Java内部类解析及其运用

Java的语言特性相比c#来说少了很多，最明显的一个地方是java里没有函数指针这样的机制，而c#里有“委托”这个概念来充当函数指针的作用。那么java里用什么来实现类似c#里用函数指针完成的任务——比如委托事件这样的机制？其实java用的是普通的接口，而这里有一个概念起了很重要的作用，那就是内部类。本文就围绕java内部类，先介绍基本用法和编译器对内部类的解释。在以上两点的基础上理解java内部类如何在一些重要的编程场合发挥作用。

第一部分：基本用法

**从语法层面上来看，内部类是作为拥有它的外部类的一个成员或是方法里的局部变量来用的**。

当内部类作为类的成员来用的时候可以像下面这样声明一个内部类:

**class** Outer{

**private** **class** Inner{

}

}

声明一个类Outer，然后在Outer的里面声明一个内部类Inner，这样就ok了。这里姑且称Outer为“外部类（相对其内部类Inner而言）”（下文同）。要注意的是在内部类上有个访问修饰符private，这在普通类上是不允许的。为什么可以这样做？其实不奇怪，思考一下这部分开头加粗的那句话就很容易理解了。除了private，常用于一般成员变量的修饰符如：static，final（此时final还是代表该内部类不能被继承，而不是”常类“的意思）都可以用来修饰内部类，而他们的意义也与一般的成员变量类似。比如由于Inner被修饰为private，那么在Outer外就不能访问到Inner。这种情况下还有其他的一些限制：

1. 当内部类由static修饰时，内部不能访问其外部类的任何非静态成员。
2. 当内部类由static修饰时，不能在内部类里声明静态成员，但是可以声明静态常量。

当内部类作为局部变量来用的时候可以像下面这样声明一个内部类：

**class** Outer{

**public** **void** someMethod() {

**class** LocalClass{}

LocalClass instance = **new** LocalClass();

// do other thing...

}

}

上面的代码在Outer的someMethod方法里声明了一个LocalClass内部类，接着在方法内部初始化了该内部类的一个对象。你甚至可以在任意的局部作用域里声明一个内部类，然后在该作用于里使用它。这一切看起来是那么地怪异，但无论如何请先从语法层面来理解这里所做的事情：在局部作用域里声明的内部类被当作该作用域里的局部变量来使用。由于这个内部类被当作了一个”局部变量“，所以相比第一种情况，这里多了几个限制：

1. 访问限制：出了作用域，其他任何地方都访问不到该内部类。
2. 修饰符限制：不能为内部类加访问修饰符，static修饰符。但是可以加final修饰符。

上面是内部类的基本使用情况，更详细的可以参考其他资料。除了语法限制，可以像往常定义普通类一样在内部类里添加成员，并在合适的地方使用它。

第二部分：编译器对内部类的”解释“

**JVM对内部类一无所知，换句话说，在JVM看来，内部类和普通类没有什么区别。**

这一部分要一探究竟java编译器这货到底对内部类做了什么！所以开始需要频繁地用到javap工具来查看编译器编译后的字节码。不过在这之前先来看下面一个代码片段：

**class** Outer{

**private** String outerField = "justin";

**void** outerMethod(){

System.***out***.println(outerField);

**new** Inner().innerMethod();

System.***out***.println(outerField);

}

**class** Inner{

**void** innerMethod() {

outerField = "hello justin";

}

}

}/\*output:

justin

hello justin

\*///~

在主函数中初始化一个Outer实例，然后执行Outer的outerMethod方法。可以看到程序输出结果，不管这个结果是不是在你的预料之中，我们得先来分析一下这个程序。现在要求你从语法层面”解脱出来”。开头说过：在JVM开来，内部类和普通类没有什么区别。如果是这样，上面这两个类的定义在JVM看来应该是像下面这样的：

**class** Outer{

**private** String outerField = "justin";

**void** outerMethod(){

System.***out***.println(outerField);

**new** Inner().innerMethod();

System.***out***.println(outerField);

}

}

**class** Inner{

**void** innerMethod() {

outerField = "hello justin";

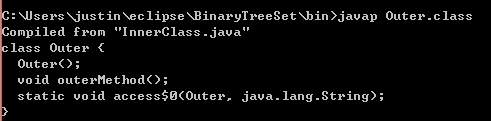
}

}

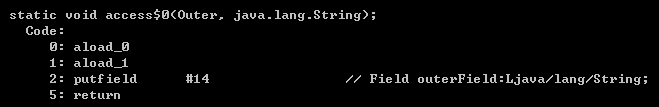
但是问题来了，Inner类的innerMethod方法如何访问Outer类的outerField，它是私有的！

如果想要让程序顺利运行，你可能会想到如下方法：在Outer类里定义一个公共方法，这个方法要求传递一个Outer类的实例，和一个String实例，在该方法内部修改outerField的值(因为它只能在Outer类内部被修改)。Inner类的innerMethod方法必须持有一个Outer类实例，然后调用Outer类的那个“不知名”方法。但是这一切都不是我们完成的，我们也不可能这样去做！

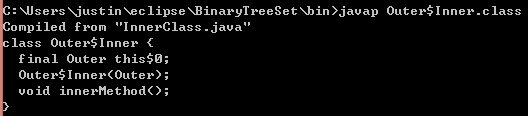
下面通过查看编译后的字节码来验证编译器是否“迎合”了我们的想法。编译后源文件后确实会产生两个class文件：Outer.class 和 Outer$Inner.class。先用查看Outer文件，在控制台运行命令：javap Outer.class，看到输出下面的方法信息：



可以看到确实有一个我们没有定义过的，字还有点古怪的方法，它就是我们想要的。如果想看一下该方法内部的实现，可以运行命令：javap –c Outer.class，-c指示javap工具显示方法内部字节码。可以看到代码其实很简单，关键的一句是putfield指令，它将传进来的Outer实例的outerField字段修改为方法的第二个参数。

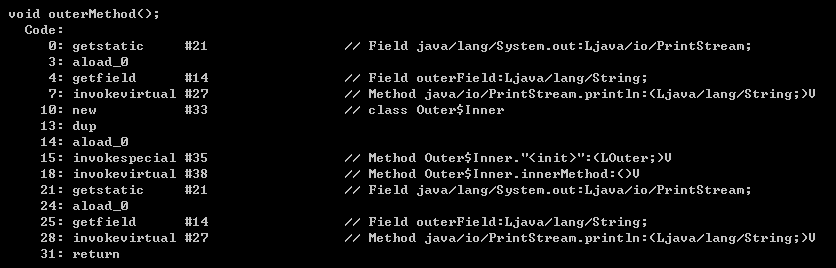


看来java编译器完全按照我们之前想的帮我们做好了一些工作，问题似乎已经解决了。但是这里还有一个问题：innerMethod方法里的Outer对象是怎么来的？毕竟outerMethod方法并没有在初始化Inner实例的时候做任何相关的事情啊。或许又是编译器做了什么小动作。所以这次要查看Outer$Inner.class文件。运行命令：javap Outer$Inner.class。



Inner类似乎多了一些东西：一个Outer类型的常量字段和一个带有Outer类型参数的构造器。看到这些应该就会很容易明白上面提出的问题了。顺便再来看看Outer的outerMethod方法的字节码，了解编译后的代码从头到尾到底是如何执行的，运行之前的

javap –c Outer.class命令看outerMethod方法内的实现：



14和15行告诉你内部类实例是如何构造的：内部类构造器的Outer参数来自于方法的第一个参数this。

下面对上面的做一下总结：

1. 编译器为每个内部类生成的类名为：外部类$内部类名
2. 对于非静态的内部类，它的实例自动地持有一个外部类实例的引用。事实上内部类的初始化必须依赖于某一个外部类实例。在初始化内部类对象的时候会将对应的外部类对象传递给内部类的构造器。Ps:构造内部类语法：外部类对象.new 内部类名()。
3. 静态内部类的实例不持有外部类实例引用，因为静态内部类根本无法访问外部类的实例成员（这个由编译器来保证）。
4. 无论是静态还是非静态，内部类要访问外部类对外不可见的成员时就会在外部类内部生成一个静态辅助方法，以提供对这些成员的访问支持。这些方法对我们是不可见的。
5. 在非静态内部类中可以通过“外部类名.this”访问它持有的外部类对象。

最后一个问题就是：对于在局部作用域里定义的内部类，编译器产生的字节码会有什么不同呢？看一个使用局部内部类的代码片段：

**class** Outer{

**private** String outerField = "justin";

**void** outerMethod(){

**final** String hello = **new** String("hello ");

**class** Inner{

**void** innerMethod() {

outerField = hello + "justin";

}

}

System.***out***.println(outerField);

**new** Inner().innerMethod();

System.***out***.println(outerField);

}

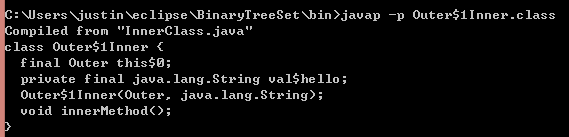
}/\*output:

justin

hello justin

\*///~

这段代码和前面的代码类似，变化就是把内部类的定义放到了outerMethod内部，然后在内部类的innerMethod方法里访问了一个outerMethod里的一个局部变量（用new关键词初始化字符串是为了防止编译器”抹去“该局部变量）。这仍旧是一个夸作用域访问变量的情况，我们要看的就是在内部类如何访问方法的局部变量。现在产生的内部类文件有点不一样：Outer$1Inner.class。我们运行命令javap -p Outer$1Inner.class –p选项告诉javap列出所有私有成员。如下结果：



可以看到内部类除了仍旧持有一个外部类对象外，还有一个String类型的私有常量字段val$hello，同时在构造器的参数里多了一个String类型的参数。看到这些估计就很容易明白内部类的方法innerMethod为什么会访问到局部变量了：在构造内部类的时候将局部变量通过构造函数参数的方式传递给了内部类实例。

总结一下局部内部类的情况：

1. 编译器生成的内部类名有所变化：外部类$+某个数字+内部类名。这样做是为了区分在不同的作用域里定义的同名内部类，毕竟这些名称相同的类并不是同一个类。
2. 内部类要访问的局部变量会通过构造函数参数的形式传递给内部类。
3. 内部类访问的局部变量必须是final的。
4. 在静态作用域（比如静态方法）里定义的内部类不需要持有外部类对象，因为这种情况下内部类访问不到外部类的实例成员。
5. 局部作用域里定义的内部类和在类作用域里定义的内部类对JVM来说没有本质的区别。

第三部分：前两部分的补充说明

1. JVM对内部类一无所知，编译器做所有的“翻译”工作，对JVM隐藏了内部类。
2. 编译器在内部类的使用上做了很多的语法限制，这可能是为了保持语法的一致性。毕竟你不希望看到对一个局部内部类的定义加上public修饰符，这样没有意义。
3. 虽然JVM对内部类一无所知，但是编译后的类文件还是留下了“内部类”的影子。编译器会在内部类文件上加上一些标记，记录这“曾经是个内部类”，同样会在外部类文件上说明“它曾经有哪些内部类”。

第四部分：内部类应用

**内部类提供了某种进入其外部类的窗口——《java编程思想》**

1. 数据封装

内部类一个很重要的特性是它可以访问外部类的私有成员，利用这一点可以保持外部类更好的封装性。查看java集合类库的源代码，可以看到大量的内部类使用，其中典型的一个运用就是迭代器。下面是一个典型的迭代器代码“模板”：

**class** SomeList<E> **implements** Iterable<E>{

**private** E[] data;

**public** Iterator<E> iterator(){

**return** **new** Itr();

}

**private** **class** Itr **implements** Iterator<E>{

// 在内部类里面访问集合（外部类）对象数据

// the field data is accessable...

}

}

在这里集合的iterator方法返回一个迭代器，而这个迭代器就是一个内部类实例，内部类可以对外界完全隐藏，因为外界只需要获取一个抽象的迭代器。作为迭代器的内部类需要和外部集合类有很强的联系，因为迭代器需要访问集合的私有数据，这就是内部类的好处。如果不用内部类实现迭代器，那么我们必须在集合类里定义访问私有数据的接口，比如在上面就要定义一个方法返回私有的data成员，这样必然会破坏集合的封装性。

1. “多继承”特性

Java集合类库中还可以看到一种现象就是一个集合可以返回不同的迭代器，已满足不同的迭代方式。比如一个很自然的需求就是既要求集合可以顺序迭代元素，也可以以逆序方式迭代，or any other way you like。考虑下面的代码：

**interface** Selector{

**void** select();

}

**class** SomeList<E> **implements** Selector {

**private** E[] data;

**public** **void** select() {

**for** (**int** i = 0; i < data.length; i++) {

System.***out***.println(data[i]);

}

}

**public** **static** **void** test(Selector selector) {

selector.select();

}

}

上面的代码片段没有使用java集合里的代码模式，为了简单起见，我只是用一个相对简单但是不偏离用意的模式。SomeList<E>是某一个数据容器，他可以以某种方式遍历其中的数据（实现Selector接口）。这里的问题是，SomeList<E>只是Selector的某一个特定实现，他不能再以其他方式实现Selector，或者说SomeList<E>只能以一种固定的方式遍历。为了让SomeList<E>满足多种遍历规则，可以使用内部类对代码作如下修改：

**class** SomeList<E> **implements** Selector {

**private** E[] data;

**public** **void** select() {

**for** (**int** i = 0; i < data.length; i++) {

System.***out***.println(data[i]);

}

}

**public** Selector reverseSelector(){

**class** reverseSelector **implements** Selector{

**public** **void** select() {

**for** (**int** i = 0; i < data.length; i++) {

System.***out***.println(data[data.length - 1 - i]);

}

}

}

**return** **new** reverseSelector();

}

**public** **static** **void** test(Selector selector) {

selector.select();

}

}

我在SomeList<E>里加了reverseSelector方法，他返回一个Selector对象，该对象实现了Selector接口，它支持以逆序的方式遍历数据。这就好像SomeList<E>对别人说，我是一个Selector，我可以以正常顺序的方式遍历自己，也可以以逆序的方式。看起来SomeList<E>好像具有多种行为方式，而这不需要SomeList<E>通过继承某一个超类或者实现某一个接口来实现。

1. 回调与闭包

在讲这一部分之前有必要先解释一下匿名内部类。我使用匿名内部类的方式把前面SomeList<E>类的reverseSelector方法做如下修改：

**public** Selector reverseSelector(){

**return** **new** Selector() {

**public** **void** select() {

**for** (**int** i = 0; i < data.length; i++) {

System.***out***.println(data[data.length - 1 - i]);

}

}

};

}

语法看上去有点怪，如果查看两个版本编译后的字节码，可以发现两个版本其实没有本质的区别，编译器会自动为匿名内部类加上名字。匿名内部类的存在是为了简化语法，我们不需要在一个局部作用域里声明一个完整的类（这样看起来才怪呢）。由于没有名字，匿名内部类的初始化必须和某一个超类或接口相关，代表创建的对象其实是某一个继承它的子类的对象。在匿名内部类里可以覆写父类的方法，或者定义自己的成员（不过对于成员方法似乎没有什么意义，因为无法调用到它）。

回归正题，题目中的“回调”是指一段将来某一时刻执行的代码，”闭包“是一个可以调用的对象。下面以一个控制系统的例子来说明匿名内部类在这方面的用武之地:

**interface** Callable{

**void** call();

}

**abstract** **class** Event{

**private** Callable callBack;

**public** Event(Callable callBack){**this**.callBack = callBack;}

**abstract** **void** start();

**public** **void** go() {

start();

**if**(callBack != **null**)

callBack.call();

}

}

**abstract** **class** ControlSystem{

List<Event> eventList = **new** ArrayList<>();

**public** **void** addEvent(Event e) {

eventList.add(e);

}

**public** **void** start() {

**while** (eventList.size() > 0) {

**for** (Event event : **new** ArrayList<>(eventList)) {

event.go();

eventList.remove(event);

}

}

}

}

上面是这个控制系统的框架代码，抽象类ControlSystem包含了一系列的事件，当它启动的（start方法）就会依次地调用事件，每一个事件结束后会回调系统外的某段代码。接下来以一个厨房做饭的特定实现来使用这个控制系统:

**class** KitchenSystem **extends** ControlSystem{

**int** boilWaterCount = 0;

KitchenSystem(){

addEvent(**new** BoilWater(**new** WaterBoiled()));

addEvent(**new** Cooking(**new** Callable() {

**public** **void** call() {

System.***out***.println("end cooking");

}

}));

addEvent(**new** Washing(**new** Callable() {

**public** **void** call() {

System.***out***.println("end washing");

}

}));

start();

}

**class** WaterBoiled **implements** Callable{

**public** **void** call() {

**if**(boilWaterCount < 10)

addEvent(**new** BoilWater(**new** WaterBoiled()));

**else**

System.***out***.println("end boiling water");

}

}

**class** BoilWater **extends** Event{

**public** BoilWater(Callable callBack){**super**(callBack);}

**public** **void** start(){

System.***out***.println("boiling water..." + ++boilWaterCount);

}

}

**class** Cooking **extends** Event{

**public** Cooking(Callable callBack){**super**(callBack);}

**public** **void** start(){

System.***out***.println("cooking...");

}

}

**class** Washing **extends** Event{

**public** Washing(Callable callBack){**super**(callBack);}

**public** **void** start(){

System.***out***.println("cooking...");

}

}

}

厨房控制系统继承自抽象控制系统，当它被初始化的时候，依次往事件列表里加入了烧水，做饭，洗碗三个事件。为三个事件各传递一个回调对象，代表事件结束后要用的代码。对于后两个事件，我直接以匿名内部类的方式声明回调对象。而第一个烧水事件有点不同，他的回调函数（WaterBoiled的call方法）要求烧水达到一定数目，否则就向控制系统里添加烧水事件（也就是继续烧水）。这里的回调和闭包是分不开的，声明每一个事件时为事件定义的回掉函数是以闭包的形式传进去的，为每一个事件传递的内部类对象就是一个可调用的对象（闭包）。

这里很容易让人联想到在GUI编程时使用的事件，我们在为控件添加监听对象的时候会大量地用到类似的编码模式：匿名内部类。为控件传递的内部类实例不就是一个回调对象吗？

文章结束，谢谢阅读~^.^