结构型模式

1. 类和对象按照某种布局组成更大的结构。

类结构型模式：采用继承机制。

对象结构型模式：采用对象的组合和聚合。

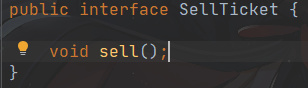
1. 代理模式

需要给某对象提供一个代理以控制对该对象的访问。

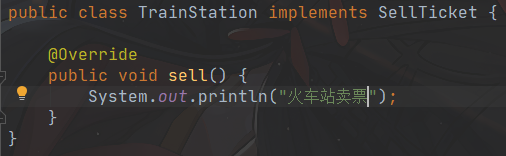
根据代理类生成时机不同：静态代理（编译期生成），动态代理。

组成：

1. 抽象主题类：真实主题和代理对象实现的业务方法。



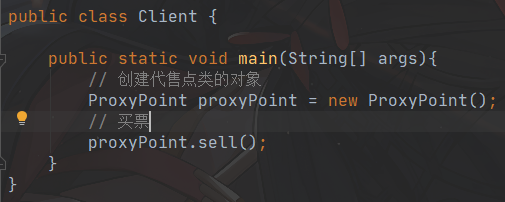
1. 真实主题类：具体业务。



1. 代理类：对真实主题的引用。可以访问、控制或扩展真实主题的功能（在这里是输出）。



1. 访问类：访问代理对象而非目标对象。



优点：

1. 在客户端和目标对象之间起中介和保护目标对象的作用；
2. 代理对象可以扩展目标对象的功能；
3. 将客户端和目标对象分离，降低耦合度。

缺点：增加了系统的复杂度。

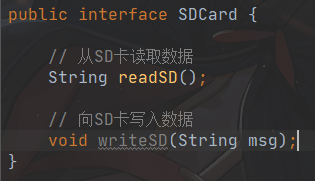
适用场景：

1. 远程代理。
2. 防火墙代理（VPN）。
3. 保护代理，给不同的用户提供不同级别的使用权限。
4. 适配器模式

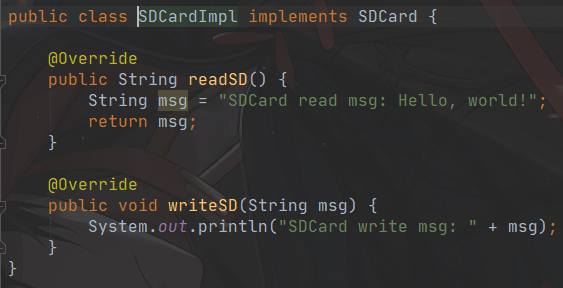
将一个类的接口转换成客户希望的另一个接口，使得原本因为接口不兼容不能一起工作的类能一起工作。

①适配器类模式组成：

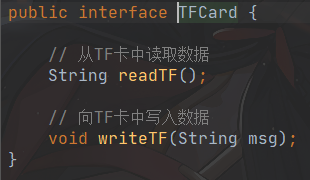
1. 目标接口：当前系统业务希望的接口；



目标接口的具体实现：



1. 适配者接口：现存组件库中的组件接口；

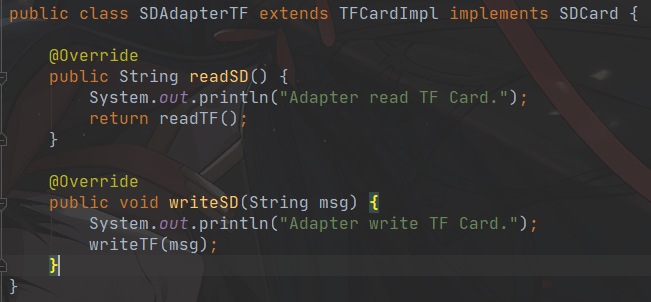


具体实现：

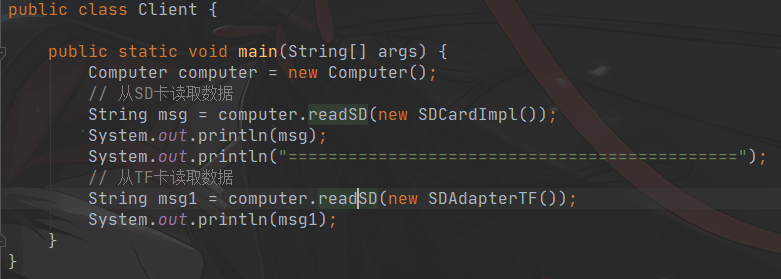


1. 适配器类：转换器。

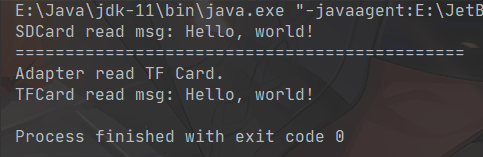
继承现有组件的接口，实现目标接口。



1. 客户类



输出结果：

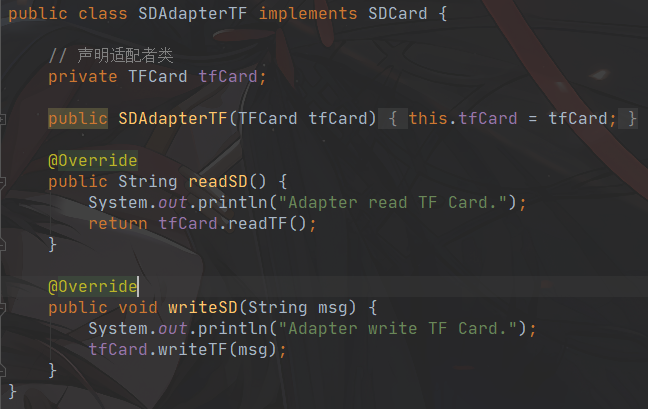


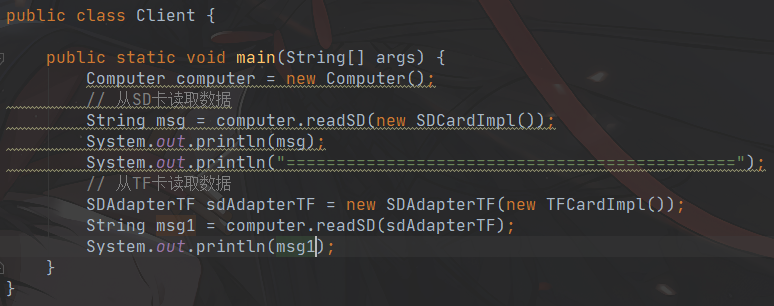
要求系统必须拥有一个目标接口，提供给适配器类实现。

不满足合成复用原则：尽可能使用对象组合而不是继承来达成复用的目的。

②对象适配器模式组成

适配器组合现有的组件而非继承。





满足合成复用原则。

当不存在目标接口时，也可以直接继承实现类。

③应用场景

1. 原有系统的接口和新系统的接口不一致；
2. 使用第三方提供的组件，但是组件的接口定义和我们系统要求的接口不同。
3. 装饰者模式

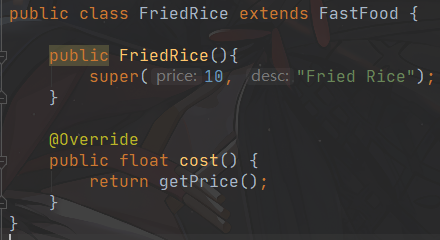
在不改变现有对象结构的情况下，动态地给该对象增加一些职责（功能）的模式。

①组成

1. 抽象构件：抽象接口



1. 具体构件：实现抽象构件，通过装饰角色增加一些职责



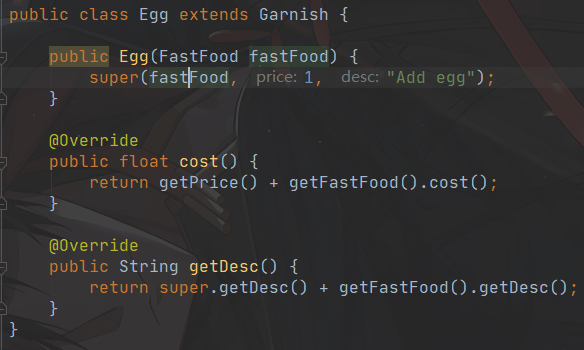
1. 抽象装饰：继承或者实现抽象构件角色，包含具体构件的实例，通过其子类扩展具体构件的功能。



继承抽象构件角色：在添加新的职责时，可以使用抽象构件的方法。

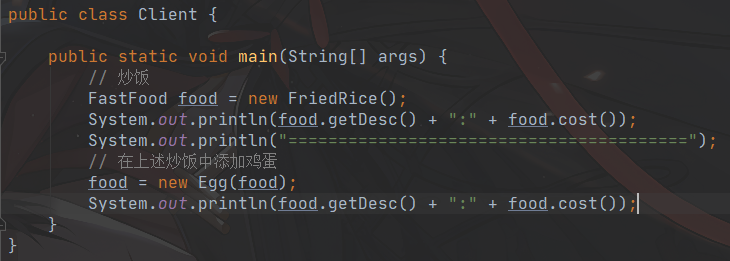
包含具体构件的实例：可以接受具体构件的对象作为参数，在其上叠加。

1. 具体装饰



这里，把鸡蛋作为一个FastFood类的新对象，因此，通过FastFood类的getPrice方法可以获取其价格（不能用cost因为在FastFood中cost没有实现）；传入的参数本身是一个FastFood类的对象（而且应该是FriedRice或FriedNoddles中的一种），通过父类的getFastFood方法可以获取，再用cost方法求其价格。

1. 客户类



直接传递food作为参数，在其上叠加鸡蛋或培根。

②优点：

1. 比继承更灵活的扩展，是动态的扩展方式；
2. 装饰者和被装饰者可以独立的发展。

③适用场景

1. 不适合使用继承的场景，例如系统中存在大量独立的扩展，类的定义不能被继承等；
2. 以动态透明的方式给单个对象添加职责；
3. 对象的功能要求可以动态地添加或动态地撤销。

④装饰者模式和静态代理

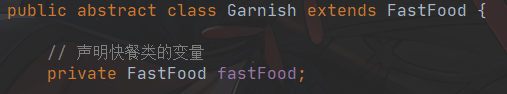
相同点：

1. 都需要实现目标类的接口；
2. 都需要组合目标类的对象；
3. 都可以增强目标类的对象。

不同点：

1. 目的不同：装饰者模式的目的在于增强对象，静态代理模式的目的在于保护和隐藏对象；
2. 组合目标类的对象的方法不同：静态代理模式在代理类中新建目标类的对象，装饰者模式不是。





1. 桥接

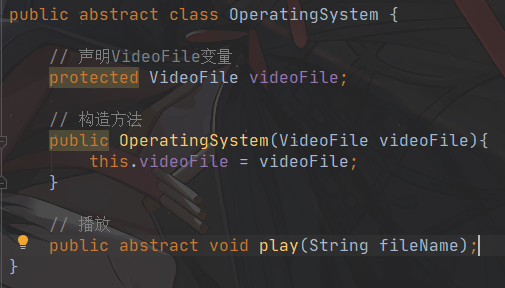
①定义

在一个有多种可能会变化的维度的系统中，将抽象和实现分离。

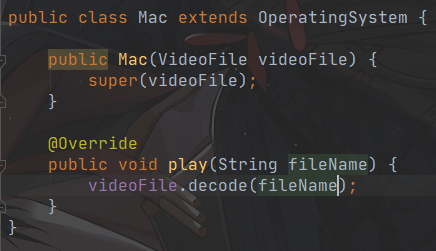
例如绘制一个图形，图形的类型和颜色都是可以变化的维度。

②组成

1. 抽象化角色：抽象类，包含一个对实现化对象的引用；



1. 扩展抽象化角色：实现抽象化角色的业务方法，通过组合关系调用实现化角色中的业务方法；



1. 实现化角色：实现化角色的接口，供扩展抽象化角色使用；



1. 具体实现化角色：给出接口的具体实现。



1. 客户端



客户端使用扩展抽象化角色，扩展抽象化角色基于具体实现化角色提供的方法。

③优点

1. 提高了系统的扩展性；
2. 实现细节对客户透明。

④适用场景

1. 当一个类存在两个独立变化的维度，且这两个维度都需要进行扩展；
2. 系统不希望使用继承导致类爆炸；
3. 桥接模式可以在构件的抽象化角色和具体化角色间建立联系。
4. 外观

①定义

为多个复杂的子系统提供一个一致的接口，外部应用程序不必关系内部子系统的具体细节。

②组成

1. 外观角色：为多个子系统提供一个共同的接口；



1. 子系统角色：实现系统的部分功能。



③优点

1. 降低子系统和客户端之间的耦合度；
2. 对客户屏蔽了子系统组件。

④缺点

不符合开闭原则。

⑤使用场景

1. 分层系统构建，使用外观模式定义每层子系统的入口点；
2. 复杂系统的子系统很多；
3. 提高子系统的独立性和可移植性。
4. 组合

①定义

把一组相似的对象当作一个单一的对象，创建对象组的树形结构。

②组成

1. 抽象根节点：定义各层次对象共有的方法和属性；



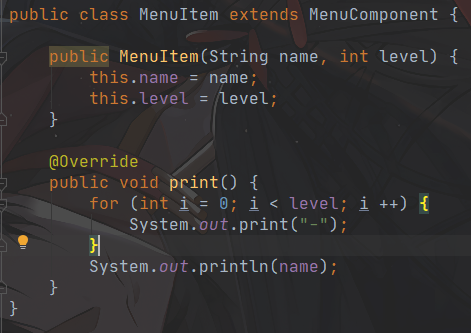
1. 树枝节点：定义行为，存储子节点，组合其它树枝节点和叶子节点形成一个树形结构；



打印菜单的名称，并且递归的打印子菜单和子菜单项。

1. 叶子节点：系统遍历的最小单位。

只覆盖和自己有关的操作。



1. 客户端



输出：



③组合模式的分类

1. 透明组合模式（标准实现方式）：在抽象根节点中声明所有用于管理成员对象的方法。例如在上例中的add，remove等方法。缺点是需要针对树枝节点和叶子节点的不同定义异常处理（throw new UnsupportedOperationException()）。
2. 安全组合模式：在抽象根节点中不声明任何用于管理成员对象的方法。缺点是必须有区别地对待叶子节点和树枝节点。

④优点

1. 定义分层次的复杂对象；
2. 客户端可以一致地使用一个组合结构或单个对象；
3. 符合开闭原则；
4. 可以形成复杂的树枝结构，但对树形结构的控制非常简单。

⑤适用场景

出现树形结构的位置。

1. 享元

①定义

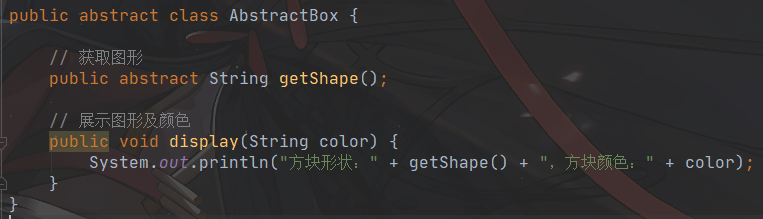
运用共享技术对大量细粒度对象的复用，大幅度减少需要创建的对象个数。

②状态

1. 内部状态：不会随着环境的改变而改变的可共享部分；
2. 外部状态：随着环境的改变而改变的不可共享的部分。

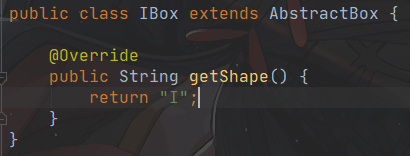
③组成

1. 抽象享元角色：接口或抽象类，声明了具体享元类公共的方法，可以向外界提供享元的内部状态或者设置享元的外部状态。

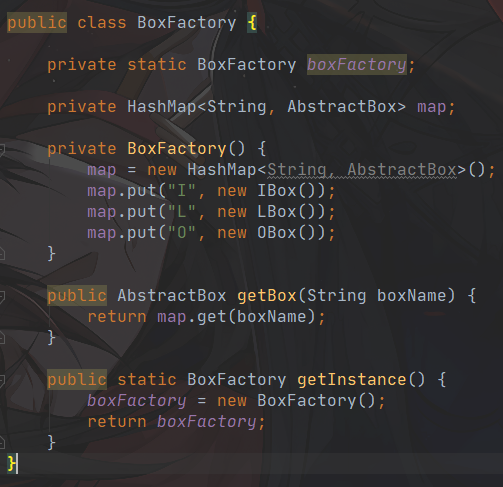


在这里，内部状态是图形的形状，外部状态是图形的颜色。

1. 具体享元角色：继承抽象享元角色。

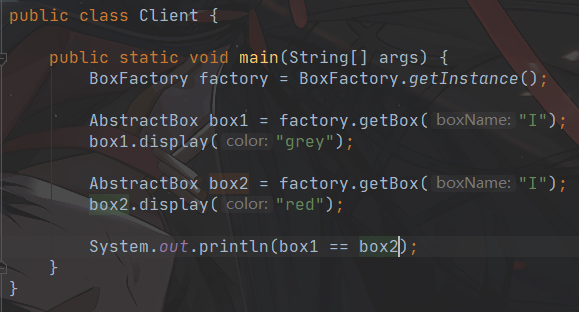


1. 非享元角色：继承抽象享元角色，但是不能被共享。
2. 享元工厂：负责创建和管理享元角色。

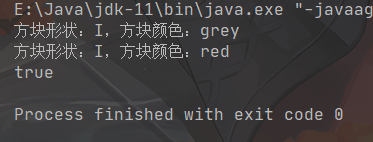


在这里，享元工厂使用了单例模式（懒汉式）。

1. 客户类



输出：



两次产生的方块对象都是从同一个哈希表里面取出的，只是修改了颜色。

④优点

1. 极大减少内存中相似或相同的对象数量，节省内存；
2. 享元模式中外部模式相对独立，不影响内部状态。

⑤缺点

享元对象的部分状态必须外部化，分离内部状态和外部状态，使程序逻辑更加复杂。

⑥使用场景

1. 一个系统中有大量相同或相似的对象；
2. 对象的大部分状态可以外部化；
3. 需要多次使用享元对象（定义享元池需要耗费系统资源）。