软件危机＆抽象层次

麻锦涛 16281262 计科1602

**一、软件危机**

软件危机是指计算机软件开发和维护过程中所遇到的一系列严重问题。

典型表现有：

（1）对软件开发成本和进度的估计常常很不准确

（2）软件产品的质量往往靠不住

（3）用户对已完成的软件系统不满意的现象经常发生

（4）软件常常是不可维护的

（5）软件中没有适当的文档资料

（6）软件成本在计算机系统总成本所占的比例逐年上升

（7）软件开发生产率提高的速度，往往跟不上计算机应用迅速普及深入的趋势

原因：

（1）软件本身独有的特点确实给开发和维护带来了困难

（2）与软件开发和维护的许多错误认识和做法的形成有关

（3）程序只是完整软件的一个组成部分

（4）轻视是一个最大的错误

**二、抽象层次**

开发中的抽象

在软件开发里面，最重要的抽象就可能是分层了。分层随处可见，例如我们的系统就是分层的。最早的程序是直接运行在硬件上的，开发成本非常高。然后慢慢开始有了操作系统，操作系统提供了资源管理、进程调度、输入输出等所有程序都需要的基础功能，开发程序时调用操作系统的接口就可以了。再后来发现操作系统也不够，于是又有了各种运行环境（如 JVM）。

编程语言也是一种分层的抽象。机器理解的其实是机器语言，即各种二进制的指令。但我们不可能直接用机器语言编程，于是我们发明了汇编语言、C 语言以及 Java 等各种高级语言，一直到 Ruby、Python 等动态语言。

开发中，我们应该也都听说过各种分层模型。例如经典的三层模型（展现层、业务逻辑层、数据层），还有 MVC 模型等。有一句名言：“软件领域的任何问题，都可以通过增加一个间接的中间层来解决”。分层架构的核心其实就是抽象的分层，每一层的抽象只需要而且只能关注本层相关的信息，从而简化整个系统的设计。

其实软件开发本身，就是一个不断抽象的过程。我们把业务需求抽象成数据模型、模块、服务和系统，面向对象开发时我们抽象出类和对象，面向过程开发时我们抽象出方法和函数。也即是说，上面提到的模型、模块、服务、系统、类、对象、方法、函数等，都是一种抽象。可想而知，设计一个好的抽象，对我们软件开发有多么重要。

抽象的原则

那么到底应如何做到好的抽象呢？在软件开发领域，前人们其实早帮我们整理出了 SOLID 等设计原则以及各种设计模式。对于 SOLID 原则，虽然很多人都听说过，但其实真正能理解这些原则的开发者并不多。那么我们就从抽象的角度，再来看下这些原则，也许会有更好的理解。

单一职责原则（Single Responsibility Principle, SRP）

单一职责是指一个模块应该只做一件事，并把这件事做好。其实对照应抽象的定义，可以发现这个原则本身就是抽象的核心体现。如果一个类包含了很多方法，或者一个方法特别长，就要引起我们的特别注意了。例如下面这个 Employee 类，既有业务逻辑（calculatePay）、又有数据库逻辑（saveToDb），那它其实至少做了两件事情，也就不符合单一职责原则，当然也就不是一个好的抽象。

class Employee {  
    public Pay calculatePay() {...}      
    public void saveToDb() {...}  
}

有些人觉得单一职责不太好理解，有时候很那分辨一个模块到底是不是单一职责。其实单一职责的概念，常常需要结合抽象的分层去理解。

在同一个抽象层里，如果一个类或者一个方法做了不止一件事，一般是比较容易分辨的。例如一个违反单一职责原则的典型征兆是，一个方法接受一个布尔类型或者枚举类型的参数，然后一个大大的 if／else 或者 switch／case，分支里也是大段的代码处理各种情况下的逻辑。这时我们可以用简单工厂模式、策略模式等设计模式去优化设计。

假如说我们用了简单工厂模式，改进了一段代码，重构后代码可能像是下面是这样的。

public Instance getInstance(final int type){  
    switch (type) {

case 1: return new AAInstance;

case 2: return new BBInstance;

default: return new DefaultInstance();

}  
}

有人可能会有疑问，代码里依然还是存在 if／else 或者 switch／case，这不还是做了不止一件事情么？其实不是的，使用了简单工厂模式，其实就是增加了一个抽象层。在这个抽象层里，getInstance 的职责很明确，就是创建对象。而原来分支里的逻辑处理，则下沉到了另外一个抽象层里去了，也就是 Instance 的实现所在的抽象层。

再看下面 Scala 实现的 updateOrder 方法，它似乎也只是做了一件事情：处理订单，那算不算单一职责呢？

protected def updateOrder(t: TransationEntity) = {  
// 1 获取订单

ManagedRepo.find[Order]("orderNo" -> t.tradeNo).map { order =>

// 2 检查订单是否已支付

val ps = SQL("""select statue from Order where id ={id} for update""").on("id" -> order.id).as(scalar[Long].singleOpt).getOrElse(0l)

if (ps == PAID) {

throw ServiceException(ApiError.SUBSCRIPTION\_UPDATE\_FAIL)

} else {

// 3 更新订单信息，标记为已支付

val updatedOrder = // 略...

updatedOrder.saveOrUpdate()

// 4 生成收入记录

createIncome(updatedOrder)

}

}  
}

答案当然是不算，因为很明显，这个方法里面既有业务逻辑的代码，又有数据库处理的代码，这两类应该是在不同的抽象层的。我们把数据库处理的代码抽取出来，下沉到数据层，它就能符合单一职责原则了。

protected def updateOrder(t: TransationEntity) = {  
findUnpaidOrder(rtent.tradeNo).map { order =>

val updatedOrder = updateOrderForPayment(rtent)

createIncome(updatedOrder)

}  
}

开放封闭原则（Open/Closed Principle, OCP）

开放封闭原则是指对扩展开放，对修改封闭。当需求改变时，我们可以扩展模块以满足新的需求；但扩展时，不应该需要修改原模块的实现。

下面两段代码都实现了方形、矩形以及圆形的面积计算。第一种用的是面向过程的方法，第二种用的是面向对象的方法。那么，到底哪一种更符合开放封闭原则呢？

面向过程方法：

public class Square {  
    public double side;  
}  
public class Rectangle {

public double height;

public double width;  
}  
public class Circle {

public double radius;  
}  
public class Geometry {

public double area(Object shape) {

if (shape instanceof Square) {

Square s = (Square) shape;

return s.side \* s.side;

} else if (shape instanceof Rectangle) {

Rectangle r = (Rectangle) shape;

return r.height \* r.width;

} else if (shape instanceof Square) {

Circle c = (Circle) shape;

return PI \* c.radius \* c.radius;

} else {

throw new NoSuchShareException();

}

}  
}

面向对象方法：

public class Square implements Share {  
    public double side;

public double area() {

return side \* side;

}  
}  
public class Rectangle implements Share {

public double height;

public double width;

public double area() {

return height \* width;

}  
}  
public class Circle implements Share {

public double radius;

public double area() {

return PI \* radius \* radius;

}  
}

估计很多人会觉得面向对象的方式更好，更符合开放封闭原则。但真相其实没那么简单。想象如果我们需要添加一个新的形状，比如说椭圆，那面向对象的实现肯定更方便，我们只需要实现一个椭圆的类以及它的 area 方法。这时候我们可以说面向对象的方法更符合开放封闭原则。

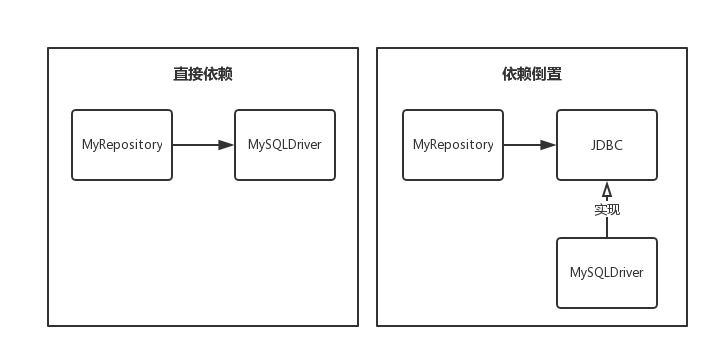
但如果我们需要添加一个新的方法呢？比如说，我们发现我们还需要计算形状的周长。这时候，面向对象的实现似乎就没那么方便了，要在每个类里面添加计算周长的方法。而面向过程的方法，则只需要添加一个方法就行了。这时候，我们反而发现面向过程的方法更符合开放封闭原则。

所以开放封闭其实是相对的，有时候，如何进行抽象，取决于我们对未来最有可能的扩展的预判。

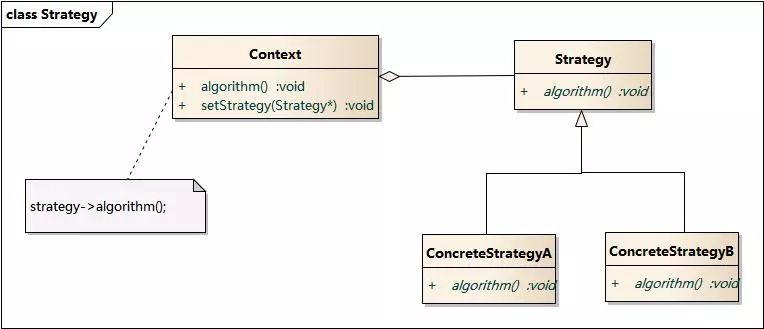
依赖倒置原则（Dependency Inversion Principle, DIP）

依赖倒置原则是指高层模块不应该依赖于低层模块的实现，两者都应该依赖于抽象。抽象不应该依赖于细节，细节应该依赖与抽象。前面提到，“软件领域的任何问题，都可以通过增加一个间接的中间层来解决” ，DIP 就是最典型的增加中间层的方式，也是我们需要解耦两个模块的最重要的方法之一。

依赖倒置原则的一个例子是 Java 的 JDBC。如果没有 JDBC，那我们的系统就会严格依赖我们使用的那个数据库。这时如果我们想要切换到另外一个数据库，就需要修改大量代码。但 Java 提供了 JDBC 接口，而所有关系数据库的连接库都实现了这个接口，我们的系统也只需要调用 JDBC 即可完成数据库操作。这时我们的系统和数据库的依赖就解除了。除了 JDBC，其实 SQL 本身也是一种依赖倒置的实现。另外一个很典型的例子就是 Java 的日志接口 Slf4j。



其实所有的协议和标准化都是 DIP 的一种实现。包括 TCP、HTTP 等网络协议、操作系统、JVM、Spring 框架的 IOC 等等。设计模式里有不少模式，也是典型的依赖倒置，例如状态模式、工厂模式、代理模式、策略模式等等，下图是策略模式的结构图。



我们日常生活中也有很多依赖倒置的例子。比如电源插座，家庭的供电只需要提供符合国家标准的电源插座，我们购买电器产品时，就不用担心买回来无法接入电源。汽车和轮胎、铅笔和笔、USB／耳机接口等等，也都是同一思想的体现。

里氏替换原则（Liskov Substitution Principle, LSP）

里氏替换原则是指子类必须能够替换成它们的基类。例如下面这个最常见的例子，Square 可以是 Rectangle 的子类吗？

public class Rectangle {  
    public double height;

public double width;

public void setHeight(int height) { ... }

public void setWidth(int width) { ... }  
}  
public class Square extends Rectangle {

???  
}

虽然几何上说，Square 是一个特殊的 Rectangle，但把 Square 作为 Rectangle 的子类，却未必合适，因为它已经不存在宽和高的概念了。如果一个抽象不能符合里氏替换原则，那我们就需要考虑下这个抽象是不是合适了。

接口隔离原则（Interface Segregation Principle, ISP）

接口隔离原则是指客户端不应该被迫依赖它们不使用的方法。例如下面的 Square 类如果继承了Shape 接口，该如何计算体积以实现volume方法？

interface Shape {  
    public function area();

public function volume();  
}  
public class Square extends Shape {

???  
}

同样，如果一个抽象不符合接口隔离原则，那可能就不是一个合适的抽象。

迪米特法则（Law of Demeter）

迪米特法则不属于 SOLID 原则，但我觉得也值得说一下。它是指模块不应该了解它所操作的对象的内部情况。想象一下，如果你想让你的狗狗快点跑的话，你会对狗狗说，还是对四条狗腿说？如果你去店里买东西，你会把钱交给店员，还是会把钱包交给店员让他自己拿？

下面是一段违反迪米特法则的典型代码。这样的代码把对象内部实现暴露了出来，应该考虑讲将功能直接暴露为接口，或者合理使用设计模式（如 Facade）。

final String outputDir = ctxt.getOptions().getScratchDir().getAbsolutePath();

总结

、软件开发并不是仅仅只依靠抽象能力就能完成的，最终我们还是要把我们抽象出来的架构、模型等，落地到真正的代码层面，那就还需要逻辑思维能力、系统分析能力等。