链滴

Java 中 final、finally 和 finalize 使用总结

作者：leopoldwu
原文链接：https://ld246.com/article/1498577336611
来源网站：链滴

Java中final、finally和finalize使用总结

---

1. final

1.1 final修饰变量

final用于修饰变量时表示该变量一旦被初始化就不可以再改变，这里的不可以再改变指的是，对于基类型它们的值不能改变，对于对象变量它们的引用不可以改变，对于后者需要注意的是，只是引用不改变，即指向初始化时的那个对象，对象中的属性等是可以改变的，例如我们有个final修饰的ArrayLis，那么这个变量只能是指向最开始初始化时的ArrayList对象，不能使它指向其他ArrayList对象，但是rrayList中的值是可以改变的。

final变量的初始化可以在定义处初始化，也可以在构造方法中初始化，另外还可以在代码块中初始化。

final一般和static一起修饰变量用于表示常量，需要注意的是static final修饰变量和final修饰变量是区别的，前者表示唯一不可改变，即类的所有对象共享一个变量，同时它在初始化后就不可以再改变而后者指的是每个对象中各自有自己的变量，它们被初始化后就不能再改变。

final修饰的成员变量不会有默认的初始化，而普通成员变量和静态成员变量会有默认的初始化，例如于int默认初始化是0，而引用类型默认初始化是null等。

局部内部类和匿名内部类中只能访问局部final变量，例如：

```
public class Test {
    public void test(final int b) {
        final int a = 10;
        new Thread(){
            public void run() {
                System.out.println(a);
                System.out.println(b);
            };
        }.start();
    }
}
```

我们可以尝试把a变量或者b变量的final给去掉，可以发现在高版本的java中是可以编译通过的，但是们加一条修改变量的语句，就会出现编译错误，提示该变量需要是final修饰的。也就是高版本的java编译器会默认帮我们添加final则个关键字。我们可以以上面的代码为例分析原因，当test方法执行完a和b的生命周期就结束了，但是Thread可能还没有结束，也就意味着再次访问a或者b变量就不可能，所以java中就用复制的方式来解决则个问题，即内部类中使用到的变量和方法中的变量不是同一个量，它只是一份拷贝，虽然这样可以解决生命周期的问题，但是因为访问的变量和原本并不是同一个量，如果修改就会出现不一致的问题，所以就需要把它设定为是final类型，让它不能够修改。

final修饰能够保证对象的安全发布，即对象在初始化完成之前能够保证不被其他线程使用，它的原理通过禁止CPU的指令集重排序。但是这样并不是能够保证线程安全，例如下面的代码：

```
public class Test{
    public static final List<Integer> list = new ArrayList<Integer>();
    public void add(Integer value){
        list.add(value);
```

```
    }
}
```

如果在多线程环境下，虽然能够保证list在没有被初始化之前不被其他线程访问到，但是里面的add方可能会出现线程同步问题，需要给方法或者里面的代码段加锁。

1.2 final修饰方法

当final修饰方法时表示该方法不能够被覆盖，但是该方法是可以被继承的；在java的早期版本中通过方法添加final可以带来性能的提升，但是现在的java版本中是不需要通过这种方式来优化的；另外，中的private方法会被隐式指定为final方法。

1.3 final修饰类

当一个类被指定为final时表示这个类不能够被继承，类中的成员变量可以指定为final，也可以不指定final，但是类中的方法都被隐式指定为final方法。

2. finally

finally用于创建在try-catch块后面执行的代码块，即无论是否发生异常都会执行到finally块中的代码但是这个不是绝对的，例如在try或者catch块中执行结束JVM进程的语句，如：System.exit(0)，这时finally块中的代码是不会执行的，还有例如线程中断等情况。

在finally块中最好不要写return语句以免使人误解，例如下面的代码：

```
try{
  return 0;
}catch(Exception e){
  return 1;
}finally{
  return 2;
}
```

在上面的代码中，如果没有异常我们的返回值是什么，这个可能会使人误解，在这种情况下返回值应是2，因为finally块中的代码是在try或者catch块中的return语句执行之后真正返回之前执行的，所以finally块中的return语句就先返回了，另外再看一个例子：

```
i = 0;
try{
  i = 1;
  return i;
}catch(Exception e){
  i = 2;
  return i;
}finally{
  i = 3;
}
```

我们先考虑没有异常情况下的返回值是什么，通过上一个例子的分析，可能会认为这个例子的返回值3，但是事实上返回值是1，这个和finally块的执行机制有关，可以把它看作类似方法的调用，我们知当我们调用一个方法时，传入一个基本变量的值给方法的形参，我们在方法中改变这个形参是不影响本的实参的，因为这是一个值传递的过程，传入形参的值只是原本实参的一个拷贝。同理，在finally中也有类似的机制，所以我们在finally块中修改i的值是不会影响try块中的返回值的。

我们继续看一个例子：

```
public static Map<String,Integer> getMap(){
    Map<String,Integer> map = new HashMap<>();
    try{
      map.put("test",1);
      return map;
    }catch(Exception e){
      map.put("test",1);
      return map;
    }finally{
      map.put("test",3);
    }
 }
public static void main(String[] args) {
    System.out.println(getMap().get("test"));
}
```

通过上一个例子的分析可以很容易知道这个例子的输出是3，这个例子和上一个例子不同的地方是一是基本变量，一个是对象，这里也可以通过类比方法调用来理解，Java中只有值传递没有地址传递，们把一个对象传入方法的形参，我们传递的只是原本实参（保存有对象在内存中的地址）的一份拷贝也就是相当于传递后有两个引用指向实际内存，那么我们通过形参修改当然可以修改到对象内容，但如果我们使形参指向另一个对象，因为原本实参中对象的地址没有改变，所以原本对象并没有改变，是形参自己指向了另一个对象而已。

3. finalize

finalize是一个方法，该方法是在垃圾回收器回收对象时调用到的一个方法，可以通过覆盖的方式在个方法中添加需要执行的代码，一般添加一些释放资源的代码。finalize相当于析构函数，不过一般需要自己来实现这个方法，也尽量不要用这个方法。用户可以自己调用这个方法，但是只是正常的方调用，与对象的销毁过程无关。