Android 开发工程师面试指南

deepwaterooo

March 30, 2018

Contents

1 国内一线互联网公司内部面试题库

1.1 java

1.1.1 接口的意义-百度

- 规范、扩展、回调

1.1.2 抽象类的意义-乐视

- 为其子类提供一个公共的类型封装子类中得重复内容定义抽象方法，子类虽然有不同的实现但是定义是一致的

1.1.3 内部类的作用-乐视

- 内部类可以用多个实例，每个实例都有自己的状态信息，并且与其他外围对象的信息相互独立。
- 在单个外围类中，可以让多个内部类以不同的方式实现同一个接口，或者继承同一个类。
- 创建内部类对象的时刻并不依赖于外围类对象的创建。
- 内部类并没有令人迷惑的“is-a”关系，他就是一个独立的实体。
- 内部类提供了更好的封装，除了该外围类，其他类都不能访问

1.1.4 父类的静态方法能否被子类重写-猎豹

- 不能
- 子类继承父类后，用相同的静态方法和非静态方法，这时非静态方法覆盖父类中的方法（即方法重写），父类的该静态方法被隐藏（如果对象是父类则调用该隐藏的方法），另外子类可继承父类的静态与非静态方法，至于方法重载我觉得它其中一要素就是在同一类中，不能说父类中的什么方法与子类里的什么方法是方法重载的体现

1.1.5 java 排序算法-美团

- http://blog.csdn.net/qy1387/article/details/7752973

1.1.6 列举 java 的集合和继承关系-百度-美团

1.1.7 java 虚拟机的特性-百度-乐视

- Java 语言的一个非常重要的特点就是与平台的无关性。而使用 Java 虚拟机是实现这一特点的关键。一般的高级语言如果要在不同的平台上运行，至少需要编译成不同的目标代码。而引入 Java 语言虚拟机后，Java 语言在不同平台上运行时不需要重新编译。Java 语言使用模式 Java 虚拟机屏蔽了与具体平台相关的信息，使得 Java 语言编译程序只需生成在 Java 虚拟机上运行的目标代码（字节码），就可以在多种平台上不加修改地运行。Java 虚拟机在执行字节码时，把字节码解释成具体平台上的机器指令执行。

1.1.8 哪些情况下的对象会被垃圾回收机制处理掉-美团-小米

- Java 垃圾回收机制最基本的做法是分代回收。内存中的区域被划分成不同的世代，对象根据其存活的时间被保存在对应世代的区域中。一般的实现是划分成 3 个世代：年轻、年老和永久。内存的分配是发生在年轻世代中的。当一个对象存活时间足够长的时候，它就会被复制到年老世代中。对于不同的世代可以使用不同的垃圾回收算法。进行世代划分的出发点是对应用中对象存活时间进行研究之后得出的统计规律。一般来说，一个应用中的大部分对象的存活时间都很短。比如局部变量的存活时间就只在方法的执行过程中。基于这一点，对于年轻世代的垃圾回收算法就可以很有针对性。

1.1.9 进程和线程的区别-猎豹-美团

- 简而言之, 一个程序至少有一个进程, 一个进程至少有一个线程。
- 线程的划分尺度小于进程，使得多线程程序的并发性高。
- 另外，进程在执行过程中拥有独立的内存单元，而多个线程共享内存，从而极大地提高了程序的运行效率。
- 线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。
- 从逻辑角度来看，多线程的意义在于一个应用程序中，有多个执行部分可以同时执行。但操作系统并没有将多个线程看做多个独立的应用，来实现进程的调度和管理以及资源分配。这就是进程和线程的重要区别。
- 进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动, 进程是系统进行资源分配和调度的一个独立单位.

- 线程是进程的一个实体, 是 CPU 调度和分派的基本单位, 它是比进程更小的能独立运行的基本单位. 线程自己基本上不拥有系统资源, 只拥有一点在运行中必不可少的资源 (如程序计数器, 一组寄存器和栈), 但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源.
- 一个线程可以创建和撤销另一个线程; 同一个进程中的多个线程之间可以并发执行.
- 进程和线程的主要差别在于它们是不同的操作系统资源管理方式。进程有独立的地址空间，一个进程崩溃后，在保护模式下不会对其它进程产生影响，而线程只是一个进程中的不同执行路径。线程有自己的堆栈和局部变量，但线程之间没有单独的地址空间，一个线程死掉就等于整个进程死掉，所以多进程的程序要比多线程的程序健壮，但在进程切换时，耗费资源较大，效率要差一些。但对于一些要求同时进行并且又要共享某些变量的并发操作，只能用线程，不能用进程。如果有兴趣深入的话，我建议你们看看《现代操作系统》或者《操作系统的设计与实现》。对就个问题说得比较清楚。

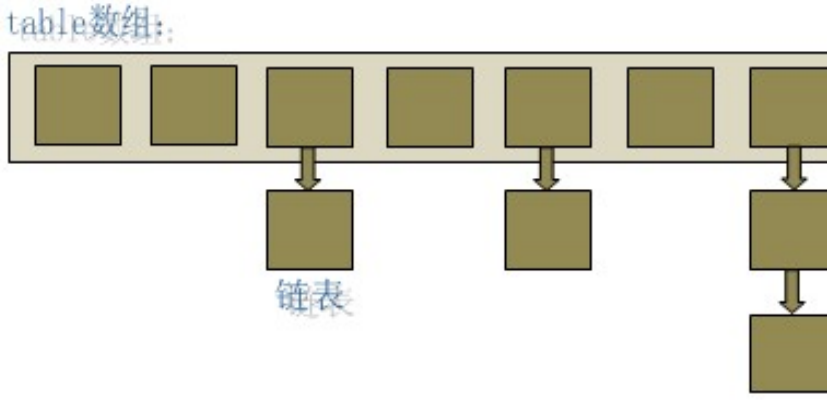
1.1.10 java 中 == 和 equals 和 hashCode 的区别-乐视

- http://blog.csdn.net/tiantiandjava/article/details/46988461

1.1.11 常见的排序算法时间复杂度-小米

1.1.12 HashMap 的实现原理-美团

- HashMap 概述: HashMap 是基于哈希表的 Map 接口的非同步实现。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。
- HashMap 的数据结构:在 java 编程语言中,最基本的结构就是两种,一个是数组,另外一个是模拟指针(引用),所有的数据结构都可以用这两个基本结构来构造的, HashMap 也不例外。HashMap 实际上是一个“链表散列”



的数据结构,即数组和链表的结合体。

- 从上图中可以看出, HashMap 底层就是一个数组结构, 数组中的每一项又是一个链表。当新建一个 HashMap 的时候，就会初始化一个数组。

1.1.13 状态机

- http://www.jdon.com/designpatterns/designpattern_State.htm

1.1.14 int-char-long 各占多少字节数

- byte 位数 8 字节数 1
- short 16 2
- int 32 4
- long 64 8
- float 32 4
- double 64 8
- char 16 2

1.1.15 int 与 integer 的区别

- http://www.cnblogs.com/shenliang123/archive/2011/10/27/2226903.html

1.1.16 string-stringbuffer-stringbuilder 区别-小米-乐视-百度

- String 字符串常量
- StringBuffer 字符串变量（线程安全）
- StringBuilder 字符串变量（非线程安全）

1. String

   - 简要的说，String 类型和 StringBuffer 类型的主要性能区别其实在于 String 是不可变的对象，因此在每次对 String 类型进行改变的时候其实都等同于生成了一个新的 String 对象，然后将指针指向新的 String 对象，所以经常改变内容的字符串最好不要用 String ，因为每次生成对象都会对系统性能产生影响，特别当内存中无引用对象多了以后,JVM 的 GC 就会开始工作，那速度是一定会相当慢的。
   - 而如果是使用 StringBuffer 类则结果就不一样了，每次结果都会对 StringBuffer 对象本身进行操作，而不是生成新的对象，再改变对象引用。所以在一般情况下我们推荐使用 StringBuffer ，特别是字符串对象经常改变的情况下。而在某些特别情况下，String 对象的字符串拼接其实是被 JVM 解释成了 StringBuffer 对象的拼接，所以这些时候 String 对象的速度并不会比 StringBuffer 对象慢，而特别是以下的字符串对象生成中，String 效率是远要比 StringBuffer 快的：

```
String S1 = "This is only a" + "simple" + " test";
StringBuffer Sb = new StringBuffer("This is only a").append("simple").append("test");
```

   - 你会很惊讶的发现，生成 String S1 对象的速度简直太快了，而这个时候 StringBuffer 居然速度上根本一点都不占优势。其实这是 JVM 的一个把戏，在 JVM 眼里，这个 String S1 = “This is only a”+ “simple”+ “test”; 其实就是：String S1 = “This is only a simple test”; 所以当然不需要太多的时间了。但大家这里要注意的是，如果你的字符串是来自另外的 String 对象的话，速度就没那么快了，譬如：String S2 = “This is only a”; String S3 = “simple”; String S4 = “test”; String S1 = S2 +S3 + S4; 这时候 JVM 会规规矩矩的按照原来的方式去做
   - 在大部分情况下 StringBuffer > String

2. StringBuffer

   - Java.lang.StringBuffer 线程安全的可变字符序列。一个类似于 String 的字符串缓冲区，但不能修改。虽然在任意时间点上它都包含某种特定的字符序列，但通过某些方法调用可以改变该序列的长度和内容。
   - 可将字符串缓冲区安全地用于多个线程。可以在必要时对这些方法进行同步，因此任意特定实例上的所有操作就好像是以串行顺序发生的，该顺序与所涉及的每个线程进行的方法调用顺序一致。
   - StringBuffer 上的主要操作是 append 和 insert 方法，可重载这些方法，以接受任意类型的数据。每个方法都能有效地将给定的数据转换成字符串，然后将该字符串的字符追加或插入到字符串缓冲区中。append 方法始终将这些字符添加到缓冲区的末端；而 insert 方法则在指定的点添加字符。
   - 例如，如果 z 引用一个当前内容是“start”的字符串缓冲区对象，则此方法调用 z.append(”le”) 会使字符串缓冲区包含“startle”，而 z.insert(4, ”le”) 将更改字符串缓冲区，使之包含“starlet”。
   - 在大部分情况下 StringBuilder > StringBuffer

3. java.lang.StringBuilder

   - java.lang.StringBuilder 一个可变的字符序列是 5.0 新增的。此类提供一个与 StringBuffer 兼容的 API，但不保证同步。该类被设计用作 StringBuffer 的一个简易替换，用在字符串缓冲区被单个线程使用的时候（这种情况很普遍）。如果可能，建议优先采用该类，因为在大多数实现中，它比 StringBuffer 要快。两者的方法基本相同

1.1.17 java 多态-乐视

- Java 多态性理解
- Java 中多态性的实现
- 什么是多态
- 面向对象的三大特性：封装、继承、多态。从一定角度来看，封装和继承几乎都是为多态而准备的。这是我们最后一个概念，也是最重要的知识点。
- 多态的定义：指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）
- 实现多态的技术称为：动态绑定（dynamic binding），是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。

1. 多态的作用：消除类型之间的耦合关系
   - 现实中，关于多态的例子不胜枚举。比方说按下 F1 键这个动作，如果当前在 Flash 界面下弹出的就是 AS 3 的帮助文档；如果当前在 Word 下弹出的就是 Word 帮助；在 Windows 下弹出的就是 Windows 帮助和支持。同一个事件发生在不同的对象上会产生不同的结果。下面是多态存在的三个必要条件，要求大家做梦时都能背出来！
   - 多态存在的三个必要条件一、要有继承；二、要有重写；三、父类引用指向子类对象。
2. 多态的好处：
   - 1. 可替换性（substitutability）。多态对已存在代码具有可替换性。例如，多态对圆 Circle 类工作，对其他任何圆形几何体，如圆环，也同样工作。
   - 2. 可扩充性（extensibility）。多态对代码具有可扩充性。增加新的子类不影响已存在类的多态性、继承性，以及其他特性的运行和操作。实际上新加子类更容易获得多态功能。例如，在实现了圆锥、半圆锥以及半球体的多态基础上，很容易增添球体类的多态性。
   - 3. 接口性（interface-ability）。多态是超类通过方法签名，向子类提供了一个共同接口，由子类来完善或者覆盖它而实现的。如图 8.3 所示。图中超类 Shape 规定了两个实现多态的接口方法，computeArea() 以及 computeVolume()。子类，如 Circle 和 Sphere 为了实现多态，完善或者覆盖这两个接口方法。
   - 4. 灵活性（flexibility）。它在应用中体现了灵活多样的操作，提高了使用效率。
   - 5. 简化性（simplicity）。多态简化对应用软件的代码编写和修改过程，尤其在处理大量对象的运算和操作时，这个特点尤为突出和重要。
3. Java 中多态的实现方式：
   - 接口实现，继承父类进行方法重写，同一个类中进行方法重载。

1.1.18 什么导致线程阻塞-58-美团

- 线程的阻塞
- 为了解决对共享存储区的访问冲突，Java 引入了同步机制，现在让我们来考察多个线程对共享资源的访问，显然同步机制已经不够了，因为在任意时刻所要求的资源不一定已经准备好了被访问，反过来，同一时刻准备好了的资源也可能不止一个。为了解决这种情况下的访问控制问题，Java 引入了对阻塞机制的支持.
- 阻塞指的是暂停一个线程的执行以等待某个条件发生（如某资源就绪），学过操作系统的同学对它一定已经很熟悉了。Java 提供了大量方法来支持阻塞，下面让我们逐一分析。
  - sleep() 方法：sleep() 允许指定以毫秒为单位的一段时间作为参数，它使得线程在指定的时间内进入阻塞状态，不能得到 CPU 时间，指定的时间一过，线程重新进入可执行状态。典型地，sleep() 被用在等待某个资源就绪的情形：测试发现条件不满足后，让线程阻塞一段时间后重新测试，直到条件满足为止。
  - suspend() 和 resume() 方法：两个方法配套使用，suspend() 使得线程进入阻塞状态，并且不会自动恢复，必须其对应的 resume() 被调用，才能使得线程重新进入可执行状态。典型地，suspend() 和 resume() 被用在等待另一个线程产生的结果的情形：测试发现结果还没有产生后，让线程阻塞，另一个线程产生了结果后，调用 resume() 使其恢复。

  - yield() 方法：yield() 使得线程放弃当前分得的 CPU 时间，但是不使线程阻塞，即线程仍处于可执行状态，随时可能再次分得 CPU 时间。调用 yield() 的效果等价于调度程序认为该线程已执行了足够的时间从而转到另一个线程.
  - wait() 和 notify() 方法：两个方法配套使用，wait() 使得线程进入阻塞状态，它有两种形式，一种允许指定以毫秒为单位的一段时间作为参数，另一种没有参数，前者当对应的 notify() 被调用或者超出指定时间时线程重新进入可执行状态，后者则必须对应的 notify() 被调用.

- 初看起来它们与 suspend() 和 resume() 方法对没有什么分别，但是事实上它们是截然不同的。区别的核心在于，前面叙述的所有方法，阻塞时都不会释放占用的锁（如果占用了的话），而这一对方法则相反。

- 上述的核心区别导致了一系列的细节上的区别。

  - 首先，前面叙述的所有方法都隶属于 Thread 类，但是这一对却直接隶属于 Object 类，也就是说，所有对象都拥有这一对方法。初看起来这十分不可思议，但是实际上却是很自然的，因为这一对方法阻塞时要释放占用的锁，而锁是任何对象都具有的，调用任意对象的 wait() 方法导致线程阻塞，并且该对象上的锁被释放。而调用任意对象的 notify() 方法则导致因调用该对象的 wait() 方法而阻塞的线程中随机选择的一个解除阻塞（但要等到获得锁后才真正可执行）。
  - 其次，前面叙述的所有方法都可在任何位置调用，但是这一对方法却必须在 synchronized 方法或块中调用，理由也很简单，只有在 synchronized 方法或块中当前线程才占有锁，才有锁可以释放。同样的道理，调用这一对方法的对象上的锁必须为当前线程所拥有，这样才有锁可以释放。因此，这一对方法调用必须放置在这样的 synchronized 方法或块中，该方法或块的上锁对象就是调用这一对方法的对象。若不满足这一条件，则程序虽然仍能编译，但在运行时会出现 IllegalMonitorStateException 异常。

- wait() 和 notify() 方法的上述特性决定了它们经常和 synchronized 方法或块一起使用，将它们和操作系统的进程间通信机制作一个比较就会发现它们的相似性：synchronized 方法或块提供了类似于操作系统原语的功能，它们的执行不会受到多线程机制的干扰，而这一对方法则相当于 block 和 wakeup 原语（这一对方法均声明为 synchronized）。它们的结合使得我们可以实现操作系统上一系列精妙的进程间通信的算法（如信号量算法），并用于解决各种复杂的线程间通信问题。

- 关于 wait() 和 notify() 方法最后再说明两点：

  - 第一：调用 notify() 方法导致解除阻塞的线程是从因调用该对象的 wait() 方法而阻塞的线程中随机选取的，我们无法预料哪一个线程将会被选择，所以编程时要特别小心，避免因这种不确定性而产生问题。
  - 第二：除了 notify()，还有一个方法 notifyAll() 也可起到类似作用，唯一的区别在于，调用 notifyAll() 方法将把因调用该对象的 wait() 方法而阻塞的所有线程一次性全部解除阻塞。当然，只有获得锁的那一个线程才能进入可执行状态。

- 谈到阻塞，就不能不谈一谈死锁，略一分析就能发现，suspend() 方法和不指定超时期限的 wait() 方法的调用都可能产生死锁。遗憾的是，Java 并不在语言级别上支持死锁的避免，我们在编程中必须小心地避免死锁。

- 以上我们对 Java 中实现线程阻塞的各种方法作了一番分析，我们重点分析了 wait() 和 notify() 方法，因为它们的功能最强大，使用也最灵活，但是这也导致了它们的效率较低，较容易出错。实际使用中我们应该灵活使用各种方法，以便更好地达到我们的目的。

1.1.19 抽象类接口区别-360

- 默认的方法实现抽象类可以有默认的方法实现完全是抽象的。接口根本不存在方法的实现

- 实现子类使用 extends 关键字来继承抽象类。如果子类不是抽象类的话，它需要提供抽象类中所有声明的方法的实现。子类使用关键字 implements 来实现接口。它需要提供接口中所有声明的方法的实现

- 构造器抽象类可以有构造器接口不能有构造器

- 与正常 Java 类的区别除了你不能实例化抽象类之外，它和普通 Java 类没有任何区接口是完全不同的类型

- 访问修饰符抽象方法可以有 public、protected 和 default 这些修饰符接口方法默认修饰符是 public。你不可以使用其它修饰符。

- main 方法抽象方法可以有 main 方法并且我们可以运行它接口没有 main 方法，因此我们不能运行它。

- 多继承抽象类在 java 语言中所表示的是一种继承关系，一个子类只能存在一个父类，但是可以存在多个接口。
- 速度它比接口速度要快接口是稍微有点慢的，因为它需要时间去寻找在类中实现的方法。
- 添加新方法如果你往抽象类中添加新的方法，你可以给它提供默认的实现。因此你不需要改变你现在的代码。如果你往接口中添加方法，那么你必须改变实现该接口的类。

1.1.20 容器类之间的区别-乐视-美团

- http://www.cnblogs.com/yuanermen/archive/2009/08/05/1539917.html http://alexyyek.github.io/2015/04/06/Collection/ http://tianmaying.com/tutorial/java_collection

1.1.21 内部类

- http://www.cnblogs.com/chenssy/p/3388487.html

1.1.22 hashmap 和 hashtable 的区别-乐视-小米

- http://www.233.com/ncre2/JAVA/jichu/20100717/084230917.html

1.1.23 ArrayMap 对比 HashMap

- http://lvable.com/?p=217

1.2 Android

1.2.1 如何导入外部数据库

- 把原数据库包括在项目源码的 res/raw
- android 系统下数据库应该存放在 *data/data/com..*（*package name*） 目录下，所以我们需要做的是把已有的数据库传入那个目录下. 操作方法是用 FileInputStream 读取原数据库，再用 FileOutputStream 把读取到的东西写入到那个目录.

1.2.2 本地广播和全局广播有什么差别

- 因广播数据在本应用范围内传播，不用担心隐私数据泄露的问题。不用担心别的应用伪造广播，造成安全隐患。相比在系统内发送全局广播，它更高效。

1.2.3 intentService 作用是什么,AIDL 解决了什么问题-小米

- 生成一个默认的且与主线程互相独立的工作者线程来执行所有传送至 onStartCommand() 方法的 Intetnt。
- 生成一个工作队列来传送 Intent 对象给你的 onHandleIntent() 方法，同一时刻只传送一个 Intent 对象，这样一来，你就不必担心多线程的问题。在所有的请求 (Intent) 都被执行完以后会自动停止服务，所以，你不需要自己去调用 stopSelf() 方法来停止。
- 该服务提供了一个 onBind() 方法的默认实现，它返回 null
- 提供了一个 onStartCommand() 方法的默认实现，它将 Intent 先传送至工作队列，然后从工作队列中每次取出一个传送至 onHandleIntent() 方法，在该方法中对 Intent 对相应的处理。
- AIDL (Android Interface Definition Language) 是一种 IDL 语言，用于生成可以在 Android 设备上两个进程之间进行进程间通信 (interprocess communication, IPC) 的代码。如果在一个进程中（例如 Activity）要调用另一个进程中（例如 Service）对象的操作，就可以使用 AIDL 生成可序列化的参数。AIDL IPC 机制是面向接口的，像 COM 或 Corba 一样，但是更加轻量级。它是使用代理类在客户端和实现端传递数据。

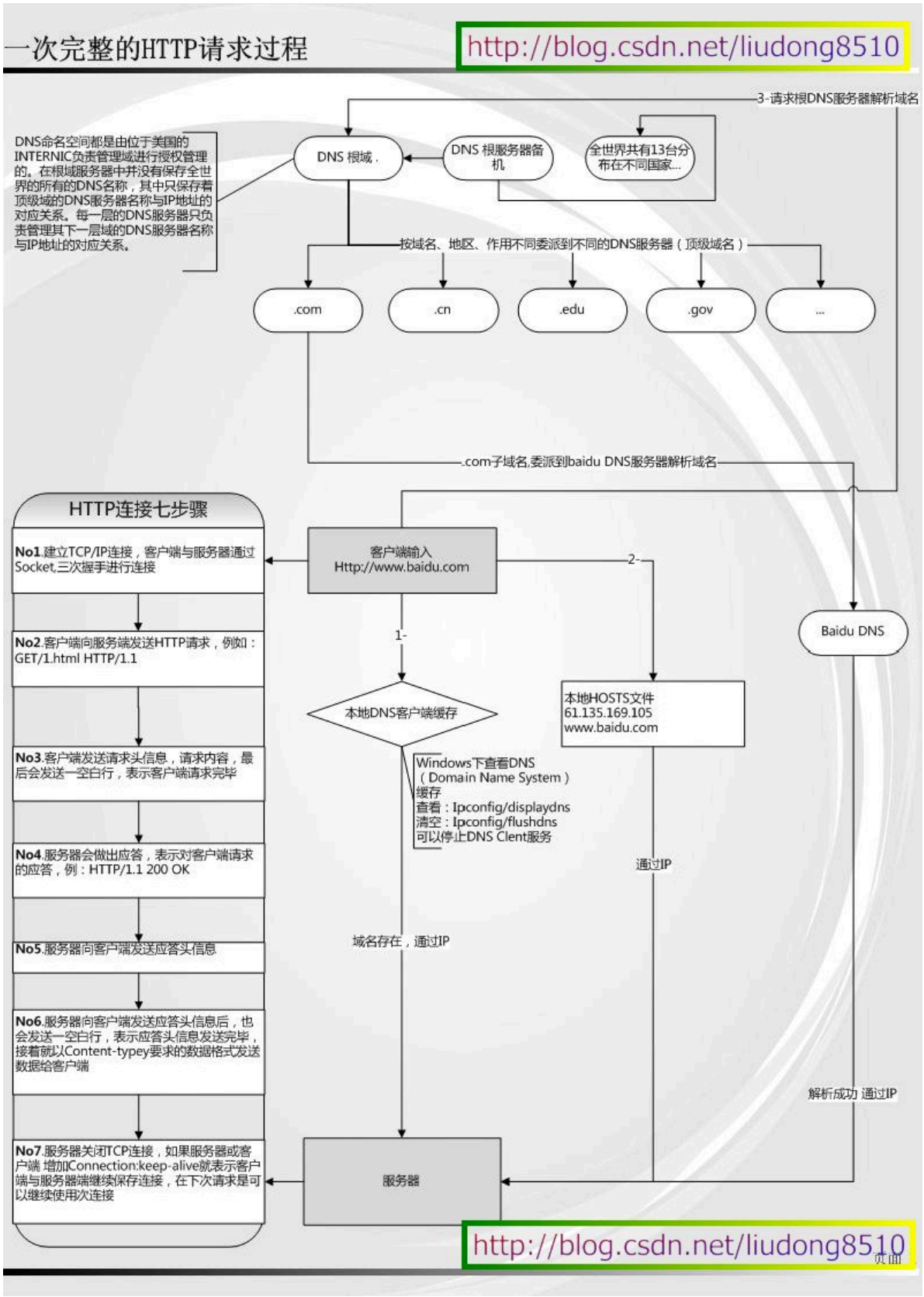
1.2.4 Activity/Window/View 三者的差别,fragment 的特点-360

- Activity 像一个工匠（控制单元），Window 像窗户（承载模型），View 像窗花（显示视图）LayoutInflater 像剪刀，Xml 配置像窗花图纸。
  - 在 Activity 中调用 attach，创建了一个 Window
  - 创建的 window 是其子类 PhoneWindow，在 attach 中创建 PhoneWindow
  - 在 Activity 中调用 setContentView(R.layout.xxx)
  - 其中实际上是调用的 getWindow().setContentView()
  - 调用 PhoneWindow 中的 setContentView 方法
  - 创建 ParentView： 作为 ViewGroup 的子类，实际是创建的 DecorView(作为 FramLayout 的子类）
  - 将指定的 R.layout.xxx 进行填充 通过布局填充器进行填充【其中的 parent 指的就是 DecorView】
  - 调用到 ViewGroup
  - 调用 ViewGroup 的 removeAllView()，先将所有的 view 移除掉
  - 添加新的 view：addView()

1. fragment 特点

   - Fragment 可以作为 Activity 界面的一部分组成出现；
   - 可以在一个 Activity 中同时出现多个 Fragment，并且一个 Fragment 也可以在多个 Activity 中使用；
   - 在 Activity 运行过程中，可以添加、移除或者替换 Fragment；
   - Fragment 可以响应自己的输入事件，并且有自己的生命周期，它们的生命周期会受宿主 Activity 的生命周期影响。

1.2.5 描述一次网络请求的流程-新浪



1.2.6 Handler,Thread 和 HandlerThread 的差别-小米

- http://blog.csdn.net/guolin_blog/article/details/9991569

- http://droidyue.com/blog/2015/11/08/make-use-of-handlerthread/
- 从 Android 中 Thread（java.lang.Thread -> java.lang.Object）描述可以看出，Android 的 Thread 没有对 Java 的 Thread 做任何封装，但是 Android 提供了一个继承自 Thread 的类 HandlerThread（android.os.HandlerThread -> java.lang.Thread），这个类对 Java 的 Thread 做了很多便利 Android 系统的封装。
- android.os.Handler 可以通过 Looper 对象实例化，并运行于另外的线程中，Android 提供了让 Handler 运行于其它线程的线程实现，也是就 HandlerThread。HandlerThread 对象 start 后可以获得其 Looper 对象，并且使用这个 Looper 对象实例 Handler。

1.2.7 低版本 SDK 实现高版本 api-小米

- 自己实现或 @TargetApi annotation

1.2.8 Ubuntu 编译安卓系统-百度

- 进入源码根目录
- . build/envsetup.sh
- lunch
- full(编译全部)
- userdebug(选择编译版本)
- make -j8(开启 8 个线程编译)

1.2.9 LaunchMode 应用场景-百度-小米-乐视

- standard，创建一个新的 Activity。
- singleTop，栈顶不是该类型的 Activity，创建一个新的 Activity。否则，onNewIntent。
- singleTask，回退栈中没有该类型的 Activity，创建 Activity，否则，onNewIntent+ClearTop。
- 注意:
  - 设置了”singleTask” 启动模式的 Activity，它在启动的时候，会先在系统中查找属性值 affinity 等于它的属性值 taskAffinity 的 Task 存在；如果存在这样的 Task，它就会在这个 Task 中启动，否则就会在新的任务栈中启动。因此，如果我们想要设置了”singleTask” 启动模式的 Activity 在新的任务中启动，就要为它设置一个独立的 taskAffinity 属性值。
  - 如果设置了”singleTask” 启动模式的 Activity 不是在新的任务中启动时，它会在已有的任务中查看是否已经存在相应的 Activity 实例，如果存在，就会把位于这个 Activity 实例上面的 Activity 全部结束掉，即最终这个 Activity 实例会位于任务的 Stack 顶端中。
  - 在一个任务栈中只有一个”singleTask”启动模式的 Activity 存在。他的上面可以有其他的 Activity。这点与 singleInstance 是有区别的。
- singleInstance，回退栈中，只有这一个 Activity，没有其他 Activity。
- singleTop 适合接收通知启动的内容显示页面。
- 例如，某个新闻客户端的新闻内容页面，如果收到 10 个新闻推送，每次都打开一个新闻内容页面是很烦人的。
- singleTask 适合作为程序入口点。
- 例如浏览器的主界面。不管从多少个应用启动浏览器，只会启动主界面一次，其余情况都会走 onNewIntent，并且会清空主界面上面的其他页面。
- singleInstance 应用场景:

- 闹铃的响铃界面。你以前设置了一个闹铃：上午 6 点。在上午 5 点 58 分，你启动了闹铃设置界面，并按 Home 键回桌面；在上午 5 点 59 分时，你在微信和朋友聊天；在 6 点时，闹铃响了，并且弹出了一个对话框形式的 Activity(名为 AlarmAlertActivity) 提示你到 6 点了 (这个 Activity 就是以 SingleInstance 加载模式打开的)，你按返回键，回到的是微信的聊天界面，这是因为 AlarmAlertActivity 所在的 Task 的栈只有他一个元素，因此退出之后这个 Task 的栈空了。如果是以 SingleTask 打开 AlarmAlertActivity，那么当闹铃响了的时候，按返回键应该进入闹铃设置界面。

1.2.10 Touch 事件传递流程-小米

http://hanhailong.com/2015/09/24/Android-%E4%B8%89%E5%BC%A0%E5%9B%BE%E6%90%9E%E5%AE%9ATouch%E4%BA%8B%E4%BB%B6%E4%BC%A0%E9%80%92%E6%9C%BA%E5%88%B6/

1.2.11 View 绘制流程-百度

http://www.codekk.com/blogs/detail/54cfab086c4761e5001b253f

1.2.12 多线程-360

- Activity.runOnUiThread(Runnable)
- View.post(Runnable),View.postDelay(Runnable,long)
- Handler
- AsyncTask

1.2.13 线程同步-百度

- http://www.itzhai.com/java-based-notebook-thread-synchronization-problem-solving-synchroni
  html#read-more
- http://www.juwends.com/tech/android/android-inter-thread-comm.html
- 单例

```
public class Singleton {
    private volatile static Singleton mSingleton;
    private Singleton(){
    }
    public static Singleton getInstance(){
        if (mSingleton == null){ // A
            synchronized(Singleton.class){ // C
                if(mSingleton == null)
                    mSingleton = new Singleton(); // B
            }
        }
        return mSingleton;
    }
}
```

1.2.14 什么情况导致内存泄漏-美团

- 1. 资源对象没关闭造成的内存泄漏
  - 描述：资源性对象比如 (Cursor，File 文件等) 往往都用了一些缓冲，我们在不使用的时候，应该及时关闭它们，以便它们的缓冲及时回收内存。它们的缓冲不仅存在于 java 虚拟机内，还存在于 java 虚拟机外。如果我们仅仅是把它的引用设置为 null, 而不关闭它们，往往会造成内存泄漏。因为有些资源性对象，比如 SQLiteCursor(在析构函数 finalize(), 如果我们没有关闭它，它自己会调 close() 关闭)，如果我们没有关闭它，系统在回收它时也会关闭它，但是这样的效率太低了。因此对于资源性对象在不使用的时候，应该调用它的 close() 函数，将其关闭掉，然后才置为 null. 在我们的程序退出时一定要确保我们的资源性对象已经关闭。程序中经常会进行查询数据库的操作，但是经常会有使用完毕 Cursor 后没有关闭的情况。如果我们的查询结果集比较小，对内存的消耗不容易被发现，只有在常时间大量操作的情况下才会复现内存问题，这样就会给以后的测试和问题排查带来困难和风险。

- 2. 构造 Adapter 时，没有使用缓存的 convertView
  - 描述：以构造 ListView 的 BaseAdapter 为例，在 BaseAdapter 中提供了方法：public View getView(int position, ViewconvertView, ViewGroup parent) 来向 ListView 提供每一个 item 所需要的 view 对象。初始时 ListView 会从 BaseAdapter 中根据当前的屏幕布局实例化一定数量的 view 对象，同时 ListView 会将这些 view 对象缓存起来。当向上滚动 ListView 时，原先位于最上面的 list item 的 view 对象会被回收，然后被用来构造新出现的最下面的 list item。这个构造过程就是由 getView() 方法完成的，getView() 的第二个形参 View convertView 就是被缓存起来的 list item 的 view 对象 (初始化时缓存中没有 view 对象则 convertView 是 null)。由此可以看出，如果我们不去使用 convertView，而是每次都在 getView() 中重新实例化一个 View 对象的话，即浪费资源也浪费时间，也会使得内存占用越来越大。ListView 回收 list item 的 view 对象的过程可以查看：android.widget.AbsListView.java –> voidaddScrapView(View scrap) 方法。示例代码：

```
public View getView(int position, ViewconvertView, ViewGroup parent) {
    View view = new Xxx(...);
    ....
    return view;
}
```

  - 修正示例代码：

```
public View getView(int position, ViewconvertView, ViewGroup parent) {
    View view = null;
    if (convertView != null) {
        view = convertView;
        populate(view, getItem(position));
    } else {
        view = new Xxx(...);
    }
    return view;
}
```

- 3.Bitmap 对象不在使用时调用 recycle() 释放内存
  - 描述：有时我们会手工的操作 Bitmap 对象，如果一个 Bitmap 对象比较占内存，当它不在被使用的时候，可以调用 Bitmap.recycle() 方法回收此对象的像素所占用的内存，但这不是必须的，视情况而定。可以看一下代码中的注释：
  - *** • Free up the memory associated with thisbitmap's pixels, and mark the • bitmap as "dead", meaning itwill throw an exception if getPixels() or • setPixels() is called, and will drawnothing. This operation cannot be • reversed, so it should only be called ifyou are sure there are no • further uses for the bitmap. This is anadvanced call, and normally need • not be called, since the normal GCprocess will free up this memory when • there are no more references to thisbitmap. *

- 4. 试着使用关于 application 的 context 来替代和 activity 相关的 context
  - 这是一个很隐晦的内存泄漏的情况。有一种简单的方法来避免 context 相关的内存泄漏。最显著地一个是避免 context 逃出他自己的范围之外。使用 Application context。这个 context 的生存周期和你的应用的生存周期一样长，而不是取决于 activity 的生存周期。如果你想保持一个长期生存的对象，并且这个对象需要一个 context，记得使用 application 对象。你可以通过调用 Context.getApplicationContext() or Activity.getApplication() 来获得。更多的请看这篇文章如何避免 Android 内存泄漏。

- 5. 注册没取消造成的内存泄漏
  - 一些 Android 程序可能引用我们的 Anroid 程序的对象 (比如注册机制)。即使我们的 Android 程序已经结束了，但是别的引用程序仍然还有对我们的 Android 程序的某个对象的引用，泄漏的内存依然不能被垃圾回收。调用 registerReceiver 后未调用 unregisterReceiver。比如：假设我们希望在锁屏界面 (LockScreen) 中，监听系统中的电话服务以获取一些信息 (如信号强度等)，则可以在 LockScreen 中定义一个 PhoneStateListener 的对象，同时将它注册到 TelephonyManager 服务中。对于 LockScreen 对象，当需要显示锁屏界面的时候就会创建一个 LockScreen 对象，而当锁屏界面消失的时候 LockScreen 对象就会被释放掉。但是如果在释放 LockScreen 对象的时候忘记取消我们之前注册的 PhoneStateListener 对象，则会导致 LockScreen 无法被垃圾回收。如果不断的使锁屏界面显示和消失，则最终会由于大量

的 LockScreen 对象没有办法被回收而引起 OutOfMemory, 使得 $system_{process}$ 进程挂掉。虽然有些系统程序，它本身好像是可以自动取消注册的 (当然不及时)，但是我们还是应该在我们的程序中明确的取消注册，程序结束时应该把所有的注册都取消掉。

- 6. 集合中对象没清理造成的内存泄漏
  - 我们通常把一些对象的引用加入到了集合中，当我们不需要该对象时，并没有把它的引用从集合中清理掉，这样这个集合就会越来越大。如果这个集合是 static 的话，那情况就更严重了。

1.2.15 ANR 定位和修正

如果开发机器上出现问题，我们可以通过查看/data/anr/traces.txt 即可，最新的 ANR 信息在最开始部分。

- 主线程被 IO 操作（从 4.0 之后网络 IO 不允许在主线程中）阻塞。
- 主线程中存在耗时的计算
- 主线程中错误的操作，比如 Thread.wait 或者 Thread.sleep 等 Android 系统会监控程序的响应状况，一旦出现下面两种情况，则弹出 ANR 对话框
- 应用在 5 秒内未响应用户的输入事件（如按键或者触摸）
- BroadcastReceiver 未在 10 秒内完成相关的处理
- Service 在特定的时间内无法处理完成 20 秒
- 使用 AsyncTask 处理耗时 IO 操作。
- 使用 Thread 或者 HandlerThread 时,调用 Process.setThreadPriority(Process.THREAD$_{PRIORITY\,BACKGROU}$ 设置优先级，否则仍然会降低程序响应，因为默认 Thread 的优先级和主线程相同。
- 使用 Handler 处理工作线程结果，而不是使用 Thread.wait() 或者 Thread.sleep() 来阻塞主线程。
- Activity 的 onCreate 和 onResume 回调中尽量避免耗时的代码
- BroadcastReceiver 中 onReceive 代码也要尽量减少耗时，建议使用 IntentService 处理。

1.2.16 什么情况导致 oom-乐视-美团

http://www.jcodecraeer.com/a/anzhuokaifa/androidkaifa/2015/0920/3478.html

- 1）使用更加轻量的数据结构
- 2）Android 里面使用 Enum
- 3）Bitmap 对象的内存占用
- 4）更大的图片
- 5）onDraw 方法里面执行对象的创建
- 6）StringBuilder

1.2.17 Service 与 Activity 之间通信的几种方式

- 通过 Binder 对象
- 通过 broadcast(广播) 的形式

1.2.18 Android 各个版本 API 的区别

- http://blog.csdn.net/lijun952048910/article/details/7980562

1.2.19 Android 代码中实现 WAP 方式联网-360

- http://blog.csdn.net/asce1885/article/details/7844159

1.2.20 如何保证 service 在后台不被 Kill

- 一、onStartCommand 方法，返回 START_{STICKY}
  - START_{STICKY} 在运行 onStartCommand 后 service 进程被 kill 后，那将保留在开始状态，但是不保留那些传入的 intent。不久后 service 就会再次尝试重新创建，因为保留在开始状态，在创建 service 后将保证调用 onstartCommand。如果没有传递任何开始命令给 service，那将获取到 null 的 intent。
  - $\text{START}_{NOTSTICKY}$ 在运行 onStartCommand 后 service 进程被 kill 后，并且没有新的 intent 传递给它。Service 将移出开始状态，并且直到新的明显的方法（startService）调用才重新创建。因为如果没有传递任何未决定的 intent 那么 service 是不会启动，也就是期间 onstartCommand 不会接收到任何 null 的 intent。
  - $\text{START}_{REDELIVERINTENT}$ 在运行 onStartCommand 后 service 进程被 kill 后，系统将会再次启动 service，并传入最后一个 intent 给 onstartCommand。直到调用 stopSelf(int) 才停止传递 intent。如果在被 kill 后还有未处理好的 intent，那被 kill 后服务还是会自动启动。因此 onstartCommand 不会接收到任何 null 的 intent。
- 二、提升 service 优先级
  - 在 AndroidManifest.xml 文件中对于 intent-filter 可以通过 android:priority = "1000" 这个属性设置最高优先级，1000 是最高值，如果数字越小则优先级越低，同时适用于广播。
- 三、提升 service 进程优先级 Android 中的进程是托管的，当系统进程空间紧张的时候，会依照优先级自动进行进程的回收。Android 将进程分为 6 个等级，它们按优先级顺序由高到低依次是:
  - 前台进程（ FOREGROUND_{APP}）
  - 可视进程 (VISIBLE_{APP})
  - 次要服务进程 ($\text{SECONDARY}_{SERVER}$)
  - 后台进程 (HIDDEN_{APP})
  - 内容供应节点 ($\text{CONTENT}_{PROVIDER}$)
  - 空进程 (EMPTY_{APP})

  当 service 运行在低内存的环境时，将会 kill 掉一些存在的进程。因此进程的优先级将会很重要，可以使用 startForeground 将 service 放到前台状态。这样在低内存时被 kill 的几率会低一些。
- 四、onDestroy 方法里重启 service
  - service +broadcast 方式，就是当 service 走 ondestory 的时候，发送一个自定义的广播，当收到广播的时候，重新启动 service;
- 五、Application 加上 Persistent 属性
- 六、监听系统广播判断 Service 状态
  - 通过系统的一些广播，比如：手机重启、界面唤醒、应用状态改变等等监听并捕获到，然后判断我们的 Service 是否还存活，别忘记加权限啊。

1.2.21 Requestlayout,onlayout,onDraw,DrawChild 区别与联系-猎豹

- requestLayout() 方法：会导致调用 measure() 过程和 layout() 过程。将会根据标志位判断是否需要 ondraw
- onLayout() 方法 (如果该 View 是 ViewGroup 对象，需要实现该方法，对每个子视图进行布局)
- 调用 onDraw() 方法绘制视图本身 (每个 View 都需要重载该方法，ViewGroup 不需要实现该方法)
- drawChild() 去重新回调每个子视图的 draw() 方法

1.2.22 invalidate() 和 postInvalidate() 的区别及使用-百度

http://blog.csdn.net/mars2639/article/details/6650876

1.2.23 Android 动画框架实现原理

- Animation 框架定义了透明度，旋转，缩放和位移几种常见的动画，而且控制的是整个 View，实现原理是每次绘制视图时 View 所在的 ViewGroup 中的 drawChild 函数获取该 View 的 Animation 的 Transformation 值，然后调用 canvas.concat(transformToApply.getMatrix())，通过矩阵运算完成动画帧，如果动画没有完成，继续调用 invalidate() 函数，启动下次绘制来驱动动画，动画过程中的帧之间间隙时间是绘制函数所消耗的时间，可能会导致动画消耗比较多的 CPU 资源，最重要的是，动画改变的只是显示，并不能相应事件。

1.2.24 Android 为每个应用程序分配的内存大小是多少-美团

- android 程序内存一般限制在 16M，也有的是 24M

1.2.25 View 刷新机制-百度-美团

- 由 ViewRoot 对象的 performTraversals() 方法调用 draw() 方法发起绘制该 View 树，值得注意的是每次发起绘图时，并不会重新绘制每个 View 树的视图，而只会重新绘制那些“需要重绘”的视图，View 类内部变量包含了一个标志位 DRAWN，当该视图需要重绘时，就会为该 View 添加该标志位。
- 调用流程:
- mView.draw() 开始绘制，draw() 方法实现的功能如下:
  - 绘制该 View 的背景
  - 为显示渐变框做一些准备操作 (见 5，大多数情况下，不需要改渐变框)
  - 调用 onDraw() 方法绘制视图本身 (每个 View 都需要重载该方法，ViewGroup 不需要实现该方法)
  - 调用 dispatchDraw () 方法绘制子视图 (如果该 View 类型不为 ViewGroup，即不包含子视图，不需要重载该方法) 值得说明的是，ViewGroup 类已经为我们重写了 dispatchDraw () 的功能实现，应用程序一般不需要重写该方法，但可以重载父类函数实现具体的功能。

1.2.26 LinearLayout 和 RelativeLayout 性能对比-百度

- RelativeLayout 会让子 View 调用 2 次 onMeasure，LinearLayout 在有 weight 时，也会调用子 View2 次 onMeasure
- RelativeLayout 的子 View 如果高度和 RelativeLayout 不同，则会引发效率问题，当子 View 很复杂时，这个问题会更加严重。如果可以，尽量使用 padding 代替 margin。
- 在不影响层级深度的情况下, 使用 LinearLayout 和 FrameLayout 而不是 RelativeLayout。
- 最后再思考一下文章开头那个矛盾的问题，为什么 Google 给开发者默认新建了个 RelativeLayout，而自己却在 DecorView 中用了个 LinearLayout。因为 DecorView 的层级深度是已知而且固定的，上面一个标题栏，下面一个内容栏。采用 RelativeLayout 并不会降低层级深度，所以此时在根节点上用 LinearLayout 是效率最高的。而之所以给开发者默认新建了个 RelativeLayout 是希望开发者能采用尽量少的 View 层级来表达布局以实现性能最优，因为复杂的 View 嵌套对性能的影响会更大一些。

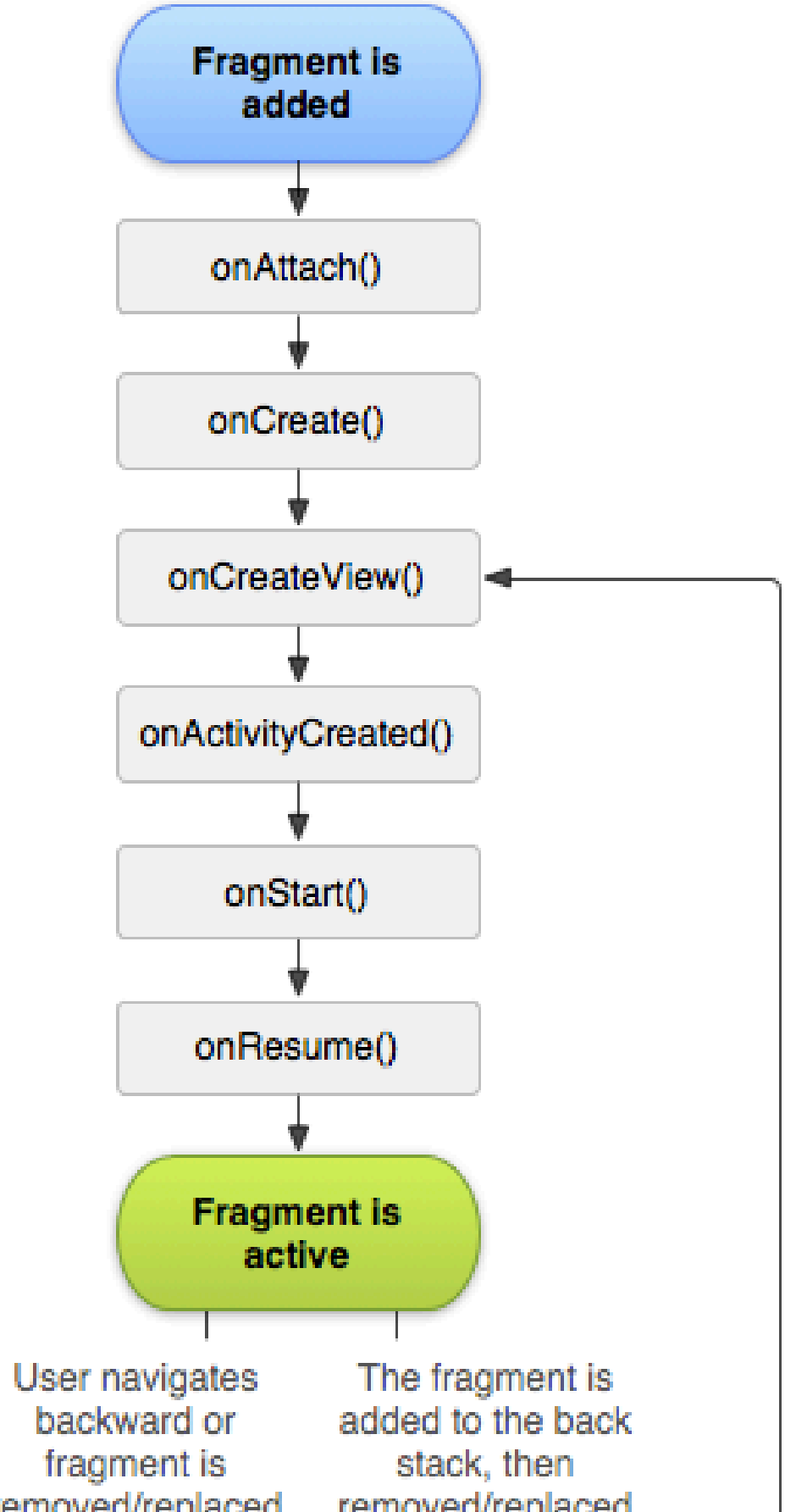
1.2.27 优化自定义 view 百度-乐视-小米

- 为了加速你的 view，对于频繁调用的方法，需要尽量减少不必要的代码。先从 onDraw 开始，需要特别注意不应该在这里做内存分配的事情，因为它会导致 GC，从而导致卡顿。在初始化或者动画间隙期间做分配内存的动作。不要在动画正在执行的时候做内存分配的事情。
- 你还需要尽可能的减少 onDraw 被调用的次数，大多数时候导致 onDraw 都是因为调用了 invalidate(). 因此请尽量减少调用 invaildate() 的次数。如果可能的话，尽量调用含有 4 个参数的 invalidate() 方法而不是没有参数的 invalidate()。没有参数的 invalidate 会强制重绘整个 view。
- 另外一个非常耗时的操作是请求 layout。任何时候执行 requestLayout()，会使得 Android UI 系统去遍历整个 View 的层级来计算出每一个 view 的大小。如果找到有冲突的值，它会需要重新计算好几次。另外需要尽量保持 View 的层级是扁平化的，这样对提高效率很有帮助。
- 如果你有一个复杂的 UI，你应该考虑写一个自定义的 ViewGroup 来执行他的 layout 操作。与内置的 view 不同，自定义的 view 可以使得程序仅仅测量这一部分，这避免了遍历整个 view 的层级结构来计算大小。这个 PieChart 例子展示了如何继承 ViewGroup 作为自定义 view 的一部分。PieChart 有子 views，但是它从来不测量它们。而是根据他自身的 layout 法则，直接设置它们的大小。

1.2.28 ContentProvider-乐视

http://blog.csdn.net/coder_pig/article/details/47858489

1.2.29 Fragment 生命周期



1.2.30 volley 解析-美团-乐视

http://a.codekk.com/detail/Android/grumoon/Volley%20%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90

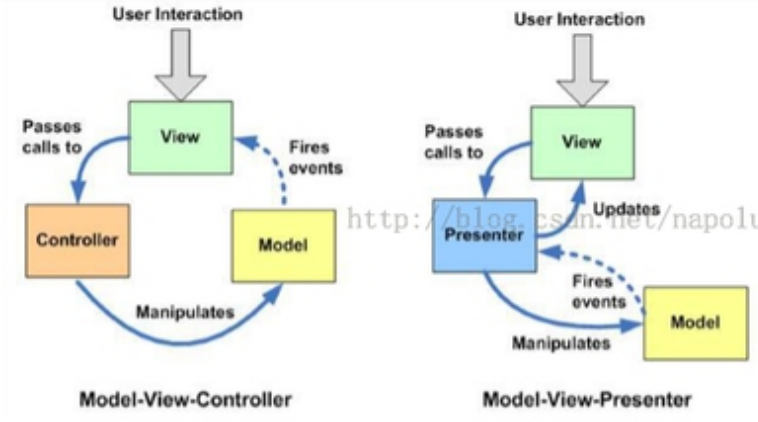
1.2.31 Glide 源码解析

http://www.lightskystreet.com/2015/10/12/glide_source_analysis/ http://frodoking.github.io/2015/10/10/android-glide/

1.2.32 Android 设计模式

http://blog.csdn.net/bboyfeiyu/article/details/44563871

1.2.33 架构设计-搜狐



http://www.tianmaying.com/tutorial/AndroidMVC

1.2.34 Android 属性动画特性-乐视-小米

- 如果你的需求中只需要对 View 进行移动、缩放、旋转和淡入淡出操作，那么补间动画确实已经足够健全了。但是很显然，这些功能是不足以覆盖所有的场景的，一旦我们的需求超出了移动、缩放、旋转和淡入淡出这四种对 View 的操作，那么补间动画就不能再帮我们忙了，也就是说它在功能和可扩展方面都有相当大的局限性，那么下面我们就来看看补间动画所不能胜任的场景。
- 注意上面我在介绍补间动画的时候都有使用“对 View 进行操作”这样的描述，没错，补间动画是只能够作用在 View 上的。也就是说，我们可以对一个 Button、TextView、甚至是 LinearLayout、或者其它任何继承自 View 的组件进行动画操作，但是如果我们想要对一个非 View 的对象进行动画操作，抱歉，补间动画就帮不上忙了。可能有的朋友会感到不能理解，我怎么会需要对一个非 View 的对象进行动画操作呢？这里我举一个简单的例子，比如说我们有一个自定义的 View，在这个 View 当中有一个 Point 对象用于管理坐标，然后在 onDraw() 方法当中就是根据这个 Point 对象的坐标值来进行绘制的。也就是说，如果我们可以对 Point 对象进行动画操作，那么整个自定义 View 的动画效果就有了。显然，补间动画是不具备这个功能的，这是它的第一个缺陷。
- 然后补间动画还有一个缺陷，就是它只能够实现移动、缩放、旋转和淡入淡出这四种动画操作，那如果我们希望可以对 View 的背景色进行动态地改变呢？很遗憾，我们只能靠自己去实现了。说白了，之前的补间动画机制就是使用硬编码的方式来完成的，功能限定死就是这些，基本上没有任何扩展性可言。
- 最后，补间动画还有一个致命的缺陷，就是它只是改变了 View 的显示效果而已，而不会真正去改变 View 的属性。什么意思呢？比如说，现在屏幕的左上角有一个按钮，然后我们通过补间动画将它移动到了屏幕的右下角，现在你可以去尝试点击一下这个按钮，点击事件是绝对不会触发的，因为实际上这个按钮还是停留在屏幕的左上角，只不过补间动画将这个按钮绘制到了屏幕的右下角而已。

1.3 专题

1.3.1 性能优化

1. Android 性能优化典范 - 第 1 季

- Render Performance Android 系统每隔 16ms 发出 VSYNC 信号，触发对 UI 进行渲染，如果每次渲染都成功，这样就能够达到流畅的画面所需要的 60fps，为了能够实现 60fps，这意味着程序的大多数操作都必须在 16ms 内完成。我们可以通过一些工具来定位问题，比如可以使用 HierarchyViewer 来查找 Activity 中的布局是否过于复杂，也可以使用手机设置里面的开发者选项，打开 Show GPU Overdraw 等选项进行观察。你还可以使用 TraceView 来观察 CPU 的执行情况，更加快捷的找到性能瓶颈。
- Understanding Overdraw Overdraw(过度绘制) 描述的是屏幕上的某个像素在同一帧的时间内被绘制了多次。在多层次的 UI 结构里面，如果不可见的 UI 也在做绘制的操作，这就会导致某些像素区域被绘制了多次。这就浪费大量的 CPU 以及 GPU 资源。Overdraw 有时候是因为你的 UI 布局存在大量重叠的部分，还有的时候是因为非必须的重叠背景。例如某个 Activity 有一个背景，然后里面的 Layout 又有自己的背景，同时子 View 又分别有自己的背景。仅仅是通过移除非必须的背景图片，这就能够减少大量的红色 Overdraw 区域，增加蓝色区域的占比。这一措施能够显著提升程序性能。
- Understanding VSYNC Refresh Rate：代表了屏幕在一秒内刷新屏幕的次数，这取决于硬件的固定参数，例如 60Hz。Frame Rate：代表了 GPU 在一秒内绘制操作的帧数，例如 30fps，60fps。通常来说，帧率超过刷新频率只是一种理想的状况，在超过 60fps 的情况下，GPU 所产生的帧数据会因为等待 VSYNC 的刷新信息而被 Hold 住，这样能够保持每次刷新都有实际的新的数据可以显示。但是我们遇到更多的情况是帧率小于刷新频率。
- Tool:Profile GPU Rendering 性能问题如此的麻烦，幸好我们可以有工具来进行调试。打开手机里面的开发者选项，选择 Profile GPU Rendering，选中 On screen as bars 的选项。
- Why 60fps? 我们通常都会提到 60fps 与 16ms，可是知道为何会是以程序是否达到 60fps 来作为 App 性能的衡量标准吗? 这是因为人眼与大脑之间的协作无法感知超过 60fps 的画面更新。开发 app 的性能目标就是保持 60fps，这意味着每一帧你只有 16ms=1000/60 的时间来处理所有的任务。
- Android, UI and the GPU 在 Android 里面那些由主题所提供的资源，例如 Bitmaps，Drawables 都是一起打包到统一的 Texture 纹理当中，然后再传递到 GPU 里面，这意味着每次你需要使用这些资源的时候，都是直接从纹理里面进行获取渲染的。当然随着 UI 组件的越来越丰富，有了更多演变的形态。例如显示图片的时候，需要先经过 CPU 的计算加载到内存中，然后传递给 GPU 进行渲染。文字的显示更加复杂，需要先经过 CPU 换算成纹理，然后再交给 GPU 进行渲染，回到 CPU 绘制单个字符的时候，再重新引用经过 GPU 渲染的内容。动画则是一个更加复杂的操作流程。为了能够使得 App 流畅，我们需要在每一帧 16ms 以内处理完所有的 CPU 与 GPU 计算，绘制，渲染等等操作。
- Invalidations, Layouts, and Performance 任何时候 View 中的绘制内容发生变化时，都会重新执行创建 DisplayList，渲染 DisplayList，更新到屏幕上等一系列操作。这个流程的表现性能取决于你的 View 的复杂程度，View 的状态变化以及渲染管道的执行性能。举个例子，假设某个 Button 的大小需要增大到目前的两倍，在增大 Button 大小之前，需要通过父 View 重新计算并摆放其他子 View 的位置。修改 View 的大小会触发整个 HierarcyView 的重新计算大小的操作。如果是修改 View 的位置则会触发 HierarchView 重新计算其他 View 的位置。如果布局很复杂，这就会很容易导致严重的性能问题。我们需要尽量减少 Overdraw。
- Overdraw, Cliprect, QuickReject 我们可以通过 canvas.clipRect() 来帮助系统识别那些可见的区域。这个方法可以指定一块矩形区域，只有在这个区域内才会被绘制，其他的区域会被忽视。这个 API 可以很好的帮助那些有多组重叠组件的自定义 View 来控制显示的区域。同时 clipRect 方法还可以帮助节约 CPU 与 GPU 资源，在 clipRect 区域之外的绘制指令都不会被执行，那些部分内容在矩形区域内的组件，仍然会得到绘制。
- Memory Churn and performance 执行 GC 操作的时候，所有线程的任何操作都会需要暂停，等待 GC 操作完成之后，其他操作才能够继续运行。Memory Churn 内存抖动，内存抖动是因为大量的对象被创建又在短时间内马上被释放。瞬间产生大量的对象会严重占用 Young Generation 的内存区域，当达到阀值，剩余空间不够的时候，也会触发 GC。即使每次分配的对象占用了很少的内存，但是他们叠加在一起会增加 Heap 的压力，从而触发更多其他类型的 GC。这个操作有可能会影响到帧率，并使得用户感知到性能问题。
- Garbage Collection in Android 原始 JVM 中的 GC 机制在 Android 中得到了很大程度上的优化。Android 里面是一个三级 Generation 的内存模型，最近分配的对象会存放在 Young Generation 区域，当这个对象在这个区域停留的时间达到一定程度，它会被移动到 Old Generation，最后到 Permanent Generation 区域。如果不小心在最小的 for 循环单元里面执行了创建对象的操作，这将很容易引起 GC 并导致性能问题。通过 Memory Monitor 我们可以查看到内存的占用情况，每一次瞬间的内存降低都是因为此时发生了 GC 操作，如果在短时间内发生大量的内存上涨与降低的事件，这说明很有可能这里有性能问题。我们还可以通过 Heap and Allocation Tracker 工具来查看此时内存中分配的到底有哪些对象。
- Performance Cost of Memory Leaks 内存泄漏指的是那些程序不再使用的对象无法被 GC 识别，这样就导致这个对象一直留在内存当中，占用了宝贵的内存空间。显然，这还使得每级 Generation 的内存区域可用空间变小，GC 就会更容易被触发，从而引起性能问题。

- Memory Performance 通常来说，Android 对 GC 做了大量的优化操作，虽然执行 GC 操作的时候会暂停其他任务，可是大多数情况下，GC 操作还是相对很安静并且高效的。但是如果我们对内存的使用不恰当，导致 GC 频繁执行，这样就会引起不小的性能问题。
- Tool - Memory Monitor Android Studio 中的 Memory Monitor 可以很好的帮助我们查看程序的内存使用情况。
- Battery Performance 我们应该尽量减少唤醒屏幕的次数与持续的时间，使用 WakeLock 来处理唤醒的问题，能够正确执行唤醒操作并根据设定及时关闭操作进入睡眠状态。某些非必须马上执行的操作，例如上传歌曲，图片处理等，可以等到设备处于充电状态或者电量充足的时候才进行。触发网络请求的操作，每次都会保持无线信号持续一段时间，我们可以把零散的网络请求打包进行一次操作，避免过多的无线信号引起的电量消耗。关于网络请求引起无线信号的电量消耗
- Understanding Battery Drain on Android 使用 WakeLock 或者 JobScheduler 唤醒设备处理定时的任务之后，一定要及时让设备回到初始状态。每次唤醒无线信号进行数据传递，都会消耗很多电量，它比 WiFi 等操作更加的耗电
- Battery Drain and WakeLocks 这正是 JobScheduler API 所做的事情。它会根据当前的情况与任务，组合出理想的唤醒时间，例如等到正在充电或者连接到 WiFi 的时候，或者集中任务一起执行。我们可以通过这个 API 实现很多免费的调度算法。

2. Android 性能优化典范 - 第 2 季

- Battery Drain and Networking 我们可以有针对性的把请求行为捆绑起来，延迟到某个时刻统一发起请求。这部分主要会涉及到 Prefetch(预取) 与 Compressed(压缩) 这两个技术。对于 Prefetch 的使用，我们需要预先判断用户在此次操作之后，后续零散的请求是否很有可能会马上被触发，可以把后面 5 分钟有可能会使用到的零散请求都一次集中执行完毕。对于 Compressed 的使用，在上传与下载数据之前，使用 CPU 对数据进行压缩与解压，可以很大程度上减少网络传输的时间。
- Wear & Sensors 首先我们需要尽量使用 Android 平台提供的既有运动数据，而不是自己去实现监听采集数据，因为大多数 Android Watch 自身记录 Sensor 数据的行为是有经过做电量优化的。其次在 Activity 不需要监听某些 Sensor 数据的时候需要尽快释放监听注册。还有我们需要尽量控制更新的频率，仅仅在需要刷新显示数据的时候才触发获取最新数据的操作。另外我们可以针对 Sensor 的数据做批量处理，待数据累积一定次数或者某个程度的时候才更新到 UI 上。最后当 Watch 与 Phone 连接起来的时候，可以把某些复杂操作的事情交给 Phone 来执行，Watch 只需要等待返回的结果。
- Smooth Android Wear Animation 在 Android 里面一个相对操作比较繁重的事情是对 Bitmap 进行旋转，缩放，裁剪等等。例如在一个圆形的钟表图上，我们把时钟的指针抠出来当做单独的图片进行旋转会比旋转一张完整的圆形图的所形成的帧率要高 56%。
- Android Wear Data Batching 仅仅在真正需要刷新界面的时候才发出请求，尽量把计算复杂操作的任务交给 Phone 来处理，Phone 仅仅在数据发生变化的时候才通知到 Wear，把零碎的数据请求捆绑一起再进行操作。
- Object Pools 使用对象池技术有很多好处，它可以避免内存抖动，提升性能，但是在使用的时候有一些内容是需要特别注意的。通常情况下，初始化的对象池里面都是空白的，当使用某个对象的时候先去对象池查询是否存在，如果不存在则创建这个对象然后加入对象池，但是我们也可以在程序刚启动的时候就事先为对象池填充一些即将要使用到的数据，这样可以在需要使用到这些对象的时候提供更快的首次加载速度，这种行为就叫做预分配。使用对象池也有不好的一面，程序员需要手动管理这些对象的分配与释放，所以我们需要慎重地使用这项技术，避免发生对象的内存泄漏。为了确保所有的对象能够正确被释放，我们需要保证加入对象池的对象和其他外部对象没有互相引用的关系。
- To Index or Iterate? for index 的方式有更好的效率，但是因为不同平台编译器优化各有差异，我们最好还是针对实际的方法做一下简单的测量比较好，拿到数据之后，再选择效率最高的那个方式。
- The Magic of LRU Cache 使用 LRU Cache 能够显著提升应用的性能，可是也需要注意 LRU Cache 中被淘汰对象的回收，否者会引起严重的内存泄露。
- Using LINT for Performance Tips Lint 已经集成到 Android Studio 中了，我们可以手动去触发这个工具，点击工具栏的 Analysis -> Inspect Code，触发之后，Lint 会开始工作，并把结果输出到底部的工具栏，我们可以逐个查看原因并根据指示做相应的优化修改。
- Hidden Cost of Transparency 通常来说，对于不透明的 View，显示它只需要渲染一次即可，可是如果这个 View 设置了 alpha 值，会至少需要渲染两次。
- Avoiding Allocations in onDraw() 首先 onDraw() 方法是执行在 UI 线程的，在 UI 线程尽量避免做任何可能影响到性能的操作。虽然分配内存的操作并不需要花费太多系统资源，但是这并不意味着是免费无代价的。设备有一定的刷新频率，导致 View 的 onDraw 方法会被频繁的调用，如果 onDraw 方法效率低下，在频繁刷新累积的效应下，效率低的问题会被扩大，然后会对性能有严重的影响。

- Tool: Strict Mode Android 提供了一个叫做 Strict Mode 的工具，我们可以通过手机设置里面的开发者选项，打开 Strict Mode 选项，如果程序存在潜在的隐患，屏幕就会闪现红色。我们也可以通过 StrictMode API 在代码层面做细化的跟踪，可以设置 StrictMode 监听那些潜在问题，出现问题时如何提醒开发者，可以对屏幕闪红色，也可以输出错误日志。
- Custom Views and Performance Useless calls to onDraw()：我们知道调用 View.invalidate() 会触发 View 的重绘，有两个原则需要遵守，第 1 个是仅仅在 View 的内容发生改变的时候才去触发 invalidate 方法，第 2 个是尽量使用 ClipRect 等方法来提高绘制的性能。Useless pixels：减少绘制时不必要的绘制元素，对于那些不可见的元素，我们需要尽量避免重绘。Wasted CPU cycles：对于不在屏幕上的元素，可以使用 Canvas.quickReject 把他们给剔除，避免浪费 CPU 资源。另外尽量使用 GPU 来进行 UI 的渲染，这样能够极大的提高程序的整体表现性能。
- Batching Background Work Until Later 1.AlarmManager 使用 AlarmManager 设置定时任务，可以选择精确的间隔时间，也可以选择非精确时间作为参数。除非程序有很强烈的需要使用精确的定时唤醒，否者一定要避免使用他，我们应该尽量使用非精确的方式。2.SyncAdapter 我们可以使用 SyncAdapter 为应用添加设置账户，这样在手机设置的账户列表里面可以找到我们的应用。这种方式功能更多，但是实现起来比较复杂。我们可以从这里看到官方的培训课程：http://developer.android.com/training/sync-adapters/index.html 3.JobSchedulor 这是最简单高效的方法，我们可以设置任务延迟的间隔，执行条件，还可以增加重试机制。
- Smaller Pixel Formats Android 的 Heap 空间是不会自动做兼容压缩的，意思就是如果 Heap 空间中的图片被收回之后，这块区域并不会和其他已经回收过的区域做重新排序合并处理，那么当一个更大的图片需要放到 heap 之前，很可能找不到那么大的连续空闲区域，那么就会触发 GC，使得 heap 腾出一块足以放下这张图片的空闲区域，如果无法腾出，就会发生 OOM。
- Smaller PNG Files 尽量减少 PNG 图片的大小是 Android 里面很重要的一条规范。相比起 JPEG，PNG 能够提供更加清晰无损的图片，但是 PNG 格式的图片会更大，占用更多的磁盘空间。到底是使用 PNG 还是 JPEG，需要设计师仔细衡量，对于那些使用 JPEG 就可以达到视觉效果的，可以考虑采用 JPEG 即可。
- Pre-scaling Bitmaps 对 bitmap 做缩放，这也是 Android 里面最遇到的问题。对 bitmap 做缩放的意义很明显，提示显示性能，避免分配不必要的内存。Android 提供了现成的 bitmap 缩放的 API，叫做 createScaledBitmap()
- Re-using Bitmaps 使用 inBitmap 属性可以告知 Bitmap 解码器去尝试使用已经存在的内存区域，新解码的 bitmap 会尝试去使用之前那张 bitmap 在 heap 中所占据的 pixel data 内存区域，而不是去问内存重新申请一块区域来存放 bitmap。利用这种特性，即使是上千张的图片，也只会仅仅只需要占用屏幕所能够显示的图片数量的内存大小。
- The Performance Lifecycle Gather：收集数据，Insight：分析数据，Action：解决问题

3. Android 性能优化典范 - 第 3 季

- Fun with ArrayMaps 为了解决 HashMap 更占内存的弊端，Android 提供了内存效率更高的 ArrayMap。它内部使用两个数组进行工作，其中一个数组记录 key hash 过后的顺序列表，另外一个数组按 key 的顺序记录 Key-Value 值
- Beware Autoboxing 有时候性能问题也可能是因为那些不起眼的小细节引起的，例如在代码中不经意的“自动装箱”。我们知道基础数据类型的大小：boolean(8 bits), int(32 bits), float(32 bits)，long(64 bits)，为了能够让这些基础数据类型在大多数 Java 容器中运作，会需要做一个 autoboxing 的操作，转换成 Boolean，Integer，Float 等对象
- SparseArray Family Ties 为了避免 HashMap 的 autoboxing 行为，Android 系统提供了 SparseBoolMap，SparseIntMap，SparseLongMap，LongSparseMap 等容器。
- The price of ENUMs Android 官方强烈建议不要在 Android 程序里面使用到 enum。
- Trimming and Sharing Memory Android 系统提供了一些回调来通知应用的内存使用情况，通常来说，当所有的 background 应用都被 kill 掉的时候，forground 应用会收到 onLowMemory() 的回调。在这种情况下，需要尽快释放当前应用的非必须内存资源，从而确保系统能够稳定继续运行。Android 系统还提供了 onTrimMemory() 的回调，当系统内存达到某些条件的时候，所有正在运行的应用都会收到这个回调
- DO NOT LEAK VIEWS 避免使用异步回调，避免使用 Static 对象，避免把 View 添加到没有清除机制的容器里面
- Location & Battery Drain 其中存在的一个优化点是，我们可以通过判断返回的位置信息是否相同，从而决定设置下次的更新间隔是否增加一倍，通过这种方式可以减少电量的消耗

- Double Layout Taxation 布局中的任何一个 View 一旦发生一些属性变化，都可能引起很大的连锁反应。例如某个 button 的大小突然增加一倍，有可能会导致兄弟视图的位置变化，也有可能导致父视图的大小发生改变。当大量的 layout() 操作被频繁调用执行的时候，就很可能引起丢帧的现象。
- Network Performance 101 减少移动网络被激活的时间与次数, 压缩传输数据
- Effective Network Batching 发起网络请求与接收返回数据都是比较耗电的，在网络硬件模块被激活之后，会继续保持几十秒的电量消耗，直到没有新的网络操作行为之后，才会进入休眠状态。前面一个段落介绍了使用 Batching 的技术来捆绑网络请求，从而达到减少网络请求的频率。那么如何实现 Batching 技术呢？通常来说，我们可以会把那些发出的网络请求，先暂存到一个 PendingQueue 里面，等到条件合适的时候再触发 Queue 里面的网络请求。
- Optimizing Network Request Frequencies 前面的段落已经提到了应该减少网络请求的频率，这是为了减少电量的消耗。我们可以使用 Batching，Prefetching 的技术来避免频繁的网络请求。Google 提供了 GCMNetworkManager 来帮助开发者实现那些功能，通过提供的 API，我们可以选择在接入 WiFi，开始充电，等待移动网络被激活等条件下再次激活网络请求。
- Effective Prefetching 类似上面的情况会频繁触发网络请求，但是如果我们能够预先请求后续可能会使用到网络资源，避免频繁的触发网络请求，这样就能够显著的减少电量的消耗。可是预先获取多少数据量是很值得考量的，因为如果预取数据量偏少，就起不到减少频繁请求的作用，可是如果预取数据过多，就会造成资源的浪费。

4. Android 性能优化典范 - 第 4 季

- Cachematters for networking 想要使得 Android 系统上的网络访问操作更加的高效就必须做好网络数据的缓存。这是提高网络访问性能最基础的步骤之一。从手机的缓存中直接读取数据肯定比从网络上获取数据要更加的便捷高效，特别是对于那些会被频繁访问到的数据，需要把这些数据缓存到设备上，以便更加快速的进行访问。
- Optimizing Network Request Frequencies 首先我们要对网络行为进行分类，区分需要立即更新数据的行为和其他可以进行延迟的更新行为，为不同的场景进行差异化处理。其次要避免客户端对服务器的轮询操作，这样会浪费很多的电量与带宽流量。解决这个问题，我们可以使用 Google Cloud Message 来对更新的数据进行推送。然后在某些必须做同步的场景下，需要避免使用固定的间隔频率来进行更新操作，我们应该在返回的数据无更新的时候，使用双倍的间隔时间来进行下一次同步。最后更进一步，我们还可以通过判断当前设备的状态来决定同步的频率，例如判断设备处于休眠，运动等不同的状态设计各自不同时间间隔的同步频率。
- Effective Prefetching 到底预取多少才比较合适呢？一个比较普适的规则是，在 3G 网络下可以预取 1-5Mb 的数据量，或者是按照提前预期后续 1-2 分钟的数据作为基线标准。在实际的操作当中，我们还需要考虑当前的网络速度来决定预取的数据量，例如在同样的时间下，4G 网络可以获取到 12 张图片的数据，而 2G 网络则只能拿到 3 张图片的数据。所以，我们还需要把当前的网络环境情况添加到设计预取数据量的策略当中去。判断当前设备的状态与网络情况，可以使用前面提到过的 GCMNetworkManager。
- Adapting to Latency 一个典型的网络操作行为，通常包含以下几个步骤：首先手机端发起网络请求，到达网络服务运营商的基站，再转移到服务提供者的服务器上，经过解码之后，接着访问本地的存储数据库，获取到数据之后，进行编码，最后按照原来传递的路径逐层返回。常来说，我们可以把网络请求延迟划分为三档：例如把网络延迟小于 60ms 的划分为 GOOD，大于 220ms 的划分为 BAD，介于两者之间的划分为 OK（这里的 60ms，220ms 会需要根据不同的场景提前进行预算推测）。
- Minimizing Asset Payload 为了能够减小网络传输的数据量，我们需要对传输的数据做压缩的处理，这样能够提高网络操作的性能。首先需要做的是减少图片的大小，其次需要做的是减少序列化数据的大小。
- Service Performance Patterns Service 是 Android 程序里面最常用的基础组件之一，但是使用 Service 很容易引起电量的过度消耗以及系统资源的未及时释放。避免错误的使用 Service，例如我们不应该使用 Service 来监听某些事件的变化，不应该搞一个 Service 在后台对服务器不断的进行轮询 (应该使用 Google Cloud Messaging)。如果已经事先知道 Service 里面的任务应该执行在后台线程 (非默认的主线程) 的时候，我们应该使用 IntentService 或者结合 HanderThread，AsycnTask Loader 实现的 Service。
- Removing unused code Android 为我们提供了 Proguard 的工具来帮助应用程序对代码进行瘦身，优化，混淆的处理。它会帮助移除那些没有使用到的代码，还可以对类名，方法名进行混淆处理以避免程序被反编译。
- Removing unused resources 所幸的是，我们可以使用 Gradle 来帮助我们分析代码，分析引用的资源，对于那些没有被引用到的资源，会在编译阶段被排除在 APK 安装包之外，要实现这个功能，对我们来说仅仅只需要在 build.gradle 文件中配置 shrinkResource 为 true 就好了

- Perf Theory: Caching 当我们讨论性能优化的时候，缓存是最常见最有效的策略之一。无论是为了提高 CPU 的计算速度还是提高数据的访问速度，在绝大多数的场景下，我们都会使用到缓存。
- Perf Theory: Approximation(近似法) 例如使用一张比较接近实际大小的图片来替代原图，换取更快的加载速度。所以对于那些对计算结果要求不需要十分精确的场景，我们可以使用近似法则来提高程序的性能。
- Perf Theory: Culling(遴选，挑选) 一个提高性能的方法是逐步对数据进行过滤筛选，减小搜索的数据集，以此提高程序的执行性能。例如我们需要搜索到居住在某个地方，年龄是多少，符合某些特定条件的候选人，就可以通过逐层过滤筛选的方式来提高后续搜索的执行效率。
- Perf Theory: Threading 使用多线程并发处理任务，从某种程度上可以快速提高程序的执行性能。对于 Android 程序来说，主线程通常也成为 UI 线程，需要处理 UI 的渲染，响应用户的操作等等。
- Perf Theory: Batching 网络请求的批量执行是另外一个比较适合说明 batching 使用场景的例子，因为每次发起网络请求都相对来说比较耗时耗电，如果能够做到批量一起执行，可以大大的减少电量的消耗。
- Serialization performance 数据序列化的行为可能发生在数据传递过程中的任何阶段，例如网络传输，不同进程间数据传递，不同类之间的参数传递，把数据存储到磁盘上等等。通常情况下，我们会把那些需要序列化的类实现 Serializable 接口 (如下图所示)，但是这种传统的做法效率不高，实施的过程会消耗更多的内存。但是我们如果使用 GSON 库来处理这个序列化的问题，不仅仅执行速度更快，内存的使用效率也更高。Android 的 XML 布局文件会在编译的阶段被转换成更加复杂的格式，具备更加高效的执行性能与更高的内存使用效率。
- Smaller Serialized Data 数据呈现的顺序以及结构会对序列化之后的空间产生不小的影响。
- Caching UI data 缓存 UI 界面上的数据，可以采用方案有存储到文件系统，Preference，SQLite 等等，做了缓存之后，这样就可以在请求数据返回结果之前，呈现给用户旧的数据，而不是使用正在加载的方式让用户什么数据都看不到，当然在请求网络最新数据的过程中，需要有正在刷新的提示。至于到底选择哪个方案来对数据进行缓存，就需要根据具体情况来做选择了。
- CPU Frequency Scaling 调节 CPU 的频率会执行的性能产生较大的影响，为了最大化的延长设备的续航时间，系统会动态调整 CPU 的频率，频率越高执行代码的速度自然就越快。我们可以使用 Systrace 工具来导出 CPU 的执行情况，以便帮助定位性能问题。

5. Android 性能优化典范 - 第 5 季

- Threading Performance AsyncTask: 为 UI 线程与工作线程之间进行快速的切换提供一种简单便捷的机制。适用于当下立即需要启动，但是异步执行的生命周期短暂的使用场景。HandlerThread: 为某些回调方法或者等待某些任务的执行设置一个专属的线程，并提供线程任务的调度机制。ThreadPool: 把任务分解成不同的单元，分发到各个不同的线程上，进行同时并发处理。IntentService: 适合于执行由 UI 触发的后台 Service 任务，并可以把后台任务执行的情况通过一定的机制反馈给 UI。
- Understanding Android Threading 通常来说，一个线程需要经历三个生命阶段: 开始，执行，结束。线程会在任务执行完毕之后结束，那么为了确保线程的存活，我们会在执行阶段给线程赋予不同的任务，然后在里面添加退出的条件从而确保任务能够执行完毕后退出。
- Memory & Threading 不要在任何非 UI 线程里面去持有 UI 对象的引用。系统为了确保所有的 UI 对象都只会被 UI 线程所进行创建，更新，销毁的操作，特地设计了对应的工作机制 (当 Activity 被销毁的时候，由该 Activity 所触发的非 UI 线程都将无法对 UI 对象进行操作，否者就会抛出程序执行异常的错误) 来防止 UI 对象被错误的使用。
- Good AsyncTask Hunting AsyncTask 虽然提供了一种简单便捷的异步机制，但是我们还是很有必要特别关注到他的缺点，避免出现因为使用错误而导致的严重系统性能问题。
- Getting a HandlerThread HandlerThread 比较合适处理那些在工作线程执行，需要花费时间偏长的任务。我们只需要把任务发送给 HandlerThread，然后就只需要等待任务执行结束的时候通知返回到主线程就好了。另外很重要的一点是，一旦我们使用了 HandlerThread，需要特别注意给 HandlerThread 设置不同的线程优先级，CPU 会根据设置的不同线程优先级对所有的线程进行调度优化。
- Swimming in Threadpools 线程池适合用在把任务进行分解，并发进行执行的场景。通常来说，系统里面会针对不同的任务设置一个单独的守护线程用来专门处理这项任务。
- The Zen of IntentService 默认的 Service 是执行在主线程的，可是通常情况下，这很容易影响到程序的绘制性能 (抢占了主线程的资源)。除了前面介绍过的 AsyncTask 与 HandlerThread，我们还可以选择使用 IntentService 来实现异步操作。IntentService 继承自普通 Service 同时又在内部创建了一个 HandlerThread，在 onHandlerIntent() 的回调里面处理扔到 IntentService 的任务。所以 IntentService 就不仅仅具备了异步线程的特性，还同时保留了 Service 不受主页面生命周期影响的特点。

- Threading and Loaders 当启动工作线程的 Activity 被销毁的时候，我们应该做点什么呢？为了方便的控制工作线程的启动与结束，Android 为我们引入了 Loader 来解决这个问题。我们知道 Activity 有可能因为用户的主动切换而频繁的被创建与销毁，也有可能是因为类似屏幕发生旋转等被动原因而销毁再重建。在 Activity 不停的创建与销毁的过程当中，很有可能因为工作线程持有 Activity 的 View 而导致内存泄漏 (因为工作线程很可能持有 View 的强引用，另外工作线程的生命周期还无法保证和 Activity 的生命周期一致，这样就容易发生内存泄漏了)。除了可能引起内存泄漏之外，在 Activity 被销毁之后，工作线程还继续更新视图是没有意义的，因为此时视图已经不在界面上显示了。
- The Importance of Thread Priority 在 Android 系统里面，我们可以通过 android.os.Process.setThreadPri 设置线程的优先级，参数范围从-20 到 24，数值越小优先级越高。Android 系统还为我们提供了以下的一些预设值，我们可以通过给不同的工作线程设置不同数值的优先级来达到更细粒度的控制。
- Profile GPU Rendering：M Update 从 Android M 系统开始，系统更新了 GPU Profiling 的工具来帮助我们定位 UI 的渲染性能问题。早期的 CPU Profiling 工具只能粗略的显示出 Process，Execute，Update 三大步骤的时间耗费情况。
- 官方性能优化系列教程 https://www.youtube.com/playlist?list=PLWz5rJ2EKKc9CBxr3BVjPTPoDPLd

1.3.2 架构分析

- MVVM https://tech.meituan.com/android_mvvm.html
- MVP https://code.tutsplus.com/series/how-to-adopt-model-view-presenter-on-android--cms-1

1.3.3 阿里面试题

- 进程间通信方式
  - 通过 Intent 在 Activity、Service 或 BroadcastReceiver 间进行进程间通信，可通过 Intent 传递数据
  - AIDL 方式
  - Messenger 方式
  - 利用 ContentProvider
  - Socket 方式
  - 基于文件共享的方式
- 什么是协程
  - 我们知道多个线程相对独立，有自己的上下文，切换受系统控制；而协程也相对独立，有自己的上下文，但是其切换由自己控制，由当前协程切换到其他协程由当前协程来控制。
- 内存泄露是怎么回事
  - 由忘记释放分配的内存导致的
- 程序计数器，引到了逻辑地址（虚地址）和物理地址及其映射关系
  - 虚拟机中的程序计数器是 Java 运行时数据区中的一小块内存区域，但是它的功能和通常的程序计数器是类似的，它指向虚拟机正在执行字节码指令的地址。具体点儿说，当虚拟机执行的方法不是 native 的时，程序计数器指向虚拟机正在执行字节码指令的地址；当虚拟机执行的方法是 native 的时，程序计数器中的值是未定义的。另外，程序计数器是线程私有的，也就是说，每一个线程都拥有仅属于自己的程序计数器。
- 数组和链表的区别
  - 数组是将元素在内存中连续存放，由于每个元素占用内存相同，可以通过下标迅速访问数组中任何元素。但是如果要在数组中增加一个元素，需要移动大量元素，在内存中空出一个元素的空间，然后将要增加的元素放在其中。同样的道理，如果想删除一个元素，同样需要移动大量元素去填掉被移动的元素。如果应用需要快速访问数据，很少或不插入和删除元素，就应该用数组。
  - 链表恰好相反，链表中的元素在内存中不是顺序存储的，而是通过存在元素中的指针联系到一起。比如：上一个元素有个指针指到下一个元素，以此类推，直到最后一个元素。如果要访问链表中一个元素，需要从第一个元素开始，一直找到需要的元素位置。但是增加和删除一个元素对于链表数据结构就非常简单了，只要修改元素中的指针就可以了。如果应用需要经常插入和删除元素你就需要用链表数据结构了。

- 二叉树的深度优先遍历和广度优先遍历的具体实现

http://www.i3geek.com/archives/794

- 堆的结构
  - 年轻代（Young Generation）、年老代（Old Generation）和持久代（Permanent Generation）。其中持久代主要存放的是 Java 类的类信息，与垃圾收集要收集的 Java 对象关系不大。年轻代和年老代的划分是对垃圾收集影响比较大的。
- bitmap 对象的理解
  - http://blog.csdn.net/angel1hao/article/details/51890938
- 什么是深拷贝和浅拷
  - 浅拷贝：使用一个已知实例对新创建实例的成员变量逐个赋值，这个方式被称为浅拷贝。
  - 深拷贝：当一个类的拷贝构造方法，不仅要复制对象的所有非引用成员变量值，还要为引用类型的成员变量创建新的实例，并且初始化为形式参数实例值。这个方式称为深拷贝
- 对象锁和类锁是否会互相影响
  - 对象锁：Java 的所有对象都含有 1 个互斥锁，这个锁由 JVM 自动获取和释放。线程进入 synchronized 方法的时候获取该对象的锁，当然如果已经有线程获取了这个对象的锁，那么当前线程会等待；synchronized 方法正常返回或者抛异常而终止，JVM 会自动释放对象锁。这里也体现了用 synchronized 来加锁的 1 个好处，方法抛异常的时候，锁仍然可以由 JVM 来自动释放。
  - 类锁：对象锁是用来控制实例方法之间的同步，类锁是用来控制静态方法（或静态变量互斥体）之间的同步。其实类锁只是一个概念上的东西，并不是真实存在的，它只是用来帮助我们理解锁定实例方法和静态方法的区别的。我们都知道，java 类可能会有很多个对象，但是只有 1 个 Class 对象，也就是说类的不同实例之间共享该类的 Class 对象。Class 对象其实也仅仅是 1 个 java 对象，只不过有点特殊而已。由于每个 java 对象都有 1 个互斥锁，而类的静态方法是需要 Class 对象。所以所谓的类锁，不过是 Class 对象的锁而已。获取类的 Class 对象有好几种，最简单的就是 MyClass.class 的方式。类锁和对象锁不是同 1 个东西，一个是类的 Class 对象的锁，一个是类的实例的锁。也就是说：1 个线程访问静态 synchronized 的时候，允许另一个线程访问对象的实例 synchronized 方法。反过来也是成立的，因为他们需要的锁是不同的。
- looper 架构
  - http://wangkuiwu.github.io/2014/08/26/MessageQueue/
- 自定义控件原理
  - http://www.jianshu.com/p/988326f9c8a3
- binder 工作原理
  - Binder 是客户端和服务端进行通讯的媒介
- ActivityThread，Ams，Wms 的工作原理
  - ActivityThread: 运行在应用进程的主线程上，响应 ActivityManangerService 启动、暂停 Activity，广播接收等消息。ams: 统一调度各应用程序的 Activity、内存管理、进程管理
- Java 中 final，finally，finalize 的区别
  - final 用于声明属性, 方法和类, 分别表示属性不可变, 方法不可覆盖, 类不可继承.
  - finally 是异常处理语句结构的一部分，表示总是执行.
  - finalize 是 Object 类的一个方法，在垃圾收集器执行的时候会调用被回收对象的此方法，可以覆盖此方法提供垃圾收集时的其他资源回收，例如关闭文件等. JVM 不保证此方法总被调用.
- 一个文件中有 100 万个整数，由空格分开，在程序中判断用户输入的整数是否在此文件中。说出最优的方法
- 两个进程同时要求写或者读，能不能实现？如何防止进程的同步？
- volatile 的意义？

- 防止 CPU 指令重排序
- 单例

```
public class Singleton {
    private volatile static Singleton mSingleton;
    private Singleton(){
    }
    public static Singleton getInstance() {
        if (mSingleton == null) {//A
            synchronized(Singleton.class) {//C
                if(mSingleton == null)
                    mSingleton = new Singleton();//B
            }
        }
        return mSingleton;
    }
}
```

- Given a string, determine if it is a palindrome（回文，如果不清楚，按字面意思脑补下）, considering only alphanumeric characters and ignoring cases.
  - For example, "A man, a plan, a canal: Panama" is a palindrome. "race a car" is not a palindrome.
  - Note: Have you consider that the string might be empty? This is a good question to ask during an interview. For the purpose of this problem, we define empty string as valid palindrome.

```
public boolean isPalindrome(String palindrome) {
    char[] palindromes = palidrome.toCharArray();
    if (palindromes.lengh == 0) {
        return true
    }
    Arraylist<Char> temp = new Arraylist();
    for (int i=0;i<palindromes.length;i++) {
 ^^I^^Iif ((palindromes[i]>'a' && palindromes[i]<'z')||palindromes[i]>'A' && palindrom
            temp.add(palindromes[i].toLowerCase());
    }
    for (int i=0;i<temp.size()/2;i++) {
        if (temp.get(i) != temp.get(temp.size()-i)) {
            //
            return false;
        }
    }
    return true;
}
```

- 烧一根不均匀的绳，从头烧到尾总共需要 1 个小时。现在有若干条材质相同的绳子，问如何用烧绳的方法来计时一个小时十五分钟呢
  - 用两根绳子，一个绳子两头烧，一个一头烧。

1.3.4 腾讯

- 2000 万个整数，找出第五十大的数字?
  - 冒泡、选择、建堆
- 从网络加载一个 10M 的图片，说下注意事项
  - 图片缓存、异常恢复、质量压缩
- 自定义 View 注意事项
  - 渲染帧率、内存

- 项目中常用的设计模式
  - 单例、观察者、适配器、建造者。。
- JVM 的理解 http://www.infoq.com/cn/articles/java-memory-model-1

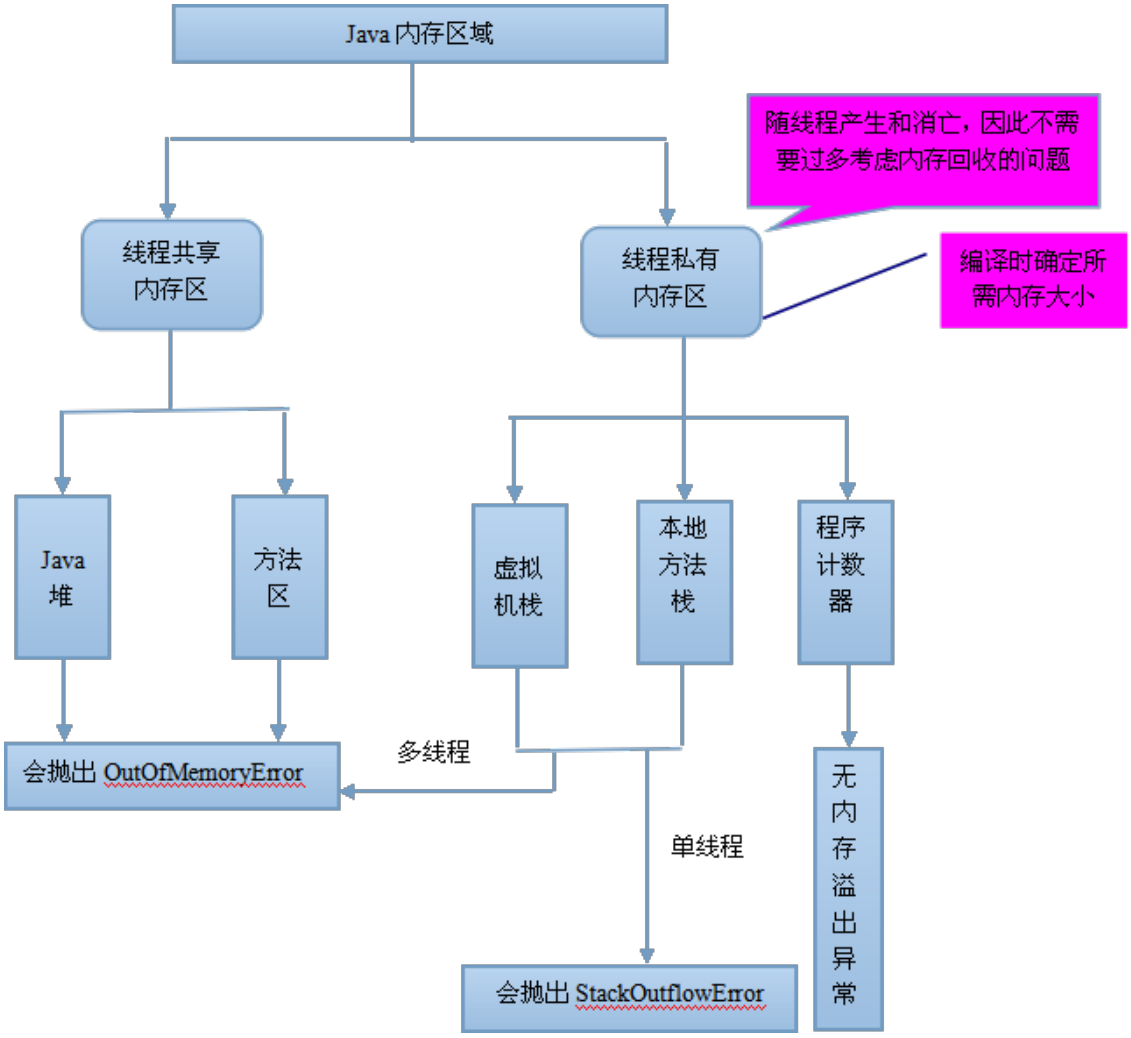
2 JVM(Java 虚拟机)

2.1 JVM 基础知识

2.1.1 内存模型以及分区，需要详细到每个区放什么。

http://blog.csdn.net/ns_code/article/details/17565503

- JVM 所管理的内存分为以下几个运行时数据区：程序计数器、Java 虚拟机栈、本地方法栈、Java 堆、方法区。



1. 程序计数器 (Program Counter Register)
   - 一块较小的内存空间，它是当前线程所执行的字节码的行号指示器，字节码解释器工作时通过改变该计数器的值来选择下一条需要执行的字节码指令，分支、跳转、循环等基础功能都要依赖它来实现。每条线程都有一个独立的的程序计数器，各线程间的计数器互不影响，因此该区域是线程私有的。
   - 当线程在执行一个 Java 方法时，该计数器记录的是正在执行的虚拟机字节码指令的地址，当线程在执行的是 Native 方法（调用本地操作系统方法）时，该计数器的值为空。另外，该内存区域是唯一一个在 Java 虚拟机规范中么有规定任何 OOM（内存溢出：OutOfMemoryError）情况的区域。
2. Java 虚拟机栈（Java Virtual Machine Stacks）

- 该区域也是线程私有的，它的生命周期也与线程相同。虚拟机栈描述的是 Java 方法执行的内存模型：每个方法被执行的时候都会同时创建一个栈帧，栈它是用于支持续虚拟机进行方法调用和方法执行的数据结构。对于执行引擎来讲，活动线程中，只有栈顶的栈帧是有效的，称为当前栈帧，这个栈帧所关联的方法称为当前方法，执行引擎所运行的所有字节码指令都只针对当前栈帧进行操作。栈帧用于存储局部变量表、操作数栈、动态链接、方法返回地址和一些额外的附加信息。在编译程序代码时，栈帧中需要多大的局部变量表、多深的操作数栈都已经完全确定了，并且写入了方法表的 Code 属性之中。因此，一个栈帧需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现。

3. 本地方法栈（Native Method Stacks）

   - 该区域与虚拟机栈所发挥的作用非常相似，只是虚拟机栈为虚拟机执行 Java 方法服务，而本地方法栈则为使用到的本地操作系统（Native）方法服务。

4. Java 堆（Java Heap）

   - Java Heap 是 Java 虚拟机所管理的内存中最大的一块，它是所有线程共享的一块内存区域。几乎所有的对象实例和数组都在这类分配内存。Java Heap 是垃圾收集器管理的主要区域，因此很多时候也被称为“GC 堆”。
   - 根据 Java 虚拟机规范的规定，Java 堆可以处在物理上不连续的内存空间中，只要逻辑上是连续的即可。如果在堆中没有内存可分配时，并且堆也无法扩展时，将会抛出 OutOfMemoryError 异常。

5. 方法区（Method Area）

   - 方法区也是各个线程共享的内存区域，它用于存储已经被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。方法区域又被称为“永久代”，但这仅仅对于 Sun HotSpot 来讲，JRockit 和 IBM J9 虚拟机中并不存在永久代的概念。Java 虚拟机规范把方法区描述为 Java 堆的一个逻辑部分，而且它和 Java Heap 一样不需要连续的内存，可以选择固定大小或可扩展，另外，虚拟机规范允许该区域可以选择不实现垃圾回收。相对而言，垃圾收集行为在这个区域比较少出现。该区域的内存回收目标主要针是对废弃常量的和无用类的回收。运行时常量池是方法区的一部分，Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池（Class 文件常量池），用于存放编译器生成的各种字面量和符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。运行时常量池相对于 Class 文件常量池的另一个重要特征是具备动态性，Java 语言并不要求常量一定只能在编译期产生，也就是并非预置入 Class 文件中的常量池的内容才能进入方法区的运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用比较多的是 String 类的 intern () 方法。
   - 根据 Java 虚拟机规范的规定，当方法区无法满足内存分配需求时，将抛出 OutOfMemoryError 异常。

2.1.2 内存泄漏和内存溢出的差别

- 内存泄露是指分配出去的内存没有被回收回来，由于失去了对该内存区域的控制，因而造成了资源的浪费。Java 中一般不会产生内存泄露，因为有垃圾回收器自动回收垃圾，但这也不绝对，当我们 new 了对象，并保存了其引用，但是后面一直没用它，而垃圾回收器又不会去回收它，这边会造成内存泄露，
- 内存溢出是指程序所需要的内存超出了系统所能分配的内存（包括动态扩展）的上限。

2.1.3 类型擦除

- http://blog.csdn.net/ns_code/article/details/18011009
- Java 语言在 JDK1.5 之后引入的泛型实际上只在程序源码中存在，在编译后的字节码文件中，就已经被替换为了原来的原生类型，并且在相应的地方插入了强制转型代码，因此对于运行期的 Java 语言来说，ArrayList<String> 和 ArrayList<Integer> 就是同一个类。所以泛型技术实际上是 Java 语言的一颗语法糖，Java 语言中的泛型实现方法称为类型擦除，基于这种方法实现的泛型被称为伪泛型。
- 下面是一段简单的 Java 泛型代码：

```
Map<Integer,String> map = new HashMap<Integer,String>();
map.put(1,"No.1");
map.put(2,"No.2");
System.out.println(map.get(1));
System.out.println(map.get(2));
```

- 将这段 Java 代码编译成 Class 文件，然后再用字节码反编译工具进行反编译后，将会发现泛型都变回了原生类型，如下面的代码所示：

```
Map map = new HashMap();
map.put(1,"No.1");
map.put(2,"No.2");
System.out.println((String)map.get(1));
System.out.println((String)map.get(2));
```

- 为了更详细地说明类型擦除，再看如下代码：

```
import java.util.List;
public class FanxingTest{
    public void method(List<String> list){
        System.out.println("List String");
    }
    public void method(List<Integer> list){
        System.out.println("List Int");
    }
}
```

- 当我用 Javac 编译器编译这段代码时，报出了如下错误：

```
FanxingTest.java:3: 名称冲突: method(java.util.List<java.lang.String>) 和 method
(java.util.List<java.lang.Integer>) 具有相同疑符
public void method(List<String> list){
^
FanxingTest.java:6: 名称冲突: method(java.util.List<java.lang.Integer>) 和 metho
d(java.util.List<java.lang.String>) 具有相同疑符
public void method(List<Integer> list){
^
```

- 2 错误
  - 这是因为泛型 List 和 List 编译后都被擦除了，变成了一样的原生类型 List，擦除动作导致这两个方法的特征签名变得一模一样，在 Class 类文件结构一文中讲过，Class 文件中不能存在特征签名相同的方法。
  - 把以上代码修改如下：

```
import java.util.List;
public class FanxingTest{
    public int method(List<String> list){
        System.out.println("List String");
        return 1;
    }
    public boolean method(List<Integer> list){
        System.out.println("List Int");
        return true;
    }
}
```

- 发现这时编译可以通过了（注意：Java 语言中 true 和 1 没有关联，二者属于不同的类型，不能相互转换，不存在 C 语言中整数值非零即真的情况）。两个不同类型的返回值的加入，使得方法的重载成功了。这是为什么呢？
- 我们知道，Java 代码中的方法特征签名只包括了方法名称、参数顺序和参数类型，并不包括方法的返回值，因此方法的返回值并不参与重载方法的选择，这样看来为重载方法加入返回值貌似是多余的。对于重载方法的选择来说，这确实是多余的，但我们现在要解决的问题是让上述代码能通过编译，让两个重载方法能够合理地共存于同一个 Class 文件之中，这就要看字节码的方法特征签名，它不仅包括了 Java 代码中方法特征签名中所包含的那些信息，还包括方法返回值及受查异常表。为两个重载方法加入不同的返回值后，因为有了不同的字节码特征签名，它们便可以共存于一个 Class 文件之中。

2.1.4 堆里面的分区：Eden，survival from to，老年代，各自的特点。

2.1.5 对象创建方法，对象的内存分配，对象的访问定位。

- 对内存分配情况分析最常见的示例便是对象实例化:

```
Object obj = new Object();
```

- 这段代码的执行会涉及 java 栈、Java 堆、方法区三个最重要的内存区域。假设该语句出现在方法体中，及时对 JVM 虚拟机不了解的 Java 使用这，应该也知道 obj 会作为引用类型（reference）的数据保存在 Java 栈的本地变量表中，而会在 Java 堆中保存该引用的实例化对象，但可能并不知道，Java 堆中还必须包含能查找到此对象类型数据的地址信息（如对象类型、父类、实现的接口、方法等），这些类型数据则保存在方法区中。

- 另外，由于 reference 类型在 Java 虚拟机规范里面只规定了一个指向对象的引用，并没有定义这个引用应该通过哪种方式去定位，以及访问到 Java 堆中的对象的具体位置，因此不同虚拟机实现的对象访问方式会有所不同，主流的访问方式有两种：使用句柄池和直接使用指针。

2.1.6 GC 的两种判定方法：引用计数与引用链。

- 引用计数方式最基本的形态就是让每个被管理的对象与一个引用计数器关联在一起，该计数器记录着该对象当前被引用的次数，每当创建一个新的引用指向该对象时其计数器就加 1，每当指向该对象的引用失效时计数器就减 1。当该计数器的值降到 0 就认为对象死亡。

- Java 的内存回收机制可以形象地理解为在堆空间中引入了重力场，已经加载的类的静态变量和处于活动线程的堆栈空间的变量是这个空间的牵引对象。这里牵引对象是指按照 Java 语言规范，即便没有其它对象保持对它的引用也不能够被回收的对象，即 Java 内存空间中的本原对象。当然类可能被去加载，活动线程的堆栈也是不断变化的，牵引对象的集合也是不断变化的。对于堆空间中的任何一个对象，如果存在一条或者多条从某个或者某几个牵引对象到该对象的引用链，则就是可达对象，可以形象地理解为从牵引对象伸出的引用链将其拉住，避免掉到回收池中。

2.1.7 GC 的三种收集方法：标记清除、标记整理、复制算法的原理与特点，分别用在什么地方，如果让你优化收集方法，有什么思路?

- 标记清除算法是最基础的收集算法，其他收集算法都是基于这种思想。标记清除算法分为“标记”和“清除”两个阶段：首先标记出需要回收的对象，标记完成之后统一清除对象。它的主要缺点：. 标记和清除过程效率不高。. 标记清除之后会产生大量不连续的内存碎片。

- 标记整理，标记操作和“标记-清除”算法一致，后续操作不只是直接清理对象，而是在清理无用对象完成后让所有存活的对象都向一端移动，并更新引用其对象的指针。主要缺点：在标记-清除的基础上还需进行对象的移动，成本相对较高，好处则是不会产生内存碎片。

- 复制算法，它将可用内存容量划分为大小相等的两块，每次只使用其中的一块。当这一块用完之后，就将还存活的对象复制到另外一块上面，然后在把已使用过的内存空间一次理掉。这样使得每次都是对其中的一块进行内存回收，不会产生碎片等情况，只要移动堆订的指针，按顺序分配内存即可，实现简单，运行高效。主要缺点：内存缩小为原来的一半。

2.1.8 Minor GC 与 Full GC 分别在什么时候发生?

- Minor GC：通常是指对新生代的回收。指发生在新生代的垃圾收集动作，因为 Java 对象大多都具备朝生夕灭的特性，所以 Minor GC 非常频繁，一般回收速度也比较快

- Major GC：通常是指对年老代的回收。

- Full GC：Major GC 除并发 gc 外均需对整个堆进行扫描和回收。指发生在老年代的 GC，出现了 Major GC，经常会伴随至少一次的 Minor GC（但非绝对的，在 ParallelScavenge 收集器的收集策略里就有直接进行 Major GC 的策略选择过程）。MajorGC 的速度一般会比 Minor GC 慢 10 倍以上。

- 几种常用的内存调试工具：jmap、jstack、jconsole。

- jmap（linux 下特有，也是很常用的一个命令）观察运行中的 jvm 物理内存的占用情况。参数如下：-heap：打印 jvm heap 的情况 -histo：打印 jvm heap 的直方图。其输出信息包括类名，对象数量，对象占用大小。-histo：live ： 同上，但是只答应存活对象的情况 -permstat：打印 permanent generation heap 情况 jstack（linux 下特有）可以观察到 jvm 中当前所有线程的运行情况和线程当前状态 jconsole 一个图形化界面，可

以观察到 java 进程的 gc，class，内存等信息 jstat 最后要重点介绍下这个命令。这是 jdk 命令中比较重要，也是相当实用的一个命令，可以观察到 classloader，compiler，gc 相关信息具体参数如下：-class：统计 class loader 行为信息 -compile：统计编译行为信息 -gc：统计 jdk gc 时 heap 信息 -gccapacity：统计不同的 generations（不知道怎么翻译好，包括新生区，老年区，permanent 区）相应的 heap 容量情况 -gccause：统计 gc 的情况，（同-gcutil）和引起 gc 的事件 -gcnew：统计 gc 时，新生代的情况 -gcnewcapacity：统计 gc 时，新生代 heap 容量 -gcold：统计 gc 时，老年区的情况 -gcoldcapacity：统计 gc 时，老年区 heap 容量 -gcpermcapacity：统计 gc 时，permanent 区 heap 容量 -gcutil：统计 gc 时，heap 情况 -printcompilation：不知道干什么的，一直没用过。

2.1.9 类加载的五个过程：加载、验证、准备、解析、初始化。

1. 类加载过程

   - 类从被加载到虚拟机内存中开始，到卸载出内存为止，它的整个生命周期包括加载、验证、准备、解析、初始化、使用、卸载。
   - 其中类加载的过程包括了加载、验证、准备、解析、初始化五个阶段。在这五个阶段中，加载、验证、准备和初始化这四个阶段发生的顺序是确定的，而解析阶段则不一定，它在某些情况下可以在初始化阶段之后开始，这是为了支持 Java 语言的运行时绑定（也成为动态绑定或晚期绑定）。另外注意这里的几个阶段是按顺序开始，而不是按顺序进行或完成，因为这些阶段通常都是互相交叉地混合进行的，通常在一个阶段执行的过程中调用或激活另一个阶段。
   - 这里简要说明下 Java 中的绑定：绑定指的是把一个方法的调用与方法所在的类 (方法主体) 关联起来，对 java 来说，绑定分为静态绑定和动态绑定：
     - 静态绑定：即前期绑定。在程序执行前方法已经被绑定，此时由编译器或其它连接程序实现。针对 java，简单的可以理解为程序编译期的绑定。java 当中的方法只有 final，static，private 和构造方法是前期绑定的。
     - 动态绑定：即晚期绑定，也叫运行时绑定。在运行时根据具体对象的类型进行绑定。在 java 中，几乎所有的方法都是后期绑定的。
   - “加载”(Loading) 阶段是“类加载”(Class Loading) 过程的第一个阶段，在此阶段，虚拟机需要完成以下三件事情：
     - 通过一个类的全限定名来获取定义此类的二进制字节流。
     - 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。
     - 在 Java 堆中生成一个代表这个类的 java.lang.Class 对象，作为方法区这些数据的访问入口。
   - 验证是连接阶段的第一步，这一阶段的目的是为了确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。
   - 准备阶段是为类的静态变量分配内存并将其初始化为默认值，这些内存都将在方法区中进行分配。准备阶段不分配类中的实例变量的内存，实例变量将会在对象实例化时随着对象一起分配在 Java 堆中。
   - 解析阶段是虚拟机将常量池内的符号引用替换为直接引用的过程。
   - 类初始化是类加载过程的最后一步，前面的类加载过程，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的 Java 程序代码。

2. 双亲委派模型：Bootstrap ClassLoader、Extension ClassLoader、ApplicationClassLoader。

   - 启动类加载器，负责将存放在 <JAVA$_{HOME}$>\lib 目录中的，或者被-Xbootclasspath 参数所指定的路径中，并且是虚拟机识别的（仅按照文件名识别，如 rt.jar，名字不符合的类库即时放在 lib 目录中也不会被加载）类库加载到虚拟机内存中。启动类加载器无法被 java 程序直接引用。
   - 扩展类加载器：负责加载 <JAVA$_{HOME}$>\ext 目录中的，或者被 java.ext.dirs 系统变量所指定的路径中的所有类库，开发者可以直接使用该类加载器。
   - 应用程序类加载器：负责加载用户路径上所指定的类库，开发者可以直接使用这个类加载器，也是默认的类加载器。三种加载器的关系：启动类加载器-> 扩展类加载器-> 应用程序类加载器-> 自定义类加载器。

   这种关系即为类加载器的双亲委派模型。其要求除启动类加载器外，其余的类加载器都应当有自己的父类加载器。这里类加载器之间的父子关系一般不以继承关系实现，而是用组合的方式来复用父类的代码。

- 双亲委派模型的工作过程：如果一个类加载器接收到了类加载的请求，它首先把这个请求委托给他的父类加载器去完成，每个层次的类加载器都是如此，因此所有的加载请求都应该传送到顶层的启动类加载器中，只有当父加载器反馈自己无法完成这个加载请求（它在搜索范围中没有找到所需的类）时，子加载器才会尝试自己去加载。
- 好处：java 类随着它的类加载器一起具备了一种带有优先级的层次关系。例如类 java.lang.Object，它存放在 rt.jar 中，无论哪个类加载器要加载这个类，最终都会委派给启动类加载器进行加载，因此 Object 类在程序的各种类加载器环境中都是同一个类。相反，如果用户自己写了一个名为 java.lang.Object 的类，并放在程序的 Classpath 中，那系统中将会出现多个不同的 Object 类，java 类型体系中最基础的行为也无法保证，应用程序也会变得一片混乱。
- 实现：在 java.lang.ClassLoader 的 loadClass() 方法中，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，若父加载器为空则默认使用启动类加载器作为父加载器。如果父加载失败，则抛出 ClassNotFoundException 异常后，再调用自己的 findClass() 方法进行加载。

3. 分派：静态分派与动态分派。

- 静态分派与重载有关，虚拟机在重载时是通过参数的静态类型，而不是运行时的实际类型作为判定依据的；静态类型在编译期是可知的；动态分派与重写（Override）相关，invokevirtual(调用实例方法) 指令执行的第一步就是在运行期确定接收者的实际类型，根据实际类型进行方法调用；
- GC 收集器有哪些？CMS 收集器与 G1 收集器的特点。
- 自动内存管理机制，GC 算法，运行时数据区结构，可达性分析工作原理，如何分配对象内存
- 反射机制，双亲委派机制，类加载器的种类
- Jvm 内存模型，先行发生原则，violate 关键字作用

2.2 JVM 类加载机制

- 虚拟机把描述类的数据从 Class 文件加载到内存，并对数据进行校验、转换解析和初始化，最终形成可以被 Java 虚拟机直接使用的 Java 类型，这就是虚拟机的类加载机制。
- 类从被加载到虚拟内存中开始，到卸载内存为止，它的整个生命周期包括了：加载 (Loading)、验证 (Verification)、准备 (Preparation)、解析 (Resolution)、初始化 (Initialization)、使用 (Using) 和卸载 (Unloading) 七个阶段。其中，验证，准备和解析三个部分统称为连接 (Linking)。

2.2.1 类加载的过程类加载的全过程，加载，验证，准备，解析和初始化这五个阶段。

1. 加载在加载阶段，虚拟机需要完成以下三件事情：
   - 通过一个类的全限定名来获取定义此类的二进制字节流
   - 将这个字节流所代表的静态存储结构转换为方法区的运行时数据结构
   - 在 Java 堆中生成一个代表这个类的 java.lang.Class 对象，作为方法区这些数据的访问入口
2. 验证这一阶段的目的是为了确保 Class 文件的字节流中包含的信息符合当前虚拟机的要求，并且不会危害虚拟机自身的安全。不同的虚拟机对类验证的实现可能有所不同，但大致上都会完成下面四个阶段的检验过程：文件格式验证、元数据验证、字节码验证和符号引用验证。
   - 文件格式验证
     - 第一阶段要验证字节流是否符合 Class 文件格式的规范，并且能被当前版本的虚拟机处理。
   - 元数据验证
     - 第二阶段是对字节码描述的信息进行语义分析，以保证其描述的信息符合 Java 语言规范的要求。
   - 字节码验证
     - 第三阶段时整个验证过程中最复杂的一个阶段，主要工作是数据流和控制流的分析。在第二阶段对元数据信息中的数据类型做完校验后，这阶段将对类的方法体进行校验分析。这阶段的任务是保证被校验类的方法在运行时不会做出危害虚拟机安全的行为。
   - 符号引用验证
     - 最后一个阶段的校验发生在虚拟机将符号引用直接转化为直接引用的时候，这个转化动作将在连接的第三个阶段- 解析阶段产生。符号引用验证可以看作是对类自身以外（常量池中的各种符号引用）的信息进行匹配性的校验。

    * 准备准备阶段是正式为类变量分配内存并设置类变量初始值的阶段，这些内存都将在方法区进行分配。
    * 解析解析阶段是虚拟机将常量池的符号引用转换为直接引用的过程。解析动作主要针对类或接口、字段、类方法、接口方法四类符号引用进行。
  - 类或接口的解析
  - 字段解析
  - 类方法解析
  - 接口方法解析

3. 初始化 前面的类加载过程中，除了在加载阶段用户应用程序可以通过自定义类加载器参与之外，其余动作完全由 Java 虚拟机主导和控制。到了初始化阶段，才真正开始执行类中定义的 Java 程序代码（或者说是字节码）。在准备阶段，变量已经赋过一次系统要求的初始值，而在初始化阶段，则是根据程序员通过程序制定的主观计划去初始化类变量和其他资源，或者说初始化阶段是执行类构造器 () 方法的过程。

2.2.2 类加载器

1. 类与类加载器

   - 虚拟机设计团队把类加载阶段中的" 通过一个类的全限定名来获取描述此类的二进制字节流" 这个动作放到 Java 虚拟机外部去实现，以便让程序自己决定如何去获取所需的类。实现这个动作的代码模块被称为" 类加载器"。

2. 双亲委派模型

   - 站在 Java 虚拟机的角度讲，只存在两种不同的类加载器：一种是启动类加载器 (Bootstrap ClassLoader)，这个类加载器使用 C++ 语言实现，是虚拟机自身的一部分；另外一种就是所有其他的类加载器，这些类加载器都由 Java 语言实现，独立于虚拟机外部，并且全部继承自抽象类 java.lang.ClassLoader。从 Java 开发人员的角度来看，类加载器还可以分得更细致一些，绝大部分 Java 程序都会使用到以下三种系统提供的类加载器：
     - 启动类加载器
     - 扩展类加载器
     - 应用程序类加载器

2.3 Java 内存区域与内存溢出

- Java 虚拟机在执行 Java 程序的过程中会把他所管理的内存划分为若干个不同的数据区域。Java 虚拟机规范将 JVM 所管理的内存分为以下几个运行时数据区：程序计数器，Java 虚拟机栈，本地方法栈，Java 堆，方法区。下面详细阐述各数据区所存储的数据类型。

这里写图片描述

2.3.1 程序计数器（Program Counter Register）

- 一块较小的内存空间，它是当前线程所执行的子节码的行号指示器，字节码解释器工作时通过改变该计数器的值来选择下一条需要执行的子节码指令，分支、跳转、循环等基础功能都要依赖它来实现。每条线程都有一个独立的程序计数器，各线程间的计数器互不影响，因此该区域是线程私有的。

- 当线程在执行一个 Java 方法时，该计数器纪录的是正在执行的虚拟机字节吗指令的地址，当线程在执行的是 Native 方法 (调用本地操作系统方法) 时，该计数器的值为空。另外，该内存区域是唯一一个在 Java 虚拟机规范中没有任何 OOM（内存溢出：OutOfMemoryError）情况的区域。

2.3.2 Java 虚拟机栈（Java Virtual Machine Stacks）

- 该区域也是线程私有的，它的生命周期也与线程相同。虚拟机栈描述的是 Java 方法执行的内存模型：每个方法被执行的时候都会创建一个帧栈，栈它是用于支持虚拟机进行方法调用和方法执行的数据结构。对于执行引擎来讲，活动线程中，只有栈顶的栈帧是有效的，称为当前栈，这个栈帧所关联的方法称为当前方法，执行引擎所运行的所有字节码都只针对当前的栈帧进行操作。栈帧用于存储局部变量表、操作数栈、动态链接、方法返回地址和一些额外的附加信息。在编译程序代码时，栈帧中需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现。在 Java 虚拟机规范中，对这个区域规定了两种异常情况：

- 如果线程请求的栈深度大于虚拟机所允许的深度，将抛出 StackOverflowError 异常。
- 如果虚拟机在动态扩展栈时无法申请到足够的内存空间，则抛出 OutOfMemory 异常。

- 这两种情况存在着一些互相重叠的部分：当栈空间无法继续分配时，到底是内存太小，还是已使用的栈空间太大，其本质只是对同一件事情的两种描述而已。其本质上只是对一件事情的两种描述而已。在单线程的操作中，无论是由于栈帧太大，还是虚拟机栈空间太小，当栈空间无法分配时，虚拟机抛出的都是 StackOverflowError 异常，而不会得到 OutOfMemoryError 异常。而在多线程环境下，则会抛出 OutOfMemory 异常。

2.3.3 下面详细说明栈帧中所存放的各部分信息的作用和数据结构。

- 局部变量表是一组变量值存储空间，用于存放方法参数和方法内部定义的局部变量，其中存放的数据的类型是编译期可知的各种基本数据类型、对象引用（reference）和 returnAddress 类型（它指向了一条字节码指令的地址）。局部变量表所需的内存空间在编译期间完成分配，即在 Java 程序被编译成 Class 文件时，就确定了所需分配的最大局部变量表的容量。当进入一个方法时，这个方法需要在栈中分配多大的局部变量空间是完全确定的，在方法运行期间不会改变局部变量表的大小。下面详细说明栈帧中所存放的各部分信息的作用和数据结构。

  - 1、局部变量表局部变量表的容量以变量槽（Slot）为最小单位。在虚拟机规范中并没有明确指明一个 Slot 应占用的内存空间大小（允许其随着处理器、操作系统或虚拟机的不同而发生变化），一个 Slot 可以存放一个 32 位以内的数据类型：boolean、byte、char、short、int、float、reference 和 returnAddresss。reference 是对象的引用类型，returnAddress 是为字节指令服务的，它执行了一条字节码指令的地址。对于 64 位的数据类型（long 和 double），虚拟机会以高位在前的方式为其分配两个连续的 Slot 空间。
    * 虚拟机通过索引定位的方式使用局部变量表，索引值的范围是从 0 开始到局部变量表最大的 Slot 数量，对于 32 位数据类型的变量，索引 n 代表第 n 个 Slot，对于 64 位的，索引 n 代表第 n 和第 n+1 两个 Slot。
    * 在方法执行时，虚拟机是使用局部变量表来完成参数值到参数变量列表的传递过程的，如果是实例方法（非 static），则局部变量表中的第 0 位索引的 Slot 默认是用于传递方法所属对象实例的引用，在方法中可以通过关键字“this”来访问这个隐含的参数。其余参数则按照参数表的顺序来排列，占用从 1 开始的局部变量 Slot，参数表分配完毕后，再根据方法体内部定义的变量顺序和作用域分配其余的 Slot。
    * 局部变量表中的 Slot 是可重用的，方法体中定义的变量，作用域并不一定会覆盖整个方法体，如果当前字节码 PC 计数器的值已经超过了某个变量的作用域，那么这个变量对应的 Slot 就可以交给其他变量使用。这样的设计不仅仅是为了节省空间，在某些情况下 Slot 的复用会直接影响到系统的而垃圾收集行为。
  - 2、操作数栈
    * 操作数栈又常被称为操作栈，操作数栈的最大深度也是在编译的时候就确定了。32 位数据类型所占的栈容量为 1,64 为数据类型所占的栈容量为 2。当一个方法开始执行时，它的操作栈是空的，在方法的执行过程中，会有各种字节码指令（比如：加操作、赋值元算等）向操作栈中写入和提取内容，也就是入栈和出栈操作。
    * Java 虚拟机的解释执行引擎称为“基于栈的执行引擎”，其中所指的“栈”就是操作数栈。因此我们也称 Java 虚拟机是基于栈的，这点不同于 Android 虚拟机，Android 虚拟机是基于寄存器的。
    * 基于栈的指令集最主要的优点是可移植性强，主要的缺点是执行速度相对会慢些；而由于寄存器由硬件直接提供，所以基于寄存器指令集最主要的优点是执行速度快，主要的缺点是可移植性差。
  - 3、动态连接
    * 每个栈帧都包含一个指向运行时常量池（在方法区中，后面介绍）中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态连接。Class 文件的常量池中存在有大量的符号引用，字节码中的方法调用指令就以常量池中指向方法的符号引用为参数。这些符号引用，一部分会在类加载阶段或第一次使用的时候转化为直接引用（如 final、static 域等），称为静态解析，另一部分将在每一次的运行期间转化为直接引用，这部分称为动态连接。
  - 4、方法返回地址
    * 当一个方法被执行后，有两种方式退出该方法：执行引擎遇到了任意一个方法返回的字节码指令或遇到了异常，并且该异常没有在方法体内得到处理。无论采用何种退出方式，在方法退出之后，都需要返回到方法被调用的位置，程序才能继续执行。方法返回时可能需要在栈帧中保存一些信息，用来帮助恢复它的上层方法的执行状态。一般来说，方法正常退出时，调用者的 PC 计数器的值就可以作为返回地址，栈帧中很可能保存了这个计数器值，而方法异常退出时，返回地址是要通过异常处理器来确定的，栈帧中一般不会保存这部分信息。

* 方法退出的过程实际上等同于把当前栈帧出站，因此退出时可能执行的操作有：恢复上层方法的局部变量表和操作数栈，如果有返回值，则把它压入调用者栈帧的操作数栈中，调整 PC 计数器的值以指向方法调用指令后面的一条指令。

2.3.4 本地方法栈（Native Method Stacks）

- 该区域与虚拟机栈所发挥的作用非常相似，只是虚拟机栈为虚拟机执行 Java 方法服务，而本地方法栈则为使用到的本地操作系统（Native）方法服务。

2.3.5 Java 堆（Java Heap）

- Java Heap 是 Java 虚拟机所管理的内存中的最大的一块，它是所有线程共享的一块内存区域。几乎所有的对象实例和数组都在这类分配内存。Java Heap 是垃圾收集器管理的主要区域，因此很多时候也被称为"GC 堆"。
- 根据 Java 虚拟机的规定，Java 堆可以处在物理上不连续的内存空间中，只要逻辑上是连续的即可。如果在堆中没有内存可分配时，并且堆也无法扩展时，将会抛出 OutOfMemory。

2.3.6 方法区（Method Area）

- 方法区也是各个线程共享的内存区域，它用于存储已经被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。方法区域又被称为" 永久代"。但着这仅仅对于 Sun HotSpot 来讲，JRocket 和 IBMJ9 虚拟机中并不存在永久代的概念。Java 虚拟机规范把方法区描述为 Java 堆的一个逻辑部分，而且它和 Java Heap 一样不需要连续的内存，可以选择固定大小或可扩展，另外，虚拟机规范允许该区域可以选择不实现垃圾回收。相对而言，垃圾收集行为在这个区域比较少出现。该区域的内存回收目标主要针是对废弃常量的和无用类的回收。运行时常量池是方法区的一部分，Class 文件中除了有类的版本、字段、方法、接口等描述信息外，还有一项信息是常量池（Class 文件常量池），用于存放编译器生成的各种字面量和符号引用，这部分内容将在类加载后存放到方法区的运行时常量池中。运行时常量池相对于 Class 文件常量池的另一个重要特征是具备动态性，Java 语言并不要求常量一定只能在编译期产生，也就是并非预置入 Class 文件中的常量池的内容才能进入方法区的运行时常量池，运行期间也可能将新的常量放入池中，这种特性被开发人员利用比较多的是 String 类的 intern （） 方法。
- 根据 Java 虚拟机规范的规定，当方法区无法满足内存分配需求时，将抛出 OutOfMemoryError 异常。

2.3.7 直接内存（Direct Memory）

- 直接内存并不是虚拟机运行内存时数据区的一部分，也不是 Java 虚拟机规范中定义的内存区域，它直接从操作系统中分配内存，因此不受 Java 堆的大小的限制，但是会受到本机总内存的大小及处理器寻址空间的限制，因此它也可能导致 OutOfMemoryError 异常出现。在 Java1.4 中新引入了 NIO 机制，它是一种基于通道与缓冲区的新 I/O 方式，可以直接从操作系统中分配直接内存，可以直接从操作系统中分配直接内存，即在堆外分配内存，这样能在一些场景中提高性能，因为避免了在 Java 堆和 Native 堆中来回复制数据。

2.3.8 内存溢出

- 下面给出个内存区域内存溢出的简单测试方法
- 这里写图片描述

<table>
<tr><th>内存区域</th><th colspan="2">内存溢出的测试方法</th></tr>
<tr><td>Java 堆</td><td colspan="2">无限循环地 new 对象出来，在 List 中保存引用，以不被垃圾收集器回收。另外，该区域也有可能会发生内存泄露（Memory Leak），出现问题时，要注意区别。</td></tr>
<tr><td>方法区</td><td colspan="2">生成大量的动态类，或无线循环调用 String 的 intern（）方法产生不同的 String 对象实例，并在 List 中保存其引用，以不被垃圾收集器回收。后者测试常量池，前者测试方法区的非常量池部分。</td></tr>
<tr><td rowspan="2">虚拟机栈和本地方法栈</td><td>单线程</td><td>多线程</td></tr>
<tr><td>递归调用一个简单的方法：如不断累积的方法。
会抛出 StackOverflowError</td><td>无线循环地创建线程，并未每个线程无限循环地增加内存。
会抛出 OutOfMemoryError</td></tr>
</table>

- 这里有一点要重点说明，在多线程情况下，给每个线程的栈分配的内存越大，反而越容易产生内存产生内存溢出一场。操作系统为每个进程分配的内存是有限制的，虚拟机提供了参数来控制 Java 堆和方法区这两部分内存的最大值，忽略掉程序计数器消耗的内存（很小），以及进程本身消耗的内存，剩下的内存便给了虚拟机栈和本地方法栈，每个线程分配到的栈容量越大，可以建立的线程数量自然就越少。因此，如果是建立过多的线程导致的内存溢出，在不能减少线程数的情况下，就只能通过减少最大堆和每个线程的栈容量来换取更多的线程。另外，由于 Java 堆内也可能发生内存泄露（Memory Leak），这里简要说明一下内存泄露和内存溢出的区别：
- 内存泄漏是指分配出去的内存没有被回收回来，由于失去了对该内存区域的控制，因而造成了资源的浪费。Java 中一般不会产生内存泄漏，因为有垃圾回收器自动回收垃圾，但这也不绝对，当我们 new 了对象，并保存了其引用，但是后面一直没用它，而垃圾回收器又不会去回收它，这就会造成内存泄漏。
- 内存溢出是指程序所需要的内存超过了系统所能分配的内存（包括动态扩展）的上限。

2.3.9 对象实例化分析

- 对内存分配情况分析最常见的示例便是对象实例化：

```
Object obj = new Object();
```

- 这段代码的执行会涉及 java 栈、Java 堆、方法区三个最重要的内存区域。假设该语句出现在方法体中，及时对 JVM 虚拟机不了解的 Java 使用这，应该也知道 obj 会作为引用类型（reference）的数据保存在 Java 栈的本地变量表中，而会在 Java 堆中保存该引用的实例化对象，但可能并不知道，Java 堆中还必须包含能查找到此对象类型数据的地址信息（如对象类型、父类、实现的接口、方法等），这些类型数据则保存在方法区中。
- 另外，由于 reference 类型在 Java 虚拟机规范里面只规定了一个指向对象的引用，并没有定义这个引用应该通过哪种方式去定位，以及访问到 Java 堆中的对象的具体位置，因此不同虚拟机实现的对象访问方式会有所不同，主流的访问方式有两种：使用句柄池和直接使用指针。
- 通过句柄池访问的方式如下：
- 这里写图片描述
- 通过直接指针访问的方式如下：
- 这里写图片描述
- 这两种对象的访问方式各有优势，使用句柄访问方式的最大好处就是 reference 中存放的是稳定的句柄地址，在对象被移动（垃圾收集时移动对象是非常普遍的行为）时只会改变句柄中的实例数据指针，而 reference 本身不需要修改。使用直接指针访问方式的最大好处是速度快，它节省了一次指针定位的时间开销。目前 Java 默认使用的 HotSpot 虚拟机采用的便是是第二种方式进行对象访问的。

2.4 垃圾回收算法

- 引用计数法：缺点是无法处理循环引用问题
- 标记-清除法：标记所有从根结点开始的可达对象，缺点是会造成内存空间不连续，不连续的内存空间的工作效率低于连续的内存空间，不容易分配内存
- 复制算法：将内存空间分成两块，每次将正在使用的内存中存活对象复制到未使用的内存块中，之后清除正在使用的内存块。算法效率高，但是代价是系统内存折半。适用于新生代 (存活对象少，垃圾对象多)
- 标记- 压缩算法：标记- 清除的改进，清除未标记的对象时还将所有的存活对象压缩到内存的一端，之后，清理边界所有空间既避免碎片产生，又不需要两块同样大小的内存快，性价比高。适用于老年代。
- 分代

3 JavaConcurrent(Java 并发)

3.1 Java 并发基础知识

Executor 框架和多线程基础）

3.1.1 Thread 与 Runable 如何实现多线程

- Java 5 以前实现多线程有两种实现方法：一种是继承 Thread 类；另一种是实现 Runnable 接口。两种方式都要通过重写 run() 方法来定义线程的行为，推荐使用后者，因为 Java 中的继承是单继承，一个类有一个父类，如果继承了 Thread 类就无法再继承其他类了，显然使用 Runnable 接口更为灵活。
- 实现 Runnable 接口相比继承 Thread 类有如下优势：
  - 可以避免由于 Java 的单继承特性而带来的局限
  - 增强程序的健壮性，代码能够被多个程序共享，代码与数据是独立的
  - 适合多个相同程序代码的线程区处理同一资源的情况
- 补充：Java 5 以后创建线程还有第三种方式：实现 Callable 接口，该接口中的 call 方法可以在线程执行结束时产生一个返回值，代码如下所示：

```
class MyTask implements Callable<Integer> {
    private int upperBounds;

    public MyTask(int upperBounds) {
        this.upperBounds = upperBounds;
    }

    @Override
    public Integer call() throws Exception {
        int sum = 0;
        for (int i = 1; i <= upperBounds; i++)
            sum += i;
        return sum;
    }
}

public class Test {
    public static void main(String[] args) throws Exception {
        List<Future<Integer>> list = new ArrayList<>();
        ExecutorService service = Executors.newFixedThreadPool(10);
        for(int i = 0; i < 10; i++) {
            list.add(service.submit(new MyTask((int) (Math.random() * 100))));
        }

        int sum = 0;
        for(Future<Integer> future : list) {
            while(!future.isDone()) ;
            sum += future.get();
        }

        System.out.println(sum);
    }
}
```

- 线程同步的方法有什么；锁，synchronized 块，信号量等
- 锁的等级：方法锁、对象锁、类锁
- 生产者消费者模式的几种实现，阻塞队列实现，sync 关键字实现，lock 实现,reentrantLock 等
- ThreadLocal 的设计理念与作用，ThreadPool 用法与优势（这里在 Android SDK 原生的 AsyncTask 底层也有使用）
- 线程池的底层实现和工作原理（建议写一个雏形简版源码实现）
- 几个重要的线程 api，interrupt，wait，sleep，stop 等等

- 写出生产者消费者模式。
- ThreadPool 用法与优势。
- Concurrent 包里的其他东西：ArrayBlockingQueue、CountDownLatch 等等。
- wait() 和 sleep() 的区别。
  - sleep() 方法是线程类（Thread）的静态方法，导致此线程暂停执行指定时间，将执行机会给其他线程，但是监控状态依然保持，到时后会自动恢复（线程回到就绪（ready）状态），因为调用 sleep 不会释放对象锁。wait() 是 Object 类的方法，对此对象调用 wait() 方法导致本线程放弃对象锁（线程暂停执行），进入等待此对象的等待锁定池，只有针对此对象发出 notify 方法（或 notifyAll）后本线程才进入对象锁定池准备获得对象锁进入就绪状态。

3.1.2 IO（IO,NIO，目前 okio 已经被集成 Android 包）

- IO 框架主要用到什么设计模式
- JDK 的 I/O 包中就主要使用到了两种设计模式：Adatper 模式和 Decorator 模式。
- NIO 包有哪些结构？分别起到的作用？
- NIO 针对什么情景会比 IO 有更好的优化？
- OKIO 底层实现

3.2 生产者和消费者问题

```
package 生产者消费者;
public class ProducerConsumerTest {
^^Ipublic static void main(String[] args) {
^^I^^IPublicResource resource = new PublicResource();
^^I^^Inew Thread(new ProducerThread(resource)).start();
        new Thread(new ConsumerThread(resource)).start();
        new Thread(new ProducerThread(resource)).start();
        new Thread(new ConsumerThread(resource)).start();
        new Thread(new ProducerThread(resource)).start();
        new Thread(new ConsumerThread(resource)).start();
^^I}
}


package 生产者消费者;
/**
 * 生产者线程，负责生产公共资源
 * @author dream
 *
 */
public class ProducerThread implements Runnable{
^^Iprivate PublicResource resource;

^^Ipublic ProducerThread(PublicResource resource) {
^^I^^Ithis.resource = resource;
^^I}

^^I@Override
^^Ipublic void run() {
^^I^^Iwhile (true) {
^^I^^I^^Itry {
^^I^^I^^I^^IThread.sleep((long) (Math.random() * 1000));
^^I^^I^^I} catch (InterruptedException e) {
^^I^^I^^I^^Ie.printStackTrace();
```

```
^^I^^I^^I}
^^I^^I^^Iresource.increase();
^^I^^I}
^^I}


}


package 生产者消费者;
/**
 * 消费者线程，负责消费公共资源
 * @author dream
 *
 */
public class ConsumerThread implements Runnable{
^^Iprivate PublicResource resource;

^^Ipublic ConsumerThread(PublicResource resource) {
^^I^^Ithis.resource = resource;
^^I}

^^I@Override
^^Ipublic void run() {
^^I^^Iwhile (true) {
^^I^^I^^Itry {
^^I^^I^^I^^IThread.sleep((long) (Math.random() * 1000));
^^I^^I^^I} catch (InterruptedException e) {
^^I^^I^^I^^I// TODO Auto-generated catch block
^^I^^I^^I^^Ie.printStackTrace();
^^I^^I^^I}
^^I^^I^^Iresource.decrease();
^^I^^I}

^^I}

}
package 生产者消费者;
/**
 * 公共资源类
 * @author dream
 *
 */
public class PublicResource {
^^Iprivate int number = 0;
^^Iprivate int size = 10;

^^I/**
^^I * 增加公共资源
^^I */
^^Ipublic synchronized void increase() {
^^I^^Iwhile (number >= size) {
^^I^^I^^Itry {
^^I^^I^^I^^Iwait();
^^I^^I^^I} catch (InterruptedException e) {
^^I^^I^^I^^I// TODO Auto-generated catch block
^^I^^I^^I^^Ie.printStackTrace();
^^I^^I^^I}
```

```
^^I^^I}
^^I^^Inumber++;
^^I^^ISystem.out.println(" 生产了 1 个，总共有" + number);
^^I^^InotifyAll();
^^I}

^^I/**
^^I * 减少公共资源
^^I */
^^Ipublic synchronized void decrease() {
^^I^^Iwhile (number <= 0) {
^^I^^I^^Itry {
^^I^^I^^I^^Iwait();
^^I^^I^^I} catch (InterruptedException e) {
^^I^^I^^I^^I// TODO Auto-generated catch block
^^I^^I^^I^^Ie.printStackTrace();
^^I^^I^^I}
^^I^^I}
^^I^^Inumber--;
^^I^^ISystem.out.println(" 消费了 1 个，总共有" + number);
^^I^^InotifyAll();
^^I}
}
```

3.3 Thread 和 Runnable 实现多线程的区别

- Java 中实现多线程有两种方法：继承 Thread、实现 Runnable 接口，在程序开发中只要是多线程，肯定永远以实现 Runnable 接口为主，因为实现 Runnable 接口相比继承 Thread 类有如下优势：
  - 可以避免由于 Java 的单继承特性而带来的局限
  - 增强程序的健壮性，代码能够被多个线程共享，代码与数据是独立的
  - 适合多个相同程序的线程区处理同一资源的情况
- 首先通过 Thread 类实现

```
class MyThread extends Thread{
    private int ticket = 5;
    public void run(){
        for (int i=0;i<10;i++) {
            if(ticket > 0)
                System.out.println("ticket = " + ticket--);
        }
    }
}

public class ThreadDemo{
    public static void main(String[] args){
        new MyThread().start();
        new MyThread().start();
        new MyThread().start();
    }
}
```

- 运行结果：

```
ticket = 5
ticket = 4
ticket = 5
ticket = 5
ticket = 4
```

```
ticket = 3
ticket = 2
ticket = 1
ticket = 4
ticket = 3
ticket = 3
ticket = 2
ticket = 1
ticket = 2
ticket = 1
```

- 每个线程单独卖了 5 张票，即独立的完成了买票的任务，但实际应用中，比如火车站售票，需要多个线程去共同完成任务，在本例中，即多个线程共同买 5 张票。

- 通过实现 Runnable 借口实现的多线程程序

```
class MyThread implements Runnable{
    private int ticket = 5;
    public void run(){
        for (int i=0;i<10;i++)
        {
            if(ticket > 0){
                System.out.println("ticket = " + ticket--);
            }
        }
    }
}

public class RunnableDemo{
    public static void main(String[] args){
        MyThread my = new MyThread();
        new Thread(my).start();
        new Thread(my).start();
        new Thread(my).start();
    }
}
```

- 运行结果

```
ticket = 5
ticket = 2
ticket = 1
ticket = 3
ticket = 4
```

- 在第二种方法 (Runnable) 中，ticket 输出的顺序并不是 54321，这是因为线程执行的时机难以预测。ticket 并不是原子操作。

- 在第一种方法中，我们 new 了 3 个 Thread 对象，即三个线程分别执行三个对象中的代码，因此便是三个线程去独立地完成卖票的任务；而在第二种方法中，我们同样也 new 了 3 个 Thread 对象，但只有一个 Runnable 对象，3 个 Thread 对象共享这个 Runnable 对象中的代码，因此，便会出现 3 个线程共同完成卖票任务的结果。如果我们 new 出 3 个 Runnable 对象，作为参数分别传入 3 个 Thread 对象中，那么 3 个线程便会独立执行各自 Runnable 对象中的代码，即 3 个线程各自卖 5 张票。

- 在第二种方法中，由于 3 个 Thread 对象共同执行一个 Runnable 对象中的代码，因此可能会造成线程的不安全，比如可能 ticket 会输出-1（如果我们 System.out.... 语句前加上线程休眠操作，该情况将很有可能出现），这种情况的出现是由于，一个线程在判断 ticket 为 1>0 后，还没有来得及减 1，另一个线程已经将 ticket 减 1，变为了 0，那么接下来之前的线程再将 ticket 减 1，便得到了-1。这就需要加入同步操作（即互斥锁），确保同一时刻只有一个线程在执行每次 for 循环中的操作。而在第一种方法中，并不需要加入同步操作，因为每个线程执行自己 Thread 对象中的代码，不存在多个线程共同执行同一个方法的情况。

3.4 线程中断

3.4.1 使用 interrupt() 中断线程

- 当一个线程运行时，另一个线程可以调用对应的 Thread 对象的 interrupt () 方法来中断它，该方法只是在目标线程中设置一个标志，表示它已经被中断，并立即返回。这里需要注意的是，如果只是单纯的调用 interrupt () 方法，线程并没有实际被中断，会继续往下执行。

- 演示休眠线程的中断

```
public class SleepInterrupt extends Object implements Runnable{
^^I@Override
^^Ipublic void run() {

^^I^^Itry {
^^I^^I^^ISystem.out.println("in run() - about to sleep for 20 seconds");
^^I^^I^^IThread.sleep(20000);
^^I^^I^^ISystem.out.println("in run() - woke up");
^^I^^I} catch (InterruptedException e) {
^^I^^I^^ISystem.out.println("in run() - interrupted while sleeping");
^^I^^I^^I//处理完中断异常后，返回到 run() 方法入口
^^I^^I^^I//如果没有 return，线程不会实际被中断，它会继续打印下面的信息
^^I^^I^^Ireturn;
^^I^^I}
^^I^^ISystem.out.println("in run() - leaving normally");
^^I}

^^Ipublic static void main(String[] args) {
^^I^^ISleepInterrupt si = new SleepInterrupt();
^^I^^IThread t = new Thread(si);
^^I^^It.start();
^^I^^I//住线程休眠 2 秒，从而确保刚才启动的线程有机会执行一段时间
^^I^^Itry {
^^I^^I^^IThread.sleep(2000);
^^I^^I} catch (InterruptedException e) {
^^I^^I^^Ie.printStackTrace();
^^I^^I}
^^I^^ISystem.out.println("in main() - interrupting other thread");
^^I^^I//中断线程 t
^^I^^It.interrupt();
^^I^^ISystem.out.println("in main() - leaving");
^^I}
}
```

- 运行结果如下：

```
in run() - about to sleep for 20 seconds
in main() - interrupting other thread
in main() - leaving
in run() - interrupted while sleeping
```

- 主线程启动新线程后，自身休眠 2 秒钟，允许新线程获得运行时间。新线程打印信息“about to sleep for 20 seconds”后，继而休眠 20 秒钟，大约 2 秒钟后，main 线程通知新线程中断，那么新线程的 20 秒的休眠将被打断，从而抛出 InterruptException 异常，执行跳转到 catch 块，打印出“interrupted while sleeping”信息，并立即从 run () 方法返回，然后消亡，而不会打印出 catch 块后面的“leaving normally”信息。

- 请注意：由于不确定的线程规划，上图运行结果的后两行可能顺序相反，这取决于主线程和新线程哪个先消亡。但前两行信息的顺序必定如上图所示。

- 另外，如果将 catch 块中的 return 语句注释掉，则线程在抛出异常后，会继续往下执行，而不会被中断，从而会打印出”leaving normally“信息。

3.4.2 待决中断

- 在上面的例子中，sleep () 方法的实现检查到休眠线程被中断，它会相当友好地终止线程，并抛出 InterruptedException 异常。另外一种情况，如果线程在调用 sleep () 方法前被中断，那么该中断称为待决中断，它会在刚调用 sleep () 方法时，立即抛出 InterruptedException 异常。

```
public class PendingInterrupt extends Object{
^^Ipublic static void main(String[] args) {
^^I^^I//如果输入了参数，则在 main 线程中中断当前线程（即 main 线程）
^^I^^Iif(args.length > 0){
^^I^^I^^IThread.currentThread().interrupt();
^^I^^I}
^^I^^I//获取当前时间
^^I^^Ilong startTime = System.currentTimeMillis();
^^I^^Itry {
^^I^^I^^IThread.sleep(2000);
^^I^^I^^ISystem.out.println("was NOT interrupted");
^^I^^I} catch (InterruptedException e) {
^^I^^I^^ISystem.out.println("was interrupted");
^^I^^I}
^^I^^I//计算中间代码执行的时间
^^I^^ISystem.out.println("elapsedTime=" + (System.currentTimeMillis() - startTime));
^^I}
}
```

- 如果 PendingInterrupt 不带任何命令行参数，那么线程不会被中断，最终输出的时间差距应该在 2000 附近（具体时间由系统决定，不精确），如果 PendingInterrupt 带有命令行参数，则调用中断当前线程的代码，但 main 线程仍然运行，最终输出的时间差距应该远小于 2000，因为线程尚未休眠，便被中断，因此，一旦调用 sleep () 方法，会立即打印出 catch 块中的信息。执行结果如下:

```
was NOT interrupted
elapsedTime=2001
```

- 这种模式下，main 线程中断它自身。除了将中断标志（它是 Thread 的内部标志）设置为 true 外，没有其他任何影响。线程被中断了，但 main 线程仍然运行，main 线程继续监视实时时钟，并进入 try 块，一旦调用 sleep () 方法，它就会注意到待决中断的存在，并抛出 InterruptException。于是执行跳转到 catch 块，并打印出线程被中断的信息。最后，计算并打印出时间差。

3.4.3 使用 isInterrupted () 方法判断中断状态

- 可以在 Thread 对象上调用 isInterrupted () 方法来检查任何线程的中断状态。这里需要注意：线程一旦被中断，isInterrupted () 方法便会返回 true，而一旦 sleep () 方法抛出异常，它将清空中断标志，此时 isInterrupted () 方法将返回 false。

- 下面的代码演示了 isInterrupted () 方法的使用:

```
public class InterruptCheck extends Object{

^^Ipublic static void main(String[] args) {
^^I^^IThread t = Thread.currentThread();
^^I^^ISystem.out.println("Point A: t.isInterrupted()=" + t.isInterrupted());
        //待决中断，中断自身
        t.interrupt();
        System.out.println("Point B: t.isInterrupted()=" + t.isInterrupted());
        System.out.println("Point C: t.isInterrupted()=" + t.isInterrupted());

        try {
^^I^^I^^IThread.sleep(2000);
^^I^^I^^ISystem.out.println("was NOT interrupted");
^^I^^I} catch (InterruptedException e) {
^^I^^I^^ISystem.out.println("was interrupted");
```

```
^^I^^I}
        //跑出异常后，会清除中断标志，这里会返回 false
        System.out.println("Point D: t.isInterrupted()=" + t.isInterrupted());
^^I}
}
```

- 运行结果如下：

```
Point A: t.isInterrupted()=false
Point B: t.isInterrupted()=true
Point C: t.isInterrupted()=true
was interrupted
Point D: t.isInterrupted()=false
```

3.4.4 使用 Thread.interrupted () 方法判断中断状态

- 可以使用 Thread.interrupted () 方法来检查当前线程的中断状态（并隐式重置为 false）。又由于它是静态方法，因此不能在特定的线程上使用，而只能报告调用它的线程的中断状态，如果线程被中断，而且中断状态尚不清楚，那么，这个方法返回 true。与 isInterrupted () 不同，它将自动重置中断状态为 false，第二次调用 Thread.interrupted () 方法，总是返回 false，除非中断了线程。

- 如下代码演示了 Thread.interrupted () 方法的使用：

```
public class InterruptReset extends Object{

^^Ipublic static void main(String[] args) {
^^I^^ISystem.out.println(
^^I            "Point X: Thread.interrupted()=" + Thread.interrupted());
^^I        Thread.currentThread().interrupt();
^^I        System.out.println(
^^I            "Point Y: Thread.interrupted()=" + Thread.interrupted());
^^I        System.out.println(
^^I            "Point Z: Thread.interrupted()=" + Thread.interrupted());
^^I}
}
运行结果
Point X: Thread.interrupted()=false
Point Y: Thread.interrupted()=true
Point Z: Thread.interrupted()=false
```

- 从结果中可以看出，当前线程中断自身后，在 Y 点，中断状态为 true，并由 Thread.interrupted () 自动重置为 false，那么下次调用该方法得到的结果便是 false。

3.4.5 补充

- yield 和 join 方法的使用
  - join 方法用线程对象调用，如果在一个线程 A 中调用另一个线程 B 的 join 方法，线程 A 将会等待线程 B 执行完毕后再执行。
  - yield 可以直接用 Thread 类调用，yield 让出 CPU 执行权给同等级的线程，如果没有相同级别的线程在等待 CPU 的执行权，则该线程继续执行。

3.5 守护线程与阻塞线程

- Java 中有两类线程：User Thread(用户线程)、Daemon Thread(守护线程)

- 用户线程即运行在前台的线程，而守护线程是运行在后台的线程。守护线程作用是为其他前台线程的运行提供便利服务，而且仅在普通、非守护线程仍然运行时才需要，比如垃圾回收线程就是一个守护线程。当 VM 检测仅剩一个守护线程，而用户线程都已经退出运行时，VM 就会退出，因为如果没有了守护者，也就没有继续运行程序的必要了。如果有非守护线程仍然活着，VM 就不会退出。

- 守护线程并非只有虚拟机内部提供，用户在编写程序时也可以自己设置守护线程。用户可以用 Thread 的 setDaemon(true) 方法设置当前线程为守护线程。
- 虽然守护线程可能非常有用，但必须小心确保其它所有非守护线程消亡时，不会由于它的终止而产生任何危害。因为你不可能知道在所有的用户线程退出运行前，守护线程是否已经完成了预期的服务任务。一旦所有的用户线程退出了，虚拟机也就退出运行了。因此，不要再守护线程中执行业务逻辑操作 (比如对数据的读写等)。
- 还有几点：
  - setDaemon(true) 必须在调用线程的 start() 方法之前设置，否则会跑出 IllegalThreadStateException 异常。
  - 在守护线程中产生的新线程也是守护线程
  - 不要认为所有的应用都可以分配给守护线程来进行服务，比如读写操作或者计算逻辑。

3.5.1 线程阻塞

- 线程可以阻塞于四种状态：
  - 当线程执行 Thread.sleep() 时，它一直阻塞到指定的毫秒时间之后，或者阻塞被另一个线程打断
  - 当线程碰到一条 wait() 语句时，它会一直阻塞到接到通知 (notify())、被中断或经过了指定毫秒时间为止 (若指定了超时值的话)
  - 线程阻塞与不同的 I/O 的方式有多种。常见的一种方式是 InputStream 的 read() 方法，该方法一直阻塞到从流中读取一个字节的数据为止，它可以无限阻塞，因此不能指定超时时间
  - 线程也可以阻塞等待获取某个对象锁的排它性访问权限 (即等待获得 synchronized 语句必须的锁时阻塞)
- 并非所有的阻塞状态都是可中断的，以上阻塞状态的前两种可以被中断，后两种不会对中断做出反应。

3.6 synchronized

- 在并发编程中，多线程同时并发访问的资源叫做临界资源，当多个线程同时访问对象并要求操作相同资源时，分割了原子操作就有可能出现数据的不一致或数据不完整的情况，为避免这种情况的发生，我们会采取同步机制，以确保在某一时刻，方法内只允许有一个线程。
- 采用 synchronized 修饰符实现的同步机制叫做互斥锁机制，它所获得的锁叫做互斥锁。每个对象都有一个 monitor(锁标记)，当线程拥有这个锁标记时才能访问这个资源，没有锁标记便进入锁池。任何一个对象系统都会为其创建一个互斥锁，这个锁是为了分配给线程的，防止打断原子操作。每个对象的锁只能分配给一个线程，因此叫做互斥锁。
- 这里就使用同步机制获取互斥锁的情况，进行几点说明：
  - 如果同一个方法内同时有两个或更多线程，则每个线程有自己的局部变量拷贝。
  - 类的每个实例都有自己的对象级别锁。当一个线程访问实例对象中的 synchronized 同步代码块或同步方法时，该线程便获取了该实例的对象级别锁，其他线程这时如果要访问 synchronized 同步代码块或同步方法，便需要阻塞等待，直到前面的线程从同步代码块或方法中退出，释放掉了该对象级别锁。
  - 访问同一个类的不同实例对象中的同步代码块，不存在阻塞等待获取对象锁的问题，因为它们获取的是各自实例的对象级别锁，相互之间没有影响。
  - 持有一个对象级别锁不会阻止该线程被交换出来，也不会阻塞其他线程访问同一示例对象中的非 synchronized 代码。当一个线程 A 持有一个对象级别锁（即进入了 synchronized 修饰的代码块或方法中）时，线程也有可能被交换出去，此时线程 B 有可能获取执行该对象中代码的时间，但它只能执行非同步代码（没有用 synchronized 修饰），当执行到同步代码时，便会被阻塞，此时可能线程规划器又让 A 线程运行，A 线程继续持有对象级别锁，当 A 线程退出同步代码时（即释放了对象级别锁），如果 B 线程此时再运行，便会获得该对象级别锁，从而执行 synchronized 中的代码。
  - 持有对象级别锁的线程会让其他线程阻塞在所有的 synchronized 代码外。例如，在一个类中有三个 synchronized 方法 a，b，c，当线程 A 正在执行一个实例对象 M 中的方法 a 时，它便获得了该对象级别锁，那么其他的线程在执行同一实例对象（即对象 M）中的代码时，便会在所有的 synchronized 方法处阻塞，即在方法 a，b，c 处都要被阻塞，等线程 A 释放掉对象级别锁时，其他的线程才可以去执行方法 a，b 或者 c 中的代码，从而获得该对象级别锁。

- 使用 synchronized（obj）同步语句块，可以获取指定对象上的对象级别锁。obj 为对象的引用，如果获取了 obj 对象上的对象级别锁，在并发访问 obj 对象时时，便会在其 synchronized 代码处阻塞等待，直到获取到该 obj 对象的对象级别锁。当 obj 为 this 时，便是获取当前对象的对象级别锁。
- 类级别锁被特定类的所有示例共享，它用于控制对 static 成员变量以及 static 方法的并发访问。具体用法与对象级别锁相似。
- 互斥是实现同步的一种手段，临界区、互斥量和信号量都是主要的互斥实现方式。synchronized 关键字经过编译后，会在同步块的前后分别形成 monitorenter 和 monitorexit 这两个字节码指令。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，如果获得了锁，把锁的计数器加 1，相应地，在执行 monitorexit 指令时会将锁计数器减 1，当计数器为 0 时，锁便被释放了。由于 synchronized 同步块对同一个线程是可重入的，因此一个线程可以多次获得同一个对象的互斥锁，同样，要释放相应次数的该互斥锁，才能最终释放掉该锁。

3.6.1 内存可见性

- 加锁（synchronized 同步）的功能不仅仅局限于互斥行为，同时还存在另外一个重要的方面：内存可见性。我们不仅希望防止某个线程正在使用对象状态而另一个线程在同时修改该状态，而且还希望确保当一个线程修改了对象状态后，其他线程能够看到该变化。而线程的同步恰恰也能够实现这一点。
- 内置锁可以用于确保某个线程以一种可预测的方式来查看另一个线程的执行结果。为了确保所有的线程都能看到共享变量的最新值，可以在所有执行读操作或写操作的线程上加上同一把锁。下图示例了同步的可见性保证。

  ./pic/a
- 当线程 A 执行某个同步代码块时，线程 B 随后进入由同一个锁保护的同步代码块，这种情况下可以保证，当锁被释放前，A 看到的所有变量值（锁释放前，A 看到的变量包括 y 和 x）在 B 获得同一个锁后同样可以由 B 看到。换句话说，当线程 B 执行由锁保护的同步代码块时，可以看到线程 A 之前在同一个锁保护的同步代码块中的所有操作结果。如果在线程 A unlock M 之后，线程 B 才进入 lock M，那么线程 B 都可以看到线程 A unlock M 之前的操作，可以得到 i=1，j=1。如果在线程 B unlock M 之后，线程 A 才进入 lock M，那么线程 B 就不一定能看到线程 A 中的操作，因此 j 的值就不一定是 1。
- 现在考虑如下代码：

```
public class  MutableInteger {
    private int value;

    public int get(){
        return value;
    }

    public void set(int value){
        this.value = value;
    }
}
```

- 以上代码中，get 和 set 方法都在没有同步的情况下访问 value。如果 value 被多个线程共享，假如某个线程调用了 set，那么另一个正在调用 get 的线程可能会看到更新后的 value 值，也可能看不到。
- 通过对 set 和 get 方法进行同步，可以使 MutableInteger 成为一个线程安全的类，如下：

```
public class  SynchronizedInteger {
    private int value;

    public synchronized int get(){
        return value;
    }

    public synchronized void set(int value){
        this.value = value;
    }
}
```

- 对 set 和 get 方法进行了同步，加上了同一把对象锁，这样 get 方法可以看到 set 方法中 value 值的变化，从而每次通过 get 方法取得的 value 的值都是最新的 value 值。

3.7 多线程环境中安全使用集合 API

- 在集合 API 中，最初设计的 Vector 和 Hashtable 是多线程安全的。例如：对于 Vector 来说，用来添加和删除元素的方法是同步的。如果只有一个线程与 Vector 的实例交互，那么，要求获取和释放对象锁便是一种浪费，另外在不必要的时候如果滥用同步化，也有可能会带来死锁。因此，对于更改集合内容的方法，没有一个是同步化的。集合本质上是非多线程安全的，当多个线程与集合交互时，为了使它多线程安全，必须采取额外的措施。

- 在 Collections 类中有多个静态方法，它们可以获取通过同步方法封装非同步集合而得到的集合：

```
public static Collection synchronizedCollention(Collection c)
public static List synchronizedList(list l)
public static Map synchronizedMap(Map m)
public static Set synchronizedSet(Set s)
public static SortedMap synchronizedSortedMap(SortedMap sm)
public static SortedSet synchronizedSortedSet(SortedSet ss)
```

- 这些方法基本上返回具有同步集合方法版本的新类。比如，为了创建多线程安全且由 ArrayList 支持的 List，可以使用如下代码：

```
List list = Collection.synchronizedList(new ArrayList());
```

- 注意，ArrayList 实例马上封装起来，不存在对未同步化 ArrayList 的直接引用（即直接封装匿名实例）。这是一种最安全的途径。如果另一个线程要直接引用 ArrayList 实例，它可以执行非同步修改。

- 下面给出一段多线程中安全遍历集合元素的示例。我们使用 Iterator 逐个扫描 List 中的元素，在多线程环境中，当遍历当前集合中的元素时，一般希望阻止其他线程添加或删除元素。安全遍历的实现方法如下：

```
import java.util.*;

public class SafeCollectionIteration extends Object {
    public static void main(String[] args) {
        //为了安全起见，仅使用同步列表的一个引用，这样可以确保控制了所有访问
        //集合必须同步化，这里是一个 List
        List wordList = Collections.synchronizedList(new ArrayList());

        //wordList 中的 add 方法是同步方法，会获取 wordList 实例的对象锁
        wordList.add("Iterators");
        wordList.add("require");
        wordList.add("special");
        wordList.add("handling");

        //获取 wordList 实例的对象锁，
        //迭代时，阻塞其他线程调用 add 或 remove 等方法修改元素
        synchronized ( wordList ) {
            Iterator iter = wordList.iterator();
            while ( iter.hasNext() ) {
                String s = (String) iter.next();
                System.out.println("found string: " + s + ", length=" + s.length());
            }
        }
    }
}
```

- 这里需要注意的是：在 Java 语言中，大部分的线程安全类都是相对线程安全的，它能保证对这个对象单独的操作时线程安全的，我们在调用的时候不需要额外的保障措施，但是对于一些特定的连续调用，就可能需要在调用端使用额外的同步手段来保证调用的正确性。例如 Vector、HashTable、Collections 的 synchronizedXxxx() 方法包装的集合等。

3.8 实现内存可见的两种方法比较：加锁和 volatile 变量

- volatile 变量是一种稍弱的同步机制在访问 volatile 变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此 volatile 变量是一种比 synchronized 关键字更轻量级的同步机制。
- 从内存可见性的角度看，写入 volatile 变量相当于退出同步代码块，而读取 volatile 变量相当于进入同步代码块。
- 在代码中如果过度依赖 volatile 变量来控制状态的可见性，通常会比使用锁的代码更脆弱，也更难以理解。仅当 volatile 变量能简化代码的实现以及对同步策略的验证时，才应该使用它。一般来说，用同步机制会更安全些。
- 加锁机制（即同步机制）既可以确保可见性又可以确保原子性，而 volatile 变量只能确保可见性，原因是声明为 volatile 的简单变量如果当前值与该变量以前的值相关，那么 volatile 关键字不起作用，也就是说如下的表达式都不是原子操作："count++"、"count = count+1"。

当且仅当满足以下所有条件时，才应该使用 volatile 变量：

- 对变量的写入操作不依赖变量的当前值，或者你能确保只有单个线程更新变量的值。
- 该变量没有包含在具有其他变量的不变式中。

总结：在需要同步的时候，第一选择应该是 synchronized 关键字，这是最安全的方式，尝试其他任何方式都是有风险的。尤其在、jdK1.5 之后，对 synchronized 同步机制做了很多优化，如：自适应的自旋锁、锁粗化、锁消除、轻量级锁等，使得它的性能明显有了很大的提升。

3.9 死锁

- 当线程需要同时持有多个锁时，有可能产生死锁。考虑如下情形：
- 线程 A 当前持有互斥所锁 lock1，线程 B 当前持有互斥锁 lock2。接下来，当线程 A 仍然持有 lock1 时，它试图获取 lock2，因为线程 B 正持有 lock2，因此线程 A 会阻塞等待线程 B 对 lock2 的释放。如果此时线程 B 在持有 lock2 的时候，也在试图获取 lock1，因为线程 A 正持有 lock1，因此线程 B 会阻塞等待 A 对 lock1 的释放。二者都在等待对方所持有锁的释放，而二者却又都没释放自己所持有的锁，这时二者便会一直阻塞下去。这种情形称为死锁。
- 下面给出一个两个线程间产生死锁的示例，如下：

```
public class Deadlock {
^^Iprivate String objID;
^^Ipublic Deadlock(String id) {
^^I^^IobjID = id;
^^I}
^^Ipublic synchronized void checkOther(Deadlock other) {
^^I^^Iprint("entering checkOther()");
        try { Thread.sleep(2000); }
        catch ( InterruptedException x ) { }
        print("in checkOther() - about to " + "invoke 'other.action()'");
        //调用 other 对象的 action 方法，由于该方法是同步方法，因此会试图获取 other 对象的对象锁
        other.action();
        print("leaving checkOther()");
^^I}

^^Ipublic synchronized void action() {
        print("entering action()");
        try { Thread.sleep(500); }
        catch ( InterruptedException x ) { }
        print("leaving action()");
    }
^^Ipublic void print(String msg) {
^^I^^IthreadPrint("objID=" + objID + " - " + msg);
^^I}
```

```
^^Ipublic static void threadPrint(String msg) {
        String threadName = Thread.currentThread().getName();
        System.out.println(threadName + ": " + msg);
    }

^^Ipublic static void main(String[] args) {
^^I^^Ifinal Deadlock obj1 = new Deadlock("obj1");
        final Deadlock obj2 = new Deadlock("obj2");

        Runnable runA = new Runnable() {
                public void run() {
                    obj1.checkOther(obj2);
                }
            };

        Thread threadA = new Thread(runA, "threadA");
        threadA.start();

        try { Thread.sleep(200); }
        catch ( InterruptedException x ) { }

        Runnable runB = new Runnable() {
                public void run() {
                    obj2.checkOther(obj1);
                }
            };

        Thread threadB = new Thread(runB, "threadB");
        threadB.start();

        try { Thread.sleep(5000); }
        catch ( InterruptedException x ) { }

        threadPrint("finished sleeping");

        threadPrint("about to interrupt() threadA");

        threadA.interrupt();

        try { Thread.sleep(1000); }
        catch ( InterruptedException x ) { }

        threadPrint("about to interrupt() threadB");
        threadB.interrupt();

        try { Thread.sleep(1000); }
        catch ( InterruptedException x ) { }

        threadPrint("did that break the deadlock?");
^^I}
}
```

- 运行结果：

```
threadA: objID=obj1 - entering checkOther()
threadB: objID=obj2 - entering checkOther()
threadA: objID=obj1 - in checkOther() - about to invoke 'other.action()'
threadB: objID=obj2 - in checkOther() - about to invoke 'other.action()'
```

```
main: finished sleeping
main: about to interrupt() threadA
main: about to interrupt() threadB
main: did that break the deadlock?
```

- 从结果中可以看出，在执行到 other.action () 时，由于两个线程都在试图获取对方的锁，但对方都没有释放自己的锁，因而便产生了死锁，在主线程中试图中断两个线程，但都无果。
- 大部分代码并不容易产生死锁，死锁可能在代码中隐藏相当长的时间，等待不常见的条件地发生，但即使是很小的概率，一旦发生，便可能造成毁灭性的破坏。避免死锁是一件困难的事，遵循以下原则有助于规避死锁：
  - 只在必要的最短时间内持有锁，考虑使用同步语句块代替整个同步方法；
  - 尽量编写不在同一时刻需要持有多个锁的代码，如果不可避免，则确保线程持有第二个锁的时间尽量短暂；
  - 创建和使用一个大锁来代替若干小锁，并把这个锁用于互斥，而不是用作单个对象的对象级别锁；

3.10 可重入内置锁

- 每个 Java 对象都可以用做一个实现同步的锁，这些锁被称为内置锁或监视器锁。线程在进入同步代码块之前会自动获取锁，并且在退出同步代码块时会自动释放锁。获得内置锁的唯一途径就是进入由这个锁保护的同步代码块或方法。
- 当某个线程请求一个由其他线程持有的锁时，发出请求的线程就会阻塞。然而，由于内置锁是可重入的，因此如果摸个线程试图获得一个已经由它自己持有的锁，那么这个请求就会成功。“重入”意味着获取锁的操作的粒度是“线程”，而不是调用。重入的一种实现方法是，为每个锁关联一个获取计数值和一个所有者线程。当计数值为 0 时，这个锁就被认为是没有被任何线程所持有，当线程请求一个未被持有的锁时，JVM 将记下锁的持有者，并且将获取计数值置为 1，如果同一个线程再次获取这个锁，计数值将递增，而当线程退出同步代码块时，计数器会相应地递减。当计数值为 0 时，这个锁将被释放。
- 重入进一步提升了加锁行为的封装性，因此简化了面向对象并发代码的开发。分析如下程序：

```
public class Father   {
    public synchronized void doSomething(){
        ......
    }
}

public class Child extends Father   {
    public synchronized void doSomething(){
        ......
        super.doSomething();
    }
}
```

- 子类覆写了父类的同步方法，然后调用父类中的方法，此时如果没有可重入的锁，那么这段代码件产生死锁。
- 由于 Fither 和 Child 中的 doSomething 方法都是 synchronized 方法，因此每个 doSomething 方法在执行前都会获取 Child 对象实例上的锁。如果内置锁不是可重入的，那么在调用 super.doSomething 时将无法获得该 Child 对象上的互斥锁，因为这个锁已经被持有，从而线程会永远阻塞下去，一直在等待一个永远也无法获取的锁。重入则避免了这种死锁情况的发生。
- 同一个线程在调用本类中其他 synchronized 方法/块或父类中的 synchronized 方法/块时，都不会阻碍该线程地执行，因为互斥锁时可重入的。

3.11 使用 wait/notify/notifyAll 实现线程间通信

3.12 NIO

- Java NIO(New IO) 是一个可以替代标准 Java IO API 的 IO API(从 Java1.4 开始)，Java NIO 提供了与标准 IO 不同的 IO 工作方式。

3.12.1 Java NIO: Channels and Buffers（通道和缓冲区）

- 标准的俄 IO 基于字节流和字符流进行操作的，而 NIO 是基于通道 (Channel) 和缓冲区 (Buffer) 进行操作，数据总是从通道读取到缓冲区中，或者从缓冲区写入通道也类似。

3.12.2 Java NIO: Non-blocking IO（非阻塞 IO）

- Java NIO 可以让你非阻塞的使用 IO，例如：当线程从通道读取数据到缓冲区时，线程还是进行其他事情。当数据被写入到缓冲区时，线程可以继续处理它。从缓冲区写入通道也类似。

3.12.3 Java NIO: Selectors(选择器)

- Java NIO 引入了选择器的概念，选择器用于监听多个通道的事件 (比如：连接打开，数据到达)。因此，单个的线程可以监听多个数据通道。
- NIO 由以下核心部分组成：
  - Channels
  - Buffers
  - Selectors

1. Channel 和 Buffer
   - 基本上，所有的 IO 和 NIO 都从一个 Channel 开始。Channel 有点像流。数据可以从 Channel 读到 Buffer 中，也可以从 Buffer 写到 Channel 中
   - Channel 的实现
     - FileChannel
     - DatagramChannel
     - SocketChannel
     - ServerSocketChannel
   - 这些通道涵盖了 UDP 和 TCP 网络 IO，以及文件 IO。
   - 以下是 Java NIO 里关键的 Buffer 实现
     - ByteBuffer
     - CharBuffer
     - DoubleBuffer
     - FloatBuffer
     - IntBuffer
     - LongBuffer
     - ShortBuffer
   - 这些 Buffer 覆盖了你能通过 IO 发送的基本数据类型：byte,short,int,long,float,double 和 char
   - Java NIO 还有个 MappedByteBuffer，用于表示内存映射文件。
2. Selextor
   - Selector 允许单线程处理多个 Channel。如果你的应用打开了多个连接 (通道)，但每一个连接的流量都很低，使用 Selector 就会很方便。
   - 例如，在一个聊天服务器中
   - 这是在一个单线程中使用一个 Selector 处理 3 个 Channel 的图示：
   - 要使用 Selector，得向 Selector 注册 Channel，然后调用它的 select() 方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新连接进来，数据接送等。

3.12.4 Channel

- Java NIO 的通道类似流，但又有些不同：
  - 既可以从通道中读取数据，又可以写数据到通道。但流的读写通常是单向的。
  - 通道可以异步的读写。
  - 通道的数据总是要先读到一个 Buffer，或者总要从一个 Buffer 中写入。
- 正如上面所说，从通道读取数据到缓冲区，从缓冲区写入数据到通道。

1. Channel 的实现
   - FileChannel 从文件中读取数据
   - DataChannel 能通过 UDP 读写网络中的数据
   - SocketChannel 能通过 TCP 读写网络中的数据
   - ServerSocketChannel 可以监听新进来的 TCP 连接，像 Web 服务器那样。对每一个新进来的连接都会创建一个 SocketChannel
2. 基本的 Channel 示例
   - 下面是一个使用 FileChannel 读取数据到 Buffer 中的示例

```
RandomAccessFile aFile = new RandomAccessFile("data/nio-data.txt", "rw");
FileChannel inChannel = aFile.getChannel();
ByteBuffer buf = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buf);

while (bytesRead != -1) {
    System.out.println("Read " + bytesRead);
    buf.flip();
    while(buf.hasRemaining()){
        System.out.print((char) buf.get());
    }
    buf.clear();
    bytesRead = inChannel.read(buf);
}
aFile.close();
```

   - 注意 buf.flip() 的调用，首先读取数据到 Buffer，然后反转 Buffer, 接着再从 Buffer 中读取数据。

3.12.5 Buffer

- Java NIO 中的 Buffer 用于和 NIO 通道进行交互。如你所知，数据是从通道读入缓冲区，从缓冲区写入到通道中的。
- 缓冲区本质是一块可以写入数据，然后可以从中读取数据的内存。这块内存被包装成 NIO Buffer 对象，并提供了一组方法，用来方便的访问这块内存。

1. Buffer 的基本用法
   - 使用 Buffer 读写数据一般遵循以下四个步骤：
     - 写入数据到 Buffer
     - 调用 flip() 方法
     - 从 Buffer 中读取数据
     - 调用 clear() 方法或者 compact() 方法

4 JavaSE(Java 基础)

4.1 Java 基础知识

4.1.1 基础

- 八种基本数据类型的大小，以及他们的封装类。
  - 八种基本数据类型，int ,double ,long ,float, short,byte,character,boolean
  - 对应的封装类型是：Integer ,Double ,Long ,Float, Short,Byte,Character,Boolean
- Switch 能否用 string 做参数?
- 在 Java 5 以前，switch(expr) 中，expr 只能是 byte、short、char、int。从 Java 5 开始，Java 中引入了枚举类型，expr 也可以是 enum 类型，从 Java 7 开始，expr 还可以是字符串（String），但是长整型（long）在目前所有的版本中都是不可以的。
- equals 与 == 的区别。
  - http://www.importnew.com/6804.html
  - == 与 equals 的主要区别是：== 常用于比较原生类型，而 equals() 方法用于检查对象的相等性。另一个不同的点是：如果 == 和 equals() 用于比较对象，当两个引用地址相同，== 返回 true。而 equals() 可以返回 true 或者 false 主要取决于重写实现。最常见的一个例子，字符串的比较，不同情况 == 和 equals() 返回不同的结果。equals() 方法最重要的一点是，能够根据业务要求去重写，按照自定义规则去判断两个对象是否相等。重写 equals() 方法的时候，要注意一下 hashCode 是否会因为对象的属性改变而改变，否则在使用散列集合储存该对象的时候会碰到坑 !! 理解 equals() 方法的存在是很重要的。
  - 使用 == 比较有两种情况：
    * 比较基础数据类型 (Java 中基础数据类型包括八中：short,int,long,float,double,char,byte,boolen)：这种情况下，== 比较的是他们的值是否相等。
    * 引用间的比较：在这种情况下，== 比较的是他们在内存中的地址，也就是说，除非引用指向的是同一个 new 出来的对象，此时他们使用 ‘==‘去比较得到 true，否则，得到 false。
  - 使用 equals 进行比较：
    * equals 追根溯源，是 Object 类中的一个方法，在该类中，equals 的实现也仅仅只是比较两个对象的内存地址是否相等，但在一些子类中，如：String、Integer 等，该方法将被重写。
  - 以 String 类为例子说明 eqauls 与 == 的区别：
    * 在开始这个例子之前，同学们需要知道 JVM 处理 String 的一些特性。Java 的虚拟机在内存中开辟出一块单独的区域，用来存储字符串对象，这块内存区域被称为字符串缓冲池。当使用 String a = ”abc” 这样的语句进行定义一个引用的时候，首先会在字符串缓冲池中查找是否已经相同的对象，如果存在，那么就直接将这个对象的引用返回给 a，如果不存在，则需要新建一个值为”abc” 的对象，再将新的引用返回 a。String a = new String(”abc”); 这样的语句明确告诉 JVM 想要产生一个新的 String 对象，并且值为”abc”，于是就在堆内存中的某一个小角落开辟了一个新的 String 对象。

```
- `==`在比较引用的情况下，会去比较两个引用的内存地址是否相等。
```
```
    String str1 = "abc";
    String str2 = "abc";

    System.out.println(str1 == str2);
    System.out.println(str1.equals(str2));

    String str2 = new String("abc");
    System.out.println(str1 == str2);
    System.out.println(str1.equals(str2));

```
```
    以上代码将会输出
    true
    true
```

```
    false
    true
    **第一个 true: **因为在 str2 赋值之前, str1 的赋值操作就已经在内存中创建了一个值为"abc"的
    **第二个 true: **因为`String`已经重写了`equals`方法: 为了方便大家阅读我贴出来, 并且在注释
    ```
    public boolean equals(Object anObject) {
    //如果比较的对象与自身内存地址相等的话
    //就说明他两指向的是同一个对象
    //所以此时 equals 的返回值跟 == 的结果是一样的。
    if (this == anObject) {
        return true;
    }
    //当比较的对象与自身的内存地址不相等, 并且
    //比较的对象是 String 类型的时候
    //将会执行这个分支
    if (anObject instanceof String) {
        String anotherString = (String)anObject;
        int n = value.length;
        if (n == anotherString.value.length) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = 0;
            //在这里循环遍历两个 String 中的 char
            while (n-- != 0) {
                //只要有一个不相等, 那么就会返回 false
                if (v1[i] != v2[i])
                    return false;
                i++;
            }
            return true;
        }
    }
    return false;
}
    ```
    进行以上分析之后, 就不难理解第一段代码中的实例程序输出了。
```

- Object 有哪些公用方法?

http://www.cnblogs.com/yumo/p/4908315.html

- 1．clone 方法
  - 保护方法，实现对象的浅复制，只有实现了 Cloneable 接口才可以调用该方法，否则抛出 CloneNotSupportedException 异常。
  - 主要是 JAVA 里除了 8 种基本类型传参数是值传递，其他的类对象传参数都是引用传递，我们有时候不希望在方法里讲参数改变，这是就需要在类中复写 clone 方法。
- 2．getClass 方法
  - final 方法，获得运行时类型。
- 3．toString 方法
  - 该方法用得比较多，一般子类都有覆盖。
- 4．finalize 方法
  - 该方法用于释放资源。因为无法确定该方法什么时候被调用，很少使用。
- 5．equals 方法

- 该方法是非常重要的一个方法。一般 equals 和 == 是不一样的，但是在 Object 中两者是一样的。子类一般都要重写这个方法。

- 6．hashCode 方法
  - 该方法用于哈希查找，可以减少在查找中使用 equals 的次数，重写了 equals 方法一般都要重写 hashCode 方法。这个方法在一些具有哈希功能的 Collection 中用到。
  - 一般必须满足 obj1.equals(obj2)==true。可以推出 obj1.hashCode()==obj2.hashCode()，但是 hashCode 相等不一定就满足 equals。不过为了提高效率，应该尽量使上面两个条件接近等价。
  - 如果不重写 hashCode(), 在 HashSet 中添加两个 equals 的对象，会将两个对象都加入进去。

- 7．wait 方法
  - wait 方法就是使当前线程等待该对象的锁，当前线程必须是该对象的拥有者，也就是具有该对象的锁。wait() 方法一直等待，直到获得锁或者被中断。wait(long timeout) 设定一个超时间隔，如果在规定时间内没有获得锁就返回。
  - 调用该方法后当前线程进入睡眠状态，直到以下事件发生。
    * （1）其他线程调用了该对象的 notify 方法。
    * （2）其他线程调用了该对象的 notifyAll 方法。
    * （3）其他线程调用了 interrupt 中断该线程。
    * （4）时间间隔到了。
  - 此时该线程就可以被调度了，如果是被中断的话就抛出一个 InterruptedException 异常。

- 8．notify 方法
  - 该方法唤醒在该对象上等待的某个线程。

- 9．notifyAll 方法
  - 该方法唤醒在该对象上等待的所有线程。

- Java 的四种引用，强弱软虚，用到的场景。
  - JDK1.2 之前只有强引用, 其他几种引用都是在 JDK1.2 之后引入的.
  - 强引用（Strong Reference）最常用的引用类型，如 Object obj = new Object(); 。只要强引用存在则 GC 时则必定不被回收。
  - 软引用（Soft Reference）用于描述还有用但非必须的对象，当堆将发生 OOM（Out Of Memory）时则会回收软引用所指向的内存空间，若回收后依然空间不足才会抛出 OOM。一般用于实现内存敏感的高速缓存。当真正对象被标记 finalizable 以及的 finalize() 方法调用之后并且内存已经清理, 那么如果 SoftReference object 还存在就被加入到它的 ReferenceQueue. 只有前面几步完成后,Soft Reference 和 Weak Reference 的 get 方法才会返回 null
  - 弱引用（Weak Reference）发生 GC 时必定回收弱引用指向的内存空间。和软引用加入队列的时机相同
  - 虚引用（Phantom Reference）又称为幽灵引用或幻影引用，虚引用既不会影响对象的生命周期，也无法通过虚引用来获取对象实例，仅用于在发生 GC 时接收一个系统通知。当一个对象的 finalize 方法已经被调用了之后，这个对象的幽灵引用会被加入到队列中。通过检查该队列里面的内容就知道一个对象是不是已经准备要被回收了. 虚引用和软引用和弱引用都不同, 它会在内存没有清理的时候被加入引用队列. 虚引用的建立必须要传入引用队列, 其他可以没有

- Hashcode 的作用。

http://c610367182.iteye.com/blog/1930676

- 以 Java.lang.Object 来理解,JVM 每 new 一个 Object, 它都会将这个 Object 丢到一个 Hash 哈希表中去, 这样的话, 下次做 Object 的比较或者取这个对象的时候, 它会根据对象的 hashcode 再从 Hash 表中取这个对象。这样做的目的是提高取对象的效率。具体过程是这样:
  - new Object(),JVM 根据这个对象的 Hashcode 值, 放入到对应的 Hash 表对应的 Key 上, 如果不同的对象确产生了相同的 hash 值, 也就是发生了 Hash key 相同导致冲突的情况, 那么就在这个 Hash key 的地方产生一个链表, 将所有产生相同 hashcode 的对象放到这个单链表上去, 串在一起。

  - 比较两个对象的时候, 首先根据他们的 hashcode 去 hash 表中找他的对象, 当两个对象的 hashcode 相同, 那么就是说他们这两个对象放在 Hash 表中的同一个 key 上, 那么他们一定在这个 key 上的链表上。那么此时就只能根据 Object 的 equal 方法来比较这个对象是否 equal。当两个对象的 hashcode 不同的话，肯定他们不能 equal.

- String、StringBuffer 与 StringBuilder 的区别。
  - Java 平台提供了两种类型的字符串：String 和 StringBuffer / StringBuilder，它们可以储存和操作字符串。其中 String 是只读字符串，也就意味着 String 引用的字符串内容是不能被改变的。而 StringBuffer 和 StringBulder 类表示的字符串对象可以直接进行修改。StringBuilder 是 JDK1.5 引入的，它和 StringBuffer 的方法完全相同，区别在于它是单线程环境下使用的，因为它的所有方面都没有被 synchronized 修饰，因此它的效率也比 StringBuffer 略高。

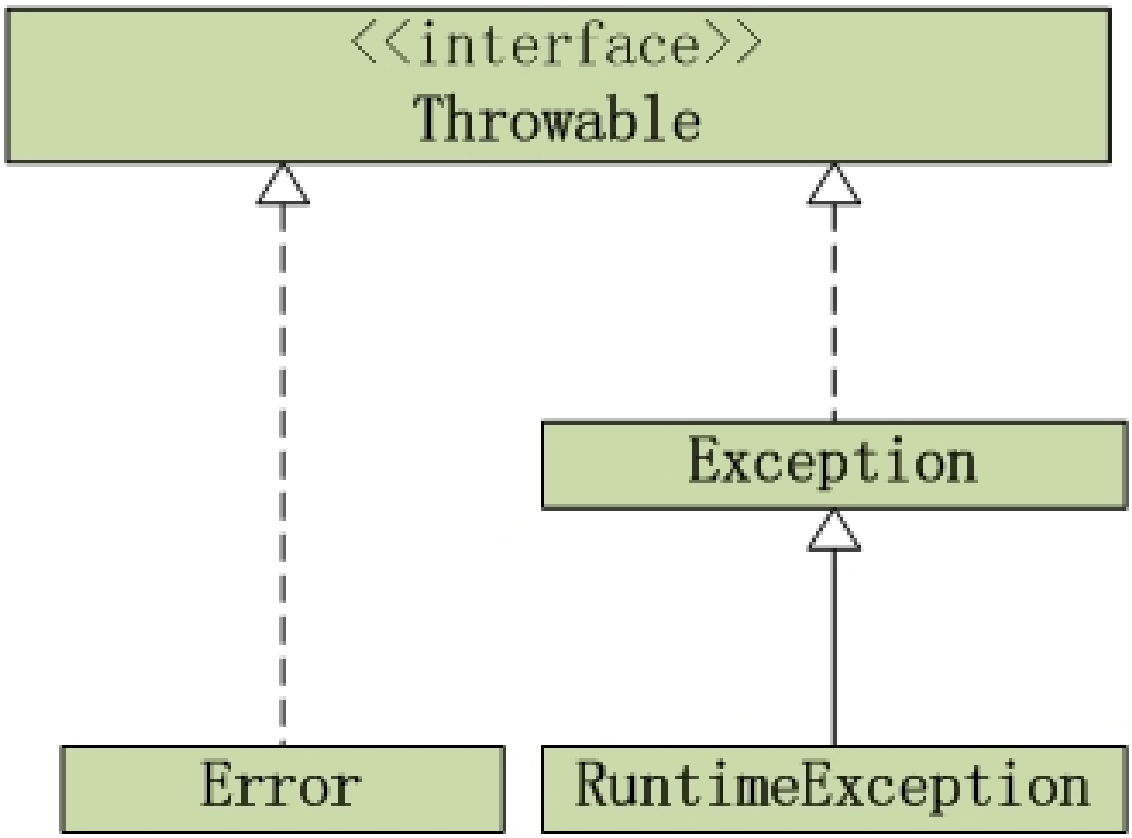
- try catch finally，try 里有 return，finally 还执行么?
  - 会执行，在方法返回调用者前执行。Java 允许在 finally 中改变返回值的做法是不好的，因为如果存在 finally 代码块，try 中的 return 语句不会立马返回调用者，而是纪录下返回值待 finally 代码块执行完毕之后再向调用者返回其值，然后如果在 finally 中修改了返回值，这会对程序造成很大的困扰，C# 中就从语法规定不能做这样的事。

- Excption 与 Error 区别
  - Error 表示系统级的错误和程序不必处理的异常，是恢复不是不可能但很困难的情况下的一种严重问题；比如内存溢出，不可能指望程序能处理这样的状况；Exception 表示需要捕捉或者需要程序进行处理的异常，是一种设计或实现问题；也就是说，它表示如果程序运行正常，从不会发生的情况。

- Excption 与 Error 包结构。OOM 你遇到过哪些情况，SOF 你遇到过哪些情况。

http://www.cnblogs.com/yumo/p/4909617.html

- Java 异常架构图



  - Throwable Throwable 是 Java 语言中所有错误或异常的超类。Throwable 包含两个子类: Error 和 Exception 。它们通常用于指示发生了异常情况。Throwable 包含了其线程创建时线程执行堆栈的快照，它提供了 printStackTrace() 等接口用于获取堆栈跟踪数据等信息。

- Exception Exception 及其子类是 Throwable 的一种形式，它指出了合理的应用程序想要捕获的条件。
- RuntimeException RuntimeException 是那些可能在 Java 虚拟机正常运行期间抛出的异常的超类。编译器不会检查 RuntimeException 异常。例如，除数为零时，抛出 ArithmeticException 异常。RuntimeException 是 ArithmeticException 的超类。当代码发生除数为零的情况时，倘若既” 没有通过 throws 声明抛出 ArithmeticException 异常”，也” 没有通过 try…catch…处理该异常”，也能通过编译。这就是我们所说的” 编译器不会检查 RuntimeException 异常”！如果代码会产生 RuntimeException 异常，则需要通过修改代码进行避免。例如，若会发生除数为零的情况，则需要通过代码避免该情况的发生！
- Error 和 Exception 一样，Error 也是 Throwable 的子类。它用于指示合理的应用程序不应该试图捕获的严重问题，大多数这样的错误都是异常条件。和 RuntimeException 一样，编译器也不会检查 Error。

Java 将可抛出 (Throwable) 的结构分为三种类型：被检查的异常 (Checked Exception)，运行时异常 (RuntimeException) 和错误 (Error)。

- (01) 运行时异常定义: RuntimeException 及其子类都被称为运行时异常。特点: Java 编译器不会检查它。也就是说，当程序中可能出现这类异常时，倘若既” 没有通过 throws 声明抛出它”，也” 没有用 try-catch 语句捕获它”，还是会编译通过。例如，除数为零时产生的 ArithmeticException 异常，数组越界时产生的 IndexOutOfBoundsException 异常，fail-fail 机制产生的 ConcurrentModificationException 异常等，都属于运行时异常。虽然 Java 编译器不会检查运行时异常，但是我们也可以通过 throws 进行声明抛出，也可以通过 try-catch 对它进行捕获处理。如果产生运行时异常，则需要通过修改代码来进行避免。例如，若会发生除数为零的情况，则需要通过代码避免该情况的发生！
- (02) 被检查的异常定义: Exception 类本身，以及 Exception 的子类中除了” 运行时异常” 之外的其它子类都属于被检查异常。特点: Java 编译器会检查它。此类异常，要么通过 throws 进行声明抛出，要么通过 try-catch 进行捕获处理，否则不能通过编译。例如，CloneNotSupportedException 就属于被检查异常。当通过 clone() 接口去克隆一个对象，而该对象对应的类没有实现 Cloneable 接口，就会抛出 CloneNotSupportedException 异常。被检查异常通常都是可以恢复的。
- (03) 错误定义: Error 类及其子类。特点: 和运行时异常一样，编译器也不会对错误进行检查。当资源不足、约束失败、或是其它程序无法继续运行的条件发生时，就产生错误。程序本身无法修复这些错误的。例如，VirtualMachineError 就属于错误。按照 Java 惯例，我们是不应该是实现任何新的 Error 子类的！

对于上面的 3 种结构，我们在抛出异常或错误时，到底该哪一种?《Effective Java》中给出的建议是：对于可以恢复的条件使用被检查异常，对于程序错误使用运行时异常。

- OOM:
  - OutOfMemoryError 异常
    - 除了程序计数器外，虚拟机内存的其他几个运行时区域都有发生 OutOfMemoryError(OOM) 异常的可能，
    - Java Heap 溢出
    - 一般的异常信息：java.lang.OutOfMemoryError:Java heap spacess
    - java 堆用于存储对象实例，我们只要不断的创建对象，并且保证 GC Roots 到对象之间有可达路径来避免垃圾回收机制清除这些对象，就会在对象数量达到最大堆容量限制后产生内存溢出异常。
    - 出现这种异常，一般手段是先通过内存映像分析工具 (如 Eclipse Memory Analyzer) 对 dump 出来的堆转存快照进行分析，重点是确认内存中的对象是否是必要的，先分清是因为内存泄漏 (Memory Leak) 还是内存溢出 (Memory Overflow)。
    - 如果是内存泄漏，可进一步通过工具查看泄漏对象到 GC Roots 的引用链。于是就能找到泄漏对象时通过怎样的路径与 GC Roots 相关联并导致垃圾收集器无法自动回收。
    - 如果不存在泄漏，那就应该检查虚拟机的参数 (-Xmx 与-Xms) 的设置是否适当。
  - 虚拟机栈和本地方法栈溢出
    - 如果线程请求的栈深度大于虚拟机所允许的最大深度，将抛出 StackOverflowError 异常。
    - 如果虚拟机在扩展栈时无法申请到足够的内存空间，则抛出 OutOfMemoryError 异常
    - 这里需要注意当栈的大小越大可分配的线程数就越少。
  - 运行时常量池溢出
    - 异常信息：java.lang.OutOfMemoryError:PermGen space

    * 如果要向运行时常量池中添加内容，最简单的做法就是使用 String.intern() 这个 Native 方法。该方法的作用是：如果池中已经包含一个等于此 String 的字符串，则返回代表池中这个字符串的 String 对象；否则，将此 String 对象包含的字符串添加到常量池中，并且返回此 String 对象的引用。由于常量池分配在方法区内，我们可以通过-XX:PermSize 和-XX:MaxPermSize 限制方法区的大小，从而间接限制其中常量池的容量。
  - 方法区溢出
    * 方法区用于存放 Class 的相关信息，如类名、访问修饰符、常量池、字段描述、方法描述等。
    * 异常信息：java.lang.OutOfMemoryError:PermGen space
    * 方法区溢出也是一种常见的内存溢出异常，一个类如果要被垃圾收集器回收，判定条件是很苛刻的。在经常动态生成大量 Class 的应用中，要特别注意这点。

- Java 面向对象的三个特征与含义。
  - 继承：继承是从已有类得到继承信息创建新类的过程。提供继承信息的类被称为父类（超类、基类）；得到继承信息的类被称为子类（派生类）。继承让变化中的软件系统有了一定的延续性，同时继承也是封装程序中可变因素的重要手段。
  - 封装：通常认为封装是把数据和操作数据的方法绑定起来，对数据的访问只能通过已定义的接口。面向对象的本质就是将现实世界描绘成一系列完全自治、封闭的对象。我们在类中编写的方法就是对实现细节的一种封装；我们编写一个类就是对数据和数据操作的封装。可以说，封装就是隐藏一切可隐藏的东西，只向外界提供最简单的编程接口（可以想想普通洗衣机和全自动洗衣机的差别，明显全自动洗衣机封装更好因此操作起来更简单；我们现在使用的智能手机也是封装得足够好的，因为几个按键就搞定了所有的事情）。
  - 多态：多态性是指允许不同子类型的对象对同一消息作出不同的响应。简单的说就是用同样的对象引用调用同样的方法但是做了不同的事情。多态性分为编译时的多态性和运行时的多态性。如果将对象的方法视为对象向外界提供的服务，那么运行时的多态性可以解释为：当 A 系统访问 B 系统提供的服务时，B 系统有多种提供服务的方式，但一切对 A 系统来说都是透明的（就像电动剃须刀是 A 系统，它的供电系统是 B 系统，B 系统可以使用电池供电或者用交流电，甚至还有可能是太阳能，A 系统只会通过 B 类对象调用供电的方法，但并不知道供电系统的底层实现是什么，究竟通过何种方式获得了动力）。方法重载（overload）实现的是编译时的多态性（也称为前绑定），而方法重写（override）实现的是运行时的多态性（也称为后绑定）。运行时的多态是面向对象最精髓的东西，要实现多态需要做两件事：1. 方法重写（子类继承父类并重写父类中已有的或抽象的方法）；2. 对象造型（用父类型引用引用子类型对象，这样同样的引用调用同样的方法就会根据子类对象的不同而表现出不同的行为）。

- Override 和 Overload 的含义与区别。
  - Overload：顾名思义，就是 Over(重新)——load（加载），所以中文名称是重载。它可以表现类的多态性，可以是函数里面可以有相同的函数名但是参数名、类型不能相同；或者说可以改变参数、类型但是函数名字依然不变。
  - Override：就是 ride(重写) 的意思，在子类继承父类的时候子类中可以定义某方法与其父类有相同的名称和参数，当子类在调用这一函数时自动调用子类的方法，而父类相当于被覆盖（重写）了。
  - 方法的重写 Overriding 和重载 Overloading 是 Java 多态性的不同表现。重写 Overriding 是父类与子类之间多态性的一种表现，重载 Overloading 是一个类中多态性的一种表现。如果在子类中定义某方法与其父类有相同的名称和参数，我们说该方法被重写 (Overriding)。子类的对象使用这个方法时，将调用子类中的定义，对它而言，父类中的定义如同被“屏蔽”了。如果在一个类中定义了多个同名的方法，它们或有不同的参数个数或有不同的参数类型，则称为方法的重载 (Overloading)。Overloaded 的方法是可以改变返回值的类型。

- Interface 与 abstract 类的区别。
  - 抽象类和接口都不能够实例化，但可以定义抽象类和接口类型的引用。一个类如果继承了某个抽象类或者实现了某个接口都需要对其中的抽象方法全部进行实现，否则该类仍然需要被声明为抽象类。接口比抽象类更加抽象，因为抽象类中可以定义构造器，可以有抽象方法和具体方法，而接口中不能定义构造器而且其中的方法全部都是抽象方法。抽象类中的成员可以是 private、默认、protected、public 的，而接口中的成员全都是 public 的。抽象类中可以定义成员变量，而接口中定义的成员变量实际上都是常量。有抽象方法的类必须被声明为抽象类，而抽象类未必要有抽象方法。

- Static class 与 non static class 的区别。

  - 内部静态类不需要有指向外部类的引用。但非静态内部类需要持有对外部类的引用。非静态内部类能够访问外部类的静态和非静态成员。静态类不能访问外部类的非静态成员。他只能访问外部类的静态成员。一个非静态内部类不能脱离外部类实体被创建，一个非静态内部类可以访问外部类的数据和方法，因为他就在外部类里面。

- java 多态的实现原理。http://blog.csdn.net/zzzhangzhun/article/details/51095075

  - 当 JVM 执行 Java 字节码时，类型信息会存储在方法区中，为了优化对象的调用方法的速度，方法区的类型信息会增加一个指针，该指针指向一个记录该类方法的方法表，方法表中的每一个项都是对应方法的指针。
  - 方法区：方法区和 JAVA 堆一样，是各个线程共享的内存区域，用于存储已被虚拟机加载的类信息、常量、静态变量、即时编译器编译后的代码等数据。运行时常量池：它是方法区的一部分，Class 文件中除了有类的版本、方法、字段等描述信息外，还有一项信息是常量池，用于存放编译器生成的各种符号引用，这部分信息在类加载时进入方法区的运行时常量池中。方法区的内存回收目标是针对常量池的回收及对类型的卸载。
  - 方法表的构造
    * 由于 java 的单继承机制，一个类只能继承一个父类，而所有的类又都继承 Object 类，方法表中最先存放的是 Object 的方法，接下来是父类的方法，最后是该类本身的方法。如果子类改写了父类的方法，那么子类和父类的那些同名的方法共享一个方法表项。
    * 由于这样的特性，使得方法表的偏移量总是固定的，例如，对于任何类来说，其方法表的 equals 方法的偏移量总是一个定值，所有继承父类的子类的方法表中，其父类所定义的方法的偏移量也总是一个定值。
  - 实例
    * 假设 Class A 是 Class B 的子类，并且 A 改写了 B 的方法的 method()，那么 B 来说，method 方法的指针指向 B 的 method 方法入口；对于 A 来说，A 的方法表的 method 项指向自身的 method 而非父类的。
    * 流程：调用方法时，虚拟机通过对象引用得到方法区中类型信息的方法表的指针入口，查询类的方法表，根据实例方法的符号引用解析出该方法在方法表的偏移量，子类对象声明为父类类型时，形式上调用的是父类的方法，此时虚拟机会从实际的方法表中找到方法地址，从而定位到实际类的方法。注：所有引用为父类，但方法区的类型信息中存放的是子类的信息，所以调用的是子类的方法表。

- foreach 与正常 for 循环效率对比。

http://904510742.iteye.com/blog/2118331

- 直接 for 循环效率最高，其次是迭代器和 ForEach 操作。作为语法糖，其实 ForEach 编译成字节码之后，使用的是迭代器实现的，反编译后，testForEach 方法如下：

```
public static void testForEach(List list) {
    for (Iterator iterator = list.iterator(); iterator.hasNext();) {
        Object t = iterator.next();
        Object obj = t;
    }
}
```

可以看到，只比迭代器遍历多了生成中间变量这一步，因为性能也略微下降了一些。

- 反射机制

JAVA 反射机制是在运行状态中, 对于任意一个类, 都能够知道这个类的所有属性和方法; 对于任意一个对象, 都能够调用它的任意一个方法和属性; 这种动态获取的信息以及动态调用对象的方法的功能称为 java 语言的反射机制.

- 主要作用有三：

  - 运行时取得类的方法和字段的相关信息。
  - 创建某个类的新实例 (.newInstance())
  - 取得字段引用直接获取和设置对象字段，无论访问修饰符是什么。

- 用处如下：
  - 观察或操作应用程序的运行时行为。
  - 调试或测试程序，因为可以直接访问方法、构造函数和成员字段。
  - 通过名字调用不知道的方法并使用该信息来创建对象和调用方法。
- String 类内部实现，能否改变 String 对象内容
  - String 源码分析

http://blog.csdn.net/zhangjg_blog/article/details/18319521

- try catch 块，try 里有 return，finally 也有 return，如何执行

http://qing0991.blog.51cto.com/1640542/1387200

- 泛型的优缺点
  - 优点：
    * 使用泛型类型可以最大限度地重用代码、保护类型的安全以及提高性能。
    * 泛型最常见的用途是创建集合类。
  - 缺点：
    * 在性能上不如数组快。
- 泛型常用特点，List<String> 能否转为 List<Object>
  - 能，但是利用类都继承自 Object，所以使用是每次调用里面的函数都要通过强制转换还原回原来的类，这样既不安全，运行速度也慢。
- 解析 XML 的几种方式的原理与特点：DOM、SAX、PULL。

http://www.cnblogs.com/HaroldTihan/p/4316397.html

- Java 与 C++ 对比。

http://developer.51cto.com/art/201106/270422.htm

- Java1.7 与 1.8 新特性。

http://blog.chinaunix.net/uid-29618857-id-4416835.html

- JNI 的使用。

http://landerlyoung.github.io/blog/2014/10/16/java-zhong-jnide-shi-yong/

4.1.2 集合

- ArrayList、LinkedList、Vector 的底层实现和区别
  - 从同步性来看，ArrayList 和 LinkedList 是不同步的，而 Vector 是的。所以线程安全的话，可以使用 ArrayList 或 LinkedList，可以节省为同步而耗费的开销。但在多线程下，有时候就不得不使用 Vector 了。当然，也可以通过一些办法包装 ArrayList、LinkedList，使我们也达到同步，但效率可能会有所降低。
  - 从内部实现机制来讲 ArrayList 和 Vector 都是使用 Object 的数组形式来存储的。当你向这两种类型中增加元素的时候，如果元素的数目超出了内部数组目前的长度它们都需要扩展内部数组的长度，Vector 缺省情况下自动增长原来一倍的数组长度，ArrayList 是原来的 50%，所以最后你获得的这个集合所占的空间总是比你实际需要的要大。如果你要在集合中保存大量的数据，那么使用 Vector 有一些优势，因为你可以通过设置集合的初始化大小来避免不必要的资源开销。
  - ArrayList 和 Vector 中，从指定的位置（用 index）检索一个对象，或在集合的末尾插入、删除一个对象的时间是一样的，可表示为 O(1)。但是，如果在集合的其他位置增加或者删除元素那么花费的时间会呈线性增长 O(n-i)，其中 n 代表集合中元素的个数，i 代表元素增加或移除元素的索引位置，因为在进行上述操作的时候集合中第 i 和第 i 个元素之后的所有元素都要执行 (n-i) 个对象的位移操作。LinkedList 底层是由双向循环链表实现的，LinkedList 在插入、删除集合中任何位置的元素所花费的时间都是一样的 O(1)，但它在索引一个元素的时候比较慢，为 O(i)，其中 i 是索引的位置，如果只是查找特定位置的元素或只在集合的末端增加、移除元素，那么使用 Vector 或 ArrayList 都可以。如果是对其它指定位置的插入、删除操作，最好选择 LinkedList。

- HashMap 和 HashTable 的底层实现和区别，两者和 ConcurrentHashMap 的区别。

http://blog.csdn.net/xuefeng0707/article/details/40834595

- HashTable 线程安全则是依靠方法简单粗暴的 sychronized 修饰，HashMap 则没有相关的线程安全问题考虑。。
- 在以前的版本貌似 ConcurrentHashMap 引入了一个“分段锁”的概念，具体可以理解为把一个大的 Map 拆分成 N 个小的 HashTable，根据 key.hashCode() 来决定把 key 放到哪个 HashTable 中。在 ConcurrentHashMap 中，就是把 Map 分成了 N 个 Segment，put 和 get 的时候，都是现根据 key.hashCode() 算出放到哪个 Segment 中。
- 通过把整个 Map 分为 N 个 Segment（类似 HashTable），可以提供相同的线程安全，但是效率提升 N 倍。
- HashMap 的 hashcode 的作用？什么时候需要重写？如何解决哈希冲突？查找的时候流程是如何？
  - 从源码分析 HashMap
- Arraylist 和 HashMap 如何扩容？负载因子有什么作用？如何保证读写进程安全？

http://m.blog.csdn.net/article/details?id=48956087 http://hovertree.com/h/bjaf/2jdr60li.htm

- ArrayList 本身不是线程安全的。所以正确的做法是去用 java.util.concurrent 里的 CopyOnWriteArrayList 或者某个同步的 Queue 类。
- HashMap 实现不是同步的。如果多个线程同时访问一个哈希映射，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。（结构上的修改是指添加或删除一个或多个映射关系的任何操作；仅改变与实例已经包含的键关联的值不是结构上的修改。）这一般通过对自然封装该映射的对象进行同步操作来完成。如果不存在这样的对象，则应该使用 Collections.synchronizedMap 方法来“包装”该映射。最好在创建时完成这一操作，以防止对映射进行意外的非同步访问.
- TreeMap、HashMap、LinkedHashMap 的底层实现区别。

http://blog.csdn.net/lolashe/article/details/20806319

- Collection 包结构，与 Collections 的区别。
  - Collection 是一个接口，它是 Set、List 等容器的父接口；Collections 是一个工具类，提供了一系列的静态方法来辅助容器操作，这些方法包括对容器的搜索、排序、线程安全化等等。
- Set、List 之间的区别是什么？

http://developer.51cto.com/art/201309/410205_all.htm

- Map、Set、List、Queue、Stack 的特点与用法。

http://www.cnblogs.com/yumo/p/4908718.html

- Collection 是对象集合，Collection 有两个子接口 List 和 Set
- List 可以通过下标 (1,2..) 来取得值，值可以重复
- 而 Set 只能通过游标来取值，并且值是不能重复的
  - ArrayList ，Vector ，LinkedList 是 List 的实现类
  - ArrayList 是线程不安全的，Vector 是线程安全的，这两个类底层都是由数组实现的
  - LinkedList 是线程不安全的，底层是由链表实现的
- Map 是键值对集合
  - HashTable 和 HashMap 是 Map 的实现类
  - HashTable 是线程安全的，不能存储 null 值
  - HashMap 不是线程安全的，可以存储 null 值
- Stack 类：继承自 Vector，实现一个后进先出的栈。提供了几个基本方法，push、pop、peak、empty、search 等。
- Queue 接口：提供了几个基本方法，offer、poll、peek 等。已知实现类有 LinkedList、PriorityQueue 等。
-

4.2 Java 中的内存泄漏

4.2.1 1.Java 内存回收机制

- 不论哪种语言的内存分配方式，都需要返回所分配内存的真实地址，也就是返回一个指针到内存块的首地址。Java 中对象是采用 new 或者反射的方法创建的，这些对象的创建都是在堆（Heap）中分配的，所有对象的回收都是由 Java 虚拟机通过垃圾回收机制完成的。GC 为了能够正确释放对象，会监控每个对象的运行状况，对他们的申请、引用、被引用、赋值等状况进行监控，Java 会使用有向图的方法进行管理内存，实时监控对象是否可以达到，如果不可到达，则就将其回收，这样也可以消除引用循环的问题。在 Java 语言中，判断一个内存空间是否符合垃圾收集标准有两个：一个是给对象赋予了空值 null，以下再没有调用过，另一个是给对象赋予了新值，这样重新分配了内存空间。

4.2.2 2.Java 内存泄漏引起的原因

- 内存泄漏是指无用对象（不再使用的对象）持续占有内存或无用对象的内存得不到及时释放，从而造成内存空间的浪费称为内存泄漏。内存泄露有时不严重且不易察觉，这样开发者就不知道存在内存泄露，但有时也会很严重，会提示你 Out of memory。

Java 内存泄漏的根本原因是什么呢？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漏，尽管短生命周期对象已经不再需要，但是因为长生命周期持有它的引用而导致不能被回收，这就是 Java 中内存泄漏的发生场景。具体主要有如下几大类：

1. 1、静态集合类引起内存泄漏：-像 HashMap、Vector 等的使用最容易出现内存泄露，这些静态变量的生命周期和应用程序一致，他们所引用的所有的对象 Object 也不能被释放，因为他们也将一直被 Vector 等引用着。

```
Static Vector v = new Vector(10);
for (int i = 1; i<100; i++) {
        Object o = new Object();
        v.add(o);
        o = null;
    }
```

   - 在这个例子中，循环申请 Object 对象，并将所申请的对象放入一个 Vector 中，如果仅仅释放引用本身（o=null），那么 Vector 仍然引用该对象，所以这个对象对 GC 来说是不可回收的。因此，如果对象加入到 Vector 后，还必须从 Vector 中删除，最简单的方法就是将 Vector 对象设置为 null。

2. 2、当集合里面的对象属性被修改后，再调用 remove() 方法时不起作用。

```
public static void main(String[] args) {
    Set<Person> set = new HashSet<Person>();
    Person p1 = new Person(" 唐僧","pwd1",25);
    Person p2 = new Person(" 孙悟空","pwd2",26);
    Person p3 = new Person(" 猪八戒","pwd3",27);
    set.add(p1);
    set.add(p2);
    set.add(p3);
    System.out.println(" 总共有:"+set.size()+" 个元素!"); //结果: 总共有:3 个元素!
    p3.setAge(2); //修改 p3 的年龄, 此时 p3 元素对应的 hashcode 值发生改变
    set.remove(p3); //此时 remove 不掉, 造成内存泄漏
    set.add(p3); //重新添加, 居然添加成功
    System.out.println(" 总共有:"+set.size()+" 个元素!"); //结果: 总共有:4 个元素!
    for (Person person : set) {
            System.out.println(person);
    }
}
```

3. 3、监听器

   - 在 java 编程中，我们都需要和监听器打交道，通常一个应用当中会用到很多监听器，我们会调用一个控件的诸如 addXXXListener() 等方法来增加监听器，但往往在释放对象的时候却没有记住去删除这些监听器，从而增加了内存泄漏的机会。

4. 4、各种连接

   - 比如数据库连接（dataSourse.getConnection()），网络连接 (socket) 和 io 连接，除非其显式的调用了其 close () 方法将其连接关闭，否则是不会自动被 GC 回收的。对于 Resultset 和 Statement 对象可以不进行显式回收，但 Connection 一定要显式回收，因为 Connection 在任何时候都无法自动回收，而 Connection 一旦回收，Resultset 和 Statement 对象就会立即为 NULL。但是如果使用连接池，情况就不一样了，除了要显式地关闭连接，还必须显式地关闭 Resultset Statement 对象（关闭其中一个，另外一个也会关闭），否则就会造成大量的 Statement 对象无法释放，从而引起内存泄漏。这种情况下一般都会在 try 里面去的连接，在 finally 里面释放连接。

5. 5、内部类和外部模块的引用

   - 内部类的引用是比较容易遗忘的一种，而且一旦没释放可能导致一系列的后继类对象没有释放。此外程序员还要小心外部模块不经意的引用，例如程序员 A 负责 A 模块，调用了 B 模块的一个方法如：public void registerMsg(Object b); 这种调用就要非常小心了，传入了一个对象，很可能模块 B 就保持了对该对象的引用，这时候就需要注意模块 B 是否提供相应的操作去除引用。

6. 6、单例模式

   - 不正确使用单例模式是引起内存泄漏的一个常见问题，单例对象在初始化后将在 JVM 的整个生命周期中存在（以静态变量的方式），如果单例对象持有外部的引用，那么这个对象将不能被 JVM 正常回收，导致内存泄漏，考虑下面的例子：

```
class A{
    public A(){
        B.getInstance().setA(this);
    }
    ....
}

//B 类采用单例模式
class B{
    private A a;
    private static B instance=new B();
    public B(){}
    public static B getInstance(){
        return instance;
    }
    public void setA(A a){
        this.a=a;
    }
    //getter...
}
```

   - 显然 B 采用 singleton 模式，它持有一个 A 对象的引用，而这个 A 类的对象将不能被回收。想象下如果 A 是个比较复杂的对象或者集合类型会发生什么情况

4.3 String 源码分析

从一段代码说起：

```
public void stringTest(){
    String a = "a"+"b"+1;
    String b = "ab1";
    System.out.println(a == b);
}
```

大家猜一猜结果如何？如果你的结论是 true。好吧，再来一段代码：

```
public void stringTest(){
    String a = new String("ab1");
    String b = "ab1";
    System.out.println(a == b);
}
```

结果如何呢？正确答案是 false。
让我们看看经过编译器编译后的代码如何

```
//第一段代码
public void stringTest() {
    String a = "ab1";
    String b = "ab1";
    System.out.println(a == b);
}
//第二段代码
public void stringTest() {
    String a1 = new String("ab1");
    String b = "ab1";
    System.out.println(a1 == b);
}
```

也就是说第一段代码经过了编译期优化，原因是编译器发现"a"+"b"+1 和"ab1" 的效果是一样的，都是不可变量组成。但是为什么他们的内存地址会相同呢？如果你对此还有兴趣，那就一起看看 String 类的一些重要源码吧。

4.3.1 一 String 类

- String 类被 final 所修饰，也就是说 String 对象是不可变量，并发程序最喜欢不可变量了。String 类实现了 Serializable, Comparable, CharSequence 接口。
- Comparable 接口有 compareTo(String s) 方法，CharSequence 接口有 length()，charAt(int index)，subSequence(int start,int end) 方法。

4.3.2 二 String 属性

- String 类中包含一个不可变的 char 数组用来存放字符串，一个 int 型的变量 hash 用来存放计算后的哈希值。

```
/** The value is used for character storage. */
private final char value[];
/** Cache the hash code for the string */
private int hash; // Default to 0
/** use serialVersionUID from JDK 1.0.2 for interoperability */
private static final long serialVersionUID = -6849794470754667710L;
```

4.3.3 三 String 构造函数

```
//不含参数的构造函数，一般没什么用，因为 value 是不可变量
public String() {
    this.value = new char[0];
}
//参数为 String 类型
public String(String original) {
    this.value = original.value;
    this.hash = original.hash;
}
//参数为 char 数组，使用 java.utils 包中的 Arrays 类复制
public String(char value[]) {
    this.value = Arrays.copyOf(value, value.length);
}
//从 bytes 数组中的 offset 位置开始，将长度为 length 的字节，以 charsetName 格式编码，拷贝到 valu
public String(byte bytes[], int offset, int length, String charsetName)
        throws UnsupportedEncodingException {
    if (charsetName == null)
        throw new NullPointerException("charsetName");
    checkBounds(bytes, offset, length);
    this.value = StringCoding.decode(charsetName, bytes, offset, length);
}
```

```
//调用 public String(byte bytes[], int offset, int length, String charsetName) 构造函数
public String(byte bytes[], String charsetName)
        throws UnsupportedEncodingException {
    this(bytes, 0, bytes.length, charsetName);
}
```

4.3.4 四 String 常用方法

```
boolean equals(Object anObject)
public boolean equals(Object anObject) {
    //如果引用的是同一个对象，返回真
    if (this == anObject) {
        return true;
    }
    //如果不是 String 类型的数据，返回假
    if (anObject instanceof String) {
        String anotherString = (String) anObject;
        int n = value.length;
        //如果 char 数组长度不相等，返回假
        if (n == anotherString.value.length) {
            char v1[] = value;
            char v2[] = anotherString.value;
            int i = 0;
            //从后往前单个字符判断，如果有不相等，返回假
            while (n-- != 0) {
                if (v1[i] != v2[i])
                        return false;
                i++;
            }
            //每个字符都相等，返回真
            return true;
        }
    }
    return false;
}
```

- equals 方法经常用得到，它用来判断两个对象从实际意义上是否相等，String 对象判断规则：
- 内存地址相同，则为真。
- 如果对象类型不是 String 类型，则为假。否则继续判断。
- 如果对象长度不相等，则为假。否则继续判断。
- 从后往前，判断 String 类中 char 数组 value 的单个字符是否相等，有不相等则为假。如果一直相等直到第一个数，则返回真。
- 由此可以看出，如果对两个超长的字符串进行比较还是非常费时间的。

```
int compareTo(String anotherString)
public int compareTo(String anotherString) {
    //自身对象字符串长度 len1
    int len1 = value.length;
    //被比较对象字符串长度 len2
    int len2 = anotherString.value.length;
    //取两个字符串长度的最小值 lim
    int lim = Math.min(len1, len2);
    char v1[] = value;
    char v2[] = anotherString.value;
    int k = 0;
    //从 value 的第一个字符开始到最小长度 lim 处为止，如果字符不相等，返回自身（对象不相等处字符-被
```

```
    while (k < lim) {
        char c1 = v1[k];
        char c2 = v2[k];
        if (c1 != c2) {
            return c1 - c2;
        }
        k++;
    }
    //如果前面都相等，则返回（自身长度-被比较对象长度）
    return len1 - len2;
}
```

- 这个方法写的很巧妙，先从 0 开始判断字符大小。如果两个对象能比较字符的地方比较完了还相等，就直接返回自身长度减被比较对象长度，如果两个字符串长度相等，则返回的是 0，巧妙地判断了三种情况。

```
int hashCode()
public int hashCode() {
    int h = hash;
    //如果 hash 没有被计算过，并且字符串不为空，则进行 hashCode 计算
    if (h == 0 && value.length > 0) {
        char val[] = value;
        //计算过程
        //s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
        for (int i = 0; i < value.length; i++) {
            h = 31 * h + val[i];
        }
        //hash 赋值
        hash = h;
    }
    return h;
}
```

- String 类重写了 hashCode 方法，Object 中的 hashCode 方法是一个 Native 调用。String 类的 hash 采用多项式计算得来，我们完全可以通过不相同的字符串得出同样的 hash，所以两个 String 对象的 hashCode 相同，并不代表两个 String 是一样的。

```
boolean startsWith(String prefix,int toffset)
public boolean startsWith(String prefix, int toffset) {
    char ta[] = value;
    int to = toffset;
    char pa[] = prefix.value;
    int po = 0;
    int pc = prefix.value.length;
    // Note: toffset might be near -1>>>1.
    //如果起始地址小于 0 或者（起始地址 + 所比较对象长度）大于自身对象长度，返回假
    if ((toffset < 0) || (toffset > value.length - pc)) {
        return false;
    }
    //从所比较对象的末尾开始比较
    while (--pc >= 0) {
        if (ta[to++] != pa[po++]) {
            return false;
        }
    }
    return true;
}
public boolean startsWith(String prefix) {
    return startsWith(prefix, 0);
}
public boolean endsWith(String suffix) {
```

```
    return startsWith(suffix, value.length - suffix.value.length);
}
```

- 起始比较和末尾比较都是比较经常用得到的方法，例如在判断一个字符串是不是 http 协议的，或者初步判断一个文件是不是 mp3 文件，都可以采用这个方法进行比较。

```
String concat(String str)
public String concat(String str) {
    int otherLen = str.length();
    //如果被添加的字符串为空，返回对象本身
    if (otherLen == 0) {
        return this;
    }
    int len = value.length;
    char buf[] = Arrays.copyOf(value, len + otherLen);
    str.getChars(buf, len);
    return new String(buf, true);
}
```

- concat 方法也是经常用的方法之一，它先判断被添加字符串是否为空来决定要不要创建新的对象。

```
String replace(char oldChar,char newChar)
public String replace(char oldChar, char newChar) {
    //新旧值先对比
    if (oldChar != newChar) {
        int len = value.length;
        int i = -1;
        char[] val = value; /* avoid getfield opcode */
        //找到旧值最开始出现的位置
        while (++i < len) {
            if (val[i] == oldChar) {
                break;
            }
        }
        //从那个位置开始，直到末尾，用新值代替出现的旧值
        if (i < len) {
            char buf[] = new char[len];
            for (int j = 0; j < i; j++) {
                buf[j] = val[j];
            }
            while (i < len) {
                char c = val[i];
                buf[i] = (c == oldChar) ? newChar : c;
                i++;
            }
            return new String(buf, true);
        }
    }
    return this;
}
```

- 这个方法也有讨巧的地方，例如最开始先找出旧值出现的位置，这样节省了一部分对比的时间。replace(String oldStr,String newStr) 方法通过正则表达式来判断。

```
String trim()
public String trim() {
    int len = value.length;
    int st = 0;
    char[] val = value;     /* avoid getfield opcode */
    //找到字符串前段没有空格的位置
```

```
    while ((st < len) && (val[st] <= ' ')) {
        st++;
    }
    //找到字符串末尾没有空格的位置
    while ((st < len) && (val[len - 1] <= ' ')) {
        len--;
    }
    //如果前后都没有出现空格，返回字符串本身
    return ((st > 0) || (len < value.length)) ? substring(st, len) : this;
}
```

- trim 方法用起来也 6 的飞起

```
String intern()
public native String intern();
```

- intern 方法是 Native 调用，它的作用是在方法区中的常量池里通过 equals 方法寻找等值的对象，如果没有找到则在常量池中开辟一片空间存放字符串并返回该对应 String 的引用，否则直接返回常量池中已存在 String 对象的引用。
- 将引言中第二段代码

```
//String a = new String("ab1");
//改为
String a = new String("ab1").intern();
```

- 则结果为为真，原因在于 a 所指向的地址来自于常量池，而 b 所指向的字符串常量默认会调用这个方法，所以 a 和 b 都指向了同一个地址空间。

```
int hash32()
private transient int hash32 = 0;
int hash32() {
    int h = hash32;
    if (0 == h) {
       // harmless data race on hash32 here.
       h = sun.misc.Hashing.murmur3_32(HASHING_SEED, value, 0, value.length);
       // ensure result is not zero to avoid recalcing
       h = (0 != h) ? h : 1;
       hash32 = h;
    }
    return h;
}
```

- 在 JDK1.7 中，Hash 相关集合类在 String 类作 key 的情况下，不再使用 hashCode 方式离散数据，而是采用 hash32 方法。这个方法默认使用系统当前时间，String 类地址，System 类地址等作为因子计算得到 hash 种子，通过 hash 种子在经过 hash 得到 32 位的 int 型数值。

```
public int length() {
    return value.length;
}
public String toString() {
    return this;
}
public boolean isEmpty() {
    return value.length == 0;
}
public char charAt(int index) {
    if ((index < 0) || (index >= value.length)) {
        throw new StringIndexOutOfBoundsException(index);
    }
    return value[index];
}
```

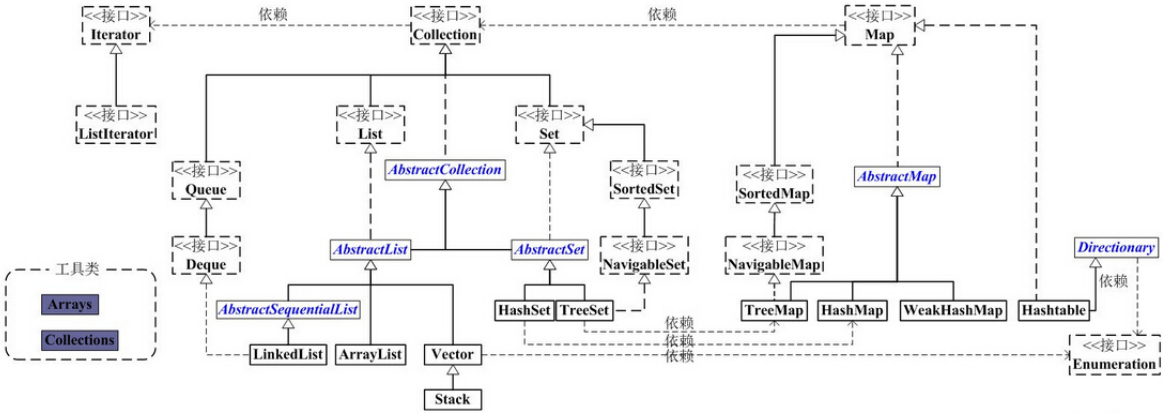
- 以上是一些简单的常用方法。

4.3.5 总结

- String 对象是不可变类型，返回类型为 String 的 String 方法每次返回的都是新的 String 对象，除了某些方法的某些特定条件返回自身。
- String 对象的三种比较方式：
  - == 内存比较：直接对比两个引用所指向的内存值，精确简洁直接明了。
  - equals 字符串值比较：比较两个引用所指对象字面值是否相等。
  - hashCode 字符串数值化比较：将字符串数值化。两个引用的 hashCode 相同，不保证内存一定相同，不保证字面值一定相同。

4.4 Java 集合框架

- Java 集合工具包位于 Java.util 包下，包含了很多常用的数据结构，如数组、链表、栈、队列、集合、哈希表等。学习 Java 集合框架下大致可以分为如下五个部分：List 列表、Set 集合、Map 映射、迭代器（Iterator、Enumeration）、工具类（Arrays、Collections）。
- Java 集合类的整体框架如下：



- 从上图中可以看出，集合类主要分为两大类：Collection 和 Map。
- Collection 是 List、Set 等集合高度抽象出来的接口，它包含了这些集合的基本操作，它主要又分为两大部分：List 和 Set。
- List 接口通常表示一个列表（数组、队列、链表、栈等），其中的元素可以重复，常用实现类为 ArrayList 和 LinkedList，另外还有不常用的 Vector。另外，LinkedList 还是实现了 Queue 接口，因此也可以作为队列使用。
- Set 接口通常表示一个集合，其中的元素不允许重复（通过 hashcode 和 equals 函数保证），常用实现类有 HashSet 和 TreeSet，HashSet 是通过 Map 中的 HashMap 实现的，而 TreeSet 是通过 Map 中的 TreeMap 实现的。另外，TreeSet 还实现了 SortedSet 接口，因此是有序的集合（集合中的元素要实现 Comparable 接口，并覆写 Compartor 函数才行）。我们看到，抽象类 AbstractCollection、AbstractList 和 AbstractSet 分别实现了 Collection、List 和 Set 接口，这就是在 Java 集合框架中用的很多的适配器设计模式，用这些抽象类去实现接口，在抽象类中实现接口中的若干或全部方法，这样下面的一些类只需直接继承该抽象类，并实现自己需要的方法即可，而不用实现接口中的全部抽象方法。
- Map 是一个映射接口，其中的每个元素都是一个 key-value 键值对，同样抽象类 AbstractMap 通过适配器模式实现了 Map 接口中的大部分函数，TreeMap、HashMap、WeakHashMap 等实现类都通过继承 AbstractMap 来实现，另外，不常用的 HashTable 直接实现了 Map 接口，它和 Vector 都是 JDK1.0 就引入的集合类。
- Iterator 是遍历集合的迭代器（不能遍历 Map，只用来遍历 Collection），Collection 的实现类都实现了 iterator() 函数，它返回一个 Iterator 对象，用来遍历集合，ListIterator 则专门用来遍历 List。而 Enumeration 则是 JDK1.0 时引入的，作用与 Iterator 相同，但它的功能比 Iterator 要少，它只能再 Hashtable、Vector 和 Stack 中使用。
- Arrays 和 Collections 是用来操作数组、集合的两个工具类，例如在 ArrayList 和 Vector 中大量调用了 Arrays.Copyof() 方法，而 Collections 中有很多静态方法可以返回各集合类的 synchronized 版本，即线程安全的版本，当然了，如果要用线程安全的结合类，首选 Concurrent 并发包下的对应的集合类。

4.5 ArrayList 源码剖析

- ArrayList 是基于数组实现的，是一个动态数组，其容量能自动增长，类似于 C 语言中的动态申请内存，动态增长内存。
- ArrayList 不是线程安全的，只能在单线程环境下，多线程环境下可以考虑用 collections.synchronizedList(List l) 函数返回一个线程安全的 ArrayList 类，也可以使用 concurrent 并发包下的 CopyOnWriteArrayList 类。
- ArrayList 实现了 Serializable 接口，因此它支持序列化，能够通过序列化传输，实现了 RandomAccess 接口，支持快速随机访问，实际上就是通过下标序号进行快速访问，实现了 Cloneable 接口，能被克隆。

4.5.1 ArrayList 源码剖析

- ArrayList 的源码如下（加入了比较详细的注释）：

```
package java.util;

public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable {

    // 序列版本号
    private static final long serialVersionUID = 8683452581122892189L;

    // ArrayList 基于该数组实现，用该数组保存数据
    private transient Object[] elementData;

    // ArrayList 中实际数据的数量
    private int size;

    // ArrayList 带容量大小的构造函数。
    public ArrayList(int initialCapacity) {
        super();
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity);
        // 新建一个数组
        this.elementData = new Object[initialCapacity];
    }

    // ArrayList 无参构造函数。默认容量是 10。
    public ArrayList() {
        this(10);
    }

    // 创建一个包含 collection 的 ArrayList
    public ArrayList(Collection<? extends E> c) {
        elementData = c.toArray();
        size = elementData.length;
        if (elementData.getClass() != Object[].class)
            elementData = Arrays.copyOf(elementData, size, Object[].class);
    }

    // 将当前容量值设为实际元素个数
    public void trimToSize() {
        modCount++;
        int oldCapacity = elementData.length;
        if (size < oldCapacity) {
            elementData = Arrays.copyOf(elementData, size);
        }
    }
```

```java
// 确定 ArrarList 的容量。
// 若 ArrayList 的容量不足以容纳当前的全部元素，设置 新的容量 =“(原始容量 x3)/2 + 1”
public void ensureCapacity(int minCapacity) {
    // 将“修改统计数”+1，该变量主要是用来实现 fail-fast 机制的
    modCount++;
    int oldCapacity = elementData.length;
    // 若当前容量不足以容纳当前的元素个数，设置 新的容量 =“(原始容量 x3)/2 + 1”
    if (minCapacity > oldCapacity) {
        Object oldData[] = elementData;
        int newCapacity = (oldCapacity * 3)/2 + 1;
        //如果还不够，则直接将 minCapacity 设置为当前容量
        if (newCapacity < minCapacity)
            newCapacity = minCapacity;
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
}

// 添加元素 e
public boolean add(E e) {
    // 确定 ArrayList 的容量大小
    ensureCapacity(size + 1);  // Increments modCount!!
    // 添加 e 到 ArrayList 中
    elementData[size++] = e;
    return true;
}

// 返回 ArrayList 的实际大小
public int size() {
    return size;
}

// ArrayList 是否包含 Object(o)
public boolean contains(Object o) {
    return indexOf(o) >= 0;
}

//返回 ArrayList 是否为空
public boolean isEmpty() {
    return size == 0;
}

// 正向查找，返回元素的索引值
public int indexOf(Object o) {
    if (o == null) {
        for (int i = 0; i < size; i++)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = 0; i < size; i++)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}

// 反向查找，返回元素的索引值
public int lastIndexOf(Object o) {
    if (o == null) {
```

```
        for (int i = size-1; i >= 0; i--)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = size-1; i >= 0; i--)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}

// 反向查找（从数组末尾向开始查找），返回元素（o）的索引值
public int lastIndexOf(Object o) {
    if (o == null) {
        for (int i = size-1; i >= 0; i--)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = size-1; i >= 0; i--)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}

// 返回 ArrayList 的 Object 数组
public Object[] toArray() {
    return Arrays.copyOf(elementData, size);
}

// 返回 ArrayList 元素组成的数组
public <T> T[] toArray(T[] a) {
    // 若数组 a 的大小 < ArrayList 的元素个数;
    // 则新建一个 T[] 数组，数组大小是“ArrayList 的元素个数”，并将“ArrayList”全部拷贝到新数
    if (a.length < size)
        return (T[]) Arrays.copyOf(elementData, size, a.getClass());

    // 若数组 a 的大小 >= ArrayList 的元素个数;
    // 则将 ArrayList 的全部元素都拷贝到数组 a 中。
    System.arraycopy(elementData, 0, a, 0, size);
    if (a.length > size)
        a[size] = null;
    return a;
}

// 获取 index 位置的元素值
public E get(int index) {
    RangeCheck(index);

    return (E) elementData[index];
}

// 设置 index 位置的值为 element
public E set(int index, E element) {
    RangeCheck(index);

    E oldValue = (E) elementData[index];
    elementData[index] = element;
```

```
    return oldValue;
}

// 将 e 添加到 ArrayList 中
public boolean add(E e) {
    ensureCapacity(size + 1);  // Increments modCount!!
    elementData[size++] = e;
    return true;
}

// 将 e 添加到 ArrayList 的指定位置
public void add(int index, E element) {
    if (index > size || index < 0)
        throw new IndexOutOfBoundsException("Index: "+index+", Size: "+size);
    ensureCapacity(size+1);  // Increments modCount!!
    System.arraycopy(elementData, index, elementData, index + 1, size - index);
    elementData[index] = element;
    size++;
}

// 删除 ArrayList 指定位置的元素
public E remove(int index) {
    RangeCheck(index);

    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;
}

// 删除 ArrayList 的指定元素
public boolean remove(Object o) {
    if (o == null) {
        for (int index = 0; index < size; index++)
            if (elementData[index] == null) {
                fastRemove(index);
                return true;
            }
    } else {
        for (int index = 0; index < size; index++)
            if (o.equals(elementData[index])) {
                fastRemove(index);
                return true;
            }
    }
    return false;
}

// 快速删除第 index 个元素
private void fastRemove(int index) {
    modCount++;
    int numMoved = size - index - 1;
    // 从"index+1" 开始，用后面的元素替换前面的元素。
```

```
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                         numMoved);
    // 将最后一个元素设为 null
    elementData[--size] = null; // Let gc do its work
}

// 删除元素
public boolean remove(Object o) {
    if (o == null) {
        for (int index = 0; index < size; index++)
            if (elementData[index] == null) {
                fastRemove(index);
                return true;
            }
    } else {
        // 便利 ArrayList，找到“元素 o”，则删除，并返回 true。
        for (int index = 0; index < size; index++)
            if (o.equals(elementData[index])) {
                fastRemove(index);
                return true;
            }
    }
    return false;
}

// 清空 ArrayList，将全部的元素设为 null
public void clear() {
    modCount++;
    for (int i = 0; i < size; i++)
        elementData[i] = null;
    size = 0;
}

// 将集合 c 追加到 ArrayList 中
public boolean addAll(Collection<? extends E> c) {
    Object[] a = c.toArray();
    int numNew = a.length;
    ensureCapacity(size + numNew);  // Increments modCount
    System.arraycopy(a, 0, elementData, size, numNew);
    size += numNew;
    return numNew != 0;
}

// 从 index 位置开始，将集合 c 添加到 ArrayList
public boolean addAll(int index, Collection<? extends E> c) {
    if (index > size || index < 0)
        throw new IndexOutOfBoundsException("Index: " + index + ", Size: " + size);

    Object[] a = c.toArray();
    int numNew = a.length;
    ensureCapacity(size + numNew);  // Increments modCount

    int numMoved = size - index;
    if (numMoved > 0)
        System.arraycopy(elementData, index, elementData, index + numNew, numMoved);
    System.arraycopy(a, 0, elementData, index, numNew);
    size += numNew;
```

```
        return numNew != 0;
    }

    // 删除 fromIndex 到 toIndex 之间的全部元素。
    protected void removeRange(int fromIndex, int toIndex) {
        modCount++;
        int numMoved = size - toIndex;
        System.arraycopy(elementData, toIndex, elementData, fromIndex, numMoved);

        // Let gc do its work
        int newSize = size - (toIndex-fromIndex);
        while (size != newSize)
            elementData[--size] = null;
    }

    private void RangeCheck(int index) {
        if (index >= size)
            throw new IndexOutOfBoundsException("Index: "+index+", Size: "+size);
    }

    // 克隆函数
    public Object clone() {
        try {
            ArrayList<E> v = (ArrayList<E>) super.clone();
            // 将当前 ArrayList 的全部元素拷贝到 v 中
            v.elementData = Arrays.copyOf(elementData, size);
            v.modCount = 0;
            return v;
        } catch (CloneNotSupportedException e) {
            // this shouldn't happen, since we are Cloneable
            throw new InternalError();
        }
    }


    // java.io.Serializable 的写入函数
    // 将 ArrayList 的“容量，所有的元素值”都写入到输出流中
    private void writeObject(java.io.ObjectOutputStream s)
        throws java.io.IOException{
        // Write out element count, and any hidden stuff
        int expectedModCount = modCount;
        s.defaultWriteObject();

        // 写入“数组的容量”
        s.writeInt(elementData.length);

        // 写入“数组的每一个元素”
        for (int i=0; i<size; i++)
            s.writeObject(elementData[i]);

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

    }


    // java.io.Serializable 的读取函数：根据写入方式读出
```

```
    // 先将 ArrayList 的“容量”读出，然后将“所有的元素值”读出
    private void readObject(java.io.ObjectInputStream s)
        throws java.io.IOException, ClassNotFoundException {
        // Read in size, and any hidden stuff
        s.defaultReadObject();

        // 从输入流中读取 ArrayList 的“容量”
        int arrayLength = s.readInt();
        Object[] a = elementData = new Object[arrayLength];

        // 从输入流中将“所有的元素值”读出
        for (int i=0; i<size; i++)
            a[i] = s.readObject();
    }
}
```

4.5.2 几点总结

- 关于 ArrayList 的源码，给出几点比较重要的总结：
  - 1. 注意其三个不同的构造方法。无参构造方法构造的 ArrayList 的容量默认为 10，带有 Collection 参数的构造方法，将 Collection 转化为数组赋给 ArrayList 的实现数组 elementData。
  - 2. 注意扩充容量的方法 ensureCapacity。ArrayList 在每次增加元素（可能是 1 个，也可能是一组）时，都要调用该方法来确保足够的容量。当容量不足以容纳当前的元素个数时，就设置新的容量为旧的容量的 1.5 倍加 1，如果设置后的新容量还不够，则直接新容量设置为传入的参数（也就是所需的容量），而后用 Arrays.copyof() 方法将元素拷贝到新的数组（详见下面的第 3 点）。从中可以看出，当容量不够时，每次增加元素，都要将原来的元素拷贝到一个新的数组中，非常之耗时，也因此建议在事先能确定元素数量的情况下，才使用 ArrayList，否则建议使用 LinkedList。
  - 3.ArrayList 的实现中大量地调用了 Arrays.copyof() 和 System.arraycopy() 方法。我们有必要对这两个方法的实现做下深入的了解。
    * 首先来看 Arrays.copyof() 方法。它有很多个重载的方法，但实现思路都是一样的，我们来看泛型版本的源码：

```
public static <T> T[] copyOf(T[] original, int newLength) {
    return (T[]) copyOf(original, newLength, original.getClass());
}
```

- 很明显调用了另一个 copyof 方法，该方法有三个参数，最后一个参数指明要转换的数据的类型，其源码如下：

```
public static <T,U> T[] copyOf(U[] original, int newLength, Class<? extends T[]> newType) {
    T[] copy = ((Object)newType == (Object)Object[].class)
        ? (T[]) new Object[newLength]
        : (T[]) Array.newInstance(newType.getComponentType(), newLength);
    System.arraycopy(original, 0, copy, 0,
                     Math.min(original.length, newLength));
    return copy;
}
```

- 这里可以很明显地看出，该方法实际上是在其内部又创建了一个长度为 newlength 的数组，调用 System.arraycopy() 方法，将原来数组中的元素复制到了新的数组中。
- 下面来看 System.arraycopy() 方法。该方法被标记了 native，调用了系统的 C/C++ 代码，在 JDK 中是看不到的，但在 openJDK 中可以看到其源码。该函数实际上最终调用了 C 语言的 memmove() 函数，因此它可以保证同一个数组内元素的正确复制和移动，比一般的复制方法的实现效率要高很多，很适合用来批量处理数组。Java 强烈推荐在复制大量数组元素时用该方法，以取得更高的效率。
- 4. 注意 ArrayList 的两个转化为静态数组的 toArray 方法。

- 第一个，Object[] toArray() 方法。该方法有可能会抛出 java.lang.ClassCastException 异常，如果直接用向下转型的方法，将整个 ArrayList 集合转变为指定类型的 Array 数组，便会抛出该异常，而如果转化为 Array 数组时不向下转型，而是将每个元素向下转型，则不会抛出该异常，显然对数组中的元素一个个进行向下转型，效率不高，且不太方便。
- 第二个，T[] toArray(T[] a) 方法。该方法可以直接将 ArrayList 转换得到的 Array 进行整体向下转型（转型其实是在该方法的源码中实现的），且从该方法的源码中可以看出，参数 a 的大小不足时，内部会调用 Arrays.copyOf 方法，该方法内部创建一个新的数组返回，因此对该方法的常用形式如下：

```
public static Integer[] vectorToArray2(ArrayList<Integer> v) {
    Integer[] newText = (Integer[])v.toArray(new Integer[0]);
    return newText;
}
```

- 5.ArrayList 基于数组实现，可以通过下标索引直接查找到指定位置的元素，因此查找效率高，但每次插入或删除元素，就要大量地移动元素，插入删除元素的效率低。
- 6. 在查找给定元素索引值等的方法中，源码都将该元素的值分为 null 和不为 null 两种情况处理，ArrayList 中允许元素为 null。

4.6 LinkedList 源码剖析

- LinkedList 是基于双向循环链表（从源码中可以很容易看出）实现的，除了可以当作链表来操作外，它还可以当作栈，队列和双端队列来使用。
- LinkedList 同样是非线程安全的，只在单线程下适合使用。
- LinkedList 实现了 Serializable 接口，因此它支持序列化，能够通过序列化传输，实现了 Cloneable 接口，能被克隆。

4.6.1 LinkedList 源码剖析 LinkedList 的源码如下（加入了比较详细的注释）

```
package java.util;

public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable     {
    // 链表的表头，表头不包含任何数据。Entry 是个链表类数据结构。
    private transient Entry<E> header = new Entry<E>(null, null, null);

    // LinkedList 中元素个数
    private transient int size = 0;

    // 默认构造函数：创建一个空的链表
    public LinkedList() {
        header.next = header.previous = header;
    }

    // 包含“集合”的构造函数：创建一个包含“集合”的 LinkedList
    public LinkedList(Collection<? extends E> c) {
        this();
        addAll(c);
    }

    // 获取 LinkedList 的第一个元素
    public E getFirst() {
        if (size==0)
            throw new NoSuchElementException();

        // 链表的表头 header 中不包含数据。
        // 这里返回 header 所指下一个节点所包含的数据。
```

```
    return header.next.element;
}

// 获取 LinkedList 的最后一个元素
public E getLast()  {
    if (size==0)
        throw new NoSuchElementException();

    // 由于 LinkedList 是双向链表；而表头 header 不包含数据。
    // 因而，这里返回表头 header 的前一个节点所包含的数据。
    return header.previous.element;
}

// 删除 LinkedList 的第一个元素
public E removeFirst() {
    return remove(header.next);
}

// 删除 LinkedList 的最后一个元素
public E removeLast() {
    return remove(header.previous);
}

// 将元素添加到 LinkedList 的起始位置
public void addFirst(E e) {
    addBefore(e, header.next);
}

// 将元素添加到 LinkedList 的结束位置
public void addLast(E e) {
    addBefore(e, header);
}

// 判断 LinkedList 是否包含元素 (o)
public boolean contains(Object o) {
    return indexOf(o) != -1;
}

// 返回 LinkedList 的大小
public int size() {
    return size;
}

// 将元素 (E) 添加到 LinkedList 中
public boolean add(E e) {
    // 将节点 (节点数据是 e) 添加到表头 (header) 之前。
    // 即，将节点添加到双向链表的末端。
    addBefore(e, header);
    return true;
}

// 从 LinkedList 中删除元素 (o)
// 从链表开始查找，如存在元素 (o) 则删除该元素并返回 true；
// 否则，返回 false。
public boolean remove(Object o) {
    if (o==null) {
        // 若 o 为 null 的删除情况
        for (Entry<E> e = header.next; e != header; e = e.next) {
```

```
            if (e.element==null) {
                remove(e);
                return true;
            }
        }
    } else {
        // 若 o 不为 null 的删除情况
        for (Entry<E> e = header.next; e != header; e = e.next) {
            if (o.equals(e.element)) {
                remove(e);
                return true;
            }
        }
    }
    return false;
}

// 将“集合 (c)”添加到 LinkedList 中。
// 实际上，是从双向链表的末尾开始，将“集合 (c)”添加到双向链表中。
public boolean addAll(Collection<? extends E> c) {
    return addAll(size, c);
}

// 从双向链表的 index 开始，将“集合 (c)”添加到双向链表中。
public boolean addAll(int index, Collection<? extends E> c) {
    if (index < 0 || index > size)
        throw new IndexOutOfBoundsException("Index: "+index+
                                            ", Size: "+size);
    Object[] a = c.toArray();
    // 获取集合的长度
    int numNew = a.length;
    if (numNew==0)
        return false;
    modCount++;

    // 设置“当前要插入节点的后一个节点”
    Entry<E> successor = (index==size ? header : entry(index));
    // 设置“当前要插入节点的前一个节点”
    Entry<E> predecessor = successor.previous;
    // 将集合 (c) 全部插入双向链表中
    for (int i=0; i<numNew; i++) {
        Entry<E> e = new Entry<E>((E)a[i], successor, predecessor);
        predecessor.next = e;
        predecessor = e;
    }
    successor.previous = predecessor;

    // 调整 LinkedList 的实际大小
    size += numNew;
    return true;
}

// 清空双向链表
public void clear() {
    Entry<E> e = header.next;
    // 从表头开始，逐个向后遍历；对遍历到的节点执行一下操作：
    // (01) 设置前一个节点为 null
    // (02) 设置当前节点的内容为 null
```

```
    // (03) 设置后一个节点为“新的当前节点”
    while (e != header) {
        Entry<E> next = e.next;
        e.next = e.previous = null;
        e.element = null;
        e = next;
    }
    header.next = header.previous = header;
    // 设置大小为 0
    size = 0;
    modCount++;
}

// 返回 LinkedList 指定位置的元素
public E get(int index) {
    return entry(index).element;
}

// 设置 index 位置对应的节点的值为 element
public E set(int index, E element) {
    Entry<E> e = entry(index);
    E oldVal = e.element;
    e.element = element;
    return oldVal;
}

// 在 index 前添加节点，且节点的值为 element
public void add(int index, E element) {
    addBefore(element, (index==size ? header : entry(index)));
}

// 删除 index 位置的节点
public E remove(int index) {
    return remove(entry(index));
}

// 获取双向链表中指定位置的节点
private Entry<E> entry(int index) {
    if (index < 0 || index >= size)
        throw new IndexOutOfBoundsException("Index: "+index+
                                            ", Size: "+size);
    Entry<E> e = header;
    // 获取 index 处的节点。
    // 若 index < 双向链表长度的 1/2, 则从前先后查找;
    // 否则，从后向前查找。
    if (index < (size >> 1)) {
        for (int i = 0; i <= index; i++)
            e = e.next;
    } else {
        for (int i = size; i > index; i--)
            e = e.previous;
    }
    return e;
}

// 从前向后查找，返回“值为对象 (o) 的节点对应的索引”
// 不存在就返回-1
public int indexOf(Object o) {
```

```
    int index = 0;
    if (o==null) {
        for (Entry e = header.next; e != header; e = e.next) {
            if (e.element==null)
                return index;
            index++;
        }
    } else {
        for (Entry e = header.next; e != header; e = e.next) {
            if (o.equals(e.element))
                return index;
            index++;
        }
    }
    return -1;
}

// 从后向前查找，返回“值为对象 (o) 的节点对应的索引”
// 不存在就返回-1
public int lastIndexOf(Object o) {
    int index = size;
    if (o==null) {
        for (Entry e = header.previous; e != header; e = e.previous) {
            index--;
            if (e.element==null)
                return index;
        }
    } else {
        for (Entry e = header.previous; e != header; e = e.previous) {
            index--;
            if (o.equals(e.element))
                return index;
        }
    }
    return -1;
}

// 返回第一个节点
// 若 LinkedList 的大小为 0，则返回 null
public E peek() {
    if (size==0)
        return null;
    return getFirst();
}

// 返回第一个节点
// 若 LinkedList 的大小为 0，则抛出异常
public E element() {
    return getFirst();
}

// 删除并返回第一个节点
// 若 LinkedList 的大小为 0，则返回 null
public E poll() {
    if (size==0)
        return null;
    return removeFirst();
}
```

```java
// 将 e 添加双向链表末尾
public boolean offer(E e) {
    return add(e);
}

// 将 e 添加双向链表开头
public boolean offerFirst(E e) {
    addFirst(e);
    return true;
}

// 将 e 添加双向链表末尾
public boolean offerLast(E e) {
    addLast(e);
    return true;
}

// 返回第一个节点
// 若 LinkedList 的大小为 0, 则返回 null
public E peekFirst() {
    if (size==0)
        return null;
    return getFirst();
}

// 返回最后一个节点
// 若 LinkedList 的大小为 0, 则返回 null
public E peekLast() {
    if (size==0)
        return null;
    return getLast();
}

// 删除并返回第一个节点
// 若 LinkedList 的大小为 0, 则返回 null
public E pollFirst() {
    if (size==0)
        return null;
    return removeFirst();
}

// 删除并返回最后一个节点
// 若 LinkedList 的大小为 0, 则返回 null
public E pollLast() {
    if (size==0)
        return null;
    return removeLast();
}

// 将 e 插入到双向链表开头
public void push(E e) {
    addFirst(e);
}

// 删除并返回第一个节点
public E pop() {
    return removeFirst();
```

```
}

// 从 LinkedList 开始向后查找，删除第一个值为元素 (o) 的节点
// 从链表开始查找，如存在节点的值为元素 (o) 的节点，则删除该节点
public boolean removeFirstOccurrence(Object o) {
    return remove(o);
}

// 从 LinkedList 末尾向前查找，删除第一个值为元素 (o) 的节点
// 从链表开始查找，如存在节点的值为元素 (o) 的节点，则删除该节点
public boolean removeLastOccurrence(Object o) {
    if (o==null) {
        for (Entry<E> e = header.previous; e != header; e = e.previous) {
            if (e.element==null) {
                remove(e);
                return true;
            }
        }
    } else {
        for (Entry<E> e = header.previous; e != header; e = e.previous) {
            if (o.equals(e.element)) {
                remove(e);
                return true;
            }
        }
    }
    return false;
}

// 返回“index 到末尾的全部节点”对应的 ListIterator 对象 (List 迭代器)
public ListIterator<E> listIterator(int index) {
    return new ListItr(index);
}

// List 迭代器
private class ListItr implements ListIterator<E> {
    // 上一次返回的节点
    private Entry<E> lastReturned = header;
    // 下一个节点
    private Entry<E> next;
    // 下一个节点对应的索引值
    private int nextIndex;
    // 期望的改变计数。用来实现 fail-fast 机制。
    private int expectedModCount = modCount;

    // 构造函数。
    // 从 index 位置开始进行迭代
    ListItr(int index) {
        // index 的有效性处理
        if (index < 0 || index > size)
            throw new IndexOutOfBoundsException("Index: "+index+ ", Size: "+size);
        // 若 “index 小于 ‘双向链表长度的一半’”，则从第一个元素开始往后查找；
        // 否则，从最后一个元素往前查找。
        if (index < (size >> 1)) {
            next = header.next;
            for (nextIndex=0; nextIndex<index; nextIndex++)
                next = next.next;
        } else {
```

```
            next = header;
            for (nextIndex=size; nextIndex>index; nextIndex--)
                next = next.previous;
        }
    }

    // 是否存在下一个元素
    public boolean hasNext() {
        // 通过元素索引是否等于“双向链表大小”来判断是否达到最后。
        return nextIndex != size;
    }

    // 获取下一个元素
    public E next() {
        checkForComodification();
        if (nextIndex == size)
            throw new NoSuchElementException();

        lastReturned = next;
        // next 指向链表的下一个元素
        next = next.next;
        nextIndex++;
        return lastReturned.element;
    }

    // 是否存在上一个元素
    public boolean hasPrevious() {
        // 通过元素索引是否等于 0，来判断是否达到开头。
        return nextIndex != 0;
    }

    // 获取上一个元素
    public E previous() {
        if (nextIndex == 0)
            throw new NoSuchElementException();

        // next 指向链表的上一个元素
        lastReturned = next = next.previous;
        nextIndex--;
        checkForComodification();
        return lastReturned.element;
    }

    // 获取下一个元素的索引
    public int nextIndex() {
        return nextIndex;
    }

    // 获取上一个元素的索引
    public int previousIndex() {
        return nextIndex-1;
    }

    // 删除当前元素。
    // 删除双向链表中的当前节点
    public void remove() {
        checkForComodification();
        Entry<E> lastNext = lastReturned.next;
```

```
        try {
            LinkedList.this.remove(lastReturned);
        } catch (NoSuchElementException e) {
            throw new IllegalStateException();
        }
        if (next==lastReturned)
            next = lastNext;
        else
            nextIndex--;
        lastReturned = header;
        expectedModCount++;
    }

    // 设置当前节点为 e
    public void set(E e) {
        if (lastReturned == header)
            throw new IllegalStateException();
        checkForComodification();
        lastReturned.element = e;
    }

    // 将 e 添加到当前节点的前面
    public void add(E e) {
        checkForComodification();
        lastReturned = header;
        addBefore(e, next);
        nextIndex++;
        expectedModCount++;
    }

    // 判断 “modCount 和 expectedModCount 是否相等”，依次来实现 fail-fast 机制。
    final void checkForComodification() {
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
    }
}

// 双向链表的节点所对应的数据结构。
// 包含 3 部分：上一节点，下一节点，当前节点值。
private static class Entry<E> {
    // 当前节点所包含的值
    E element;
    // 下一个节点
    Entry<E> next;
    // 上一个节点
    Entry<E> previous;

    /**
     * 链表节点的构造函数。
     * 参数说明：
     *   element  -- 节点所包含的数据
     *   next      -- 下一个节点
     *   previous -- 上一个节点
     */
    Entry(E element, Entry<E> next, Entry<E> previous) {
        this.element = element;
        this.next = next;
        this.previous = previous;
```

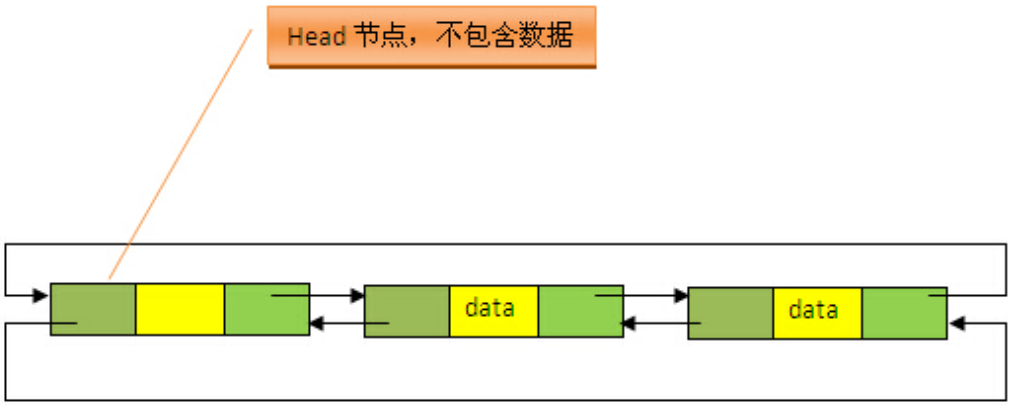
```
    }
}

// 将节点 (节点数据是 e) 添加到 entry 节点之前。
private Entry<E> addBefore(E e, Entry<E> entry) {
    // 新建节点 newEntry，将 newEntry 插入到节点 e 之前；并且设置 newEntry 的数据是 e
    Entry<E> newEntry = new Entry<E>(e, entry, entry.previous);
    newEntry.previous.next = newEntry;
    newEntry.next.previous = newEntry;
    // 修改 LinkedList 大小
    size++;
    // 修改 LinkedList 的修改统计数：用来实现 fail-fast 机制。
    modCount++;
    return newEntry;
}

// 将节点从链表中删除
private E remove(Entry<E> e) {
    if (e == header)
        throw new NoSuchElementException();

    E result = e.element;
    e.previous.next = e.next;
    e.next.previous = e.previous;
    e.next = e.previous = null;
    e.element = null;
    size--;
    modCount++;
    return result;
}

// 反向迭代器
public Iterator<E> descendingIterator() {
    return new DescendingIterator();
}

// 反向迭代器实现类。
private class DescendingIterator implements Iterator {
    final ListItr itr = new ListItr(size());
    // 反向迭代器是否下一个元素。
    // 实际上是判断双向链表的当前节点是否达到开头
    public boolean hasNext() {
        return itr.hasPrevious();
    }
    // 反向迭代器获取下一个元素。
    // 实际上是获取双向链表的前一个节点
    public E next() {
        return itr.previous();
    }
    // 删除当前节点
    public void remove() {
        itr.remove();
    }
}

// 返回 LinkedList 的 Object[] 数组
public Object[] toArray() {
    // 新建 Object[] 数组
```

```
    Object[] result = new Object[size];
    int i = 0;
    // 将链表中所有节点的数据都添加到 Object[] 数组中
    for (Entry<E> e = header.next; e != header; e = e.next)
        result[i++] = e.element;
    return result;
}

// 返回 LinkedList 的模板数组。所谓模板数组，即可以将 T 设为任意的数据类型
public <T> T[] toArray(T[] a) {
    // 若数组 a 的大小 < LinkedList 的元素个数 (意味着数组 a 不能容纳 LinkedList 中全部元素)
    // 则新建一个 T[] 数组，T[] 的大小为 LinkedList 大小，并将该 T[] 赋值给 a。
    if (a.length < size)
        a = (T[])java.lang.reflect.Array.newInstance(
                                                a.getClass().getComponentType(), size);
    // 将链表中所有节点的数据都添加到数组 a 中
    int i = 0;
    Object[] result = a;
    for (Entry<E> e = header.next; e != header; e = e.next)
        result[i++] = e.element;

    if (a.length > size)
        a[size] = null;

    return a;
}

// 克隆函数。返回 LinkedList 的克隆对象。
public Object clone() {
    LinkedList<E> clone = null;
    // 克隆一个 LinkedList 克隆对象
    try {
        clone = (LinkedList<E>) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new InternalError();
    }

    // 新建 LinkedList 表头节点
    clone.header = new Entry<E>(null, null, null);
    clone.header.next = clone.header.previous = clone.header;
    clone.size = 0;
    clone.modCount = 0;

    // 将链表中所有节点的数据都添加到克隆对象中
    for (Entry<E> e = header.next; e != header; e = e.next)
        clone.add(e.element);

    return clone;
}

// java.io.Serializable 的写入函数
// 将 LinkedList 的“容量，所有的元素值”都写入到输出流中
private void writeObject(java.io.ObjectOutputStream s)
    throws java.io.IOException {
    // Write out any hidden serialization magic
    s.defaultWriteObject();

    // 写入“容量”
```

```
        s.writeInt(size);

        // 将链表中所有节点的数据都写入到输出流中
        for (Entry e = header.next; e != header; e = e.next)
            s.writeObject(e.element);
    }

    // java.io.Serializable 的读取函数：根据写入方式反向读出
    // 先将 LinkedList 的“容量”读出，然后将“所有的元素值”读出
    private void readObject(java.io.ObjectInputStream s)
        throws java.io.IOException, ClassNotFoundException {
        // Read in any hidden serialization magic
        s.defaultReadObject();

        // 从输入流中读取“容量”
        int size = s.readInt();

        // 新建链表表头节点
        header = new Entry<E>(null, null, null);
        header.next = header.previous = header;

        // 从输入流中将“所有的元素值”并逐个添加到链表中
        for (int i=0; i<size; i++)
            addBefore((E)s.readObject(), header);
    }
}
```

4.6.2 几点总结

- 关于 LinkedList 的源码，给出几点比较重要的总结：
  - 1、从源码中很明显可以看出，LinkedList 的实现是基于双向循环链表的，且头结点中不存放数据，如下图；



  - 2、注意两个不同的构造方法。无参构造方法直接建立一个仅包含 head 节点的空链表，包含 Collection 的构造方法，先调用无参构造方法建立一个空链表，然后将 Collection 中的数据加入到链表的尾部后面。
  - 3、在查找和删除某元素时，源码中都划分为该元素为 null 和不为 null 两种情况来处理，LinkedList 中允许元素为 null。
  - 4、LinkedList 是基于链表实现的，因此不存在容量不足的问题，所以这里没有扩容的方法。
  - 5、注意源码中的 Entry entry(int index) 方法。该方法返回双向链表中指定位置处的节点，而链表中是没有下标索引的，要指定位置出的元素，就要遍历该链表，从源码的实现中，我们看到这里有一个加速动作。源码中先将 index 与长度 size 的一半比较，如果 index<size/2，就只从位置 0 往后遍历到位置

index 处，而如果 index>size/2，就只从位置 size 往前遍历到位置 index 处。这样可以减少一部分不必要的遍历，从而提高一定的效率（实际上效率还是很低）。

- 6、注意链表类对应的数据结构 Entry。如下;

```
// 双向链表的节点所对应的数据结构。
// 包含 3 部分：上一节点，下一节点，当前节点值。
private static class Entry<E> {
    // 当前节点所包含的值
    E element;
    // 下一个节点
    Entry<E> next;
    // 上一个节点
    Entry<E> previous;

    /**
     * 链表节点的构造函数。
     * 参数说明：
     *   element  —— 节点所包含的数据
     *   next     —— 下一个节点
     *   previous —— 上一个节点
     */
    Entry(E element, Entry<E> next, Entry<E> previous) {
        this.element = element;
        this.next = next;
        this.previous = previous;
    }
}
```

- 7、LinkedList 是基于链表实现的，因此插入删除效率高，查找效率低（虽然有一个加速动作）。
- 8、要注意源码中还实现了栈和队列的操作方法，因此也可以作为栈、队列和双端队列来使用。

4.7 Vector 源码剖析

- ector 也是基于数组实现的，是一个动态数组，其容量能自动增长。
- Vector 是 JDK1.0 引入了，它的很多实现方法都加入了同步语句，因此是线程安全的（其实也只是相对安全，有些时候还是要加入同步语句来保证线程的安全），可以用于多线程环境。
- Vector 没有实现 Serializable 接口，因此它不支持序列化，实现了 Cloneable 接口，能被克隆，实现了 RandomAccess 接口，支持快速随机访问。

4.7.1 Vector 源码剖析 Vector 的源码如下（加入了比较详细的注释）：

```
package java.util;

public class Vector<E>
    extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable     {

    // 保存 Vector 中数据的数组
    protected Object[] elementData;

    // 实际数据的数量
    protected int elementCount;

    // 容量增长系数
    protected int capacityIncrement;

    // Vector 的序列版本号
```

```
private static final long serialVersionUID = -2767605614048989439L;

// Vector 构造函数。默认容量是 10。
public Vector() {
    this(10);
}

// 指定 Vector 容量大小的构造函数
public Vector(int initialCapacity) {
    this(initialCapacity, 0);
}

// 指定 Vector" 容量大小 " 和 " 增长系数 " 的构造函数
public Vector(int initialCapacity, int capacityIncrement) {
    super();
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: "+
                                           initialCapacity);
    // 新建一个数组，数组容量是 initialCapacity
    this.elementData = new Object[initialCapacity];
    // 设置容量增长系数
    this.capacityIncrement = capacityIncrement;
}

// 指定集合的 Vector 构造函数。
public Vector(Collection<? extends E> c) {
    // 获取“集合 (c)”的数组，并将其赋值给 elementData
    elementData = c.toArray();
    // 设置数组长度
    elementCount = elementData.length;
    // c.toArray might (incorrectly) not return Object[] (see 6260652)
    if (elementData.getClass() != Object[].class)
        elementData = Arrays.copyOf(elementData, elementCount, Object[].class);
}

// 将数组 Vector 的全部元素都拷贝到数组 anArray 中
public synchronized void copyInto(Object[] anArray) {
    System.arraycopy(elementData, 0, anArray, 0, elementCount);
}

// 将当前容量值设为 = 实际元素个数
public synchronized void trimToSize() {
    modCount++;
    int oldCapacity = elementData.length;
    if (elementCount < oldCapacity) {
        elementData = Arrays.copyOf(elementData, elementCount);
    }
}

// 确认“Vector 容量”的帮助函数
private void ensureCapacityHelper(int minCapacity) {
    int oldCapacity = elementData.length;
    // 当 Vector 的容量不足以容纳当前的全部元素，增加容量大小。
    // 若 容量增量系数 >0(即 capacityIncrement>0)，则将容量增大当 capacityIncrement
    // 否则，将容量增大一倍。
    if (minCapacity > oldCapacity) {
        Object[] oldData = elementData;
        int newCapacity = (capacityIncrement > 0) ?
```

```
            (oldCapacity + capacityIncrement) : (oldCapacity * 2);
        if (newCapacity < minCapacity) {
            newCapacity = minCapacity;
        }
        elementData = Arrays.copyOf(elementData, newCapacity);
    }
}

// 确定 Vector 的容量。
public synchronized void ensureCapacity(int minCapacity) {
    // 将 Vector 的改变统计数 +1
    modCount++;
    ensureCapacityHelper(minCapacity);
}

// 设置容量值为 newSize
public synchronized void setSize(int newSize) {
    modCount++;
    if (newSize > elementCount) {
        // 若 "newSize 大于 Vector 容量", 则调整 Vector 的大小。
        ensureCapacityHelper(newSize);
    } else {
        // 若 "newSize 小于/等于 Vector 容量", 则将 newSize 位置开始的元素都设置为 null
        for (int i = newSize ; i < elementCount ; i++) {
            elementData[i] = null;
        }
    }
    elementCount = newSize;
}

// 返回“Vector 的总的容量”
public synchronized int capacity() {
    return elementData.length;
}

// 返回“Vector 的实际大小”, 即 Vector 中元素个数
public synchronized int size() {
    return elementCount;
}

// 判断 Vector 是否为空
public synchronized boolean isEmpty() {
    return elementCount == 0;
}

// 返回“Vector 中全部元素对应的 Enumeration”
public Enumeration<E> elements() {
    // 通过匿名类实现 Enumeration
    return new Enumeration<E>() {
        int count = 0;

        // 是否存在下一个元素
        public boolean hasMoreElements() {
            return count < elementCount;
        }

        // 获取下一个元素
        public E nextElement() {
```

```
            synchronized (Vector.this) {
                if (count < elementCount) {
                    return (E)elementData[count++];
                }
            }
            throw new NoSuchElementException("Vector Enumeration");
        }
    };
}

// 返回 Vector 中是否包含对象 (o)
public boolean contains(Object o) {
    return indexOf(o, 0) >= 0;
}


// 从 index 位置开始向后查找元素 (o)。
// 若找到，则返回元素的索引值；否则，返回-1
public synchronized int indexOf(Object o, int index) {
    if (o == null) {
        // 若查找元素为 null，则正向找出 null 元素，并返回它对应的序号
        for (int i = index ; i < elementCount ; i++)
            if (elementData[i]==null)
                return i;
    } else {
        // 若查找元素不为 null，则正向找出该元素，并返回它对应的序号
        for (int i = index ; i < elementCount ; i++)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}

// 查找并返回元素 (o) 在 Vector 中的索引值
public int indexOf(Object o) {
    return indexOf(o, 0);
}

// 从后向前查找元素 (o)。并返回元素的索引
public synchronized int lastIndexOf(Object o) {
    return lastIndexOf(o, elementCount-1);
}

// 从后向前查找元素 (o)。开始位置是从前向后的第 index 个数；
// 若找到，则返回元素的“索引值”；否则，返回-1。
public synchronized int lastIndexOf(Object o, int index) {
    if (index >= elementCount)
        throw new IndexOutOfBoundsException(index + " >= "+ elementCount);

    if (o == null) {
        // 若查找元素为 null，则反向找出 null 元素，并返回它对应的序号
        for (int i = index; i >= 0; i--)
            if (elementData[i]==null)
                return i;
    } else {
        // 若查找元素不为 null，则反向找出该元素，并返回它对应的序号
        for (int i = index; i >= 0; i--)
            if (o.equals(elementData[i]))
```

```
                return i;
        }
        return -1;
    }

    // 返回 Vector 中 index 位置的元素。
    // 若 index 月结，则抛出异常
    public synchronized E elementAt(int index) {
        if (index >= elementCount) {
            throw new ArrayIndexOutOfBoundsException(index + " >= " + elementCount);
        }

        return (E)elementData[index];
    }

    // 获取 Vector 中的第一个元素。
    // 若失败，则抛出异常!
    public synchronized E firstElement() {
        if (elementCount == 0) {
            throw new NoSuchElementException();
        }
        return (E)elementData[0];
    }

    // 获取 Vector 中的最后一个元素。
    // 若失败，则抛出异常!
    public synchronized E lastElement() {
        if (elementCount == 0) {
            throw new NoSuchElementException();
        }
        return (E)elementData[elementCount - 1];
    }

    // 设置 index 位置的元素值为 obj
    public synchronized void setElementAt(E obj, int index) {
        if (index >= elementCount) {
            throw new ArrayIndexOutOfBoundsException(index + " >= " +
                                                     elementCount);
        }
        elementData[index] = obj;
    }

    // 删除 index 位置的元素
    public synchronized void removeElementAt(int index) {
        modCount++;
        if (index >= elementCount) {
            throw new ArrayIndexOutOfBoundsException(index + " >= " +
                                                     elementCount);
        } else if (index < 0) {
            throw new ArrayIndexOutOfBoundsException(index);
        }

        int j = elementCount - index - 1;
        if (j > 0) {
            System.arraycopy(elementData, index + 1, elementData, index, j);
        }
        elementCount--;
        elementData[elementCount] = null; /* to let gc do its work */
```

```
}

// 在 index 位置处插入元素 (obj)
public synchronized void insertElementAt(E obj, int index) {
    modCount++;
    if (index > elementCount) {
        throw new ArrayIndexOutOfBoundsException(index
                                                 + " > " + elementCount);
    }
    ensureCapacityHelper(elementCount + 1);
    System.arraycopy(elementData, index, elementData, index + 1, elementCount - index);
    elementData[index] = obj;
    elementCount++;
}

// 将“元素 obj”添加到 Vector 末尾
public synchronized void addElement(E obj) {
    modCount++;
    ensureCapacityHelper(elementCount + 1);
    elementData[elementCount++] = obj;
}

// 在 Vector 中查找并删除元素 obj。
// 成功的话，返回 true；否则，返回 false。
public synchronized boolean removeElement(Object obj) {
    modCount++;
    int i = indexOf(obj);
    if (i >= 0) {
        removeElementAt(i);
        return true;
    }
    return false;
}

// 删除 Vector 中的全部元素
public synchronized void removeAllElements() {
    modCount++;
    // 将 Vector 中的全部元素设为 null
    for (int i = 0; i < elementCount; i++)
        elementData[i] = null;

    elementCount = 0;
}

// 克隆函数
public synchronized Object clone() {
    try {
        Vector<E> v = (Vector<E>) super.clone();
        // 将当前 Vector 的全部元素拷贝到 v 中
        v.elementData = Arrays.copyOf(elementData, elementCount);
        v.modCount = 0;
        return v;
    } catch (CloneNotSupportedException e) {
        // this shouldn't happen, since we are Cloneable
        throw new InternalError();
    }
}
```

```java
// 返回 Object 数组
public synchronized Object[] toArray() {
    return Arrays.copyOf(elementData, elementCount);
}

// 返回 Vector 的模板数组。所谓模板数组，即可以将 T 设为任意的数据类型
public synchronized <T> T[] toArray(T[] a) {
    // 若数组 a 的大小 < Vector 的元素个数;
    // 则新建一个 T[] 数组，数组大小是“Vector 的元素个数”，并将“Vector”全部拷贝到新数组中
    if (a.length < elementCount)
        return (T[]) Arrays.copyOf(elementData, elementCount, a.getClass());

    // 若数组 a 的大小 >= Vector 的元素个数;
    // 则将 Vector 的全部元素都拷贝到数组 a 中。
    System.arraycopy(elementData, 0, a, 0, elementCount);

    if (a.length > elementCount)
        a[elementCount] = null;

    return a;
}

// 获取 index 位置的元素
public synchronized E get(int index) {
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);

    return (E)elementData[index];
}

// 设置 index 位置的值为 element，并返回 index 位置的原始值
public synchronized E set(int index, E element) {
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);

    Object oldValue = elementData[index];
    elementData[index] = element;
    return (E)oldValue;
}

// 将“元素 e”添加到 Vector 最后。
public synchronized boolean add(E e) {
    modCount++;
    ensureCapacityHelper(elementCount + 1);
    elementData[elementCount++] = e;
    return true;
}

// 删除 Vector 中的元素 o
public boolean remove(Object o) {
    return removeElement(o);
}

// 在 index 位置添加元素 element
public void add(int index, E element) {
    insertElementAt(element, index);
}
```

```
// 删除 index 位置的元素，并返回 index 位置的原始值
public synchronized E remove(int index) {
    modCount++;
    if (index >= elementCount)
        throw new ArrayIndexOutOfBoundsException(index);
    Object oldValue = elementData[index];

    int numMoved = elementCount - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                         numMoved);
    elementData[--elementCount] = null; // Let gc do its work

    return (E)oldValue;
}

// 清空 Vector
public void clear() {
    removeAllElements();
}

// 返回 Vector 是否包含集合 c
public synchronized boolean containsAll(Collection<?> c) {
    return super.containsAll(c);
}

// 将集合 c 添加到 Vector 中
public synchronized boolean addAll(Collection<? extends E> c) {
    modCount++;
    Object[] a = c.toArray();
    int numNew = a.length;
    ensureCapacityHelper(elementCount + numNew);
    // 将集合 c 的全部元素拷贝到数组 elementData 中
    System.arraycopy(a, 0, elementData, elementCount, numNew);
    elementCount += numNew;
    return numNew != 0;
}

// 删除集合 c 的全部元素
public synchronized boolean removeAll(Collection<?> c) {
    return super.removeAll(c);
}

// 删除“非集合 c 中的元素”
public synchronized boolean retainAll(Collection<?> c)  {
    return super.retainAll(c);
}

// 从 index 位置开始，将集合 c 添加到 Vector 中
public synchronized boolean addAll(int index, Collection<? extends E> c) {
    modCount++;
    if (index < 0 || index > elementCount)
        throw new ArrayIndexOutOfBoundsException(index);

    Object[] a = c.toArray();
    int numNew = a.length;
    ensureCapacityHelper(elementCount + numNew);
```

```
        int numMoved = elementCount - index;
        if (numMoved > 0)
            System.arraycopy(elementData, index, elementData, index + numNew, numMoved);

        System.arraycopy(a, 0, elementData, index, numNew);
        elementCount += numNew;
        return numNew != 0;
    }

    // 返回两个对象是否相等
    public synchronized boolean equals(Object o) {
        return super.equals(o);
    }

    // 计算哈希值
    public synchronized int hashCode() {
        return super.hashCode();
    }

    // 调用父类的 toString()
    public synchronized String toString() {
        return super.toString();
    }

    // 获取 Vector 中 fromIndex(包括) 到 toIndex(不包括) 的子集
    public synchronized List<E> subList(int fromIndex, int toIndex) {
        return Collections.synchronizedList(super.subList(fromIndex, toIndex), this);
    }

    // 删除 Vector 中 fromIndex 到 toIndex 的元素
    protected synchronized void removeRange(int fromIndex, int toIndex) {
        modCount++;
        int numMoved = elementCount - toIndex;
        System.arraycopy(elementData, toIndex, elementData, fromIndex,
                         numMoved);

        // Let gc do its work
        int newElementCount = elementCount - (toIndex-fromIndex);
        while (elementCount != newElementCount)
            elementData[--elementCount] = null;
    }

    // java.io.Serializable 的写入函数
    private synchronized void writeObject(java.io.ObjectOutputStream s)
        throws java.io.IOException {
        s.defaultWriteObject();
    }
}
```

4.7.2 几点总结

- Vector 的源码实现总体与 ArrayList 类似，关于 Vector 的源码，给出如下几点总结：
  - 1、Vector 有四个不同的构造方法。无参构造方法的容量为默认值 10，仅包含容量的构造方法则将容量增长量（从源码中可以看出容量增长量的作用，第二点也会对容量增长量详细说）明置为 0。
  - 2、注意扩充容量的方法 ensureCapacityHelper。与 ArrayList 相同，Vector 在每次增加元素（可能是 1 个，也可能是一组）时，都要调用该方法来确保足够的容量。当容量不足以容纳当前的元素个数时，就先看构造方法中传入的容量增长量参数 CapacityIncrement 是否为 0，如果不为 0，就设置新的容量为就容量加上容量增长量，如果为 0，就设置新的容量为旧的容量的 2 倍，如果设置后的新容量还不够，

则直接新容量设置为传入的参数（也就是所需的容量），而后同样用 Arrays.copyof() 方法将元素拷贝到新的数组。

- 3、很多方法都加入了 synchronized 同步语句，来保证线程安全。
- 4、同样在查找给定元素索引值等的方法中，源码都将该元素的值分为 null 和不为 null 两种情况处理，Vector 中也允许元素为 null。
- 5、其他很多地方都与 ArrayList 实现大同小异，Vector 现在已经基本不再使用。

4.8 HashMap 源码剖析

- HashMap 是基于哈希表实现的，每一个元素都是一个 key-value 对，其内部通过单链表解决冲突问题，容量不足（超过了阈值）时，同样会自动增长。
- HashMap 是非线程安全的，只是用于单线程环境下，多线程环境下可以采用 concurrent 并发包下的 concurrentHashMap。
- HashMap 实现了 Serializable 接口，因此它支持序列化，实现了 Cloneable 接口，能被克隆。

4.8.1 HashMap 源码剖析

- HashMap 的源码如下（加入了比较详细的注释）：

```
package java.util;
import java.io.*;

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

    // 默认的初始容量（容量为 HashMap 中槽的数目）是 16，且实际容量必须是 2 的整数次幂。
    static final int DEFAULT_INITIAL_CAPACITY = 16;

    // 最大容量（必须是 2 的幂且小于 2 的 30 次方，传入容量过大将被这个值替换）
    static final int MAXIMUM_CAPACITY = 1 << 30;

    // 默认加载因子为 0.75
    static final float DEFAULT_LOAD_FACTOR = 0.75f;

    // 存储数据的 Entry 数组，长度是 2 的幂。
    // HashMap 采用链表法解决冲突，每一个 Entry 本质上是一个单向链表
    transient Entry[] table;

    // HashMap 的底层数组中已用槽的数量
    transient int size;

    // HashMap 的阈值，用于判断是否需要调整 HashMap 的容量（threshold = 容量 * 加载因子）
    int threshold;

    // 加载因子实际大小
    final float loadFactor;

    // HashMap 被改变的次数
    transient volatile int modCount;

    // 指定“容量大小”和“加载因子”的构造函数
    public HashMap(int initialCapacity, float loadFactor) {
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal initial capacity: " +
                                               initialCapacity);
        // HashMap 的最大容量只能是 MAXIMUM_CAPACITY
```

```
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    //加载因此不能小于 0
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal load factor: " +
                                           loadFactor);

    // 找出“大于 initialCapacity”的最小的 2 的幂
    int capacity = 1;
    while (capacity < initialCapacity)
        capacity <<= 1;

    // 设置“加载因子”
    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
    threshold = (int)(capacity * loadFactor);
    // 创建 Entry 数组，用来保存数据
    table = new Entry[capacity];
    init();
}


// 指定“容量大小”的构造函数
public HashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_LOAD_FACTOR);
}

// 默认构造函数。
public HashMap() {
    // 设置“加载因子”为默认加载因子 0.75
    this.loadFactor = DEFAULT_LOAD_FACTOR;
    // 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
    threshold = (int)(DEFAULT_INITIAL_CAPACITY * DEFAULT_LOAD_FACTOR);
    // 创建 Entry 数组，用来保存数据
    table = new Entry[DEFAULT_INITIAL_CAPACITY];
    init();
}

// 包含“子 Map”的构造函数
public HashMap(Map<? extends K, ? extends V> m) {
    this(Math.max((int) (m.size() / DEFAULT_LOAD_FACTOR) + 1,
                  DEFAULT_INITIAL_CAPACITY), DEFAULT_LOAD_FACTOR);
    // 将 m 中的全部元素逐个添加到 HashMap 中
    putAllForCreate(m);
}

//求 hash 值的方法，重新计算 hash 值
static int hash(int h) {
    h ^= (h >>> 20) ^ (h >>> 12);
    return h ^ (h >>> 7) ^ (h >>> 4);
}

// 返回 h 在数组中的索引值，这里用 & 代替取模，旨在提升效率
// h & (length-1) 保证返回值的小于 length
static int indexFor(int h, int length) {
    return h & (length-1);
}
```

```
public int size() {
    return size;
}

public boolean isEmpty() {
    return size == 0;
}

// 获取 key 对应的 value
public V get(Object key) {
    if (key == null)
        return getForNullKey();
    // 获取 key 的 hash 值
    int hash = hash(key.hashCode());
    // 在“该 hash 值对应的链表”上查找“键值等于 key”的元素
    for (Entry<K,V> e = table[indexFor(hash, table.length)];
         e != null;
         e = e.next) {
        Object k;
        //判断 key 是否相同
        if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
            return e.value;
    }
    //没找到则返回 null
    return null;
}

// 获取“key 为 null”的元素的值
// HashMap 将“key 为 null”的元素存储在 table[0] 位置，但不一定是该链表的第一个位置!
private V getForNullKey() {
    for (Entry<K,V> e = table[0]; e != null; e = e.next) {
        if (e.key == null)
            return e.value;
    }
    return null;
}

// HashMap 是否包含 key
public boolean containsKey(Object key) {
    return getEntry(key) != null;
}

// 返回“键为 key”的键值对
final Entry<K,V> getEntry(Object key) {
    // 获取哈希值
    // HashMap 将“key 为 null”的元素存储在 table[0] 位置，“key 不为 null”的则调用 hash() 计
    int hash = (key == null) ? 0 : hash(key.hashCode());
    // 在“该 hash 值对应的链表”上查找“键值等于 key”的元素
    for (Entry<K,V> e = table[indexFor(hash, table.length)];
         e != null;
         e = e.next) {
        Object k;
        if (e.hash == hash &&
            ((k = e.key) == key || (key != null && key.equals(k))))
            return e;
    }
    return null;
}
```

```
// 将“key-value”添加到 HashMap 中
public V put(K key, V value) {
    // 若“key 为 null”，则将该键值对添加到 table[0] 中。
    if (key == null)
        return putForNullKey(value);
    // 若“key 不为 null”，则计算该 key 的哈希值，然后将其添加到该哈希值对应的链表中。
    int hash = hash(key.hashCode());
    int i = indexFor(hash, table.length);
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        // 若“该 key”对应的键值对已经存在，则用新的 value 取代旧的 value。然后退出!
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }

    // 若“该 key”对应的键值对不存在，则将“key-value”添加到 table 中
    modCount++;
    //将 key-value 添加到 table[i] 处
    addEntry(hash, key, value, i);
    return null;
}

// putForNullKey() 的作用是将“key 为 null”键值对添加到 table[0] 位置
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;
        }
    }
    // 如果没有存在 key 为 null 的键值对，则直接题阿见到 table[0] 处!
    modCount++;
    addEntry(0, null, value, 0);
    return null;
}

// 创建 HashMap 对应的“添加方法”，
// 它和 put() 不同。putForCreate() 是内部方法，它被构造函数等调用，用来创建 HashMap
// 而 put() 是对外提供的往 HashMap 中添加元素的方法。
private void putForCreate(K key, V value) {
    int hash = (key == null) ? 0 : hash(key.hashCode());
    int i = indexFor(hash, table.length);

    // 若该 HashMap 表中存在“键值等于 key”的元素，则替换该元素的 value 值
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        if (e.hash == hash &&
            ((k = e.key) == key || (key != null && key.equals(k)))) {
            e.value = value;
            return;
        }
```

```
    }

    // 若该 HashMap 表中不存在“键值等于 key”的元素，则将该 key-value 添加到 HashMap 中
    createEntry(hash, key, value, i);
}

// 将“m”中的全部元素都添加到 HashMap 中。
// 该方法被内部的构造 HashMap 的方法所调用。
private void putAllForCreate(Map<? extends K, ? extends V> m) {
    // 利用迭代器将元素逐个添加到 HashMap 中
    for (Iterator<? extends Map.Entry<? extends K, ? extends V>> i = m.entrySet().iterator(
        Map.Entry<? extends K, ? extends V> e = i.next();
        putForCreate(e.getKey(), e.getValue());
    }
}

// 重新调整 HashMap 的大小，newCapacity 是调整后的容量
void resize(int newCapacity) {
    Entry[] oldTable = table;
    int oldCapacity = oldTable.length;
    //如果就容量已经达到了最大值，则不能再扩容，直接返回
    if (oldCapacity == MAXIMUM_CAPACITY) {
        threshold = Integer.MAX_VALUE;
        return;
    }

    // 新建一个 HashMap，将“旧 HashMap”的全部元素添加到“新 HashMap”中，
    // 然后，将“新 HashMap”赋值给“旧 HashMap”。
    Entry[] newTable = new Entry[newCapacity];
    transfer(newTable);
    table = newTable;
    threshold = (int)(newCapacity * loadFactor);
}

// 将 HashMap 中的全部元素都添加到 newTable 中
void transfer(Entry[] newTable) {
    Entry[] src = table;
    int newCapacity = newTable.length;
    for (int j = 0; j < src.length; j++) {
        Entry<K,V> e = src[j];
        if (e != null) {
            src[j] = null;
            do {
                Entry<K,V> next = e.next;
                int i = indexFor(e.hash, newCapacity);
                e.next = newTable[i];
                newTable[i] = e;
                e = next;
            } while (e != null);
        }
    }
}

// 将"m" 的全部元素都添加到 HashMap 中
public void putAll(Map<? extends K, ? extends V> m) {
    // 有效性判断
    int numKeysToBeAdded = m.size();
    if (numKeysToBeAdded == 0)
```

```
        return;

    // 计算容量是否足够，
    // 若“当前阀值容量 < 需要的容量”，则将容量 x2。
    if (numKeysToBeAdded > threshold) {
        int targetCapacity = (int)(numKeysToBeAdded / loadFactor + 1);
        if (targetCapacity > MAXIMUM_CAPACITY)
            targetCapacity = MAXIMUM_CAPACITY;
        int newCapacity = table.length;
        while (newCapacity < targetCapacity)
            newCapacity <<= 1;
        if (newCapacity > table.length)
            resize(newCapacity);
    }

    // 通过迭代器，将“m”中的元素逐个添加到 HashMap 中。
    for (Iterator<? extends Map.Entry<? extends K, ? extends V>> i = m.entrySet().iterator(
        Map.Entry<? extends K, ? extends V> e = i.next();
        put(e.getKey(), e.getValue());
    }
}

// 删除“键为 key”元素
public V remove(Object key) {
    Entry<K,V> e = removeEntryForKey(key);
    return (e == null ? null : e.value);
}

// 删除“键为 key”的元素
final Entry<K,V> removeEntryForKey(Object key) {
    // 获取哈希值。若 key 为 null，则哈希值为 0；否则调用 hash() 进行计算
    int hash = (key == null) ? 0 : hash(key.hashCode());
    int i = indexFor(hash, table.length);
    Entry<K,V> prev = table[i];
    Entry<K,V> e = prev;

    // 删除链表中“键为 key”的元素
    // 本质是“删除单向链表中的节点”
    while (e != null) {
        Entry<K,V> next = e.next;
        Object k;
        if (e.hash == hash &&
            ((k = e.key) == key || (key != null && key.equals(k)))) {
            modCount++;
            size--;
            if (prev == e)
                table[i] = next;
            else
                prev.next = next;
            e.recordRemoval(this);
            return e;
        }
        prev = e;
        e = next;
    }

    return e;
}
```

```
// 删除“键值对”
final Entry<K,V> removeMapping(Object o) {
    if (!(o instanceof Map.Entry))
        return null;

    Map.Entry<K,V> entry = (Map.Entry<K,V>) o;
    Object key = entry.getKey();
    int hash = (key == null) ? 0 : hash(key.hashCode());
    int i = indexFor(hash, table.length);
    Entry<K,V> prev = table[i];
    Entry<K,V> e = prev;

    // 删除链表中的“键值对 e”
    // 本质是“删除单向链表中的节点”
    while (e != null) {
        Entry<K,V> next = e.next;
        if (e.hash == hash && e.equals(entry)) {
            modCount++;
            size--;
            if (prev == e)
                table[i] = next;
            else
                prev.next = next;
            e.recordRemoval(this);
            return e;
        }
        prev = e;
        e = next;
    }

    return e;
}

// 清空 HashMap，将所有的元素设为 null
public void clear() {
    modCount++;
    Entry[] tab = table;
    for (int i = 0; i < tab.length; i++)
        tab[i] = null;
    size = 0;
}

// 是否包含“值为 value”的元素
public boolean containsValue(Object value) {
    // 若“value 为 null”，则调用 containsNullValue() 查找
    if (value == null)
        return containsNullValue();

    // 若“value 不为 null”，则查找 HashMap 中是否有值为 value 的节点。
    Entry[] tab = table;
    for (int i = 0; i < tab.length ; i++)
        for (Entry e = tab[i] ; e != null ; e = e.next)
            if (value.equals(e.value))
                return true;
    return false;
}
```

```
    // 是否包含 null 值
    private boolean containsNullValue() {
        Entry[] tab = table;
        for (int i = 0; i < tab.length ; i++)
            for (Entry e = tab[i] ; e != null ; e = e.next)
                if (e.value == null)
                    return true;
        return false;
    }

    // 克隆一个 HashMap，并返回 Object 对象
    public Object clone() {
        HashMap<K,V> result = null;
        try {
            result = (HashMap<K,V>)super.clone();
        } catch (CloneNotSupportedException e) {
            // assert false;
        }
        result.table = new Entry[table.length];
        result.entrySet = null;
        result.modCount = 0;
        result.size = 0;
        result.init();
        // 调用 putAllForCreate() 将全部元素添加到 HashMap 中
        result.putAllForCreate(this);

        return result;
    }

    // Entry 是单向链表。
    // 它是 “HashMap 链式存储法”对应的链表。
    // 它实现了 Map.Entry 接口，即实现 getKey(), getValue(), setValue(V value), equals(Object o)
    static class Entry<K,V> implements Map.Entry<K,V> {
        final K key;
        V value;
        // 指向下一个节点
        Entry<K,V> next;
        final int hash;

        // 构造函数。
        // 输入参数包括" 哈希值 (h)", " 键 (k)", " 值 (v)", " 下一节点 (n)"
        Entry(int h, K k, V v, Entry<K,V> n) {
            value = v;
            next = n;
            key = k;
            hash = h;
        }

        public final K getKey() {
            return key;
        }

        public final V getValue() {
            return value;
        }

        public final V setValue(V newValue) {
            V oldValue = value;
```

```
            value = newValue;
            return oldValue;
        }

        // 判断两个 Entry 是否相等
        // 若两个 Entry 的“key”和“value”都相等，则返回 true。
        // 否则，返回 false
        public final boolean equals(Object o) {
            if (!(o instanceof Map.Entry))
                return false;
            Map.Entry e = (Map.Entry)o;
            Object k1 = getKey();
            Object k2 = e.getKey();
            if (k1 == k2 || (k1 != null && k1.equals(k2))) {
                Object v1 = getValue();
                Object v2 = e.getValue();
                if (v1 == v2 || (v1 != null && v1.equals(v2)))
                    return true;
            }
            return false;
        }

        // 实现 hashCode()
        public final int hashCode() {
            return (key==null   ? 0 : key.hashCode()) ^
                (value==null ? 0 : value.hashCode());
        }

        public final String toString() {
            return getKey() + "=" + getValue();
        }

        // 当向 HashMap 中添加元素时，绘调用 recordAccess()。
        // 这里不做任何处理
        void recordAccess(HashMap<K,V> m) {
        }

        // 当从 HashMap 中删除元素时，绘调用 recordRemoval()。
        // 这里不做任何处理
        void recordRemoval(HashMap<K,V> m) {
        }
    }

    // 新增 Entry。将“key-value”插入指定位置，bucketIndex 是位置索引。
    void addEntry(int hash, K key, V value, int bucketIndex) {
        // 保存“bucketIndex”位置的值到“e”中
        Entry<K,V> e = table[bucketIndex];
        // 设置“bucketIndex”位置的元素为“新 Entry”，
        // 设置“e”为“新 Entry 的下一个节点”
        table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
        // 若 HashMap 的实际大小 不小于 “阈值”，则调整 HashMap 的大小
        if (size++ >= threshold)
            resize(2 * table.length);
    }

    // 创建 Entry。将“key-value”插入指定位置。
    void createEntry(int hash, K key, V value, int bucketIndex) {
        // 保存“bucketIndex”位置的值到“e”中
```

```
    Entry<K,V> e = table[bucketIndex];
    // 设置“bucketIndex”位置的元素为“新 Entry”,
    // 设置“e”为“新 Entry 的下一个节点”
    table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
    size++;
}

// HashIterator 是 HashMap 迭代器的抽象出来的父类，实现了公共了函数。
// 它包含“key 迭代器 (KeyIterator)”、“Value 迭代器 (ValueIterator)”和“Entry 迭代器 (EntryI
private abstract class HashIterator<E> implements Iterator<E> {
    // 下一个元素
    Entry<K,V> next;
    // expectedModCount 用于实现 fast-fail 机制。
    int expectedModCount;
    // 当前索引
    int index;
    // 当前元素
    Entry<K,V> current;

    HashIterator() {
        expectedModCount = modCount;
        if (size > 0) { // advance to first entry
            Entry[] t = table;
            // 将 next 指向 table 中第一个不为 null 的元素。
            // 这里利用了 index 的初始值为 0，从 0 开始依次向后遍历，直到找到不为 null 的元素就
            while (index < t.length && (next = t[index++]) == null)
                ;
        }
    }

    public final boolean hasNext() {
        return next != null;
    }

    // 获取下一个元素
    final Entry<K,V> nextEntry() {
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        Entry<K,V> e = next;
        if (e == null)
            throw new NoSuchElementException();

        // 注意!!!
        // 一个 Entry 就是一个单向链表
        // 若该 Entry 的下一个节点不为空，就将 next 指向下一个节点;
        // 否则，将 next 指向下一个链表 (也是下一个 Entry) 的不为 null 的节点。
        if ((next = e.next) == null) {
            Entry[] t = table;
            while (index < t.length && (next = t[index++]) == null)
                ;
        }
        current = e;
        return e;
    }

    // 删除当前元素
    public void remove() {
        if (current == null)
```

```
            throw new IllegalStateException();
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        Object k = current.key;
        current = null;
        HashMap.this.removeEntryForKey(k);
        expectedModCount = modCount;
    }

}

// value 的迭代器
private final class ValueIterator extends HashIterator<V> {
    public V next() {
        return nextEntry().value;
    }
}

// key 的迭代器
private final class KeyIterator extends HashIterator<K> {
    public K next() {
        return nextEntry().getKey();
    }
}

// Entry 的迭代器
private final class EntryIterator extends HashIterator<Map.Entry<K,V>> {
    public Map.Entry<K,V> next() {
        return nextEntry();
    }
}

// 返回一个“key 迭代器”
Iterator<K> newKeyIterator()   {
    return new KeyIterator();
}
// 返回一个“value 迭代器”
Iterator<V> newValueIterator()   {
    return new ValueIterator();
}
// 返回一个“entry 迭代器”
Iterator<Map.Entry<K,V>> newEntryIterator()   {
    return new EntryIterator();
}

// HashMap 的 Entry 对应的集合
private transient Set<Map.Entry<K,V>> entrySet = null;

// 返回“key 的集合”，实际上返回一个“KeySet 对象”
public Set<K> keySet() {
    Set<K> ks = keySet;
    return (ks != null ? ks : (keySet = new KeySet()));
}

// Key 对应的集合
// KeySet 继承于 AbstractSet，说明该集合中没有重复的 Key。
private final class KeySet extends AbstractSet<K> {
    public Iterator<K> iterator() {
```

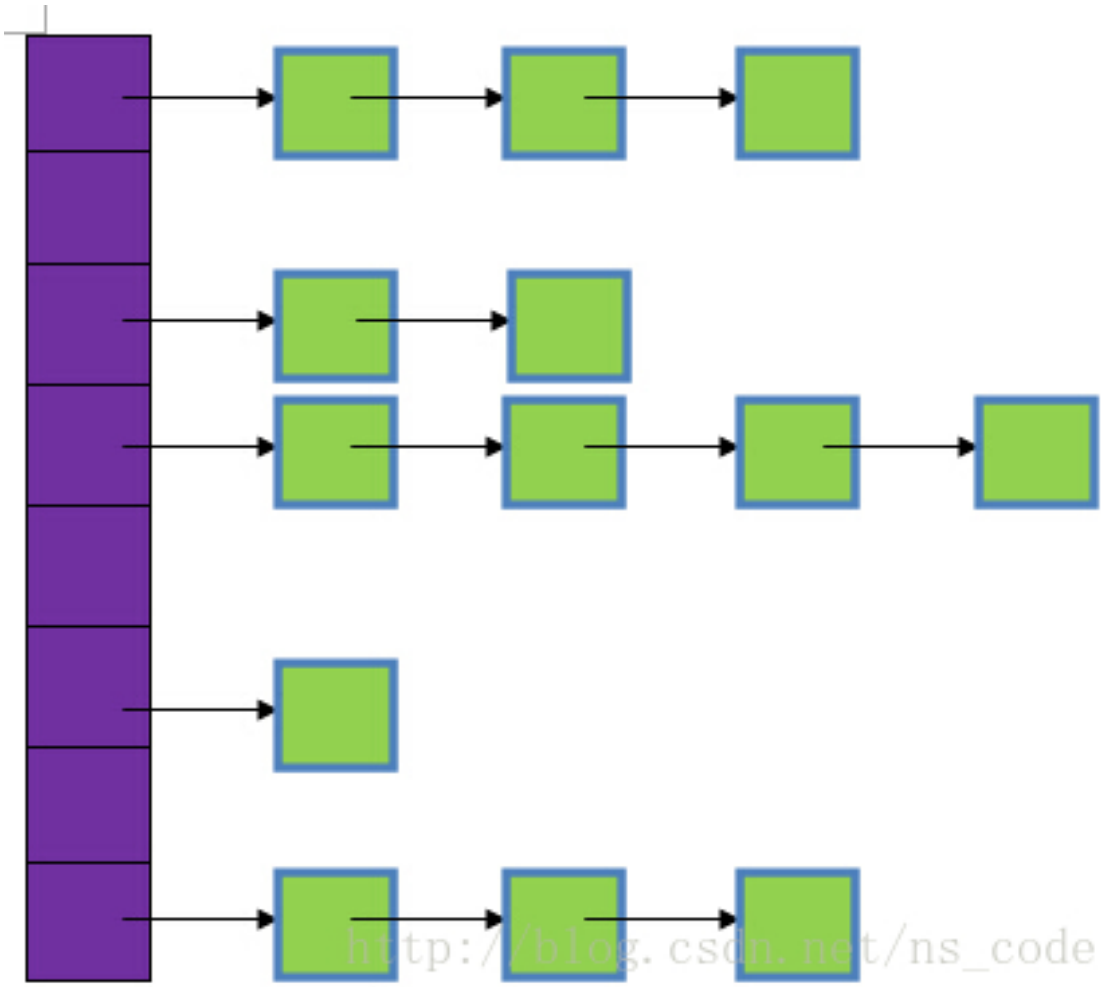
```
        return newKeyIterator();
    }
    public int size() {
        return size;
    }
    public boolean contains(Object o) {
        return containsKey(o);
    }
    public boolean remove(Object o) {
        return HashMap.this.removeEntryForKey(o) != null;
    }
    public void clear() {
        HashMap.this.clear();
    }
}

// 返回“value 集合”，实际上返回的是一个 Values 对象
public Collection<V> values() {
    Collection<V> vs = values;
    return (vs != null ? vs : (values = new Values()));
}

// “value 集合”
// Values 继承于 AbstractCollection，不同于“KeySet 继承于 AbstractSet”，
// Values 中的元素能够重复。因为不同的 key 可以指向相同的 value。
private final class Values extends AbstractCollection<V> {
    public Iterator<V> iterator() {
        return newValueIterator();
    }
    public int size() {
        return size;
    }
    public boolean contains(Object o) {
        return containsValue(o);
    }
    public void clear() {
        HashMap.this.clear();
    }
}

// 返回“HashMap 的 Entry 集合”
public Set<Map.Entry<K,V>> entrySet() {
    return entrySet0();
}

// 返回“HashMap 的 Entry 集合”，它实际是返回一个 EntrySet 对象
private Set<Map.Entry<K,V>> entrySet0() {
    Set<Map.Entry<K,V>> es = entrySet;
    return es != null ? es : (entrySet = new EntrySet());
}

// EntrySet 对应的集合
// EntrySet 继承于 AbstractSet，说明该集合中没有重复的 EntrySet。
private final class EntrySet extends AbstractSet<Map.Entry<K,V>> {
    public Iterator<Map.Entry<K,V>> iterator() {
        return newEntryIterator();
    }
    public boolean contains(Object o) {
```

```
        if (!(o instanceof Map.Entry))
            return false;
        Map.Entry<K,V> e = (Map.Entry<K,V>) o;
        Entry<K,V> candidate = getEntry(e.getKey());
        return candidate != null && candidate.equals(e);
    }
    public boolean remove(Object o) {
        return removeMapping(o) != null;
    }
    public int size() {
        return size;
    }
    public void clear() {
        HashMap.this.clear();
    }
}

// java.io.Serializable 的写入函数
// 将 HashMap 的“总的容量，实际容量，所有的 Entry”都写入到输出流中
private void writeObject(java.io.ObjectOutputStream s)
    throws IOException     {
    Iterator<Map.Entry<K,V>> i =
        (size > 0) ? entrySet0().iterator() : null;

    // Write out the threshold, loadfactor, and any hidden stuff
    s.defaultWriteObject();

    // Write out number of buckets
    s.writeInt(table.length);

    // Write out size (number of Mappings)
    s.writeInt(size);

    // Write out keys and values (alternating)
    if (i != null) {
        while (i.hasNext()) {
            Map.Entry<K,V> e = i.next();
            s.writeObject(e.getKey());
            s.writeObject(e.getValue());
        }
    }
}


private static final long serialVersionUID = 362498820763181265L;

// java.io.Serializable 的读取函数：根据写入方式读出
// 将 HashMap 的“总的容量，实际容量，所有的 Entry”依次读出
private void readObject(java.io.ObjectInputStream s)
    throws IOException, ClassNotFoundException     {
    // Read in the threshold, loadfactor, and any hidden stuff
    s.defaultReadObject();

    // Read in number of buckets and allocate the bucket array;
    int numBuckets = s.readInt();
    table = new Entry[numBuckets];

    init();  // Give subclass a chance to do its thing.
```

```
        // Read in size (number of Mappings)
        int size = s.readInt();

        // Read the keys and values, and put the mappings in the HashMap
        for (int i=0; i<size; i++) {
            K key = (K) s.readObject();
            V value = (V) s.readObject();
            putForCreate(key, value);
        }
    }

    // 返回“HashMap 总的容量”
    int   capacity()     { return table.length; }
    // 返回“HashMap 的加载因子”
    float loadFactor()   { return loadFactor;   }
}
```

4.8.2 几点总结

- 1、首先要清楚 HashMap 的存储结构，如下图所示：



- 图中，紫色部分即代表哈希表，也称为哈希数组，数组的每个元素都是一个单链表的头节点，链表是用来解决冲突的，如果不同的 key 映射到了数组的同一位置处，就将其放入单链表中。
- 2、首先看链表中节点的数据结构：

```
// Entry 是单向链表。
// 它是 “HashMap 链式存储法”对应的链表。
```

```java
// 它实现了 Map.Entry 接口，即实现 getKey(), getValue(), setValue(V value), equals(Object o), ha
static class Entry<K,V> implements Map.Entry<K,V> {
    final K key;
    V value;
    // 指向下一个节点
    Entry<K,V> next;
    final int hash;

    // 构造函数。
    // 输入参数包括" 哈希值 (h)", " 键 (k)", " 值 (v)", " 下一节点 (n)"
    Entry(int h, K k, V v, Entry<K,V> n) {
        value = v;
        next = n;
        key = k;
        hash = h;
    }

    public final K getKey() {
        return key;
    }

    public final V getValue() {
        return value;
    }

    public final V setValue(V newValue) {
        V oldValue = value;
        value = newValue;
        return oldValue;
    }

    // 判断两个 Entry 是否相等
    // 若两个 Entry 的“key”和“value”都相等，则返回 true。
    // 否则，返回 false
    public final boolean equals(Object o) {
        if (!(o instanceof Map.Entry))
            return false;
        Map.Entry e = (Map.Entry)o;
        Object k1 = getKey();
        Object k2 = e.getKey();
        if (k1 == k2 || (k1 != null && k1.equals(k2))) {
            Object v1 = getValue();
            Object v2 = e.getValue();
            if (v1 == v2 || (v1 != null && v1.equals(v2)))
                return true;
        }
        return false;
    }

    // 实现 hashCode()
    public final int hashCode() {
        return (key==null   ? 0 : key.hashCode()) ^
               (value==null ? 0 : value.hashCode());
    }

    public final String toString() {
        return getKey() + "=" + getValue();
    }
```

```
    // 当向 HashMap 中添加元素时，绘调用 recordAccess()。
    // 这里不做任何处理
    void recordAccess(HashMap<K,V> m) {
    }

    // 当从 HashMap 中删除元素时，绘调用 recordRemoval()。
    // 这里不做任何处理
    void recordRemoval(HashMap<K,V> m) {
    }
}
```

- 它的结构元素除了 key、value、hash 外，还有 next，next 指向下一个节点。另外，这里覆写了 equals 和 hashCode 方法来保证键值对的独一无二。
- 3、HashMap 共有四个构造方法。构造方法中提到了两个很重要的参数：初始容量和加载因子。这两个参数是影响 HashMap 性能的重要参数，其中容量表示哈希表中槽的数量（即哈希数组的长度），初始容量是创建哈希表时的容量（从构造函数中可以看出，如果不指明，则默认为 16），加载因子是哈希表在其容量自动增加之前可以达到多满的一种尺度，当哈希表中的条目数超出了加载因子与当前容量的乘积时，则要对该哈希表进行 resize 操作（即扩容）。
  - 下面说下加载因子，如果加载因子越大，对空间的利用更充分，但是查找效率会降低（链表长度会越来越长）；如果加载因子太小，那么表中的数据将过于稀疏（很多空间还没用，就开始扩容了），对空间造成严重浪费。如果我们在构造方法中不指定，则系统默认加载因子为 0.75，这是一个比较理想的值，一般情况下我们是无需修改的。
  - 另外，无论我们指定的容量为多少，构造方法都会将实际容量设为不小于指定容量的 2 的次方的一个数，且最大值不能超过 2 的 30 次方
- 4、HashMap 中 key 和 value 都允许为 null。
- 5、要重点分析下 HashMap 中用的最多的两个方法 put 和 get。先从比较简单的 get 方法着手，源码如下：

```
1  // 获取 key 对应的 value
2  public V get(Object key) {
3      if (key == null)
4          return getForNullKey();
5      // 获取 key 的 hash 值
6      int hash = hash(key.hashCode());
7      // 在“该 hash 值对应的链表”上查找“键值等于 key”的元素
8      for (Entry<K,V> e = table[indexFor(hash, table.length)];
9           e != null;
10          e = e.next) {
11         Object k;
12 /判断 key 是否相同
13         if (e.hash == hash && ((k = e.key) == key || key.equals(k)))
14             return e.value;
15     }
16 没找到则返回 null
17     return null;
18 }
19
20 // 获取“key 为 null”的元素的值
21 // HashMap 将“key 为 null”的元素存储在 table[0] 位置，但不一定是该链表的第一个位置!
22 private V getForNullKey() {
23     for (Entry<K,V> e = table[0]; e != null; e = e.next) {
24         if (e.key == null)
25             return e.value;
26     }
27     return null;
28 }
```

  - 首先，如果 key 为 null，则直接从哈希表的第一个位置 table[1]对应的链表上查找。记住，key 为 null 的键值对永远都放在以 table[1]为头结点的链表中，当然不一定是存放在头结点 table[1]中。
  - 如果 key 不为 null，则先求的 key 的 hash 值，根据 hash 值找到在 table 中的索引，在该索引对应的单链表中查找是否有键值对的 key 与目标 key 相等，有就返回对应的 value，没有则返回 null。
  - put 方法稍微复杂些，代码如下：

```
// 将“key-value”添加到 HashMap 中
public V put(K key, V value) {
    // 若“key 为 null”，则将该键值对添加到 table[0] 中。
    if (key == null)
        return putForNullKey(value);
    // 若“key 不为 null”，则计算该 key 的哈希值，然后将其添加到该哈希值对应的链表中。
    int hash = hash(key.hashCode());
    int i = indexFor(hash, table.length);
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        // 若“该 key”对应的键值对已经存在，则用新的 value 取代旧的 value。然后退出！
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }

    // 若“该 key”对应的键值对不存在，则将“key-value”添加到 table 中
    modCount++;
//将 key-value 添加到 table[i] 处
    addEntry(hash, key, value, i);
    return null;
}
```

- 如果 key 为 null，则将其添加到 table[1]对应的链表中，putForNullKey 的源码如下：

```
// putForNullKey() 的作用是将“key 为 null”键值对添加到 table[0] 位置
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;
        }
    }
    // 如果没有存在 key 为 null 的键值对，则直接题阿见到 table[0] 处！
    modCount++;
    addEntry(0, null, value, 0);
    return null;
}
```

- 如果 key 不为 null，则同样先求出 key 的 hash 值，根据 hash 值得出在 table 中的索引，而后遍历对应的单链表，如果单链表中存在与目标 key 相等的键值对，则将新的 value 覆盖旧的 value，比将旧的 value 返回，如果找不到与目标 key 相等的键值对，或者该单链表为空，则将该键值对插入到改单链表的头结点位置（每次新插入的节点都是放在头结点的位置），该操作是有 addEntry 方法实现的，它的源码如下：

```
// 新增 Entry。将“key-value”插入指定位置，bucketIndex 是位置索引。
void addEntry(int hash, K key, V value, int bucketIndex) {
    // 保存“bucketIndex”位置的值到“e”中
```

[1]DEFINITION NOT FOUND.

```
    Entry<K,V> e = table[bucketIndex];
    // 设置“bucketIndex”位置的元素为“新 Entry”，
    // 设置“e”为“新 Entry 的下一个节点”
    table[bucketIndex] = new Entry<K,V>(hash, key, value, e);
    // 若 HashMap 的实际大小 不小于 “阈值”，则调整 HashMap 的大小
    if (size++ >= threshold)
        resize(2 * table.length);
}
```

- 注意这里倒数第三行的构造方法，将 key-value 键值对赋给 table[bucketIndex]，并将其 next 指向元素 e，这便将 key-value 放到了头结点中，并将之前的头结点接在了它的后面。该方法也说明，每次 put 键值对的时候，总是将新的该键值对放在 table[bucketIndex] 处（即头结点处）。
- 两外注意最后两行代码，每次加入键值对时，都要判断当前已用的槽的数目是否大于等于阀值（容量 * 加载因子），如果大于等于，则进行扩容，将容量扩为原来容量的 2 倍。
- 6、关于扩容。上面我们看到了扩容的方法，resize 方法，它的源码如下：

```
// 重新调整 HashMap 的大小，newCapacity 是调整后的单位
void resize(int newCapacity) {
    Entry[] oldTable = table;
    int oldCapacity = oldTable.length;
    if (oldCapacity == MAXIMUM_CAPACITY) {
        threshold = Integer.MAX_VALUE;
        return;
    }

    // 新建一个 HashMap，将“旧 HashMap”的全部元素添加到“新 HashMap”中，
    // 然后，将“新 HashMap”赋值给“旧 HashMap”。
    Entry[] newTable = new Entry[newCapacity];
    transfer(newTable);
    table = newTable;
    threshold = (int)(newCapacity * loadFactor);
}
```

- 很明显，是新建了一个 HashMap 的底层数组，而后调用 transfer 方法，将就 HashMap 的全部元素添加到新的 HashMap 中（要重新计算元素在新的数组中的索引位置）。transfer 方法的源码如下：

```
// 将 HashMap 中的全部元素都添加到 newTable 中
void transfer(Entry[] newTable) {
    Entry[] src = table;
    int newCapacity = newTable.length;
    for (int j = 0; j < src.length; j++) {
        Entry<K,V> e = src[j];
        if (e != null) {
            src[j] = null;
            do {
                Entry<K,V> next = e.next;
                int i = indexFor(e.hash, newCapacity);
                e.next = newTable[i];
                newTable[i] = e;
                e = next;
            } while (e != null);
        }
    }
}
```

- 很明显，扩容是一个相当耗时的操作，因为它需要重新计算这些元素在新的数组中的位置并进行复制处理。因此，我们在用 HashMap 的时，最好能提前预估下 HashMap 中元素的个数，这样有助于提高 HashMap 的性能。

- 7、注意 containsKey 方法和 containsValue 方法。前者直接可以通过 key 的哈希值将搜索范围定位到指定索引对应的链表，而后者要对哈希数组的每个链表进行搜索。
- 8、我们重点来分析下求 hash 值和索引值的方法，这两个方法便是 HashMap 设计的最为核心的部分，二者结合能保证哈希表中的元素尽可能均匀地散列。
  - 计算哈希值的方法如下：

```
static int hash(int h) {
        h ^= (h >>> 20) ^ (h >>> 12);
        return h ^ (h >>> 7) ^ (h >>> 4);
}
```

- 它只是一个数学公式，IDK 这样设计对 hash 值的计算，自然有它的好处，至于为什么这样设计，我们这里不去追究，只要明白一点，用的位的操作使 hash 值的计算效率很高。
- 由 hash 值找到对应索引的方法如下：

```
static int indexFor(int h, int length) {
        return h & (length-1);
}
```

- 这个我们要重点说下，我们一般对哈希表的散列很自然地会想到用 hash 值对 length 取模（即除法散列法），Hashtable 中也是这样实现的，这种方法基本能保证元素在哈希表中散列的比较均匀，但取模会用到除法运算，效率很低，HashMap 中则通过 h&(length-1) 的方法来代替取模，同样实现了均匀的散列，但效率要高很多，这也是 HashMap 对 Hashtable 的一个改进。
- 接下来，我们分析下为什么哈希表的容量一定要是 2 的整数次幂。首先，length 为 2 的整数次幂的话，h&(length-1) 就相当于对 length 取模，这样便保证了散列的均匀，同时也提升了效率；其次，length 为 2 的整数次幂的话，为偶数，这样 length-1 为奇数，奇数的最后一位是 1，这样便保证了 h&(length-1) 的最后一位可能为 0，也可能为 1（这取决于 h 的值），即与后的结果可能为偶数，也可能为奇数，这样便可以保证散列的均匀性，而如果 length 为奇数的话，很明显 length-1 为偶数，它的最后一位是 0，这样 h&(length-1) 的最后一位肯定为 0，即只能为偶数，这样任何 hash 值都只会被散列到数组的偶数下标位置上，这便浪费了近一半的空间，因此，length 取 2 的整数次幂，是为了使不同 hash 值发生碰撞的概率较小，这样就能使元素在哈希表中均匀地散列。

4.9 HashTable 源码剖析

- HashTable 同样是基于哈希表实现的，同样每个元素都是 key-value 对，其内部也是通过单链表解决冲突问题，容量不足（超过了阈值）时，同样会自动增长。
- Hashtable 也是 JDK1.0 引入的类，是线程安全的，能用于多线程环境中。
- Hashtable 同样实现了 Serializable 接口，它支持序列化，实现了 Cloneable 接口，能被克隆。

4.9.1 Hashtable 源码剖析 Hashtable 的源码的很多实现都和 HashMap 差不多，源码如下（加入了比较详细的注释）：

```
package java.util;
import java.io.*;

public class Hashtable<K,V>
    extends Dictionary<K,V>
    implements Map<K,V>, Cloneable, java.io.Serializable {

    // 保存 key-value 的数组。
    // Hashtable 同样采用单链表解决冲突，每一个 Entry 本质上是一个单向链表
    private transient Entry[] table;

    // Hashtable 中键值对的数量
    private transient int count;
```

```
// 阈值，用于判断是否需要调整 Hashtable 的容量（threshold = 容量 * 加载因子）
private int threshold;

// 加载因子
private float loadFactor;

// Hashtable 被改变的次数，用于 fail-fast 机制的实现
private transient int modCount = 0;

// 序列版本号
private static final long serialVersionUID = 1421746759512286392L;

// 指定“容量大小”和“加载因子”的构造函数
public Hashtable(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity);
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal Load: "+loadFactor);

    if (initialCapacity==0)
        initialCapacity = 1;
    this.loadFactor = loadFactor;
    table = new Entry[initialCapacity];
    threshold = (int)(initialCapacity * loadFactor);
}

// 指定“容量大小”的构造函数
public Hashtable(int initialCapacity) {
    this(initialCapacity, 0.75f);
}

// 默认构造函数。
public Hashtable() {
    // 默认构造函数，指定的容量大小是 11；加载因子是 0.75
    this(11, 0.75f);
}

// 包含“子 Map”的构造函数
public Hashtable(Map<? extends K, ? extends V> t) {
    this(Math.max(2*t.size(), 11), 0.75f);
    // 将“子 Map”的全部元素都添加到 Hashtable 中
    putAll(t);
}

private int hash(Object k) {
    if (useAltHashing) {
        if (k.getClass() == String.class) {
            return sun.misc.Hashing.stringHash32((String) k);
        } else {
            int h = hashSeed ^ 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);
        }
    } else  {
        return k.hashCode();
```

```
    }
}

public synchronized int size() {
    return count;
}

public synchronized boolean isEmpty() {
    return count == 0;
}

// 返回“所有 key”的枚举对象
public synchronized Enumeration<K> keys() {
    return this.<K>getEnumeration(KEYS);
}

// 返回“所有 value”的枚举对象
public synchronized Enumeration<V> elements() {
    return this.<V>getEnumeration(VALUES);
}

// 判断 Hashtable 是否包含“值 (value)”
public synchronized boolean contains(Object value) {
    //注意，Hashtable 中的 value 不能是 null，
    // 若是 null 的话，抛出异常！
    if (value == null) {
        throw new NullPointerException();
    }

    // 从后向前遍历 table 数组中的元素 (Entry)
    // 对于每个 Entry(单向链表)，逐个遍历，判断节点的值是否等于 value
    Entry tab[] = table;
    for (int i = tab.length ; i-- > 0 ;) {
        for (Entry<K,V> e = tab[i] ; e != null ; e = e.next) {
            if (e.value.equals(value)) {
                return true;
            }
        }
    }
    return false;
}

public boolean containsValue(Object value) {
    return contains(value);
}

// 判断 Hashtable 是否包含 key
public synchronized boolean containsKey(Object key) {
    Entry tab[] = table;
    //计算 hash 值，直接用 key 的 hashCode 代替
    int hash = key.hashCode();
    // 计算在数组中的索引值
    int index = (hash & 0x7FFFFFFF) % tab.length;
    // 找到“key 对应的 Entry(链表)”，然后在链表中找出“哈希值”和“键值”与 key 都相等的元素
    for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {
        if ((e.hash == hash) && e.key.equals(key)) {
            return true;
        }
```

```
    }
    return false;
}

// 返回 key 对应的 value，没有的话返回 null
public synchronized V get(Object key) {
    Entry tab[] = table;
    int hash = hash(key);
    // 计算索引值，
    int index = (hash & 0x7FFFFFFF) % tab.length;
    // 找到“key 对应的 Entry(链表)”，然后在链表中找出“哈希值”和“键值”与 key 都相等的元素
    for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {
        if ((e.hash == hash) && e.key.equals(key)) {
            return e.value;
        }
    }
    return null;
}

// 调整 Hashtable 的长度，将长度变成原来的 2 倍 +1
protected void rehash() {
    int oldCapacity = table.length;
    Entry[] oldMap = table;

    //创建新容量大小的 Entry 数组
    int newCapacity = oldCapacity * 2 + 1;
    Entry[] newMap = new Entry[newCapacity];

    modCount++;
    threshold = (int)(newCapacity * loadFactor);
    table = newMap;

    //将“旧的 Hashtable”中的元素复制到“新的 Hashtable”中
    for (int i = oldCapacity ; i-- > 0 ;) {
        for (Entry<K,V> old = oldMap[i] ; old != null ; ) {
            Entry<K,V> e = old;
            old = old.next;
            //重新计算 index
            int index = (e.hash & 0x7FFFFFFF) % newCapacity;
            e.next = newMap[index];
            newMap[index] = e;
        }
    }
}

// 将“key-value”添加到 Hashtable 中
public synchronized V put(K key, V value) {
    // Hashtable 中不能插入 value 为 null 的元素!!!
    if (value == null) {
        throw new NullPointerException();
    }

    // 若“Hashtable 中已存在键为 key 的键值对”，
    // 则用“新的 value”替换“旧的 value”
    Entry tab[] = table;
    int hash = hash(key);
    int index = (hash & 0x7FFFFFFF) % tab.length;
    for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {
```

```
        if ((e.hash == hash) && e.key.equals(key)) {
            V old = e.value;
            e.value = value;
            return old;
        }
    }

    // 若“Hashtable 中不存在键为 key 的键值对”,
    // 将“修改统计数”+1
    modCount++;
    //  若“Hashtable 实际容量” > “阈值”(阈值 = 总的容量 * 加载因子)
    //  则调整 Hashtable 的大小
    if (count >= threshold) {
        rehash();

        tab = table;
        index = (hash & 0x7FFFFFFF) % tab.length;
    }

    //将新的 key-value 对插入到 tab[index] 处（即链表的头结点）
    Entry<K,V> e = tab[index];
    tab[index] = new Entry<K,V>(hash, key, value, e);
    count++;
    return null;
}

// 删除 Hashtable 中键为 key 的元素
public synchronized V remove(Object key) {
    Entry tab[] = table;
    int hash = hash(key);
    int index = (hash & 0x7FFFFFFF) % tab.length;

    //从 table[index] 链表中找出要删除的节点，并删除该节点。
    //因为是单链表，因此要保留带删节点的前一个节点，才能有效地删除节点
    for (Entry<K,V> e = tab[index], prev = null ; e != null ; prev = e, e = e.next) {
        if ((e.hash == hash) && e.key.equals(key)) {
            modCount++;
            if (prev != null) {
                prev.next = e.next;
            } else {
                tab[index] = e.next;
            }
            count--;
            V oldValue = e.value;
            e.value = null;
            return oldValue;
        }
    }
    return null;
}

// 将“Map(t)”的中全部元素逐一添加到 Hashtable 中
public synchronized void putAll(Map<? extends K, ? extends V> t) {
    for (Map.Entry<? extends K, ? extends V> e : t.entrySet())
        put(e.getKey(), e.getValue());
}

// 清空 Hashtable
```

```java
// 将 Hashtable 的 table 数组的值全部设为 null
public synchronized void clear() {
    Entry tab[] = table;
    modCount++;
    for (int index = tab.length; --index >= 0; )
        tab[index] = null;
    count = 0;
}

// 克隆一个 Hashtable，并以 Object 的形式返回。
public synchronized Object clone() {
    try {
        Hashtable<K,V> t = (Hashtable<K,V>) super.clone();
        t.table = new Entry[table.length];
        for (int i = table.length ; i-- > 0 ; ) {
            t.table[i] = (table[i] != null)
                ? (Entry<K,V>) table[i].clone() : null;
        }
        t.keySet = null;
        t.entrySet = null;
        t.values = null;
        t.modCount = 0;
        return t;
    } catch (CloneNotSupportedException e) {
        throw new InternalError();
    }
}

public synchronized String toString() {
    int max = size() - 1;
    if (max == -1)
        return "{}";

    StringBuilder sb = new StringBuilder();
    Iterator<Map.Entry<K,V>> it = entrySet().iterator();

    sb.append('{');
    for (int i = 0; ; i++) {
        Map.Entry<K,V> e = it.next();
        K key = e.getKey();
        V value = e.getValue();
        sb.append(key   == this ? "(this Map)" : key.toString());
        sb.append('=');
        sb.append(value == this ? "(this Map)" : value.toString());

        if (i == max)
            return sb.append('}').toString();
        sb.append(", ");
    }
}

// 获取 Hashtable 的枚举类对象
// 若 Hashtable 的实际大小为 0，则返回“空枚举类”对象；
// 否则，返回正常的 Enumerator 的对象。
private <T> Enumeration<T> getEnumeration(int type) {
    if (count == 0) {
        return (Enumeration<T>)emptyEnumerator;
    } else {
```

```
            return new Enumerator<T>(type, false);
        }
    }

    // 获取 Hashtable 的迭代器
    // 若 Hashtable 的实际大小为 0, 则返回“空迭代器”对象;
    // 否则, 返回正常的 Enumerator 的对象。(Enumerator 实现了迭代器和枚举两个接口)
    private <T> Iterator<T> getIterator(int type) {
        if (count == 0) {
            return (Iterator<T>) emptyIterator;
        } else {
            return new Enumerator<T>(type, true);
        }
    }

    // Hashtable 的“key 的集合”。它是一个 Set, 没有重复元素
    private transient volatile Set<K> keySet = null;
    // Hashtable 的“key-value 的集合”。它是一个 Set, 没有重复元素
    private transient volatile Set<Map.Entry<K,V>> entrySet = null;
    // Hashtable 的“key-value 的集合”。它是一个 Collection, 可以有重复元素
    private transient volatile Collection<V> values = null;

    // 返回一个被 synchronizedSet 封装后的 KeySet 对象
    // synchronizedSet 封装的目的是对 KeySet 的所有方法都添加 synchronized, 实现多线程同步
    public Set<K> keySet() {
        if (keySet == null)
            keySet = Collections.synchronizedSet(new KeySet(), this);
        return keySet;
    }

    // Hashtable 的 Key 的 Set 集合。
    // KeySet 继承于 AbstractSet, 所以, KeySet 中的元素没有重复的。
    private class KeySet extends AbstractSet<K> {
        public Iterator<K> iterator() {
            return getIterator(KEYS);
        }
        public int size() {
            return count;
        }
        public boolean contains(Object o) {
            return containsKey(o);
        }
        public boolean remove(Object o) {
            return Hashtable.this.remove(o) != null;
        }
        public void clear() {
            Hashtable.this.clear();
        }
    }

    // 返回一个被 synchronizedSet 封装后的 EntrySet 对象
    // synchronizedSet 封装的目的是对 EntrySet 的所有方法都添加 synchronized, 实现多线程同步
    public Set<Map.Entry<K,V>> entrySet() {
        if (entrySet==null)
            entrySet = Collections.synchronizedSet(new EntrySet(), this);
        return entrySet;
    }
```

```
// Hashtable 的 Entry 的 Set 集合。
// EntrySet 继承于 AbstractSet，所以，EntrySet 中的元素没有重复的。
private class EntrySet extends AbstractSet<Map.Entry<K,V>> {
    public Iterator<Map.Entry<K,V>> iterator() {
        return getIterator(ENTRIES);
    }

    public boolean add(Map.Entry<K,V> o) {
        return super.add(o);
    }

    // 查找 EntrySet 中是否包含 Object(O)
    // 首先，在 table 中找到 o 对应的 Entry 链表
    // 然后，查找 Entry 链表中是否存在 Object
    public boolean contains(Object o) {
        if (!(o instanceof Map.Entry))
            return false;
        Map.Entry entry = (Map.Entry)o;
        Object key = entry.getKey();
        Entry[] tab = table;
        int hash = hash(key);
        int index = (hash & 0x7FFFFFFF) % tab.length;

        for (Entry e = tab[index]; e != null; e = e.next)
            if (e.hash==hash && e.equals(entry))
                return true;
        return false;
    }

    // 删除元素 Object(O)
    // 首先，在 table 中找到 o 对应的 Entry 链表
    // 然后，删除链表中的元素 Object
    public boolean remove(Object o) {
        if (!(o instanceof Map.Entry))
            return false;
        Map.Entry<K,V> entry = (Map.Entry<K,V>) o;
        K key = entry.getKey();
        Entry[] tab = table;
        int hash = hash(key);
        int index = (hash & 0x7FFFFFFF) % tab.length;

        for (Entry<K,V> e = tab[index], prev = null; e != null;
             prev = e, e = e.next) {
            if (e.hash==hash && e.equals(entry)) {
                modCount++;
                if (prev != null)
                    prev.next = e.next;
                else
                    tab[index] = e.next;

                count--;
                e.value = null;
                return true;
            }
        }
        return false;
    }
```

```
        public int size() {
            return count;
        }

        public void clear() {
            Hashtable.this.clear();
        }
    }

    // 返回一个被 synchronizedCollection 封装后的 ValueCollection 对象
    // synchronizedCollection 封装的目的是对 ValueCollection 的所有方法都添加 synchronized，实现
    public Collection<V> values() {
        if (values==null)
            values = Collections.synchronizedCollection(new ValueCollection(),
                                                        this);
        return values;
    }

    // Hashtable 的 value 的 Collection 集合。
    // ValueCollection 继承于 AbstractCollection，所以，ValueCollection 中的元素可以重复的。
    private class ValueCollection extends AbstractCollection<V> {
        public Iterator<V> iterator() {
            return getIterator(VALUES);
        }
        public int size() {
            return count;
        }
        public boolean contains(Object o) {
            return containsValue(o);
        }
        public void clear() {
            Hashtable.this.clear();
        }
    }

    // 重新 equals() 函数
    // 若两个 Hashtable 的所有 key-value 键值对都相等，则判断它们两个相等
    public synchronized boolean equals(Object o) {
        if (o == this)
            return true;

        if (!(o instanceof Map))
            return false;
        Map<K,V> t = (Map<K,V>) o;
        if (t.size() != size())
            return false;

        try {
            // 通过迭代器依次取出当前 Hashtable 的 key-value 键值对
            // 并判断该键值对，存在于 Hashtable 中。
            // 若不存在，则立即返回 false；否则，遍历完“当前 Hashtable”并返回 true。
            Iterator<Map.Entry<K,V>> i = entrySet().iterator();
            while (i.hasNext()) {
                Map.Entry<K,V> e = i.next();
                K key = e.getKey();
                V value = e.getValue();
                if (value == null) {
                    if (!(t.get(key)==null && t.containsKey(key)))
```

```
                    return false;
            } else {
                if (!value.equals(t.get(key)))
                    return false;
            }
        }
    } catch (ClassCastException unused)   {
        return false;
    } catch (NullPointerException unused) {
        return false;
    }

    return true;
}

// 计算 Entry 的 hashCode
// 若 Hashtable 的实际大小为 0 或者 加载因子 <0，则返回 0。
// 否则，返回“Hashtable 中的每个 Entry 的 key 和 value 的异或值 的总和”。
public synchronized int hashCode() {
    int h = 0;
    if (count == 0 || loadFactor < 0)
        return h;  // Returns zero

    loadFactor = -loadFactor;  // Mark hashCode computation in progress
    Entry[] tab = table;
    for (int i = 0; i < tab.length; i++)
        for (Entry e = tab[i]; e != null; e = e.next)
            h += e.key.hashCode() ^ e.value.hashCode();
    loadFactor = -loadFactor;  // Mark hashCode computation complete

    return h;
}

// java.io.Serializable 的写入函数
// 将 Hashtable 的“总的容量，实际容量，所有的 Entry”都写入到输出流中
private synchronized void writeObject(java.io.ObjectOutputStream s)
    throws IOException     {
    // Write out the length, threshold, loadfactor
    s.defaultWriteObject();

    // Write out length, count of elements and then the key/value objects
    s.writeInt(table.length);
    s.writeInt(count);
    for (int index = table.length-1; index >= 0; index--) {
        Entry entry = table[index];

        while (entry != null) {
            s.writeObject(entry.key);
            s.writeObject(entry.value);
            entry = entry.next;
        }
    }
}

// java.io.Serializable 的读取函数：根据写入方式读出
// 将 Hashtable 的“总的容量，实际容量，所有的 Entry”依次读出
private void readObject(java.io.ObjectInputStream s)
    throws IOException, ClassNotFoundException     {
```

```
    // Read in the length, threshold, and loadfactor
    s.defaultReadObject();

    // Read the original length of the array and number of elements
    int origlength = s.readInt();
    int elements = s.readInt();

    // Compute new size with a bit of room 5% to grow but
    // no larger than the original size.  Make the length
    // odd if it's large enough, this helps distribute the entries.
    // Guard against the length ending up zero, that's not valid.
    int length = (int)(elements * loadFactor) + (elements / 20) + 3;
    if (length > elements && (length & 1) == 0)
        length--;
    if (origlength > 0 && length > origlength)
        length = origlength;

    Entry[] table = new Entry[length];
    count = 0;

    // Read the number of elements and then all the key/value objects
    for (; elements > 0; elements--) {
        K key = (K)s.readObject();
        V value = (V)s.readObject();
        // synch could be eliminated for performance
        reconstitutionPut(table, key, value);
    }
    this.table = table;
}

private void reconstitutionPut(Entry[] tab, K key, V value)
    throws StreamCorruptedException     {
    if (value == null) {
        throw new java.io.StreamCorruptedException();
    }
    // Makes sure the key is not already in the hashtable.
    // This should not happen in deserialized version.
    int hash = key.hashCode();
    int index = (hash & 0x7FFFFFFF) % tab.length;
    for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {
        if ((e.hash == hash) && e.key.equals(key)) {
            throw new java.io.StreamCorruptedException();
        }
    }
    // Creates the new entry.
    Entry<K,V> e = tab[index];
    tab[index] = new Entry<K,V>(hash, key, value, e);
    count++;
}

// Hashtable 的 Entry 节点，它本质上是一个单向链表。
// 也因此，我们才能推断出 Hashtable 是由拉链法实现的散列表
private static class Entry<K,V> implements Map.Entry<K,V> {
    // 哈希值
    int hash;
    K key;
    V value;
    // 指向的下一个 Entry，即链表的下一个节点
```

```
    Entry<K,V> next;

    // 构造函数
    protected Entry(int hash, K key, V value, Entry<K,V> next) {
        this.hash = hash;
        this.key = key;
        this.value = value;
        this.next = next;
    }

    protected Object clone() {
        return new Entry<K,V>(hash, key, value,
                              (next==null ? null : (Entry<K,V>) next.clone()));
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }

    // 设置 value。若 value 是 null，则抛出异常。
    public V setValue(V value) {
        if (value == null)
            throw new NullPointerException();

        V oldValue = this.value;
        this.value = value;
        return oldValue;
    }

    // 覆盖 equals() 方法，判断两个 Entry 是否相等。
    // 若两个 Entry 的 key 和 value 都相等，则认为它们相等。
    public boolean equals(Object o) {
        if (!(o instanceof Map.Entry))
            return false;
        Map.Entry e = (Map.Entry)o;

        return (key==null ? e.getKey()==null : key.equals(e.getKey())) &&
            (value==null ? e.getValue()==null : value.equals(e.getValue()));
    }

    public int hashCode() {
        return hash ^ (value==null ? 0 : value.hashCode());
    }

    public String toString() {
        return key.toString()+"="+value.toString();
    }
}

private static final int KEYS = 0;
private static final int VALUES = 1;
private static final int ENTRIES = 2;

// Enumerator 的作用是提供了“通过 elements() 遍历 Hashtable 的接口” 和 “通过 entrySet() 遍
```

```
private class Enumerator<T> implements Enumeration<T>, Iterator<T> {
    // 指向 Hashtable 的 table
    Entry[] table = Hashtable.this.table;
    // Hashtable 的总的大小
    int index = table.length;
    Entry<K,V> entry = null;
    Entry<K,V> lastReturned = null;
    int type;

    // Enumerator 是 “迭代器 (Iterator)” 还是 “枚举类 (Enumeration)”的标志
    // iterator 为 true，表示它是迭代器；否则，是枚举类。
    boolean iterator;

    // 在将 Enumerator 当作迭代器使用时会用到，用来实现 fail-fast 机制。
    protected int expectedModCount = modCount;

    Enumerator(int type, boolean iterator) {
        this.type = type;
        this.iterator = iterator;
    }

    // 从遍历 table 的数组的末尾向前查找，直到找到不为 null 的 Entry。
    public boolean hasMoreElements() {
        Entry<K,V> e = entry;
        int i = index;
        Entry[] t = table;
        /* Use locals for faster loop iteration */
        while (e == null && i > 0) {
            e = t[--i];
        }
        entry = e;
        index = i;
        return e != null;
    }

    // 获取下一个元素
    // 注意：从 hasMoreElements() 和 nextElement() 可以看出“Hashtable 的 elements() 遍历方式
    // 首先，从后向前的遍历 table 数组。table 数组的每个节点都是一个单向链表 (Entry)。
    // 然后，依次向后遍历单向链表 Entry。
    public T nextElement() {
        Entry<K,V> et = entry;
        int i = index;
        Entry[] t = table;
        /* Use locals for faster loop iteration */
        while (et == null && i > 0) {
            et = t[--i];
        }
        entry = et;
        index = i;
        if (et != null) {
            Entry<K,V> e = lastReturned = entry;
            entry = e.next;
            return type == KEYS ? (T)e.key : (type == VALUES ? (T)e.value : (T)e);
        }
        throw new NoSuchElementException("Hashtable Enumerator");
    }

    // 迭代器 Iterator 的判断是否存在下一个元素
```

```
    // 实际上，它是调用的 hasMoreElements()
    public boolean hasNext() {
        return hasMoreElements();
    }

    // 迭代器获取下一个元素
    // 实际上，它是调用的 nextElement()
    public T next() {
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        return nextElement();
    }

    // 迭代器的 remove() 接口。
    // 首先，它在 table 数组中找出要删除元素所在的 Entry，
    // 然后，删除单向链表 Entry 中的元素。
    public void remove() {
        if (!iterator)
            throw new UnsupportedOperationException();
        if (lastReturned == null)
            throw new IllegalStateException("Hashtable Enumerator");
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();

        synchronized(Hashtable.this) {
            Entry[] tab = Hashtable.this.table;
            int index = (lastReturned.hash & 0x7FFFFFFF) % tab.length;

            for (Entry<K,V> e = tab[index], prev = null; e != null;
                 prev = e, e = e.next) {
                if (e == lastReturned) {
                    modCount++;
                    expectedModCount++;
                    if (prev == null)
                        tab[index] = e.next;
                    else
                        prev.next = e.next;
                    count--;
                    lastReturned = null;
                    return;
                }
            }
            throw new ConcurrentModificationException();
        }
    }
}


private static Enumeration emptyEnumerator = new EmptyEnumerator();
private static Iterator emptyIterator = new EmptyIterator();

// 空枚举类
// 当 Hashtable 的实际大小为 0；此时，又要通过 Enumeration 遍历 Hashtable 时，返回的是“空枚举
private static class EmptyEnumerator implements Enumeration<Object> {

    EmptyEnumerator() {
    }
```

```
        // 空枚举类的 hasMoreElements() 始终返回 false
        public boolean hasMoreElements() {
            return false;
        }

        // 空枚举类的 nextElement() 抛出异常
        public Object nextElement() {
            throw new NoSuchElementException("Hashtable Enumerator");
        }
    }


    // 空迭代器
    // 当 Hashtable 的实际大小为 0; 此时，又要通过迭代器遍历 Hashtable 时，返回的是“空迭代器”的对
    private static class EmptyIterator implements Iterator<Object> {

        EmptyIterator() {
        }

        public boolean hasNext() {
            return false;
        }

        public Object next() {
            throw new NoSuchElementException("Hashtable Iterator");
        }

        public void remove() {
            throw new IllegalStateException("Hashtable Iterator");
        }

    }
}
```

4.9.2 几点总结

- 针对 Hashtable，我们同样给出几点比较重要的总结，但要结合与 HashMap 的比较来总结。
  - 二者的存储结构和解决冲突的方法都是相同的。
  - HashTable 在不指定容量的情况下的默认容量为 11，而 HashMap 为 16，Hashtable 不要求底层数组的容量一定要为 2 的整数次幂，而 HashMap 则要求一定为 2 的整数次幂。
  - Hashtable 中 key 和 value 都不允许为 null，而 HashMap 中 key 和 value 都允许为 null（key 只能有一个为 null，而 value 则可以有多个为 null）。但是如果在 Hashtable 中有类似 put(null,null) 的操作，编译同样可以通过，因为 key 和 value 都是 Object 类型，但运行时会抛出 NullPointerException 异常，这是 JDK 的规范规定的。我们来看下 ContainsKey 方法和 ContainsValue 的源码：

```
// 判断 Hashtable 是否包含“值 (value)”
 public synchronized boolean contains(Object value) {
     //注意，Hashtable 中的 value 不能是 null，
     // 若是 null 的话，抛出异常！
     if (value == null) {
         throw new NullPointerException();
     }

     // 从后向前遍历 table 数组中的元素 (Entry)
     // 对于每个 Entry(单向链表)，逐个遍历，判断节点的值是否等于 value
     Entry tab[] = table;
     for (int i = tab.length ; i-- > 0 ;) {
         for (Entry<K,V> e = tab[i] ; e != null ; e = e.next) {
```

```
            if (e.value.equals(value)) {
                return true;
            }
        }
    }
    return false;
}

public boolean containsValue(Object value) {
    return contains(value);
}

// 判断 Hashtable 是否包含 key
public synchronized boolean containsKey(Object key) {
    Entry tab[] = table;
/计算 hash 值, 直接用 key 的 hashCode 代替
    int hash = hash(key);
    // 计算在数组中的索引值
    int index = (hash & 0x7FFFFFFF) % tab.length;
    // 找到“key 对应的 Entry(链表)”，然后在链表中找出“哈希值”和“键值”与 key 都相等的元素
    for (Entry<K,V> e = tab[index] ; e != null ; e = e.next) {
        if ((e.hash == hash) && e.key.equals(key)) {
            return true;
        }
    }
    return false;
}
```

- 很明显，如果 value 为 null，会直接抛出 NullPointerException 异常，但源码中并没有对 key 是否为 null 判断，有点小不解！不过 NullPointerException 属于 RuntimeException 异常，是可以由 JVM 自动抛出的，也许对 key 的值在 JVM 中有所限制吧。4. Hashtable 扩容时，将容量变为原来的 2 倍加 1，而 HashMap 扩容时，将容量变为原来的 2 倍。5. Hashtable 和 HashMap 都重新计算了 key 的 hash 值，Hashtable 在求 hash 值对应的位置索引时，用取模运算，而 HashMap 在求位置索引时，则用与运算，且这里一般先用 hash&0x7FFFFFFF 后，再对 length 取模，&0x7FFFFFFF 的目的是为了将负的 hash 值转化为正值，因为 hash 值有可能为负数，而 &0x7FFFFFFF 后，只有符号外改变，而后面的位都不变。

4.10 LinkedHashMap 源码剖析

- LinkedHashMap 是 HashMap 的子类，与 HashMap 有着同样的存储结构，但它加入了一个双向链表的头结点，将所有 put 到 LinkedHashmap 的节点一一串成了一个双向循环链表，因此它保留了节点插入的顺序，可以使节点的输出顺序与输入顺序相同。
- LinkedHashMap 可以用来实现 LRU 算法（这会在下面的源码中进行分析）。
- LinkedHashMap 同样是非线程安全的，只在单线程环境下使用。

4.10.1 LinkedHashMap 源码剖析

- LinkedHashMap 源码如下（加入了详细的注释）：

```
package java.util;
import java.io.*;

public class LinkedHashMap<K,V>
    extends HashMap<K,V>
    implements Map<K,V>   {

    private static final long serialVersionUID = 3801124242820219131L;

    //双向循环链表的头结点，整个 LinkedHashMap 中只有一个 header，
```

```
//它将哈希表中所有的 Entry 贯穿起来，header 中不保存 key-value 对，只保存前后节点的引用
private transient Entry<K,V> header;

//双向链表中元素排序规则的标志位。
//accessOrder 为 false，表示按插入顺序排序
//accessOrder 为 true，表示按访问顺序排序
private final boolean accessOrder;

//调用 HashMap 的构造方法来构造底层的数组
public LinkedHashMap(int initialCapacity, float loadFactor) {
    super(initialCapacity, loadFactor);
    accessOrder = false;    //链表中的元素默认按照插入顺序排序
}

//加载因子取默认的 0.75f
public LinkedHashMap(int initialCapacity) {
    super(initialCapacity);
    accessOrder = false;
}

//加载因子取默认的 0.75f，容量取默认的 16
public LinkedHashMap() {
    super();
    accessOrder = false;
}

//含有子 Map 的构造方法，同样调用 HashMap 的对应的构造方法
public LinkedHashMap(Map<? extends K, ? extends V> m) {
    super(m);
    accessOrder = false;
}

//该构造方法可以指定链表中的元素排序的规则
public LinkedHashMap(int initialCapacity,float loadFactor,boolean accessOrder) {
    super(initialCapacity, loadFactor);
    this.accessOrder = accessOrder;
}

//覆写父类的 init() 方法（HashMap 中的 init 方法为空），
//该方法在父类的构造方法和 Clone、readObject 中在插入元素前被调用，
//初始化一个空的双向循环链表，头结点中不保存数据，头结点的下一个节点才开始保存数据。
void init() {
    header = new Entry<K,V>(-1, null, null, null);
    header.before = header.after = header;
}


//覆写 HashMap 中的 transfer 方法，它在父类的 resize 方法中被调用，
//扩容后，将 key-value 对重新映射到新的 newTable 中
//覆写该方法的目的是为了提高复制的效率，
//这里充分利用双向循环链表的特点进行迭代，不用对底层的数组进行 for 循环。
void transfer(HashMap.Entry[] newTable) {
    int newCapacity = newTable.length;
    for (Entry<K,V> e = header.after; e != header; e = e.after) {
        int index = indexFor(e.hash, newCapacity);
        e.next = newTable[index];
        newTable[index] = e;
    }
```

```
}


//覆写 HashMap 中的 containsValue 方法，
//覆写该方法的目的同样是为了提高查询的效率，
//利用双向循环链表的特点进行查询，少了对数组的外层 for 循环
public boolean containsValue(Object value) {
    // Overridden to take advantage of faster iterator
    if (value==null) {
        for (Entry e = header.after; e != header; e = e.after)
            if (e.value==null)
                return true;
    } else {
        for (Entry e = header.after; e != header; e = e.after)
            if (value.equals(e.value))
                return true;
    }
    return false;
}

//覆写 HashMap 中的 get 方法，通过 getEntry 方法获取 Entry 对象。
//注意这里的 recordAccess 方法，
//如果链表中元素的排序规则是按照插入的先后顺序排序的话，该方法什么也不做，
//如果链表中元素的排序规则是按照访问的先后顺序排序的话，则将 e 移到链表的末尾处。
public V get(Object key) {
    Entry<K,V> e = (Entry<K,V>)getEntry(key);
    if (e == null)
        return null;
    e.recordAccess(this);
    return e.value;
}

//清空 HashMap，并将双向链表还原为只有头结点的空链表
public void clear() {
    super.clear();
    header.before = header.after = header;
}

//Enty 的数据结构，多了两个指向前后节点的引用
private static class Entry<K,V> extends HashMap.Entry<K,V> {
    // These fields comprise the doubly linked list used for iteration.
    Entry<K,V> before, after;

    //调用父类的构造方法
    Entry(int hash, K key, V value, HashMap.Entry<K,V> next) {
        super(hash, key, value, next);
    }

    //双向循环链表中，删除当前的 Entry
    private void remove() {
        before.after = after;
        after.before = before;
    }

    //双向循环立链表中，将当前的 Entry 插入到 existingEntry 的前面
    private void addBefore(Entry<K,V> existingEntry) {
        after  = existingEntry;
        before = existingEntry.before;
```

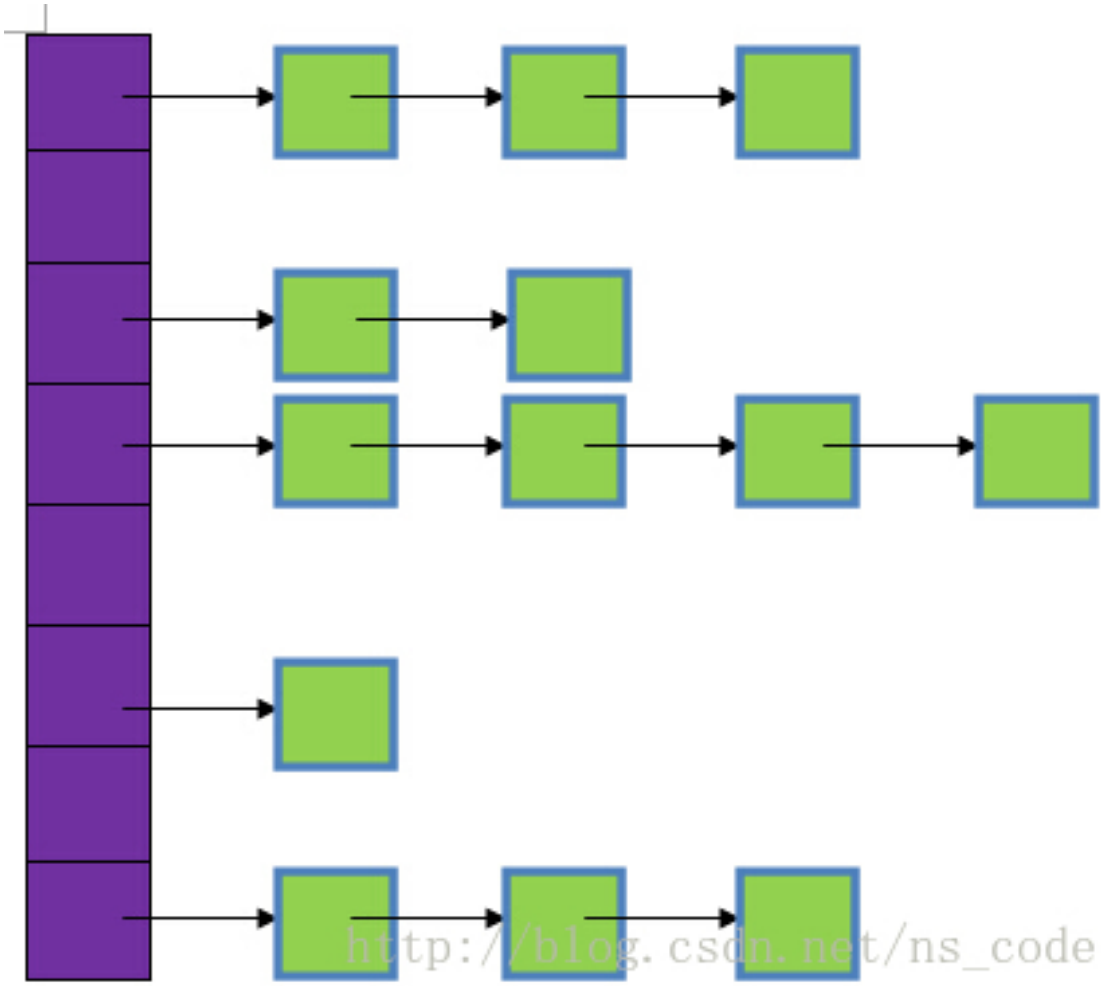
```
            before.after = this;
            after.before = this;
        }

        //覆写 HashMap 中的 recordAccess 方法 (HashMap 中该方法为空),
        //当调用父类的 put 方法，在发现插入的 key 已经存在时，会调用该方法，
        //调用 LinkedHashmap 覆写的 get 方法时，也会调用到该方法，
        //该方法提供了 LRU 算法的实现，它将最近使用的 Entry 放到双向循环链表的尾部，
        //accessOrder 为 true 时，get 方法会调用 recordAccess 方法
        //put 方法在覆盖 key-value 对时也会调用 recordAccess 方法
        //它们导致 Entry 最近使用，因此将其移到双向链表的末尾
        void recordAccess(HashMap<K,V> m) {
            LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;
            //如果链表中元素按照访问顺序排序，则将当前访问的 Entry 移到双向循环链表的尾部，
            //如果是按照插入的先后顺序排序，则不做任何事情。
            if (lm.accessOrder) {
                lm.modCount++;
                //移除当前访问的 Entry
                remove();
                //将当前访问的 Entry 插入到链表的尾部
                addBefore(lm.header);
            }
        }

        void recordRemoval(HashMap<K,V> m) {
            remove();
        }
    }

    //迭代器
    private abstract class LinkedHashIterator<T> implements Iterator<T> {
        Entry<K,V> nextEntry    = header.after;
        Entry<K,V> lastReturned = null;

        /**
         * The modCount value that the iterator believes that the backing
         * List should have.  If this expectation is violated, the iterator
         * has detected concurrent modification.
         */
        int expectedModCount = modCount;

        public boolean hasNext() {
            return nextEntry != header;
        }

        public void remove() {
            if (lastReturned == null)
                throw new IllegalStateException();
            if (modCount != expectedModCount)
                throw new ConcurrentModificationException();

            LinkedHashMap.this.remove(lastReturned.key);
            lastReturned = null;
            expectedModCount = modCount;
        }

        //从 head 的下一个节点开始迭代
        Entry<K,V> nextEntry() {
```

```
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        if (nextEntry == header)
            throw new NoSuchElementException();

        Entry<K,V> e = lastReturned = nextEntry;
        nextEntry = e.after;
        return e;
    }
}

//key 迭代器
private class KeyIterator extends LinkedHashIterator<K> {
    public K next() { return nextEntry().getKey(); }
}

//value 迭代器
private class ValueIterator extends LinkedHashIterator<V> {
    public V next() { return nextEntry().value; }
}

//Entry 迭代器
private class EntryIterator extends LinkedHashIterator<Map.Entry<K,V>> {
    public Map.Entry<K,V> next() { return nextEntry(); }
}

// These Overrides alter the behavior of superclass view iterator() methods
Iterator<K> newKeyIterator()   { return new KeyIterator();   }
Iterator<V> newValueIterator() { return new ValueIterator(); }
Iterator<Map.Entry<K,V>> newEntryIterator() { return new EntryIterator(); }


//覆写 HashMap 中的 addEntry 方法，LinkedHashmap 并没有覆写 HashMap 中的 put 方法，
//而是覆写了 put 方法所调用的 addEntry 方法和 recordAccess 方法，
//put 方法在插入的 key 已存在的情况下，会调用 recordAccess 方法，
//在插入的 key 不存在的情况下，要调用 addEntry 插入新的 Entry
void addEntry(int hash, K key, V value, int bucketIndex) {
    //创建新的 Entry，并插入到 LinkedHashMap 中
    createEntry(hash, key, value, bucketIndex);

    //双向链表的第一个有效节点（header 后的那个节点）为近期最少使用的节点
    Entry<K,V> eldest = header.after;
    //如果有必要，则删除掉该近期最少使用的节点，
    //这要看对 removeEldestEntry 的覆写，由于默认为 false，因此默认是不做任何处理的。
    if (removeEldestEntry(eldest)) {
        removeEntryForKey(eldest.key);
    } else {
        //扩容到原来的 2 倍
        if (size >= threshold)
            resize(2 * table.length);
    }
}

void createEntry(int hash, K key, V value, int bucketIndex) {
    //创建新的 Entry，并将其插入到数组对应槽的单链表的头结点处，这点与 HashMap 中相同
    HashMap.Entry<K,V> old = table[bucketIndex];
    Entry<K,V> e = new Entry<K,V>(hash, key, value, old);
    table[bucketIndex] = e;
```

```
        //每次插入 Entry 时，都将其移到双向链表的尾部，
        //这便会按照 Entry 插入 LinkedHashMap 的先后顺序来迭代元素，
        //同时，新 put 进来的 Entry 是最近访问的 Entry，把其放在链表末尾 ，符合 LRU 算法的实现
        e.addBefore(header);
        size++;
    }

    //该方法是用来被覆写的，一般如果用 LinkedHashmap 实现 LRU 算法，就要覆写该方法，
    //比如可以将该方法覆写为如果设定的内存已满，则返回 true，这样当再次向 LinkedHashMap 中 put
    //Entry 时，在调用的 addEntry 方法中便会将近期最少使用的节点删除掉（header 后的那个节点）。
    protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
        return false;
    }
}
```

4.10.2 几点总结

- 关于 LinkedHashMap 的源码，给出以下几点比较重要的总结：
  - 1、从源码中可以看出，LinkedHashMap 中加入了一个 head 头结点，将所有插入到该 LinkedHashMap 中的 Entry 按照插入的先后顺序依次加入到以 head 为头结点的双向循环链表的尾部。



  - 1、实际上就是 HashMap 和 LinkedList 两个集合类的存储结构的结合。在 LinkedHashMapMap 中，所有 put 进来的 Entry 都保存在如第一个图所示的哈希表中，但它又额外定义了一个以 head 为头结点的空的双向循环链表，每次 put 进来 Entry，除了将其保存到对哈希表中对应的位置上外，还要将其插入到双向循环链表的尾部。
  - 2、LinkedHashMap 由于继承自 HashMap，因此它具有 HashMap 的所有特性，同样允许 key 和 value 为 null。

- 3、注意源码中的 accessOrder 标志位，当它 false 时，表示双向链表中的元素按照 Entry 插入 LinkedHashMap 到中的先后顺序排序，即每次 put 到 LinkedHashMap 中的 Entry 都放在双向链表的尾部，这样遍历双向链表时，Entry 的输出顺序便和插入的顺序一致，这也是默认的双向链表的存储顺序；当它为 true 时，表示双向链表中的元素按照访问的先后顺序排列，可以看到，虽然 Entry 插入链表的顺序依然是按照其 put 到 LinkedHashMap 中的顺序，但 put 和 get 方法均有调用 recordAccess 方法（put 方法在 key 相同，覆盖原有的 Entry 的情况下调用 recordAccess 方法），该方法判断 accessOrder 是否为 true，如果是，则将当前访问的 Entry（put 进来的 Entry 或 get 出来的 Entry）移到双向链表的尾部（key 不相同时，put 新 Entry 时，会调用 addEntry，它会调用 creatEntry，该方法同样将新插入的元素放入到双向链表的尾部，既符合插入的先后顺序，又符合访问的先后顺序，因为这时该 Entry 也被访问了），否则，什么也不做。
- 4、注意构造方法，前四个构造方法都将 accessOrder 设为 false，说明默认是按照插入顺序排序的，而第五个构造方法可以自定义传入的 accessOrder 的值，因此可以指定双向循环链表中元素的排序规则，一般要用 LinkedHashMap 实现 LRU 算法，就要用该构造方法，将 accessOrder 置为 true。
- 5、LinkedHashMap 并没有覆写 HashMap 中的 put 方法，而是覆写了 put 方法中调用的 addEntry 方法和 recordAccess 方法，我们回过头来再看下 HashMap 的 put 方法：

```
// 将“key-value”添加到 HashMap 中
public V put(K key, V value) {
    // 若“key 为 null”，则将该键值对添加到 table[0] 中。
    if (key == null)
        return putForNullKey(value);
    // 若“key 不为 null”，则计算该 key 的哈希值，然后将其添加到该哈希值对应的链表中。
    int hash = hash(key.hashCode());
    int i = indexFor(hash, table.length);
    for (Entry<K,V> e = table[i]; e != null; e = e.next) {
        Object k;
        // 若“该 key”对应的键值对已经存在，则用新的 value 取代旧的 value。然后退出！
        if (e.hash == hash && ((k = e.key) == key || key.equals(k))) {
            V oldValue = e.value;
            e.value = value;
            e.recordAccess(this);
            return oldValue;
        }
    }

    // 若“该 key”对应的键值对不存在，则将“key-value”添加到 table 中
    modCount++;
    //将 key-value 添加到 table[i] 处
    addEntry(hash, key, value, i);
    return null;
}
```

- 当要 put 进来的 Entry 的 key 在哈希表中已经在存在时，会调用 recordAccess 方法，当该 key 不存在时，则会调用 addEntry 方法将新的 Entry 插入到对应槽的单链表的头部。
- 我们先来看 recordAccess 方法：

```
//覆写 HashMap 中的 recordAccess 方法 (HashMap 中该方法为空),
//当调用父类的 put 方法，在发现插入的 key 已经存在时，会调用该方法，
//调用 LinkedHashmap 覆写的 get 方法时，也会调用到该方法，
//该方法提供了 LRU 算法的实现，它将最近使用的 Entry 放到双向循环链表的尾部，
//accessOrder 为 true 时，get 方法会调用 recordAccess 方法
//put 方法在覆盖 key-value 对时也会调用 recordAccess 方法
//它们导致 Entry 最近使用，因此将其移到双向链表的末尾
      void recordAccess(HashMap<K,V> m) {
          LinkedHashMap<K,V> lm = (LinkedHashMap<K,V>)m;
    //如果链表中元素按照访问顺序排序，则将当前访问的 Entry 移到双向循环链表的尾部，
    //如果是按照插入的先后顺序排序，则不做任何事情。
          if (lm.accessOrder) {
```

```
            lm.modCount++;
        //移除当前访问的 Entry
            remove();
        //将当前访问的 Entry 插入到链表的尾部
            addBefore(lm.header);
        }
    }
```

- 该方法会判断 accessOrder 是否为 true，如果为 true，它会将当前访问的 Entry（在这里指 put 进来的 Entry）移动到双向循环链表的尾部，从而实现双向链表中的元素按照访问顺序来排序（最近访问的 Entry 放到链表的最后，这样多次下来，前面就是最近没有被访问的元素，在实现、LRU 算法时，当双向链表中的节点数达到最大值时，将前面的元素删去即可，因为前面的元素是最近最少使用的），否则什么也不做。

- 再来看 addEntry 方法：

```
//覆写 HashMap 中的 addEntry 方法，LinkedHashmap 并没有覆写 HashMap 中的 put 方法，
//而是覆写了 put 方法所调用的 addEntry 方法和 recordAccess 方法，
//put 方法在插入的 key 已存在的情况下，会调用 recordAccess 方法，
//在插入的 key 不存在的情况下，要调用 addEntry 插入新的 Entry
   void addEntry(int hash, K key, V value, int bucketIndex) {
    //创建新的 Entry，并插入到 LinkedHashMap 中
       createEntry(hash, key, value, bucketIndex);

       //双向链表的第一个有效节点（header 后的那个节点）为近期最少使用的节点
       Entry<K,V> eldest = header.after;
    //如果有必要，则删除掉该近期最少使用的节点，
    //这要看对 removeEldestEntry 的覆写，由于默认为 false，因此默认是不做任何处理的。
       if (removeEldestEntry(eldest)) {
           removeEntryForKey(eldest.key);
       } else {
        //扩容到原来的 2 倍
           if (size >= threshold)
               resize(2 * table.length);
       }
   }

   void createEntry(int hash, K key, V value, int bucketIndex) {
    //创建新的 Entry，并将其插入到数组对应槽的单链表的头结点处，这点与 HashMap 中相同
       HashMap.Entry<K,V> old = table[bucketIndex];
    Entry<K,V> e = new Entry<K,V>(hash, key, value, old);
       table[bucketIndex] = e;
    //每次插入 Entry 时，都将其移到双向链表的尾部，
    //这便会按照 Entry 插入 LinkedHashMap 的先后顺序来迭代元素，
    //同时，新 put 进来的 Entry 是最近访问的 Entry，把其放在链表末尾 ，符合 LRU 算法的实现
       e.addBefore(header);
       size++;
   }
```

- 同样是将新的 Entry 插入到 table 中对应槽所对应单链表的头结点中，但可以看出，在 createEntry 中，同样把新 put 进来的 Entry 插入到了双向链表的尾部，从插入顺序的层面来说，新的 Entry 插入到双向链表的尾部，可以实现按照插入的先后顺序来迭代 Entry，而从访问顺序的层面来说，新 put 进来的 Entry 又是最近访问的 Entry，也应该将其放在双向链表的尾部。

- 上面还有个 removeEldestEntry 方法，该方法如下：

```
    //该方法是用来被覆写的，一般如果用 LinkedHashmap 实现 LRU 算法，就要覆写该方法，
    //比如可以将该方法覆写为如果设定的内存已满，则返回 true，这样当再次向 LinkedHashMap 中 put
    //Entry 时，在调用的 addEntry 方法中便会将近期最少使用的节点删除掉（header 后的那个节点）。
    protected boolean removeEldestEntry(Map.Entry<K,V> eldest) {
        return false;
```

```
    }
}
```

- 该方法默认返回 false，我们一般在用 LinkedHashMap 实现 LRU 算法时，要覆写该方法，一般的实现是，当设定的内存（这里指节点个数）达到最大值时，返回 true，这样 put 新的 Entry（该 Entry 的 key 在哈希表中没有已经存在）时，就会调用 removeEntryForKey 方法，将最近最少使用的节点删除（head 后面的那个节点，实际上是最近没有使用）。

- 6、LinkedHashMap 覆写了 HashMap 的 get 方法：

```
//覆写 HashMap 中的 get 方法，通过 getEntry 方法获取 Entry 对象。
//注意这里的 recordAccess 方法，
//如果链表中元素的排序规则是按照插入的先后顺序排序的话，该方法什么也不做，
//如果链表中元素的排序规则是按照访问的先后顺序排序的话，则将 e 移到链表的末尾处。
   public V get(Object key) {
       Entry<K,V> e = (Entry<K,V>)getEntry(key);
       if (e == null)
           return null;
       e.recordAccess(this);
       return e.value;
   }
```

- 先取得 Entry，如果不为 null，一样调用 recordAccess 方法，上面已经说得很清楚，这里不在多解释了。

- 7、最后说说 LinkedHashMap 是如何实现 LRU 的。首先，当 accessOrder 为 true 时，才会开启按访问顺序排序的模式，才能用来实现 LRU 算法。我们可以看到，无论是 put 方法还是 get 方法，都会导致目标 Entry 成为最近访问的 Entry，因此便把该 Entry 加入到了双向链表的末尾（get 方法通过调用 recordAccess 方法来实现，put 方法在覆盖已有 key 的情况下，也是通过调用 recordAccess 方法来实现，在插入新的 Entry 时，则是通过 createEntry 中的 addBefore 方法来实现），这样便把最近使用了的 Entry 放入到了双向链表的后面，多次操作后，双向链表前面的 Entry 便是最近没有使用的，这样当节点个数满的时候，删除的最前面的 Entry(head 后面的那个 Entry) 便是最近最少使用的 Entry。