/////5{sign="<>",name="剩余的写在前面的"}////////////////////////////////////////

总结  线程 剩余的点

编译器常量

访问一个空对象的，静态常量，和编译器常亮的理解 4

ReentrantLock和synchronized持有的对象监视器不同

对象是放在堆中的，栈中存放的是对象的引用（地址）。由此可见'=='是对栈中的值进行比较的。如果要比较堆中对象的内容是否相同，那么就要重写equals方法了

（对段落底色hJ）

(对象是放在ls方法)

《对象是放在ls方法》

<对象是放在ls方法>

[Java Study](#_/////5{sign=\"<>\",name=\"Java的一些知识\"} ////////////////////////////////////////)

[算法 Study](#_/////5{sign=\"<>\",name=\"算法的一些知识\"} ////////////////////////////////////////)

[Android Study](#_/////5{sign=\"<>\",name=\"Android的一些知识\"} ////////////////////////////////////////)

adb push /Users/wjw/Documents/work\_here/StudyExample/a\_Doc/a\_\_base\_beed/String字符串常量池.docx /sdcard/android\_wjw\_study



#### /////5{sign="<>",name="算法的一些知识"} ///////////////////////////////////////

#### /////5{sign="<>",name="冒泡排序"} ////////////////////////////////////////

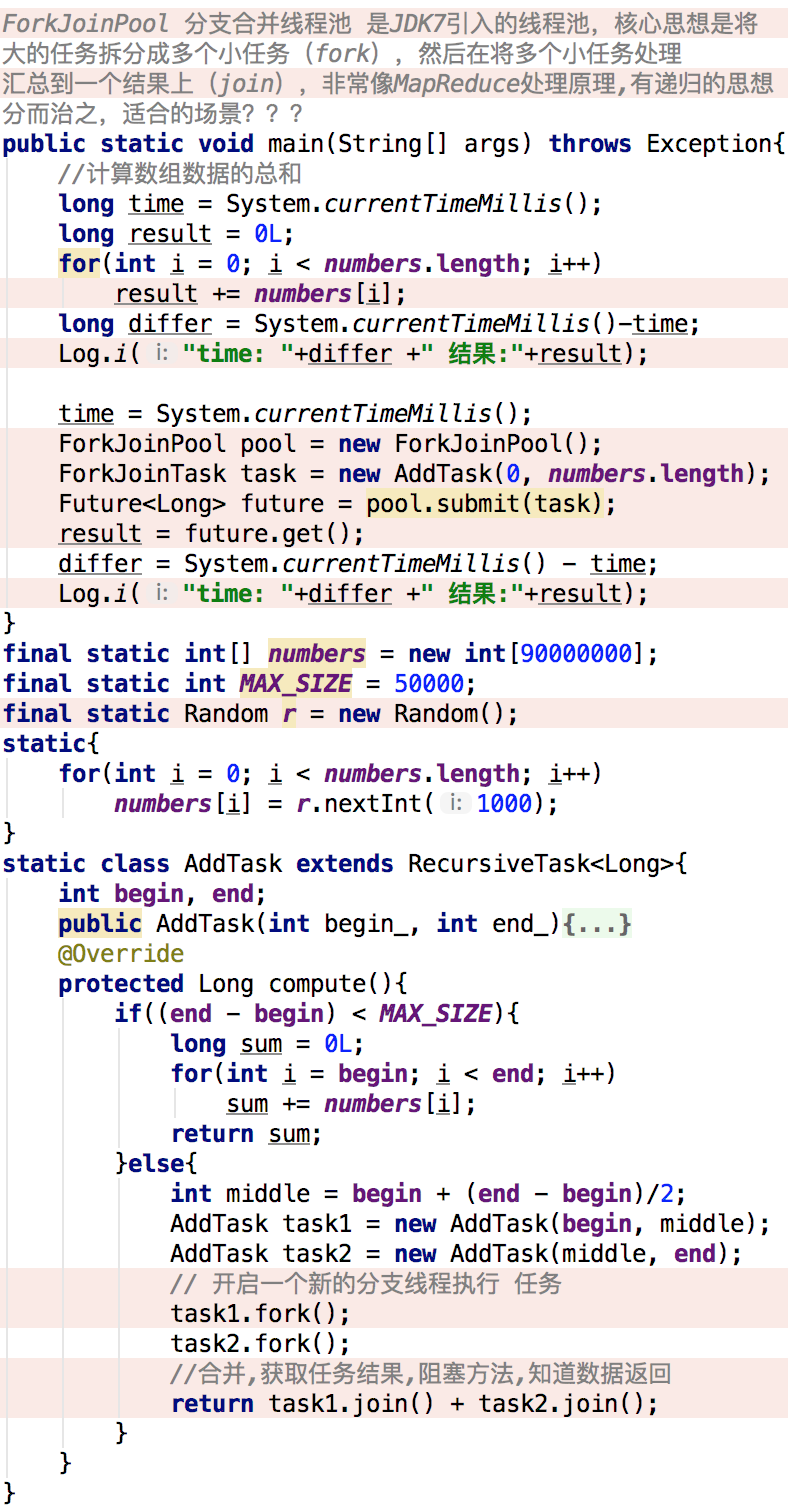
#### /////5{sign="<>",name="插入排序"} ////////////////////////////////////////

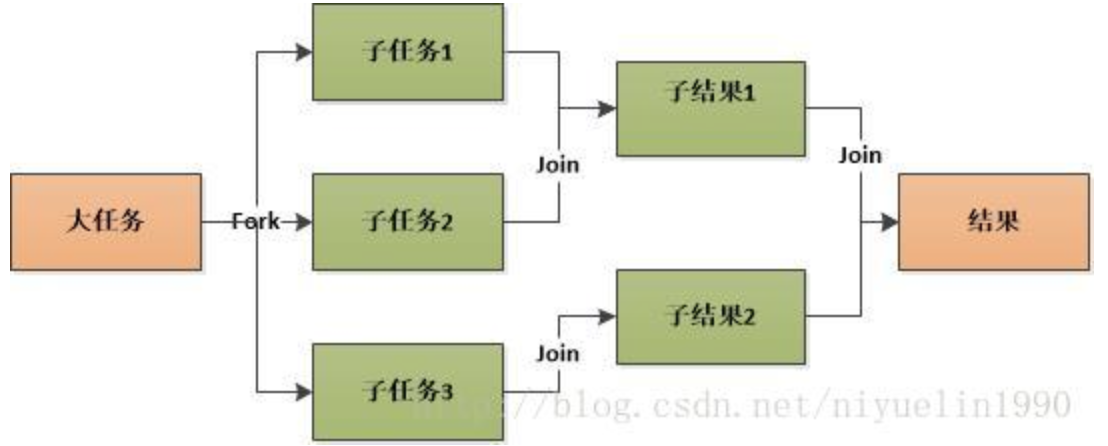
/////5{sign="<>",name="shellSort希尔排序"} ////////////////////////////////////////

/////5{sign="<>",name="mergeSort并归排序"} ////////////////////////////////////////

/////5{sign="<>",name="quickSort"} ////////////////////////////////////////

/////5{sign="<>",name="ForkJoinPool 分支合并线程池"} ////////////////////////////////////////





<通常情况下我们不需要直接继承ForkJoinTask类，而只需要继承它的子类，Fork/Join框架提供了以下两个子类：

RecursiveAction：用于没有返回结果的任务。

RecursiveTask ：用于有返回结果的任务。>

getQueuedSubmissionCount—获取所有待执行的任务数；

getRunningThreadCount—获取正在运行的任务数。

Java 8为ForkJoinPool添加了一个通用线程池，这个线程池用来处理那些没有被显式提交到任何线程池的任务。它是ForkJoinPool类型上的一个静态元素，它拥有的默认线程数量等于运行计算机上的处理器数量。当调用Arrays类上添加的新方法时，自动并行化就会发生。比如用来排序一个数组的并行快速排序，用来对一个数组中的元素进行并行遍历。自动并行化也被运用在Java 8新添加的Stream API中。

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

numbers.parallelStream().forEach(out::println);

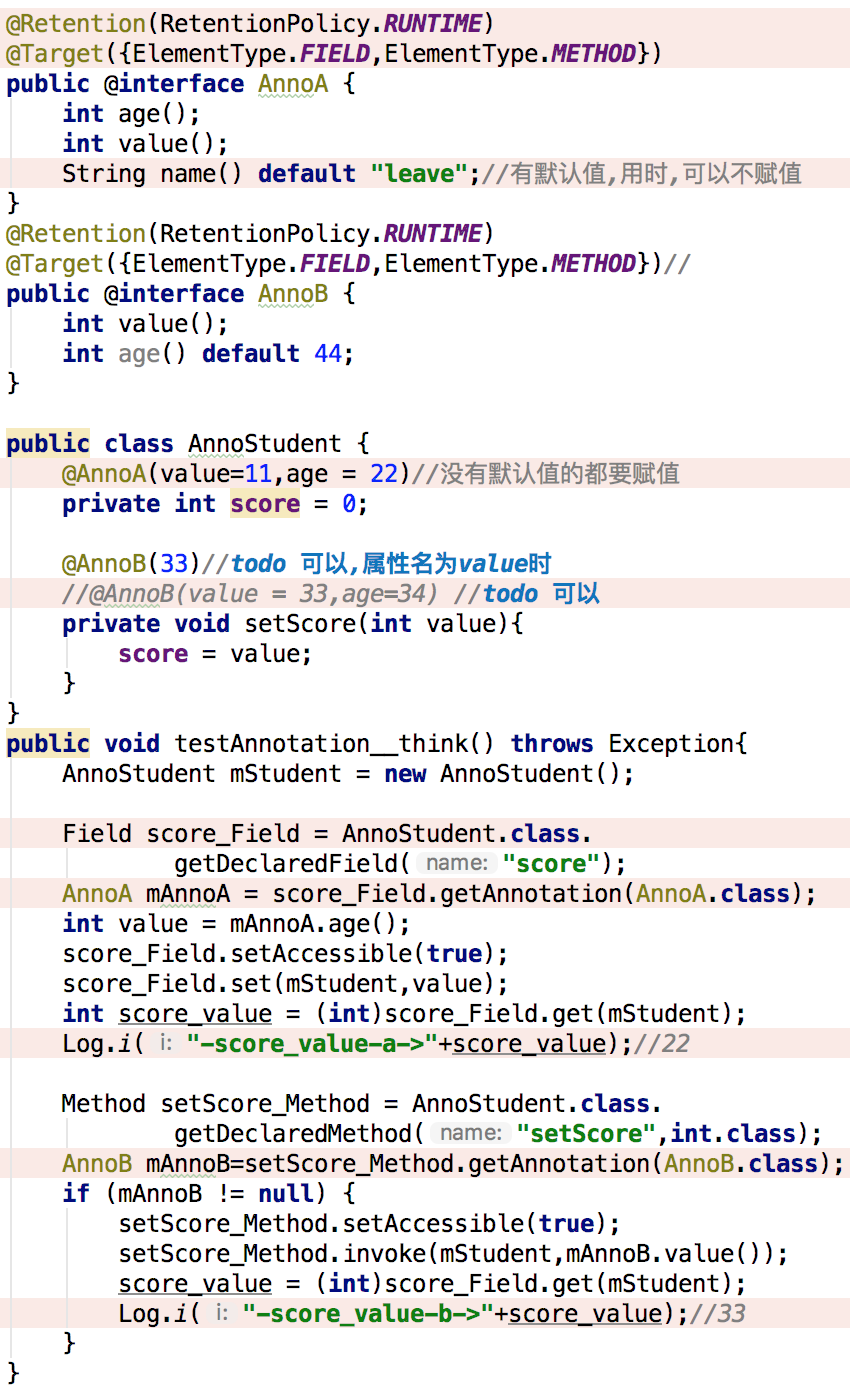
/////5{sign="<>",name="AsyncTask<Params, Progress, Result>"} ////////////////////////////////////////



/////5{sign="<>",name="自定义注解类的声明"} ////////////////////////////////////////



/////5{sign="<>",name="自定义注解的使用"} ////////////////////////////////////////



/////5{sign="<>",name="剩下的还未总结的"} ////////////////////////////////////////

类的继承，方法和变量的(覆盖、隐藏)，类的初始化顺序

JAVA中方法和变量在继承时的覆盖和隐藏规则

1.父类的实例变量和静态变量能被子类的同名变量隐藏

2.父类的静态方法被子类的同名静态方法隐藏(在可访问时)

3.父类的实例方法被子类的同名实例变量覆盖(在可访问时)

a.不能用子类的(静态方法、实例方法)隐藏父类中同样标志（也就是返回值 名字 参数都一样）的(实例方法、静态方法)(在可访问时)

变量只会被隐藏，不会被覆盖 ，无论他是实例变量还是静态变量，而且，

子类的静态变量可以隐藏父类的实例变量，子类的实例变量可以隐藏父类的静态变量

隐藏 和覆盖 的区别在于，子类对象转换成父类对象后，能够访问父类被隐藏的变量和方法，而不能访问父类被覆盖的方法

如果需要访问父类被隐藏的实例变量，加上super就好了，比如访问父类的name，写上super.name就好了

<方法重载Overloading,是在一个类中，有多个方法，方法名相同，但是参数类型和参数个数不同，返回值类型可以相同，也可以不同，不能以返回值类型判断方法是否重载。

方法重写覆盖Overriding,是存在于父类与子类之间

a.前提，在可访问时，父类的方法，在子类可以被访问

b.若子类中的方法与父类中的某一方法具有相同的同样”标志“，如需调用父类方法用super关键字

c.子类的重写方法的权限修饰符不能小于父类的，要大于等于父类>

java反射无法实现用一个子类实例调用父类方法又,不调用子类被重写的方法

后半句是不严谨的，可以调用子类的一个方法，这个方法里面调用super.父类方法

类的变量的声明，使用顺序

成员变量的赋值可以在声明之前 ，成员变量的使用需要在声明之后(属于编译器报错),（编译时会对变量的赋值顺序进行优化调整）

静态变量存放在常量池之中。

静态变量的声明与初始化是两个不同的操作；

静态变量的声明在编译时已经明确了内存的位置。

<java中子类继承父类和实现接口有同名方法怎么处理

子类直接继承父类和实现接口不实现接口方法不报错，说明这种情况下默认父类实现该方法

https://blog.csdn.net/s3395719/article/details/78140088

子类和父类实现同一个接口的意义

最后得出的结论是：效果上没什么特别用途，仅仅是为了让人阅读源码时知道子类实现了该核心接口。就像很多人都知道 ArrayList实现了List接口就够了，而不需要知道它继承AbstractList的相关细节。>

/////5{sign="<>",name="编译期常量"} ////////////////////////////////////////

java中的编译期常量包括java八大基本类型和直接声明的string类型。

什么是直接声明的string类型呢？就是没有通过new关键字声明的字符串（String str = "abc";）。

编译期常量用（public）static final 修饰,编译期常量必须要在声明时进行初始化

进行常量折叠需要满足：

1. 编译时常量（数字字面值，字符串字面值等）。

2. 编译时常量进行简单运算的结果也是编译时常量，如1 + 2，”a”+ “b”。

3. 被 final 修饰的基本类型和字符串变量也是编译时常量。

final类型变量的赋值要在构造函数结束之前，

static final类型变量的赋值要在所有静态代码块结束之前，

使用最终变量的默认值只允许通过方法,

/\*short s1 =1;

s1 = s1 + 1;\*/

short s2 =1;

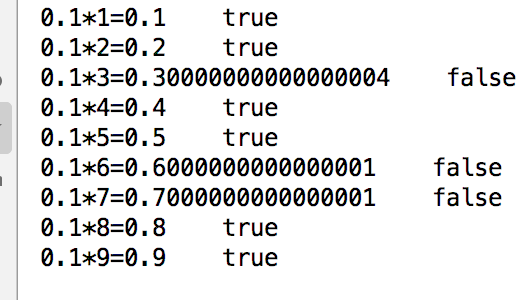
s2 += 1;

/////5{sign="<>",name="浮点数的==的慎用"} ////////////////////////////////////////

因为浮点数不能完全精确的表示出来，一般都会损失精度。

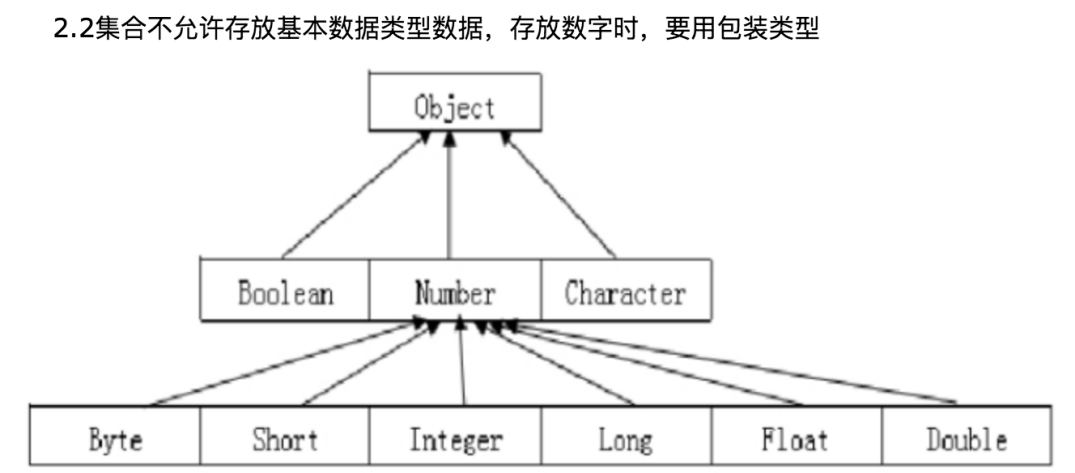
和浮点数在电脑上的存储方式有关,用“==”要确认

这几个记下来，注意此类问题



/////5{sign="<>",name="包装类型装箱拆箱"} ////////////////////////////////////////

Java 包装类型装箱拆箱基础



自身范围缓存

boolean

char[0,127]ASCII的范围

byte[-128,127]

范围为256的缓存[-128,127]

short

int (可以通过参数配置上限)

long

不被缓存

float

Doubles

byte：8位，最大存储数据量是255，存放的数据范围是-128~127之间。

short：16位，最大数据存储量是65536，数据范围是-32768~32767之间。

int：32位，最大数据存储容量是2的32次方减1，数据范围是负的2的31次方到正的2的31次方减1。

long：64位，最大数据存储容量是2的64次方减1，数据范围为负的2的63次方到正的2的63次方减1。

boolean：只有true和false两个取值。

char：16位，存储Unicode码，用单引号赋值。

float：32位，数据范围在3.4e-45~1.4e38，直接赋值时必须在数字后加上f或F。

double：64位，数据范围在4.9e-324~1.8e308，赋值时可以加d或D也可以不加。

java 1.5 开始的自动装箱拆箱机制其实是编译器自动完成的替换，

装箱阶段自动替换为了 valueOf 方法，拆箱阶段自动替换为了 xxxValue 方法。

对于 Integer 类型的 valueOf 方法参数如果是 -128~127 之间的值会直接返回内部缓存池中已经存在对象的引用，

参数是其他范围值则返回新建对象；而 Double 类型与 Integer 类型一样会调用到 Double 的 valueOf 方法，

但是 Double 的区别在于不管传入的参数值是多少都会 new 一个对象来表达该数值（因为在指定范围内浮点型数据个数是不确定的，

整型等个数是确定的，所以可以 Cache）。 注意：Integer、Short、Byte、Character、Long 的 valueOf 方法实现类似，

而 Double 和 Float 比较特殊，每次返回新包装对象。

对于两边都是包装类型的比较 == 比较的是引用，equals 比较的是值，

对于两边有一边是表达式（包含算数运算）则 == 比较的是数值（自动触发拆箱过程），

对于包装类型 equals 方法不会进行类型转换,是不同类型的包装类直接返回false。

用Short的equals()方法与short进行比较的时候，short类型会被判断为是Short类型的实例，然后两个对象都会被转化为基本类型用==进行比较，所以结果为true。

用==比较Short和short的时候，Short类型对象被拆箱（转为short基本类型），所以结果为true。

用Short的equals()方法与int进行比较的时候，由于类型判断那里就已经为false了，直接返回false。

用Short的equals()方法与Integer进行比较的时候，与用Short的equals()方法与int进行比较的时候同样的原因，返回结果为false。

用==比较Short和int的时候，Short首先是进行了拆箱（转为short基本类型），然后是自动提升类型（转为int），之后才进行比较，所以结果为true。

/////5{sign="<>",name="String，常量池的，比较"} ////////////////////////////////////////

String，常量池的，比较

关于String.intern()

intern方法使用：一个初始为空的字符串池，它由类String独自维护。当调用 intern方法时，

如果池已经包含一个等于此String对象的字符串（用equals(oject)方法确定），则返回池中的字符串。

否则，将此String对象添加到池中，并返回此String对象的引用。

它遵循以下规则：对于任意两个字符串 s 和 t，当且仅当 s.equals(t) 为 true 时，s.intern() == t.intern() 才为 true。

String.intern();

再补充介绍一点：存在于.class文件中的常量池，在运行期间被jvm装载，并且可以扩充。String的intern()方法

就是扩充常量池的一个方法；当一个String实例str调用intern()方法时，java查找常量池中是否有

相同unicode的字符串常量，如果有，则返回其引用，如果没有，则在常量池中增加一个unicode等于str的字符串并返回它的引用。

1）单独使用""引号创建的字符串都是常量,编译期就已经确定存储到String Pool中；

（2）使用new String("")创建的对象会存储到heap中,是运行期新创建的；

new创建字符串时首先查看池中是否有相同值的字符串，如果有，则拷贝一份到堆中，然后返回堆中的地址；

如果池中没有，则在堆中创建一份，然后返回堆中的地址（注意，此时不需要从堆中复制到池中，否则，

将使得堆中的字符串永远是池中的子集，导致浪费池的空间）！

（3）使用只包含常量的字符串连接符如"aa" + "aa"创建的也是常量,编译期就能确定,已经确定存储到String Pool中；

（4）使用包含变量的字符串连接符如"aa" + s1创建的对象是运行期才创建的,存储在heap中；

对于final修饰的变量，它在编译时被解析为常量值的一个本地拷贝存储到自己的常量池中或嵌入到它的字节码流中

4.使用String不一定创建对象

在执行到双引号包含字符串的语句时，如String a = "123"，JVM会先到常量池里查找，如果有的话返回常量池里的这个实例的引用，

否则的话创建一个新实例并置入常量池里。所以，当我们在使用诸如String str = "abc"；的格式定义对象时，总是想当然地认为，

创建了String类的对象str。担心陷阱！对象可能并没有被创建！而可能只是指向一个先前已经创建的对象。

只有通过new()方法才能保证每次都创建一个新的对象。

5.使用new String，一定创建对象

可变与不可变：String是不可变字符串对象，StringBuilder和StringBuffer是可变字符串对象（其内部的字符数组长度可变）。

（2）是否多线程安全：String中的对象是不可变的，也就可以理解为常量，显然线程安全。StringBuffer 与 StringBuilder

中的方法和功能完全是等价的，只是StringBuffer 中的方法大都采用了synchronized 关键字进行修饰，因此是线程安全的，

而 StringBuilder 没有这个修饰，可以被认为是非线程安全的。

很显然，new只调用了一次，也就是说只创建了一个对象。而这道题目让人混淆的地方就是这里，这段代码在运行期间确实只创建了一个对象，

即在堆上创建了"abc"对象。而为什么大家都在说是2个对象呢，这里面要澄清一个概念，该段代码执行过程和类的加载过程是有区别的。

在类加载的过程中，确实在运行时常量池中创建了一个"abc"对象，而在代码执行过程中确实只创建了一个String对象。

因此，这个问题如果换成 String str = new String("abc")涉及到几个String对象？合理的解释是2个。

个人觉得在面试的时候如果遇到这个问题，可以向面试官询问清楚”是这段代码执行过程中创建了多少个对象还是涉及到多少个对象“再根据具体的来进行回答。

首先 count++ 是一个有返回值的表达式，返回值是 count 自加前的值，

Java 对自加处理的流程是先把 count 的值（不是引用）拷贝到一个临时变量区，

然后对 count 变量加1，接着返回临时变量区的值。

i++,返回使用之前的值，++i返回使用之后的值

public static void test\_01() {

    int count = 0 , num = 0;

    for (int i = 0; i < 100; i++) {

        count = ++count;

        num = num++;//

    }

    System.out.println("count:"+count+",  num:"+num);

}

默认情况（没有覆盖equals方法）下equals方法都是调用Object类的equals方法，

而Object的equals方法主要用于判断对象的内存地址引用是不是同一个地址（是不是同一个对象）。

一些基类有重写，如String

Object类提供的默认实现确实保证每个对象的hash码不同（在对象的内存地址基础上经过特定算法返回一个hash码）

hashCode() 的作用是获取“哈希码”，也称为“散列码”；它实际上是返回一个int整数。这个哈希码的作用是确定该对象在“哈希表”中的“索引位置”

hashCode() 在散列表中才有用，在其它情况下没用。在散列表中hashCode() 的作用是获取对象的散列码，

在对对象进行散列的时候作为key输入，进而确定该对象在散列表中的位置。

—这里所说的“不会创建类对应的散列表”是说：我们不会在HashSet, Hashtable, HashMap等等这些本质是散列表的数据结构中，用到该类。

1. 第一种 不会创建“类对应的散列表”

在这种情况下，类的“hashCode() 和 equals() ”没有半毛钱关系的！equals() 用来比较该类的两个对象是否相等。

而hashCode() 则根本没有任何作用，所以，不用理会hashCode()。

2. 第二种 会创建“类对应的散列表”

“在标准环境下”

也就是说对于两个对象，

如果调用equals方法得到的结果为true，则两个对象的hashcode值必定相等；

如果equals方法得到的结果为false，则两个对象的hashcode值不一定不同；

如果两个对象的hashcode值不等，则equals方法得到的结果必定为false；

如果两个对象的hashcode值相等，则equals方法得到的结果未知。

补充说一句：“两个不同的键值对，哈希值相等”，这就是哈希冲突。

hashCode() 的作用是为了提高在散列结构存储中查找的效率，在线性表中没有作用。

当 equals 方法被重写时通常有必要重写 hashCode 方法来维护 hashCode 方法的常规协定。

instanceof 是比较运算符。instanceof 后面的类名在编译时就已知且固定了，即 obj instanceof ClassA，ClassA 必须是已经存在的类名，不然编译都过不了。

isInstance() 是动态比较的方法。isInstance() 的左边可以在运行时决定，即可以这样 objA.getClass().isInstance(objB)，

objA 可以作为某个方法的参数被传进来，这样可以动态的看两个对象是否类型兼容。

System.out.println(Base.class instanceof Object);Object是个特殊吧？

调用 getClass().getName() 方法的实质就是在调用从父类继承的 getClass() 方法，

等效于调用 super.getClass().getName() 方法；然而 getClass() 方法来自 Object 类且其返回对象在运行时的类型，

在 java 中重写方法抛出的异常不能是原方法抛出异常的父类，

这里 func 方法在父类中抛出了 IOException，所有在子类中的

func 方法只能抛出 IOExcepition 或是其子类，但不能是其父类。

避免在 finally 中使用 return 语句或者抛出异常，如果调用的其他代码可能抛出异常则应该捕获异常并进行处理，

因为 finally 中 return 不仅会覆盖 try 和 catch 内的返回值且还会掩盖 try 和 catch 内的异常，

就像异常没有发生一样（特别注意，当 try-finally 中没有 return 时该方法运行会继续抛出异常）。

尽量不要在 catch 块中压制异常（即什么也不处理直接 return），

因为这样以后无论抛出什么异常都会被忽略，以至没有留下任何问题线索，

如果在这一层不知道如何处理异常最好将异常重新抛出由上层决定如何处理异常。

Throwable 是所有异常的父类，它有两个直接子类 Error 和 Exception，

Error 表示系统错误，通常不能预期和恢复（譬如 JVM 崩溃、内存不足等）；

Exception 又被继续划分为被检查的异常和运行时的异常；

被检查的异常在程序中能预期且要尝试修复,如我们必须捕获 FileNotFoundException 异常并为用户提供有用信息和合适日志来进行调试，

Exception 是所有被检查的异常的父类）；

运行时异常又称为不受检查异常，譬如我们检索数组元素之前必须确认数组的长度，

否则就可能会抛出 ArrayIndexOutOfBoundException 运行时异常，

RuntimeException 是所有运行时异常的父类。

避免在 finally 中使用 return 语句或者抛出异常，如果调用的其他代码可能抛出异常则应该捕获异常并进行处理，

因为 finally 中 return 不仅会覆盖 try 和 catch 内的返回值且还会掩盖 try 和 catch 内的异常，

就像异常没有发生一样（特别注意，当 try-finally 中没有 return 时该方法运行会继续抛出异常）。

尽量不要在 catch 块中压制异常（即什么也不处理直接 return），

因为这样以后无论抛出什么异常都会被忽略，以至没有留下任何问题线索，

如果在这一层不知道如何处理异常最好将异常重新抛出由上层决定如何处理异常。

Java中提供了三个特殊的浮点数值：正无穷大、负无穷大、非数，用于表示溢出和出错。

正无穷大：用一个正数除以0将得到一个正无穷大，通过Double或Float的POSITIVE\_INFINITY表示。

负无穷大：用一个负数除以0将得到一个负无穷大，通过Double或Float的NEGATIVE\_INFINITY表示。

非数：0.0除以0.0或对一个负数开放将得到一个非数，通过Double或Float的NaN表示。

所有的正无穷大的数值都是相等的，所有的负无穷大的数值都是相等；而NaN不与任何数值相等，甚至和NaN都不等。

问：谈谈 Java 中 final、finally、finalize 的区别？

答：这道题其实没有任何意义，无非就是考察开发者有没有区分这几个关键字的含义，仅仅关联是因为长得像而已。

final 是一个修饰符，如果一个类被声明为 final 则其不能再派生出新的子类，所以一个类不能既被声明为 abstract 又被声明为 final 的；将变量或方法声明为 final 可以保证它们在使用中不被改变（对于对象变量来说其引用不可变，即不能再指向其他的对象，但是对象的值可变），被声明为 final 的变量必须在声明时给定初值，而在以后的引用中只能读取不可修改，被声明为 final 的方法也同样只能使用不能重载。使用 final 关键字如果编译器能够在编译阶段确定某变量的值则编译器就会把该变量当做编译期常量来使用，如果需要在运行时确定（譬如方法调用）则编译器就不会优化相关代码；将类、方法、变量声明为 final 能够提高性能，这样 JVM 就有机会进行估计并进行优化；接口中的变量都是 public static final 的。

finally 用来在异常处理时提供块来执行任何清除操作，如果抛出一个异常，则相匹配的 catch 子句就会执行，然后控制就会进入 finally 块。

finalize 是一个方法名，Java 允许使用 finalize() 方法在垃圾收集器将对象从内存中清除出去之前做必要的清理工作，这个方法是由垃圾收集器在确定这个对象没有被引用时对这个对象调用的，它是在 Object 类中定义的，因此所有的类都继承了它，子类覆盖 finalize() 方法以整理系统资源或者执行其他清理工作，finalize() 方法在垃圾收集器删除对象之前对这个对象调用的。

由于java有自己的垃圾回收机制，所以在写代码的时候千万不能依赖于它的一些方法来回收资源（比如finalize，gc），因为这些方法的调用时机都是不可预知的。

/////////////////////////////////////////////////////////////

问：java 中 static、final、static final 的区别是什么？

答：final 可以修饰属性、方法、类、局部变量（方法中的变量），修饰属性的初始化可以在编译期，也可以在运行期，初始化后不能被改变；修饰的属性表明是一个常数；修饰方法表示方法不能在子类中被重写；修饰类表示类不能被继承。

static 可以修饰属性、方法、代码段、内部类（静态内部类或嵌套内部类），修饰属性的初始化在编译期（类加载的时候），初始化后可以被修改值；修饰的属性、方法、代码段跟该类的具体对象无关，不创建对象也能调用 static 修饰的属性、方法等；static 不可以修饰局部变量。

static final（或者 final static）是组合修饰，static 修饰的属性强调它们只有一个，final 修饰的属性表明是一个常数（创建后不能被修改），static final 修饰的属性表示一旦给值就不可修改并且可以通过类名访问，static final 也可以修饰方法，表示该方法不能重写，可以在不 new 对象的情况下调用。

/////////////////////////////////////////////////////////////

问：下面程序的有问题吗，结果是什么？

class Test {

   public static String foo(){

       System.out.println("foo called.");

       return "return called.";

   }

}

public class Demo {

   public static void main(String[] args) {

       Test obj = null;

       System.out.println(obj.foo());

   }

}

答：没有问题，运行结果如下：

foo called.

return called.

因为 jvm 内存里有栈区、堆区，栈区主要用来存放基础类型数据和局部变量，堆区主要存放 new 出来的对象，在堆区又有一个叫做方法区的内存区域用来存放常量、static 变量和 static 方法、还有类的信息，static 的变量和方法不依赖对象，即使对象没有创建，在类加载的时候已经存在信息了（Test 在声明时就被加载了），jvm 识别出是 static 方法就直接调用了在方法区内存里的方法，没有报空指针异常。

/////////////////////////////////////////////////////////////

问：下面程序的运行结果是什么？为什么？

public class Test {

   public static void main(String[] args) {

       String a = "hello2";

       final String b = "hello";

       String c = b + 2;

       String d = "hello";

       String e = d + 2;

       System.out.println((a == c));

       System.out.println((a == e));

   }

}

答：运行结果如下：

true

false

因为当 final 变量是基本数据类型以及 String 类型时如果在编译期间能知道它的确切值则编译器会把它当做编译期常量使用，也就是说在用到该 final 变量的地方相当于直接访问了这个常量，不需要在运行时确定，所以上面代码中由于变量 b 被 final 修饰从而被当做编译器常量，故在使用到 b 的地方会直接将变量 b 替换为它的值，而对于变量 d 的访问却需要在运行时通过链接来进行。

/////////////////////////////////////////////////////////////

Java String类实例化方式以及入池操作

1. 实例化方式

String有两种实例化方式，一种是直接赋值，另一种是使用String类的构造方法进行实例化。

直接赋值

String str = "Hello World" ;

1

 在任何的语言的底层，都不会提供有直接的字符串类型。字符串类型只是高级语言提供给用户方便开发的支持而已。

 在java之中，本身也没有直接提供字符串类型的概念，java本身提供的只有八大基本数据类型，而所有使用双引号 ” “定义的字符串本质上来讲都是String的匿名对象。

 所以，String通过直接赋值所创建的字符串就是在堆上的一个String匿名对象。

构造方法构造：

String str = new String("Hello World");

虽然使用构造方法创建对象是最标准的创建方式，但是字符串创建一般推荐使用第一种方式。

2. 两种实例化的区别

 String类的设计使用了共享设计模式。在JVM底层实际上会自动维护一个对象池（字符串对象池），如果现在采用了直接赋值的方式进行String类的对象实例化操作，那么该实例化对象（字符串内容）将自动保存到这个对象池之中。如果下次继续使用直接赋值的模式声明String类对象，此时对象池之中如若有指定内容，将直接进行引用；如若没有，则开辟新的字符串对象而后将其保存在对象池之中以供下次使用。所谓的对象池就是一个对象数组（目的就是减少开销）。

 == 操作符两端为数字时，比较的是它们的值，如果是对象，比较的是它们的地址。那么用其进行String直接赋值的验证，如下：

String str1 = "hello" ;

String str2 = "hello" ;

String str3 = "hello" ;

System.out.println(str1 == str2); // true

System.out.println(str1 == str3); // true

System.out.println(str2 == str3); // true

另外，new出来的字符串是不会入池的，但是我们可以采用String提供的实例方法intern()将字符串进行入池操作。

综上，可以看出两种方式的区别：

方式 特点

直接赋值 只会开辟一块堆内存空间，并且该字符串对象可以自动保存在对象池中以供下次使用。

构造方法 会开辟两块堆内存空间，其中一块成为垃圾空间，不会自动保存在对象池中，可以使用intern()方法手工入池。

推荐使用第一种，效率高而且语法简单。

/////////////////////////////////////////////////////////////

”==”本身是进行数值比较的，如果现在用于对象比较，那么所比较的就应该是两个对象所保存的内存地址数值比

较，而并没有比较对象的内容

当然，如果都使用String直接赋值的方式进行字符串的创建，如上面str1的创建，由于入池操作，内容相同的字符串实际上使用一片空间，所以这时可以进行字符串内容的比较。

那么要想比较内容比较，应该采用String类提供的equals方法。

/////////////////////////////////////////////////////////////

String hel2="hel";

StringBuilder helloBuilder = new StringBuilder("hel");

System.out.println(helloBuilder.equals(hel2)); --false

System.out.println(hel2.equals(helloBuilder.toString())); --true

System.out.println(helloBuilder.toString().equals(hel2)); --true

最后一个StringBuilder的测试我们发现虽然使用equals来判断，但是返回的是false，这是为什么呢？

首先，当我们使用StringBuilder创建对象时，肯定会在内存中开辟一个新的专属的地址用于存放对象内容，但是即使StringBuilder中存放的内容与其他字符串的内容相同，使用equals来判断也是返回false，这是因为StringBuilder并没有重写equals函数，即StringBuilder的equals为：

/////////////////////////////////////////////////////////////

首先得从赋值讲起：你在进行==或者equals判断的时候,肯定是先赋值对吧，string s=null;(这里是建立一个s的对象，但是这个对象未指向任何引用)string s="";(这里是建立一个s的对象，但是这个对象指向（“”）一个空的字符串的引用的)如果string s=null;之后再进行判断:1.s==null这个返回true， 2.s==""这个返回false ， 3.s.equals("")这里会抛出异常，因为equals是对s这个对象指向的引用做的判断，而s赋值时是没有指向任何引用的。如果string s=“”;之后再进行判断:1.s==null这个返回false， 2.s==""这个返回true，3.string.equals("")这里true

/////////////////////////////////////////////////////////////

java中的数据类型，可分为两类：

1.基本数据类型，也称原始数据类型。byte,short,char,int,long,float,double,boolean

他们之间的比较，应用双等号（==）,比较的是他们的值。

2.复合数据类型(类)

当他们用（==）进行比较的时候，比较的是他们在内存中的存放地址，所以，除非是同一个new出来的对象，他们的比较后的结果为true，否则比较后结果为false。 JAVA当中所有的类都是继承于Object这个基类的，在Object中的基类中定义了一个equals的方法，这个方法的初始行为是比较对象的内存地 址，但在一些类库当中这个方法被覆盖掉了，如String,Integer,Date在这些类当中equals有其自身的实现，而不再是比较类在堆内存中的存放地址了。

对于复合数据类型之间进行equals比较，在没有覆写equals方法的情况下，他们之间的比较还是基于他们在内存中的存放位置的地址值的，因为Object的equals方法也是用双等号（==）进行比较的，所以比较后的结果跟双等号（==）的结果相同。

/////////////////////////////////////////////////////////////

String c = "xx" + "yy " + a + "zz" + "mm" + b; 实质上的实现过程是： String c = new StringBuilder("xxyy").append(a).append("zz").append("mm").append(b).toString();

由于得出结论：当使用+进行多个字符串连接时，实际上是产生了一个StringBuilder对象和一个String对象。

/////////////////////////////////////////////////////////////

String字符串常量池

JVM为了提高性能和减少内存开销，内部维护了一个字符串常量池，每当创建字符串常量时，JVM首先检查字符串常量池，如果常量池中已经存在，则返回池中的字符串对象引用，否则创建该字符串对象并放入池中。

因此下述结果返回true。

String a = "abc";

String b = "abc";

System.out.print(a == b); //true

但与创建字符串常量方式不同的是，当使用new String(String str)方式等创建字符串对象时，不管字符串常量池中是否有与此相同内容的字符串，都会在堆内存中创建新的字符串对象。

因此，下面代码片段有如下结果。

String a = "Hello";

String b = new String("Hello");

System.out.println(a == b); //false

System.out.println(a.equals(b)); //true

即使字符串内容相同，字符串常量池中的字符串与通过new String(..)等方式创建的字符串对象之间没有直接的关系，但是，可以通过字符串的intern()方法找到此种关联。intern()方法返回字符串对象在字符串常量池中的对象引用，若字符串常量池中尚未有此字符串，则创建一新的字符串常量放置于池中。

于是，很据如上理解，很自然的，可以得到如下结果。

String a = "Hello";

System.out.println(a == a.intern()); //true

String b = new String("corn");

String c = b.intern();

System.out.println(b == c); //false

String d = "corn";

System.out.println(c == d); //true

/////5{sign="<>",name="Fragment、两种Fragment管理器"}///////////

/////5{sign="<>",name="Glide"}///////////







我们就完全不用担心图片内存浪费，甚至是内存溢出的问题。因为Glide从来都不会直接将图片的完整尺寸全部加载到内存中，而是用多少加载多少。Glide会自动判断ImageView的大小，然后只将这么大的图片像素加载到内存当中，帮助我们节省内存开支。

ImageView默认的scaleType是FIT\_CENTER

FitCenter的效果：会将图片按照原始的长宽比充满全屏，

Glide VS Picasso

双胞胎兄弟之间的对比，使用方式相同，但 Glide 之所以胜出，不仅仅是 Google的推荐，更多应该归功于 GIF 的支持。 在没有 Glide 之前，常用的做法就是写了个自定义 view 然后 用一个 media 去播放。有了 Glide 之后几乎对于 GIF 无感知了的， 内部已经支持了的。可以像普通图片那样去加载并且显示出来动图。

安卓图片显示的质量配置主要分为四种:

ARGB\_8888 :32位图,带透明度,每个像素占4个字节

ARGB\_4444 :16位图,带透明度,每个像素占2个字节

RGB\_565 :16位图,不带透明度,每个像素占2个字节

ALPHA\_8 :32位图,只有透明度,不带颜色,每个像素占4个字节

(A代表透明度,RGB代表红绿蓝:即颜色)

Picasso的默认质量是 ARGB\_8888

Glide的默认质量则为 RGB\_565

加载一张4000 \* 2000(一般手机拍摄的都超过这个像素)的图片

Picasso需要占用的内存为: 32MB

4000 \* 2000 \* 4 / 1024 / 1024 = 30 (MB)

Glide需要占用的内存为: 16MB

4000 \* 2000 \* 2 / 1024 / 1024 = 15 (MB)

也就是说只要同时加载几张图片,你的应用就会OOM(内存溢出了),最恐怖的是就算你的ImageView的宽高只有10px,同样会占用那么多内存,这就是为什么需要做图片压缩的原因了

Glide VS fresco

两个都支持 GIF。所以 GIF 这一关pass掉。说到这里不得不提到一个头疼的OOM问题，fresco 之所以很快闯入大家的视线，大概就是因为 Facebook 说他们使用了 native 内存规避掉了 OutOfMemoryError 问题。而且官方还专门写了个demo，把几大流行的开源库都集成进去，为了说明自己的图片加载库加载同样的图片速度更快，内存占用更低。所以 fresco 相比较于 Glide 的（官方）优势就是这两点： 内存以及加载速度。但是我为什么依旧坚持抛弃了 fresco ?

“ In Android 4.x and lower, Fresco puts images in a special region of Android memory. This lets your application run faster - and suffer the dreaded OutOfMemoryError much less often.” 官方的原话是这么说的，所以在高版本上面依旧使用的Java 内存，所以不可避免依旧会占用内存。

提到内存，不得不说到另外一个笑话，fresco 最大只支持图片文件大小为 2M 。记得有一次帮其他团队跟踪问题，看到了 fresco 源码中有一个 最大 size 2M 常量 。于是当场找了一个10M的图片作为测试。 Glide 正常显示， fresco显示黑屏。。。

使用方式上，fresco 推荐的是用他提供的 SimpleDraweeView . 这个方式意味着我们的迁移成本会非常的高，要改布局文件，其次还必须给定大小（或者比例）。 当然他也支持代码来加载图片，比如 DraweeHierarchy，但是写起来还是真心很费劲的，很不友好，改动成本居高。

fresco 更多是native实现。所以需要对NDK有所了解，但个人对NDK不太了解，相比较于 Glide， 同样遇到问题之后，修改源码的成本，Glide 成本更可控。前者可能就不太好下手了的。

Glide 各种 BitmapTransformation，比如圆形，圆角等，更让人喜欢。

这一点就当随意吐槽一下，当然也可以说心疼一下 Facebook。因为在没有 Android studio （gradle构建）的情况下，想必大家都用的是 eclipse 吧。那么就意味着 fresco 得提供 Jar 包. 这一点当时也是把很多人拒之门外了的，可笑的是当 Facebook 费了老大劲的搞出来 jar 包之后，大家早就纷纷转战 gradle 构建工程, 直接 maven 依赖啦。大写的尴尬。

EngineKey key = keyFactory.buildKey(id, signature, width, height, loadProvider.getCacheDecoder(),

loadProvider.getSourceDecoder(), transformation, loadProvider.getEncoder(),

transcoder, loadProvider.getSourceEncoder());

可见，决定缓存Key的条件非常多，即使你用override()方法改变了一下图片的width或者height，也会生成一个完全不同的缓存Key。

1、LruCache算法

LruCache算法，又称为近期最少使用算法。主要算法原理就是把最近所使用的对象的强引用存储在LinkedHashMap上，并且，把最近最少使用的对象在缓存池达到预设值之前从内存中移除。

Q2:简单说一下Glide的三级缓存？

Android图片加载框架最全解析（三），深入探究Glide的缓存机制

【Android - 进阶】之图片三级缓存的原理及实现

Android Glide缓存策略分析

严格来讲，并没有什么Glide的三级缓存，因为Glide的缓存只有两个模块，一个是内存缓存，一个是硬盘缓存。其中内存缓存又分为Lru算法的缓存和弱引用缓存。

Glide缓存机制大致分为三层：Lru算法缓存、弱引用缓存、磁盘缓存。

读取的顺序是：Lru算法缓存、弱引用缓存、磁盘缓存

写入的顺序是：弱引用缓存、Lru算法缓存、磁盘缓存

我们先来看读取：Lru算法缓存、弱引用缓存、磁盘缓存

内存缓存的主要作用是防止应用重复将图片数据读取到内存当中，而硬盘缓存的主要作用是防止应用重复从网络或其他地方重复下载和读取数据

内存缓存读取：先获取LruResourceCache（Lru算法缓存），然后放入activeResources（是一个弱引用的HashMap）来缓存正在使用中的图片，可以保护这些图片不会被LruCache算法回收掉。

内存缓存写入：回调过来的EngineResource先被put到了activeResources当中，也就是在这里写入的缓存。如果图片正在使用中，也就应该放到activeResources弱引用缓存当中。如果图片不再被使用了，首先会将缓存图片从activeResources中移除，然后再将它put到LruResourceCache当中。这样也就实现了正在使用中的图片使用弱引用来进行缓存，不在使用中的图片使用LruCache来进行缓存的功能。

进一步总结：正在使用中的图片使用弱引用来进行缓存，不在使用中的图片使用LruCache来进行缓存的功能。

6、没有布局的Fragment的作用

没有布局文件Fragment实际上是为了保存，当Activity重启时，保存大量数据准备的

请参考博客：Android 屏幕旋转 处理 AsyncTask 和 ProgressDialog 的最佳方案

https://blog.csdn.net/lmj623565791/article/details/37936275

7、使用Fragment创建对话框

这是Google推荐的方式，我也单独写过博客介绍，请参考：Android 官方推荐 : DialogFragment 创建对话框

https://blog.csdn.net/lmj623565791/article/details/37815413

/////5{sign="<>",name="RxAndroid"} ////////////////////////////////////////



<创建被观察者对象Observable

创建观察者Observer

连接观察者和被观察者subscribe

被观察者通过onNext函数给观察者通知结果

被观察者onComplete函数通知观察者执行结束

连接观察者和被观察者我们使用subscribe函数>

通过打印的log我们可以看到观察者函数调用情况,调用subscribe函数去绑定观察者和被观察者时候，  
观察者的onSubscribe函数会被回调表示建立关联。  
接着每当被观察者调用onNext给观察者发送数据时候，观察者的onNext 会收到回调，并且得到所发送的数据。  
当被观察者调用onComplete函数时候,代表着完成，观察者的onComplete回调会被触发，  
并且断开了两者的关联，这时被观察者再发送数据，观察者也不会收到。  
  
关于onComplete和onError唯一并且互斥这一点, 是需要自行在代码中进行控制,  
如果你的代码逻辑中违背了这个规则, 并不一定会导致程序崩溃.  
比如发送多个onComplete是可以正常运行的, 依然是收到第一个onComplete就不再接收了,  
但若是发送多个onError, 则收到第二个onError事件会导致程序会崩溃.当我们写多个onComplete时，不会报错。  
  
除了被观察者能断开关联，观察者也能主动断开连接，  
调用onSubscribe函数中传入的对象Disposable的dispose()函数即可完成断开连接，  
同样关联断开后，被观察者依然会继续发送数据  
  
我们需要先改变Observable发送事件的线程, 让它去子线程中发送事件,  
然后再改变Observer的线程, 让它去主线程接收事件.  
通过RxAndroid内置的线程调度器可以很轻松的做到这一点

subscribeOn() 指定的是Observable发送事件的线程,指定Observable自身在哪个调度器上执行  
observeOn() 指定的是Observer接收事件的线程.指定一个观察者在哪个调度器上观察这个Observable  
多次指定Observable的线程只有第一次指定的有效, 也就是说多次调用subscribeOn() 只有第一次的有效, 其余的会被忽略.  
多次指定Observer的线程是可以的, 也就是说每调用一次observeOn() , Observer的线程就会切换一次  
  
  
Schedulers.io(): I/O 操作（读写文件、数据库、网络请求等），与newThread()差不多，区别在于io() 的内部实现是是用一个无数量上限的线程池，可以重用空闲的线程，因此多数情况下 io() 效率比 newThread() 更高。值得注意的是，在 io() 下，不要进行大量的计算，以免产生不必要的线程；  
Schedulers.newThread(): 开启新线程操作；  
Schedulers.immediate(): 默认指定的线程，也就是当前线程；  
Schedulers.computation():计算所使用的调度器。这个计算指的是 CPU 密集型计算，即不会被 I/O等操作限制性能的操作，例如图形的计算。这个 Scheduler 使用的固定的线程池，大小为 CPU 核数。值得注意的是，不要把 I/O 操作放在 computation() 中，否则 I/O 操作的等待时间会浪费 CPU；  
AndroidSchedulers.mainThread(): Rxndroid 扩展的 Android 主线程；

<事件产生create() , just() , from() 等 ---

事件加工map() , flapMap() , scan() , filter() 等 --

事件消费subscribe() --

事件产生：默认运行在当前线程，可以由 subscribeOn() 自定义线程

事件加工：默认跟事件产生的线程保持一致, 可由 observeOn() 自定义线程

事件消费：默认运行在当前线程，可以有observeOn() 自定义>

三种方式创建的Observable指定IO操作使用区别：  
Observable.create().....subscribeOn(Schedulers.io()) 创建子线程  
Observable.just().....subscribeOn(Schedulers.io())未创建子线程  
Observable.from().....subscribeOn(Schedulers.io())未创建子线程  
  
Map是RxAndroid中最简单的一个变换操作符了,  
它的作用就是对Observable发送的每一个事件应用一个函数,  
使得每一个事件都按照指定的函数去变化。  
Observable –> map变换 –> Observable  
url -> drawable -> bitmap

<Zip通过一个函数将多个Observable发送的事件结合到一起，然后发送这些组合到一起的事件.

它按照严格的顺序应用这个函数。它只发射与发射数据项最少的那个Observable一样多的数据。

take 最多保留的事件数。只影响 onNext

filter 条件过滤，去除不符合某些条件的事件。举个栗子:

fromArray(T[]) / fromArray(Iterable < ? extends T>):将传入的数组或 Iterable 拆分成具体对象后，依次发送出来。

just(T...): 将传入的参数依次发送出来。>

//

end