Android 开发工程师面试指南

yumufeng

JavaSE(Java基础)

Java基础知识

J2SE

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。

== 和 Equals 的区别
1. == 是一个运算符。
2.Equals则是string对象的方法，可以.（点）出来。

我们比较无非就是这两种 1、基本数据类型比较 2、引用对象比较
1、基本数据类型比较
==和Equals都比较两个值是否相等。相等为true 否则为false；

2、引用对象比较
==和Equals都是比较栈内存中的地址是否相等 。相等为true 否则为false；

详解 http://www.importnew.com/6804.html

主要区别在于前者是方法后者是操作符。“==”的行为对于每个对象来说与equals()是完全相同的，但是equals()可以基于业务规则的不同而重写（overridden）。“==”习惯用于原生（primitive）类型之间的比较，而equals()仅用于对象之间的比较。

==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。

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.hash- Code()==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:

1. 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。
2. 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#中国就从语法规定不能做这样的事。

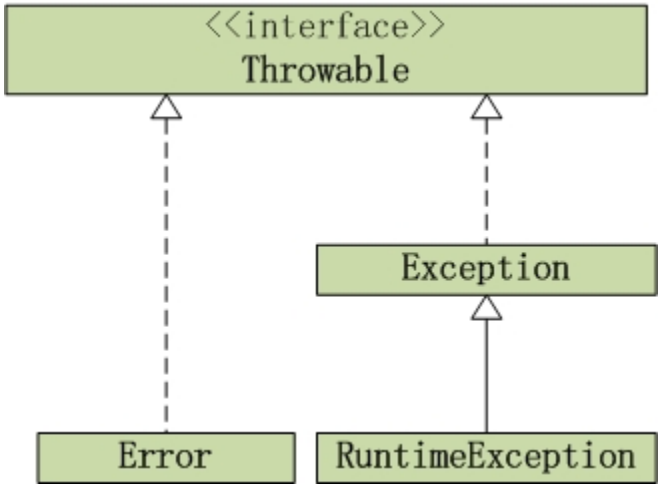
Excption与Error区别

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



1. 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。

2. 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。

3. 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！

4. 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：

1. 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。

2. 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。

3. 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。

4. 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/

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。

Java中的内存泄漏

Java中的内存泄漏

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。

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、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。

例如

```
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、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。

例如：

```
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、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、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、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、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

String源码分析

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。

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。

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;
```

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value
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);
}
```

三 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-被比较对象不相等字符）
    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。

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。

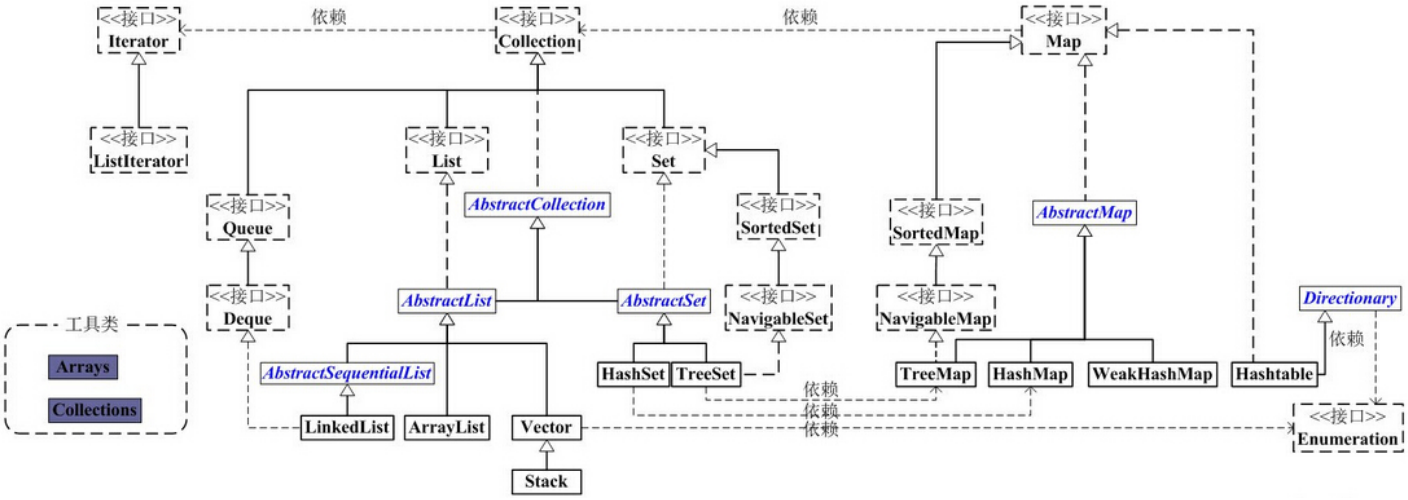
equals字符串值比较：比较两个引用所指对象字面值是否相等。

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。

Java集合结构

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。

ArrayList源码剖析

ArrayList简介

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。

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);
        }
    }


    // 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”全部拷贝到新数组中
        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(
```

```
            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);
```

```
                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();
    }
}
```

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。

1. 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。

HashMap源码剖析

HashMap简介

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。

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的容量加倍。
        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的容量加倍。
        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()计算哈希值
        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”
```

```
    // 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(); i.hasNex
t(); ) {
            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(); i.hasNex
t(); ) {
            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;
```

```
        Entry<K,V> prev = e;

        // 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), hashCo
de()这些函数
    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(EntryIterator)”
3个子类。
    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的元素就退出循环。
                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() {
```

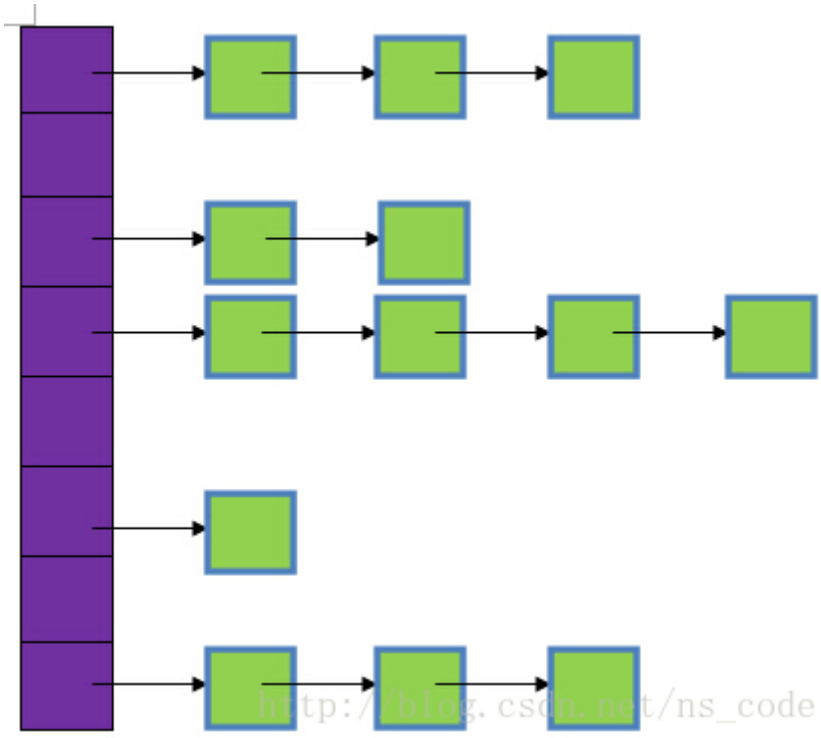
```
            HashMap.this.clear();
        }
    }

    // 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;   }
}
```

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。
// 它实现了Map.Entry 接口，即实现getKey(), getValue(), setValue(V value), equals(Object o), hashCode(
)这些函数
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：

```
// 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;
}
```

首先，如果key为null，则直接从哈希表的第一个位置table[0]对应的链表上查找。记住，key为null的键值对永远都放在以table[0]为头结点的链表中，当然不一定是存放在头结点table[0]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[0]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
    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。

Hashtable简介

Hashtable简介

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。

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);
    }

    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 = key.hashCode();
        // 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 = 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)) {
                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 = key.hashCode();
        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
    // 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(0)
        // 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 = key.hashCode();
            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(0)
        // 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 = key.hashCode();
            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，实现多线程同步
    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];
```

```
            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) {
```

```
        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()遍历Hashtable的接口” 。
    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“空枚举类”的对象。
    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”的对象。

    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");
        }

    }
}
```

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。

1. 二者的存储结构和解决冲突的方法都是相同的。
2. 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。
3. 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 = 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;
 }
```

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。

1. 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。
2. Hashtable计算hash值，直接用key的hashCode()，而HashMap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。

Vector源码剖析

Vector简介

Vector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。

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
```

```
    // 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();
        }
    }

    // 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();
    }
}
```

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。

LinkedHashMap简介

LinkedHashMap简介

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。

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> {
```

```
    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;

    /**
```

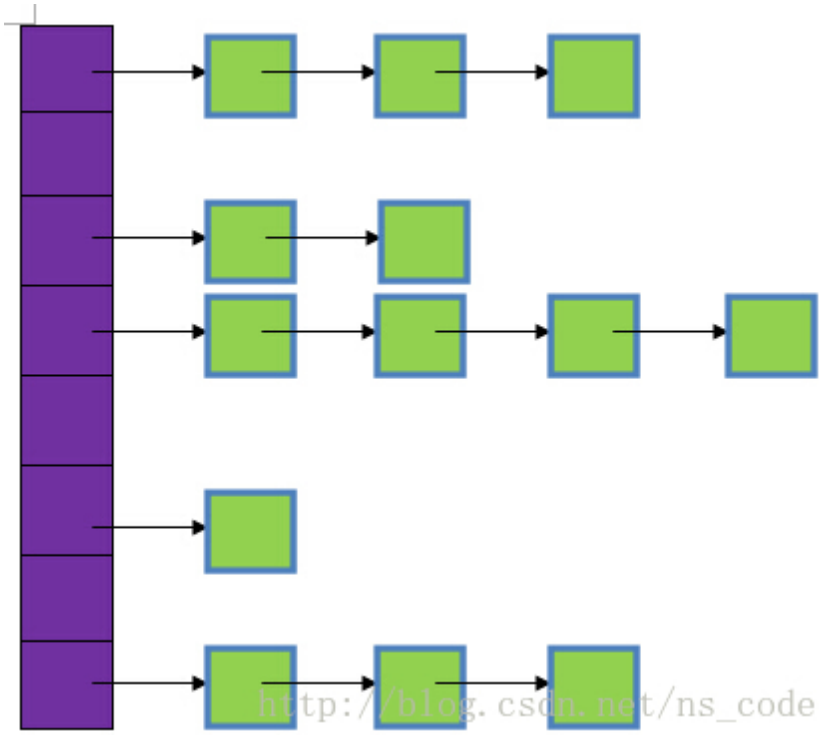
```
     * The modCount value that the iterator believes that the backing
     * List should have.  If this expectation is violated, the iterator
     * has detected concurrent modification.
     */
    int expectedModCount = modCount;

    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;
    }
}
```

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。

LinkedList简介

LinkedList简介

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。

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();
}
```

```
    }

    // 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);
```

```
            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;
```

```
        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。
```

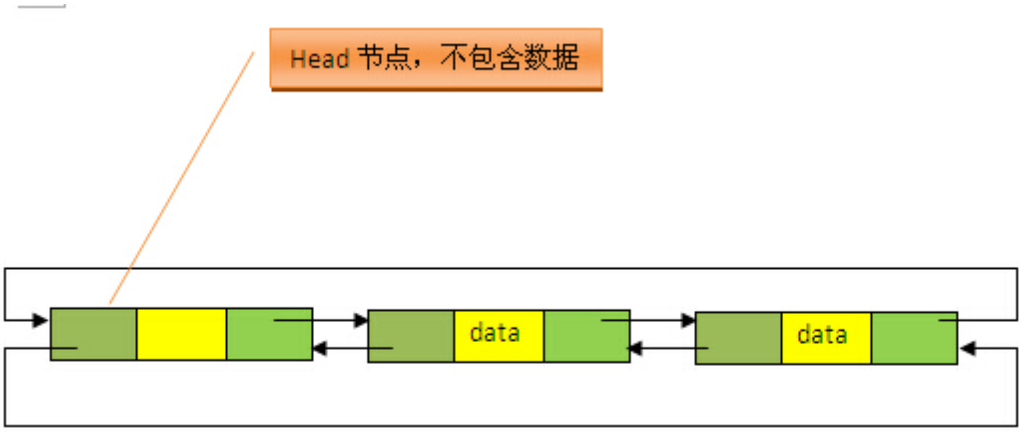
```
        // 实际上是判断双向链表的当前节点是否达到开头
        public boolean hasNext() {
            return itr.hasPrevious();
        }
        // 反向迭代器获取下一个元素。
        // 实际上是获取双向链表的前一个节点
        public E next() {
            return itr.previous();
        }
        // 删除当前节点
        public void remove() {
            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();
```

```
            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);
    }

}
```

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indexsize/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。

JVM(Java虚拟机)

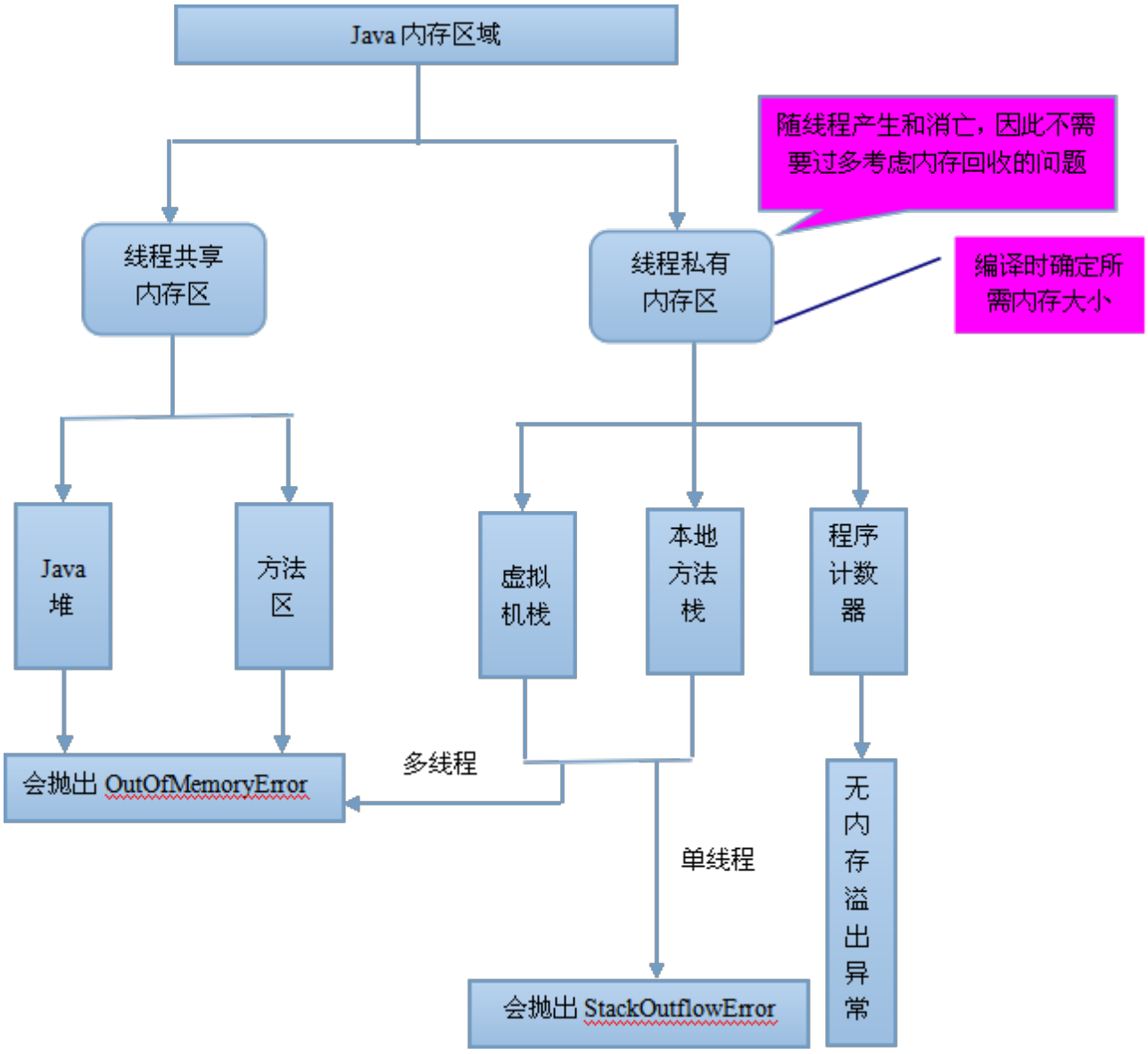
JVM基础知识

JVM

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。



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。

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。

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。

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。

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。

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。

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。

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

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。

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。

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：①.标记和清除

过程效率不高 。②.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。

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。

类加载的五个过程：加载、验证、准备、解析、初始化。

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：

1. 通过一个类的全限定名来获取定义此类的二进制字节流。
2. 将这个字节流所代表的静态存储结构转化为方法区的运行时数据结构。
3. 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。

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

1. 启动类加载器，负责将存放在\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。
2. 扩展类加载器：负责加载\lib\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。
3. 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。

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

JVM类加载机制

虚拟机类加载机制

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)。

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。

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

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

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。

类加载器

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"。

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：

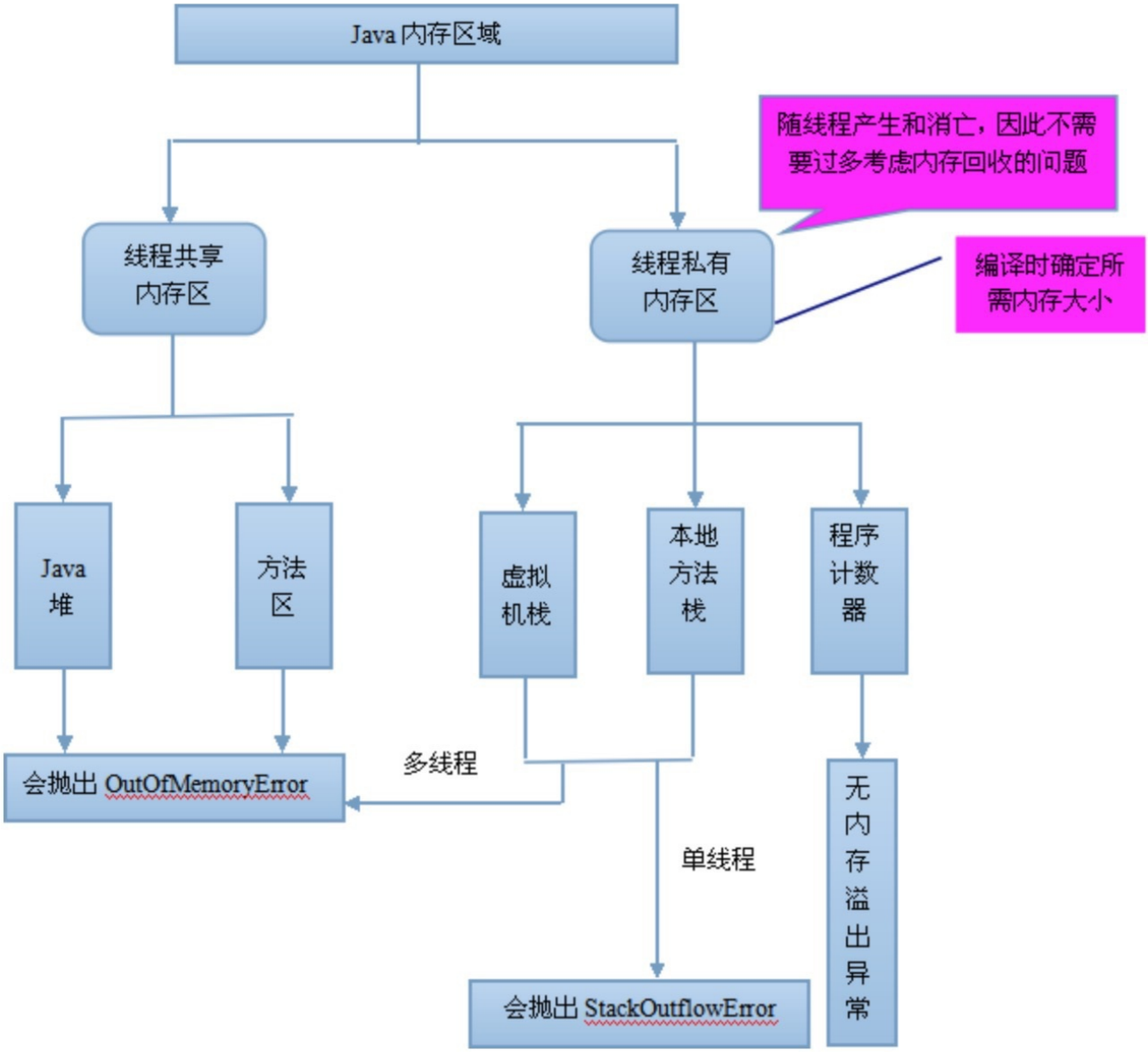
- 启动类加载器
- 扩展类加载器

- 应用程序类加载器

Java内存区域与内存溢出

内存区域

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。



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。

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：

1. 如果线程请求的栈深度大于虚拟机所允许的深度，将抛出StackOverflowError异常。
2. 如果虚拟机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。

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。

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。

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。

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。

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。

内存溢出

下面给出个内存区域内存溢出的简单测试方法

<table>
<tr><td>内存区域</td><td colspan="2">内存溢出的测试方法</td></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。

对象实例化分析

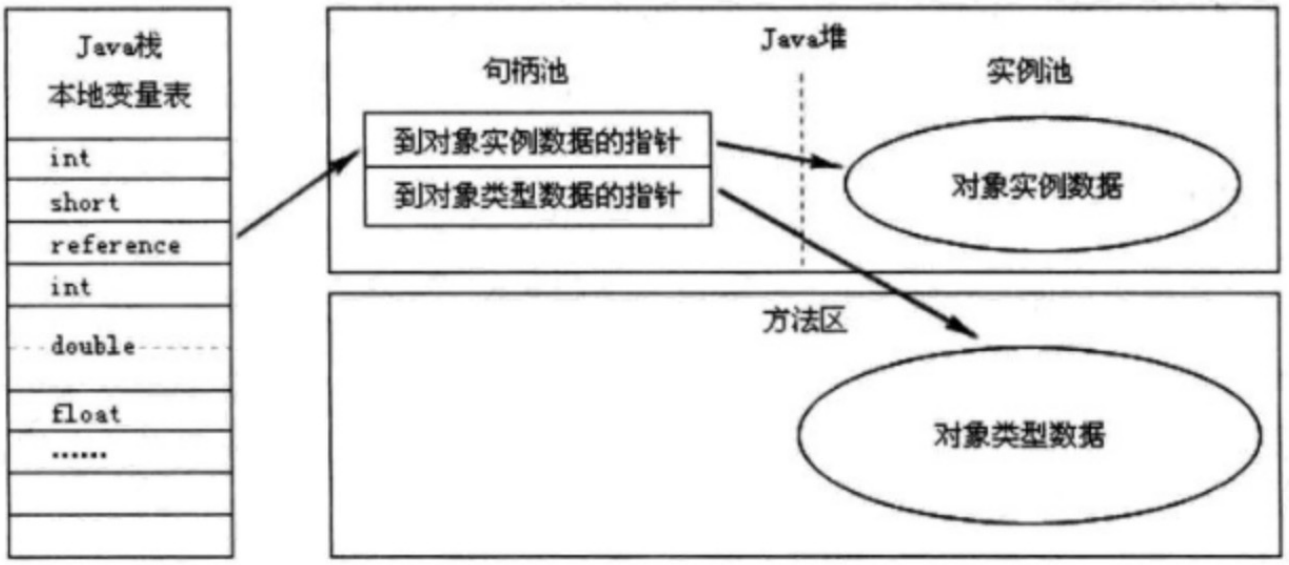
对内存分配情况分析最常见的示例便是对象实例化：

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

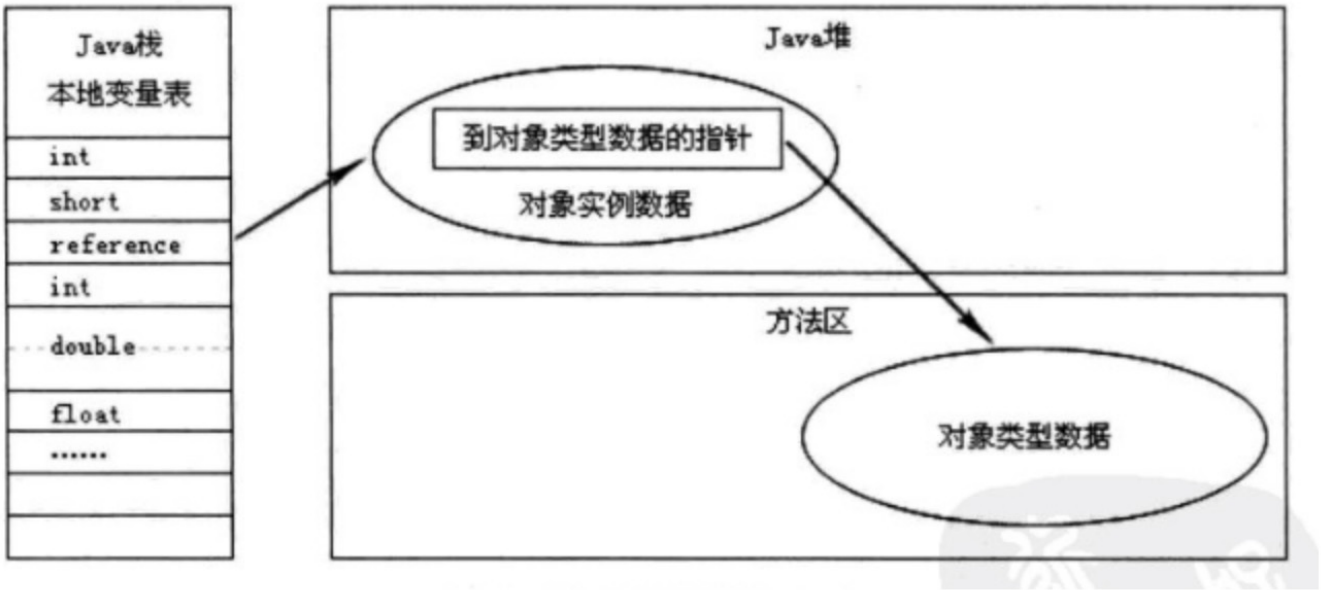
这段代码的执行会涉及java栈、Java堆、方法区三个最重要的内存区域。假设该语句出现在方法体中，及时对JVM虚拟机不了解的Java使用这，应该也知道obj会作为引用类型（reference）的数据保存在Java栈的本地变量表中，而会在Java堆中保存该引用的实例化对象，但可能并不知道，Java堆中还必须包含能查找到此对象类型数据的地址信息（如对象类型、父类、实现的接口、方法等），这些类型数据则保存在方法区中。

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：



通过直接指针访问的方式如下：



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。

垃圾回收算法

垃圾回收算法

1. 引用计数法：缺点是无法处理循环引用问题
2. 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
3. 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)
4. 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。
5. 分代

Java并发（JavaConcurrent）

Java并发基础知识

Java并发

（Executor框架和多线程基础）

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：

1. 可以避免由于Java的单继承特性而带来的局限
2. 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
3. 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、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

生产者和消费者问题

生产者和消费者问题

```
package 生产者消费者;

public class ProducerConsumerTest {

    public static void main(String[] args) {
        PublicResource resource = new PublicResource();
        new 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();
    }
}
```

```
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{

    private PublicResource resource;


    public ProducerThread(PublicResource resource) {
        this.resource = resource;
    }


    @Override
    public void run() {
        while (true) {
            try {
                Thread.sleep((long) (Math.random() * 1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            resource.increase();
        }
    }


}
```

```
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{

    private PublicResource resource;


    public ConsumerThread(PublicResource resource) {
        this.resource = resource;
    }


    @Override
    public void run() {
        while (true) {
            try {
                Thread.sleep((long) (Math.random() * 1000));
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            resource.decrease();
        }

    }


}
```

```
package 生产者消费者;

/**
 * 公共资源类
 * @author dream
 *
 */
public class PublicResource {

    private int number = 0;
    private int size = 10;

    /**
     * 增加公共资源
     */
    public synchronized void increase()
    {
        while (number >= size) {
            try {
                wait();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
            number++;
            System.out.println("生产了1个，总共有" + number);
            notifyAll();
        }
    }


    /**
     * 减少公共资源
     */
    public synchronized void decrease()
    {
        while (number <= 0) {
            try {
                wait();
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
        number--;
        System.out.println("消费了1个，总共有" + number);
        notifyAll();
    }
}
```

Thread和Runnable实现多线程的区别

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：

1. 可以避免由于Java的单继承特性而带来的局限
2. 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
3. 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。

线程中断

线程中断

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{

    @Override
    public void run() {

        try {
            System.out.println("in run() - about to sleep for 20 seconds");
            Thread.sleep(20000);
            System.out.println("in run() - woke up");
        } catch (InterruptedException e) {
            System.out.println("in run() - interrupted while sleeping");
            //处理完中断异常后，返回到run()方法入口
            //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
            return;
        }
        System.out.println("in run() - leaving normally");
    }

    public static void main(String[] args) {
        SleepInterrupt si = new SleepInterrupt();
        Thread t = new Thread(si);
        t.start();
        //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
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("in main() - interrupting other thread");
        //中断线程t
        t.interrupt();
        System.out.println("in main() - leaving");
    }

}
```

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。

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{

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

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。

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{

    public static void main(String[] args) {
        Thread t = Thread.currentThread();
        System.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 {
            Thread.sleep(2000);
            System.out.println("was NOT interrupted");
        } catch (InterruptedException e) {
            System.out.println("was interrupted");
        }
        //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());
    }

}
```

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
```

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{

    public static void main(String[] args) {
        System.out.println(
                "Point X: Thread.interrupted()=" + Thread.interrupted());
            Thread.currentThread().interrupt();
            System.out.println(
                "Point Y: Thread.interrupted()=" + Thread.interrupted());
            System.out.println(
                "Point Z: Thread.interrupted()=" + Thread.interrupted());
    }

}
```

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。

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。

守护线程与阻塞线程的情况

守护线程与阻塞线程的四种情况

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：

1. 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。
2. 在守护线程中产生的新线程也是守护线程
3. 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。

线程阻塞

线程可以阻塞于四种状态：

1. 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
2. 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)
3. 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
4. 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。

Synchronized

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。

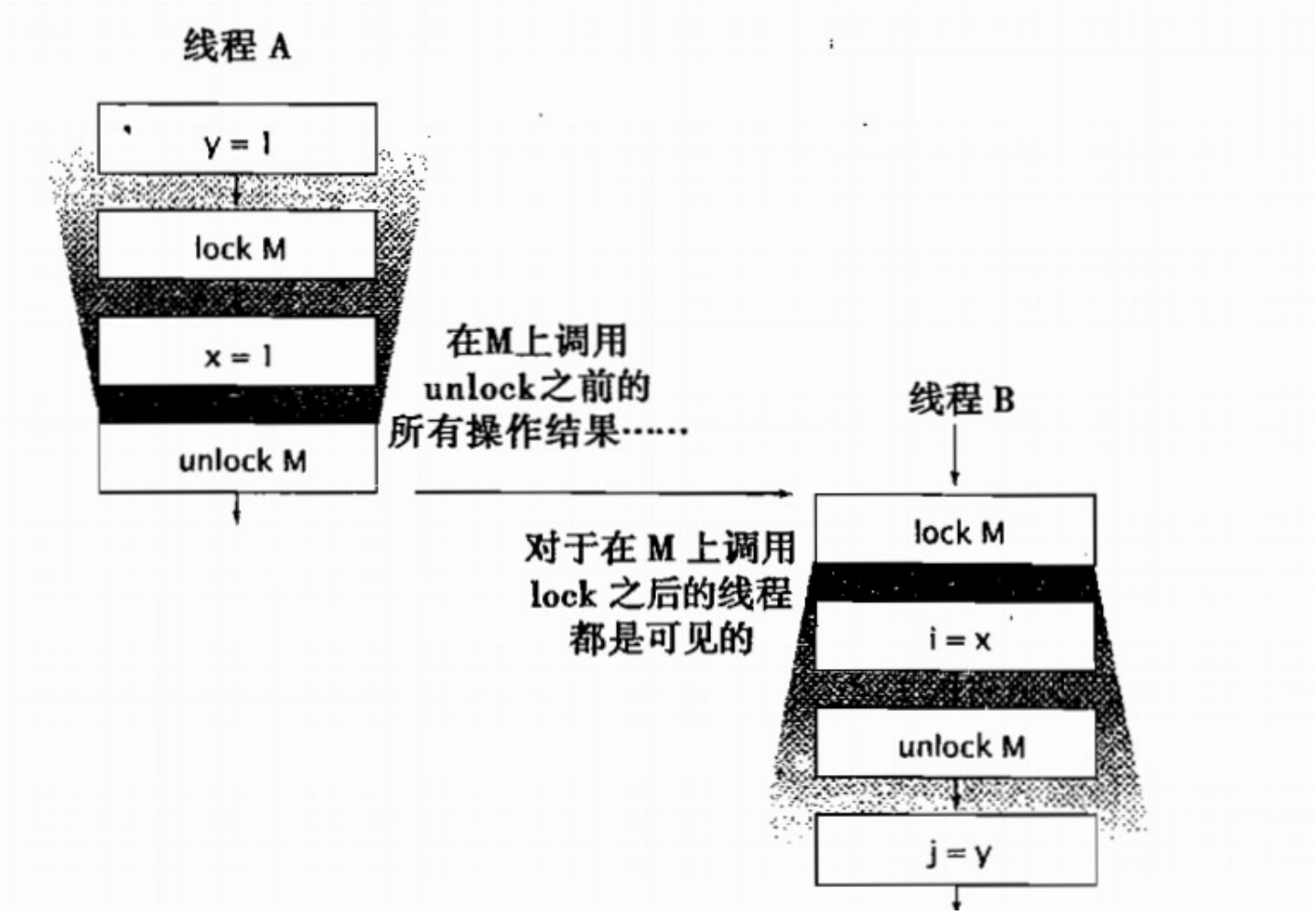
这里就使用同步机制获取互斥锁的情况，进行几点说明：

1. 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。
2. 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。
3. 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。
4. 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。
5. 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。
6. 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。
7. 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。
8. 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。

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。



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。

多线程环境中安全使用集合API

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。

实现内存可见的两种方法比较：加锁和volatile变量

并发编程中实现内存可见的两种方法比较：加锁和volatile变量

1. 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。

1. 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。
2. 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。
3. 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：

1. 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。
2. 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。

死锁

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 {

    private String objID;

    public Deadlock(String id) {
        objID = id;
    }

    public synchronized void checkOther(Deadlock other) {
```

```
        print("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()");
    }

    public synchronized void action() {
        print("entering action()");
        try { Thread.sleep(500); }
        catch ( InterruptedException x ) { }
        print("leaving action()");
    }

    public void print(String msg) {
        threadPrint("objID=" + objID + " - " + msg);
    }

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

    public static void main(String[] args) {
        final 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?");
    }
}
```

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：

1. 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；
2. 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；
3. 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；

可重入内置锁

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。

使用wait/notify/notifyAll实现线程间通信

Java并发编程：线程间协作的两种方式：wait、notify、notifyAll和Condition

转载自： http://www.cnblogs.com/dolphin0520/p/3920385.html

在现实中，需要线程之间的协作。比如说最经典的生产者-消费者模型：当队列满时，生产者需要等待队列有空间才能继续往里面放入商品，而在等待的期间内，生产者必须释放对临界资源（即队列）的占用权。因为生产者如果不释放对临界资源的占用权，那么消费者就无法消费队列中的商品，就不会让队列有空间，那么生产者就会一直无限等待下去。因此，一般情况下，当队列满时，会让生产者交出对临界资源的占用权，并进入挂起状态。然后等待消费者消费了商品，然后消费者通知生产者队列有空间了。同样地，当队列空时，消费者也必须等待，等待生产者通知它队列中有商品了。这种互相通信的过程就是线程间的协作。

今天我们就来探讨一下Java中线程协作的最常见的两种方式：利用Object.wait()、Object.notify()和使用Condition

以下是本文目录大纲：

一.wait()、notify()和notifyAll()

二.Condition

三.生产者-消费者模型的实现

若有不正之处请多多谅解，并欢迎批评指正。

一.wait()、notify()和notifyAll()

wait()、notify()和notifyAll()是Object类中的方法：

```
 /**
 * Wakes up a single thread that is waiting on this object's
 * monitor. If any threads are waiting on this object, one of them
 * is chosen to be awakened. The choice is arbitrary and occurs at
 * the discretion of the implementation. A thread waits on an object's
 * monitor by calling one of the wait methods
 */
public final native void notify();

/**
 * Wakes up all threads that are waiting on this object's monitor. A
 * thread waits on an object's monitor by calling one of the
 * wait methods.
 */
public final native void notifyAll();

/**
 * Causes the current thread to wait until either another thread invokes the
 * {@link java.lang.Object#notify()} method or the
 * {@link java.lang.Object#notifyAll()} method for this object, or a
 * specified amount of time has elapsed.
 * <p>
 * The current thread must own this object's monitor.
 */
public final native void wait(long timeout) throws InterruptedException;
```

从这三个方法的文字描述可以知道以下几点信息：

1）wait()、notify()和notifyAll()方法是本地方法，并且为final方法，无法被重写。

2）调用某个对象的wait()方法能让当前线程阻塞，并且当前线程必须拥有此对象的monitor（即锁）

3）调用某个对象的notify()方法能够唤醒一个正在等待这个对象的monitor的线程，如果有多个线程都在等待这个对象的monitor，则只能唤醒其中一个线程；

4）调用notifyAll()方法能够唤醒所有正在等待这个对象的monitor的线程；

有朋友可能会有疑问：为何这三个不是Thread类声明中的方法，而是Object类中声明的方法（当然由于Thread类继承了Object类，所以Thread也可以调用者三个方法）？其实这个问题很简单，由于每个对象都拥有monitor（即锁），所以让当前线程等待某个对象的锁，当然应该通过这个对象来操作了。而不是用当前线程来操作，因为当前线程可能会等待多个线程的锁，如果通过线程来操作，就非常复杂了。

上面已经提到，如果调用某个对象的wait()方法，当前线程必须拥有这个对象的monitor（即锁），因此调用wait()方法必须在同步块或者同步方法中进行（synchronized块或者synchronized方法）。

调用某个对象的wait()方法，相当于让当前线程交出此对象的monitor，然后进入等待状态，等待后续再次获得此对象的锁（Thread类中的sleep方法使当前线程暂停执行一段时间，从而让其他线程有机会继续执行，但它并不释放对象锁）；

notify()方法能够唤醒一个正在等待该对象的monitor的线程，当有多个线程都在等待该对象的monitor的话，则只能唤醒其中一个线程，具体唤醒哪个线程则不得而知。

同样地，调用某个对象的notify()方法，当前线程也必须拥有这个对象的monitor，因此调用notify()方法必须在同步块或者同步方法中进行（synchronized块或者synchronized方法）。

nofityAll()方法能够唤醒所有正在等待该对象的monitor的线程，这一点与notify()方法是不同的。

这里要注意一点：notify()和notifyAll()方法只是唤醒等待该对象的monitor的线程，并不决定哪个线程能够获取到monitor。

举个简单的例子：假如有三个线程Thread1、Thread2和Thread3都在等待对象objectA的monitor，此时Thread4拥有对象objectA的monitor，当在Thread4中调用objectA.notify()方法之后，Thread1、Thread2和Thread3只有一个能被唤醒。注意，被唤醒不等于立刻就获取了objectA的monitor。假若在Thread4中调用objectA.notifyAll()方法，则Thread1、Thread2和Thread3三个线程都会被唤醒，至于哪个线程接下来能够获取到objectA的monitor就具体依赖于操作系统的调度了。

上面尤其要注意一点，一个线程被唤醒不代表立即获取了对象的monitor，只有等调用完notify()或者notifyAll()并退出synchronized块，释放对象锁后，其余线程才可获得锁执行。

下面看一个例子就明白了：

```
public class Test {
    public static Object object = new Object();
    public static void main(String[] args) {
        Thread1 thread1 = new Thread1();
        Thread2 thread2 = new Thread2();

        thread1.start();

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

        thread2.start();
    }

    static class Thread1 extends Thread{
        @Override
        public void run() {
            synchronized (object) {
                try {
                    object.wait();
                } catch (InterruptedException e) {
                }
                System.out.println("线程"+Thread.currentThread().getName()+"获取到了锁");
            }
        }
    }

    static class Thread2 extends Thread{
        @Override
        public void run() {
            synchronized (object线程Thread-1调用了object.notify()
线程Thread-1释放了锁
线程Thread-0获取到了锁;
                System.out.println("线程"+Thread.currentThread().getName()+"调用了object.notify()");
            }
            System.out.println("线程"+Thread.currentThread().getName()+"释放了锁");
        }
    }
}
```

无论运行多少次，运行结果必定是：

> 线程Thread-1调用了object.notify()
> 线程Thread-1释放了锁
> 线程Thread-0获取到了锁

NIO

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。

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。

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。

Java NIO: Selectors(选择器)

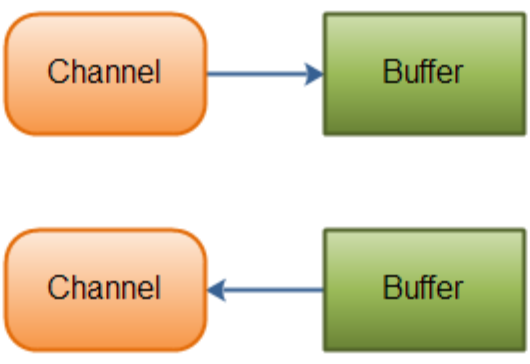
Java NIO引入了选择器的概念，选择器用于监听多个通道的事件(比如：连接打开，数据到达)。因此，单个的线程可以监听多个数据通道。

NIO由以下核心部分组成：

- Channels
- Buffers
- Selectors

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

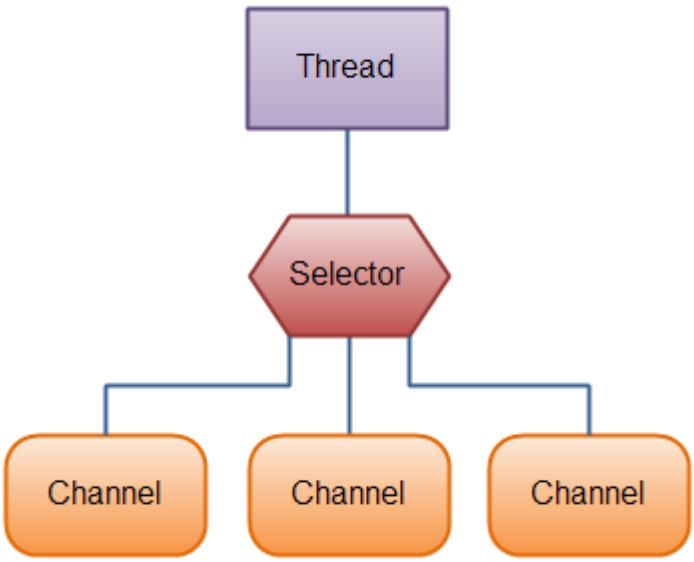
Java NIO还有个MappedByteBuffer，用于表示内存映射文件。

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。

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。

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

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。

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。

Buffer的基本用法

使用Buffer读写数据一般遵循以下四个步骤：

1. 写入数据到Buffer
2. 调用flip()方法
3. 从Buffer中读取数据
4. 调用clear()方法或者compact()方法

数据结构(DataStructure)

数组

数组

```
/**
 * 普通数组的Java代码
 * @author dream
 *
 */
public class GeneralArray {

    private int[] a;
    private int size;   //数组的大小
    private int nElem;   //数组中有多少项

    public GeneralArray(int max){
        this.a = new int[max];
        this.size = max;
        this.nElem = 0;
    }

    public boolean find(int searchNum){   //查找某个值
        int j;
        for(j = 0; j < nElem; j++){
            if(a[j] == searchNum){
                break;
            }
        }
        if(j == nElem){
            return false;
        }else {
            return true;
        }
    }


    public boolean insert(int value){   //插入某个值
        if(nElem == size){
            System.out.println("数组已满");
            return false;
        }
        a[nElem] = value;
        nElem++;
        return true;
    }
```

```
    }

    public boolean delete(int value){   //删除某个值
        int j;
        for(j = 0; j < nElem; j++){
            if(a[j] == value){
                break;
            }
        }
        if(j == nElem){
            return false;
        }
        if(nElem == size){
            for(int k = j; k < nElem - 1; k++){
                a[k] = a[k+1];
            }
        }else {
            for(int k = j; k < nElem; k++){
                a[k] = a[k+1];
            }
        }
        nElem--;
        return true;
    }

    public void display(){   //打印整个数组
        for(int i = 0; i < nElem; i++){
            System.out.println(a[i] + " ");
        }
        System.out.println("");
    }
}
```

```
/**
 * 有序数组的Java代码
 * @author dream
 *
 */

/**
 * 对于数组这种数据结构，
 * 线性查找的话，时间复杂度为O(N)，
 * 二分查找的话时间为O(longN)，
 * 无序数组插入的时间复杂度为O(1)，
 * 有序数组插入的时间复杂度为O(N)，
 * 删除操作的时间复杂度均为O(N)。
 * @author dream
 *
 */
public class OrderedArray {

    private long[] a;
    private int size;   //数组的大小
    private int nElem;  //数组中有多少项
```

```
    public OrderedArray(int max){   //初始化数组
        this.a = new long[max];
        this.size = max;
        this.nElem = 0;
    }

    public int size(){   //返回数组实际有多少值
        return this.nElem;
    }

    /**
     * 二分查找
     * @param searchNum
     * @return
     */
    public int find(long searchNum){
        int lower = 0;
        int upper = nElem - 1;
        int curr;
        while (true) {
            curr = (lower + upper) / 2;
            if(a[curr] == searchNum){
                return curr;
            }else if(lower > upper){
                return -1;
            }else {
                if(a[curr] < searchNum){
                    lower = curr + 1;
                }else {
                    upper = curr - 1;
                }
            }
        }
    }


    public boolean insert(long value){   //插入某个值
        if(nElem == size){
            System.out.println("数组已满!");
            return false;
        }
        int j;
        for(j = 0; j < nElem; j++){
            if(a[j] > value){
                break;
            }
        }

        for(int k = nElem; k > j; k++){
            a[k] = a[k-1];
        }
        a[j] = value;
        nElem++;
```

```
        return true;
    }



    public boolean delete(long value){   //删除某个值
        int j = find(value);
        if(j == -1){
            System.out.println("没有该元素!");
            return false;
        }

        if(nElem == size){
            for(int k = j; k < nElem - 1; k++){
                a[k] = a[k+1];
            }
            a[nElem-1] = 0;
        }else {
            for(int k = j; k < nElem; k++){
                a[k] = a[k+1];
            }
        }
        nElem--;
        return true;
    }


    public void display(){   //打印整个数组
        for(int i = 0; i < nElem; i++){
            System.out.println(a[i] + " ");
        }
        System.out.println("");
    }
}
```

栈和队列

栈和队列

栈

栈只允许访问一个数据项：即最后插入的数据。溢出这个数据才能访问倒数第二个插入的数据项。它是一种"后进先出"的数据结构。

栈最基本的操作是出栈(Pop)、入栈(Push)，还有其他扩展操作，如查看栈顶元素，判断栈是否为空、是否已满，读取栈的大小等。

```
/**
 * 栈是先进后出
 * 只能访问栈顶的数据
 * @author dream
 *
 */

/**
 * 基于数组来实现栈的基本操作
 * 数据项入栈和出栈的时间复杂度均为O(1)
 * @author dream
 *
 */
public class ArrayStack {

    private long[] a;
    private int size;   //栈数组的大小
    private int top;   //栈顶

    public ArrayStack(int maxSize){
        this.size = maxSize;
        this.a = new long[size];
        this.top = -1;   //表示空栈
    }

    public void push(long value){   //入栈
        if(isFull()){
            System.out.println("栈已满!");
            return;
        }
        a[++top] = value;
    }

    public long peek(){   //返回栈顶内容，但不删除
        if(isEmpty()){
            System.out.println("栈中没有数据");
            return 0;
        }
        return a[top];
    }


    public long pop(){   //弹出栈顶内容
        if(isEmpty()){
            System.out.println("栈中没有数据!");
            return 0;
        }
        return a[top--];
    }

    public int size(){
        return top + 1;
    }
```

```
    /**
     * 判断是否满了
     * @return
     */
    public boolean isFull(){
        return (top == size - 1);
    }

    /**
     * 是否为空
     * @return
     */
    public boolean isEmpty(){
        return (top == -1);
    }


    public void display(){
        for (int i = top; i >= 0; i--) {
            System.out.println(a[i] + " ");
        }
        System.out.println("");
    }

}
```

队列

依然使用数组作为底层容器来实现一个队列的封装

```
/**
 * 队列也可以用数组来实现，不过这里有个问题，当数组下标满了后就不能再添加了，
 * 但是数组前面由于已经删除队列头的数据了，导致空。所以队列我们可以用循环数组来实现，
 * @author dream
 *
 */
public class RoundQueue {

    private long[] a;
    private int size;     //数组大小
    private int nItems;    //实际存储数量
    private int front;     //头
    private int rear;       //尾

    public RoundQueue(int maxSize){
        this.size = maxSize;
        a = new long[size];
        front = 0;
        rear = -1;
        nItems = 0;
    }
```

```
    public void insert(long value){
        if(isFull()){
            System.out.println("队列已满");
            return;
        }
        rear = ++rear % size;
        a[rear] = value;  //尾指针满了就循环到0处，这句相当于下面注释内容
        nItems++;
    }

    public long remove(){
        if(isEmpty()){
            System.out.println("队列为空!");
            return 0;
        }
        nItems--;
        front = front % size;
        return a[front++];
    }

    public void display(){
        if(isEmpty()){
            System.out.println("队列为空!");
            return;
        }
        int item = front;
        for(int i = 0;i < nItems; i++){
            System.out.println(a[item++ % size] + " ");
        }
        System.out.println("");
    }

    public long peek(){
        if(isEmpty()){
            System.out.println("队列为空！");
            return 0;
        }
        return a[front];
    }

    public boolean isFull(){
        return (nItems == size);
    }

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

    public int size(){
        return nItems;
    }


}
```

和栈一样，队列中插入数据项和删除数据项的时间复杂度均为O(1)

还有个优先级队列，优先级队列是比栈和队列更专用的数据结构。优先级队列与上面普通的队列相比，主要区别在于队列中的元素是有序的，关键字最小（或者最大）的数据项总在队头。数据项插入的时候会按照顺序插入到合适的位置以确保队列的顺序。优先级队列的内部实现可以用数组或者一种特别的树——堆来实现。这里用数组实现优先级队列。

```

public class PriorityQueue {

    private long[] a;
    private int size;
    private int nItems;  //元素个数

    public PriorityQueue(int maxSize){
        size = maxSize;
        nItems = 0;
        a = new long[size];
    }

    public void insert(long value){
        if(isFull()){
            System.out.println("队列已满！");
            return;
        }
        int j;
        if(nItems == 0){  //空队列直接添加
            a[nItems++] = value;
        }else {
            //将数组中的数字依照下标按照从大到小排列
            for(j=nItems-1; j>=0; j--){
                if(value > a[j]){
                    a[j+1] = a[j];
                }
                else {
                    break;
                }
            }
            a[j+1] = value;
            nItems++;
        }
    }

    public long remove(){
        if(isFull()){
            System.out.println("队列为空!");
            return 0;
        }
        return a[--nItems];
    }
```

```
    public long peekMin(){
        return a[nItems - 1];
    }

    public boolean isFull(){
        return (nItems == size);
    }

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

    public int size(){
        return nItems;
    }

    public void display(){
        for(int i = nItems - 1;i >= 0; i--){
            System.out.println(a[i] + " ");
        }
        System.out.println(" ");
    }
```

}

```

优先级队列中，插入操作需要O(N)的时间，而删除操作则需要O(1)的时间。
```

Algorithm(算法)

排序

100644 blob ccf96ac4f6767fb448ba37297d785409306678c0 冒泡排序.md
100644 blob 689469b273c3f6180401b483ffa631fd7efd2143 归并排序.md
100644 blob 20a6987ec05b2784450685312c2f68586e6a2ff7 快速排序.md
100644 blob 75465ab7057e7ce90685793036eaa4b11b62974d 选择排序.md

选择排序

选择排序：

- 背景介绍： 选择排序（Selection sort）是一种简单直观的排序算法。它的工作原理如下。首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。 ----- 来自 wikipedia
- 算法规则： 将待排序集合(0...n)看成两部分，在起始状态中，一部分为(k..n)的待排序unsorted集合，另一部分为(0...k)的已排序sorted集合,在待排序集合中挑选出最小元素并且记录下标i，若该下标不等于k，那么 unsorted[i] 与 sorted[k]交换 ，一直重复这个过程，直到unsorted集合中元素为空为止。
- 代码实现（Java版本）

```
public void sort(int[] args)
{
        int len = args.length;
        for (int i = 0,k = 0; i < len; i++,k = i) {
            // 在这一层循环中找最小
            for (int j = i + 1; j < len; j++) {
                // 如果后面的元素比前面的小，那么就交换下标，每一趟都会选择出来一个最小值的下标
                if (args[k] > args[j]) k = j;
            }

            if (i != k) {
                int tmp = args[i];
                args[i] = args[k];
                args[k] = tmp;
            }
        }
    }
```

冒泡排序

冒泡排序：

- 背景介绍： 是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元素，如果他们的顺序错误就把他们交换过来。走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。----- 来自 wikipedia
- 算法规则： 由于算法每次都将一个最大的元素往上冒，我们可以将待排序集合(0...n)看成两部分，一部分为(k..n)的待排序unsorted集合，另一部分为(0...k)的已排序sorted集合，每一次都在unsorted集合从前往后遍历，选出一个数，如果这个数比其后面的数大，则进行交换。完成一轮之后，就肯定能将这一轮unsorted集合中最大的数移动到集合的最后，并且将这个数从unsorted中删除，移入sorted中。
- 代码实现（Java版本）

```
public void sort(int[] args)
    {
        //第一层循环从数组的最后往前遍历
        for (int i = args.length - 1; i > 0 ; --i) {
            //这里循环的上界是 i - 1，在这里体现出 “将每一趟排序选出来的最大的数从sorted中移除”
            for (int j = 0; j < i; j++) {
                //保证在相邻的两个数中比较选出最大的并且进行交换(冒泡过程)
                if (args[j] > args[j+1]) {
                    int temp = args[j];
                    args[j] = args[j+1];
                    args[j+1] = temp;
                }
            }
        }
    }
```

快速排序

快速排序：

- 背景介绍： 又称划分交换排序（partition-exchange sort），一种排序算法，最早由东尼·霍尔提出。在平均状况下，排序n个项目要O(n log n)次比较。在最坏状况下则需要O(n2)次比较，但这种状况并不常见。事实上，快速排序通常明显比其他O(n log n)算法更快，因为它的内部循环（inner loop）可以在大部分的架构上很有效率地被实现出来 ----- 来自 wikipedia **
- 算法规则： 本质来说，快速排序的过程就是不断地将无序元素集递归分割，一直到所有的分区只包含一个元素为止。
  由于快速排序是一种分治算法，我们可以用分治思想将快排分为三个步骤：
  1.分：设定一个分割值，并根据它将数据分为两部分
  2.治：分别在两部分用递归的方式，继续使用快速排序法
  3.合：对分割的部分排序直到完成
- 代码实现（Java版本）

```
public int dividerAndChange(int[] args, int start, int end)
{
    //标准值
    int pivot = args[start];
    while (start < end) {
        // 从右向左寻找，一直找到比参照值还小的数值，进行替换
        // 这里要注意，循环条件必须是 当后面的数 小于 参照值的时候
        // 我们才跳出这一层循环
        while (start < end && args[end] >= pivot)
            end--;

        if (start < end) {
            swap(args, start, end);
            start++;
        }

        // 从左向右寻找，一直找到比参照值还大的数组，进行替换
        while (start < end && args[start] < pivot)
            start++;

        if (start < end) {
            swap(args, end, start);
            end--;
        }
    }

    args[start] = pivot;
    return start;
}

public void sort(int[] args, int start, int end)
{
    //当分治的元素大于1个的时候，才有意义
    if ( end - start > 1) {
        int mid = 0;
        mid = dividerAndChange(args, start, end);
        // 对左部分排序
        sort(args, start, mid);
        // 对右部分排序
        sort(args, mid + 1, end);
    }
}

private void swap(int[] args, int fromIndex, int toIndex)
{
    args[fromIndex] = args[toIndex];
}
```

归并排序

归并排序：

- 背景介绍： 是创建在归并操作上的一种有效的排序算法，效率为O(n log n)。1945年由约翰·冯·诺伊曼首次提出。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用，且各层分治递归可以同时进行。 ----- 来自 wikipedia
- 算法规则： 像快速排序一样，由于归并排序也是分治算法，因此可使用分治思想：
  1.申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列
  2.设定两个指针，最初位置分别为两个已经排序序列的起始位置
  3.比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置
  4.重复步骤3直到某一指针到达序列尾
  5.将另一序列剩下的所有元素直接复制到合并序列尾
- 代码实现（Java版本）

```
public void mergeSort(int[] ints, int[] merge, int start, int end)
{
    if (start >= end) return;

    int mid = (end + start) >> 1;

    mergeSort(ints, merge, start, mid);
    mergeSort(ints, merge, mid + 1, end);

    merge(ints, merge, start, end, mid);

}

private void merge(int[] a, int[] merge, int start, int end,int mid)
{
    int i = start;
    int j = mid+1;
    int pos = start;
    while( i <= mid || j <= end ){
        if( i > mid ){
            while( j <= end ) merge[pos++] = a[j++];
            break;
        }

        if( j > end ){
            while( i <= mid ) merge[pos++] = a[i++];
            break;
        }

        merge[pos++] = a[i] >= a[j] ? a[j++] : a[i++];
    }

    for (pos = start; pos <= end; pos++)
        a[pos] = merge[pos];

}
```

查找

顺序查找

顺序查找

基本原理：依次遍历

```
public class Solution {

    public static int SequenceSearch(int[] sz, int key) {
        for (int i = 0; i < sz.length; i++) {
            if (sz[i] == key) {
                return i;
            }
        }
        return -1;
    }
}
```

折半查找

折半查找

基本原理：每次查找都对半分，但要求数组是有序的

```
public class Solution {

    public static int BinarySearch(int[] sz,int key){
        int low = 0;
        int high = sz.length - 1;

        while (low <= high) {
            int middle = (low + high) / 2;
            if(sz[middle] == key){
                return middle;
            }else if(sz[middle] > key){
                high = middle - 1;
            }else {
                low = middle + 1;
            }
        }
        return -1;
    }
}
```

Network(网络)

100644 blob 03f7cf5419decec503c76dffc5052f4ac8b624d2 Http协议.md
100644 blob 11d6d5ed07bb502dbc347c0774ab8e9d3aaed2ad Socket.md
100644 blob b4544309ef27ddd8a9d73fc4d3d463835cfc38b7 TCP与UDP.md

TCP/UDP

TCP与UDP

面向报文的传输方式是应用层交给UDP多长的报文，UDP就照样发送，即一次发送一个报文。因此，应用程序必须选择合适大小的报文。若报文太长，则IP层需要分片，降低效率。若太短，会是IP太小。UDP对应用层交下来的报文，既不合并，也不拆分，而是保留这些报文的边界。这也就是说，应用层交给UDP多长的报文，UDP就照样发送，即一次发送一个报文。
面向字节流的话，虽然应用程序和TCP的交互是一次一个数据块（大小不等），但TCP把应用程序看成是一连串的无结构的字节流。TCP有一个缓冲，当应用程序传送的数据块太长，TCP就可以把它划分短一些再传送。如果应用程序一次只发送一个字节，TCP也可以等待积累有足够多的字节后再构成报文段发送出去。

TCP协议

- Transmission Control Protocol，传输控制协议
- 面向连接的协议
- 需要三次握手建立连接
- 需要四次挥手断开连接
- TCP报头最小长度：20字节

三次握手的过程：

---

1. 客户端发送：SYN = 1, SEQ = X, 端口号
2. 服务器回复：SYN = 1, ACK = X + 1, SEQ = Y
3. 客户端发送：ACK = Y + 1, SEQ = X + 1

> 确认应答信号ACK = 收到的SEQ + 1。
> 连接建立中，同步信号SYN始终为1。连接建立后，同步信号SYN=0。

四次挥手过程

---

1. A向B提出停止连接请求，FIN = 1
2. B收到，ACK = 1
3. B向A提出停止连接请求，FIN = 1
4. A收到，ACK = 1

优点：

- 可靠，稳定
  1、传递数据前，会有三次握手建立连接
  2、传递数据时，有确认、窗口、重传、拥塞控制
  3、传递数据后，会断开连接节省系统资源

缺点：

- 传输慢，效率低，占用系统资源高
  1、传递数据前，建立连接需要耗时
  2、传递数据时，确认、重传、拥塞等会消耗大量时间以及CPU和内存等硬件资源
- 易被攻击
  1、因为有确认机制，三次握手等机制，容易被人利用，实现DOS 、DDOS攻击

如何保证接收的顺序性：

TCP协议使用SEQ和ACK机制保证了顺序性
TCP的每个报文都是有序号的。确认应答信号ACK=收到的SEQ+1

UDP协议

- User Data Protocol，用户数据包协议
- 面向无连接的协议
- UDP报头只有8字节

简介：

- 传输数据之前源端和终端不建立连接，当它想传送时就简单地去抓取来自应用程序的数据，并尽可能快的把它扔到网络上
- 在发送端，UDP传送数据的速度仅仅是受应用程序生成数据的速度、计算机的能力和传输带宽的限制
- 在接收端，UDP把每个消息段放在队列中，应用程序每次从队列中读一个消息段
- 由于传输数据不建立连接，因此也就不需要维护连接状态，包括收发状态等，因此一台服务机可同时向多个客户机传输相同的消息
- UDP信息包的标题很短，只有8个字节，相对于TCP的20个字节信息包的额外开销很小
- 吞吐量不受拥挤控制算法的调节，只受应用软件生成数据的速率、传输带宽、源端和终端主机性能的

限制

- UDP使用尽最大努力交付，即不保证可靠交付，因此主机不需要维持复杂的链接状态表。
- UDP是面向报文的。发送方的UDP对应用程序交下来的报文，在添加首部后就向下交付给IP层。既不拆分，也不合并，而是保留这些报文的边界，因此，应用程序需要选择合适的报文大小。

> 使用“ping”命令来测试两台主机之间TCP/IP通信是否正常，其实“ping”命令的原理就是向对方主机发送UDP数据包，然后对方主机确认收到数据包，如果数据包是否到达的消息及时反馈回来，那么网络就是通的。

优点：

- 传输速率快
  1、传输数据前，不需要像TCP一样建立连接
  2、传输数据时，没有确认、窗口、重传、拥塞控制等机制
- 较安全
  1、由于没有了TCP的一些机制，被攻击者利用的漏洞就少了

缺点：

- 不可靠，不稳定
  1、由于没有了TCP的机制，在数据传输时如果网络不好，很可能丢包

用UDP协议通讯时怎样得知目标机是否获得了数据包

仿造TCP的做法，每发一个UDP包，都在里面加一个SEQ序号，接收方收到包后，将SEQ序号回复给发送方。如果发送方在指定时间以内没有收到回应，说明丢包了。

TCP与UDP的区别

TCP面向有链接的通信服务	UDP面向无连接的通信服务
TCP提供可靠的通信传输	UDP不可靠,会丢包
TCP保证数据顺序	UDP不保证
TCP数据无边界	UDP有边界
TCP速度快	UDP速度慢
TCP面向字节流	UDP面向报文
TCP一对一	UDP可以一对一，一对多
TCP报头至少20字节	UDP报头8字节
TCP有流量控制，拥塞控制	UDP没有

为什么UDP比TCP快

1. TCP需要三次握手
2. TCP有拥塞控制，控制流量等机制

为什么TCP比UDP可靠

1. TCP是面向有连接的，建立连接之后才发送数据；而UDP则不管对方存不存在都会发送数据。
2. TCP有确认机制，接收端每收到一个正确包都会回应给发送端。超时或者数据包不完整的话发送端会重传。UDP没有。因此可能丢包。

什么时候使用TCP

当对网络通讯质量有要求的时候，比如：整个数据要准确无误的传递给对方，这往往用于一些要求可靠的应用，比如HTTP、HTTPS、FTP等传输文件的协议，POP、SMTP等邮件传输的协议。
在日常生活中，常见使用TCP协议的应用如下：
浏览器，用的HTTP
FlashFXP，用的FTP
Outlook，用的POP、SMTP
Putty，用的Telnet、SSH
QQ文件传输

什么时候应该使用UDP：

当对网络通讯质量要求不高的时候，要求网络通讯速度能尽量的快，这时就可以使用UDP。
比如，日常生活中，常见使用UDP协议的应用如下：
QQ语音
QQ视频
TFTP

TCP无边界，UDP有边界

TCP无边界

客户端分多次发送数据给服务器，若服务器的缓冲区够大，那么服务器端会在客户端发送完之后一次性接收过来，所以是无边界的；

UDP有边界

客户端每发送一次，服务器端就会接收一次，也就是说发送多少次就会接收多少次，因此是有边界的。

HTTP

Http协议

- 默认端口：80

Http协议的主要特点

1. 支持客户 / 服务器模式
2. 简单快速：客户向服务端请求服务时，只需传送请求方式和路径。
3. 灵活：允许传输任意类型的数据对象。由Content-Type加以标记。
4. 无连接：每次响应一个请求，响应完成以后就断开连接。
5. 无状态：服务器不保存浏览器的任何信息。每次提交的请求之间没有关联。

非持续性和持续性

- HTTP1.0默认非持续性；HTTP1.1默认持续性

持续性

浏览器和服务器建立TCP连接后，可以请求多个对象

非持续性

浏览器和服务器建立TCP连接后，只能请求一个对象

非流水线和流水线

类似于组成里面的流水操作

- 流水线：不必等到收到服务器的回应就发送下一个报文。
- 非流水线：发出一个报文，等到响应，再发下一个报文。类似TCP。

POST和GET的区别

Post一般用于更新或者添加资源信息	**Get一般用于查询操作，而且应该是安全和幂等的**
Post更加安全	Get会把请求的信息放到URL的后面
Post传输量一般无大小限制	Get不能大于2KB
Post执行效率低	Get执行效率略高

为什么POST效率低，Get效率高

- Get将参数拼成URL,放到header消息头里传递
- Post直接以键值对的形式放到消息体中传递。

- 但两者的效率差距很小很小

Https

- 端口号是443
- 是由SSL+Http协议构建的可进行加密传输、身份认证的网络协议。

Socket

Socket

使用TCP

客户端

```
Socket socket = new Socket("ip", 端口);

InputStream is = socket.getInputStream();
DataInputStream dis = new DataInputStream(is);

OutputStream os = socket.getOutputStream();
DataInputStream dos = new DataOutputStream(os);
```

服务器端

```
ServerSocket serverSocket = new ServerSocket(端口);
Socket socket = serverSocket.accept();
//获取流的方式与客户端一样
```

读取输入流

```
byte[] buffer = new byte[1024];
do{
    int count = is.read(buffer);
    if(count <= 0){ break; }
    else{
    // 对buffer保存或者做些其他操作
        }
    }
while(true);
```

使用UDP

客户端和服务器端一样的

```
DatagramSocket socket = new DatagramSocket(端口);
InetAddress serverAddress = InetAddress.getbyName("ip");
//发送
DatagramPackage packet = new DatagramPacket(buffer, length, host, port);
socket.send(packet);
//接收
byte[] buf = new byte[1024];
DatagramPacket packet = new DatagramPacket(buf, 1024);
Socket.receive(packet);
```

OperatingSystem(操作系统)

OS

进程和线程

关系：

一个进程可以创建和撤销另一个线程，同一个进程中的线程可以并发执行。

死锁的必要条件，怎么处理死锁。

Window内存管理方式：段存储，页存储，段页存储。

进程的几种状态和转换

什么是虚拟内存。

Linux下的IPC几种通信方式

1. 管道(pipe):管道可用于具有亲缘关系的进程间的通信，是一种半双工的方式，数据只能单向流动，允许一个进程和另一个与它有公共祖先的进程之间进行通信。
2. 命名管道(named pipe):命名管道克服了管道没有名字的限制，同时除了具有管道的功能外(也是半双工)，它还允许无亲缘关系进程间的通信。命令管道在文件系统中有对应的文件名。命令管道通过命令mkfifo或系统调用mkfifo来创建。
3. 信号(signal):信号是比较复杂的通信方式，用于通知接收进程有某种事件发生了，除了进程间通信外，进程还可以发送信号给进程本身。
4. 消息队列:消息队列是消息的链接表，包括Posix消息队列和system V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程可以读走队列中的消息。消息队列克服了信号承载信息少，管道只能承载无格式字节流以及缓冲区大小受限等缺点。
5. 共享内存:使得多个进程可以访问同一块内存空间，是最快的IPC形式。是针对其他通信机制运行效率低而设计的。往往与其他通信机制，如信号量结合使用，来达到进程间的同步及互斥。
6. 内存映射:内存映射允许任何多个进程间通信每一个使用该机制的进程通过把一个共享的文件映射到自己的进程地址空间来实现它。
7. 信号量(semaphore):主要作为进程间以及同一进程不同线程之间的同步手段。
8. 套接字(Socket):更为一般的进程间通信机制，可用于不同机器之间的进程间通信。

逻辑地址、物理地址的区别

进程调度算法

Linux系统的IPC

Linux系统的IPC

线程之间不存在通信,因为本来就共享同一片内存

各个线程可以访问进程中的公共变量，资源，所以使用多线程的过程中需要注意的问题是如何防止两个或两个以上的线程同时访问同一个数据，以免破坏数据的完整性。数据之间的相互制约包括
1、直接制约关系,即一个线程的处理结果,为另一个线程的输入,因此线程之间直接制约着，这种关系可以称之为同步关系
2、间接制约关系，即两个线程需要访问同一资源,该资源在同一时刻只能被一个线程访问,这种关系称之为线程间对资源的互斥访问，某种意义上说互斥是一种制约关系更小的同步

线程间同步

- 临界区
  - 每个线程中访问临界资源的代码,一个线程拿到临界区的所有权后,可以多次重入.只有前一个线程放弃,后一个才可以进来
- 互斥量
  - 互斥量就是简化版的信号量
  - 互斥量由于也有线程所有权的概念，故也只能进行线程间的资源互斥访问，不能由于线程同步
  - 由于互斥量是内核对象，因此其可以进行进程间通信，同时还具有一个很好的特性，就是在进程间通信时完美的解决了"遗弃"问题
- 信号量
  - 信号量的用法和互斥的用法很相似，不同的是它可以同一时刻允许多个线程访问同一个资源，PV操作
  - 信号量也是内核对象,也可以进行进程间通信

进程间通信

- 管道
  - 半双工
  - 只能在具有父子关系的进程间使用
  - FIFO的共享内存实现
- 命名管道

    - 以linux中的文件的形式存在
    - 不要求进程有用父子关系,甚至通过网络也是可以的
    - 半双工
- 信号量
    - 有up和down操作
    - 共享内存实现
- 消息队列
    - 队列数据结构
    - 共享内存实现
- 信号
    - 较为复杂的方式,用于通知进程某事件已经发生.例如kill信号
- 共享内存
    - 将同一个物理内存附属到两个进程的虚拟内存中
- socket
    - 可以跨网络通信

android中常用设计模式

面向对象六大原则

常见的面向对象设计原则

1. 单一职责原则 SRP
   一个类应该仅有一个引起它变化的原因。
2. 开放关闭原则 OCP
   一个类应该对外扩展开放，对修改关闭。
3. 里氏替换原则 LSP
   子类型能够替换掉它们的父类型。
4. 依赖倒置原则 DIP
   要依赖于抽象，不要依赖于具体类，要做到依赖倒置，应该做到：
   - 高层模块不应该依赖底层模块，二者都应该依赖于抽象。
   - 抽象不应该依赖于具体实现，具体实现应该依赖于抽象。
5. 接口隔离原则 ISP
   不应该强迫客户依赖于他们不用的方法。
6. 最少知识原则 LKP
   只和你的朋友谈话。
7. 其他原则
   - 面向接口编程
   - 优先使用组合，而非继承
   - 一个类需要的数据应该隐藏在类的内部
   - 类之间应该零耦合，或者只有传导耦合，换句话说，类之间要么没关系，要么只使用另一个类的接口提供的操作
   - 在水平方向上尽可能统一地分布系统功能

单例模式

单例模式

定义

> 保证一个类仅有一个实例，并提供一个访问它的全局访问点。
>
> Singleton:负责创建Singleton类自己的唯一实例，并提供一个getInstance的方法，让外部来访问这个类的唯一实例。

- 饿汉式：

```
private static Singleton uniqueInstance = new Singleton();
```

- 懒汉式

```
private static Singleton uniqueInstance = null;
```

功能

单例模式是用来保证这个类在运行期间只会被创建一个类实例，另外，单例模式还提供了一个全局唯一访问这个类实例的访问点，就是getInstance方法。

范围

Java里面实现的单例是一个虚拟机的范围。因为装载类的功能是虚拟机的，所以一个虚拟机在通过自己的ClassLoader装载饿汉式实现单例类的时候就会创建一个类的实例。

懒汉式单例有延迟加载和缓存的思想

优缺点

- 懒汉式是典型的时间换空间
- 饿汉式是典型的空间换时间

- 不加同步的懒汉式是线程不安全的。比如，有两个线程，一个是线程A，一个是线程B，它们同时调用getInstance方法，就可能导致并发问题。
- 饿汉式是线程安全的，因为虚拟机保证只会装载一次，在装载类的时候是不会发生并发的。

如何实现懒汉式的线程安全？

加上synchronized即可

```
public static synchronized Singleton getInstance(){}
```

但这样会降低整个访问的速度，而且每次都要判断。可以用双重检查加锁。

双重加锁机制，指的是：并不是每次进入getInstance方法都需要同步，而是先不同步，进入方法过后，先检查实例是否存在，如果不存在才进入下面的同步块，这是第一重检查。进入同步块后，再次检查实例是

否存在，如果不存在，就在同步的情况下创建一个实例。这是第二重检查。

双重加锁机制的实现会使用一个关键字volatile，它的意思是：被volatile修饰的变量的值，将不会被本地线程缓存，所有对该变量的读写都是直接操作共享内存，从而确保多个线程能正确的处理该变量。

```
/**
 * 双重检查加锁的单例模式
 * @author dream
 *
 */
public class Singleton {

    /**
     * 对保存实例的变量添加volitile的修饰
     */
    private volatile static Singleton instance = null;
    private Singleton(){

    }

    public static Singleton getInstance(){
        //先检查实例是否存在，如果不存在才进入下面的同步块
        if(instance == null){
            //同步块，线程安全的创建实例
            synchronized (Singleton.class) {
                //再次检查实例是否存在，如果不存在才真正的创建实例
                instance = new Singleton();
            }
        }
        return instance;
    }

}
```

一种更好的单例实现方式

```
public class Singleton {

    /**
     * 类级的内部类，也就是静态类的成员式内部类，该内部类的实例与外部类的实例
     * 没有绑定关系，而且只有被调用时才会装载，从而实现了延迟加载
     * @author dream
     *
     */
    private static class SingletonHolder{
        /**
         * 静态初始化器，由JVM来保证线程安全
         */
        private static Singleton instance = new Singleton();
    }

    /**
     * 私有化构造方法
     */
    private Singleton(){

    }

    public static Singleton getInstance(){
        return SingletonHolder.instance;
    }
}
```

根据《高效Java第二版》中的说法，单元素的枚举类型已经成为实现Singleton的最佳方法。

```
package example6;

/**
 * 使用枚举来实现单例模式的示例
 * @author dream
 *
 */
public class Singleton {

    /**
     * 定义一个枚举的元素，它就代表了Singleton的一个实例
     */
    uniqueInstance;

    /**
     * 示意方法，单例可以有自己的操作
     */
    public void singletonOperation(){
        //功能树立
    }
}
```

本质

控制实例数量

何时选用单例模式

当需要控制一个类的实例只能有一个，而且客户只能从一个全局访问点访问它时，可以选用单例模式，这些功能恰好是单例模式要解决的问题。

Builder模式

Builder模式

模式介绍

模式的定义

将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

模式的使用场景

1. 相同的方法，不同的执行顺序，产生不同的事件结果时；
2. 多个部件或零件，都可以装配到一个对象中，但是产生的运行结果又不相同时；
3. 产品类非常复杂，或者产品类中的调用顺序不同产生了不同的效能，这个时候使用建造者模式非常合适；

Android源码中的模式实现

在Android源码中，我们最常用到的Builder模式就是AlertDialog.Builder， 使用该Builder来构建复杂的AlertDialog对象。简单示例如下：

```
//显示基本的AlertDialog
    private void showDialog(Context context) {
        AlertDialog.Builder builder = new AlertDialog.Builder(context);
        builder.setIcon(R.drawable.icon);
        builder.setTitle("Title");
        builder.setMessage("Message");
        builder.setPositiveButton("Button1",
                new DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog, int whichButton) {
                        setTitle("点击了对话框上的Button1");
                    }
                });
        builder.setNeutralButton("Button2",
                new DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog, int whichButton) {
                        setTitle("点击了对话框上的Button2");
                    }
                });
        builder.setNegativeButton("Button3",
                new DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog, int whichButton) {
                        setTitle("点击了对话框上的Button3");
                    }
                });
        builder.create().show();  // 构建AlertDialog， 并且显示
    }
```

优点与缺点

优点

- 良好的封装性， 使用建造者模式可以使客户端不必知道产品内部组成的细节；
- 建造者独立，容易扩展；
- 在对象创建过程中会使用到系统中的一些其它对象，这些对象在产品对象的创建过程中不易得到。

缺点

- 会产生多余的Builder对象以及Director对象，消耗内存；
- 对象的构建过程暴露。

原型模式

原型模式

模式介绍

模式的定义

用原型实例指定创建对象的种类，并通过拷贝这些原型创建新的对象。

模式的使用场景

1. 类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等，通过原型拷贝避免这些消耗；
2. 通过 new 产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式；
3. 一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用，即保护性拷贝。

Android源码中的模式实现

Intent中使用了原型模式

```
Uri uri = Uri.parse("smsto:0800000123");
    Intent shareIntent = new Intent(Intent.ACTION_SENDTO, uri);
    shareIntent.putExtra("sms_body", "The SMS text");

    Intent intent = (Intent)shareIntent.clone() ;
    startActivity(intent);
```

优点与缺点

优点

原型模式是在内存二进制流的拷贝，要比直接 new 一个对象性能好很多，特别是要在一个循环体内产生大量的对象时，原型模式可以更好地体现其优点。

缺点

这既是它的优点也是缺点，直接在内存中拷贝，构造函数是不会执行的，在实际开发当中应该注意这个潜在的问题。优点就是减少了约束，缺点也是减少了约束，需要大家在实际应用时考虑。

简单工厂

简单工厂

接口

接口是一种特殊的抽象类，跟一般的抽象类相比，接口里的所有方法都是抽象方法，接口里的所有属性都是常量。也就是说接口里面只有方法定义没有任何方法实现。

接口的思想是"封装隔离"

简单工厂

示例代码：
https://github.com/GeniusVJR/DesignMode_Java/tree/master/SimpleFactory

客户端在调用的时候，不但知道了接口，同时还知道了具体的实现。接口的思想是"封装隔离"，而实现类Impl应该是被接口Api封装并同客户端隔离开来的，客户端不应该知道具体的实现类是Impl。

简单工厂的功能

不仅可以利用简单工厂来创建接口，也可以用简单工厂来创造抽象类，甚至是一个具体的实例。

静态工厂

没有创建工厂实例的必要，把简单工厂实现成一个工具类，直接使用静态方法。

万能工厂

一个简单哪工厂可以包含很多用来构造东西的方法，这些方法可以创建不同的接口、抽象类或者是类实例。

简单工厂的优缺点

1. 优点

- 帮助封装
- 解耦

1. 缺点

- 可能增加客户端的复杂度
- 不方便扩展子工厂

思考

简单工厂的本质是选择实现。

策略模式

策略模式

模式的定义

策略模式定义了一系列的算法，并将每一个算法封装起来，而且使它们还可以相互替换。策略模式让算法独立于使用它的客户而独立变化。

注：针对同一类型操作，将复杂多样的处理方式分别开来，有选择的实现各自特有的操作。

模式的使用场景

- 针对同一类型问题的多种处理方式，仅仅是具体行为有差别时。
- 需要安全的封装多种同一类型的操作时。
- 出现同一抽象多个子类，而又需要使用if-else 或者 switch-case来选择时。

Android源码中的模式实现

策略模式主要用来分离算法，根据相同的行为抽象来做不同的具体策略实现。

优缺点

优点：

- 结构清晰明了、使用简单直观。
- 耦合度相对而言较低，扩展方便。
- 操作封装也更为彻底，数据更为安全。

缺点：

- 随着策略的增加，子类也会变得繁多。

责任链模式

责任链模式

模式介绍

模式的定义

一个请求沿着一条“链”传递，直到该“链”上的某个处理者处理它为止。

模式的使用场景

一个请求可以被多个处理者处理或处理者未明确指定时。

观察者模式

观察者模式

首先在Android中，我们往ListView添加数据后，都会调用Adapter的notifyDataChanged()方法，其中使用了观察者模式。

当ListView的数据发生变化时，调用Adapter的notifyDataSetChanged函数，这个函数又会调用DataSetObservable的notifyChanged函数，这个函数会调用所有观察者(AdapterDataSetObserver)的onChanged方法，在onChanged函数中又会调用ListView重新布局的函数使得ListView刷新界面。

Android中应用程序发送广播的过程：

- 通过sendBroadcast把一个广播通过Binder发送给ActivityManagerService，ActivityManagerService根据这个广播的Action类型找到相应的广播接收器，然后把这个广播放进自己的消息队列中，就完成第一阶段对这个广播的异步分发。
- ActivityManagerService在消息循环中处理这个广播，并通过Binder机制把这个广播分发给注册的ReceiverDispatcher，ReceiverDispatcher把这个广播放进MainActivity所在线程的消息队列中，就完成第二阶段对这个广播的异步分发：
- ReceiverDispatcher的内部类Args在MainActivity所在的线程消息循环中处理这个广播，最终是将这个广播分发给所注册的BroadcastReceiver实例的onReceive函数进行处理：

代理模式

代理模式

模式介绍

代理模式是对象的结构模式。代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。

模式的使用场景

就是一个人或者机构代表另一个人或者机构采取行动。在一些情况下，一个客户不想或者不能够直接引用一个对象，而代理对象可以在客户端和目标对象之间起到中介的作用。

角色介绍

- 抽象对象角色：声明了目标对象和代理对象的共同接口，这样一来在任何可以使用目标对象的地方都可以使用代理对象。
- 目标对象角色：定义了代理对象所代表的目标对象。
- 代理对象角色：代理对象内部含有目标对象的引用，从而可以在任何时候操作目标对象；代理对象提供一个与目标对象相同的接口，以便可以在任何时候替代目标对象。代理对象通常在客户端调用传递给目标对象之前或之后，执行某个操作，而不是单纯地将调用传递给目标对象。

优点与缺点

优点

给对象增加了本地化的扩展性，增加了存取操作控制

缺点

会产生多余的代理类

适配器模式

适配器模式

定义：

> 将一个类的接口转换成客户希望的另一个接口。适配器模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

功能：

> 进行转换匹配，目的是复用已有的功能，而不是来实现新的接口。在适配器里实现功能，这种适配器称为智能适配器。

优点：

- 更好的复用性
- 更好的扩展性

缺点：

- 过多的使用适配器，会让系统非常零乱，不容易整体进行把握。

本质：

转换匹配，复用功能。

何时选用适配器模式：

- 如果你想要使用一个已经存在的类，但是它的接口不符合你的需求，这种情况可以使用适配器模式，来把已有的实现转换成你需要的接口。
- 如果你想创建一个可以复用的类，这个类可能和一些不兼容的类一起工作，这种情况可以使用适配器模式，到时候需要什么就适配什么。
- 如果你想使用一些已经存在的子类，但是不可能对每一个子类都进行适配，这种情况可以选用对象适配器，直接适配这些子类的父类就可以了。

外观模式

外观模式

定义

为子系统中的一组接口提供一个一致的界面，Facade模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

外观模式的目的

不是给子系统添加新的功能接口，而是为了让外部减少与子系统内多个模块的交互，松散耦合，从而让外部能够更简单的使用子系统。

优缺点

1. 优点
   - 松散耦合
   - 简单易用
   - 更好的划分访问的层次
2. 缺点
   - 过多的或者是不太合理的Facade也容易让人迷惑。到底是调用Facade好还是直接调用模块好。

本质

何时选用外观模式

- 如果你希望为复杂的子系统提供一个简单接口的时候，可以考虑使用外观模式。使用外观对象对实现大部分客户需要的功能，从而简化客户的使用。
- 如果想要让客户程序和抽象类的实现部分松散耦合，可以考虑使用外观模式，使用外观对象来将这个子系统与它的客户分离开来，从而提高子系统的独立性和可移植性。
- 如果构建多层结构的系统，可以考虑使用外观模式，使用外观对象作为每层的入口，这样就可以简化层间调用，也可以松散层次之间的依赖关系。

Android(安卓面试点)

Android基础知识

Android：

五种布局： FrameLayout 、 LinearLayout 、 AbsoluteLayout 、 RelativeLayout 、 TableLayout 全都继承自ViewGroup，各自特点及绘制效率对比。

- FrameLayout(框架布局)

  此布局是五中布局中最简单的布局，Android中并没有对child view的摆布进行控制，这个布局中所有的控件都会默认出现在视图的左上角，我们可以使用 android:layout_margin ， android:layout_gravity 等属性去控制子控件相对布局的位置。

- LinearLayout(线性布局)

  一行只控制一个控件的线性布局，所以当有很多控件需要在一个界面中列出时，可以用LinearLayout布局。
  此布局有一个需要格外注意的属性: android:orientation= “horizontal|vertical 。

  - 当 android:orientation="horizontal 时，*说明你希望将水平方向的布局交给LinearLayout*，其子元素的 android:layout_gravity="right|left" 等控制水平方向的gravity值都是被忽略的*，此时LinearLayout中的子元素都是默认的按照水平从左向右来排*，我们可以用 android:layout_gravity="top|bottom" 等gravity值来控制垂直展示。
  - 反之，可以知道 当 android:orientation="vertical 时，LinearLayout对其子元素展示上的的处理方式。

- AbsoluteLayout(绝对布局)

  可以放置多个控件，并且可以自己定义控件的x,y位置

- RelativeLayout(相对布局)

  这个布局也是相对自由的布局，Android 对该布局的child view的 水平layout& 垂直layout做了解析，由此我们可以FrameLayout的基础上使用标签或者Java代码对垂直方向 以及 水平方向 布局中的views任意的控制.

  - 相关属性：

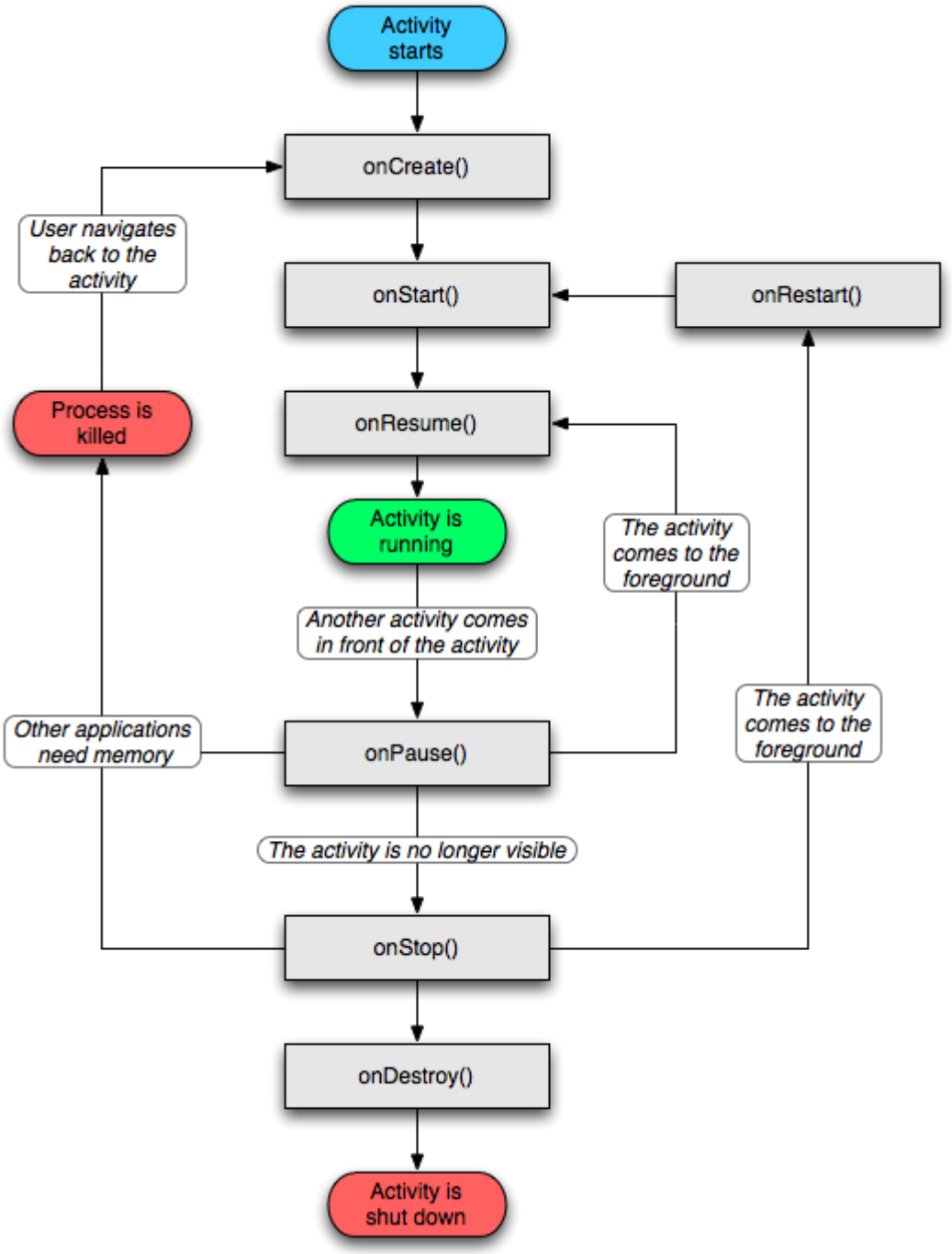
```
android:layout_centerInParent="true|false"
android:layout_centerHorizontal="true|false"
android:layout_alignParentRight="true|false"
```

- TableLayout(表格布局)

  将子元素的位置分配到行或列中，一个TableLayout由许多的TableRow组成

Activity生命周期。

- 启动Activity:
  onCreate()—>onStart()—>onResume()，Activity进入运行状态。
- Activity退居后台:
  当前Activity转到新的Activity界面或按Home键回到主屏：
  onPause()—>onStop()，进入停滞状态。
- Activity返回前台:
  onRestart()—>onStart()—>onResume()，再次回到运行状态。
- Activity退居后台，且系统内存不足，
  系统会杀死这个后台状态的Activity（此时这个Activity引用仍然处在任务栈中，只是这个时候引用指向的对象已经为null），若再次回到这个Activity,则会走onCreate()–>onStart()—>onResume()(将重新走一次Activity的初始化生命周期)
- 锁定屏与解锁屏幕
  只会调用onPause()，而不会调用onStop()方法，开屏后则调用onResume()
- 更多流程分支，请参照以下生命周期流程图



通过Acitivty的xml标签来改变任务栈的默认行为

- 使用 android:launchMode="standard|singleInstance|singleTask|singleTop" 来控制Acivity任务栈。

  任务栈是一种后进先出的结构。位于栈顶的Activity处于焦点状态,当按下back按钮的时候,栈内的Activity会一个一个的出栈,并且调用其 onDestory() 方法。如果栈内没有Activity,那么系统就会回收这个栈,每个APP默认只有一个栈,以APP的包名来命名.

  - standard : 标准模式,每次启动Activity都会创建一个新的Activity实例,并且将其压入任务栈栈顶,而不管这个Activity是否已经存在。Activity的启动三回调(*onCreate()->onStart()->onResume()*)都会执行。
  - singleTop : 栈顶复用模式.这种模式下,如果新Activity已经位于任务栈的栈顶,那么此Activity不会

被重新创建,所以它的启动三回调就不会执行,同时Activity的 `onNewIntent()` 方法会被回调.如果Activity已经存在但是不在栈顶,那么作用于*standard模式*一样.

- singleTask: 栈内复用模式.创建这样的Activity的时候,系统会先确认它所需任务栈已经创建,否则先创建任务栈.然后放入Activity,如果栈中已经有一个Activity实例,那么这个Activity就会被调到栈顶, `onNewIntent()` ,并且singleTask会清理在当前Activity上面的所有Activity.(clear top)
- singleInstance : 加强版的singleTask模式,这种模式的Activity只能单独位于一个任务栈内,由于栈内复用的特性,后续请求均不会创建新的Activity,除非这个独特的任务栈被系统销毁了

Activity的堆栈管理以ActivityRecord为单位,所有的ActivityRecord都放在一个List里面.可以认为一个ActivityRecord就是一个Activity栈

Activity缓存方法。

有a、b两个Activity，当从a进入b之后一段时间，可能系统会把a回收，这时候按back，执行的不是a的onRestart而是onCreate方法，a被重新创建一次，这是a中的临时数据和状态可能就丢失了。

可以用Activity中的onSaveInstanceState()回调方法保存临时数据和状态，这个方法一定会在活动被回收之前调用。方法中有一个Bundle参数，putString()、putInt()等方法需要传入两个参数，一个键一个值。数据保存之后会在onCreate中恢复，onCreate也有一个Bundle类型的参数。

示例代码：

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

    //这里，当Acivity第一次被创建的时候为空
    //所以我们需要判断一下
    if( savedInstanceState != null ){
        savedInstanceState.getString("anAnt");
    }
}

@Override
protected void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);

    outState.putString("anAnt","Android");

}
```

一、onSaveInstanceState (Bundle outState)

当某个activity变得“容易”被系统销毁时，该activity的onSaveInstanceState就会被执行，除非该activity是被用户主动销毁的，例如当用户按BACK键的时候。

注意上面的双引号，何为“容易”？言下之意就是该activity还没有被销毁，而仅仅是一种可能性。这种可能性有哪些？通过重写一个activity的所有生命周期的onXXX方法，包括onSaveInstanceState和onRestoreInstanceState方法，我们可以清楚地知道当某个activity（假定为activity A）显示在当前task的最上层时，其onSaveInstanceState方法会在什么时候被执行，有这么几种情况：

1、当用户按下HOME键时。

这是显而易见的，系统不知道你按下HOME后要运行多少其他的程序，自然也不知道activity A是否会被销毁，故系统会调用onSaveInstanceState，让用户有机会保存某些非永久性的数据。以下几种情况的分析都遵循该原则

2、长按HOME键，选择运行其他的程序时。

3、按下电源按键（关闭屏幕显示）时。

4、从activity A中启动一个新的activity时。

5、屏幕方向切换时，例如从竖屏切换到横屏时。（如果不指定configchange属性）
在屏幕切换之前，系统会销毁activity A，在屏幕切换之后系统又会自动地创建activity A，所以onSaveInstanceState一定会被执行

总而言之，onSaveInstanceState的调用遵循一个重要原则，即当系统“未经你许可”时销毁了你的activity，则onSaveInstanceState会被系统调用，这是系统的责任，因为它必须要提供一个机会让你保存你的数据（当然你不保存那就随便你了）。另外，需要注意的几点：

1.布局中的每一个View默认实现了onSaveInstanceState()方法，这样的话，这个UI的任何改变都会自动的存储和在activity重新创建的时候自动的恢复。但是这种情况只有在你为这个UI提供了唯一的ID之后才起作用，如果没有提供ID，将不会存储它的状态。

2.由于默认的onSaveInstanceState()方法的实现帮助UI存储它的状态，所以如果你需要覆盖这个方法去存储额外的状态信息时，你应该在执行任何代码之前都调用父类的onSaveInstanceState()方法（super.onSaveInstanceState()）。
既然有现成的可用，那么我们到底还要不要自己实现onSaveInstanceState()?这得看情况了，如果你自己的派生类中有变量影响到UI，或你程序的行为，当然就要把这个变量也保存了，那么就需要自己实现，否则就不需要。

3.由于onSaveInstanceState()方法调用的不确定性，你应该只使用这个方法去记录activity的瞬间状态（UI的状态）。不应该用这个方法去存储持久化数据。当用户离开这个activity的时候应该在onPause()方法中存储持久化数据（例如应该被存储到数据库中的数据）。

4.onSaveInstanceState()如果被调用，这个方法会在onStop()前被触发，但系统并不保证是否在onPause()之前或者之后触发。

二、onRestoreInstanceState (Bundle outState)

至于onRestoreInstanceState方法，需要注意的是，onSaveInstanceState方法和onRestoreInstanceState方法“不一定”是成对的被调用的，（本人注：我昨晚调试时就发现原来不一定成对被调用的！）

onRestoreInstanceState被调用的前提是，activity A“确实”被系统销毁了，而如果仅仅是停留在有这种可能性的情况下，则该方法不会被调用，例如，当正在显示activity A的时候，用户按下HOME键回到主界面，然后用户紧接着又返回到activity A，这种情况下activity A一般不会因为内存的原因被系统销毁，故activity A的onRestoreInstanceState方法不会被执行

另外，onRestoreInstanceState的bundle参数也会传递到onCreate方法中，你也可以选择在onCreate方法中做数据还原。
还有onRestoreInstanceState在onstart之后执行。
至于这两个函数的使用，给出示范代码（留意自定义代码在调用super的前或后）：

```
@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
        savedInstanceState.putBoolean("MyBoolean", true);
        savedInstanceState.putDouble("myDouble", 1.9);
        savedInstanceState.putInt("MyInt", 1);
        savedInstanceState.putString("MyString", "Welcome back to Android");
        // etc.
        super.onSaveInstanceState(savedInstanceState);
}

@Override
public void onRestoreInstanceState(Bundle savedInstanceState) {
        super.onRestoreInstanceState(savedInstanceState);

        boolean myBoolean = savedInstanceState.getBoolean("MyBoolean");
        double myDouble = savedInstanceState.getDouble("myDouble");
        int myInt = savedInstanceState.getInt("MyInt");
        String myString = savedInstanceState.getString("MyString");
}
```

Fragment的生命周期和activity如何的一个关系

这我们引用本知识库里的一张图片：

为什么在Service中创建子线程而不是Activity中

这是因为Activity很难对Thread进行控制，当Activity被销毁之后，就没有任何其它的办法可以再重新获取到之前创建的子线程的实例。而且在一个Activity中创建的子线程，另一个Activity无法对其进行操作。但是Service就不同了，所有的Activity都可以与Service进行关联，然后可以很方便地操作其中的方法，即使Activity被销毁了，之后只要重新与Service建立关联，就又能够获取到原有的Service中Binder的实例。因此，使用Service来处理后台任务，Activity就可以放心地finish，完全不需要担心无法对后台任务进行控制

的情况。

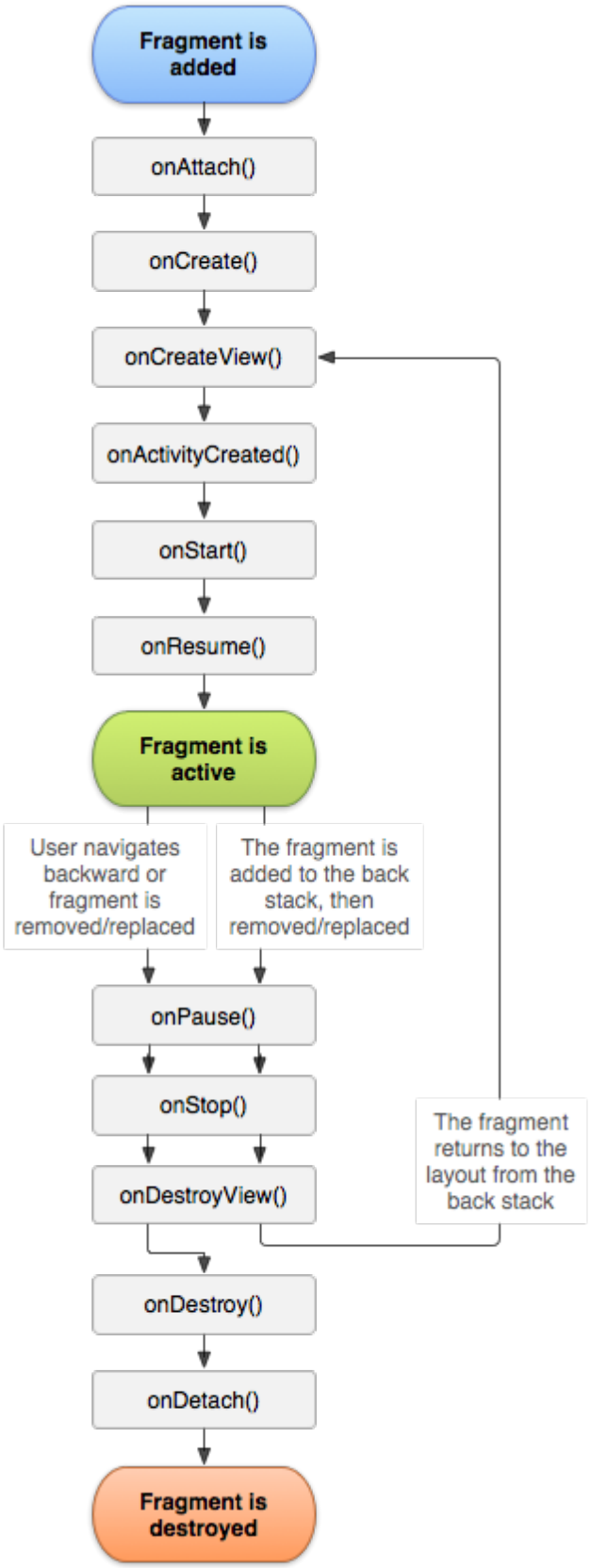
Intent的使用方法，可以传递哪些数据类型。

通过查询Intent/Bundle的API文档，我们可以获知，Intent/Bundle支持传递基本类型的数据和基本类型的数组数据，以及String/CharSequence类型的数据和String/CharSequence类型的数组数据。而对于其它类型的数据貌似无能为力，其实不然，我们可以在Intent/Bundle的API中看到Intent/Bundle还可以传递Parcelable（包裹化，邮包）和Serializable（序列化）类型的数据，以及它们的数组/列表数据。

所以要让非基本类型和非String/CharSequence类型的数据通过Intent/Bundle来进行传输，我们就需要在数据类型中实现Parcelable接口或是Serializable接口。

http://blog.csdn.net/kkk0526/article/details/7214247

Fragment生命周期



注意和Activity的相比的区别,按照执行顺序

- onAttach(),onDetach()
- onCreateView(),onDestroyView()

Service的两种启动方法，有什么区别 1.在Context中通过 `public boolean bindService(Intent service,ServiceConnection conn,int flags)` 方法来进行Service与

Context的关联并启动，并且Service的生命周期依附于Context(不求同时同分同秒生！但求同时同分同秒屎！！)。

2.通过 public ComponentName startService(Intent service) 方法去启动一个Service，此时Service的生命周期与启动它的Context无关。

3.要注意的是，whatever，都需要在xml里注册你的Service，就像这样:

```
<service
      android:name=".packnameName.youServiceName"
      android:enabled="true" />
```

广播(Boardcast Receiver)的两种动态注册和静态注册有什么区别。

- 静态注册：在AndroidManifest.xml文件中进行注册，当App退出后，Receiver仍然可以接收到广播并且进行相应的处理
- 动态注册：在代码中动态注册，当App退出后，也就没办法再接受广播了

ContentProvider使用方法

http://blog.csdn.net/juetion/article/details/17481039

目前能否保证service不被杀死

Service设置成START_STICKY

- kill 后会被重启（等待5秒左右），重传Intent，保持与重启前一样

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。
- 【结论】目前看来，priority这个属性貌似只适用于broadcast，对于Service来说可能无效

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
- 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，可以在startForeground()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。
- 【结论】如果在极度极度低内存的压力下，该service还是会被kill掉，并且不一定会restart()

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

- 也可以直接在onDestroy()里startService
- 【结论】当使用类似口口管家等第三方应用或是在setting里-应用-强制停止时，APP进程可能就直接被干掉了，onDestroy方法都进不来，所以还是无法保证

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
- 【结论】这也能算是一种措施，不过感觉监听多了会导致Service很混乱，带来诸多不便

在JNI层,用C代码fork一个进程出来

- 这样产生的进程,会被系统认为是两个不同的进程.但是Android5.0之后可能不行

root之后放到system/app变成系统级应用

大招: 放一个像素在前台(手机QQ)

动画有哪两类，各有什么特点？三种动画的区别

- tween 补间动画。通过指定View的初末状态和变化时间、方式，对View的内容完成一系列的图形变换来实现动画效果。
  Alpha
  Scale
  Translate
  Rotate。
- frame 帧动画
  AnimationDrawable 控制
  animation-list xml布局
- PropertyAnimation 属性动画

Android的数据存储形式。

- SQLite：SQLite是一个轻量级的数据库，支持基本的SQL语法，是常被采用的一种数据存储方式。Android为此数据库提供了一个名为SQLiteDatabase的类，封装了一些操作数据库的api
- SharedPreference： 除SQLite数据库外，另一种常用的数据存储方式，其本质就是一个xml文件，常用于存储较简单的参数设置。
- File： 即常说的文件（I/O）存储方法，常用语存储大数量的数据，但是缺点是更新数据将是一件困难的事情。
- ContentProvider: Android系统中能实现所有应用程序共享的一种数据存储方式，由于数据通常在各应用间的是互相私密的，所以此存储方式较少使用，但是其又是必不可少的一种存储方式。例如音

频，视频，图片和通讯录，一般都可以采用此种方式进行存储。每个Content Provider都会对外提供一个公共的URI（包装成Uri对象），如果应用程序有数据需要共享时，就需要使用Content Provider为这些数据定义一个URI，然后其他的应用程序就通过Content Provider传入这个URI来对数据进行操作。

Sqlite的基本操作。

http://blog.csdn.net/zgljl2012/article/details/44769043

如何判断应用被强杀

在Applicatio中定义一个static常量，赋值为 - 1，在欢迎界面改为0，如果被强杀，application重新初始化，在父类Activity判断该常量的值。

应用被强杀如何解决

如果在每一个Activity的onCreate里判断是否被强杀，冗余了，封装到Activity的父类中，如果被强杀，跳转回主界面，如果没有被强杀，执行Activity的初始化操作，给主界面传递intent参数，主界面会调用onNewIntent方法，在onNewIntent跳转到欢迎页面，重新来一遍流程。

Json有什么优劣势。

怎样退出终止App

Asset目录与res目录的区别。

Android怎么加速启动Activity。

Android内存优化方法：ListView优化，及时关闭资源，图片缓存等等。

Android中弱引用与软引用的应用场景。

Bitmap的四种属性，与每种属性队形的大小。

View与View Group分类。自定义View过程：onMeasure()、onLayout()、onDraw()。

如何自定义控件：

1. 自定义属性的声明和获取
    - 分析需要的自定义属性
    - 在res/values/attrs.xml定义声明
    - 在layout文件中进行使用
    - 在View的构造方法中进行获取
2. 测量onMeasure
3. 布局onLayout(ViewGroup)

4. 绘制onDraw
5. onTouchEvent
6. onInterceptTouchEvent(ViewGroup)
7. 状态的恢复与保存

Android长连接，怎么处理心跳机制。

View树绘制流程

下拉刷新实现原理

你用过什么框架，是否看过源码，是否知道底层原理。

Retrofit

EventBus

glide

Android5.0、6.0新特性。

Android5.0新特性：

- MaterialDesign设计风格
- 支持多种设备
- 支持64位ART虚拟机

Android6.0新特性

- 大量漂亮流畅的动画
- 支持快速充电的切换
- 支持文件夹拖拽应用
- 相机新增专业模式

Android7.0新特性

- 分屏多任务
- 增强的Java8语言模式
- 夜间模式

Context区别

- Activity和Service以及Application的Context是不一样的,Activity继承自ContextThemeWraper.其他的继承自ContextWrapper

- 每一个Activity和Service以及Application的Context都是一个新的ContextImpl对象
- getApplication()用来获取Application实例的，但是这个方法只有在Activity和Service中才能调用的到。那么也许在绝大多数情况下我们都是在Activity或者Service中使用Application的，但是如果在一些其它的场景，比如BroadcastReceiver中也想获得Application的实例，这时就可以借助getApplicationContext()方法，getApplicationContext()比getApplication()方法的作用域会更广一些，任何一个Context的实例，只要调用getApplicationContext()方法都可以拿到我们的Application对象。
- Activity在创建的时候会new一个ContextImpl对象并在attach方法中关联它，Application和Service也差不多。ContextWrapper的方法内部都是转调ContextImpl的方法
- 创建对话框传入Application的Context是不可以的
- 尽管Application、Activity、Service都有自己的ContextImpl，并且每个ContextImpl都有自己的mResources成员，但是由于它们的mResources成员都来自于唯一的ResourcesManager实例，所以它们看似不同的mResources其实都指向的是同一块内存
- Context的数量等于Activity的个数 + Service的个数 + 1，这个1为Application

IntentService的使用场景与特点。

> IntentService是Service的子类，是一个异步的，会自动停止的服务，很好解决了传统的Service中处理完耗时操作忘记停止并销毁Service的问题

优点：

- 一方面不需要自己去new Thread
- 另一方面不需要考虑在什么时候关闭该Service

onStartCommand中回调了onStart，onStart中通过mServiceHandler发送消息到该handler的handleMessage中去。最后handleMessage中回调onHandleIntent(intent)。

图片缓存

查看每个应用程序最高可用内存：

```
int maxMemory = (int) (Runtime.getRuntime().maxMemory() / 1024);
Log.d("TAG", "Max memory is " + maxMemory + "KB");
```

Gradle

构建工具、Groovy语法、Java

Jar包里面只有代码，aar里面不光有代码还包括

你是如何自学Android

首先是看书和看视频敲代码，然后看大牛的博客，做一些项目，向github提交代码，觉得自己API掌握的不错之后，开始看进阶的书，以及看源码，看完源码学习到一些思想，开始自己造轮子，开始想代码的提升，比如设计模式，架构，重构等。

Android内存泄漏总结

Android 内存泄漏总结

内存管理的目的就是让我们在开发中怎么有效的避免我们的应用出现内存泄漏的问题。内存泄漏大家都不陌生了，简单粗俗的讲，就是该被释放的对象没有释放，一直被某个或某些实例所持有却不再被使用导致 GC 不能回收。最近自己阅读了大量相关的文档资料，打算做个 总结 沉淀下来跟大家一起分享和学习，也给自己一个警示，以后 coding 时怎么避免这些情况，提高应用的体验和质量。

我会从 java 内存泄漏的基础知识开始，并通过具体例子来说明 Android 引起内存泄漏的各种原因，以及如何利用工具来分析应用内存泄漏，最后再做总结。

Java 内存分配策略

Java 程序运行时的内存分配策略有三种,分别是静态分配,栈式分配,和堆式分配，对应的，三种存储策略使用的内存空间主要分别是静态存储区（也称方法区）、栈区和堆区。

- 静态存储区（方法区）：主要存放静态数据、全局 static 数据和常量。这块内存在程序编译时就已经分配好，并且在程序整个运行期间都存在。
- 栈区 ：当方法被执行时，方法体内的局部变量（其中包括基础数据类型、对象的引用）都在栈上创建，并在方法执行结束时这些局部变量所持有的内存将会自动被释放。因为栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。
- 堆区 ： 又称动态内存分配，通常就是指在程序运行时直接 new 出来的内存，也就是对象的实例。这部分内存在不使用时将会由 Java 垃圾回收器来负责回收。

栈与堆的区别：

在方法体内定义的（局部变量）一些基本类型的变量和对象的引用变量都是在方法的栈内存中分配的。当在一段方法块中定义一个变量时，Java 就会在栈中为该变量分配内存空间，当超过该变量的作用域后，该变量也就无效了，分配给它的内存空间也将被释放掉，该内存空间可以被重新使用。

堆内存用来存放所有由 new 创建的对象（包括该对象其中的所有成员变量）和数组。在堆中分配的内存，将由 Java 垃圾回收器来自动管理。在堆中产生了一个数组或者对象后，还可以在栈中定义一个特殊的变量，这个变量的取值等于数组或者对象在堆内存中的首地址，这个特殊的变量就是我们上面说的引用变

量。我们可以通过这个引用变量来访问堆中的对象或者数组。

举个例子:

```
public class Sample() {
    int s1 = 0;
    Sample mSample1 = new Sample();

    public void method() {
        int s2 = 1;
        Sample mSample2 = new Sample();
    }
}

Sample mSample3 = new Sample();
```

Sample 类的局部变量 s2 和引用变量 mSample2 都是存在于栈中，但 mSample2 指向的对象是存在于堆上的。
mSample3 指向的对象实体存放在堆上，包括这个对象的所有成员变量 s1 和 mSample1，而它自己存在于栈中。

结论：

局部变量的基本数据类型和引用存储于栈中，引用的对象实体存储于堆中。—— 因为它们属于方法中的变量，生命周期随方法而结束。

成员变量全部存储与堆中（包括基本数据类型，引用和引用的对象实体）—— 因为它们属于类，类对象终究是要被new出来使用的。

了解了 Java 的内存分配之后，我们再来看看 Java 是怎么管理内存的。

Java是如何管理内存

Java的内存管理就是对象的分配和释放问题。在 Java 中，程序员需要通过关键字 new 为每个对象申请内存空间 (基本类型除外)，所有的对象都在堆 (Heap)中分配空间。另外，对象的释放是由 GC 决定和执行的。在 Java 中，内存的分配是由程序完成的，而内存的释放是由 GC 完成的，这种收支两条线的方法确实简化了程序员的工作。但同时，它也加重了JVM的工作。这也是 Java 程序运行速度较慢的原因之一。因为，GC 为了能够正确释放对象，GC 必须监控每一个对象的运行状态，包括对象的申请、引用、被引用、赋值等，GC 都需要进行监控。

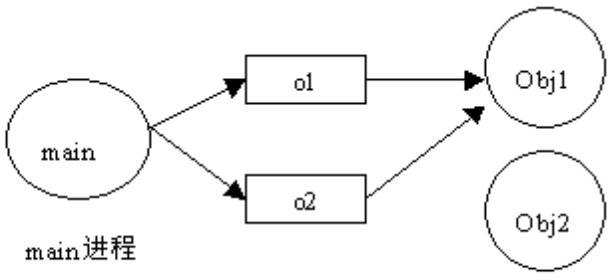
监视对象状态是为了更加准确地、及时地释放对象，而释放对象的根本原则就是该对象不再被引用。

为了更好理解 GC 的工作原理，我们可以将对象考虑为有向图的顶点，将引用关系考虑为图的有向边，有向边从引用者指向被引对象。另外，每个线程对象可以作为一个图的起始顶点，例如大多程序从 main 进程开始执行，那么该图就是以 main 进程顶点开始的一棵根树。在这个有向图中，根顶点可达的对象都是有效对象，GC将不回收这些对象。如果某个对象 (连通子图)与这个根顶点不可达(注意，该图为有向图)，

那么我们认为这个(这些)对象不再被引用，可以被 GC 回收。

以下，我们举一个例子说明如何用有向图表示内存管理。对于程序的每一个时刻，我们都有一个有向图表示JVM的内存分配情况。以下右图，就是左边程序运行到第6行的示意图。

```
class test{
 Public static void  main(String a[]){
          Object  o1 =new Object();
          Object  o2 =new Object();
          o2=o1;
          //此行为第 6 行
          }
}
```



该图描述了第 6 行的内存管理的有向图，
Obj2 是第二次申请的对象，此时为可回收对象

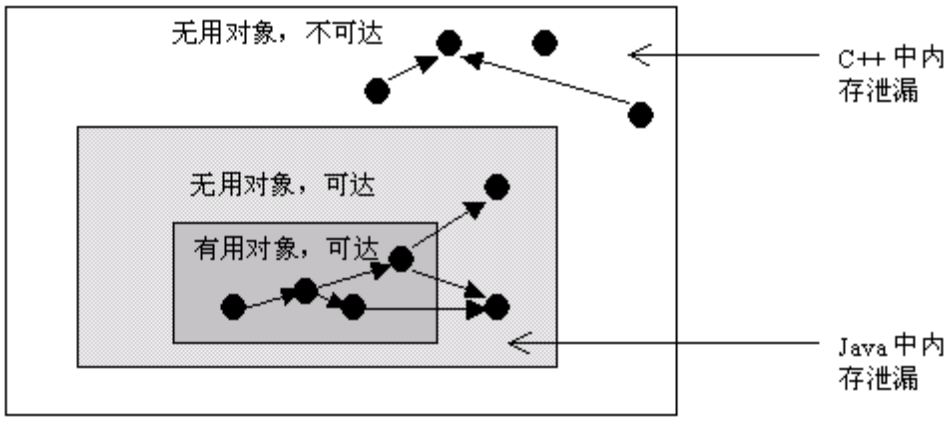
Java使用有向图的方式进行内存管理，可以消除引用循环的问题，例如有三个对象，相互引用，只要它们和根进程不可达的，那么GC也是可以回收它们的。这种方式的优点是管理内存的精度很高，但是效率较低。另外一种常用的内存管理技术是使用计数器，例如COM模型采用计数器方式管理构件，它与有向图相比，精度行低(很难处理循环引用的问题)，但执行效率很高。

什么是Java中的内存泄露

在Java中，内存泄漏就是存在一些被分配的对象，这些对象有下面两个特点，首先，这些对象是可达的，即在有向图中，存在通路可以与其相连；其次，这些对象是无用的，即程序以后不会再使用这些对象。如果对象满足这两个条件，这些对象就可以判定为Java中的内存泄漏，这些对象不会被GC所回收，然而它却占用内存。

在C++中，内存泄漏的范围更大一些。有些对象被分配了内存空间，然后却不可达，由于C++中没有GC，这些内存将永远收不回来。在Java中，这些不可达的对象都由GC负责回收，因此程序员不需要考虑这部分的内存泄露。

通过分析，我们得知，对于C++，程序员需要自己管理边和顶点，而对于Java程序员只需要管理边就可以了(不需要管理顶点的释放)。通过这种方式，Java提高了编程的效率。



因此，通过以上分析，我们知道在Java中也有内存泄漏，但范围比C++要小一些。因为Java从语言上保

证，任何对象都是可达的，所有的不可达对象都由GC管理。

对于程序员来说，GC基本是透明的，不可见的。虽然，我们只有几个函数可以访问GC，例如运行GC的函数System.gc()，但是根据Java语言规范定义， 该函数不保证JVM的垃圾收集器一定会执行。因为，不同的JVM实现者可能使用不同的算法管理GC。通常，GC的线程的优先级别较低。JVM调用GC的策略也有很多种，有的是内存使用到达一定程度时，GC才开始工作，也有定时执行的，有的是平缓执行GC，有的是中断式执行GC。但通常来说，我们不需要关心这些。除非在一些特定的场合，GC的执行影响应用程序的性能，例如对于基于Web的实时系统，如网络游戏等，用户不希望GC突然中断应用程序执行而进行垃圾回收，那么我们需要调整GC的参数，让GC能够通过平缓的方式释放内存，例如将垃圾回收分解为一系列的小步骤执行，Sun提供的HotSpot JVM就支持这一特性。

同样给出一个 Java 内存泄漏的典型例子，

```
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，我们循环申请Object对象，并将所申请的对象放入一个 Vector 中，如果我们仅仅释放引用本身，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。

详细Java中的内存泄漏

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。

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。j

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、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。

例如

```
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、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。

例如：

```
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、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、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、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、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

Android中常见的内存泄漏汇总

集合类泄漏

集合类如果仅仅有添加元素的方法，而没有相应的删除机制，导致内存被占用。如果这个集合类是全局性的变量 (比如类中的静态属性，全局性的 map 等即有静态引用或 final 一直指向它)，那么没有相应的删除机制，很可能导致集合所占用的内存只增不减。比如上面的典型例子就是其中一种情况，当然实际上我们在项目中肯定不会写这么 2B 的代码，但稍不注意还是很容易出现这种情况，比如我们都喜欢通过 HashMap 做一些缓存之类的事，这种情况就要多留一些心眼。

单例造成的内存泄漏

由于单例的静态特性使得其生命周期跟应用的生命周期一样长，所以如果使用不恰当的话，很容易造成内存泄漏。比如下面一个典型的例子，

```
public class AppManager {
private static AppManager instance;
private Context context;
private AppManager(Context context) {
this.context = context;
}
public static AppManager getInstance(Context context) {
if (instance == null) {
instance = new AppManager(context);
}
return instance;
}
}
```

这是一个普通的单例模式，当创建这个单例的时候，由于需要传入一个Context，所以这个Context的生命周期的长短至关重要：

1、如果此时传入的是 Application 的 Context，因为 Application 的生命周期就是整个应用的生命周期，所以这将没有任何问题。

2、如果此时传入的是 Activity 的 Context，当这个 Context 所对应的 Activity 退出时，由于该 Context 的引用被单例对象所持有，其生命周期等于整个应用程序的生命周期，所以当前 Activity 退出时它的内存并不会被回收，这就造成泄漏了。

正确的方式应该改为下面这种方式：

```
public class AppManager {
private static AppManager instance;
private Context context;
private AppManager(Context context) {
this.context = context.getApplicationContext();// 使用Application 的context
}
public static AppManager getInstance(Context context) {
if (instance == null) {
instance = new AppManager(context);
}
return instance;
}
}
```

或者这样写，连 Context 都不用传进来了：

```
在你的 Application 中添加一个静态方法，getContext() 返回 Application 的 context，

...

context = getApplicationContext();

...
   /**
     * 获取全局的context
     * @return 返回全局context对象
     */
    public static Context getContext(){
        return context;
    }

public class AppManager {
private static AppManager instance;
private Context context;
private AppManager() {
this.context = MyApplication.getContext();// 使用Application 的context
}
public static AppManager getInstance() {
if (instance == null) {
instance = new AppManager();
}
return instance;
}
}
```

匿名内部类/非静态内部类和异步线程

非静态内部类创建静态实例造成的内存泄漏

有的时候我们可能会在启动频繁的Activity中，为了避免重复创建相同的数据资源，可能会出现这种写法：

```
public class MainActivity extends AppCompatActivity {
private static TestResource mResource = null;
@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_main);
if(mManager == null){
mManager = new TestResource();
}
//...
}
class TestResource {
//...
}
}
```

这样就在Activity内部创建了一个非静态内部类的单例，每次启动Activity时都会使用该单例的数据，这样虽然避免了资源的重复创建，不过这种写法却会造成内存泄漏，因为非静态内部类默认会持有外部类的引用，而该非静态内部类又创建了一个静态的实例，该实例的生命周期和应用的一样长，这就导致了该静态实例一直会持有该Activity的引用，导致Activity的内存资源不能正常回收。正确的做法为：

将该内部类设为静态内部类或将该内部类抽取出来封装成一个单例，如果需要使用Context，请按照上面推荐的使用Application 的 Context。当然，Application 的 context 不是万能的，所以也不能随便乱用，对于有些地方则必须使用 Activity 的 Context，对于Application，Service，Activity三者的Context的应用场景如下：

功能	Application	Service	Activity
Start an Activity	NO1	NO1	YES
Show a Dialog	NO	NO	YES
Layout Inflation	YES	YES	YES
Start a Service	YES	YES	YES
Bind to a Service	YES	YES	YES
Send a Broadcast	YES	YES	YES
Register BroadcastReceiver	YES	YES	YES
Load Resource Values	YES	YES	YES

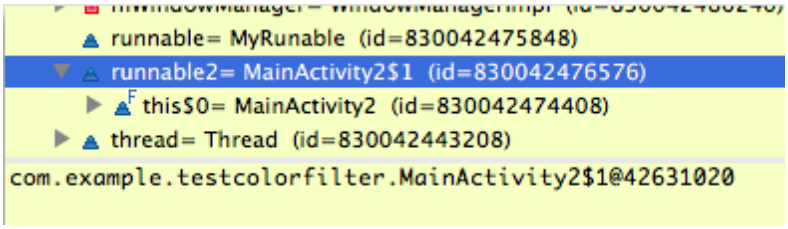
其中： NO1表示 Application 和 Service 可以启动一个 Activity，不过需要创建一个新的 task 任务队列。而对于 Dialog 而言，只有在 Activity 中才能创建

匿名内部类

android开发经常会继承实现Activity/Fragment/View，此时如果你使用了匿名类，并被异步线程持有了，那要小心了，如果没有任何措施这样一定会导致泄露

```
public class MainActivity extends Activity {
...
Runnable ref1 = new MyRunable();
Runnable ref2 = new Runnable() {
    @Override
    public void run() {

    }
};
  ...
}
```

ref1和ref2的区别是，ref2使用了匿名内部类。我们来看看运行时这两个引用的内存：



可以看到，ref1没什么特别的。

但ref2这个匿名类的实现对象里面多了一个引用：

this$0这个引用指向MainActivity.this，也就是说当前的MainActivity实例会被ref2持有，如果将这个引用再传入一个异步线程，此线程和此Acitivity生命周期不一致的时候，就造成了Activity的泄露。

Handler 造成的内存泄漏

Handler 的使用造成的内存泄漏问题应该说是最为常见了，很多时候我们为了避免 ANR 而不在主线程进行耗时操作，在处理网络任务或者封装一些请求回调等api都借助Handler来处理，但 Handler 不是万能的，对于 Handler 的使用代码编写一不规范即有可能造成内存泄漏。另外，我们知道 Handler、Message 和 MessageQueue 都是相互关联在一起的，万一 Handler 发送的 Message 尚未被处理，则该 Message 及发送它的 Handler 对象将被线程 MessageQueue 一直持有。

由于 Handler 属于 TLS(Thread Local Storage) 变量, 生命周期和 Activity 是不一致的。因此这种实现方式一般很难保证跟 View 或者 Activity 的生命周期保持一致，故很容易导致无法正确释放。

举个例子：

```
public class SampleActivity extends Activity {

private final Handler mLeakyHandler = new Handler() {
@Override
public void handleMessage(Message msg) {
  // ...
}
}

@Override
protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);

// Post a message and delay its execution for 10 minutes.
mLeakyHandler.postDelayed(new Runnable() {
  @Override
  public void run() { /* ... */ }
}, 1000 * 60 * 10);

// Go back to the previous Activity.
finish();
}
}
```

在该 SampleActivity 中声明了一个延迟10分钟执行的消息 Message，mLeakyHandler 将其 push 进了消息队列 MessageQueue 里。当该 Activity 被 finish() 掉时，延迟执行任务的 Message 还会继续存在于主线程中，它持有该 Activity 的 Handler 引用，所以此时 finish() 掉的 Activity 就不会被回收了从而造成内存泄漏（因 Handler 为非静态内部类，它会持有外部类的引用，在这里就是指 SampleActivity）。

修复方法：在 Activity 中避免使用非静态内部类，比如上面我们将 Handler 声明为静态的，则其存活期跟 Activity 的生命周期就无关了。同时通过弱引用的方式引入 Activity，避免直接将 Activity 作为 context 传进去，见下面代码：

```
public class SampleActivity extends Activity {

  /**
   * Instances of static inner classes do not hold an implicit
   * reference to their outer class.
   */
  private static class MyHandler extends Handler {
    private final WeakReference<SampleActivity> mActivity;

    public MyHandler(SampleActivity activity) {
      mActivity = new WeakReference<SampleActivity>(activity);
    }

    @Override
    public void handleMessage(Message msg) {
      SampleActivity activity = mActivity.get();
      if (activity != null) {
        // ...
      }
    }
  }

  private final MyHandler mHandler = new MyHandler(this);

  /**
   * Instances of anonymous classes do not hold an implicit
   * reference to their outer class when they are "static".
   */
  private static final Runnable sRunnable = new Runnable() {
      @Override
      public void run() { /* ... */ }
  };

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Post a message and delay its execution for 10 minutes.
    mHandler.postDelayed(sRunnable, 1000 * 60 * 10);

    // Go back to the previous Activity.
    finish();
  }
}
```

综述，即推荐使用静态内部类 + WeakReference 这种方式。每次使用前注意判空。

前面提到了 WeakReference，所以这里就简单的说一下 Java 对象的几种引用类型。

Java对引用的分类有 Strong reference, SoftReference, WeakReference, PhatomReference 四种。

级别	回收时机	用途	生存时间
强	从来不会	对象的一般状态	JVM停止运行时终止
软	在内存不足时	联合ReferenceQueue构造有效期短/占内存大/生命周期长的对象的二级高速缓冲器（内存不足才清空）	内存不足时终止
弱	在垃圾回收时	联合ReferenceQueue构造有效期短/占内存大/生命周期长的对象的一级高速缓冲器（系统发生gc则清空）	gc运行后终止
虚	在垃圾回收时	联合ReferenceQueue来跟踪对象被垃圾回收器回收的活动	gc运行后终止

在Android应用的开发中，为了防止内存溢出，在处理一些占用内存大而且声明周期较长的对象时候，可以尽量应用软引用和弱引用技术。

软/弱引用可以和一个引用队列（ReferenceQueue）联合使用，如果软引用所引用的对象被垃圾回收器回收，Java虚拟机就会把这个软引用加入到与之关联的引用队列中。利用这个队列可以得知被回收的软/弱引用的对象列表，从而为缓冲器清除已失效的软/弱引用。

假设我们的应用会用到大量的默认图片，比如应用中有默认的头像，默认游戏图标等等，这些图片很多地方会用到。如果每次都去读取图片，由于读取文件需要硬件操作，速度较慢，会导致性能较低。所以我们考虑将图片缓存起来，需要的时候直接从内存中读取。但是，由于图片占用内存空间比较大，缓存很多图片需要很多的内存，就可能比较容易发生OutOfMemory异常。这时，我们可以考虑使用软/弱引用技术来避免这个问题发生。以下就是高速缓冲器的雏形：

首先定义一个HashMap，保存软引用对象。

```
private Map <String, SoftReference<Bitmap>> imageCache = new HashMap <String, SoftReference<
Bitmap>> ();
```

再来定义一个方法，保存Bitmap的软引用到HashMap。

```
public class CacheBySoftRef {
    // 首先定义一个HashMap，保存软引用对象。
    private Map<String, SoftReference<Bitmap>> imageCache = new HashMap<String,
SoftReference<Bitmap>>();
    // 再来定义一个方法，保存Bitmap的软引用到HashMap。
    public void addBitmapToCache(String path) {
        // 强引用的Bitmap对象
        Bitmap bitmap = BitmapFactory.decodeFile(path);
        // 软引用的Bitmap对象
        SoftReference<Bitmap> softBitmap = new SoftReference<Bitmap>(bitmap);
        // 添加该对象到Map中使其缓存
        imageCache.put(path, softBitmap);
    }
    // 获取的时候，可以通过SoftReference的get()方法得到Bitmap对象。
    public Bitmap getBitmapByPath(String path) {
        // 从缓存中取软引用的Bitmap对象
        SoftReference<Bitmap> softBitmap = imageCache.get(path);
        // 判断是否存在软引用
        if (softBitmap == null) {
            return null;
        }
        // 通过软引用取出Bitmap对象，如果由于内存不足Bitmap被回收，将取得空 ，如果未被回收，则可重复使
用，提高速度。
        Bitmap bitmap = softBitmap.get();
        return bitmap;
    }
}
```

使用软引用以后，在OutOfMemory异常发生之前，这些缓存的图片资源的内存空间可以被释放掉的，从而避免内存达到上限，避免Crash发生。

如果只是想避免OutOfMemory异常的发生，则可以使用软引用。如果对于应用的性能更在意，想尽快回收一些占用内存比较大的对象，则可以使用弱引用。

另外可以根据对象是否经常使用来判断选择软引用还是弱引用。如果该对象可能会经常使用的，就尽量用软引用。如果该对象不被使用的可能性更大些，就可以用弱引用。

ok，继续回到主题。前面所说的，创建一个静态Handler内部类，然后对 Handler 持有的对象使用弱引用，这样在回收时也可以回收 Handler 持有的对象，但是这样做虽然避免了 Activity 泄漏，不过 Looper 线程的消息队列中还是可能会有待处理的消息，所以我们在 Activity 的 Destroy 时或者 Stop 时应该移除消息队列 MessageQueue 中的消息。

下面几个方法都可以移除 Message：

```
public final void removeCallbacks(Runnable r);

public final void removeCallbacks(Runnable r, Object token);

public final void removeCallbacksAndMessages(Object token);

public final void removeMessages(int what);

public final void removeMessages(int what, Object object);
```

尽量避免使用 static 成员变量

如果成员变量被声明为 static，那我们都知道其生命周期将与整个app进程生命周期一样。

这会导致一系列问题，如果你的app进程设计上是长驻内存的，那即使app切到后台，这部分内存也不会被释放。按照现在手机app内存管理机制，占内存较大的后台进程将优先回收，yi'wei如果此app做过进程互保保活，那会造成app在后台频繁重启。当手机安装了你参与开发的app以后一夜时间手机被消耗空了电量、流量，你的app不得不被用户卸载或者静默。

这里修复的方法是：

不要在类初始时初始化静态成员。可以考虑lazy初始化。
架构设计上要思考是否真的有必要这样做，尽量避免。如果架构需要这么设计，那么此对象的生命周期你有责任管理起来。

避免 override finalize()

1、finalize 方法被执行的时间不确定，不能依赖与它来释放紧缺的资源。时间不确定的原因是：
虚拟机调用GC的时间不确定
Finalize daemon线程被调度到的时间不确定

2、finalize 方法只会被执行一次，即使对象被复活，如果已经执行过了 finalize 方法，再次被 GC 时也不会再执行了，原因是：

含有 finalize 方法的 object 是在 new 的时候由虚拟机生成了一个 finalize reference 在来引用到该 Object的，而在 finalize 方法执行的时候，该 object 所对应的 finalize Reference 会被释放掉，即使在这个时候把该 object 复活(即用强引用引用住该 object)，再第二次被 GC 的时候由于没有了 finalize reference 与之对应，所以 finalize 方法不会再执行。

3、含有Finalize方法的object需要至少经过两轮GC才有可能被释放。

资源未关闭造成的内存泄漏

对于使用了BraodcastReceiver，ContentObserver，File，游标 Cursor，Stream，Bitmap等资源的使

用，应该在Activity销毁时及时关闭或者注销，否则这些资源将不会被回收，造成内存泄漏。

一些不良代码造成的内存压力

有些代码并不造成内存泄露，但是它们，或是对没使用的内存没进行有效及时的释放，或是没有有效的利用已有的对象而是频繁的申请新内存。

比如：
Bitmap 没调用 recycle()方法，对于 Bitmap 对象在不使用时,我们应该先调用 recycle() 释放内存，然后才它设置为 null. 因为加载 Bitmap 对象的内存空间，一部分是 java 的，一部分 C 的（因为 Bitmap 分配的底层是通过 JNI 调用的）。 而这个 recyle() 就是针对 C 部分的内存释放。
构造 Adapter 时，没有使用缓存的 convertView ,每次都在创建新的 converView。这里推荐使用 ViewHolder。

总结

对 Activity 等组件的引用应该控制在 Activity 的生命周期之内； 如果不能就考虑使用 getApplicationContext 或者 getApplication，以避免 Activity 被外部长生命周期的对象引用而泄露。

尽量不要在静态变量或者静态内部类中使用非静态外部成员变量（包括context），即使要使用，也要考虑适时把外部成员变量置空；也可以在内部类中使用弱引用来引用外部类的变量。

对于生命周期比Activity长的内部类对象，并且内部类中使用了外部类的成员变量，可以这样做避免内存泄漏：

> 将内部类改为静态内部类
> 静态内部类中使用弱引用来引用外部类的成员变量

Handler 的持有的引用对象最好使用弱引用，资源释放时也可以清空 Handler 里面的消息。比如在 Activity onStop 或者 onDestroy 的时候，取消掉该 Handler 对象的 Message和 Runnable.

在 Java 的实现过程中，也要考虑其对象释放，最好的方法是在不使用某对象时，显式地将此对象赋值为 null，比如使用完Bitmap 后先调用 recycle()，再赋为null,清空对图片等资源有直接引用或者间接引用的数组（使用 array.clear()；array = null）等，最好遵循谁创建谁释放的原则。

正确关闭资源，对于使用了BraodcastReceiver，ContentObserver，File，游标 Cursor，Stream，Bitmap等资源的使用，应该在Activity销毁时及时关闭或者注销。

保持对对象生命周期的敏感，特别注意单例、静态对象、全局性集合等的生命周期。

Handler内存泄漏分析及解决

Handler内存泄漏分析及解决

一、介绍

首先，请浏览下面这段handler代码：

```
public class SampleActivity extends Activity {
  private final Handler mLeakyHandler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
      // ...
    }
  }
}
```

在使用handler时，这是一段很常见的代码。但是，它却会造成严重的内存泄漏问题。在实际编写中，我们往往会得到如下警告：

> ⚠ In Android, Handler classes should be static or leaks might occur.

二、分析

1、 Android角度

当Android应用程序启动时，framework会为该应用程序的主线程创建一个Looper对象。这个Looper对象包含一个简单的消息队列Message Queue，并且能够循环的处理队列中的消息。这些消息包括大多数应用程序framework事件，例如Activity生命周期方法调用、button点击等，这些消息都会被添加到消息队列中并被逐个处理。

另外，主线程的Looper对象会伴随该应用程序的整个生命周期。

然后，当主线程里，实例化一个Handler对象后，它就会自动与主线程Looper的消息队列关联起来。所有发送到消息队列的消息Message都会拥有一个对Handler的引用，所以当Looper来处理消息时，会据此回调[Handler#handleMessage(Message)]方法来处理消息。

2、 Java角度

在java里，非静态内部类 和 匿名类 都会潜在的引用它们所属的外部类。但是，静态内部类却不会。

三、泄漏来源

请浏览下面一段代码：

```
public class SampleActivity extends Activity {

  private final Handler mLeakyHandler = new Handler() {
    @Override
    public void handleMessage(Message msg) {
      // ...
    }
  }

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Post a message and delay its execution for 10 minutes.
    mLeakyHandler.postDelayed(new Runnable() {
      @Override
      public void run() { /* ... */ }
    }, 1000 * 60 * 10);

    // Go back to the previous Activity.
    finish();
  }
}
```

当activity结束(finish)时，里面的延时消息在得到处理前，会一直保存在主线程的消息队列里持续10分钟。而且，由上文可知，这条消息持有对handler的引用，而handler又持有对其外部类（在这里，即SampleActivity）的潜在引用。这条引用关系会一直保持直到消息得到处理，从而，这阻止了SampleActivity被垃圾回收器回收，同时造成应用程序的泄漏。

<<<<<<< HEAD
注意，上面代码中的Runnable类--非静态匿名类--同样持有对其外部类的引用。从而也导致泄漏。

=======
注意，上面代码中的Runnable类--非静态匿名类--同样持有对其外部类的引用。从而也导致泄漏。

c67abfcfd66909095068cb5f0c8632dc5547131b

四、泄漏解决方案

首先，上面已经明确了内存泄漏来源：

只要有未处理的消息，那么消息会引用handler，非静态的handler又会引用外部类，即Activity，导致Activity无法被回收，造成泄漏；

Runnable类属于非静态匿名类，同样会引用外部类。

为了解决遇到的问题，我们要明确一点：静态内部类不会持有对外部类的引用。所以，我们可以把handler类放在单独的类文件中，或者使用静态内部类便可以避免泄漏。

另外，如果想要在handler内部去调用所在的外部类Activity，那么可以在handler内部使用弱引用的方式指向所在Activity，这样统一不会导致内存泄漏。

对于匿名类Runnable，同样可以将其设置为静态类。因为静态的匿名类不会持有对外部类的引用。

```
public class SampleActivity extends Activity {

  /**
   * Instances of static inner classes do not hold an implicit
   * reference to their outer class.
   */
  private static class MyHandler extends Handler {
    private final WeakReference<SampleActivity> mActivity;

    public MyHandler(SampleActivity activity) {
      mActivity = new WeakReference<SampleActivity>(activity);
    }

    @Override
    public void handleMessage(Message msg) {
      SampleActivity activity = mActivity.get();
      if (activity != null) {
        // ...
      }
    }
  }

  private final MyHandler mHandler = new MyHandler(this);

  /**
   * Instances of anonymous classes do not hold an implicit
   * reference to their outer class when they are "static".
   */
  private static final Runnable sRunnable = new Runnable() {
      @Override
      public void run() { /* ... */ }
  };

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    // Post a message and delay its execution for 10 minutes.
    mHandler.postDelayed(sRunnable, 1000 * 60 * 10);

    // Go back to the previous Activity.
    finish();
  }
}
```

五、小结

<<<<<<< HEAD

虽然静态类与非静态类之间的区别并不大，但是对于Android开发者而言却是必须理解的。至少我们要清楚，如果一个内部类实例的生命周期比Activity更长，那么我们千万不要使用非静态的内部类。最好的做法是，使用静态内部类，然后在该类里使用弱引用来指向所在的Activity。

原文链接：

http://www.jianshu.com/p/cb9b4b71a820

虽然静态类与非静态类之间的区别并不大，但是对于Android开发者而言却是必须理解的。至少我们要清楚，如果一个内部类实例的生命周期比Activity更长，那么我们千万不要使用非静态的内部类。最好的做法是，使用静态内部类，然后在该类里使用弱引用来指向所在的Activity。

原文链接：

http://www.jianshu.com/p/cb9b4b71a820

c67abfcfd66909095068cb5f0c8632dc5547131b

Android性能优化

Android性能优化

合理管理内存

节制的使用Service

如果应用程序需要使用Service来执行后台任务的话，只有当任务正在执行的时候才应该让Service运行起来。当启动一个Service时，系统会倾向于将这个Service所依赖的进程进行保留，系统可以在LRUcache当中缓存的进程数量也会减少，导致切换程序的时候耗费更多性能。我们可以使用IntentService，当后台任务执行结束后会自动停止，避免了Service的内存泄漏。

当界面不可见时释放内存

当用户打开了另外一个程序，我们的程序界面已经不可见的时候，我们应当将所有和界面相关的资源进行释放。重写Activity的onTrimMemory()方法，然后在这个方法中监听TRIM_MEMORY_UI_HIDDEN这个级别，一旦触发说明用户离开了程序，此时就可以进行资源释放操作了。

当内存紧张时释放内存

onTrimMemory()方法还有很多种其他类型的回调，可以在手机内存降低的时候及时通知我们，我们应该根据回调中传入的级别来去决定如何释放应用程序的资源。

避免在Bitmap上浪费内存

读取一个Bitmap图片的时候，千万不要去加载不需要的分辨率。可以压缩图片等操作。

是有优化过的数据集合

Android提供了一系列优化过后的数据集合工具类，如SparseArray、SparseBooleanArray、LongSparseArray，使用这些API可以让我们的程序更加高效。HashMap工具类会相对比较低效，因为它需要为每一个键值对都提供一个对象入口，而SparseArray就避免掉了基本数据类型转换成对象数据类型的时间。

知晓内存的开支情况

- 使用枚举通常会比使用静态常量消耗两倍以上的内存，尽可能不使用枚举
- 任何一个Java类，包括匿名类、内部类，都要占用大概500字节的内存空间
- 任何一个类的实例要消耗12-16字节的内存开支，因此频繁创建实例也是会在一定程序上影响内存的
- 使用HashMap时，即使你只设置了一个基本数据类型的键，比如说int，但是也会按照对象的大小来分配内存，大概是32字节，而不是4字节，因此最好使用优化后的数据集合

谨慎使用抽象编程

在Android使用抽象编程会带来额外的内存开支，因为抽象的编程方法需要编写额外的代码，虽然这些代码根本执行不到，但是也要映射到内存中，不仅占用了更多的内存，在执行效率上也会有所降低。所以需要合理的使用抽象编程。

尽量避免使用依赖注入框架

使用依赖注入框架貌似看上去把findViewById()这一类的繁琐操作去掉了，但是这些框架为了要搜寻代码中的注解，通常都需要经历较长的初始化过程，并且将一些你用不到的对象也一并加载到内存中。这些用不到的对象会一直站用着内存空间，可能很久之后才会得到释放，所以可能多敲几行代码是更好的选择。

使用多个进程

谨慎使用，多数应用程序不该在多个进程中运行的，一旦使用不当，它甚至会增加额外的内存而不是帮我们节省内存。这个技巧比较适用于哪些需要在后台去完成一项独立的任务，和前台是完全可以区分开的场景。比如音乐播放，关闭软件，已经完全由Service来控制音乐播放了，系统仍然会将许多UI方面的内存进行保留。在这种场景下就非常适合使用两个进程，一个用于UI展示，另一个用于在后台持续的播放音乐。关于实现多进程，只需要在Manifast文件的应用程序组件声明一个android:process属性就可以了。进程名可以自定义，但是之前要加个冒号，表示该进程是一个当前应用程序的私有进程。

分析内存的使用情况

系统不可能将所有的内存都分配给我们的应用程序，每个程序都会有可使用的内存上限，被称为堆大小。

不同的手机堆大小不同，如下代码可以获得堆大小：

```
ActivityManager manager = (ActivityManager)getSystemService(Context.ACTIVITY_SERVICE);
int heapSize = manager.getMemoryClass();
```

结果以MB为单位进行返回，我们开发时应用程序的内存不能超过这个限制，否则会出现OOM。

Android的GC操作

Android系统会在适当的时机触发GC操作，一旦进行GC操作，就会将一些不再使用的对象进行回收。GC操作会从一个叫做Roots的对象开始检查，所有它可以访问到的对象就说明还在使用当中，应该进行保留，而其他的对系那个就表示已经不再被使用了。

Android中内存泄漏

Android中的垃圾回收机制并不能防止内存泄漏的出现导致内存泄漏最主要的原因就是某些长存对象持有了一些其它应该被回收的对象的引用，导致垃圾回收器无法去回收掉这些对象，也就是出现内存泄漏了。比如说像Activity这样的系统组件，它又会包含很多的控件甚至是图片，如果它无法被垃圾回收器回收掉的话，那就算是比较严重的内存泄漏情况了。
举个例子，在MainActivity中定义一个内部类，实例化内部类对象，在内部类新建一个线程执行死循环，会导致内部类资源无法释放，MainActivity的控件和资源无法释放，导致OOM,可借助一系列工具，比如LeakCanary。

高性能编码优化

都是一些微优化，在性能方面看不出有什么显著的提升的。使用合适的算法和数据结构是优化程序性能的最主要手段。

避免创建不必要的对象

不必要的对象我们应该避免创建：

- 如果有需要拼接的字符串，那么可以优先考虑使用StringBuffer或者StringBuilder来进行拼接，而不是加号连接符，因为使用加号连接符会创建多余的对象，拼接的字符串越长，加号连接符的性能越低。
- 在没有特殊原因的情况下，尽量使用基本数据类型来代替封装数据类型，int比Integer要更加有效，其它数据类型也是一样。
- 当一个方法的返回值是String的时候，通常需要去判断一下这个String的作用是什么，如果明确知道调用方会将返回的String再进行拼接操作的话，可以考虑返回一个StringBuffer对象来代替，因为这样可以将一个对象的引用进行返回，而返回String的话就是创建了一个短生命周期的临时对象。
- 基本数据类型的数组也要优于对象数据类型的数组。另外两个平行的数组要比一个封装好的对象数组更加高效，举个例子，Foo[]和Bar[]这样的数组，使用起来要比Custom(Foo,Bar)[]这样的一个数组高

效的多。

尽可能地少创建临时对象，越少的对象意味着越少的GC操作。

静态优于抽象

如果你并不需要访问一个对系那个中的某些字段，只是想调用它的某些方法来去完成一项通用的功能，那么可以将这个方法设置成静态方法，调用速度提升15%-20%，同时也不用为了调用这个方法去专门创建对象了，也不用担心调用这个方法后是否会改变对象的状态(静态方法无法访问非静态字段)。

对常量使用static final修饰符

```
static int intVal = 42;
static String strVal = "Hello, world!";
```

编译器会为上面的代码生成一个初始方法，称为方法，该方法会在定义类第一次被使用的时候调用。这个方法会将42的值赋值到intVal当中，从字符串常量表中提取一个引用赋值到strVal上。当赋值完成后，我们就可以通过字段搜寻的方式去访问具体的值了。

final进行优化:

```
static final int intVal = 42;
static final String strVal = "Hello, world!";
```

这样，定义类就不需要方法了，因为所有的常量都会在dex文件的初始化器当中进行初始化。当我们调用intVal时可以直接指向42的值，而调用strVal会用一种相对轻量级的字符串常量方式，而不是字段搜寻的方式。

这种优化方式只对基本数据类型以及String类型的常量有效，对于其他数据类型的常量是无效的。

使用增强型for循环语法

```
static class Counter {
    int mCount;
}

Counter[] mArray = ...

public void zero() {
    int sum = 0;
    for (int i = 0; i < mArray.length; ++i) {
        sum += mArray[i].mCount;
    }
}

public void one() {
    int sum = 0;
    Counter[] localArray = mArray;
    int len = localArray.length;
    for (int i = 0; i < len; ++i) {
        sum += localArray[i].mCount;
    }
}

public void two() {
    int sum = 0;
    for (Counter a : mArray) {
        sum += a.mCount;
    }
}
```

zero()最慢，每次都要计算mArray的长度，one()相对快得多，two()fangfa在没有JIT(Just In Time Compiler)的设备上是运行最快的，而在有JIT的设备上运行效率和one()方法不相上下，需要注意这种写法需要JDK1.5之后才支持。

Tips:ArrayList手写的循环比增强型for循环更快，其他的集合没有这种情况。因此默认情况下使用增强型for循环，而遍历ArrayList使用传统的循环方式。

多使用系统封装好的API

系统提供不了的Api完成不了我们需要的功能才应该自己去写，因为使用系统的Api很多时候比我们自己写的代码要快得多，它们的很多功能都是通过底层的汇编模式执行的。
举个例子，实现数组拷贝的功能，使用循环的方式来对数组中的每一个元素一一进行赋值当然可行，但是直接使用系统中提供的System.arraycopy()方法会让执行效率快9倍以上。

避免在内部调用Getters/Setters方法

面向对象中封装的思想是不要把类内部的字段暴露给外部，而是提供特定的方法来允许外部操作相应类的内部字段。但在Android中，字段搜寻比方法调用效率高得多，我们直接访问某个字段可能要比通过getters方法来去访问这个字段快3到7倍。但是编写代码还是要按照面向对象思维的，我们应该在能优化的

地方进行优化，比如避免在内部调用getters/setters方法。

布局优化技巧

重用布局文件

标签可以允许在一个布局当中引入另一个布局，那么比如说我们程序的所有界面都有一个公共的部分，这个时候最好的做法就是将这个公共的部分提取到一个独立的布局中，然后每个界面的布局文件当中来引用这个公共的布局。

Tips:如果我们要在标签中覆写layout属性，必须要将layout_width和layout_height这两个属性也进行覆写，否则覆写xiaoguo将不会生效。

标签是作为标签的一种辅助扩展来使用的，它的主要作用是为了防止在引用布局文件时引用文件时产生多余的布局嵌套。布局嵌套越多，解析起来就越耗时，性能就越差。因此编写布局文件时应该让嵌套的层数越少越好。

举例：比如在LinearLayout里边使用一个布局。里边又有一个LinearLayout，那么其实就存在了多余的布局嵌套，使用merge可以解决这个问题。

仅在需要时才加载布局

某个布局当中的元素不是一起显示出来的，普通情况下只显示部分常用的元素，而那些不常用的元素只有在用户进行特定操作时才会显示出来。

举例：填信息时不是需要全部填的，有一个添加更多字段的选项，当用户需要添加其他信息的时候，才将另外的元素显示到界面上。用VISIBLE性能表现一般，可以用ViewStub。ViewStub也是View的一种，但是没有大小，没有绘制功能，也不参与布局，资源消耗非常低，可以认为完全不影响性能。

```
<ViewStub
      android:id="@+id/view_stub"
      android:layout="@layout/profile_extra"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      />
```

```
public void onMoreClick() {
   ViewStub viewStub = (ViewStub) findViewById(R.id.view_stub);
   if (viewStub != null) {
      View inflatedView = viewStub.inflate();
      editExtra1 = (EditText) inflatedView.findViewById(R.id.edit_extra1);
      editExtra2 = (EditText) inflatedView.findViewById(R.id.edit_extra2);
      editExtra3 = (EditText) inflatedView.findViewById(R.id.edit_extra3);
   }
}
```

tips：ViewStub所加载的布局是不可以使用标签的，因此这有可能导致加载出来出来的布局存在着多余的嵌套结构。

ListView详解

ListView详解

直接继承自AbsListView，AbsListView继承自AdapterView，AdapterView继承自AdapterView，AdapterView又继承自ViewGroup。

Adpater在ListView和数据源之间起到了一个桥梁的作用

RecycleBin机制

RecycleBin机制是ListView能够实现成百上千条数据都不会OOM最重要的一个原因。RecycleBin是AbsListView的一个内部类。

- RecycleBin当中使用mActiveViews这个数组来存储View，调用这个方法后就会根据传入的参数来将ListView中的指定元素存储到mActiveViews中。
- mActiveViews当中所存储的View，一旦被获取了之后就会从mActiveViews当中移除，下次获取同样位置的时候将会返回null，所以mActiveViews不能被重复利用。
- addScrapView()用于将一个废弃的View进行缓存，该方法接收一个View参数，当有某个View确定要废弃掉的时候（比如滚动出了屏幕）就应该调用这个方法来对View进行缓存，RecycleBin当中使用mScrapV
- iews和mCurrentScrap这两个List来存储废弃View。
- getScrapView 用于从废弃缓存中取出一个View，这些废弃缓存中的View是没有顺序可言的，因此getScrapView()方法中的算法也非常简单，就是直接从mCurrentScrap当中获取尾部的一个scrap view进行返回。
- 我们都知道Adapter当中可以重写一个getViewTypeCount()来表示ListView中有几种类型的数据项，而setViewTypeCount()方法的作用就是为每种类型的数据项都单独启用一个RecycleBin缓存机制。

View的流程分三步，onMeasure()用于测量View的大小，onLayout()用于确定View的布局，onDraw()用于将View绘制到界面上。

RecyclerView和ListView的异同

RecyclerView和ListView的异同

- ViewHolder是用来保存视图引用的类，无论是ListView亦或是RecyclerView。只不过在ListView中，ViewHolder需要自己来定义，且这只是一种推荐的使用方式，不使用当然也可以，这不是必须的。只不过不使用ViewHolder的话，ListView每次getView的时候都会调用findViewById(int)，这将导致ListView性能展示迟缓。而在RecyclerView中使用RecyclerView.ViewHolder则变成了必须，尽管实现起来稍显复杂，但它却解决了ListView面临的上述不使用自定义ViewHolder时所面临的问题。

- 我们知道ListView只能在垂直方向上滚动，Android API没有提供ListView在水平方向上面滚动的支持。或许有多种方式实现水平滑动，但是请想念我，ListView并不是设计来做这件事情的。但是RecyclerView相较于ListView，在滚动上面的功能扩展了许多。它可以支持多种类型列表的展示要求，主要如下：

  1. LinearLayoutManager，可以支持水平和竖直方向上滚动的列表。
  2. StaggeredGridLayoutManager，可以支持交叉网格风格的列表，类似于瀑布流或者Pinterest。
  3. GridLayoutManager，支持网格展示，可以水平或者竖直滚动，如展示图片的画廊。

- 列表动画是一个全新的、拥有无限可能的维度。起初的Android API中，删除或添加item时，item是无法产生动画效果的。后面随着Android的进化，Google的Chat Hasse推荐使用ViewPropertyAnimator属性动画来实现上述需求。
  相比较于ListView，RecyclerView.ItemAnimator则被提供用于在RecyclerView添加、删除或移动item时处理动画效果。同时，如果你比较懒，不想自定义ItemAnimator，你还可以使用DefaultItemAnimator。

- ListView的Adapter中，getView是最重要的方法，它将视图跟position绑定起来，是所有神奇的事情发生的地方。同时我们也能够通过registerDataObserver在Adapter中注册一个观察者。RecyclerView也有这个特性，RecyclerView.AdapterDataObserver就是这个观察者。ListView有三个Adapter的默认实现，分别是ArrayAdapter、CursorAdapter和SimpleCursorAdapter。然而，RecyclerView的Adapter则拥有除了内置的内DB游标和ArrayList的支持之外的所有功能。RecyclerView.Adapter的实现的，我们必须采取措施将数据提供给Adapter，正如BaseAdapter对ListView所做的那样。

- 在ListView中如果我们想要在item之间添加间隔符，我们只需要在布局文件中对ListView添加如下属性即可：

```
android:divider="@android:color/transparent"
android:dividerHeight="5dp"
```

- ListView通过AdapterView.OnItemClickListener接口来探测点击事件。而RecyclerView则通过RecyclerView.OnItemTouchListener接口来探测触摸事件。它虽然增加了实现的难度，但是却给予开发人员拦截触摸事件更多的控制权限。

- ListView可以设置选择模式，并添加MultiChoiceModeListener，如下所示：

```
listView.setChoiceMode(ListView.CHOICE_MODE_MULTIPLE_MODAL);
listView.setMultiChoiceModeListener(new MultiChoiceModeListener() {
public boolean onCreateActionMode(ActionMode mode, Menu menu) { ... }
public void onItemCheckedStateChanged(ActionMode mode, int position,
long id, boolean checked) { ... }
public boolean onActionItemClicked(ActionMode mode, MenuItem item) {
    switch (item.getItemId()) {
        case R.id.menu_item_delete_crime:
        CrimeAdapter adapter = (CrimeAdapter)getListAdapter();
        CrimeLab crimeLab = CrimeLab.get(getActivity());
        for (int i = adapter.getCount() - 1; i >= 0; i--) {
            if (getListView().isItemChecked(i)) {
                crimeLab.deleteCrime(adapter.getItem(i));
            }
      }
    mode.finish();
    adapter.notifyDataSetChanged();
    return true;
    default:
        return false;
}
public boolean onPrepareActionMode(ActionMode mode, Menu menu) { ... }
public void onDestroyActionMode(ActionMode mode) { ... }
});
```

而RecyclerView则没有此功能。

http://www.cnblogs.com/littlepanpc/p/4497290.html

AsyncTask源码分析

AsyncTask

首先从Android3.0开始，系统要求网络访问必须在子线程中进行，否则网络访问将会失败并抛出NetworkOnMainThreadException这个异常，这样做是为了避免主线程由于耗时操作所阻塞从而出现ANR现象。AsyncTask封装了线程池和Handler。AsyncTask有两个线程池：SerialExecutor和THREAD_POOL_EXECUTOR。前者是用于任务的排队，默认是串行的线程池：后者用于真正的执行任务。AsyncTask还有一个Handler，叫InternalHandler，用于将执行环境从线程池切换到主线程。AsyncTask内部就是通过InternalHandler来发送任务执行的进度以及执行结束等消息。

AsyncTask排队执行过程：系统先把参数Params封装为FutureTask对象，它相当于Runnable，接着FutureTask交给SerialExcutor的execute方法，它先把FutureTask插入到任务队列tasks中，如果这个时候没有正在活动的AsyncTask任务，那么就会执行下一个AsyncTask任务，同时当一个AsyncTask任务执行完毕之后，AsyncTask会继续执行其他任务直到所有任务都被执行为止。

关于线程池，AsyncTask对应的线程池ThreadPoolExecutor都是进程范围内共享的，都是static的，所以是AsyncTask控制着进程范围内所有的子类实例。由于这个限制的存在，当使用默认线程池时，如果线程数超过线程池的最大容量，线程池就会爆掉(3.0默认串行执行，不会出现这个问题)。针对这种情况。可以尝试自定义线程池，配合AsyncTask使用。

插件化技术

Android动态加载dex技术初探

http://blog.csdn.net/u013478336/article/details/50734108

Android使用Dalvik虚拟机加载可执行程序，所以不能直接加载基于class的jar，而是需要将class转化为dex字节码。

Android支持动态加载的两种方式是：DexClassLoader和PathClassLoader，DexClassLoader可加载jar/apk/dex，且支持从SD卡加载；PathClassLoader据说只能加载已经安装在Android系统内APK文件。

Android插件化基础

Android简单来说就是如下操作：

- 开发者将插件代码封装成Jar或者APK
- 宿主下载或者从本地加载Jar或者APK到宿主中
- 将宿主调用插件中的算法或者Android特定的Class（如Activity）

插件化开发—动态加载技术加载已安装和未安装的apk

http://blog.csdn.net/u010687392/article/details/47121729?hmsr=toutiao.io&utm_medium=toutiao.io&utm_source=toutiao.io 为什么引入动态加载技术？

- 一个应用程序dex文件的方法数最大不能超过65536个
- 可以让应用程序实现插件化、插拔式结构，对后期维护有益

什么是动态加载技术

动态加载技术就是使用类加载器加载相应的apk、dex、jar(必须含有dex文件)，再通过反射获得该apk、dex、jar内部的资源（class、图片、color等等）进而供宿主app使用。

关于动态加载使用的类加载器

- PathClassLoader - 只能加载已经安装的apk，即/data/app目录下的apk。
- DexClassLoader - 能加载手机中未安装的apk、jar、dex，只要能在找到对应的路径。

插件化技术学习

原因：

各大厂商都碰到了AndroidNative平台的瓶颈：

1. 从技术上讲，业务逻辑的复杂代码急剧膨胀，各大厂商陆续触到65535方法数的天花板；同时，对模块热更新提出了更高的要求。
2. 在业务层面上，功能模块的解耦以及维护团队的分离也是大势所趋。

插件化技术主要解决两个问题：

1. 代码加载
2. 资源加载

代码加载

类的加载可以使用Java的ClassLoader机制，还需要组件生命周期管理。

资源加载

用AssetManager的隐藏方法addAssetPath。

Android插件化原理解析——Hook机制之动态代理

使用代理机制进行API Hook进而达到方法增强。

静态代理

动态代理：可以简单理解为JVM可以在运行时帮我们动态生成一系列的代理类。

代理Hook

如果我们自己创建代理对象，然后把原始对象替换为我们的代理对象，就可以在这个代理对象中为所欲为了；修改参数，替换返回值，称之为Hook。

整个Hook过程简要总结如下：

1. 寻找Hook点，原则是静态变量或者单例对象，尽量Hook public的对象和方法，非public不保证每个版本都一样，需要适配。
2. 选择合适的代理方式，如果是接口可以用动态代理；如果是类可以手动写代理也可以使用cglib。
3. 偷梁换柱 - 用代理对象替换原始对象

Android插件化原理解析——Hook机制之Binder Hook

自定义控件

自定义控件

自定义View的步骤：

- 自定义View的属性
- 在View的构造方法中获得我们自定义View的步骤
- ［3.重写onMeasure］(不必须)
- 重写onDraw

ANR问题

ANR

1、ANR排错一般有三种类型

1. KeyDispatchTimeout(5 seconds) --主要是类型按键或触摸事件在特定时间内无响应
2. BroadcastTimeout(10 seconds) --BroadcastReceiver在特定时间内无法处理完成
3. ServiceTimeout(20 secends) --小概率事件 Service在特定的时间内无法处理完成

2、哪些操作会导致ANR
在主线程执行以下操作：

1. 高耗时的操作，如图像变换
2. 磁盘读写，数据库读写操作
3. 大量的创建新对象

3、如何避免

1. UI线程尽量只做跟UI相关的工作
2. 耗时的操作(比如数据库操作，I/O，连接网络或者别的有可能阻塞UI线程的操作)把它放在单独的线程处理
3. 尽量用Handler来处理UIThread和别的Thread之间的交互

4、解决的逻辑

1. 使用AsyncTask
    1. 在doInBackground()方法中执行耗时操作
    2. 在onPostExecuted()更新UI

2. 使用Handler实现异步任务

    1. 在子线程中处理耗时操作
    2. 处理完成之后，通过handler.sendMessage()传递处理结果
    3. 在handler的handleMessage()方法中更新UI
    4. 或者使用handler.post()方法将消息放到Looper中

5、如何排查

1. 首先分析log
2. 从trace.txt文件查看调用stack，adb pull data/anr/traces.txt ./mytraces.txt
3. 看代码
4. 仔细查看ANR的成因(iowait?block?memoryleak?)

6、监测ANR的Watchdog

最近出来一个叫LeakCanary

FC(Force Close)

什么时候会出现

1. Error
2. OOM，内存溢出
3. StackOverFlowError
4. Runtime,比如说空指针异常

解决的办法

1. 注意内存的使用和管理
2. 使用Thread.UncaughtExceptionHandler接口

Art和Dalvik的区别

ART和Dalvik区别

Art上应用启动快，运行快，但是耗费更多存储空间，安装时间长，总的来说ART的功效就是"空间换时间"。

ART: Ahead of Time Dalvik: Just in Time

什么是Dalvik：Dalvik是Google公司自己设计用于Android平台的Java虚拟机。Dalvik虚拟机是Google等厂商合作开发的Android移动设备平台的核心组成部分之一，它可以支持已转换为.dex(即Dalvik Executable)格式的Java应用程序的运行，.dex格式是专为Dalvik应用设计的一种压缩格式，适合内存和处理器速度有限的系统。Dalvik经过优化，允许在有限的内存中同时运行多个虚拟机的实例，并且每一个Dalvik应用作为独立的Linux进程执行。独立的进程可以防止在虚拟机崩溃的时候所有程序都被关闭。

什么是ART:Android操作系统已经成熟，Google的Android团队开始将注意力转向一些底层组件，其中之一是负责应用程序运行的Dalvik运行时。Google开发者已经花了两年时间开发更快执行效率更高更省电的替代ART运行时。ART代表Android Runtime,其处理应用程序执行的方式完全不同于Dalvik，Dalvik是依靠一个Just-In-Time(JIT)编译器去解释字节码。开发者编译后的应用代码需要通过一个解释器在用户的设备上运行，这一机制并不高效，但让应用能更容易在不同硬件和架构上运行。ART则完全改变了这套做法，在应用安装的时候就预编译字节码到机器语言，这一机制叫Ahead-Of-Time(AOT)编译。在移除解释代码这一过程后，应用程序执行将更有效率，启动更快。

ART优点：

1. 系统性能的显著提升
2. 应用启动更快、运行更快、体验更流畅、触感反馈更及时
3. 更长的电池续航能力
4. 支持更低的硬件

ART缺点：

1. 更大的存储空间占用，可能会增加10%-20%
2. 更长的应用安装时间

Android关于OOM的解决方案

Android关于OOM的解决方案

OOM

- 内存溢出（Out Of Memory）
- 也就是说内存占有量超过了VM所分配的最大

出现OOM的原因

1. 加载对象过大
2. 相应资源过多，来不及释放

如何解决

1. 在内存引用上做些处理，常用的有软引用、强化引用、弱引用
2. 在内存中加载图片时直接在内存中作处理，如边界压缩
3. 动态回收内存
4. 优化Dalvik虚拟机的堆内存分配
5. 自定义堆内存大小

Fragment

Fragment

为何产生

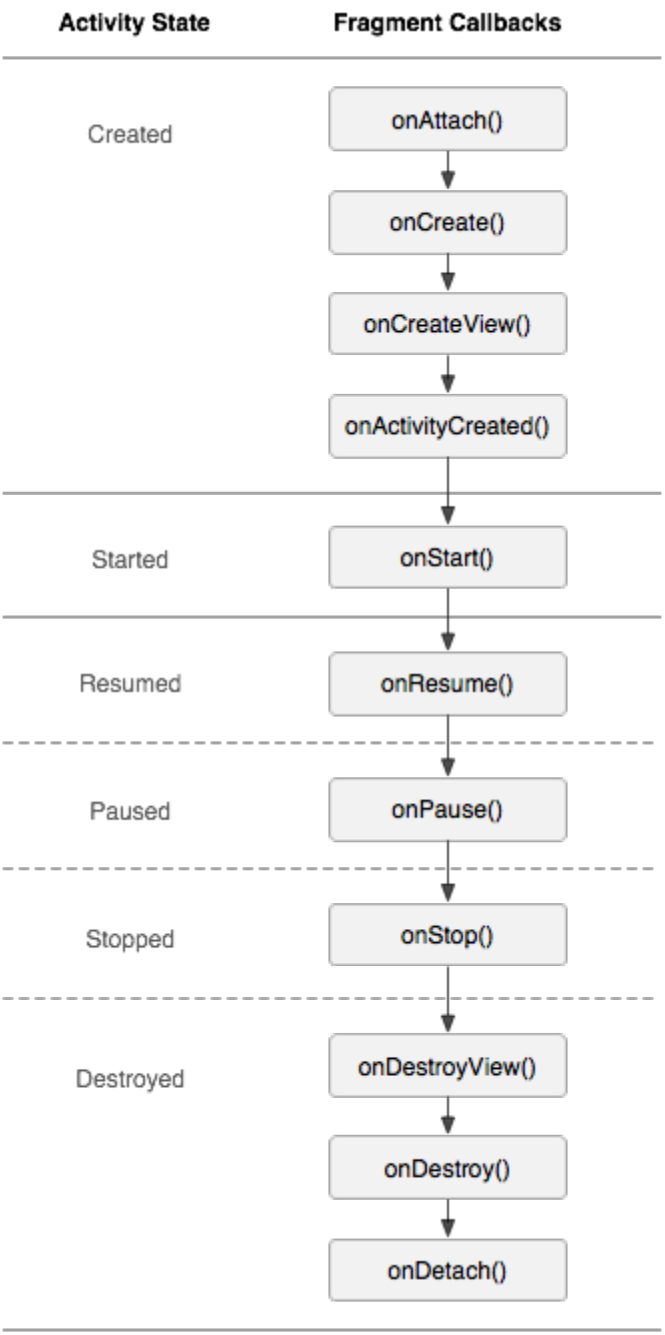
- 同时适配手机和平板、UI和逻辑的共享。

介绍

- Fragment也会被加入回退栈中。
- Fragment拥有自己的生命周期和接受、处理用户的事件
- 可以动态的添加、替换和移除某个Fragment

生命周期

- 必须依存于Activity



- Fragment依附于Activity的生命状态
-

生命周期中那么多方法，懵逼了的话我们就一起来看一下每一个生命周期方法的含义吧。

Fragment生命周期方法含义：

- `public void onAttach(Context context)`
  - onAttach方法会在Fragment于窗口关联后立刻调用。从该方法开始，就可以通过Fragment.getActivity方法获取与Fragment关联的窗口对象，但因为Fragment的控件未初始化，所以不能够操作控件。
- `public void onCreate(Bundle savedInstanceState)`
  - 在调用完onAttach执行完之后立刻调用onCreate方法，可以在Bundle对象中获取一些在Activity

中传过来的数据。通常会在该方法中读取保存的状态，获取或初始化一些数据。在该方法中不要进行耗时操作，不然窗口不会显示。

- `public View onCreateView(LayoutInflater inflater, ViewGroup container,Bundle savedInstanceS`
  - 该方法是Fragment很重要的一个生命周期方法，因为会在该方法中创建在Fragment显示的View，其中inflater是用来装载布局文件的，container是`<fragment>`标签的父标签对应对象，savedInstanceState参数可以获取Fragment保存的状态，如果未保存那么就为null。
- `public void onViewCreated(View view,Bundle savedInstanceState)`
  - Android在创建完Fragment中的View对象之后，会立刻回调该方法。其种view参数就是onCreateView中返回的view，而bundle对象用于一般用途。
- `public void onActivityCreated(Bundle savedInstanceState)`
  - 在Activity的onCreate方法执行完之后，Android系统会立刻调用该方法，表示窗口已经初始化完成，从这一个时候开始，就可以在Fragment中使用getActivity().findViewById(Id);来操控Activity中的view了。
- `public void onStart()`
  - 这个没啥可讲的，但有一个细节需要知道，当系统调用该方法的时候，fragment已经显示在ui上，但还不能进行互动，因为onResume方法还没执行完。
- `public void onResume()`
  - 该方法为fragment从创建到显示Android系统调用的最后一个生命周期方法，调用完该方法时候，fragment就可以与用户互动了。
- `public void onPause()`
  - fragment由活跃状态变成非活跃状态执行的第一个回调方法，通常可以在这个方法中保存一些需要临时暂停的工作。如保存音乐播放进度，然后在onResume中恢复音乐播放进度。
- `public void onStop()`
  - 当onStop返回的时候，fragment将从屏幕上消失。
- `public void onDestoryView()`
  - 该方法的调用意味着在`onCreateView`中创建的视图都将被移除。
- `public void onDestroy()`
  - Android在Fragment不再使用时会调用该方法，要注意的是~这时Fragment还和Activity藕断丝连！并且可以获得Fragment对象，但无法对获得的Fragment进行任何操作（呵~呵呵~我已经不

听你的了）。

- `public void onDetach()`
  - 为Fragment生命周期中的最后一个方法，当该方法执行完后，Fragment与Activity不再有关联(分手！我们分手！！(ﾍ'□')ﾍ ┻┻)。

Fragment比Activity多了几个额外的生命周期回调方法：

- onAttach(Activity):当Fragment和Activity发生关联时使用
- onCreateView(LayoutInflater, ViewGroup, Bundle):创建该Fragment的视图
- onActivityCreate(Bundle):当Activity的onCreate方法返回时调用
- onDestoryView():与onCreateView相对应，当该Fragment的视图被移除时调用
- onDetach():与onAttach相对应，当Fragment与Activity关联被取消时调用

注意：除了onCreateView，其他的所有方法如果你重写了，必须调用父类对于该方法的实现

Fragment与Activity之间的交互

- Fragment与Activity之间的交互可以通过 `Fragment.setArguments(Bundle args)` 以及 `Fragment.getArguments()` 来实现。

Fragment状态的持久化。

由于Activity会经常性的发生配置变化，所以依附它的Fragment就有需要将其状态保存起来问题。下面有两个常用的方法去将Fragment的状态持久化。

- 方法一：
  - 可以通过 `protected void onSaveInstanceState(Bundle outState)`，`protected void onRestoreInstanceState(Bundle savedInstanceState)` 状态保存和恢复的方法将状态持久化。
- 方法二(更方便,让Android自动帮我们保存Fragment状态)：
  - 我们只需要将Fragment在Activity中作为一个变量整个保存，只要保存了Fragment，那么Fragment的状态就得到保存了，所以呢.....
    - `FragmentManager.putFragment(Bundle bundle, String key, Fragment fragment)` 是在Activity中保存Fragment的方法。
    - `FragmentManager.getFragment(Bundle bundle, String key)` 是在Activity中获取所保存的Frament的方法。

- 很显然，key就传入Fragment的id，fragment就是你要保存状态的fragment，但，我们注意到上面的两个方法，第一个参数都是Bundle，这就意味着*FragmentManager*是通过Bundle去保存Fragment的。但是，这个方法仅仅能够保存Fragment中的控件状态，比如说EditText中用户已经输入的文字（*注意！在这里，控件需要设置一个id，否则Android将不会为我们保存控件的状态*），而Fragment中需要持久化的变量依然会丢失，但依然有解决办法，就是利用方法一！

- 下面给出状态持久化的事例代码：

```
    /** Activity中的代码 **/
    FragmentB fragmentB;

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.fragment_activity);
        if( savedInstanceState != null ){
            fragmentB = (FragmentB) getSupportFragmentManager().getFragment(savedInstance
State,"fragmentB");
        }
        init();
    }

    @Override
    protected void onSaveInstanceState(Bundle outState) {
        if( fragmentB != null ){
            getSupportFragmentManager().putFragment(outState,"fragmentB",fragmentB);
        }

        super.onSaveInstanceState(outState);
    }

    /** Fragment中保存变量的代码 **/

    @Nullable
    @Override
    public View onCreateView(LayoutInflater inflater, @Nullable ViewGroup container, @Nullab
le Bundle savedInstanceState) {
        AppLog.e("onCreateView");
        if ( null != savedInstanceState ){
            String savedString = savedInstanceState.getString("string");
            //得到保存下来的string
        }
        View root = inflater.inflate(R.layout.fragment_a,null);
        return root;
    }

    @Override
    public void onSaveInstanceState(Bundle outState) {
        outState.putString("string","anAngryAnt");
        super.onSaveInstanceState(outState);
    }
```

静态的使用Fragment

1. 继承Fragment，重写onCreateView决定Fragment的布局
2. 在Activity中声明此Fragment,就和普通的View一样

Fragment常用的API

- android.support.v4.app.Fragment 主要用于定义Fragment
- android.support.v4.app.FragmentManager 主要用于在Activity中操作Fragment，可以使用FragmentManager.findFragmenById，FragmentManager.findFragmentByTag等方法去找到一个Fragment
- android.support.v4.app.FragmentTransaction 保证一些列Fragment操作的原子性，熟悉事务这个词
- 主要的操作都是FragmentTransaction的方法
  (一般我们为了向下兼容，都使用support.v4包里面的Fragment)

```
getFragmentManager() // Fragment若使用的是support.v4包中的，那就使用getSupportFragm
```

- 主要的操作都是FragmentTransaction的方法

```
FragmentTransaction transaction = fm.benginTransatcion();//开启一个事务
transaction.add()
//往Activity中添加一个Fragment

transaction.remove()
//从Activity中移除一个Fragment，如果被移除的Fragment没有添加到回退栈（回退栈后面会详细说），这个Fragment实例将会被销毁。

transaction.replace()
//使用另一个Fragment替换当前的，实际上就是remove()然后add()的合体~

transaction.hide()
//隐藏当前的Fragment，仅仅是设为不可见，并不会销毁

transaction.show()
//显示之前隐藏的Fragment

detach()
//当fragment被加入到回退栈的时候，该方法与*remove()*的作用是相同的，
//反之，该方法只是将fragment从视图中移除，
//之后仍然可以通过*attach()*方法重新使用fragment，
//而调用了*remove()*方法之后，
//不仅将Fragment从视图中移除，fragment还将不再可用。

attach()
//重建view视图，附加到UI上并显示。

transatcion.commit()
//提交一个事务
```

管理Fragment回退栈

- 跟踪回退栈状态
  - 我们通过实现 *OnBackStackChangedListener* 接口来实现回退栈状态跟踪，具体如下

    ```
    public class XXX implements FragmentManager.OnBackStackChangedListener

    /* 实现接口所要实现的方法 */

@Override
public void onBackStackChanged() {
//do whatevery you want
}

/* 设置回退栈监听接口 */
getSupportFragmentManager().addOnBackStackChangedListener(this);

```

- 管理回退栈
    - `FragmentTransaction.addToBackStack(String)` *--将一个刚刚添加的Fragment加入到回退栈中*
    - `getSupportFragmentManager().getBackStackEntryCount()` *- 获取回退栈中实体数量*
    - `getSupportFragmentManager().popBackStack(String name, int flags)` *- 根据name立刻弹出栈顶的fragment*
    - `getSupportFragmentManager().popBackStack(int id, int flags)` *- 根据id立刻弹出栈顶的fragment*

SurfaceView

为什么要使用SurfaceView来实现动画？

因为View的绘图存在以下缺陷：

1. View缺乏双缓冲机制
2. 当程序需要更新View上的图像时，程序必须重绘View上显示的整张图片
3. 新线程无法直接更新View组件

SurfaceView的绘图机制

- 一般会与SurfaceView结合使用
- 调用SurfaceView的getHolder()方法即可获得SurfaceView关联的SurfaceHolder

SurfaceHolder提供了如下方法来获取Canvas对象

1. Canvas lockCanvas():锁定整个SurfaceView对象，获取该Surface上的Canvas
2. Canvas lockCanvas(Rect dirty):锁定SurfaceView上Rect划分的区域，获取该Surface上的Canvas
3. unlockCanvasAndPost(canvas):释放绘图、提交所绘制的图形，需要注意，当调用SurfaceHolder上的unlockCanvasAndPost方法之后，该方法之前所绘制的图形还处于缓冲之中，下一次lockCanvas()方法锁定的区域可能会“遮挡”它

```

```

```
public class SurfaceViewTest extends Activity
{
// SurfaceHolder负责维护SurfaceView上绘制的内容
private SurfaceHolder holder;
private Paint paint;
```

```
@Override
public void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    paint = new Paint();
    SurfaceView surface = (SurfaceView) findViewById(R.id.show);
    // 初始化SurfaceHolder对象
    holder = surface.getHolder();
    holder.addCallback(new Callback()
    {
        @Override
        public void surfaceChanged(SurfaceHolder arg0, int arg1, int arg2,
                int arg3)
        {
        }

        @Override
        public void surfaceCreated(SurfaceHolder holder)
        {
            // 锁定整个SurfaceView
            Canvas canvas = holder.lockCanvas();
            // 绘制背景
            Bitmap back = BitmapFactory.decodeResource(
                SurfaceViewTest.this.getResources()
                , R.drawable.sun);
            // 绘制背景
            canvas.drawBitmap(back, 0, 0, null);
            // 绘制完成，释放画布，提交修改
            holder.unlockCanvasAndPost(canvas);
            // 重新锁一次，"持久化"上次所绘制的内容
            holder.lockCanvas(new Rect(0, 0, 0, 0));
            holder.unlockCanvasAndPost(canvas);
        }

        @Override
        public void surfaceDestroyed(SurfaceHolder holder)
        {
        }
    });
    // 为surface的触摸事件绑定监听器
    surface.setOnTouchListener(new OnTouchListener()
    {
        @Override
        public boolean onTouch(View source, MotionEvent event)
        {
            // 只处理按下事件
            if (event.getAction() == MotionEvent.ACTION_DOWN)
            {
                int cx = (int) event.getX();
                int cy = (int) event.getY();
                // 锁定SurfaceView的局部区域，只更新局部内容
                Canvas canvas = holder.lockCanvas(new Rect(cx - 50,
                        cy - 50, cx + 50, cy + 50));
```

```
                // 保存canvas的当前状态
                canvas.save();
                // 旋转画布
                canvas.rotate(30, cx, cy);
                paint.setColor(Color.RED);
                // 绘制红色方块
                canvas.drawRect(cx - 40, cy - 40, cx, cy, paint);
                // 恢复Canvas之前的保存状态
                canvas.restore();
                paint.setColor(Color.GREEN);
                // 绘制绿色方块
                canvas.drawRect(cx, cy, cx + 40, cy + 40, paint);
                // 绘制完成，释放画布，提交修改
                holder.unlockCanvasAndPost(canvas);
            }
            return false;
        }
    });
}
}
```

上面的程序为SurfaceHolder添加了一个CallBack实例，该Callback中定义了如下三个方法：

- void surfaceChanged(SurfaceHolder holder, int format, int width, int height):当一个surface的格式或大小发生改变时回调该方法。
- void surfaceCreated(SurfaceHolder holder):当surface被创建时回调该方法
- void surfaceDestroyed(SurfaceHolder holder):当surface将要被销毁时回调该方法

Android几种进程

Android几种进程

1. 前台进程：即与用户正在交互的Activity或者Activity用到的Service等，如果系统内存不足时前台进程是最后被杀死的
2. 可见进程：可以是处于暂停状态(onPause)的Activity或者绑定在其上的Service，即被用户可见，但由于失去了焦点而不能与用户交互
3. 服务进程：其中运行着使用startService方法启动的Service，虽然不被用户可见，但是却是用户关心的，例如用户正在非音乐界面听的音乐或者正在非下载页面自己下载的文件等；当系统要空间运行前两者进程时才会被终止
4. 后台进程：其中运行着执行onStop方法而停止的程序，但是却不是用户当前关心的，例如后台挂着的QQ，这样的进程系统一旦没了有内存就首先被杀死

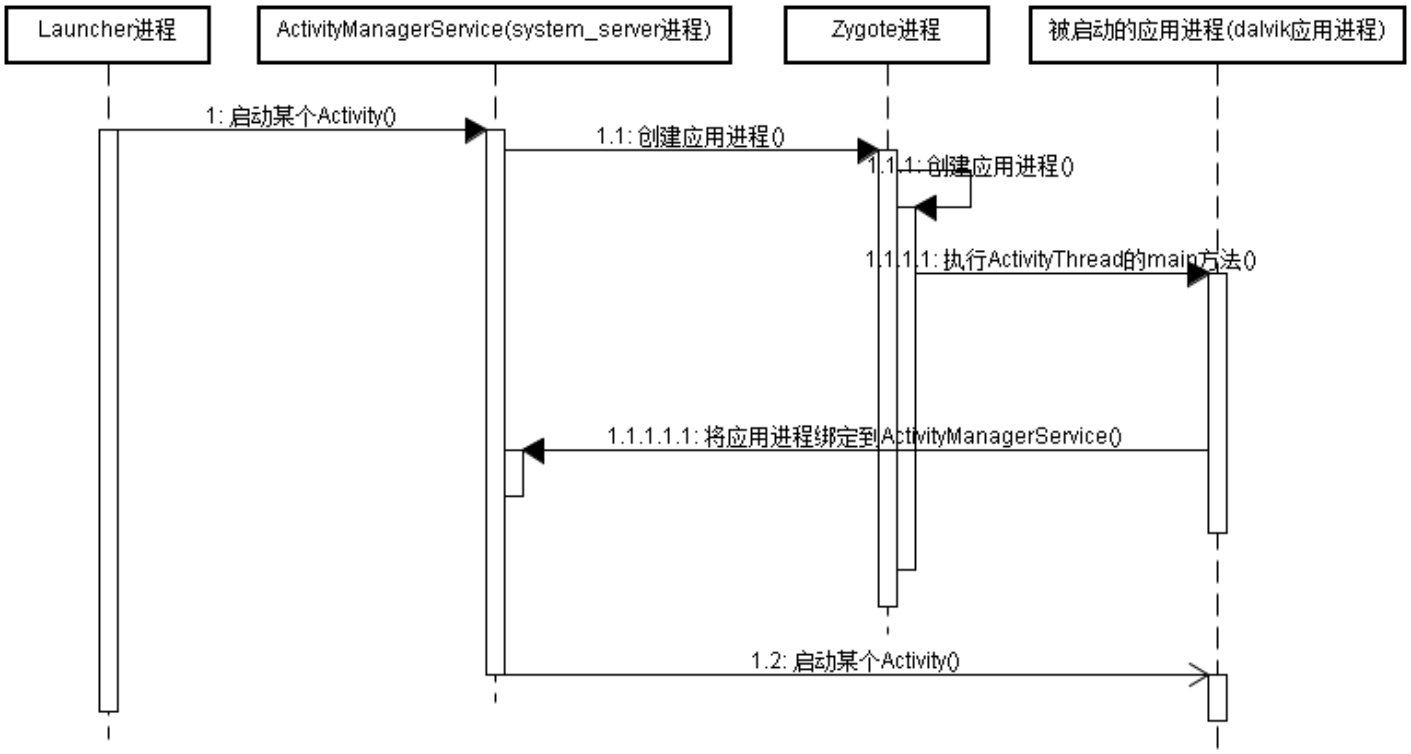
5. 空进程：不包含任何应用程序的程序组件的进程，这样的进程系统是一般不会让他存在的

如何避免后台进程被杀死：

1. 调用startForegound，让你的Service所在的线程成为前台进程
2. Service的onStartCommond返回START_STICKY或START_REDELIVER_INTENT
3. Service的onDestroy里面重新启动自己

APP启动过程

APP启动过程



- 上图就可以很好的说明App启动的过程
- ActivityManagerService组织回退栈时以ActivityRecord为基本单位，所有的ActivityRecord放在同一个ArrayList里，可以将mHistory看作一个栈对象，索引0所指的对象位于栈底，索引mHistory.size()-1所指的对象位于栈顶
- Zygote进程孵化出新的应用进程后，会执行ActivityThread类的main方法.在该方法里会先准备好Looper和消息队列，然后调用attach方法将应用进程绑定到ActivityManagerService，然后进入loop循环，不断地读取消息队列里的消息，并分发消息。
- ActivityThread的main方法执行后,应用进程接下来通知ActivityManagerService应用进程已启动，ActivityManagerService保存应用进程的一个代理对象，这样ActivityManagerService可以通过这个代理对象控制应用进程，然后ActivityManagerService通知应用进程创建入口Activity的实例，并执行它的生命周期方法

图片三级缓存

Android图片中的三级缓存

为什么要使用三级缓存

- 如今的 Android App 经常会需要网络交互，通过网络获取图片是再正常不过的事了
- 假如每次启动的时候都从网络拉取图片的话，势必会消耗很多流量。在当前的状况下，对于非wifi用户来说，流量还是很贵的，一个很耗流量的应用，其用户数量级肯定要受到影响
- 特别是，当我们想要重复浏览一些图片时，如果每一次浏览都需要通过网络获取，流量的浪费可想而知
- 所以提出三级缓存策略，通过网络、本地、内存三级缓存图片，来减少不必要的网络交互，避免浪费流量

什么是三级缓存

- 网络加载，不优先加载，速度慢，浪费流量
- 本地缓存，次优先加载，速度快
- 内存缓存，优先加载，速度最快

三级缓存原理

- 首次加载 Android App 时，肯定要通过网络交互来获取图片，之后我们可以将图片保存至本地SD卡和内存中
- 之后运行 App 时，优先访问内存中的图片缓存，若内存中没有，则加载本地SD卡中的图片
- 总之，只在初次访问新内容时，才通过网络获取图片资源

参考链接

http://www.jianshu.com/p/2cd59a79ed4a

Bitmap的分析与使用

Bitmap的分析与使用

- Bitmap的创建

- 创建Bitmap的时候，Java不提供 new Bitmap() 的形式去创建，而是通过 BitmapFactory 中的静态方法去创建,如: BitmapFactory.decodeStream(is);//通过InputStream去解析生成Bitmap (这里就不贴 BitmapFactory 中创建 Bitmap 的方法了，大家可以自己去看它的源码)，我们跟进 BitmapFactory 中创建 Bitmap 的源码，最终都可以追溯到这几个native函数

```
private static native Bitmap nativeDecodeStream(InputStream is, byte[] storage,
        Rect padding, Options opts);
private static native Bitmap nativeDecodeFileDescriptor(FileDescriptor fd,
      Rect padding, Options opts);
private static native Bitmap nativeDecodeAsset(long nativeAsset, Rect padding, Options o
private static native Bitmap nativeDecodeByteArray(byte[] data, int offset,
      int length, Options opts);
```

  而 Bitmap 又是Java对象，这个Java对象又是从native，也就是C/C++中产生的，所以，在Android中Bitmap的内存管理涉及到两部分，一部分是*native*，另一部分是*dalvik*，也就是我们常说的java堆(如果对java堆与栈不了解的同学可以戳)，到这里基本就已经了解了创建Bitmap的一些内存中的特性(大家可以使用 adb shell dumpsys meminfo 去查看Bitmap实例化之后的内存使用情况)。

- Bitmap的使用
  - 我们已经知道了 BitmapFactory 是如何通过各种资源创建 Bitmap 了，那么我们如何合理的使用它呢？以下是几个我们使用 Bitmap 需要关注的点

    1. Size
       - 这里我们来算一下，在Android中，如果采用 Config.ARGB_8888 的参数去创建一个 Bitmap ，这是Google推荐的配置色彩参数，也是Android4.4及以上版本默认创建Bitmap的Config参数(Bitmap.Config.inPreferredConfig 的默认值)，那么每一个像素将会占用4byte，如果一张手机照片的尺寸为1280×720，那么我们可以很容易的计算出这张图片占用的内存大小为 1280x720x4 = 3686400(byte) = 3.5M，一张未经处理的照片就已经3.5M了! 显而易见，在开发当中，这是我们最需要关注的问题，否则分分钟OOM!
       - *那么，我们一般是如何处理Size这个重要的因素的呢？*，当然是调整 Bitmap 的大小到适合的程度啦！辛亏在 BitmapFactory 中，我们可以很方便的通过 BitmapFactory.Options 中的 options.inSampleSize 去设置 Bitmap 的压缩比，官方给出的说法是

         > If set to a value > 1, requests the decoder to subsample the original image, returning a smaller image to save memory....For example, inSampleSize == 4

> returns
> an image that is 1/4 the width/height of the original, and 1/16 the number of pixels. Any value <= 1 is treated the same as 1.

很简洁明了啊！也就是说，只要按计算方法设置了这个参数，就可以完成我们Bitmap的Size调整了。那么，应该怎么调整姿势才比较舒服呢？下面先介绍其中一种通过 `InputStream` 的方式去创建 `Bitmap` 的方法，上一段从Gallery中获取照片并且将图片Size调整到合适手机尺寸的代码：

```
static final int PICK_PICS = 9;

public void startGallery(){
Intent i = new Intent();
i.setAction(Intent.ACTION_PICK);
i.setType("image/*");
startActivityForResult(i,PICK_PICS);
}

private int[] getScreenWithAndHeight(){
WindowManager wm = (WindowManager)
getSystemService(Context.WINDOW_SERVICE);
DisplayMetrics dm = new DisplayMetrics();
wm.getDefaultDisplay().getMetrics(dm);
return new int[]{dm.widthPixels,dm.heightPixels};
}

/**
*
```

- @param actualWidth 图片实际的宽度，也就是options.outWidth
- @param actualHeight 图片实际的高度，也就是options.outHeight
- @param desiredWidth 你希望图片压缩成为的目的宽度
- @param desiredHeight 你希望图片压缩成为的目的高度
- @return

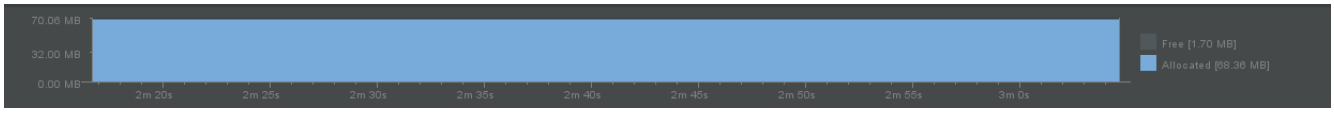
```
*/
private int findBestSampleSize(int actualWidth, int actualHeight, int desiredWidth,
int desiredHeight) {
double wr = (double) actualWidth / desiredWidth;
double hr = (double) actualHeight / desiredHeight;
```

```
        double ratio = Math.min(wr, hr);
        float n = 1.0f;
        //这里我们为什么要寻找 与ratio最接近的2的倍数呢？
        //原因就在于API中对于inSimpleSize的注释：最终的inSimpleSize应该为2的倍数，我们应该向上取与压缩比最接近的2的倍数。
        while ((n * 2) <= ratio) {
        n *= 2;
        }

        return (int) n;
        }

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
if(resultCode == RESULT_OK){
switch (requestCode){
case PICK_PICS:
Uri uri = data.getData();
InputStream is = null;
try {
is = getContentResolver().openInputStream(uri);
} catch (FileNotFoundException e) {
e.printStackTrace();
}
```

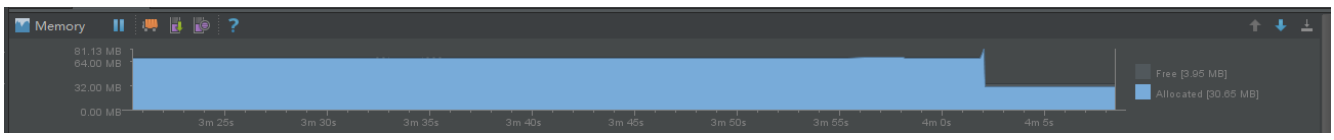
```
            BitmapFactory.Options options = new BitmapFactory.Options();
            //当这个参数为true的时候,意味着你可以在解析时候不申请内存的情况下去获取Bitmap的
宽和高
            //这是调整Bitmap Size一个很重要的参数设置
            options.inJustDecodeBounds = true;
            BitmapFactory.decodeStream( is,null,options );

            int realHeight = options.outHeight;
            int realWidth = options.outWidth;

            int screenWidth = getScreenWithAndHeight()[0];

            int simpleSize = findBestSampleSize(realWidth,realHeight,screenWidth,300);
            options.inSampleSize = simpleSize;
            //当你希望得到Bitmap实例的时候，不要忘了将这个参数设置为false
            options.inJustDecodeBounds = false;

            try {
                is = getContentResolver().openInputStream(uri);
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            }

            Bitmap bitmap = BitmapFactory.decodeStream(is,null,options);

            iv.setImageBitmap(bitmap);

            try {
                is.close();
                is = null;
            } catch (IOException e) {
                e.printStackTrace();
            }

            break;
        }
    }
    super.onActivityResult(requestCode, resultCode, data);

}
```

我们来看看这段代码的功效：

压缩前：



压缩后：



对比条件为：1080P的魅族Note3拍摄的高清无码照片

1. Reuse 上面介绍了 BitmapFactory 通过 InputStream 去创建 Bitmap 的这种方式，以及 BitmapFactory.Options.inSimpleSize 和 BitmapFactory.Options.inJustDecodeBounds 的使用方法，但将单个Bitmap加载到UI是简单的，但是如果我们需要一次性加载大量的图片，事情就会变得复杂起来。 Bitmap 是吃内存大户，我们不希望多次解析相同的 Bitmap ，也不希望可能不会用到的 Bitmap 一直存在于内存中，所以，这个场景下， Bitmap 的重用变得异常的重要。
   在这里只介绍一种 BitmapFactory.Options.inBitmap 的重用方式，下一篇文章会介绍使用三级缓存来实现Bitmap的重用。

   根据官方文档在Android 3.0 引进了BitmapFactory.Options.inBitmap，如果这个值被设置了，decode方法会在加载内容的时候去重用已经存在的bitmap. 这意味着bitmap的内存是被重新利用的，这样可以提升性能, 并且减少了内存的分配与回收。然而，使用inBitmap有一些限制。特别是在Android 4.4 之前，只支持同等大小的位图。
   我们看来看看这个参数最基本的运用方法。

   ```
   new BitmapFactory.Options options = new BitmapFactory.Options();
   //inBitmap只有当inMutable为true的时候是可用的。
   options.inMutable = true;
   Bitmap reusedBitmap = BitmapFactory.decodeResource(getResources(),R.drawable.reused_btimap,options);
   options.inBitmap = reusedBitmap;
   ```

   这样，当你在下一次decodeBitmap的时候，将设置了 options.inMutable=true 以及 options.inBitmap 的 Options 传入，Android就会复用你的Bitmap了，具体实例：

```
@Override
protected void onCreate(@Nullable Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(reuseBitmap());
}

private LinearLayout reuseBitmap(){
    LinearLayout linearLayout = new LinearLayout(this);
    linearLayout.setLayoutParams(new ViewGroup.LayoutParams(ViewGroup.LayoutParams
.MATCH_PARENT, ViewGroup.LayoutParams.MATCH_PARENT));
    linearLayout.setOrientation(LinearLayout.VERTICAL);

    ImageView iv = new ImageView(this);
    iv.setLayoutParams(new ViewGroup.LayoutParams(500,300));

    options = new BitmapFactory.Options();
    options.inJustDecodeBounds = true;
    //inBitmap只有当inMutable为true的时候是可用的。
    options.inMutable = true;
    BitmapFactory.decodeResource(getResources(),R.drawable.big_pic,options);

    //压缩Bitmap到我们希望的尺寸
    //确保不会OOM
    options.inSampleSize = findBestSampleSize(options.outWidth,options.outHeight,500,3
00);
    options.inJustDecodeBounds = false;

    Bitmap bitmap = BitmapFactory.decodeResource(getResources(),R.drawable.big_pic,op
tions);
    options.inBitmap = bitmap;

    iv.setImageBitmap(bitmap);

    linearLayout.addView(iv);

    ImageView iv1 = new ImageView(this);
    iv1.setLayoutParams(new ViewGroup.LayoutParams(500,300));
    iv1.setImageBitmap( BitmapFactory.decodeResource(getResources(),R.drawable.big_pic
,options));
    linearLayout.addView(iv1);

    ImageView iv2 = new ImageView(this);
    iv2.setLayoutParams(new ViewGroup.LayoutParams(500,300));
    iv2.setImageBitmap( BitmapFactory.decodeResource(getResources(),R.drawable.big_pic
,options));
    linearLayout.addView(iv2);


    return linearLayout;
}
```

以上代码中，我们在解析了一次一张1080P分辨率的图片，并且设置在 options.inBitmap

中，然后分别decode了同一张图片，并且传入了相同的`options`。最终只占用一份第一次解析`Bitmap`的内存。

2. Recycle 一定要记得及时回收Bitmap，否则如上分析，你的native以及dalvik的内存都会被一直占用着，最终导致OOM

```
// 先判断是否已经回收
if(bitmap != null && !bitmap.isRecycled()){
    // 回收并且置为null
    bitmap.recycle();
    bitmap = null;
}
System.gc();
```

- Enjoy Android :) 如果有误，轻喷，欢迎指正。

热修复的原理

热修复技术

APP提早发出去的包，如果出现客户端的问题，实在是干着急，覆水难收。因此线上修复方案迫在眉睫。

概述

基于Xposed中的思想，通过修改c层的Method实例描述，来实现更改与之对应的java方法的行为，从而达到修复的目的。

Xposed

诞生于XDA论坛，类似一个应用平台，不同的是其提供诸多系统级的应用。可实现许多神奇的功能。Xposed需要以越狱为前提，像是iOS中的cydia。

Xposed可以修改任何程序的任何java方法（需root），github上提供了XposedInstaller，是一个android app。提供很多framework层，应用层级的程序。开发者可以为其开发一些系统或应用方面的插件，自定义android系统，它甚至可以做动态权限管理（XposedMods）。

Android系统启动与应用启动

Zygote进程是Android手机系统启动后，常驻的一个名为‘受精卵’的进程。

- zygote的启动实现脚本在/init.rc文件中
- 启动过程中执行的二进制文件在/system/bin/app_process

任何应用程序启动时，会从zygote进程fork出一个新的进程。并装载一些必要的class，invoke一些初始化方法。这其中包括像：

- ActivityThread
- ServiceThread
- ApplicationPackageManager

等应用启动中必要的类，触发必要的方法，比如：handleBindApplication，将此进程与对应的应用绑定的初始化方法；同时，会将zygote进程中的dalvik虚拟机实例复制一份，因此每个应用程序进程都有自己的dalvik虚拟机实例；会将已有Java运行时加载到进程中；会注册一些android核心类的jni方法到虚拟机中，支撑从c到java的启动过程。

Xposed做了手脚

Xposed在这个过程改写了app_process(源码在Xposed : a modified app_process binary)，替换/system/bin/app_process这个二进制文件。然后做了两个事：

1. 通过Xposed的hook技术，在上述过程中，对上面提到的那些加载的类的方法hook。
2. 加载XposedBridge.jar

这时hook必要的方法是为了方便开发者为它开发插件，加载XposedBridge.jar是为动态hook提供了基础。在这个时候加载它意味着，所有的程序在启动时，都可以加载这个jar（因为上面提到的fork过程）。结合hook技术，从而达到了控制所有程序的所有方法。

为获得/system/bin/目录的读写权限，因而需要以root为前提。

Xposed的hook思想

那么Xposed是怎么hook java方法的呢？要从XposedBridge看起，重点在XposedBridge.hookmethod(原方法的Member对象，含有新方法的XC_MethodHook对象)；，这里会调到

```
private native synchronized static void hookMethodNative(Member method, Class<?> declaringClass,
 int slot, Object additionalInfo);
```

这个native的方法，通过这个方法，可以让所hook的方法，转向native层的一个c方法。如何做到？

```
When a transmit from java to native occurs, dvm sets up a native stack.
In dvmCallJNIMethod(), dvmPlatformInvoke is used to call the native method(signature in Method.ins
ns).
```

在jni这个中间世界里，类型数据由jni表来沟通java和c的世界；方法由c++指针结合DVM*系(如dvmSlotToMethod,dvmDecodeIndirectRef等方法)的api方法，操作虚拟机，从而实现java方法与c方法的世界。

那么hook的过程是这样：首先通过dexclassload来load所要hook的方法，分析类后，进c层，见代码XposedBridge_hookMethodNative方法，拿到要hook的Method类，然后通过dvmslotTomethod方法获取Method*指针，

```
Method* method = dvmSlotToMethod(declaredClass, slot);
```

declaredClass就是所hook方法所在的类，对应的jobject。slot是Method类中，描述此java对象在vm中的索引；那么通过这个方法，我们就获取了c层的Method指针,通过

```
SET_METHOD_FLAG(method, ACC_NATIVE);
```

将该方法标记为一个native方法，然后通过

```
method->nativeFunc = &hookedMethodCallback;
```

定向c层方法到hookedMethodCallback，这样当被hook的java方法执行时，就会调到c层的hookedMethodCallback方法。

通过meth->nativeFunc重定向MethodCallBridge到hookedMethodCallback这个方法上，控制这个c++指针是无视java的private的。

另外，在method结构体中有

```
method->insns = (const u2*) hookInfo;
```

用insns指向替换成为的方法，以便hookedMethodCallback可以获取真正期望执行的java方法。

现在所有被hook的方法，都指向了hookedMethodCallbackc方法中，然后在此方法中实现调用替换成为的java方法。

从Xposed提炼精髓

回顾Xposed，以root为必要条件，在app_process加载XposedBidge.jar，从而实现有hook所有应用的所有方法的能力；而后续动态hook应用内的方法，其实只是load了从zypote进程复制出来的运行时的这个XposedBidge.jar，然后hook而已。因此，若在一个应用范围内的hook，root不是必须的，只是单纯的加载hook的实现方法，即可修改本应用的方法。

业界内也不乏通过「修改BaseDexClassLoader中的pathList，来动态加载dex」方式实现热修复。后者纯java实现，但需要hack类的优化流程，将打CLASS_ISPREVERIFIED标签的类，去除此标签，以解决类与类引用不在一个dex中的异常问题。这会放弃dex optimize对启动运行速度的优化。原则上，这对于方法数没有大到需要multidex的应用，损失更明显。而前者不触犯原有的优化流程，只点杀需要hook的方法，更为纯粹、有效。

AIDL

AIDL

1. 创建一个接口，再里面定义方法

```
package com.example.taidl;
interface ICalcAIDL
{
    int add(int x , int y);
    int min(int x , int y );
}
```

build一下gen目录下会生成ICalcAIDL.java文件

```
/*
 * This file is auto-generated.  DO NOT MODIFY.
 * Original file: /Users/dream/Downloads/android/androidProject/TAIDL/src/com/example/taidl/ICalc
AIDL.aidl
 */
package com.example.taidl;
public interface ICalcAIDL extends android.os.IInterface
{
/** Local-side IPC implementation stub class. */
public static abstract class Stub extends android.os.Binder implements com.example.taidl.ICalcAIDL
{
private static final java.lang.String DESCRIPTOR = "com.example.taidl.ICalcAIDL";
/** Construct the stub at attach it to the interface. */
public Stub()
{
this.attachInterface(this, DESCRIPTOR);
}
/**
 * Cast an IBinder object into an com.example.taidl.ICalcAIDL interface,
 * generating a proxy if needed.
 */
public static com.example.taidl.ICalcAIDL asInterface(android.os.IBinder obj)
{
if ((obj==null)) {
return null;
}
android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);
if (((iin!=null)&&(iin instanceof com.example.taidl.ICalcAIDL))) {
return ((com.example.taidl.ICalcAIDL)iin);
}
return new com.example.taidl.ICalcAIDL.Stub.Proxy(obj);
}
@Override public android.os.IBinder asBinder()
```

```
{
return this;
}
@Override public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int fla
gs) throws android.os.RemoteException
{
switch (code)
{
case INTERFACE_TRANSACTION:
{
reply.writeString(DESCRIPTOR);
return true;
}
case TRANSACTION_add:
{
data.enforceInterface(DESCRIPTOR);
int _arg0;
_arg0 = data.readInt();
int _arg1;
_arg1 = data.readInt();
int _result = this.add(_arg0, _arg1);
reply.writeNoException();
reply.writeInt(_result);
return true;
}
case TRANSACTION_min:
{
data.enforceInterface(DESCRIPTOR);
int _arg0;
_arg0 = data.readInt();
int _arg1;
_arg1 = data.readInt();
int _result = this.min(_arg0, _arg1);
reply.writeNoException();
reply.writeInt(_result);
return true;
}
}
return super.onTransact(code, data, reply, flags);
}
private static class Proxy implements com.example.taidl.ICalcAIDL
{
private android.os.IBinder mRemote;
Proxy(android.os.IBinder remote)
{
mRemote = remote;
}
@Override public android.os.IBinder asBinder()
{
return mRemote;
}
public java.lang.String getInterfaceDescriptor()
{
return DESCRIPTOR;
```

```
}
@Override public int add(int x, int y) throws android.os.RemoteException
{
android.os.Parcel _data = android.os.Parcel.obtain();
android.os.Parcel _reply = android.os.Parcel.obtain();
int _result;
try {
_data.writeInterfaceToken(DESCRIPTOR);
_data.writeInt(x);
_data.writeInt(y);
mRemote.transact(Stub.TRANSACTION_add, _data, _reply, 0);
_reply.readException();
_result = _reply.readInt();
}
finally {
_reply.recycle();
_data.recycle();
}
return _result;
}
@Override public int min(int x, int y) throws android.os.RemoteException
{
android.os.Parcel _data = android.os.Parcel.obtain();
android.os.Parcel _reply = android.os.Parcel.obtain();
int _result;
try {
_data.writeInterfaceToken(DESCRIPTOR);
_data.writeInt(x);
_data.writeInt(y);
mRemote.transact(Stub.TRANSACTION_min, _data, _reply, 0);
_reply.readException();
_result = _reply.readInt();
}
finally {
_reply.recycle();
_data.recycle();
}
return _result;
}
}
static final int TRANSACTION_add = (android.os.IBinder.FIRST_CALL_TRANSACTION + 0);
static final int TRANSACTION_min = (android.os.IBinder.FIRST_CALL_TRANSACTION + 1);
}
public int add(int x, int y) throws android.os.RemoteException;
public int min(int x, int y) throws android.os.RemoteException;
}
```

1. 新建一个Service

```
package com.example.taidl;

import android.app.Service;
import android.content.Intent;
import android.os.IBinder;
```

```
import android.os.IBinder;
import android.os.RemoteException;
import android.util.Log;

public class CalcService extends Service{

    private static final String TAG = "server";

    public void onCreate()
    {
        Log.e(TAG, "onCreate");
    }

    public IBinder onBind(Intent t)
    {
        Log.e(TAG, "onBind");
        return mBinder;
    }

    public void onDestroy()
    {
        Log.e(TAG, "onDestroy");
        super.onDestroy();
    }

    public boolean onUnbind(Intent intent)
    {
        Log.e(TAG, "onUnbind");
        return super.onUnbind(intent);
    }

    public void onRebind(Intent intent)
    {
        Log.e(TAG, "onRebind");
        super.onRebind(intent);
    }
    private final ICalcAIDL.Stub mBinder = new ICalcAIDL.Stub() {

        @Override
        public int min(int x, int y) throws RemoteException {
            return x + y;
        }

        @Override
        public int add(int x, int y) throws RemoteException {
            // TODO Auto-generated method stub
            return x - y;
        }
    };



}
```

创建了一个mBinder对象，并在Service的onBind方法中返回

注册：

```
        <service android:name="com.example.taidl.CalcService">
            <intent-filter>
                <action android:name="com.example.taidl.calc" />

                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </service>
```

我们一会会在别的应用程序中通过Intent来查找此Service；这个不需要Activity，所以我也就没写Activity，安装完成也看不到安装图标，悄悄在后台运行着。服务端编写完毕。下面开始编写客户端:

```
package com.example.tclient;

import com.example.taidl.ICalcAIDL;

import android.app.Activity;
import android.content.ComponentName;
import android.content.Context;
import android.content.Intent;
import android.content.ServiceConnection;
import android.os.Bundle;
import android.os.IBinder;
import android.util.Log;
import android.view.View;
import android.widget.Toast;

public class MainActivity extends Activity {

    private ICalcAIDL mCalcAidl;

    private ServiceConnection mServiceConn = new ServiceConnection()
    {
        @Override
        public void onServiceDisconnected(ComponentName name)
        {
            Log.e("client", "onServiceDisconnected");
            mCalcAidl = null;
        }

        @Override
        public void onServiceConnected(ComponentName name, IBinder service)
        {
            Log.e("client", "onServiceConnected");
            mCalcAidl = ICalcAIDL.Stub.asInterface(service);
        }
    };

    @Override
```

```
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);

}

/**
 * 点击BindService按钮时调用
 * @param view
 */
public void bindService(View view)
{
    Intent intent = new Intent();
    intent.setAction("com.example.taidl.calc");
    bindService(intent, mServiceConn, Context.BIND_AUTO_CREATE);
}
/**
 * 点击unBindService按钮时调用
 * @param view
 */
public void unbindService(View view)
{
    unbindService(mServiceConn);
}
/**
 * 点击12+12按钮时调用
 * @param view
 */
public void addInvoked(View view) throws Exception
{

    if (mCalcAidl != null)
    {
        int addRes = mCalcAidl.add(12, 12);
        Toast.makeText(this, addRes + "", Toast.LENGTH_SHORT).show();
    } else
    {
        Toast.makeText(this, "服务器被异常杀死，请重新绑定服务端", Toast.LENGTH_SHORT)
                .show();

    }

}
/**
 * 点击50-12按钮时调用
 * @param view
 */
public void minInvoked(View view) throws Exception
{

    if (mCalcAidl != null)
    {
        int addRes = mCalcAidl.min(50, 12);
```

```
            Toast.makeText(this, addRes + "", Toast.LENGTH_SHORT).show();
        } else
        {
            Toast.makeText(this, "服务器未绑定或被异常杀死，请重新绑定服务端", Toast.LENGTH_SHORT)
                    .show();

        }

    }
}
```

将服务端的aidl文件完整的复制过来，包名一定要一致。

分析AIDL生成的代码

1. 服务端

```
private final ICalcAIDL.Stub mBinder = new ICalcAIDL.Stub()
    {

        @Override
        public int add(int x, int y) throws RemoteException
        {
            return x + y;
        }

        @Override
        public int min(int x, int y) throws RemoteException
        {
            return x - y;
        }

    };
```

ICalcAILD.Stub来执行的，让我们来看看Stub这个类的声明：

```
public static abstract class Stub extends android.os.Binder implements com.zhy.calc.aidl.ICalcAIDL
```

清楚的看到这个类是Binder的子类，是不是符合我们文章开通所说的服务端其实是一个Binder类的实例
接下来看它的onTransact()方法：

```
@Override public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply, int fla
gs) throws android.os.RemoteException
{
switch (code)
{
case INTERFACE_TRANSACTION:
{
reply.writeString(DESCRIPTOR);
return true;
}
case TRANSACTION_add:
{
data.enforceInterface(DESCRIPTOR);
int _arg0;
_arg0 = data.readInt();
int _arg1;
_arg1 = data.readInt();
int _result = this.add(_arg0, _arg1);
reply.writeNoException();
reply.writeInt(_result);
return true;
}
case TRANSACTION_min:
{
data.enforceInterface(DESCRIPTOR);
int _arg0;
_arg0 = data.readInt();
int _arg1;
_arg1 = data.readInt();
int _result = this.min(_arg0, _arg1);
reply.writeNoException();
reply.writeInt(_result);
return true;
}
}
return super.onTransact(code, data, reply, flags);
}
```

文章开头也说到服务端的Binder实例会根据客户端依靠Binder驱动发来的消息，执行onTransact方法，然后由其参数决定执行服务端的代码。
可以看到onTransact有四个参数
code ， data ，replay ， flags

- code 是一个整形的唯一标识，用于区分执行哪个方法，客户端会传递此参数，告诉服务端执行哪个方法
- data客户端传递过来的参数
- replay服务器返回回去的值
- flags标明是否有返回值，0为有（双向），1为没有（单向）

我们仔细看case TRANSACTION_min中的代码

data.enforceInterface(DESCRIPTOR);

与客户端的writeInterfaceToken对用，标识远程服务的名称

```
int _arg0;
_arg0 = data.readInt();
int _arg1;
_arg1 = data.readInt();
```

接下来分别读取了客户端传入的两个参数

```
int _result = this.min(_arg0, _arg1);
reply.writeNoException();
reply.writeInt(_result);
```

然后执行this.min，即我们实现的min方法；返回result由reply写回。

add同理，可以看到服务端通过AIDL生成Stub的类，封装了服务端本来需要写的代码。

客户端

客户端主要通过ServiceConnected与服务端连接

```
private ServiceConnection mServiceConn = new ServiceConnection()
    {
        @Override
        public void onServiceDisconnected(ComponentName name)
        {
            Log.e("client", "onServiceDisconnected");
            mCalcAidl = null;
        }

        @Override
        public void onServiceConnected(ComponentName name, IBinder service)
        {
            Log.e("client", "onServiceConnected");
            mCalcAidl = ICalcAIDL.Stub.asInterface(service);
        }
    };
```

如果你比较敏锐，应该会猜到这个onServiceConnected中的IBinder实例，其实就是我们文章开通所说的Binder驱动，也是一个Binder实例
在ICalcAIDL.Stub.asInterface中最终调用了：

```
return new com.zhy.calc.aidl.ICalcAIDL.Stub.Proxy(obj);
```

这个Proxy实例传入了我们的Binder驱动，并且封装了我们调用服务端的代码，文章开头说，客户端会通过Binder驱动的transact()方法调用服务端代码

直接看Proxy中的add方法

```
@Override public int add(int x, int y) throws android.os.RemoteException
{
android.os.Parcel _data = android.os.Parcel.obtain();
android.os.Parcel _reply = android.os.Parcel.obtain();
int _result;
try {
_data.writeInterfaceToken(DESCRIPTOR);
_data.writeInt(x);
_data.writeInt(y);
mRemote.transact(Stub.TRANSACTION_add, _data, _reply, 0);
_reply.readException();
_result = _reply.readInt();
}
finally {
_reply.recycle();
_data.recycle();
}
return _result;
}
```

首先声明两个Parcel对象，一个用于传递数据，一个用户接收返回的数据

```
_data.writeInterfaceToken(DESCRIPTOR);与服务器端的enforceInterfac对应
_data.writeInt(x);
_data.writeInt(y);写入需要传递的参数
mRemote.transact(Stub.TRANSACTION_add, _data, _reply, 0);
```

终于看到了我们的transact方法，第一个对应服务端的code,_data,_repay分别对应服务端的data，reply，0表示是双向的

```
_reply.readException();
_result = _reply.readInt();
```

最后读出我们服务端返回的数据，然后return。可以看到和服务端的onTransact基本是一行一行对应的。

我们已经通过AIDL生成的代码解释了Android Binder框架的工作原理。Service的作用其实就是为我们创建Binder驱动，即服务端与客户端连接的桥梁。

Binder机制

Binder机制

首先Binder是Android系统进程间通信(IPC)方式之一。

Binder使用Client - Server通信方式。Binder框架定义了四个角色：Server,Client,ServiceManager以及Binder驱动。其中Server,Client,ServiceManager运行于用户空间，驱动运行于内核空间。Binder驱动程序提供设备文件/dev/binder与用户空间交互，Client、Server和Service Manager通过open和ioctl文件操作函数与Binder驱动程序进行通信。

Server创建了Binder实体，为其取一个字符形式，可读易记的名字，将这个Binder连同名字以数据包的形式通过Binder驱动发送给ServiceManager，通知ServiceManager注册一个名字为XX的Binder，它位于Server中。驱动为这个穿过进程边界的Binder创建位于内核中的实体结点以及ServiceManager对实体的引用，将名字以及新建的引用打包给ServiceManager。ServiceManager收数据包后，从中取出名字和引用填入一张查找表中。但是一个Server若向ServiceManager注册自己Binder就必须通过0这个引用和ServiceManager的Binder通信。Server向ServiceManager注册了Binder实体及其名字后，Client就可以通过名字获得该Binder的引用了。Clent也利用保留的0号引用向ServiceManager请求访问某个Binder：我申请名字叫XX的Binder的引用。ServiceManager收到这个连接请求，从请求数据包里获得Binder的名字，在查找表里找到该名字对应的条目，从条目中取出Binder引用，将该引用作为回复发送给发起请求的Client。

当然，不是所有的Binder都需要注册给ServiceManager广而告之的。Server端可以通过已经建立的Binder连接将创建的Binder实体传给Client，当然这条已经建立的Binder连接必须是通过实名Binder实现。由于这个Binder没有向ServiceManager注册名字，所以是匿名Binder。Client将会收到这个匿名Binder的引用，通过这个引用向位于Server中的实体发送请求。匿名Binder为通信双方建立一条私密通道，只要Server没有把匿名Binder发给别的进程，别的进程就无法通过穷举或猜测等任何方式获得该Binder的引用，向该Binder发送请求。

为什么Binder只进行了一次数据拷贝？

Linux内核实际上没有从一个用户空间到另一个用户空间直接拷贝的函数，需要先用copy_from_user()拷贝到内核空间，再用copy_to_user()拷贝到另一个用户空间。为了实现用户空间到用户空间的拷贝，mmap()分配的内存除了映射进了接收方进程里，还映射进了内核空间。所以调用copy_from_user()将数据拷贝进内核空间也相当于拷贝进了接收方的用户空间，这就是Binder只需一次拷贝的‘秘密’。

最底层的是Android的ashmen(Anonymous shared memory)机制，它负责辅助实现内存的分配，以及跨进程所需要的内存共享。AIDL(android interface definition language)对Binder的使用进行了封装，可以让开发者方便的进行方法的远程调用，后面会详细介绍。Intent是最高一层的抽象，方便开发者进行常用的跨进程调用。

从英文字面上意思看，Binder具有粘结剂的意思那么它是把什么东西粘接在一起呢？在Android系统的Binder机制中，由一系统组件组成，分别是Client、Server、Service Manager和Binder驱动，其中

Client、Server、Service Manager运行在用户空间，Binder驱动程序运行内核空间。Binder就是一种把这四个组件粘合在一起的粘连剂了，其中，核心组件便是Binder驱动程序了，ServiceManager提供了辅助管理的功能，Client和Server正是Binder驱动和ServiceManager提供的基础设施上，进行Client-Server之间的通信。

1. Client、Server和ServiceManager实现在用户空间中，Binder驱动实现在内核空间中
2. Binder驱动程序和ServiceManager在Android平台中已经实现，开发者只需要在用户空间实现自己的Client和Server
3. Binder驱动程序提供设备文件/dev/binder与用户空间交互，Client、Server和ServiceManager通过open和ioctl文件操作函数与Binder驱动程序进行通信
4. Client和Server之间的进程间通信通过Binder驱动程序间接实现
5. ServiceManager是一个守护进程，用来管理Server，并向Client提供查询Server接口的能力

服务器端：一个Binder服务器就是一个Binder类的对象。当创建一个Binder对象后，内部就会开启一个线程，这个线程用户接收binder驱动发送的消息，收到消息后，会执行相关的服务代码。

Binder驱动：当服务端成功创建一个Binder对象后，Binder驱动也会相应创建一个mRemote对象，该对象的类型也是Binder类，客户就可以借助这个mRemote对象来访问远程服务。

客户端：客户端要想访问Binder的远程服务，就必须获取远程服务的Binder对象在binder驱动层对应的binder驱动层对应的mRemote引用。当获取到mRemote对象的引用后，就可以调用相应Binde对象的服务了。

在这里我们可以看到，客户是通过Binder驱动来调用服务端的相关服务。首先，在服务端创建一个Binder对象，接着客户端通过获取Binder驱动中Binder对象的引用来调用服务端的服务。在Binder机制中正是借着Binder驱动将不同进程间的组件bind(粘连)在一起，实现通信。

mmap将一个文件或者其他对象映射进内存。文件被映射进内存。文件被映射到多个页上，如果文件的大小不是所有页的大小之和，最后一个页不被使用的空间将会凋零。munmap执行相反的操作，删除特定地址区域的对象映射。

当使用mmap映射文件到进程后，就可以直接操作这段虚拟地址进行文件的读写等操作，不必再调用read,write等系统调用。但需注意，直接对该段内存写时不会写入超过当前文件大小的内容。

使用共享内存通信的一个显而易见的好处是效率高，因为进程可以直接读写内存，而不需要任何数据的拷贝。对于像管道和消息队列等通信方式，则需要在内核和用户空间进行四次的数据拷贝，而共享内存则只拷贝两次内存数据：一次从输入文件到共享内存区，另一次从共享内存到输出文件。实际上，进程之间在共享内存时，并不总是读写少量数据后就解除映射，有新的通信时，再重新建立共享内存区域，而是保持共享区域，直到通信完成为止，这样，数据内容一直保存在共享内存中，并没有写回文件。共享内存中的内容往往是在解除内存映射时才写回文件的。因此，采用共享内存的通信方式效率是非常高的。

aidl主要就帮助了我们完成了包装数据和解包的过程，并调用了transact过程，而用来传递的数据包我们就称为parcel

AIDL:xxx.aidl -> xxx.java ,注册service

1. 用aidl定义需要被调用方法接口
2. 实现这些方法
3. 调用这些方法

Zygote和System进程的启动过程

Zygote和System进程的启动过程

init脚本的启动

```
+------------+    +-------+   +-----------+
|Linux Kernel+--> |init.rc+-> |app_process|
+------------+    +-------+   +-----------+
                create and public
                  server socket
```

linux内核加载完成后，运行init.rc脚本

```
service zygote /system/bin/app_process -Xzygote /system/bin --zygote --start-system-server socket z
ygote stream 666
```

- /system/bin/app_process Zygote服务启动的进程名
- --start-system-server 表明Zygote启动完成之后，要启动System进程。
- socket zygote stream 666 在Zygote启动时，创建一个权限为666的socket。此socket用来请求Zygote创建新进程。socket的fd保存在名称为“ANDROID_SOCKET_zygote”的环境变量中。

Zygote进程的启动过程

app_process进程

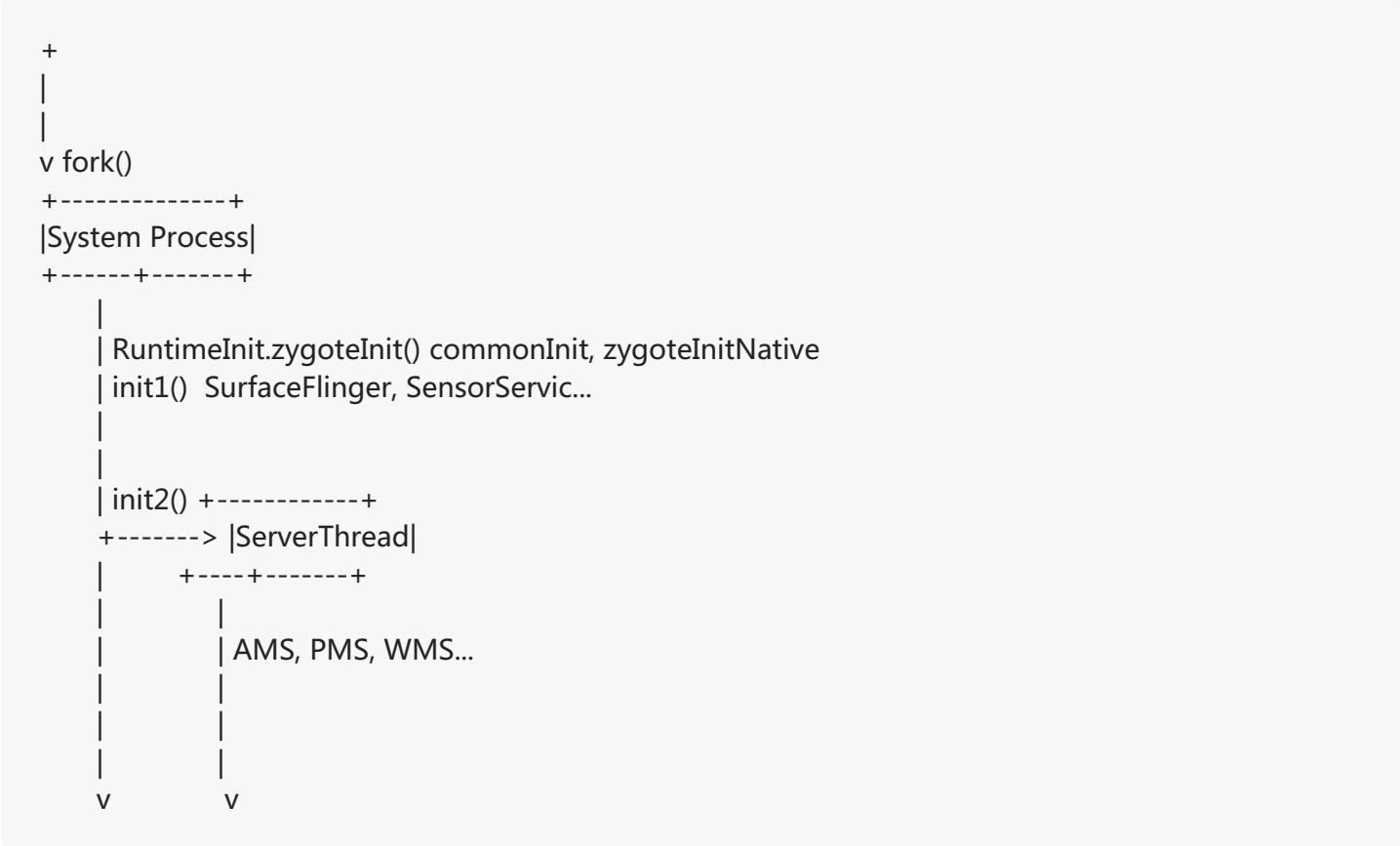
/system/bin/app_process 启动时创建了一个AppRuntime对象。通过AppRuntime对象的start方法，通过JNI调用创建了一个虚拟机实例，然后运行com.android.internal.os.ZygoteInit类的静态main方法，传递true(boolean startSystemServer)参数。

ZygoteInit类

ZygoteInit类的main方法运行时，会通过registerZygoteSocket方法创建一个供ActivityManagerService使用的server socket。然后通过调用startSystemServer方法来启动System进程。最后通过runSelectLoopMode来等待AMS的新建进程请求。

1. 在registerZygoteSocket方法中，通过名为ANDROID_SOCKET_zygote的环境获取到zygote启动时创建的socket的fd，然后以此来创建server socket。
2. 在startSystemServer方法中，通过Zygote.forkSystemServer方法创建了一个子进程，并将其用户和用户组的ID设置为1000。
3. 在runSelectLoopMode方法中，会将之前建立的server socket保存起来。然后进入一个无限循环，在其中通过selectReadable方法，监听socket是否有数据可读。有数据则说明接收到了一个请求。selectReadable方法会返回一个整数值index。如果index为0，则说明这个是AMS发过来的连接请求。这时会与AMS建立一个新的socket连接，并包装成ZygoteConnection对象保存起来。如果index大于0，则说明这是AMS发过来的一个创建新进程的请求。此时会取出之前保存的ZygoteConnection对象，调用其中的runOnce方法创建新进程。调用完成后将connection删除。
   这就是Zygote处理一次AMS请求的过程。

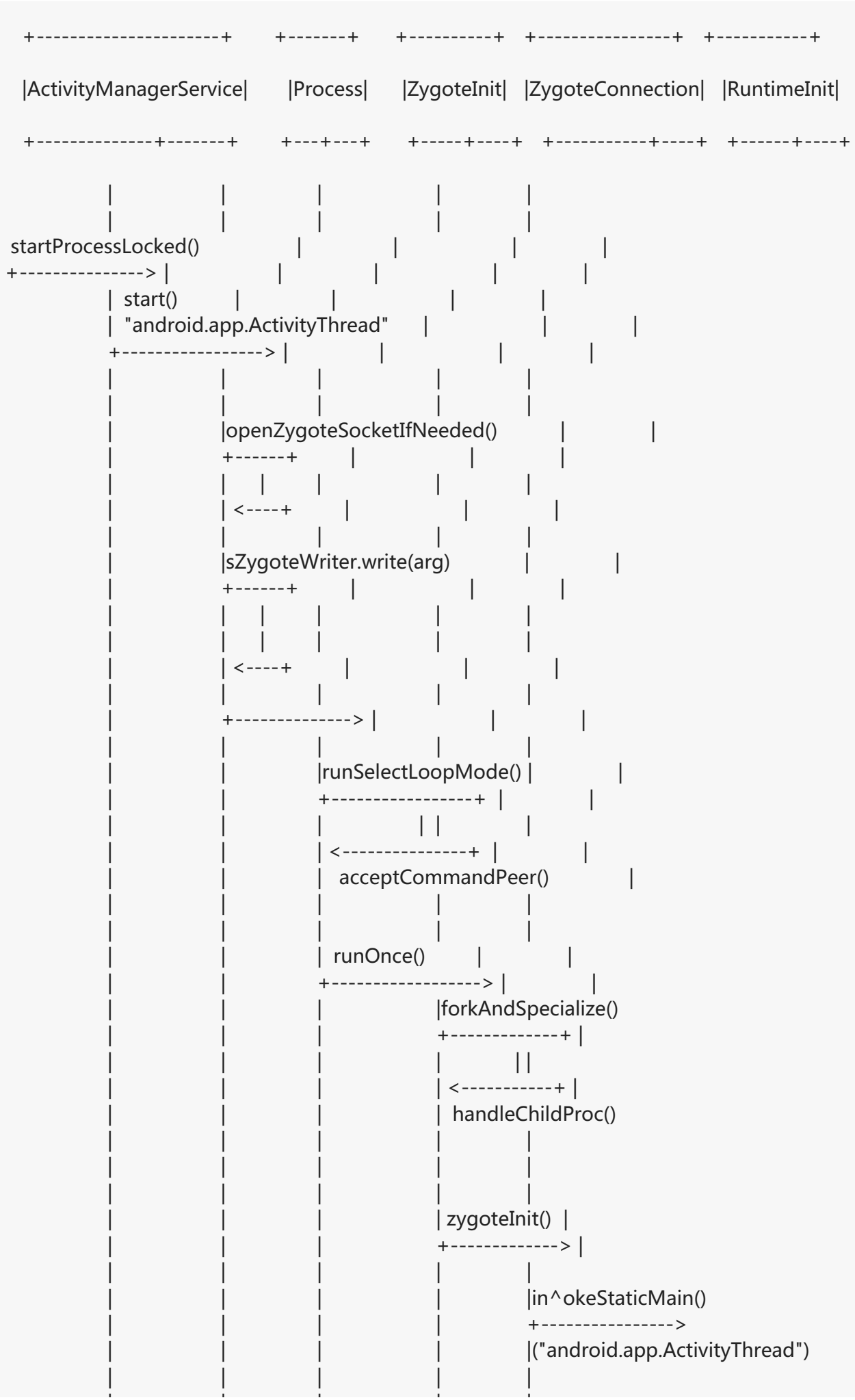
System进程的启动



System进程是在ZygoteInit的handleSystemServerProcess中开始启动的。

1. 首先，因为System进程是直接fork Zygote进程的，所以要先通过closeServerSocket方法关掉server socket。
2. 调用RuntimeInit.zygoteInit方法进一步启动System进程。在zygoteInit中，通过commonInit方法设置时区和键盘布局等通用信息，然后通过zygoteInitNative方法启动了一个Binder线程池。最后通过invokeStaticMain方法调用SystemServer类的静态Main方法。
3. SystemServer类的main通过JNI调用cpp实现的init1方法。在init1方法中，会启动各种以C++开发的系统服务（例如SurfaceFlinger和SensorService）。然后回调ServerServer类的init2方法来启动以Java开发的系统服务。
4. 在init2方法中，首先会新建名为"android.server.ServerThread"的ServerThread线程，并调用其start方法。然后在该线程中启动各种Service（例如AMS，PMS，WMS等）。启动的方式是调用对应Service类的静态main方法。
5. 首先，AMS会被创建，但未注册到ServerManager中。然后PMS被创建，AMS这时候才注册到ServerManager中。然后到ContentService、WMS等。
   注册到ServerManager中时会制定Service的名字，其后其他进程可以通过这个名字来获取到Binder Proxy对象，以访问Service提供的服务。
6. 执行到这里，System就将系统的关键服务启动起来了，这时候其他进程便可利用这些Service提供的基础服务了。
7. 最后会调用ActivityManagerService的systemReady方法，在该方法里会启动系统界面以及Home程序。

Android进程启动


ActivityManagerService
Process
ZygoteInit
ZygoteConnection
RuntimeInit
startProcessLocked()
start()
"android.app.ActivityThread"
openZygoteSocketIfNeeded()
sZygoteWriter.write(arg)
runSelectLoopMode()
acceptCommandPeer()
runOnce()
forkAndSpecialize()
handleChildProc()
zygoteInit()
in^okeStaticMain()
("android.app.ActivityThread")

```
          |          |          |          |          |
          +          +          +          +          +
```

- AMS向Zygote发起请求（通过之前保存的socket），携带各种参数，包括“android.app.ActivityThread”。
- Zygote进程fork自己，然后在新Zygote进程中调用RuntimeInit.zygoteInit方法进行一系列的初始化（commonInit、Binder线程池初始化等）。
- 新Zygote进程中调用ActivityThread的main函数，并启动消息循环。

Android中的MVC，MVP和MVVM

MVP

MVP

为什么需要MVP

1. 尽量简单
   大部分的安卓应用只使用View-Model结构,程序员现在更多的是和复杂的View打交道而不是解决业务逻辑。当你在应用中只使用Model-View时，到最后，你会发现“所有的事物都被连接到一起”。复杂的任务被分成细小的任务，并且很容易解决。越小的东西，bug越少，越容易debug，更好测试。在MVP模式下的View层将会变得简单，所以即便是他请求数据的时候也不需要回调函数。View逻辑变成十分直接。
2. 后台任务
   当你编写一个Actviity、Fragment、自定义View的时候，你会把所有的和后台任务相关的方法写在一个静态类或者外部类中。这样，你的Task不再和Activity联系在一起，这既不会导致内存泄露，也不依赖于Activity的重建。

优缺点

优点：

1. 降低耦合度，实现了Model和View真正的完全分离，可以修改View而不影响Modle
2. 模块职责划分明显，层次清晰
3. 隐藏数据
4. Presenter可以复用，一个Presenter可以用于多个View，而不需要更改Presenter的逻辑（当然是在View的改动不影响业务逻辑的前提下）

5. 利于测试驱动开发。以前的Android开发是难以进行单元测试的（虽然很多Android开发者都没有写过测试用例，但是随着项目变得越来越复杂，没有测试是很难保证软件质量的；而且近几年来Android上的测试框架已经有了长足的发展——开始写测试用例吧），在使用MVP的项目中Presenter对View是通过接口进行，在对Presenter进行不依赖UI环境的单元测试的时候。可以通过Mock一个View对象，这个对象只需要实现了View的接口即可。然后依赖注入到Presenter中，单元测试的时候就可以完整的测试Presenter应用逻辑的正确性。
6. View可以进行组件化。在MVP当中，View不依赖Model。这样就可以让View从特定的业务场景中脱离出来，可以说View可以做到对业务完全无知。它只需要提供一系列接口提供给上层操作。这样就可以做到高度可复用的View组件。
7. 代码灵活性

缺点：

1. Presenter中除了应用逻辑以外，还有大量的View->Model，Model->View的手动同步逻辑，造成Presenter比较笨重，维护起来会比较困难。
2. 由于对视图的渲染放在了Presenter中，所以视图和Presenter的交互会过于频繁。
3. 如果Presenter过多地渲染了视图，往往会使得它与特定的视图的联系过于紧密。一旦视图需要变更，那么Presenter也需要变更了。
4. 额外的代码复杂度及学习成本。

在MVP模式里通常包含4个要素：

1. View :负责绘制UI元素、与用户进行交互(在Android中体现为Activity);
2. View interface :需要View实现的接口，View通过View interface与Presenter进行交互，降低耦合，方便进行单元测试;
3. Model :负责存储、检索、操纵数据(有时也实现一个Model interface用来降低耦合);
4. Presenter :作为View与Model交互的中间纽带，处理与用户交互的负责逻辑。

Android开机过程

Android开机过程

- BootLoder引导,然后加载Linux内核.
- 0号进程init启动.加载init.rc配置文件,配置文件有个命令启动了zygote进程
- zygote开始fork出SystemServer进程
- SystemServer加载各种JNI库,然后init1,init2方法,init2方法中开启了新线程ServerThread.
- 在SystemServer中会创建一个socket客户端，后续AMS（ActivityManagerService）会通过此客户端和zygote通信
- ServerThread的run方法中开启了AMS,还孵化新进程ServiceManager,加载注册了一溜的服务,最后一

句话进入loop 死循环

- run方法的SystemReady调用resumeTopActivityLocked打开锁屏界面

EventBus用法详解

EventBus

概述

EventBus是一款针对Android优化的发布/订阅（publish/subscribe）事件总线。主要功能是替代Intent,Handler,BroadCast在Fragment，Activity，Service，线程之间传递消息。简化了应用程序内各组件间、组件与后台线程间的通信。优点是开销小，代码更优雅。以及将发送者和接收者解耦。比如请求网络，等网络返回时通过Handler或Broadcast通知UI，两个Fragment之间需要通过Listener通信，这些需求都可以通过EventBus实现。

EventBus作为一个消息总线，有三个主要的元素：

- Event：事件。可以是任意类型的对象
- Subscriber：事件订阅者，接收特定的事件。在EventBus中，使用约定来指定事件订阅者以简化使用。即所有事件订阅都都是以onEvent开头的函数，具体来说，函数的名字是onEvent,onEventMainThread，onEventBackgroundThread，onEventAsync这四个，这个和ThreadMode（下面讲）有关。
- Publisher：事件发布者，用于通知 Subscriber 有事件发生。可以在任意线程任意位置发送事件，直接调用eventBus.post(Object) 方法，可以自己实例化 EventBus
  对象，但一般使用默认的单例就好了：EventBus.getDefault()， 根据post函数参数的类型，会自动调用订阅相应类型事件的函数。

关于ThreadMode

前面说了，Subscriber的函数只能是那4个，因为每个事件订阅函数都是和一个ThreadMode相关联的，ThreadMode指定了会调用的函数。有以下四个ThreadMode：

- PostThread：事件的处理在和事件的发送在相同的进程，所以事件处理时间不应太长，不然影响事件的发送线程，而这个线程可能是UI线程。对应的函数名是onEvent。
- MainThread: 事件的处理会在UI线程中执行。事件处理时间不能太长，这个不用说的，长了会ANR的，对应的函数名是onEventMainThread。
- BackgroundThread：事件的处理会在一个后台线程中执行，对应的函数名是onEventBackgroundThread，虽然名字是BackgroundThread，事件处理是在后台线程，但事件处理时间还是不应该太长，因为如果发送事件的线程是后台线程，会直接执行事件，如果当前线程是UI

线程，事件会被加到一个队列中，由一个线程依次处理这些事件，如果某个事件处理时间太长，会阻塞后面的事件的派发或处理。

- Async：事件处理会在单独的线程中执行，主要用于在后台线程中执行耗时操作，每个事件会开启一个线程（有线程池），但最好限制线程的数目。

根据事件订阅都函数名称的不同，会使用不同的ThreadMode，比如果在后台线程加载了数据想在UI线程显示，订阅者只需把函数命名onEventMainThread。

对相应的函数名，进一步解释一下：

onEvent:如果使用onEvent作为订阅函数，那么该事件在哪个线程发布出来的，onEvent就会在这个线程中运行，也就是说发布事件和接收事件线程在同一个线程。使用这个方法时，在onEvent方法中不能执行耗时操作，如果执行耗时操作容易导致事件分发延迟。

onEventMainThread:如果使用onEventMainThread作为订阅函数，那么不论事件是在哪个线程中发布出来的，onEventMainThread都会在UI线程中执行，接收事件就会在UI线程中运行，这个在Android中是非常有用的，因为在Android中只能在UI线程中跟新UI，所以在onEvnetMainThread方法中是不能执行耗时操作的。

onEventBackground:如果使用onEventBackgrond作为订阅函数，那么如果事件是在UI线程中发布出来的，那么onEventBackground就会在子线程中运行，如果事件本来就是子线程中发布出来的，那么onEventBackground函数直接在该子线程中执行。

onEventAsync：使用这个函数作为订阅函数，那么无论事件在哪个线程发布，都会创建新的子线程在执行onEventAsync。

基本用法

引入EventBus:

```
compile 'org.greenrobot:eventbus:3.0.0'
```

定义事件:

```
public class MessageEvent { /* Additional fields if needed */ }
```

注册事件接收者：

```
eventBus.register(this);
```

发送事件:

```
eventBus.post(event)
```

接收消息并处理:

```
public void onEvent(MessageEvent event) {}
```

注销事件接收：

```
eventBus.unregister(this);
```

最后，proguard 需要做一些额外处理:

```
#EventBus
 -keepclassmembers class ** {
    public void onEvent*(**);
    void onEvent*(**);
 }
```

查漏补缺

查漏补缺

请分析一张400*500尺寸的PNG图片加载到程序中占用内存中的大小

http://m.blog.csdn.net/article/details?id=7856519

Git操作

Git 操作

git 命令

- 创建本地仓库

```
git init
```

- 获取远程仓库

```
git clone [url]
例：git clone https://github.com/you/yourpro.git
```

- 创建远程仓库

```
// 添加一个新的 remote 远程仓库
git remote add [remote-name] [url]
例：git remote add origin https://github.com/you/yourpro.git
origin：相当于该远程仓库的别名

// 列出所有 remote 的别名
git remote

// 列出所有 remote 的 url
git remote -v

// 删除一个 renote
git remote rm [name]

// 重命名 remote
git remote rename [old-name] [new-name]
```

- 从本地仓库中删除

```
git rm file.txt         // 从版本库中移除，删除文件
git rm file.txt -cached // 从版本库中移除，不删除原始文件
git rm -r xxx           // 从版本库中删除指定文件夹
```

- 从本地仓库中添加新的文件

```
git add .               // 添加所有文件
git add file.txt        // 添加指定文件
```

- 提交，把缓存内容提交到 HEAD 里

```
git commit -m "注释"
```

- 撤销

```
// 撤销最近的一个提交.
git revert HEAD

// 取消 commit + add
git reset --mixed

// 取消 commit
git reset --soft

// 取消 commit + add + local working
git reset --hard
```

- 把本地提交 push 到远程服务器

```
git push [remote-name] [loca-branch]:[remote-branch]
例：git push origin master:master
```

- 查看状态

```
git status
```

- 从远程库中下载新的改动

```
git fetch [remote-name]/[branch]
```

- 合并下载的改动到分支

```
git merge [remote-name]/[branch]
```

- 从远程库中下载新的改动

```
pull = fetch + merge

git pull [remote-name] [branch]
例：git pull origin master
```

- 分支

```
// 列出分支
git branch

// 创建一个新的分支
git branch (branch-name)

// 删除一个分支
git branch -d (branch-nam)

// 删除 remote 的分支
git push (remote-name) :(remote-branch)
```

- 切换分支

```
// 切换到一个分支
git checkout [branch-name]

// 创建并切换到该分支
git checkout -b [branch-name]
```

与github建立ssh通信，让Git操作免去输入密码的繁琐。

- 首先呢，我们先建立ssh密匙。

> ssh key must begin with 'ssh-ed25519', 'ssh-rsa', 'ssh-dss', 'ecdsa-sha2-nistp256', 'ecdsa-sha2-nistp384', or 'ecdsa-sha2-nistp521'. -- from github

根据以上文段我们可以知道github所支持的ssh密匙类型，这里我们创建ssh-rsa密匙。
在command line 中输入以下指令: `ssh-keygen -t rsa` 去创建一个ssh-rsa密匙。如果你并不需要为你的密匙创建密码和修改名字，那么就一路回车就OK，如果你需要，请您自行Google翻译，因为只是英文问题。

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
//您可以根据括号中的路径来判断你的.ssh文件放在了什么地方
Enter file in which to save the key (/c/Users/Liang Guan Quan/.ssh/id_rsa):
```

- 到 https://github.com/settings/keys 这个地址中去添加一个新的SSH key，然后把你的xx.pub文件下的内容文本都复制到Key文本域中，然后就可以提交了。
- 添加完成之后 我们用 `ssh git@github.com` 命令来连通一下github，如果你在response里面看到了你github账号名，那么就说明配置成功了。 *let's enjoy github ;)*

gitignore

在本地仓库根目录创建 .gitignore 文件。Win7 下不能直接创建，可以创建 ".gitignore." 文件，后面的标点自动被忽略；

```
/.idea          // 过滤指定文件夹
/fd/*           // 忽略根目录下的 /fd/ 目录的全部内容；
*.iml           // 过滤指定的所有文件
!.gitignore     // 不忽略该文件
```