链滴

回看一下领域模型中的贫血模型

作者：ykz200

原文链接：https://ld246.com/article/1620203647773

来源网站：链滴

本文来源回看一下领域模型中的贫血模型

本篇笑点

我要从甲方跳到乙方公司了。

你得知道这篇讲的些什么

本文1517字，阅读可能需要2分59秒

Java Web开发中最常见的领域模型 - **贫血模式**

说明

前几个月和杭州的大妖怪说要做一个xxxx项目，其中聊到了**领域模型**。

准备写一篇领域模型回顾篇，基本是针对网络上的总结进行一些细节丰富，去除一些不必要的理解。

贫血模式

项目要采用的是 **贫血模型**，我们先来看一下贫血模型。

而所谓 贫血模型 就是模型对象之间存在完整的关联(可能存在多余的关联)，但是对象除了get和set方外几乎就没有其它的方法，整个对象充当的就是一个数据容器。

所有的业务方法都在一个无状态的Service类中实现，Service类仅仅包含一些行为。

这是大多数JavaWeb程序采用的最常用开发模型，看完下面的包结构你会更清晰这个模型。

包结构

贫血模型的实现一般包括如下包：

包	内容
dao	负责持久化逻辑
model	包含数据对象，是service操纵的对象
service	放置所有的服务类，其中包含了所有的业逻辑
facade	提供对UI层访问的入口

代码实现

先看model包的两个类，Account和TransferTransaction对象，分别代表帐户和一次转账事务。由它们不包含业务逻辑，就是一个普通的Java Bean，下面的代码省略了get和set方法。

```
/**
 * 账户
 **/
public class Account {
    private String accountId;
    private BigDecimal balance;

    public Account() {}
    public Account(String accountId, BigDecimal balance) {
        this.accountId = accountId;
        this.balance = balance;
    }
    // getter and setter ....

}

/**
 * 转账
 **/
public class TransferTransaction {
    private Date timestamp;
    private String fromAccountId;
    private String toAccountId;
    private BigDecimal amount;

    public TransferTransaction() {}

    public TransferTransaction(String fromAccountId, String toAccountId, BigDecimal amount,
ate timestamp) {
        this.fromAccountId = fromAccountId;
        this.toAccountId = toAccountId;
        this.amount = amount;
        this.timestamp = timestamp;
    }

    // getter and setter ....
}
```

这两个类很常见，就是数据容器。

接下来看service包中TransferService接口和它的实现TransferServiceImpl。TransferService定义了账服务的接口，TransferServiceImpl则提供了转账服务的实现。

```
public interface TransferService {
    TransferTransaction transfer(String fromAccountId, String toAccountId, BigDecimal amount)

            throws AccountNotExistedException, AccountUnderflowException;
}

public class TransferServiceImpl implements TransferService {
    private AccountDAO accountDAO;
    private TransferTransactionDAO transferTransactionDAO;

    public TransferServiceImpl(AccountDAO accountDAO,
            TransferTransactionDAO transferTransactionDAO) {
        this.accountDAO = accountDAO;
        this.transferTransactionDAO = transferTransactionDAO;

    }

   /**
     * 转账逻辑
     * @param fromAccountId 转账人
     * @param toAccountId  收账人
     * @param amount 金额
     * @return
     * @throws AccountNotExistedException
     * @throws AccountUnderflowException
     */
    public TransferTransaction transfer(String fromAccountId, String toAccountId,
            BigDecimal amount) throws AccountNotExistedException, AccountUnderflowException
{
    Validate.isTrue(0 < amount.compareTo(BigDecimal.ZERO));

        Account fromAccount = accountDAO.findAccount(fromAccountId);
        if (null == fromAccount) throw new AccountNotExistedException(fromAccountId);
        if (0 > fromAccount.getBalance().compareTo(amount)) {
            throw new AccountUnderflowException(fromAccount, amount);
        }

        Account toAccount = accountDAO.findAccount(toAccountId);
        if (null == toAccount) throw new AccountNotExistedException(toAccountId);
        fromAccount.setBalance(fromAccount.getBalance().subtract(amount));
        toAccount.setBalance(toAccount.getBalance().add(amount));

        accountDAO.updateAccount(fromAccount);      // 对Hibernate来说这不是必须的
        accountDAO.updateAccount(toAccount);        // 对Hibernate来说这不是必须的
        return transferTransactionDAO.create(fromAccountId, toAccountId, amount);
    }
}
```

TransferServiceImpl类使用了AccountDAO和TranferTransactionDAO，其中transfer方法负责整转帐操作业务。TransferServiceImpl负责所有的业务逻辑，验证是否超额提取并更新帐户余额。

一切并不复杂，对于这个例子来说，贫血模型工作得非常好！这是因为这个例子相当简单，业务逻辑不复杂，一旦业务逻辑变得复杂，TransferServiceImpl就会膨胀。

我有幸见过一个电商项目下单方法写了3k行的代码，最后被上司安排了对这部分代码开了分支，花了个周的时间，针对性的做了一些各模块化方法优化。

总结

简单来说，就是domain ojbect包含了不依赖于持久化的领域逻辑，而那些依赖持久化的领域逻辑被离到Service层。

优点：

1、各层单向依赖，结构清楚，易于实现和维护

2、设计简单易行，底层模型非常稳定

缺点：

1、domain object的部分比较紧密依赖的持久化domain logic被分离到Service层，显得不够OO

2、Service层过于厚重

附言

本篇如有错误，请及时指出，马上修改。

非常非常重要的事情