https://www.cnblogs.com/dolphin0520/p/3811445.html

Java内部类详解

　　说起内部类这个词，想必很多人都不陌生，但是又会觉得不熟悉。原因是平时编写代码时可能用到的场景不多，用得最多的是在有事件监听的情况下，并且即使用到也很少去总结内部类的用法。今天我们就来一探究竟。下面是本文的目录大纲：

　　一.内部类基础

　　二.深入理解内部类

　　三.内部类的使用场景和好处

　　四.常见的与内部类相关的笔试面试题

　　若有不正之处，请多谅解并欢迎批评指正。

　　请尊重作者劳动成果，转载请标明原文链接：

　　http://www.cnblogs.com/dolphin0520/p/3811445.html

**一.内部类基础**

　　在Java中，可以将一个类定义在另一个类里面或者一个方法里面，这样的类称为内部类。广泛意义上的内部类一般来说包括这四种：成员内部类、局部内部类、匿名内部类和静态内部类。下面就先来了解一下这四种内部类的用法。

　　1.成员内部类

　　成员内部类是最普通的内部类，它的定义为位于另一个类的内部，形如下面的形式：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | class Circle {      double radius = 0;        public Circle(double radius) {          this.radius = radius;      }        class Draw {     //内部类          public void drawSahpe() {              System.out.println("drawshape");          }      }  } |

　　这样看起来，类Draw像是类Circle的一个成员，Circle称为外部类。成员内部类可以无条件访问外部类的所有成员属性和成员方法（包括private成员和静态成员）。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | class Circle {      private double radius = 0;      public static int count =1;      public Circle(double radius) {          this.radius = radius;      }        class Draw {     //内部类          public void drawSahpe() {              System.out.println(radius);  //外部类的private成员              System.out.println(count);   //外部类的静态成员          }      }  } |

　　不过要注意的是，当成员内部类拥有和外部类同名的成员变量或者方法时，会发生隐藏现象，即默认情况下访问的是成员内部类的成员。如果要访问外部类的同名成员，需要以下面的形式进行访问：

|  |  |
| --- | --- |
| 1  2 | 外部类.this.成员变量  外部类.this.成员方法 |

　　虽然成员内部类可以**无条件地访问外部类的成员**，而外部类想访问成员内部类的成员却不是这么随心所欲了。在**外部类中如果要访问成员内部类的成员，必须先创建一个成员内部类的对象，再通过指向这个对象的引用来访问**：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class Circle {      private double radius = 0;        public Circle(double radius) {          this.radius = radius;          getDrawInstance().drawSahpe();   //必须先创建成员内部类的对象，再进行访问      }        private Draw getDrawInstance() {          return new Draw();      }        class Draw {     //内部类          public void drawSahpe() {              System.out.println(radius);  //外部类的private成员          }      }  } |

　　成员内部类是依附外部类而存在的，也就是说，如果要创建成员内部类的对象，前提是必须存在一个外部类的对象。创建成员内部类对象的一般方式如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29 | public class Test {      public static void main(String[] args)  {          //第一种方式：          Outter outter = new Outter();          Outter.Inner inner = outter.new Inner();  //必须通过Outter对象来创建            //第二种方式：          Outter.Inner inner1 = outter.getInnerInstance();      }  }    class Outter {      private Inner inner = null;      public Outter() {        }        public Inner getInnerInstance() {          if(inner == null)              inner = new Inner();          return inner;      }        class Inner {          public Inner() {            }      }  } |

　　内部类可以拥有private访问权限、protected访问权限、public访问权限及包访问权限。比如上面的例子，如果成员内部类Inner用private修饰，则只能在外部类的内部访问，如果用public修饰，则任何地方都能访问；如果用protected修饰，则只能在同一个包下或者继承外部类的情况下访问；如果是默认访问权限，则只能在同一个包下访问。这一点和外部类有一点不一样，外部类只能被public和包访问两种权限修饰。我个人是这么理解的，由于成员内部类看起来像是外部类的一个成员，所以可以像类的成员一样拥有多种权限修饰。

　　2.局部内部类

　　局部内部类是定义在一个方法或者一个作用域里面的类，它和成员内部类的区别在于局部内部类的访问仅限于方法内或者该作用域内。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | class People{      public People() {        }  }    class Man{      public Man(){        }        public People getWoman(){          class Woman extends People{   //局部内部类              int age =0;          }          return new Woman();      }  } |

　　注意，局部内部类就像是方法里面的一个局部变量一样，是不能有public、protected、private以及static修饰符的。

　　3.匿名内部类

　　匿名内部类应该是平时我们编写代码时用得最多的，在编写事件监听的代码时使用匿名内部类不但方便，而且使代码更加容易维护。下面这段代码是一段Android事件监听代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | scan\_bt.setOnClickListener(new OnClickListener() {                @Override              public void onClick(View v) {                  // TODO Auto-generated method stub                }          });            history\_bt.setOnClickListener(new OnClickListener() {                @Override              public void onClick(View v) {                  // TODO Auto-generated method stub                }          }); |

　　这段代码为两个按钮设置监听器，这里面就使用了匿名内部类。这段代码中的：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | new OnClickListener() {                @Override              public void onClick(View v) {                  // TODO Auto-generated method stub                }          } |

　　就是匿名内部类的使用。代码中需要给按钮设置监听器对象，使用匿名内部类能够在实现父类或者接口中的方法情况下同时产生一个相应的对象，但是前提是这个父类或者接口必须先存在才能这样使用。当然像下面这种写法也是可以的，跟上面使用匿名内部类达到效果相同。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | private void setListener()  {      scan\_bt.setOnClickListener(new Listener1());      history\_bt.setOnClickListener(new Listener2());  }    class Listener1 implements View.OnClickListener{      @Override      public void onClick(View v) {      // TODO Auto-generated method stub        }  }    class Listener2 implements View.OnClickListener{      @Override      public void onClick(View v) {      // TODO Auto-generated method stub        }  } |

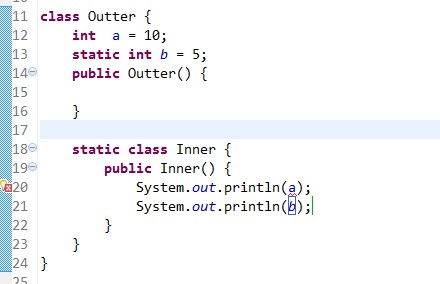
　　这种写法虽然能达到一样的效果，但是既冗长又难以维护，所以一般使用匿名内部类的方法来编写事件监听代码。同样的，匿名内部类也是不能有访问修饰符和static修饰符的。

　　匿名内部类是唯一一种没有构造器的类。正因为其没有构造器，所以匿名内部类的使用范围非常有限，大部分匿名内部类用于接口回调。匿名内部类在编译的时候由系统自动起名为Outter$1.class。一般来说，匿名内部类用于继承其他类或是实现接口，并不需要增加额外的方法，只是对继承方法的实现或是重写。

　　4.静态内部类

　　静态内部类也是定义在另一个类里面的类，只不过在类的前面多了一个关键字static。静态内部类是不需要依赖于外部类的，这点和类的静态成员属性有点类似，并且它不能使用外部类的非static成员变量或者方法，这点很好理解，因为在没有外部类的对象的情况下，可以创建静态内部类的对象，如果允许访问外部类的非static成员就会产生矛盾，因为外部类的非static成员必须依附于具体的对象。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | public class Test {      public static void main(String[] args)  {          Outter.Inner inner = new Outter.Inner();      }  }    class Outter {      public Outter() {        }        static class Inner {          public Inner() {            }      }  } |



**二.深入理解内部类**

　　1.为什么成员内部类可以无条件访问外部类的成员？

　　在此之前，我们已经讨论过了成员内部类可以无条件访问外部类的成员，那具体究竟是如何实现的呢？下面通过反编译字节码文件看看究竟。事实上，编译器在进行编译的时候，会将成员内部类单独编译成一个字节码文件，下面是Outter.java的代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | public class Outter {      private Inner inner = null;      public Outter() {        }        public Inner getInnerInstance() {          if(inner == null)              inner = new Inner();          return inner;      }        protected class Inner {          public Inner() {            }      }  } |

　　编译之后，出现了两个字节码文件：



　　反编译Outter$Inner.class文件得到下面信息：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58 | E:\Workspace\Test\bin\com\cxh\test2>javap -v Outter$Inner  Compiled from "Outter.java"  public class com.cxh.test2.Outter$Inner extends java.lang.Object    SourceFile: "Outter.java"    InnerClass:     #24= #1 of #22; //Inner=class com/cxh/test2/Outter$Inner of class com/cxh/tes  t2/Outter    minor version: 0    major version: 50    Constant pool:  const #1 = class        #2;     //  com/cxh/test2/Outter$Inner  const #2 = Asciz        com/cxh/test2/Outter$Inner;  const #3 = class        #4;     //  java/lang/Object  const #4 = Asciz        java/lang/Object;  const #5 = Asciz        this$0;  const #6 = Asciz        Lcom/cxh/test2/Outter;;  const #7 = Asciz        <init>;  const #8 = Asciz        (Lcom/cxh/test2/Outter;)V;  const #9 = Asciz        Code;  const #10 = Field       #1.#11; //  com/cxh/test2/Outter$Inner.this$0:Lcom/cxh/t  est2/Outter;  const #11 = NameAndType #5:#6;//  this$0:Lcom/cxh/test2/Outter;  const #12 = Method      #3.#13; //  java/lang/Object."<init>":()V  const #13 = NameAndType #7:#14;//  "<init>":()V  const #14 = Asciz       ()V;  const #15 = Asciz       LineNumberTable;  const #16 = Asciz       LocalVariableTable;  const #17 = Asciz       this;  const #18 = Asciz       Lcom/cxh/test2/Outter$Inner;;  const #19 = Asciz       SourceFile;  const #20 = Asciz       Outter.java;  const #21 = Asciz       InnerClasses;  const #22 = class       #23;    //  com/cxh/test2/Outter  const #23 = Asciz       com/cxh/test2/Outter;  const #24 = Asciz       Inner;    {  final com.cxh.test2.Outter this$0;    public com.cxh.test2.Outter$Inner(com.cxh.test2.Outter);    Code:     Stack=2, Locals=2, Args\_size=2     0:   aload\_0     1:   aload\_1     2:   putfield        #10; //Field this$0:Lcom/cxh/test2/Outter;     5:   aload\_0     6:   invokespecial   #12; //Method java/lang/Object."<init>":()V     9:   return    LineNumberTable:     line 16: 0     line 18: 9      LocalVariableTable:     Start  Length  Slot  Name   Signature     0      10      0    this       Lcom/cxh/test2/Outter$Inner;      } |

　　第11行到35行是常量池的内容，下面逐一第38行的内容：

final com.cxh.test2.Outter this$0;

　　这行是一个指向外部类对象的指针，看到这里想必大家豁然开朗了。也就是说编译器会默认为成员内部类添加了一个指向外部类对象的引用，那么这个引用是如何赋初值的呢？下面接着看内部类的构造器：

public com.cxh.test2.Outter$Inner(com.cxh.test2.Outter);

　　从这里可以看出，虽然我们在定义的内部类的构造器是无参构造器，编译器还是会默认添加一个参数，该参数的类型为指向外部类对象的一个引用，所以成员内部类中的Outter this&0 指针便指向了外部类对象，因此可以在成员内部类中随意访问外部类的成员。从这里也间接说明了成员内部类是依赖于外部类的，如果没有创建外部类的对象，则无法对Outter this&0引用进行初始化赋值，也就无法创建成员内部类的对象了。

　　2.为什么局部内部类和匿名内部类只能访问局部final变量？

　　想必这个问题也曾经困扰过很多人，在讨论这个问题之前，先看下面这段代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public class Test {      public static void main(String[] args)  {        }        public void test(final int b) {          final int a = 10;          new Thread(){              public void run() {                  System.out.println(a);                  System.out.println(b);              };          }.start();      }  } |

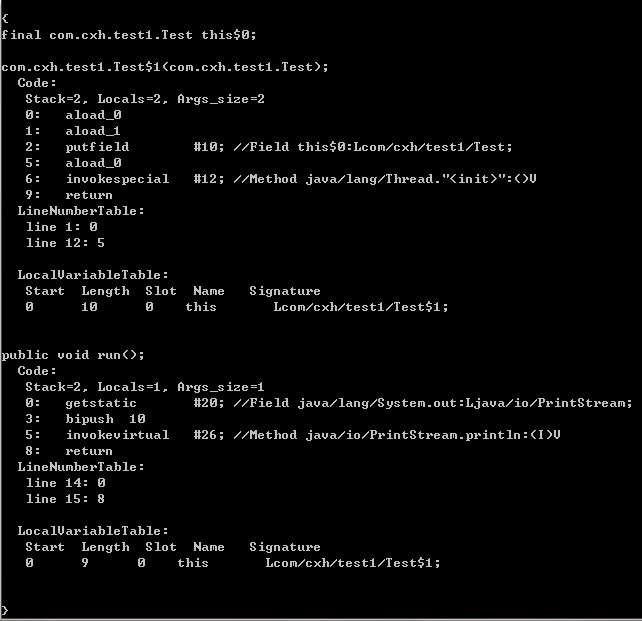
　　这段代码会被编译成两个class文件：Test.class和Testhttps://mathjax.cnblogs.com/2_6_1/fonts/HTML-CSS/TeX/png/Main/Regular/400/0031.png?rev=2.6.1https://mathjax.cnblogs.com/2_6_1/fonts/HTML-CSS/TeX/png/Main/Regular/400/002E.png?rev=2.6.1https://mathjax.cnblogs.com/2_6_1/fonts/HTML-CSS/TeX/png/Math/Italic/400/0063.png?rev=2.6.1https://mathjax.cnblogs.com/2_6_1/fonts/HTML-CSS/TeX/png/Math/Italic/400/006C.png?rev=2.6.1https://mathjax.cnblogs.com/2_6_1/fonts/HTML-CSS/TeX/png/Math/Italic/400/0061.png?rev=2.6.1https://mathjax.cnblogs.com/2_6_1/fonts/HTML-CSS/TeX/png/Math/Italic/400/0073.png?rev=2.6.1https://mathjax.cnblogs.com/2_6_1/fonts/HTML-CSS/TeX/png/Math/Italic/400/0073.png?rev=2.6.1。默认情况下，编译器会为匿名内部类和局部内部类起名为https://mathjax.cnblogs.com/2_6_1/fonts/HTML-CSS/TeX/png/Math/Italic/400/004F.png?rev=2.6.1https://mathjax.cnblogs.com/2_6_1/fonts/HTML-CSS/TeX/png/Math/Italic/400/0075.png?rev=2.6.1https://mathjax.cnblogs.com/2_6_1/fonts/HTML-CSS/TeX/png/Math/Italic/400/0074.png?rev=2.6.1https://mathjax.cnblogs.com/2_6_1/fonts/HTML-CSS/TeX/png/Math/Italic/400/0074.png?rev=2.6.1https://mathjax.cnblogs.com/2_6_1/fonts/HTML-CSS/TeX/png/Math/Italic/400/0065.png?rev=2.6.1https://mathjax.cnblogs.com/2_6_1/fonts/HTML-CSS/TeX/png/Math/Italic/400/0072.png?rev=2.6.11.class。默认情况下，编译器会为匿名内部类和局部内部类起名为Outterx.class（x为正整数）。



　　根据上图可知，test方法中的匿名内部类的名字被起为 Test$1。

　　上段代码中，如果把变量a和b前面的任一个final去掉，这段代码都编译不过。我们先考虑这样一个问题：

　　当test方法执行完毕之后，变量a的生命周期就结束了，而此时Thread对象的生命周期很可能还没有结束，那么在Thread的run方法中继续访问变量a就变成不可能了，但是又要实现这样的效果，怎么办呢？Java采用了 复制  的手段来解决这个问题。将这段代码的字节码反编译可以得到下面的内容：



　　我们看到在run方法中有一条指令：

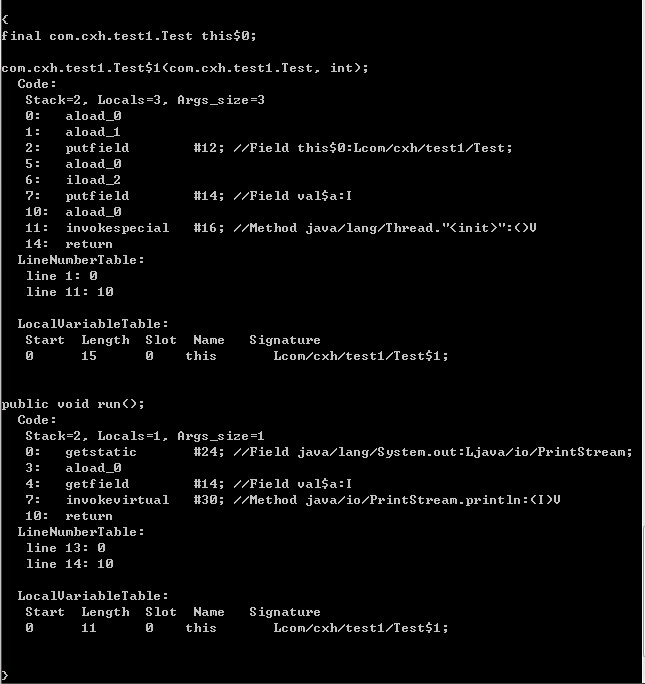
bipush 10

　　这条指令表示将操作数10压栈，表示使用的是一个本地局部变量。这个过程是在编译期间由编译器默认进行，如果这个变量的值在编译期间可以确定，则编译器默认会在匿名内部类（局部内部类）的常量池中添加一个内容相等的字面量或直接将相应的字节码嵌入到执行字节码中。这样一来，匿名内部类使用的变量是另一个局部变量，只不过值和方法中局部变量的值相等，因此和方法中的局部变量完全独立开。

　　下面再看一个例子：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | public class Test {      public static void main(String[] args)  {        }        public void test(final int a) {          new Thread(){              public void run() {                  System.out.println(a);              };          }.start();      }  } |

　　反编译得到：



　　我们看到匿名内部类Test$1的构造器含有两个参数，一个是指向外部类对象的引用，一个是int型变量，很显然，这里是将变量test方法中的形参a以参数的形式传进来对匿名内部类中的拷贝（变量a的拷贝）进行赋值初始化。

　　也就说如果局部变量的值在编译期间就可以确定，则直接在匿名内部里面创建一个拷贝。如果局部变量的值无法在编译期间确定，则通过构造器传参的方式来对拷贝进行初始化赋值。

　　从上面可以看出，在run方法中访问的变量a根本就不是test方法中的局部变量a。这样一来就解决了前面所说的 生命周期不一致的问题。但是新的问题又来了，既然在run方法中访问的变量a和test方法中的变量a不是同一个变量，当在run方法中改变变量a的值的话，会出现什么情况？

　　对，会造成数据不一致性，这样就达不到原本的意图和要求。为了解决这个问题，java编译器就限定必须将变量a限制为final变量，不允许对变量a进行更改（对于引用类型的变量，是不允许指向新的对象），这样数据不一致性的问题就得以解决了。

　　到这里，想必大家应该清楚为何 方法中的局部变量和形参都必须用final进行限定了。

　　3.静态内部类有特殊的地方吗？

　　从前面可以知道，静态内部类是不依赖于外部类的，也就说可以在不创建外部类对象的情况下创建内部类的对象。另外，静态内部类是不持有指向外部类对象的引用的，这个读者可以自己尝试反编译class文件看一下就知道了，是没有Outter this&0引用的。

**三.内部类的使用场景和好处**

　　为什么在Java中需要内部类？总结一下主要有以下四点：

　　1.每个内部类都能独立的继承一个接口的实现，所以无论外部类是否已经继承了某个(接口的)实现，对于内部类都没有影响。内部类使得多继承的解决方案变得完整，

　　2.方便将存在一定逻辑关系的类组织在一起，又可以对外界隐藏。

　　3.方便编写事件驱动程序

　　4.方便编写线程代码

　　个人觉得第一点是最重要的原因之一，内部类的存在使得Java的多继承机制变得更加完善。

四.常见的与内部类相关的笔试面试题

 1.根据注释填写(1)，(2)，(3)处的代码

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | public class Test{      public static void main(String[] args){             // 初始化Bean1             (1)             bean1.I++;             // 初始化Bean2             (2)             bean2.J++;             //初始化Bean3             (3)             bean3.k++;      }      class Bean1{             public int I = 0;      }        static class Bean2{             public int J = 0;      }  }    class Bean{      class Bean3{             public int k = 0;      }  } |

　　从前面可知，对于成员内部类，必须先产生外部类的实例化对象，才能产生内部类的实例化对象。而静态内部类不用产生外部类的实例化对象即可产生内部类的实例化对象。

　　创建静态内部类对象的一般形式为：  外部类类名.内部类类名 xxx = new 外部类类名.内部类类名()

　　创建成员内部类对象的一般形式为：  外部类类名.内部类类名 xxx = 外部类对象名.new 内部类类名()

　　因此，（1），（2），（3）处的代码分别为：

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

2.下面这段代码的输出结果是什么？

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public class Test {      public static void main(String[] args)  {          Outter outter = new Outter();          outter.new Inner().print();      }  }      class Outter  {      private int a = 1;      class Inner {          private int a = 2;          public void print() {              int a = 3;              System.out.println("局部变量：" + a);              System.out.println("内部类变量：" + this.a);              System.out.println("外部类变量：" + Outter.this.a);          }      }  } |

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif View Code

　　最后补充一点知识：关于成员内部类的继承问题。一般来说，内部类是很少用来作为继承用的。但是当用来继承的话，要注意两点：

　　1）成员内部类的引用方式必须为 Outter.Inner.

　　2）构造器中必须有指向外部类对象的引用，并通过这个引用调用super()。这段代码摘自《Java编程思想》

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | class WithInner {      class Inner{        }  }  class InheritInner extends WithInner.Inner {        // InheritInner() 是不能通过编译的，一定要加上形参      InheritInner(WithInner wi) {          wi.super(); //必须有这句调用      }        public static void main(String[] args) {          WithInner wi = new WithInner();          InheritInner obj = new InheritInner(wi);      }  } |

 参考资料：

　　《java编程思想》

<http://www.cnblogs.com/chenssy/p/3388487.html>

<http://blog.csdn.net/zhangjg_blog/article/details/20000769>

<http://blog.csdn.net/zhangjg_blog/article/details/19996629>

<http://blog.csdn.net/zhaoqianjava/article/details/6849812>

<http://www.cnblogs.com/nerxious/archive/2013/01/24/2875649.html>