http://blog.csdn.net/zhangjg\_blog/article/details/20000769

1. **深入理解Java中为什么内部类可以访问外部类的成员**

原创 2014年02月26日 22:54:26

* 标签：
* [Java](http://so.csdn.net/so/search/s.do?q=Java&t=blog) /
* [内部类](http://so.csdn.net/so/search/s.do?q=%E5%86%85%E9%83%A8%E7%B1%BB&t=blog) /
* [访问外部类的成员](http://so.csdn.net/so/search/s.do?q=%E8%AE%BF%E9%97%AE%E5%A4%96%E9%83%A8%E7%B1%BB%E7%9A%84%E6%88%90%E5%91%98&t=blog) /
* [字节码反编译](http://so.csdn.net/so/search/s.do?q=%E5%AD%97%E8%8A%82%E7%A0%81%E5%8F%8D%E7%BC%96%E8%AF%91&t=blog) /
* [javap](http://so.csdn.net/so/search/s.do?q=javap&t=blog)
* 20608

1. 内部类简介

虽然Java是一门相对比较简单的编程语言，但是对于初学者， 还是有很多东西感觉云里雾里， 理解的不是很清晰。内部类就是一个经常让初学者感到迷惑的特性。 即使现在我自认为Java学的不错了， 但是依然不是很清楚。其中一个疑惑就是为什么内部类对象可以访问外部类对象中的成员（包括成员变量和成员方法）？ 早就想对内部类这个特性一探究竟了，今天终于抽出时间把它研究了一下。

内部类就是定义在一个类内部的类。定义在类内部的类有两种情况：一种是被static关键字修饰的， 叫做静态内部类， 另一种是不被static关键字修饰的， 就是普通内部类。 在下文中所提到的内部类都是指这种不被static关键字修饰的普通内部类。 静态内部类虽然也定义在外部类的里面， 但是它只是在形式上（写法上）和外部类有关系， 其实在逻辑上和外部类并没有直接的关系。而一般的内部类，不仅在形式上和外部类有关系（写在外部类的里面）， 在逻辑上也和外部类有联系。 这种逻辑上的关系可以总结为以下两点：

**1 内部类对象的创建依赖于外部类对象；**

**2 内部类对象持有指向外部类对象的引用。**

上边的第二条可以解释为什么在内部类中可以访问外部类的成员。就是因为内部类对象持有外部类对象的引用。但是我们不禁要问， 为什么会持有这个引用？ 接着向下看， 答案在后面。

1. 通过反编译字节码获得答案

在源代码层面， 我们无法看到原因，因为Java为了语法的简洁， 省略了很多该写的东西， 也就是说很多东西本来应该在源代码中写出， 但是为了方便起见， 不必在源码中写出，编译器在编译时会加上一些代码。 现在我们就看看Java的编译器为我们加上了什么？

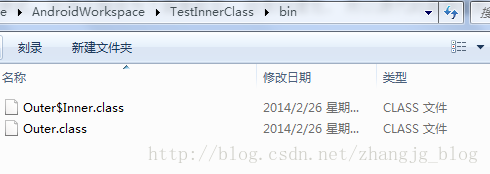
首先建一个工程TestInnerClass用于测试。 在该工程中为了简单起见， 没有创建包， 所以源代码直接在默认包中。在该工程中， 只有下面一个简单的文件。

**[java]** [view plain](http://blog.csdn.net/zhangjg_blog/article/details/20000769) [copy](http://blog.csdn.net/zhangjg_blog/article/details/20000769)

1. **public** **class** Outer {
2. **int** outerField = 0;
4. **class** Inner{
5. **void** InnerMethod(){
6. **int** i = outerField;
7. }
8. }
9. }

该文件很简单， 就不用过多介绍了。 在外部类Outer中定义了内部类Inner， 并且在Inner的方法中访问了Outer的成员变量outerField。

虽然这两个类写在同一个文件中， 但是编译完成后， 还是生成各自的class文件：



这里我们的目的是探究内部类的行为， 所以只反编译内部类的class文件Outer$Inner.class 。 在命令行中， 切换到工程的bin目录， 输入以下命令反编译这个类文件：

**[plain]** [view plain](http://blog.csdn.net/zhangjg_blog/article/details/20000769) [copy](http://blog.csdn.net/zhangjg_blog/article/details/20000769)

1. javap -classpath . -v Outer$Inner

-classpath .   说明在当前目录下寻找要反编译的class文件

-v   加上这个参数输出的信息比较全面。包括常量池和方法内的局部变量表， 行号， 访问标志等等。

注意， 如果有包名的话， 要写class文件的全限定名， 如：

**[plain]** [view plain](http://blog.csdn.net/zhangjg_blog/article/details/20000769) [copy](http://blog.csdn.net/zhangjg_blog/article/details/20000769)

1. javap -classpath . -v com.baidu.Outer$Inner

反编译的输出结果很多， 为了篇幅考虑， 在这里我们省略了常量池。 下面给出除了常量池之外的输出信息。

**[plain]** [view plain](http://blog.csdn.net/zhangjg_blog/article/details/20000769) [copy](http://blog.csdn.net/zhangjg_blog/article/details/20000769)

1. {
2. final Outer this$0;
3. flags: ACC\_FINAL, ACC\_SYNTHETIC

6. Outer$Inner(Outer);
7. flags:
8. Code:
9. stack=2, locals=2, args\_size=2
10. 0: aload\_0
11. 1: aload\_1
12. 2: putfield      #10                 // Field this$0:LOuter;
13. 5: aload\_0
14. 6: invokespecial #12                 // Method java/lang/Object."<init>":()V
15. 9: return
16. LineNumberTable:
17. line 5: 0
18. LocalVariableTable:
19. Start  Length  Slot  Name   Signature
20. 0      10     0  this   LOuter$Inner;
22. void InnerMethod();
23. flags:
24. Code:
25. stack=1, locals=2, args\_size=1
26. 0: aload\_0
27. 1: getfield      #10                 // Field this$0:LOuter;
28. 4: getfield      #20                 // Field Outer.outerField:I
29. 7: istore\_1
30. 8: return
31. LineNumberTable:
32. line 7: 0
33. line 8: 8
34. LocalVariableTable:
35. Start  Length  Slot  Name   Signature
36. 0       9     0  this   LOuter$Inner;
37. 8       1     1     i   I
38. }

首先我们会看到， 第一行的信息如下：

**[plain]** [view plain](http://blog.csdn.net/zhangjg_blog/article/details/20000769) [copy](http://blog.csdn.net/zhangjg_blog/article/details/20000769)

1. final Outer this$0;

这句话的意思是， 在内部类Outer$Inner中， 存在一个名字为this$0 ， 类型为Outer的成员变量， 并且这个变量是final的。 其实这个就是所谓的“在内部类对象中存在的指向外部类对象的引用”。但是我们在定义这个内部类的时候， 并没有声明它， 所以这个成员变量是编译器加上的。

虽然编译器在创建内部类时为它加上了一个指向外部类的引用， 但是这个引用是怎样赋值的呢？毕竟必须先给他赋值，　它才能指向外部类对象。　下面我们把注意力转移到构造函数上。 下面这段输出是关于构造函数的信息。

**[plain]** [view plain](http://blog.csdn.net/zhangjg_blog/article/details/20000769) [copy](http://blog.csdn.net/zhangjg_blog/article/details/20000769)

1. Outer$Inner(Outer);
2. flags:
3. Code:
4. stack=2, locals=2, args\_size=2
5. 0: aload\_0
6. 1: aload\_1
7. 2: putfield      #10                 // Field this$0:LOuter;
8. 5: aload\_0
9. 6: invokespecial #12                 // Method java/lang/Object."<init>":()V
10. 9: return
11. LineNumberTable:
12. line 5: 0
13. LocalVariableTable:
14. Start  Length  Slot  Name   Signature
15. 0      10     0  this   LOuter$Inner;

我们知道， 如果在一个类中， 不声明构造方法的话， 编译器会默认添加一个无参数的构造方法。 但是这句话在这里就行不通了， 因为我们明明看到， 这个构造函数有一个构造方法， 并且类型为Outer。 所以说，编译器会为内部类的构造方法添加一个参数， 参数的类型就是外部类的类型。

下面我们看看在构造参数中如何使用这个默认添加的参数。 我们来分析一下构造方法的字节码。 下面是每行字节码的意义：

aload\_0 ：

 将局部变量表中的第一个引用变量加载到操作数栈。 这里有几点需要说明。 局部变量表中的变量在方法执行前就已经初始化完成；局部变量表中的变量包括方法的参数；成员方法的局部变量表中的第一个变量永远是this；操作数栈就是执行当前代码的栈。所以这句话的意思是： 将this引用从局部变量表加载到操作数栈。

aload\_1：

将局部变量表中的第二个引用变量加载到操作数栈。 这里加载的变量就是构造方法中的Outer类型的参数。

putfield      #10                 // Field this$0:LOuter;

使用操作数栈顶端的引用变量为指定的成员变量赋值。 这里的意思是将外面传入的Outer类型的参数赋给成员变量this$0 。

这一句putfield字节码就揭示了， 指向外部类对象的这个引用变量是如何赋值的。

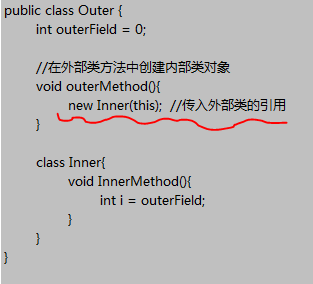
下面几句字节码和本文讨论的话题无关， 只做简单的介绍。 下面几句字节码的含义是： 使用this引用调用父类（Object）的构造方法然后返回。

用我们比较熟悉的形式翻译过来， 这个内部类和它的构造函数有点像这样： （注意， 这里不符合Java的语法， 只是为了说明问题）

**[java]** [view plain](http://blog.csdn.net/zhangjg_blog/article/details/20000769) [copy](http://blog.csdn.net/zhangjg_blog/article/details/20000769)

1. **class** Outer$Inner{
2. **final** Outer **this**$0;
4. **public** Outer$Inner(Outer outer){
5. **this**.**this**$0 = outer;
6. **super**();
7. }
8. }

说到这里， 可以推想到， 在调用内部类的构造器初始化内部类对象的时候， 编译器默认也传入外部类的引用。 调用形式有点像这样： （注意， 这里不符合java的语法， 只是为了说明问题）



这也印证了上面所说的内部类和外部类逻辑关系的第一条： 内部类对象的创建依赖于外部类对象。

关于在内部类中如何使用指向外部类的引用访问外部类成员， 就不用多做解释了， 其实和普通的通过引用访问成员的方式是相同的。 在内部类的InnerMethod方法中， 访问了外部类的成员变量outerField， 下面的字节码揭示了访问是如何进行的：

**[plain]** [view plain](http://blog.csdn.net/zhangjg_blog/article/details/20000769) [copy](http://blog.csdn.net/zhangjg_blog/article/details/20000769)

1. void InnerMethod();
2. flags:
3. Code:
4. stack=1, locals=2, args\_size=1
5. 0: aload\_0
6. 1: getfield      #10                 // Field this$0:LOuter;
7. 4: getfield      #20                 // Field Outer.outerField:I
8. 7: istore\_1
9. 8: return

getfield      #10                 // Field this$0:LOuter;

将成员变量this$0加载到操作数栈上来

getfield      #20                 // Field Outer.outerField:I

使用上面加载的this$0引用， 将外部类的成员变量outerField加载到操作数栈

istore\_1

将操作数栈顶端的int类型的值保存到局部变量表中的第二个变量上（注意， 第一个局部变量被this占用， 第二个局部变量是i）。操作数栈顶端的int型变量就是上一步加载的outerField变量。 所以， 这句字节码的含义就是： 使用outerField为i赋值。

上面三步就是内部类中是如何通过指向外部类对象的引用， 来访问外部类成员的。

1. 总结

文章写到这里， 相信读者对整个原理就会有一个清晰的认识了。 下面做一下总结：

本文通过反编译内部类的字节码， 说明了内部类是如何访问外部类对象的成员的，除此之外， 我们也对编译器的行为有了一些了解， 编译器在编译时会自动加上一些逻辑， 这正是我们感觉困惑的原因。

关于内部类如何访问外部类的成员， 分析之后其实也很简单， 主要是通过以下几步做到的：

1 编译器自动为内部类添加一个成员变量， 这个成员变量的类型和外部类的类型相同， 这个成员变量就是指向外部类对象的引用；

2 编译器自动为内部类的构造方法添加一个参数， 参数的类型是外部类的类型， 在构造方法内部使用这个参数为1中添加的成员变量赋值；

3 在调用内部类的构造函数初始化内部类对象时， 会默认传入外部类的引用。