No.1：重复代码的提炼

重复**代码**是重构收效最大的手法之一，进行这项重构的原因不需要多说。它有很多很明显的好处，比如总**代码**量大大减少，维护方便，**代码**条理更加清晰易读。

它的重点就在于寻找**代码**当中完成某项子功能的重复**代码**，找到以后请毫不犹豫将它移动到合适的方法当中，并存放在合适的类当中。

小实例

```
class BadExample { public void someMethod1(){ //code System.out.println("重复代码");/* 重复代码块 */
//code } public void someMethod2(){ //code System.out.println("重复代码");/* 重复代码块 */ //code } }
/* ----------------------分割线---------------------- */ class GoodExample { public void
someMethod1(){ //code someMethod3(); //code } public void someMethod2(){ //code someMethod3();
//code } public void someMethod3(){ System.out.println("重复代码");/* 重复代码块 */ } }
```

No.2：冗长方法的分割

有关冗长方法的分割，其实有时候与重复**代码**的提炼是有着不可分割的关系的，往往在我们提炼重复**代码**的过程中，就不知不觉的完成了对某一个超长方法的分割。倘若在你提炼了大部分的重复**代码**之后，某一些冗长方法依然留存，此时就要静下心来专门处理这些冗长方法了。

这其中有一点是值得注意的，由于我们在分割一个大方法时，大部分都是针对其中的一些子功能分割，因此我们需要给每一个子功能起一个恰到好处的方法名，这很重要。可以说，能否给方法起一个好名字，有时候能体现出一个程序猿的大致水准。

小实例

```
class BadExample { public void someMethod(){ //function[1] //function[2] //function[3] } } /* -----
----------------分割线---------------------- */ class GoodExample { public void someMethod(){
function1(); function2(); function3(); } private void function1(){ //function[1] } private void
function2(){ //function[2] } private void function3(){ //function[3] } }
```

No.3：嵌套条件分支的优化（1）

大量的嵌套条件分支是很容易让人望而却步的**代码**，我们应该极力避免这种**代码**的出现。尽管结构化原则一直在说一个函数只能有一个出口，但是在这么大量的嵌套条件分支下，让我们忘了这所谓的规则吧。

有一个专业名词叫卫语句，可以治疗这种恐怖的嵌套条件语句。它的核心思想是，将不满足某些条件的情况放在方法前面，并及时跳出方法，以免对后面的判断造成影响。经过这项手术的**代码**看起来会非常的清晰，下面 LZ 就给各位举一个经典的例子，各位可以自行评判一下这两种方式，哪个让你看起来更清晰一点。

小实例

```
class BadExample { public void someMethod(Object A,Object B){ if (A != null) { if (B != null) {
//code[1] }else { //code[3] } }else { //code[2] } } } /* ----------------------分割线----------------
------ */ class GoodExample { public void someMethod(Object A,Object B){ if (A == null) { //code[2]
return; } if (B == null) { //code[3] return; } //code[1] } }
```

No.4：嵌套条件分支的优化（2）

此处所说的嵌套条件分支与上面的有些许不同，它无法使用卫语句进行优化，而应该是将条件分支合并，以此来达到**代码**清晰的目的。由这两条也可以看出，嵌套条件分支在编码当中应当尽量避免，它会大大降低**代码**的可读性。

下面请尚且不明觉厉的猿友看下面这个典型的小例子。

小实例

```
class BadExample { public void someMethod(Object A,Object B){ if (A != null) { if (B != null) {
//code } } } } /* ----------------------分割线---------------------- */ class GoodExample { public
void someMethod(Object A,Object B){ if (A != null && B != null) { //code } } }
```

No.5：去掉一次性的临时变量

(《重构》P122)

生活当中我们都经常用一次性筷子，这无疑是对树木的摧残。然而在程序当中，一次性的临时变量不仅是对性能上小小的摧残，更是对**代码**可读性的亵渎。因此我们有必要对一些一次性的临时变量进行手术。

小实例

```
class BadExample { private int i; public int someMethod(){ int temp = getVariable(); return temp *
100; } public int getVariable(){ return i; } } /* ----------------------分割线----------------------
*/ class GoodExample { private int i; public int someMethod(){ return getVariable() * 100; } public
int getVariable(){ return i; } }
```

No.6：消除过长参数列表

对于一些传递了大批参数的方法，对于追求**代码**整洁的程序猿来说，是无法接受的。我们可以尝试将这些参数封装成一个对象传递给方法，从而去除过长的参数列表。大部分情况下，当你尝试寻找这样一个对象的时候，它往往已经存在了，因此绝大多

数情况下，我们并不需要做多余的工作。
小实例
```
class BadExample { public void someMethod(int i,int j,int k,int l,int m,int n){ //code } } /* ----------分割线---------- */ class GoodExample { public void someMethod(Data data){ //code } } class Data{ private int i; private int j; private int k; private int l; private int m; private int n;     //getter&&setter }
```
No.7：提取类或继承体系中的常量
这项重构的目的是为了**消除**一些魔数或者是字符串常量等等，魔数所带来的弊端自不用说，它会让人对程序的意图产生迷惑。而对于字符串等类型的常量的**消除**，更多的好处在于维护时的方便。因为我们只需要修改一个常量，就可以完成对程序中所有使用该常量的**代码**的修改。
顺便提一句，与此类情况类似并且最常见的，就是 Action 基类中，对于 INPUT、LIST、SUCCESS 等这些常量的提取。
小实例
```
class BadExample { public void someMethod1(){ send("您的操作已成功！"); } public void someMethod2(){ send("您的操作已成功！"); } public void someMethod3(){ send("您的操作已成功！"); } private void send(String message){ //code } }
/* ----------分割线---------- */
class GoodExample { protected static final String SUCCESS_MESSAGE = "您的操作已成功！"; public void someMethod1(){ send(SUCCESS_MESSAGE); } public void someMethod2(){ send(SUCCESS_MESSAGE); } public void someMethod3(){ send(SUCCESS_MESSAGE); } private void send(String message){ //code } }
```
No.8：让类提供应该提供的方法
很多时候，我们经常会操作一个类的大部分属性，从而得到一个最终我们想要的结果。这种时候，我们应该让这个类做它该做的事情，而不应该让我们替它做。而且大部分时候，这个过程最终会成为重复**代码**的根源。
小实例
```
class BadExample { public int someMethod(Data data){ int i = data.getI(); int j = data.getJ(); int k = data.getK(); return i * j * k; } public static class Data{ private int i; private int j; private int k; public Data(int i, int j, int k) { super(); this.i = i; this.j = j; this.k = k; } public int getI() { return i; } public int getJ() { return j; } public int getK() { return k; } } }
/* ----------分割线---------- */
class GoodExample { public int someMethod(Data data){ return data.getResult(); } public static class Data{ private int i; private int j; private int k; public Data(int i, int j, int k) { super(); this.i = i; this.j = j; this.k = k; } public int getI() { return i; } public int getJ() { return j; } public int getK() { return k; } public int getResult(){ return i * j * k; } } }
```
No.9：拆分冗长的类
这项技巧其实也是属于非常实用的一个技巧，只不过由于它的难度相对较高，因此被 LZ 排在了后面。针对这个技巧，LZ 很难像上面的技巧一样，给出一个即简单又很容易说明问题的小例子，因为它已经不仅仅是小手段了。
大部分时候，我们拆分一个类的关注点应该主要集中在类的属性上面。拆分出来的两批属性应该在逻辑上是可以分离的，并且在**代码**当中，这两批属性的使用也都分别集中于某一些方法当中。如果实在有一些属性同时存在于拆分后的两批方法内部，那么可以通过参数传递的方式解决这种依赖。
类的拆分是一个相对较大的工程，毕竟一个大类往往在程序中已经被很多类所使用着，因此这项重构的难度相当之大，一定要谨慎，并做好足够的测试

No.10：提取继承体系中重复的属性与方法到父类
这项技巧大部分时候需要足够的判断力，很多时候，这其实是在向模板方法模式迈进的过程。它的实例 LZ 这里无法给出，原因是因为它的小实例会毫无意义，无非就是子类有一样的属性或者方法，然后删除子类的重复属性或方法放到父类当中。
往往这一类重构都不会是小工程，因此这一项重构与第九种类似，都需要足够的谨慎与测试。而且需要在你足够确认，这些提取到父类中的属性或方法，应该是子类的共性的时候，才可以使用这项技巧。

-