Spring的本质系列(2)-AOP

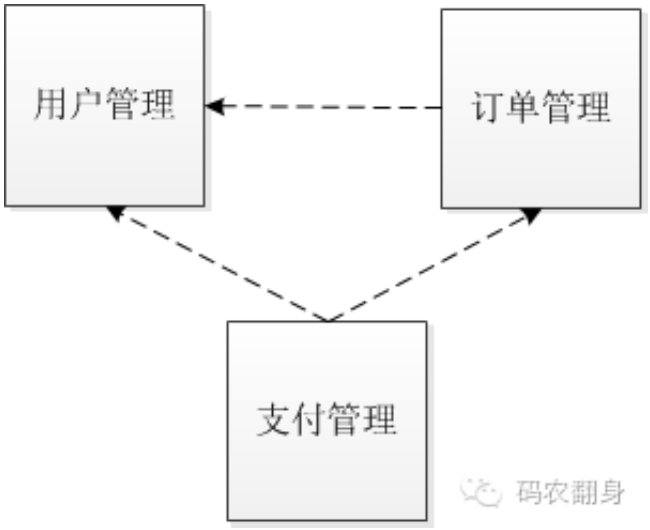
原创：刘欣

> 据说有些词汇非常热门和神奇， 如果你经常把它挂在嘴边，就能让自己功力大涨， 可以轻松找到理想的高薪的工作 :-)
>
> 这些词就包括上一篇文章(《Spring本质系列(1) --依赖注入》)中聊过的**IoC** 和 **DI**， 也包括今天要聊的**AOP**。
>
> AOP（Aspect Oriented Programming）就是面向切面的编程， 为什么是面向切面， 而不是面向对象呢?

> (码农翻身提示：如果你对设计模式不太熟， 可以跳过下文中的第2和第3节)

1 问题来源

我们在做系统设计的时候，一个非常重要的工作就是把一个大系统做分解， 按业务功能分解成一个个低耦合、高内聚的模块，就像这样：



但是分解以后就会发现有些很有趣的东西， 这些东西是通用的，或者是跨越多个模块的：

日志： 对特定的操作输出日志来记录

安全：在执行操作之前进行操作检查

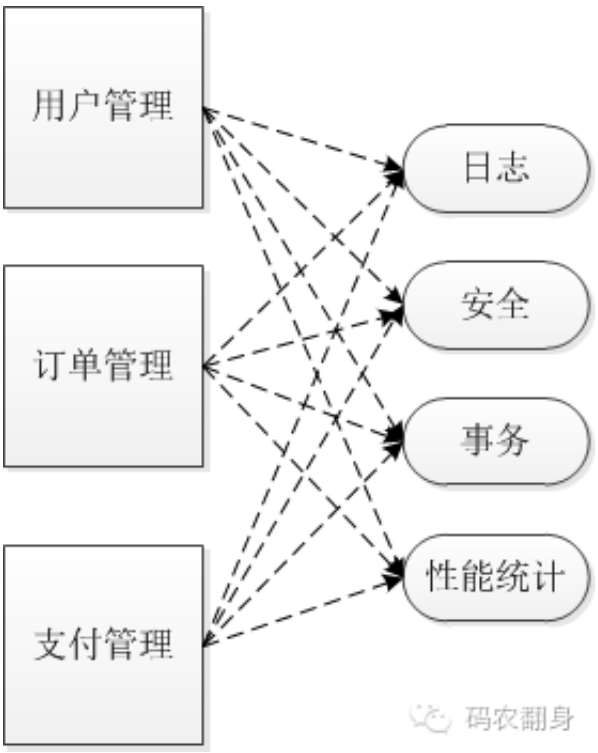
性能：要统计每个方法的执行时间

事务：方法开始之前要开始事务， 结束后要提交或者回滚事务

等等....

这些可以称为是非功能需求， 但他们是多个业务模块都需要的， 是跨越模块的， 把他们放到什么地方呢?

最简单的办法就是把这些通用模块的接口写好， 让程序员在实现业务模块的时候去调用就可以了，码农嘛，辛苦一下也没什么。



这样做看起来没问题， 只是会产生类似这样的代码：

```
public class PlaceOrderCommand {
  public void execute() {
    Logger logger = Logger.getLog(...);
    logger.debug("...");
    PerformanceUtil.startTimer(...);
    if(! user.hasPreviledge(...)) {
      //抛出异常
    }
    //开始事务
    beginTransaction();
    //执行业务
    commitTransaction();
    PerformanceUtil.endTimer();
    logger.debug("...");
  }
}
```

这样的代码也实现了功能，但是看起来非常的不爽， 那就是日志，性能，事务 相关的代码几乎要把真正的业务代码给淹没了。

不仅仅这一个类需要这么干， 其他类都得这么干， 重复代码会非常的多。

有经验的程序员还好， 新手忘记写这样的非业务代码简直是必然的。

2 设计模式：模板方法

用设计模式在某些情况下可以部分解决上面的问题，例如著名的**模板方法**：

```
public abstract class BaseCommand {
  public void execute() {
    Logger logger = Logger.getLog(...);
    logger.debug("...");
    PerformanceUtil.startTimer(...);
    if(! user.hasPreviledge(...)) {
      //抛出异常
    }
    //开始事务
    beginTransaction();
    //这是一个需要子类实现的抽象方法
    doBusiness();
    commitTransaction();
    PerformanceUtil.endTimer();
    logger.debug("...");
  }
  public abstract void doBusiness();
}
class PlaceOrderCommand extends BaseCommand {
  public void doBusiness() {
    //执行下单操作
  }
}
class PaymentCommand extends BaseCommand {
  public void doBusiness() {
    //执行支付操作
  }
}
```

在父类（BaseCommand）中已经把那些"乱七八糟"的非功能代码都写好了， 只是留了一个口子（抽象方法doBusiness()）让子类去实现。

子类变的清爽， 只需要关注业务逻辑就可以了。

调用也很简单，例如：

BaseCommand cmd = ... 获得PlaceOrderCommand的实例...

cmd.execute();

但是这样方式的巨大缺陷就是父类会定义一切： **要执行哪些非功能代码， 以什么顺序执行等等**

子类只能无条件接受，完全没有反抗余地。

如果有个子类， 根本不需要事务， 但是它也没有办法把事务代码去掉。

3 设计模式：装饰者

如果利用装饰者模式， 针对上面的问题，可以带来更大的灵活性：

```
public interface Command {
  public void execute();
}
//一个用于记录日志的装饰器
public class LoggerDecorator implements Command {
  Command cmd;
  public LoggerDecorator(Command command) {
    this.cmd = command;
  }
  public void execute() {
    Logger logger = Logger.getLog(...);
    logger.debug("...");
    this.cmd.execute();
    logger.debug("...");
  }
}
//一个用于性能统计的装饰器
public class PerformanceDecorator implements Command {
  Command cmd;
  public PerformanceDecorator(Command command) {
    this.cmd = command;
  }
  public void execute() {
    PerformanceUtil.startTimer(...);
    this.cmd.execute();
    PerformanceUtil.endTimer(...);
  }
}
class PlaceOrderCommand extends BaseCommand {
  public void doBusiness() {
    //执行下单操作
  }
}
class PaymentCommand extends BaseCommand {
  public void doBusiness() {
    //执行支付操作
  }
}
```

现在让这个PlaceOrderCommand 能够打印日志，进行性能统计

Command cmd = new **LoggerDecorator**(

new **PerformanceDecorator**(

new **PlaceOrderCommand**()));

cmd.execute();

如果PaymentCommand 只需要打印日志，装饰一次就可以了：

Command cmd = new **LoggerDecorator**(

new **PaymentCommand**());

cmd.execute();

可以使用任意数量装饰器，还可以以任意次序执行（严格意义上来说是不行的）， 是不是很灵活?

4 AOP

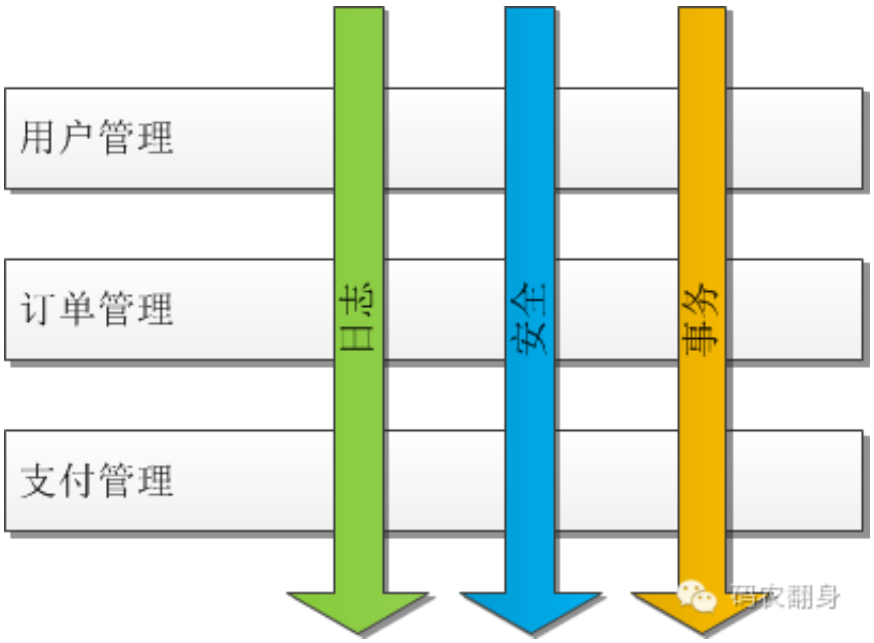
如果仔细思考一下就会发现装饰者模式的不爽之处:

(1) 一个处理日志/性能/事务 的类为什么要实现 业务接口（Command）呢?

(2) 如果别的业务模块，没有实现Command接口，但是也想利用日志/性能/事务等功能，该怎么办呢?

最好把日志/安全/事务这样的代码和业务代码完全隔离开来，因为他们的关注点和业务代码的关注点完全不同 ，他们之间应该是正交的，他们之间的关系

应该是这样的:



如果把这个业务功能看成一层层面包的话， 这些日志/安全/事务 像不像一个个“切面”(Aspect)？

如果我们能让这些“切面“能和业务独立， 并且能够非常灵活的“织入”到业务方法中， 那就实现了面向切面编程(AOP)!

5 实现AOP

现在我们来实现AOP吧， 首先我们得有一个所谓的“切面“类(Aspect)， 这应该是一个普通的java 类 ，不用实现什么“乱七八糟”的接口。

以一个事务类为例:

```
public class Transaction {
  public void beginTx() {
    //开始事务
  }
  public void commitTx() {
    //提交事务
  }
}
```

我们想达到的目的只这样的： 对于com.coderising这个包中所有类的execute方法， 在方法调用之前，需要执行Transaction.beginTx()方法， 在调用之后， 需要执行Transaction.commitTx()方法。

暂时停下脚步分析一下。

“对于com.coderising这个包中所有类的execute方法”， 用一个时髦的词来描述就是**切入点（PointCut）**, 它可以是一个方法或一组方法（可以通过通配符来支持，你懂的）

”在方法调用之前/之后， 需要执行xxx“, 用另外一个时髦的词来描述就是**通知（Advice）**

码农翻身认为，PointCut,Advice 这些词实在是不直观， 其实Spring的作者们也是这么想的 : These terms are not Spring-specific... unfortunately, AOP terminology is not particularly intuitive; however, it would be even more confusing if Spring used its own terminology.

当然，想描述这些规则， xml依然是不二之选：

```
<aspect id="tx" class="com.coderising.Transaction">
  <pointcut id="place-order" expression="com.coderising.*.execute()">
  <before pointcut-ref="place-order" method="beginTx"/>
  <after pointcut-ref="place-order" method="commitTx"/>
</aspect>
```

注意：现在**Transaction这个类和业务类在源代码层次上没有一点关系，完全隔离了。**

隔离是一件好事情， 但是马上给我们带来了大麻烦 。

Java 是一门静态的强类型语言， 代码一旦写好， 编译成java class 以后 ， 可以在运行时通过反射（Reflection）来查看类的信息， 但是想对类进行修改非常困难。

而AOP要求的恰恰就是在不改变业务类的源代码（其实大部分情况下你也拿不到）的情况下， 修改业务类的方法, 进行功能的增强，就像上面给所有的业务类增加事务支持。

为了突破这个限制，大家可以说是费尽心机， 现在基本是有这么几种技术：

(1) 在编译的时候， 根据AOP的配置信息，悄悄的把日志，安全，事务等“切面”代码 和业务类编译到一起去。

(2) 在运行期，业务类加载以后， 通过Java动态代理技术为业务类生产一个代理类， 把“切面”代码放到代理类中， Java 动态代理要求业务类需要实现接口才行。

(3) 在运行期， 业务类加载以后， 动态的使用字节码构建一个业务类的子类，将“切面”逻辑加入到子类当中去, CGLIB就是这么做的。

Spring采用的就是(1) +(2) 的方式，限于篇幅，这里不再展开各种技术了， 不管使用哪一种方式， 在运行时，真正干活的“业务类”其实已经不是原来单纯的业务类了， 它们被AOP了！