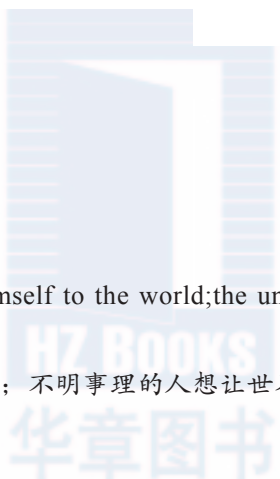




## 第 1 章

# Java 开发中通用的方法和准则



The reasonable man adapts himself to the world; the unreasonable one persists in trying to adapt the world to himself.

明白事理的人使自己适应世界；不明事理的人想让世界适应自己。

——萧伯纳

Java 的世界丰富又多彩，但同时也布满了荆棘陷阱，大家一不小心就可能跌入黑暗深渊，只有在了解了其通行规则后才能使自己在技术的海洋里遨游飞翔，恣意驰骋。

“千里之行始于足下”，本章主要讲述与 Java 语言基础有关的问题及建议的解决方案，例如常量和变量的注意事项、如何更安全地序列化、断言到底该如何使用等。

### 建议 1：不要在常量和变量中出现易混淆的字母

包名全小写，类名首字母全大写，常量全部大写并用下划线分隔，变量采用驼峰命名法（Camel Case）命名等，这些都是最基本的 Java 编码规范，是每个 Javaer 都应熟知的规则，但是在变量的声明中要注意不要引入容易混淆的字母。尝试阅读如下代码，思考一下打印出的 `i` 等于多少：

```
public class Client {  
    public static void main(String[] args) {  
        long i = 11;  
        System.out.println("i 的两倍是: " + (i+i));  
    }  
}
```

肯定有人会说：这么简单的例子还能出错？运行结果肯定是 22！实践是检验真理的唯一标准，将其拷贝到 Eclipse 中，然后 Run 一下看看，或许你会很奇怪，结果是 2，而不是 22，难道是 Eclipse 的显示有问题，少了个“2”？

因为赋给变量 `i` 的数字就是“1”，只是后面加了长整型变量的标示字母“1”而已。别说是我挖坑让你跳，如果有类似程序出现在项目中，当你试图通过阅读代码来理解作者的思想时，此情此景就有可能出现。所以，为了让您的程序更容易理解，字母“1”（还包括大写字母“O”）尽量不要和数字混用，以免使阅读者的理解与程序意图产生偏差。如果字母和数字必须混合使用，字母“1”务必大写，字母“O”则增加注释。

---

**注意** 字母“1”作为长整型标志时务必大写。

---

### 建议 2：莫让常量蜕变成变量

常量蜕变成变量？你胡扯吧，加了 `final` 和 `static` 的常量怎么可能会变呢？不可能二次赋值的呀。真的不可能吗？看我们神奇的魔术，代码如下：

```
public class Client {  
    public static void main(String[] args) {  
        System.out.println("常量会变哦: " + Const.RAND_CONST);  
    }  
}
```

```

}
/* 接口常量 */
interface Const{
    // 这还是常量吗?
    public static final int RAND_CONST = new Random().nextInt();
}

```

RAND\_CONST 是常量吗？它的值会变吗？绝对会变！这种常量的定义方式是极不可取的，常量就是常量，在编译期就必须确定其值，不应该在运行期更改，否则程序的可读性会非常差，甚至连作者自己都不能确定在运行期发生了何种神奇的事情。

甭想着使用常量会变的这个功能来实现序列号算法、随机种子生成，除非这真的是项目中的唯一方案，否则就放弃吧，常量还是当常量使用。

---

**注意** 务必让常量的值在运行期保持不变。

---

### 建议 3：三元操作符的类型务必一致

三元操作符是 if-else 的简化写法，在项目中使用它的地方很多，也非常好用，但是好用又简单的东西并不表示就可以随便用，我们来看看下面这段代码：

```

public class Client {
    public static void main(String[] args) {
        int i = 80;
        String s = String.valueOf(i<100?90:100);
        String s1 = String.valueOf(i<100?90:100.0);
        System.out.println(" 两者是否相等："+s.equals(s1));
    }
}

```

分析一下这段程序：i 是 80，那它当然小于 100，两者的返回值肯定都是 90，再转成 String 类型，其值也绝对相等，毋庸置疑的。恩，分析得有点道理，但是变量 s 中三元操作符的第二个操作数是 100，而 s1 的第二个操作数是 100.0，难道没有影响吗？不可能有影响吧，三元操作符的条件都为真了，只返回第一个值嘛，与第二个值有一毛钱的关系吗？貌似有道理。

果真如此吗？我们通过结果来验证一下，运行结果是：“两者是否相等：false”，什么？不相等，Why？

问题就出在了 100 和 100.0 这两个数字上，在变量 s 中，三元操作符中的第一个操作数（90）和第二个操作数（100）都是 int 类型，类型相同，返回的结果也就是 int 类型的 90，而变量 s1 的情况就有点不同了，第一个操作数是 90（int 类型），第二个操作数却是 100.0，而这是个浮点数，也就是说两个操作数的类型不一致，可三元操作符必须要返回一个数据，

而且类型要确定，不可能条件为真时返回 int 类型，条件为假时返回 float 类型，编译器是不允许如此的，所以它就会进行类型转换了，int 型转换为浮点数 90.0，也就是说三元操作符的返回值是浮点数 90.0，那这当然与整型的 90 不相等了。这里可能有读者疑惑了：为什么是整型转为浮点，而不是浮点转为整型呢？这就涉及三元操作符类型的转换规则：

- ❑ 若两个操作数不可转换，则不做转换，返回值为 Object 类型。
- ❑ 若两个操作数是明确类型的表达式（比如变量），则按照正常的二进制数字来转换，int 类型转换为 long 类型，long 类型转换为 float 类型等。
- ❑ 若两个操作数中有一个是数字 S，另外一个表达式，且其类型标示为 T，那么，若数字 S 在 T 的范围内，则转换为 T 类型；若 S 超出了 T 类型的范围，则 T 转换为 S 类型（可以参考“建议 22”，会对该问题进行展开描述）。
- ❑ 若两个操作数都是直接量数字（Literal）<sup>⊖</sup>，则返回值类型为范围较大者。

知道是什么原因了，相应的解决办法也就有了：保证三元操作符中的两个操作数类型一致，即可减少可能错误的发生。

## 建议 4：避免带有变长参数的方法重载

在项目和系统的开发中，为了提高方法的灵活度和可复用性，我们经常要传递不确定数量的参数到方法中，在 Java 5 之前常用的设计技巧就是把形参定义成 Collection 类型或其子类类型，或者是数组类型，这种方法的缺点就是需要对空参数进行判断和筛选，比如实参为 null 值和长度为 0 的 Collection 或数组。而 Java 5 引入变长参数（varargs）就是为了更好地提高方法的复用性，让方法的调用者可以“随心所欲”地传递实参数量，当然变长参数也是要遵循一定规则的，比如变长参数必须是方法中的最后一个参数；一个方法不能定义多个变长参数等，这些基本规则需要牢记，但是即使记住了这些规则，仍然有可能出现错误，我们来看如下代码：

```
public class Client {
    // 简单折扣计算
    public void calPrice(int price,int discount){
        float knockdownPrice =price * discount / 100.0F;
        System.out.println(" 简单折扣后的价格是: "+formateCurrency(knockdownPrice));
    }
    // 复杂多折扣计算
    public void calPrice(int price,int... discounts){
        float knockdownPrice = price;
        for(int discount:discounts){
            knockdownPrice = knockdownPrice * discount / 100;
        }
    }
}
```

<sup>⊖</sup> “Literal” 也译作“字面量”。

```

    }
    System.out.println(" 复杂折扣后的价格是: " +formateCurrency(knockdownPrice));
}
// 格式化成本的货币形式
private String formateCurrency(float price){
    return NumberFormat.getCurrencyInstance().format(price/100);
}

public static void main(String[] args) {
    Client client = new Client();
    //499 元的货物, 打 75 折
    client.calPrice(49900, 75);
}
}

```

这是一个计算商品价格折扣的模拟类，带有两个参数的 calPrice 方法（该方法的业务逻辑是：提供商品的原价和折扣率，即可获得商品的折扣价）是一个简单的折扣计算方法，该方法在实际项目中经常会用到，这是单一的打折方法。而带有变长参数的 calPrice 方法则是较复杂的折扣计算方式，多种折扣的叠加运算（模拟类是一种比较简单的实现）在实际生活中也是经常见到的，比如在大甩卖期间对 VIP 会员再度进行打折；或者当天是你的生日，再给你打个 9 折，也就是俗话说的“折上折”。

业务逻辑清楚了，我们来仔细看看这两个方法，它们是重载吗？当然是了，重载的定义是“方法名相同，参数类型或数量不同”，很明显这两个方法是重载。但是再仔细瞧瞧，这个重载有点特殊：calPrice (int price,int... discounts) 的参数范畴覆盖了 calPrice (int price,int discount) 的参数范畴。那问题就出来了：对于 calPrice (49900,75) 这样的计算，到底该调用哪个方法来处理呢？

我们知道 Java 编译器是很聪明的，它在编译时会根据方法签名 (Method Signature) 来确定调用哪个方法，比如 calPrice (499900,75,95) 这个调用，很明显 75 和 95 会被转成一个包含两个元素的数组，并传递到 calPrice (int price,in.. discounts) 中，因为只有这一个方法签名符合该实参类型，这很容易理解。但是我们现在面对的是 calPrice (49900,75) 调用，这个“75”既可以被编译成 int 类型的“75”，也可以被编译成 int 数组“{75}”，即只包含一个元素的数组。那到底该调用哪一个方法呢？

我们先运行一下看看结果，运行结果是：

简单折扣后的价格是：¥374.25。

看来是调用了第一个方法，为什么会调用第一个方法，而不是第二个变长参数方法呢？因为 Java 在编译时，首先会根据实参的数量和类型（这里是 2 个实参，都为 int 类型，注意没有转成 int 数组）来进行处理，也就是查找到 calPrice(int price,int discount) 方法，而且确认它是否符合方法签名条件。现在的问题是编译器为什么会首先根据 2 个 int 类型的实参而不是 1 个 int 类型、1 个 int 数组类型的实参来查找方法呢？这是个好问题，也非常好回答：

因为 `int` 是一个原生数据类型，而数组本身是一个对象，编译器想要“偷懒”，于是它会从最简单的开始“猜想”，只要符合编译条件的即可通过，于是就出现了此问题。

问题是阐述清楚了，为了让我们的程序能被“人类”看懂，还是慎重考虑变长参数的方法重载吧，否则让人伤脑筋不说，说不定哪天就陷入这类小陷阱里了。

## 建议 5：别让 `null` 值和空值威胁到变长方法

上一建议讲解了变长参数的重载问题，本建议还会继续讨论变长参数的重载问题。上一建议的例子是变长参数的范围覆盖了非变长参数的范围，这次我们从两个都是变长参数的方法说起，代码如下：

```
public class Client {
    public void methodA(String str,Integer... is){
    }

    public void methodA(String str,String... str){
    }

    public static void main(String[] args) {
        Client client = new Client();
        client.methodA("China", 0);
        client.methodA("China", "People");
        client.methodA("China");
        client.methodA("China",null);
    }
}
```

两个 `methodA` 都进行了重载，现在的问题是：上面的代码编译通不过，问题出在什么地方？看似很简单哦。

有两处编译通不过：`client.methodA("China")` 和 `client.methodA("China",null)`，估计你已经猜到了，两处的提示是相同的：方法模糊不清，编译器不知道调用哪一个方法，但这两处代码反映的代码味道可是不同的。

对于 `methodA("China")` 方法，根据实参“China”（`String` 类型），两个方法都符合形参格式，编译器不知道该调用哪个方法，于是报错。我们来思考这个问题：Client 类是一个复杂的商业逻辑，提供了两个重载方法，从其他模块调用（系统内本地调用或系统外远程调用）时，调用者根据变长参数的规范调用，传入变长参数的实参数量可以是  $N$  个（ $N \geq 0$ ），那当然可以写成 `client.methodA("china")` 方法啊！完全符合规范，但是这却让编译器和调用者都很郁闷，程序符合规则却不能运行，如此问题，谁之责任呢？是 Client 类的设计者，他违反了 KISS 原则（Keep It Simple, Stupid，即懒人原则），按照此规则设计的方法应该很容易调用，可是现在在遵循规范的情况下，程序竟然出错了，这对设计者和开发者而言都是

应该严禁出现的。

对于 `client.methodA("china",null)` 方法，直接量 `null` 是没有类型的，虽然两个 `methodA` 方法都符合调用请求，但不知道调用哪一个，于是报错了。我们来体会一下它的坏味道：除了不符合上面的懒人原则外，这里还有一个非常不好的编码习惯，即调用者隐藏了实参类型，这是非常危险的，不仅仅调用者需要“猜测”该调用哪个方法，而且被调用者也可能产生内部逻辑混乱的情况。对于本例来说应该做如下修改：

```
public static void main(String[] args) {
    Client client = new Client();
    String[] strs = null;
    client.methodA("China",strs);
}
```

也就是说让编译器知道这个 `null` 值是 `String` 类型的，编译即可顺利通过，也就减少了错误的发生。

## 建议 6：覆写变长方法也循规蹈矩

在 Java 中，子类覆写父类中的方法很常见，这样做既可以修正 Bug 也可以提供扩展的业务功能支持，同时还符合开闭原则（Open-Closed Principle），我们来看一下覆写必须满足的条件：

- ❑ 重写方法不能缩小访问权限。
- ❑ 参数列表必须与被重写方法相同。
- ❑ 返回类型必须与被重写方法的相同或是其子类。
- ❑ 重写方法不能抛出新的异常，或者超出父类范围的异常，但是可以抛出更少、更有限的异常，或者不抛出异常。

估计你已经猜测出下面要讲的内容了，为什么“参数列表必须与被重写方法的相同”采用不同的字体，这其中是不是有什么玄机？是的，还真有那么一点点小玄机。参数列表相同包括三层意思：参数数量相同、类型相同、顺序相同，看上去好像没什么问题，那我们来看一个例子，业务场景与上一个建议相同，商品打折，代码如下：

```
public class Client {
    public static void main(String[] args) {
        // 向上转型
        Base base = new Sub();
        base.fun(100, 50);
        // 不转型
        Sub sub = new Sub();
        sub.fun(100, 50);
    }
}
```



```

    }
    // 基类
    class Base{
        void fun(int price,int... discounts){
            System.out.println("Base.....fun");
        }
    }

    // 子类，覆写父类方法
    class Sub extends Base{
        @Override
        void fun(int price,int[] discounts){
            System.out.println("Sub.....fun");
        }
    }

```

请问：该程序有问题吗？——编译通不过。那问题出在什么地方呢？

@Override 注解吗？非也，覆写是正确的，因为父类的 calPrice 编译成字节码后的形参是一个 int 类型的形参加上一个 int 数组类型的形参，子类的参数列表也与此相同，那覆写是理所当然的了，所以加上 @Override 注解没有问题，只是 Eclipse 会提示这不是一种很好的编码风格。

难道是“sub.fun(100, 50)”这条语句？正解，确实是这条语句报错，提示找不到 fun(int,int) 方法。这太奇怪了：子类继承了父类的所有属性和方法，甭管是私有的还是公开的访问权限，同样的参数、同样的方法名，通过父类调用没有任何问题，通过子类调用却编译通不过，为啥？难道是没继承下来？或者子类缩小了父类方法的前置条件？那如果是这样，就不应该覆写，@Override 就应该报错，真是奇妙的事情！

事实上，base 对象是把子类对象 Sub 做了向上转型，形参列表是由父类决定的，由于是变长参数，在编译时，“base.fun(100, 50)”中的“50”这个实参会被编译器“猜测”而编译成“{50}”数组，再由子类 Sub 执行。我们再来看看直接调用子类的情况，这时编译器并不会把“50”做类型转换，因为数组本身也是一个对象，编译器还没有聪明到要在两个没有继承关系的类之间做转换，要知道 Java 是要求严格的类型匹配的，类型不匹配编译器自然就会拒绝执行，并给予错误提示。

这是个特例，覆写的方法参数列表竟然与父类不相同，这违背了覆写的定义，并且会引发莫名其妙的错误。所以读者在对变长参数进行覆写时，如果要使用此类似的方法，请找个小黑屋仔细想想是不是一定要如此。

---

**注意** 覆写的方法参数与父类相同，不仅仅是类型、数量，还包括显示形式。

---

## 建议 7：警惕自增的陷阱

记得大学刚开始学 C 语言时，老师就说：自增有两种形式，分别是 i++ 和 ++i，i++ 表



示的是先赋值后加 1, ++i 是先加 1 后赋值, 这样理解了很多年也没出现问题, 直到遇到如下代码, 我才怀疑我的理解是不是错了:

```
public class Client {
    public static void main(String[] args) {
        int count = 0;
        for(int i=0;i<10;i++){
            count=count++;
        }
        System.out.println("count="+count);
    }
}
```

这个程序输出的 count 等于几? 是 count 自加 10 次吗? 答案等于 10? 可以非常肯定地告诉你, 答案错误! 运行结果是 count 等于 0。为什么呢?

count++ 是一个表达式, 是有返回值的, 它的返回值就是 count 自加前的值, Java 对自加是这样处理的: 首先把 count 的值 (注意是值, 不是引用) 拷贝到一个临时变量区, 然后对 count 变量加 1, 最后返回临时变量区的值。程序第一次循环时的详细处理步骤如下:

**步骤 1** JVM 把 count 值 (其值是 0) 拷贝到临时变量区。

**步骤 2** count 值加 1, 这时候 count 的值是 1。

**步骤 3** 返回临时变量区的值, 注意这个值是 0, 没修改过。

**步骤 4** 返回值赋值给 count, 此时 count 值被重置成 0。

“count=count++” 这条语句可以按照如下代码来理解:

```
public static int mockAdd(int count){
    // 先保存初始值
    int temp = count;
    // 做自增操作
    count = count+1;
    // 返回原始值
    return temp;
}
```

于是第一次循环后 count 的值还是 0, 其他 9 次的循环也是一样的, 最终你会发现 count 的值始终没有改变, 仍然保持着最初的状态。

此例中代码作者的本意是希望 count 自增, 所以想当然地认为赋值给自身就成了, 不曾想掉到 Java 自增的陷阱中了。解决方法很简单, 只要把 “count=count++” 修改为 “count++” 即可。该问题在不同的语言环境有不同的实现: C++ 中 “count=count++” 与 “count++” 是等效的, 而在 PHP 中则保持着与 Java 相同的处理方式。每种语言对自增的实现方式各不同, 读者有兴趣可以多找几种语言测试一下, 思考一下原理。

下次如果看到某人 T 恤上印着 “i=i++”, 千万不要鄙视他, 记住, 能够以不同的语言解释清楚这句话的人绝对不简单, 应该表现出 “如滔滔江水” 般的敬仰, 心理默念着 “高人,

绝世高人哪”。

## 建议 8：不要让旧语法困扰你

N 多年前接手了一个除了源码以外什么都没有的项目，没需求、没文档、没设计，原创者也已鸟兽散了，我们只能通过阅读源码来进行维护。期间，同事看到一段很“奇妙”的代码，让大家帮忙分析，代码片段如下：

```
public class Client {  
    public static void main(String[] args) {  
        // 数据定义及初始化  
        int fee=200;  
        // 其他业务处理  
        saveDefault:save(fee);  
        // 其他业务处理  
    }  
  
    static void saveDefault(){  
    }  
  
    static void save(int fee){  
    }  
}
```

该代码的业务含义是计算交易的手续费，最低手续费是 2 元，其业务逻辑大致看懂了，但是此代码非常神奇，“saveDefault:save (fee)” 这句代码在此处出现后，后续就再也没有与此有关的代码了，这做何解释呢？更神奇的是，编译竟然还没有错，运行也很正常。Java 中竟然有冒号操作符，一般情况下，它除了在唯一一个三元操作符中存在外就没有其他地方可用了呀。当时连项目组里的高手也是一愣一愣的，翻语法书，也没有介绍冒号操作符的内容，而且，也不可能出现连括号都可以省掉的方法调用、方法级联啊！这也太牛了吧！

隔壁做 C 项目的同事过来串门，看我们在讨论这个问题，很惊奇地说“耶，Java 中还有标号呀，我以为 Java 这么高级的语言已经抛弃 goto 语句了……”，一语点醒梦中人：项目的原创者是 C 语言转过来的开发人员，所以他把 C 语言的 goto 习惯也带到项目中了，后来由于经过 N 手交接，重构了多次，到我们这里 goto 语句已经被重构掉了，但是跳转标号还保留着，估计上一届的重构者也是稀里糊涂的，不敢贸然修改，所以把这个重任留给了我们。

goto 语句中有着“double face”作用的关键字，它可以让程序从多层的循环中跳出，不用一层一层地退出，类似高楼着火了，来不及一楼一楼的下，goto 语句就可以让你“biu~”的一声从十层楼跳到地面上。这点确实很好，但同时也带来了代码结构混乱的问题，而且程序跳来跳去让人看着就头晕，还怎么调试？！这样做甚至会隐祸连连，比如标号前后对象构造或变量初始化，一旦跳到这个标号，程序就不可想象了，所以 Java 中抛弃了 goto 语法，

但还是保留了该关键字，只是不进行语义处理而已，与此类似的还有 `const` 关键字。

Java 中虽然没有了 `goto` 关键字，但是扩展了 `break` 和 `continue` 关键字，它们的后面都可以加上标号做跳转，完全实现了 `goto` 功能，同时也把 `goto` 的诟病带了进来，所以我们在阅读大牛的开源程序时，根本就看不到 `break` 或 `continue` 后跟标号的情况，甚至是 `break` 和 `continue` 都很少看到，这是提高代码可读性的一剂良药，旧语法就让它随风而去吧！

## 建议 9：少用静态导入

从 Java 5 开始引入了静态导入语法（`import static`），其目的是为了减少字符输入量，提高代码的可阅读性，以便更好地理解程序。我们先来看一个不使用静态导入的例子，也就是一般导入：

```
public class MathUtils{
    // 计算圆面积
    public static double calCircleArea(double r){
        return Math.PI * r * r;
    }
    // 计算球面积
    public static double calBallArea(double r){
        return 4* Math.PI * r * r;
    }
}
```

这是很简单的数学工具类，我们在这两个计算面积的方法中都引入了 `java.lang.Math` 类（该类是默认导入的）中的 `PI`（圆周率）常量，而 `Math` 这个类写在这里有点多余，特别是如果 `MathUtils` 中的方法比较多时，如果每次都要敲入 `Math` 这个类，繁琐且多余，静态导入可解决此类问题，使用静态导入后的程序如下：

```
import static java.lang.Math.PI;
public class MathUtils{
    // 计算圆面积
    public static double calCircleArea(double r){
        return PI * r * r;
    }
    // 计算球面积
    public static double calBallArea(double r){
        return 4 * PI * r * r;
    }
}
```

静态导入的作用是把 `Math` 类中的 `PI` 常量引入到本类中，这会使程序更简单，更容易阅读，只要看到 `PI` 就知道这是圆周率，不用每次都要把类名写全了。但是，滥用静态导入会使程序更难阅读，更难维护。静态导入后，代码中就不用再写类名了，但是我们知道类是“一

类事物的描述”，缺少了类名的修饰，静态属性和静态方法的表象意义可以被无限放大，这会让阅读者很难弄清楚其属性或方法代表何意，甚至是哪一个类的属性（方法）都要思考一番（当然，IDE 友好提示功能是另说），特别是在一个类中有多个静态导入语句时，若还使用了\*（星号）通配符，把一个类的所有静态元素都导入进来了，那简直就是恶梦。我们来看一段例子：

```
import static java.lang.Double.*;
import static java.lang.Math.*;
import static java.lang.Integer.*;
import static java.text.NumberFormat.*;

public class Client {
    // 输入半径和精度要求，计算面积
    public static void main(String[] args) {
        double s = PI * parseDouble(args[0]);
        NumberFormat nf = getInstance();
        nf.setMaximumFractionDigits(parseInt(args[1]));
        formatMessage(nf.format(s));
    }
    // 格式化消息输出
    public static void formatMessage(String s){
        System.out.println(" 圆面积是: "+s);
    }
}
```

就这么一段程序，看着就让人火大：常量 PI，这知道，是圆周率；parseDouble 方法可能是 Double 类的一个转换方法，这看名称也能猜测到。那紧接着的 getInstance 方法是哪个类的？是 Client 本地类？不对呀，没有这个方法，哦，原来是 NumberFormate 类的方法，这和 formateMessage 本地方法没有任何区别了——这代码也太难阅读了，非机器不可阅读。

所以，对于静态导入，一定要遵循两个规则：

- ❑ 不使用\*（星号）通配符，除非是导入静态常量类（只包含常量的类或接口）。
- ❑ 方法名是具有明确、清晰表象意义的工具类。

何为具有明确、清晰表象意义的工具类？我们来看看 JUnit 4 中使用的静态导入的例子，代码如下：

```
import static org.junit.Assert.*;
public class DaoTest {
    @Test
    public void testInsert(){
        // 断言
        assertEquals("foo", "foo");
        assertFalse(Boolean.FALSE);
    }
}
```

我们从程序中很容易判断出 `assertEquals` 方法是用来断言两个值是否相等的，`assertFalse` 方法则是断言表达式为假，如此确实减少了代码量，而且代码的可读性也提高了，这也是静态导入用到正确地方所带来的好处。

## 建议 10：不要在本类中覆盖静态导入的变量和方法

如果一个类中的方法及属性与静态导入的方法及属性重名会出现什么问题呢？我们先来看一个正常的静态导入，代码如下：

```
import static java.lang.Math.PI;
import static java.lang.Math.abs;

public class Client {
    public static void main(String[] args) {
        System.out.println("PI="+PI);
        System.out.println("abs(100)=" +abs(-100));
    }
}
```

很简单的例子，打印出静态常量 `PI` 值，计算 `-100` 的绝对值。现在的问题是：如果我们在 `Client` 类中也定义了 `PI` 常量和 `abs` 方法，会出现什么问题？代码如下：

```
import static java.lang.Math.PI;
import static java.lang.Math.abs;

public class Client {
    // 常量名与静态导入的 PI 相同
    public final static String PI="祖冲之";
    // 方法名与静态导入的相同
    public static int abs(int abs){
        return 0;
    }

    public static void main(String[] args) {
        System.out.println("PI="+PI);
        System.out.println("abs(100)=" +abs(-100));
    }
}
```

以上代码中，定义了一个 `PI` 字符串类型的常量，又定义了一个 `abs` 方法，与静态导入的相同。首先说好消息：编译器没有报错，接下来是不好的消息了：我们不知道哪个属性和哪个方法被调用了，因为常量名和方法名相同，到底调用了哪一个方法呢？我们运行一下看看结果：

PI= 祖冲之

```
abs(100)=0
```

很明显是本地的属性和方法被引用了，为什么不是 Math 类中的属性和方法呢？那是因为编译器有一个“最短路径”原则：如果能够在本类中查找到的变量、常量、方法，就不会到其他包或父类、接口中查找，以确保本类中的属性、方法优先。

因此，如果要变更一个被静态导入的方法，最好的办法是在原始类中重构，而不是在本类中覆盖。

## 建议 11：养成良好习惯，显式声明 UID

我们编写一个实现了 Serializable 接口（序列化标志接口）的类，Eclipse 马上就会给一个黄色警告：需要增加一个 Serial Version ID。为什么要增加？它是怎么计算出来的？有什么用？本章就来解释该问题。

类实现 Serializable 接口的目的是为了可持久化，比如网络传输或本地存储，为系统的分布和异构部署提供先决支持条件。若没有序列化，现在我们熟悉的远程调用、对象数据库都不可能存在，我们来看一个简单的序列化类：

```
public class Person implements Serializable{
    private String name;
    /*name 属性的 getter/setter 方法省略 */
}
```

这是一个简单 JavaBean，实现了 Serializable 接口，可以在网络上传输，也可以本地存储然后读取。这里我们以 Java 消息服务（Java Message Service）方式传递该对象（即通过网络传递一个对象），定义在消息队列中的数据类型为 ObjectMessage，首先定义一个消息的生产者（Producer），代码如下：

```
public class Producer {
    public static void main(String[] args) throws Exception {
        Person person = new Person();
        person.setName(" 混世魔王 ");
        // 序列化，保存到磁盘上
        SerializationUtils.writeObject(person);
    }
}
```

这里引入了一个工具类 SerializationUtils，其作用是对一个类进行序列化和反序列化，并存储到硬盘上（模拟网络传输），其代码如下：

```
public class SerializationUtils {
    private static String FILE_NAME = "c:/obj.bin";
    // 序列化
    public static void writeObject(Serializable s) {
```



```

        try {
            ObjectOutputStream oos = new ObjectOutputStream(new
                FileOutputStream(FILE_NAME));
            oos.writeObject(s);
            oos.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public static Object readObject(){
        Object obj=null;
        // 反序列化
        try {
            ObjectInput input = new ObjectInputStream(new
                FileInputStream(FILE_NAME));
            obj = input.readObject();
            input.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
        return obj;
    }
}

```

通过对象序列化过程，把一个对象从内存块转化为可传输的数据流，然后通过网络发送到消息消费者（Consumer）那里，并进行反序列化，生成实例对象，代码如下：

```

public class Consumer {
    public static void main(String[] args) throws Exception {
        // 反序列化
        Person p = (Person) SerializationUtils.readObject();
        System.out.println("name="+p.getName());
    }
}

```

这是一个反序列化过程，也就是对象数据流转换为一个实例对象的过程，其运行后的输出结果为：混世魔王。这太 easy 了，是的，这就是序列化和反序列化典型的 demo。但此处隐藏着一个问题：如果消息的生产者和消息的消费者所参考的类（Person 类）有差异，会出现何种神奇事件？比如：消息生产者中的 Person 类增加了一个年龄属性，而消费者没有增加该属性。为啥没有增加？！因为这是个分布式部署的应用，你甚至都不知道这个应用部署在何处，特别是通过广播（broadcast）方式发送消息的情况，漏掉一两个订阅者也是很正常的。

在这种序列化和反序列化的类不一致的情形下，反序列化时会报一个 `InvalidClassException` 异常，原因是序列化和反序列化所对应的类版本发生了变化，JVM 不能把数据流转换为实例

对象。接着刨根问底：JVM 是根据什么来判断一个类版本的呢？

好问题，通过 `SerialVersionUID`，也叫做流标识符（Stream Unique Identifier），即类的版本定义的，它可以显式声明也可以隐式声明。显式声明格式如下：

```
private static final long serialVersionUID = XXXXXL;
```

而隐式声明则是不声明，你编译器在编译的时候帮我生成。生成的依据是通过包名、类名、继承关系、非私有的方法和属性，以及参数、返回值等诸多因子计算得出的，极度复杂，基本上计算出来的这个值是唯一的。

`serialVersionUID` 如何生成已经说明了，我们再来看看 `serialVersionUID` 的作用。JVM 在反序列化时，会比较数据流中的 `serialVersionUID` 与类的 `serialVersionUID` 是否相同，如果相同，则认为类没有发生改变，可以把数据流 load 为实例对象；如果不相同，对不起，我 JVM 不干了，抛个异常 `InvalidClassException` 给你瞧瞧。这是一个非常好的校验机制，可以保证一个对象即使在网络或磁盘中“滚过”一次，仍能做到“出淤泥而不染”，完美地实现类的一致性。

但是，有时候我们需要一点特例场景，例如：我的类改变不大，JVM 是否可以把以前的对象反序列化过来？就是依靠显式声明 `serialVersionUID`，向 JVM 撒谎说“我的类版本没有变更”，如此，我们编写的类就实现了向上兼容。我们修改一下上面的 `Person` 类，代码如下：

```
public class Person implements Serializable{
    private static final long serialVersionUID = 55799L;
    /* 其他保持不变 */
}
```

刚开始生产者和消费者持有的 `Person` 类版本一致，都是 V1.0，某天生产者的 `Person` 类版本变更了，增加了一个“年龄”属性，升级为 V2.0，而由于种种原因（比如程序员疏忽、升级时间窗口不同等）消费端的 `Person` 还保持为 V1.0 版本，代码如下：

```
public class Person implements Serializable{
    private static final long serialVersionUID = 5799L;
    private int age;
    /*age、name 的 getter/setter 方法省略 */
}
```

此时虽然生产者和消费者对应的类版本不同，但是显式声明的 `serialVersionUID` 相同，反序列化也是可以运行的，所带来的业务问题就是消费端不能读取到新增的业务属性（age 属性）而已。

通过此例，我们的反序列化实现了版本向上兼容的功能，使用 V1.0 版本的应用访问了一个 V2.0 版本的对象，这无疑提高了代码的健壮性。我们在编写序列化类代码时，随手加上 `serialVersionUID` 字段，也不会给我们带来太多的工作量，但它却可以在关键时刻发挥异

乎寻常的作用。

---

**注意** 显式声明 serialVersionUID 可以避免对象不一致，但尽量不要以这种方式向 JVM “撒谎”。

---

## 建议 12：避免用序列化类在构造函数中为不变量赋值

我们知道带有 final 标识的属性是不变量，也就是说只能赋值一次，不能重复赋值，但是在序列化类中就有点复杂了，比如有这样一个类：

```
public class Person implements Serializable{
    private static final long serialVersionUID = 71282334L;
    // 不变量
    public final String name=" 混世魔王 ";
}
```

这个 Person 类（此时 V1.0 版本）被序列化，然后存储在磁盘上，在反序列化时 name 属性会重新计算其值（这与 static 变量不同，static 变量压根就没有保存到数据流中），比如 name 属性修改成了“德天使”（版本升级为 V2.0），那么反序列化对象的 name 值就是“德天使”。保持新旧对象的 final 变量相同，有利于代码业务逻辑统一，这是序列化的基本规则之一，也就是说，如果 final 属性是一个直接量，在反序列化时就会重新计算。对这基本规则不多说，我们要说的是 final 变量另外一种赋值方式：通过构造函数赋值。代码如下：

```
public class Person implements Serializable{
    private static final long serialVersionUID = 91282334L;
    // 不变量初始不赋值
    public final String name;
    // 构造函数为不变量赋值
    public Person(){
        name=" 混世魔王 ";
    }
}
```

这也是我们常用的一种赋值方式，可以把这个 Person 类定义为版本 V1.0，然后进行序列化，看看有什么问题没有，序列化的代码如下所示：

```
public class Serialize {
    public static void main(String[] args) {
        // 序列化以持久保存
        SerializationUtils.writeObject(new Person());
    }
}
```

Person 的实例对象保存到了磁盘上，它是一个贫血对象（承载业务属性定义，但不包含其行为定义），我们做一个简单的模拟，修改一下 name 值代表变更，要注意的是

serialVersionUID 保持不变，修改后的代码如下：

```
public class Person implements Serializable{
    private static final long serialVersionUID = 91282334L;
    // 不变量初始不赋值
    public final String name;
    // 构造函数为不变量赋值
    public Person(){
        name=" 德天使 ";
    }
}
```

此时 Person 类的版本是 V2.0，但 serialVersionUID 没有改变，仍然可以反序列化，其代码如下：

```
public class Deserialize {
    public static void main(String[] args) {
        // 反序列化
        Person p = (Person)SerializationUtils.readObject();
        System.out.println(p.name);
    }
}
```

现在问题来了：打印的结果是什么？是混世魔王还是德天使？

答案即将揭晓，答案是：混世魔王。

final 类型的变量不是会重新计算吗？答案应该是“德天使”才对啊，为什么会是“混世魔王”？这是因为这里触及了反序列化的另一个规则：反序列化时构造函数不会执行。

反序列化的执行过程是这样的：JVM 从数据流中获取一个 Object 对象，然后根据数据流中的类文件描述信息（在序列化时，保存到磁盘的对象文件中包含了类描述信息，注意是类描述信息，不是类）查看，发现是 final 变量，需要重新计算，于是引用 Person 类中的 name 值，而此时 JVM 又发现 name 竟然没有赋值，不能引用，于是它很“聪明”地不再初始化，保持原值状态，所以结果就是“混世魔王”了。

读者不要以为这样的情况很少发生，如果使用 Java 开发过桌面应用，特别是参与过对性能要求较高的项目（比如交易类项目），那么很容易遇到这样的问题。比如一个 C/S 结构的在线外汇交易系统，要求提供 24 小时的联机服务，如果在升级的类中有一个 final 变量是构造函数赋值的，而且新旧版本还发生了变化，则在应用请求热切的过程中（非常短暂，可能只有 30 秒），很可能就会出现反序列化生成的 final 变量值与新产生的实例值不相同的情况，于是业务异常就产生了，情况严重的话甚至会影响交易数据，那可是天大的事故了。

---

**注意** 在序列化类中，不使用构造函数为 final 变量赋值。

---

## 建议 13: 避免为 final 变量复杂赋值

为 final 变量赋值还有一种方式：通过方法赋值，即直接在声明时通过方法返回值赋值。还是以 Person 类为例来说明，代码如下：

```
public class Person implements Serializable{
    private static final long serialVersionUID = 91282334L;
    // 通过方法返回值为 final 变量赋值
    public final String name=initName();
    // 初始化方法名
    public String initName(){
        return "混世魔王";
    }
}
```

name 属性是通过 initName 方法的返回值赋值的，这在复杂类中经常用到，这比使用构造函数赋值更简洁、易修改，那么如此用法在序列化时会不会有问题呢？我们一起来看看。Person 类写好了（定义为 V1.0 版本），先把它序列化，存储到本地文件，其代码与上一建议的 Serialize 类相同，不再赘述。

现在，Person 类的代码需要修改，initName 的返回值也改变了，代码如下：

```
public class Person implements Serializable{
    private static final long serialVersionUID = 91282334L;
    // 通过方法返回值为 final 变量赋值
    public final String name=initName();
    // 初始化方法名
    public String initName(){
        return "德天使";
    }
}
```

上段代码仅仅修改了 initName 的返回值（Person 类为 V2.0 版本），也就是说通过 new 生成的 Person 对象的 final 变量值都是“德天使”。那么我们把之前存储在磁盘上的实例加载上来，name 值会是什么呢？

结果是：混世魔王。很诧异，上一建议说过 final 变量会被重新赋值，但是这个例子又没有重新赋值，为什么？

上个建议所说 final 会被重新赋值，其中的“值”指的是简单对象。简单对象包括：8 个基本类型，以及数组、字符串（字符串情况很复杂，不通过 new 关键字生成 String 对象的情况下，final 变量的赋值与基本类型相同），但是不能方法赋值。

其中的原理是这样的，保存到磁盘上（或网络传输）的对象文件包括两部分：

（1）类描述信息

包括包路径、继承关系、访问权限、变量描述、变量访问权限、方法签名、返回值，以

及变量的关联类信息。要注意的一点是，它并不是 class 文件的翻版，它不记录方法、构造函数、static 变量等的具体实现。之所以类描述会被保存，很简单，是因为能去也能回嘛，这保证反序列化的健壮运行。

#### (2) 非瞬态 (transient 关键字) 和非静态 (static 关键字) 的实例变量值

注意，这里的值如果是一个基本类型，好说，就是一个简单值保存下来；如果是复杂对象，也简单，连该对象和关联类信息一起保存，并且持续递归下去（关联类也必须实现 Serializable 接口，否则会出现序列化异常），也就是说递归到最后，其实还是基本数据类型的保存。

正是因为这两点原因，一个持久化后的对象文件会比一个 class 类文件大很多，有兴趣的读者可以自己写个 Hello word 程序检验一下，其体积确实膨胀了不少。

总结一下，反序列化时 final 变量在以下情况下不会被重新赋值：

- ❑ 通过构造函数为 final 变量赋值。
- ❑ 通过方法返回值为 final 变量赋值。
- ❑ final 修饰的属性不是基本类型。

## 建议 14：使用序列化类的私有方法巧妙解决部分属性持久化问题

部分属性持久化问题看似很简单，只要把不需要持久化的属性加上瞬态关键字 (transient 关键字) 即可。这是一种解决方案，但有时候行不通。例如一个计税系统和人力资源系统 (HR 系统) 通过 RMI (Remote Method Invocation, 远程方法调用) 对接，计税系统需要从 HR 系统获得人员的姓名和基本工资，以作为纳税的依据，而 HR 系统的工资分为两部分：基本工资和绩效工资，基本工资没什么秘密，根据工作岗位和年限自己都可以计算出来，但绩效工资却是保密的，不能泄露到外系统，很明显这是两个相互关联的类。先来看薪水类 Salary 类的代码：

```
public class Salary implements Serializable{
    private static final long serialVersionUID = 44663L;
    // 基本工资
    private int basePay;
    // 绩效工资
    private int bonus;

    public Salary(int _basePay,int _bonus){
        basePay = _basePay;
        bonus = _bonus;
    }
    /*getter/setter 方法省略*/
}
```

Peron 类与 Salary 类是关联关系，代码如下：



```

public class Person implements Serializable{
    private static final long serialVersionUID =60407L;
    // 姓名
    private String name;
    // 薪水
    private Salary salary;

    public Person(String _name,Salary _salary){
        name=_name;
        salary=_salary;
    }
    /*getter/setter 方法省略*/
}

```

这是两个简单的 JavaBean，都实现了 Serializable 接口，都具备了持久化条件。首先计税系统请求 HR 系统对某一个 Person 对象进行序列化，把人员和工资信息传递到计税系统中，代码如下：

```

public class Serialize {
    public static void main(String[] args) {
        // 基本工资 1000 元，绩效工资 2500 元
        Salary salary = new Salary(1000,2500);
        // 记录人员信息
        Person person = new Person("张三",salary);
        //HR 系统持久化，并传递到计税系统
        SerializationUtils.writeObject(person);
    }
}

```

在通过网络传送到计税系统后，进行反序列化，代码如下：

```

public class Deserialize {
    public static void main(String[] args) {
        // 技术系统反序列化，并打印信息
        Person p = (Person)SerializationUtils.readObject();
        StringBuffer sb = new StringBuffer();
        sb.append("姓名：" + p.getName());
        sb.append("\t基本工资：" + p.getSalary().getBasePay());
        sb.append("\t绩效工资：" + p.getSalary().getBonus());
        System.out.println(sb);
    }
}

```

打印出的结果很简单：

姓名：张三          基本工资：1000          绩效工资：2500。

但是这不符合需求，因为计税系统只能从 HR 系统中获得人员姓名和基本工资，而绩效工资是不能获得的，这是个保密数据，不允许发生泄露。怎么解决这个问题呢？你可能马上

会想到四种方案：

(1) 在 bonus 前加上 transient 关键字

这是一个方法，但不是一个好方法，加上 transient 关键字就标志着 Salary 类失去了分布式部署的功能，它可是 HR 系统最核心的类了，一旦遭遇性能瓶颈，想再实现分布式部署就不可能了，此方案否定。

(2) 新增业务对象

增加一个 Person4Tax 类，完全为计税系统服务，就是说它只有两个属性：姓名和基本工资。符合开闭原则，而且对原系统也没有侵入性，只是增加了工作量而已。这是个方法，但不是最优方法。

(3) 请求端过滤

在计税系统获得 Person 对象后，过滤掉 Salary 的 bonus 属性，方案可行但不合规矩，因为 HR 系统中的 Salary 类安全性竟然让外系统（计税系统）来承担，设计严重失职。

(4) 变更传输契约

例如改用 XML 传输，或者重建一个 Web Service 服务。可以做，但成本太高。

可能有读者会说了，你都在说别人的方案不好，你提供个优秀的方案看看！好的，这就展示一个优秀的方案。其中，实现了 Serializable 接口的类可以实现两个私有方法：writeObject 和 readObject，以影响和控制序列化和反序列化的过程。我们把 Person 类稍做修改，看看如何控制序列化和反序列化，代码如下：

```
public class Person implements Serializable{
    private static final long serialVersionUID =60407L;
    // 姓名
    private String name;
    // 薪水
    private transient Salary salary;

    public Person(String _name,Salary _salary){
        name=_name;
        salary=_salary;
    }
    // 序列化委托方法
    private void writeObject(java.io.ObjectOutputStream out) throws IOException {
        out.defaultWriteObject();
        out.writeInt(salary.getBasePay());
    }
    // 反序列化时委托方法
    private void readObject(java.io.ObjectInputStream in) throws IOException,Class-
        NotFoundException {
        in.defaultReadObject();
        salary = new Salary(in.readInt(),0);
    }
}
```

其他代码不做任何改动，我们先运行看看，结果为：

姓名：张三          基本工资：1000          绩效工资：0。

我们在 Person 类中增加了 writeObject 和 readObject 两个方法，并且访问权限都是私有级别，为什么这会改变程序的运行结果呢？其实这里使用了序列化独有的机制：序列化回调。Java 调用 ObjectOutputStream 类把一个对象转换成流数据时，会通过反射（Reflection）检查被序列化的类是否有 writeObject 方法，并且检查其是否符合私有、无返回值的特性。若有，则会委托该方法进行对象序列化，若没有，则由 ObjectOutputStream 按照默认规则继续序列化。同样，在从流数据恢复成实例对象时，也会检查是否有一个私有的 readObject 方法，如果有，则会通过该方法读取属性值。此处有几个关键点要说明：

(1) out.defaultWriteObject()

告知 JVM 按照默认的规则写入对象，惯例的写法是写在第一句话里。

(2) in.defaultReadObject()

告知 JVM 按照默认规则读入对象，惯例的写法也是写在第一句话里。

(3) out.writeXX 和 in.readXX

分别是写入和读出相应的值，类似一个队列，先进先出，如果此处有复杂的数据逻辑，建议按封装 Collection 对象处理。

可能有读者会提出，这似乎不是一种优雅的处理方案呀，为什么 JDK 没有对此提供一个更好的解决办法呢？比如访问者模式，或者设置钩子函数（Hook），完全可以更优雅地解决此类问题。我查阅了大量的文档，得出的结论是：无解，只能说这是一个可行的解决方案而已。

再回到我们的业务领域，通过上述方法重构后，其代码的修改量减少了许多，也优雅了许多。可能你又要反问了：如此一来，Person 类也失去了分布式部署的能力啊。确实是，但是 HR 系统的难点和重点是薪水计算，特别是绩效工资，它所依赖的参数很复杂（仅从数量上说就有上百甚至上千种），计算公式也不简单（一般是引入脚本语言，个性化公式定制），而相对来说 Person 类基本上都是“静态”属性，计算的可能性不大，所以即使为性能考虑，Person 类为分布式部署的意义也不大。

## 建议 15: break 万万不可忘

我们经常会写一些转换类，比如货币转换、日期转换、编码转换等，在金融领域里用到最多的要数中文数字转换了，比如把“1”转换为“壹”，不过，开源世界是不会提供此工具类的，因为它太贴合中国文化了，要转换还是得自己动手写，代码片段如下：

```
public class Client {  
    public static void main(String[] args) {  
        System.out.println("2 = "+toChineseNumberCase(2));  
    }  
  
    // 把阿拉伯数字翻译成中文大写数字  
    public static String toChineseNumberCase(int n) {  
        String chineseNumber = "";  
        switch (n) {  
            case 0:chineseNumber = "零";  
            case 1:chineseNumber = "壹";  
            case 2:chineseNumber = "贰";  
            case 3:chineseNumber = "叁";  
            case 4:chineseNumber = "肆";  
            case 5:chineseNumber = "伍";  
            case 6:chineseNumber = "陆";  
            case 7:chineseNumber = "柒";  
            case 8:chineseNumber = "捌";  
            case 9:chineseNumber = "玖";  
        }  
        return chineseNumber;  
    }  
}
```

这是一个简单的转换类，并没有完整实现，只是一个金融项目片段。如此简单的代码应该不会有错吧，我们运行看看，结果是：2 = 玖。

恩？错了？回头再来看程序，马上醒悟了：每个 case 语句后面少加了 break 关键字。程序从“case 2”后面的语句开始执行，直到找到最近的 break 语句结束，但可惜的是我们的程序中没有 break 语句，于是在程序执行的过程中，chineseNumber 的赋值语句会多次执行，会等于“贰”、等于“叁”、等于“肆”，一直变换到等于“玖”，switch 语句执行结束了，于是结果也就如此了。

此类问题发生得非常频繁，但也很容易发现，只要做一下单元测试（Unit Test），问题立刻就会被发现并解决掉，但如果是在一堆的 case 语句中，其中某一条漏掉了 break 关键字，特别是在单元测试覆盖率不够高的时候（为什么不够高？在大点的项目中蹲过坑、打过仗的兄弟们可能都知道，项目质量是与项目工期息息相关的，而项目工期往往不是由项目人员决定的，所以如果一个项目的单元测试覆盖率能够达到 60%，你就可以笑了），也就是说分支条件可能覆盖不到的时候，那就会在生产中出现大事故了。

我曾遇到过一个类似的故事，那是开发一个通过会员等级决定相关费率的系统，由于会员等级有 100 多个，所以测试时就采用了抽样测试的方法，测试时一切顺利，直到系统上线后，财务报表系统发现一个小概率的会员费率竟然出奇的低，于是就跟踪分析，发现是少了一个 break，此事不仅造成甲方经济上的损失，而且在外部也产生了不良的影响，最后该代码的作者被辞退了，测试人员、质量负责人、项目经理都做了相应的处罚。希望读者能引以

为戒，记住在 case 语句后面随手写上 break，养成良好的习惯。

对于此类问题，还有一个最简单的解决办法：修改 IDE 的警告级别，例如在 Eclipse 中，可以依次点击 Performances → Java → Compiler → Errors/Warnings → Potential Programming problems，然后修改 ‘switch’ case fall-through 为 Errors 级别，如果你胆敢不在 case 语句中加入 break，那 Eclipse 直接就报个红叉给你看，这样就可以完全避免该问题的发生了。

## 建议 16：易变业务使用脚本语言编写

Java 世界一直在遭受着异种语言的入侵，比如 PHP、Ruby、Groovy、JavaScript 等，这些“入侵者”都有一个共同特征：全是同一类语言——脚本语言，它们都是在运行期解释执行的。为什么 Java 这种强编译型语言会需要这些脚本语言呢？那是因为脚本语言的三大特征，如下所示：

- ❑ 灵活。脚本语言一般都是动态类型，可以不用声明变量类型而直接使用，也可以在运行期改变类型。
- ❑ 便捷。脚本语言是一种解释型语言，不需要编译成二进制代码，也不需要像 Java 一样生成字节码。它的执行是依靠解释器解释的，因此在运行期变更代码非常容易，而且不用停止应用。
- ❑ 简单。只能说部分脚本语言简单，比如 Groovy，Java 程序员若转到 Groovy 程序语言上，只需要两个小时，看完语法说明，看完 Demo 即可使用了，没有太多的技术门槛。

脚本语言的这些特性是 Java 所缺少的，引入脚本语言可以使 Java 更强大，于是 Java 6 开始正式支持脚本语言。但是因为脚本语言比较多，Java 的开发者也很难确定该支持哪种语言，于是 JCP（Java Community Process）很聪明地提出了 JSR223 规范，只要符合该规范的语言都可以在 Java 平台上运行（它对 JavaScript 是默认支持的），诸位读者有兴趣的话可以自己写个脚本语言，然后再实现 ScriptEngine，即可在 Java 平台上运行。

我们来分析一个案例，展现一下脚本语言是如何实现“拥抱变化”的。咱们编写一套模型计算公式，预测下一个工作日的股票走势（如果真有，那巴菲特就羞愧死了），即把国家政策、汇率、利率、地域系数等参数输入到公式中，然后计算出明天这支股票是涨还是跌，该公式是依靠历史数据推断而来的，会根据市场环境逐渐优化调整，也就是逐渐趋向“真理”的过程，在此过程中，公式经常需要修改（这里的修改不仅仅是参数修改，还涉及公式的算法修改），如果把这个公式写到一个类中（或者几个类中），就需要经常发布重启等操作（比如业务中断，需要冒烟测试（Smoke Testing）等），使用脚本语言则可以很好地简化这一过程，我们写一个简单公式来模拟一下，代码如下：

```
function formula(var1,var2){
    return var1 + var2 * factor;
}
```

这就是一个简单的脚本语言函数，可能你会很疑惑：factor（因子）这个变量是从哪儿来的？它是从上下文来的，类似于一个运行的环境变量。该 JavaScript 保存在 C:/model.js 中。下一步 Java 需要调用 JavaScript 公式，代码如下：

```
public static void main(String[] args) throws Exception {
    // 获得一个 JavaScript 的执行引擎
    ScriptEngine engine=new ScriptEngineManager().getEngineByName("javascript");
    // 建立上下文变量
    Bindings bind=engine.createBindings();
    bind.put("factor", 1);
    // 绑定上下文，作用域是当前引擎范围
    engine.setBindings(bind,ScriptContext.ENGINE_SCOPE);
    Scanner input = new Scanner(System.in);
    while(input.hasNextInt()){
        int first = input.nextInt();
        int sec = input.nextInt();
        System.out.println(" 输入参数是: "+first+", "+sec);
        // 执行 js 代码
        engine.eval(new FileReader("c:/model.js"));
        // 是否可调用方法
        if(engine instanceof Invocable){
            Invocable in=(Invocable)engine;
            // 执行 js 中的函数
            Double result = (Double)in.invokeFunction("formula",first,sec);
            System.out.println(" 运算结果: "+result.intValue());
        }
    }
}
```

上段代码使用 Scanner 类接受键盘输入的两个数字，然后调用 JavaScript 脚本的 formula 函数计算其结果，注意，除非输入了一个非 int 数字，否则当前 JVM 会一直运行，这也是模拟生产系统的在线变更状况。运行结果如下：

```
输入参数是: 1,2
运算结果: 3
```

此时，保持 JVM 的运行状态，我们修改一下 formula 函数，代码如下：

```
function formula(var1,var2){
    return var1 + var2 - factor;
}
```

其中，乘号变成了减号，计算公式发生了重大改变。回到 JVM 中继续输入，运行结果如下。



输入参数是: 1,2

运算结果: 2

修改 Java 代码, JVM 没有重启, 输入参数也没有任何改变, 仅仅改变脚本函数即可产生不同的结果。这就是脚本语言对系统设计最有利的地方: 可以随时发布而不用重新部署; 这也是我们 Javaer 最喜爱它的地方——即使进行变更, 也能提供不间断的业务服务。

Java 6 不仅仅提供了代码级的脚本内置, 还提供了一个 jrunscript 命令工具, 它可以在批处理中发挥最大效能, 而且不需要通过 JVM 解释脚本语言, 可以直接通过该工具运行脚本。想想看, 这是多么大的诱惑力呀! 而且这个工具是可以跨操作系统的, 脚本移植就更容易了。但是有一点需要注意: 该工具是实验性的, 在以后的 JDK 中会不会继续提供就很难说了。

## 建议 17: 慎用动态编译

动态编译一直是 Java 的梦想, 从 Java 6 版本它开始支持动态编译了, 可以在运行期直接编译 .java 文件, 执行 .class, 并且能够获得相关的输入输出, 甚至还能监听相关的事件。不过, 我们最期望的还是给定一段代码, 直接编译, 然后运行, 也就是空中编译执行 (on-the-fly), 来看如下代码:

```
public class Client {
    public static void main(String[] args) throws Exception {
        //Java 源代码
        String sourceStr = "public class Hello{public String sayHello (String name)
            {return \"Hello,\" + name + \"!\";}}";
        // 类名及文件名
        String clsName = "Hello";
        // 方法名
        String methodName = "sayHello";
        // 当前编译器
        JavaCompiler cmp = ToolProvider.getSystemJavaCompiler();
        //Java 标准文件管理器
        StandardJavaFileManager fm = cmp.getStandardFileManager(null,null,null);
        //Java 文件对象
        JavaFileObject jfo = new StringJavaObject(clsName,sourceStr);
        // 编译参数, 类似于 javac <options> 中的 options
        List<String> optionsList = new ArrayList<String>();
        // 编译文件的存放地方, 注意: 此处是为 Eclipse 工具特设的
        optionsList.addAll(Arrays.asList("-d", "./bin"));
        // 要编译的单元
        List<JavaFileObject> jfos = Arrays.asList(jfo);
        // 设置编译环境
        JavaCompiler.CompilationTask task = cmp.getTask(null, fm, null,
            optionsList,null,jfos);
        // 编译成功
        if(task.call()){
```

```

        // 生成对象
        Object obj = Class.forName(clsName).newInstance();
        Class<? extends Object> cls = obj.getClass();
        // 调用 sayHello 方法
        Method m = cls.getMethod(methodName, String.class);
        String str = (String) m.invoke(obj, "Dynamic Compilation");
        System.out.println(str);
    }
}
// 文本中的 Java 对象
class StringJavaObject extends SimpleJavaFileObject{
    // 源代码
    private String content = "";
    // 遵循 Java 规范的类名及文件
    public StringJavaObject(String _javaFileName, String _content){
        super(_createStringJavaObjectUri(_javaFileName), Kind.SOURCE);
        content = _content;
    }
    // 产生一个 URL 资源路径
    private static URI _createStringJavaObjectUri(String name){
        // 注意此处没有设置包名
        return URI.create("String:/// " + name + Kind.SOURCE.extension);
    }
    // 文本文件代码
    @Override
    public CharSequence getCharContent(boolean ignoreEncodingErrors)
        throws IOException {
        return content;
    }
}

```

上面的代码较多，这是一个动态编译的模板程序，读者可以拷贝到项目中使用，代码中的中文注释也较多，相信读者看得懂，不多解释，读者只要明白一件事：只要是在本地静态编译能够实现的任务，比如编译参数、输入输出、错误监控等，动态编译就都能实现。

Java 的动态编译对源提供了多个渠道。比如，可以是字符串（例子中就是字符串），可以是文本文件，也可以是编译过的字节码文件（.class 文件），甚至可以是存放在数据库中的明文代码或是字节码。汇总成一句话，只要是符合 Java 规范的就都可以在运行期动态加载，其实现方式就是实现 JavaFileObject 接口，重写 getCharContent、openInputStream、openOutputStream，或者实现 JDK 已经提供的两个 SimpleJavaFileObject、ForwardingJavaFileObject，具体代码可以参考上个例子。

动态编译虽然是很好的工具，让我们可以更加自如地控制编译过程，但是在我目前所接触的项目中还是使用得较少。原因很简单，静态编译已经能够帮我们处理大部分的工作，甚至是全部的工作，即使真的需要动态编译，也有很好的替代方案，比如 JRuby、Groovy 等无缝的脚本语言。

另外，我们在使用动态编译时，需要注意以下几点：

(1) 在框架中谨慎使用

比如要在 Struts 中使用动态编译，动态实现一个类，它若继承自 ActionSupport 就希望它成为一个 Action。能做到，但是 debug 很困难；再比如在 Spring 中，写一个动态类，要让它动态注入到 Spring 容器中，这是需要花费老大功夫的。

(2) 不要在要求高性能的项目使用

动态编译毕竟需要一个编译过程，与静态编译相比多了一个执行环节，因此在高性能项目中不要使用动态编译。不过，如果是在工具类项目中它则可以很好地发挥其优越性，比如在 Eclipse 工具中写一个插件，就可以很好地使用动态编译，不用重启即可实现运行、调试功能，非常方便。

(3) 动态编译要考虑安全问题

如果你在 Web 界面上提供了一个功能，允许上传一个 Java 文件然后运行，那就等于说：“我的机器没有密码，大家都来看我的隐私吧”，这是非常典型的注入漏洞，只要上传一个恶意 Java 程序就可以让你所有的安全工作毁于一旦。

(4) 记录动态编译过程

建议记录源文件、目标文件、编译过程、执行过程等日志，不仅仅是为了诊断，还是为了安全和审计，对 Java 项目来说，空中编译和运行是很不让人放心的，留下这些依据可以更好地优化程序。

## 建议 18：避免 instanceof 非预期结果

instanceof 是一个简单的二元操作符，它是用来判断一个对象是否是一个类实例的，其操作类似于  $\geq$ 、 $=$ ，非常简单，我们来看段程序，代码如下：

```
public class Client {
    public static void main(String[] args) {
        //String 对象是否是 Object 的实例
        boolean b1 = "Sting" instanceof Object;
        //String 对象是否是 String 的实例
        boolean b2 = new String() instanceof String;
        //Object 对象是否是 String 的实例
        boolean b3 = new Object() instanceof String;
        // 拆箱类型是否是装箱类型的实例
        boolean b4 = 'A' instanceof Character;
        // 空对象是否是 String 的实例
        boolean b5 = null instanceof String;
        // 类型转换后的空对象是否是 String 的实例
        boolean b6 = (String)null instanceof String;
        //Date 对象是否是 String 的实例
```

```

        boolean b7 = new Date() instanceof String;
        // 在泛型类中判断 String 对象是否是 Date 的实例
        boolean b8 = new GenericClass<String>().isDateInstance("");
    }
}

class GenericClass<T>{
    // 判断是否是 Date 类型
    public boolean isDateInstance(T t){
        return t instanceof Date;
    }
}

```

就这么一段程序，instanceof 的所有应用场景都出现了，同时问题也产生了：这段程序中哪些语句会编译通不过？我们一个一个地来解说。

#### ❑ "String" instanceof Object

返回值是 true，这很正常，“String”是一个字符串，字符串又继承了 Object，那当然是返回 true 了。

#### ❑ new String() instanceof String

返回值是 true，没有任何问题，一个类的对象当然是它的实例了。

#### ❑ new Object() instanceof String

返回值是 false，Object 是父类，其对象当然不是 String 类的实例了。要注意的是，这句话其实完全可以编译通过，只要 instanceof 关键字的左右两个操作数有继承或实现关系，就可以编译通过。

#### ❑ 'A' instanceof Character

这句话可能有读者会猜错，事实上它编译不通过，为什么呢？因为 'A' 是一个 char 类型，也就是一个基本类型，不是一个对象，instanceof 只能用于对象的判断，不能用于基本类型的判断。

#### ❑ null instanceof String

返回值是 false，这是 instanceof 特有的规则：若左操作数是 null，结果就直接返回 false，不再运算右操作数是什么类。这对我们的程序非常有利，在使用 instanceof 操作符时，不用关心被判断的类（也就是左操作数）是否为 null，这与我们经常用到的 equals、toString 方法不同。

#### ❑ (String)null instanceof String

返回值是 false，不要看这里有个强制类型转换就认为结果是 true，不是的，null 是一个万用类型，也可以说它没类型，即使做类型转换还是个 null。

#### ❑ new Date() instanceof String

编译通不过，因为 Date 类和 String 没有继承或实现关系，所以在编译时直接就报错了，

instanceof 操作符的左右操作数必须有继承或实现关系，否则编译会失败。

```
❏ new GenericClass<String>().isDateInstance("")
```

编译通不过？非也，编译通过了，返回值是 false，T 是个 String 类型，与 Date 之间没有继承或实现关系，为什么 "t instanceof Date" 会编译通过呢？那是因为 Java 的泛型是为编码服务的，在编译成字节码时，T 已经是 Object 类型了，传递的实参是 String 类型，也就是说 T 的表面类型是 Object，实际类型是 String，那 "t instanceof Date" 这句话就等价于 "Object instanceof Date" 了，所以返回 false 就很正常了。

就这么一个简单的 instanceof，你答对几个？

## 建议 19：断言绝对不是鸡肋

在防御式编程中经常会用断言（Assertion）对参数和环境做出判断，避免程序因不当的输入或错误的环境而产生逻辑异常，断言在很多语言中都存在，C、C++、Python 都有不同的断言表示形式。在 Java 中的断言使用的是 assert 关键字，其基本的用法如下：

```
assert <布尔表达式>
assert <布尔表达式> : <错误信息>
```

在布尔表达式为假时，抛出 AssertionError 错误，并附带了错误信息。assert 的语法较简单，有以下两个特性：

（1）assert 默认是不启用的

我们知道断言是为调试程序服务的，目的是为了能够快速、方便地检查到程序异常，但 Java 在默认条件下是不启用的，要启用就需要在编译、运行时加上相关的关键字，这就不多说，有需要的话可以参考一下 Java 规范。

（2）assert 抛出的异常 AssertionError 是继承自 Error 的

断言失败后，JVM 会抛出一个 AssertionError 错误，它继承自 Error，注意，这是一个错误，是不可恢复的，也就表示这是一个严重问题，开发者必须予以关注并解决之。

assert 虽然是做断言的，但不能将其等价于 if...else...这样的条件判断，它在以下两种情况不可使用：

（1）在对外公开的方法中

我们知道防御式编程最核心的一点就是：所有的外部因素（输入参数、环境变量、上下文）都是“邪恶”的，都存在着企图摧毁程序的罪恶本源，为了抵制它，我们要在程序中处处检验，满地设卡，不满足条件就不再执行后续程序，以保护主程序的正确性，处处设卡没问题，但就是不能用断言做输入校验，特别是公开方法。我们来看一个例子：

```
public class Client {
```

```

    public static void main(String[] args) {
        StringUtils.encode(null);
    }
}
// 字符串处理工具类
class StringUtils{
    public static String encode(String str){
        assert str!=null:"加密的字符串为 null";
        /* 加密处理 */
    }
}

```

encode 方法对输入参数做了不为空的假设，如果为空，则抛出 AssertionError 错误，但这段程序存在一个严重的问题，encode 是一个 public 方法，这标志着是它对外公开的，任何一个类只要能够传递一个 String 类型的参数（遵守契约）就可以调用，但是 Client 类按照规范和契约调用 encode 方法，却获得了一个 AssertionError 错误信息，是谁破坏了契约协定？——是 encode 方法自己。

#### (2) 在执行逻辑代码的情况下

assert 的支持是可选的，在开发时可以让它运行，但在生产系统中则不需要其运行了（以便提高性能），因此在 assert 的布尔表达式中不能执行逻辑代码，否则会因为环境不同而产生不同的逻辑，例如：

```

public void doSomething(List list, Object element){
    assert list.remove(element):"删除元素 " + element + " 失败";
    /* 业务处理 */
}

```

这段代码在 assert 启用的环境下，没有任何问题，但是一旦投入到生产环境，就不会启用断言了，而这个方法也就彻底完蛋了，list 的删除动作永远都不会执行，所以也就永远不会报错或异常，因为根本就没有执行嘛！

以上两种情况下不能使用 assert，那在什么情况下能够使用 assert 呢？一句话：按照正常执行逻辑不可能到达的代码区域可以放置 assert。具体分为三种情况：

#### (1) 在私有方法中放置 assert 作为输入参数的校验

在私有方法中可以放置 assert 校验输入参数，因为私有方法的使用者是作者自己，私有方法的调用者和被调用者之间是一种弱契约关系，或者说没有契约关系，其间的约束是依靠作者自己控制的，因此加上 assert 可以更好地预防自己犯错，或者无意的程序犯错。

#### (2) 流程控制中不可能达到的区域

这类似于 JUnit 的 fail 方法，其标志性的意义就是：程序执行到这里就是错误的，例如：

```

public void doSomething(){
    int i = 7;
    while(i > 7){

```



```

        /* 业务处理 */
    }
    assert false: " 到达这里就表示错误 ";
}

```

### (3) 建立程序探针

我们可能会在一段程序中定义两个变量，分别代表两个不同的业务含义，但是两者有固定的关系，例如 `var1=var2*2`，那我们就可以在程序中到处设“桩”，断言这两者的关系，如果不满足即表明程序已经出现了异常，业务也就没有必要运行下去了。

## 建议 20：不要只替换一个类

我们经常在系统中定义一个常量接口（或常量类），以囊括系统中所涉及的常量，从而简化代码，方便开发，在很多的开源项目中已采用了类似的方法，比如在 Struts2 中，`org.apache.struts2.StrutsConstants` 就是一个常量类，它定义了 Struts 框架中与配置有关的常量，而 `org.apache.struts2.StrutsStatics` 则是一个常量接口，其中定义了 OGNL 访问的关键字。

关于常量接口（类）我们来看一个例子，首先定义一个常量类：

```

public class Constant {
    // 定义人类寿命极限
    public final static int MAX_AGE = 150;
}

```

这是一个非常简单的常量类，定义了人类的最大年龄，我们引用这个常量，代码如下：

```

public class Client {
    public static void main(String[] args) {
        System.out.println("人类寿命极限是: " + Constant.MAX_AGE);
    }
}

```

运行的结果非常简单（结果省略）。目前的代码编写都是在“智能型”IDE 工具中完成的，下面我们暂时回溯到原始时代，也就是回归到用记事本编写代码的年代，然后看看会发生什么奇妙事情（为什么要如此，稍后会给出答案）。

修改常量 Constant 类，人类的寿命增加了，最大能活到 180 岁，代码如下：

```

public class Constant {
    // 定义人类寿命极限
    public final static int MAX_AGE = 180;
}

```

然后重新编译：`javac Constant`，编译完成后执行：`java Client`，大家想看看输出的极限年龄是多少岁吗？

输出的结果是：“人类寿命极限是：150”，竟然没有改变为 180，太奇怪了，这是为何？

原因是：对于 `final` 修饰的基本类型和 `String` 类型，编译器会认为它是稳定态（Immutable Status），所以在编译时就直接把值编译到字节码中了，避免了在运行期引用（Run-time Reference），以提高代码的执行效率。针对我们的例子来说，`Client` 类在编译时，字节码中就写上了“150”这个常量，而不是一个地址引用，因此无论你后续怎么修改常量类，只要不重新编译 `Client` 类，输出还是照旧。

而对于 `final` 修饰的类（即非基本类型），编译器认为它是不稳定态（Mutable Status），在编译时建立的则是引用关系（该类型也叫做 Soft Final），如果 `Client` 类引入的常量是一个类或实例，即使不重新编译也会输出最新值。

千万不可小看了这点知识，细坑也能绊倒大象，比如在一个 Web 项目中，开发人员修改一个 `final` 类型的值（基本类型），考虑到重新发布风险较大，或者是时间较长，或者是审批流程过于繁琐，反正是为了偷懒，于是直接采用替换 `class` 类文件的方式发布。替换完毕后应用服务器自动重启，然后简单测试一下（比如本类引用 `final` 类型的常量），一切 OK。可运行几天后发现业务数据对不上，有的类（引用关系的类）使用了旧值，有的类（继承关系的类）使用的是新值，而且毫无头绪，让人一筹莫展，其实问题的根源就在于此。

恩，还有个问题没有说明，我们的例子为什么不在 IDE 工具（比如 Eclipse）中运行呢？那是因为在 IDE 中不能重现该问题，若修改了 `Constant` 类，IDE 工具会自动编译所有的引用类，“智能”化屏蔽了该问题，但潜在的风险其实仍然存在。

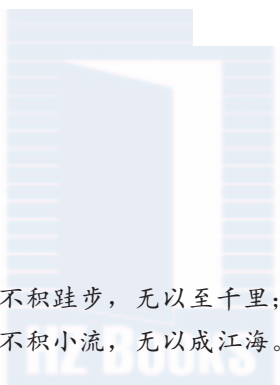
---

**注意** 发布应用系统时禁止使用类文件替换方式，整体 WAR 包发布才是万全之策。

---



## 第2章 基本类型



不积跬步，无以至千里；  
不积小流，无以成江海。

——荀子《劝学篇》

华章图书

Java 中的基本数据类型（Primitive Data Types）有 8 个：byte、char、short、int、long、float、double、boolean，它们是 Java 最基本的单元，我们的每一段程序中都有它们的身影，但我们对如此熟悉的“伙伴”又了解多少呢？

积少成多，积土成山，本章我们就来一探这最基本的 8 个数据类型。

## 建议 21：用偶判断，不用奇判断

判断一个数是奇数还是偶数是小学里学的基本知识，能够被 2 整除的整数是偶数，不能被 2 整除的是奇数，这规则简单又明了，还有什么好考虑的？好，我们来看一个例子，代码如下：

```
public class Client {
    public static void main(String[] args) {
        // 接收键盘输入参数
        Scanner input = new Scanner(System.in);
        System.out.print(" 请输入多个数字判断奇偶: ");
        while(input.hasNextInt()){
            int i = input.nextInt();
            String str = i+ "->" + (i%2 ==1?" 奇数 ":" 偶数 ");
            System.out.println(str);
        }
    }
}
```

输入多个数字，然后判断每个数字的奇偶性，不能被 2 整除就是奇数，其他的都是偶数，完全是根据奇偶数的定义编写的程序，我们来看看打印的结果：

```
请输入多个数字判断奇偶: 1 2 0 -1 -2
1-> 奇数
2-> 偶数
0-> 偶数
-1-> 偶数
-2-> 偶数
```

前三个还很靠谱，第四个参数 -1 怎么可能会是偶数呢，这 Java 也太差劲了，如此简单的计算也会错！别忙着下结论，我们先来了解一下 Java 中的取余（% 标示符）算法，模拟代码如下：

```
// 模拟取余计算，dividend 被除数，divisor 除数
public static int remainder(int dividend,int divisor){
    return dividend - dividend / divisor * divisor;
}
```

看到这段程序，相信大家都会心地笑了，原来 Java 是这么处理取余计算的呀。根据上面

的模拟取余可知，当输入 -1 的时候，运算结果是 -1，当然不等于 1 了，所以它就被判定为偶数了，也就是说我们的判断失误了。问题明白了，修正也很简单，改为判断是否是偶数即可，代码如下：

```
i%2 ==0?" 偶数 ":" 奇数 "
```

---

**注意** 对于基础知识，我们应该“知其然，并知其所以然”。

---

## 建议 22：用整数类型处理货币

在日常生活中，最容易接触到的小数就是货币，比如你付给售货员 10 元钱购买一个 9.60 元的零食，售货员应该找你 0.4 元也就是 4 毛钱才对，我们来看下面的程序：

```
public class Client {
    public static void main(String[] args) {
        System.out.println(10.00-9.60) ;
    }
}
```

我们期望的结果是 0.4，也应该是这个数字，但是打印出来的却是 0.40000000000000036，这是为什么呢？

这是因为在计算机中浮点数有可能（注意是可能）是不准确的，它只能无限接近准确值，而不能完全精确。为什么会如此呢？这是由浮点数的存储规则所决定的，我们先来看 0.4 这个十进制小数如何转换成二进制小数，使用“乘 2 取整，顺序排列”法（不懂？这就没招了，太基础了），我们发现 0.4 不能使用二进制准确的表示，在二进制数世界里它是一个无限循环的小数，也就是说，“展示”都不能“展示”，更别说是在内存中存储了（浮点数的存储包括三部分：符号位、指数位、尾数，具体不再介绍），可以这样理解，在十进制的世界里没有办法准确表示  $1/3$ ，那在二进制世界里当然也无法准确表示  $1/5$ （如果二进制也有分数的话倒是可以表示），在二进制的世界里  $1/5$  是一个无限循环小数。

各位要说了，那我对结果取整不就对了吗？代码如下：

```
public class Client {
    public static void main(String[] args) {
        NumberFormat f = new DecimalFormat("#.##");
        System.out.println(f.format(10.00-9.60));
    }
}
```

打印出结果是 0.4，看似解决了，但是隐藏了一个很深的问题。我们来思考一下金融行业的计算方法，会计系统一般记录小数点后的 4 位小数，但是在汇总、展现、报表中，则只记录小数点后的 2 位小数，如果使用浮点数来计算货币，想想看，在大批量的加减乘除后结

果会有多大的差距（其中还涉及后面会讲到的四舍五入问题）！会计系统要的就是准确，但是却因为计算机的缘故不准确了，那真是罪过。要解决此问题有两种方法：

#### （1）使用 BigDecimal

BigDecimal 是专门为弥补浮点数无法精确计算的缺憾而设计的类，并且它本身也提供了加减乘除的常用数学算法。特别是与数据库 Decimal 类型的字段映射时，BigDecimal 是最优的解决方案。

#### （2）使用整型

把参与运算的值扩大 100 倍，并转变为整型，然后在展现时再缩小 100 倍，这样处理的好处是计算简单、准确，一般在非金融行业（如零售行业）应用较多。此方法还会用于某些零售 POS 机，它们的输入和输出全部是整数，那运算就更简单。

## 建议 23：不要让类型默默转换

我们出一个小学生的题目给大家做做看，光速是每秒 30 万公里，根据光线旅行的时间，计算月亮与地球、太阳与地球之间的距离。代码如下：

```
public class Client {
    // 光速是 30 万公里 / 秒，常量
    public static final int LIGHT_SPEED = 30 * 10000 * 1000;
    public static void main(String[] args) {
        System.out.println(" 题目 1: 月亮光照射到地球需要 1 秒，计算月亮和地球的距离。");
        long dis1 = LIGHT_SPEED * 1;
        System.out.println(" 月亮与地球的距离是: " + dis1 + " 米");
        System.out.println("-----");
        System.out.println(" 题目 2: 太阳光照射到地球上需要 8 分钟，计算太阳到地球的距离。");
        // 可能要超出整数范围，使用 long 型
        long dis2 = LIGHT_SPEED * 60 * 8;
        System.out.println(" 太阳与地球的距离是: " + dis2 + " 米");
    }
}
```

估计你要鄙视了，这种小学生乘法计算有什么可做的。不错，确实就是一个乘法运算，我们运行一下看看结果：

```
题目 1: 月亮光照射到地球需要 1 秒，计算月亮和地球的距离。
月亮与地球的距离是: 300000000 米
-----
题目 2: 太阳光照射到地球上需要 8 分钟，计算太阳到地球的距离。
太阳与地球的距离是: -2028888064 米
```

太阳和地球的距离竟然是负的，诡异。dis2 不是已经考虑到 int 类型可能越界的问题，并使用了 long 型吗，为什么还会出现负值呢？



那是因为 Java 是先运算然后再进行类型转换的，具体地说就是因为 disc2 的三个运算参数都是 int 类型，三者相乘的结果虽然也是 int 类型，但是已经超过了 int 的最大值，所以其值就是负值了（为什么是负值？因为过界了就会从头开始），再转换成 long 型，结果还是负值。

问题知道了，解决起来也很简单，只要加个小小的“L”即可，代码如下：

```
long dis2 = LIGHT_SPEED * 60L * 8;
```

60L 是一个长整型，乘出来的结果也是一个长整型（此乃 Java 的基本转换规则，向数据范围大的方向转换，也就是加宽类型），在还没有超过 int 类型的范围时就已经转换为 long 型了，彻底解决了越界问题。在实际开发中，更通用的做法是主动声明式类型转化（注意不是强制类型转换），代码如下：

```
long dis2 = 1L * LIGHT_SPEED * 60 * 8;
```

既然期望的结果是 long 型，那就让第一个参与运算的参数也是 long 型（1L）吧，也就是明说“嗨，我已经是长整型了，你们都跟着我一起转为长整型吧”。

---

**注意** 基本类型转换时，使用主动声明方式减少不必要的 Bug。

---

## 建议 24：边界，边界，还是边界

某商家生产的电子产品非常畅销，需要提前 30 天预订才能抢到手，同时它还规定了一个会员可拥有的最多产品数量，目的是防止囤积压货肆意加价。会员的预定过程是这样的：先登录官方网站，选择产品型号，然后设置需要预订的数量，提交，符合规则即提示下单成功，不符合规则提示下单失败。后台的处理逻辑模拟如下：

```
public class Client {
    // 一个会员拥有产品的最多数量
    public final static int LIMIT = 2000;
    public static void main(String[] args) {
        // 会员当前拥有的产品数量
        int cur = 1000;
        Scanner input = new Scanner(System.in);
        System.out.print(" 请输入需要预定的数量: ");
        while(input.hasNextInt()){
            int order = input.nextInt();
            // 当前拥有的与准备订购的产品数量之和
            if(order>0 && order+cur<=LIMIT){
                System.out.println(" 你已经成功预定的 "+order+" 个产品! ");
            }else{
```

```
        System.out.println(" 超过限额，预订失败！ ");  
    }  
}  
}
```

这是一个简易的订单处理程序，其中 `cur` 代表的是会员已经拥有的产品数量，`LIMIT` 是一个会员最多拥有的产品数量（现实中这两个参数当然是从数据库中获得的，不过这里是一个模拟程序），如果当前预订数量与拥有数量之和超过了最大数量，则预订失败，否则下单成功。业务逻辑很简单，同时在 Web 界面上对订单数量做了严格的校验，比如不能是负值、不能超过最大数量等，但是人算不如天算，运行不到两小时数据库中就出现了异常数据：某会员拥有产品的数量与预订数量之和远远大于限额。怎么会这样？程序逻辑上不可能有问题呀，这是如何产生的呢？我们来模拟一下，第一次输入：

```
请输入需要预定的数量：800  
你已经成功预定的 800 个产品！
```

这完全满足条件，没有任何问题，继续输入：

```
请输入需要预定的数量：2147483647  
你已经成功预定的 2147483647 个产品！
```

看到没，这个数字远远超过了 2000 的限额，但是竟然预订成功了，真是神奇！

看着 2147483647 这个数字很眼熟？那就对了，它是 `int` 类型的最大值，没错，有人输入了一个最大值，使校验条件失效了，Why？我们来看程序，`order` 的值是 2147483647，那再加上 1000 就超出 `int` 的范围了，其结果是 -2147482649，那当然是小于正数 2000 了！一句话可归结其原因：数字越界使检验条件失效。

在单元测试中，有一项测试叫做边界测试（也有叫做临界测试），如果一个方法接收的是 `int` 类型的参数，那以下三个值是必测的：0、正最大、负最小，其中正最大和负最小是边界值，如果这三个值都没有问题，方法才是比较安全可靠的。我们的例子就是因为缺少边界测试，致使生产系统产生了严重的偏差。

也许你要疑惑了，Web 界面既然已经做了严格的校验，为什么还能输入 2147483647 这么大的数字呢？是否说明 Web 校验不严格？错了，不是这样的，Web 校验都是在页面上通过 JavaScript 实现的，只能限制普通用户（这里的普通用户是指不懂 HTML、不懂 HTTP、不懂 Java 的简单使用者），而对于高手，这些校验基本上就是摆设，HTTP 是明文传输的，将其拦截几次，分析一下数据结构，然后再写一个模拟器，一切前端校验就都成了浮云！想往后台提交个什么数据那还不是信手拈来？！

## 建议 25: 不要让四舍五入亏了一方

本建议还是来重温一个小学数学问题：四舍五入。四舍五入是一种近似精确的计算方法，在 Java 5 之前，我们一般是通过使用 `Math.round` 来获得指定精度的整数或小数的，这种方法使用非常广泛，代码如下：

```
public class Client {
    public static void main(String[] args) {
        System.out.println("10.5 近似值: " + Math.round(10.5));
        System.out.println("-10.5 近似值: " + Math.round(-10.5));
    }
}
```

输出结果为：

```
10.5 近似值: 11
-10.5 近似值: -10
```

这是四舍五入的经典案例，也是初级面试官很乐意选择的考题，绝对值相同的两个数字，近似值为什么就不同了？这是由 `Math.round` 采用的舍入规则所决定的（采用的是正无穷方向舍入规则，后面会讲解）。我们知道四舍五入是有误差的：其误差值是舍入位的一半。我们以舍入运用最频繁的银行利息计算为例来阐述该问题。

我们知道银行的盈利渠道主要是利息差，从储户手里收拢资金，然后放贷出去，其间的利息差额便是所获得的利润。对一个银行来说，对付给储户的利息的计算非常频繁，人民银行规定每个季度末月的 20 日为银行结息日，一年有 4 次的结息日。

场景介绍完毕，我们回过头来看四舍五入，小于 5 的数字被舍去，大于等于 5 的数字进位后舍去，由于所有位上的数字都是自然计算出来的，按照概率计算可知，被舍入的数字均匀分布在 0 到 9 之间，下面以 10 笔存款利息计算作为模型，以银行家的身份来思考这个算法：

- ❑ 四舍。舍弃的数值：0.000、0.001、0.002、0.003、0.004，因为是舍弃的，对银行家来说，就不用付款给储户了，那每舍弃一个数字就会赚取相应的金额：0.000、0.001、0.002、0.003、0.004。
- ❑ 五入。进位的数值：0.005、0.006、0.007、0.008、0.009，因为是进位，对银行家来说，每进一位就会多付款给储户，也就是亏损了，那亏损部分就是其对应的 10 进制补数：0.005、0.004、0.003、0.002、0.001。

因为舍弃和进位的数字是在 0 到 9 之间均匀分布的，所以对于银行家来说，每 10 笔存款的利息因采用四舍五入而获得的盈利是：

$$0.000 + 0.001 + 0.002 + 0.003 + 0.004 - 0.005 - 0.004 - 0.003 - 0.002 - 0.001 = -0.005$$

也就是说，每 10 笔的利息计算中就损失 0.005 元，即每笔利息计算损失 0.0005 元，这

对一家有 5 千万储户的银行来说（对国内的银行来说，5 千万是个很小的数字），每年仅仅因为四舍五入的误差而损失的金额是：

```
public class Client {
    public static void main(String[] args) {
        // 银行账户数量，5 千万
        int accountNum = 5000 * 10000;
        // 按照人行的规定，每个季度末月的 20 日为银行结息日
        double cost = 0.0005 * accountNum * 4;
        System.out.println(" 银行每年损失的金额: " + cost);
    }
}
```

输出的结果是：“银行每年损失的金额：100000.0”。即，每年因为一个算法误差就损失了 10 万元，事实上以上的假设条件都是非常保守的，实际情况可能损失得更多。那各位可能要说了，银行还要放贷呀，放出去这笔计算误差不就抵消掉了吗？不会抵销，银行的贷款数量是非常有限的，其数量级根本没有办法和存款相比。

这个算法误差是由美国银行家发现的（那可是私人银行，钱是自己的，白白损失了可不行），并且对此提出了一个修正算法，叫做银行家舍入（Banker's Round）的近似算法，其规则如下：

- ❑ 舍去位的数值小于 5 时，直接舍去；
- ❑ 舍去位的数值大于等于 6 时，进位后舍去；
- ❑ 当舍去位的数值等于 5 时，分两种情况：5 后面还有其他数字（非 0），则进位后舍去；若 5 后面是 0（即 5 是最后一个数字），则根据 5 前一位数的奇偶性来判断是否需要进位，奇数进位，偶数舍去。

以上规则汇总成一句话：四舍六入五考虑，五后非零就进一，五后为零看奇偶，五前为偶应舍去，五前为奇要进一。我们举例说明，取 2 位精度：

```
round(10.5551) = 10.56
round(10.555)  = 10.56
round(10.545)  = 10.54
```

要在 Java 5 以上的版本中使用银行家的舍入法则非常简单，直接使用 RoundingMode 类提供的 Round 模式即可，示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        // 存款
        BigDecimal d = new BigDecimal(888888);
        // 月利率，乘 3 计算季利率
        BigDecimal r = new BigDecimal(0.001875 * 3);
        // 计算利息
        BigDecimal i = d.multiply(r).setScale(2, RoundingMode.HALF_EVEN);
    }
}
```

```

        System.out.println(" 季利息是: "+i);
    }
}

```

在上面的例子中，我们使用了 `BigDecimal` 类，并且采用 `setScale` 方法设置了精度，同时传递了一个 `RoundingMode.HALF_EVEN` 参数表示使用银行家舍入法则进行近似计算，`BigDecimal` 和 `RoundingMode` 是一个绝配，想要采用什么舍入模式使用 `RoundingMode` 设置即可。目前 Java 支持以下七种舍入方式：

❑ **ROUND\_UP**：远离零方向舍入。

向远离 0 的方向舍入，也就是说，向绝对值最大的方向舍入，只要舍弃位非 0 即进位。

❑ **ROUND\_DOWN**：趋向零方向舍入。

向 0 方向靠拢，也就是说，向绝对值最小的方向输入，注意：所有的位都舍弃，不存在进位情况。

❑ **ROUND\_CEILING**：向正无穷方向舍入。

向正最大方向靠拢，如果是正数，舍入行为类似于 `ROUND_UP`；如果为负数，则舍入行为类似于 `ROUND_DOWN`。注意：`Math.round` 方法使用的即为此模式。

❑ **ROUND\_FLOOR**：向负无穷方向舍入。

向负无穷方向靠拢，如果是正数，则舍入行为类似于 `ROUND_DOWN`；如果是负数，则舍入行为类似于 `ROUND_UP`。

❑ **HALF\_UP**：最近数字舍入（5 进）。

这就是我们最经典的四舍五入模式。

❑ **HALF\_DOWN**：最近数字舍入（5 舍）。

在四舍五入中，5 是进位的，而在 `HALF_DOWN` 中却是舍弃不进位。

❑ **HALF\_EVEN**：银行家算法。

在普通的项目中舍入模式不会有太多影响，可以直接使用 `Math.round` 方法，但在大量与货币数字交互的项目中，一定要选择好近似的计算模式，尽量减少因算法不同而造成的损失。

---

**注意** 根据不同的场景，慎重选择不同的舍入模式，以提高项目的精准度，减少算法损失。

---

## 建议 26：提防包装类型的 null 值

我们知道 Java 引入包装类型（Wrapper Types）是为了解决基本类型的实例化问题，以便让一个基本类型也能参与到面向对象的编程世界中。而在 Java 5 中泛型更是对基本类型说了“不”，如想把一个整型放到 `List` 中，就必须使用 `Integer` 包装类型。我们来看一段代码：

```
// 计算 list 中所有元素之和
public static int f(List<Integer> list){
    int count = 0;
    for(int i:list){
        count += i;
    }
    return count;
}
```

接收一个元素是整型的 List 参数，计算所有元素之和，这在统计、报表项目中很常见，我们来看看这段代码有没有问题。遍历一个列表，然后相加，应该没有问题。那我们来写一个方法调用，代码如下：

```
public static void main(String[] args) {
    List<Integer> list = new ArrayList<Integer>();
    list.add(1);
    list.add(2);
    list.add(null);
    System.out.println(f(list));
}
```

把 1、2 和空值都放到 List 中，然后调用方法计算，现在来思考一下会不会出错。应该不会出错吧，基本类型和包装类型都是可以通过自动装箱（Autoboxing）和自动拆箱（AutoUnboxing）自由转换的，null 应该可以转为 0 吧，真的是这样吗？我们运行一下看看结果：

```
Exception in thread "main" java.lang.NullPointerException
```

运行失败，报空指针异常，我们稍稍思考一下很快就知道原因了：在程序的 for 循环中，隐含了一个拆箱过程，在此过程中包装类型转换为了基本类型。我们知道拆箱过程是通过调用包装对象的 intValue 方法来实现的，由于包装对象是 null 值，访问其 intValue 方法报空指针异常也就在所难免了。问题清楚了，修改也很简单，加入 null 值检查即可，代码如下：

```
public static int f(List<Integer> list) {
    int count = 0;
    for (Integer i : list) {
        count += (i!=null)?i:0;
    }
    return count;
}
```

上面以 Integer 和 int 为例说明了拆箱问题，其他 7 个包装对象的拆箱过程也存在着同样的问题。包装对象和拆箱对象可以自由转换，这不假，但是要剔除 null 值，null 值并不能转化为基本类型。对于此类问题，我们谨记一点：包装类型参与运算时，要做 null 值校验。



## 建议 27：谨慎包装类型的大小比较

基本类型是可以比较大小的，其所对应的包装类型都实现了 Comparable 接口也说明了此问题，那我们来比较一下两个包装类型的大小，代码如下：

```
public class Client {
    public static void main(String[] args) {
        Integer i = new Integer(100);
        Integer j = new Integer(100);
        compare(i, j);
    }
    // 比较两个包装对象大小
    public static void compare(Integer i, Integer j) {
        System.out.println(i == j);
        System.out.println(i > j);
        System.out.println(i < j);
    }
}
```

代码很简单，产生了两个 Integer 对象，然后比较两者的大小关系，既然基本类型和包装类型是可以自由转换的，那上面的代码是不是就可打印出两个相等的值呢？让事实说话，运行结果如下：

```
false
false
false
```

竟然是 3 个 false，也就是说两个值之间不等，也没大小关系，这也太奇怪了吧。不奇怪，我们来一一解释。

□  $i == j$

在 Java 中 “==” 是用来判断两个操作数是否有相等关系的，如果是基本类型则判断值是否相等，如果是对象则判断是否是一个对象的两个引用，也就是地址是否相等，这里很明显是两个对象，两个地址，不可能相等。

□  $i > j$  和  $i < j$

在 Java 中，“>” 和 “<” 用来判断两个数字类型的大小关系，注意只能是数字型的判断，对于 Integer 包装类型，是根据其 intValue() 方法的返回值（也就是其相应的基本类型）进行比较的（其他包装类型是根据相应的 value 值来比较的，如 doubleValue、floatValue 等），那很显然，两者不可能有大小关系的。

问题清楚了，修改总是比较容易的，直接使用 Integer 实例的 compareTo 方法即可。但是这类问题的产生更应该说是习惯问题，只要是两个对象之间的比较就应该采用相应的方法，而不是通过 Java 的默认机制来处理，除非你确定对此非常了解。

## 建议 28：优先使用整型池

上一建议我们解释了包装对象的比较问题，本建议将继续深入讨论相关问题，首先看如下代码：

```
public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    while(input.hasNextInt()){
        int ii = input.nextInt();
        System.out.println("\n===="+ii+" 的相等判断 =====");
        // 两个通过 new 产生的 Integer 对象
        Integer i =new Integer(ii);
        Integer j = new Integer(ii);
        System.out.println("new 产生的对象: " + (i==j));

        // 基本类型转为包装类型后比较
        i=ii;
        j=ii;
        System.out.println("基本类型转换的对象: " + (i==j));

        // 通过静态方法生成一个实例
        i=Integer.valueOf(ii);
        j = Integer.valueOf(ii);
        System.out.println("valueOf 产生的对象: " + (i==j));
    }
}
```

输入多个数字，然后按照 3 种不同的方式产生 Integer 对象，判断其是否相等，注意这里使用了“==”，这说明判断的不是同一个对象。我们输入三个数字 127、128、555，结果如下：

```
====127 的相等判断=====
new 产生的对象: false
基本类型转换的对象: true
valueOf 产生的对象: true

====128 的相等判断=====
new 产生的对象: false
基本类型转换的对象: false
valueOf 产生的对象: false

====555 的相等判断=====
new 产生的对象: false
基本类型转换的对象: false
valueOf 产生的对象: false
```

很不可思议呀，数字 127 的比较结果竟然与其他两个数字不同，它的装箱动作所产生的对象竟然是同一个对象，valueOf 产生的也是同一个对象，但是大于 127 的数字 128 和 555

在比较过程中所产生的却不是同一个对象，这是为什么？我们一个一个来解释。

### (1) new 产生的 Integer 对象

new 声明的就是要生成一个新的对象，没二话，这是两个对象，地址肯定不等，比较结果为 false。

### (2) 装箱生成的对象

对于这一点，首先要说明的是装箱动作是通过 valueOf 方法实现的，也就是说后两个算法是相同的，那结果肯定也是一样的，现在的问题是：valueOf 是如何生成对象的呢？我们来阅读一下 Integer.valueOf 的实现代码：

```
public static Integer valueOf(int i) {
    final int offset = 128;
    if (i >= -128 && i <= 127) { // must cache
        return IntegerCache.cache[i + offset];
    }
    return new Integer(i);
}
```

这段代码的意思已经很明了了，如果是 -128 到 127 之间的 int 类型转换为 Integer 对象，则直接从 cache 数组中获得，那 cache 数组里是什么东西，代码如下：

```
static final Integer cache[] = new Integer[-(-128) + 127 + 1];

static {
    for(int i = 0; i < cache.length; i++)
        cache[i] = new Integer(i - 128);
}
```

cache 是 IntegerCache 内部类的一个静态数组，容纳的是 -128 到 127 之间的 Integer 对象。通过 valueOf 产生包装对象时，如果 int 参数在 -128 和 127 之间，则直接从整型池中获得对象，不在该范围的 int 类型则通过 new 生成包装对象。

明白了这一点，要理解上面的输出结果就迎刃而解了，127 的包装对象是直接从整型池中获得的，不管你输入多少次 127 这个数字，获得的对象都是同一个，那地址当然都是相等的。而 128、555 超出了整型池范围，是通过 new 产生一个新的对象，地址不同，当然也就不相等了。

以上的解释也是整型池的原理，整型池的存在不仅仅提高了系统性能，同时也节约了内存空间，这也是我们使用整型池的原因，也就是在声明包装对象的时候使用 valueOf 生成，而不是通过构造函数来生成的原因。顺便提醒大家，在判断对象是否相等的时候，最好是用 equals 方法，避免用 “==” 产生非预期结果。

---

**注意** 通过包装类的 valueOf 生成包装实例可以显著提高空间和时间性能。

---

## 建议 29：优先选择基本类型

包装类型是一个类，它提供了诸如构造方法、类型转换、比较等非常实用的功能，而且在 Java 5 之后又实现了与基本类型之间的自动转换，这使包装类型如虎添翼，更是应用广泛了，在开发中包装类型已经随处可见，但无论是从安全性、性能方面来说，还是从稳定性方面来说，基本类型都是首选方案。我们来看一段代码：

```
public class Client {
    public static void main(String[] args) {
        Client cilent = new Client();
        int i=140;
        // 分别传递 int 类型和 Integer 类型
        cilent.f(i);
        cilent.f(Integer.valueOf(i));
    }
    public void f(long a) {
        System.out.println(" 基本类型的方法被调用 ");
    }
    public void f(Long a) {
        System.out.println(" 包装类型的方法被调用 ");
    }
}
```

在上面的程序中首先声明了一个 int 变量 i，然后加宽转变成 long 型，再调用 f() 方法，分别传递 int 和 long 的基本类型和包装类型，诸位想想该程序是否能够编译？如果能编译输出结果又是什么呢？

首先，这段程序绝对是能够编译的。不过，说不能编译的同学还是很动了一番脑筋的，只是还欠缺点火候，你可能会猜测以下这些地方不能编译：

- ❑ f() 方法重载有问题。定义的两个 f() 方法实现了重载，一个形参是基本类型，一个形参是包装类型，这类重载很正常。虽然基本类型和包装类型有自动装箱、自动拆箱的功能，但并不影响它们的重载，自动拆箱（装箱）只有在赋值时才会发生，和重载没有关系。
- ❑ cilent.f(i) 报错。i 是 int 类型，传递到 fun(long l) 是没有任何问题的，编译器会自动把 i 的类型加宽，并将其转变为 long 型，这是基本类型的转换规则，也没有任何问题。
- ❑ cilent.f(Integer.valueOf(i)) 报错。代码中没有 f(Integer i) 方法，不可能接收一个 Integer 类型的参数，而且 Integer 和 Long 两个包装类型是兄弟关系，不是继承关系，那就是说肯定编译失败了？不，编译是成功的，稍后再解释为什么这里编译成功。

既然编译通过了，我们来看一下输出：

```
基本类型的方法被调用
基本类型的方法被调用
```

cilent.f(i) 的输出是正常的，我们已经解释过了。那第二个输出就让人很困惑了，为什么

会调用 `f(long a)` 方法呢？这是因为自动装箱有一个重要的原则：基本类型可以先加宽，再转变成宽类型的包装类型，但不能直接转变成宽类型的包装类型。这句话比较拗口，简单地讲就是，`int` 可以加宽转变成 `long`，然后再转变成 `Long` 对象，但不能直接转变成包装类型，注意这里指的都是自动转换，不是通过构造函数生成。为了解释这个原则，我们再来看一个例子：

```
public class Client {
    public static void main(String[] args) {
        int i=100;
        f(i);
    }
    public static void f(Long l){
    }
}
```

这段程序编译是通不过的，因为 `i` 是一个 `int` 类型，不能自动转变为 `Long` 型。但是修改成以下代码就可以编译通过了：

```
public static void main(String[] args) {
    int i=100;
    long l = (long)i;
    f(l);
}
```

这就是 `int` 先加宽转变为 `long` 型，然后自动转换成 `Long` 型。规则说明白了，我们继续来看 `f(Integer.valueOf(i))` 是如何调用的，`Integer.valueOf(i)` 返回的是一个 `Integer` 对象，这没错，但是 `Integer` 和 `int` 是可以互相转换的。没有 `f(Integer i)` 方法？没关系，编译器会尝试转换成 `int` 类型的实参调用，OK，这次成功了，与 `f(i)` 相同了，于是乎被加宽转变成 `long` 型——结果也很明显了。整个 `f(Integer.valueOf(i))` 的执行过程是这样的：

- ❑ `i` 通过 `valueOf` 方法包装成一个 `Integer` 对象。
- ❑ 由于没有 `f(Integer i)` 方法，编译器“聪明”地把 `Integer` 对象转换成 `int`。
- ❑ `int` 自动拓宽为 `long`，编译结束。

使用包装类型确实有方便的地方，但是也会引起一些不必要的困惑，比如我们这个例子，如果 `f()` 的两个重载方法使用的是基本类型，而且实参也是基本类型，就不会产生以上问题，而且程序的可读性更强。自动装箱（拆箱）虽然很方便，但引起的问题也非常严重——我们甚至都不知道执行的是哪个方法。

---

**注意** 重申，基本类型优先考虑。

---

## 建议 30：不要随便设置随机种子

随机数在太多的地方使用了，比如加密、混淆数据等，我们使用随机数是期望获得一个

唯一的、不可仿造的数字，以避免产生相同的业务数据造成混乱。在 Java 项目中通常是通过 `Math.random` 方法和 `Random` 类来获得随机数的，我们来看一段代码：

```
public class Client {  
    public static void main(String[] args) {  
        Random r = new Random();  
        for(int i=1;i<4;i++){  
            System.out.println(" 第 "+i+" 次: "+r.nextInt());  
        }  
    }  
}
```

代码很简单，我们一般都是这样获得随机数的，运行此程序可知：三次打印的随机数都不相同，即使多次运行结果也不同，这也正是我们想要随机数的原因。我们再来看下面的程序：

```
public class Client {  
    public static void main(String[] args) {  
        Random r = new Random(1000);  
        for(int i=1;i<4;i++){  
            System.out.println(" 第 "+i+" 次: "+r.nextInt());  
        }  
    }  
}
```

上面使用了 `Random` 的有参构造，运行结果如下：

```
第 1 次: -498702880  
第 2 次: -858606152  
第 3 次: 1942818232
```

计算机不同输出的随机数也不同，但是有一点是相同的：在同一台机器上，甭管运行多少次，所打印的随机数都是相同的，也就是说第一次运行，会打印出这三个随机数，第二次运行还是打印出这三个随机数，只要是在同一台硬件机器上，就永远都会打印出相同的随机数，似乎随机数不随机了，问题何在？

那是因为产生随机数的种子被固定了，在 Java 中，随机数的产生取决于种子，随机数和种子之间的关系遵从以下两个规则：

- ❑ 种子不同，产生不同的随机数。
- ❑ 种子相同，即使实例不同也产生相同的随机数。

看完上面两个规则，我们再来看这个例子，会发现问题就出在有参构造上，`Random` 类的默认种子（无参构造）是 `System.nanoTime()` 的返回值（JDK 1.5 版本以前默认种子是 `System.currentTimeMillis()` 的返回值），注意这个值是距离某一个固定时间点的纳秒数，不同的操作系统和硬件有不同的固定时间点，也就是说不同的操作系统其纳秒值是不同的，而同一个操作系统纳秒值也会不同，随机数自然也就不同了。（顺便说下，`System.nanoTime` 不



能用于计算日期，那是因为“固定”的时间点是不确定的，纳秒值甚至可能是负值，这点与 `System.currentTimeMillis` 不同。)

`new Random(1000)` 显式地设置了随机种子为 1000，运行多次，虽然实例不同，但都会获得相同的三个随机数。所以，除非必要，否则不要设置随机种子。

顺便提一下，在 Java 中有两种方法可以获得不同的随机数：通过 `java.util.Random` 类获得随机数的原理和 `Math.random` 方法相同，`Math.random()` 方法也是通过生成一个 `Random` 类的实例，然后委托 `nextDouble()` 方法的，两者是殊途同归，没有差别。

---

**注意** 若非必要，不要设置随机数种子。

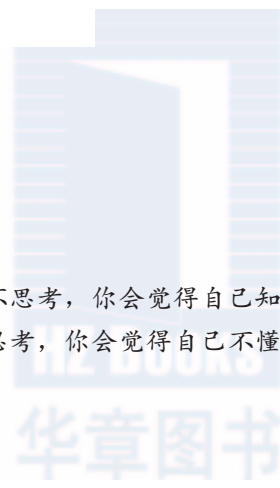
---





## 第 3 章

# 类、对象及方法



书读得多而不思考，你会觉得自己知道的很多。

书读得多而思考，你会觉得自己不懂的越来越多。

——伏尔泰

在面向对象编程（Object-Oriented Programming, OOP）的世界里，类和对象是真实世界的描述工具，方法是行为和动作的展示形式，封装、继承、多态则是其多姿多彩的主要实现方式，如此，OOP 才会像现在这样繁荣昌盛、欣欣向荣。

本章主要讲述关于 Java 类、对象、方法的种种规则、限制及建议，让读者在面向对象编程的世界中走得更远，飞得更高。

## 建议 31：在接口中不要存在实现代码

看到这样的标题读者可能会纳闷：接口中有实现代码？这怎么可能呢？确实，接口中可以声明常量，声明抽象方法，也可以继承父接口，但就是不能有具体实现，因为接口是一种契约（Contract），是一种框架性协议，这表明它的实现类都是同一种类型，或者是具备相似特征的一个集合体。对于一般程序，接口确实没有任何实现，但是在那些特殊的程序中就例外了，阅读如下代码：

```
public class Client {
    public static void main(String[] args) {
        // 调用接口的实现
        B.s.doSomething();
    }
}
// 在接口中存在实现代码
interface B{
    public static final S s = new S(){
        public void doSomething(){
            System.out.println(" 我在接口中实现了 ");
        }
    };
}
// 被实现的接口
interface S{
    public void doSomething();
}
```

仔细看 main 方法，注意那个 B 接口。它调用了接口常量，在没有任何显式实现类的情况下，它竟然打印出了结果，那 B 接口中的 s 常量（接口是 S）是在什么地方被实现的呢？答案是在 B 接口中。

在 B 接口中声明了一个静态常量 s，其值是一个匿名内部类（Anonymous Inner Class）的实例对象，就是该匿名内部类（当然，可以不用匿名，直接在接口中实现内部类也是允许的）实现了 S 接口。你看，在接口中存在着实现代码吧！

这确实很好，很强大，但是在一般的项目中，此类代码是严禁出现的，原因很简单：这是一种不好的编码习惯，接口是用来干什么的？接口是一个契约，不仅仅约束着实现者，同

时也是一个保证，保证提供的服务（常量、方法）是稳定、可靠的，如果把实现代码写到接口中，那接口就绑定了可能变化的因素，这就会导致实现不再稳定和可靠，是随时都可能被抛弃、被更改、被重构的。所以，接口中虽然可以有实现，但应避免使用。

---

**注意** 接口中不能存在实现代码。

---

## 建议 32：静态变量一定要先声明后赋值

这标题看着让人很纳闷，什么叫做变量一定要先声明后赋值？Java 中的变量不都是先声明后使用的吗？难道还能先使用后声明？能不能暂且不说，我们先来看一个例子，代码如下：

```
public class Client {
    public static int i=1;
    static{
        i=100;
    }
    public static void main(String[] args) {
        System.out.println(i);
    }
}
```

这段程序很简单，输出 100 嘛！对，确实是 100，我们再稍稍修改一下，代码如下：

```
public class Client {
    static{
        i=100;
    }
    public static int i=1;
    public static void main(String[] args) {
        System.out.println(i);
    }
}
```

注意，变量 *i* 的声明和赋值调换了位置，现在的问题是：这段程序能否编译？如果可以编译那输出是多少？还要注意：这个变量 *i* 可是先使用（也就是赋值）后声明的。

答案是：可以编译，没有任何问题，输出是 1。对，你没有看错，输出确实是 1，而不是 100。仅仅调换了一下位置，输出就变了，而且变量 *i* 还真是先使用后声明的，难道这世界真的颠倒了？

这要从静态变量的诞生说起了，静态变量是类加载时被分配到数据区（Data Area）的，它在内存中只有一个拷贝，不会被分配多次，其后的所有赋值操作都是值改变，地址则保持不变。我们知道 JVM 初始化变量是先声明空间，然后再赋值的，也就是说：

```
int i=100;
```

在 JVM 中是分开执行，等价于：

```
int i;    // 分配地址空间
i=100;   // 赋值
```

静态变量是在类初始化时首先被加载的，JVM 会去查找类中所有的静态声明，然后分配空间，注意这时候只是完成了地址空间的分配，还没有赋值，之后 JVM 会根据类中静态赋值（包括静态类赋值和静态块赋值）的先后顺序来执行。对于程序来说，就是先声明了 int 类型的地址空间，并把地址传递给了 i，然后按照类中的先后顺序执行赋值动作，首先执行静态块中 i=100，接着执行 i=1，那最后的结果就是 i=1 了。

哦，如此而已，那再问一个问题：如果有多个静态块对 i 继续赋值呢？i 当然还是等于 1 了，谁的位置最靠后谁有最终的决定权。

有些程序员喜欢把变量定义放到类的底部，如果这是实例变量还好说，没有任何问题，但如果是静态变量，而且还在静态块中进行了赋值，那这结果可就和期望的不一样了，所以遵循 Java 通用的开发规范“变量先声明后使用”是一个良好的编码风格。

---

**注意** 再次重申变量要先声明后使用，这不是一句废话。

---

## 建议 33：不要覆写静态方法

我们知道在 Java 中可以通过覆写（Override）来增强或减弱父类的方法和行为，但覆写是针对非静态方法（也叫做实例方法，只有生成实例才能调用的方法）的，不能针对静态方法（static 修饰的方法，也叫做类方法），为什么呢？我们先看一个例子，代码如下：

```
public class Client {
    public static void main(String[] args) {
        Base base = new Sub();
        // 调用非静态方法
        base.doAnything();
        // 调用静态方法
        base.doSomething();
    }
}

class Base{
    // 父类静态方法
    public static void doSomething(){
        System.out.println("我是父类静态方法");
    }
    // 父类非静态方法
    public void doAnything(){
        System.out.println("我是父类非静态方法");
    }
}
```

```

    }
}

class Sub extends Base{
    // 子类同名、同参数的静态方法
    public static void doSomething(){
        System.out.println(" 我是子类静态方法 ");
    }
    // 覆写父类的非静态方法
    @Override
    public void doAnything(){
        System.out.println(" 我是子类非静态方法 ");
    }
}

```

注意看程序，子类的 doAnything 方法覆写了父类方法，这没有任何问题，那 doSomething 方法呢？它与父类的方法名相同，输入、输出也相同，按道理来说应该是覆写，不过到底是不是覆写呢？我们先看输出结果：

```

我是子类非静态方法
我是父类静态方法

```

这个结果很让人困惑，同样是调用子类方法，一个执行了子类方法，一个执行了父类方法，两者的差别仅仅是有无 static 修饰，却得到不同的输出结果，原因何在呢？

我们知道一个实例对象有两个类型：表面类型（Apparent Type）和实际类型（Actual Type），表面类型是声明时的类型，实际类型是对象产生时的类型，比如我们例子，变量 base 的表面类型是 Base，实际类型是 Sub。对于非静态方法，它是根据对象的实际类型来执行的，也就是执行了 Sub 类中的 doAnything 方法。而对于静态方法来说就比较特殊了，首先静态方法不依赖实例对象，它是通过类名访问的；其次，可以通过对象访问静态方法，如果是通过对象调用静态方法，JVM 则会通过对象的表面类型查找到静态方法的入口，继而执行之。因此上面的程序打印出“我是父类静态方法”，也就不足为奇了。

在子类中构建与父类相同的方法名、输入参数、输出参数、访问权限（权限可以扩大），并且父类、子类都是静态方法，此种行为叫做隐藏（Hide），它与覆写有两点不同：

- ❑ 表现形式不同。隐藏用于静态方法，覆写用于非静态方法。在代码上的表现是：@Override 注解可以用于覆写，不能用于隐藏。
- ❑ 职责不同。隐藏的目的是为了抛弃父类静态方法，重现子类方法，例如我们的例子，Sub.doSomething 的出现是为了遮盖父类的 Base.doSomething 方法，也就是期望父类的静态方法不要破坏子类的业务行为；而覆写则是将父类的行为增强或减弱，延续父类的职责。

解释了这么多，我们回头看一下本建议的标题：静态方法不能覆写，可以再续上一句话，虽然不能覆写，但是可以隐藏。顺便说一下，通过实例对象访问静态方法或静态属性不是好



习惯，它给代码带来了“坏味道”，建议读者阅之戒之。

## 建议 34：构造函数尽量简化

我们知道在通过 new 关键字生成对象时必然会调用构造函数，构造函数的简繁情况会直接影响实例对象的创建是否繁琐。在项目开发中，我们一般都会制订构造函数尽量简单，尽可能不抛异常，尽量不做复杂算法等规范，那如果一个构造函数确实复杂了会怎么样？我们来看一段代码：

```
public class Client {
    public static void main(String[] args) {
        Server s = new SimpleServer(1000);
    }
}
// 定义一个服务
abstract class Server{
    public final static int DEFAULT_PORT = 40000;
    public Server(){
        // 获得子类提供的端口号
        int port = getPort();
        System.out.println(" 端口号: " + port);
        /* 进行监听动作 */
    }
    // 由子类提供端口号，并做可用性检查
    protected abstract int getPort();
}

class SimpleServer extends Server{
    private int port=100;
    // 初始化传递一个端口号
    public SimpleServer(int _port){
        port = _port;
    }
    // 检查端口号是否有效，无效则使用默认端口，这里使用随机数模拟
    @Override
    protected int getPort() {
        return Math.random() > 0.5?port:DEFAULT_PORT;
    }
}
```

该代码是一个服务类的简单模拟程序，Server 类实现了服务器的创建逻辑，子类只要在生成实例对象时传递一个端口号即可创建一个监听该端口的服务，该代码的意图如下：

- 通过 SimpleServer 的构造函数接收端口参数。
- 子类的构造函数默认调用父类的构造函数。
- 父类构造函数调用子类的 getPort 方法获得端口号。

❑ 父类构造函数建立端口监听机制。

❑ 对象创建完毕，服务监听启动，正常运行。

貌似很合理，再仔细看看代码，确实也和我们的意图相吻合，那我们尝试多次运行看看，输出结果要么是“端口号：40000”，要么是“端口号：0”，永远不会出现“端口号：100”或是“端口号：1000”，这就奇怪了，40000 还好说，但那个 0 是怎么冒出来的呢？代码在什么地方出现问题了？

要解释这个问题，我们首先要说说子类是如何实例化的。子类实例化时，会首先初始化父类（注意这里是初始化，可不是生成父类对象），也就是初始化父类的变量，调用父类的构造函数，然后才会初始化子类的变量，调用子类自己的构造函数，最后生成一个实例对象。了解了相关知识，我们再来看上面的程序，其执行过程如下：

❑ 子类 SimpleServer 的构造函数接收 int 类型的参数：1000。

❑ 父类初始化常量，也就是 DEFAULT\_PORT 初始化，并设置为 40000。

❑ 执行父类无参构造函数，也就是子类的有参构造中默认包含了 super() 方法。

❑ 父类无参构造函数执行到“int port = getPort()”方法，调用子类的 getPort 方法实现。

❑ 子类的 getPort 方法返回 port 值（注意，此时 port 变量还没有赋值，是 0）或 DEFAULT\_PORT（此时已经是 40000）了。

❑ 父类初始化完毕，开始初始化子类的实例变量，port 赋值 100。

❑ 执行子类构造函数，port 被重新赋值为 1000。

❑ 子类 SimpleServer 实例化结束，对象创建完毕。

终于清楚了，在类初始化时 getPort 方法返回的 port 值还没有赋值，port 只是获得了默认初始值（int 类的实例变量默认初始值是 0），因此 Server 永远监听的是 40000 端口了（0 端口是没有意义的）。这个问题的产生从浅处说是由类元素初始化顺序导致的，从深处说是因为构造函数太复杂而引起的。构造函数用作初始化变量，声明实例的上下文，这都是简单的实现，没有任何问题，但我们的例子却实现了一个复杂的逻辑，而这放在构造函数里就不合适了。

问题知道了，修改也很简单，把父类的无参构造函数中的所有实现都移动到一个叫做 start 的方法中，将 SimpleServer 类初始化完毕，再调用其 start 方法即可实现服务器的启动工作，简洁而又直观，这也是大部分 JEE 服务器的实现方式。

---

**注意** 构造函数简化，再简化，应该达到“一眼洞穿”的境界。

---

## 建议 35：避免在构造函数中初始化其他类

构造函数是一个类初始化必须执行的代码，它决定着类的初始化效率，如果构造函数比

较复杂，而且还关联了其他类，则可能产生意想不到的问题，我们来看如下代码：

```
public class Client {
    public static void main(String[] args) {
        Son s = new Son();
        s.doSomething();
    }
}
// 父类
class Father{
    Father(){
        new Other();
    }
} // 子类
class Son extends Father{
    public void doSomething(){
        System.out.println("Hi,show me something");
    }
}
// 相关类
class Other{
    public Other(){
        new Son();
    }
}
```

这段代码并不复杂，只是在构造函数中初始化了其他类，想想看这段代码的运行结果是什么？是打印“Hi,show me something”吗？

答案是这段代码不能运行，报 `StackOverflowError` 异常，栈（Stack）内存溢出。这是因为声明 `s` 变量时，调用了 `Son` 的无参构造函数，JVM 又默认调用了父类 `Father` 的无参构造函数，接着 `Father` 类又初始化了 `Other` 类，而 `Other` 类又调用了 `Son` 类，于是一个死循环就诞生了，直到栈内存被消耗完毕为止。

可能有读者会觉得这样的场景不可能在开发中出现，那我们来思考这样的场景：`Father` 是由框架提供的，`Son` 类是我们自己编写的扩展代码，而 `Other` 类则是框架要求的拦截类（`Interceptor` 类或者 `Handle` 类或者 `Hook` 方法），再来看看该问题，这种场景不可能出现吗？

那有读者可能要说了，这种问题只要系统一运行就会发现，不可能对项目产生影响。

那是因为我们在这里展示的代码比较简单，很容易一眼洞穿，一个项目中的构造函数可不止一两个，类之间的关系也不会这么简单的，要想瞥一眼就能明白是否有缺陷这对所有人员来说都是不可能完成的任务，解决此类问题的最好办法就是：不要在构造函数中声明初始化其他类，养成良好的习惯。

## 建议 36：使用构造代码块精炼程序

什么叫代码块（Code Block）？用大括号把多行代码封装在一起，形成一个独立的数据体，实现特定算法的代码集合即为代码块，一般来说代码块是不能单独运行的，必须要有运行主体。在 Java 中一共有四种类型的代码块：

### （1）普通代码块

就是在方法后面使用“{}”括起来的代码片段，它不能单独执行，必须通过方法名调用执行。

### （2）静态代码块

在类中使用 static 修饰，并使用“{}”括起来的代码片段，用于静态变量的初始化或对象创建前的环境初始化。

### （3）同步代码块

使用 synchronized 关键字修饰，并使用“{}”括起来的代码片段，它表示同一时间只能有一个线程进入到该方法块中，是一种多线程保护机制。

### （4）构造代码块

在类中没有任何的前缀或后缀，并使用“{}”括起来的代码片段。

我们知道，一个类至少有一个构造函数（如果没有，编译器会无私地为其创建一个无参构造函数），构造函数是在对象生成时调用的，那现在的问题来了：构造函数和构造代码块是什么关系？构造代码块是在什么时候执行的？在回答这个问题之前，我们先来看看编译器是如何处理构造代码块的，看如下代码：

```
public class Client {  
    {  
        // 构造代码块  
        System.out.println(" 执行构造代码块 ");  
    }  
  
    public Client() {  
        System.out.println(" 执行无参构造 ");  
    }  
  
    public Client(String _str) {  
        System.out.println(" 执行有参构造 ");  
    }  
}
```

这是一段非常简单的代码，它包含了构造代码块、无参构造、有参构造，我们知道代码块不具有独立执行的能力，那么编译器是如何处理构造代码块呢？很简单，编译器会把构造代码块插入到每个构造函数的最前端，上面的代码与如下代码等价：

```

public class Client {
    public Client() {
        System.out.println(" 执行构造代码块 ");
        System.out.println(" 执行无参构造 ");
    }

    public Client(String _str) {
        System.out.println(" 执行构造代码块 ");
        System.out.println(" 执行有参构造 ");
    }
}

```

每个构造函数的最前端都被插入了构造代码块，很显然，在通过 new 关键字生成一个实例时会先执行构造代码块，然后再执行其他代码，也就是说：构造代码块会在每个构造函数内首先执行（需要注意的是：构造代码块不是在构造函数之前运行的，它依托于构造函数的执行），明白了这一点，我们就可以把构造代码块应用到如下场景中：

#### （1）初始化实例变量（Instance Variable）

如果每个构造函数都要初始化变量，可以通过构造代码块来实现。当然也可以通过定义一个方法，然后在每个构造函数中调用该方法来实现，没错，可以解决，但是要在每个构造函数中都调用该方法，而这就是其缺点，若采用构造代码块的方式则不用定义和调用，会直接由编译器写入到每个构造函数中，这才是解决此类问题的绝佳方式。

#### （2）初始化实例环境

一个对象必须在适当的场景下才能存在，如果没有适当的场景，则就需要在创建对象时创建此场景，例如在 JEE 开发中，要产生 HTTP Request 必须首先建立 HTTP Session，在创建 HTTP Request 时就可以通过构造代码块来检查 HTTP Session 是否已经存在，不存在则创建之。

以上两个场景利用了构造代码块的两个特性：在每个构造函数中都运行和在构造函数中它会首先运行。很好地利用构造代码块的这两个特性不仅可以减少代码量，还可以让程序更容易阅读，特别是当所有的构造函数都要实现逻辑，而且这部分逻辑又很复杂时，这时就可以通过编写多个构造代码块来实现。每个代码块完成不同的业务逻辑（当然了，构造函数尽量简单，这是基本原则），按照业务顺序依次存放，这样在创建实例对象时 JVM 也就会按照顺序依次执行，实现复杂对象的模块化创建。

## 建议 37：构造代码块会想你所想

上一个建议中我们提议使用构造代码块来简化代码，并且也了解到编译器会自动把构造代码块插入到各个构造函数中，那我们接下来看看编译器是不是足够聪明，能够为我们解决真实的开发问题。有这样一个案例：统计一个类的实例数量。可能你要说了，这很简单，在

每个构造函数中加入一个对象计数器不就解决问题了吗？或者使用我们上一个建议介绍的，使用构造代码块也可以。确实如此，我们来看如下代码是否可行：

```
public class Client {
    public static void main(String[] args) {
        new Base();
        new Base("");
        new Base(0);
        System.out.println(" 实例对象数量: " + Base.getNumOfObjects());
    }
}

class Base{
    // 对象计数器
    private static int numOfObjects = 0;

    {
        // 构造代码块，计算产生对象数量
        numOfObjects++;
    }

    public Base(){
    }

    // 有参构造调用无参构造
    public Base(String _str){
        this();
    }

    // 有参构造不调用其他构造
    public Base(int _i){
    }

    // 返回在一个 JVM 中，创建了多少个实例对象
    public static int getNumOfObjects(){
        return numOfObjects;
    }
}
```

这段代码是可行的吗？能计算出实例对象的数量吗？哎，好像不对呀，如果编译器把构造代码块插入到各个构造函数中，那带有 String 形参的构造函数可就有问题，它会调用无参构造，那通过它生成 Base 对象时就会执行两次构造代码块：一次是由无参构造函数调用构造代码块，一次是执行自身的构造代码块，这样的话计算可就不准确了，main 函数实际在内存中产生了 3 个对象，但结果却会是 4。不过真是这样的吗？Are you sure？我们运行一下看看结果：

实例对象数量：3

非常遗憾，你错了，实例对象的数量还是 3，程序没有任何问题。奇怪吗？不奇怪，上



一个建议是说编译器会把构造代码块插入到每一个构造函数中，但是有一个例外的情况没有说明：如果遇到 `this` 关键字（也就是构造函数调用自身其他的构造函数时）则不插入构造代码块，对于我们的例子来说，编译器在编译时发现 `String` 形参的构造函数调用了无参构造，于是放弃插入构造代码块，所以只执行了一次构造代码块——结果就是如此。

那 Java 编译器为什么会这么聪明呢？这还要从构造代码块的诞生说起，构造代码块是为了提取构造函数的共同量，减少各个构造函数的代码而产生的，因此，Java 就很聪明地认为把代码块插入到没有 `this` 方法的构造函数中即可，而调用其他构造函数的则不插入，确保每个构造函数只执行一次构造代码块。

还有一点需要说明，读者千万不要以为 `this` 是特殊情况，那 `super` 也会类似处理了。其实不会，在构造代码块的处理上，`super` 方法没有任何特殊的地方，编译器只是把构造代码块插入到 `super` 方法之后执行而已，仅此不同。

---

**注意** 放心地使用构造代码块吧，Java 已经想你所想了。

---

## 建议 38：使用静态内部类提高封装性

Java 中的嵌套类（Nested Class）分为两种：静态内部类（也叫静态嵌套类，Static Nested Class）和内部类（Inner Class）。内部类我们介绍过很多了，现在来看看静态内部类。什么是静态内部类呢？是内部类，并且是静态（`static` 修饰）的即为静态内部类。只有在是静态内部类的情况下才能把 `static` 修饰符放在类前，其他任何时候 `static` 都是不能修饰类的。

静态内部类的形式很好理解，但是为什么需要静态内部类呢？那是因为静态内部类有两个优点：加强了类的封装性和提高了代码的可读性，我们通过一段代码来解释这两个优点，如下所示：

```
public class Person{
    // 姓名
    private String name;
    // 家庭
    private Home home;
    // 构造函数设置属性值
    public Person(String _name){
        name = _name;
    }
    /* home、name 的 getter/setter 方法省略 */

    public static class Home{
        // 家庭地址
        private String address;
        // 家庭电话
        private String tel;
```



```

        public Home(String _address,String _tel){
            address = _address;
            tel = _tel;
        }
        /* address、tel 的 getter/setter 方法省略 */
    }
}

```

其中，Person 类中定义了一个静态内部类 Home，它表示的意思是“人的家庭信息”，由于 Home 类封装了家庭信息，不用在 Person 类中再定义 homeAddre、homeTel 等属性，这就使封装性提高了。同时我们仅仅通过代码就可以分析出 Person 和 Home 之间的强关联关系，也就是说语义增强了，可读性提高了。所以在使用时就会非常清楚它要表达的含义：

```

public static void main(String[] args) {
    // 定义张三这个人
    Person p = new Person("张三");
    // 设置张三的家庭信息
    p.setHome(new Person.Home("上海","021"));
}

```

定义张三这个人，然后通过 Person.Home 类设置张三的家庭信息，这是不是就和我们真实世界的情形相同了？先登记人的主要信息，然后登记人员的分类信息。可能你又要问了，这和我们一般定义的类有什么区别呢？又有什么吸引人的地方呢？如下所示：

- ❑ 提高封装性。从代码位置上来讲，静态内部类放置在外部的类内，其代码层意义就是：静态内部类是外部类的子行为或子属性，两者直接保持着一定的关系，比如在我们的例子中，看到 Home 类就知道它是 Person 的 Home 信息。
- ❑ 提高代码的可读性。相关联的代码放在一起，可读性当然提高了。
- ❑ 形似内部，神似外部。静态内部类虽然存在于外部类内，而且编译后的类文件名也包含外部类（格式是：外部类+\$+内部类），但是它可以脱离外部类存在，也就是说我们仍然可以通过 new Home() 声明一个 Home 对象，只是需要导入“Person.Home”而已。

解释了这么多，读者可能会觉得外部类和静态内部类之间是组合关系（Composition）了，这是错误的，外部类和静态内部类之间有强关联关系，这仅仅表现在“字面”上，而深层次的抽象意义则依赖于类的设计。

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。

## 建议 39：使用匿名类的构造函数

阅读如下代码，看看是否可以编译：

```
public static void main(String[] args) {
    List l1 = new ArrayList();
    List l2 = new ArrayList();
    List l3 = new ArrayList();
    System.out.println(l1.getClass() == l2.getClass());
    System.out.println(l2.getClass() == l3.getClass());
    System.out.println(l1.getClass() == l3.getClass());
}
```

注意 ArrayList 后面的不同点：l1 变量后面什么都没有，l2 后面有一对 {}，l3 后面有 2 对嵌套的 {}，这段程序能不能编译呢？若能编译，那输出是多少呢？

答案是能编译，输出的是 3 个 false。l1 很容易解释，就是声明了 ArrayList 的实例对象，那 l2 和 l3 代表的是什么呢？

#### (1) l2=new ArrayList()

l2 代表的是一个匿名类的声明和赋值，它定义了一个继承于 ArrayList 的匿名类，只是没有任何的覆写方法而已，其代码类似于：

```
// 定义一个继承 ArrayList 的内部类
class Sub extends ArrayList{
}
// 声明和赋值
List l2 = new Sub();
```

#### (2) l3=new ArrayList({})

这个语句就有点怪了，还带了两对大括号，我们分开来解释就会明白了，这也是一个匿名类的定义，它的代码类似于：

```
// 定义一个继承 ArrayList 的内部类
class Sub extends ArrayList{
{
}
```

```

        // 初始化块
    }
}
// 声明和赋值
List l3 = new Sub();

```

看到了吧，就是多了一个初始化块而已，起到构造函数的功能。我们知道一个类肯定有一个构造函数，且构造函数的名称和类名相同，那问题来了：匿名类的构造函数是什么呢？它没有名字呀！很显然，初始化块就是它的构造函数。当然，一个类中的构造函数块可以是多个，也就是说可以出现如下代码：

```
List l3 = new ArrayList(){{}}{{}}{{}};
```

上面的代码是正确无误，没有任何问题的。现在清楚了：匿名函数虽然没有名字，但也是可以有构造函数的，它用构造函数块来代替，那上面的 3 个输出就很清楚了：虽然父类相同，但是类还是不同的。

## 建议 40：匿名类的构造函数很特殊

在上一个建议中我们讲到匿名类虽然没有名字，但可以有一个初始化块来充当构造函数，那这个构造函数是否就和普通的构造函数完全一样呢？我们来看一个例子，设计一个计算器，进行加减乘除运算，代码如下：

```

// 定义一个枚举，限定操作符
enum Ops {ADD, SUB}
class Calculator {
    private int i, j, result;
    // 无参构造
    public Calculator() {}
    // 有参构造
    public Calculator(int _i, int _j) {
        i = _i;
        j = _j;
    }
    // 设置符号，是加法运算还是减法运算
    protected void setOperator(Ops _op) {
        result = _op.equals(Ops.ADD)?i+j:i-j;
    }
    // 取得运算结果
    public int getResult(){
        return result;
    }
}

```

代码的意图是，通过构造函数输入两个 int 类型的数字，然后根据设置的操作符（加法

还是减法) 进行计算, 编写一个客户端调用:

```
public static void main(String[] args) {
    Calculator c1 = new Calculator(1,2) {
        {
            setOperator(Ops.ADD);
        }
    };
    System.out.println(c1.getResult());
}
```

这段匿名类的代码非常清晰: 接收两个参数 1 和 2, 然后设置一个操作符号, 计算其值, 结果是 3, 这毫无疑问, 但是这中间隐藏着一个问题: 带有参数的匿名类声明时到底是调用的哪一个构造函数呢? 我们把这段程序模拟一下:

```
// 加法计算
class Add extends Calculator {
    {
        setOperator(Ops.ADD);
    }
    // 覆写父类的构造方法
    public Add(int _i, int _j) {
    }
}
```

匿名类和这个 Add 类是等价的吗? 可能有人会说: 上面只是把匿名类增加了一个名字, 其他的都没有改动, 那肯定是等价的啦! 毫无疑问! 那好, 你再写个客户端调用 Add 类的方法看看。是不是输出结果为 0 (为什么是 0? 这很容易, 有参构造没有赋值)。这说明两者不等价, 不过, 原因何在呢?

原来是因为匿名类的构造函数特殊处理机制, 一般类 (也就是具有显式名字的类) 的所有构造函数默认都是调用父类的无参构造的, 而匿名类因为没有名字, 只能由构造代码块代替, 也就无所谓的有参和无参构造函数了, 它在初始化时直接调用了父类的同参数构造, 然后再调用了自己的构造代码块, 也就是说上面的匿名类与下面的代码是等价的:

```
// 加法计算
class Add extends Calculator {
    {
        setOperator(Ops.ADD);
    }
    // 覆写父类的构造方法
    public Add(int _i, int _j) {
        super(_i, _j);
    }
}
```

它首先会调用父类有两个参数的构造函数, 而不是无参构造, 这是匿名类的构造函数与

普通类的差别，但是这一点也确实鲜有人细细琢磨，因为它的处理机制符合习惯呀，我传递两个参数，就是希望先调用父类有两个参数的构造，然后再执行我自己的构造函数，而 Java 的处理机制也正是如此处理的！

## 建议 41：让多重继承成为现实

在 Java 中一个类可以多重实现，但不能多重继承，也就是说一个类能够同时实现多个接口，但不能同时继承多个类。但有时候我们确实需要继承多个类，比如希望拥有两个类的行为功能，就很难使用单继承来解决问题了（当然，使用多层继承是可以解决的）。幸运的是 Java 中提供的内部类可以曲折地解决此问题，我们来看一个案例，定义一个父亲、母亲接口，描述父亲强壮、母亲温柔的理想情形，代码如下：

```
// 父亲
interface Father{
    public int strong();
}
// 母亲
interface Mother{
    public int kind();
}
```

其中 strong 和 kind 的返回值表示强壮和温柔的指数，指数越高强壮度和温柔度也就越高，这与在游戏中设置人物的属性值是一样的。我们继续来看父亲、母亲这两个实现：

```
class FatherImpl implements Father{
    // 父亲的强壮指数是 8
    public int strong(){
        return 8;
    }
}

class MotherImpl implements Mother{
    // 母亲的温柔指数是 8
    public int kind(){
        return 8;
    }
}
```

父亲强壮指数是 8，母亲温柔指数也是 8，门当户对，那他们生的儿子、女儿一定更优秀了，我们先来看儿子类，代码如下：

```
class Son extends FatherImpl implements Mother{
    @Override
    public int strong(){
```

```

        // 儿子比父亲强壮
        return super.strong() + 1;
    }

    @Override
    public int kind(){
        return new MotherSpecial().kind();
    }

    private class MotherSpecial extends MotherImpl{
        public int kind(){
            // 儿子温柔指数降低了
            return super.kind() - 1;
        }
    }
}

```

儿子继承自父亲，变得比父亲更强壮了（覆写父类 strong 方法），同时儿子也具有母亲的优点，只是温柔指数降低了。注意看，这里构造了 MotherSpecial 类继承母亲类，也就是获得了母亲类的行为方法，这也是内部类的一个重要特性：内部类可以继承一个与外部类无关的类，保证了内部类的独立性，正是基于这一点，多重继承才会成为可能。MotherSpecial 的这种内部类叫做成员内部类（也叫做实例内部类，Instance Inner Class）。我们再来看看女儿类，代码如下：

```

class Daughter extends MotherImpl implements Father{

    @Override
    public int strong() {
        return new FatherImpl(){
            @Override
            public int strong() {
                // 女儿的强壮指数降低了
                return super.strong() - 2 ;
            }
        }.strong();
    }
}

```

女儿继承了母亲的温柔指数，同时又覆写父类的强壮指数，不多解释。注意看覆写的 strong 方法，这里是创建了一个匿名内部类（Anonymous Inner Class）来覆写父类的方法，以完成继承父亲行为的功能。

多重继承指的是一个类可以同时从多于一个的父类那里继承行为与特征，按照这个定义来看，我们的儿子类、女儿类都实现了从父亲类、母亲类那里所继承的功能，应该属于多重继承。这要完全归功于内部类，诸位在需要用到多重继承时，可以思考一下内部类。

在现实生活中，也确实存在多重继承的问题，上面的例子是说后人即继承了父亲也继承

了母亲的行为和特征，再比如我国的特产动物“四不像”（学名麋鹿），其外形“似鹿非鹿，似马非马，似牛非牛，似驴非驴”，这你要是想用单继承表示就麻烦了，如果用多继承则可以很好地解决问题：定义鹿、马、牛、驴四个类，然后建立麋鹿类的多个内部类，继承它们即可。

## 建议 42：让工具类不可实例化

Java 项目中使用的工具类非常多，比如 JDK 自己的工具类 `java.lang.Math`、`java.util.Collections` 等都是我们经常用到的。工具类的方法和属性都是静态的，不需要生成实例即可访问，而且 JDK 也做了很好的处理，由于不希望被初始化，于是就设置构造函数为 `private` 访问权限，表示除了类本身外，谁都不能产生一个实例，我们来看一下 `java.lang.Math` 代码：

```
public final class Math {  
    /**  
     * Don't let anyone instantiate this class.  
     */  
    private Math() {}  
}
```

之所以要将“Don't let anyone instantiate this class.”留下来，是因为 `Math` 的构造函数设置为 `private` 了：我就是一个工具类，我只想要其他类通过类名来访问，我不想你通过实例对象访问。这在平台型或框架型项目中已经足够了。但是如果已经告诉你不能这么做了，你还要生成一个 `Math` 实例来访问静态方法和属性（Java 的反射是如此的发达，修改个构造函数的访问权限易如反掌），那我就不保证正确性了，隐藏问题随时都有可能爆发！那我们在项目开发中有没有更好的限制办法呢？有，即不仅仅设置成 `private` 访问权限，还抛异常，代码如下：

```
public class UtilsClass {  
    private UtilsClass() {  
        throw new Error("不要实例化我！");  
    }  
}
```

如此做才能保证一个工具类不会实例化，并且保证所有的访问都是通过类名来进行的。需要注意一点的是，此工具类最好不要做继承的打算，因为如果子类可以实例化的话，那就要调用父类的构造函数，可是父类没有可以被访问的构造函数，于是问题就会出现。

---

**注意** 如果一个类不允许实例化，就要保证“平常”渠道都不能实例化它。

---



## 建议 43: 避免对象的浅拷贝

我们知道一个类实现了 Cloneable 接口就表示它具备了被拷贝的能力，如果再覆写 clone() 方法就会完全具备拷贝能力。拷贝是在内存中进行的，所以在性能方面比直接通过 new 生成对象要快很多，特别是在大对象的生成上，这会使性能的提升非常显著。但是对象拷贝也有一个比较容易忽略的问题：浅拷贝（Shadow Clone，也叫做影子拷贝）存在对象属性拷贝不彻底的问题。我们来看这样一段代码：

```
public class Client {
    public static void main(String[] args) {
        // 定义父亲
        Person f = new Person("父亲");
        // 定义大儿子
        Person s1 = new Person("大儿子", f);
        // 小儿子的信息是通过大儿子拷贝过来的
        Person s2 = s1.clone();
        s2.setName("小儿子");
        System.out.println(s1.getName() + " 的父亲是 " + s1.getFather().getName());
        System.out.println(s2.getName() + " 的父亲是 " + s2.getFather().getName());
    }
}

class Person implements Cloneable{
    // 姓名
    private String name;
    // 父亲
    private Person father;

    public Person(String _name){
        name = _name;
    }
    public Person(String _name, Person _parent){
        name = _name;
        father = _parent;
    }
    /*name 和 parent 的 getter/setter 方法省略*/

    // 拷贝的实现
    @Override
    public Person clone(){
        Person p = null;
        try {
            p = (Person) super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return p;
    }
}
```

程序中，我们描述了这样一个场景：一个父亲，有两个儿子，大小儿子同根同种，所以小儿子对象就通过拷贝大儿子对象来生成，运行输出的结果如下：

```
大儿子 的父亲是 父亲
小儿子 的父亲是 父亲
```

这很正确，没有问题。突然有一天，父亲心血来潮想让我大儿子去认个干爹，也就是大儿子的父亲名称需要重新设置一下，代码如下：

```
public static void main(String[] args) {
    // 定义父亲
    Person f = new Person(" 父亲 ");
    // 定义大儿子
    Person s1 = new Person(" 大儿子 ",f);
    // 小儿子的信息是通过大儿子拷贝过来的
    Person s2 = s1.clone();
    s2.setName(" 小儿子 ");
    // 认干爹
    s1.getFather().setName(" 干爹 ");
    System.out.println(s1.getName() + " 的父亲是 " + s1.getFather().getName());
    System.out.println(s2.getName() + " 的父亲是 " + s2.getFather().getName());
}
```

上面仅仅修改了加粗字体部分，大儿子重新设置了父亲名称，我们期望的输出是：将大儿子父亲的名称修改为干爹，小儿子的父亲名称保持不变。下面来检查一下结果是否如此：

```
大儿子 的父亲是 干爹
小儿子 的父亲是 干爹
```

怎么回事，小儿子的父亲也成了“干爹”？两个儿子都没有，岂不是要气死“父亲”了！出现这个问题的原因就在于 clone 方法，我们知道所有类都继承自 Object，Object 提供了一个对象拷贝的默认方法，即上面代码中的 super.clone 方法，但是该方法是有缺陷的，它提供的是一种浅拷贝方式，也就是说它并不会把对象的所有属性全部拷贝一份，而是有选择性的拷贝，它的拷贝规则如下：

#### (1) 基本类型

如果变量是基本类型，则拷贝其值，比如 int、float 等。

#### (2) 对象

如果变量是一个实例对象，则拷贝地址引用，也就是说此时新拷贝出的对象与原有对象共享该实例变量，不受访问权限的限制。这在 Java 中是很疯狂的，因为它突破了访问权限的定义：一个 private 修饰的变量，竟然可以被两个不同的实例对象访问，这让 Java 的访问权限体系情何以堪！

#### (3) String 字符串

这个比较特殊，拷贝的也是一个地址，是个引用，但是在修改时，它会从字符串池

(String Pool) 中重新生成新的字符串，原有的字符串对象保持不变，在此处我们可以认为 String 是一个基本类型。(有关字符串的知识详见第 4 章。)

明白了这三个规则，上面的例子就很清晰了，小儿子对象是通过拷贝大儿子产生的，其父亲都是同一个人，也就是同一个对象，大儿子修改了父亲名称，小儿子也就跟着修改了——于是，父亲的两个儿子都没了！其实要更正也很简单，clone 方法的代码如下：

```
public Person clone(){
    Person p = null;
    try {
        p = (Person) super.clone();
        p.setFather(new Person(p.getFather().getName()));
    } catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }
    return p;
}
```

然后再运行，小儿子的父亲就不会是“干爹”了。如此就实现了对象的深拷贝 (Deep Clone)，保证拷贝出来的对象自成一體，不受“母体”的影响，和 new 生成的对象没有任何区别。

---

**注意** 浅拷贝只是 Java 提供的一种简单拷贝机制，不便于直接使用。

---

## 建议 44：推荐使用序列化实现对象的拷贝

上一个建议说了对象的浅拷贝问题，实现 Cloneable 接口就具备了拷贝能力，那我们来思考这样一个问题：如果一个项目中有大量的对象是通过拷贝生成的，那我们该如何处理？每个类都写一个 clone 方法，并且还要深拷贝？想想看这是何等巨大的工作量呀，是否有更好的方法呢？

其实，可以通过序列化方式来处理，在内存中通过字节流的拷贝来实现，也就是把母对象写到一个字节流中，再从字节流中将其读出来，这样就可以重建一个新对象了，该新对象与母对象之间不存在引用共享的问题，也就相当于深拷贝了一个新对象，代码如下：

```
public class CloneUtils {
    // 拷贝一个对象
    @SuppressWarnings("unchecked")
    public static <T extends Serializable> T clone(T obj) {
        // 拷贝产生的对象
        T clonedObj = null;
        try {
            // 读取对象字节数据
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
```

```

        ObjectOutputStream oos = new ObjectOutputStream(baos);
        oos.writeObject(obj);
        oos.close();
        // 分配内存空间，写入原始对象，生成新对象
        ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());
        ObjectInputStream ois = new ObjectInputStream(bais);
        // 返回新对象，并做类型转换
        clonedObj = (T)ois.readObject();
        ois.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return clonedObj;
}
}

```

此工具类要求被拷贝的对象必须实现 `Serializable` 接口，否则是没办法拷贝的（当然，使用反射那是另外一种技巧），上一个建议中的例子只要稍微修改一下即可实现深拷贝，代码如下：

```

class Person implements Serializable{
    private static final long serialVersionUID = 1611293231L;
    /* 删除掉 clone 方法，其他代码保持不变 */
}

```

被拷贝的类只要实现 `Serializable` 这个标志性接口即可，不需要任何实现，当然 `serialVersionUID` 常量还是要加上去的，然后我们就可以通过 `CloneUtils` 工具进行对象的深拷贝了。用此方法进行对象拷贝时需要注意两点：

（1）对象的内部属性都是可序列化的

如果有内部属性不可序列化，则会抛出序列化异常，这会让调试者很纳闷：生成一个对象怎么会出现序列化异常呢？从这一点来考虑，也需要把 `CloneUtils` 工具的异常进行细化处理。

（2）注意方法和属性的特殊修饰符

比如 `final`、`static` 变量的序列化问题会被引入到对象拷贝中来（参考第 1 章），这点需要特别注意，同时 `transient` 变量（瞬态变量，不进行序列化的变量）也会影响到拷贝的效果。

当然，采用序列化方式拷贝时还有一个更简单的办法，即使用 Apache 下的 `commons` 工具包中的 `SerializationUtils` 类，直接使用更加简洁方便。

## 建议 45：覆写 `equals` 方法时不要识别不出自己

我们在写一个 `JavaBean` 时，经常会覆写 `equals` 方法，其目的是根据业务规则判断两个对象是否相等，比如我们写一个 `Person` 类，然后根据姓名判断两个实例对象是否相同，这在

DAO (Data Access Objects) 层是经常用到的。具体操作是先从数据库中获得两个 DTO (Data Transfer Object, 数据传输对象), 然后判断它们是否是相等的, 代码如下:

```
class Person{
    private String name;

    public Person(String _name){
        name = _name;
    }
    /*name 的 getter/setter 方法省略 */

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof Person){
            Person p = (Person) obj;
            return name.equalsIgnoreCase(p.getName().trim());
        }
        return false;
    }
}
```

覆写的 equals 做了多个校验, 考虑到从 Web 上传递过来的对象有可能输入了前后空格, 所以用 trim 方法剪切一下, 看看代码有没有问题, 我们写一个 main:

```
public static void main(String[] args) {
    Person p1 = new Person("张三");
    Person p2 = new Person("张三 ");

    List<Person> l =new ArrayList<Person>();
    l.add(p1);
    l.add(p2);
    System.out.println("列表中是否包含张三: "+l.contains(p1));
    System.out.println("列表中是否包含张三 : "+l.contains(p2));
}
```

上面的代码产生了两个 Person 对象 (注意 p2 变量中的那个张三后面有一个空格), 然后放到 List 中, 最后判断 List 是否包含了这两个对象。看上去没有问题, 应该打印出两个 true 才是, 但是结果却是:

```
列表中是否包含张三: true
列表中是否包含张三 : false
```

刚刚放到 list 中的对象竟然说没有, 这太让人失望了, 原因何在呢? List 类检查是否包含元素时是通过调用对象的 equals 方法来判断的, 也就是说 contains(p2) 传递进去, 会依次执行 p2.equals(p1)、p2.equals(p2), 只要有一个返回 true, 结果就是 true, 可惜的是比较结果都是 false, 那问题就出来了: 难道 p2.equals(p2) 也为 false 不成?

还真说对了，`p2.equals(p2)` 确实是 `false`，看看我们的 `equals` 方法，它把第二个参数进行了剪切！也就是说比较的是如下等式：

```
" 张三 ".equalsIgnoreCase(" 张三 ")
```

注意前面的“张三”是有空格的，那这个结果肯定是 `false` 了，错误也就此产生了。这是一个想做好事却办成了“坏事”的典型案列，它违背了 `equals` 方法的自反性原则：对于任何非空引用 `x`，`x.equals(x)` 应该返回 `true`。

问题知道了，解决也非常容易，只要把 `trim()` 去掉即可，注意解决的只是当前问题，该 `equals` 方法还存在其他问题。

## 建议 46: equals 应该考虑 null 值情景

继续上一建议的问题，我们解决了覆写 `equals` 的自反性问题，是不是就很完美了呢？再把 `main` 方法重构一下：

```
public static void main(String[] args) {
    Person p1 = new Person(" 张三 ");
    Person p2 = new Person(null);

    /* 其他部分没有任何修改，不再赘述 */
}
```

很小的改动，那运行结果是什么呢？是两个 `true` 吗？我们来看运行结果：

列表中是否包含张三: `true`

**Exception in thread "main" java.lang.NullPointerException**

竟然抛异常了！为什么 `p1` 就能在 `List` 中检查一遍，并且执行 `p1.equals` 方法，而到了 `p2` 就开始报错了呢？仔细分析一下程序，马上明白了：当执行到 `p2.equals(p1)` 时，由于 `p2` 的 `name` 是一个 `null` 值，所以调用 `name.equalsIgnoreCase` 方法时就会报空指针异常了！出现这种情形是因为覆写 `equals` 没有遵循对称性原则：对于任何引用 `x` 和 `y` 的情形，如果 `x.equals(y)` 返回 `true`，那么 `y.equals(x)` 也应该返回 `true`。

问题知道了，解决也很简单，增加 `name` 是否为空进行判断即可，修改后的 `equals` 代码如下：

```
public boolean equals(Object obj) {
    if(obj instanceof Person){
        Person p = (Person) obj;
        if(p.getName()==null || name==null){
            return false;
        }else{
            return name.equalsIgnoreCase(p.getName());
        }
    }
}
```

```

    }
}
return false;
}

```

## 建议 47：在 equals 中使用 getClass 进行类型判断

本节我们继续讨论覆写 equals 的问题。这次我们编写一个员工 Employee 类继承 Person 类，这很正常，员工也是人嘛，而且在 JEE 中 JavaBean 有继承关系也很常见，代码如下：

```

class Employee extends Person{
    private int id;
    /*id的getter/setter方法省略*/
    public Employee(String _name,int _id) {
        super(_name);
        id = _id;
    }

    @Override
    public boolean equals(Object obj) {
        if(obj instanceof Employee){
            Employee e = (Employee) obj;
            return super.equals(obj)&& e.getId() == id;
        }
        return false;
    }
}

```

员工类增加了工号 ID 属性，同时也覆写了 equals 方法，只有在姓名和 ID 号都相同的情况下才表示是同一个员工，这是为了避免在一个公司中出现同名同姓员工的情况。看看上面的代码，这里校验条件已经相当完备了，应该不会再出错了，那我们编写一个 main 方法来看看，代码如下：

```

public static void main(String[] args) {
    Employee e1 = new Employee("张三",100);
    Employee e2 = new Employee("张三",1001);
    Person p1 = new Person("张三");
    System.out.println(p1.equals(e1));
    System.out.println(p1.equals(e2));
    System.out.println(e1.equals(e2));
}

```

上面定义了 2 个员工和 1 个社会闲杂人员，虽然他们同名同姓，但肯定不是同一个，输出应该都是 false，那我们看看运行结果：

```

true

```



```
true
false
```

很不给力嘛，p1 竟然等于 e1，也等于 e2，为什么不是同一个类的两个实例竟然也会相等呢？这很简单，因为 p1.equals(e1) 是调用父类 Person 的 equals 方法进行判断的，它使用 instanceof 关键字检查 e1 是否是 Person 的实例，由于两者存在继承关系，那结果当然是 true 了，相等也就没有任何问题了，但是反过来就不成立了，e1 或 e2 可不等于 p1，这也是违反对称性原则的一个典型案例。

更玄的是 p1 与 e1、e2 相等，但 e1 竟然与 e2 不相等，似乎一个简单的等号传递都不能实现。这才是我们要分析的真正重点：e1.equals(e2) 调用的是子类 Employee 的 equals 方法，不仅仅要判断姓名相同，还要判断工号是否相同，两者工号是不同的，不相等也是自然的了。等式不传递是因为违反了 equals 的传递性原则，传递性原则是指对于实例对象 x、y、z 来说，如果 x.equals(y) 返回 true，y.equals(z) 返回 true，那么 x.equals(z) 也应该返回 true。

这种情况发生的关键是父类使用了 instanceof 关键字，它是用来判断是否是一个类的实例对象的，这很容易让子类“钻空子”。想要解决也很简单，使用 getClass 来代替 instanceof 进行类型判断，Person 类的 equals 方法修改后如下所示：

```
public boolean equals(Object obj) {
    if(obj!=null && obj.getClass() == this.getClass()){
        Person p = (Person) obj;
        if(p.getName()==null || name==null){
            return false;
        }else{
            return name.equalsIgnoreCase(p.getName());
        }
    }
    return false;
}
```

当然，考虑到 Employee 也有可能被继承，也需要把它的 instanceof 修改为 getClass。总之，在覆写 equals 时建议使用 getClass 进行类型判断，而不要使用 instanceof。

## 建议 48：覆写 equals 方法必须覆写 hashCode 方法

覆写 equals 方法必须覆写 hashCode 方法，这条规则基本上每个 Javaer 都知道，这也是 JDK API 上反复说明的，不过为什么要这样做呢？这两个方法之间有什么关系呢？本建议就来解释该问题，我们先来看如下代码：

```
public static void main(String[] args) {
    // Person 类的实例作为 Map 的 key
    Map<Person, Object> map = new HashMap<Person, Object>() {
```

```

        {
            put(new Person(" 张三 "), new Object());
        }
    };
    // Person 类的实例作为 List 的元素
    List<Person> list = new ArrayList<Person>() {
        {
            add(new Person(" 张三 "));
        }
    };
    // 列表中是否包含
    boolean b1 = list.contains(new Person(" 张三 "));
    // Map 中是否包含
    boolean b2 = map.containsKey(new Person(" 张三 "));
}

```

代码中的 Person 类与上一建议相同，equals 方法完美无缺。在这段代码中，我们在声明时直接调用方法赋值，这其实也是一个内部匿名类的操作（下一个建议会详细说明）。现在的问题是 b1 和 b2 这两个 boolean 值是否都为 true？

我们先来看 b1，Person 类的 equals 覆写了，不再判断两个地址是否相等，而是根据人员的姓名来判断两个对象是否相等，所以不管我们的 new Person（“张三”）产生了多少个对象，它们都是相等的。把“张三”对象放入 List 中，再检查 List 中是否包含，那结果肯定是 true 了。

接着来看 b2，我们把张三这个对象作为 Map 的键（Key），放进去的对象是张三，检查的对象还是张三，那应该和 List 的结果相同了，但是很遗憾，结果是 false。原因何在呢？

原因就是 HashMap 的底层处理机制是以数组的方式保存 Map 条目（Map Entry）的，这其中的关键就是这个数组下标的处理机制：依据传入元素 hashCode 方法的返回值决定其数组的下标，如果该数组位置上已经有了 Map 条目，且与传入的键值相等则不处理，若不相等则覆盖；如果数组位置没有条目，则插入，并加入到 Map 条目的链表中。同理，检查键是否存在也是根据哈希码确定位置，然后遍历查找键值的。

接着深入探讨，那对象元素的 hashCode 方法返回的是什么呢？它是一个对象的哈希码，是由 Object 类的本地方法生成的，确保每个对象有一个哈希码（这也是哈希算法的基本要求：任意输入 k，通过一定算法 f(k)，将其转换为非可逆的输出，对于两个输入 k1 和 k2，要求若 k1=k2，则必须 f(k1)=f(k2)，但也允许 k1 ≠ k2，f(k1)=f(k2) 的情况存在）。

那回到我们的例子上，由于我们没有重写 hashCode 方法，两个张三对象的 hashCode 方法返回值（也就是哈希码）肯定是不相同的了，在 HashMap 的数组中也就找不到对应的 Map 条目了，于是就返回了 false。

问题清楚了，修改也非常简单，重写一下 hashCode 方法即可，代码如下：

```

class Person {

```

```

/* 其他代码相同，不再赘述 */
@Override
public int hashCode() {
    return new HashCodeBuilder().append(name).hashCode();
}
}

```

其中 `HashCodeBuilder` 是 `org.apache.commons.lang.builder` 包下的一个哈希码生成工具，使用起来非常方便，诸位可以直接在项目中集成。（为什么不直接写 `hashCode` 方法？因为哈希码的生成有很多种算法，自己写麻烦，事儿又多，所以采用拿来主义是最好的方法。）

## 建议 49：推荐覆写 `toString` 方法

为什么要覆写 `toString` 方法，这个问题很简单，因为 Java 提供的默认 `toString` 方法不友好，打印出来看不懂，不覆写不行，看这样一段代码：

```

public class Client {
    public static void main(String[] args) {
        System.out.println(new Person("张三"));
    }
}

class Person{
    private String name;

    public Person(String _name){
        name = _name;
    }
    /*name 的 getter/setter 方法省略 */
}

```

输出的结果是：`Person@1fc4bec`。如果机器不同，`@` 后面的内容也会不同，但格式都是相同的：类名 + `@` + `hashCode`，这玩意就是给机器看的，人哪能看得懂呀！这就是因为我们没有覆写 `Object` 类的 `toString` 方法的缘故，修改一下，代码如下所示：

```

public String toString(){
    return String.format("%s.name=%s",this.getClass(),name);
}

```

如此就可以在需要的时候输出可调试信息了，而且也非常友好，特别是在 `Bean` 流行的项目中（一般的 `Web` 项目就是这样），有了这样的输出才能更好的 `debug`，否则查找错误就如海底捞针呀！当然，当 `Bean` 的属性较多时，自己实现就不可取了，不过可以使用 `apache` 的 `commons` 工具包中的 `ToStringBuilder` 类，简洁、实用又方便。

可能有读者要说了，为什么通过 `println` 方法打印一个对象会调用 `toString` 方法？那是源

于 println 的实现机制：如果是一个原始类型就直接打印，如果是一个类类型，则打印出其 toString 方法的返回值，如此而已！

## 建议 50：使用 package-info 类为包服务

Java 中有一个特殊的类：package-info 类，它是专门为本包服务的，为什么说它特殊呢？主要体现在 3 个方面：

### （1）它不能随便被创建

在一般的 IDE 中，Eclipse、package-info 等文件是不能随便被创建的，会报 “Type name is not valid” 错误，类名无效。在 Java 变量定义规范中规定如下字符是允许的：字母、数字、下划线，以及那个不怎么常用的 \$ 符号，不过中划线可不在之列，那怎么创建这个文件呢？很简单，用记事本创建一个，然后拷贝进去再改一下就成了，更直接的办法就是从别的项目中拷贝过来。

### （2）它服务的对象很特殊

一个类是一类或一组事物的描述，比如 Dog 这个类，就是描述 “旺财” 的，那 package-info 这个类是描述什么的呢？它总要有个被描述或被陈述的对象吧，它是描述和记录本包信息的。

### （3）package-info 类不能有实现代码

package-info 类再怎么特殊也是一个类，也会被编译成 package-info.class，但是在 package-info.java 文件里不能声明 package-info 类。

package-info 类还有几个特殊的地方，比如不可以继承，没有接口，没有类间关系（关联、组合、聚合等）等，不再赘述，Java 中既然允许存在这么一个特殊的类，那肯定有其特殊的作用了，我们来看看它的作用，主要表现在以下三个方面：

### （1）声明友好类和包内访问常量

这个比较简单，而且很实用，比如一个包中有很多内部访问的类或常量，就可以统一放到 package-info 类中，这样很方便，而且便于集中管理，可以减少友好类到处游走的情况，代码如下：

```
// 这里是包类，声明一个包使用的公共类
class PkgClass{
    public void test(){ }
}
// 包常量，只允许包内访问
class PkgConst{
    static final String PACKAGE_CONST="ABC";
}
```

注意以上代码是存放在 package-info.java 中的，虽然它没有编写 package-info 的实现，但是 package-info.class 类文件还是会生成。通过这样的定义，我们把一个包需要的类和常量都放置在本包下，在语义上和习惯上都能让程序员更适应。

### (2) 为在包上标注注解提供便利

比如我们要写一个注解 (Annotation)，查看一个包下的所有对象，只要把注解标注到 package-info 文件中即可，而且在很多开源项目也采用了此方法，比如 Struts2 的 @namespace、Hibernate 的 @FilterDef 等。

### (3) 提供包的整体注释说明

如果是分包开发，也就是说一个包实现了一个业务逻辑或功能点或模块或组件，则该包需要有一个很好的说明文档，说明这个包是做什么用的，版本变迁历史，与其他包的逻辑关系等，package-info 文件的作用在此就发挥出来了，这些都可以直接定义到此文件中，通过 javadoc 生成文档时，会把这些说明作为包文档的首页，让读者更容易对该包有一个整体的认识。当然在这点上它与 package.htm 的作用是相同的，不过 package-info 可以在代码中维护文档的完整性，并且可以实现代码与文档的同步更新。

解释了这么多，总结成一句话：在需要用到包的地方，就可以考虑一下 package-info 这个特殊类，也许能起到事半功倍的作用。

## 建议 51：不要主动进行垃圾回收

很久很久以前，在 Java 1.1 的年代里，我们经常会看到 System.gc 这样的调用——主动对垃圾进行回收。不过，在 Java 知识深入人心后，这样的代码就逐渐销声匿迹了——这是好现象，因为主动进行垃圾回收是一个非常危险的动作。

之所以危险，是因为 System.gc 要停止所有的响应 (Stop the world)，才能检查内存中是否有可回收的对象，这对一个应用系统来说风险极大，如果是一个 Web 应用，所有的请求都会暂停，等待垃圾回收器执行完毕，若此时堆内存 (Heap) 中的对象少的话则还可以接受，一旦对象较多 (现在的 Web 项目是越做越大，框架、工具也越来越多，加载到内存中的对象当然也就更多了)，那这个过程就非常耗时了，可能 0.01 秒，也可能是 1 秒，甚至是 20 秒，这就会严重影响到业务的正常运行。

例如，我们写这样一段代码：new String("abc")，该对象没有任何引用，对 JVM 来说就是个垃圾对象。JVM 的垃圾回收器线程第一次扫描 (扫描时间不确定，在系统不繁忙的时候执行) 时把它贴上一个标签，说“你是可以被回收的”，第二次扫描时才真正地回收该对象，并释放内存空间，如果我们直接调用 System.gc，则是在说“嗨，你，那个垃圾回收器过来检查一下有没有垃圾对象，回收一下”。瞧瞧看，程序主动招来了垃圾回收器，这意味着正在运行着的系统要让出资源，以供垃圾回收器执行，想想看吧，它会把所有的对象都检查一遍，然后处理掉那些垃圾对象。注意哦，是检查每个对象。

不要调用 System.gc，即使经常出现内存溢出也不要调用，内存溢出是可分析的，是可以查找出原因的，GC 可不是一个好招数！