的实现所提供的。如在 Apache Xerces 中，实现的类是 org.apache.xerces.jaxp.DocumentBuilderFactoryImpl。而问题在于，SPI 的接口是 Java 核心库的一部分，是由引导类加载器来加载的；SPI 实现的 Java 类一般是由系统类加载器来加载的。引导类加载器是无法找到 SPI 的实现类的，因为它只加载 Java 的核心库。它也不能代理给系统类加载器，因为它是系统类加载器的祖先类加载器。也就是说，类加载器的代理模式无法解决这个问题。  
 线程上下文类加载器正好解决了这个问题。如果不做任何的设置，Java 应用的线程的上下文类加载器默认就是**系统上下文类加载器**。在 SPI 接口的代码中使用线程上下文类加载器，就可以成功的加载到 SPI 实现的类。线程上下文类加载器在很多 SPI 的实现中都会用到。

**6、类的加载过程**  
JVM将类加载过程分为三个步骤：**装载（Load），链接（Link）和初始化(Initialize)**.

链接又分为三个步骤，如下图所示：



1) 装载：查找并加载类的二进制数据；  
2)链接：  
验证：确保被加载类的正确性；  
准备：为类的静态变量分配内存，并将其初始化为默认值；  
解析：把类中的符号引用转换为直接引用；  
3)初始化：为类的静态变量赋予正确的初始值；

那为什么我要有验证这一步骤呢？

首先如果由编译器生成的class文件，它肯定是符合JVM字节码格式的，但是万一有高手自己写一个class文件，让JVM加载并运行，用于恶意用途，就不妙了，因此这个class文件要先过验证这一关，不符合的话不会让它继续执行的，也是为了安全考虑吧。  
 准备阶段和初始化阶段看似有点牟盾，其实是不牟盾的，如果类中有语句：private static int a = 10，它的执行过程是这样的，首先字节码文件被加载到内存后，先进行链接的验证这一步骤，验证通过后准备阶段，给a分配内存，因为变量a是static的，所以此时a等于int类型的默认初始值0，即a=0,然后到解析（后面在说），到初始化这一步骤时，才把a的真正的值10赋给a,此时a=10。  
**7、类的初始化**  
类什么时候才被初始化：  
1）创建类的实例，也就是new一个对象  
2）访问某个类或接口的静态变量，或者对该静态变量赋值  
3）调用类的静态方法  
4）反射（Class.forName("com.lyj.load")）  
5）初始化一个类的子类（会首先初始化子类的父类）  
6）JVM启动时标明的启动类，即文件名和类名相同的那个类  
 只有这6中情况才会导致类的类的初始化。  
 类的初始化步骤：  
 1）如果这个类还没有被加载和链接，那先进行加载和链接  
 2）假如这个类存在直接父类，并且这个类还没有被初始化（注意：在一个类加载器中，类只能初始化一次），那就初始化直接的父类（不适用于接口）  
 3)加入类中存在初始化语句（如static变量和static块），那就依次执行这些初始化语句。  
————————————————  
版权声明：本文为CSDN博主「wbsjhbl」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/weisg81/article/details/77415937

### 3.Java 内存分配。

答案引用-https://blog.csdn.net/u014596551/article/details/79757932

Java基础（三）java内存分配  
原创TISIULS 最后发布于2018-03-30 15:54:49 阅读数 4733 收藏

1、Java中的内存分配

2、一维数组的内存

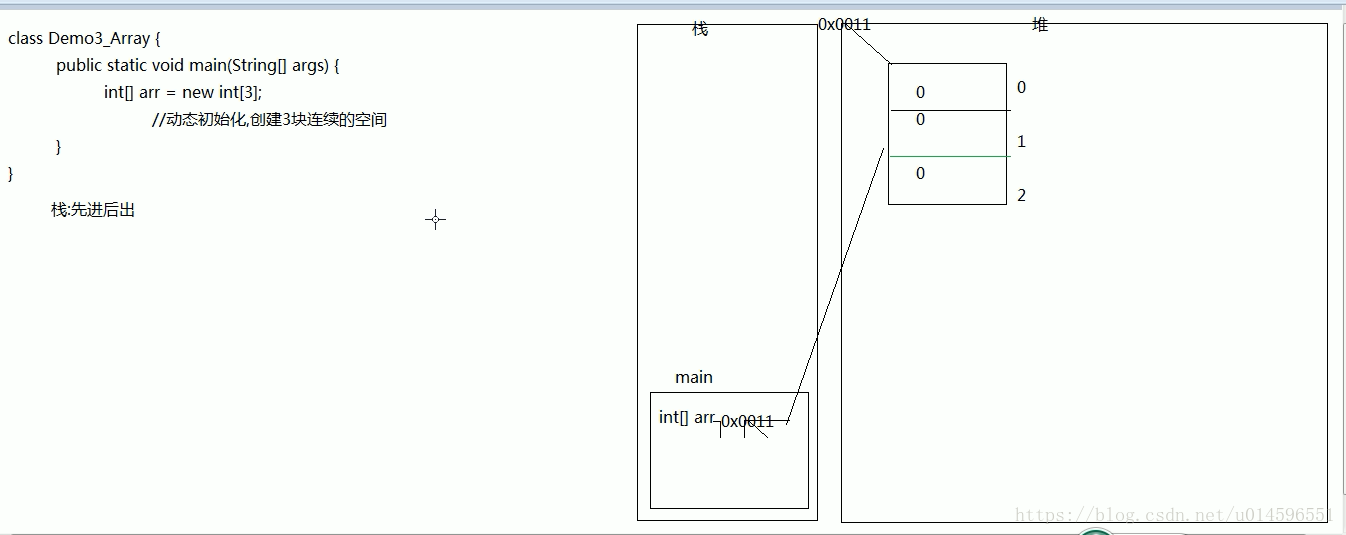
3.二维数组的内存

4.Java中的参数传递问题

5.对象

**1、Java中的内存分配**

* A:栈(掌握)
  + 存储局部变量
  + 局部变量：定义在方法声明上和方法中的变量
* B:堆(掌握)
  + 存储new出来的数组或对象
* C:方法区
  + 代码
* D:本地方法区
  + 和系统相关
* E:寄存器
  + 给CPU使用



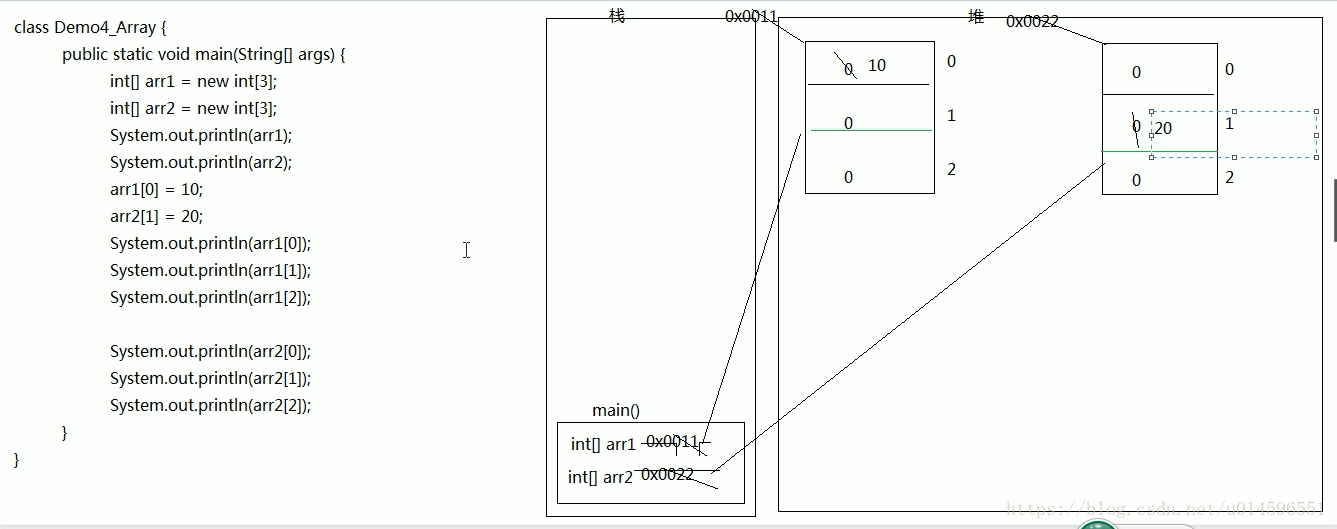
**2.一维数组的内存**  
**（1）一个数组的内存图解**

首先是方法进栈，main方法圧进栈，随后变量进栈，new的对象进入堆，要使用时取堆里的地址值



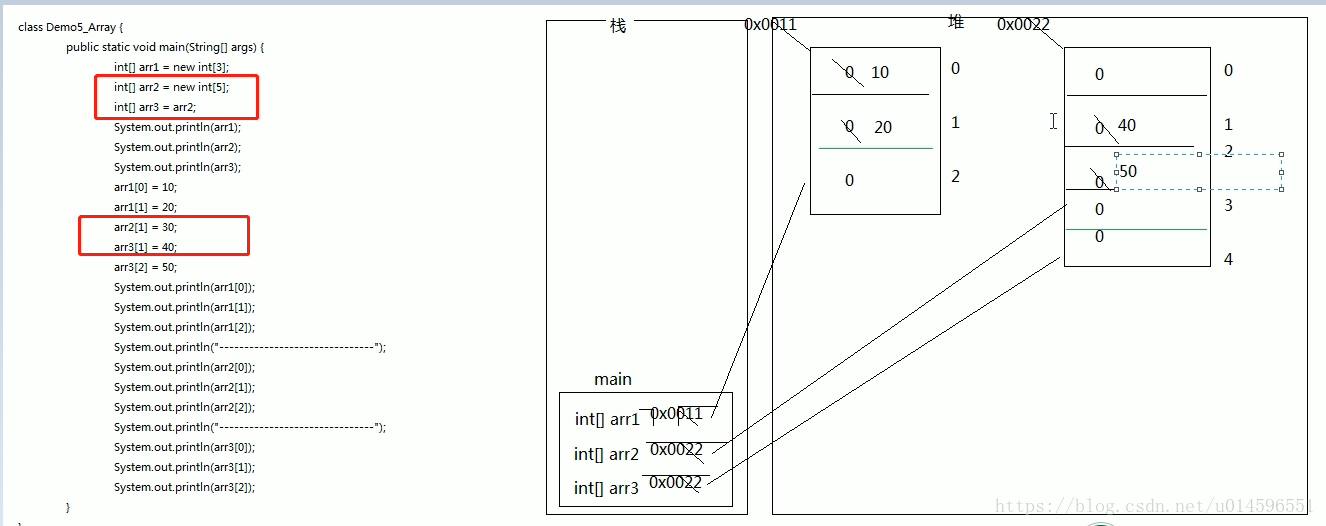
**（2）两个数组的内存图解**

会在堆中新建连个地址空间

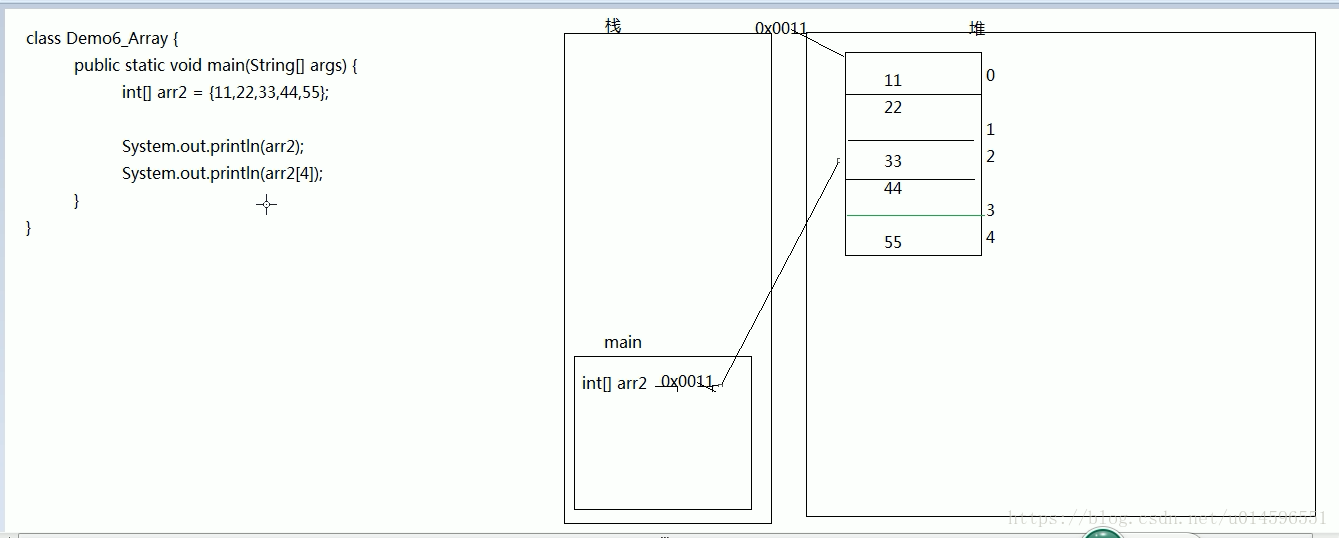


**（3）三个引用两个数组**

两个引用指向同一个实体，共用一个地址



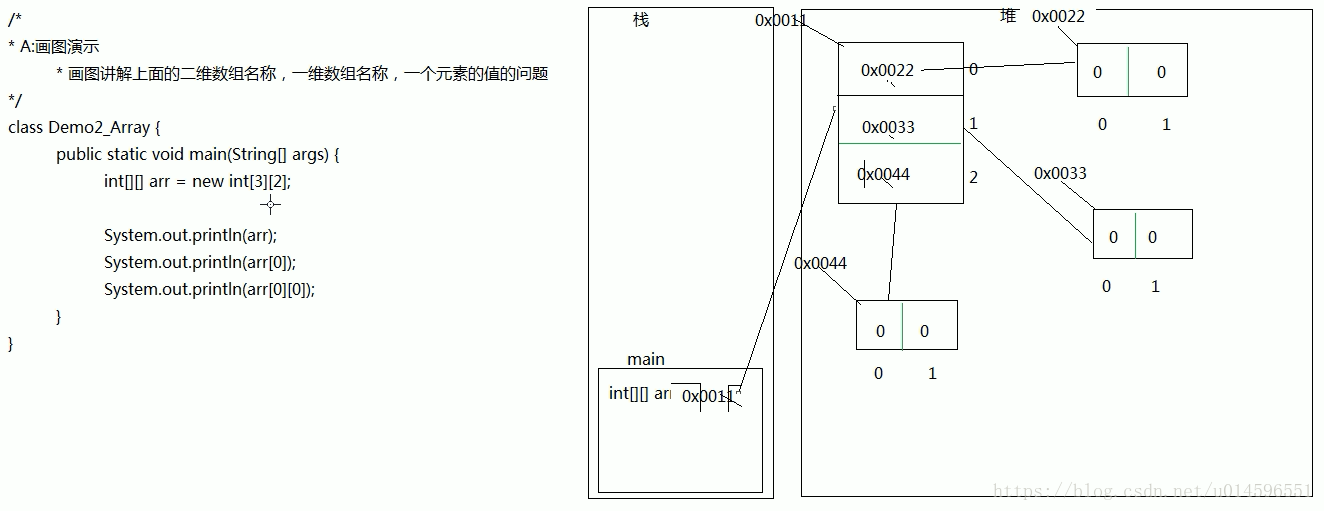
**（4）数组的初始化静态初始化**



**3.二维数组的内存**  
**（1）二维数组格式**

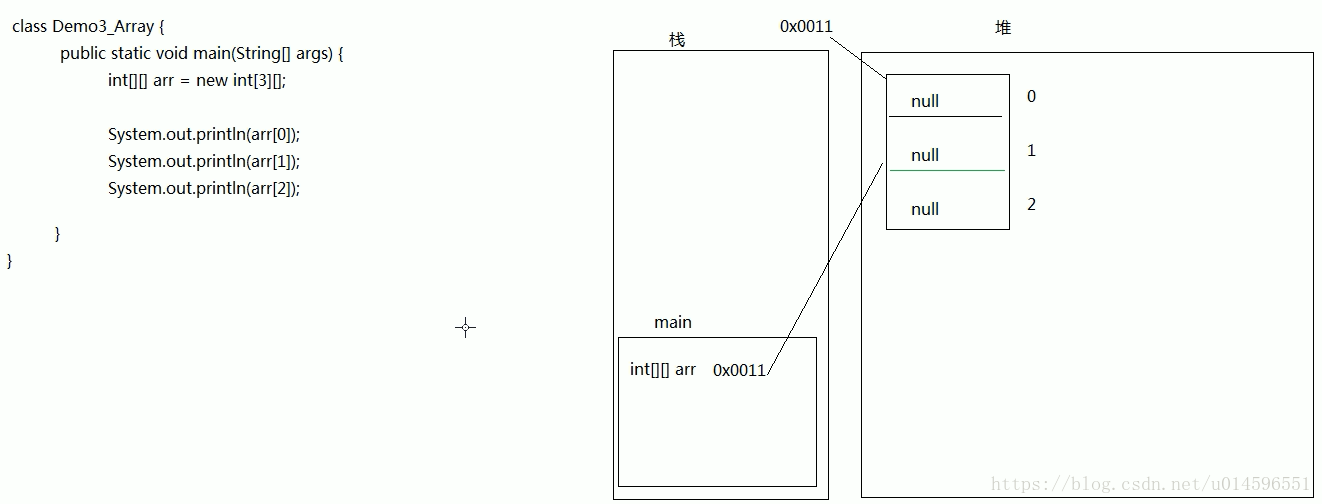
int [][] arr = new int[3][2]

在堆中先建立一个一维数组，一维数组的每个索引存的是地址, 默认初始化值是null



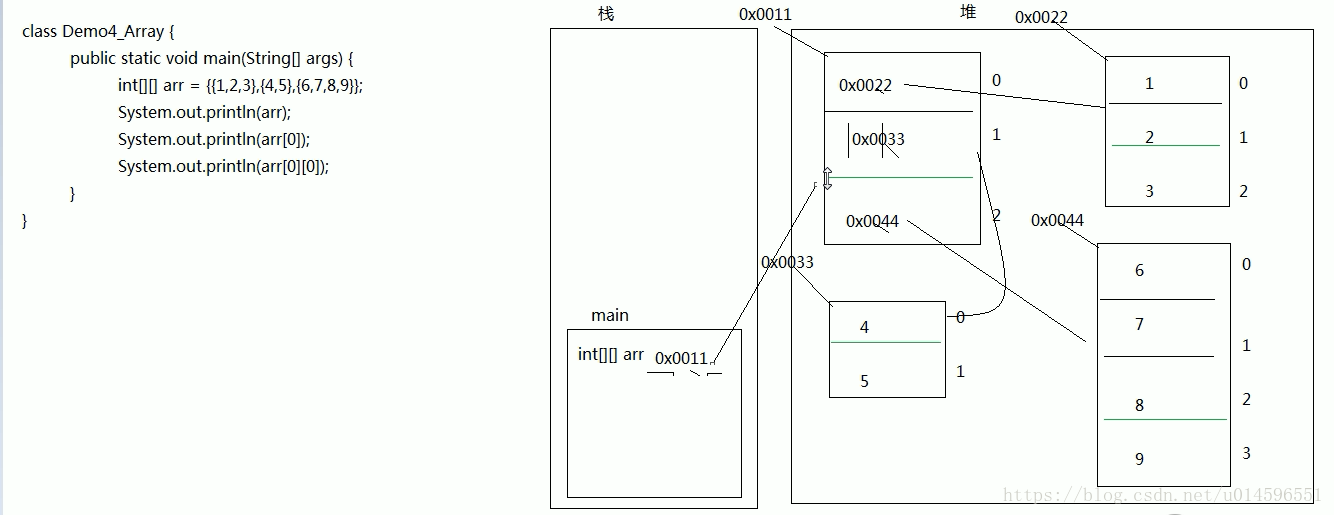
**（2）二维数组格式**

int arr[][] = new int[3][];



**（3）二维数组格式**

int[][] arr = {{1,2,3},{4,5},{6,7,8,9}}



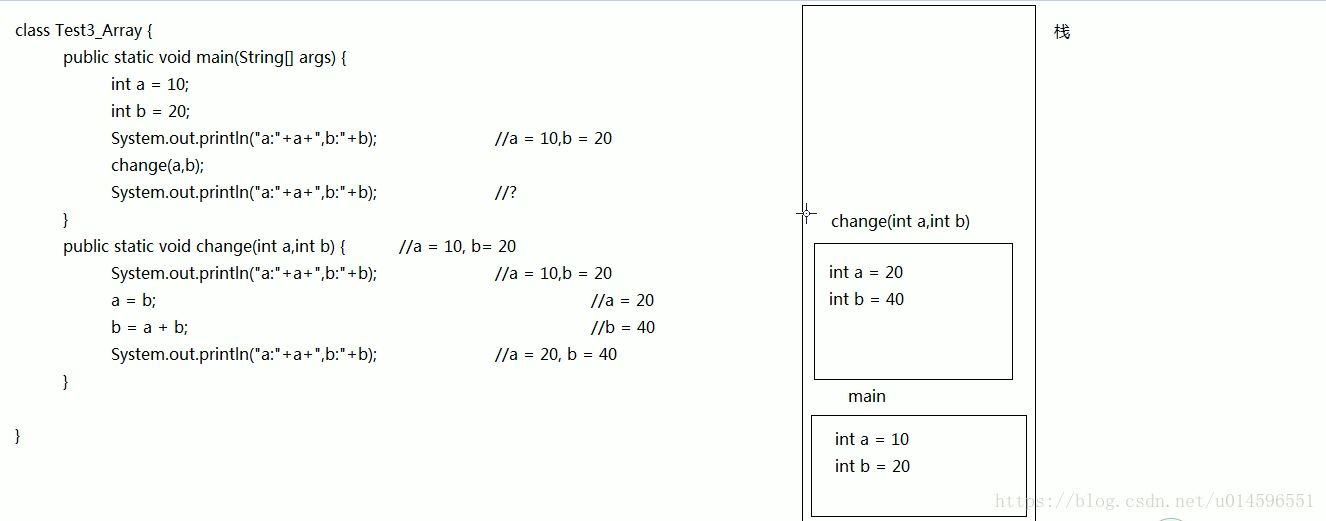
**4.Java中的参数传递问题**

程序一：**基本数据类型的值传递，不改变原值，因为调用后就会弹栈，局部变量随之消失**

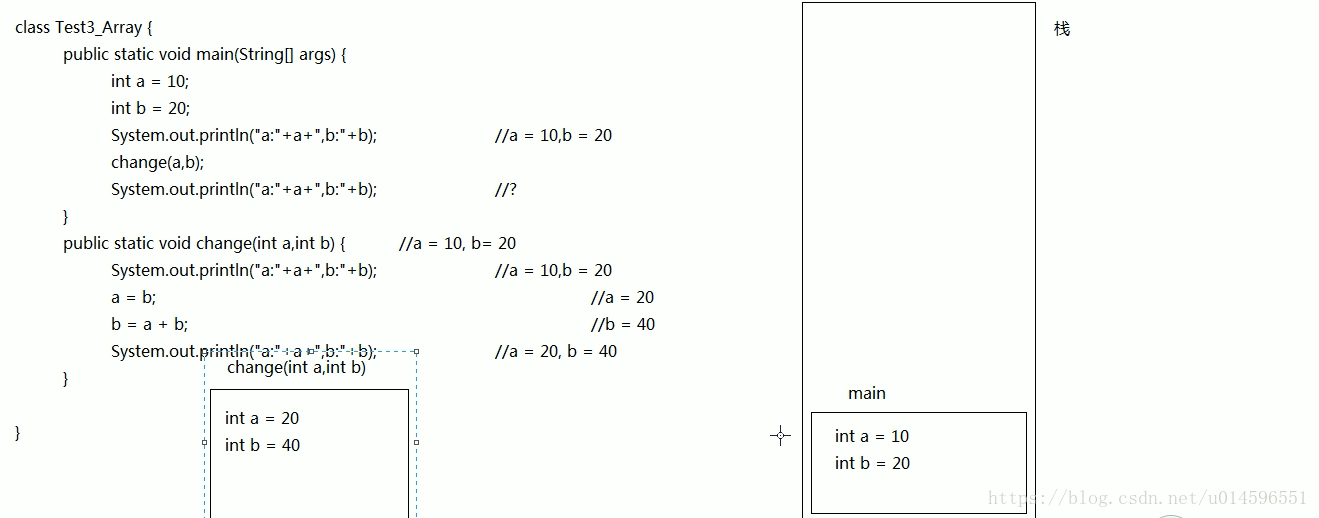
public static void main(String[] args) {  
 int a = 10;  
 int b = 20;  
 System.out.println("a:"+a+",b:"+b);  
 change(a,b);  
 System.out.println("a:"+a+",b:"+b);  
 }  
   
 public static void change(int a,int b) {  
 System.out.println("a:"+a+",b:"+b);  
 a = b;  
 b = a + b;  
 System.out.println("a:"+a+",b:"+b);  
 }

输出结果： 10 20  
 10 20  
 20 40  
 10 20

main方法进栈，随后change方法进栈



change执行完毕后，弹栈

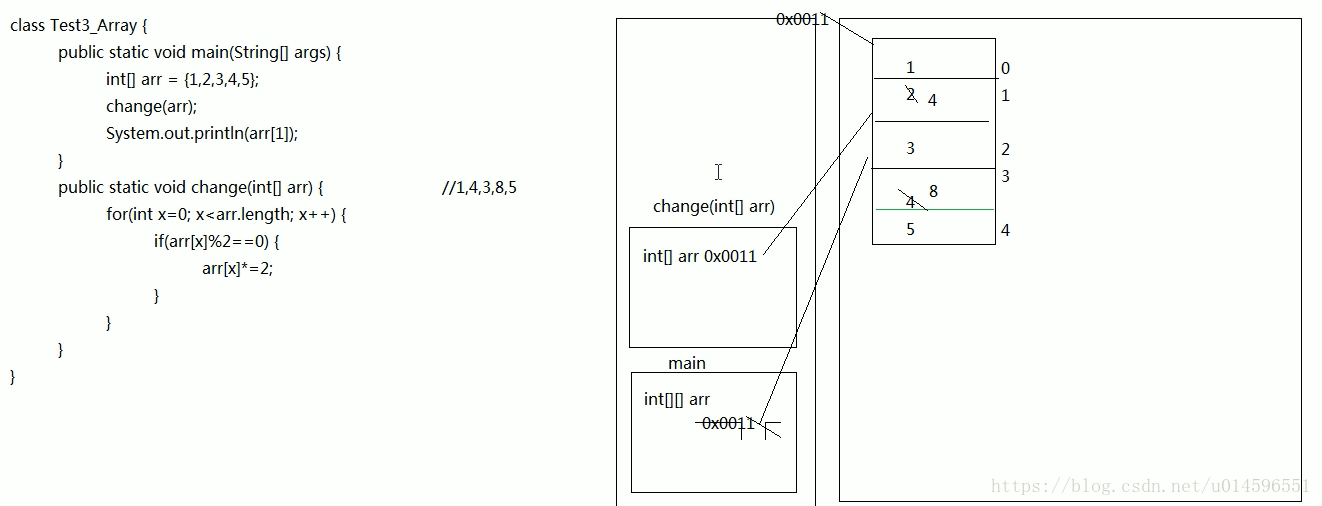


**程序二：引用数据类型的值传递，改变原值，因为即使方法弹栈，但是堆内存数组对象还在，可以通过地址继续访问**

public static void main(String[] args) {  
 int[] arr = {1,2,3,4,5};  
 change(arr);  
 System.out.println(arr[1]);  
}  
public static void change(int[] arr) {  
 for(int x=0; x<arr.length; x++) {  
 if(arr[x]%2==0) {  
 arr[x]\*=2;  
 }  
 }  
}

输出 4

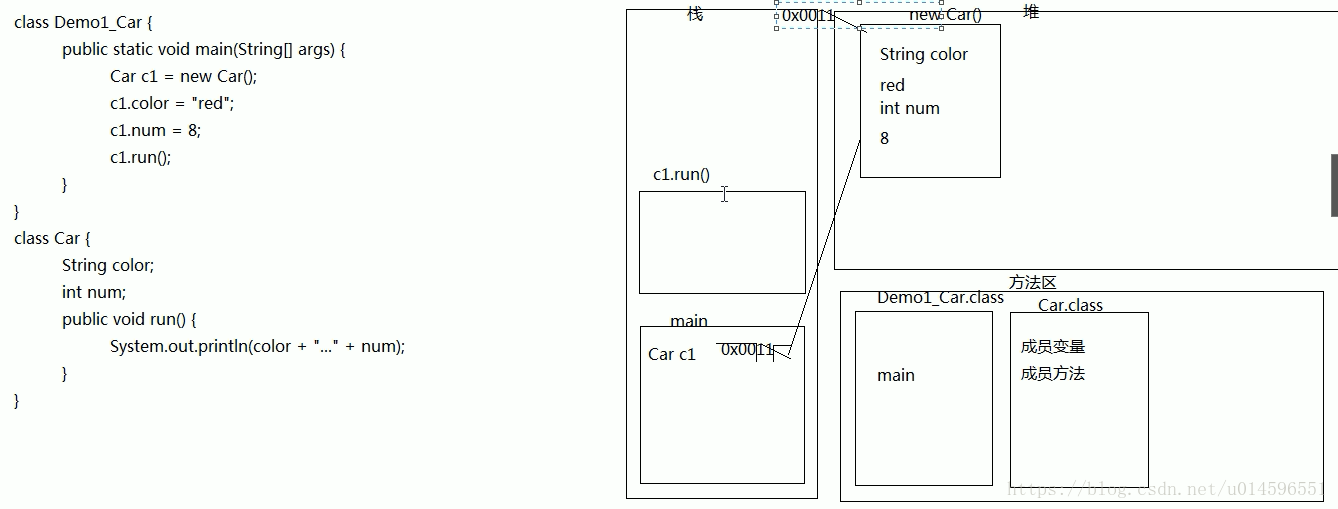
同样也是先main进栈，随后change进栈



最后change出栈，但堆内存区的值不会改变

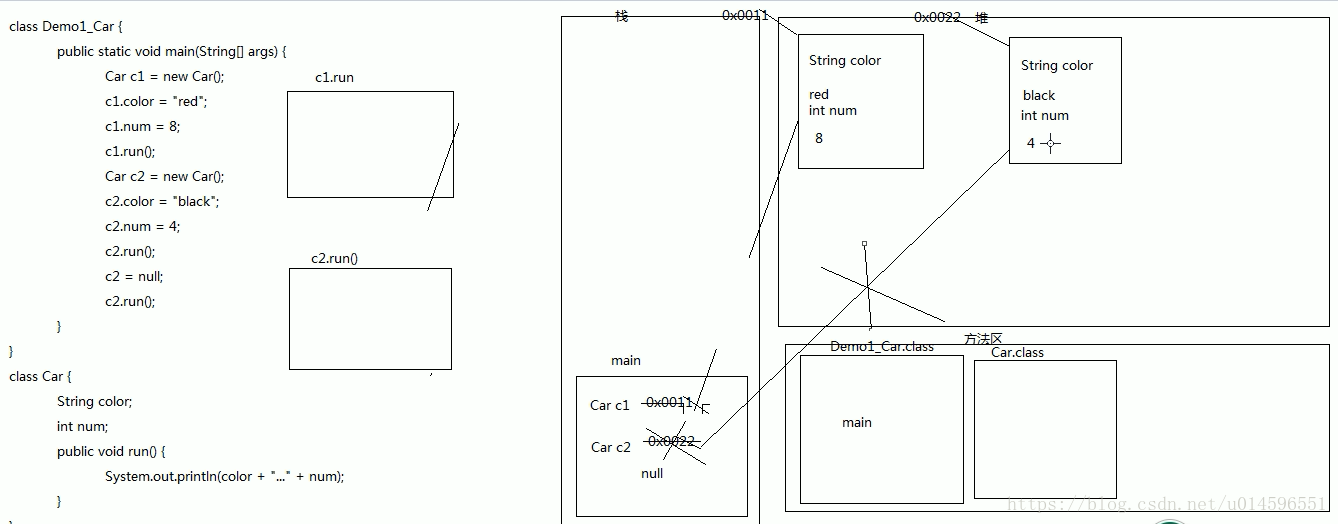
**5.对象**  
**（1）一个对象**

将磁盘上的文件编译成字节码文件，而在运行时字节码文件进入内存的方法区，首先是Demo1\_Car ，虚拟机调用main，main进栈，Car先将Car.class加载入内存，随后通过Car创建对象，在堆里创建对象，并将成员变量赋初值，String的赋值null，int的赋值为0，将对象的地址传入，在通过地址找到对象，对成员变量赋值，调用run方法，run方法进栈，运行完之后弹栈



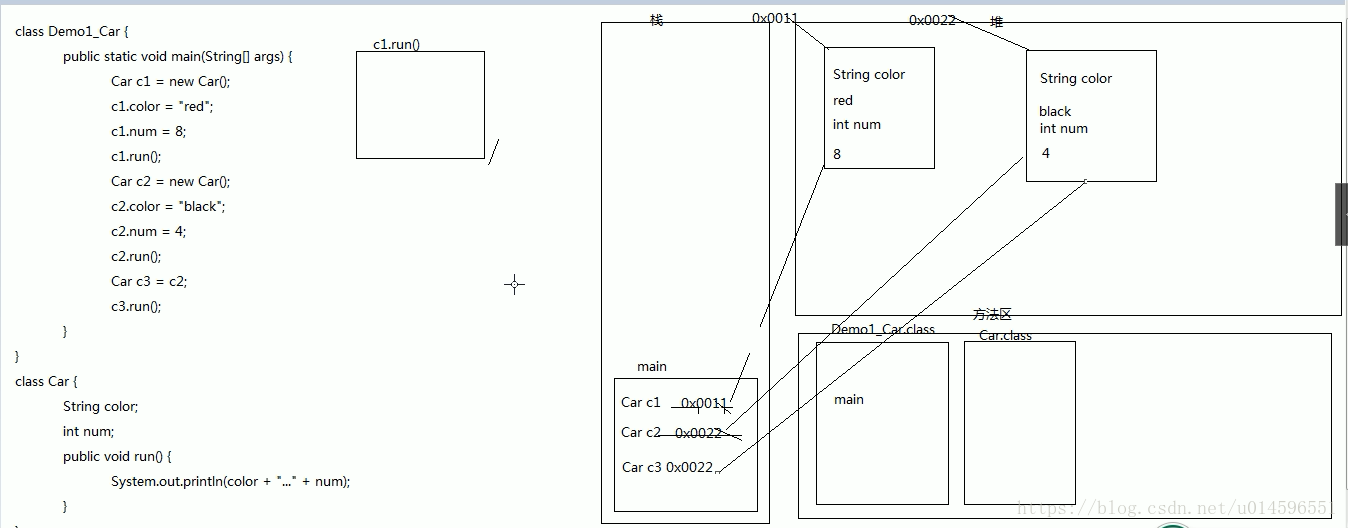
**（2）二个对象**

如果没有任何引用指向该对象，那么该对象就会变成垃圾，java中有完善的垃圾回收机制，会在不定时对其进行回收



**（3）三个引用两个对象**

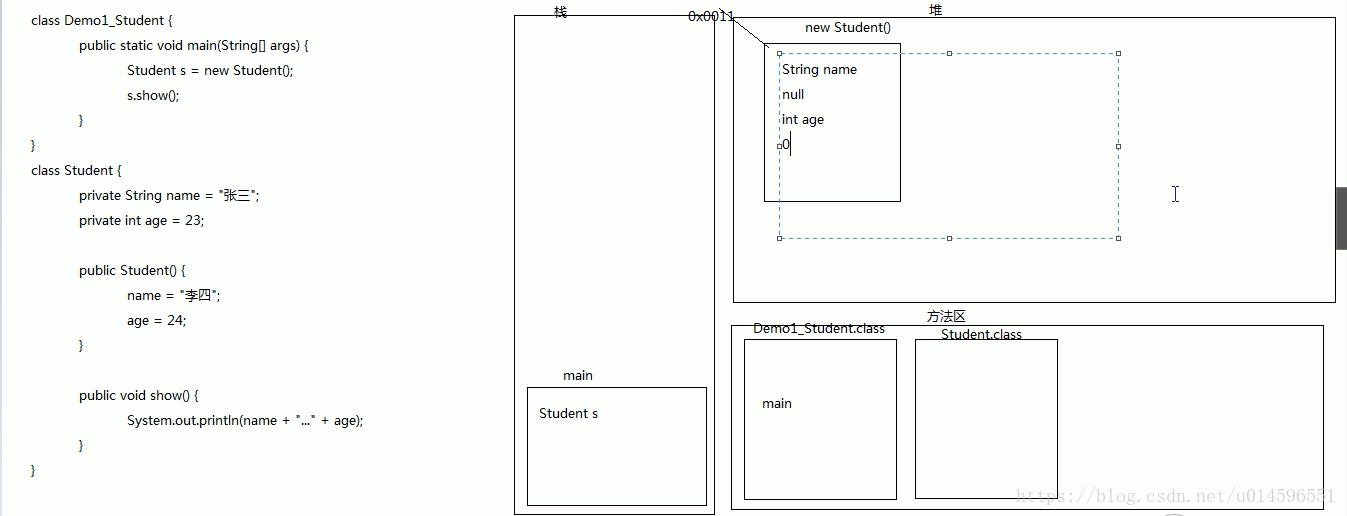
同两个对象类似，将从c2的地址赋值给c3



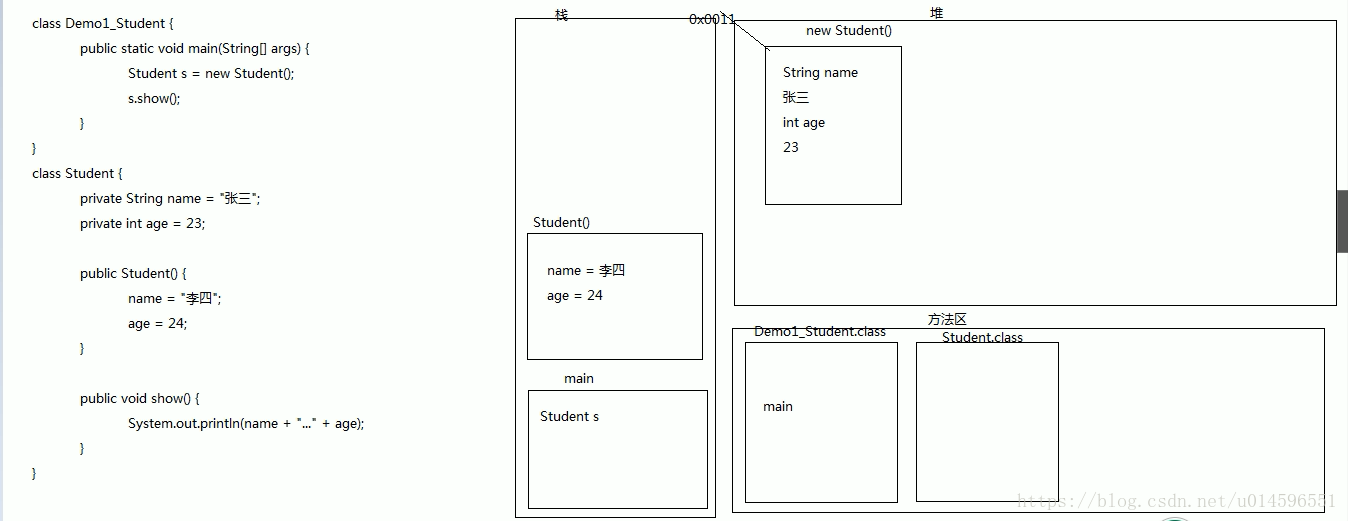
**（4） 创建对象的步骤**

* Student s = new Student();
  + 1,Student.class加载进内存
  + 2,声明一个Student类型引用s
  + 3,在堆内存创建对象,
  + 4,给对象中属性默认初始化值
  + 5,属性进行显示初始化
  + 6,构造方法进栈,对对象中的属性赋值,构造方法弹栈
  + 7,将对象的地址值赋值给s

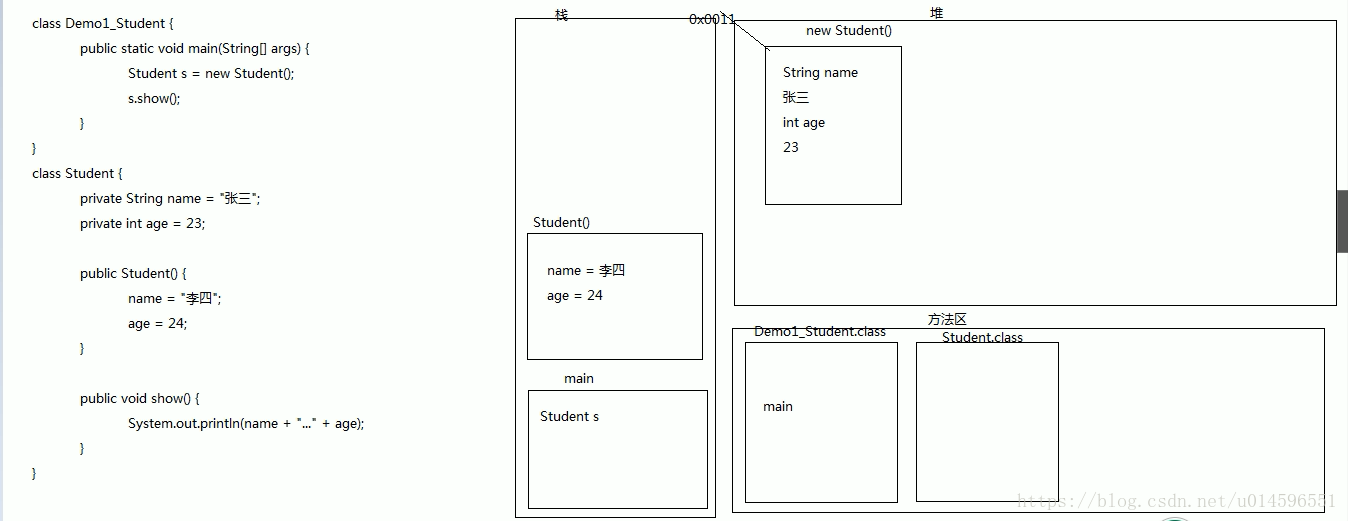
同以前，创建对象后，赋初始值



再根据对属性进行显示初始化，随后虚拟机自动调用构造方法



对象创建完毕，构造方法弹栈



点赞 2  
————————————————  
版权声明：本文为CSDN博主「TISIULS」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/u014596551/article/details/79757932

答案应用二：https://www.cnblogs.com/reformdai/p/10758316.html

Java 中的内存分配

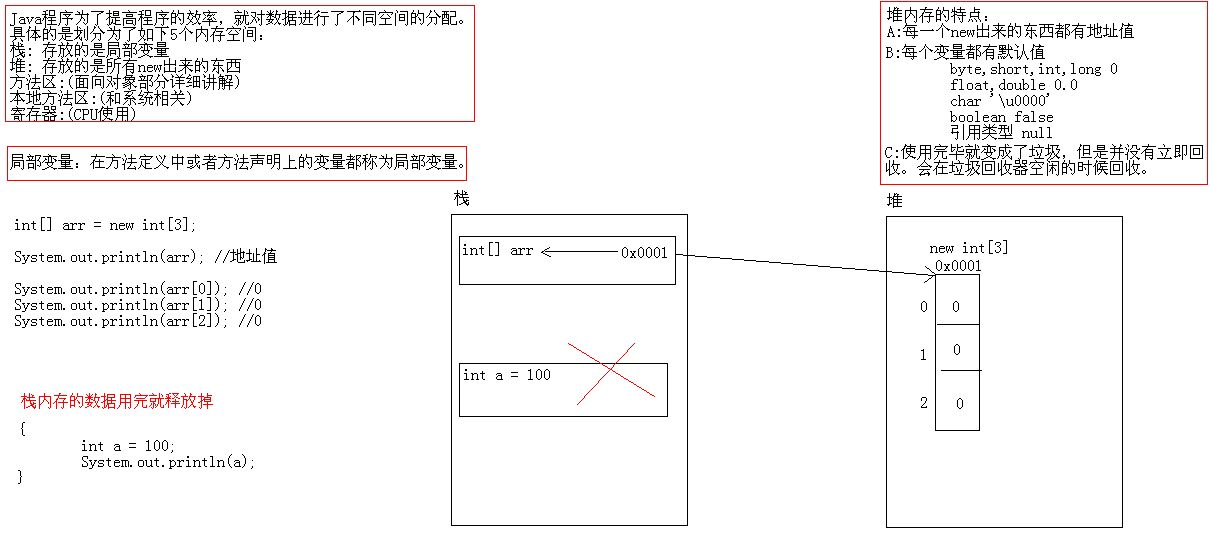
Java 程序运行时，需要在内存中分配空间。为了提高运算效率，就对空间进行了不同区域的划分，因为每一片区域都有特定的处理数据方式和内存管理方式。

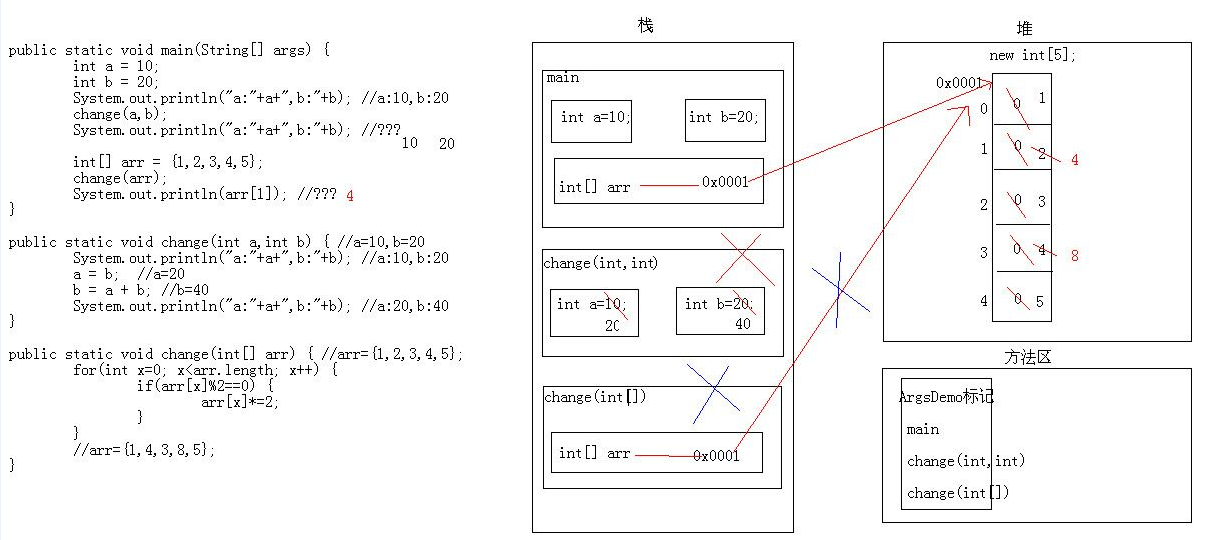
一、栈：储存局部变量

* 局部变量：在方法的定义中或者在方法声明上的变量称为局部变量。
* 特点：栈内存的数据用完就释放。

二、堆：储存 new 出来的东西

* 特点：
  + 每一个 new 出来的东西都有地址值；
  + 每个变量都有默认值 （byte, short, int, long 的默认值为 0；float, double 的默认值为 0.0；char 的默认值为 “\u0000”；boolean 的默认值为 false；引用类型为 null）；
  + 使用完毕就变成垃圾，但是并没有立即回收。会有垃圾回收器空闲的时候回收。

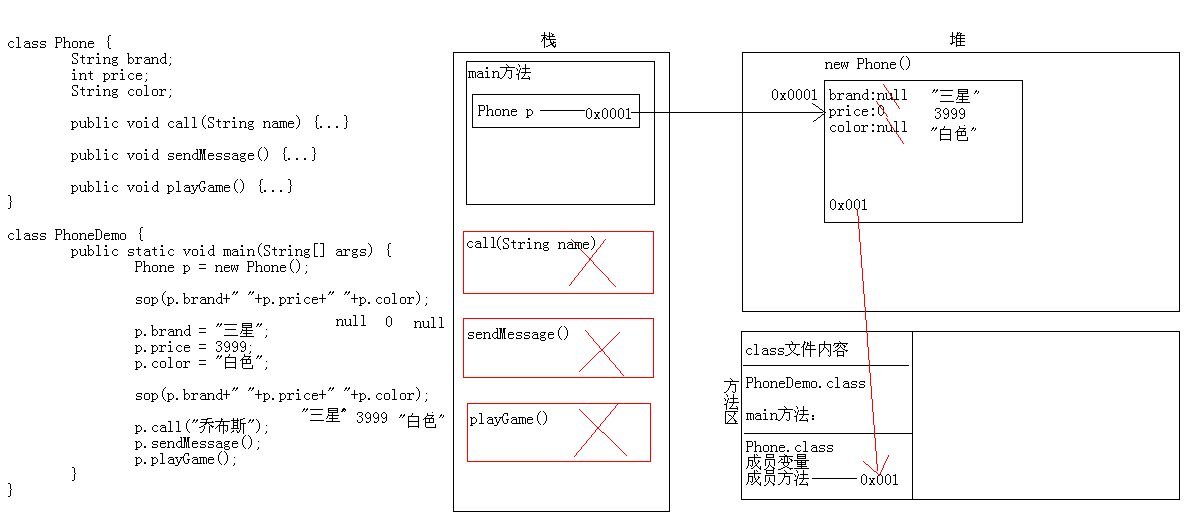




三、方法区：

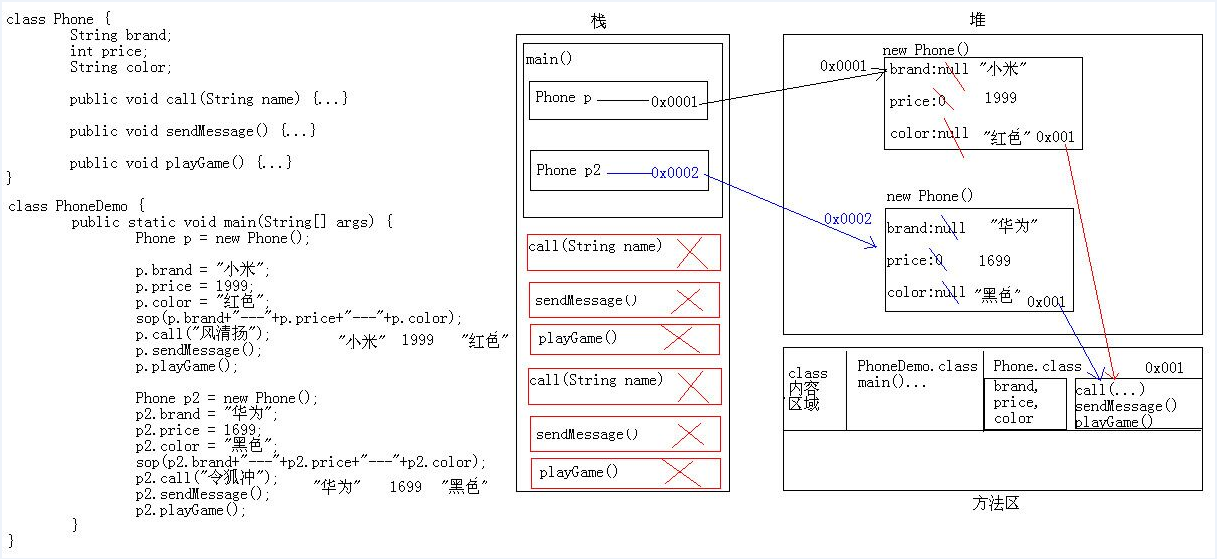
一个对象的运行过程：

1. **程序从 main 方法中进入；运行到 Phone p 时，在栈中开辟了一个空间；**
2. **new Phone() 时，在队中开了一个内存空间，此时会有一个内存值为 0x0001；此时会找到对应的 Phone 的 class 文件，发现有三个变量和三个方法，于是将三个成员变量放在了堆中，但是此时的值为默认值（具体默认值见上）。注意，在方法区里也有一个地址值，假设为 0x001，可以认为在堆中也有一个位置，在堆中的位置，可以找到方法区中相对应的方法；**
3. **继续运行，p.brand = "三星"；将三星赋值给 p.brand，通过栈中的 p 找到了堆中的 brand，此时的 null 值变为“三星”。剩下的类似；**
4. **当运行到 p.call("乔布斯") 时，通过栈中的 p 找到堆中存在的方法区的内存地址，从而指引到方法区中的 Phone.class 中的方法。从而将 call 方法加载到栈内存中，注意：当执行完毕后，call 方法就从栈内存中消失！剩余的如上。**
5. **最后，main 方法消失！**



**两个对象的运行过程：**

1. **程序从 main() 方法进入，运行到 Phone p 时，栈内存中开内存空间；**
2. **new Phone() 时，在队中开了一个内存空间，内存值为 0x0001；此时会找到对应的 Phone 类，发现有三个变量，于是将三个成员变量放在了堆中，但是此时的值为默认值。又发现该类还存在方法，于是将该方法的内存值留在了堆中，在方法区里也有一个地址值，假设为 0x001，这个值与堆中的值相对应；**
3. **程序继续运行，到 p.brand 时，进行了负值，同上；**
4. **当程序运行到 Phone p2 时；到 new Phone() 时，在堆内存中开辟了内存空间 0x0002，赋值给 Phone p2；**
5. **剩下跟一个对象的内存相同。**



**三个对象的运行过程：**

1. **基本流程跟前两个无差别；**
2. **但是当运行到 Phone p3 时，在栈内存中分配了一个空间，然后将 p1 的内存赋值给了 p3，即此时 Phone p3 的内存是指向 0x0001 的；**
3. **继续给变量赋值，会将原来已经赋值的变量给替换掉。**



答案三：Java内存分配全面浅析https://blog.csdn.net/iteye*4537/article/details/82341051?depth*1-utm*source=distribute.pc*relevant.none-task-blog-BlogCommendFromBaidu-1&utm*source=distribute.pc*relevant.none-task-blog-BlogCommendFromBaidu-1

Java内存分配全面浅析

[iteye\_4537](https://me.csdn.net/iteye_4537) 最后发布于2012-06-11 09:46:00 阅读数 101 收藏

本文将由浅入深详细介绍Java内存分配的原理，以帮助新手更轻松的学习Java。这类文章网上有很多，但大多比较零碎。本文从认知过程角度出发，将带给读者一个系统的介绍。

进入正题前首先要知道的是Java程序运行在JVM(Java VirtualMachine，Java虚拟机)上，可以把JVM理解成Java程序和操作系统之间的桥梁，JVM实现了Java的平台无关性，由此可见JVM的重要性。所以在学习Java内存分配原理的时候一定要牢记这一切都是在JVM中进行的，JVM是内存分配原理的基础与前提。

**简单通俗的讲，一个完整的Java程序运行过程会涉及以下内存区域：**

l **寄存器：**JVM内部虚拟寄存器，存取速度非常快，程序不可控制。

l **栈：**保存局部变量的值，包括：1.用来保存基本数据类型的值；2.保存类的**实例**，即堆区**对象**的引用(指针)。也可以用来保存加载方法时的帧。

l **堆：**用来存放动态产生的数据，比如new出来的**对象**。注意创建出来的对象只包含属于各自的成员变量，并不包括成员方法。因为同一个类的对象拥有各自的成员变量，存储在各自的堆中，但是他们共享该类的方法，并不是每创建一个对象就把成员方法复制一次。

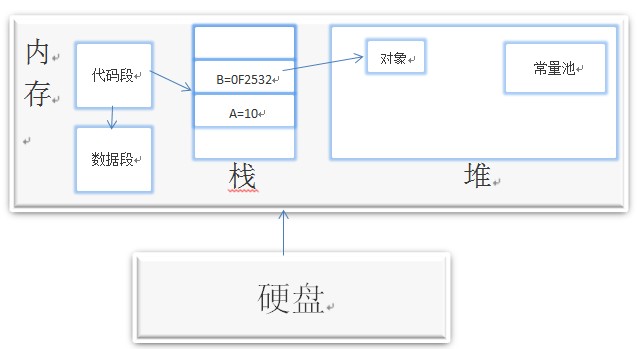
l **常量池：**JVM为每个已加载的类型维护一个常量池，常量池就是这个类型用到的常量的一个有序集合。包括直接常量(基本类型，String)和对其他类型、方法、字段的**符号引用(1)**。池中的数据和数组一样通过索引访问。由于常量池包含了一个类型所有的对其他类型、方法、字段的符号引用，所以常量池在Java的动态链接中起了核心作用。**常量池存在于堆中**。

l **代码段：**用来存放从硬盘上读取的源程序代码。

l **数据段：**用来存放static定义的静态成员。

**下面是内存表示图：**

\*\*  
\*\*



上图中大致描述了Java内存分配，接下来通过实例详细讲解Java程序是如何在内存中运行的（注：以下图片引用自尚学堂马士兵老师的J2SE课件，图右侧是程序代码，左侧是内存分配示意图，我会一一加上注释）。

**预备知识：**

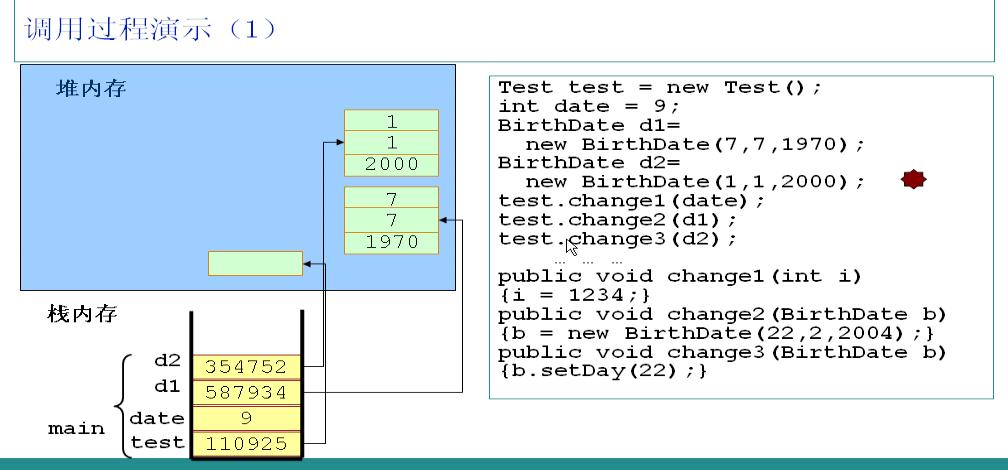
\*\*  
\*\*

**1.**一个Java文件，只要有main入口方法，我们就认为这是一个Java程序，可以单独编译运行。

**2.**无论是普通类型的变量还是引用类型的变量(俗称实例)，都可以作为局部变量，他们都可以出现在栈中。只不过普通类型的变量在栈中直接保存它所对应的值，而引用类型的变量保存的是一个指向堆区的指针，通过这个指针，就可以找到这个实例在堆区对应的对象。因此，普通类型变量只在栈区占用一块内存，而引用类型变量要在栈区和堆区各占一块内存。

**示例：**

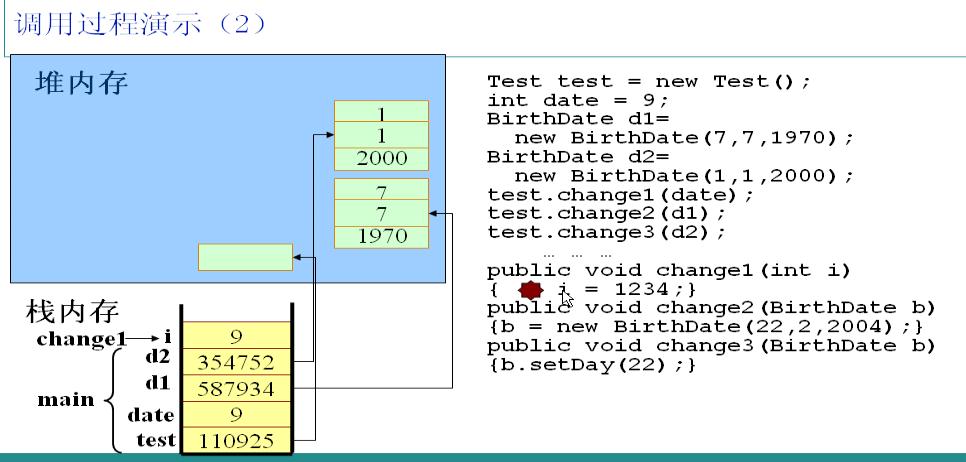
\*\*  
\*\*



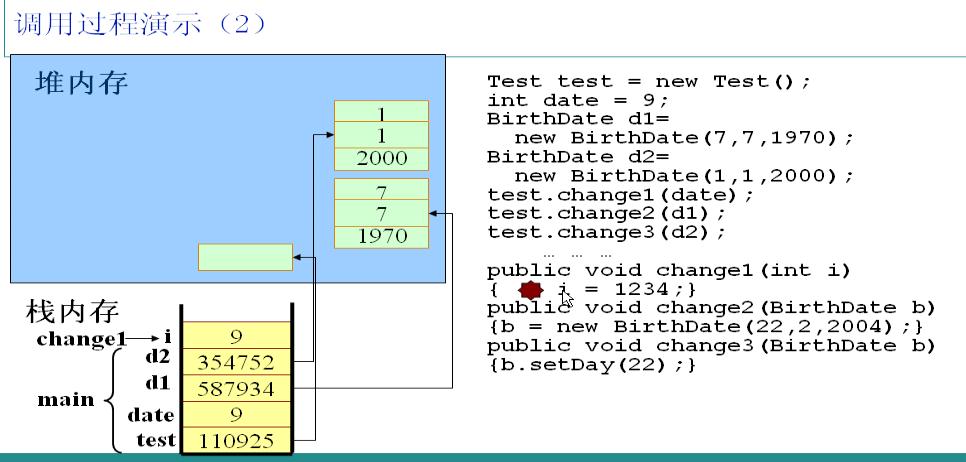
**1.**JVM自动寻找main方法，执行第一句代码，创建一个Test类的实例，在栈中分配一块内存，存放一个指向堆区对象的指针110925。

**2.**创建一个int型的变量date，由于是基本类型，直接在栈中存放date对应的值9。

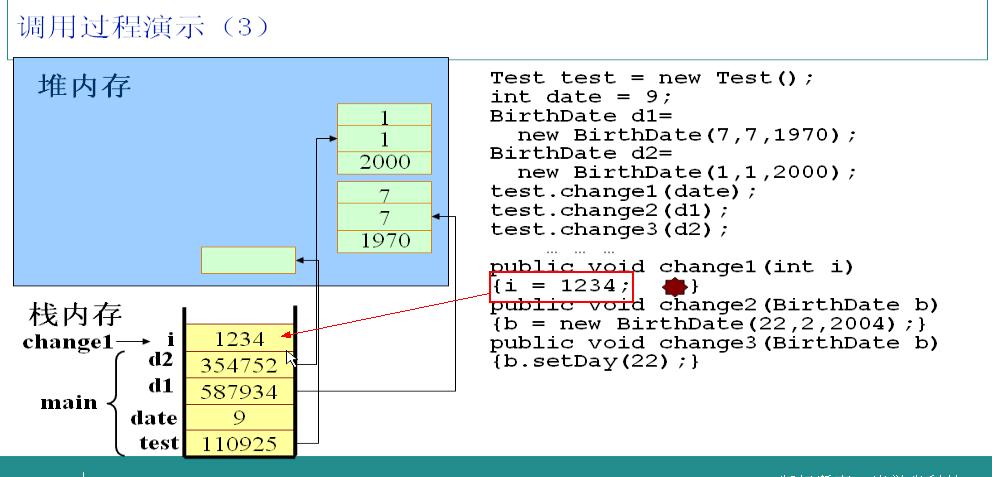
**3.**创建两个BirthDate类的实例d1、d2，在栈中分别存放了对应的指针指向各自的对象。他们在实例化时调用了有参数的构造方法，因此对象中有自定义初始值。



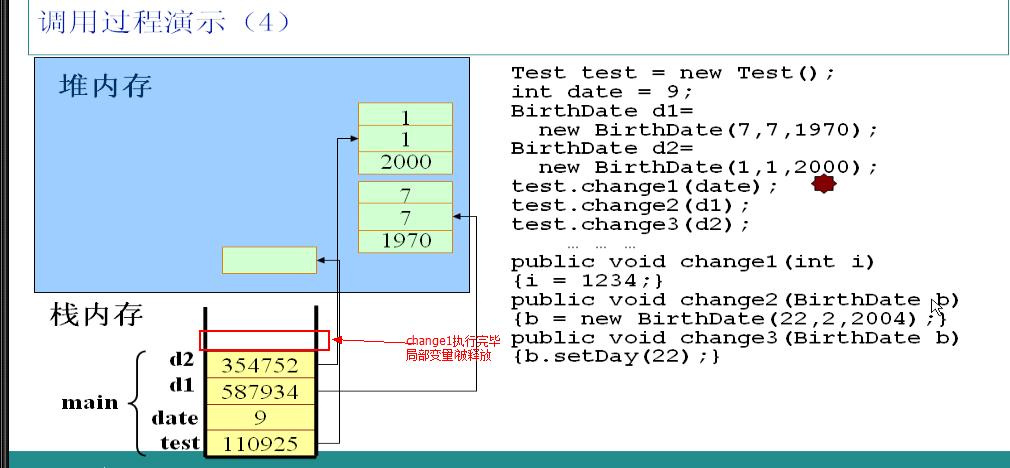
调用test对象的change1方法，并且以date为参数。JVM读到这段代码时，检测到i是局部变量，因此会把i放在栈中，并且把date的值赋给i。



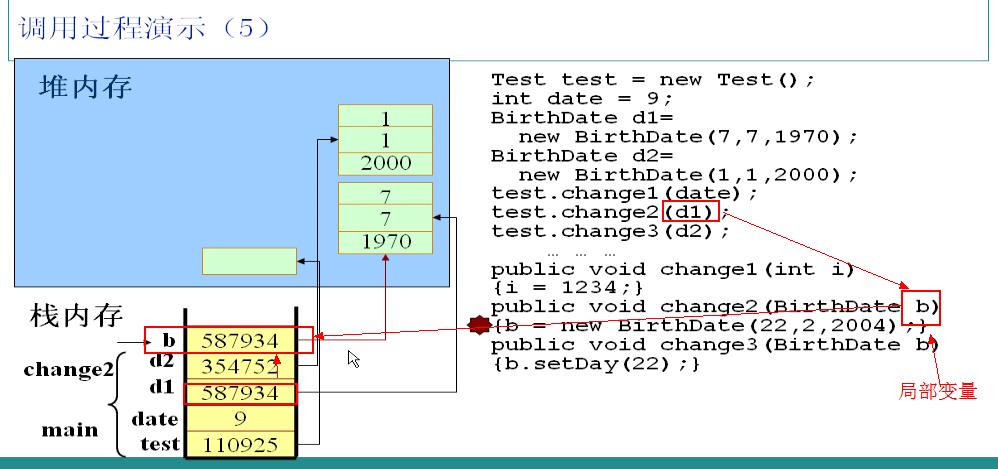
把1234赋给i。很简单的一步。



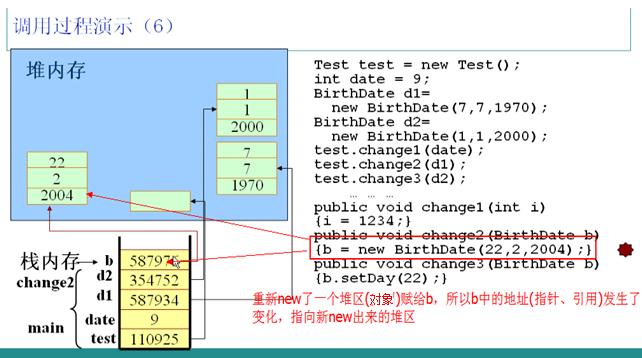
change1方法执行完毕，立即释放局部变量i所占用的栈空间。



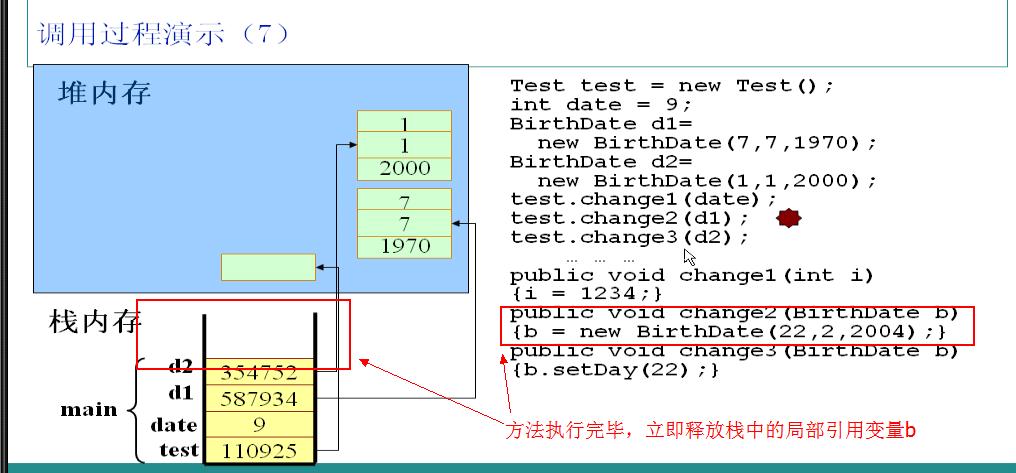
调用test对象的change2方法，以实例d1为参数。JVM检测到change2方法中的b参数为局部变量，立即加入到栈中，由于是引用类型的变量，所以b中保存的是d1中的指针，此时b和d1指向同一个堆中的对象。在b和d1之间传递是指针。



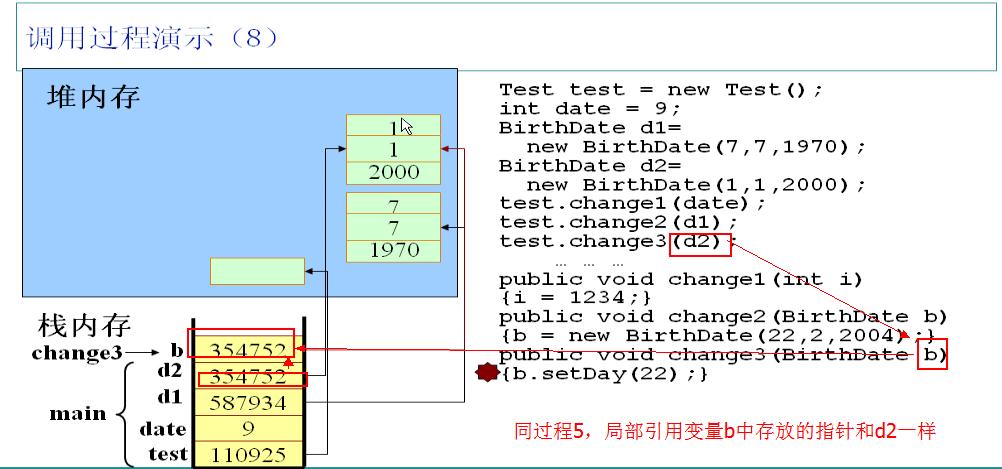
change2方法中又实例化了一个BirthDate对象，并且赋给b。在内部执行过程是：在堆区new了一个对象，并且把该对象的指针保存在栈中的b对应空间，此时实例b不再指向实例d1所指向的对象，但是实例d1所指向的对象并无变化，这样无法对d1造成任何影响。



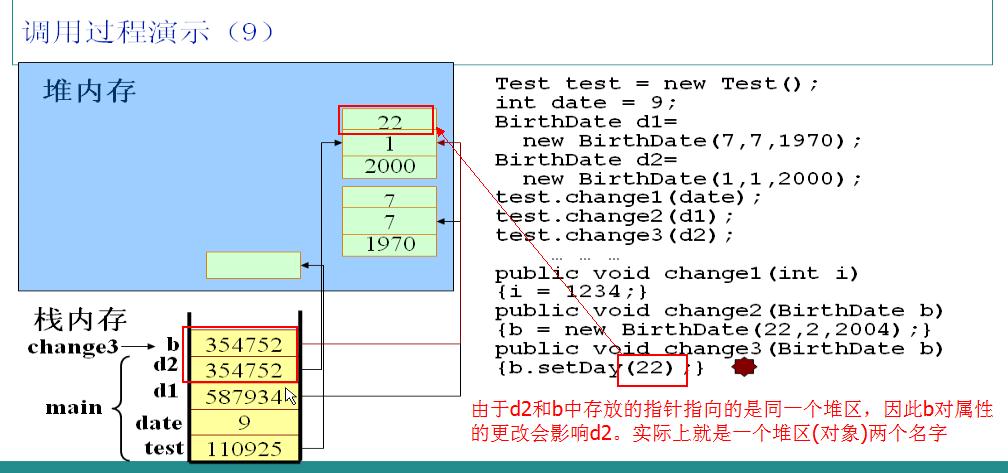
change2方法执行完毕，立即释放局部引用变量b所占的栈空间，注意只是释放了栈空间，堆空间要等待自动回收。



调用test实例的change3方法，以实例d2为参数。同理，JVM会在栈中为局部引用变量b分配空间，并且把d2中的指针存放在b中，此时d2和b指向同一个对象。再调用实例b的setDay方法，其实就是调用d2指向的对象的setDay方法。



调用实例b的setDay方法会影响d2，因为二者指向的是同一个对象。



change3方法执行完毕，立即释放局部引用变量b。

以上就是Java程序运行时内存分配的大致情况。其实也没什么，掌握了思想就很简单了。无非就是两种类型的变量：基本类型和引用类型。二者作为局部变量，都放在栈中，基本类型直接在栈中保存值，引用类型只保存一个指向堆区的指针，真正的对象在堆里。作为参数时基本类型就直接传值，引用类型传指针。

**小结：**

\*\*  
\*\*

**1.**分清什么是实例什么是对象。Class a= new Class();此时a叫实例，而不能说a是对象。实例在栈中，对象在堆中，操作实例实际上是通过实例的指针间接操作对象。多个实例可以指向同一个对象。

**2.**栈中的数据和堆中的数据销毁并不是同步的。方法一旦结束，栈中的局部变量立即销毁，但是堆中对象不一定销毁。因为可能有其他变量也指向了这个对象，直到栈中没有变量指向堆中的对象时，它才销毁，而且还不是马上销毁，要等垃圾回收扫描时才可以被销毁。

**3.**以上的栈、堆、代码段、数据段等等都是相对于应用程序而言的。每一个应用程序都对应唯一的一个JVM实例，每一个JVM实例都有自己的内存区域，互不影响。并且这些内存区域是所有线程共享的。这里提到的栈和堆都是整体上的概念，这些堆栈还可以细分。

**4.**类的成员变量在不同对象中各不相同，都有自己的存储空间(成员变量在堆中的对象中)。而类的方法却是该类的所有对象共享的，只有一套，对象使用方法的时候方法才被压入栈，方法不使用则不占用内存。

以上分析只涉及了栈和堆，还有一个非常重要的内存区域：常量池，这个地方往往出现一些莫名其妙的问题。常量池是干嘛的上边已经说明了，也没必要理解多么深刻，只要记住它维护了一个已加载类的常量就可以了。接下来结合一些例子说明常量池的特性。

**预备知识：**

\*\*  
\*\*

基本类型和基本类型的包装类。基本类型有：byte、short、char、int、long、boolean。基本类型的包装类分别是：Byte、Short、Character、Integer、Long、Boolean。注意区分大小写。二者的区别是：基本类型体现在程序中是普通变量，基本类型的包装类是类，体现在程序中是引用变量。因此二者在内存中的存储位置不同：基本类型存储在栈中，而基本类型包装类存储在堆中。上边提到的这些包装类都实现了常量池技术，另外两种浮点数类型的包装类则没有实现。另外，String类型也实现了常量池技术。

***\*实例：\***\*

public class test { public static void main(String[] args) { objPoolTest(); } public static void objPoolTest() { int i = 40; int i0 = 40; Integer i1 = 40; Integer i2 = 40; Integer i3 = 0; Integer i4 = new Integer(40); Integer i5 = new Integer(40); Integer i6 = new Integer(0); Double d1=1.0; Double d2=1.0; System.out.println("i=i0\t" + (i == i0)); System.out.println("i1=i2\t" + (i1 == i2)); System.out.println("i1=i2+i3\t" + (i1 == i2 + i3)); System.out.println("i4=i5\t" + (i4 == i5)); System.out.println("i4=i5+i6\t" + (i4 == i5 + i6)); System.out.println("d1=d2\t" + (d1==d2)); System.out.println(); }}

***\*结果：\***\*

i=i0 true  
i1=i2 true  
i1=i2+i3 true  
i4=i5 false  
i4=i5+i6 true  
d1=d2 false

***\*结果\*\*\*\*\*\*分析\*\*\*\*\*\*：\***\*

***\***  
**\***\*

***\*1.\***\*i和i0均是普通类型(int)的变量，所以数据直接存储在栈中，而栈有一个很重要的特性：**栈中的数据可以共享**。当我们定义了int i = 40;，再定义int i0 = 40;这时候会自动检查栈中是否有40这个数据，如果有，i0会直接指向i的40，不会再添加一个新的40。

***\*2.\***\*i1和i2均是引用类型，在栈中存储指针，因为Integer是包装类。由于Integer 包装类实现了常量池技术，因此i1、i2的40均是从常量池中获取的，均指向同一个地址，因此i1=12。

***\*3.\****很明显这是一个加法运算，***\*Java\**\**的数学运算都是在栈中进行的\*****\*，***\*Java\*\*\*\*会自动对\*\*\*\*i1\*\*\*\*、\*\*\*\*i2\*\*\*\*进行拆箱操作转化成整型\*\*\*\*，因此i1在数值上等于i2+i3。

***\*4.i\***\*4和i5 均是引用类型，在栈中存储指针，因为Integer是包装类。但是由于他们各自都是new出来的，因此不再从常量池寻找数据，而是从堆中各自new一个对象，然后各自保存指向对象的指针，所以i4和i5不相等，因为他们所存指针不同，所指向对象不同。

***\*5.\***\*这也是一个加法运算，和3同理。

***\*6.\***\*d1和d2均是引用类型，在栈中存储指针，因为Double是包装类。但Double包装类没有实现常量池技术，因此Doubled1=1.0;相当于Double d1=new Double(1.0);，是从堆new一个对象，d2同理。因此d1和d2存放的指针不同，指向的对象不同，所以不相等。

***\*小结：\***\*

***\***  
**\***\*

***\*1.\***\*以上提到的几种基本类型包装类均实现了常量池技术，但他们维护的常量仅仅是【-128至127】这个范围内的常量，如果常量值超过这个范围，就会从堆中创建对象，不再从常量池中取。比如，把上边例子改成Integer i1 = 400; Integer i2 = 400;，很明显超过了127，无法从常量池获取常量，就要从堆中new新的Integer对象，这时i1和i2就不相等了。

***\*2.\***\*String类型也实现了常量池技术，但是稍微有点不同。String型是先检测常量池中有没有对应字符串，如果有，则取出来；如果没有，则把当前的添加进去。

凡是涉及内存原理，一般都是博大精深的领域，切勿听信一家之言，多读些文章。我在这只是浅析，里边还有很多猫腻，就留给读者探索思考了。希望本文能对大家有所帮助！

***\*脚注：\***\*

***\***  
**\***\*

**(1)** 符号引用，顾名思义，就是一个符号，符号引用被使用的时候，才会解析这个符号。如果熟悉linux或unix系统的，可以把这个符号引用看作一个文件的软链接，当使用这个软连接的时候，才会真正解析它，展开它找到实际的文件

对于符号引用，在类加载层面上讨论比较多，源码级别只是一个形式上的讨论。

当一个类被加载时，该类所用到的别的类的符号引用都会保存在常量池，实际代码执行的时候，首次遇到某个别的类时，JVM会对常量池的该类的符号引用展开，转为直接引用，这样下次再遇到同样的类型时，JVM就不再解析，而直接使用这个已经被解析过的直接引用。

除了上述的类加载过程的符号引用说法，对于源码级别来说，就是依照引用的解析过程来区别代码中某些数据属于符号引用还是直接引用，如，System.out.println("test" +"abc");//这里发生的效果相当于直接引用，而假设某个Strings = "abc"; System.out.println("test" + s);//这里的发生的效果相当于符号引用，即把s展开解析，也就相当于s是"abc"的一个符号链接，也就是说在编译的时候，class文件并没有直接展看s，而把这个s看作一个符号，在实际的代码执行时，才会展开这个。

***\*参考文章：\***\*

java内存分配研究：http://www.blogjava.net/Jack2007/archive/2008/05/21/202018.html

Java常量池详解之一道比较蛋疼的面试题：http://www.cnblogs.com/DreamSea/archive/2011/11/20/2256396.html

jvm常量池：http://www.cnblogs.com/wenfeng762/archive/2011/08/14/2137820.html

深入Java核心 Java内存分配原理精讲：http://developer.51cto.com/art/201009/225071.htm

四、本地方法区（和系统相关）

五、寄存器（给 CPU 使用）

答案4--https://www.jianshu.com/p/ef0c80d1782c

**Java内存分配总结**

[](https://www.jianshu.com/u/347890c1ed1e)

[奔跑吧李博](https://www.jianshu.com/u/347890c1ed1e)关注

12018.12.29 10:50:01字数 1,601阅读 3,964

##### Java 的内存管理就是对象的分配和释放的处理

1.分配：通过关键字new创建对象分配内存空间，对象存在堆中。  
2.释放 ：对象的释放是由垃圾回收机制决定和执行的，开发人员可以将经历集中在业务的开发上

##### Java内存泄漏：

当对象存在内存的引用，却不会再继续使用，对象会占用内存无法被GC回收，这些对象就会判定为内存泄漏。

##### Java内存区域划分：

1.栈：  
在函数中定义的基本类型变量和对象的引用变量都在函数的栈内存中分配。栈数据可以共享。但缺点是，存在栈中的数据大小与生存期必须是确定的，缺乏灵活性。

2.堆：  
通过new生成的对象都存放在堆中，对于堆中的对象生命周期的管理由Java虚拟机的垃圾回收机制GC进行回收和统一管理。优点是可以动态分配内存大小，缺点是由于动态分配内存导致存取速度慢。

3.方法区：  
是各个线程共享的内存区域，它用于存储class二进制文件，包含了虚拟机加载的类信息、常量(常量池)、静态变量(静态域)、即时编译后的代码等数据。

1.常量池：  
常量池在编译期间就将一部分数据存放于该区域，包含以final修饰的基本数据类型的常量值、String字符串。

2.静态域：  
存放类中以static声明的静态成员变量。

3.程序计数器  
当前线程所执行的行号指示器。通过改变计数器的值来确定下一条指令，比如循环，分支，跳转，异常处理，线程恢复等都是依赖计数器来完成。

##### 堆和栈在内存中的区别是什么

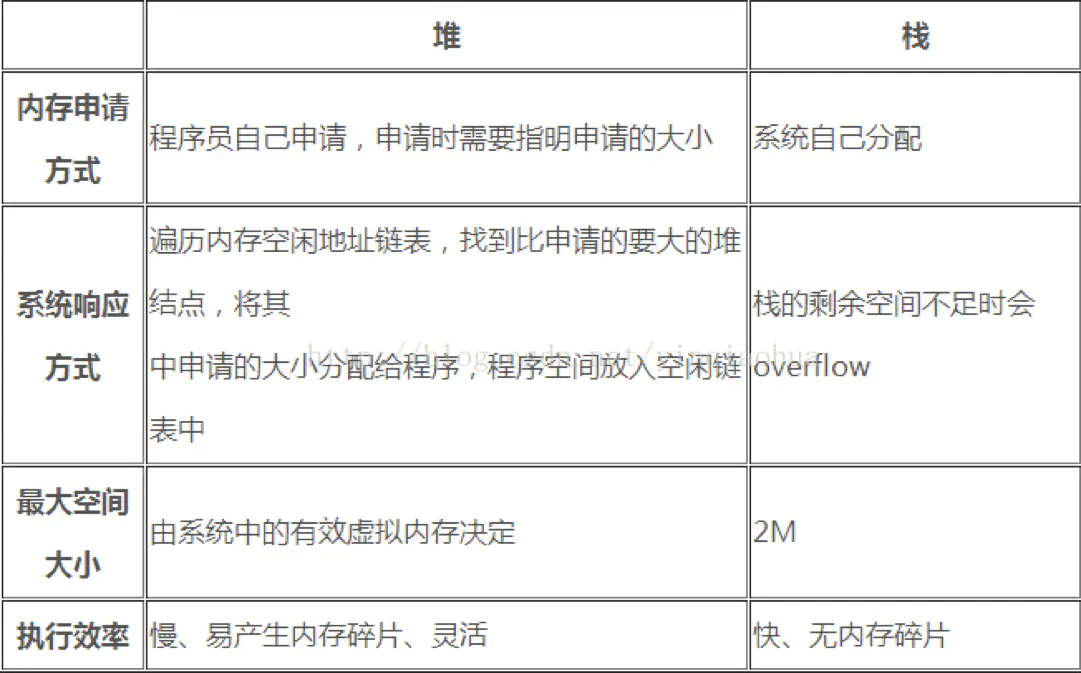


image.png

各种数据在内存中的存储方式：

**1.基本数据类型**

众所周知，基本数据类型有8种，它们存储于栈中。  
int num = 5,这里的 num 是一个指向 int 类型的引用，指向5这个字面值。这些字面值的数据，由于大小可知，生存期可知，出于追求速度的原因，就存在于栈中。

栈中的数据共享：已存在的值不会再次创建

int a = 3;  
int b = 3;

编译器先处理 int a = 3；首先它会在栈中创建一个变量为 a 的引用，然后查找有没有字面值为3的地址，没找到，就开辟一个存放3这个字面值的地址，然后将 a 指向3的地址。接着处理 int b = 3；在创建完 b 这个引用变量后，由于在栈中已经有3这个字面值，便将 b 直接指向3的地址。这样，就出现了 a 与 b 同时均指向3的情况。

2.对象

举一个万年不变的Person类：

public class Person {  
 private String name;  
 private int age;  
  
 public Person(String name, int age) {  
 this.name = name;  
 this.age = age;  
 }  
  
 public String getName() {  
 return name;  
 }  
  
 public void setName(String name) {  
 this.name = name;  
 }  
  
 public int getAge() {  
 return age;  
 }  
  
 public void setAge(int age) {  
 this.age = age;  
 }  
}

一个类通过使用new运算符可以创建多个不同的对象实例，这些对象实例将在堆中被分配不同的内存空间，改变其中一个对象的状态不会影响其他对象的状态。

Person one = new Person("小明",15);  
 Person two = new Person("小王",17);

在堆内存中只创建了一个对象实例，在栈内存中创建了两个对象引用，两个对象引用同时指向一个对象实例。

Person one = new Person("小明",15);  
 Person two = one;

3.包装类  
基本类型的定义都是直接在栈中，如果用包装类来创建对象，就和普通对象一样了。  
比如int a = 5,a存储在栈中。而Integer i = new Integer(5)，i 对象数据存储在堆中，i 的引用存储在栈中。

4.数组  
数组是一种引用类型，数组用来存储同一种数据类型的数据，一旦初始化完成，即所占的空间就已固定下来，即使某个元素被清空，但其所在空间仍然保留，因此数组长度将不能被改变。

举例：执行int[] array = new int[5]时，首先会在栈中创建引用变量，在堆中开辟5个int型数据的空间，该引用变量存放数组首地址，即实现数组名来引用数组。

5.静态变量  
static 的修饰的变量和方法会存在静态域，类变量会共同使用同一块变量地址。即改变该静态变量，该所有类对象所使用的变量值都会随之改变。

##### 对象==和基本数据==

"=="符号判断的内存地址是否是同一块。

int a=5;  
int b=5;  
System.out.println(a==b);//true  
  
String str1 = "hello";  
String str2 = "he" + new String("llo");  
System.err.println(str1 == str2); //false

因为a在栈中创建了存放值为5的地址，创建b的时候，由于数据共享，已经有5的值就不会再创建，将b指向5，所有是同样的地址。但是String拼接总是会创建新的对象，是在常量池中创建的是不同的地址。

##### 内存分配综合情况分析：

public class Dog {  
Collar c;  
String name;  
//1. main()方法位于栈上  
public static void main(String[] args) {  
//2. 在栈上创建引用变量d,但Dog对象尚未存在  
Dog d;  
//3. 创建新的Dog对象，并将其赋予d引用变量  
d = new Dog();  
//4. 将引用变量的一个副本传递给go()方法  
d.go(d);  
}  
//5. 将go()方法置于栈上，并将dog参数作为局部变量  
void go(Dog dog){  
//6. 在堆上创建新的Collar对象，并将其赋予Dog的实例变量  
c =new Collar();  
}  
//7.将setName()添加到栈上，并将dogName参数作为其局部变量  
void setName(String dogName){  
//8. name的实例对象也引用String对象  
name=dogName;  
}  
//9. 程序执行完成后，setName()将会完成并从栈中清除，此时，局部变量dogName也会消失，尽管它所引用的String仍在堆上  
}

##### Java中==和equals的区别，equals和hashCode的区别

Java中==和equals的区别，equals和hashCode的区别  
 ==用于基本数据类型用比较，比较的是值是否相等  
 ==用于对象，比较的是在内存中的地址是否相等  
 Equals表示引用所指内容是否相等。

##### Android为每个应用程序分配的默认内存大小是多少

16M,可以申请使用更大内存

##### String s1=”ab”,String s2=”a”+”b”,String s3=”a”,String s4=”b”,s5=s3+s4请问s5==s2返回什么?

返回false.在编译过程中,编译器会将s2直接优化为”ab”,会将其放置在常量池当中,s5则是被创建在堆区,相当于s5=new String(“ab”);内存地址不一样

作者：奔跑吧李博  
链接：https://www.jianshu.com/p/ef0c80d1782c  
来源：简书  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

### 4.GC 是什么? 为什么要有 GC？

答案：https://blog.csdn.net/hustwht/article/details/52109343

GC是什么？为什么要有GC？

转载[千里码万里行](https://me.csdn.net/hustwht) 最后发布于2016-08-04 01:24:33 阅读数 15062 收藏

GC是**垃圾收集**的意思，***内存处理***是编程人员容易出现问题的地方，

忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java提供的GC功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，

Java语言没有提供释放已分配内存的显示操作方法。Java程序员不用担心内存管理，因为垃圾收集器会自动进行管理。要请求垃圾收集，可以调用下面的方法之一：System.gc()或Runtime.getRuntime().gc()，但JVM可以屏蔽掉显示的垃圾回收调用。

垃圾回收可以有效的防止内存泄露，有效的使用可以使用的内存。垃圾回收器通常是作为一个单独的低优先级的线程运行，不可预知的情况下对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收，程序员不能实时的调用垃圾回收器对某个对象或所有对象进行垃圾回收。

在Java诞生初期，**垃圾回收**是Java最大的亮点之一，因为*服务器端的编程需要有效的防止内存泄露问题*，然而时过境迁，如今Java的垃圾回收机制已经成为被诟病的东西。移动智能终端用户通常觉得iOS的系统比Android系统有更好的用户体验，其中一个深层次的原因就在于**Android系统中垃圾回收的不可预知性**。

采用“**分代式垃圾收集**”。这种方法会跟Java对象的生命周期将堆内存划分为不同的区域，在垃圾收集过程中，可能会将对象移动到不同区域：   
- 伊甸园（Eden）：这是对象最初诞生的区域，并且对大多数对象来说，这里是它们唯一存在过的区域。   
- 幸存者乐园（Survivor）：从伊甸园幸存下来的对象会被挪到这里。   
- 终身颐养园（Tenured）：这是足够老的幸存对象的归宿。年轻代收集（Minor-GC）过程是不会触及这个地方的。当年轻代收集不能把对象放进终身颐养园时，就会触发一次完全收集（Major-GC），这里可能还会牵扯到压缩，以便为大对象腾出足够的空间。

与垃圾回收相关的JVM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

**补充：**

Java是由C++发展来的。

它摈弃了C++中一些繁琐容易出错的东西。其中有一条就是这个GC。

写C/C++程序，程序员定义了一个变量，就是在内存中开辟了一段相应的空间来存值。内存再大也是有限的，所以当程序不再需要使用某个变量的时候，就需要释放这个内存空间资源，好让别的变量来用它。在C/C++中，释放无用变量内存空间的事情要由程序员自己来解决。就是说当程序员认为变量没用了，就应当写一条代码，释放它占用的内存。这样才能最大程度地避免内存泄露和资源浪费。

但是这样显然是非常繁琐的。程序比较大，变量多的时候往往程序员就忘记释放内存或者在不该释放的时候释放内存了。而且释放内存这种事情，从开发角度说，不应当是程序员所应当关注的。程序员所要做的应该是实现所需要的程序功能，而不是耗费大量精力在内存的分配释放上。

Java有了GC，就不需要程序员去人工释放内存空间。当Java虚拟机发觉内存资源紧张的时候，就会自动地去清理无用变量所占用的内存空间。当然，如果需要，程序员可以在Java程序中显式地使用System.gc()来强制进行一次立即的内存清理。

因为显式声明是做堆内存全扫描，也就是 Full GC，是需要停止所有的活动的（Stop The World Collection），你的应用能承受这个吗？而其显示调用System.gc()只是给虚拟机一个建议，不一定会执行，因为System.gc()在一个优先级很低的线程中执行。

### 5.简述 Java 垃圾回收机制

java基础:简述垃圾回收机制  
原创君尔 最后发布于2019-04-08 13:37:10 阅读数 4095 收藏

1、什么是“垃圾回收”机制

2、垃圾回收机制的特点

3.对象在内存中的状态？

4.强制垃圾回收

**1.什么是“垃圾回收”机制？**

当程序创建对象，数组等引用类型实体时，系统会在堆内存中为之分配一块内存区，对象就保存在内存区中，当内存不再被任何引用变量引用时，这块内存就变成了垃圾，等待垃圾回收机制去进行回收。

**2.垃圾回收机制的特点:**

垃圾回收机制只负责回收**堆内存**中的对象，不会回收任何物理资源（网络io等）

程序无法精准控制垃圾回收的运行，垃圾回收在合适的时候进行，当对象永久性失去了引用后。系统会在合适的时候回收它所占的内存

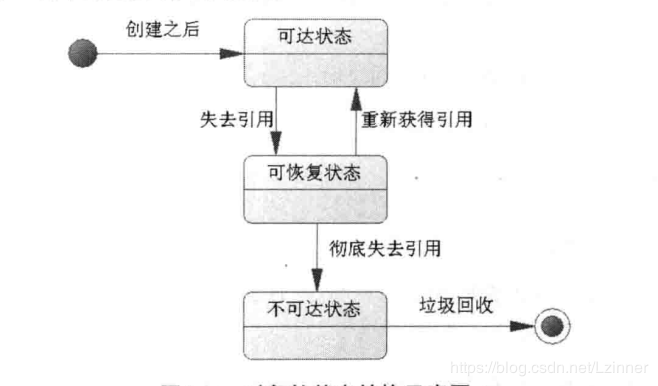
在垃圾回收机制回收任何对象之前，总会**调用它的finalize（）方法**，该方法可能使该对象重新复活（使用一个引用变量重新去引用该对象），从而导致垃圾回收机制取消回收

**3.对象在内存中的状态？**

可达状态:当一个对象创建后，如果有一个以上的引用变量去引用它，则处于可达状态

**可恢复状态**:如果程序中的某个对象不再有任何引用变量去引用它，它就进入了可恢复状态，在这种状态下，系统的垃圾回收机制准备回收该对象所占用的内存，在回收该对象前，系统会调用它的finalize（）方法实现资源清理，如果这时有引用变量进行引用该对象，则该对象会再次变为可达状态，否则会变为不可达状态

**不可达状态**:当对象和所有引用变量的关联都被切断，且系统已经调用所有对象的finalize(）后，对象依然没有达到可达状态，则该对象永久性失去引用，最后达到不可达状态，系统回收该对象所占有的所有资源



**4.强制垃圾回收**

当对象失去引用后，程序不知晓什么时候进行资源清理，垃圾回收，程序只能控制对象何时不再被任何引用变量引用，绝不能控制对象何时被回收。

程序无法控制java垃圾回收的时机，但是可以通知系统进行垃圾回收（强制垃圾回收），但是只是通知，系统是否垃圾回收还是不确定。大部分时候，程序强制系统回收后总会有一些效果

强制回收方法

System.gc()  
  
Runtime.getRuntime().gc()

**5.什么是finalize方法？**

1.在垃圾回收机制回收某个对象占用的资源之前，通常要求程序调用适当的方法来进行资源的清理，在java中提供了默认的机制来进行资源的清理（finalize（）方法）

原型:protected void finalize() throws Throwable

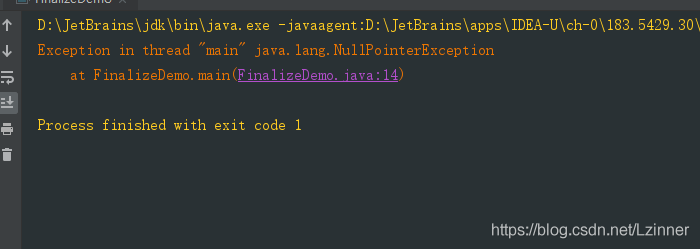
特点:

* 不要主动去调用finalize（）方法，由垃圾回收机制调用。
* 该方法的调用具有随机性。
* 在调用该方法的时候，可能使该对象或者系统中的对象重新变成可达状态（有其他对象引用）。
* 当JVM执行finalize方法出现异常时，垃圾回收机制不会报告异常，程序会进行执行。
* 如果想人为去清理某个类的资源时，由于finalize方法具有随机性，因此不要采用该方法。

举个栗子:

public class FinalizeDemo {  
 private static FinalizeDemo fd=null;  
 public void info(){  
 System.out.println("测试资源调度的Finalize方法");  
 }  
 public static void main(String[]args) throws Exception{  
 new FinalizeDemo();//创建新的对象  
 //通知系统进行资源清理  
 System.gc();  
 //让系统执行finalize方法 强制回收  
 Runtime.getRuntime().runFinalization();  
 System.runFinalization();  
 fd.info();  
 }  
 //重写该方法  
 @Override  
 public void finalize(){  
 //使对象重新变成可达状态  
 fd=this;  
 }  
}

2、使用system.gc（）可以提醒系统去清空资源 如果调用取消强制垃圾回收，会出现空指针异常（因为程序没有通知系统进行垃圾回收，内存也不紧张，所以不会调用finalize方法，fd变量依旧为null。出现空指针异常）



————————————————  
版权声明：本文为CSDN博主「君尔」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/Lzinner/article/details/89086312

### 6.如何判断一个对象是否存活？（或者 GC 对象的判定方法）

答案：https://www.cnblogs.com/java-spring/p/9855129.html

[如何判断对象是否存活/死去](https://www.cnblogs.com/java-spring/p/9855129.html)

在堆里面存放着Java世界中几乎所有的对象实例，垃圾收集器对堆内存进行回收前，都会先判断这些

对象之中哪些还“存活”着，哪些已经“死去”(即不可能在被任何途径使用的对象)。一共有两种算法：

**1、引用计数算法**

给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器

值就减1；任何时刻计数器为0的对象就是不可能再被使用的。

JVM里面并没有选用引用计数算法来管理内存，主要原因是它很难解决对象之间相互循环引用的问题。

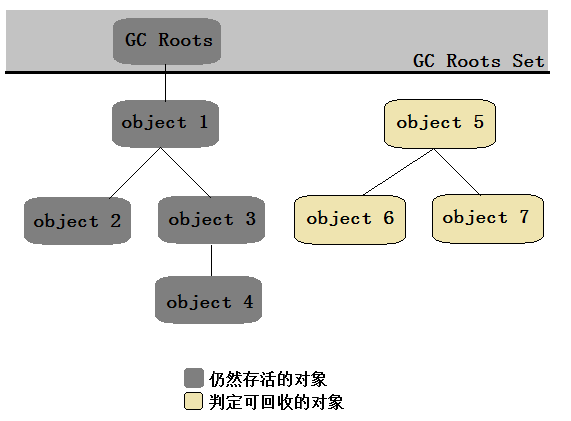
**2、可达性分析算法**

通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为

引用链(Reference Chain)，当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的。

用下面一个图说明一下：

可达性分析算法判定对象是否可回收



在Java语言中，可作为GC Roots的对象包括下面几种：

* 虚拟机栈（栈帧中的本地变量表）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JNI（即一般说的Native方法）引用的对象。

需要说明的是，即使再可达性分析算法中不可达的对象，也并非是“非死不可”的，这时候它们暂时处于

“缓刑”阶段，要真正宣告一个对象死亡，至少要经历两次标记过程：

1、对象在进行可达性分析后被发现不可达，它将会被第一次标记并进行一次筛选，筛选的条件是此对象是否有必要执行finalise()方法，当对象没有覆盖finalize()方法或者finalize()方法已经被JVM调用过，那么就没必要执行finalize()方法；

2、如果被判定为有必要执行finalize()方法，那么此对象将会放置在一个叫做F-Quenen的队列之中，并在稍后由一个虚拟机自动建立的、低优先级的Finalize线程去触发这个方法。finalize()方法是对象逃脱死亡的最后一次机会，稍后GC将对F-Quenen中的对象进行第二次小规模的标记，如果对象要在finalize()中成功拯救自己——只要重新与引用链上的任何一个对象建立关系即可，譬如把自己(this关键字)赋值给某个类变量或者对象的成员变量，那么在第二次标记时它将被移出“即将回收”集合；如果对象这时候还么有成功逃脱，那他就会真的被回收了。

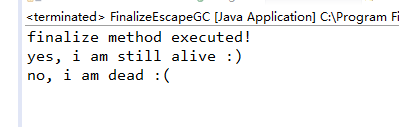
用一段代码来看一下一个对象的finalize()被执行，但是它仍然可以存活：

[fig:](javascript:void(0);)

/\*\*  
 \* 此代码演示了两点：   
 \* 1.对象可以被GC时自我拯救   
 \* 2.这种自救的机会只有一次，因为一个对象的finzlize()方法最多只会被系统自动调用一次  
 \*/  
public class FinalizeEscapeGC {  
 public static FinalizeEscapeGC SAVE\_HOOK = null;  
  
 public void isAlive() {  
 System.out.println("yes, i am still alive :)");  
 }  
  
 protected void finalize() throws Throwable {  
 super.finalize();  
 System.out.println("finalize method executed!");  
 FinalizeEscapeGC.SAVE\_HOOK = this;  
 }  
  
 public static void main(String[] args) throws Throwable {  
 SAVE\_HOOK = new FinalizeEscapeGC();  
  
 // 对象第一次成功拯救自己  
 SAVE\_HOOK = null;  
 System.gc();  
 // 因为finalize方法优先级很低，所以暂停0.5s以等待它  
 Thread.sleep(500);  
 if (SAVE\_HOOK != null) {  
 SAVE\_HOOK.isAlive();  
 } else {  
 System.out.println("no, i am dead :(");  
 }  
  
 // 下面这段代码与上面的完全相同，但是这次自救却失败了  
 SAVE\_HOOK = null;  
 System.gc();  
 // 因为finalize方法优先级很低，所以暂停0.5s以等待它  
 Thread.sleep(500);  
 if (SAVE\_HOOK != null) {  
 SAVE\_HOOK.isAlive();  
 } else {  
 System.out.println("no, i am dead :(");  
 }  
  
 }  
}

[fig:](javascript:void(0);)

运行结果：



SAVE\_HOOK对象的finalize()方法确实被GC收集器触发了，并且在被收集前成功逃脱了。

注意代码中有两段完全一样的代码片段，执行结果确实一次成功逃脱，一次失败，这是因为

任何一个对象的finalize()方法都只会被系统自动调用一次，如果对象面临下一次回收，它的

finalize()方法不会被再次执行，因此第二段代码的自救行动失败了。

### 7.垃圾回收的优点和原理。并考虑 2 种回收机制

答案https://www.cnblogs.com/pypua/articles/8779469.html

[【Java面试题】49 垃圾回收的优点和原理。并考虑2种回收机制。](https://www.cnblogs.com/pypua/articles/8779469.html)

1、[Java](http://lib.csdn.net/base/javase)语言最显著的特点就是引入了垃圾回收机制，它使java程序员在编写程序时不再考虑内存管理的问题。   
2、由于有这个垃圾回收机制，java中的对象不再有“作用域”的概念，只有引用的对象才有“作用域”。   
3、垃圾回收机制有效的**防止了内存泄露**，可以**有效**的**使用**可使用的**内存。**   
4、垃圾回收器通常作为一个单独的低级别的线程运行，在***不可预知的***情况下对内存堆中已经死亡的或很长时间没有用过的对象进行清除和回收。   
5、程序员不能实时的对某个对象或所有对象调用垃圾回收器进行垃圾回收。

垃圾回收机制有**分代复制垃圾回收**、**标记垃圾回收**、**增量垃圾回收**。

**一、分代复制垃圾回收**

不同的对象的生命周期是不一样的。因此，不同生命周期的对象可以采取不同的收集方式，以便提高回收效率。 在[Java](http://lib.csdn.net/base/javase)程序运行的过程中，会产生大量的对象，其中有些对象是与业务信息相关，比如Http请求中的

Session对象、线程、Socket连接，这类对象跟业务直接挂钩，因此生命周期比较长。但是还有一些对象，主要是程序运行过程中生成的临时变量，这些对象生命周期会比较短，比如：String对象，由于其不变类的特性，系统会产生大量的这些对象，有些对象甚至只用一次即可回收。如果每次垃圾回收都是对整个堆空间进行回收，花费时间相对会 长，并且生命周期长的对象依旧存在，因此引入分代回收，把不同生命周期的对象放在不同代上，不同代上采用最适合它的垃圾回收方式进行回收。

虚拟机中的共划分为三个代：年轻代（Young Generation）、年老点（Old Generation）和持久代（Permanent Generation）。由于对象进行了分代处理，因此垃圾回收区域、时间也不一样。GC有两种类型：Scavenge GC和Full GC。

Scavenge GC：一般情况下，当新对象生成，并且在Eden申请空间失败时，就会触发Scavenge GC，对Eden区域进行GC，清除非存活对象，并且把尚且存活的对象移动到Survivor区。然后整理Survivor的两个区。这种方式的GC是对年轻代的Eden区进行，不会影响到年老代。因为大部分对象都是从Eden区开始的，同时Eden区不会分配的很大，所以Eden区的GC会频繁进行。因而，一般在这里需要使用速度快、效率高的[算法](http://lib.csdn.net/base/datastructure)，使Eden去能尽快空闲出来。

Full GC ：对整个堆进行整理，包括Young、Tenured和Perm。Full GC因为需要对整个对进行回收，所以比ScavengeGC要慢，因此应该尽可能减少Full GC的次数。在对JVM调优的过程中，很大一部分工作就是对于FullGC的调节。有如下原因可能导致Full GC：

· 年老代（Tenured）被写满

· 持久代（Perm）被写满

· System.gc()被显示调用

上一次GC之后Heap的各域分配策略动态变化

**二、标记垃圾回收**

它是第一个可以回收被循环引用的[数据结构](http://lib.csdn.net/base/datastructure)的垃圾回收算法.现在仍旧有许多常用的垃圾回收技术使用各种各样的标记清除算法的变体。

在使用标记清除算法时,未引用对象并不会被立即回收.取而代之的做法是,垃圾对象将一直累计到内存耗尽为止.当内存耗尽时,程序将会被挂起,垃圾回收开始执行.当所有的未引用对象被清理完毕时,程序才会继续执行。

标记清除算法由两个阶段组成:

① 标记阶段，标记所有的可访问对象。

② 收集阶段，垃圾收集算法扫描堆并回收所有的未标记对象。

标记垃圾回收的优点：标记清除式的垃圾回收跟踪了由根(root)访问的所有对象,所以即使是在有循环引用时,它也可以正确地标记并执行垃圾回收工作。另外，对于引用对象的常规操作不会产生任何的额外开销。

缺点：当垃圾回收算法执行时，正常的程序会被挂起。特别是,如果一个程序是交互式程序或者正在有一些实时运算时，这就会成为一个问题。比如，一个正在进行垃圾回收的交互式程序会周期的无响应。

**三、增量垃圾回收**

对这种垃圾回收机制始终无法理解透彻，只能在此稍作解释，具体该如何定义也请自行google。

简单地说，它的存在是为了解决标记清除的长停顿问题。增量回收是将GC分成几部分来执行。设置「GC最多中断10ms」这样的条件限制来使GC的终端时间视作可预测的。但是，在两段的GC程序之间，引用关系可能发生了变化。所以，这种GC算法也要写屏障，来记录引用关系的变化。虽然这种方式控制了中断最高时间，但是由于中断次数增加，GC总时间是增加的。

### 8.垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收？

[垃圾回收器的基本原理是什么？垃圾回收器可以马上回收内存吗？有什么办法主动通知虚拟机进行垃圾回收？](https://blog.csdn.net/weixin_42697074/article/details/88904346?depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromBaidu-4&utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromBaidu-4)

**1、基本原理（对象引用遍历方式）：**

对于GC（垃圾收集）来说，当程序员创建对象时，GC就开始监控这个对象的地址、大小以及使用情况。  
通常，GC采用有向图的方式记录和管理堆(heap)中的所有对象。  
通过这种方式确定哪些对象是"可达的"，哪些对象是"不可达的"。  
当GC确定一些对象为"不可达"时，GC就有责任回收这些内存空间。

2、垃圾回收器不可以马上回收内存。  
垃圾收集器不可以被强制执行，但程序员可以通过调研System.gc方法来建议执行垃圾收集。  
记住，只是建议。一般不建议自己写System.gc，因为会加大垃圾收集工作量

程序员可以手动执行System.gc()，通知GC运行，但是Java语言规范并不保证GC一定会执行。

————————————————  
版权声明：本文为CSDN博主「老新人」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/weixin\_42697074/article/details/88904346

### 9.Java 中会存在内存泄漏吗，请简单描述

答案https://www.cnblogs.com/guweiwei/p/6641762.html

所谓**内存泄露**就是指一个不再被程序使用的对象或变量一直被占据在内存中。

[Java](http://lib.csdn.net/base/javase)中有垃圾回收机制，它可以保证一对象不再被引用的时候，即对象变成了孤儿的时候，对象将自动被垃圾回收器从内存中清除掉。由于Java 使用**有向图的方式**进行垃圾回收管理，可以*消除引用循环*的问题，例如有两个对象，相互引用，只要它们和根进程不可达的，那么GC也是可以回收它们的。

package com.huawei.interview;  
  
import java.io.IOException;  
  
 public class GarbageTest {  
 /\*\*  
 \* @param args  
 \* @throws IOException   
 \*/  
 public static void main(String[] args) throws IOException {  
 // TODO Auto-generated method stub  
 try {  
 gcTest();  
 } catch (IOException e) {  
 // TODO Auto-generated catch block  
 e.printStackTrace();  
 }  
 System.out.println("has exited gcTest!");  
 System.in.read();  
 System.in.read();   
 System.out.println("out begin gc!");   
 for(int i=0;i<100;i++)  
 {  
 System.gc();  
 System.in.read();   
 System.in.read();   
 }  
 }  
  
 private static void gcTest() throws IOException {  
 System.in.read();  
 System.in.read();   
 Person p1 = new Person();  
 System.in.read();  
 System.in.read();   
 Person p2 = new Person();  
 p1.setMate(p2);  
 p2.setMate(p1);  
 System.out.println("before exit gctest!");  
 System.in.read();  
 System.in.read();   
 System.gc();  
 System.out.println("exit gctest!");  
 }  
  
 private static class Person  
 {  
 byte[] data = new byte[20000000];  
 Person mate = null;  
 public void setMate(Person other)  
 {  
 mate = other;  
 }  
 }  
}

java中的内存泄露的情况：**长生命周期的对象持有短生命周期对象的引用就很可能发生内存泄露**，尽管短生命周期对象已经不再需要，但是因为长生命周期对象持有它的引用而导致不能被回收，这就是java中内存泄露的发生场景，通俗地说，就是<u>程序员可能创建了一个对象，以后一直不再使用这个对象，这个对象却一直被引用，即这个对象无用但是却无法被垃圾回收器回收的</u>，这就是java中可能出现内存泄露的情况，例如，缓存系统，我们加载了一个对象放在缓存中(例如放在一个全局map对象中)，然后一直不再使用它，这个对象一直被缓存引用，但却不再被使用。   
检查java中的内存泄露，一定要让程序将各种分支情况都完整执行到程序结束，然后看某个对象是否被使用过，如果没有，则才能判定这个对象属于内存泄露。

如果一个外部类的实例对象的方法返回了一个内部类的实例对象，这个内部类对象被长期引用了，即使那个外部类实例对象不再被使用，但由于内部类持久外部类的实例对象，这个外部类对象将不会被垃圾回收，这也会造成内存泄露。

public class Stack {   
 private Object[] elements=new Object[10];   
 private int size = 0;   
 public void push(Object e){   
 ensureCapacity();   
 elements[size++] = e;   
 }   
 public Object pop(){   
 if( size == 0)   
 throw new EmptyStackException();   
 return elements[–size];   
 }   
 private void ensureCapacity(){   
 if(elements.length == size){   
 Object[] oldElements = elements;   
 elements = new Object[2 \* elements.length+1];   
 System.arraycopy(oldElements,0, elements, 0, size);   
 }   
 }   
}

内存泄露的另外一种情况：当一个对象被存储进HashSet集合中以后，就不能修改这个对象中的那些参与计算哈希值的字段了，否则，对象修改后的哈希值与最初存储进HashSet集合中时的哈希值就不同了，在这种情况下，即使在contains方法使用该对象的当前引用作为的参数去HashSet集合中检索对象，也将返回找不到对象的结果，这也会导致无法从HashSet集合中单独删除当前对象，造成内存泄露

### 10.深拷贝和浅拷贝。

答案：https://www.cnblogs.com/lgxblog/p/11568153.html

[一文搞懂Java中深拷贝和浅拷贝的区别](https://www.cnblogs.com/lgxblog/p/11568153.html)

Java深拷贝和浅拷贝的区别

**1、浅拷贝**

被复制的对象的所有的变量都与原对象有相同的值，而所有的引用对象仍然指向原来的对象。换言之，浅拷贝只是复制所考虑的对象，不复制引用对象。

[fig:](javascript:void(0);)

1 class Experience {  
 2 private String skill;  
 3 public void setSkill(String skill){  
 4 this.skill = skill;  
 5 }  
 6 public void setExperience(String skill){  
 7 this.skill = skill;  
 8 }  
 9   
10 @Override  
11 public String toString() {  
12 return skill;  
13 }  
14 }  
15   
16 public class CloneTest implements Cloneable{  
17   
18 private int age ;  
19 private Experience experience;   
20   
21 public CloneTest(){  
22 this.age = 10;  
23 this.experience = new Experience();  
24 }  
25 public Experience getExperience() {  
26 return experience;  
27 }  
28 public void setExperience(String skill){  
29 experience.setExperience(skill);  
30 }  
31 public void show(){  
32 System.out.println(experience.toString());  
33 }  
34 public int getAge() {  
35 return age;  
36 }  
37 @Override  
38 protected Object clone() throws CloneNotSupportedException {  
39 return (CloneTest) super.clone();  
40 }  
41 }  
42 //测试类  
43 class MianTest{  
44   
45 public static void main(String[] args) throws CloneNotSupportedException {  
46 CloneTest test = new CloneTest();  
47 test.setExperience("我是小明，我精通Java，C++的复制粘贴");  
48 test.show();  
49 CloneTest cloneTest = (CloneTest) test.clone();  
50 cloneTest.show();  
51 cloneTest.setExperience("我是小明的副本，我精通Java,C++");  
52 cloneTest.show();  
53 test.show();  
54 System.out.println(cloneTest.getAge());  
55 }  
56 }

[fig:](javascript:void(0);)

输出的结果：

**我是小明，我精通Java，C++的复制粘贴**

**我是小明，我精通Java，C++的复制粘贴**

**我是小明的副本，我精通Java,C++**

**我是小明的副本，我精通Java,C++**

10

从结果中不难看出，拷贝的副本改变了Experience的skill属性，原类中的skill属性打印出来也是修改后的结果，说明引用 类型的拷贝没有将对象拷贝，引用的指向还是原类中的指向

**2、深拷贝**

除了被复制的对象的所有变量都有原来对象的值之外，还把引用对象也指向了被复制的新对象

[fig:](javascript:void(0);)

1 class Experience implements Cloneable{  
 2 private String skill;  
 3 public void setSkill(String skill){  
 4 this.skill = skill;  
 5 }  
 6 public void setExperience(String skill){  
 7 this.skill = skill;  
 8 }  
 9   
10 @Override  
11 public String toString() {  
12 return skill;  
13 }  
14   
15 @Override  
16 protected Object clone() throws CloneNotSupportedException {  
17 return super.clone();  
18 }  
19 }  
20   
21 public class CloneTest implements Cloneable{  
22   
23 private int age ;  
24 private Experience experience;  
25   
26 public CloneTest(){  
27 this.age = 10;  
28 this.experience = new Experience();  
29 }  
30 public Experience getExperience() {  
31 return experience;  
32 }  
33 public void setExperience(String skill){  
34 experience.setExperience(skill);  
35 }  
36 public void show(){  
37 System.out.println(experience.toString());  
38 }  
39 public int getAge() {  
40 return age;  
41 }  
42 @Override  
43 protected Object clone() throws CloneNotSupportedException {  
44 CloneTest o = (CloneTest) super.clone();  
45 o.experience = (Experience) o.getExperience().clone();  
46 return o;  
47 }  
48 }  
49 class MianTest{  
50   
51 public static void main(String[] args) throws CloneNotSupportedException {  
52 CloneTest test = new CloneTest();  
53 test.setExperience("我是小明，我精通Java，C++的复制粘贴");  
54 test.show();  
55 CloneTest cloneTest = (CloneTest) test.clone();  
56 cloneTest.show();  
57 cloneTest.setExperience("我是小明的副本，我精通Java,C++");  
58 cloneTest.show();  
59 test.show();  
60 System.out.println(cloneTest.getAge());  
61 }  
62 }

[fig:](javascript:void(0);)

输出的结果：

**我是小明，我精通Java，C++的复制粘贴**

**我是小明，我精通Java，C++的复制粘贴**

**我是小明的副本，我精通Java,C++**

**我是小明，我精通Java，C++的复制粘贴**

**10**

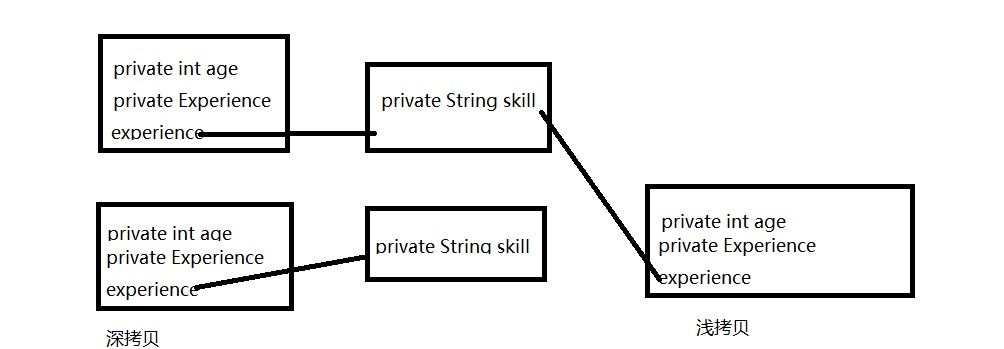
*可以看出和第一次的结果不同了。*

o.experience =(Experience) o.getExperience().clone();

加了这句之后效果不同的。说明实现了深拷贝，原引用对象也复制过来了

总结

深拷贝和浅拷贝的区别用一张图来区分：



分类: [java](https://www.cnblogs.com/lgxblog/category/1513909.html)

### 11.System.gc() 和 Runtime.gc() 会做什么事情？

答案：https://blog.csdn.net/timmobile/article/details/49452921

(1) GC是垃圾收集的意思（Gabage Collection）,内存处理是编程人员容易出现问题的地方，忘记或者错误的内存回收会导致程序或系统的不稳定甚至崩溃，Java提供的GC功能可以自动监测对象是否超过作用域从而达到自动回收内存的目的，[Java语言](https://www.baidu.com/s?wd=Java语言&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YvPjcYryP-ujfvuj0kPvmk0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6K1TL0qnfK1TL0z5HD0IgF_5y9YIZ0lQzqlpA-bmyt8mh7GuZR8mvqVQL7dugPYpyq8Q1R4njnkPWn1P0)没有提供释放已[分配内存](https://www.baidu.com/s?wd=分配内存&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YvPjcYryP-ujfvuj0kPvmk0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6K1TL0qnfK1TL0z5HD0IgF_5y9YIZ0lQzqlpA-bmyt8mh7GuZR8mvqVQL7dugPYpyq8Q1R4njnkPWn1P0)的显示操作方法。

(2) 对于GC来说，当程序员创建对象时，GC就开始监控这个对象的地址、大小以及使用情况。通常，GC采用有向图的方式记录和管理堆(heap)中的所有对象。通过这种方式确定哪些对象是”可达的”，哪些对象是”不可达的”。当GC确定一些对象为”不可达”时，GC就有责任回收这些内存空间。可以。程序员可以手动执行System.gc()，通知GC运行，但是[Java语言](https://www.baidu.com/s?wd=Java语言&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YvPjcYryP-ujfvuj0kPvmk0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6K1TL0qnfK1TL0z5HD0IgF_5y9YIZ0lQzqlpA-bmyt8mh7GuZR8mvqVQL7dugPYpyq8Q1R4njnkPWn1P0)规范并不保证GC一定会执行。

(3) [垃圾回收](https://www.baidu.com/s?wd=垃圾回收&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YvPjcYryP-ujfvuj0kPvmk0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6K1TL0qnfK1TL0z5HD0IgF_5y9YIZ0lQzqlpA-bmyt8mh7GuZR8mvqVQL7dugPYpyq8Q1R4njnkPWn1P0)是一种动态存储管理技术，它自动地释放不再被程序引用的对象，当一个对象不再被引用的时候,按照特定的垃圾收集算法来实现资源自动回收的功能。

(4) System.gc();就是呼叫java虚拟机的[垃圾回收](https://www.baidu.com/s?wd=垃圾回收&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YvPjcYryP-ujfvuj0kPvmk0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6K1TL0qnfK1TL0z5HD0IgF_5y9YIZ0lQzqlpA-bmyt8mh7GuZR8mvqVQL7dugPYpyq8Q1R4njnkPWn1P0)器运行回收内存的垃圾。

(5) 当不存在对一个对象的引用时，我们就假定不再需要那个对象，那个对象所占有的存储单元可以被收回，可通过System.gc()方法回收，但一般要把不再引用的对象标志为null为佳。

(6) 每个 Java 应用程序都有一个 Runtime 类实例，使应用程序能够与其运行的环境相连接。可以通过 getRuntime 方法获取当前运行时。 Runtime.getRuntime().gc();

(7) java.lang.System.gc()只是java.lang.Runtime.getRuntime().gc()的简写，两者的行为没有任何不同。

(8) 唯一的区别就是System.gc()写起来比Runtime.getRuntime().gc()简单点. 其实基本没什么机会用得到这个命令, 因为这个命令只是建议JVM安排GC运行, 还有可能完全被拒绝。 GC本身是会周期性的自动运行的,由JVM决定运行的时机,而且现在的版本有多种更智能的模式可以选择,还会根据运行的机器自动去做选择,就算真的有性能上的需求,也应该去对GC的运行机制进行微调,而不是通过使用这个命令来实现性能的优化。

重点分析：浅谈Java的System.gc()实现(https://blog.csdn.net/huangdeijia/article/details/89403936?depth*1-utm*source=distribute.pc*relevant.none-task-blog-BlogCommendFromBaidu-1&utm*source=distribute.pc\_relevant.none-task-blog-BlogCommendFromBaidu-1)

### 12.finalize() 方法什么时候被调用？析构函数 (finalization) 的目的是什么？

链接：https://www.nowcoder.com/questionTerminal/d8eab06913084e42b515633604eef7cd?pos=28&mutiTagIds=570&orderByHotValue=0&done=0  
来源：牛客网

参考：《深入理解Java虚拟机》

对于Java而言：

调用时机：当垃圾回收器要宣告一个对象死亡时，至少要经过两次标记过程：如果对象在进行可达性分析后发现没有和GC Roots相连接的引用链，就会被第一次标记，并且判断是否执行finalizer( )方法，如果对象覆盖finalizer( )方法且未被虚拟机调用过，那么这个对象会被放置在F-Queue队列中，并在稍后由一个虚拟机自动建立的低优先级的Finalizer线程区执行触发finalizer( )方法，但不承诺等待其运行结束。

finalization的目的：对象逃脱死亡的最后一次机会。（只要重新与引用链上的任何一个对象建立关联即可。）但是不建议使用，运行代价高昂，不确定性大，且无法保证各个对象的调用顺序。可用try-finally或其他替代。

答案二https://www.cnblogs.com/kissazi2/p/3620743.html

[**扯扯Java中Finalization的意义**](https://www.cnblogs.com/kissazi2/p/3620743.html)

这是Stack Overflow上关于Finalizetion意义的两段讨论，这两个观点是互为补充的。

**观点1：**

垃圾回收器（The garbage collector）自动在后台运行（虽然它也可以被直接调用，但是一般不这么干），基本上它就是清理那些没有被其他对象引用的对象。（垃圾回收器的整个工作原理要比上面说的复杂，但基本就是这样）。所以它不会改变活动对象（live object）上的任何引用。如果一个对象不能被其他活动对象引用（can not be accessed），那么就意味着它可以被安全地回收。

Finalizetion主要用来释放被对象占用的资源（不是指内存，而是指其他资源，比如文件(File Handle)、端口(ports)、数据库连接(DB Connection)等）。然而，它不能真正有效地工作。

●finalize()的调用事件是不可预知的。

●事实上，没有机制能保证finalize()一定会被调用。

但是即使finalize()被保证一定会被调用，它也不是一个释放资源的好地方；当它被调用准备清理所有你打开的数据库连接(DB Connections)时，系统可能已经耗尽了所有空闲的连接，然后你的app就不能继续跑了。

**观点2：**

一旦垃圾回收器运行（VM决定它什么时候需要清理内存，你不能强迫它运行）并决定从某个对象（this object）中回收内存时（这意味着已经没有引用（reference）指向这个对象，至少它不是可达（reachable）的对象），在它删除这个对象占用的内存时，它将调用对象上的finalize()方法。你可以肯定的是如果进行了垃圾回收，那么在对象消失（dispperars）前finalize()方法会被调用，但你无法确定它是否会被GC回收，所以你不能依赖这个方法做清理工作。你应该在finally{}里面做清理工作，不要使用finalize()方法，因为它不一定保证会被运行。

从以上观点总结：GC是用来回收对象占用的内存的，而finalize()这个东西会被finalizer调用，用来回收资源（文件、端口、数据库连接等）。

本文内容翻译自[What is the purpose of Finalization in java?](http://stackoverflow.com/questions/2450580/what-is-the-purpose-of-finalization-in-java)

作者：[kissazi2](http://www.cnblogs.com/kissazi2/)  
出处：http://www.cnblogs.com/kissazi2/  
本文版权归作者所有，欢迎转载，但未经作者同意必须保留此段声明，且在文章页面明显位置给出原文连接，否则保留追究法律责任的权利。

### 13.如果对象的引用被置为 null，垃圾收集器是否会立即释放对象占用的内存？

答案：https://blog.csdn.net/qq\_32534441/article/details/95456973

如果对象的引用被置为null，垃圾收集器是否会立即释放对象占用的内存？

阅读：  
Java的finalize()：  
https://blog.csdn.net/qq\_32534441/article/details/83585359

Java中的垃圾回收一般是在Java堆中进行，因为堆中几乎存放了Java中所有的对象实例。谈到Java堆中的垃圾回收，自然要谈到引用。在JDK1.2之前，Java中的引用定义很很纯粹：如果reference类型的数据中存储的数值代表的是另外一块内存的起始地址，就称这块内存代表着一个引用。但在JDK1.2之后，Java对引用的概念进行了扩充，将其分为强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）、虚引用（Phantom Reference）四种，引用强度依次减弱。

* 强引用：如“Object obj = new Object（）”，这类引用是Java程序中最普遍的。只要强引用还存在，垃圾收集器就永远不会回收掉被引用的对象。
* 软引用：它用来描述一些可能还有用，但并非必须的对象。在系统内存不够用时，这类引用关联的对象将被垃圾收集器回收。JDK1.2之后提供了SoftReference类来实现软引用。
* 弱引用：它也是用来描述非需对象的，但它的强度比软引用更弱些，被弱引用关联的对象只能生存岛下一次垃圾收集发生之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。在JDK1.2之后，提供了WeakReference类来实现弱引用。
* 虚引用：最弱的一种引用关系，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。为一个对象设置虚引用关联的唯一目的是希望能在这个对象被收集器回收时收到一个系统通知。JDK1.2之后提供了PhantomReference类来实现虚引用。

一旦垃圾回收器准备好释放对象占用的存储空间，首先会去调用finalize()方法  
①进行一些必要的清理工作（对垃圾回收器不能处理的特殊情况进行处理）（例子在下边）  
②也有可能使该对象重新被引用，我习惯叫这种作用为复活。注意！！每个对象的finalize()方法只能被执行一次，第二次就会直接跳过finalize()方法，这就是为了防止出现对象无限复活，内存空间只增不减。  
 一般忽略第二种情况，概念就变成了：一旦垃圾收集器准备好释放对象占用的存储空间（进入第一个回收周期），首先会去调用finalize()方法进行一些必要的清理工作，只有到下一次再进行垃圾回收动作（下一个回收周期）的时候，才会真正释放这个对象所占用的内存空间。

答：  
 不会立即释放对象占用的内存。 如果对象的引用被置为null，只是断开了当前线程栈帧中对该对象的引用关系，而 垃圾收集器是运行在后台的线程，只有当用户线程运行到安全点(safe point)或者安全区域才会扫描对象引用关系，扫描到对象没有被引用则会标记对象，这时候仍然不会立即释放该对象内存，因为有些对象是可恢复的（在 finalize方法中恢复引用 ）。只有确定了对象无法恢复引用的时候才会清除对象内存。

### 14.什么是分布式垃圾回收（DGC）？它是如何工作的？

答案：https://www.nowcoder.com/questionTerminal/eed60f12be6340ef96e0a661e2dea0d6

RMI 子系统实现基于引用计数的“分布式垃圾回收”(DGC)，以便为远程服务器对象提供自动内存管理设施。

当客户机创建（序列化）远程引用时，会在服务器端 DGC 上调用 dirty()。当客户机完成远程引用后，它会调用对应的 clean() 方法。

针对远程对象的引用由持有该引用的客户机租用一段时间。租期从收到 dirty() 调用开始。在此类租约到期之前，客户机必须通过对远程引用额外调用 dirty() 来更新租约。如果客户机不在租约到期前进行续签，那么分布式垃圾收集器会假设客户机不再引用远程对象。

DGC叫做分布式垃圾回收。RMI使用DGC来做自动垃圾回收。因为RMI包含了跨虚拟机的远程对象的引用，垃圾回收是很困难的。DGC使用引用计数算法来给远程对象提供自动内存管理。

分布式垃圾回收概念：

1)Java虚拟机中，一个远程对象不仅会被本地虚拟机内的变量引用，还会被远程引用。

2)只有当一个远程对象不受到任何本地引用和远程引用，这个远程对象才会结束生命周期。

说明：

1)服务端的一个远程对象在3个地方被引用：

1>服务端的一个本地对象持有它的本地引用

2>服务端的远程对象已经注册到rmiregistry注册表中，也就是说，rmiregistry注册表持有它的远程引用。

3>客户端获得远程对象的存根对象，也就是说，客户端持有它的远程引用。

2)服务端判断客户端是否持有远程对象引用的方法：

1>当客户端获得一个服务端的远程对象的存根时，就会向服务器发送一条租约(lease)通知，以告诉服务器自己持有了这个远程对象的引用了。

2>客户端定期地向服务器发送租约通知，以保证服务器始终都知道客户端一直持有着远程对象的引用。

3>租约是有期限的，如果租约到期了，服务器则认为客户端已经不再持有远程对象的引用了。

JDK-API解释：http://jszx-jxpt.cuit.edu.cn/JavaAPI/java/rmi/dgc/DGC.html

DGC 抽象用于分布式垃圾回收算法的服务器端。此接口包含了两个方法：dirty 和 clean。当一个远程引用在客户机（客户机由其 VMID 表示）被解组时，则进行一次脏 (dirty) 调用。当客户机上不存任何针对远程引用的更多引用时，则进行一次相应的洁 (clean) 调用。一次失败的脏调用必须安排一次强洁调用，这样调用的序列号才能保持，以检测未来由分布式垃圾回收器接收的无序调用。针对远程对象的引用由保持该引用的客户机租借一段时间。租借期从接收到脏调用时开始。对租借进行续期是客户机的职责，其方式是：在租借期满之前，在客户机保持的远程引用上进行附加的脏调用。如果客户机在期满之前没有对租借进行续期，则分式布垃圾回收器假定远程对象已不再为该客户机所保持。

### 15.串行（serial）收集器和吞吐量（throughput）收集器的区别是什么？

答案一https://blog.csdn.net/u013898617/article/details/78824047：

串行GC：整个扫描和复制过程均采用**单线程**的方式，相对于吞吐量GC来说简单；适合于**单CPU、客户端**级别。

吞吐量GC：采用**多线程**的方式来完成垃圾收集；适合于**吞吐量**要求较高的场合，比较适合中等和大规模的应用程序。

吞吐量收集器使用并行版本的新生代垃圾收集器，它用于中等规模和大规模数据的应用程序。而串行收集器对大多数的小应用(在现代处理器上需要大概100M左右的内存)就足够了。  
————————————————  
版权声明：本文为CSDN博主「bai020」的原创文章，遵循CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/u013898617/article/details/78824047

答案二：https://www.nowcoder.com/questionTerminal/46b6030921164ab0a3cb3dbd6d64f01a?pos=384&orderByHotValue=1

**串行收集器：**

**特性**：

这个收集器是一个单线程的收集器，但它的“单线程”的意义并不仅仅说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束（Stop The World）。

**应用场景：**

Serial收集器是虚拟机运行在Client模式下的默认新生代收集器。

**优势：**

简单而高效（与其他收集器的单线程比），对于限定单个CPU的环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得最高的单线程收集效率。

**吞吐量收集器：**

**先讲一下吞吐量的定义：**  
 吞吐量就是CPU用于运行用户代码的时间与CPU总消耗时间的比值，即吞吐量 = 运行用户代码时间 /（运行用户代码时间 + 垃圾收集时间）。  
 虚拟机总共运行了100分钟，其中垃圾收集花掉1分钟，那吞吐量就是99%。

**特性：**

吞吐量收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器。

**应用场景：**

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验，而高吞吐量则可以高效率地利用CPU时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

### 16.在 Java 中，对象什么时候可以被垃圾回收？

答案：https://www.jianshu.com/p/faa30c8e52e8

一个对象什么时候才能被回收？

**目录：**

1、怎样判断一个对象“已死”？  
2、引用的分类  
3、回收方法区的数据

**1、怎样判断一个对象“已死”？**

在堆里面存放着 Java 世界中几乎所有的对象实例，垃圾收集器在对堆进行回收前，第一件事情就是要确定这些对象之中哪些还“存活”着，哪些已经“死去”（即不可能再被任何途径使用的对象）。

那么怎么判断一个对象“已死”呢，目前有两种算法可以判断对象“已死”。

1. 引用计数算法：  
   这个算法的判断依据是通过给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加 1；当引用失效时，计数器值就减 1；任何时刻计数器为 0 的对象就不可能再被使用的。  
   客观的说，引用计数算法的实现简单，判断效率也很高，在大部分情况下它都是一个不错的算法，也有一些比较著名的应用案例，例如微软公司的 COM（Component Object Model）技术。但是，至少主流的 Java 虚拟机里面没有选用引用计算法来管理内存，其中最主要的原因是它很难解决对象之间相互循环引用的问题。

举个简单的例子，请看下面代码中的 testGC() 方法：对象 objA 和 objB 都有字段 mInstance，赋值令 objA.mInstance = objB 及 objB.mInstance = objA，除此之外，这两个对象再无任何实际上这两个对象已经不可能再被访问，但是它们因为互相引用着对方，导致它们的引用计数都不为 0，于是引用技术算法无法通知 GC 收集器回收它们。

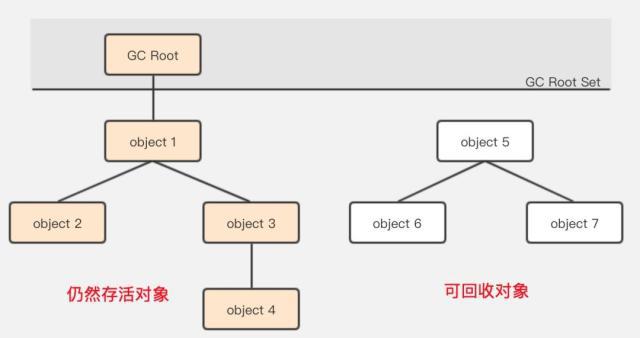
/\*\*   
 \* testGC()方法执行后，objA和objB会不会被GC呢？   
 \* @author zzm   
 \*/   
 public class ReferenceCountingGC {   
   
 public Object instance = null;   
   
 private static final int \_1MB = 1024 \* 1024;   
   
 /\*\*   
 \* 这个成员属性的唯一意义就是占点内存，以便能在GC日志中看清楚是否被回收过   
 \*/   
 private byte[] bigSize = new byte[2 \* \_1MB];   
   
 public static void testGC() {   
 ReferenceCountingGC objA = new ReferenceCountingGC();   
 ReferenceCountingGC objB = new ReferenceCountingGC();   
 objA.instance = objB;   
 objB.instance = objA;   
   
 objA = null;   
 objB = null;   
   
 //假设在这行发生GC，objA和objB是否能被回收？   
 System.gc();   
 }   
 }

1 [Full GC (System) [Tenured: 0K->210K(10240K), 0.0149142 secs] 4603K->210K(19456K), [Perm : 2999K->2999K(21248K)], 0.0150007 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]   
2. 2 Heap   
3. 3 def new generation total 9216K, used 82K [0x00000000055e0000, 0x0000000005fe0000, 0x0000000005fe0000)   
4. 4 Eden space 8192K, 1% used [0x00000000055e0000, 0x00000000055f4850, 0x0000000005de0000)   
5. 5 from space 1024K, 0% used [0x0000000005de0000, 0x0000000005de0000, 0x0000000005ee0000)   
6. 6 to space 1024K, 0% used [0x0000000005ee0000, 0x0000000005ee0000, 0x0000000005fe0000)   
7. 7 tenured generation total 10240K, used 210K [0x0000000005fe0000, 0x00000000069e0000, 0x00000000069e0000)   
8. 8 the space 10240K, 2% used [0x0000000005fe0000, 0x0000000006014a18, 0x0000000006014c00, 0x00000000069e0000)   
9. 9 compacting perm gen total 21248K, used 3016K [0x00000000069e0000, 0x0000000007ea0000, 0x000000000bde0000)   
10.10 the space 21248K, 14% used [0x00000000069e0000, 0x0000000006cd2398, 0x0000000006cd2400, 0x0000000007ea0000)   
11.11 No shared spaces configured.

从运行结果中可以清楚看到，GC日志中包含“4603K->210K”，意味着虚拟机并没有因为这两个对象互相引用就不回收它们，这也从侧面说明虚拟机并不是通过引用计数算法来判断对象是否存活的。

//对象不存在时引用计数器不为 0 的情况  
 public class Example{  
 public Object instance = null;  
   
 public static void test(){  
 Example objA=new Example();  
 Example objB=new Example();  
   
 objA.instance=objB;  
 objB.instance=objA;  
   
 objA=null;  
 objB=null;  
 }  
 }

1. 可达性分析算法：

* 在主流的商用程序语言（如Java）的主流实现中，都是称通过可达性分析来判断对象是否存活的。这个算法的基本思路就是通过一系列的称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是不可用的。
* 
* (以上图片来自网络：https://baijiahao.baidu.com/s?id=1639843226656912938&wfr=spider&for=pc)
* 图中，对象object 5、object 6、object 7虽然互相有关联，但是它们到GC Roots是不可达的，所以它们将会被判定为是可回收的对象。
* 可达性分析算法判定对象是否可回收
* 在 Java 语言中，可作为 GC Roots 的对象包括下面几种：
* 虚拟机栈（栈帧中的本地变量表）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 JNI（即一般说的 Native 方法）引用的对象。

**2、引用的分类**

无论是通过引用计数算法判断对象的引用数量，还是通过可达性分析判断对象的引用链是否可达，判断对象是否存活都与“引用”有关。

在 JDK 1.2 之后，Java 对引用的概念进行了扩充，将引用分为强引用（Strong Reference）、软引用（Soft Reference）、弱引用（Weak Reference）、虚引用（Phantom Reference） 4 种，这 4 种引用强度一次逐渐减弱。

（以下内容有引用：https://www.cnblogs.com/study-everyday/p/7018977.html）

* 强引用就是指在程序代码之中普遍存在的，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象。
* 强引用是使用最普遍的引用。如果一个对象具有强引用，那垃圾回收器绝不会回收它。如下：
* Object o = new Object(); // 强引用
* 当内存空间不足，Java虚拟机宁愿抛出OutOfMemoryError错误，使程序异常终止，也不会靠随意回收具有强引用的对象来解决内存不足的问题。如果不使用时，要通过如下方式来弱化引用，如下：
* o = null; // 帮助垃圾收集器回收此对象
* 显式地设置o为null，或超出对象的生命周期范围，则gc认为该对象不存在引用，这时就可以回收这个对象。具体什么时候收集这要取决于gc的算法。
* 软引用是用来描述一些还有用但并非必须的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常之前（即内存紧张）， 将会把这些对象列进垃圾回收范围之中进行第二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。在 JDK1.2 之后， 提供了 SoftReference 类累实现软引用。

用来描述一些还有用，但并非必需的对象。

如果一个对象只具有软引用，则内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，就会回收这些对象的内存。只要垃圾回收器没有回收它，该对象就可以被程序使用。软引用可用来实现内存敏感的高速缓存。

1 String str = new String("abc"); // 强引用  
2 SoftReference<String> softRef = new SoftReference<String>(str); // 软引用

当内存不足时，等价于：

1 If(JVM.内存不足()) {  
2 str = null; // 转换为软引用  
3 System.gc(); // 垃圾回收器进行回收  
4 }

软引用在实际中有重要的应用，例如浏览器的后退按钮。按后退时，这个后退时显示的网页内容是重新进行请求还是从缓存中取出呢？这就要看具体的实现策略了。

（1）如果一个网页在浏览结束时就进行内容的回收，则按后退查看前面浏览过的页面时，需要重新构建

（2）如果将浏览过的网页存储到内存中会造成内存的大量浪费，甚至会造成内存溢出

这时候就可以使用软引用。

1 Browser prev = new Browser(); // 获取页面进行浏览2 SoftReference sr = new SoftReference(prev); // 浏览完毕后置为软引用 3 if(sr.get()!=null){ 4 rev = (Browser) sr.get(); // 还没有被回收器回收，直接获取5 }else{6 prev = new Browser(); // 由于内存吃紧，所以对软引用的对象回收了7 sr = new SoftReference(prev); // 重新构建8 }

[![复制代码](https://common.cnblogs.com/images/copycode.gif

这样就很好的解决了实际的问题。

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

* 弱引用是用来描述非必需的对象的，但是它的强度比软引用更弱一些，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。 当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。使用 WeekReference 类来实现弱引用。
* 也是用来描述非必需对象的，但是它的强度比软引用更弱一些，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。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。在JDK 1.2之后，提供了WeakReference类来实现弱引用。
* 弱引用与软引用的区别：
* 只具有弱引用的对象拥有更短暂的生命周期。在垃圾回收器线程扫描它所管辖的内存区域的过程中，一旦发现了只具有弱引用的对象，不管当前内存空间足够与否，都会回收它的内存。不过，由于垃圾回收器是一个优先级很低的线程，因此不一定会很快发现那些只具有弱引用的对象。
* String str = new String("abc");  
  WeakReference abcWeakRef = new WeakReference(str);  
  str=``null``;
* 当垃圾回收器进行扫描回收时等价于：
* str = "null";  
  System.gc();
* 如果这个对象是偶尔的使用，并且希望在使用时随时就能获取到，但又不想影响此对象的垃圾收集，那么你应该用 Weak Reference 来记住此对象。
* 下面的代码会让str再次变为一个强引用：
* String abc = abcWeakRef.get();
* 弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果弱引用所引用的对象被垃圾回收，Java虚拟机就会把这个弱引用加入到与之关联的引用队列中。
* 当你想引用一个对象，但是这个对象有自己的生命周期，你不想介入这个对象的生命周期，这时候你就是用弱引用。
* 虚引用也成为幽灵引用或者幻影引用，它是最弱的一种引用。一个对象是否有虚引用的存在，完全不会对其生存周期时间构成影响， 也无法通过虚引用来取得一个对象实例。唯一的作用就是能在这个对象被收集器回收时收到一个系统通知。使用 PhantomReference 表示。
* “虚引用”顾名思义，就是形同虚设，也称为幽灵引用或者幻影引用，它是最弱的一种引用关系。一个对象是否有虚引用的存在，完全不会对其生存时间构成影响，也无法通过虚引用来取得一个对象实例。***为一个对象设置虚引用关联的唯一目的就是希望能在这个对象被收集器回收时收到一个系统通知。***
* 在JDK 1.2之后，提供了PhantomReference类来实现虚引用。
* 虚引用，与其他几种引用都不同，虚引用并不会决定对象的生命周期。如果一个对象仅持有虚引用，那么它就和没有任何引用一样，在任何时候都可能被垃圾回收器回收。
* 虚引用主要用来跟踪对象被垃圾回收器回收的活动。
* **虚引用与软引用和弱引用的一个区别在于：**虚引用必须和引用队列 （ReferenceQueue）联合使用。当垃圾回收器准备回收一个对象时，如果发现它还有虚引用，就会在回收对象的内存之前，把这个虚引用加入到与之关联的引用队列中。
* 我们可以声明虚引用来引用我们感兴趣的对象，在GC要回收的时候，GC收集器会把这个对象添加到ReferenceQueue，这样我们如果检测到ReferenceQueue中有我们感兴趣的对象的时候，说明GC将要回收这个对象了。此时我们可以在GC回收之前做一些其他事情，比如记录些日志什么的。

**3、回收方法区数据**

方法区的垃圾收集主要回收两部分内容：废弃常量和无用的类。回收废弃常量与回收 Java 堆中的对象非常类似。以常量池中字面量的回收为例，假如一个字符串“abc”已经进入了常量池，但是当前系统没有任何一个 String 对象是叫做 “abc”的，换句话说，就是没有任何一个 String 对象引用常量池中的“abc”常量，也没有其他地方引用了这个字面量，如果这时发生内存回收，而且必要的话，这个“abc”常量就会被系统清理出常量池。常量池中的其他类（接口）、方法、字段的符号引用也与此类似。

判定一个常量为“废弃常量”比较简单，而要判断一个常量池中的类是否是“无用的类”条件则苛刻很多。类需要同时满足下面 3 个条件才能称为“无用的类”：

* 该类所有的实例都已经被全部回收，也就是说 Java 堆中不再存在任何该类的实例。
* 加载该类的 ClassLoader 已经被回收。
* 该类对应的 java.lang.Class 对象没有在任何地方引用，并且无法再任何地方通过反射调用该类的方法。

最后做个总结：

1. 我们可以通过 引用计数器 和 可达性算法 来判断一个对象是否“已死”。引用计数器很难解决对象之间互相循环引用的问题，所以在主流的商用程序语言（如Java）的主流实现中，都是称通过可达性分析来判断对象是否存活的。
2. 对象的引用可以分为 强引用、软引用、弱引用 以及 虚引用 4 种，其中被 强引用 引用的对象垃圾收集器永远不会回收掉；被 软引用 引用的对象，只有当系统将要发生内存溢出时，才会去回收软引用引用的对象；只被 弱引用关联的对象，只要发生垃圾收集事件，只被弱引用关联的对象就会被回收；被虚引用关联的对象的唯一作用是能在这个对象被回收器回收时受到一个系统通知。
3. 回收方法区的数据，垃圾收集器主要回收 废弃常量和无用的类 两部分内容。

* 引用自以下:
* [Java虚拟机笔记（二）：GC垃圾回收和对象的引用](https://www.cnblogs.com/study-everyday/p/7018977.html)
* [jvm(4)---垃圾回收（哪些对象可以被回收）](https://www.cnblogs.com/nijunyang/p/11108441.html)

### 17.简述 Java 内存分配与回收策率以及 Minor GC 和 Major GC。

答案：https://www.liangzl.com/get-article-detail-145652.html

**内存分配**：

\1. 栈区：栈可分为Java虚拟机和本地方法栈

\2. 堆区：堆被所有线程共享，在虚拟机启动时创建，是唯一的目的是存放对象实例，是gc的主要区域。通常可分为两个区块年轻代和年老代。更新一点年轻代可分为Eden区，主要放新创建对象，From survivor 和 To survivor 保存 gc 后幸存下的对象，默认情况下各自占比 8:1:1。

\3. 方法区：被所有线程共享，用于存放已被虚拟机加载的信息，常量，静态变量等，是Java虚拟机描述为堆的一个逻辑部分。习惯被称为永久代。

\4. 程序计数器：是当前线程所执行的行号指示器，跳转指令等都依赖这个完成，线程私有。

**回收策略以及Minor GC 和 Major GC(Full GC)**

\1. 对象优先在堆区的Eden区分配。

\2. 大对象直接进入老年代。

\3. 长期存活的对象直接进入老年代。

**回收**：当Eden区没有足够的空间分配时，虚拟机会执行一次Minor GC .Minor GC通常发生在Eden新生代，因为这个区的对象生存期短，发生频率高，回收速度快。Major GC发生在老年代，一般触发老年代的GC不会触发Minor GC ,但是通过配置，可以在之前进行一次Minor GC,能加快老年代的回收速度

### 18.JVM 的永久代中会发生垃圾回收么？

答案：https://www.sohu.com/a/245293459\_661203

垃圾回收不会发生在永久代，如果永久代满了或者是超过了临界值，会触发完全垃圾回收(Full GC)。如果你仔细查看垃圾收集器的输出信息，就会发现永久代也是被回收的。这就是为什么正确的永久代大小对避免Full GC是非常重要的原因。

(注：Java8中已经移除了永久代，新加了一个叫做元数据区的native内存区) 异常处理

答案二：链接：https://www.nowcoder.com/questionTerminal/8f393a761e0f4b67b1c442d092eb484d?toCommentId=139898  
来源：牛客网

永生代也是可以回收的，条件是

1.该类的实例都被回收。

2.加载该类的classLoader已经被回收

3.该类不能通过反射访问到其方法，而且该类的java.lang.class没有被引用

当满足这3个条件时，是可以回收，但回不回收还得看jvm。

### 19.Java 中垃圾收集的方法有哪些？

答案：https://www.cnblogs.com/wangqingming/p/9766803.html

[java中垃圾收集的方法有哪些？](https://www.cnblogs.com/wangqingming/p/9766803.html)

java中垃圾收集的方法有哪些?

一、引用计数算法（Reference Counting）

介绍：给对象添加一个引用计数器，每当一个地方引用它时，数据器加1；当引用失效时，计数器减1；计数器为0的即可被回收。

优点：实现简单，判断效率高

缺点：很难解决对象之间的相互循环引用（objA.instance = objB; objB.instance = objA）的问题，所以java语言并没有选用引用计数法管理内存

二、根搜索算法（GC Root Tracing）

Java和C#都是使用根搜索算法来判断对象是否存活。通过一系列的名为“GC Root”的对象作为起始点，从这些节点开始向下搜索，搜索所有走过的路径称为引用链（Reference Chain）,当一个对象到GC Root没有任何引用链相连时（用图论来说就是GC Root到这个对象不可达时），证明该对象是可以被回收的。

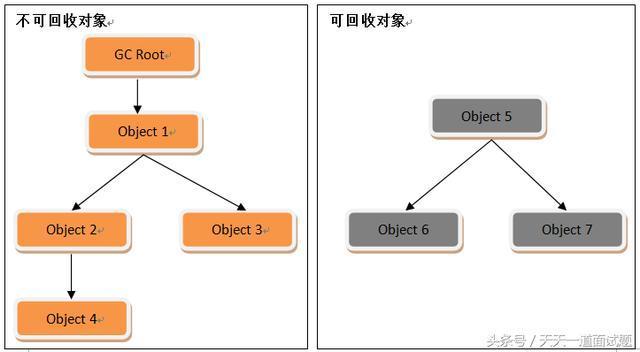
在Java中哪些对象可以成为GC Root?

虚拟机栈（栈帧中的本地变量表）中的引用对象

方法区中的类静态属性引用的对象

方法区中的常量引用对象

本地方法栈中JNI（即Native方法）的引用对象



三、标记-清除算法（Mark-Sweep）

这是垃圾收集算法中最基础的，根据名字就可以知道，它的思想就是标记那些要被回收的对象，然后统一回收。这种方法很简单，但是会有两个主要问题：

1.效率不高，标记和清除的效率都很低；

2.会产生大量不连续的内存碎片，导致以后程序在分配交大的对象时，由于没有充足的连续内存而提前触发一次GC动作。



四、复制算法（Copying）

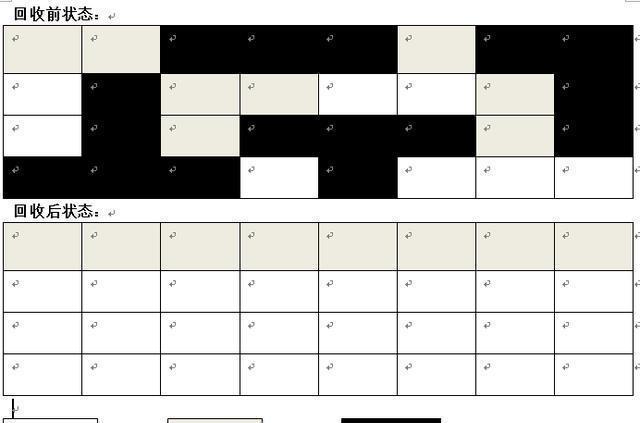
为了解决效率问题，复制算法将可用内存按容量划分相等的两部分，然后每次只使用其中的一块，当第一块内存用完时，就将还存活的对象复制到第二块内存上，然后一次性清除完第一块内存，在将第二块上的对象复制到第一块。但是这种方式，内存的代价太高，每次基本上都要浪费一块内存。

于是将该算法进行了改进，内存区域不再是按照1：1去划分，而是将内存划分为8：1：1三部分，较大的那份内存叫Eden区，其余两块较小的内存叫Survior区。每次都会先使用Eden区，若Eden区满，就将对象赋值到第二块内存上，然后清除Eden区，如果此时存活的对象太多，以至于Survivor不够时，会将这些对象通过分配担保机制赋值到老年代中。（java堆又分为新生代和老年代）。



五、标记-整理算法（Mark-Compact）

该算法是为了解决标记-清楚，产生大量内存碎片的问题；当对象存活率较高时，也解决了复制算法的效率问题。它的不同之处就是在清除对象的时候现将可回收的对象移动一端，然后清除掉端边界以为的对象，这样就不会产生内存碎片。



六、分代收集算法（Generational Collection）

根据对象的存活周期的不同将内存划分为几块，一般就分为新生代和老年代，根据各个年代的特点采用不同的收集算法。新生代（少量存活）用复制算法，老年代（对象存活率高）“标记-清理”算法

补充：分代划分内存介绍

整个JVM内存总共划分为三代：年轻代（Young Generation）、年老代（Old Generation）、持久代（Permanent Generation）

1、年轻代：所有新生成的对象首先都放在年轻代内存中。年轻代的目标就是尽可能快速的手机掉那些生命周期短的对象。年轻代内存分为一块较大的Eden空间和两块较小的Survior空间，每次使用Eden和其中的一块Survior.当回收时，将Eden和Survior中还存活的对象一次性拷贝到另外一块Survior空间上，最后清理Eden和刚才用过的Survior空间。

2、年老代：在年轻代经历了N次GC后，仍然存活的对象，就会被放在老年代中。因此可以认为老年代存放的都是一些生命周期较长的对象。

3、持久代：基本固定不变，用于存放静态文件，例如Java类和方法。持久代对GC没有显著的影响。持久代可以通过-XX:MaxPermSize=进行设置。

### 20.什么是类加载器，类加载器有哪些？

答案：深入探讨 Java 类加载器:https://www.ibm.com/developerworks/cn/java/j-lo-classloader/

**深入探讨 Java 类加载器**

成 富 2010 年 3 月 01 日发布

类加载器是 Java 语言的一个创新，也是 Java 语言流行的重要原因之一。它使得 Java 类可以被动态加载到 Java 虚拟机中并执行。类加载器从 JDK 1.0 就出现了，最初是为了满足 Java Applet 的需要而开发出来的。Java Applet 需要从远程下载 Java 类文件到浏览器中并执行。现在类加载器在 Web 容器和 OSGi 中得到了广泛的使用。一般来说，Java 应用的开发人员不需要直接同类加载器进行交互。Java 虚拟机默认的行为就已经足够满足大多数情况的需求了。不过如果遇到了需要与类加载器进行交互的情况，而对类加载器的机制又不是很了解的话，就很容易花大量的时间去调试 ClassNotFoundException和 NoClassDefFoundError等异常。本文将详细介绍 Java 的类加载器，帮助读者深刻理解 Java 语言中的这个重要概念。下面首先介绍一些相关的基本概念。

**类加载器基本概念**

顾名思义，类加载器（class loader）用来加载 Java 类到 Java 虚拟机中。一般来说，Java 虚拟机使用 Java 类的方式如下：Java 源程序（.java 文件）在经过 Java 编译器编译之后就被转换成 Java 字节代码（.class 文件）。类加载器负责读取 Java 字节代码，并转换成 java.lang.Class类的一个实例。每个这样的实例用来表示一个 Java 类。通过此实例的 newInstance()方法就可以创建出该类的一个对象。实际的情况可能更加复杂，比如 Java 字节代码可能是通过工具动态生成的，也可能是通过网络下载的。

基本上所有的类加载器都是 java.lang.ClassLoader类的一个实例。下面详细介绍这个 Java 类。

*java.lang.ClassLoader类介绍*

java.lang.ClassLoader类的基本职责就是根据一个指定的类的名称，找到或者生成其对应的字节代码，然后从这些字节代码中定义出一个 Java 类，即 java.lang.Class类的一个实例。除此之外，ClassLoader还负责加载 Java 应用所需的资源，如图像文件和配置文件等。不过本文只讨论其加载类的功能。为了完成加载类的这个职责，ClassLoader提供了一系列的方法，比较重要的方法如 [表 1](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#minor1.1)所示。关于这些方法的细节会在下面进行介绍。

表 1. ClassLoader 中与加载类相关的方法

|  |  |
| --- | --- |
| 方法 | 说明 |
| getParent() | 返回该类加载器的父类加载器。 |
| loadClass(String name) | 加载名称为 name的类，返回的结果是 java.lang.Class类的实例。 |
| findClass(String name) | 查找名称为 name的类，返回的结果是 java.lang.Class类的实例。 |
| findLoadedClass(String name) | 查找名称为 name的已经被加载过的类，返回的结果是 java.lang.Class类的实例。 |
| defineClass(String name, byte[] b, int off, int len) | 把字节数组 b中的内容转换成 Java 类，返回的结果是 java.lang.Class类的实例。这个方法被声明为 final的。 |
| resolveClass(Class c) | 链接指定的 Java 类。 |

对于 [表 1](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#minor1.1)中给出的方法，表示类名称的 name参数的值是类的二进制名称。需要注意的是内部类的表示，如 com.example.Sample$1和 com.example.Sample$Inner等表示方式。这些方法会在下面介绍类加载器的工作机制时，做进一步的说明。下面介绍类加载器的树状组织结构。

**类加载器的树状组织结构**

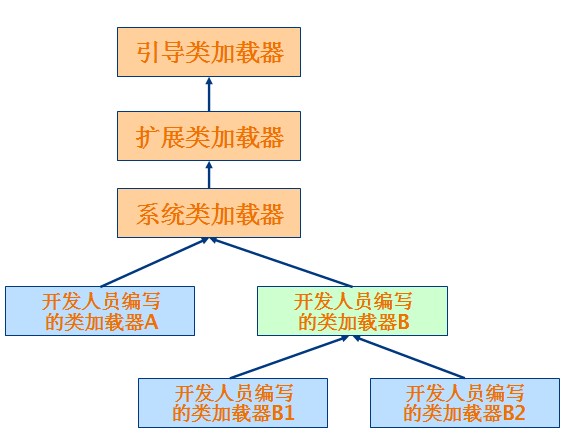
Java 中的类加载器大致可以分成两类，一类是系统提供的，另外一类则是由 Java 应用开发人员编写的。系统提供的类加载器主要有下面三个：

* 引导类加载器（bootstrap class loader）：它用来加载 Java 的核心库，是用原生代码来实现的，并不继承自 java.lang.ClassLoader。
* 扩展类加载器（extensions class loader）：它用来加载 Java 的扩展库。Java 虚拟机的实现会提供一个扩展库目录。该类加载器在此目录里面查找并加载 Java 类。
* 系统类加载器（system class loader）：它根据 Java 应用的类路径（CLASSPATH）来加载 Java 类。一般来说，Java 应用的类都是由它来完成加载的。可以通过 ClassLoader.getSystemClassLoader()来获取它。

除了系统提供的类加载器以外，开发人员可以通过继承 java.lang.ClassLoader类的方式实现自己的类加载器，以满足一些特殊的需求。

除了引导类加载器之外，所有的类加载器都有一个父类加载器。通过 [表 1](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#minor1.1)中给出的 getParent()方法可以得到。对于系统提供的类加载器来说，系统类加载器的父类加载器是扩展类加载器，而扩展类加载器的父类加载器是引导类加载器；对于开发人员编写的类加载器来说，其父类加载器是加载此类加载器 Java 类的类加载器。因为类加载器 Java 类如同其它的 Java 类一样，也是要由类加载器来加载的。一般来说，开发人员编写的类加载器的父类加载器是系统类加载器。类加载器通过这种方式组织起来，形成树状结构。树的根节点就是引导类加载器。[图 1](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#fig1)中给出了一个典型的类加载器树状组织结构示意图，其中的箭头指向的是父类加载器。

**图 1. 类加载器树状组织结构示意图**



[代码清单 1](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#code1)演示了类加载器的树状组织结构。

清单 1. 演示类加载器的树状组织结构

public class ClassLoaderTree {   
   
 public static void main(String[] args) {   
 ClassLoader loader = ClassLoaderTree.class.getClassLoader();   
 while (loader != null) {   
 System.out.println(loader.toString());   
 loader = loader.getParent();   
 }   
 }   
}

每个 Java 类都维护着一个指向定义它的类加载器的引用，通过 getClassLoader()方法就可以获取到此引用。[代码清单 1](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#code1)中通过递归调用 getParent()方法来输出全部的父类加载器。

[代码清单 1](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#code1)的运行结果如 [代码清单 2](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#code2)所示。

清单 2. 演示类加载器的树状组织结构的运行结果

sun.misc.Launcher$AppClassLoader@9304b1   
sun.misc.Launcher$ExtClassLoader@190d11

如 [代码清单 2](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#code2)所示，第一个输出的是 ClassLoaderTree类的类加载器，即系统类加载器。它是 sun.misc.Launcher$AppClassLoader类的实例；第二个输出的是扩展类加载器，是 sun.misc.Launcher$ExtClassLoader类的实例。需要注意的是这里并没有输出引导类加载器，这是由于有些 JDK 的实现对于父类加载器是引导类加载器的情况，getParent()方法返回 null。

在了解了类加载器的树状组织结构之后，下面介绍类加载器的代理模式。

**类加载器的代理模式**

类加载器在尝试自己去查找某个类的字节代码并定义它时，会先代理给其父类加载器，由父类加载器先去尝试加载这个类，依次类推。在介绍代理模式背后的动机之前，首先需要说明一下 Java 虚拟机是如何判定两个 Java 类是相同的。Java 虚拟机不仅要看类的全名是否相同，还要看加载此类的类加载器是否一样。只有两者都相同的情况，才认为两个类是相同的。即便是同样的字节代码，被不同的类加载器加载之后所得到的类，也是不同的。比如一个 Java 类 com.example.Sample，编译之后生成了字节代码文件 Sample.class。两个不同的类加载器 ClassLoaderA和 ClassLoaderB分别读取了这个 Sample.class文件，并定义出两个 java.lang.Class类的实例来表示这个类。这两个实例是不相同的。对于 Java 虚拟机来说，它们是不同的类。试图对这两个类的对象进行相互赋值，会抛出运行时异常 ClassCastException。下面通过示例来具体说明。[代码清单 3](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#code3)中给出了 Java 类 com.example.Sample。

清单 3. com.example.Sample 类

package com.example;   
   
public class Sample {   
 private Sample instance;   
   
 public void setSample(Object instance) {   
 this.instance = (Sample) instance;   
 }   
}

如 [代码清单 3](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#code3)所示，com.example.Sample类的方法 setSample接受一个 java.lang.Object类型的参数，并且会把该参数强制转换成 com.example.Sample类型。测试 Java 类是否相同的代码如 [代码清单 4](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#code4)所示。

清单 4. 测试 Java 类是否相同

public void testClassIdentity() {   
   String classDataRootPath = "C:\\workspace\\Classloader\\classData";   
   FileSystemClassLoader fscl1 = new FileSystemClassLoader(classDataRootPath);   
   FileSystemClassLoader fscl2 = new FileSystemClassLoader(classDataRootPath);   
   String className = "com.example.Sample";      
   try {   
       Class<?> class1 = fscl1.loadClass(className);   
       Object obj1 = class1.newInstance();   
       Class<?> class2 = fscl2.loadClass(className);   
       Object obj2 = class2.newInstance();   
       Method setSampleMethod = class1.getMethod("setSample", java.lang.Object.class);   
       setSampleMethod.invoke(obj1, obj2);   
   } catch (Exception e) {   
       e.printStackTrace();   
   }   
}

[代码清单 4](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#code4)中使用了类 FileSystemClassLoader的两个不同实例来分别加载类 com.example.Sample，得到了两个不同的 java.lang.Class的实例，接着通过 newInstance()方法分别生成了两个类的对象 obj1和 obj2，最后通过 Java 的反射 API 在对象 obj1上调用方法 setSample，试图把对象 obj2赋值给 obj1内部的 instance对象。[代码清单 4](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#code4)的运行结果如 [代码清单 5](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#code5)所示。

清单 5. 测试 Java 类是否相同的运行结果

java.lang.reflect.InvocationTargetException   
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)   
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:39)   
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:25)  
at java.lang.reflect.Method.invoke(Method.java:597)   
at classloader.ClassIdentity.testClassIdentity(ClassIdentity.java:26)   
at classloader.ClassIdentity.main(ClassIdentity.java:9)   
Caused by: java.lang.ClassCastException: com.example.Sample   
cannot be cast to com.example.Sample   
at com.example.Sample.setSample(Sample.java:7)   
... 6 more

从 [代码清单 5](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#code5)给出的运行结果可以看到，运行时抛出了 java.lang.ClassCastException异常。虽然两个对象 obj1和 obj2的类的名字相同，但是这两个类是由不同的类加载器实例来加载的，因此不被 Java 虚拟机认为是相同的。

了解了这一点之后，就可以理解代理模式的设计动机了。代理模式是为了保证 Java 核心库的类型安全。所有 Java 应用都至少需要引用 java.lang.Object类，也就是说在运行的时候，java.lang.Object这个类需要被加载到 Java 虚拟机中。如果这个加载过程由 Java 应用自己的类加载器来完成的话，很可能就存在多个版本的 java.lang.Object类，而且这些类之间是不兼容的。通过代理模式，对于 Java 核心库的类的加载工作由引导类加载器来统一完成，保证了 Java 应用所使用的都是同一个版本的 Java 核心库的类，是互相兼容的。

不同的类加载器为相同名称的类创建了额外的名称空间。相同名称的类可以并存在 Java 虚拟机中，只需要用不同的类加载器来加载它们即可。不同类加载器加载的类之间是不兼容的，这就相当于在 Java 虚拟机内部创建了一个个相互隔离的 Java 类空间。这种技术在许多框架中都被用到，后面会详细介绍。

下面具体介绍类加载器加载类的详细过程。

**加载类的过程**

在前面介绍类加载器的代理模式的时候，提到过类加载器会首先代理给其它类加载器来尝试加载某个类。这就意味着真正完成类的加载工作的类加载器和启动这个加载过程的类加载器，有可能不是同一个。真正完成类的加载工作是通过调用 defineClass来实现的；而启动类的加载过程是通过调用 loadClass来实现的。前者称为一个类的定义加载器（defining loader），后者称为初始加载器（initiating loader）。在 Java 虚拟机判断两个类是否相同的时候，使用的是类的定义加载器。也就是说，哪个类加载器启动类的加载过程并不重要，重要的是最终定义这个类的加载器。两种类加载器的关联之处在于：一个类的定义加载器是它引用的其它类的初始加载器。如类 com.example.Outer引用了类 com.example.Inner，则由类 com.example.Outer的定义加载器负责启动类 com.example.Inner的加载过程。

方法 loadClass()抛出的是 java.lang.ClassNotFoundException异常；方法 defineClass()抛出的是 java.lang.NoClassDefFoundError异常。

类加载器在成功加载某个类之后，会把得到的 java.lang.Class类的实例缓存起来。下次再请求加载该类的时候，类加载器会直接使用缓存的类的实例，而不会尝试再次加载。也就是说，对于一个类加载器实例来说，相同全名的类只加载一次，即 loadClass方法不会被重复调用。

下面讨论另外一种类加载器：线程上下文类加载器。

**线程上下文类加载器**

线程上下文类加载器（context class loader）是从 JDK 1.2 开始引入的。类 java.lang.Thread中的方法 getContextClassLoader()和 setContextClassLoader(ClassLoader cl)用来获取和设置线程的上下文类加载器。如果没有通过 setContextClassLoader(ClassLoader cl)方法进行设置的话，线程将继承其父线程的上下文类加载器。Java 应用运行的初始线程的上下文类加载器是系统类加载器。在线程中运行的代码可以通过此类加载器来加载类和资源。

前面提到的类加载器的代理模式并不能解决 Java 应用开发中会遇到的类加载器的全部问题。Java 提供了很多服务提供者接口（Service Provider Interface，SPI），允许第三方为这些接口提供实现。常见的 SPI 有 JDBC、JCE、JNDI、JAXP 和 JBI 等。这些 SPI 的接口由 Java 核心库来提供，如 JAXP 的 SPI 接口定义包含在 javax.xml.parsers包中。这些 SPI 的实现代码很可能是作为 Java 应用所依赖的 jar 包被包含进来，可以通过类路径（CLASSPATH）来找到，如实现了 JAXP SPI 的 [Apache Xerces](http://xerces.apache.org/)所包含的 jar 包。SPI 接口中的代码经常需要加载具体的实现类。如 JAXP 中的 javax.xml.parsers.DocumentBuilderFactory类中的 newInstance()方法用来生成一个新的 DocumentBuilderFactory的实例。这里的实例的真正的类是继承自 javax.xml.parsers.DocumentBuilderFactory，由 SPI 的实现所提供的。如在 Apache Xerces 中，实现的类是 org.apache.xerces.jaxp.DocumentBuilderFactoryImpl。而问题在于，SPI 的接口是 Java 核心库的一部分，是由引导类加载器来加载的；SPI 实现的 Java 类一般是由系统类加载器来加载的。引导类加载器是无法找到 SPI 的实现类的，因为它只加载 Java 的核心库。它也不能代理给系统类加载器，因为它是系统类加载器的祖先类加载器。也就是说，类加载器的代理模式无法解决这个问题。

线程上下文类加载器正好解决了这个问题。如果不做任何的设置，Java 应用的线程的上下文类加载器默认就是系统上下文类加载器。在 SPI 接口的代码中使用线程上下文类加载器，就可以成功的加载到 SPI 实现的类。线程上下文类加载器在很多 SPI 的实现中都会用到。

下面介绍另外一种加载类的方法：Class.forName。

**Class.forName**

Class.forName是一个静态方法，同样可以用来加载类。该方法有两种形式：Class.forName(String name, boolean initialize, ClassLoader loader)和 Class.forName(String className)。第一种形式的参数 name表示的是类的全名；initialize表示是否初始化类；loader表示加载时使用的类加载器。第二种形式则相当于设置了参数 initialize的值为 true，loader的值为当前类的类加载器。Class.forName的一个很常见的用法是在加载数据库驱动的时候。如 Class.forName("org.apache.derby.jdbc.EmbeddedDriver").newInstance()用来加载 Apache Derby 数据库的驱动。

在介绍完类加载器相关的基本概念之后，下面介绍如何开发自己的类加载器。

**开发自己的类加载器**

虽然在绝大多数情况下，系统默认提供的类加载器实现已经可以满足需求。但是在某些情况下，您还是需要为应用开发出自己的类加载器。比如您的应用通过网络来传输 Java 类的字节代码，为了保证安全性，这些字节代码经过了加密处理。这个时候您就需要自己的类加载器来从某个网络地址上读取加密后的字节代码，接着进行解密和验证，最后定义出要在 Java 虚拟机中运行的类来。下面将通过两个具体的实例来说明类加载器的开发。

**文件系统类加载器**

第一个类加载器用来加载存储在文件系统上的 Java 字节代码。完整的实现如 [代码清单 6](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#code6)所示。

清单 6. 文件系统类加载器

public class FileSystemClassLoader extends ClassLoader {   
   
 private String rootDir;   
   
 public FileSystemClassLoader(String rootDir) {   
 this.rootDir = rootDir;   
 }   
   
 protected Class<?> findClass(String name) throws ClassNotFoundException {   
 byte[] classData = getClassData(name);   
 if (classData == null) {   
 throw new ClassNotFoundException();   
 }   
 else {   
 return defineClass(name, classData, 0, classData.length);   
 }   
 }   
   
 private byte[] getClassData(String className) {   
 String path = classNameToPath(className);   
 try {   
 InputStream ins = new FileInputStream(path);   
 ByteArrayOutputStream baos = new ByteArrayOutputStream();   
 int bufferSize = 4096;   
 byte[] buffer = new byte[bufferSize];   
 int bytesNumRead = 0;   
 while ((bytesNumRead = ins.read(buffer)) != -1) {   
 baos.write(buffer, 0, bytesNumRead);   
 }   
 return baos.toByteArray();   
 } catch (IOException e) {   
 e.printStackTrace();   
 }   
 return null;   
 }   
   
 private String classNameToPath(String className) {   
 return rootDir + File.separatorChar   
 + className.replace('.', File.separatorChar) + ".class";   
 }   
}

如 [代码清单 6](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#code6)所示，类 FileSystemClassLoader继承自类 java.lang.ClassLoader。在 [表 1](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#minor1.1)中列出的 java.lang.ClassLoader类的常用方法中，一般来说，自己开发的类加载器只需要覆写 findClass(String name)方法即可。java.lang.ClassLoader类的方法 loadClass()封装了前面提到的代理模式的实现。该方法会首先调用 findLoadedClass()方法来检查该类是否已经被加载过；如果没有加载过的话，会调用父类加载器的 loadClass()方法来尝试加载该类；如果父类加载器无法加载该类的话，就调用 findClass()方法来查找该类。因此，为了保证类加载器都正确实现代理模式，在开发自己的类加载器时，最好不要覆写 loadClass()方法，而是覆写 findClass()方法。

类 FileSystemClassLoader的 findClass()方法首先根据类的全名在硬盘上查找类的字节代码文件（.class 文件），然后读取该文件内容，最后通过 defineClass()方法来把这些字节代码转换成 java.lang.Class类的实例。

**网络类加载器**

下面将通过一个网络类加载器来说明如何通过类加载器来实现组件的动态更新。即基本的场景是：Java 字节代码（.class）文件存放在服务器上，客户端通过网络的方式获取字节代码并执行。当有版本更新的时候，只需要替换掉服务器上保存的文件即可。通过类加载器可以比较简单的实现这种需求。

类 NetworkClassLoader负责通过网络下载 Java 类字节代码并定义出 Java 类。它的实现与 FileSystemClassLoader类似。在通过 NetworkClassLoader加载了某个版本的类之后，一般有两种做法来使用它。第一种做法是使用 Java 反射 API。另外一种做法是使用接口。需要注意的是，并不能直接在客户端代码中引用从服务器上下载的类，因为客户端代码的类加载器找不到这些类。使用 Java 反射 API 可以直接调用 Java 类的方法。而使用接口的做法则是把接口的类放在客户端中，从服务器上加载实现此接口的不同版本的类。在客户端通过相同的接口来使用这些实现类。网络类加载器的具体代码见 [下载](https://www.ibm.com/developerworks/cn/java/j-lo-classloader/#artdownload)。

在介绍完如何开发自己的类加载器之后，下面说明类加载器和 Web 容器的关系。

**类加载器与 Web 容器**

对于运行在 Java EE™容器中的 Web 应用来说，类加载器的实现方式与一般的 Java 应用有所不同。不同的 Web 容器的实现方式也会有所不同。以 Apache Tomcat 来说，每个 Web 应用都有一个对应的类加载器实例。该类加载器也使用代理模式，所不同的是它是首先尝试去加载某个类，如果找不到再代理给父类加载器。这与一般类加载器的顺序是相反的。这是 Java Servlet 规范中的推荐做法，其目的是使得 Web 应用自己的类的优先级高于 Web 容器提供的类。这种代理模式的一个例外是：Java 核心库的类是不在查找范围之内的。这也是为了保证 Java 核心库的类型安全。

绝大多数情况下，Web 应用的开发人员不需要考虑与类加载器相关的细节。下面给出几条简单的原则：

* 每个 Web 应用自己的 Java 类文件和使用的库的 jar 包，分别放在 WEB-INF/classes和 WEB-INF/lib目录下面。
* 多个应用共享的 Java 类文件和 jar 包，分别放在 Web 容器指定的由所有 Web 应用共享的目录下面。
* 当出现找不到类的错误时，检查当前类的类加载器和当前线程的上下文类加载器是否正确。

在介绍完类加载器与 Web 容器的关系之后，下面介绍它与 OSGi 的关系。

**类加载器与 OSGi**

OSGi™是 Java 上的动态模块系统。它为开发人员提供了面向服务和基于组件的运行环境，并提供标准的方式用来管理软件的生命周期。OSGi 已经被实现和部署在很多产品上，在开源社区也得到了广泛的支持。Eclipse 就是基于 OSGi 技术来构建的。

OSGi 中的每个模块（bundle）都包含 Java 包和类。模块可以声明它所依赖的需要导入（import）的其它模块的 Java 包和类（通过 Import-Package），也可以声明导出（export）自己的包和类，供其它模块使用（通过 Export-Package）。也就是说需要能够隐藏和共享一个模块中的某些 Java 包和类。这是通过 OSGi 特有的类加载器机制来实现的。OSGi 中的每个模块都有对应的一个类加载器。它负责加载模块自己包含的 Java 包和类。当它需要加载 Java 核心库的类时（以 java开头的包和类），它会代理给父类加载器（通常是启动类加载器）来完成。当它需要加载所导入的 Java 类时，它会代理给导出此 Java 类的模块来完成加载。模块也可以显式的声明某些 Java 包和类，必须由父类加载器来加载。只需要设置系统属性 org.osgi.framework.bootdelegation的值即可。

假设有两个模块 bundleA 和 bundleB，它们都有自己对应的类加载器 classLoaderA 和 classLoaderB。在 bundleA 中包含类 com.bundleA.Sample，并且该类被声明为导出的，也就是说可以被其它模块所使用的。bundleB 声明了导入 bundleA 提供的类 com.bundleA.Sample，并包含一个类 com.bundleB.NewSample继承自 com.bundleA.Sample。在 bundleB 启动的时候，其类加载器 classLoaderB 需要加载类 com.bundleB.NewSample，进而需要加载类 com.bundleA.Sample。由于 bundleB 声明了类 com.bundleA.Sample是导入的，classLoaderB 把加载类 com.bundleA.Sample的工作代理给导出该类的 bundleA 的类加载器 classLoaderA。classLoaderA 在其模块内部查找类 com.bundleA.Sample并定义它，所得到的类 com.bundleA.Sample实例就可以被所有声明导入了此类的模块使用。对于以 java开头的类，都是由父类加载器来加载的。如果声明了系统属性 org.osgi.framework.bootdelegation=com.example.core.\*，那么对于包 com.example.core中的类，都是由父类加载器来完成的。

OSGi 模块的这种类加载器结构，使得一个类的不同版本可以共存在 Java 虚拟机中，带来了很大的灵活性。不过它的这种不同，也会给开发人员带来一些麻烦，尤其当模块需要使用第三方提供的库的时候。下面提供几条比较好的建议：

* 如果一个类库只有一个模块使用，把该类库的 jar 包放在模块中，在 Bundle-ClassPath中指明即可。
* 如果一个类库被多个模块共用，可以为这个类库单独的创建一个模块，把其它模块需要用到的 Java 包声明为导出的。其它模块声明导入这些类。
* 如果类库提供了 SPI 接口，并且利用线程上下文类加载器来加载 SPI 实现的 Java 类，有可能会找不到 Java 类。如果出现了 NoClassDefFoundError异常，首先检查当前线程的上下文类加载器是否正确。通过 Thread.currentThread().getContextClassLoader()就可以得到该类加载器。该类加载器应该是该模块对应的类加载器。如果不是的话，可以首先通过 class.getClassLoader()来得到模块对应的类加载器，再通过 Thread.currentThread().setContextClassLoader()来设置当前线程的上下文类加载器。

**总结**

类加载器是 Java 语言的一个创新。它使得动态安装和更新软件组件成为可能。本文详细介绍了类加载器的相关话题，包括基本概念、代理模式、线程上下文类加载器、与 Web 容器和 OSGi 的关系等。开发人员在遇到 ClassNotFoundException和 NoClassDefFoundError等异常的时候，应该检查抛出异常的类的类加载器和当前线程的上下文类加载器，从中可以发现问题的所在。在开发自己的类加载器的时候，需要注意与已有的类加载器组织结构的协调。

**下载资源**

* [类加载器示例代码](http://www.ibm.com/developerworks/apps/download/index.jsp?contentid=470295&filename=classloader.zip&method=http&locale=zh_CN) (classloader.zip | 13 KB)

**相关主题**

* “[The Java Language Specification](http://java.sun.com/docs/books/jls/)”的第 12 章“[Execution](http://java.sun.com/docs/books/jls/third_edition/html/execution.html)”和“[The Java Virtual Machine Specification](http://java.sun.com/docs/books/jvms/)”的第 5 章“[Loading, Linking, and Initializing](http://java.sun.com/docs/books/jvms/second_edition/html/ConstantPool.doc.html)” 详细介绍了 Java 类的加载、链接和初始化。
* “[OSGi Service Platform Core Specification](http://www.osgi.org/Specifications/HomePage)”：OSGi 规范文档
* “[The Apache Tomcat 5.5 Servlet/JSP Container - Class Loader HOW-TO](http://tomcat.apache.org/tomcat-5.5-doc/class-loader-howto.html)”：详细介绍了 Tomcat 5.5 中的类加载器机制。
* [developerWorks Java 技术专区](http://www.ibm.com/developerworks/cn/java/)：数百篇关于 Java 编程各个方面的文章。

### 21.类加载器双亲委派模型机制？

答案：https://blog.csdn.net/zhangliangzi/article/details/51338291

**JVM类加载机制详解（二）类加载器与双亲委派模型**  
原创leeon\_l 最后发布于2016-05-07 21:19:35 阅读数 16790 收藏  
展开  
在上一篇JVM类加载机制详解

（一）JVM类加载过程中说到，类加载机制的第一个阶段加载做的工作有：

1、通过一个类的全限定名（包名与类名）来获取定义此类的二进制字节流（Class文件）。而获取的方式，可以通过jar包、war包、网络中获取、JSP文件生成等方式。

2、将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。这里只是转化了数据结构，并未合并数据。（方法区就是用来存放已被加载的类信息，常量，静态变量，编译后的代码的运行时内存区域）

3、在内存中生成一个代表这个类的java.lang.Class对象，作为方法区这个类的各种数据的访问入口。这个Class对象并没有规定是在Java堆内存中，它比较特殊，虽为对象，但存放在方法区中。

其中，实现第一个工作的代码块就被称为“**类加载器**”。

类加载器的作用不仅仅是实现类的加载，它还与类的的“相等”判定有关，关系着Java“相等”判定方法的返回结果，只有在满足如下三个**类、相等、判定条件**，才能判定两个类相等。

1、两个类来自同一个Class文件

2、两个类是由同一个虚拟机加载

3、两个类是由同一个类加载器加载

**Java“相等”判定相关方法：**

1、判断两个实例对象的引用是否指向内存中同一个实例对象，使用 Class对象的equals()方法，obj1.equals(obj2)；  
2、判断实例对象是否为某个类、接口或其子类、子接口的实例对象，使用Class对象的isInstance()方法，class.isInstance(obj)；

3、判断实例对象是否为某个类、接口的实例，使用instanceof关键字，obj instanceof class；  
4、判断一个类是否为另一个类本身或其子类、子接口，可以使用Class对象的isAssignableFrom()方法，class1.isAssignableFrom(class2)。

**JVM类加载器分类详解：**

1、Bootstrap ClassLoader：启动类加载器，也叫根类加载器，它负责加载Java的核心类库，加载如(%JAVA\_HOME%/lib)目录下的rt.jar（包含System、String这样的核心类）这样的核心类库。根类加载器非常特殊，它不是java.lang.ClassLoader的子类，它是JVM自身内部由C/C++实现的，并不是Java实现的。

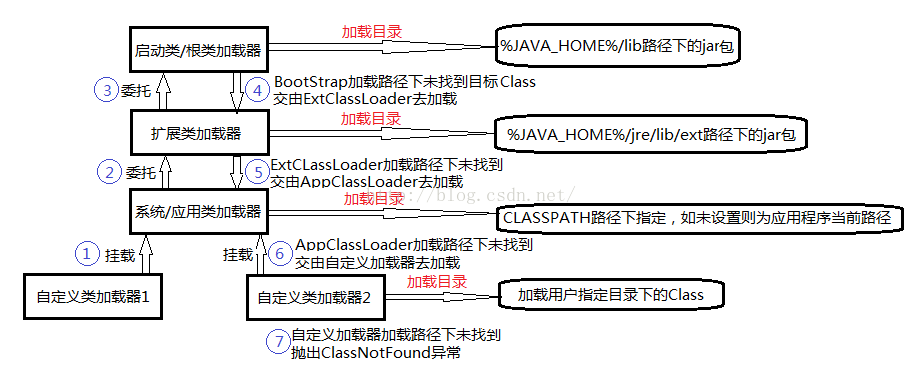
2、Extension ClassLoader：扩展类加载器，它负责加载扩展目录(%JAVA\_HOME%/jre/lib/ext)下的jar包，用户可以把自己开发的类打包成jar包放在这个目录下即可扩展核心类以外的新功能。

3、System ClassLoader\APP ClassLoader：系统类加载器或称为应用程序类加载器，是加载CLASSPATH环境变量所指定的jar包与类路径。一般来说，用户自定义的类就是由APP ClassLoader加载的。

**各种类加载器间关系**：以组合关系复用父类加载器的父子关系，注意，这里的父子关系并不是以继承关系实现的。

//验证类加载器与类加载器间的父子关系  
 public static void main(String[] args) throws Exception{  
 //获取系统/应用类加载器  
 ClassLoader appClassLoader = ClassLoader.getSystemClassLoader();  
 System.out.println("系统/应用类加载器：" + appClassLoader);  
 //获取系统/应用类加载器的父类加载器，得到扩展类加载器  
 ClassLoader extcClassLoader = appClassLoader.getParent();  
 System.out.println("扩展类加载器" + extcClassLoader);  
 System.out.println("扩展类加载器的加载路径：" + System.getProperty("java.ext.dirs"));  
 //获取扩展类加载器的父加载器，但因根类加载器并不是用Java实现的所以不能获取  
 System.out.println("扩展类的父类加载器：" + extcClassLoader.getParent());  
 }  
}  
**类加载器的双亲委派加载机制（重点）**：当一个类收到了类加载请求，他首先不会尝试自己去加载这个类，而是把这个请求委派给父类去完成，每一个层次类加载器都是如此，因此所有的加载请求都应该传送到启动类加载其中，只有当父类加载器反馈自己无法完成这个请求的时候（在它的加载路径下没有找到所需加载的Class），子类加载器才会尝试自己去加载。

这个过程如下图标号过程所示：



**双亲委派模型的源码实现：**

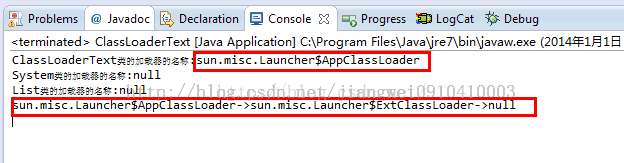
主要体现在ClassLoader的loadClass()方法中，思路很简单：先检查是否已经被加载过，若没有加载则调用父类加载器的loadClass()方法，若父类加载器为空则默认使用启动类加载器作为父类加载器。如果父类加载器加载失败，抛出ClassNotFoundException异常后，调用自己的findClass()方法进行加载。

public Class<?> loadClass(String name) throws ClassNotFoundException {  
 return loadClass(name, false);  
 }  
protected synchronized Class<?> loadClass(String name, boolean resolve)  
 throws ClassNotFoundException  
 {  
 // First, check if the class has already been loaded  
 Class c = findLoadedClass(name);  
 if (c == null) {  
 try {  
 if (parent != null) {  
 c = parent.loadClass(name, false);  
 } else {  
 c = findBootstrapClassOrNull(name);  
 }  
 } catch (ClassNotFoundException e) {  
 // ClassNotFoundException thrown if class not found  
 // from the non-null parent class loader  
 }  
 if (c == null) {  
 // If still not found, then invoke findClass in order  
 // to find the class.  
 c = findClass(name);  
 }  
 }  
 if (resolve) {  
 resolveClass(c);  
 }  
 return c;  
}

下面看一个**简单的双亲委派模型代码实例验证**：

public class ClassLoaderTest {  
 public static void main(String[] args){  
 //输出ClassLoaderText的类加载器名称  
 System.out.println("ClassLoaderText类的加载器的名称:"+ClassLoaderTest.class.getClassLoader().getClass().getName());  
 System.out.println("System类的加载器的名称:"+System.class.getClassLoader());  
 System.out.println("List类的加载器的名称:"+List.class.getClassLoader());  
 ClassLoader cl = ClassLoaderTest.class.getClassLoader();  
 while(cl != null){  
 System.out.print(cl.getClass().getName()+"->");  
 cl = cl.getParent();  
 }  
 System.out.println(cl);  
 }

输出结果为：

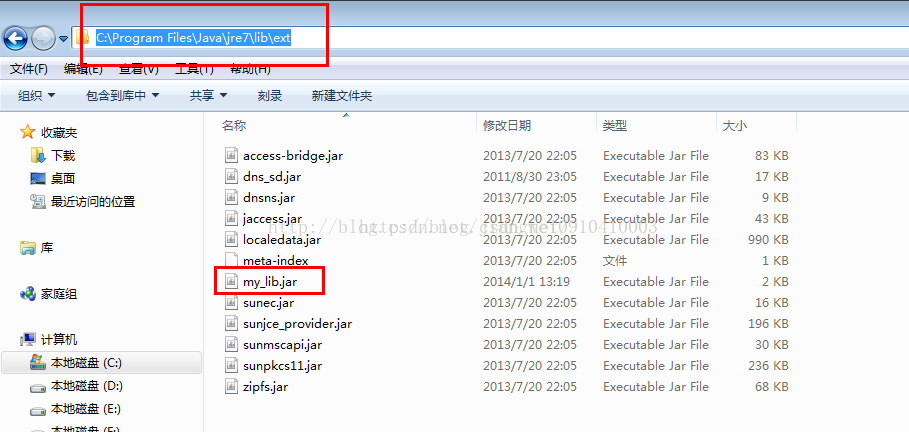


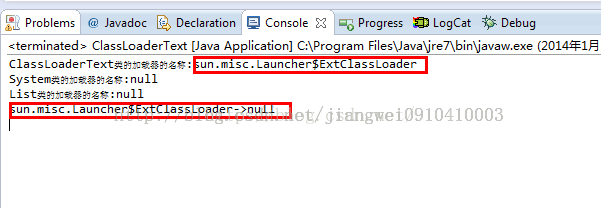
解释一下：  
1、ClassLoaderTest类是用户定义的类，位于CLASSPATH下，由系统/应用程序类加载器加载。

2、System类与List类都属于Java核心类，由祖先类启动类加载器加载，而启动类加载器是在JVM内部通过C/C++实现的，并不是Java，自然也就不能继承ClassLoader类，自然就不能输出其名称。

3、而箭头项代表的就是类加载的流程，层级委托，从祖先类加载器开始，直到系统/应用程序类加载器处才被加载。

那么我们做个测试，把类打成jar包，拷贝入%JAVA\_HOME%/jre/lib/ext目录下，再次运行ClassLoaderTest类





解释一下，因为类的Jar包放到了ExtClassLoader的加载目录下，所以在根目录找不到相应类后，在ExtClassLoader处就完成了类加载，而忽略了APPClassLoader阶段。

上一篇：JVM类加载机制详解（一）JVM类加载过程

（未完待续，下一篇：自定义的类加载器）

————————————————  
版权声明：本文为CSDN博主「leeon\_l」的原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/zhangliangzi/article/details/51338291

## 三、并发编程28题

### 1.Synchronized 用过吗，其原理是什么？

答案：https://blog.csdn.net/kuangsonghan/article/details/82178916

1、概述

2、synchronized锁的实现原理

3、synchronized的锁存放在哪里？

4、JavaSE 1.6中synchronized中锁的升级

**1、概述**

在并发编程中，synchronized一直是被使用非常频繁的，很多人会把它称为**重量级锁**，但是，JavaSE1.6对它有一个重大优化，在1.6中为了减少获得锁和释放锁带来的性能消耗而引入了**偏向锁和轻量级锁**。

我们知道Java中的每一个对象都可以作为锁，具体表现为三种形式：

1.对于普通同步方法，锁是当前实例对象

2.对于静态同步方法，锁是当前类的Class对象

3.对于同步代码块，锁是synchronized括号中配置的对象

我们知道，当一个线程试图访问同步代码块或方法时，它首先要得到锁，退出或者抛出异常时必须释放锁，那么我们所说的锁到底存放在哪里？锁中会存放什么信息呢？

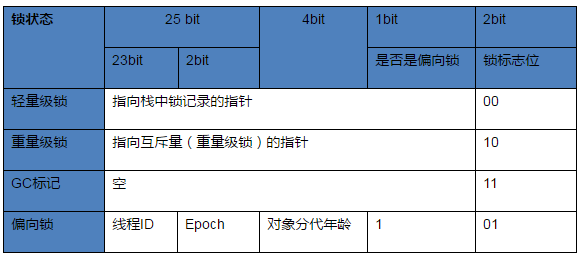
**2、synchronized锁的实现原理**  
在JVM规范中，*任何一个对象都与一个Monitor与之关联*，当且一个Monitor被持有后，它将处于锁定状态。Synchronized在JVM里的实现都是基于进入和退出Monitor对象来实现方法同步和代码块同步，虽然具体实现细节不一样，但是都可以通过成对的MonitorEnter和MonitorExit指令来实现。MonitorEnter指令插入在同步代码块的开始位置，当代码执行到该指令时，将会尝试获取该对象Monitor的所有权，即**尝试获得该对象的锁**，而monitorExit指令则插入在方法结束处和异常处，JVM保证每个MonitorEnter必须有对应的MonitorExit。

也就是说，在并发的环境下，假如有一个线程A要执行同步代码块，该同步块括号中配置的对象对应了一个Monitor，当该线程执行到MonitorEnter指令时，线程A将尝试获取该对象对应的Monitor，如果获取到了，那么线程A就获得了该对象锁。

**3、synchronized的锁存放在哪里？**  
synchronized用的锁是存放在Java对象的对象头里的，在理解这个之前，我们有必要先了解什么是对象头。我们知道对于Java对象，对象的组成一般分为三个部分，**对象头、对象实例数据、以及对齐填充**。

对象头里面又分为两（如果是数组是三个，第三个部分存放着数组的长度）部分，**第一个部分存放着对象运行时的数据**：如hashcode、GC分代年龄、锁状态标志、线程持有的锁、偏向线程ID、偏向时间戳等信息。这部分信息，我们称它为**“mark word”**。第二个部分**存放着对象类型指针**。那么我们所说的synchronized锁是存在对象头的第一个部分mark word中。

总结一下上面的话，synchronized使用的锁是存放在Java对象头里面，具体位置是对象头里面的MarkWord，MarkWord里默认数据是存储对象的HashCode等信息，但是会随着对象的运行改变而发生变化，不同的锁状态对应着不同的记录存储方式，可能值如下所示：



**无锁状态 ： 对象的HashCode + 对象分代年龄 + 状态位001**

**4、JavaSE 1.6中synchronized中锁的升级**  
1.6中为了减少获得锁和释放锁带来的性能消耗，引入了“偏向锁”和“轻量级锁”，加上1.6之前的两种状态，在1.6及以后锁就有4中状态，分别是：**无锁状态、偏向锁状态、轻量级锁状态、重量级锁状态**。这几个状态会随着竞争情况逐渐升级。锁可以升级而不能降级。

**4.1偏向锁**  
为什么要加入偏向锁？因为在大多数情况下，锁不仅存在多个线程竞争，而且总是由同一个线程多次获得，为了让线程获得锁的代价更低而引入偏向锁。引入偏向锁是为了在多线程竞争的情况下尽量减少不必要的轻量级锁执行路径，因为**轻量级锁的获取及释放依赖多次CAS原子指令，而偏向锁只需要在置换ThreadID的时候依赖一次CAS原子指令（**由于一旦出现多线程竞争的情况就必须撤销偏向锁，所以偏向锁的撤销操作的性能损耗必须小于节省下来的CAS原子指令的性能消耗）。***轻量级锁是为了在线程交替执行同步块时提高性能，而偏向锁则是在只有一个线程执行同步块时进一步提高性能***。

**4.1.1偏向锁的获得**  
**当一个线程访问同步块并获得锁时，会在对象头和栈帧中的锁记录中存储锁偏向的线程ID，以后该线程在进入和退出同步块时不需要进行CAS操作来加锁和解锁，只需要简单地测试一下对象头的Mark word里是否存储着指向当前线程的偏向锁。如果测试成功，表示线程已经获得了锁，如果测试失败，则需要再测试下Mark word中偏向锁的标识是否设置为1（表示当前是偏向锁），如果没有设置，则需要使用CAS进行竞争锁。如果设置了则尝试使用CAS将对象的偏向锁指向当前线程。**

根据上面的描述，整理一下偏向锁获取过程：

（1）访问Mark Word中偏向锁的标识是否设置成1，锁标志位是否为01——确认为可偏向状态。

（2）如果为可偏向状态，则测试线程ID是否指向当前线程，

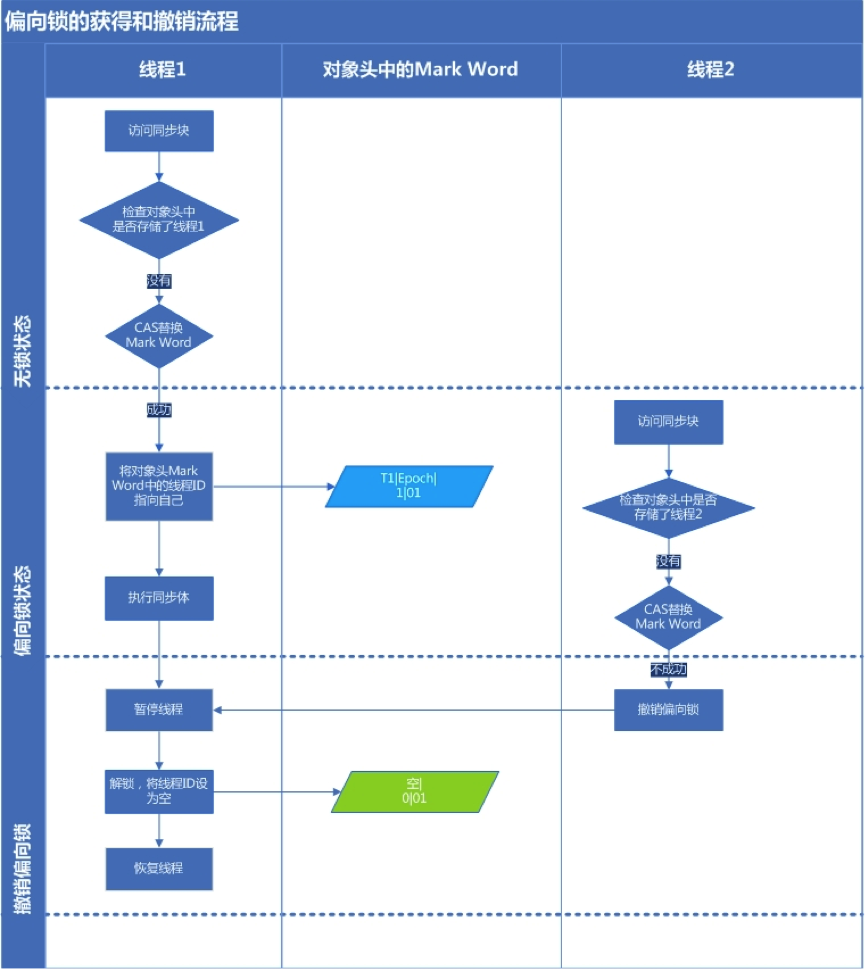
如果是，进入步骤(5)，否则进入步骤（3）。

（3）如果线程ID并未指向当前线程，则通过CAS操作竞争锁。如果竞争成功，则将Mark Word中线程ID设置为当前线程ID，然后执行（5）；如果竞争失败，执行（4）。

（4）如果CAS获取偏向锁失败，则表示有竞争。当到达全局安全点（safepoint）时获得偏向锁的线程被挂起，偏向锁升级为轻量级锁，然后被阻塞在安全点的线程继续往下执行同步代码。

（5）执行同步代码。

**4.1.2偏向锁的撤销**  
 偏向锁的撤销在上述第4步骤中有提到。偏向锁只有遇到其他线程尝试竞争偏向锁时，持有偏向锁的线程才会释放锁，线程不会主动去释放偏向锁。偏向锁的撤销，需要等待全局安全点（在这个时间点上没有字节码正在执行），它会首先暂停拥有偏向锁的线程，判断锁对象是否处于被锁定状态，撤销偏向锁后恢复到未锁定（标志位为“01”）或轻量级锁（标志位为“00”）的状态。



**4.2轻量级锁**  
**4.2.1为什么要引入轻量级锁**  
轻量级锁实现的背后基于这样一种假设，即在真实的情况下我们程序中的大部分同步代码一般都处于无锁竞争状态（即单线程执行环境），在无锁竞争的情况下完全可以避免调用操作系统层面的**重量级互斥锁**，取而代之的是在monitorenter和monitorexit中**只需要依靠一条CAS原子指令就可以完成锁的获取及释放**。当存在锁竞争的情况下，执行CAS指令失败的线程将调用操作系统互斥锁进入到阻塞状态，当锁被释放的时候被唤醒。

需要注意的是，“轻量级”是相对于使用操作系统互斥量来实现的传统锁而言的。但是，首先需要强调一点的是，轻量级锁并不是用来代替重量级锁的，它的本意是在没有多线程竞争的前提下，减少传统的重量级锁使用产生的性能消耗。在解释轻量级锁的执行过程之前，先明白一点，轻量级锁所适应的场景是线程交替执行同步块的情况，如果存在同一时间访问同一锁的情况，就会导致轻量级锁膨胀为重量级锁。

**4.2.2轻量级锁的加锁过程**  
 （1）在代码进入同步块的时候，如果同步对象锁状态为无锁状态（锁标志位为“01”状态，是否为偏向锁为“0”），虚拟机首先将在当前线程的栈帧中建立一个名为锁记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝，官方称之为 **Displaced Mark Word**。

（2）拷贝对象头中的Mark Word复制到锁记录中。

（3）拷贝成功后，虚拟机将使用CAS操作尝试将对象的Mark Word更新为指向Lock Record的指针，并将Lock record里的owner指针指向object mark word。如果更新成功，则执行步骤（3），否则执行步骤（4）。

（4）如果这个更新动作成功了，那么这个线程就拥有了该对象的锁，并且对象Mark Word的锁标志位设置为“00”，即表示此对象处于轻量级锁定状态。

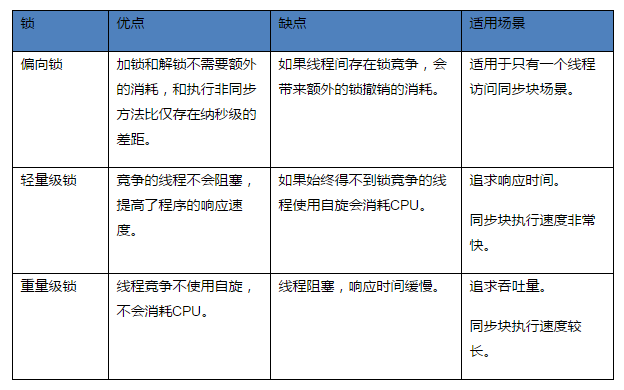
（5）如果这个更新操作失败了，虚拟机首先会检查对象的Mark Word是否指向当前线程的栈帧，如果是就说明当前线程已经拥有了这个对象的锁，那就可以直接进入同步块继续执行。否则说明多个线程竞争锁，轻量级锁就要膨胀为重量级锁，锁标志的状态值变为“10”，Mark Word中存储的就是指向重量级锁（互斥量）的指针，后面等待锁的线程也要进入阻塞状态。 而当前线程便尝试使用自旋来获取锁，自旋就是为了不让线程阻塞，而采用循环去获取锁的过程。

**4.2.3轻量级锁的解锁过程**  
 （1）通过CAS操作尝试把线程中复制的Displaced Mark Word对象替换当前的Mark Word。

（2）如果替换成功，整个同步过程就完成了。

（3）如果替换失败，说明有其他线程尝试过获取该锁（此时锁已膨胀），那就要在释放锁的同时，唤醒被挂起的线程。

**42.4不同锁的比较**



————————————————  
版权声明：本文为CSDN博主「kingdoooom」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/kuangsonghan/article/details/82178916

### 2.你刚才提到获取对象的锁，这个“锁”到底是什么？如何确定对象的锁？

答案：https://www.cnblogs.com/huashuohehe/p/10665733.html

1、概述

一、线程的先来后到

一直对多线程有点迷糊，不知道什么是锁对象，锁的到底是个什么玩意，最近看了下面这位大佬的解释，感觉很到位

以下是转载：https://www.cnblogs.com/coprince/p/5848614.html

**1、概述**

更新：在一次和一位专家的交谈中，他对一下代码能否能够成功同步，给予了否定的答案， 他的理由是”以构造函数的成员变量作为synchronized的锁，在多线程的情况下，每一个线程都持有自己私有变量的锁，这个锁的地址能一样吗？怎么可能成功同步？这是最错误的写法！“，哎呀妈，说实话，当时我都被惊了一下，都开始怀疑这个代码的正确性了。

我亲自测试了一下本文中的代码（测试代码在资源和硬盘中已保存）。结论是可以同步的， 然后我回忆了一下我们当时谈话的出发点：**线程安全**（代码和本文中的代码类似，就是让打印的代码能够线程安全）。所以我在想，他是否把线程同步和线程安全混 为一谈了? 让多线程按启动顺序去执行那段循环打印还是保证线程间不相互影响（当某线程在打印的过程中，其他的线程不可进行打印）即可？

运行测试代码后，可以清晰的看到，在以那个字符串成员变量为同步锁的代码 的输出中，每个线程都是完成了自己的打印而没有出现穿插。所以我认为这个，就可以视作线程安全了。至于他，是否想的是让线程按顺序执行？如此，可以用 join方法。但是在真实的应用中，没必要线程按照顺序执行，线程启动后，可能被阻塞或者其他的原因的wait。

**2、synchronized特性**

**一段synchronized的代码被一个线程执行之前，他要先拿到执行这段代码的权限，在**[**Java**](http://lib.csdn.net/base/17)**里 边就是拿到某个同步对象的锁（一个对象只有一把锁）； 如果这个时候同步对象的锁被其他线程拿走了，他（这个线程）就只能等了（线程阻塞在锁池等待队列 中）。 取到锁后，他就开始执行同步代码(被synchronized修饰的代码）；线程执行完同步代码后马上就把锁还给同步对象，其他在锁池中等待的某 个线程就可以拿到锁执行同步代码了。这样就保证了同步代码在统一时刻只有一个线程在执行。**

**众所周知，在Java多线程编程中，一个非常重要的方面就是线程的同步问题。**

**关于线程的同步，一般有以下解决方法：**

**1. 在需要同步的方法的方法签名中加入synchronized关键字。**

**2. 使用synchronized块对需要进行同步的代码段进行同步。**

**3. 使用JDK 5中提供的java.util.concurrent.lock包中的Lock对象。**

另外，为了解决多个线程对同一变量进行访问时可能发生的安全性问题，我们不仅可以采用同步机制，更可以通过JDK 1.2中加入的ThreadLocal来保证更好的并发性。

本篇中，将详细的讨论Java多线程同步机制

**3、Java多线程同步机制**

**3.1线程的先来后到**

我们来举一个Dirty的例子：某餐厅的卫生间很小，几乎只能容纳一个人如厕。为了保证不受干扰，如厕的人进入卫生间，就要锁上房门。我们可以把卫生间想 象成是共享的资源，而众多需要如厕的人可以被视作多个线程。假如卫生间当前有人占用，那么其他人必须等待，直到这个人如厕完毕，打开房门走出来为止。这就 好比多个线程共享一个资源的时候，是一定要分出先来后到的。

有人说：那如果我没有这道门会怎样呢？让两个线程相互竞争，谁抢先了，谁就可以先干活，这样多好阿？但是我们知道：如果厕所没有门的话，如厕的人一起涌向 厕所，那么必然会发生争执，正常的如厕步骤就会被打乱，很有可能会发生意想不到的结果，例如某些人可能只好被迫在不正确的地方施肥……

正是因为有这道门，任何一个单独进入如厕的人都可以顺利的完成他们的如厕过程，而不会被干扰，甚至发生以外的结果。这就是说，如厕的时候要讲究先来后到。

那么在Java 多线程程序当中，当多个线程竞争同一个资源的时候，如何能够保证他们不会产生“打架”的情况呢？有人说是使用同步机制。没错，像上面这个例子，就是典型的 同步案例，一旦第一位开始如厕，则第二位必须等待第一位结束，才能开始他的如厕过程。一个线程，一旦进入某一过程，必须等待正常的返回，并退出这一过程， 下一个线程才能开始这个过程。这里，最关键的就是卫生间的门。其实，卫生间的门担任的是资源锁的角色，只要如厕的人锁上门，就相当于获得了这个锁，而当他 打开锁出来以后，就相当于释放了这个锁。

也就是说，多线程的线程同步机制实际上是**靠锁的概念来控制的**。那么在Java程序当中，锁是如何体现的呢？

**3.2锁的概念**

让我们从JVM的角度来看看锁这个概念：

在Java程序运行时环境中，JVM需要对两类线程共享的数据进行协调：  
1）保存在堆中的实例变量  
2）保存在方法区中的类变量

这两类数据是被所有线程共享的。  
（程序不需要协调保存在Java 栈当中的数据。因为这些数据是属于拥有该栈的线程所私有的。）

**3.2锁和监视器**

在java虚拟机中，每个对象和类在逻辑上都是和一个监视器相关联的。  
对于对象来说，相关联的监视器保护对象的实例变量。

对于类来说，监视器保护类的类变量。

（如果一个对象没有实例变量，或者一个类没有变量，相关联的监视器就什么也不监视。）   
为了实现监视器的排他性监视能力，java虚拟机为每一个对象和类都关联一个锁。代表任何时候只允许一个线程拥有的特权。线程访问实例变量或者类变量不需锁。

但是如果线程获取了锁，那么在它释放这个锁之前，就没有其他线程可以获取同样数据的锁了。（锁住一个对象就是获取对象相关联的监视器）

类锁实际上用对象锁来实现。当虚拟机装载一个class文件的时候，它就会创建一个java.lang.Class类的实例。当锁住一个对象的时候，实际上锁住的是那个类的Class对象。

一个线程可以多次对同一个对象上锁。对于每一个对象，java虚拟机维护一个加锁计数器，线程每获得一次该对象，计数器就加1，每释放一次，计数器就减 1，当计数器值为0时，锁就被完全释放了。

java编程人员不需要自己动手加锁，对象锁是java虚拟机内部使用的。

在java程序中，只需要使用synchronized块或者synchronized方法就可以标志一个监视区域。当每次进入一个监视区域时，java 虚拟机都会自动锁上对象或者类。

看到这里，我想你们一定都疲劳了吧？o(∩\_∩)o...哈哈。让我们休息一下，但是在这之前，请你们一定要记着：  
***当一个有限的资源被多个线程共享的时候，为了保证对共享资源的互斥访问，我们一定要给他们排出一个先来后到。而要做到这一点，对象锁在这里起着非常重要的作用。***\*

在上一篇中，我们讲到了多线程是如何处理共享资源的，以及保证他们对资源进行互斥访问所依赖的重要机制：对象锁。

本篇中，我们来看一看传统的同步实现方式以及这背后的原理。

4、实现原理

很多人都知道，在Java多线程编程中，有一个重要的关键字，synchronized。但是很多人看到这个东西会感到困惑：“都说同步机制是通过对象锁来实现的，但是这么一个关键字，我也看不出来Java程序锁住了哪个对象阿？“

public class ThreadTest extends Thread {  
 private int threadNo;  
   
 public ThreadTest(int threadNo) {  
 this.threadNo = threadNo;  
 }  
   
 public static void main(String[] args) throws Exception {  
 for (int i = 1; i < 10; i++) {  
 new ThreadTest(i).start();  
 Thread.sleep(1);  
 }  
 }  
   
 @Override  
 public synchronized void run() {  
 for (int i = 1; i < 10; i++) {  
 System.out.println("No." + threadNo + ":" + i);  
 }  
 }  
}

这个程序其实就是让10个线程在控制台上数数，从1数到9999。理想情况下，我们希望看到一个线程数完，然后才是另一个线程开始数数。但是这个程序的执行过程告诉我们，这些线程还是乱糟糟的在那里抢着报数，丝毫没有任何规矩可言。  
 但是细心的读者注意到：run方法还是加了一个synchronized关键字的，按道理说，这些线程应该可以一个接一个的执行这个run方法才对阿。  
 但是通过上一篇中，我们提到的，对于一个成员方法加synchronized关键字，这实际上是以这个成员方法所在的对象本身作为对象锁。在本例中，就是 以ThreadTest类的一个具体对象，也就是该线程自身作为对象锁的。一共十个线程，每个线程持有自己 线程对象的那个对象锁。这必然不能产生同步的效果。换句话说，**如果要对这些线程进行同步，那么这些线程所持有的对象锁应当是共享且唯一的！**

我们来看下面的例子：

public class ThreadTest2 extends Thread {  
 private int threadNo;  
 private String lock;  
  
 public ThreadTest2(int threadNo, String lock) {  
 this.threadNo = threadNo;  
 this.lock = lock;  
 }  
  
 public static void main(String[] args) throws Exception {  
 String lock = new String("lock");  
 for (int i = 1; i < 10; i++) {  
 new ThreadTest2(i, lock).start();  
 Thread.sleep(1);  
 }  
 }  
  
 public void run() {  
 synchronized (lock) {  
 for (int i = 1; i < 10; i++) {  
 System.out.println("No." + threadNo + ":" + i);  
 }  
 }  
 }  
}

我们注意到，该程序通过在main方法启动10个线程之前，创建了一个String类型的对象。并通过ThreadTest2的构造函数，将这个对象赋值 给每一个ThreadTest2线程对象中的私有变量lock。根据Java方法的传值特点，我们知道，这些线程的lock变量实际上指向的是堆内存中的 同一个区域，即存放main函数中的lock变量的区域。  
 程序将原来run方法前的synchronized关键字去掉，换用了run方法中的一个synchronized块来实现。这个同步块的对象锁，就是 main方法中创建的那个String对象。换句话说，**他们指向的是同一个String类型的对象，对象锁是共享且唯一的！**

于是，我们看到了预期的效果：10个线程不再是争先恐后的报数了，而是一个接一个的报数。

再来看下面的例子：

public class ThreadTest3 extends Thread {  
  
 private int threadNo;  
 //private String lock;  
  
 public ThreadTest3(int threadNo) {  
 this.threadNo = threadNo;  
 }  
  
 public static void main(String[] args) throws Exception {  
 for (int i = 1; i < 20; i++) {  
 new ThreadTest3(i).start();  
 Thread.sleep(1);  
 }  
 }  
  
 public static synchronized void abc(int threadNo) {  
 for (int i = 1; i < 10; i++) {  
 System.out.println("No." + threadNo + ":" + i);  
 }  
 }  
  
 public void run() {  
 abc(threadNo);  
 }  
}

细心的读者发现了：这段代码没有使用main方法中创建的String对象作为这10个线程的线程锁。而是通过在run方法中调用本线程中一 个静态的同步 方法abc而实现了线程的同步。我想看到这里，你们应该很困惑：这里synchronized静态方法是用什么来做对象锁的呢？

我们知道，对于同步静态方法，对象锁就是该静态放发所在的类的Class实例，由于在JVM中，所有被加载的类都有唯一的类对象，具体到本例，就是唯一的 ThreadTest3.class对象。不管我们创建了该类的多少实例，但是它的类实例仍然是一个！

这样我们就知道了：

**1、对于同步的方法或者代码块来说，必须获得对象锁才能够进入同步方法或者代码块进行操作；**

**2、如果采用method级别的同步，则对象锁即为method所在的对象，如果是静态方法，对象锁即指method所在的Class对象(唯一)；**

**3、对于代码块，对象锁即指synchronized(abc)中的abc；**

**4、因为第一种情况，对象锁即为每一个线程对象，因此有多个，所以同步失效，第二种共用同一个对象锁lock，因此同步生效，第三个因为是**  
**static因此对象锁为ThreadTest3的class 对象，因此同步生效。**

如上述正确，则同步有两种方式，同步块和同步方法（为什么没有wait和notify？这个我会在补充章节中做出阐述）

如果是同步代码块，则对象锁需要编程人员自己指定，一般有些代码为synchronized(this)只有在单态模式才生效；  
（本类的实例有且只有一个）

如果是同步方法，**则分静态和非静态两种** 。

静态方法则一定会同步，非静态方法需在单例模式才生效，推荐用静态方法(不用担心是否单例)。

所以说，在Java多线程编程中，最常见的synchronized关键字实际上是依靠对象锁的机制来实现线程同步的。  
我们似乎可以听到synchronized在向我们说：“给我**一把** 锁，我能创造一个规矩”。

### 3.什么是可重入性，为什么说 Synchronized 是可重入锁？

答案：https://blog.csdn.net/weixin\_37627441/article/details/103163069

什么是可重入性，为什么说 Synchronized 是可重入锁？

可重入性是锁的一个基本要求，是为了解决自己锁死自己的情况。  
一个类中的同步方法调用另一个同步方法，假如 Synchronized 不支持重入，进入 method2 方法时当前线程获得锁，method2 方法里面执行 method1 时当前线程又要去尝试获取锁，对 Synchronized 来说，可重入性是显而易见的，刚才提到，在执行 monitorenter 指令时，如果这个对象没有锁定，或者当前线程已经拥这时如果不支持重入，它就要等释放，把自己阻塞，导致自己锁死自己。

有了这个对象的锁（而不是已拥有了锁则不能继续获取），就把锁的计数器 +1，其实本质上就通过这种方式实现了可重入性。

————————————————  
版权声明：本文为CSDN博主「离散小维」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/weixin\_37627441/article/details/103163069

答案二：https://blog.csdn.net/hw120219/article/details/94165926

“可重入锁”的概念：自己可以再次获取自己的内部锁。比如一个线程获得了某个对象的锁，此时锁还没释放，当再次获取这个对象锁的时候还可以获取。（如果不能获取，就会造成死锁）。

可重入锁也支持在父子类的继承中。比如父类一个方法，子类有另外一个方法，这两个方法都是synchronized修饰，子类方法可以调用父类的方法。

synchronized其它的特性  
1.运行出现异常，自动释放锁。  
2.同步不具有继承性，父类的方法用synchronized修饰后，子类的重写的父类方法也要手动加synchronize的关键字  
3.如果synchronized修饰的方法执行效率过低，可以用同步代码块来提高效率，即synchronized不在修饰方法，只把关键的数据放入同步代码块。此时该方法一半同步一半异步。  
4.synchronized(this)锁定的是当前对象，synchronized修饰方法锁定的也是当前对象。  
5.同步代码块synchronized(),括号中可以是任意的字符串也可以是对象（如果是对象，即使对象的属性改变，方法仍然是同步的。但如果是字符串，字符串改变，则方法不能保证同步），这样做的好处是在执行同步代码块的过程中，可以同时调用其他的同步方法（如果业务有这个需求的话）。  
6.一个类内部有两个内部类，内部类中各有一个synchronized修饰的方法，那么synchronized锁定的对象是类的对象，不是内部类的对象。

可重入锁的例子如下：

package com.hanw.testKafka;  
  
public class Test1 {  
 public static void main(String[] args) {  
 MyThread t = new MyThread();  
 t.start();  
 }  
}  
class Service {  
 synchronized public void service1() {  
 System.out.println("service1");  
 service2();  
 }  
   
 synchronized public void service2() {  
 System.out.println("service2");  
 service3();  
 }  
   
 synchronized public void service3() {  
 System.out.println("service3");  
 }  
}  
  
class MyThread extends Thread {  
 @Override  
 public void run() {  
 Service service = new Service();  
 service.service1();  
 }  
}

执行结果 为

service1  
service2  
service3

————————————————  
版权声明：本文为CSDN博主「hw120219」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/hw120219/article/details/94165926

### 4.JVM 对 Java 的原生锁做了哪些优化？

答案：https://blog.csdn.net/weixin\_37627441/article/details/103163629

JVM 对 Java 的原生锁做了哪些优化？

离散小维 2019-11-20 16:51:49 541 收藏 展开  
**1、自旋锁**  
在 Java 6 之前，Monitor 的实现**完全依赖**底层操作系统的**互斥锁**来实现。  
由于 Java 层面的线程与操作系统的原生线程有映射关系，如果要将一个线程进行阻塞或唤起都需要操作系统的协助，这就需要从用户态切换到内核态来执行，这种切换代价十分昂贵，很耗处理器时间，现代 JDK 中做了大量的优化。  
一种优化是**使用自旋锁**，即在把线程进行阻塞操作之前先让线程自旋等待一段时间，可能在等待期间其他线程已经解锁，这时就无需再让线程执行阻塞操作，避免了用户态到内核态的切换。

Java 虚拟机的开发工程师们在分析过大量数据后发现：共享数据的锁定状态一般只会持续很短的一段时间，为了这段时间去挂起和恢复线程其实并不值得。

自旋锁在 JDK 1.4 中引入，在 JDK 1.6 中默认开启。

自旋等待虽然避免了线程切换的开销，但自旋的线程要占用处理器时间的，所以若锁被占用的时间很短，自旋等待的效果就会非常好，反之锁被占用的时间很长，那么自旋的线程只会白白消耗 CPU 资源。

因此自旋等待的时间必须要有一定的限度，超过限定的次数仍然没有成功获得锁，就应当挂起（阻塞）线程了。*自旋次数的默认值是 10 次*。，用户可以通过使用参数-XX:PreBlockSpin来更改。

**2、自适应自旋锁**  
在 JDK 1.6 中引入了自适应自旋锁。

自适应意味着自旋的时间不再固定了，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。

如果在同一个锁对象上，自旋等待刚刚成功获得过锁，并且持有锁的线程正在运行中，那么虚拟机就会认为这次自旋也很有可能再次成功，进而它将允许自旋等待持续相对更长的时间，比如100个循环。

如果对于某个锁，自旋很少成功获得过，那在以后要获取这个锁时将可能省略掉自旋过程，以避免浪费处理器资源。

**3、锁消除**  
在动态编译同步块的时候，JIT 编译器可以借助一种被称为**逃逸分析（Escape Analysis）**的技术来判断同步块所使用的锁对象是否只能够被一个线程访问而没有被发布到其他线程。从而取消对这部分代码的同步。

锁消除：指虚拟机即时编译器在运行时，对一些代码上要求同步，但被检测到不可能存在共享数据竞争的锁进行消除。主要根据逃逸分析。  
程序员怎么会在明知道不存在数据竞争的情况下使用同步呢？很多不是程序员自己加入的。

**4、锁粗化**  
当 JIT 编译器发现一系列连续的操作都对同一个对象反复加锁和解锁，甚至加锁操作出现在循环体中的时候，会将加锁同步的范围扩散（粗化）到整个操作序列的外部。

在编写代码的时候，总是推荐将同步块的作用范围（锁粒度）限制得尽量小（只在共享数据的实际作用域中才进行同步），这样是为了使得需要同步的操作数量尽可能变小，如果存在锁竞争，那等待锁的线程可以尽快的拿到锁。

锁粒度：不要锁住一些无关的代码。锁粗化：可以一次执行完的不要多次加锁执行

————————————————  
版权声明：本文为CSDN博主「离散小维」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/weixin\_37627441/article/details/103163629

### 5.为什么说 Synchronized 是非公平锁？

答案：https://blog.csdn.net/wab719591157/article/details/86221109

synchronized 是公平锁吗？可以重入吗？详细的来说说 synchronized

1、公平锁：

获取不到锁的时候，会自动加入队列，等待线程释放后，**队列的第一个线程获取锁**

2、非公平锁:

获取不到锁的时候，会自动加入队列，**等待线程释放锁后所有等待的线程同时去竞争**

非 公 平 主 要 表 现 在 获 取 锁 的 行 为 上 ， 并 非 是 按 照 申 请 锁 的 时 间 前 后 给 等  
待 线 程 分 配 锁 的 ， 每 当 锁 被 释 放 后 ， 任 何 一 个 线 程 都 有 机 会 竞 争 到 锁 ，  
这 样 做 的 目 的 是 为 了 提 高 执 行 性 能 ， 缺 点 是 可 能 会 产 生 线 程 饥 饿 现 象 。  
————————————————  
版权声明：本文为CSDN博主「mischen520」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/miachen520/article/details/104874058

3、什么是可重入？

同一个线程可以反复获取锁多次，然后需要释放多次

回答标题问题：**synchronized 是非公平锁，可以重入。**

附录：

在来看几个问题：

1、 synchronized 加在 static 修饰的 方法上锁的是哪个对象？

答：锁的是 Class 对象

2、synchronized(this) 锁的是哪个对象？

答：锁的是当前对象的实例也就是 new 出来的对象

3、synchronized() 锁的是哪个对象？

答：同 synchronized(this) 锁的也是当前对象的实例

4、synchronized(lock) 锁的是哪个对象？

答：锁的是 lock 对象。

5、同一个对象里面 有 static 方法加了 synchronized 还有一个普通方法也加了 synchronized 如果这个时候有2个线程，一个先获取 了 static 方法上的锁，另外一个线程可以在 第一个线程释放锁之前获取 普通方法上的锁吗？反过来呢？

答：可以的，因为这是 2 把锁，2 个线程分别获取2把锁，没有任何问题。

————————————————

版权声明：本文为CSDN博主「坚强一点」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。

原文链接：[synchronized 是公平锁吗？可以重入吗？详细的来说说 synchronized](https://link.zhihu.com/?target=https%3A//blog.csdn.net/wab719591157/article/details/86221109)

### 6.什么是锁消除和锁粗化？

答案：原创 [我咋这么优秀呢](https://me.csdn.net/zj57356498318)

锁消除和锁粗化 https://blog.csdn.net/zj57356498318/article/details/103364024

**锁消除和锁粗化**

锁消除和锁粗化是虚拟机对低效的锁操作而进行的一个优化。

**锁消除**

简单来说，锁消除就是虚拟机根据一个对象是否真正存在同步情况，若不存在同步情况，则对该对象的访问无需经过加锁解锁的操作。（比如说程序员使用了StringBuffer的append方法，因为append方法需要判断对象是否被占用，而如果代码不存在锁的竞争，那么这部分的性能消耗是无意义的。于是虚拟机在即时编译的时候就会将上面代码进行优化，也就是锁消除。）

那么虚拟机如何判断不存在同步情况呢？

通过逃逸分析。可见下面代码

public static String createStringBuffer(String str1, String str2) {  
 StringBuffer sBuf = new StringBuffer();  
 sBuf.append(str1);// append方法是同步操作  
 sBuf.append(str2);  
 return sBuf.toString();  
 }   
   
@Override  
public synchronized String toString() {  
 if (toStringCache == null) {  
 toStringCache = Arrays.copyOfRange(value, 0, count);  
 }  
 return new String(toStringCache, true);  
}

可以看到sBuf对象使用的范围仅仅只在这个方法栈中，因为return回去的对象是一个新的String对象。  
也就是说sBuf对象是不会逃逸出去从而被其他线程访问到，那就可以把它们当作栈上的数据对待，认为它们是线程私有，同步锁无需进行。

锁粗化

public static StringBuffer createStringBuffer(String str1, String str2) {  
 StringBuffer sBuf = new StringBuffer();  
 sBuf.append(str1);// append方法是同步操作  
 sBuf.append(str2);  
 sBuf.append("abc");  
 return sBuf;  
}

频繁的对sBuf进行加锁、解锁，会造成性能上的损失。如果虚拟机探测到有一串零碎操作都是对同一对象加锁，将会把加锁同步的范围扩展到整个操作序列的外部，也就是在第一个和最后一个append操作之后。

用下面代码对锁粗化进行说明

for(int i=0;i<100000;i++){   
 synchronized(this){   
 do();   
}   
  
//在锁粗化之后运行逻辑如下列代码  
synchronized(this){   
 for(int i=0;i<100000;i++){   
 do();   
}

————————————————  
版权声明：本文为CSDN博主「我咋这么优秀呢」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/zj57356498318/article/details/103364024

另一篇参考：https://blog.csdn.net/qq\_26222859/article/details/80546917

**锁粗化**

通常情况下，为了保证多线程间的有效并发，会要求每个线程持有锁的时间尽可能短，但是某些情况下，一个程序对同一个锁不间断、高频地请求、同步与释放，会消耗掉一定的系统资源，因为锁的请求、同步与释放本身会带来性能损耗，这样高频的锁请求就反而不利于系统性能的优化了，虽然单次同步操作的时间可能很短。**锁粗化就是告诉我们任何事情都有个度，有些情况下我们反而希望把很多次锁的请求合并成一个请求，以降低短时间内大量锁请求、同步、释放带来的性能损耗。**

一种极端的情况如下：

public void doSomethingMethod(){   
 synchronized(lock){   
 //do some thing   
 }   
 //这是还有一些代码，做其它不需要同步的工作，但能很快执行完毕   
 synchronized(lock){   
 //do other thing   
 }  
}

上面的代码是有两块需要同步操作的，但在这两块需要同步操作的代码之间，需要做一些其它的工作，而这些工作只会花费很少的时间，那么我们就可以把这些工作代码放入锁内，将两个同步代码块合并成一个，以降低多次锁请求、同步、释放带来的系统性能消耗，合并后的代码如下:

public void doSomethingMethod(){ //进行锁粗化：整合成一次锁请求、同步、释放 synchronized(lock){ //do some thing //做其它不需要同步但能很快执行完的工作 //do other thing }}

注意：这样做是有前提的，就是中间不需要同步的代码能够很快速地完成，如果不需要同步的代码需要花很长时间，就会导致同步块的执行需要花费很长的时间，这样做也就不合理了。

另一种需要锁粗化的极端的情况是：

for(int i=0;i<size;i++){ synchronized(lock){ }}

上面代码每次循环都会进行锁的请求、同步与释放，看起来貌似没什么问题，且在jdk内部会对这类代码锁的请求做一些优化，但是还不如把加锁代码写在循环体的外面，这样一次锁的请求就可以达到我们的要求，除非有特殊的需要：循环需要花很长时间，但其它线程等不起，要给它们执行的机会。

锁粗化后的代码如下：

synchronized(lock){ for(int i=0;i<size;i++){ }}

**锁消除**

锁消除是发生在编译器级别的一种锁优化方式。  
有时候我们写的代码完全不需要加锁，却执行了加锁操作。

比如，StringBuffer类的append操作：

@Overridepublic synchronized StringBuffer append(String str) { toStringCache = null; super.append(str); return this;}

从源码中可以看出，append方法用了synchronized关键词，它是线程安全的。但我们可能仅在线程内部把StringBuffer当作局部变量使用：

package com.leeib.thread;public class Demo { public static void main(String[] args) { long start = System.currentTimeMillis(); int size = 10000; for (int i = 0; i < size; i++) { createStringBuffer("Hyes", "为分享技术而生"); } long timeCost = System.currentTimeMillis() - start; System.out.println("createStringBuffer:" + timeCost + " ms"); } public static String createStringBuffer(String str1, String str2) { StringBuffer sBuf = new StringBuffer(); sBuf.append(str1);// append方法是同步操作 sBuf.append(str2); return sBuf.toString(); }}

代码中createStringBuffer方法中的局部对象sBuf，就只在该方法内的作用域有效，不同线程同时调用createStringBuffer()方法时，都会创建不同的sBuf对象，因此此时的append操作若是使用同步操作，就是白白浪费的系统资源。

这时我们可以通过编译器将其优化，将锁消除，前提是java必须运行在server模式（server模式会比client模式作更多的优化），同时必须开启逃逸分析:

-server -XX:+DoEscapeAnalysis -XX:+EliminateLocks

其中+DoEscapeAnalysis表示开启逃逸分析，+EliminateLocks表示锁消除。

逃逸分析：比如上面的代码，它要看sBuf是否可能逃出它的作用域？如果将sBuf作为方法的返回值进行返回，那么它在方法外部可能被当作一个全局对象使用，就有可能发生线程安全问题，这时就可以说sBuf这个对象发生逃逸了，因而不应将append操作的锁消除，但我们上面的代码没有发生锁逃逸，锁消除就可以带来一定的性能提升。

转载自：[java锁优化的5个方法](http://www.hao124.net/article/70)

====================================================

锁削除是指虚拟机即时编译器在运行时，对一些代码上要求同步，但是被检测到不可能存在共享数据竞争的锁进行削除。锁削除的主要判定依据来源于逃逸分析的数据支持，如果判断到一段代码中，在堆上的所有数据都不会逃逸出去被其他线程访问到，那就可以把它们当作栈上数据对待，认为它们是线程私有的，同步加锁自然就无须进行。   
 也许读者会有疑问，变量是否逃逸，对于虚拟机来说需要使用数据流分析来确定，但是程序员自己应该是很清楚的，怎么会在明知道不存在数据争用的情况下要求同步呢？答案是有许多同步措施并不是程序员自己加入的，同步的代码在Java程序中的普遍程度也许超过了大部分读者的想象。我们来看看下面代码清单13-6中的例子，这段非常简单的代码仅仅是输出三个字符串相加的结果，无论是源码字面上还是程序语义上都没有同步。

**代码清单 13-6 一段看起来没有同步的代码**

Java代码 [fig:](http://icyfenix.iteye.com/blog/1018932)

1. **public** String concatString(String s1, String s2, String s3) {
2. **return** s1 + s2 + s3;
3. }

我们也知道，由于String是一个不可变的类，对字符串的连接操作总是通过生成新的String对象来进行的，因此Javac编译器会对String连接做自动优化。在JDK 1.5之前，会转化为StringBuffer对象的连续append()操作，在JDK 1.5及以后的版本中，会转化为StringBuilder对象的连续append()操作。即代码清单13-6中的代码可能会变成代码清单13-7的样子 。

**代码清单 13-7 Javac转化后的字符串连接操作**

Java代码 [fig:](http://icyfenix.iteye.com/blog/1018932)

1. **public** String concatString(String s1, String s2, String s3) {
2. StringBuffer sb = **new** StringBuffer();
3. sb.append(s1);
4. sb.append(s2);
5. sb.append(s3);
6. **return** sb.toString();
7. }

（注1：实事求是地说，既然谈到锁削除与逃逸分析，那虚拟机就不可能是JDK 1.5之前的版本，所以实际上会转化为非线程安全的StringBuilder来完成字符串拼接，并不会加锁。但是这也不影响笔者用这个例子证明Java对象中同步的普遍性。）

现在大家还认为这段代码没有涉及同步吗？每个StringBuffer.append()方法中都有一个同步块，锁就是sb对象。虚拟机观察变量sb，很快就会发现它的动态作用域被限制在concatString()方法内部。也就是sb的所有引用永远不会“逃逸”到concatString()方法之外，其他线程无法访问到它，所以这里虽然有锁，但是可以被安全地削除掉，在即时编译之后，这段代码就会忽略掉所有的同步而直接执行了。

=====================================

This is an example of manual "[lock coarsening](http://www.oracle.com/technetwork/java/6-performance-137236.html#2.1.2)" and may have been done to get a performance boost.

Consider these two snippets:

StringBuffer b = new StringBuffer();for(int i = 0 ; i < 100; i++){ b.append(i);}

versus:

StringBuffer b = new StringBuffer();synchronized(b){ for(int i = 0 ; i < 100; i++){ b.append(i); }}

In the first case, the StringBuffer must acquire and release a lock 100 times (because append is a synchronized method), whereas in the second case, the lock is acquired and released only once. This can give you a performance boost and is probably why the author did it. In some cases, the compiler can perform this [lock coarsening](http://www.oracle.com/technetwork/java/6-performance-137236.html#2.1.2) for you (but not around looping constructs because you could end up holding a lock for long periods of time).

By the way, the compiler can detect that an object is not "escaping" from a method and so remove acquiring and releasing locks on the object altogether (lock elision) since no other thread can access the object anyway. A lot of work has been done on this in [JDK7](http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html).

转载自：[Synchronization on the local variables](https://stackoverflow.com/questions/6123024/synchronization-on-the-local-variables)

===================================

转载自：[Java锁消除](https://blog.csdn.net/winwill2012/article/details/46376679)

非商业转载，可联系本人删除

概述

锁消除是Java虚拟机在JIT编译是，通过对运行上下文的扫描，去除不可能存在共享资源竞争的锁，通过锁消除，可以节省毫无意义的请求锁时间。

实验

看如下代码：

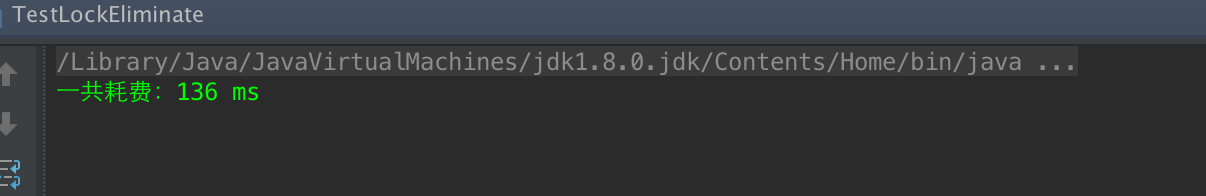
package com.winwill.lock; /\*\* \* @author qifuguang \* @date 15/6/5 14:11 \*/public class TestLockEliminate { public static String getString(String s1, String s2) { StringBuffer sb = new StringBuffer(); sb.append(s1); sb.append(s2); return sb.toString(); } public static void main(String[] args) { long tsStart = System.currentTimeMillis(); for (int i = 0; i < 1000000; i++) { getString("TestLockEliminate ", "Suffix"); } System.out.println("一共耗费：" + (System.currentTimeMillis() - tsStart) + " ms"); }}

getString()方法中的StringBuffer数以函数内部的局部变量，进作用于方法内部，不可能逃逸出该方法，因此他就不可能被多个线程同时访问，也就没有资源的竞争，但是StringBuffer的append操作却需要执行同步操作:

@Override public synchronized StringBuffer append(String str) { toStringCache = null; super.append(str); return this; }

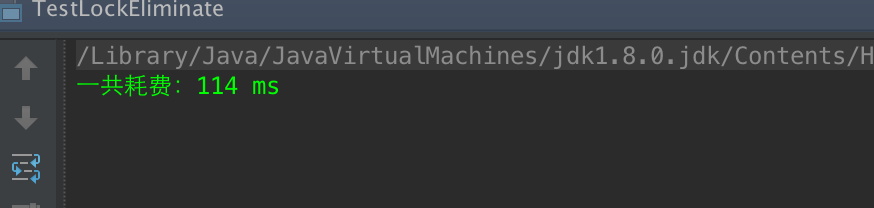
逃逸分析和锁消除分别可以使用参数-XX:+DoEscapeAnalysis和-XX:+EliminateLocks(锁消除必须在-server模式下)开启。使用如下参数运行上面的程序：

-XX:+DoEscapeAnalysis -XX:-EliminateLocks

得到如下结果：   


使用如下命令运行程序：

-XX:+DoEscapeAnalysis -XX:+EliminateLocks

得到如下结果：   


结论

由上面的例子可以看出，关闭了锁消除之后，StringBuffer每次append都会进行锁的申请，浪费了不必要的时间，开启锁消除之后性能得到了提高。

### 7.为什么说 Synchronized 是一个悲观锁？乐观锁的实现原理又是什么？什么是 CAS，它有什么特性？

答案：https://blog.csdn.net/qq\_36998190/article/details/103310223

Synchronized 显然是一个悲观锁，因为它的并发策略是悲观的:

不管是否会产生竞争，任何的数据操作都必须要加锁、用户态核心态转 换、维护锁计数器和检查是否有被阻塞的线程需要被唤醒等操作。

随着硬件指令集的发展，我们可以使用基于冲突检测的乐观并发策略。 先进行操作，如果没有其他线程征用数据，那操作就成功了; 如果共享数据有征用，产生了冲突，那就再进行其他的补偿措施。这种 乐观的并发策略的许多实现不需要线程挂起，所以被称为非阻塞同步。 乐观锁的核心算法是 CAS(Compareand Swap，比较并交换)，它涉 及到三个操作数:内存值、预期值、新值。当且仅当预期值和内存值相 等时才将内存值修改为新值。 这样处理的逻辑是，首先检查某块内存的值是否跟之前我读取时的一 样，如不一样则表示期间此内存值已经被别的线程更改过，舍弃本次操 作，否则说明期间没有其他线程对此内存值操作，可以把新值设置给此 块内存。  
CAS 具有原子性，它的原子性由 CPU 硬件指令实现保证，即使用 JNI 调用 Native 方法调用由 C++ 编写的硬件级别指令，JDK 中提 供了 Unsafe 类执行这些操作。  
————————————————  
版权声明：本文为CSDN博主「qq*36998190」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。*  
*原文链接：https://blog.csdn.net/qq*36998190/article/details/103310223

另一篇参考：乐观锁的原理

[Java乐观锁的实现原理（案例）](https://www.cnblogs.com/baxianhua/p/9378031.html)

简要说明:

表设计时，需要往表里加一个**version字段**。每次查询时，**查出带有version的数据记录**，**更新数据时，判断数据库里对应id的记录的version是否和查出的version相同。若相同，则更新数据并把版本号+1；若不同，则说明，该数据发送并发，被别的线程使用了，进行递归操作，再次执行递归方法，知道成功更新数据为止**

简单说说乐观锁。乐观锁是相对于悲观锁而言。悲观锁认为，这个线程，发生并发的可能性极大，线程冲突几率大，比较悲观。一般用synchronized实现，保证每次操作数据不会冲突。乐观锁认为，线程冲突可能性小，比较乐观，直接去操作数据，如果发现数据已经被更改（通过**版本号**控制），则不更新数据，再次去重复 所需操作，知道没有冲突（使用**递归**算法）。

因为乐观锁使用递归+版本号控制 实现，所以，如果线程冲突几率大，使用乐观锁会重复很多次操作（包括查询数据库），尤其是递归部分逻辑复杂，耗时和耗性能，是低效不合适的，应考虑使用悲观锁。

乐观锁悲观锁的选择：

乐观锁：并发冲突几率小，对应模块递归操作简单 时使用

悲观锁：并发几率大，对应模块操作复杂 时使用

案例一

/\*\*  
  
 \* 自动派单  
  
 \* 只查出一条 返回list只是为了和查询接口统一  
  
 \* 视频审核订单不派送  
  
 \* @param paramMap  
  
 \* @return  
  
 \*/  
  
public List<AutomaticAssignDto> automaticAssign(Map<String, Object> paramMap){  
  
 //派送规则  
  
 String changeSortSet = RedisCacheUtil.getValue(CACHE\_TYPE.APP, "changeSortSet");  
  
 if (StringUtils.isBlank(changeSortSet)) {  
  
 changeSortSet = customerManager.getDictionaryByCode("changeSortSet");  
  
 if (StringUtils.isNotBlank(changeSortSet)) {  
  
 RedisCacheUtil.addValue(CACHE\_TYPE.APP, "changeSortSet", changeSortSet,30,TimeUnit.DAYS);  
  
 } else {  
  
 changeSortSet = ConstantsUtil.AssignRule.FIFO; // 默认先进先审  
  
 }  
  
 }  
  
 AutomaticAssignDto automaticAssignDto = new AutomaticAssignDto();  
  
 automaticAssignDto.setChangeSortSet(changeSortSet);  
  
 automaticAssignDto.setUserTeam(CommonUtils.getValue(paramMap, "userTeam"));  
  
 List<AutomaticAssignDto> waitCheckList = automaticAssignMybatisDao.automaticAssignOrder(automaticAssignDto);  
  
 if(waitCheckList != null && waitCheckList.size()>0){  
  
 automaticAssignDto = waitCheckList.get(0);  
  
 automaticAssignDto.setSendStatus(ConstantsUtil.SendStatus.SEND);  
  
 automaticAssignDto.setBindTime(new Date());  
  
 automaticAssignDto.setUserId(Long.parseLong(paramMap.get("userId").toString()) );  
  
 int sum = automaticAssignMybatisDao.bindAutomaticAssignInfo(automaticAssignDto);  
  
 if(sum == 1){  
  
 　　return waitCheckList;  
  
 }else{  
  
 //已被更新 则再次获取  
  
 return automaticAssign(paramMap);  
  
 }  
  
 }else{  
  
 return null;  
  
 }  
  
}

学习自 https://blog.csdn.net/zhangdehua678/article/details/79594212

**案例二**

package what21.thread.lock;  
  
   
  
public class OptimLockMain {  
  
   
  
 // 文件版本号  
  
 static int version = 1;  
  
 // 操作文件  
  
 static String file = "d://IT小奋斗.txt";  
  
   
  
 /\*\*  
  
 \* 获取版本号  
  
 \*  
  
 \* @return  
  
 \*/  
  
 public static int getVersion(){  
  
 return version;  
  
 }  
  
   
  
 /\*\*  
  
 \* 更新版本号  
  
 \*/  
  
 public static void updateVersion(){  
  
 version+=1;  
  
 }  
  
   
  
 /\*\*  
  
 \* @param args  
  
 \*/  
  
 public static void main(String[] args) {  
  
 for(int i=1;i<=5;i++){  
  
 new OptimThread(String.valueOf(i),getVersion(),file).start();  
  
 }  
  
 }  
  
   
  
}  
  
package what21.thread.lock;  
  
import java.io.BufferedReader;  
import java.io.File;  
import java.io.FileReader;  
import java.io.FileWriter;  
import java.io.IOException;  
  
   
  
public class OptimThread extends Thread {  
  
   
  
 // 文件版本号  
  
 public int version;  
  
 // 文件  
  
 public String file;  
  
   
  
 public OptimThread(String name,int version,String file){  
  
 this.setName(name);  
  
 this.version = version;  
  
 this.file = file;  
  
 }  
  
   
  
 public void run() {  
  
 // 1. 读取文件  
  
 String text = read(file);  
  
 println("线程"+ getName() + "，文件版本号为：" + OptimLockMain.getVersion());  
  
 println("线程"+ getName() + "，版本号为：" + getVersion());  
  
 // 2. 写入文件  
  
 if(OptimLockMain.getVersion() == getVersion()){  
  
 println("线程" + getName() + "，版本号为：" + version + "，正在执行");  
  
 // 文件操作，这里用synchronized就相当于文件锁  
  
 // 如果是数据库，相当于表锁或者行锁  
  
 synchronized(OptimThread.class){  
  
 if(OptimLockMain.getVersion() == this.version){  
  
 // 写入操作  
  
 write(file, text);  
  
 // 更新文件版本号  
  
 OptimLockMain.updateVersion();  
  
 return ;  
  
 }  
  
 }  
  
 }  
  
 // 3. 版本号不正确的线程，需要重新读取，重新执行  
  
 println("线程"+ getName() + "，文件版本号为：" + OptimLockMain.getVersion());  
  
 println("线程"+ getName() + "，版本号为：" + getVersion());  
  
 System.err.println("线程"+ getName() + "，需要重新执行。");  
  
 }  
  
   
  
 /\*\*  
  
 \* @return  
  
 \*/  
  
 private int getVersion(){  
  
 return this.version;  
  
 }  
  
   
  
 /\*\*  
  
 \* 写入数据  
  
 \*  
  
 \* @param file  
  
 \* @param text  
  
 \*/  
  
 public static void write(String file,String text){  
  
 try {  
  
 FileWriter fw = new FileWriter(file,false);  
  
 fw.write(text + "\r\n");  
  
 fw.flush();  
  
 fw.close();  
  
 } catch (IOException e) {  
  
 e.printStackTrace();  
  
 }  
  
 }  
  
   
  
 /\*\*  
  
 \* 读取数据  
  
 \*  
  
 \* @param file  
  
 \* @return  
  
 \*/  
  
 public static String read(String file){  
  
 StringBuilder sb = new StringBuilder();  
  
 try {  
  
 File rFile = new File(file);  
  
 if(!rFile.exists()){  
  
 rFile.createNewFile();  
  
 }  
  
 FileReader fr = new FileReader(rFile);  
  
 BufferedReader br = new BufferedReader(fr);  
  
 String r = null;  
  
 while((r=br.readLine())!=null){  
  
 sb.append(r).append("\r\n");  
  
 }  
  
 br.close();  
  
 fr.close();  
  
 } catch (IOException e) {  
  
 e.printStackTrace();  
  
 }  
  
 return sb.toString();  
  
 }  
  
   
  
 /\*\*  
  
 \* @param content  
  
 \*/  
  
 public static void println(String content){  
  
 System.out.println(content);  
  
 }  
  
   
  
}

学习自https://blog.csdn.net/qq897958555/article/details/79337064

### 8.乐观锁一定就是好的吗？

乐观锁和悲观锁的区别（最全面的分析） https://blog.csdn.net/xiaobing\_122613/article/details/78672708

悲观锁(Pessimistic Lock), 顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

乐观锁(Optimistic Lock), 顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于write\_condition机制的其实都是提供的乐观锁。

两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果经常产生冲突，上层应用会不断的进行retry，这样反倒是降低了性能，所以这种情况下用悲观锁就比较合适。

本质上，数据库的乐观锁做法和悲观锁做法主要就是解决下面假设的场景，避免丢失更新问题：  
 一个比较清楚的场景  
 下面这个假设的实际场景可以比较清楚的帮助我们理解这个问题：  
 假设当当网上用户下单买了本书，这时数据库中有条订单号为001的订单，其中有个status字段是’有效’,表示该订单是有效的；  
 后台管理人员查询到这条001的订单，并且看到状态是有效的  
 用户发现下单的时候下错了，于是撤销订单，假设运行这样一条SQL: update order*table set status = ‘取消’ where order*id = 001;  
后台管理人员由于在b这步看到状态有效的，这时，虽然用户在c这步已经撤销了订单，可是管理人员并未刷新界面，看到的订单状态还是有效的，于是点击”发货”按钮，将该订单发到物流部门，同时运行类似如下SQL，将订单状态改成已发货:update order*table set status = ‘已发货’ where order*id = 001

观点1：只有冲突非常严重的系统才需要悲观锁；“所有悲观锁的做法都适合于状态被修改的概率比较高的情况，具体是否合适则需要根据实际情况判断。”，表达的也是这个意思，不过说法不够准确；的确，之所以用悲观锁就是因为两个用户更新同一条数据的概率高，也就是冲突比较严重的情况下，所以才用悲观锁。

观点2：最后提交前作一次select for update检查，然后再提交update也是一种乐观锁的做法，的确，这符合传统乐观锁的做法，就是到最后再去检查。但是wiki在解释悲观锁的做法的时候，’It is not appropriate for use in web application development.’， 现在已经很少有悲观锁的做法了，所以我自己将这种二次检查的做法也归为悲观锁的变种，因为这在所有乐观锁里面，做法和悲观锁是最接近的，都是先select for update，然后update

在实际应用中我们在更新数据的时候，更严谨的做法是带上更新前的“状态”，如

update order*table set status = ‘取消’ where order*id = 001 and status = ‘待支付’ and ..........;

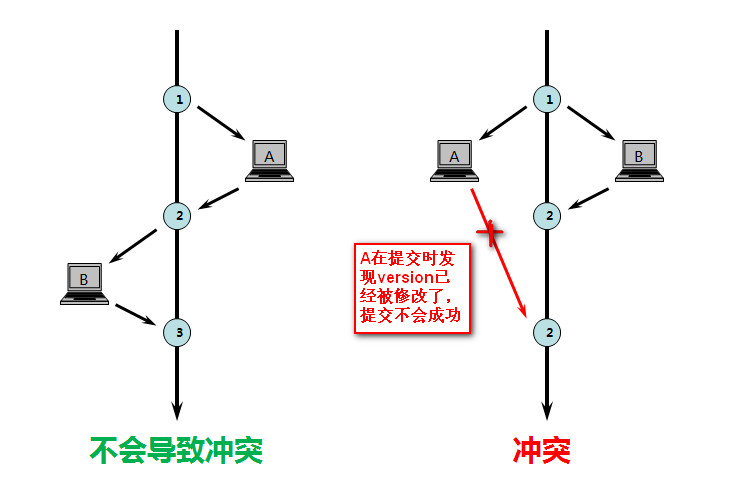
update order*table set status = ‘已发货’ where order*id = 001 and status = ‘已支付’ and ..........;

然后在业务逻辑代码里判断更新的记录数，为0说明数据库已经被更新了，否则是正常的。

悲观锁(Pessimistic Lock), 顾名思义，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

乐观锁(Optimistic Lock), 顾名思义，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制。乐观锁适用于多读的应用类型，这样可以提高吞吐量，像数据库如果提供类似于write\_condition机制的其实都是提供的乐观锁。

两种锁各有优缺点，不可认为一种好于另一种，像乐观锁适用于写比较少的情况下，即冲突真的很少发生的时候，这样可以省去了锁的开销，加大了系统的整个吞吐量。但如果经常产生冲突，上层应用会不断的进行retry，这样反倒是降低了性能，所以这种情况下用悲观锁就比较合适。

最近因为在工作中需要，学习了乐观锁与悲观锁的相关知识，这里我通过这篇文章，把我自己对这两个“锁家”兄弟理解记录下来;  
 **- 悲观锁**：正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）的修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。  
 以常用的mysql InnoDB存储引擎为例：加入商品表items表中有一个字段status，status=1表示该商品未被下单，status=2表示该商品已经被下单，那么我们对每个商品下单前必须确保此商品的status=1。假设有一件商品，其id为10000；如果不使用锁，那么操作方法如下：  
 //查出商品状态  
 select status from items where id=10000;  
 //根据商品信息生成订单  
 insert into orders(id,item*id) values(null,10000);*  
 *//修改商品状态为2*  
 *update Items set status=2 where id=10000;*  
 *上述场景在高并发环境下可能出现问题：*  
 *前面已经提到只有商品的status=1是才能对它进行下单操作，上面第一步操作中，查询出来的商品status为1。但是当我们执行第三步update操作的时候，有可能出现其他人先一步对商品下单把Item的status修改为2了，但是我们并不知道数据已经被修改了，这样就可能造成同一个商品被下单2次，使得数据不一致。所以说这种方式是不安全的。*  
 *使用悲观锁来实现：在上面的场景中，商品信息从查询出来到修改，中间有一个处理订单的过程，使用悲观锁的原理就是，当我们在查询出items信息后就把当前的数据锁定，直到我们修改完毕后再解锁。那么在这个过程中，因为items被锁定了，就不会出现有第三者来对其进行修改了。*  
 *注：要使用悲观锁，我们必须关闭mysql数据库的自动提交属性，因为MySQL默认使用autocommit模式，也就是说，当你执行一个更新操作后，MySQL会立刻将结果进行提交。我们可以使用命令设置MySQL为非autocommit模式：*  
 *set autocommit=0;*  
 *设置完autocommit后，我们就可以执行我们的正常业务了。具体如下：*  
 *//开始事务*  
 *begin;/begin work;/start transaction; (三者选一就可以)*  
 *//查询出商品信息*  
 *select status from items where id=10000 for update;*  
 *//根据商品信息生成订单*  
 *insert into orders (id,item*id) values (null,10000);  
 //修改商品status为2  
 update items set status=2 where id=10000;  
 //提交事务  
 commit;/commit work;  
 注：上面的begin/commit为事务的开始和结束，因为在前一步我们关闭了mysql的autocommit，所以需要手动控制事务的提交，在这里就不细表了。  
 上面的第一步我们执行了一次查询操作：select status from items where id=10000 for update;与普通查询不一样的是，我们使用了select…for update的方式，这样就通过数据库实现了悲观锁。此时在items表中，id为10000的 那条数据就被我们锁定了，其它的事务必须等本次事务提交之后才能执行。这样我们可以保证当前的数据不会被其它事务修改。  
 注：需要注意的是，在事务中，只有SELECT ... FOR UPDATE 或LOCK IN SHARE MODE 同一笔数据时会等待其它事务结束后才执行，一般SELECT ... 则不受此影响。拿上面的实例来说，当我执行select status from items where id=10000 for update;后。我在另外的事务中如果再次执行select status from items where id=10000 for update;则第二个事务会一直等待第一个事务的提交，此时第二个查询处于阻塞的状态，但是如果我是在第二个事务中执行select status from items where id=10000;则能正常查询出数据，不会受第一个事务的影响。  
 上面我们提到，使用select…for update会把数据给锁住，不过我们需要注意一些锁的级别，MySQL InnoDB默认Row-Level Lock，所以只有明确地指定主键，MySQL 才会执行Row lock (只锁住被选取的数据) ，否则MySQL 将会执行Table Lock (将整个数据表单给锁住)。除了主键外，使用索引也会影响数据库的锁定级别。  
 悲观锁并不是适用于任何场景，它也有它存在的一些不足，因为悲观锁大多数情况下依靠数据库的锁机制实现，以保证操作最大程度的独占性。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，特别是对长事务而言，这样的开销往往无法承受。所以与悲观锁相对的，我们有了乐观锁，乐观锁的概念如下：  
 **- 乐观锁**（ Optimistic Locking ） 相对悲观锁而言，乐观锁假设认为数据一般情况下不会造成冲突，所以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，则让返回用户错误的信息，让用户决定如何去做。那么我们如何实现乐观锁呢，一般来说有以下2种方式：  
 1.使用数据版本（Version）记录机制实现，这是乐观锁最常用的一种实现方式。何谓数据版本？即为数据增加一个版本标识，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，对此version值+1。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。用下面的一张图来说明：  
[](http://img1.tbcdn.cn/L1/461/1/aba656681527f77a6c28ab596e8f1938bacb043c) 如上图所示，如果更新操作顺序执行，则数据的版本（version）依次递增，不会产生冲突。但是如果发生有不同的业务操作对同一版本的数据进行修改，那么，先提交的操作（图中B）会把数据version更新为2，当A在B之后提交更新时发现数据的version已经被修改了，那么A的更新操作会失败。  
 2.乐观锁定的第二种实现方式和第一种差不多，同样是在需要乐观锁控制的table中增加一个字段，名称无所谓，字段类型使用时间戳（timestamp）, 和上面的version类似，也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比，如果一致则OK，否则就是版本冲突。  
 以mysql InnoDB存储引擎为例，还是拿之前的例子商品表items表中有一个字段status，status=1表示该商品未被下单，status=2表示该商品已经被下单，那么我们对每个商品下单前必须确保此商品的status=1。假设有一件商品，其id为10000；  
 下单操作包括3步骤：  
 //查询出商品信息  
 select (status,version) from items where id=#{id}  
 //根据商品信息生成订单  
 //修改商品status为2  
 update items set status=2,version=version+1 where id=#{id} and version=#{version};  
 为了使用乐观锁，我们需要首先修改items表，增加一个version字段，数据默认version可设为1；  
 其实我们周围的很多产品都有乐观锁的使用，比如我们经常使用的分布式存储引擎XXX，XXX中存储的每个数据都有版本号，版本号在每次更新后都会递增，相应的，在XXX put接口中也有此version参数，这个参数是为了解决并发更新同一个数据而设置的，这其实就是乐观锁；  
 很多情况下，更新数据是先get，修改get回来的数据，然后put回系统。如果有多个客户端get到同一份数据，都对其修改并保存，那么先保存的修改就会被后到达的修改覆盖，从而导致数据一致性问题,在大部分情况下应用能够接受，但在少量特殊情况下，这个是我们不希望发生的。  
 比如系统中有一个值”1”, 现在A和B客户端同时都取到了这个值。之后A和B客户端都想改动这个值，假设A要改成12，B要改成13，如果不加控制的话，无论A和B谁先更新成功，它的更新都会被后到的更新覆盖。XXX引入的乐观锁机制避免了这样的问题。刚刚的例子中，假设A和B同时取到数据，当时版本号是10，A先更新，更新成功后，值为12，版本为11。当B更新的时候，由于其基于的版本号是10，此时服务器会拒绝更新，返回version error，从而避免A的更新被覆盖。B可以选择get新版本的value，然后在其基础上修改，也可以选择强行更新。

原文链接：https://blog.csdn.net/xiaobing\_122613/article/details/78672708

### 9.跟 Synchronized 相比，可重入锁 ReentrantLock 其实现原理有什么不同？

https://blog.csdn.net/qq838642798/article/details/65441415

ReenTrantLock可重入锁（和synchronized的区别）总结  
可重入性：  
从名字上理解，ReenTrantLock的字面意思就是再进入的锁，其实synchronized关键字所使用的锁也是可重入的，两者关于这个的区别不大。两者都是同一个线程没进入一次，锁的计数器都自增1，所以要等到锁的计数器下降为0时才能释放锁。

锁的实现：  
Synchronized是依赖于JVM实现的，而ReenTrantLock是JDK实现的，有什么区别，说白了就类似于操作系统来控制实现和用户自己敲代码实现的区别。前者的实现是比较难见到的，后者有直接的源码可供阅读。

性能的区别：  
在Synchronized优化以前，synchronized的性能是比ReenTrantLock差很多的，但是自从Synchronized引入了偏向锁，轻量级锁（自旋锁）后，两者的性能就差不多了，在两种方法都可用的情况下，官方甚至建议使用synchronized，其实synchronized的优化我感觉就借鉴了ReenTrantLock中的CAS技术。都是试图在用户态就把加锁问题解决，避免进入内核态的线程阻塞。

功能区别：  
便利性：很明显Synchronized的使用比较方便简洁，并且由编译器去保证锁的加锁和释放，而ReenTrantLock需要手工声明来加锁和释放锁，为了避免忘记手工释放锁造成死锁，所以最好在finally中声明释放锁。  
锁的细粒度和灵活度：很明显ReenTrantLock优于Synchronized

ReenTrantLock独有的能力：

1. ReenTrantLock可以指定是公平锁还是非公平锁。而synchronized只能是非公平锁。所谓的公平锁就是先等待的线程先获得锁。
2. ReenTrantLock提供了一个Condition（条件）类，用来实现分组唤醒需要唤醒的线程们，而不是像synchronized要么随机唤醒一个线程要么唤醒全部线程。
3. ReenTrantLock提供了一种能够中断等待锁的线程的机制，通过lock.lockInterruptibly()来实现这个机制。

ReenTrantLock实现的原理：  
在网上看到相关的源码分析，本来这块应该是本文的核心，但是感觉比较复杂就不一一详解了，简单来说，ReenTrantLock的实现是一种自旋锁，通过循环调用CAS操作来实现加锁。它的性能比较好也是因为避免了使线程进入内核态的阻塞状态。想尽办法避免线程进入内核的阻塞状态是我们去分析和理解锁设计的关键钥匙。

什么情况下使用ReenTrantLock：  
答案是，如果你需要实现ReenTrantLock的三个独有功能时。  
————————————————  
版权声明：本文为CSDN博主「qq838642798」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/qq838642798/article/details/65441415

另一篇：

详解synchronized 和ReentrantLock两种锁实现原理https://blog.csdn.net/ly199108171231/article/details/88098614

版权声明：本文为博主原创文章，遵循[CC 4.0 BY-SA](http://creativecommons.org/licenses/by-sa/4.0/) 版权协议，转载请附上原文出处链接和本声明。

本文链接：https://blog.csdn.net/ly199108171231/article/details/88098614

**1、概述**

Java 中的并发锁大致分为隐式锁和显式锁两种。隐式锁就是我们最常使用的 synchronized 关键字，显式锁主要包含两个接口：**Lock 和 ReadWriteLock**，主要实现类分别为**ReentrantLock 和 ReentrantReadWriteLock**，这两个类都是基于**AQS**(AbstractQueuedSynchronizer) 实现的。还有的地方将 **CAS** 也称为一种锁，在包括 AQS 在内的很多并发相关类中，CAS 都扮演了很重要的角色。

**2、概念辨析**

**2.1悲观锁和乐观锁**

悲观锁和独占锁是一个意思，它假设一定会发生冲突，因此获取到锁之后会阻塞其他等待线程。这么做的好处是简单安全，但是挂起线程和恢复线程都需要转入内核态进行，这样做会带来很大的性能开销。悲观锁的代表是 synchronized。然而在真实环境中，大部分时候都不会产生冲突。悲观锁会造成很大的浪费。而乐观锁不一样，它假设不会产生冲突，先去尝试执行某项操作，失败了再进行其他处理（一般都是不断循环重试）。这种锁不会阻塞其他的线程，也不涉及上下文切换，性能开销小。代表实现是 CAS。

**2.2公平锁和非公平锁**

公平锁是指各个线程在加锁前先检查有无排队的线程，按排队顺序去获得锁。

非公平锁是指线程加锁前不考虑排队问题，直接尝试获取锁，获取不到再去队尾排队。值得注意的是，在 AQS 的实现中，一旦线程进入排队队列，即使是非公平锁，线程也得乖乖排队。

**2.3可重入锁和不可重入锁**

如果一个线程已经获取到了一个锁，那么它可以访问被这个锁锁住的所有代码块。不可重入锁与之相反。

**2.4Synchronized 关键字**

Synchronized 是一种独占锁。在修饰静态方法时，锁的是类对象，如 Object.class。修饰非静态方法时，锁的是对象，即 this。修饰方法块时，锁的是括号里的对象。

每个对象有一个锁和一个等待队列，锁只能被一个线程持有，其他需要锁的线程需要阻塞等待。锁被释放后，对象会从队列中取出一个并唤醒，唤醒哪个线程是不确定的，不保证公平性。

**2.5类锁与对象锁**

synchronized 修饰静态方法时，锁的是类对象,如 Object.class。修饰非静态方法时，锁的是对象，即 this。

多个线程是可以同时执行同一个synchronized实例方法的，只要它们访问的对象是不同的。

synchronized 锁住的是对象而非代码，只要访问的是同一个对象的 synchronized 方法，即使是不同的代码，也会被同步顺序访问。

此外，需要说明的，synchronized方法不能防止非synchronized方法被同时执行，所以，一般在保护变量时，需要在所有访问该变量的方法上加上synchronized。

**3、实现原理**

synchronized 是基于 Java 对象头和 Monitor 机制来实现的。

**3.1 Java 对象头**

一个对象在内存中包含三部分：对象头，实例数据和对齐填充。其中 Java 对象头包含两部分：

**Class Metadata Address （类型指针）**。存储类的元数据的指针。虚拟机通过这个指针找到它是哪个类的实例。

**Mark Word（标记字段）**。存出一些对象自身运行时的数据。包括哈希码，GC 分代年龄，锁状态标志等。

**Monitor**

Mark Word 有一个字段指向 monitor 对象。monitor 中记录了锁的持有线程，等待的线程队列等信息。前面说的每个对象都有一个锁和一个等待队列，就是在这里实现的。

monitor 对象由 C++ 实现。其中有三个关键字段：

* \_owner 记录当前持有锁的线程
* \_EntryList 是一个队列，记录所有阻塞等待锁的线程
* \_WaitSet 也是一个队列，记录调用 wait() 方法并还未被通知的线程。

**3.2 Monitor的操作机制如下：**

多个线程竞争锁时，会先进入 EntryList 队列。竞争成功的线程被标记为 Owner。其他线程继续在此队列中阻塞等待。

如果 Owner 线程调用 wait() 方法，则其释放对象锁并进入 WaitSet 中等待被唤醒。Owner 被置空，EntryList 中的线程再次竞争锁。

如果 Owner 线程执行完了，便会释放锁，Owner 被置空，EntryList 中的线程再次竞争锁。

**3.3JVM 对 synchronized 的处理**

上面了解了 monitor 的机制，那虚拟机是如何将 synchronized 和 monitor 关联起来的呢？分两种情况：

如果同步的是代码块，编译时会直接在同步代码块前加上 monitorenter 指令，代码块后加上 monitorexit 指令。这称为显示同步。

如果同步的是方法，虚拟机会为方法设置 ACC\_SYNCHRONIZED 标志。调用的时候 JVM 根据这个标志判断是否是同步方法。

JVM 对 synchronized 的优化

synchronized 是重量级锁，由于消耗太大，虚拟机对其做了一些优化。

**4自旋锁与自适应自旋**

在许多应用中，锁定状态只会持续很短的时间，为了这么一点时间去挂起恢复线程，不值得。我们可以让等待线程执行一定次数的循环，在循环中去获取锁。这项技术称为自旋锁，它可以节省系统切换线程的消耗，但仍然要占用处理器。在 JDK1.4.2 中，自选的次数可以通过参数来控制。 JDK 1.6又引入了自适应的自旋锁，不再通过次数来限制，而是由前一次在同一个锁上的自旋时间及锁的拥有者的状态来决定。

**锁消除**

虚拟机在运行时，如果发现一段被锁住的代码中不可能存在共享数据，就会将这个锁清除。

**锁粗化**

当虚拟机检测到有一串零碎的操作都对同一个对象加锁时，会把锁扩展到整个操作序列外部。如 StringBuffer 的 append 操作。

**轻量级锁**

对绝大部分的锁来说，在整个同步周期内都不存在竞争。如果没有竞争，轻量级锁可以使用 CAS 操作避免使用互斥量的开销。

**偏向锁**

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，当这个线程再次请求锁时，无需再做任何同步操作，即可获取锁。

**CAS**

**操作模型**

CAS 是 compare and swap 的简写，即比较并交换。它是指一种操作机制，而不是某个具体的类或方法。在 Java 平台上对这种操作进行了包装。在 Unsafe 类中，调用代码如下：

unsafe.compareAndSwapInt(this, valueOffset, expect, update);

复制代码它需要三个参数，分别是内存位置 V，旧的预期值 A 和新的值 B。操作时，先从内存位置读取到值，然后和预期值A比较。如果相等，则将此内存位置的值改为新值 B，返回 true。如果不相等，说明和其他线程冲突了，则不做任何改变，返回 false。

这种机制在不阻塞其他线程的情况下避免了并发冲突，比独占锁的性能高很多。 CAS 在 Java 的原子类和并发包中有大量使用。

**重试机制（循环 CAS）**

有很多文章说，CAS 操作失败后会一直重试直到成功，这种说法很不严谨。

第一，CAS 本身并未实现失败后的处理机制，它只负责返回成功或失败的布尔值，后续由调用者自行处理。只不过我们最常用的处理方式是重试而已。

第二，这句话很容易理解错，被理解成重新比较并交换。实际上失败的时候，原值已经被修改，如果不更改期望值，再怎么比较都会失败。而新值同样需要修改。

所以正确的方法是，使用一个死循环进行 CAS 操作，成功了就结束循环返回，失败了就重新从内存读取值和计算新值，再调用 CAS。看下 AtomicInteger 的源码就什么都懂了：

public final int incrementAndGet () {  
 for (;;) {  
 int current = get();  
 int next = current + 1;  
 if (compareAndSet(current, next))  
 return next;  
 }  
}

**底层实现**

CAS 主要分三步，读取-比较-修改。其中比较是在检测是否有冲突，如果检测到没有冲突后，其他线程还能修改这个值，那么 CAS 还是无法保证正确性。所以最关键的是要保证比较-修改这两步操作的原子性。

CAS 底层是靠调用 CPU 指令集的 cmpxchg 完成的，它是 x86 和 Intel 架构中的 compare and exchange 指令。在多核的情况下，这个指令也不能保证原子性，需要在前面加上 lock 指令。lock 指令可以保证一个 CPU 核心在操作期间独占一片内存区域。那么 这又是如何实现的呢？

在处理器中，一般有两种方式来实现上述效果：总线锁和缓存锁。在多核处理器的结构中，CPU 核心并不能直接访问内存，而是统一通过一条总线访问。总线锁就是锁住这条总线，使其他核心无法访问内存。这种方式代价太大了，会导致其他核心停止工作。而缓存锁并不锁定总线，只是锁定某部分内存区域。当一个 CPU 核心将内存区域的数据读取到自己的缓存区后，它会锁定缓存对应的内存区域。锁住期间，其他核心无法操作这块内存区域。

CAS 就是通过这种方式实现比较和交换操作的原子性的。值得注意的是， CAS 只是保证了操作的原子性，并不保证变量的可见性，因此变量需要加上 volatile 关键字。

**ABA 问题**

上面提到，CAS 保证了比较和交换的原子性。但是从读取到开始比较这段期间，其他核心仍然是可以修改这个值的。如果核心将 A 修改为 B，CAS 可以判断出来。但是如果核心将 A 修改为 B 再修改回 A。那么 CAS 会认为这个值并没有被改变，从而继续操作。这是和实际情况不符的。解决方案是加一个版本号。

**可重入锁 ReentrantLock**

ReentrantLock 使用代码实现了和 synchronized 一样的语义，包括可重入，保证内存可见性和解决竞态条件问题等。相比 synchronized，它还有如下好处：

支持以非阻塞方式获取锁

可以响应中断

可以限时

支持了公平锁和非公平锁

基本用法如下：

public class Counter {  
 private final Lock lock = new ReentrantLock();  
 private volatile int count;  
 public void incr() {  
 lock.lock();  
 try {  
 count++;  
 } finally {  
 lock.unlock();  
 }  
 }  
 public int getCount() {  
 return count;  
 }  
}

ReentrantLock 内部有两个内部类，分别是 FairSync 和 NoFairSync，对应公平锁和非公平锁。他们都继承自 Sync。Sync 又继承自AQS。

AQS

AQS 全称 AbstractQueuedSynchronizer。AQS 中有两个重要的成员：

成员变量 state。用于表示锁现在的状态，用 volatile 修饰，保证内存一致性。同时所用对 state 的操作都是使用 CAS 进行的。state 为0表示没有任何线程持有这个锁，线程持有该锁后将 state 加1，释放时减1。多次持有释放则多次加减。

还有一个双向链表，链表除了头结点外，每一个节点都记录了线程的信息，代表一个等待线程。这是一个 FIFO 的链表。

下面以 ReentrantLock 非公平锁的代码看看 AQS 的原理。

**请求锁**

请求锁时有三种可能：

1. 如果没有线程持有锁，则请求成功，当前线程直接获取到锁。
2. 如果当前线程已经持有锁，则使用 CAS 将 state 值加1，表示自己再次申请了锁，释放锁时减1。这就是可重入性的实现。
3. 如果由其他线程持有锁，那么将自己添加进等待队列。

final void lock() {  
if (compareAndSetState(0, 1))   
setExclusiveOwnerThread(Thread.currentThread()); //没有线程持有锁时，直接获取锁，对应情况1  
else  
acquire(1);  
}  
public final void acquire(int arg) {  
if (!tryAcquire(arg) && //在此方法中会判断当前持有线程是否等于自己，对应情况2  
acquireQueued(addWaiter(Node.EXCLUSIVE), arg)) //将自己加入队列中，对应情况3  
selfInterrupt();  
}

创建 Node 节点并加入链表

如果没竞争到锁，这时候就要进入等待队列。队列是默认有一个 head 节点的，并且不包含线程信息。上面情况3中，addWaiter 会创建一个 Node，并添加到链表的末尾，Node 中持有当前线程的引用。同时还有一个成员变量 waitStatus，表示线程的等待状态，初始值为0。我们还需要关注两个值：

CANCELLED，值为1，表示取消状态，就是说我不要这个锁了，请你把我移出去。

SINGAL，值为-1，表示下一个节点正在挂起等待，注意是下一个节点，不是当前节点。

同时，加到链表末尾的操作使用了 CAS+死循环的模式，很有代表性，拿出来看一看：

Node node = new Node(mode);  
for (;;) {  
Node oldTail = tail;  
if (oldTail != null) {  
U.putObject(node, Node.PREV, oldTail);  
if (compareAndSetTail(oldTail, node)) {  
oldTail.next = node;  
return node;  
}  
} else {  
initializeSyncQueue();  
}  
}

可以看到，在死循环里调用了 CAS 的方法。如果多个线程同时调用该方法，那么每次循环都只有一个线程执行成功，其他线程进入下一次循环，重新调用。N个线程就会循环N次。这样就在无锁的模式下实现了并发模型。

挂起等待

如果此节点的上一个节点是头部节点，则再次尝试获取锁，获取到了就移除并返回。获取不到就进入下一步；

判断前一个节点的 waitStatus，如果是 SINGAL，则返回 true，并调用 LockSupport.park() 将线程挂起；

如果是 CANCELLED，则将前一个节点移除；

如果是其他值，则将前一个节点的 waitStatus 标记为 SINGAL，进入下一次循环。

可以看到，一个线程最多有两次机会，还竞争不到就去挂起等待。

final boolean acquireQueued(final Node node, int arg) {  
try {  
boolean interrupted = false;  
for (;;) {  
final Node p = node.predecessor();  
if (p == head && tryAcquire(arg)) {  
setHead(node);  
p.next = null; // help GC  
return interrupted;  
}  
if (shouldParkAfterFailedAcquire(p, node) &&  
parkAndCheckInterrupt())  
interrupted = true;  
}  
} catch (Throwable t) {  
cancelAcquire(node);  
throw t;  
}  
}

**释放锁**

调用 tryRelease，此方法由子类实现。实现非常简单，如果当前线程是持有锁的线程，就将 state 减1。减完后如果 state 大于0，表示当前线程仍然持有锁，返回 false。如果等于0，表示已经没有线程持有锁，返回 true，进入下一步；

如果头部节点的 waitStatus 不等于0，则调用LockSupport.unpark()唤醒其下一个节点。头部节点的下一个节点就是等待队列中的第一个线程，这反映了 AQS 先进先出的特点。另外，即使是非公平锁，进入队列之后，还是得按顺序来。

public final boolean release(int arg) {  
if (tryRelease(arg)) { //将 state 减1  
Node h = head;  
if (h != null && h.waitStatus != 0)  
unparkSuccessor(h);  
return true;  
}  
return false;  
}  
private void unparkSuccessor(Node node) {  
int ws = node.waitStatus;  
if (ws < 0)  
node.compareAndSetWaitStatus(ws, 0);  
Node s = node.next;  
if (s == null || s.waitStatus > 0) {   
 s = null;  
 for (Node p = tail; p != node && p != null; p = p.prev)  
 if (p.waitStatus <= 0)  
 s = p;  
}  
if (s != null) //唤醒第一个等待的线程  
 LockSupport.unpark(s.thread);  
}

**公平锁如何实现**

上面分析的是非公平锁，那公平锁呢？很简单，在竞争锁之前判断一下等待队列中有没有线程在等待就行了。

protected final boolean tryAcquire(int acquires) {  
final Thread current = Thread.currentThread();  
int c = getState();  
if (c == 0) {  
if (!hasQueuedPredecessors() && //判断等待队列是否有节点  
compareAndSetState(0, acquires)) {  
setExclusiveOwnerThread(current);  
return true;  
}  
}  
......  
return false;  
}

**可重入读写锁 ReentrantReadWriteLock**

**读写锁机制**

理解 ReentrantLock 和 AQS 之后，再来理解读写锁就很简单了。读写锁有一个读锁和一个写锁，分别对应读操作和锁操作。锁的特性如下：

只有一个线程可以获取到写锁。在获取写锁时，只有没有任何线程持有任何锁才能获取成功；

如果有线程正持有写锁，其他任何线程都获取不到任何锁；

没有线程持有写锁时，可以有多个线程获取到读锁。

上面锁的特点保证了可以并发读取，这大大提高了效率，在实际开发中非常有用。那么在具体是如何实现的呢？

实现原理

读写锁虽然有两个锁，但实际上只有一个等待队列。

获取写锁时，要保证没有任何线程持有锁；

写锁释放后，会唤醒队列第一个线程，可能是读锁和写锁；

获取读锁时，先判断写锁有没有被持有，没有就可以获取成功；

获取读锁成功后，会将队列中等待读锁的线程挨个唤醒，知道遇到等待写锁的线程位置；

释放读锁时，要检查读锁数，如果为0，则唤醒队列中的下一个线程，否则不进行操作。

### 10.那么请谈谈 AQS 框架是怎么回事儿？

[AQS框架的理解](https://www.cnblogs.com/jihuabai/p/12239062.html) https://www.cnblogs.com/jihuabai/p/12239062.html

在实习的时候，需要对公司内部的分布式框架（RPC框架）进行拓展。在阅读该RPC框架源码的时候，发现该框架中较多地方使用了自增原子类，而原子类又是基于AQS实现，在秋招之前阅读过AQS框架，但是都是粗粗的阅读了一些博客，并没有对源码进行阅读。如今，趁着过年有时间对AQS源码进行梳理。

\1. 原理简介

\2. 部分Node类分析

根据原理可知道，AQS是一个线程同步工具，其主要作用是内部维持了一个双向队列，以及一个状态，如果没有获取到状态，那么该线程则会被加入等待队列。而这个队列中的节点(Node)则是AQS内部实现的类，其主要的属性如下：

static final AbstractQueuedSynchronizer.Node SHARED = new AbstractQueuedSynchronizer.Node();  
  
static final AbstractQueuedSynchronizer.Node EXCLUSIVE = null;  
  
static final int CANCELLED = 1;   
  
static final int SIGNAL = -1;  
  
static final int CONDITION = -2;  
  
static final int PROPAGATE = -3;  
  
volatile int waitStatus; // 等待状态（是否超时等）  
  
volatile AbstractQueuedSynchronizer.Node prev; // 指向前一个节点  
  
volatile AbstractQueuedSynchronizer.Node next; // 指向后一个节点  
  
volatile Thread thread; // 获取线程信息  
  
AbstractQueuedSynchronizer.Node nextWaiter;   
  
private static final VarHandle NEXT; // VarHandle为本地同步类  
  
private static final VarHandle PREV;  
  
private static final VarHandle THREAD;  
  
private static final VarHandle WAITSTATUS;

注：在java9 之后均使用VarHandle来实现CAS操作，代替了原先的Unsafe类，其好处是屏蔽了内存模型，使得可以适应不同的系统。

**框架意义**

只需要根据AQS提供的工具实现排斥锁和共享锁即可，而无须关注状态是否获取成功，是否需要排队，唤醒等操作。

**框架原理**

**1. 属性**

首先，AQS框架是为了设计锁而存在的，而锁的就是对状态的获取。在AQS中通过private volatile int state来表示锁的状态，在独占锁时只有0、1两个状态，在共享锁时大于等于0表示锁的状态。其次，我们说过AQS解决了解决了线程等待、排队以及唤醒的问题，这一过程是通过双向链表来完成的。所以在AQS中有head以及tail两个节点。AQS的基础原理是基于CAS来完成状态获取操作（无论是获取排他锁还是共享锁），CAS操作又是基于VarHandle来完成，所以在类属性中存在

private static final VarHandle STATE; // 操作状态  
  
private static final VarHandle HEAD; // 操作头节点  
  
private static final VarHandle TAIL; // 操作尾节点

（对于这三个状态的操作采用了三个工具，本身也是为了互不干扰，如果是采用一个VarHandle变量，会影响框架的效率。比方说：在A线程需要通过VarHandle设置头节点，此时B线程需要通过VarHandle设置尾节点，需要等待A线程操作结束之后才行。当然以上是我的猜测。）

具体的属性列表如下：

private static final long serialVersionUID = 7373984972572414691L;  
  
private transient volatile AbstractQueuedSynchronizer.Node head;  
  
private transient volatile AbstractQueuedSynchronizer.Node tail;  
  
private volatile int state;  
  
static final long SPIN\_FOR\_TIMEOUT\_THRESHOLD = 1000L;  
  
private static final VarHandle STATE;  
  
private static final VarHandle HEAD;  
  
private static final VarHandle TAIL;

**2. 方法**

protected final boolean compareAndSetState(int expect, int update) {  
  
 return STATE.compareAndSet(this, expect, update);  
  
}

上面一段代码没啥可以说的，就是利用STATE完成CAS操作。

[?](https://www.cnblogs.com/jihuabai/p/12239062.html#)

private AbstractQueuedSynchronizer.Node enq(AbstractQueuedSynchronizer.Node node) {  
  
 while(true) {  
  
 AbstractQueuedSynchronizer.Node oldTail = this.tail; // 获取尾节点  
  
 if (oldTail != null) {  
  
 node.setPrevRelaxed(oldTail); // 将node的前一个节点设置为oldTail节点  
  
 if (this.compareAndSetTail(oldTail, node)) {  
  
 oldTail.next = node;  
  
 return oldTail;  
  
 }  
  
 } else {  
  
 this.initializeSyncQueue();  
  
 }  
  
 }  
  
 }<br>// 下面就是setPrevRelaxed方法的签名  
 final void setPrevRelaxed(AbstractQueuedSynchronizer.Node p) {  
 PREV.set(this, p);  
 }

enq方法的主要作用就是通过CAS+自旋完成尾节点的插入。

private AbstractQueuedSynchronizer.Node addWaiter(AbstractQueuedSynchronizer.Node mode) {  
  
 AbstractQueuedSynchronizer.Node node = new AbstractQueuedSynchronizer.Node(mode);  
  
   
  
 AbstractQueuedSynchronizer.Node oldTail;  
  
 do {  
  
 while(true) {  
  
 oldTail = this.tail;  
  
 if (oldTail != null) {  
  
 node.setPrevRelaxed(oldTail); // node节点的前向指针指向当前尾节点  
  
 break;  
  
 }  
  
   
  
 this.initializeSyncQueue(); // 更新  
  
 }  
  
 } while(!this.compareAndSetTail(oldTail, node));  
  
   
  
 oldTail.next = node; // 尾节点的后向指针指向node节点  
  
 return node;  
  
}

addWaiter方法的主要作用就是CAS + 自旋完成节点的添加，与enq方法的不同时，这里的插入是形成了双向链表。

[?](https://www.cnblogs.com/jihuabai/p/12239062.html#)

private void unparkSuccessor(AbstractQueuedSynchronizer.Node node) {  
  
 int ws = node.waitStatus;  
  
 if (ws < 0) {  
  
 node.compareAndSetWaitStatus(ws, 0);  
  
 }  
  
   
  
 AbstractQueuedSynchronizer.Node s = node.next;  
  
 if (s == null || s.waitStatus > 0) {  
  
 s = null;  
  
   
  
 for(AbstractQueuedSynchronizer.Node p = this.tail; p != node && p != null; p = p.prev) {  
  
 if (p.waitStatus <= 0) {  
  
 s = p;  
  
 }  
  
 }  
  
 }  
  
   
  
 if (s != null) {  
  
 LockSupport.unpark(s.thread);  
  
 }  
  
   
  
}

unparkSuccessor方法的主要任务是：唤醒其他等待锁的节点。

其主要流程是：

\1. 如果当前节点的状态为取消状态，则对其进行初始化。

\2. 拿到node后面的一个节点。

\3. 如果后边的节点为空或者被取消（waitStatus > 0 则表明节点已经被取消）

\4. 开始从尾部开始进行迭代。原因是：节点被阻塞的时候，是在 acquireQueued 方法里面被阻塞的，唤醒时也一定会在 acquireQueued 方法里面被唤醒，唤醒之后的条件是，判断当前节点的前置节点是否是头节点，这里是判断当前节点的前置节点，所以这里必须使用从尾到头的迭代顺序才行，目的就是为了过滤掉无效的前置节点，不然节点被唤醒时，发现其前置节点还是无效节点，就又会陷入阻塞。

**条件队列**

主要是因为并不是所有场景一个同步队列就可以搞定的，在遇到锁 + 队列结合的场景时，就需要 Lock + Condition 配合才行，先使用 Lock 来决定哪些线程可以获得锁，哪些线程需要到同步队列里面排队阻塞；获得锁的多个线程在碰到队列满或者空的时候，可以使用 Condition 来管理这些线程，让这些线程阻塞等待，然后在合适的时机后，被正常唤醒。

说白了就是同步队列的预备队列。在同步队列满了之后，阻塞的线程无法进入同步队列，这时候会进入条件队列（代表获得了抢占锁的机会）。同步队列是负责互斥，也就是如果没有获取锁就等着，等待获取锁的那一刻。而条件队列是告诉你什么时候可以去等待获取锁。类似于银行排队，一个窗口只能负责一个人，但是一个窗口有很多人在排队，如果排队的人过多，让你先不要排队，先去等候区坐着，等到了满足条件的时候，大堂经理会告诉你要准备了（可以进入排队）。

### 11.请尽可能详尽地对比下 Synchronized 和 ReentrantLock 的异同。

Synchronized与ReentrantLock区别总结（简单粗暴，一目了然）https://blog.csdn.net/zxd8080666/article/details/83214089

这篇文章是关于这两个同步锁的简单总结比较，关于底层源码实现原理没有过多涉及，后面会有关于这两个同步锁的底层原理篇幅去介绍。

相似点：

这两种同步方式有很多相似之处，它们都是加锁方式同步，而且都是阻塞式的同步，也就是说当如果一个线程获得了对象锁，进入了同步块，其他访问该同步块的线程都必须阻塞在同步块外面等待，而进行线程阻塞和唤醒的代价是比较高的（操作系统需要在用户态与内核态之间来回切换，代价很高，不过可以通过对锁优化进行改善）。

功能区别：

这两种方式最大区别就是对于Synchronized来说，它是java语言的关键字，是原生语法层面的互斥，需要jvm实现。而ReentrantLock它是JDK 1.5之后提供的API层面的互斥锁，需要lock()和unlock()方法配合try/finally语句块来完成

便利性：很明显Synchronized的使用比较方便简洁，并且由编译器去保证锁的加锁和释放，而ReenTrantLock需要手工声明来加锁和释放锁，为了避免忘记手工释放锁造成死锁，所以最好在finally中声明释放锁。

锁的细粒度和灵活度：很明显ReenTrantLock优于Synchronized

性能的区别：

在Synchronized优化以前，synchronized的性能是比ReenTrantLock差很多的，但是自从Synchronized引入了偏向锁，轻量级锁（自旋锁）后，两者的性能就差不多了，在两种方法都可用的情况下，官方甚至建议使用synchronized，其实synchronized的优化我感觉就借鉴了ReenTrantLock中的CAS技术。都是试图在用户态就把加锁问题解决，避免进入内核态的线程阻塞。

1.Synchronized

Synchronized进过编译，会在同步块的前后分别形成monitorenter和monitorexit这个两个字节码指令。在执行monitorenter指令时，首先要尝试获取对象锁。如果这个对象没被锁定，或者当前线程已经拥有了那个对象锁，把锁的计算器加1，相应的，在执行monitorexit指令时会将锁计算器就减1，当计算器为0时，锁就被释放了。如果获取对象锁失败，那当前线程就要阻塞，直到对象锁被另一个线程释放为止。

public class SynDemo{ public static void main(String[] arg){ Runnable t1=new MyThread(); new Thread(t1,"t1").start(); new Thread(t1,"t2").start(); } }class MyThread implements Runnable { @Override public void run() { synchronized (this) { for(int i=0;i<10;i++) System.out.println(Thread.currentThread().getName()+":"+i); } } }

2.ReentrantLock

由于ReentrantLock是java.util.concurrent包下提供的一套互斥锁，相比Synchronized，ReentrantLock类提供了一些高级功能，主要有以下3项：

1.等待可中断，持有锁的线程长期不释放的时候，正在等待的线程可以选择放弃等待，这相当于Synchronized来说可以避免出现死锁的情况。通过lock.lockInterruptibly()来实现这个机制。

2.公平锁，多个线程等待同一个锁时，必须按照申请锁的时间顺序获得锁，Synchronized锁非公平锁，ReentrantLock默认的构造函数是创建的非公平锁，可以通过参数true设为公平锁，但公平锁表现的性能不是很好。

公平锁、非公平锁的创建方式：

//创建一个非公平锁，默认是非公平锁  
Lock lock = new ReentrantLock();Lock lock = new ReentrantLock(false);   
//创建一个公平锁，构造传参  
trueLock lock = new ReentrantLock(true);

3.锁绑定多个条件，一个ReentrantLock对象可以同时绑定对个对象。ReenTrantLock提供了一个Condition（条件）类，用来实现分组唤醒需要唤醒的线程们，而不是像synchronized要么随机唤醒一个线程要么唤醒全部线程。

ReenTrantLock实现的原理：

之后还会总结一篇ReenTrantLock相关的原理底层原理分析，简单来说，ReenTrantLock的实现是一种自旋锁，通过循环调用CAS操作来实现加锁。它的性能比较好也是因为避免了使线程进入内核态的阻塞状态。想尽办法避免线程进入内核的阻塞状态是我们去分析和理解锁设计的关键钥匙。

什么情况下使用ReenTrantLock：

答案是，如果你需要实现ReenTrantLock的三个独有功能时。

ReentrantLock的用法如下：

public class SynDemo{ public static void main(String[] arg){ Runnable t1=new MyThread(); new Thread(t1,"t1").start(); new Thread(t1,"t2").start(); } }class MyThread implements Runnable { private Lock lock=new ReentrantLock(); public void run() { lock.lock(); try{ for(int i=0;i<5;i++) System.out.println(Thread.currentThread().getName()+":"+i); }finally{ lock.unlock(); } } }  
————————————————  
版权声明：本文为CSDN博主「奋力奔跑的蜗牛」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/zxd8080666/article/details/83214089

对ReentrantLock的源码分析这有一篇很好的文章

http://www.blogjava.net/zhanglongsr/articles/356782.html

后续在补充的问题：

Synchronized的原理

ReentrantLock的原理。

ReentrantLock为什么是可重入的。

公平锁和非公平锁是什么？有什么区别。  
————————————————  
版权声明：本文为CSDN博主「奋力奔跑的蜗牛」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/zxd8080666/article/details/83214089

### 12.ReentrantLock 是如何实现可重入性的？

答案：什么是可重入性？ReentrantLock和Synchronized是怎么实现可重入的？https://www.jianshu.com/p/ca7df6c1110f

1、什么是可重入

2 synchronized如何实现可重入性

3 ReentrantLock如何实现可重入性

众所周知，ReentrantLock和synchronized是可重入锁。下面分析什么是可重入性，以及它们如何实现可重入性

**1 什么是可重入？**

先看定义，定义来自维基百科

若一个程序或子程序可以“在任意时刻被中断然后操作系统调度执行另外一段代码，这段代码又调用了该子程序不会出错”，则称其为可重入（reentrant或re-entrant）的。即当该子程序正在运行时，执行线程可以再次进入并执行它，仍然获得符合设计时预期的结果。与多线程并发执行的线程安全不同，可重入强调对单个线程执行时重新进入同一个子程序仍然是安全的。

代码测试，两个方法的锁都是MyReent对象，在run()内部调用了function()方法，在执行run()方法时，已经拿到了锁对象reent，假设synchronized是不可重入的，那么在run()内部调用function() 时会被阻塞，但是事实却没有，所以可以看出synchronized是可重入的。即使是两个方法相互调用也不会发生死锁，只会抛出栈溢出异常。  
 简言之，**一个线程持有锁时，当其他线程尝试获取该锁时，会被阻塞；而这个线程尝试获取自己持有锁时，如果成功说明该锁是可重入的，反之则不可重入。**

public class ReentrantTest {  
  
  
 public static void main(String[] args) {  
 MyReent reent = new MyReent();  
 new Thread(reent).start();  
 }  
  
}  
class MyReent implements Runnable{  
  
  
 @Override  
 public synchronized void run() {  
 System.out.println(Thread.currentThread().getName() + " run");  
 function();  
 }  
  
  
 public synchronized void function(){  
 System.out.println(Thread.currentThread().getName()+ " function");  
 }  
}  
Thread-0 run  
Thread-0 function

**2 synchronized如何实现可重入性**

synchronized关键字经过编译之后，会在同步块的前后分别形成**monitorenter和monitorexit**这两个字节码指令。每个锁对象内部维护一个计数器，该计数器初始值为0，表示任何线程都可以获取该锁并执行相应的方法。根据虚拟机规范的要求，在执行monitorenter指令时，首先要尝试获取对象的锁。如果这个对象没被锁定，或者当前线程已经拥有了那个对象的锁，把锁的计数器加1，相应的，在执行monitorexit指令时会将锁计数器减1，当计数器为0时，锁就被释放。如果获取对象锁失败，那当前线程就要阻塞等待，直到对象锁被另外一个线程释放为止。

**3 ReentrantLock如何实现可重入性**

ReentrantLock在内部使用了内部类Sync来管理锁，所以真正的获取锁是由Sync的实现类控制的。Sync有两个实现，分别为NonfairSync（非公平锁）和FairSync（公平锁）。Sync通过继承AQS实现，在AQS中维护了一个private volatile int state来计数重入次数，避免了频繁的持有释放操作带来效率问题。  
 下面是部分ReentrantLock源码

// Sync继承于AQS  
abstract static class Sync extends AbstractQueuedSynchronizer {  
 ...  
}  
// ReentrantLock默认是非公平锁  
public ReentrantLock() {  
 sync = new NonfairSync();  
 }  
// 可以通过向构造方法中传true来实现公平锁  
public ReentrantLock(boolean fair) {  
 sync = fair ? new FairSync() : new NonfairSync();  
}

线程抢锁过程（公平锁实现）：

protected final boolean tryAcquire(int acquires) {  
 // 当前想要获取锁的线程  
 final Thread current = Thread.currentThread();  
 // 当前锁的状态  
 int c = getState();  
 // state == 0 此时此刻没有线程持有锁  
 if (c == 0) {  
 // 虽然此时此刻锁是可以用的，但是这是公平锁，既然是公平，就得讲究先来后到，  
 // 看看有没有别人在队列中等了半天了  
 if (!hasQueuedPredecessors() &&  
 // 如果没有线程在等待，那就用CAS尝试一下，成功了就获取到锁了，  
 // 不成功的话，只能说明一个问题，就在刚刚几乎同一时刻有个线程抢先了 =\_=  
 // 因为刚刚还没人的，我判断过了  
 compareAndSetState(0, acquires)) {  
  
 // 到这里就是获取到锁了，标记一下，告诉大家，现在是我占用了锁  
 setExclusiveOwnerThread(current);  
 return true;  
 }  
 }  
 // 会进入这个else if分支，说明是重入了，需要操作：state=state+1  
 // 这里不存在并发问题  
 else if (current == getExclusiveOwnerThread()) {  
 int nextc = c + acquires;  
 if (nextc < 0)  
 throw new Error("Maximum lock count exceeded");  
 setState(nextc);  
 return true;  
 }  
 // 如果到这里，说明前面的if和else if都没有返回true，说明没有获取到锁  
 return false;  
 }

从上面可以看出：

(1) 当一个线程在获取锁过程中，先判断state的值是否为0，如果是表示没有线程持有锁，就可以尝试获取锁（不一定获取成功）。  
(2) 当state的值不为0时，表示锁已经被一个线程占用了，这时会做一个判断current == getExclusiveOwnerThread()（这个方法返回的是当前持有锁的线程），这个判断是看当前持有锁的线程是不是自己，如果是自己，那么将state的值加1就可以了，表示重入返回即可。

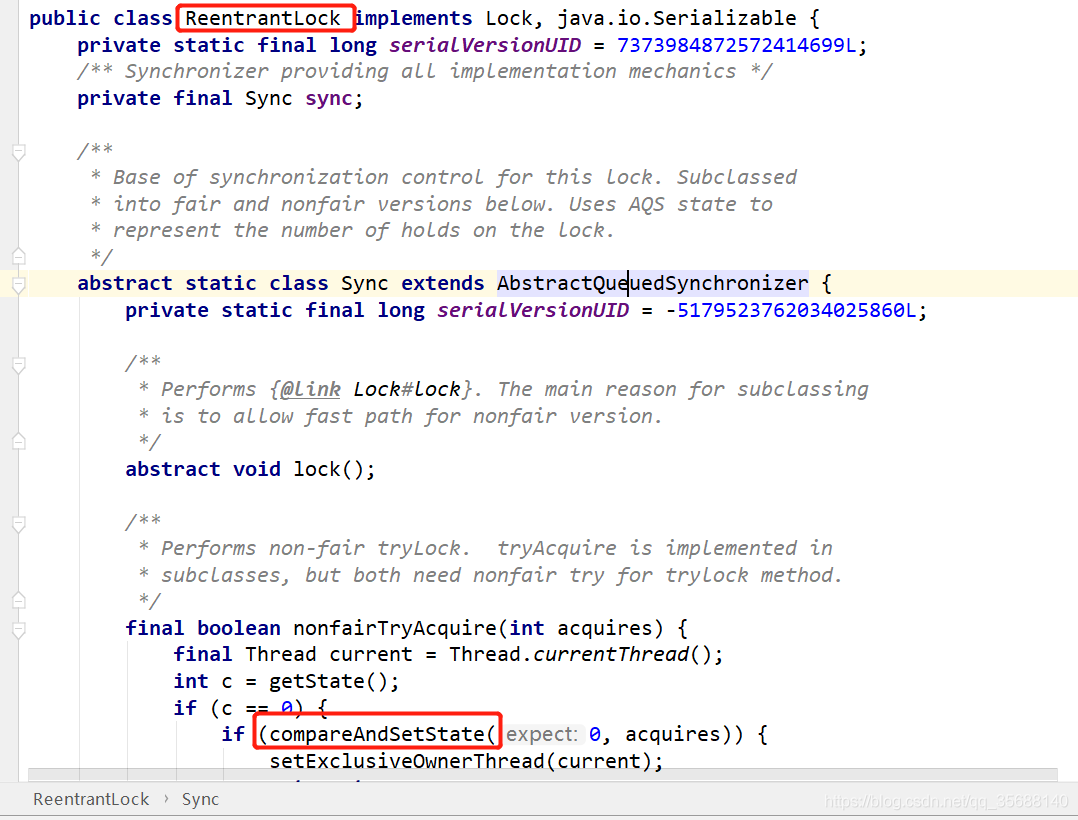
**本文完**

本文参考：《深入理解Java虚拟机第二版》  
 源码分析参考：[一行一行源码分析清楚AbstractQueuedSynchronizer](https://links.jianshu.com/go?to=https%3A%2F%2Fjavadoop.com%2Fpost%2FAbstractQueuedSynchronizer)，写的非常好！！！

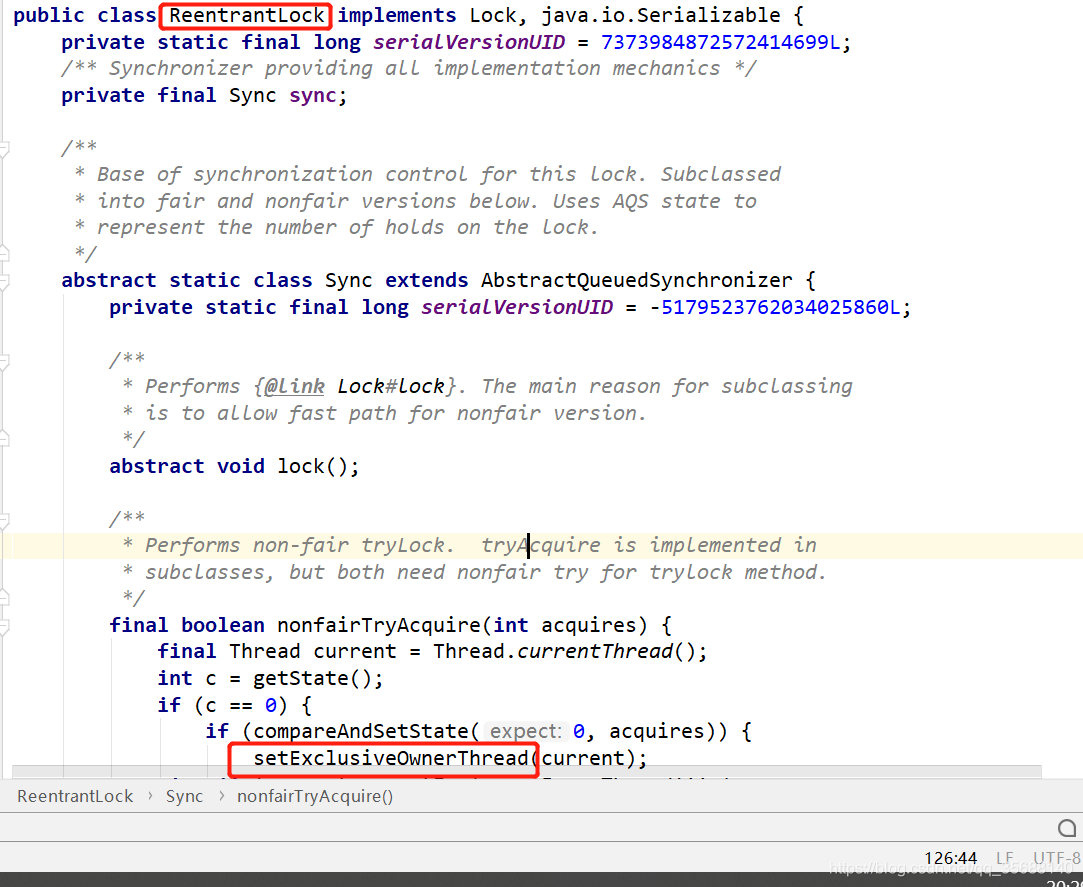
附录：

关于为什么ReentrantLock不是乐观锁的一些猜想https://blog.csdn.net/qq\_35688140/article/details/101223701

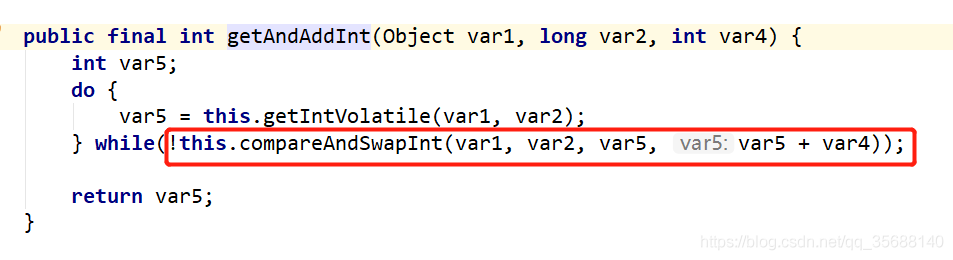
1.众所周知，ReentrantLock是一个悲观锁，但是查看源码，发现底层实现使用的是compareAndSet相关方法实现的，于是产生疑问：为什么ReentrantLock使用的和CAS一样的compareAndSet相关的方法实现的，CAS确实乐观锁，ReentrantLock却是悲观锁？



猜想一：仔细看代码发现ReentrantLock使用了setExclusiveOwnerThread方法，这个方法是将某一个线程设置为独占线程。就是我们常说的互斥锁。该线程占用该方法以后就无法被其他线程占有，也就是线程的互斥。所以这不符合乐观锁的定义：“认为自己在使用数据时不会有别的线程修改数据”。



猜想二：下面是AtomicInetger实现getAndAddInt的源码，实际上是调用了Unsafe.compareAndSwapInt()方法进行判断，但是这是依赖的是自旋锁（一种乐观的思想），他们虽然都是调用了compareAndSet实现的，但是基于的思想不同导致他们一个是乐观的，一个是悲观的。

————————————————  
版权声明：本文为CSDN博主「LUK流」的原创文章，遵循CC 4.0 BY-SA版权协议，转载请附上原文出处链接及本声明。  
原文链接：https://blog.csdn.net/qq\_35688140/article/details/101223701

### 13.除了 ReetrantLock，你还接触过 JUC 中的哪些并发工具？

### 14.请谈谈 ReadWriteLock 和 StampedLock。

### 15.如何让 Java 的线程彼此同步？你了解过哪些同步器？请分别介绍下。

### 16.CyclicBarrier 和 CountDownLatch 看起来很相似，请对比下呢？

### 17.Java 线程池相关问题

### 18.Java 中的线程池是如何实现的？

### 19.创建线程池的几个核心构造参数？

### 20.线程池中的线程是怎么创建的？是一开始就随着线程池的启动创建好的吗？

### 21.既然提到可以通过配置不同参数创建出不同的线程池，那么 Java 中默认实现好的线程池又有哪些呢？请比较它们的异同

### 22.如何在 Java 线程池中提交线程？

### 23.什么是 Java 的内存模型，Java 中各个线程是怎么彼此看到对方的变量的？

### 24.请谈谈 volatile 有什么特点，为什么它能保证变量对所有线程的可见性？

### 25.既然 volatile 能够保证线程间的变量可见性，是不是就意味着基于 volatile 变量的运算就是并发安全的？

### 26.请对比下 volatile 对比 Synchronized 的异同。

### 27.请谈谈 ThreadLocal 是怎么解决并发安全的？

### 28.很多人都说要慎用 ThreadLocal，谈谈你的理解，使用 ThreadLocal 需要注意些什么？

### 附加：29synchronized与lock 对象锁、互斥锁、共享锁以及公平锁和非公平锁

答案https://www.cnblogs.com/20140930zn/p/4016256.html[synchronized与lock 对象锁、互斥锁、共享锁以及公平锁和非公平锁](https://www.cnblogs.com/20140930zn/p/4016256.html)

synchronized与lock 都是用来实现线程同步的锁，synchronized对象锁，lock是一个接口，她的实现有reentrantlock互斥锁以及ReentrantReadWriteLock共享锁。

这里说明一下ReentrantReadWriteLock共享锁，所谓共享就是该锁提供读读锁共享，即可以多个线程共享一个读取锁，但是读写锁以及读读锁是互斥的。

看到网上有好多介绍介绍锁的种类的，有对象锁、互斥锁、共享锁以及公平锁和非公平锁，但是说明都不够详细了然，在这里用直白的说法记录一下，以供查阅，

对象锁：一般指的是synchronized这个锁，这个锁和对象有关，而且每个对象都会有个隐形的监视器，用于线程的同步，和这个对象线程交互的方法一般有wait和notify及notifyall,

调用wait方法，线程进入对象的监视器等待其他线程调用notify或notifyall,需要注意的是wait与notify或notifyall不能精确唤醒特定的线程,关于这个锁的用法，请参考[这篇文章。](http://www.cnblogs.com/20140930zn/p/4015831.html)

互斥锁：字面理解相互排斥的，即两个线程获得锁是互斥的，一个线程获得了锁，其他线程在此线程释放锁之前是不能获得锁的，java提供了一个互斥锁为reentrantlock，该锁是可以重入的

重入就是指，线程获得了锁之后，此线程可以进入任一其他方法中关于这个对象的锁。我们知道synchronized通过wait和notify等实现线程之间的交互，那么reentrantlock是通过

什么实现交互的呢，在这里就引入了api中一个重要的接口**Condition,**condition是什么呢，Condition 将 Object 监视器方法（[wait](http://www.cnblogs.com/java/lang/Object.html#wait())、[notify](http://www.cnblogs.com/java/lang/Object.html#notify()) 和 [notifyAll](http://www.cnblogs.com/java/lang/Object.html#notifyAll())）分解成截然不同的对象，

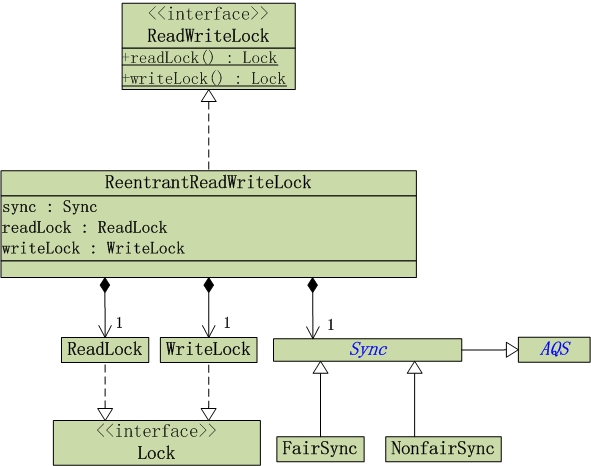
以便通过将这些对象与任意 [Lock](http://www.cnblogs.com/java/util/concurrent/locks/Lock.html) 实现组合使用，为每个对象提供多个等待 set（wait-set）。其中，Lock 替代了 synchronized 方法和语句的使用，Condition 替代了 Object 监视

器方法的使用。这个是官网对condition的解释，可以看出来condition提供的功能和wait与notify是一样的，只不过condition用于lock中，而wait与notify是和synchronized一起使用

，这里需要说明的就是，condition提供的等待与唤醒机制，更明确指出wait哪一个线程，唤醒哪一个线程，关于Reentrantlock与condition的用法，请参考[这篇文章](http://www.cnblogs.com/20140930zn/p/4016264.html)

共享锁：共享锁前面已经讲过，就是多个线程可以共享锁，这里指的就是lock的实现类ReentrantReadWriteLock，ReentrantReadWriteLock分为读锁与写锁，这里锁的伟大之处就是，提供了读写锁分离

，分离的好处就是可以让读锁共享，就是多个线程可以共享读锁，读读共享以及写写互斥以及读写互斥，是这个锁的主要特点。下面是这个类的实现源码结构图:



上图有个AQS，这个是**AbstractQueuedSynchronizer**类的缩写，这个类是lock实现的重点，有兴趣的可以研究一下源码，Sync是AQS的实现，在ReentrantLock以及ReentrantReadWriteLock

类中，都有具体的实现，而FairSync和NonFairSync就是所谓的公平锁以及非公平锁

## 四、spring 25题

### 1.什么是 Spring 框架？Spring 框架有哪些主要模块？

### 2.使用 Spring 框架能带来哪些好处？

### 3.什么是控制反转(IOC)？什么是依赖注入？

### 4.请解释下 Spring 框架中的 IoC？

### 5.BeanFactory 和 ApplicationContext 有什么区别？

### 6.Spring 有几种配置方式？

### 7.如何用基于 XML 配置的方式配置 Spring？

### 8.如何用基于 Java 配置的方式配置 Spring？

### 9.怎样用注解的方式配置 Spring？

### 10.请解释 Spring Bean 的生命周期？

### 11.Spring Bean 的作用域之间有什么区别？

### 12.什么是 Spring inner beans？

### 13.Spring 框架中的单例 Beans 是线程安全的么？

### 14.请举例说明如何在 Spring 中注入一个 Java Collection？

### 15.如何向 Spring Bean 中注入一个 Java.util.Properties？

### 16.请解释 Spring Bean 的自动装配？

### 17.请解释自动装配模式的区别？

### 18.如何开启基于注解的自动装配？

### 19.请举例解释@Required 注解？

### 20.请举例解释@Autowired 注解？

### 21.请举例说明@Qualifier 注解？

### 22.构造方法注入和设值注入有什么区别？

### 23.Spring 框架中有哪些不同类型的事件？

### 24.FileSystemResource 和 ClassPathResource 有何区别？

### 25.Spring 框架中都用到了哪些设计模式？

## 五、设计模式 10题

### 1.请列举出在 JDK 中几个常用的设计模式？

### 2.什么是设计模式？你是否在你的代码里面使用过任何设计模式？

### 3.Java 中什么叫单例设计模式？请用 Java 写出线程安全的单例模式

### 4.在 Java 中，什么叫观察者设计模式（observer design pattern）？

### 5.使用工厂模式最主要的好处是什么？在哪里使用？

### 6.举一个用 Java 实现的装饰模式(decorator design pattern)？它是作用于对象层次还是类层次？

### 7.在 Java 中，为什么不允许从静态方法中访问非静态变量？

### 8.设计一个 ATM 机，请说出你的设计思路？

### 9.在 Java 中，什么时候用重载，什么时候用重写？

### 10.举例说明什么情况下会更倾向于使用抽象类而不是接口

## 六、springboot 22题

### 1.什么是 Spring Boot？

### 2.Spring Boot 有哪些优点？

### 3.什么是 JavaConfig？

### 4.如何重新加载 Spring Boot 上的更改，而无需重新启动服务器？

### 5.Spring Boot 中的监视器是什么？

### 6.如何在 Spring Boot 中禁用 Actuator 端点安全性？

### 7.如何在自定义端口上运行 Spring Boot 应用程序？

### 8.什么是 YAML？

### 9.如何实现 Spring Boot 应用程序的安全性？

### 10.如何集成 Spring Boot 和 ActiveMQ？

### 11.如何使用 Spring Boot 实现分页和排序？

### 12.什么是 Swagger？你用 Spring Boot 实现了它吗？

### 13.什么是 Spring Profiles？

### 14.什么是 Spring Batch？

### 15.什么是 FreeMarker 模板？

### 16.如何使用 Spring Boot 实现异常处理？

### 17.您使用了哪些 starter maven 依赖项？

### 18.什么是 CSRF 攻击？

### 19.什么是 WebSockets？

### 20.什么是 AOP？

### 21.什么是 Apache Kafka？

### 22.我们如何监视所有 Spring Boot 微服务？

## 七、Redis 16题

### 1.什么是redis?

### 2.Reids的特点

### 3.Redis支持的数据类型

### 4.Redis是单进程单线程的

### 5.redis的虚拟内存

### 6.Redis锁

### 7.读写分离模型

### 8.数据分片模型

### 9.Redis的回收策略

### 10.使用Redis有哪些好处？

### 11.redis相比memcached有哪些优势？

### 12.redis常见性能问题和解决方案

### 13.MySQL里有2000w数据，redis中只存20w的数据，如何保证redis中的数据都是热点数据

### 14.Memcache与Redis的区别都有哪些？

### 15.Redis 常见的性能问题都有哪些？如何解决？

### 16.Redis 最适合的场景

## 八、Netty10题

### 1.BIO、NIO和AIO的区别？

### 2.NIO的组成？

### 3.Netty的特点？

### 4.Netty的线程模型？

### 5.TCP 粘包/拆包的原因及解决方法？

### 6.了解哪几种序列化协议？

### 7.如何选择序列化协议？

### 8.Netty的零拷贝实现？

### 9.Netty的高性能表现在哪些方面？

### 10.NIOEventLoopGroup源码？

欢迎加入我的知识星球，一起探讨架构，交流源码。加入方式，长按下方二维码噢：

已在知识星球更新源码解析如下：

最近更新《芋道 SpringBoot 2.X 入门》系列，已经 20 余篇，覆盖了 MyBatis、Redis、MongoDB、ES、分库分表、读写分离、SpringMVC、Webflux、权限、WebSocket、Dubbo、RabbitMQ、RocketMQ、Kafka、性能测试等等内容。  
提供近 3W 行代码的 SpringBoot 示例，以及超 4W 行代码的电商微服务项目。  
获取方式：点“在看”，关注公众号并回复 666 领取，更多内容陆续奉上。  
如果你喜欢这篇文章，喜欢，转发。  
生活很美好，明天见(｡･ω･｡)ﾉ♡  
阅读原文  
阅读 4931  
在看67