https://zhuanlan.zhihu.com/p/103258844?utm_source=qq

目录

内部类初探

什么是内部类?

内部类的默认构造器传递的参数是外部类对象，作为内部类对象的一个成员属性保存。因此内部类持有外部类对象的引用

内部类是指在一个外部类的内部再定义一个类。内部类作为外部类的一个成员，并且依附于外部类而存在的。内部类可为静态，可用protected和private修饰（而外部类只能使用public和缺省的包访问权限）。内部类主要有以下几类：成员内部类、局部内部类、静态内部类、匿名内部类

内部类的共性

> (1)内部类仍然是一个独立的类，在编译之后内部类会被编译成独立的.class文件，但是前面冠以外部类的类名和$符号 。
>
> (2)内部类不能用普通的方式访问。
>
> (3)内部类声明成静态的，就不能随便的访问外部类的成员变量了，此时内部类只能访问外部类的静态成员变量 。
>
> (4)外部类不能直接访问内部类的的成员，但可以通过内部类对象来访问

内部类是外部类的一个成员，因此内部类可以自由地访问外部类的成员变量，无论是否是private的。

因为当某个外围类的对象创建内部类的对象时，此内部类会捕获一个隐式引用，它引用了实例化该内部对象的外围类对象。通过这个指针，可以访问外围类对象的全部状态。

通过反编译内部类的字节码，分析之后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。

使用内部类的好处：

> 静态内部类的作用：
>
> 1 只是为了降低包的深度，方便类的使用，静态内部类适用于包含类当中，但又不依赖与外在的类。
>
> 2 由于Java规定静态内部类不能用使用外在类的非静态属性和方法，所以只是为了方便管理类结构而定义。于是我们在创建静态内部类的时候，不需要外部类对象的引用。
>
> 非静态内部类的作用：
>
> 1 内部类继承自某个类或实现某个接口，内部类的代码操作创建其他外围类的对象。所以你可以认为内部类提供了某种进入其外围类的窗口。
>
> 2 使用内部类最吸引人的原因是:每个内部类都能独立地继承自一个(接口的)实现，所以无论外围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

> 3 如果没有内部类提供的可以继承多个具体的或抽象的类的能力，一些设计与编程问题就很难解决。 从这个角度看，内部类使得多重继承的解决方案变得完整。接口解决了部分问题，而内部类有效地实现了"多重继承"。

那静态内部类与普通内部类有什么区别呢?

问得好，区别如下：

> （1）静态内部类不持有外部类的引用 在普通内部类中，我们可以直接访问外部类的属性、方法，即使是private类型也可以访问，这是因为内部类持有一个外部类的引用，可以自由访问。而静态内部类，则只可以访问外部类的静态方法和静态属性（如果是private权限也能访问，这是由其代码位置所决定的），其他则不能访问。
>
> （2）静态内部类不依赖外部类 普通内部类与外部类之间是相互依赖的关系，内部类实例不能脱离外部类实例，也就是说它们会同生同死，一起声明，一起被垃圾回收器回收。而静态内部类是可以独立存在的，即使外部类消亡了，静态内部类还是可以存在的。
>
> （3）普通内部类不能声明static的方法和变量 普通内部类不能声明static的方法和变量，注意这里说的是变量，常量（也就是final static修饰的属性）还是可以的，而静态内部类形似外部类，没有任何限制。

为什么普通内部类不能有静态变量呢?

> 1 成员内部类 之所以叫做成员 就是说他是类实例的一部分 而不是类的一部分

> 2 结构上来说 他和你声明的成员变量是一样的地位 一个特殊的成员变量 而静态的变量是类的一部分和实例无关

> 3 你若声明一个成员内部类 让他成为主类的实例一部分 然后又想在内部类声明和实例无关的静态的东西 你让JVM情何以堪啊

> 4 若想在内部类内声明静态字段 就必须将其内部类本身声明为静态

非静态内部类有一个很大的优点：可以自由使用外部类的所有变量和方法

下面的例子大概地介绍了

1 非静态内部类和静态内部类的区别。

2 不同访问权限的内部类的使用。

3 外部类和它的内部类之间的关系

```
//本节讨论内部类以及不同访问权限的控制
//内部类只有在使用时才会被加载。
//外部类B
public class B{
```

```
    int i = 1;
    int j = 1;
    static int s = 1;
    static int ss = 1;
    A a;
    AA aa;
    AAA aaa;
    //内部类A

    public class A {
//        static void go () {
//
//        }
//        static {
//
//        }
//      static int b = 1;//非静态内部类不能有静态成员变量和静态代码块和静态方法，
        // 因为内部类在外部类加载时并不会被加载和初始化。
        //所以不会进行静态代码的调用
        int i = 2;//外部类无法读取内部类的成员，而内部类可以直接访问外部类成员

        public void test() {
            System.out.println(j);
            j = 2;
            System.out.println(j);
            System.out.println(s);//可以访问类的静态成员变量
        }
        public void test2() {
            AA aa = new AA();
            AAA aaa = new AAA();
        }

    }
    //静态内部类S，可以被外部访问
    public static class S {
        int i = 1;//访问不到非静态变量。
        static int s = 0;//可以有静态变量

        public static void main(String[] args) {
            System.out.println(s);
        }
        @Test
        public void test () {
//            System.out.println(j);//报错，静态内部类不能读取外部类的非静态变量
            System.out.println(s);
            System.out.println(ss);
            s = 2;
            ss = 2;
            System.out.println(s);
            System.out.println(ss);
        }
    }

    //内部类AA，其实这里加protected相当于default
    //因为外部类要调用内部类只能通过B。并且无法直接继承AA，所以必须在同包
```

```
    //的类中才能调用到(这里不考虑静态内部类)，那么就和default一样了。
    protected class AA{
        int i = 2;//内部类之间不共享变量
        public void test (){
            A a = new A();
            AAA aaa = new AAA();
            //内部类之间可以互相访问。
        }
    }
    //包外部依然无法访问，因为包没有继承关系，所以找不到这个类
    protected static class SS{
        int i = 2;//内部类之间不共享变量
        public void test (){

            //内部类之间可以互相访问。
        }
    }
    //私有内部类A，对外不可见，但对内部类和父类可见
    private class AAA {
        int i = 2;//内部类之间不共享变量

        public void test() {
            A a = new A();
            AA aa = new AA();
            //内部类之间可以互相访问。
        }
    }
    @Test
    public void test(){
        A a = new A();
        a.test();
        //内部类可以修改外部类的成员变量
        //打印出 1 2
        B b = new B();

    }
}
```

//另一个外部类 class C { @Test public void test() { //首先，其他类内部类只能通过外部类来获取其实例。 B.S s = new B.S(); //静态内部类可以直接通过B类直接获取，不需要B的实例，和静态成员变量类似。 //B.A a = new B.A(); //当A不是静态类时这行代码会报错。 //需要使用B的实例来获取A的实例 B b = new B(); B.A a = b.new A(); B.AA aa = b.new AA();//B和C同包，所以可以访问到AA // B.AAA aaa = b.new AAA();AAA为私有内部类，外部类不可见 //当A使用private修饰时，使用B的实例也无法获取A的实例，这一点和私有变量是一样的。 //所有普通的内部类与类中的一个变量是类似的。静态内部类则与静态成员类似。 } }

内部类的加载

可能刚才的例子中没办法直观地看到内部类是如何加载的，接下来用例子展示一下内部类加载的过程。

> 1 内部类是延时加载的，也就是说只会在第一次使用时加载。不使用就不加载，所以可以很好的实现单例模式。
>
> 2 不论是静态内部类还是非静态内部类都是在第一次使用时才会被加载。
>
> 3 对于非静态内部类是不能出现静态模块（包含静态块，静态属性，静态方法等）
>
> 4 非静态类的使用需要依赖于外部类的对象，详见上述对象innerClass 的初始化。

简单来说，类的加载都是发生在类要被用到的时候。内部类也是一样

> 1 普通内部类在第一次用到时加载，并且每次实例化时都会执行内部成员变量的初始化，以及代码块和构造方法。
>
> 静态内部类执行clinit方法初始化静态变量
>
> 2 静态内部类也是在第一次用到时被加载。但是当它加载完以后就会将静态成员变量初始化，运行静态代码块，并且只执行一次。当然，非静态成员和代码块每次实例化时也会执行。

总结一下Java类代码加载的顺序，万变不离其宗。

> 规律一、初始化构造时，先父后子；只有在父类所有都构造完后子类才被初始化
>
> 规律二、类加载先是静态、后非静态、最后是构造函数。
>
> 只执行一次　　new一次就执行
>
> 静态构造块、静态类属性按出现在类定义里面的先后顺序初始化，同理非静态的也是一样的，只是静态的只在加载字节码时执行一次，不管你new多少次，非静态会在new多少次就执行多少次
>
> 规律三、java中的类只有在被用到的时候才会被加载
>
> 规律四、java类只有在类字节码被加载后才可以被构造成对象实例

成员内部类　当作“局部变量”理解

在方法中定义的内部类称为局部内部类。与局部变量类似，局部内部类不能有访问说明符，因为它不是外围类的一部分，但是它可以访问当前代码块内的常量，和此外围类所有的成员。

需要注意的是：局部内部类只能在定义该内部类的方法内实例化，不可以在此方法外对其实例化。

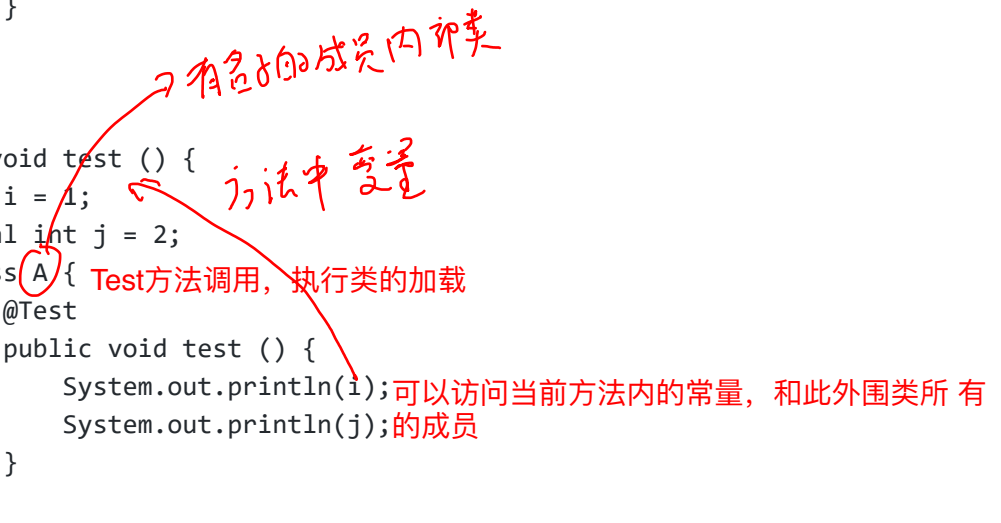
```
public class 局部内部类 {
    class A {//局部内部类就是写在方法里的类，只在方法执行时加载，一次性使用。
        public void test() {
            class B {
                public void test () {
```

```
                    class C {

                    }
                }
            }
        }
    }
    @Test
    public void test () {
        int i = 1;
        final int j = 2;
        class A {
            @Test
            public void test () {
                System.out.println(i);
                System.out.println(j);
            }
        }
        A a = new A();
        System.out.println(a);
    }

    static class B {
        public static void test () {
            //static class A报错，方法里不能定义静态内部类。
            //因为只有在方法调用时才能进行类加载和初始化。

        }
    }
}
```



匿名内部类

简单地说：匿名内部类就是没有名字的内部类，并且，匿名内部类是局部内部类的一种特殊形式。什么情况下需要使用匿名内部类？如果满足下面的一些条件，使用匿名内部类是比较合适的： 只用到类的一个实例。 类在定义后马上用到。 类非常小（SUN推荐是在4行代码以下） 给类命名并不会导致你的代码更容易被理解。 在使用匿名内部类时，要记住以下几个原则：

1 匿名内部类不能有构造方法。

2 匿名内部类不能定义任何静态成员、方法和类。

3 匿名内部类不能是public,protected,private,static。

4 只能创建匿名内部类的一个实例。

5 一个匿名内部类一定是在new的后面，用其隐含实现一个接口或实现一个类。

6 因匿名内部类为局部内部类，所以局部内部类的所有限制都对其生效。

一个匿名内部类的例子：

```
    public class 匿名内部类 {

}
interface D{
    void run ();
}
abstract class E{
    E (){

    }
    abstract void work();
}
class A {

        @Test
        public void test (int k) {
            //利用接口写出一个实现该接口的类的实例。
            //有且仅有一个实例，这个类无法重用。
            new Runnable() {
                @Override
                public void run() {
//                      k = 1;报错，当外部方法中的局部变量在内部类使用中必须改为final类型。
                    //因为方外部法中即使改变了这个变量也不会反映到内部类中。
                    //所以对于内部类来讲这只是一个常量。
                    System.out.println(100);
                    System.out.println(k);
                }
            };
            new D(){ D是一个接口，这里匿名内部类没有名字，直接实现接口中方法
                //实现接口的匿名类
                int i =1;
                @Override
                public void run() {
                    System.out.println("run");
                    System.out.println(i);
                    System.out.println(k);
                }
            }.run();
            new E(){只用到类的一个实例
                //继承抽象类的匿名类
                int i = 1;
                void run (int j) {
                    j = 1;
                }

                @Override
                void work() {

                }
            };
        }
```

```
}
```

匿名内部类里的final

使用的形参为何要为final

参考文件：http://android.blog.51cto.com/268543/384844

> 我们给匿名内部类传递参数的时候，若该形参在内部类中需要被使用，那么该形参必须要为final。也就是说：当所在的方法的形参需要被内部类里面使用时，该形参必须为final。
>
> 为什么必须要为final呢?
>
> 首先我们知道在内部类编译成功后，它会产生一个class文件，该class文件与外部类并不是同一class文件，仅仅只保留对外部类的引用。当外部类传入的参数需要被内部类调用时，从java程序的角度来看是直接被调用：

```
public class OuterClass {
    public void display(final String name,String age){
        class InnerClass{
            void display(){
                System.out.println(name);
            }
        }
    }
}
```

从上面代码中看好像name参数应该是被内部类直接调用？其实不然，在java编译之后实际的操作如下：

```
public class OuterClass$InnerClass {
    public InnerClass(String name,String age){内部类生成包含display参数的构造函数
        this.InnerClass$name = name;
        this.InnerClass$age = age;
    }
public void display(){ System.out.println(this.InnerClass$name + "----" +
this.InnerClass$age ); } }
```

所以从上面代码来看，内部类并不是直接调用方法传递的参数，而是利用自身的构造器对传入的参数进行备份，自己内部方法调用的实际上时自己的属性而不是外部方法传递进来的参数。

> 直到这里还没有解释为什么是final

> 在内部类中的属性和外部方法的参数两者从外表上看是同一个东西，但实际上却不是，所以他们两者是可以任意变化的，也就是说在内部类中我对属性的改变并不会影响到外部的形参，而然这从程序员的角度来看这是不可行的。
> 引用对外部方法参数，通过内部类的构造函数，进行了复制一份，使用时候使用了复制后的参数
> 毕竟站在程序的角度来看这两个根本就是同一个，如果内部类该变了，而外部方法的形参却没有改变这是难以理解和不可接受的，所以为了保持参数的一致性，就规定使用final来避免形参的不改变。 为什么局部内部类传参必须是final?

简单理解就是，拷贝引用，为了避免引用值发生改变，例如被外部类的方法修改等，而导致内部类得到的值不一致，于是用final来让该引用不可改变。

故如果定义了一个匿名内部类，并且希望它使用一个其外部定义的参数，那么编译器会要求该参数引用是final的。

匿名内部类初始化

我们一般都是利用构造器来完成某个实例的初始化工作的，但是匿名内部类是没有构造器的！那怎么来初始化匿名内部类呢？使用构造代码块！利用构造代码块能够达到为匿名内部类创建一个构造器的效果。

```
public class OutClass {
    public InnerClass getInnerClass(final int age,final String name){
        return new InnerClass() {
            int age_ ;
            String name_;
            //构造代码块完成初始化工作
            {
                if(0 < age && age < 200){
                    age_ = age;
                    name_ = name;
                }
            }
            public String getName() {
                return name_;
            }

            public int getAge() {
                return age_;
            }
        };
    }
```

内部类的重载

如果你创建了一个内部类，然后继承其外围类并重新定义此内部类时，会发生什么呢？也就是说，内部类可以被重载吗？这看起来似乎是个很有用的点子，但是“重载”内部类就好像它是外围类的一个方法，其实并不起什么作用：

```
class Egg {
       private Yolk y;

       protected class Yolk { //内部类命名空间Egg$1
              public Yolk() {
                     System.out.println("Egg.Yolk()");
              }
       }

       public Egg() {
              System.out.println("New Egg()");
              y = new Yolk();
       }
}

public class BigEgg extends Egg {
    public class Yolk {    //内部类命名空间BigEgg$1
              public Yolk() {
                     System.out.println("BigEgg.Yolk()");
              }
       }

       public static void main(String[] args) {
              new BigEgg();
       }
}
复制代码
输出结果为：
New Egg()
Egg.Yolk()
```

缺省的构造器是编译器自动生成的，这里是调用基类的缺省构造器。你可能认为既然创建了BigEgg 的对象，那么所使用的应该是被“重载”过的Yolk，但你可以从输出中看到实际情况并不是这样的。 这个例子说明，当你继承了某个外围类的时候，内部类并没有发生什么特别神奇的变化。这两个内部类是完全独立的两个实体，各自在自己的命名空间内。

内部类的继承

因为内部类的构造器要用到其外围类对象的引用，所以在你继承一个内部类的时候，事情变得有点复杂。问题在于，那个“秘密的”外围类对象的引用必须被初始化，而在被继承的类中并不存在要联接的缺省对象。要解决这个问题，需使用专门的语法来明确说清它们之间的关联：

```
class WithInner {
        class Inner {
                Inner(){
                        System.out.println("this is a constructor in
WithInner.Inner");
                };
        }
}

public class InheritInner extends WithInner.Inner {
        // ! InheritInner() {} // Won't compile
        InheritInner(WithInner wi) {
                wi.super();
                System.out.println("this is a constructor in InheritInner");
        }

        public static void main(String[] args) {
                WithInner wi = new WithInner();
                InheritInner ii = new InheritInner(wi);
        }
}
```

复制代码 输出结果为： this is a constructor in WithInner.Inner this is a constructor in InheritInner

可以看到，InheritInner 只继承自内部类，而不是外围类。但是当要生成一个构造器时，缺省的构造器并不算好，而且你不能只是传递一个指向外围类对象的引用。此外，你必须在构造器内使用如下语法： enclosingClassReference.super(); 这样才提供了必要的引用，然后程序才能编译通过。

有关匿名内部类实现回调，事件驱动，委托等机制的文章将在下一节讲述。

Java内部类的实现原理

内部类为什么能够访问外部类的成员?

定义内部类如下：

```java
package test02;

public class Main{
    public int m = 10;
    // 静态内部类
    class Inner{
        public void go() {
            System.out.println(m);
        }
    }
}
```

使用javap命令进行反编译。

编译后得到Main.class Main$Inner.class两个文件，反编译Main$Inner.class文件如下：

```
public class Main { //反编译字节码之后，对比字节码得到的信息得到的伪代码
    public int age=10;
    class MyInnerClass{
        final synthetic this$0=外部类Main对象//编译器添加的synthetic变量，持有外部类对象引用
        MyInnerClass(外部类main对象){//构造函数中传递外部类对象的引用
            this.this$0=外部类main对象//外部类对象引用保存到内部类的编译器添加的属性中
        }
        public void go(){
            获取 this$0  //通过内部类拷贝的外部类对象的引用，获取外部类对象的属性age
            获取age数值 this$0.age
            System.out.println(打印获取到的age数值);
        }
    }

}
```

外部类Main和内部类MyInnerClass的class文件中都保存了外部类、内部类的路径名

可以看到，内部类其实拥有外部类的一个引用，在构造函数中将外部类的引用传递进来。

匿名内部类为什么只能访问局部的final变量？

局部内部类对象完全可以逃逸出方法体（比如挂到外部类的static变量上）

其实可以这样想，当方法执行完毕后，局部变量的生命周期就结束了，而局部内部类对象的生命周期可能还没有结束，那么在局部内部类中访问局部变量就不可能了，所以将局部变量改为final，改变其生命周期。

编写代码如下：

```java
public class Main{
    public void go(final int n) {
        final int m = 10;
        // 创建匿名内部类
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println(m);
                System.out.println(n);
            }
        }).start();
    }
}
```

编译期间生成的字节码run方法中，已经确定操作数栈数值。bipush 10

直接复制m值，放入操作数栈

是Thread对象初始化，复制到Thread中成员属性val$n字段上

n通过当前内部类对象，构造期间传递的外部final数值，复制给内部类对象的成员变量。使用n使用直接从当前内部类对象中调用成员变量getfield字节码

这段代码编译为Main.class Main$1.class两个文件，反编译Main$1.class文件如下：

C:\WINDOWS\system32\cmd.exe

```
E:\rubbish\software\java\hello\bin\test02>javap -v Main$1.class
Classfile /E:/rubbish/software/java/hello/bin/test02/Main$1.class
  Last modified 2018-10-12; size 677 bytes
  MD5 checksum 0956c90075d2da52625c239ed4691085
  Compiled from "Main.java"
class test02.Main$1 implements java.lang.Runnable
  minor version: 0
  major version: 52
  flags: ACC_SUPER
Constant pool:
   #1 = Class              #2             // test02/Main$1
   #2 = Utf8               test02/Main$1
   #3 = Class              #4             // java/lang/Object
   #4 = Utf8               java/lang/Object
   #5 = Class              #6             // java/lang/Runnable
   #6 = Utf8               java/lang/Runnable
   #7 = Utf8               this$0
   #8 = Utf8               Ltest02/Main;
   #9 = Utf8               val$n
  #10 = Utf8               I
  #11 = Utf8               <init>
  #12 = Utf8               (Ltest02/Main;I)V
  #13 = Utf8               Code
  #14 = Fieldref           #1.#15         // test02/Main$1.this$0:Ltest02/Main;
  #15 = NameAndType        #7:#8          // this$0:Ltest02/Main;
  #16 = Fieldref           #1.#17         // test02/Main$1.val$n:I
  #17 = NameAndType        #9:#10         // val$n:I
  #18 = Methodref          #3.#19         // java/lang/Object."<init>":()V
  #19 = NameAndType        #11:#20        // "<init>":()V
  #20 = Utf8               ()V
  #21 = Utf8               LineNumberTable
  #22 = Utf8               LocalVariableTable
  #23 = Utf8               this
  #24 = Utf8               Ltest02/Main$1;
  #25 = Utf8               run
  #26 = Fieldref           #27.#29        // java/lang/System.out:Ljava/io/PrintStream;
  #27 = Class              #28            // java/lang/System
  #28 = Utf8               java/lang/System
  #29 = NameAndType        #30:#31        // out:Ljava/io/PrintStream;
  #30 = Utf8               out
  #31 = Utf8               Ljava/io/PrintStream;
  #32 = Methodref          #33.#35        // java/io/PrintStream.println:(I)V
  #33 = Class              #34            // java/io/PrintStream
  #34 = Utf8               java/io/PrintStream
  #35 = NameAndType        #36:#37        // println:(I)V
  #36 = Utf8               println
  #37 = Utf8               (I)V
  #38 = Utf8               SourceFile
  #39 = Utf8               Main.java
  #40 = Utf8               EnclosingMethod
  #41 = Class              #42            // test02/Main
  #42 = Utf8               test02/Main
  #43 = NameAndType        #44:#37        // go:(I)V
  #44 = Utf8               go
  #45 = Utf8               InnerClasses
{
  final test02.Main this$0;
    descriptor: Ltest02/Main;
    flags: ACC_FINAL, ACC_SYNTHETIC

  test02.Main$1(test02.Main, int);
    descriptor: (Ltest02/Main;I)V
    flags:
    Code:
      stack=2, locals=3, args_size=3
         0: aload_0
         1: aload_1
         2: putfield      #14                 // Field this$0:Ltest02/Main;
         5: aload_0
         6: iload_2
         7: putfield      #16                 // Field val$n:I
        10: aload_0
        11: invokespecial #18                 // Method java/lang/Object."<init>":()V
        14: return
      LineNumberTable:
        line 1: 0
        line 7: 10
      LocalVariableTable:
        Start  Length  Slot  Name   Signature
            0      15     0  this   Ltest02/Main$1;
```

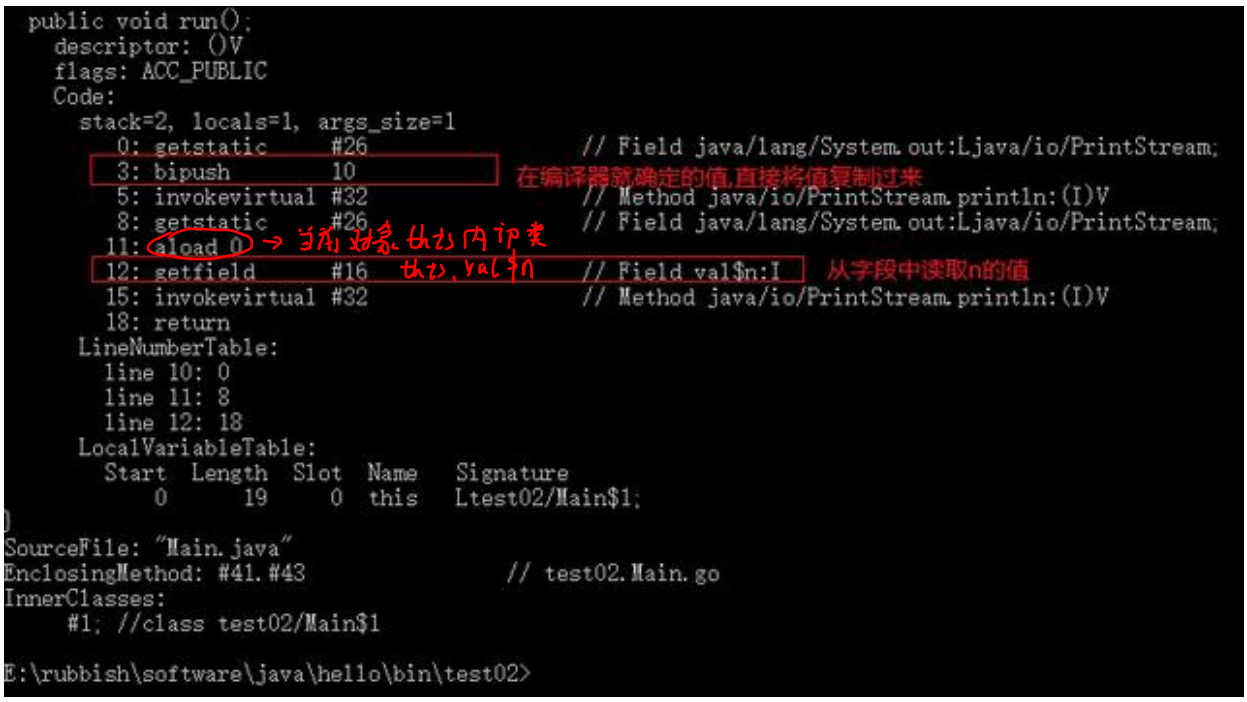
保存变量

常量池中保存final常量数值

内部类构造器传递外部类对象，和外部方法中final参数。到内部类的构造器中，进一步赋值到内部类的成员属性中，供内部类方法使用。

外部类对象写入内部类成员属性中

从常量池中读取值保存到字段中



可以看到，java将编译时已经确定的值直接复制，进行替换，将无法确定的值放到了内部类的常量池中，并在构造函数中将其从常量池取出到字段中。

可以看出，java将局部变量m直接进行复制，所以其并不是原来的值，若在内部类中将m更改，局部变量的m值不会变，就会出现数据不一致，所以java就将其限制为final，使其不能进行更改，这样数据不一致的问题就解决了。

参考文章

https://www.cnblogs.com/hujingnb/p/10181621.html
https://blog.csdn.net/codingtu/article/details/79336026
https://www.cnblogs.com/woshimrf/p/java-inner-class.html
https://www.cnblogs.com/dengchengchao/p/9713979.html

内部类

(一) 概述

把类定义在另一个类的内部，该类就被称为内部类。

举例：把类Inner定义在类Outer中，类Inner就被称为内部类。

```
class Outer {
    class Inner {
    }
}
```

(二) 内部类的访问规则

A:可以直接访问外部类的成员，包括私有

B:外部类要想访问内部类成员，必须创建对象

(三) 内部类的分类

A：成员内部类

B：局部内部类

C：静态内部类

D：匿名内部类

(1) 成员内部类

> 成员内部类——就是位于外部类成员位置的类
> 特点：可以使用外部类中所有的成员变量和成员方法（包括private的）

A：格式：

```
class Outer {
    private int age = 20;
    //成员位置
    class Inner {
        public void show() {
            System.out.println(age);
        }
    }
}

class Test {
    public static void main(String[] ages) {
        //成员内部类是非静态的演示
        Outer.Inner oi = new Outer().new Inner();
```

```
        成员内部类不是静态的：
    外部类名.内部类名 对象名 = new 外部类名.new        ();
//成员内部类是静态的：
外部类名.内部类名 对象名 = new 外部类名.内部类名();
```

C：成员内部类常见修饰符：

A：private

如果我们的内部类不想轻易被任何人访问，可以选择使用private修饰内部类，这样我们就无法通过创建对象的方法来访问，想要访问只需要在外部类中定义一个public修饰的方法，间接调用。这样做的好处就是，我们可以在这个public方法中增加一些判断语句，起到数据安全的作用。

```
class Outer {
    private class Inner {// 外部如果这样定义变量Outer.Inner oi=null就会报错，只能通过调用Outer类实例的方法后获取
        public void show() {
            System.out.println(“密码备份文件”);
        }
    }

    public void method() {
        if(你是管理员){
            Inner i = new Inner();
            i.show();
        }else {
            System.out.println(“你没有权限访问”);
        }
    }
}
```

下面我们给出一个更加规范的写法

```
class Outer {
    private class Inner {
        public void show() {
            System.out.println(“密码备份文件”);
        }
```

```
        }

        Inner i = new Inner();
        i.show();
    }
}
    public static void main(String[] args) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.getInner();
        inner.show();
    }
}
```

B：static

这种被 static 所修饰的内部类，按位置分，属于成员内部类，但也可以称作静态内部类，也常叫做嵌套内部类。具体内容我们在下面详细讲解。

```
        public int age = 20;
        public viod showAge() {
            int age  = 25;
            System.out.println(age);//空1
            System.out.println(this.age);//空2
            System.out.println(Outer.this.age);//空3
        }
    }
}
```

(2) 局部内部类

> 局部内部类——就是定义在一个方法或者一个作用域里面的类
> 特点：主要是作用域发生了变化，只能在自身所在方法和属性中被使用

A 格式：

```
class Outer {
    public void method(){
        class Inner {
        }
    }
}
```

B：访问时：

```
//在局部位置，可以创建内部类对象，通过对象调用和内部类方法
class Outer {
    private int age = 20;
    public void method() {
        final int age2 = 30;
        class Inner {
            public void show() {
                System.out.println(age);
                //从内部类中访问方法内变量age2
                System.out.println(age2);
          }

          Inner i = new Inner();
          i.show();
      }
  }
```

代码 1

```
class Outer {
   private int age = 20;
   public void method() {
      int age2 = 30;//如果非final得时候，可以编译运行，但是下面age2=40不能通过编译，因为内部类要求必须使用final常量
      class Inner {
         public void show() {
            System.out.println(age);
//从内部类中访问方法内变量age2，需要将变量声明为最终类型。
            System.out.println(age2);
         }
      }
      //age2 = 40;不能通过编译
      Inner i = new Inner();
```

代码2

如果访问的局部变量不加final内部类如何处理？比如代码2 中int age2=30编译器怎么处理？

反编译代码2字节码

```
class se基础.demo.内部类.Outer$1Inner {
  final int val$age2; 编译器为局部内部类访问的局部变量，通过创建final类成员变量进行保存，编译期间就保存类字节码中。

  final se基础.demo.内部类.Outer this$0; 编译器为局部内部类增加外部类对象的引用，作为内部类对象访问外部类对象入口

  se基础.demo.内部类.Outer$1Inner();内部类构造器中初始化成员变量（final局部变量和外部类对象的引用）
    Code:
       0: aload_0
       1: aload_1
       2: putfield      #1                  // Field this$0:Lse基础/demo/内部类/Outer;外部类对象保存到内部类对象的成员属性中
       5: aload_0
       6: iload_2
       7: putfield      #2                  // Field val$age2:I 方法中局部变量（不管是否final修饰）都保存到内部类final修饰的成员变量中
      10: aload_0
      11: invokespecial #3                  // Method java/lang/Object."<init>":()V
      14: return

  public void show();
    Code:
       0: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
       3: aload_0
       4: getfield      #1                  // Field this$0:Lse基础/demo/内部类/Outer;
       7: invokestatic  #5                  // Method se基础/demo/内部类/Outer.access$000:(Lse基础/demo/内部类/Outer;)I
这里编译器为外部类添加了access$000方法，字节码内容如下：本质上还是通过内部类持有的外部类引用this$0获取外部类对象的age
0 aload_0
1 getfield #1 <se基础/demo/内部类/Outer.age>
4 ireturn
      10: invokevirtual #6                  // Method java/io/PrintStream.println:(I)V
      13: getstatic     #4                  // Field java/lang/System.out:Ljava/io/PrintStream;
      16: aload_0
      17: getfield      #2                  // Field val$age2:I  通过内部类对象获得“拷贝”的局部变量 age2
      20: invokevirtual #6                  // Method java/io/PrintStream.println:(I)V
      23: return
}
```

总结：局部内部类访问的局部变量如果不用final修饰，编译器如何处理。
通过字节码可以看到，编译器把外部类对象和局部内部类访问的局部变量，通过内部类构造器“拷贝”到内部类对象的成员属性中，并且这些属性都是final修饰的，当需要使用外部类对象的属性或者方法中局部变量时候，通过内部类对象持有的外部类对象的引用获取外部类对象的成员变量和属性，通过内部类对象持有的final变量获取局部变量被“拷贝”的数值。反编译结果就是int age2 = 30;被替换为final int age2 = 30;

C: 为什么局部内部类访问局部变量必须加final修饰呢?

因为**局部变量是随着方法的调用而调用，使用完毕就消失，而堆内存的数据并不会立即消失。**所以，堆内存还是用该变量，而该变量已经没有了。**为了让该值还存在，就加final修饰。**原因是，当我们使用final修饰变量后，堆内存直接存储的**是值**，而**不是变量名。**final修饰之后在编译阶段class文件的常量池中就确定了该变量数值，以后也不再改变，保证了内部类对象无论什么时候访问都是同一个数值。

（即上例 age2 的位置存储着常量30 而不是 age2 这个变量名）

简单理解:

即使没有外部类对象，也可以创建静态内部类对象，而外部类的非static成员必须依赖于对象的调用，静态成员则可以直接使用类调用，不必依赖于外部类的对象，所以静态内部类只能访问静态的外部属性和方法。

```
class Outter {
    int age = 10;
    static age2 = 20;
    public Outter() {
    }

    static class Inner {
        public method() {
            System.out.println(age);//错误
            System.out.println(age2);//正确
        }
    }
}

public class Test {
    public static void main(String[] args)  {
        Outter.Inner inner = new Outter.Inner();
        inner.method();
    }
}
```

外部类先加载执行clinit方法，初始化了静态结构，然后内部类才被new,所以静态内部类只能访问外部类静态属性和方法

(4) 匿名内部类

> 一个没有名字的类，是内部类的简化写法

A 格式:

```
new 类名或者接口名() {
    重写方法();
}
```

本质：其实是继承该类或者实现接口的子类匿名对象

这也就是下例中，可以直接使用 new Inner() {}.show(); 的原因 == **子类**对象.show();

```
interface Inner {
    public abstract void show();
}

class Outer {
    public void method(){
        new Inner() {
            public void show() {
                System.out.println("HelloWorld");
            }
        }.show();
    }
}
class Test {
    public static void main(String[] args)  {
```



```
Inter i = new Inner() {  //多态，因为new Inner(){}代表的是接口的子类对象
    public void show() {
    System.out.println("HelloWorld");
    }
};
```

B：匿名内部类在开发中的使用

我们在开发的时候，会看到抽象类，或者接口作为参数。

而这个时候，实际需要的是一个子类对象。

如果该方法仅仅调用一次，我们就可以使用匿名内部类的格式简化。

使用内部类的原因

(一) 封装性

作为一个类的编写者，我们很显然需要对这个类的使用访问者的访问权限做出一定的限制，我们需要将一些我们不愿意让别人看到的操作隐藏起来，

如果我们的内部类不想轻易被任何人访问，可以选择使用private修饰内部类，这样我们就无法通过创建对象的方法来访问，想要访问只需要在外部类中定义一个public修饰的方法，间接调用。

```
public interface Demo {
    void show();
}

class Outer {
    private class test implements Demo {
        public void show() {
```

```
            System.out.println("密码备份文件");
        }
    }

    public Demo getInner() {
        return new test();
    }

}
```

我们来看其测试

```
    public static void main(String[] args) {
        Outer outer = new Outer();
        Demo d = outer.getInner();
        i.show();
    }

//
```

(二) 实现多继承 ※

我们之前的学习知道，java是不可以实现多继承的，一次只能继承一个类，我们学习接口的时候，有提到可以用接口来实现多继承的效果，即一个接口有多个实现，但是这里也是有一点弊端的，那就是，一旦实现一个接口就必须实现里面的所有方法，有时候就会出现一些累赘，但是使用内部类可以很好的解决这些问题

```
public class Demo1 {
    public String name() {
        return "BWH_Steven";
    }
}

public class Demo2 {
    public String email() {
        return "xxx.@163.com";
    }
}

public class MyDemo {

    private class test1 extends Demo1 {
        public String name() {
            return super.name();
        }
    }

    private class test2 extends Demo2  {
        public String email() {
            return super.email();
        }
    }

    public String name() {
        return new test1().name(); 内部类test1继承自Demo1的name方法
    }

    public String email() {
         return new test2().email(); 内部类test2继承自Demo2的email1方法
    }
```

```
    public static void main(String args[]) {
        MyDemo md = new MyDemo();
        System.out.println("我的姓名:" + md.name());
        System.out.println("我的邮箱:" + md.email());
    }
}
```

我们编写了两个待继承的类Demo1和Demo2，在MyDemo类中书写了两个内部类，test1和test2两者分别继承了Demo1和Demo2类，这样MyDemo中就间接的实现了多继承

(三) 用匿名内部类实现回调功能

我们用通俗讲解就是说在Java中，通常就是编写一个接口，然后你来实现这个接口，然后把这个接口的一个对象作以参数的形式传到另一个程序方法中， 然后通过接口调用你的方法，匿名内部类就可以很好的展现了这一种回调功能

```
    public static void main(String[] args) {
        MyDemo md = new MyDemo();
        //这里我们使用匿名内部类的方式将接口对象作为参数传递到test方法中去了
        md.test(new Demo){
            public void demoMethod(){
                System.out.println("具体实现接口")
            }
        }
    }
}
```

(四) 解决继承及实现接口出现同名方法的问题

编写一个接口 Demo

```
public interface Demo {
    void test();
}
```

编写一个类 MyDemo

```
public class MyDemo {

    public void test() {
        System.out.println("父类的test方法");
    }

}
```

编写一个测试类

```
public class DemoTest extends MyDemo implements Demo {
    public void test() {
    }
}
```

这样的话我就有点懵了，这样如何区分这个方法是接口的还是继承的，所以我们使用内部类解决这个问题

```
public class DemoTest extends MyDemo {


    private class inner implements Demo {
        public void test() {
            System.out.println("接口的test方法");
        }
    }

    public Demo getIn() {
        return new inner();
    }


    public static void main(String[] args) {
        //调用接口而来的test()方法
        DemoTest dt = new DemoTest();
```

```
接口的test方法
父类的test方法
```

静态内部类实现单例模式

```
public class Singleton {
    /***
     * 类的内部类，也就是静态成员的内部类，该内部类的实例与外部类的实例没有绑定关系
     * 只有被调用的时候才会装载，从而实现了延时加载
     *
     */
    public static class SingletonHolder {
         //静态初始化器，JVM保证线程安全
        private static Singleton instance = new Singleton();外部类对象的单例对象
    }
    /**
     * 私有化构造方法，不让外部可以new
    private Singleton(){
    }
    public static Singleton getInstance(){
        return SingletonHolder.instance;
    }
}
```

当getInstance方法第一次被调用的时候，她第一次读取SingletonHolder.instance，导致SingletonHolder类被初始化，而这个类在装载并初始化的时候会初始化他的静态域，从而创建singleton实例，由于是静态的域，所以只会在虚拟机装载类的时候加载一次，并由虚拟机保证她的线程安全
这个模式的优势在于，getInstance并没有被同步，并且只执行一个域的访问，因此延迟初始化并没有增加访问成本。还可以尝试使用单元素的枚举，来实现singleton，只要编写一个单元素的枚举类型即可。

问题：反射仍然可以破坏单例模式

```
Singleton singleton=Singleton.getInstance();
        Class objClass = Singleton.class;
        //获取类的构造器
        Constructor constructor = objClass.getDeclaredConstructor();
        //把构造器私有权限放开
         constructor.setAccessible(true);
         //反射创建实例
         Singleton reflectSingleton = (Singleton) constructor.newInstance();
         System.out.println(singleton);
         System.out.println(reflectSingleton);
         System.out.println(singleton==reflectSingleton);
```

```
se基础.demo.内部类.Singleton@1540e19d
se基础.demo.内部类.Singleton@677327b6
false
```

成员内部类可能导致内存泄漏：原因：生命周期长的内部类持有生命周期短的外部类

```
public class OutClass
{
  public class InnerClass
  {
  }
}
OutClass  oc= new OutClass();
OutClass .InnerClass ic= oc.new InnerClass();
```

通过上面的字节码反编译解析，我们知道内部类InnerClass对象在构造时候，构造器内部会持有外部类对象的OutClass的引用，无论InnerClass对象内部是否需要使用外部类的属性、方法。也就是说即使内部类不使用外部类对象的属性方法也会持有外部类对象的引用，如果外部类对象占用内存很大，如果我们需要内部类生命周期比外部类长，那么内部类持有短周期外部类对象的引用，导致外部类对象不能被垃圾回收，从而导致内存泄漏。

如何解决内部类可能导致的内存泄漏？

使用静态内部类，静态内部类构造器初始化时侯不持有外部类对象的引用，因为外部类和静态内部类是相对“独立”关系，不需要外部类对象就可以new静态内部类。

```
Outter.Inner inner = new Outter.Inner();//静态内部类new方式，不涉及到外部类对象
inner.method();
```