DerbySoft

Refactoring

PAN ZHU

内容大纲

1. 什么是重构 (what)
2. 何时进行重构 (when)
3. 如何进行重构 (how)
4. 一些重要常用的重构方法
5. 通过重构来引入模式

什么是重构

- 重构就是在不改变外部行为的条件下对现有代码进行修改的过程，也就是对如何如何做而不是做什么进行修改。
- 重构的目的是改变内部结构，提高可理解性，降低修改成本。

何时进行重构

在下列三种情况下我们需要进行重构

1. 存在重复的时候。 （三次法则）
2. 当我们觉得代码或代码所表达的意图不清晰的时候。
3. 当我们察觉到代码有 bad smells 的时候。

下面介绍几种常见且比较重要的 bad smells

何时进行重构-bad smells（注释）

- 注释：大多数注释存在的理由都是用来弥补拙劣代码的不足（对外接口注释除外）。如果觉得有必要编写一条注释的话，首先考虑重构或重写代码。

下面看看一个注释的例子：

何时进行重构- bad smells（注释）

这个是我们dswitch标准委员会的标准DTO里面的RequestHeader :

```
private RequestHeader createHeader() {
        RequestHeader header = new RequestHeader();
        //create source
        SourceDTO source = new SourceDTO();
        UniqueIDDTO sourceID = new UniqueIDDTO();
        sourceID.setType(UniqueIDType.HOTEL);
        sourceID.setId(SOURCE_ID);
        source.setUniqueID(sourceID);
        header.setSource(source);

        //create requestor
        RequestorDTO requestor = new RequestorDTO();
        UniqueIDDTO uniqueId = new UniqueIDDTO();
        uniqueId.setType(UniqueIDType.TRAVEL_AGENCY);
        uniqueId.setId(CHANNEL_PASSPORT);
        requestor.setUniqueID(uniqueId);
        header.setRequestor(requestor);

        header.setTaskId(TASK_ID);
        return header;
    }
```

何时进行重构- bad smells（数据类）

- 数据类：典型的数据类只包含数据而没有丝毫行为。只有一些get/set方法。

何时进行重构- bad smells（数据类）

一个简单的数据类例子：

```
public class Point {

    private int x;

    private int y;

    public Point() {
        this(0, 0);
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    //set/get方法省略
}
```

何时进行重构- bad smells（数据类）

还有一个龌龊+找抽型的版本：

```
public class Point {

    public int x;

    public int y;

    public Point() {
        this(0, 0);
    }

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    //set/get方法省略
}
```

何时进行重构- bad smells（数据类）

- 如果我们在代码中发现纯数据类了，我们就应该四处查看一下代码，找出操作这些数据类的实例类，要么把数据移到实例类中，要么把行为移到数据类中。

下面看一个使用数据类的例子:

何时进行重构- bad smells（数据类）


```
public class Shape {

    private Point center;

    public Shape() {
        center = new Point();
    }

    public void translate(int x, int y) {
        center.setX(center.getX() + x);
        center.setY(center.getY() + y);
    }

}
```

这里面的translate()中的行为应该属于Point：

何时进行重构- bad smells（数据类）

重构后的代码：

```
public class Shape {
    private Point center;
    public Shape() {
        center = new Point();
    }
    public void translate(int x, int y) {
        center.translate(x, y);
    }
}

public class Point {
    //……
    public void translate(int x, int y) {
        this.x += x;
        this.y += y;
    }
}
```

何时进行重构- bad smells（代码重复）

- 代码重复：这个是性质最恶劣的味道。各种形式的重复代码是优质代码的最大敌人。必须清楚干净。出现重复的情况有些是显而易见的，有些则不是很容易就能发现的。

  出现重复的最简单形式就是同一个表达式出现在多个方法中。这个也是大家最常见的重复例子。

何时进行重构- bad smells（代码重复）

举一个信息重复的例子：

```
public class MovieList {

    private int numberOfMovies = 0;

    private Collection movies = new ArrayList();

    public int size() {
        return movies.size();
    }

    public void add(Movie movie) {
        movies.add(movie);
        numberOfMovies++;
    }

}
```

何时进行重构- bad smells（交往不当）

- 交往不当：这个是指一个类对另一个类的内部细节知道得太多。

  遇到这种情况，我们就需要对方法进行迁移，让那些彼此了解的代码在一个位置。

何时进行重构- bad smells（交往不当）

下面看一个例子，这个方法掌握了关于Movie类结构的一切信息，
如果Movie的结构方式变化，那么像这样的方法就需要跟着改变：

```
public class MovieList {
    public void writeTo(Writer destination) {
        Iterator movieIterator = movies.iterator();
        while (movieIterator.hasNext()) {
            Movie movie = (Movie) movieIterator.next();
            destination.write(movie.getName());
            destination.write("|");
            destination.write(movie.getCategory());
            destination.write("|");
            destination.write(movie.getPersonator());
            destination.write("\n");
        }
    }
}
```

何时进行重构- bad smells（交往不当）

解决办法：把处理单个Movie输出的有关部分转移到Movie类中

```
public class MovieList {
    public void writeTo(Writer destination) {
        Iterator movieIterator = movies.iterator();
        while (movieIterator.hasNext()) {
            Movie movie = (Movie) movieIterator.next();
            movie.writeTo(destination);
        }
    }
}

public class Movie {
    //.....
    public void writeTo(Writer destination) {
        destination.write(movie.getName());
        destination.write("|");
        destination.write(movie.getCategory());
        destination.write("|");
        destination.write(movie.getPersonator());
        destination.write("\n");
    }
}
```

何时进行重构- bad smells（交往不当）

- 对继承来说也会出现同样的问题，即子类过多的了解父类的实现细节，超出它们所应该了解的。这种情况我们可以把继承换成委托(delegation)或者将父类的具体细节设为私有来解耦。

何时进行重构- bad smells（类尺寸过大）

- 类的尺寸过大：如果我们发现代码中存在比较大的类时，就应该仔细检查一下了。看看是什么原因导致它过大。想要完成的工作太多？是了解的东西太多？条件判断导致的？针对不同的情况可以采用不同的重构方法和策略。

可以用checkstyle方便的检查出代码中是不是存在类过大的情况。

何时进行重构- bad smells（懒惰类）

- 懒惰类：这个是与大尺寸类刚好相反的情况，懒惰类不能充分体现其价值，可以和别的适当的类加以合并。

何时进行重构- bad smells（方法过长）

- 方法过长：多长的代码算长了？如果看行数的话，任何超过20行的代码都算长。如果看职责的话，任何完成超出一件工作的方法都算长。

通过将功能内聚的代码块拆分到各自独立的方法中，可以消减方法的尺寸并使代码更容易理解。

何时进行重构- bad smells（switch 语句）

- switch 语句：不加约束的使用switch语句意味着没有深入理解面向对象的原则。switch语句的结果往往导致霰弹式的修改。我们常常可以使用多态以一种更好的方式来代替switch语句。

何时进行重构- bad smells（霰弹式的修改）

- 霰弹式的修改(shotgun surgery)：这个并不算是源自代码的味道，更是出于我们与代码打交道的方式。当我们要修改某一处功能，却发现需要改动多个不同的classes的代码，这就表明了出现了这种问题，这注定要成为灾难。常见的例子就是需要在多个方法的switch语句中添加一条子句。

何时进行重构- bad smells（发散式变化）

- 发散式变化(divergent change)：如果某个class经常因为不同的原因在不同的方向上发生变化，就表明发散式变化出现了。
- 霰弹式修改是指一种变化引发多个classes修改，发散式变化是指一个class受多种变化的影响。这两种情况下都需要重构代码，达到变化与修改一一对应。

如何进行重构

- 准备好自动测试环境，这些测试能在我们执行重构时及时的向我们反馈重构是否伤到原有行为。
- 小步前进，在每一步重构完成时运行测试。以便于最快的检查重构的有效性。如果失败，必须恢复到正确代码，再尝试进一步的重构。
- 擅于使用优良的重构工具可以大大减少我们重构所需的时间。

一些重要常用的重构方法

- 重构方法很多，仅《Refactoring:Improving the Design of Existing Code》中就讲述了近百种重构方法，还有很多重构方法没有列入。下面将介绍十三种比较重要且常用重构方法。

重构方法- 提取方法(Extract method)

- 提取方法：当一个方法太长或逻辑过于复杂而不易理解时，我们可以将其中的某些部分提取出来而形成各自独立的方法。

重构方法- 提取方法(Extract method)

来看看前面的dswitch标准DTO的例子：

```
private RequestHeader createHeader() {
    RequestHeader header = new RequestHeader();
    //create source
    SourceDTO source = new SourceDTO();
    UniqueIDDTO sourceID = new UniqueIDDTO();
    sourceID.setType(UniqueIDType.HOTEL);
    sourceID.setId(SOURCE_ID);
    source.setUniqueID(sourceID);
    header.setSource(source);

    //create requestor
    RequestorDTO requestor = new RequestorDTO();
    UniqueIDDTO uniqueId = new UniqueIDDTO();
    uniqueId.setType(UniqueIDType.TRAVEL_AGENCY);
    uniqueId.setId(CHANNEL_PASSPORT);
    requestor.setUniqueID(uniqueId);

    header.setRequestor(requestor);
    header.setTaskId(TASK_ID);
    return header;
}
```

重构方法- 提取方法(Extract method)

对它应用了Extract method之后如下：

```
private RequestHeader createHeader() {
    RequestHeader header = new RequestHeader();
    header.setSource(createSource());
    header.setRequestor(createRequestor());
    header.setTaskId(TASK_ID);
    return header;
}
private RequestorDTO createRequestor() {
    RequestorDTO requestor = new RequestorDTO();
    UniqueIDDTO uniqueId = new UniqueIDDTO();
    uniqueId.setType(UniqueIDType.TRAVEL_AGENCY);
    uniqueId.setId(CHANNEL_PASSPORT);
    requestor.setUniqueID(uniqueId);
    return requestor;
}
private SourceDTO createSource() {
    SourceDTO source = new SourceDTO();
    UniqueIDDTO sourceID = new UniqueIDDTO();
    sourceID.setType(UniqueIDType.HOTEL);
    sourceID.setId(SOURCE_ID);
    source.setUniqueID(sourceID);
    return source;
}
```

重构方法- 用委托来代替继承

- 用委托来代替继承(Replace Inheritance witch Delegation)：我们应当只在子类对超类进行扩展而不仅仅是覆写超类的部分功能时，才使用继承。如果只是为了重用超类的某些功能的话，应该用委托来代替继承。

重构方法- 用委托来代替继承

范例，假如有个“长方形”类别：

```
public class Rectangle {

    private double length;

    private double width;

    public double area() {
        return width * length;
    }

    public void setWidth(double width) {
        this.width = width;
    }

    public void setLength(double length) {
        this.length = length;
    }
}
```

重构方法- 用委托来代替继承

接着想设计个“正方形”类别，很多人会想到使用继承：

```
public Square extends Rectangle {

}
```

多好啊，什么都不用干，就可重复使用Rectangle的area()方法了。

这样会有什么问题呢？

重构方法- 用委托来代替继承

这个继承结构有个不足之处，Square从Rectangle继承了setWidth()和setLength()函数，这个对Square而言是副作用的，如果给Square的setWidth()和setLength()设置不一样的值，则Square的4边就不等长了。这时可以使用委托来代替继承：

```
public class Square {
    private Rectangle rectangle;
    public Square() {
        rectangle = new Rectangle();
    }
    public void setLength(double length) {
        rectangle.setLength(length);
        rectangle.setWidth(length);
    }
    public double area() {
        return rectangle.area();
    }
}
```

重构方法- 用委托来代替继承

- 滥用继承的经典范例就是java.util里面的Stack继承了Vector(JDK 1.0留下的遗憾，一失足成千古恨 了)。

从逻辑上，Stack不应该继承Vector。因为StackStack的定义是先进后出(FILO)的。而Vector有一些对Stack无意义还可能有害的方法。比如setElementAt()方法。如果调用了setElementAt()方法，Stack还能先进后出吗？

重构方法- 用子类代替型别码

- 用子类来代替型别码（Replace Type with Subclasses）： 当我们的类使用型别码来表示子类的时候（例如，雇员分工程师，销售人员和管理人员），就可以使用这种重构方法。针对每种类型分别设计子类可以消除那些根据型别码进行判别的复杂的条件判断和switch语句。

重构方法-用子类代替型别码

范例：

```
public class Employee {
    //0-engineer , 1-salesman , 2-manager
    private int employeeType;
}
```

通过把employeeType替换为各种子类得到如下结构：

```
abstract class Employee {
    //...
}
public class Engineer extends Employee {
    //...
}
public class Salesman extends Employee {
    //...
}
public class Manager extends Employee {
    //...
}
```

重构方法- 用多态来代替条件判断

DERBYSoft 德比软件（上海）有限公司

- 用多态来代替条件判断(Replace conditional with Polymorphism)：当我们发现代码中有switch语句时，可以考虑创建子类处理不同的情况，从而去掉switch语句。

重构方法-用多态来代替条件判断

范例：

```
public class Employee {
    //0-engineer , 1-salesman , 2-manager
    private int employeeType;

    public String departmentName() {
        switch (employeeType) {
            case 0:
                return "Engineering";
            case 1:
                return "Sales";
            case 2:
                return "Management";
            default:
                return "Unknown";
        }
    }
}
```

重构方法-用多态来代替条件判断

用多态替换switch语句：

```
abstract class Employee {
    public abstract String departmentName();
}
public class Engineer extends Employee {
    public String departmentName() {
        return "Engineering";
    }
}
public class Salesman extends Employee {
    public String departmentName() {
        return "Sales";
    }
}
public class Manager extends Employee {
    public String departmentName() {
        return "Management";
    }
}
```

重构方法- 塑造模板函数

- 塑造模板函数(Form Template Method)：当我们在多个类中都有某种具有相同结构但不同细节的相似方法时就可以使用这种重构方法。相同结构的方法放在父类中，提取出来的细节具体方法在子类中实现。

重构方法- 塑造模板函数

范例：输出雇员(Engineer)的XML表示方法：

```
public class Engineer extends Employee {
    public String toXML(){
        StringBuffer source = new StringBuffer();
        source.append("<employee name=\"");
        source.append(getName());
        source.append("\" department=\"Engineering\“>");
        //...
        return source.toString();
    }
    public String departmentName() {
        return "Engineering";
    }
}
```

重构方法- 塑造模板函数

范例：输出雇员(Salesman)的XML表示方法：

```
public class Salesman extends Employee {
    public String toXML(){
        StringBuffer source = new StringBuffer();
        source.append("<employee name=\"");
        source.append(getName());
        source.append("\" department=\"Sales\“>");
        //...
        return source.toString();
    }
    public String departmentName() {
        return "Sales";
    }
}
```

重构方法- 塑造模板函数

范例：输出雇员(Manager)的XML表示方法：

```
public class Manager extends Employee {
    public String toXML(){
        StringBuffer source = new StringBuffer();
        source.append("<employee name=\"");
        source.append(getName());
        source.append("\" department=\"Management\“>");
        //...
        return source.toString();
    }
    public String departmentName() {
        return "Management";
    }
}
```

重构方法- 塑造模板函数

它们之间的唯一差别就是department不同，我们把toXML方法提到父类中作为模板方法，具体细节差异方法depertmentName()在子类中实现了,在模板方法中调用那个多态方法：

```
public abstract class Employee {
    public abstract String departmentName();
    public String toXML(){
        StringBuffer source = new StringBuffer();
        source.append("<employee name=\"");
        source.append(getName());
        source.append("\" department=\"");
        source.append(departmentName());
        source.append("\">");
        //...
        return source.toString();
    }
}
```

重构方法- 提取类(Extract class)

- 提取类：当一个类变得太大或其行为逻辑组织分散时，我们需要将其分成多块内聚的行为并在需要的时候创建新类。

重构方法- 提取类(Extract class)

范例：

```
public class Person {

    private String name;

    private String officeAreaCode;

    private String officeNumber;

    public String getTelephoneNumber() {
        return "(" + officeAreaCode + ")" + officeNumber;
    }

    public String getOfficeNumber() {
        return officeNumber;
    }

    //……

}
```

重构方法- 提取类(Extract class)

可以将与电话号码相关的行为分离到一个独立的class中：

```
public class Person {
    private TelephoneNumber telephoneNumber = new TelephoneNumber();
    public String getTelephoneNumber() {
        return telephoneNumber.getTelephoneNumber();
    }
    public String getOfficeNumber() {
        return telephoneNumber.getNumber();
    }
    //......
}

class TelephoneNumber {
    private String areaCode;
    private String number;

    public String getTelephoneNumber() {
        return "(" + areaCode + ")" + number;
    }

    public String getNumber() {
        return number;
    }
    //……
}
```

重构方法- 提取接口(Extract interface)

- 提取接口：当两个classes的接口有部分相同，或对具体的实现进行抽象以方便Mock的时候，可以考虑提取接口。

重构方法- 提取接口(Extract interface)

我们需要从MovieList中提取出来一个接口：

```
public class MovieList {

    private Collection movies = new ArrayList();

    public int size() {
        return movies.size();
    }

    public void add(Movie movie){
        movies.add(movie);
    }

    public boolean contains(Movie movie){
        return movies.contains(movie);
    }

}
```

重构方法- 提取接口(Extract interface)

DERBYSoft 德比软件（上海）有限公司

```
public interface IMovieList {
    int size();
    void add(Movie movie);
    boolean contains(Movie movie);
}

public class MovieList implements IMovieList{
    //.....
}
```

最后就是把对MovieList的引用修改成对IMovieList的引用。我们就能为IMovieList接口创建包含mock对象在内的其他实现了。

重构方法-引入解释变量

- 引入解释变量(Introduce Explaining Variable)：当表达式复杂且难以理解时，我们就可以提取其中的某些部分，把中间结果保留在命名清楚的临时变量中。把表达式切分成容易理解的片段，提高整个表达式的清晰度。

重构方法-引入解释变量

范例：一个简单的计算

```
public double price() {
    //price is base price - quantity discount + shipping
    return quantity * itemPrice - Math.max(0, quantity - 500) * itemPrice * 0.05
            + Math.min(quantity * itemPrice * 0.1, 100);
}
```

重构之后就可以不用注释表达意义了：

```
public double price() {
    double basePrice = quantity * itemPrice;
    double quantityDiscount = Math.max(0, quantity - 500) * itemPrice * 0.05;
    double shipping = Math.min(basePrice * 0.1, 100);
    return basePrice - quantityDiscount + shipping;
}
```

重构方法-使用工厂方法来代替构造方法

- 使用工厂方法来代替构造方法(Replace Constructor with Factory Method)：因为构造方法具有相同的名字，所以当存在多个构造方法时，就容易让人搞不清楚调用哪个。对此我们使用静态工厂方法，这样就能给每个构造方法起一个有意义的名字。

重构方法-使用工厂方法来代替构造方法

范例：

```
public class Rating {
    private int value;
    private String source;
    private String review;
    public Rating(int value) {
        this(value, "Anonymous", "");
    }
    public Rating(int value, String source) {
        this(value, source, "");
    }
    public Rating(int value, String source, String review) {
        this.value = value;
        this.source = source;
        this.review = review;
    }
    //...
}
```

重构方法-使用工厂方法来代替构造方法

把构造方法转换成工厂方法：

```
public class Rating {
    private int value;
    private String source;
    private String review;
    public static Rating newAnonymousRating(int value) {
        return new Rating(value, "Anonymous", "");
    }
    public static Rating newRating(int value, String source) {
        return new Rating(value, source, "");
    }
    public static Rating newReview(int value, String source, String review) {
        return new Rating(value, source, review);
    }
    private Rating(int value, String source, String review) {
        this.value = value;
        this.source = source;
        this.review = review;
    }
    //...
}
```

重构方法-使用工厂方法来代替构造方法

原来构造方法的调用：

```
lists.addRating(new Rating(1));
lists.addRating(new Rating(3,"People's Daily"));
lists.addRating(new Rating(5,"Amusement news ","a really fun movie."));
```

修改为调用工厂方法：

```
lists.addRating(Rating.newAnonymousRating(1));
lists.addRating(Rating.newRating(3,"People's Daily"));
lists.addRating(Rating.newReview(5,"Amusement news ","a really fun movie."));
```

重构方法- 使用符号常量来代替魔幻数字

- 使用符号常量来代替魔化数字(Replace Magic Number with Symbolic Constant): 在代码中硬性编码的数值是一种非常不好的习惯。这样的数值很难分辨，改变它们会引起霰弹式手术，其实这也是一种重复。可以用命名良好的符号常量来代替它们。

重构方法- 使用符号常量来代替魔幻数字

一个简单的例子计算圆面积：

```
double calculateAcreage(double radial) {
    return radial * radial * 3.14;
}
```

用符号常量来代替魔幻数字：

```
private final static double CIRCUMFERENCE_RATIO = 3.14;
double calculateAcreage(double radial) {
    return radial * radial * CIRCUMFERENCE_RATIO;
}
```

这样就可以保证数值只出现一次，如果需要改变这个值，只需要修改一处就可以了。

重构方法-使用卫述句代替嵌套的条件判断

- 使用卫述句代替嵌套的条件判断(Replace Nested Conditional with Guard Clauses): 单一出口规则要求每个方法只能有一个出口(即return语句)。这样的要求没有正当的理由，肯定不是出于对代码清晰度的考虑。对应当在多种条件下退出的方法来说，就会导致复杂、嵌套的条件判断语句。这种情况下一种更好更清楚的解决办法就是采用卫述句来返回。

重构方法-使用卫述句代替嵌套的条件判断

一个简单的例子：

```
public int fib(int i) {
    int result;
    if (i == 0) {
        result = 0;
    } else if (i <= 2) {
        result = 1;
    } else {
        result = fib(i - 1) + fib(i - 2);
    }
    return result;
}
```

用卫述句重构之后：

```
public int fib(int i) {
    if (i == 0) return 0;
    if (i <= 2) return 1;
    return fib(i - 1) + fib(i - 2);
}
```

重构方法-使用卫述句代替嵌套的条件判断

一个更恶心点的嵌套范例：

```
public double getPayAmout() {
    double result;
    if (isDead) result = deadAmount();
    else {
        if (isSeparated) result = separatedAmount();
        else {
            if (isRetired) result = retiredAmount();
            else result = normalPayAmout();
        }
    }
    return result;
}
```

用卫述句重构之后：

```
public double getPayAmout() {
    if (isDead) return deadAmount();
    if (isSeparated) return separatedAmount();
    if (isRetired) return retiredAmount();
    return normalPayAmout();
}
```

重构方法-引入空对象

- 引入空对象(Introduce Null Object):将null value（无效值）替换为null object（无效物）。创建一个代表特殊情形(null)的对象，让它具备和普通对象相同的协议，但不能有任何副作用。

重构方法-引入空对象

来看看java.io.File类里面的方法：

```
public boolean setReadOnly() {
    SecurityManager security = System.getSecurityManager();
    if (security != null) {
        security.checkWrite(path);
    }
    return fs.setReadOnly(this);
}
```

在java.io.File(JDK1.6)中security != null出现了20次，这样的检查十分繁琐无味。而且还存在忘记检查的隐患。

重构方法-引入空对象

有一种可选的解决办法是创建一个新类LaxSecurity(引入空对象):

```
public class LaxSecurity {
    public void checkWrite() {
    }
}
```

SecurityManager的getSecurityManager方法改成这样：

```
public class SecurityManager {
    public static SecurityManager getSecurityManager() {
        return security == null ? new LaxSecurity() : security;
    }
    //...
}
```

重构方法-引入空对象

现在java.io.File类中20处繁琐的空值检查就可以丢掉了，代码也简洁了：

```
public boolean setReadOnly() {
    SecurityManager security = System.getSecurityManager();
    security.checkWrite(path);
    return fs.setReadOnly(this);
}
```

重构方法-以异常取代错误码

为什么要以异常取代错误码(Replace Error Code with Exception)?

- 错误码不能跨作用域传送，必须逐层向上转发，而且错误码是可以忽略的，因为调用函数时可以不处理其返回值，从而使错误处理要依赖文档约束，而不是程序本身的要求，这个错误可能会导致系统崩溃，非常不安全。
- 异常可以跨作用域传送，可以很好的把错误发现和错误处理分开，异常是不可忽略的，抛出的异常必须捕获，否则就会报错，保证了程序的安全性。
- 当然某些情况下还不能用异常代替错误码，比如对外发布的服务接口用异常处理错误就不是很合适：如果我们的服务是java语言发布的，对方是用.net调用的，那么如果我们抛出异常，调用方可能就不能识别。

通过重构来引入模式

- 设计模式是经过提炼和验证的设计思想，要想成为一名好的程序员，就应该尽可能多地了解模式，并不断地学习新的模式。
- 更重要的是要知道何时使用模式以及何时不使用模式。设计模式的危险之处就是深陷其中。
- 应该如何使用模式呢？答案就是把模式作为重构的目标（注意是目标，不是目的），在必要的时候通过重构渐进地引入设计模式。

DerbySoft

Thank You

朱 攀
zhupan@derbysoft.com