第1章 架构中的设计模式

目标1：设计模式常用知识

1、设计模式优点
2、设计模式遵循原则
3、设计模式分类

目标2：设计模式单利讲解

1、单利模式使用
2、单利模式多种实现方式
3、单利模式内存消耗对比

目标3：Spring源码设计模式的使用

1、代理模式分析
2、观察者模式分析
3、模板模式分析
4、适配器模式分析

目标4：架构中的设计模式使用

1、自定义框架，MVC+AOP
2、使用单利模式、观察者模式、代理模式、适配器模式、工厂模式

1 认识设计模式

问题：

1、什么是设计模式
2、设计模式什么时候使用
3、使用设计模式有什么好处
4、设计模式设计需要遵循哪些原则
5、设计模式有哪些分类

1.1 设计模式简介

软件设计模式（Software Design Pattern），俗称设计模式，**设计模式**是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。它描述了在软件设计过程中的一些不断重复发生的问题，以及该问题的解决方案。也就是说，它是解决特定问题的一系列套路，是前辈们的代码设计经验的总结，具有一定的普遍性，可以反复使用。使用设计模式的**目的**是为了代码重用、让代码更容易被他人理解、保证代码可靠性。

设计模式：

> 设计模式是一套被反复使用的、多数人知晓的、经过分类编目的、代码设计经验的总结。它描述了在软件设计过程中的一些不断重复发生的问题，以及该问题的解决方案。

设计模式使用场景：

> 1、在程序软件架构设计上会使用到设计模式
> 2、在软件架构设计上会使用到设计模式

设计模式的目的：

> 1、提高代码的可重用性
> 2、提高代码的可读性
> 3、保障代码的可靠性

GOF

《Design Patterns: Elements of Reusable Object-Oriented Software》（即后述《设计模式》一书），由 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 合著（Addison-Wesley，1995）。这几位作者常被称为"四人组（Gang of Four）"，而这本书也就被称为"四人组（或 GoF）"书。

在《设计模式》这本书的最大部分是一个目录，该目录列举并描述了 23 种设计模式。

1.2 设计原则

优良的系统设计具备特点：

```
1.可扩展性(Extensibility)
2.灵活性(Flexibility)
3.组件化可插拔式(Pluggability)
```

面向对象编程常用的设计原则包括7个，这些原则并不是孤立存在的，它们相互依赖，相互补充。

设计原则名称	设计原则简介	重要性
单一职责原则(Single Responsibility Principle, SRP)	类的职责要单一，不能将太多的职责放在一个类中。**该原则是实现高内聚、低耦合的指导方针**	★★★★☆
开闭原则(Open-Closed Principle, OCP)	一个软件实体应当对扩展开放，对修改关闭。**即在不修改源代码的情况下改变对象的行为。**	★★★★★
里氏代换原则(Liskov Substitution Principle, LSP)	在软件系统中，一个可以接受基类(父类)对象的地方必然可以接受一个子类对象。里氏原则属于开闭原则的实现。	★★★★☆
依赖倒转原则(Dependency Inversion Principle, DIP)	抽象不应该依赖于细节(实现)，细节应当依赖于抽象。换言之，要针对接口编程，而不是针对实现编程。 **用到接口的地方，通过依赖注入将接口的实现对象注入进去。**	★★★★★
接口隔离原则(Interface Segregation Principle, ISP)	使用多个专门的接口来取代一个统一的接口。	★★☆☆☆
合成复用原则(Composite Reuse Principle, CRP)	在系统中应该尽量多使用组合和聚合关联，尽量少使用甚至不使用继承关系。	★★★★☆
迪米特法则(Law of Demeter, LoD)	一个软件实体应当尽可能少地与其他实体发生相互作用。通过引 入一个合理的第三者来降低现有对象之间的耦合度	★★★☆☆

1.2.1 单一职责原则

定义：一个对象应该只包含单一的职责，并且该职责被完整地封装在一个类中。

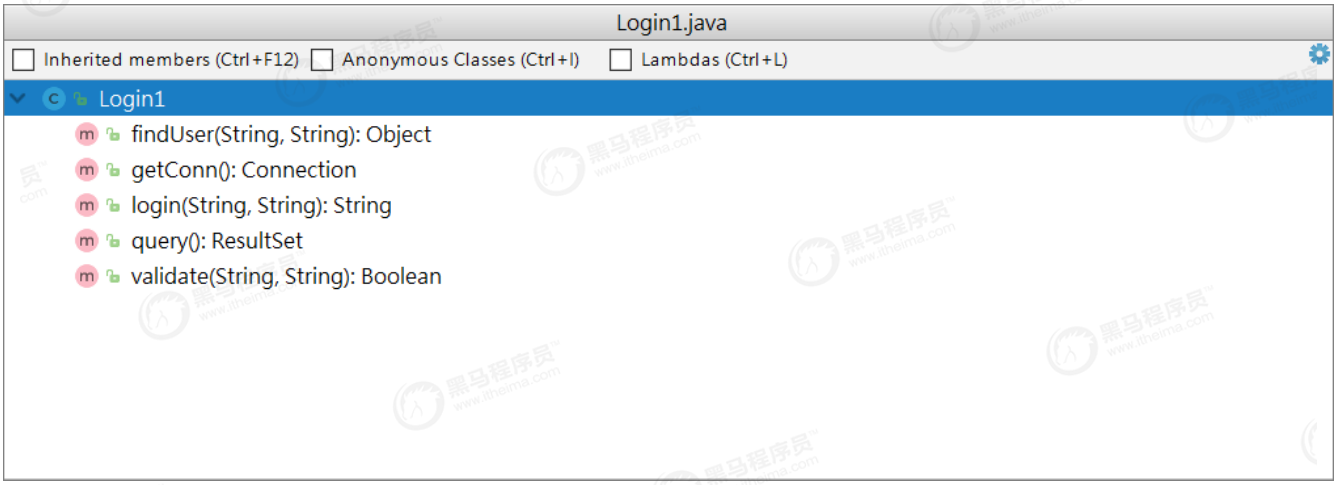
解说：一个类（或者大到模块，小到方法）承担的职责越多，它被复用的可能性越小，而且如果一个类承担的职责过多，就相当于将这些职责耦合在一起，当其中一个职责变化时，可能会影响其他职责的运作。

类的职责主要包括两个方面：数据职责和行为职责，数据职责通过其属性来体现，而行为职责通过其方法来体现。

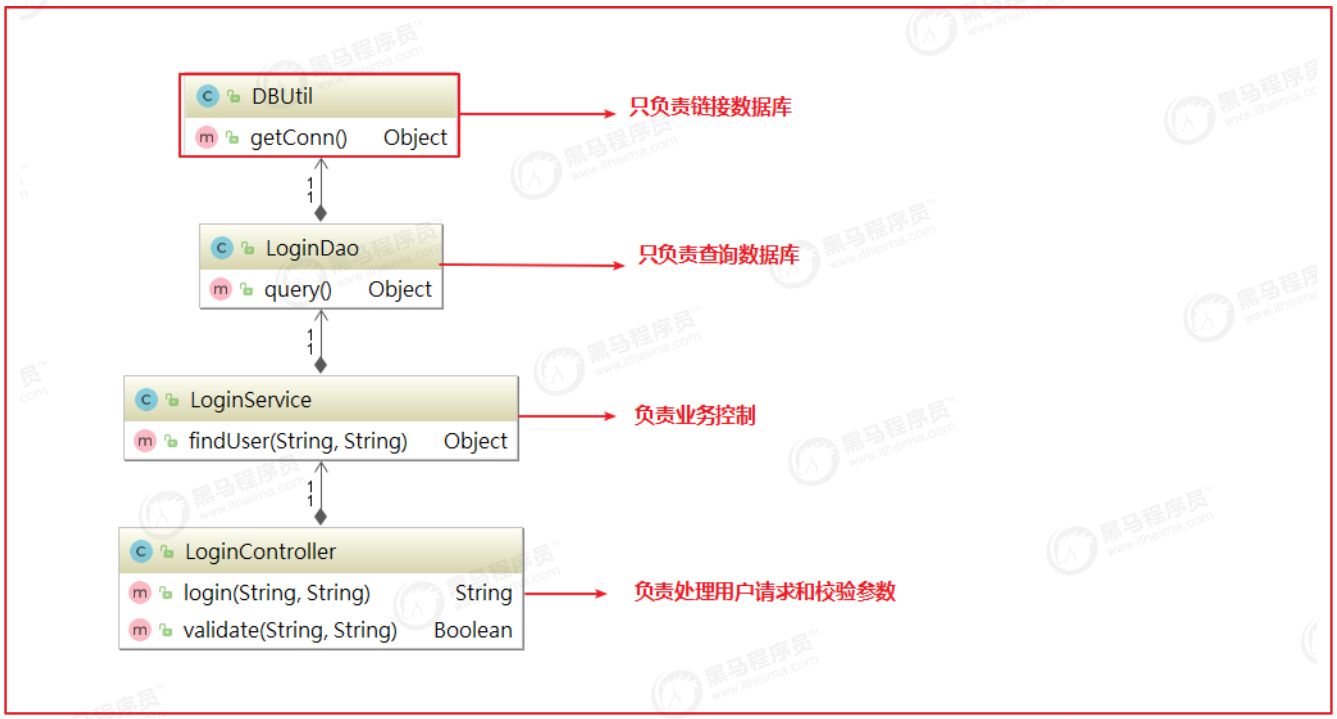
单一职责原则是实现高内聚、低耦合的指导方针，在很多代码重构方法中都能找到它的存在，它是最简单但又最难运用的原则，需要设计人员发现类的不同职责并将其分离，而发现类的多重职责需要设计人员具有较强的分析设计能力和相关重构经验。

实例：以登录实现为例:

原始设计方案:



使用单一职责原则对其进行重构:



1.2.2 开闭原则

定义：一个软件实体应当对扩展开放，对修改关闭。也就是说在设计一个模块的时候，应当使这个模块可以在不被修改的前提下被扩展，即实现在不修改源代码的情况下改变这个模块的行为。

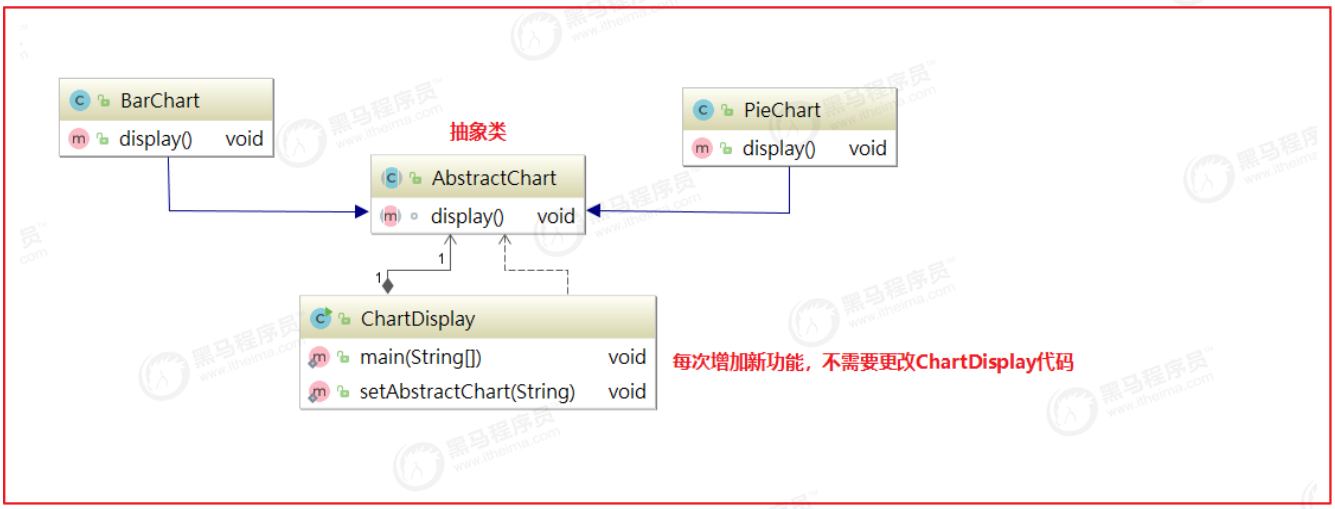
解说：开闭原则还可以通过一个更加具体的"对可变性封装原则"来描述，对可变性封装原则(EVP)要求找到系统的可变因素并将其封装起来。

如果一个软件设计符合开闭原则，那么可以非常方便地对系统进行扩展，而且在扩展时无须修改现有代码，使得软件系统在拥有适应性和灵活性的同时具备较好的稳定性和延续性。为了满足开闭原则，需要对系统进行抽象化设计，抽象化是开闭原则的关键。

实例：我们拿报表功能来说，`BarChart`和`PieChart`为不同的报表功能，此时在`ChartDisplay`中使用报表功能，可以直接new对应的功能，但如果增加新的报表功能，在`ChartDisplay`中使用，就需要改代码了，这就违背了开闭原则。

原始设计方案:

基于开闭原则进行重构：



1.2.3 里氏代换原则

定义：所有引用基类（父类）的地方必须能透明地使用其子类的对象。

解说：里氏代换原则可以通俗表述为：在软件中将一个基类对象替换成它的子类对象，程序将不会产生任 何错误和异常，反过来则不成立，如果一个软件实体使用的是一个子类对象的话，那么它不 一定能够使用基类对象。

里氏代换原则是实现开闭原则的重要方式之一，由于使用基类对象的地方都可以使用子类对 象，因此在程序中尽量使用基类类型来对对象进行定义，而在运行时再确定其子类类型，用 子类对象来替换父类对象。

使用里氏代换原则需要注意:

(1)子类的所有方法必须在父类中声明，或子类必须实现父类中声明的所有方法。根据里氏代 换原则，为了保证系统的扩展性，在程序中通常使用父类来进行定义，如果一个方法只存在子类中，在父类中不提供相应的声明，则无法在以父类定义的对象中使用该方法。

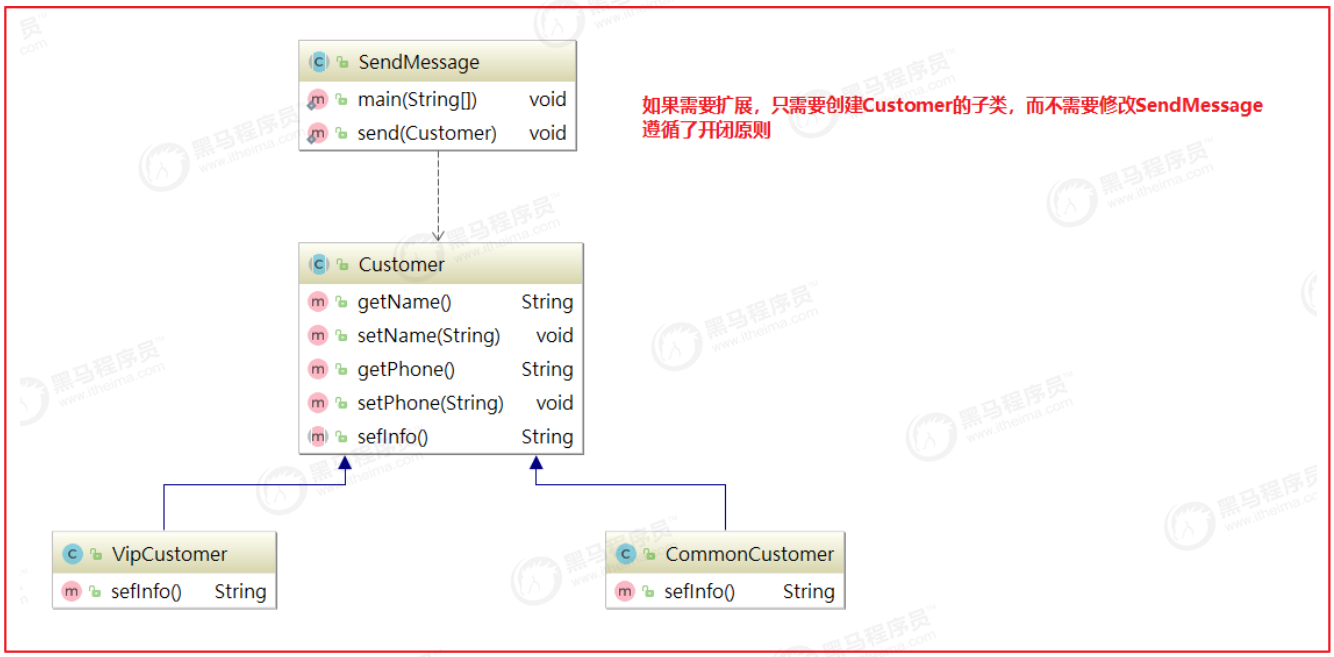
(2) 我们在运用里氏代换原则时，尽量把父类设计为抽象类或者接口，让子类继承父类或实现 父接口，并实现在父类中声明的方法，运行时，子类实例替换父类实例，我们可以很方便地 扩展系统的功能，同时无须修改原有子类的代码，增加新的功能可以通过增加一个新的子类 来实现。里氏代换原则是开闭原则的具体实现手段之一。

(3) Java语言中，在编译阶段，Java编译器会检查一个程序是否符合里氏代换原则，这是一个 与实现无关的、纯语法意义上的检查，但Java编译器的检查是有局限的。

实例：我们以给客户发消息为例，给VIP客户(VipCustomer)和普通客户(CommonCustomer)发消息,在`SendMessage`中分别定义给普通会员和VIP发消息，如果以后有新的客户分类，不仅要添加客户分类，还要修改`SendMessage`，违背了开闭原则。

原始设计方案：

基于里氏代换原则进行重构：



1.2.4 依赖倒转原则

定义：抽象不应该依赖于细节，细节应当依赖于抽象。换言之，要针对接口编程，而不是针对实现编程。

注意点：依赖倒转原则要求我们在程序代码中传递参数时或在关联关系中，尽量引用层次高的抽象层 类，即使用接口和抽象类进行变量类型声明、参数类型声明、方法返回类型声明，以及数据 类型的转换等，而不要用具体类来做这些事情。为了确保该原则的应用，一个具体类应当只 实现接口或抽象类中声明过的方法，而不要给出多余的方法，否则将无法调用到在子类中增 加的新方法。

在引入抽象层后，系统将具有很好的灵活性，在程序中尽量使用抽象层进行编程，而将具体 类写在配置文件中，这样一来，如果系统行为发生变化，只需要对抽象层进行扩展，并修改 配置文件，而无须修改原有系统的源代码，在不修改的情况下来扩展系统的功能，满足开闭 原则的要求。

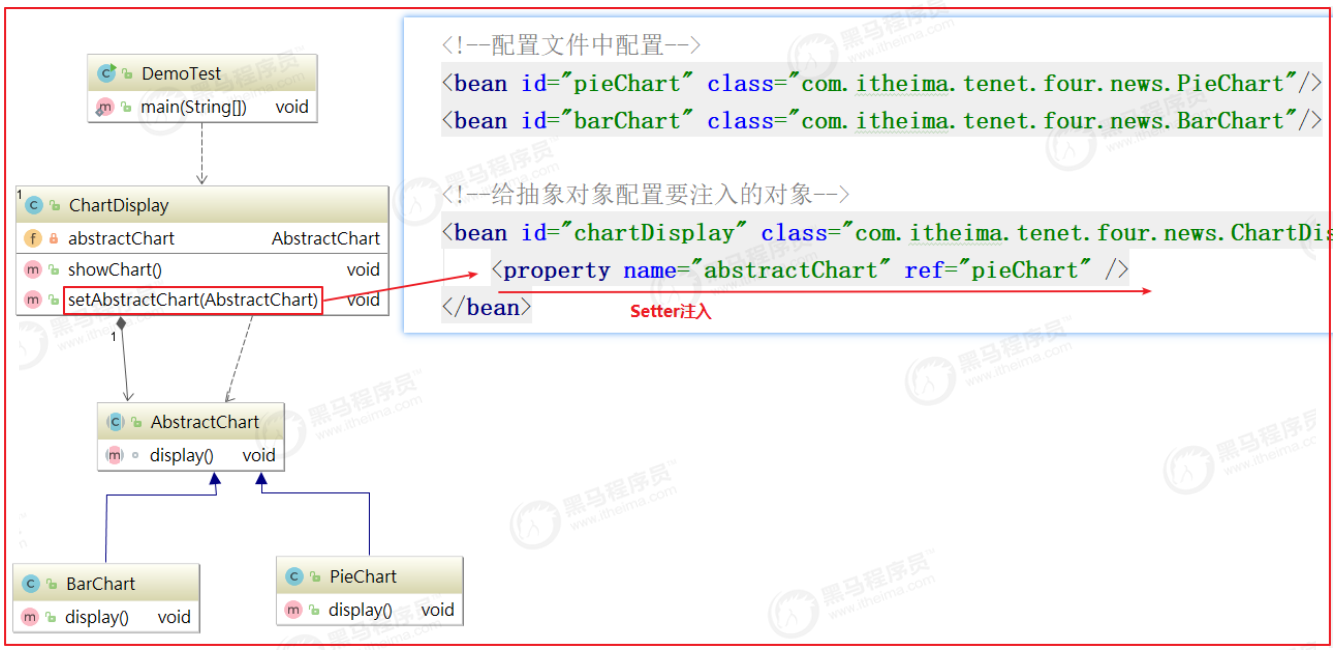
在实现依赖倒转原则时，我们需要针对抽象层编程，而将具体类的对象通过依赖注入 (DependencyInjection, DI)的方式注入到其他对象中，依赖注入是指当一个对象要与其他对象发 生依赖关系时，通过抽象来注入所依赖的对象。常用的注入方式有三种，分别是：构造注 入，设值注入（Setter注入）和接口注入。构造注入是指通过构造函数来传入具体类的对象， 设值注入是指通过Setter方法来传入具体类的对象，而接口注入是指通过在接口中声明的业务 方法来传入具体类的对象。这些方法在定义时使用的是抽象类型，在运行时再传入具体类型 的对象，由子类对象来覆盖父类对象。

总结：

1、针对接口编程
2、在接口或抽象类中定义方法、声明变量
3、类只实现接口或抽象类中的方法，不要定义多余的方法
4、给抽象对象或接口注入依赖对象时，采用依赖注入方式

实例：

我们可以把之前的开闭原则案例修改一下，利用Spring框架进行修改，可读性更强，同时遵循了开闭原则、里氏代换原则和依赖倒转原则，如下图：



1.2.5 接口隔离原则

定义：使用多个专门的接口，而不使用单一的总接口，即客户端不应该依赖那些它不需要的接口。

讲解：接口仅仅提供客户端 需要的行为，客户端不需要的行为则隐藏起来，应当为客户端提供尽可能小的单独的接口， 而不要提供大的总接口。在面向对象编程语言中，实现一个接口就需要实现该接口中定义的 所有方法，因此大的总接口使用起来不一定很方便，为了使接口的职责单一，需要将大接口 中的方法根据其职责不同分别放在不同的小接口中，以确保每个接口使用起来都较为方便， 并都承担某一单一角色。接口应该尽量细化，同时接口中的方法应该尽量少，每个接口中只 包含一个客户端（如子模块或业务逻辑类）所需的方法即可，这种机制也称为“定制服务”，即 为不同的客户端提供宽窄不同的接口。

总结：

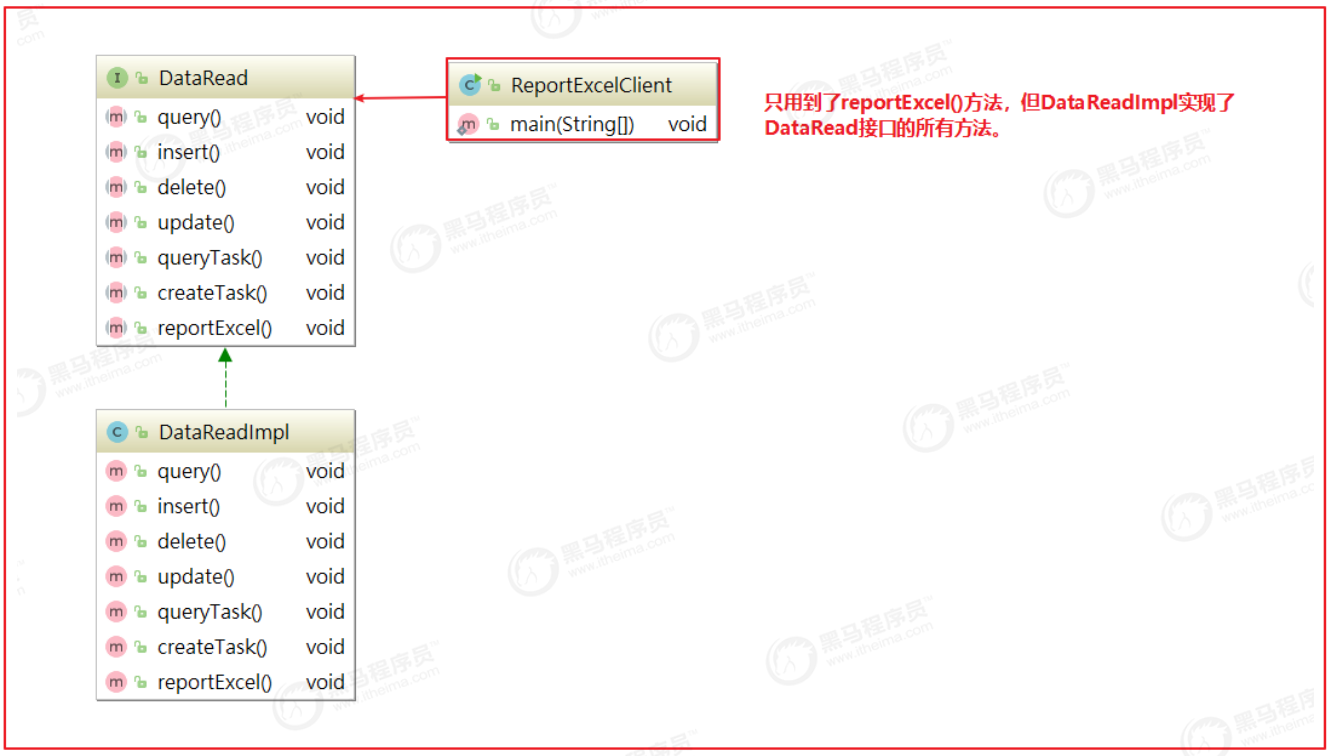
> 需要用到哪些方法，接口中就只提供哪些方法，用不到的方法，接口中不提供。

注意：

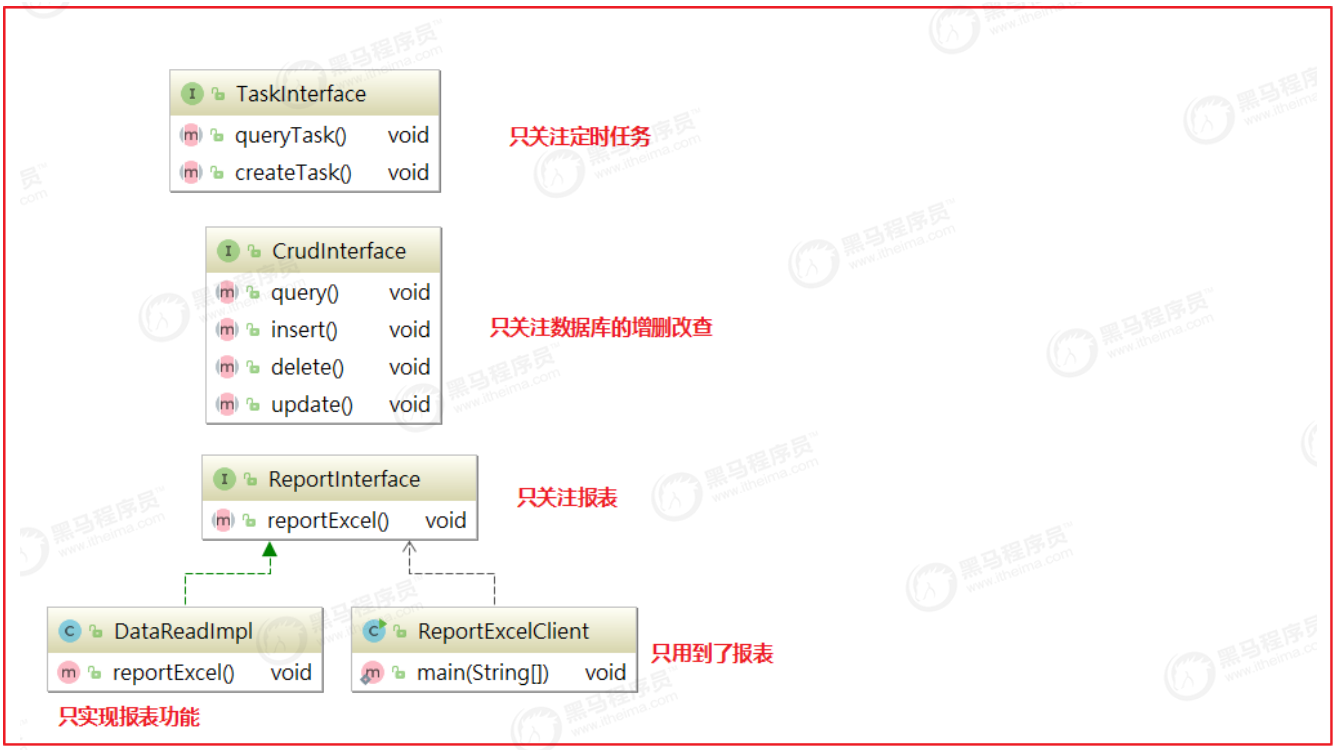
> 我们需要注意控制接口的粒度，接口不能太小，如果太小会导致系 统中接口泛滥，不利于维护；接口也不能太大，太大的接口将违背接口隔离原则，灵活性较 差，使用起来很不方便。一般而言，接口中仅包含为某一类用户定制的方法即可，不应该强 迫客户依赖于那些它们不用的方法。

实例：下图展示了一个拥有多个客户类的系统，在系统中定义了一个巨大的接口DataRead来服务所有的客户类。

原始设计方案：



基于接口隔离原则进行重构：



1.2.6 合成复用原则

定义：尽量使用对象组合，而不是继承来达到复用的目的。

讲解：合成复用原则就是在一个新的对象里通过关联关系（包括组合关系和聚合关系）来使用一些 已有的对象，使之成为新对象的一部分；新对象通过委派调用已有对象的方法达到复用功能 的目的。简言之：复用时要尽量使用组合/聚合关系（关联关系），少用继承。

在面向对象设计中，可以通过两种方法在不同的环境中复用已有的设计和实现，即通过组合/ 聚合关系或通过继承，但首先应该考虑使用组合/聚合，组合/聚合可以使系统更加灵活，降低 类与类之间的耦合度，一个类的变化对其他类造成的影响相对较少；其次才考虑继承，在使 用继承时，需要严格遵循里氏代换原则，有效使用继承会有助于对问题的理解，降低复杂 度，而滥用继承反而会增加系统构建和维护的难度以及系统的复杂度，因此需要慎重使用继 承复用。

通过继承来进行复用的主要问题在于继承复用会破坏系统的封装性，因为继承会将基类的实 现细节暴露给子类，由于基类的内部细节通常对子类来说是可见的，所以这种复用又称“白 箱”复用，如果基类发生改变，那么子类的实现也不得不发生改变；

由于组合或聚合关系可以将已有的对象（也可称为成员对象）纳入到新对象中，使之成为新对象的一部分，因此新对象可以调用已有对象的功能，这样做可以使得成员对象的内部实现 细节对于新对象不可见。

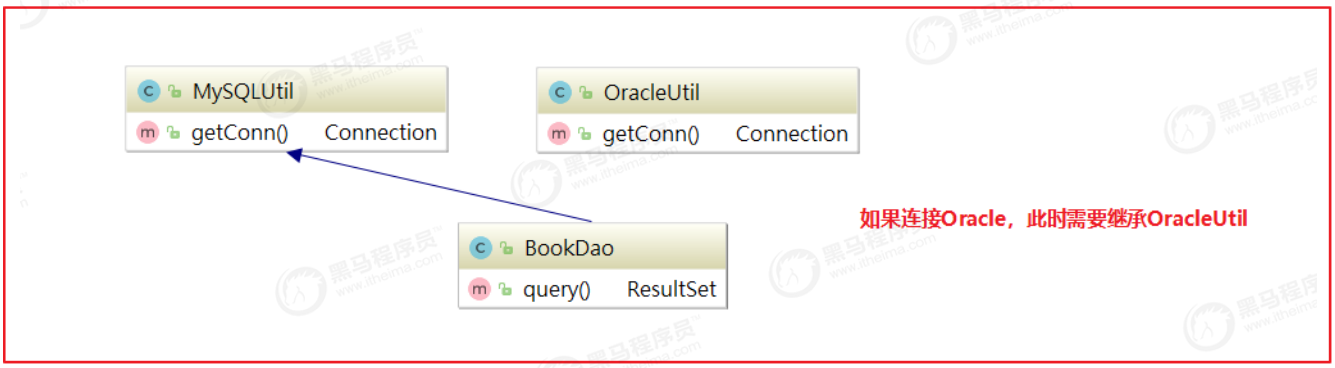
总结：

复用的方式：
①组合/聚合关系实现复用
②继承实现复用

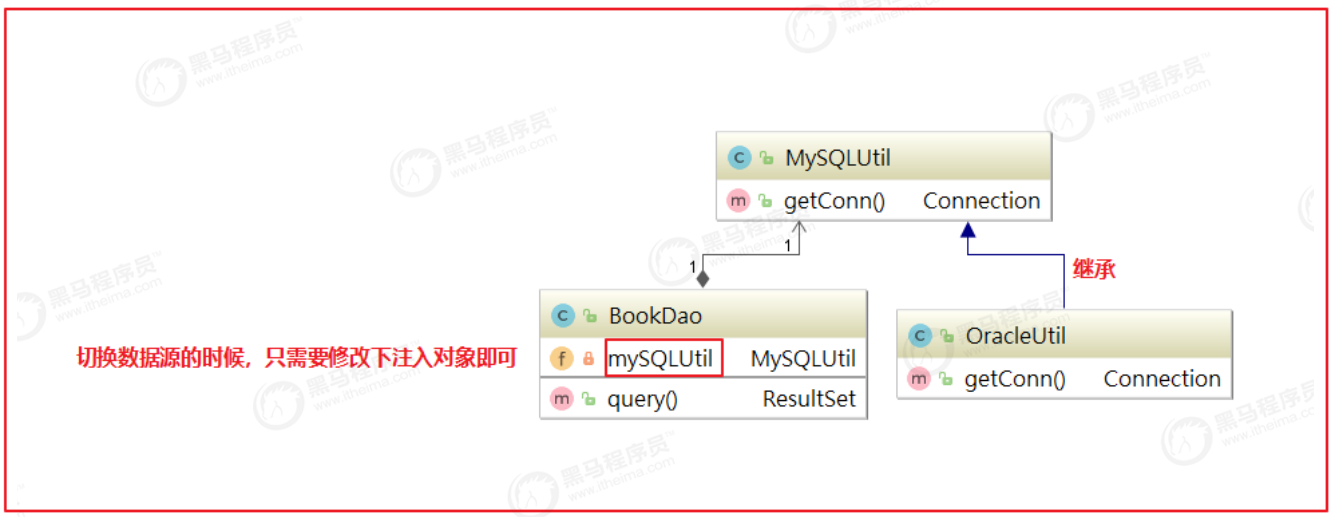
继承复用问题：会破坏系统的封装性，会把基类实现暴露给子类。
组合/聚合复用：已有对象的功能细节，对组合而成的新对象是不可见的，封装性教好。

实例：图书管理系统中，如果数据在MySQL中，我们需要创建一个链接MySQL的工具类`MySQLUtil`,Dao只需要继承该工具类即可操作数据库，如果把数据库换成Oracle，我们需要新建一个工具类`OracleUtil`，Dao需要修改继承对象改为`OracleUtil`,这就违反了开闭原则。

原始设计方案：



基于合成复用原则进行重构：



我们把`OracleUtil`作为`MySQLUtil`的子类，BookDao中把`MySQLUtil`作为一个属性组合进来，每次需要变更数据库链接的时候，只需要修改BookDao的依赖注入配置文件即可。这里符合里氏替换原则。

1.2.7 迪米特法则

定义：一个软件实体应当尽可能少地与其他实体发生相互作用。

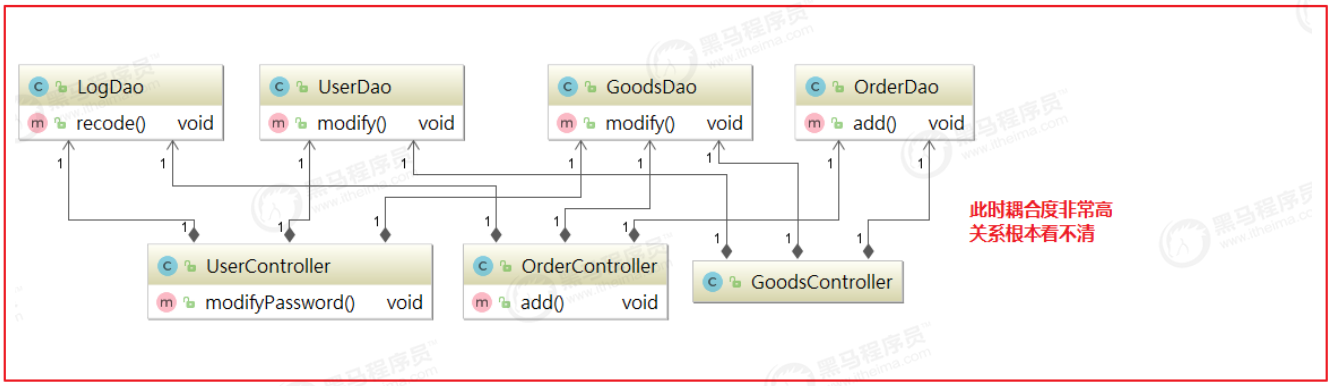
讲解：如果一个系统符合迪米特法则，那么当其中某一个模块发生修改时，就会尽量少地影响其他 模块，扩展会相对容易，这是对软件实体之间通信的限制，迪米特法则要求限制软件实体之 间通信的宽度和深度。迪米特法则可降低系统的耦合度，使类与类之间保持松散的耦合关系。

迪米特法则要求我们在设计系统时，应该尽量减少对象之间的交互，如果两个对象之间不必 彼此直接通信，那么这两个对象就不应当发生任何直接的相互作用，如果其中的一个对象需 要调用另一个对象的某一个方法的话，可以通过第三者转发这个调用。简言之，就是**通过引 入一个合理的第三者来降低现有对象之间的耦合度。**

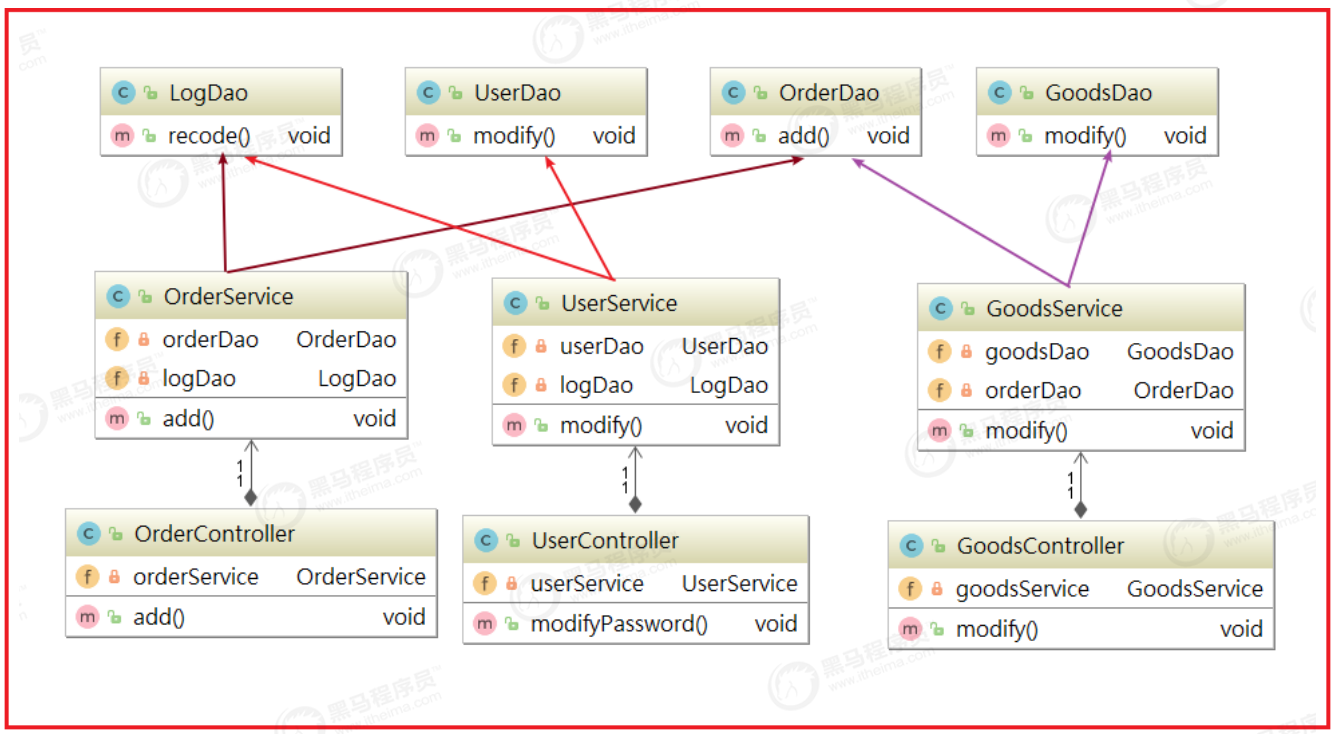
作用：降低系统的耦合度

实例：我们在做增删改查的时候，如果直接用控制层调用Dao，业务处理的关系会比较乱，我们需要合理增加一个中间对象(业务层)来解决个问题。

原始设计方案：



基于迪米特法则进行重构：



1.3 设计模式分类

GOF中共提到了23种设计模式不是孤立存在的，很多模式之间存在一定的关联关系，在大的系统开发中常常同时使用多种设计模式。这23种设计模式根据功能作用来划分，可以划分为3类：

(1)创建型模式：用于描述“怎样创建对象”，它的主要特点是“将对象的创建与使用分离”，单例、原型、工厂方法、抽象工厂、建造者5种设计模式属于创建型模式。

(2)结构型模式：用于描述如何将类或对象按某种布局组成更大的结构，代理、适配器、桥接、装饰、外观、享元、组合7种设计模式属于结构型模式。

(3)行为型模式：用于描述类或对象之间怎样相互协作共同完成单个对象都无法单独完成的任务，以及怎样分配职责。模板方法、策略、命令、职责链、状态、观察者、中介者、迭代器、访问者、备忘录、解释器11种设计模式属于行为型模式。

GOF的23种设计模式：

1、单例（Singleton）模式：某个类只能生成一个实例，该类提供了一个全局访问点供外部获取该实例，其拓展是有限多例模式。
2、原型（Prototype）模式：将一个对象作为原型，通过对其进行复制而克隆出多个和原型类似的新实例。
3、工厂方法（Factory Method）模式：定义一个用于创建产品的接口，由子类决定生产什么产品。
4、抽象工厂（AbstractFactory）模式：提供一个创建产品族的接口，其每个子类可以生产一系列相关的产品。
5、建造者（Builder）模式：将一个复杂对象分解成多个相对简单的部分，然后根据不同需要分别创建它们，最后构建成该复杂对象。

6、代理（Proxy）模式：为某对象提供一种代理以控制对该对象的访问。即客户端通过代理间接地访问该对象，从而限制、增强或修改该对象的一些特性。
7、适配器（Adapter）模式：将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。
8、桥接（Bridge）模式：将抽象与实现分离，使它们可以独立变化。它是用组合关系代替继承关系来实现，从而降低了抽象和实现这两个可变维度的耦合度。
9、装饰（Decorator）模式：动态的给对象增加一些职责，即增加其额外的功能。
10、外观（Facade）模式：为多个复杂的子系统提供一个一致的接口，使这些子系统更加容易被访问。
11、享元（Flyweight）模式：运用共享技术来有效地支持大量细粒度对象的复用。
12、组合（Composite）模式：将对象组合成树状层次结构，使用户对单个对象和组合对象具有一致的访问性。

13、模板方法（TemplateMethod）模式：定义一个操作中的算法骨架，而将算法的一些步骤延迟到子类中，使得子类可以不改变该算法结构的情况下重定义该算法的某些特定步骤。
14、策略（Strategy）模式：定义了一系列算法，并将每个算法封装起来，使它们可以相互替换，且算法的改变不会影响使用算法的客户。
15、命令（Command）模式：将一个请求封装为一个对象，使发出请求的责任和执行请求的责任分割开。
16、职责链（Chain of Responsibility）模式：把请求从链中的一个对象传到下一个对象，直到请求被响应为止。通过这种方式去除对象之间的耦合。
17、状态（State）模式：允许一个对象在其内部状态发生改变时改变其行为能力。
18、观察者（Observer）模式：多个对象间存在一对多关系，当一个对象发生改变时，把这种改变通知给其他多个对象，从而影响其他对象的行为。
19、中介者（Mediator）模式：定义一个中介对象来简化原有对象之间的交互关系，降低系统中对象间的耦合度，使原有对象之间不必相互了解。
20、迭代器（Iterator）模式：提供一种方法来顺序访问聚合对象中的一系列数据，而不暴露聚合对象的内部表示。
21、访问者（Visitor）模式：在不改变集合元素的前提下，为一个集合中的每个元素提供多种访问方式，即每个元素有多个访问者对象访问。
22、备忘录（Memento）模式：在不破坏封装性的前提下，获取并保存一个对象的内部状态，以便以后恢复它。
23、解释器（Interpreter）模式：提供如何定义语言的放法，以及对语言句子的解释方法，即解释器。

2 设计模式常用案例

2.1 单利模式

单例模式（Singleton Pattern）是 Java 中最常见的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

单利模式涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建。该类还提供了一种访问它唯一对象的方式，其他类可以直接访问该方法获取该对象实例，而不需要实例化该类的对象。

单利模式特点：

1、单例类只能有一个实例。
2、单例类必须自己创建自己的唯一实例。
3、单例类必须给所有其他对象提供这一实例。

单利模式优点：

1、在内存里只有一个实例，减少了内存的开销，尤其是频繁的创建和销毁实例。
2、避免对资源的多重占用（比如写文件操作）。

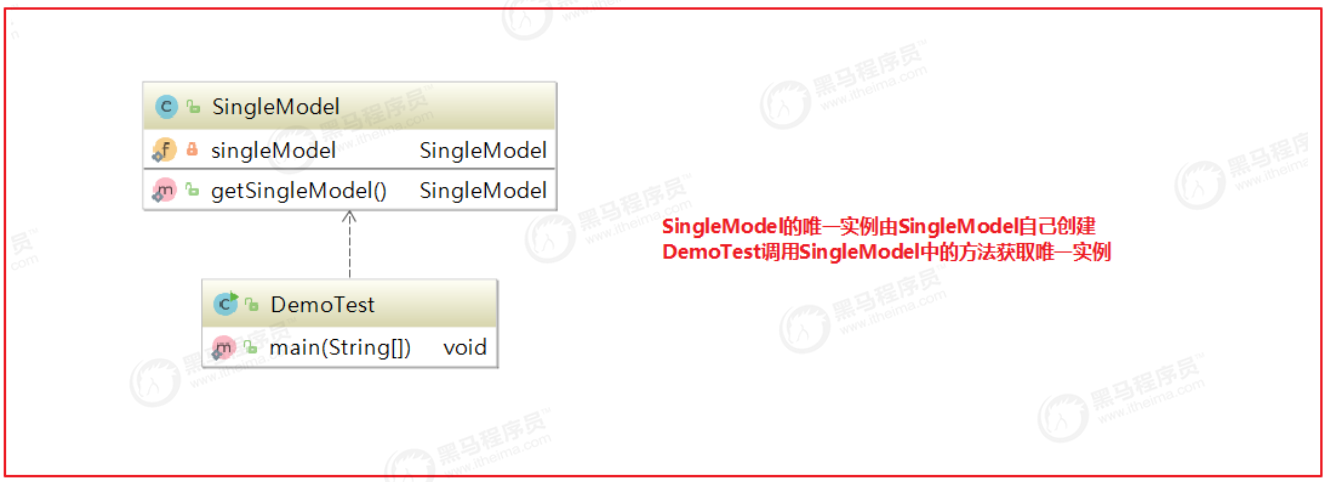
单利模式真实应用场景：

1、网站的计数器
2、应用程序的日志应用
3、数据库连接池设计
4、多线程的线程池设计

2.1.1 单利模式-饿汉式

创建一个单利对象`SingleModel`,`SingleModel`类有它的私有构造函数和本身的一个静态实例。

`SingleModel`类提供了一个静态方法，供外界获取它的静态实例。`DesignTest`我们的演示类使用`SingleModel`类来获取`SingleModel`对象。



创建`SingleModel`：

```
public class SingleModel {

    //创建 SingleModel 的一个对象
    private static SingleModel instance = new SingleModel();

    //让构造函数为 private，这样该类就不会被实例化
    private SingleModel(){}

    //获取唯一可用的对象
    public static SingleModel getInstance(){
        return instance;
    }

    public void useMessage(){
        System.out.println("Single Model!");
    }
}
```

单利测试：

```
public class DemoTest {

    /****
     * 单利模式测试
     */
    @Test
    public void testSingleModel(){
        //不合法的构造函数
        //编译时错误：构造函数 SingleModel() 是不可见的
        //SingleModel singleModel = new SingleModel();

        //获取唯一可用的对象
        SingleModel singleModel1 = SingleModel.getInstance();
        SingleModel singleModel2 = SingleModel.getInstance();

        //显示消息
        singleModel1.useMessage();

        //创建的2个对象是同一个对象
        System.out.println(singleModel1 == singleModel2);
    }
}
```

输入结果如下：

```
Single Model!
true
```

我们测试创建10万个对象，用单利模式创建，仅占内存：`104` 字节，而如果用传统方式创建10万个对象，占内存大小为 `2826904` 字节。

2.1.2 多种单利模式讲解

单利模式有多种创建方式，刚才创建方式没有特别的问题，但是程序启动就需要创建对象，不管你用不用到对象，都会创建对象，都会消耗一定内存。因此在单利的创建上出现了多种方式。

懒汉式：

懒汉式有这些特点：

1、延迟加载创建，也就是用到对象的时候，才会创建
2、线程安全问题需要手动处理(不添加同步方法，线程不安全，添加了同步方法，效率低)
3、实现容易

案例如下：`SingleModel1`

```
public class SingleModel1 {

    //不实例化
    private static SingleModel1 instance;

    //让构造函数为 private，这样该类就不会被实例化
    private SingleModel1(){}

    //获取唯一可用的对象
    public static SingleModel1 getInstance(){
        //instance为空的时候才创建对象，这里线程不安全
        if(instance==null){
            instance = new SingleModel1();
        }
        return instance;
    }

    public void useMessage(){
        System.out.println("Single Model!");
    }
}
```

①不直接实例化

instance对象为空的时候，才创建对象
如果是多线程场景下，这里不安全

如果在创建对象实例的方法上添加同步 `synchronized`,但是这种方案效率低，代码如下：

```
//获取唯一可用的对象
public static synchronized SingleModel1 getInstance(){
    //instance为空的时候才创建对象，这里线程不安全
    if(instance==null){
        instance = new SingleModel1();
    }
    return instance;
}
```

添加了同步可以解决多线程安全问题，但这么做明显会降低获取对象实例的效率。

双重校验锁： `SingleModel2`

这种方式采用双锁机制，安全且在多线程情况下能保持高性能。

```
public class SingleModel2 {

    //不实例化
    private static SingleModel2 instance;

    //让构造函数为 private，这样该类就不会被实例化
    private SingleModel2(){}

    //获取唯一可用的对象
    public static SingleModel2 getInstance(){
        //instance为空的时候才创建对象
        if(instance==null){
            //同步锁，效率比懒汉式高
            synchronized (SingleModel2.class){
                //这里需要判断第2次为空
                if(instance==null){
                    instance = new SingleModel2();
                }
            }
        }
        return instance;
    }

    public void useMessage(){
        System.out.println("Single Model!");
    }
}
```

3 Spring设计模式剖析

Spring是一个分层的JavaSE/EE full-stack(一站式) 轻量级开源框架，非常受企业欢迎，他解决了业务逻辑层和其他各层的松耦合问题，它将面向接口的编程思想贯穿整个系统应用。在Spring源码中拥有多个优秀的设计模式使用场景，有非常高的学习价值。

3.1 观察者模式

定义：

对象之间存在一对多或者一对一依赖，当一个对象改变状态，依赖它的对象会收到通知并自动更新。
MQ其实就属于一种观察者模式，发布者发布信息，订阅者获取信息，订阅了就能收到信息，没订阅就收不到信息。

优点：

1、观察者和被观察者是抽象耦合的。
2、建立一套触发机制。

缺点：

1、如果一个被观察者对象有很多的直接和间接的观察者的话，将所有的观察者都通知到会花费很多时间。
2、如果在观察者和观察目标之间有循环依赖的话，观察目标会触发它们之间进行循环调用，可能导致系统崩溃。

3.1.1 Spring观察者模式

`ApplicationContext` 事件机制是观察者设计模式的实现，通过 `ApplicationEvent` 类和 `ApplicationListener` 接口，可以实现 `ApplicationContext` 事件处理。

如果容器中有一个 `ApplicationListener Bean`，每当 `ApplicationContext` 发布 `ApplicationEvent` 时，`ApplicationListener Bean` 将自动被触发。这种事件机制都必须需要程序显示的触发。

其中spring有一些内置的事件，当完成某种操作时会发出某些事件动作。比如监听 `ContextRefreshedEvent` 事件，当所有的bean都初始化完成并被成功装载后会触发该事件，实现 `ApplicationListener<ContextRefreshedEvent>` 接口可以收到监听动作，然后可以写自己的逻辑。

同样事件可以自定义、监听也可以自定义，完全根据自己的业务逻辑来处理。

对象说明：

1、ApplicationContext容器对象
2、ApplicationEvent事件对象（ContextRefreshedEvent容器刷新事件）
3、ApplicationListener事件监听对象

3.1.2 ApplicationContext事件监听

当ApplicationContext内的Bean对象初始化完成时，此时可以通过监听 `ContextRefreshedEvent` 得到通知！我们来模拟一次。

创建监听对象：`ApplicationContextListener`

```
public class ApplicationContextListener implements
ApplicationListener<ContextRefreshedEvent> {
    @Override
    public void onApplicationEvent(ContextRefreshedEvent event) {
        System.out.println("ContextRefreshedEvent事件,容器内对象发生变更");
    }
}
```

将对象添加到容器中：

```
<bean class="com.itheima.event.ApplicationContextListener" id="appListener"/>
```

测试：

```
public static void main(String[] args) throws InterruptedException {
    ApplicationContext act = new ClassPathXmlApplicationContext("spring.xml");
}
```

此时会打印如下信息：

```
ContextRefreshedEvent事件,容器内对象发生变更
```

应用场景：

```
程序启动，初始化过程中，需要确保所有对象全部初始化完成，此时在从容器中获取指定对象做相关初始化操作。
例如：将省、市、区信息初始化到缓存中。
```

3.1.3 自定义监听事件

自定义监听事件可以监听容器变化，同时也能精确定位指定事件对象，我们编写一个案例演示自定义监听事件实现流程。

定义事件监听对象：`MessageNotifier`

```
public class MessageNotifier implements ApplicationListener {

    @Override
    public void onApplicationEvent(ApplicationEvent event) {
        System.out.println("事件对象："+event.getClass().getName());
    }
}
```

定义事件对象：`MessageEvent`

```
public class MessageEvent extends ApplicationEvent {

    private String phone;
```

```
    private String message;

    /**
     * 创建事件
     */
    public MessageEvent(Object source,String phone,String message) {
        super(source);
        this.phone=phone;
        this.message=message;
    }

    /**
     * 创建事件
     */
    public MessageEvent(Object source) {
        super(source);
    }

    //get..set..
}
```

将对象添加到容器中：

```
<bean class="com.itheima.event.MessageNotifier" id="messageNotifier"/>
```

添加事件测试：

```
public static void main(String[] args) throws InterruptedException {
    ApplicationContext act = new ClassPathXmlApplicationContext("spring.xml");
    MessageEvent messageEvent = new MessageEvent("beijing","13670000000","hello!");
    act.publishEvent(messageEvent);
}
```

测试打印结果如下：

```
事件对象：org.springframework.context.event.ContextRefreshedEvent
事件对象：com.itheima.event.MessageEvent
```

3.2 代理模式

定义：

给某对象提供一个代理对象，通过代理对象可以访问该对象的功能。主要解决通过代理去访问[不能直接访问的对象]，例如租房中介，你可以直接通过中介去了解房东的房源信息，此时中介就可以称为代理。

优点：

1、职责清晰。
2、高扩展性。
3、智能化。

缺点：

1、由于在客户端和真实主题之间增加了代理对象，因此有些类型的代理模式可能会造成请求的处理速度变慢。
2、实现代理模式需要额外的工作，有些代理模式的实现非常复杂。

代理实现方式：

基于接口的动态代理
    提供者：JDK官方的Proxy类。
    要求：被代理类最少实现一个接口。
基于子类的动态代理
    提供者：第三方的CGLib，如果报asmxxxx异常，需要导入asm.jar。
    要求：被代理类不能用final修饰的类（最终类）。

3.2.1 JDK动态代理

JDK动态代理要点：

1、被代理的类必须实现一个接口
2、创建代理对象的时候，用JDK代理需要实现InvocationHandler
3、代理过程在invoke中实现

我们以王五租房为例，王五通过中介直接租用户主房屋，中介在这里充当代理角色，户主充当被代理角色。

创建房东接口对象：`LandlordService`

```
public interface LandlordService {
    void rentingPay(String name);
}
```

创建房东对象：`Landlord`

```
public class Landlord implements LandlordService{

    /****
     * @param name
     */
    @Override
    public void rentingPay(String name){
        System.out.println(name+" 来交租! ");
    }
}
```

创建代理处理过程对象：`QFangProxy`

```
public class QFangProxy implements InvocationHandler{

    private Object instance;

    public QFangProxy(Object instance) {
        this.instance = instance;
    }

    /****
     * 代理过程
     * @throws Throwable
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        args[0] = "中介QFang带领租户"+args[0];
        Object result = method.invoke(instance, args);
        return result;
    }
}
```

创建代理，并通过代理调用房东方法：`JdkProxyTest`

```
public class JdkProxyTest {
    public static void main(String[] args) {
        //给QFang产生代理
        LandlordService landlordService = new Landlord();
        QFangProxy proxy = new QFangProxy(landlordService);
        LandlordService landlordServiceProxy = (LandlordService)
Proxy.newProxyInstance(LandlordService.class.getClassLoader(), new Class[]
{LandlordService.class}, proxy);

        //通过代理对象调用Landlord对象的方法
        landlordServiceProxy.rentingPay("王五");
    }
}
```

运行结果如下：

```
中介QFang带领客户 来交租!
```

3.2.2 CGLib动态代理

CGLib动态代理要点：

```
1、代理过程可以实现MethodInterceptor(Callback)接口中的invoke来实现
2、通过Enhancer来创建代理对象
```

在上面的案例基础上，把 `QFangProxy` 换成 `SFangProxy`，代码如下：

```
public class SFangProxy implements MethodInterceptor {

    private Object instance;

    public SFangProxy(Object instance) {
        this.instance = instance;
    }

    /***
     * 代理过程
     * @throws Throwable
     */
    @Override
    public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy)
throws Throwable {
        args[0]="S房网带租户"+args[0];
        return method.invoke(instance,args);
    }
}
```

创建测试类: `CGLibProxyTest`，代码如下

```
public class CGLibProxyTest {
    public static void main(String[] args) {
        //给QFang产生代理
        LandlordService landlordService = new Landlord();
        SFangProxy proxy = new SFangProxy(landlordService);
        LandlordService landlordServiceProxy = (LandlordService)
Enhancer.create(LandlordService.class,proxy);

        //通过代理对象调用Landlord对象的方法
        landlordServiceProxy.rentingPay("王五");
    }
}
```

3.2.3 Spring AOP-动态代理

基于SpringAOP可以实现非常强大的功能，例如声明式事务、基于AOP的日志管理、基于AOP的权限管理等功能，利用AOP可以将重复的代码抽取，重复利用，节省开发时间，提升开发效率。Spring的AOP其实底层就是基于动态代理而来，并且支持JDK动态代理和CGLib动态代理，动态代理的集中体现在`DefaultAopProxyFactory`类中，我们来解析下`DefaultAopProxyFactory`类。

```
public class DefaultAopProxyFactory implements AopProxyFactory, Serializable {

    @Override
    public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigException {
        if (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppliedProxyInterfaces(config)) {
            Class<?> targetClass = config.getTargetClass();
            if (targetClass == null) {
                throw new AopConfigException("TargetSource cannot determine target class: " +
                        "Either an interface or a target is required for proxy creation.");
            }
            if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
                return new JdkDynamicAopProxy(config);
            }
            return new ObjenesisCglibAopProxy(config);
        }
        else {
            return new JdkDynamicAopProxy(config);
        }
    }
}
```

启用CGLib动态代理：`<aop:config proxy-target-class="true">`

JDK动态代理

CGLib动态代理

JDK动态代理

如果我们在spring的配置文件中不配置`<aop:config proxy-target-class="true">`，此时默认使用的将是JDK动态代理，如果配置了，则会使用CGLib动态代理。

JDK动态代理的创建`JdkDynamicAopProxy`如下：

```
final class JdkDynamicAopProxy implements AopProxy, InvocationHandler, Serializable {

    //创建代理对象
    @Override
    public Object getProxy(@Nullable ClassLoader classLoader) {
        if (logger.isDebugEnabled()) {
            logger.debug("Creating JDK dynamic proxy: target source is " +
this.advised.getTargetSource());
        }
        Class<?>[] proxiedInterfaces =
AopProxyUtils.completeProxiedInterfaces(this.advised, true);
        findDefinedEqualsAndHashCodeMethods(proxiedInterfaces);
        return Proxy.newProxyInstance(classLoader, proxiedInterfaces, this);
    }

    @Override
    @Nullable
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        //JDK动态代理过程
        }
    }
```

```
}
```

CGLib动态代理的创建`ObjenesisCglibAopProxy`如下：

```
class ObjenesisCglibAopProxy extends CglibAopProxy {

    //CGLib动态代理创建过程
    @Override
    @SuppressWarnings("unchecked")
    protected Object createProxyClassAndInstance(Enhancer enhancer, Callback[] callbacks) {
        Class<?> proxyClass = enhancer.createClass();
        Object proxyInstance = null;

        if (objenesis.isWorthTrying()) {
            try {
                proxyInstance = objenesis.newInstance(proxyClass, enhancer.getUseCache());
            }
            catch (Throwable ex) {
                logger.debug("Unable to instantiate proxy using Objenesis, " +
                        "falling back to regular proxy construction", ex);
            }
        }

        if (proxyInstance == null) {
            // Regular instantiation via default constructor...
            try {
                Constructor<?> ctor = (this.constructorArgs != null ?
                        proxyClass.getDeclaredConstructor(this.constructorArgTypes) :
                        proxyClass.getDeclaredConstructor());
                ReflectionUtils.makeAccessible(ctor);
                proxyInstance = (this.constructorArgs != null ?
                        ctor.newInstance(this.constructorArgs) : ctor.newInstance());
            }
            catch (Throwable ex) {
                throw new AopConfigException("Unable to instantiate proxy using Objenesis,
" +
                        "and regular proxy instantiation via default constructor fails as
well", ex);
            }
        }

        ((Factory) proxyInstance).setCallbacks(callbacks);
        return proxyInstance;
    }
}
```

3.3 工厂设计模式

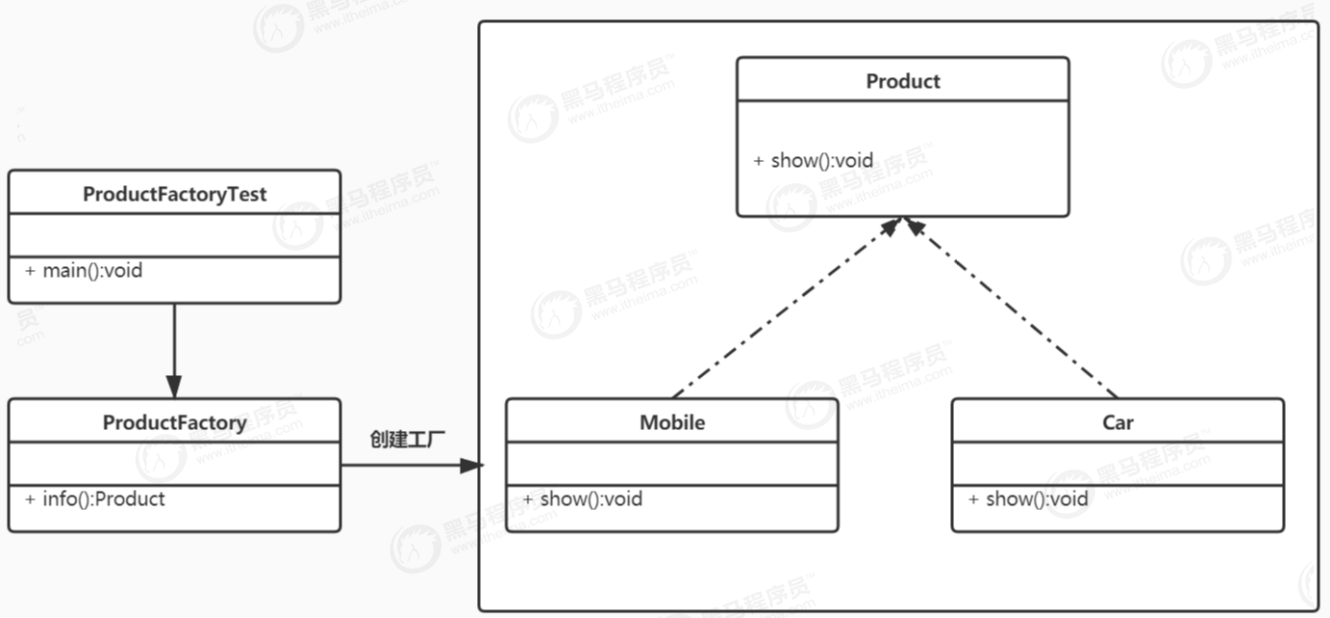
定义：

工厂模式（Factory Pattern）是 Java 中最常用的设计模式之一。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。它负责实现创建所有实例的内部逻辑。工厂类的创建产品类的方法可以被外界直接调用，创建所需的产品对象。

优点：

1、一个调用者想创建一个对象，只要知道其名称就可以了。
2、屏蔽产品的具体实现，调用者只关心产品的接口。
3、降低了耦合度

3.3.1 工厂模式案例



我们来做这么一个案例，创建一个接口Product和Product的实现类Mobile以及Car，再定义一个具体的工厂对象ProcutFactory，并通过ProductFactory来获取指定的Product。

Product接口：

```
public interface Product {
    void show();
}
```

创建接口实现类Mobile:

```
public class Mobile implements Product {
    @Override
    public void show() {
        System.out.println("HUWEI Mate 30 Pro");
    }
}
```

创建接口实现类Car:

```
public class Car implements Product {
    @Override
    public void show() {
        System.out.println("国产车 BYD");
    }
}
```

创建工厂ProductFactory，根据参数创建指定产品对象：

```
public class ProductFactory {

    /***
     * 调用info方法，传入需要创建的对象的名字来创建具体对象
     * @param name
     * @return
     */
    public Product info(String name){
        if(name.equals("mobile")){
            return new Mobile();
        }else if(name.equals("car")){
            return new Car();
        }else{
            return null;
        }
    }
}
```

使用工厂ProductFactory创建指定对象：

```
public class ProductFactoryTest {

    public static void main(String[] args) {
        //创建ProductFactory
        ProductFactory factory = new ProductFactory();

        //获取Mobile对象
        Product mobile = factory.info("mobile");
        mobile.show();
        //获取Car
        Product car = factory.info("car");
        car.show();
    }
}
```

运行结果如下：

```
HUWEI Mate 30 Pro
国产车 BYD
```

3.3.2 BeanFactory工厂模式

Spring内部源码也有工厂模式的实现，并且解决了上面我们提到的工厂模式的缺陷问题。

Spring中的BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识来获得Bean对象，但是否是在传入参数后创建还是传入参数前创建这个要根据具体情况来定。

BeanFactory源码：

```
public interface BeanFactory {

    //根据参数获取指定对象的实例
    Object getBean(String name) throws BeansException;
    <T> T getBean(String name, @Nullable Class<T> requiredType) throws BeansException;
    Object getBean(String name, Object... args) throws BeansException;
    <T> T getBean(Class<T> requiredType) throws BeansException;
    <T> T getBean(Class<T> requiredType, Object... args) throws BeansException;
}
```

在BeanFactory接口中，有多个getBean方法，该方法其实就是典型的工厂设计模式特征，在接口中定义了创建对象的方法，而对象如何创建其实在接口的实现类中实现`DefaultListableBeanFactory`。

我们用Spring的工厂对象BeanFactory来解决上面工厂模式案例所带来的问题。

案例：

xml:

```
<bean id="car" class="com.itheima.factory.Car"/>
<bean id="mobile" class="com.itheima.factory.Mobile"/>
```

测试：

```
public void testBeanFactory(){
    //加载核心配置文件  beas.xml
    Resource resource = new ClassPathResource("beans.xml");
    //创建容器对象BeanFactory
    BeanFactory beanFactory = new XmlBeanFactory(resource);
    //使用Car对象
    Product car = (Product) beanFactory.getBean("car");
    //使用Mobile对象
    Product mobile = (Product) beanFactory.getBean("mobile");
    car.show();
    mobile.show();
}
```

结果：

```
国产车 BYD
HUWEI Mate 30 Pro
```

使用Spring的BeanFactory，以后要新增一个产品，只需要创建产品对应的xml配置即可，而不需要像`ProductFactory`那样硬编码存在。

3.4 适配器模式

定义：

```
将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类能一起工作。
```

优点：

```
1、可以让任何两个没有关联的类一起运行。
2、提高了类的复用。
3、灵活性好。
```

缺点：

```
过多地使用适配器，会让系统非常零乱，不易整体进行把握。比如，明明看到调用的是 A 接口，其实内部被适配成了 B 接口的实现，一个系统如果太多出现这种情况，无异于一场灾难。
```

3.4.1 Spring Aop适配器+代理模式案例

Spring架构中涉及了很多设计模式，本文来介绍下Spring中在AOP实现时Adapter模式的使用。AOP本质上是Java动态代理模式的实现和适配器模式的使用。

我们基于Spring的前置通知来实现一个打卡案例，再基于前置通知讲解前置适配模式。

创建打卡接口：`PunchCard`定义打卡方法，代码如下：

```
public interface PunchCard {

    //打卡记录
    void info(String name);
}
```

定义打卡实现：`PunchCardImpl`实现打卡操作，代码如下：

```
public class PunchCardImpl implements PunchCard {
    @Override
    public void info(String name) {
        System.out.println(name+" 打卡成功！");
    }
}
```

前置通知创建：`PunchCardBefore`实现在打卡之前识别用户身份，代码如下：

```
public class PunchCardBefore implements MethodBeforeAdvice {
    @Override
    public void before(Method method, Object[] args, Object target) throws Throwable {
        System.out.println("身份识别通过！");
    }
}
```

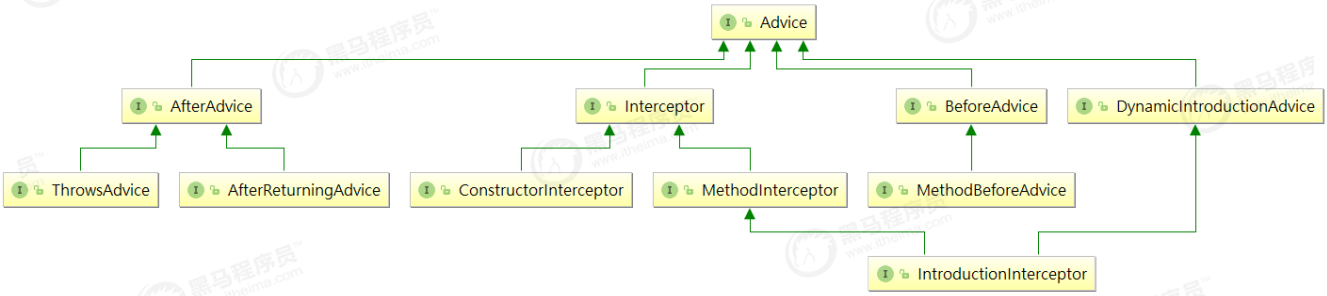
spring.xml配置前置通知：

```
<!--打卡-->
<bean id="punchCard" class="com.itheima.adapter.PunchCardImpl"></bean>

<!--前置通知-->
<bean id="punchCardBefore" class="com.itheima.adapter.PunchCardBefore"></bean>

<!--代理配置-->
<bean id="proxyFactoryBean" class="org.springframework.aop.framework.ProxyFactoryBean">
    <!-- 指定目标对象 -->
    <property name="target" ref="punchCard"/>
    <!-- 指定目标类实现的所有接口 -->
    <property name="interfaces" value="com.itheima.adapter.PunchCard"/>
    <!-- 指定切面 -->
    <property name="interceptorNames" >
        <list>
            <value>punchCardBefore</value>
        </list>
    </property>
</bean>
```

测试效果如下：

```
正在初始化打卡程序！
王五 打卡成功！
```

3.4.2 Spring AOP适配器体系



前置通知其实就是适配器模式之一，刚才我们编写的前置通知实现了接口`MethodBeforeAdvice`。Spring容器将每个具体的advice封装成对应的拦截器，返回给容器，这里对advice转换就需要用到适配器模式。我们来分析下适配器的实现：

如下代码实现了接口`BeforeAdvice`，而`BeforeAdvice`继承了`Advice`接口，在适配器接口`AdvisorAdapter`里面定义了方法拦截。

```
public interface MethodBeforeAdvice extends BeforeAdvice {

    void before(Method method, Object[] args, @Nullable Object target) throws Throwable;
}
```

`AdvisorAdapter`:定义了2个方法，分别是判断通知类型是否匹配，如果匹配就会获取对应的方法拦截。

```
public interface AdvisorAdapter {
    // 判断通知类型是否匹配
    boolean supportsAdvice(Advice advice);

    // 获取对应的拦截器
    MethodInterceptor getInterceptor(Advisor advisor);
}
```

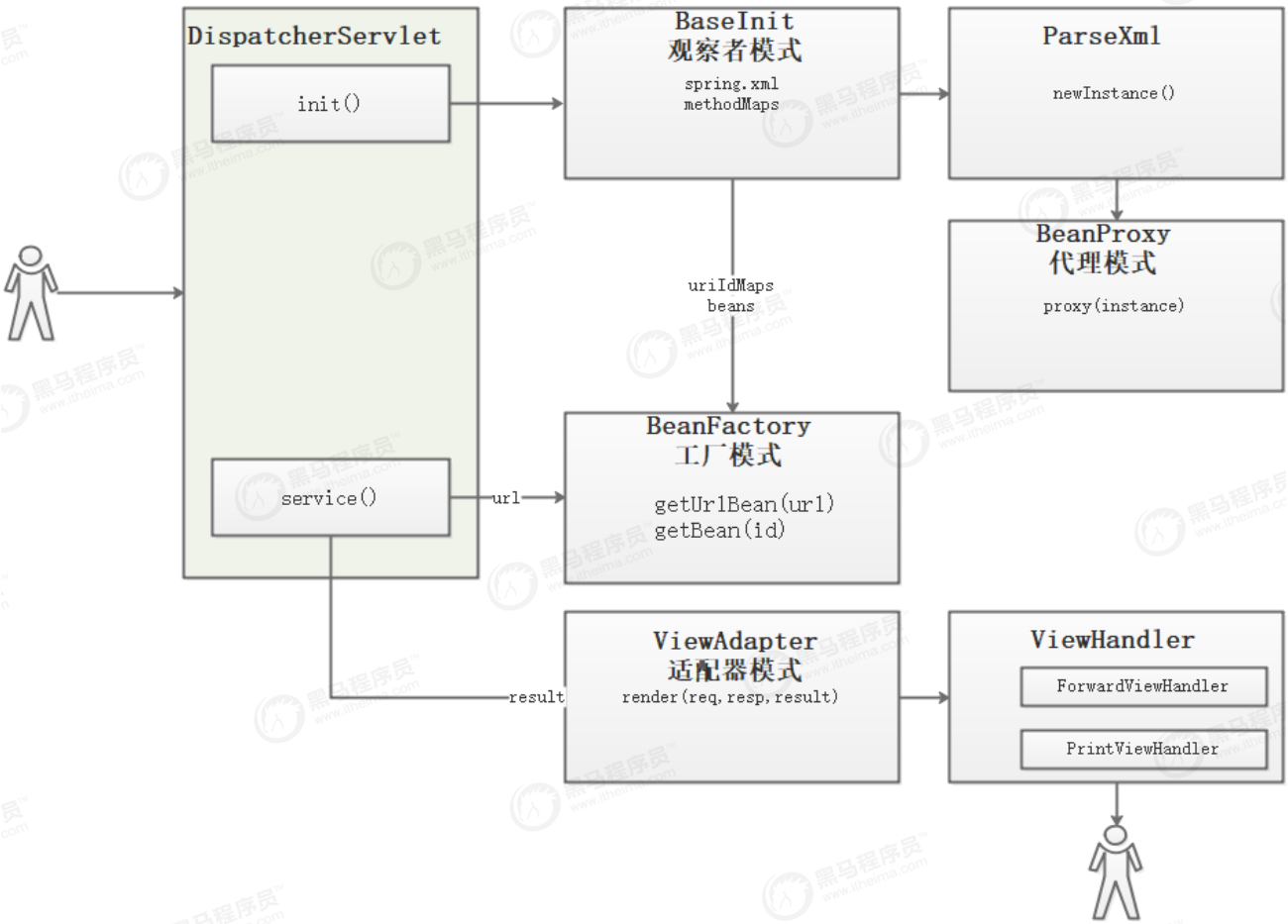
`MethodBeforeAdviceAdapter`：实现了`AdvisorAdapter`，代码如下：

```
class MethodBeforeAdviceAdapter implements AdvisorAdapter, Serializable {

    @Override
    public boolean supportsAdvice(Advice advice) {
        return (advice instanceof MethodBeforeAdvice);
    }

    @Override
    public MethodInterceptor getInterceptor(Advisor advisor) {
        MethodBeforeAdvice advice = (MethodBeforeAdvice) advisor.getAdvice();
        // 通知类型匹配对应的拦截器
        return new MethodBeforeAdviceInterceptor(advice);
    }
}
```

刚才我们在spring.xml中配置了代理类，代理类通过DefaultAdvisorAdapterRegistry类来注册相应的适配器，我们可以在

```
<!--代理配置-->
<bean id="proxyFactoryBean" class="org.springframework.aop.framework.ProxyFactoryBean">
    <!-- 指定目标对象 -->
    <property name="target" ref="punchCard"/>
    <!-- 指定目标类实现的所有接口 -->
    <property name="interfaces" value="com.itheima.adapter.PunchCard"/>
    <!-- 指定切面 -->
    <property name="interceptorNames" >
        <list>
            <value>punchCardBefore</value>
        </list>
    </property>
</bean>
```

4 架构中的设计模式



我们结合上面所学的设计模式，开发一款框架，该框架具备Spring的功能和SpringMVC功能，这里我们会提供部分工具，直接供大家使用。开发的框架流程图如上，整体基于Servlet实现。

准备工作：

搭建一个工程，引入相关依赖包，以及工具包，pom.xml代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.itheima</groupId>
    <artifactId>itheima-framework</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <properties>
        <java.version>1.8</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.1.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.commons</groupId>
            <artifactId>commons-lang3</artifactId>
            <version>3.0</version>
        </dependency>
        <!-- 解析 xml 的 dom4j -->
        <dependency>
            <groupId>dom4j</groupId>
            <artifactId>dom4j</artifactId>
            <version>1.6.1</version>
        </dependency>
        <!-- dom4j 的依赖包 jaxen -->
        <dependency>
            <groupId>jaxen</groupId>
            <artifactId>jaxen</artifactId>
            <version>1.1.6</version>
        </dependency>

        <!--fastJSON-->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>fastjson</artifactId>
            <version>1.2.51</version>
        </dependency>
    </dependencies>
</project>
```

扩展

```
@RequestMapping(value = "/account")
public class AccountController {

    private AccountService accountService;

    /***
     * 查询一条记录
     */
    @RequestMapping(value = "/one")
    public String one(){
        String result = accountService.one();
        return "/WEB-INF/pages/one.jsp";
    }
}
```

在 `AccountController` 类上有几个@RequestMapping注解，这个注解是我们自定义的，这个注解上有对应的值，我们可以通过 `ParseAnnotation` 工具类解析该注解，并将解析值存储到Map中，修改 `BaseInit`，代码如下：

```
public class BaseInit extends HttpServlet {

    //存储所有请求路径和对应的处理方法
    public static Map<String,Method> methods;

    /***
     * 做初始化工作
     * @param config
     * @throws ServletException
     */
    @Override
    public void init(ServletConfig config) throws ServletException {
        try {
            //1.解析所有controller里面拥有@RequestMapping注解的对象，并存储到Map<String,Method> methods
            methods = ParseAnnotation.parseRequestMapping();
            System.out.println(methods);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

解析所有@RequestMaping注解，将对应的路径和Method的关系存储到Map中

解析这个注解的作用，可以通过用于请求的路径来判断该路径归哪个对象处理；

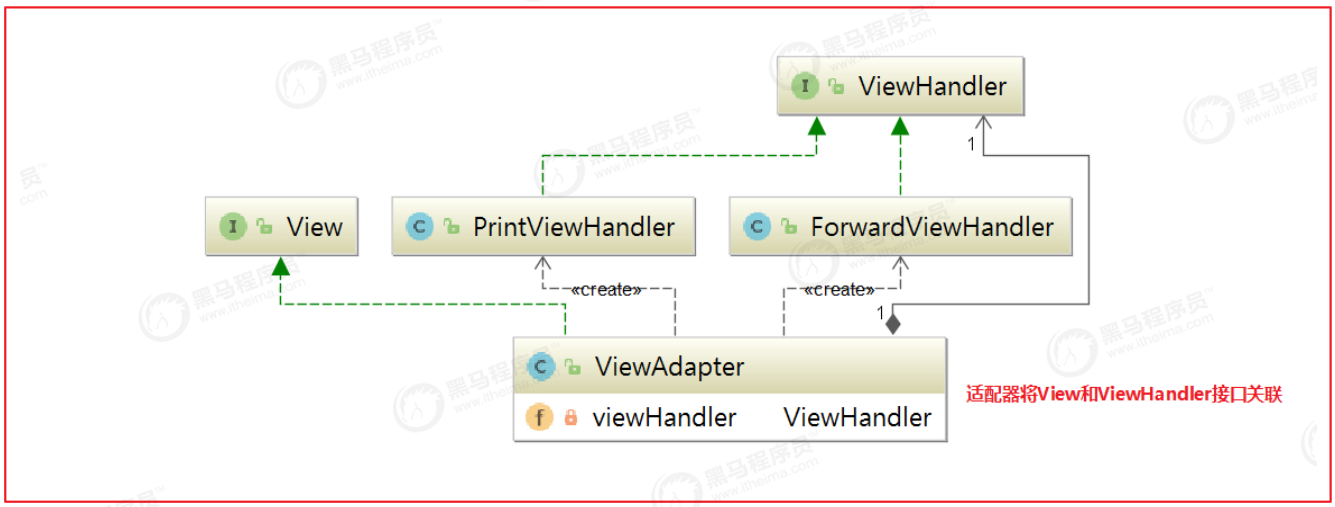
此时不要忘了配置web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
```

```
        version="3.1">
    <servlet>
        <servlet-name>itheima</servlet-name>
        <servlet-class>com.itheima.framework.DispacherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>itheima</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
</web-app>
```

4.1 自定义框架-适配器视图渲染

4.1.1 流程分析



在我们做视图解析的时候，有不同的解析方式，比如有直接输入json数据、重定向、转发、输出文件流等多种方式。通常采用哪种解析方式由执行方法的返回值决定，例如返回一个字符串，我们可以把返回的字符串当做响应的页面，这时候可以采用转发的方式，如果返回的是一个javabean，这时候我们可以采用输出json字符串的方式解析。像这一块的实现，我们可以采用适配器模式实现。

实现步骤如下：

```
1、定义一个视图解析接口ViewHandler，提供2种解析方式，分别为json输出和forward
2、为接口实现每种解析方式。分别创建PrintViewHandler和ForwardViewHandler
3、创建一个视图渲染接口View，View中提供渲染方法render
4、创建View的渲染实现ViewAdapter，通过提供的相应结果，来创建对应的试图解析器，并调用解析方式
```

4.1.2 适配器模式实现视图解析

1)创建视图解析器接口

```
public interface ViewHandler {
    //输出字符串
    default void print(HttpServletResponse response,Object result){}

    //forward转发
    default void forward(HttpServletRequest request,HttpServletResponse response,Object
result){}
}
```

2）创建json解析和转发

JSON解析对象：`PrintViewHandler`

```
public class PrintViewHandler implements ViewHandler {

    //输出渲染
    @Override
    public void print(HttpServletResponse response, Object result) {
        try {
            response.setContentType("application/json;charset=utf-8");
            PrintWriter writer = response.getWriter();
            writer.write(JSON.toJSONString(result));
            writer.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

转发解析对象：`ForwardViewHandler`

```
public class ForwardViewHandler implements ViewHandler {

    //转发渲染
    @Override
    public void forward(HttpServletRequest request, HttpServletResponse response, Object
result) {
        try {
            request.getRequestDispatcher(result.toString()).forward(request,response);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

3)视图渲染接口和实现

视图渲染接口：`View`

```
public interface View {
    /***
     * 视图渲染
     */
    void render(HttpServletRequest request, HttpServletResponse response, Object result);
}
```

视图渲染实现：`ViewAdapter`

```
public class ViewAdapter implements View {

    private ViewHandler viewHandler;

    /***
     * 渲染
     */
    @Override
    public void render(HttpServletRequest request, HttpServletResponse response, Object
result) {
        if(result instanceof String){
            viewHandler = new ForwardViewHandler();
            //forward
            viewHandler.forward(request,response,result);
        }else {
            viewHandler = new PrintViewHandler();
            //print
            viewHandler.print(response,result);
        }
    }
}
```

我们创建一个类`DispacherServlet`，继承BaseInit，同时在`web.xml`中把`BaseInit`换成`DispacherServlet`,并重写`service`方法，实现拦截所有用户请求，在`service`中使用执行反射调用，然后调用刚才写好的适配器查找对应的渲染方式执行渲染，代码如下：

```
public class DispacherServlet extends BaseInit {

    /****
     * 所有请求的入口
     */
    @Override
    protected void service(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        try {
```

```
            //获取uri
            String uri = request.getRequestURI();
            //调用
            Object result = invoke(uri);

            if(result!=null){
                //执行渲染
                View view = new ViewAdapter();
                view.render(request,response,result);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /***
     * 执行调用
     * @param uri
     * @return
     */
    public Object invoke(String uri) throws Exception{
        //从methods中获取请求方法
        Method method = methods.get(uri);
        if(method==null){
            return null;
        }
        //执行反射调用
        Class<?> clazz = method.getDeclaringClass();
        return method.invoke(clazz.newInstance());
    }
}
```

我们来测试一下，请求`<http://localhost:18081/account/info>`

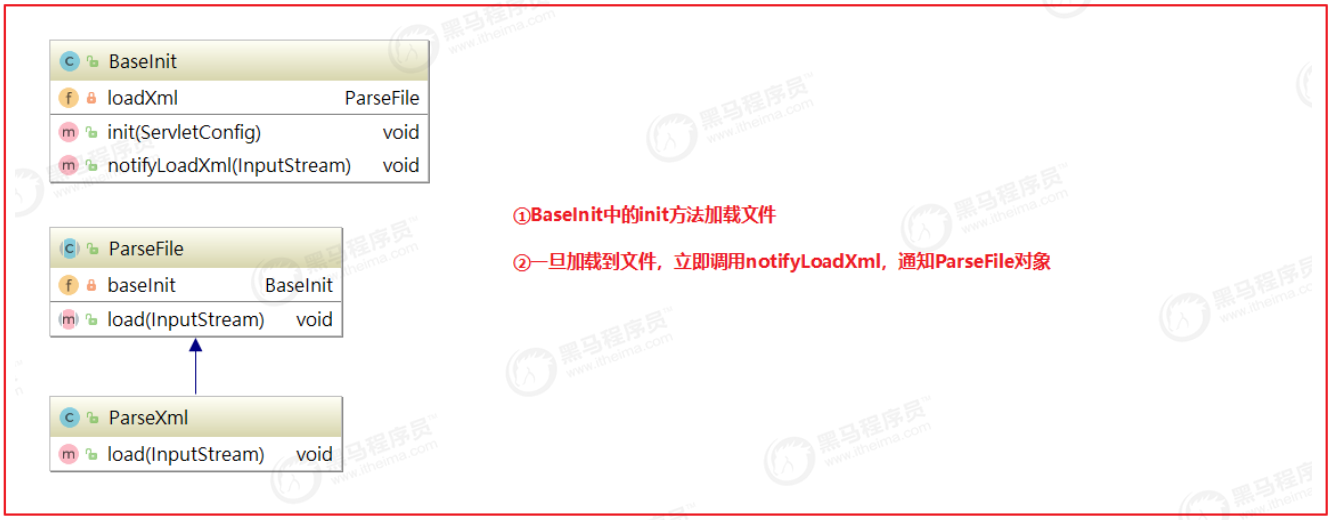
请求`<http://localhost:18081/account/one>`

one!!!

4.2 自定义框架-观察者模式

上面虽然已经实现了MVC模型对应功能，但是每次调用对象都是创建了新对象，我们可以对这里进行优化，让每次调用的对象是单利对象，这时候我们就需要初始化的时候把对象创建好了，但是对象和对象之间又存在依赖关系，我们可以在配置文件中配置这种关系。这里我们可以使用观察者模式和单利模式。

4.2.1 流程分析



我们编写一个类似Spring的MVC框架，这里采用观察者模式实现监听文件加载，实现步骤如下：

```
1、编写BaseInit类，并继承HttpServlet
2、重写HttpServlet中的init(ServletConfig config)方法
3、编写一个抽象类ParseFile，在该类中编写监听的对象baseInit，同时编写一个通知方法load(InputStream is)
4、编写ParseXml继承ParseFile，实现load(InputStream is)方法，当BaseInit中的init方法加载到变更文件时，该方法将得到变更通知。
```

4.2.2 监听文件加载并解析文件

要创建实例的对象信息以及依赖关系信息，我们可以配置到配置文件中，配置如下：

```
<beans>
    <!--id就是接口的名字，class实现类的全限定名-->
    <bean id="accountController" class="com.itheima.controller.AccountController">
        <property name="accountService" ref="accountService" />
    </bean>
    <bean id="accountService" class="com.itheima.service.impl.AccountServiceImpl">
        <property name="accountDao" ref="accountDao" />
    </bean>
    <bean id="accountDao" class="com.itheima.dao.impl.AccountDaoImpl"></bean>
</beans>
```

讲解：

```
1、配置文件中一个bean表示要创建一个对象的实例
2、id表示创建对象后的唯一id
3、class表示要创建哪个对象的全限定名
4、property表示给创建的对象指定属性赋值
5、property.name表示给指定对象的指定属性赋值
6、property.ref表示将指定对象赋值给property.name指定的属性
```

我们需要加载哪个配置文件进行解析，需要在web.xml中配置，代码如下：

```
<servlet>
    <servlet-name>itheima</servlet-name>
    <servlet-class>com.itheima.framework.DispacherServlet</servlet-class>
    <!--指定要解析的配置文件-->
    <init-param>
        <param-name>contextLocation</param-name>
        <param-value>spring.xml</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
```

当 `BaseInit` 获取到要解析的文件内容时，通知 `ParseFile` 解析对应的配置文件，在这里我们可以使用观察者模式，让当前 `BaseInit` 和 `ParseFile` 关联，这里 `ParseFile` 是一个抽象对象，给它编写一个实现 `ParseXml`，以后也许还会写 `ParseYaml` 等。这里用到的设计模式是观察者模式，观察到要解析配置文件，通知指定对象进行解析。

创建 `ParseFile`，代码如下：

```
public abstract class ParseFile {
    //监听的对象
    private BaseInit baseInit;

    //执行通知
    public abstract void load(InputStream is);
}
```

创建 `ParseXml`，代码如下：

```
public class ParseXml extends ParseFile {

    /***
     * 加载文件
     * @param is
     */
    @Override
    public void load(InputStream is){
        System.out.println("开始加载xml文件");
    }
}
```

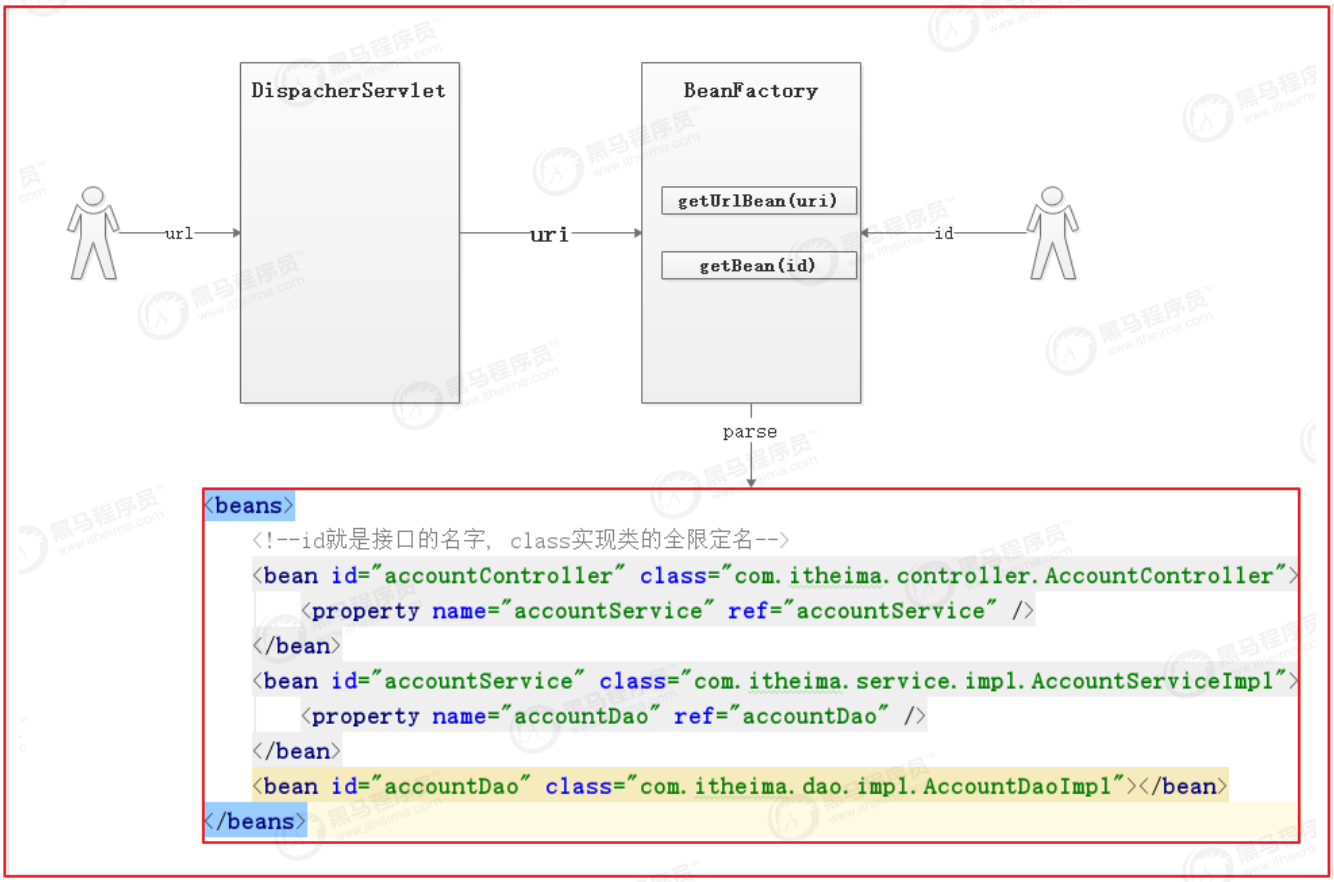
修改`BaseInit`，代码如下：

```
//需要通知的对象
private ParseFile parseFile = new ParseXml();

/***
 * 做初始化工作
 * @param config
 * @throws ServletException
 */
@Override
public void init(ServletConfig config) throws ServletException {
    try {
        //1.解析所有controller里面拥有@RequestMapping注解的对象，并存储到Map<String,Method> methods
        methods = ParseAnnotation.parseRequestMapping();

        //获取当前要解析的配置文件-观察者模式
        String cnf = config.getInitParameter( s: "contextLocation");   获取要解析的配置文件
        if(!StringUtils.isEmpty(cnf)){
            InputStream is = BaseInit.class.getClassLoader().getResourceAsStream(cnf);
            parseFile.load(is);     如果要解析配置文件，则通知parseFile进行文件解析
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

4.3 自定义框架-工厂模式

此时我们通知`ParseXml`加载需要解析的配置文件，这时候我们需要创建一个工厂对象来实现对象创建和对象获取，这块我们可以采用工厂模式实现。

4.3.1 流程分析



工厂的流程如上图，工厂需要先解析`spring.xml`，将解析的实体bean存储起来，用户每次获取对应的实例时，可以通过请求uri获取，也可以通过`spring.xml`中唯一的id获取。

4.3.2 工厂模式实现获取对象实例

1)创建工厂接口

创建工厂接口`BeanFactory`，因为以后可能会有注解实现方式，所以这里为工厂创建一个接口，代码如下：

```
public interface BeanFactory {

    /***
     * 根据ID获取对应的实例
     */
    Object getBean(String id) throws Exception;

    /***
     * 根据uri获取实例
     */
    Object getUrlBean(String url) throws Exception;
}
```

2)工厂实现

创建工厂实现类`XmlBeanFactory`,同时实现根据url和根据id获取对应实例方法，代码如下：

```
public class XmlBeanFactory implements BeanFactory {

    //存储了对应的实例对象
    //spring.xml bean id
    //spring.xml bean class->instance
    private static Map<String,Object> beans;

    public XmlBeanFactory() {
        initBeans();
    }

    /***
     * @param config:需要解析的文件
     */
    public XmlBeanFactory(String config) {
        try {
            //获取文件流
            InputStream is =
XmlBeanFactory.class.getClassLoader().getResourceAsStream(config);
            XmlBean.load(is);

            //实例化调用
            initBeans();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /***
     * 根据ID获取对应的实例
     * @param id
     * @return
     * @throws Exception
     */
    @Override
    public Object getBean(String id) throws Exception {
        return beans.get(id);
    }

    /***
     * 根据uri获取对应实例
     * @param url
     * @return
     */
    @Override
    public Object getUrlBean(String url) throws Exception {
        return null;
    }

    /***
     * 实例化Beans对象
     */
    private void initBeans(){
```

```
        try {
            //创建对应的实例
            beans = XmlBean.initBeans();
            //1.解析所有controller里面拥有@RequestMapping注解的对象，并存储到Map<String,Method>
 methods
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
 }
```

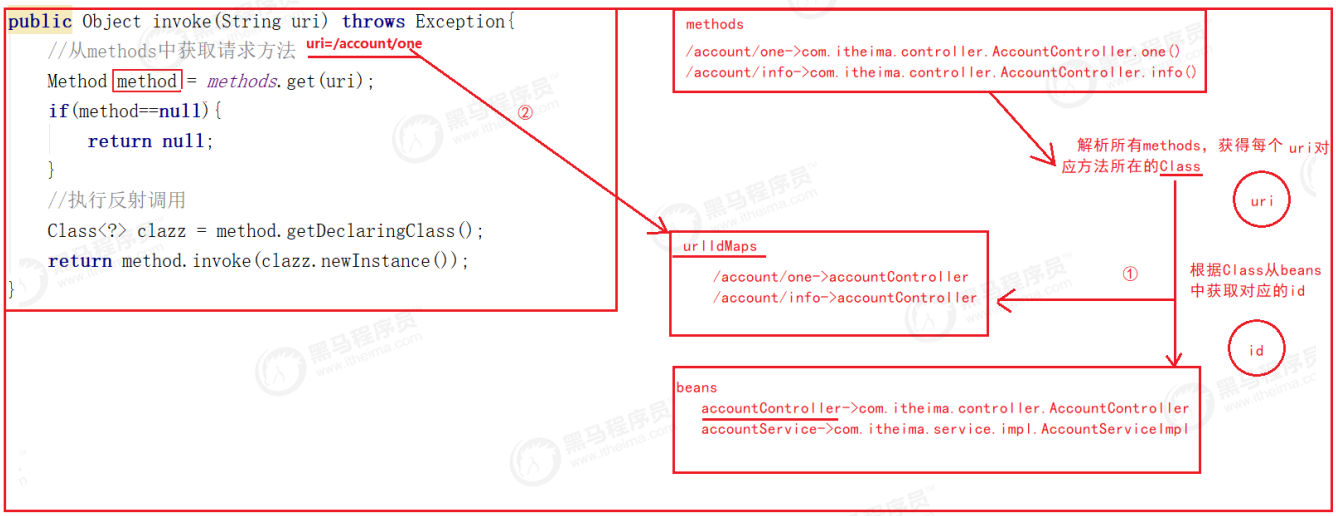
工厂测试：

```
public static void main(String[] args) throws Exception{
    BeanFactory beanFactory = new XmlBeanFactory("spring.xml");
    AccountService accountService = (AccountService) beanFactory.getBean("accountService");
    System.out.println(accountService);
}
```

输出结果 `com.itheima.service.impl.AccountServiceImpl@762efe5d`

此时容器生效

3)重写service反射调用



为了每次能使用单利对象处理用户请求从而更节省资源，我们需要根据用户的uri找到对应的实例。

我们可以按照这个步骤实现通过uri找到对应的实例：

```
1、循环所有methods，获取每个method对应的uri和method对应的Class
2、匹配beans中实例的Class和method的Class相同的对象，获取对应的key(也就是id)
3、把uri和id重新组合成一个新的Map，叫urlIdMaps
4、用户每次请求的时候，直接通过uri从urlIdMaps中获取对应实例
```

这个操作，该`ParseAnnotation.parseUrlMappingInstance(methods,beans)`方法已经帮我们实现了，所以我们只需要把实现的整体流程搬到`XmlBeanFactory`中就可以了。

修改`XmlBeanFactory`，代码如下：

```java
public class XmlBeanFactory implements BeanFactory {

    //存储了对应的实例对象
    //spring.xml bean id
    //spring.xml bean class->instance
    private static Map<String,Object> beans;

    //存储所有请求路径和对应的处理方法
    private static Map<String,Method> methods;

    //uri->id的映射关系
    private static Map<String,String> urlIdMaps;

    public XmlBeanFactory() {
        initBeans();
    }

    /**
     * @param config:需要解析的文件
     */
    public XmlBeanFactory(String config) {
        try {
            //获取文件流
            InputStream is = XmlBeanFactory.class.getClassLoader().getResourceAsStream(config);
            XmlBean.load(is);

            //实例化调用
            initBeans();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * 根据ID获取对应的实例
     * @param id
     * @return
     * @throws Exception
     */
    @Override
    public Object getBean(String id) throws Exception {
        return beans.get(id);
    }

    /**
     * 根据uri获取对应实例
     * @param url
     * @return
```

```
     */
    @Override
    public Object getUrlBean(String url) throws Exception {
        //根据uri获取id
        String id = urlIdMaps.get(url);
        //根据id获取唯一值
        if(StringUtils.isNotEmpty(id)){
            return getBean(id);
        }
        return null;
    }

    /***
     * 实例化Beans对象
     */
    private void initBeans(){
        try {
            //创建对应的实例
            beans = XmlBean.initBeans();
            //1.解析所有controller里面拥有@RequestMapping注解的对象，并存储到Map<String,Method> me
            methods = ParseAnnotation.parseRequestMapping();
            //解析uri和id的映射关系
            urlIdMaps=ParseAnnotation.parseUrlMappingInstance(methods,beans);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /***
     * 获取所有映射
     * @return
     */
    public static Map<String, Method> getMethods() {
        return methods;
    }
}
```

上图代码如下:

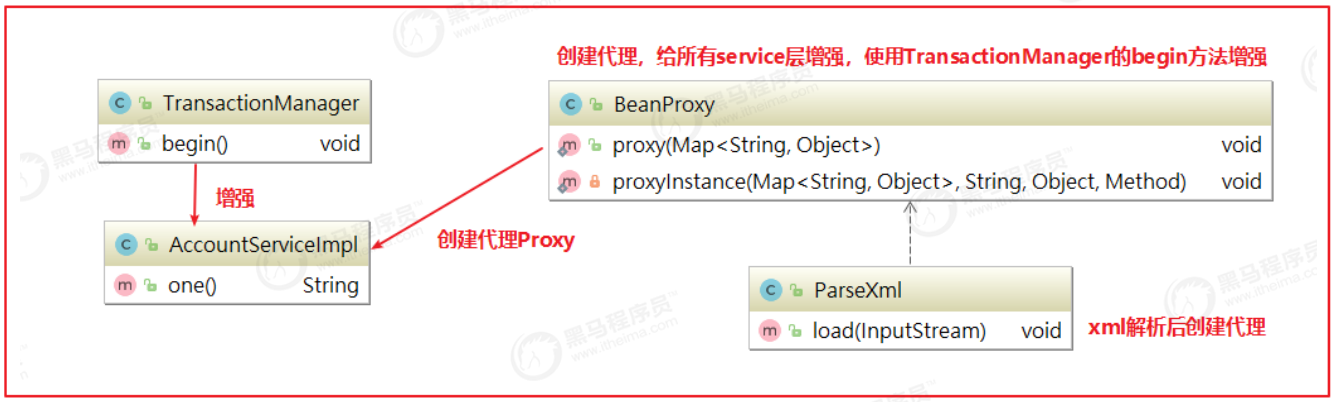
```
public class XmlBeanFactory implements BeanFactory {

    //存储了对应的实例对象
    //spring.xml bean id
    //spring.xml bean class->instance
    private static Map<String,Object> beans;

    //存储所有请求路径和对应的处理方法
    private static Map<String,Method> methods;
```

```
    //uri->id的映射关系
    private static Map<String,String> urlIdMaps;

    public XmlBeanFactory() {
        initBeans();
    }

    /***
     * @param config:需要解析的文件
     */
    public XmlBeanFactory(String config) {
        try {
            //获取文件流
            InputStream is =
XmlBeanFactory.class.getClassLoader().getResourceAsStream(config);
            XmlBean.load(is);

            //实例化调用
            initBeans();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /***
     * 根据ID获取对应的实例
     * @param id
     * @return
     * @throws Exception
     */
    @Override
    public Object getBean(String id) throws Exception {
        return beans.get(id);
    }

    /***
     * 根据uri获取对应实例
     * @param url
     * @return
     */
    @Override
    public Object getUrlBean(String url) throws Exception {
        //根据uri获取id
        String id = urlIdMaps.get(url);
        //根据id获取唯一值
        if(StringUtils.isNotEmpty(id)){
            return getBean(id);
        }
        return null;
    }

    /***
```

```
     * 实例化Beans对象
     */
    private void initBeans(){
        try {
            //创建对应的实例
            beans = XmlBean.initBeans();
            //1.解析所有controller里面拥有@RequestMapping注解的对象，并存储到Map<String,Method>
methods
            methods = ParseAnnotation.parseRequestMapping();
            //解析uri和id的映射关系
            urlIdMaps=ParseAnnotation.parseUrlMappingInstance(methods,beans);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /***
     * 获取所有映射
     * @return
     */
    public static Map<String, Method> getMethods() {
        return methods;
    }
}
```

4.4 自定义框架-代理模式增强

4.4.1 流程分析



我们模拟Spring的声明式事务控制，在业务层进行增强，我们可以按照如下步骤实现：

1、创建增强类TransactionManager，在里面编写一个增强方法begin
2、创建一个BeanProxy对象，用于给指定包下的对象创建代理
3、每次ParseXml加载解析之后，调用BeanProxy给指定包下的对象创建代理

4.4.2 业务层代理模式增强

1)指定增强位置

我们首先在spring.xml配置文件中配置一下增强的位置,before表示前置增强，`package`表示指定包下的对象进行前置增强，`ref`表示指定增强的类，`method`表示增强的类中指定的方法。

```
<!--增强类-->
<bean id="transactionManager" class="com.itheima.framework.aop.TransactionManager"></bean>
<!--
    package:给该包下所有类的所有方法产生代理
-->
<before package="com.itheima.service.impl" ref="transactionManager" method="begin" />
```

我们需要创建一个增强类`TransactionManager`：

```
public class TransactionManager {

    //前置增强
    public void begin(){
        System.out.println("开启事务! ");
    }
}
```

2)增强实现

我们接着需要对增强进行实现，我们可以按照如下步骤实现：

```
1、解析xml，需要获取增强的包，实现增强的类的方法
2、获取所有解析的实例，并判断每个实例是否是该包下的对象
3、如果是该包下的对象，给它创建代理，执行增强
4、将当前对象替换成代理对象
5、将当前对象下的引用属性替换成代理
```

我们创建该代理增强类`BeforeProxyBean`，代码如下：

```
public class BeforeProxyBean implements InvocationHandler {

    //被代理的对象
    private Object instance;

    //增强的实例
    private Object beforeInstance;

    //增强的方法
    private Method beforeMethod;

    //构造函数
```

```
    public BeforeProxyBean(Object instance, Object beforeInstance, Method beforeMethod) {
        this.instance = instance;
        this.beforeInstance = beforeInstance;
        this.beforeMethod = beforeMethod;
    }

    //增强过程
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        //前置增强
        beforeMethod.invoke(beforeInstance);
        //目标方法调用
        return method.invoke(instance);
    }
}
```

创建代理类`ProxyBeanFactory`,代码如下:

```
public class ProxyBeanFactory {

    /***
     * 增强操作
     */
    public static void proxy(Map<String,Object> beans){
        try {
            //获取对应前置增强节点信息
            Map<String, String> map = XmlBean.before();
            if(map==null){
                return;
            }
            //获取指定的包
            String packageinfo= map.get("package");
            //获取增强类实例
            Object aopInstance = beans.get(map.get("ref"));
            //增强方法
            Method method = aopInstance.getClass().getDeclaredMethod(map.get("method"));
            //增强操作
            beforeHandler(beans,aopInstance,method,packageinfo);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /***
     * 执行增强
     */
    private static void beforeHandler(Map<String,Object> beans,Object aopInstance,Method
method,String packageinfo) throws Exception {
        //循环增强
        for (Map.Entry<String, Object> entry : beans.entrySet()) {
            //当前实例的包
```

```
            String entryPackage = entry.getValue().getClass().getPackage().getName();
            //属于增强包下的类，需要增强
            if(entryPackage.equals(packageinfo)){
                //创建代理
                Object proxy = Proxy.newProxyInstance(
                        entry.getValue().getClass().getClassLoader(),
                        entry.getValue().getClass().getInterfaces(),
                        new BeforeProxyBean(entry.getValue(), aopInstance, method)
                );
                //将非代理对象替换掉
                beans.put(entry.getKey(),proxy);
                //将实例对象中对应属性替换成代理对象
                XmlBean.replaceProxy(beans,entry.getKey(),proxy);
            }
        }
    }
}
```

在工厂`XmlBeanFactory`实现调用即可：

```
private void initBeans(){
    try {
        //创建对应的实例
        beans = XmlBean.initBeans();
        //增强操作
        ProxyBeanFactory.proxy(beans);
        //1.解析所有controller里面拥有@RequestMapping注解的对象，并存储到Map<String,Method> methods
        methods = ParseAnnotation.parseRequestMapping();
        //解析uri和id的映射关系
        urlIdMaps=ParseAnnotation.parseUrlMappingInstance(methods,beans);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

运行测试结果如下：

```
开启事务!
AccountServiceImpl.one()方法执行
Dao查询!
```