码农求职小助手：设计模式高频面试题

笔记本：	9-设计模式		
创建时间：	2019/9/7 21:47	**更新时间：**	2019/9/7 21:47
作者：	pc941206@163.com		

更多资料请关注微信公众号：**码农求职小助手**

对于设计模式，知道每种设计模式的主要思想，然后挑几个重要的了解其具体使用细节。

一 创建型模式

1、简单工厂模式【重点】

1.1 思想

简单工厂模式（Simple Factory Pattern）：又称为静态工厂方法模式。它属于类创建型模式。在简单工厂模式中，可以根据参数的不同返回不同的实例。简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类。

简单工厂模式的要点在于：当你需要什么，只需要传入一个正确的参数，就可以获取你所需要的对象，而无须知道其创建细节。

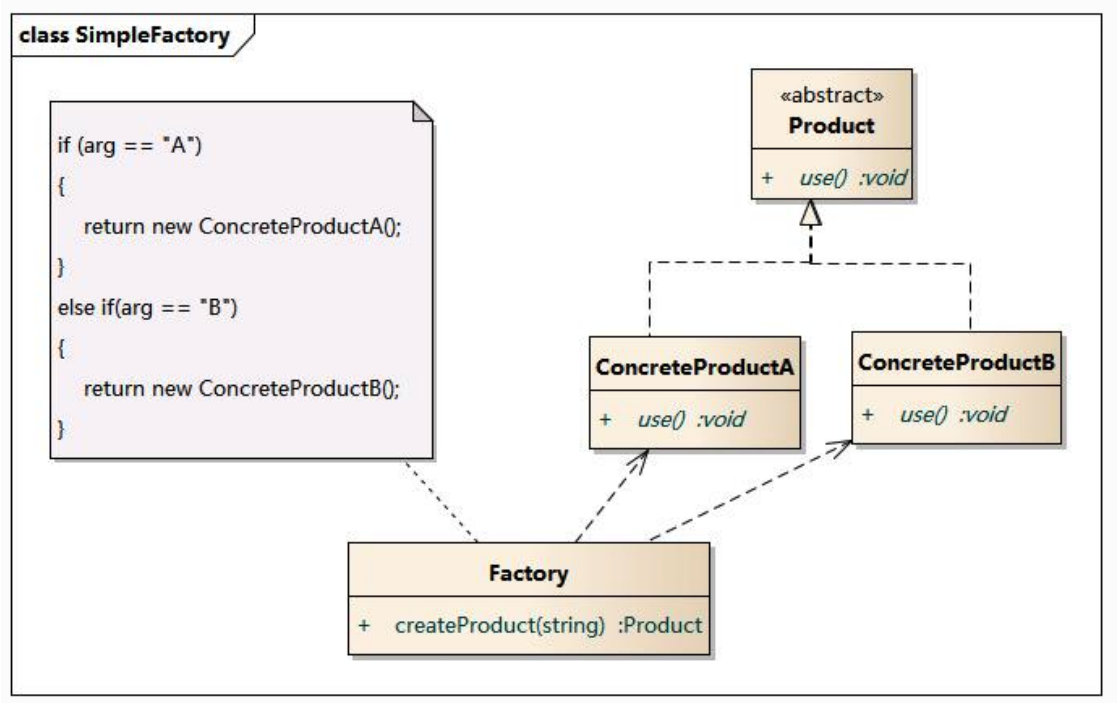
1.2 模式结构

简单工厂模式包括三个角色：

1、Factory（工厂角色）：负责实现所有实例的内部逻辑；

2、Product（抽象产品角色）：是所创建的所对象的父类，负责描述所有实例共有的公共接口；

3、ConcreteProduct（具体产品角色）：创建目标，所有创建的对象都充当这个角色的某个具体类的实例。



1.3 优缺点

优点：

在于实现对象的创建和对象的使用分离，将对象的创建交给专门的工厂类负责。

缺点：

在于工厂类不够灵活，增加新的具体产品需要修改工厂类的判断逻辑代码，而且产品较多时，工厂方法代码将会非常复杂。

1.4 模式应用

1、JDK 类库中广泛使用了简单工厂模式，如工具类 java.text.DateFormat，它用于格式化一个本地日期或者时间。

```
public final static DateFormat getDateInstance();

public final static DateFormat getDateInstance(int style);

public final static DateFormat getDateInstance(int style,Locale
locale);
```

2、Java 加密技术：获取不同加密算法的密钥生成器:

```
KeyGenerator keyGen = KeyGenerator.getInstance("DESede");
```

创建密码器:

```
Cipher cp = Cipher.getInstance("DESede");
```

2、工厂方法模式【重点】

2.1 思想

工厂方法模式（Factory Method Pattern）：又称为工厂模式或者多态工厂模式。在工厂方法模式中，工厂父类负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象，这样做的目的是将产品的实例化操作延迟到工厂子类中完成，即通过工厂子类来确定究竟应该实例化哪一个具体产品类。

2.2 模式结构

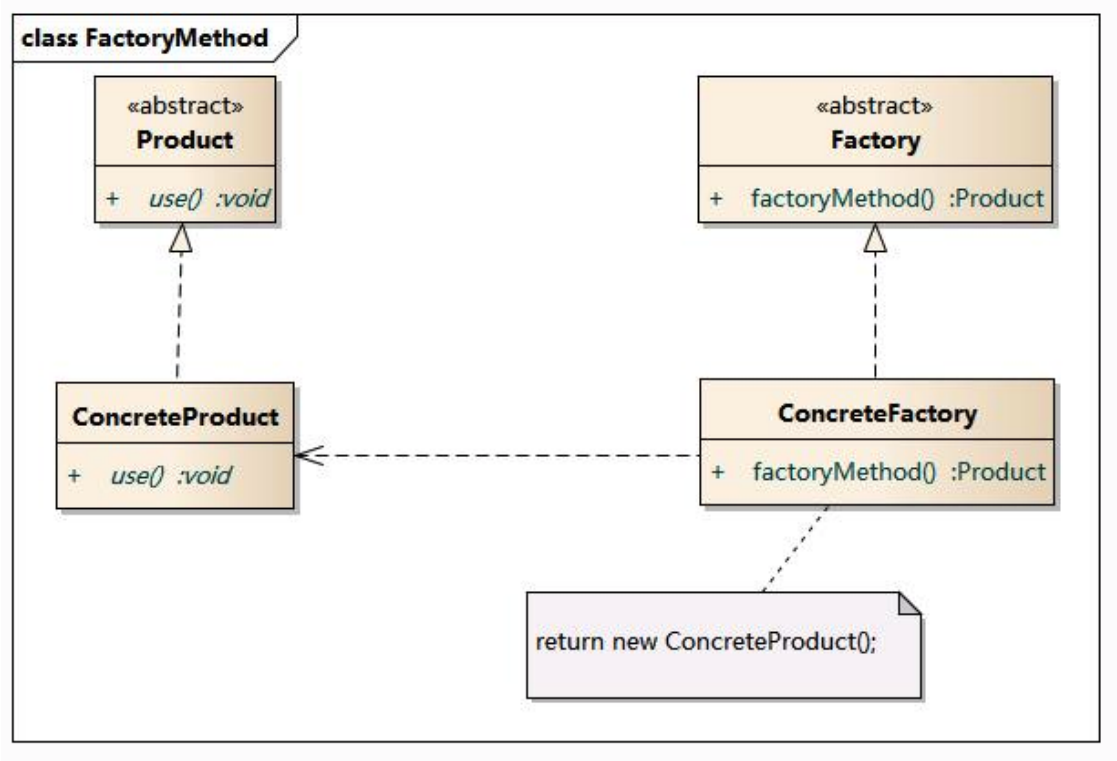
工厂方法模式包含如下四个角色:

1、Product：抽象产品

2、ConcreteProduct：具体产品

3、Factory：抽象工厂

4、ConcreteFactory：具体工厂



2.3 优缺点

工厂方法模式是简单工厂模式的进一步抽象和推广。由于使用了面向对象的多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。**在工厂方法模式中，核心的工厂类不再负责所有产品的创建，而是将具体创建工作交给子类去做。** 这个核心类仅仅负责给出具体工厂必须实现的接口，而不负责产品类被实例化这种细节，这使得工厂方法模式可以允许系统在不修改工厂角色的情况下引进新产品。

优点:

增加新的产品类时无须修改现有系统，并封装了产品对象的创建细节，系统具有良好的灵活性和可扩展性。

缺点:

在于增加新产品的同时需要增加新的工厂，导致系统类的个数成对增加，在一定程度上增加了系统的复杂性。

2.4 模式应用

- **JDBC中的工厂方法:**

```
Connection conn = DriverManager.getConnection("jdbc:microsoft:sqlserver://loc
alhost:1433;

DatabaseName = DB;user=sa;password="123");
```

```
Statement statement = conn.createStatement();

ResultSet rs = statement.executeQuery("select * from UserInfo");
```

3、抽象工厂模式【重点】

3.1 思想

在工厂方法模式中具体工厂负责生产具体的产品，每一个具体工厂对应一种具体产品，工厂方法也具有唯一性，一般情况下，一个具体工厂中只有一个工厂方法或者一组重载的工厂方法。但是有时候我们需要一个工厂可以提供多个产品对象，而不是单一的产品对象。

为了更清晰地理解工厂方法模式，需要先引入两个概念：

1、产品等级结构：产品等级结构即产品的继承结构，如一个抽象类是电视机，其子类有海尔电视机、海信电视机、TCL电视机，则抽象电视机与具体品牌的电视机之间构成了一个产品等级结构，抽象电视机是父类，而具体品牌的电视机是其子类。

2、产品族：在抽象工厂模式中，产品族是指由同一个工厂生产的，位于不同产品等级结构中的一组产品，如海尔电器工厂生产的海尔电视机、海尔电冰箱，海尔电视机位于电视机产品等级结构中，海尔电冰箱位于电冰箱产品等级结构中。

当系统所提供的工厂所需生产的具体产品并不是一个简单的对象，而是多个位于不同产品等级结构中属于不同类型的具体产品时需要使用抽象工厂模式。

思想：提供一个创建一系列相关或依赖对象的接口，而无须指定它们具体的类。抽象工厂模式又称为 Kit 模式。

3.2 模式结构

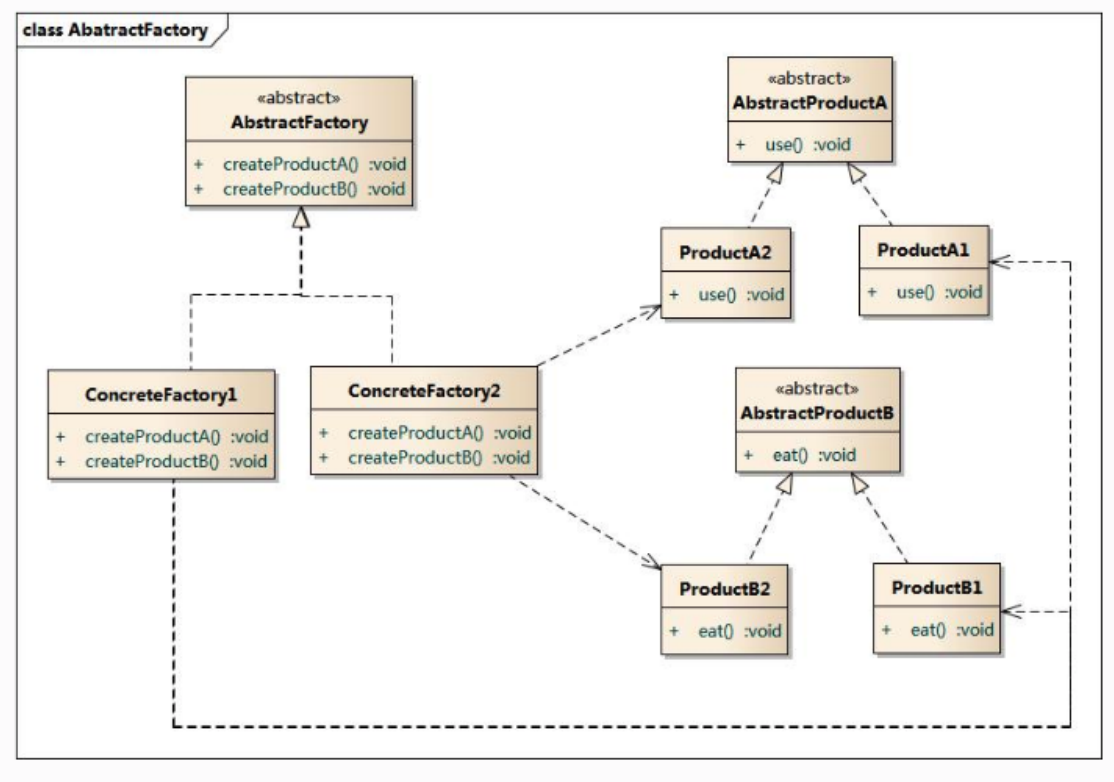
抽象工厂模式包含如下四个角色：

1、AbstractFactory：抽象工厂

2、ConcreteFactory：具体工厂

3、AbstractProduct：抽象产品

4、Product：具体产品



3.3 优缺点

抽象工厂模式是所有形式的工厂模式中最为抽象和最具一般性的一种形态。抽象工厂模式与工厂方法模式最大的区别在于：工厂方法模式针对的是一个产品等级结构，而抽象工厂模式则需要面对多个产品等级结构。

优点：

是隔离了具体类的生成，使得客户并不需要知道什么被创建，而且每次可以通过具体工厂类创建一个产品族中的多个对象，增加或者替换产品族比较方便，增加新的具体工厂和产品族很方便。

缺点：

在于增加新的产品等级结构很复杂，需要修改抽象工厂和所有的具体工厂类，对“开闭原则”的支持呈现倾斜性。

3.4 模式应用

在很多软件系统中需要更换界面主题，要求界面中的按钮、文本框、背景色等一起发生改变时，可以使用抽象工厂模式进行设计。

4、建造者模式

4.1 思想

无论是在现实世界中还是在软件系统中，都存在一些复杂的对象，它们拥有多个组成部分，如：汽车，它包括车轮、方向盘、发送机等各种部件。而对于大多数用户而言，无须知道这些部件的装配细节，也几乎不会使用单独某个部件，而是使用一辆完整的汽车，可以通过建造者模式对其进行设计与描述，建造者模式可以将部件和其组装过程分开，一步一步创建一个复杂的对象。用户只需要指定复杂对象的类型就可以得到该对象，而无须知道其内部的具体构造细节。

在软件开发中，也存在大量类似汽车一样的复杂对象，它们拥有一系列成员属性，这些成员属性中有些是引用类型的成员对象。而且在这些复杂对象中，还可能存在一些限制条件，如某些属性没有赋值则复杂对象不能作为一个完整的产品使用；有些属性的赋值必须按照某个顺序，一个属性没有赋值之前，另一个属性可能无法赋值等。

建造者模式(Builder Pattern)：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

建造者模式是一步一步创建一个复杂的对象，它允许用户只通过指定复杂对象的类型和内容就可以构建它们，用户不需要知道内部的具体构建细节。建造者模式属于对象创建型模式。根据中文翻译的不同，建造者模式又可以称为生成器模式。

4.2 模式结构

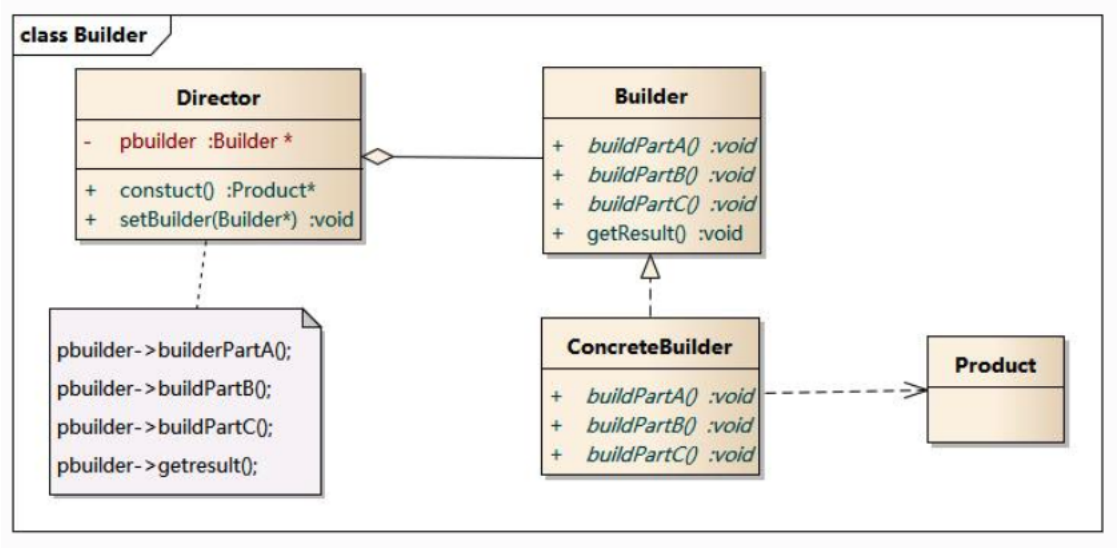
建造者模式包含如下四个角色：

1、Builder：抽象建造者：为创建一个产品对象的各个部件指定抽象接口；

2、ConcreteBuilder：具体建造者：实现了抽象建造者接口，实现各个部件的构造和装配方法，定义并明确它所创建的复杂对象，也可以提供一个方法返回创建好的复杂产品对象；

3、Director：指挥者：负责安排复杂对象的建造次序，指挥者与抽象建造者之间存在关联关系，可以在其 construct() 建造方法中调用建造者对象的部件构造与装配方法，完成复杂对象的建造；

4、Product：产品角色：被构建的复杂对象，包含多个组成部件。



4.3 优缺点

优点：

客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象，每一个具体建造者都相对独立，而与其他的具体建造者无关，因此可以很方便地替换具体建造者或增加新的具体建造者，符合“开闭原则”，还可以更加精细地控制产品的创建过程。

缺点：

由于建造者模式所创建的产品一般具有较多的共同点，其组成部分相似，因此其使用范围受到一定的限制，如果产品的内部变化复杂，可能会导致需要定义很多具体建造者类来实现这种变化，导致系统变得很庞大。

4.4 模式应用

在很多游戏软件中，地图包括天空、地面、背景等组成部分，人物角色包括人体、服装、装备等组成部分，可以使用建造者模式对其进行设计，通过不同的具体建造者创建不同类型的地图或人物。

5、单例模式【重点】

5.1 思想

单例模式(Singleton Pattern)：单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它提供全局访问的方法。

单例模式的要点有三个：

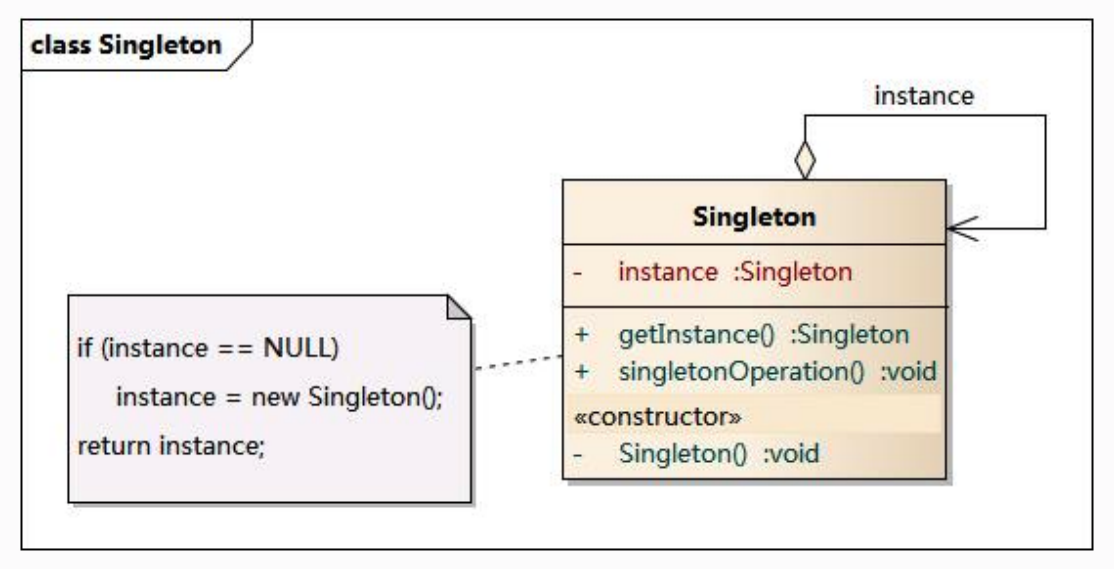
一是：某个类只能有一个实例；

二是：它必须自行创建这个实例；

三是：它必须自行向整个系统提供这个实例。

单例模式是一种对象创建型模式。单例模式又名单件模式或单态模式。

单例模式只包含一个单例角色：在单例类的内部实现只生成一个实例，同时它提供一个静态的工厂方法，让客户可以使用它的唯一实例；为了防止在外部对其实例化，将其构造函数设计为私有。



5.2 优缺点

优点：

提供了对唯一实例的受控访问并可以节约系统资源；

缺点：

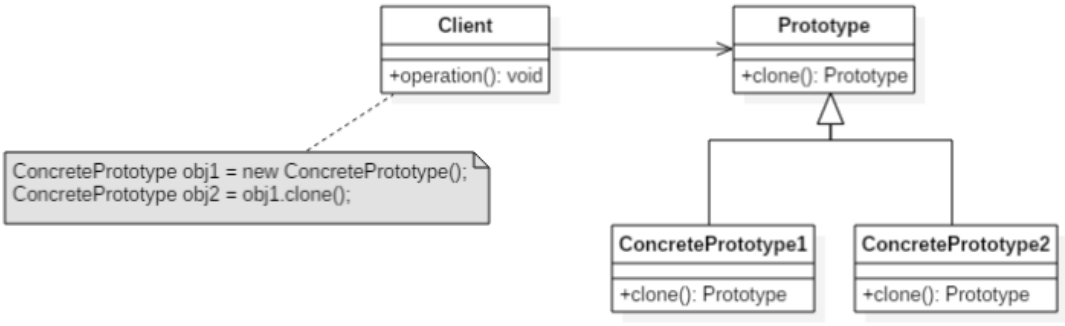
因为缺少抽象层而难以扩展，且单例类职责过重。

5.3 模式应用

一个具有自动编号主键的表可以有多个用户同时使用，但数据库中只能有一个地方分配下一个主键编号，否则会出现主键重复，因此该主键编号生成器必须具备唯一性，可以通过单例模式来实现。

6、原型模式

使用原型实例指定要创建对象的类型，通过复制这个原型来创建新对象。

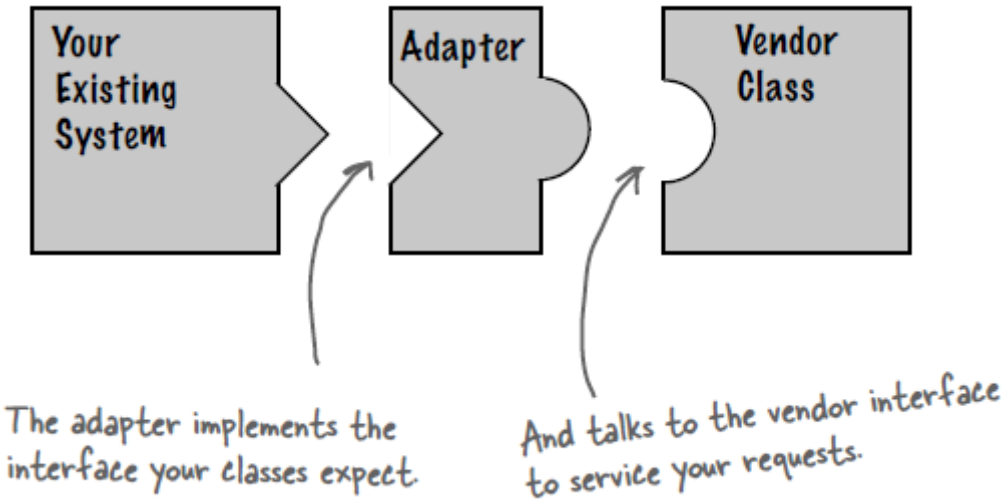


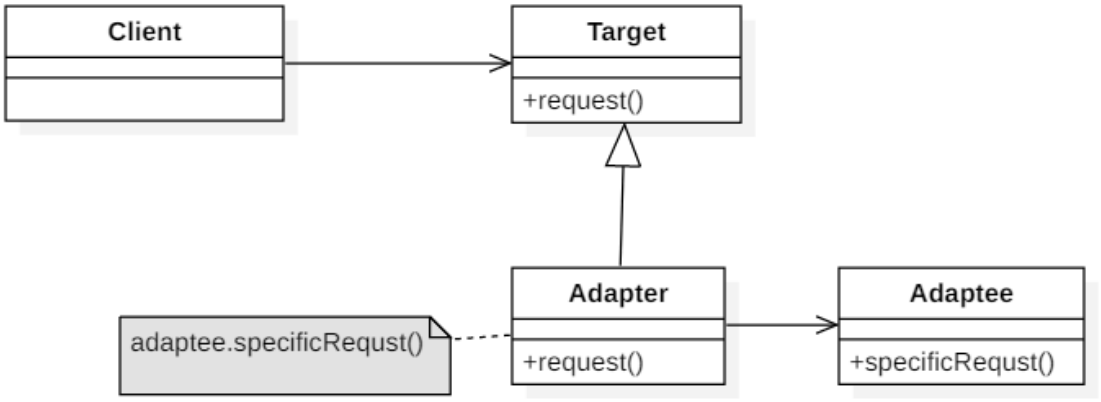
- **JDK：Object.clone() 方法**

二 结构型模式

1、适配器模式

把一个接口转换成另外一个用户需要的接口。





- **JDK 源码：**

1、java.util.Arrays.asLlist()

2、java.util.Collection.list()

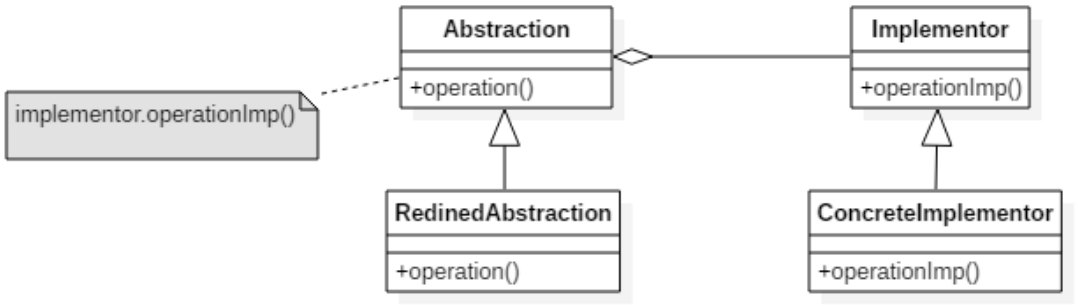
3、java.util.Collections.enumeration()

4、javax.xml.bind.annotation.adapters.XMLAdapter

2、桥接模式

将抽象与实现分离开，使它们可以独立变化。

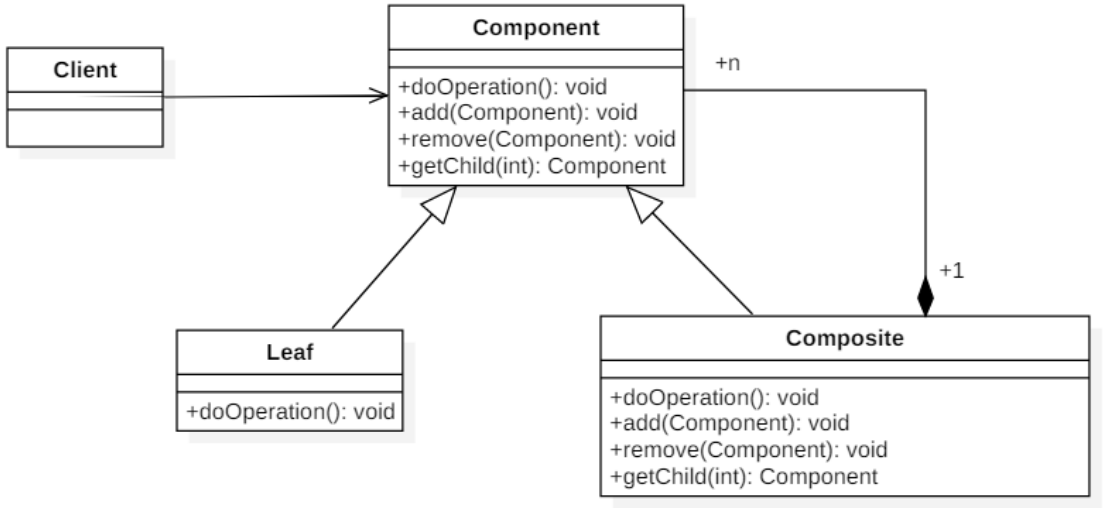
- Abstraction：定义抽象类的接口
- Implementor：定义实现类接口



3、组合模式

将对象组合成树形结构来表示“整体/部分”层次关系，允许用户以相同的方式处理单独对象和组合对象。

> 组件（Component）类是组合类（Composite）和叶子类（Leaf）的父类，可以把组合类看成是树的中间节点。
>
> 组合对象拥有一个或者多个组件对象，因此组合对象的操作可以委托给组件对象去处理，而组件对象可以是另一个组合对象或者叶子对象。



- **JDK 源码出处：**

1、java.util.Map.putAll(Map)

2、java.util.List.addAll(Collection)

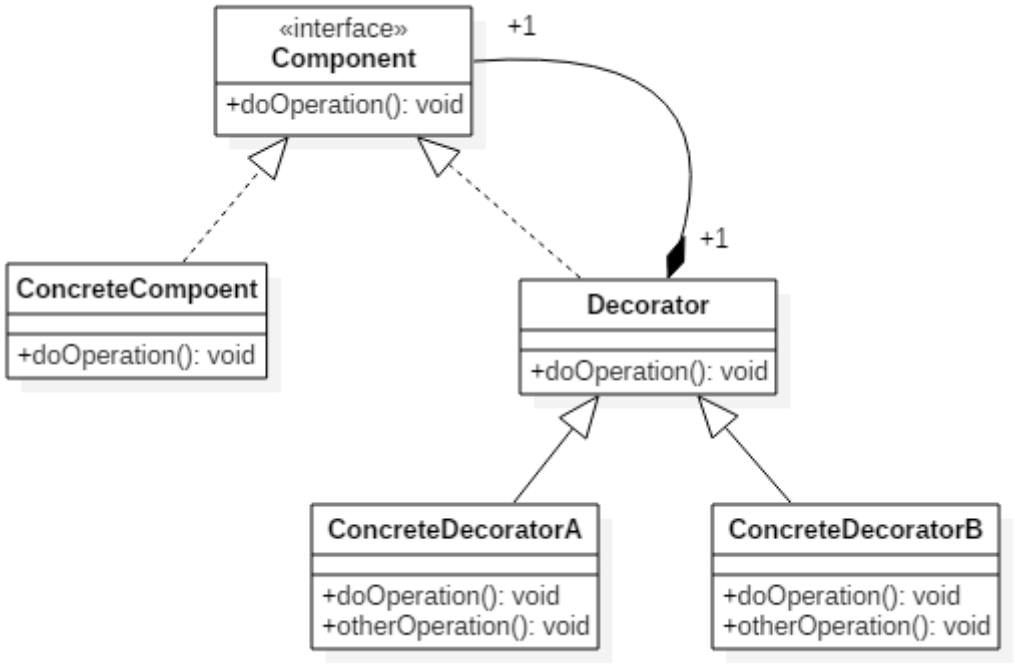
3、java.util.Set.addAll(Collection)

4、装饰模式【重点】

目的：为对象动态添加功能。

装饰者（Decorator）和具体组件（ConcreteComponent）都继承自组件（Component），具体组件的方法实现不需要依赖于其它对象，而装饰者组合了一个组

件，这样它可以装饰其它装饰者或者具体组件。**所谓装饰，就是把这个装饰者套在被装饰者之上，从而动态扩展被装饰者的功能。装饰者的方法有一部分是自己的，这属于它的功能，然后调用被装饰者的方法实现，从而也保留了被装饰者的功能。**可以看到，具体组件应当是装饰层次的最低层，因为只有具体组件的方法实现不需要依赖于其它对象。



- JDK 源码出处：

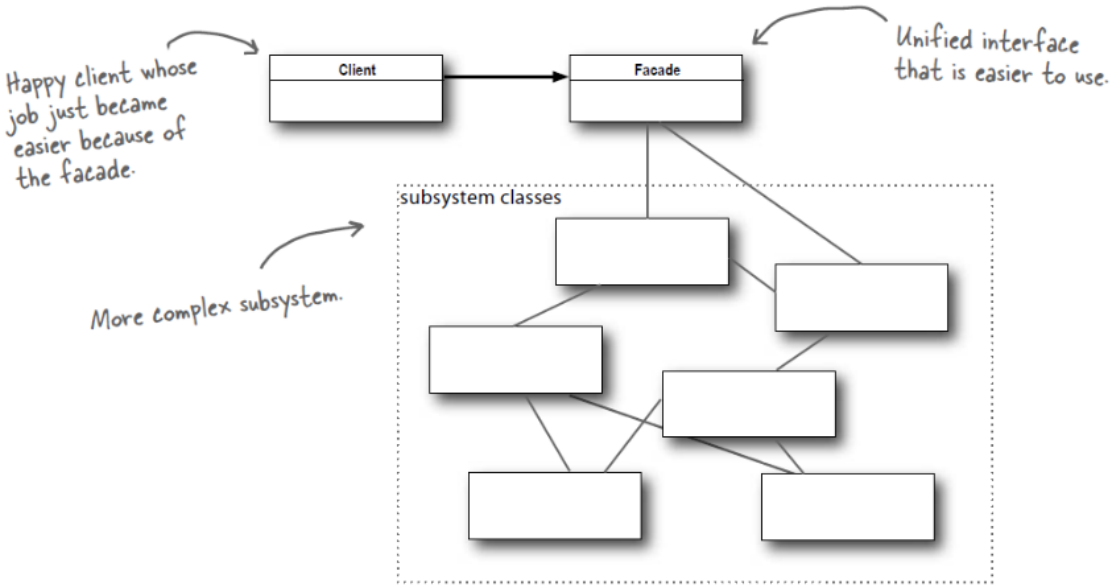
1、java.io.BufferedInputStream(InputStream)

2、java.io.DataInputStream(InputStream)

3、java.io.BufferedOutputStream(OutputStream)

5、外观模式

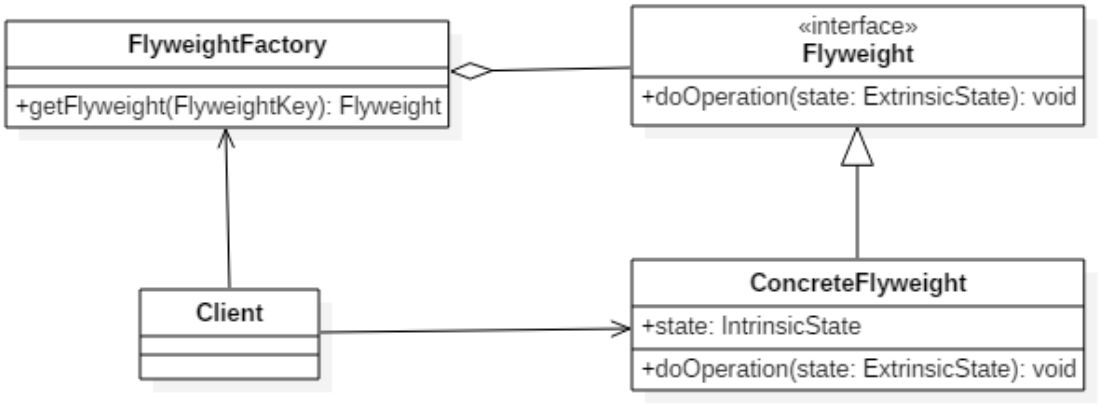
目的：提供了一个统一的接口，用来访问子系统中的一群接口，从而让子系统更容易使用。



6、享元模式

目的：利用共享的方式来支持大量细粒度的对象，这些对象一部分内部状态是相同的。

> **Flyweight**：享元对象
>
> **IntrinsicState**：内部状态，享元对象共享内部状态
>
> **ExtrinsicState**：外部状态，每个享元对象的外部状态不同



- **JDK 源码出处：**

1、java.lang.Integer.valueOf(int)

2、java.lang.Boolean.valueOf(boolan)

3、java.long.Byte.valueOf(byte)

4、java.lang.Character.valueOf(char)

7、代理模式【重要】

- **目的：控制对其它对象的访问。**

四 Java 集合类中的设计模式

2.1 迭代器模式

Collection 继承了 Iterable 接口，其中的 iterator() 方法能够产生一个 Iterator 对象，通过这个对象就可以迭代遍历 Collection 中的元素。从 JDK 1.5 之后可以使用 foreach 方法来遍历实现了 Iterable 接口的聚合对象。

```
List<String> list = new ArrayList<>();
list.add("a");
list.add("b");
for (String item : list) {
        System.out.println(item);
}
```

2.2 适配器模式

java.util.Arrays#asList() 可以把数组类型转换为 List 类型。

```
@SafeVarargs
public static <T> List<T> asList(T... a)
```

应该注意的是 asList() 的参数为泛型的变长参数，不能使用基本类型数组作为参数，只能使用相应的包装类型数组。

```
Integer[ ] arr = {1, 2, 3};
List list = Arrays.asList(arr);
```

也可以使用以下方式调用 asList()：

```
List list = Arrays.asList(1, 2, 3);
```

五 Spring 中的设计模式