设计模式&Spring场景

模式归类

创建型模式

定义：创建型模式是用来创建对象的模式，抽象了实例化的过程，帮助一个系统独立于其关联对象的创建、组合和表示方式。

创建型模式的作用：

■ 将系统所使用的具体类的信息封装起来；

■ 隐藏类的实例是如何被创建和组织的。外界对于这些对象只知道它们共同的接口，而不清楚其具体的实现细节。

■ 封装创建逻辑，不仅仅是new一个对象那么简单；

■ 封装创建逻辑变化，客户代码尽量不修改，或尽量少修改。

简述创建型模式（Creational Pattern）是对类的实例化过程的抽象化，能够提供对象的创建和管理职责。创建型模式共有5种：

■ 单例模式；

■ 工厂方法模式；

■ 抽象工厂模式；

■ 建造者模式；

■ 原型模式。

结构型模式

定义：结构型模式讨论的是类和对象的结构，它采用继承机制来组合接口或实现（类结构型模式），或者通过组合一些对象实现新的功能（对象结构型模式）

简述结构型模式（Structural Pattern）描述如何将类或者对象结合在一起形成更大的结构。结构型模式的目的是通过组合类或对象产生更大结构以适应更高层次的逻辑需求，包括以下 7种模式：

■ 代理模式

■ 装饰模式

■ 适配器模式

■ 组合模式

■ 桥梁模式

■ 外观模式

■ 享元模式

行为型模式

定义：行为型设计模式关注的是对象的行为，用来解决对象之间的联系问题。

简述行为型模式（Behavioral Pattern）是对不同的对象之间划分责任和算法的抽象化。行为型模式包括以下11个模式：

■ 模板方法模式

■ 命令模式

■ 责任链模式

■ 策略模式

■ 迭代器模式

■ 中介者模式

■ 观察者模式

■ 备忘录模式

■ 访问者模式

■ 状态模式

■ 解释器模式

创建型模式

单例设计模式

> 最简单的一种创建型设计模式，提供了创建对象的最佳方式，在此模式中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。这个类提供了一种访问其唯一的对象的方式，可以直接访问，不需要实例化该类的对象。

单例模式论述

	Singleton Pattern
别名	单例模式
描述	单例是自己创建自己的唯一实例，且给其他对象提供这一实例
设计模式	创建型
实现方式	懒汉模式 | 恶汉模式

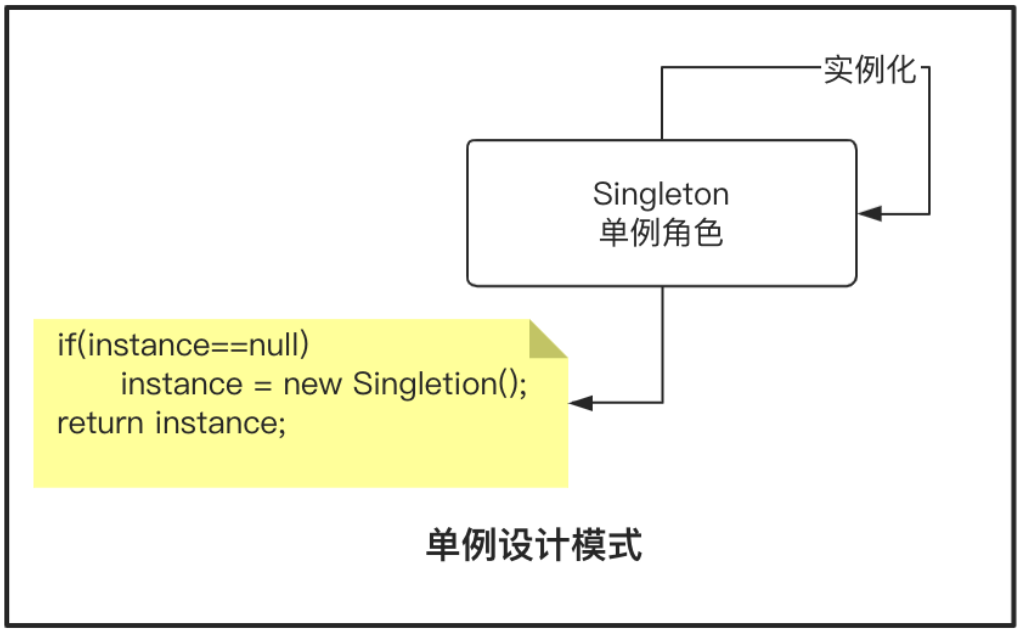
- 我们为什么需要单例模式
  - 节省内存空间：单例在内存中只有一个实例对象，节省内存空间，避免了大量创建及销毁
  - 高性能：减少高重复资源重复占用，可以进行全部访问

单例模式理论

- 单例模式

> 简单模式整体我们用的相比后两者是比较少的，因为本身它违背来开闭原则，虽然可以通过反射+配置文件解决，但总体还是不太友好。

- 什么时候用单例模式?
    1. 重复对象需要频繁实例化及销毁的对象
    2. 有状态化的工具的对象
    3. 频繁访问数据库或文件的对象
- 如何看待单例模式的角色?



单例角色：单例模式只有一个单例角色，在单例内部生成一个实例，同时提供了一个静态工厂的方法，让客户可以访问他的唯一实例，为防止外部访问，将其构造参数设置为私有并对外提供一个静态对象用于外部共享的唯一实例。

单例模式实践

- 饿汉单例模式

:) 饿汉模式是实现单例的最简单的模式，

```
/**
 * 饿汉单例模式
 */
public class EhSingleton {
    // 定义静态变量的同时，实例化单例类，所以类加载的时候已经创建了单例对象
    private static final EhSingleton instance = new EhSingleton();
    private EhSingleton() { }

    public static EhSingleton getInstance() {
        return instance;
    }
}
```

优点：

- 典型的空间换时间，节省了运行时间，简单方便
- 饿汉单例模式是线程安全的，不存在线程安全问题，因为虚拟机仅仅会装载一次，所有类加载不会产生并发

缺点：

- 如果在不需要创建对象的时候就装载的话，就会造成资源浪费

- 懒汉单例模式

  :) 懒汉单例模式是在第一次调用getInstance()进行单例对象的实例化，这种实现被称为懒加载或者延迟加载，但懒汉单例模式本身是线程不安全的，因为多个线程遇到并发会同时调用getInstance()，就会导致创建多个实例对象，所以我们需要通过以下几步方式解决线程不安全，代码如下：

```
// 第一步（存在问题）
public class LazySingleton {
    private static LazySingleton instance = null;

    private LazySingleton() { }
        // 通过实现synchronized关键词同步（我们知道，synchronized要在距离资源最近的地方进行同步，所以我们改下代码到第二种）
    public synchronized static LazySingleton getInstance() {
        if (instance == null) {
            instance = new LazySingleton();
        }
        return instance;
    }
}

// 第二步（解决第一步的疑问，代码如下）
public class LazySingleton {
    private static LazySingleton instance = null;

    private LazySingleton() { }

    public static LazySingleton getInstance() {
        if (instance == null) {
            // 当多个线程进来之后，虽然进行了同步加锁机制，但却不能有效验证，所以我们要进行多重加锁
            synchronized(LazySingleton.class){
                    instance = new LazySingleton();
            }
        }
        return instance;
    }
}

// 第三步【第一种方式完成】
public class LazySingleton {

   //volatile修饰的变量可以保证多线程环境下的可见性以及禁止指令重排序
    private volatile static LazySingleton instance = null;

    private LazySingleton() { }

    public static LazySingleton getInstance() {
        // 第一重判断
        if (instance == null) {
            //  同步加锁机制
            synchronized(LazySingleton.class){
```

```
                // 第二次判断
                if (instance == null) {
                        instance = new LazySingleton();
                }
            }
        }
        return instance;
    }
}


// 【第二中方式】
public class Singleton {
    private Singleton() {
    }

    // 通过静态内部类实现不会通过类加载进行实例化
    private static class HolderClass {
        final static Singleton instance = new Singleton();
    }

    public static Singleton getInstance() {
        // 第一次调用getInstance，加载内部类HolderClass
        return HolderClass.instance;
    }
}
```

优点：

- 如果一直没有调用，则会节省空间

缺点：

- 时间换空间，造成判断产生的时间

工厂设计模式

> 最常用的一种创建型设计模式，提供了创建对象的最佳方式，在此模式中，我们创建对象不会对客户端暴露创建逻辑，而是通过使用共同的借口来指向创建对象。

工厂模式分类

	Simple Factory	Factory Method	Abstract Factory
分类	简单工厂模式	工厂方法模式	抽象工厂模式
描述	又称为静态工厂方法	又称为多态性构造工厂模式	又称为集装箱模式（Kit | ToolKit）
设计模式	创建型	创建型	创建型

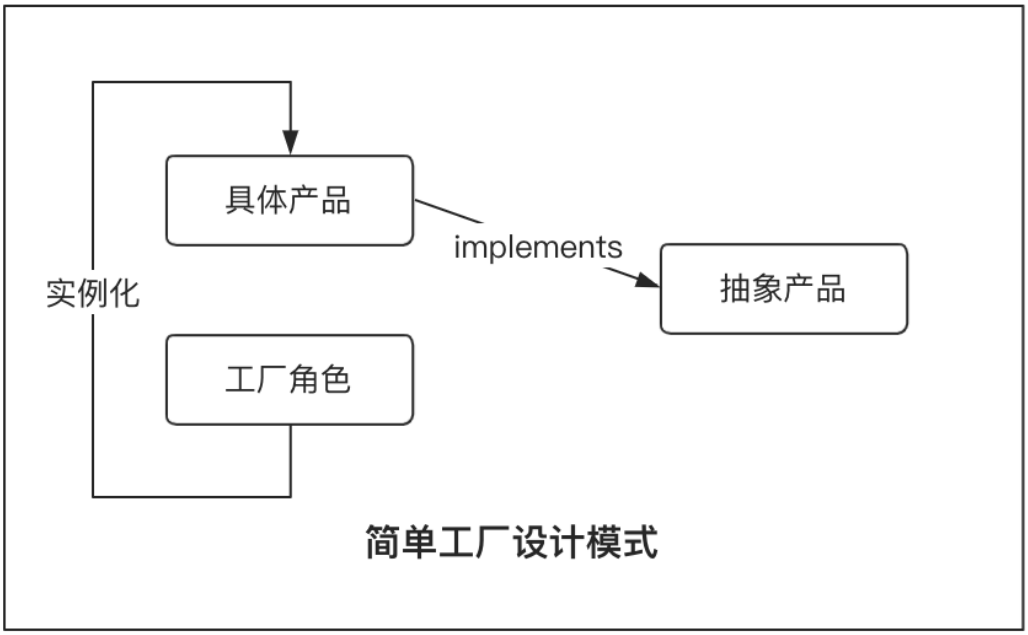
- 我们为什么需要工厂模式
  - 解耦：把对象的创建和使用分离
  - 可复用：对创建复杂过程比较复杂的对象，并且很多地方都会用到此对象，那么我们就需要通过工厂模式增强此对象创建的代码复用性
  - 降低成本：由于复杂对象通过工厂进行统一管理，此时只需要修改工厂内部的对象创建过程就可以维护对象，整体成本降低。

工厂模式理论

- 简单工厂模式

> 简单模式整体我们用的相比后两者是比较少的，因为本身它违背来开闭原则，虽然可以通过反射+配置文件解决，但总体还是不太友好。

  - 什么时候用简单工厂模式?
    1. 需要创建对象较少
    2. 客户端不需要关注对象的创建过程
  - 如何看待简单工厂模式中的角色?



工厂角色： 是整个工厂模式的核心，负责实例化内部的所有逻辑，工厂类可以被外界直接调用，创建所要的实例化对象。

抽象产品：简单工厂创建出的对象都是抽象产品类，它负责描述实例所有公共接口。

具体产品：简单工厂模式的创建目标，所有的创建对象最终都是充当这个角色的具体的实例。

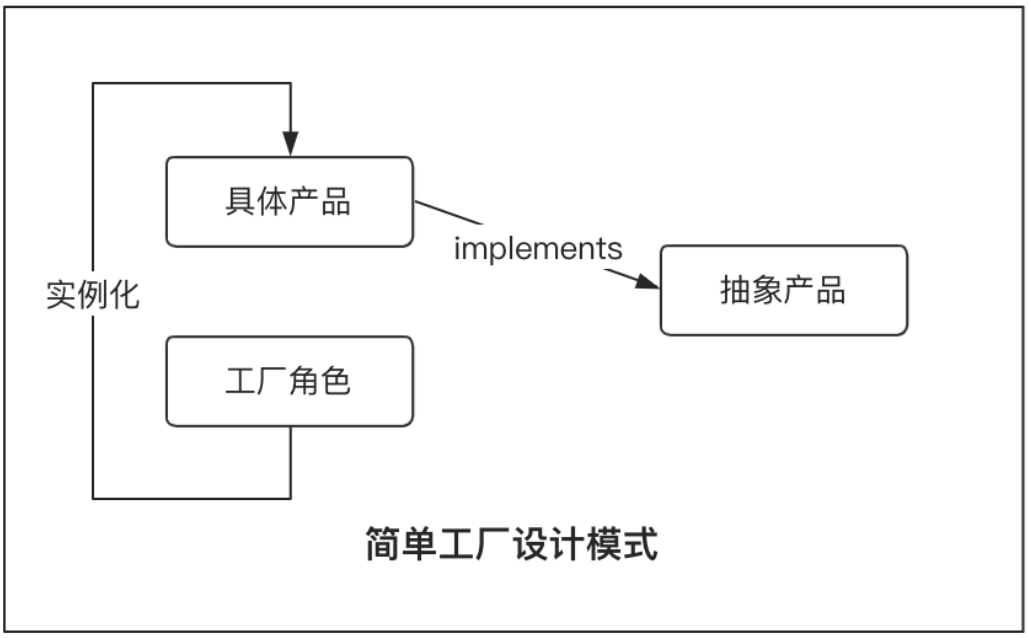
- 工厂方法模式

> 工厂方法模式是简单工厂模式的进一步深化，在工厂方法模式中，不会像简单工厂模式一样，通过一个工厂来完成所有对象的创建，而是针对不同的对象提供不同的工厂，也就是说每个对象都有一个对应的工厂。

  - 什么时候用工厂方法模式?
    1. 一个类不需要知道它所需要的对象的类，但只需要知道具体类的对应工厂类

2. 一个类通过其子类来决定创建那个对象，工厂类只需要提供一个创建产品的接口
3. 将创建对象的任务委托给多个工厂的任意子类创建，也可以动态指定由那个工厂的子类创

- 如何看待工厂方法模式中的角色?



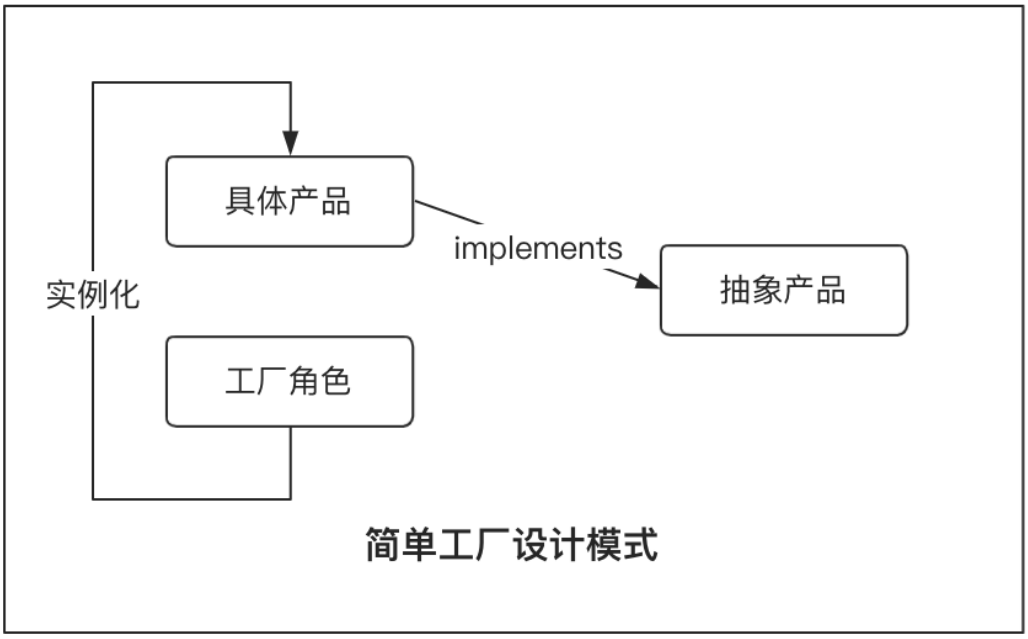
- 抽象工厂模式

> 抽象工厂模式是工厂方法的进一步深化，在工厂方法模式中，工厂仅仅可以创建一种产品，然而在抽象工厂模式中，此工厂不仅仅可以创建一种产品，还可以创建一组产品。

- 什么时候采用抽象工厂模式?

1. 客户端不需要知道所创建对象的类
2. 需要一组对象完成某种功能或者多组对象完成不同的功能，
3. 系统稳定，不会额外增加对象

- 如何看待抽象工厂模式中的角色?



工厂模式实践

- 工厂简单示例

  :) 比如一个程序猿生日想要一台电脑，有很多品牌的电脑，这个程序猿只想买自己想要的品牌电脑。

```
/**
 * Calculation接口
 * 抽象产品（电脑公用接口）
 */
public abstract interface Calculation {

    /**
     * 开机
     */
    void PowerOn();

    /**
     * 关机
     */
    void PowerOff();

}




/**
 * 戴尔电脑
 * 具体实现类
 */
public class DellComputer implements Calculation{

    public DellComputer(){
        //私有构造方法，在类之外不能通过new获得本类对象了，保证了单例
    };

    @Override
    public void PowerOn() {
        System.out.println("Dell电脑正在开机");
    }

    @Override
    public void PowerOff() {
        System.out.println("Dell电脑正在开机");
    }

}




/**
 * 小米电脑
 * 产品具体实现类
 */
public class XmComputer implements Calculation{
```

```
    public XmComputer(){
        //私有构造方法，在类之外不能通过new获得本类对象了，保证了单例
    };

    @Override
    public void PowerOn() {
        System.out.println("小米电脑正在开机");
    }

    @Override
    public void PowerOff() {
        System.out.println("小米电脑正在开机");
    }

}




/**
 * 产品工厂
 */
public class ComputerFactory {

    public Calculation create(String computers) {
        if (computers == null) {
            return null;
        }
        if (computers.equalsIgnoreCase("DELL")) {
            return new DellComputer();
        } else if (computers.equalsIgnoreCase("XM")) {
            return new XmComputer();
        }
        return null;
    }

}



public class Computer {

    public static void main(String[] args) {
        //创建工厂
        Calculation xm = new ComputerFactory().create("XM");

        //开机
        xm.PowerOn();

    }
}
```

优点：

- 一个调用者想创建一个对象，只需要知道其名称就可以实现

缺点：

- 违背了开放-封闭原则，虽然可以通过反射+配置文件解决，但用起来还是比较麻烦

- 工厂方法模式示例

  :) 比如一个程序猿生日想要一台电脑，京东从工厂直接发货，程序猿不需要在意电脑机箱是怎么制作、电脑硬件是如何组装等等。

```
/**
 * Calculation接口
 * 抽象产品（电脑公用接口）
 */
public abstract interface Calculation {

    /**
     * 开机
     */
    void PowerOn();

    /**
     * 关机
     */
    void PowerOff();

}



/**
 * 台式电脑 | 具体实现类
 */
public class DesktopComputer implements Calculation{

    public DesktopComputer(){
        //私有构造方法，在类之外不能通过new获得本类对象了，保证了单例
    };

    @Override
    public void PowerOn() {
        System.out.println("台式电脑正在开机");
    }

    @Override
    public void PowerOff() {
        System.out.println("台式电脑正在关机");
    }

}



/**
 * 抽象产品工厂
 */
```

```
public abstract class ComputerFactory {

    public abstract Calculation create();

}




/**
 * 具体产品工厂
 */
public class DesktopFactory extends ComputerFactory {

    @Override
    public Calculation create() {
        return new DesktopComputer();
    }

}




/**
 * 最终应用实例        [Computer.java]
 */
public class Computer {
    public static void main(String[] args) {
        //创建工厂
        DesktopFactory factory = new DesktopFactory();
        //通过工厂创建具体实例化的子类
        Calculation calculation = factory.create();
        //由子类实现
        calculation.PowerOn();
    }
}
```

优点:

- 一个调用者想创建一个对象，只需要知道其名称就可以实现
- 扩展性较高，如果要增加一个产品，只需要扩展一个工厂类就可以实现
- 对调用者屏蔽产品具体实现，调用者只需要关注接口

缺点:

- 当我们需要扩展增加产品时，都需要增加一个具体类和一个具体实现工厂，这样系统中类的数据会进行倍增，一定的程度上增加了系统的复杂度，同时增加了系统具体类的依赖。

- 抽象工厂模式示例

```
/**
 * 电脑产品公用接口
 */
public interface Cpu {

    /**
```

```
     * 制造CPU
     */
    public void markCpu();

}


public interface Harddisk {

    /**
     * 制造硬盘
     */
    public void markHarddisk();


}


public interface Memory {

    /**
     * 制造内存条
     */
    public void markMemory();

}


/**
 * 戴尔电脑
 * 具体CPU实现类
 */
public class DellComputerCpu implements Cpu{

    public DellComputerCpu(){
        //私有构造方法，在类之外不能通过new获得本类对象了，保证了单例
    };

    @Override
    public void markCpu() {
        System.out.printf("制作戴尔CPU");
    }

}


/**
 * 戴尔电脑
 * 具体Memory实现类
 */
public class DellComputerMemory implements Memory{

    public DellComputerMemory(){
        //私有构造方法，在类之外不能通过new获得本类对象了，保证了单例
    };

    @Override
    public void markMemory() {
```

```
        System.out.printf("制作戴尔Memory");
    }

}



/**
 * 戴尔电脑
 * 具体Harddisk实现类
 */
public class DellComputerHarddisk implements Harddisk{

    public DellComputerHarddisk(){
        //私有构造方法，在类之外不能通过new获得本类对象了，保证了单例
    };

    @Override
    public void markHarddisk() {
        System.out.printf("制作戴尔Harddisk");
    }

}


/**
 * 工厂公用接口
 */
public interface ComputerFactory {


    public Cpu madeCpu();

    public Memory madeMemory();

    public Harddisk madeHarddisk();

}



public class DellComputerFactory implements ComputerFactory{
    @Override
    public Cpu madeCpu() {
        return new DellComputerCpu();
    }

    @Override
    public Memory madeMemory() {
        return new DellComputerMemory();
    }

    @Override
    public Harddisk madeHarddisk() {
        return new DellComputerHarddisk();
    }
}
```

```
public class Computer {

    public static void main(String[] args) {
        //创建工厂
        ComputerFactory factory;
        //定义Cpu
        Cpu cpu;
        //定义内存
        Memory meory;
        //定义硬盘
        Harddisk hd;


        DellComputerFactory dellComputerFactory = new DellComputerFactory();

        cpu = dellComputerFactory.madeCpu();
        cpu.markCpu();

    }
}
```

优点：

- 扩展性较高，可以通过一组对象实现某个功能

缺点：

- 一旦增加就需要修改原有代码，不符合开闭原则，所以尽可能用在不需要修改

Spring中工厂模式示例

经过前面的应用工厂示例，我们这次直接进行Spring中工厂模式的使用，代码如下：

```
public interface BeanFactory {

        String FACTORY_BEAN_PREFIX = "&";

    Object getBean(String name) throws BeansException;

    <T> T getBean(String name, 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;

    <T> ObjectProvider<T> getBeanProvider(Class<T> requiredType);

    <T> ObjectProvider<T> getBeanProvider(ResolvableType requiredType);

    boolean containsBean(String name);

    //.....更多其他方法，具体参考链接：
}
```

```
public class DefaultListableBeanFactory extends
AbstractAutowireCapableBeanFactory
        implements ConfigurableListableBeanFactory, BeanDefinitionRegistry,
Serializable{

        @Nullable
        private static Class<?> javaxInjectProviderClass;

        @Override
    public <T> T getBean(Class<T> requiredType, @Nullable Object... args) throws
BeansException {
        Assert.notNull(requiredType, "Required type must not be null");
        Object resolved = resolveBean(ResolvableType.forRawClass(requiredType),
args, false);
        if (resolved == null) {
          throw new NoSuchBeanDefinitionException(requiredType);
        }
        return (T) resolved;
    }

    // ......BeanFactory更多实现方法姑且不提，参考链接：
}
```

原型设计模式

> 原型设计模式用于创建重复对象的同时保证性能，此模式属于创建型模式，提供了一种创建对象的最佳方式。

原型模式论述

	Prototype Pattern
别名	适配器模式
描述	用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象
设计模式	创建型

- 我们为什么需要原型设计模式
  - 提高性能：通过new创建对象不能获取当前对象运行时的状态，并且new复制给新对象并没有直接clone性能要高
  - 逃避构造函数：原型模式生成的新对象可能是一个派生类。拷贝构造函数生成的新对象只能是类本身

原型模式理论

- 原型模式

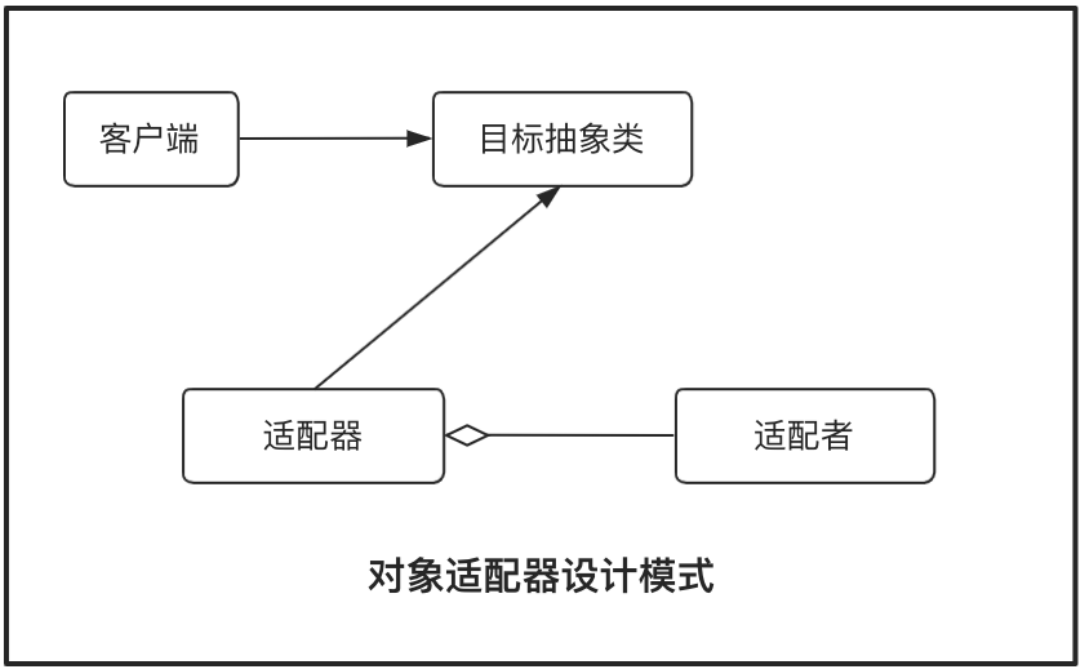
  > 用一个已经创建的实例作为原型，通过复制该原型对象来创建一个和原型相同或相似的新对象。在这里，原型实例指定了要创建的对象的种类。用这种方式创建对象非常高效，根本无须知道对象创建的细节

  - 什么时候用原型模式?

    1. 对象之间相同或相似，即只是个别的几个属性不同的时候
    2. 对象的创建过程比较麻烦，但复制比较简单的时候

  3. 资源优化场景，在实际项目中，原型模式很少单独出现，一般是和工厂方法模式一起出现，通过 clone 的方法创建一个对象，然后由工厂方法提供给调用者
  4. 一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用

- 如何看待原型模式的角色?



具体原型类：实现抽象原型类的 clone() 方法，它是可被复制的对象

问类：使用具体原型类中的 clone() 方法来复制新的对象

原型模式实践

- 原型模式

  :) 有个程序猿想买最新的MAC电脑，通过3D打印去一个美国朋友家把最新的MAC电脑进行打印了下来，制作了一个仿真模型

```
//具体原型类
class Realizetype implements Cloneable
{
    Realizetype()
    {
        System.out.println("MAC电脑模具生产成功");
    }
    public Object clone() throws CloneNotSupportedException
    {
```

```
        System.out.println("3D打印MAC电脑");
        return (Realizetype)super.clone();
    }
}


public class Prototypes {

    public static void main(String[] args)throws CloneNotSupportedException
    {
        Realizetype obj1=new Realizetype();
        Realizetype obj2=(Realizetype)obj1.clone();
        System.out.println("obj1==obj2?"+(obj1==obj2));
    }

}
```

优点：

- 简化对象创建提高性能，它直接操作内存中的二进制流，特别是复制大对象时，性能的差别非常明显

缺点：

- 实现原型模式每个派生类都必须实现 Clone接口

结构型模式

代理设计模式

> 代理模式是一种常见的结构性模式，给某一个对象提供代理，并由代理控制对源对象的引用，此时可以通过一个称之为代理层对象，类似中介，来实现客户端与目标端的链接作用，并且可以通过代理对象去掉客户端无需看到的额外资源与服务。

代理模式论述

	Proxy Pattern
别名	代理模式
描述	为其他对象提供一种代理来控制访问其对象
设计模式	结构型
实现方式	静态代理 | 动态代理

- 我们为什么需要代理模式
  - 降低耦合度：代理模式作为代理层可以协调调用者及被调用者，一定程度解决了耦合性的问题

- 高性能：通过远程代理对可以访问远程机器的对象，这样更加具有性能上的提升空间可进行数据计算分析等，并且通过一个代理的小对象可以控制一个大对象，减少了资源开销，对系统进行了优化并提高了运行速度
- 安全性：通过代理层可校验被调用者的权限

代理模式理论

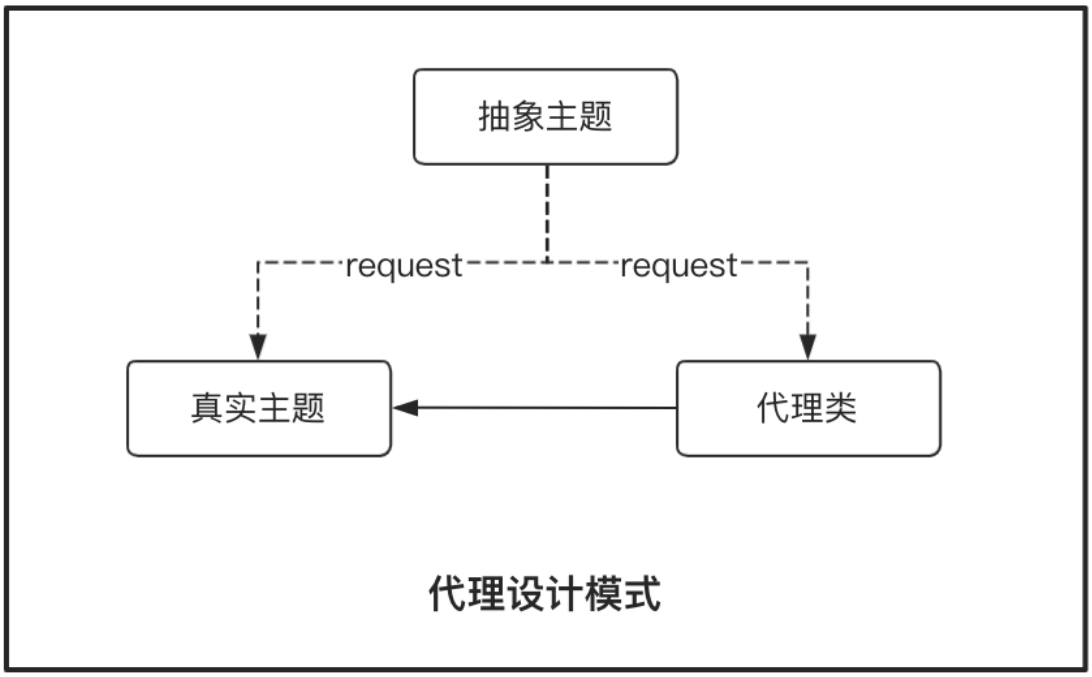
- 代理模式

> 通过引入代理对象的方式来间接访问目标对象，防止直接访问目标对象给系统带来不必要的复杂性

  - 什么时候用代理模式?

    1. 要访问的过于复杂的对象需要过费很长时间
    2. 需要屏蔽的真实对象的部分服务功能

- 如何看待代理模式的角色?



抽象主题：通过接口或者抽象类声明真实主题和代理对象的实现的业务方法

真实主题：实现抽象主题中的具体业务，是代理对象所表示的真实对象，也是最终引用的对象

代理角色：提供了与真实主题的相同接口，其中内部包含对真实主题的引用，可以包含访问、控制、扩展真实主题的功能。

代理模式实践

- 代理模式

:) 有个程序猿想买最新的MAC电脑，但是国内不能购买，只能通过寻找代理商在美国购买。

```
/**
 * mac电脑国内抽象购买
```

```
 */
public interface MacSubject {

    // 购买MAC电脑
    public void macStore();

}


/**
 * 美国真实购买
 */
public class MacRealSubject implements MacSubject{

    @Override
    public void macStore() {
        System.out.printf("美国商店购买了MAC");
    }

}



public class MacProxy implements MacSubject{

    @Override
    public void macStore() {
        //创建并引用真实对象
        MacRealSubject mac = new MacRealSubject();
        //调用真实对象的方法，购买美国MAC
        mac.macStore();
        //由代理发货
        this.sendMac();
    }

    private void sendMac(){
        System.out.println("从美国发货到国内");
    }

}



public class ProxyPattern {

    public static void main(String[] args) {
        MacProxy macProxy = new MacProxy();
        macProxy.macStore();
    }

}
```

优点：

- 通过代理降低了调用者及被调用者之间的耦合性
- 代理对象作为客户端及目标对象之间的中介，起到保护目标对象的作用，并可以做出相应扩展功能。

缺点：

- 由于调用者及被调用者之间增加了代理对象，因此会造成请求处理变慢
- 实现代理模式需要额外的工作，从而增加系统实现的复杂度

适配器设计模式

> 适配器模式可以使接口不兼容的一些类可以一起工作，其为包装器（Wrapper），适配器模式即可以作为类结构模式，也可以作为对象结构模式。

适配器模式论述

	Adapter Pattern
别名	适配器模式
描述	将一个接口转换成客户端希望的另一个接口，适配器作为不兼容的两个接口之间的桥梁
设计模式	结构型（类结构、对象结构）

- 我们为什么需要适配器模式
  - 增强灵活性：因为适配器类是适配者的子类，因此我们可以在适配器类中进行适配者的部分方法置换，从而实现适配器的灵活性
  - 扩展性：一个对象适配器可以把多个是适配者适配到同一个目标，也就是说，同一个适配器可以把适配者类和它的子类都适配到目标接口

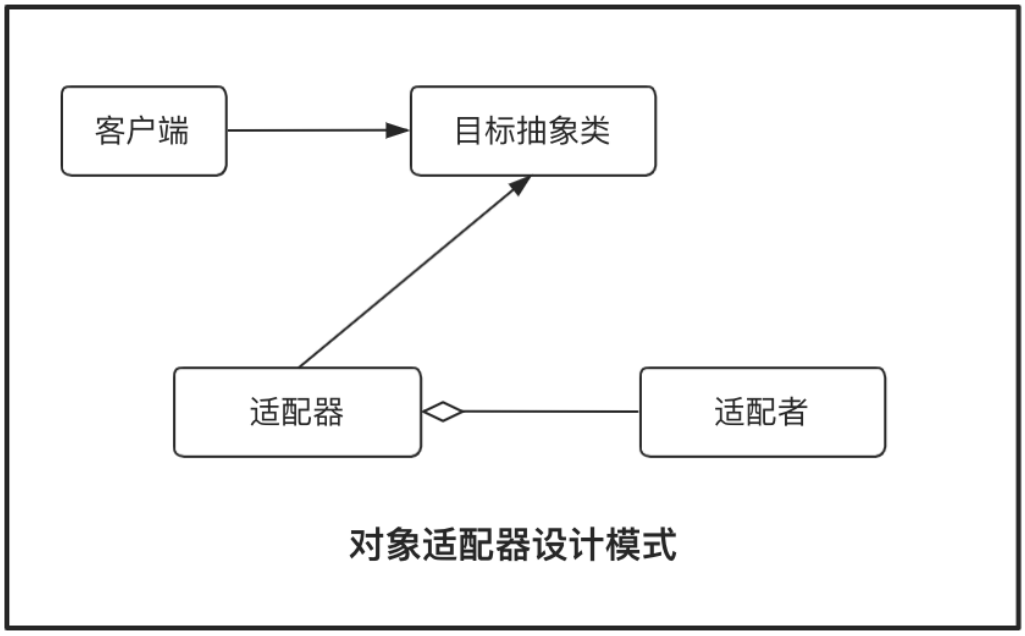
适配者模式理论

- 适配者模式

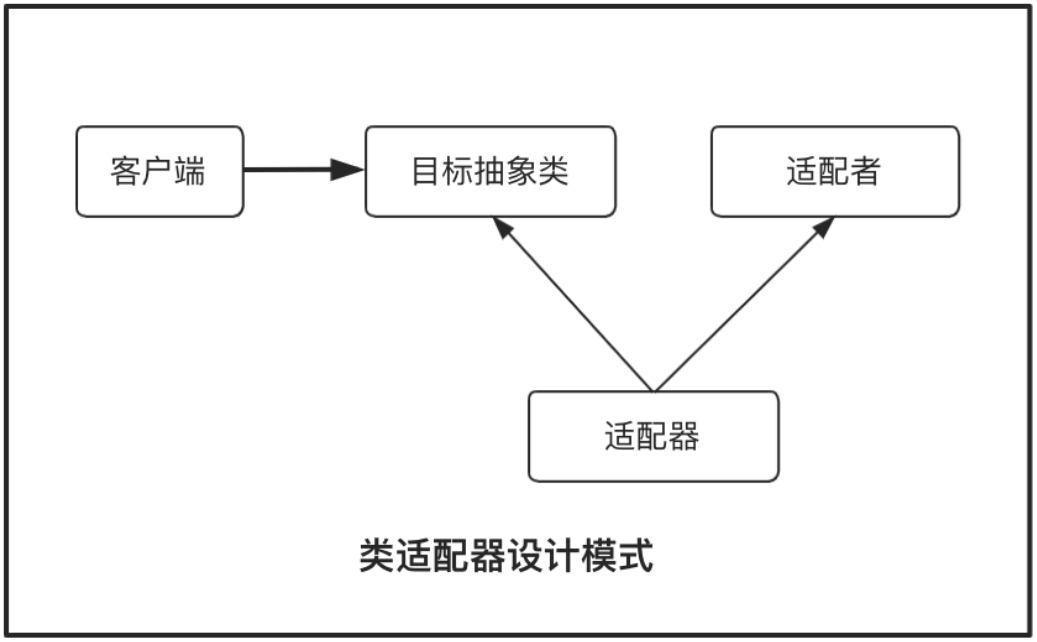
> 类适配器可以采用多重继承的方法实现，比如C++可以定义一个适配器同时继承当前业务接口和现有组件库存在的组件接口，Java不支持多继承，所以我们需要定义一个适配器来实现当前系统业务的接口，同时继承组件库存在的组件

> 对象适配器可以采用将现有组件库中已经实现的组件引入适配器中，并且适配器实现当前系统的业务接口

- 什么时候用适配器模式?
  1. 系统需要使用现有的类，而这些类的接口不符合系统需要
  2. 想要建立一个可重复使用的类，用于有一些彼此之间没有关联的类或未来引进一些新的类一起工作
- 如何看待适配器模式的角色?
  - 对象适配者角色



- 类适配者角色



目标接口：当前业务所期待的接口，它可以是抽象类或者接口

适配者类：它是访问和适配的现存组件库中的组件接口

适配器类：它是一个转换器，通过继承或引用适配者的对象，把适配者接口转成目标接口，让客户按照目标接口的格式访问适配者

适配者模式实践

- 适配者模式

  :) 有个程序猿想买最新的MAC电脑，但是国内不能购买，只能通过寻找代理商在美国购买。

```
//目标接口
interface Target
{
```

```
    public void request();
}


//适配者接口
class Adaptee
{
    public void specificRequest()
    {
        System.out.println("适配者业务代码被调用！");
    }
}


//类适配器类
class ClassAdapter extends Adaptee implements Target
{
    public void request()
    {
        specificRequest();
    }
}


/**
 * 类适配器测试
 */
public class Oadapter {

    public static void main(String[] args)
    {
        System.out.println("类适配器：");
        Target target = new ClassAdapter();
        target.request();
    }

}



//################分割线#################3

//对象适配器类
class ObjectAdapter implements Target
{
    private Adaptee adaptee;
    public ObjectAdapter(Adaptee adaptee)
    {
        this.adaptee=adaptee;
    }
    public void request()
    {
        adaptee.specificRequest();
    }
}

public class ObjectAdapters {
    public static void main(String[] args)
```

```
    {
        System.out.println("对象适配器: ");
        Adaptee adaptee = new Adaptee();
        Target target = new ObjectAdapter(adaptee);
        target.request();
    }
}
```

优点:

- 将目标类和适配者类解耦，通过引入一个适配器类来重用现有的适配者类，而无须修改原有代码
- 增加了类的透明性和复用性，将具体的实现封装在适配者类中
- 灵活性和扩展性都非常好，通过使用配置文件，可以很方便地更换适配器，也可以在不修改原有代码的基础上增加新的适配器类，完全符合“开闭原则”

缺点:

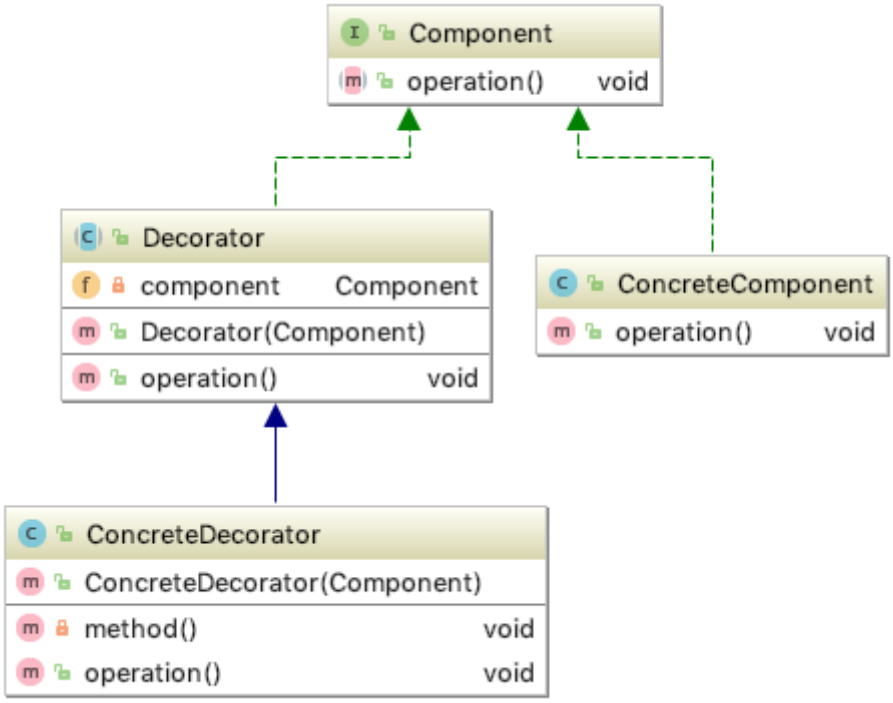
- 类适配器中对于不支持多重继承的语言，一次只能够适配一个适配者类，并且目标抽象类必须是抽象类，具有一定的局限性，不能将适配者和它的子类都适配到目标接口。
- 对象适配器中，要想置换适配者类的方法就不容易，如果一定要置换掉适配者类的一个或多个方法，就只好先做一个适配者类的子类，将适配者类的方法置换掉，然后再把适配者类的子类当做真正的适配者进行适配，实现过程较为复杂。

装饰器模式

论述

装饰器Decorator模式
动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式比生成子类更为灵活。

理论



抽象构件（Component）角色：该角色用于规范需要装饰的对象（原始对象）。
具体构件（Concrete Component）角色：该角色实现抽象构件接口，定义一个需要装饰的原始类。
装饰（Decorator）角色：该角色持有一个构件对象的实例，并定义一个与抽象构件接口一致的接口。
具体装饰（Concrete Decorator）角色：该角色负责对构件对象进行装饰。

我们为什么需要装饰器模式

装饰器模式是一种结构性模式，它作用是对对象已有功能进行增强，但是不改变原有对象结构。这避免了通过继承方式进行功能扩充导致的类体系臃肿。

装饰模式是对继承的有力补充。单纯使用继承时，在一些情况下就会增加很多子类，而且灵活性差，维护也不容易。装饰模式可以替代继承，解决类膨胀的问题，如Java基础类库中的输入输出流相关的类大量使用了装饰模式。

使用场景

■ 需要扩展一个类的功能，或给一个类增加附加功能。

■ 需要动态地给一个对象增加功能，这些功能可以再动态地撤销。

■ 需要为一批类进行改装或加装功能。

实践

模型案例

```java
package com.naixue.vip.p6.decorator;

/**
 * @Description 抽象构件Component接口
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public interface Component {

    public void operation();
}
```

```java
package com.naixue.vip.p6.decorator;

/**
 * @Description 抽象的装饰角色
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:40
 **/
public abstract class Decorator implements Component {

    private Component component = null;

    public Decorator(Component component) {
        this.component = component;
    }

    @Override
    public void operation() {
        this.component.operation();
    }
}
```

```java
package com.naixue.vip.p6.decorator;

/**
 * @Description 具体构件,被修饰者
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:38
 **/
public class ConcreteComponent implements Component {

    @Override
    public void operation() {
        //业务代码
        System.out.println("小屋");
    }
}
```

```java
package com.naixue.vip.p6.decorator;

/**
 * @Description 装饰者
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:42
```

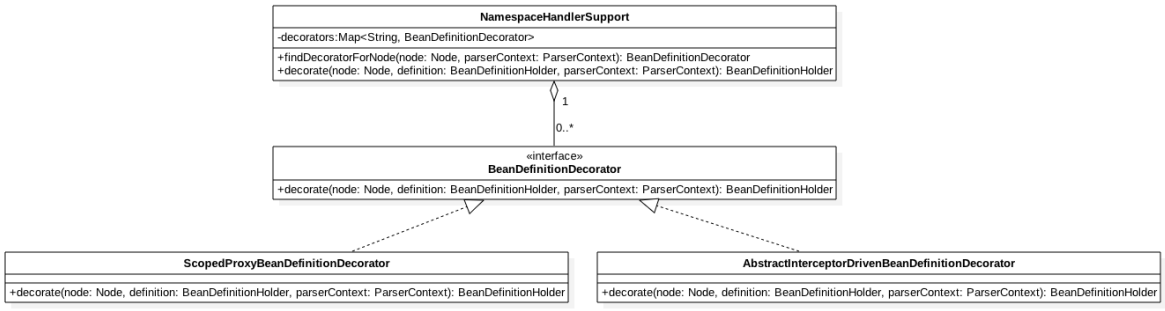
```
 **/
public class ConcreteDecorator extends Decorator {

    public ConcreteDecorator(Component component) {
        super(component);
    }

    private void method() {
        System.out.println("装饰");
    }

    @Override
    public void operation() {
        this.method();
        super.operation();
    }
}
```

```
package com.naixue.vip.p6.decorator;

/**
 * @Description
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:44
 **/
public class MainTest {

    public static void main(String[] args) {
        Component component = new ConcreteComponent();
        //进行装饰
        component = new ConcreteDecorator(component);
        component.operation();
    }
}
```

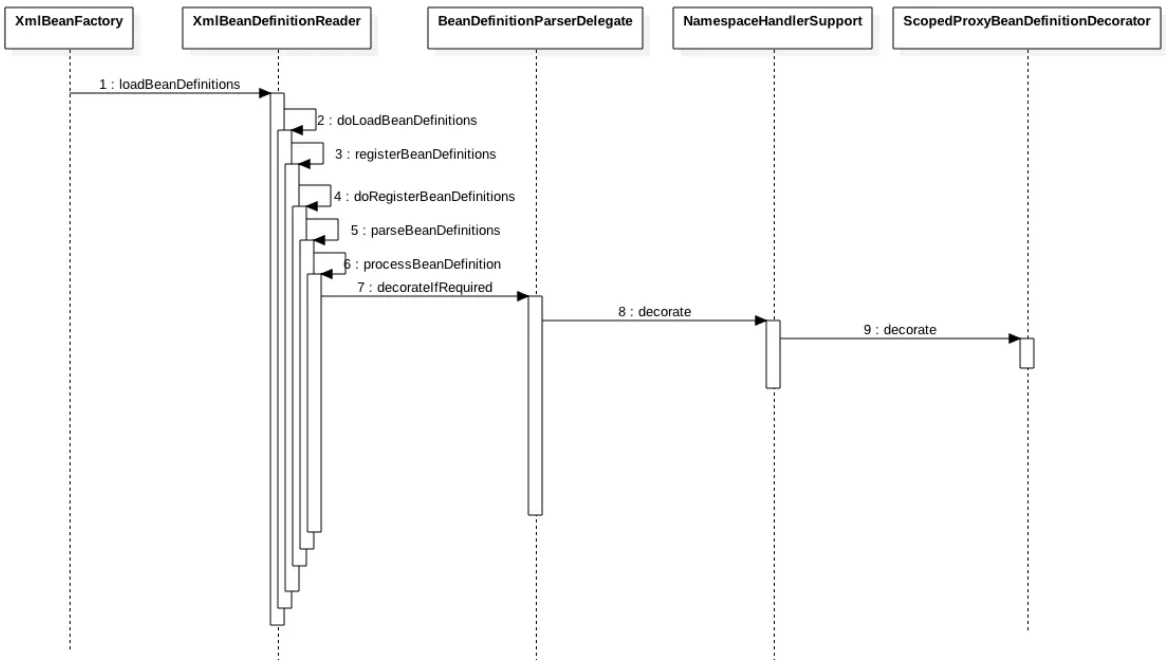
spring中的装饰器模型

Spring中用到的包装器模式在类名上有两种表现：一种是类名中含有Wrapper，另一种是类名中含有Decorator。

BeanDefinitionDecorator



如图ScopedProxyBeanDefinitionDecorator实现了decorate方法用来对scope作用域为request的bean定义进行包装。
具体时序图为：




```
class ScopedProxyBeanDefinitionDecorator implements BeanDefinitionDecorator {

    private static final String PROXY_TARGET_CLASS = "proxy-target-class";


    /**
    ** 装饰了definition
    **/
    @Override
    public BeanDefinitionHolder decorate(Node node, BeanDefinitionHolder
definition, ParserContext parserContext) {
        boolean proxyTargetClass = true;
        if (node instanceof Element) {
            Element ele = (Element) node;
            if (ele.hasAttribute(PROXY_TARGET_CLASS)) {
                proxyTargetClass =
Boolean.valueOf(ele.getAttribute(PROXY_TARGET_CLASS));
            }
        }

        // 创建scoped的代理类，并注册到容器
        BeanDefinitionHolder holder =
                ScopedProxyUtils.createScopedProxy(definition,
parserContext.getRegistry(), proxyTargetClass);
        String targetBeanName =
ScopedProxyUtils.getTargetBeanName(definition.getBeanName());
        parserContext.getReaderContext().fireComponentRegistered(
                new BeanComponentDefinition(definition.getBeanDefinition(),
targetBeanName));
        return holder;
    }

}
```

关于ScopedProxyBeanDefinitionDecorator干啥用的那：

```
<bean id="lavaPvgInfo" class="com.alibaba.lava.privilege.PrivilegeInfo"
      scope="request">
      <property name="aesKey" value="666" />
      <aop:scoped-proxy />
  </bean>
```

其实就是处理`<aop:scoped-proxy />`
的，具体作用是包装lavaPvgInfo的bean定义为ScopedProxyFactoryBean，作用是实现request作用域bean.

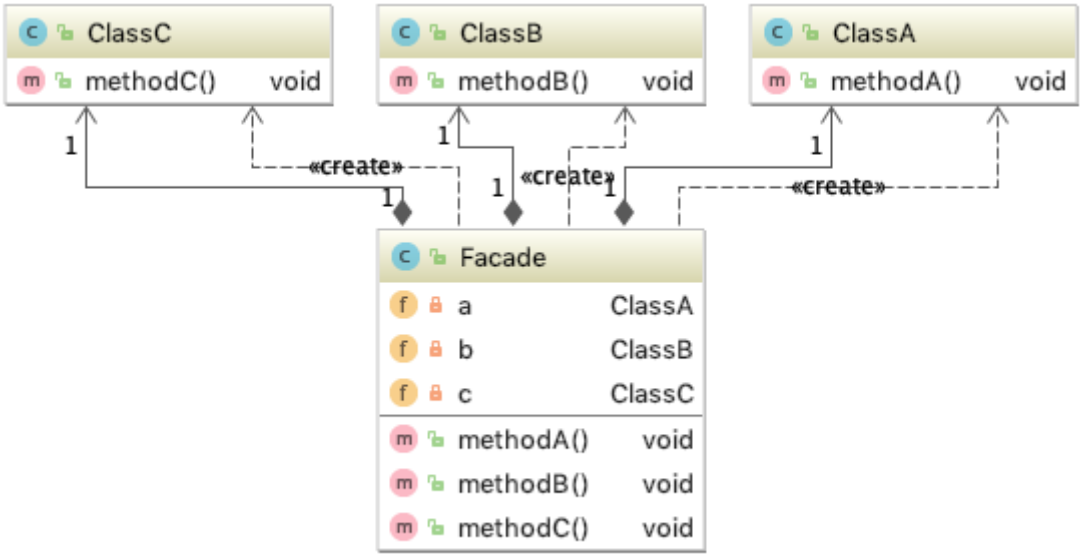
外观模式

论述

外观模式（Facade Pattern）也叫门面模式。

要求一个子系统的外部与其内部的通信必须通过一个统一的对象进行。外观模式提供一个高层次的接口，使得子系统更易使用。

理论



外观模式注重“统一的对象”，即提供一个访问子系统的接口，只有通过该接口（Facade）才能允许访问子系统的行为发生
外观（Facade）角色：客户端可以调用该角色的方法，该角色知晓相关子系统的功能和责任
子系统（Subsystem）角色：可以同时有一个或多个子系统，每一个子系统都不是一个单独的类，而是一个类的集合。

为什么使用外观模式

■ 减少系统的相互依赖，所有的依赖都是对Facade对象的依赖，与子系统无关。

■ 提高灵活性，不管子系统内部如何变化，只要不影响Facade对象，任何活动都是自由的。

■ 提高安全性，Facade中未提供的方法，外界就无法访问，提高系统的安全性。

注意外观模式最大的缺点是不符合开闭原则，对修改关闭，对扩展开放

外观模式的使用场景

■ 为一个复杂的模块或子系统提供一个供外界访问的接口。

■ 子系统相对独立，外界对子系统的访问只要黑箱操作即可。

■ 预防风险扩散，使用Facade进行访问操作控制。

实践

模型案例

```
package com.naixue.vip.p6.facade;

/**
 * @Description 隐藏子类A
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:38
 **/
public class ClassA {

    public void methodA() {
    }
}
```

```
package com.naixue.vip.p6.facade;

/**
 * @Description 隐藏子类B
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:38
 **/
public class ClassB {

    public void methodB() {
    }
}
```

```
package com.naixue.vip.p6.facade;

/**
 * @Description 隐藏子类C
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:38
 **/
public class ClassC {

    public void methodC() {
    }
}
```

```
package com.naixue.vip.p6.facade;

/**
 * @Description 对外界提供的对象
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:38
 **/
public class Facade {
```

```
    //被委托的对象
    private ClassA a = new ClassA();

    private ClassB b = new ClassB();

    private ClassC c = new ClassC();

    //提供外界的方法
    public void methodA() {
        a.methodA();
    }

    public void methodB() {
        b.methodB();
    }

    public void methodC() {
        c.methodC();
    }
}
```

Tomcat实践

`org.apache.catalina.connector.Request` 和 `org.apache.catalina.connector.RequestFacade` 这两个类都实现了 `HttpServletRequest` 接口

在 `Request` 中调用 `getRequest()` 实际获取的是 `RequestFacade` 的对象

```
protected RequestFacade facade = null;

public HttpServletRequest getRequest() {
    if (facade == null) {
        facade = new RequestFacade(this);
    }
    return facade;
}
```

在 `RequestFacade` 中再对认为是子系统的操作进行封装

```
public class RequestFacade implements HttpServletRequest {
    /**
     * The wrapped request.
     */
    protected Request request = null;

    @Override
    public Object getAttribute(String name) {
        if (request == null) {
            throw new
IllegalStateException(sm.getString("requestFacade.nullRequest"));
        }
        return request.getAttribute(name);
    }
    // ...省略...
}
```

行为型模式

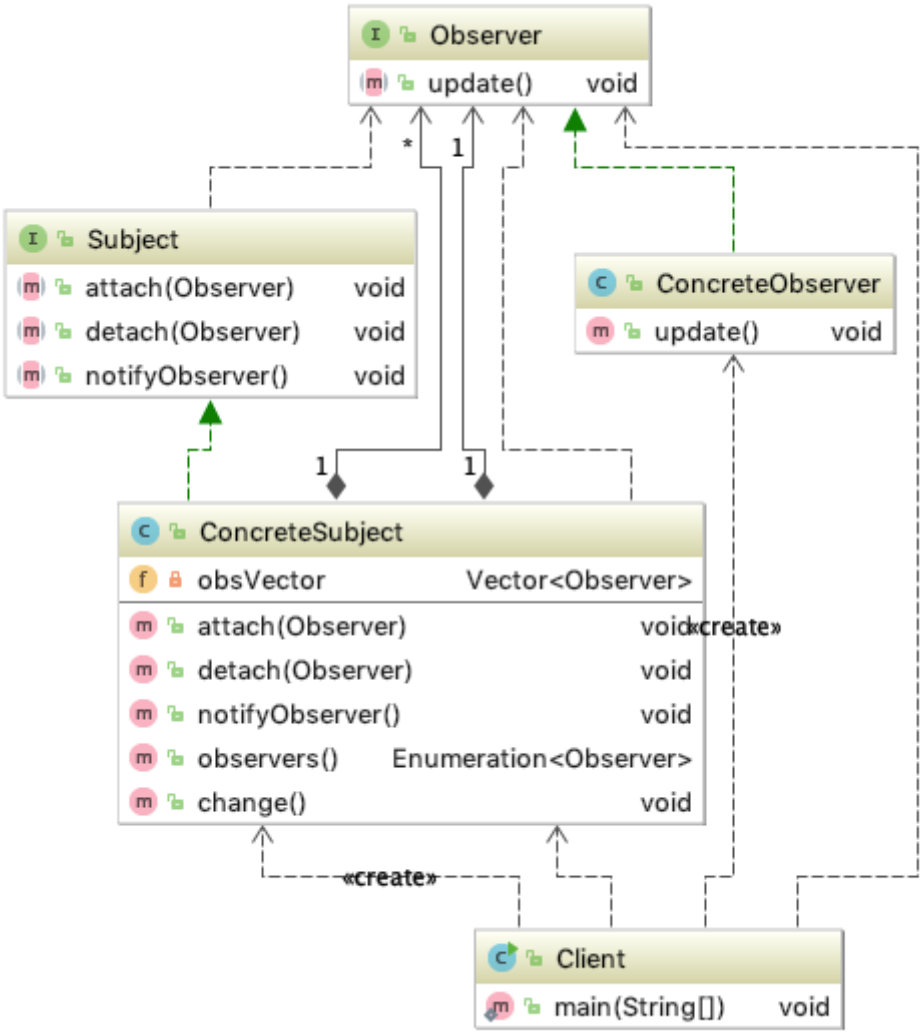
观察者模式

论述

观察者模式（Observer Pattern）也称发布订阅模式。

定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。

理论



■ 抽象主题（Subject）角色：该角色又称为“被观察者”，可以增加和删除观察者对象。

■ 抽象观察者（Observer）角色：该角色为所有的具体观察者定义一个接口，在得到主题的通知时更新自己。

■ 具体主题（Concrete Subject）角色：该角色又称为“具体被观察者”，它将有关状态存入具体观察者对象，在具体主题的内部状态改变时，给所有登记过的观察者发出通知。

■ 具体观察者（Concrete Observer）角色：该角色实现抽象观察者所要求的更新接口，以便使自身的状态与主题的状态相协调。

优点

■ 观察者和被观察者之间是抽象耦合。被观察者角色所知道的只是一个具体观察者集合，每一个具体观察者都符合一个抽象观察者的接口。被观察者并不认识任何一个具体的观察者，它只知道它们都有一个共同的接口。由于被观察者和观察者没有紧密的耦合在一起，因此它们可以属于不同的抽象化层次，且都非常容易扩展。

■ 支持广播通信。被观察者会向所有登记过的观察者发出通知，这就是一个触发机制，形成一个触发链。

缺点

■ 如果一个主题有多个直接或间接的观察者，则通知所有的观察者会花费很多时间，且开发和调试都比较复杂。

■ 如果在主题之间有循环依赖，被观察者会触发它们之间进行循环调用，导致系统崩溃。在使用观察者模式时要特别注意这一点。

■ 如果对观察者的通知是通过另外的线程进行异步投递，系统必须保证投递的顺序执行。

■ 虽然观察者模式可以随时使观察者知道所观察的对象发生了变化，但是观察者模式没有提供相应的机制使观察者知道所观察的对象是如何发生变化。

典型应用场景

■ 关联行为场景。

■ 事件多级触发场景。

■ 跨系统的消息交换场景，如消息队列的处理机制。

实践

模型案例

```
package com.naixue.vip.p6.observer;

/**
 * @Description 主题
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public interface Subject {

    //登记一个新的观察者
    public void attach(Observer obs);

    //删除一个登记过的观察者
    public void detach(Observer obs);

    //通知所有登记过的观察者对象
    public void notifyObserver();
}
```

```java
package com.naixue.vip.p6.observer;

/**
 * @Description 抽象观察者
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public interface Observer {

    //更新方法
    public void update();
}
```

```java
package com.naixue.vip.p6.observer;

/**
 * @Description 具体观察者
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public class ConcreteObserver implements Observer {

    //实现更新方法
    @Override
    public void update() {
        System.out.println("收到通知，并进行处理！");
    }
}
```

```java
package com.naixue.vip.p6.observer;

import java.util.Enumeration;
import java.util.Vector;

/**
 * @Description 具体主题，主题被多个观察者观察，可以移除观察者或者新增观察者，主题改动通知所有观察者
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public class ConcreteSubject implements Subject {

    private Vector<Observer> obsVector = new Vector<Observer>();

    //登记一个新的观察者
    @Override
    public void attach(Observer obs) {
        obsVector.add(obs);
    }

    //删除一个登记过的观察者
    @Override
    public void detach(Observer obs) {
        obsVector.remove(obs);
    }
```

```
    //通知所有登记过的观察者对象
    @Override
    public void notifyObserver() {
        for (Observer e : obsVector) {
            e.update();
        }
    }

    //返回观察者集合的Enumeration对象
    public Enumeration<Observer> observers() {
        return obsVector.elements();
    }

    //业务方法，改变状态
    public void change() {
        this.notifyObserver();
    }
}
```

```
package com.naixue.vip.p6.observer;

/**
 * @Description 客户端调用
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public class Client {

    public static void main(String args[]) {
        //创建一个主题对象（被观察者）
        ConcreteSubject subject = new ConcreteSubject();
        //创建一个观察者
        Observer obs = new ConcreteObserver();
        //登记观察者
        subject.attach(obs);
        //登记观察者
        subject.attach(obs);
        //登记观察者
        subject.attach(obs);
        //改变状态
        subject.change();
    }
}
```

在spring中的实现

spring的事件驱动模型使用的是 观察者模式， Spring中Observer模式常用的地方是listener的实现。

具体实现：

事件机制的实现需要三个部分,事件源,事件,事件监听器ApplicationEvent抽象类 [事件] 继承自jdk的EventObject,所有的事件都需要继承ApplicationEvent,并且通过构造器参数source得到事件源该类的实现类ApplicationContextEvent表示ApplicaitonContext的容器事件代码：

```
public abstract class ApplicationEvent extends EventObject {
    private static final long serialVersionUID = 7099057708183571937L;
    private final long timestamp;
    public ApplicationEvent(Object source) {
    super(source);
    this.timestamp = System.currentTimeMillis();
    }
    public final long getTimestamp() {
        return this.timestamp;
    }
}
```

ApplicationListener接口`[事件监听器]`

继承自jdk的EventListener,所有的监听器都要实现这个接口。

这个接口只有一个onApplicationEvent()方法,该方法接受一个ApplicationEvent或其子类对象作为参数,在方法体中,可以通过不同对Event类的判断来进行相应的处理。

当事件触发时所有的监听器都会收到消息。

代码:

```
public interface ApplicationListener<E extends ApplicationEvent> extends
EventListener {
     void onApplicationEvent(E event);
}
```

ApplicationContext接口`[事件源]`ApplicationContext是spring中的全局容器，翻译过来是"应用上下文"。实现了ApplicationEventPublisher接口。

职责:

负责读取bean的配置文档,管理bean的加载,维护bean之间的依赖关系,可以说是负责bean的整个生命周期,再通俗一点就是我们平时所说的IOC容器。代码:

```
public interface ApplicationEventPublisher {
        void publishEvent(ApplicationEvent event);
}

public void publishEvent(ApplicationEvent event) {
    Assert.notNull(event, "Event must not be null");
    if (logger.isTraceEnabled()) {
         logger.trace("Publishing event in " + getDisplayName() + ": " + event);
    }
    getApplicationEventMulticaster().multicastEvent(event);
    if (this.parent != null) {
    this.parent.publishEvent(event);
    }
}
```

ApplicationEventMulticaster抽象类`[事件源中publishEvent方法需要调用其方法getApplicationEventMulticaster]`属于事件广播器,它的作用是把Applicationcontext发布的Event广播给所有的监听器代码:

```
public abstract class AbstractApplicationContext extends DefaultResourceLoader
```

```
    implements ConfigurableApplicationContext, DisposableBean {
    private ApplicationEventMulticaster applicationEventMulticaster;
    protected void registerListeners() {
    // Register statically specified listeners first.
    for (ApplicationListener<?> listener : getApplicationListeners()) {
    getApplicationEventMulticaster().addApplicationListener(listener);
    }
    // Do not initialize FactoryBeans here: We need to leave all regular beans
    // uninitialized to let post-processors apply to them!
    String[] listenerBeanNames = getBeanNamesForType(ApplicationListener.class,
true, false);
    for (String lisName : listenerBeanNames) {
    getApplicationEventMulticaster().addApplicationListenerBean(lisName);
    }
  }
}
```

模板方法模式

论述

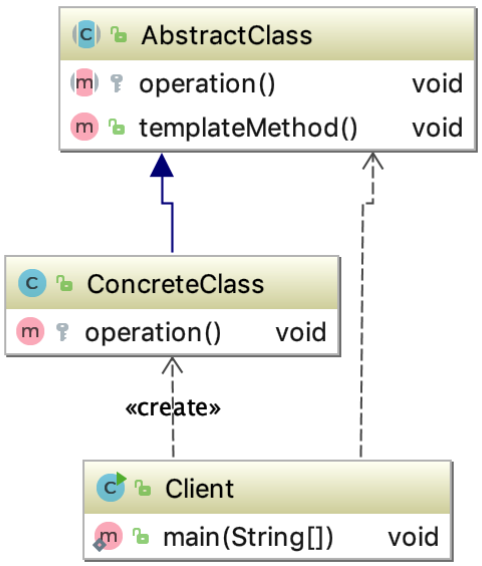
模板方法模式（Template Method Pattern）定义一个操作中的算法的框架，而将一些步骤延迟到子类中。使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

父类定义了骨架（调用哪些方法及顺序），某些特定方法由子类实现。最大的好处：代码复用，减少重复代码。除了子类要实现的特定方法，其他方法及方法调用顺序都在父类中预先写好了。所以父类模板方法中有两类方法：共同的方法：所有子类都会用到的代码不同的方法：子类要覆盖的方法，分为两种：

- 抽象方法：父类中的是抽象方法，子类必须覆盖
- 钩子方法：父类中是一个空方法，子类继承了默认也是空的

注：为什么叫钩子，子类可以通过这个钩子（方法），控制父类，因为这个钩子实际是父类的方法（空方法）！

理论



■ 抽象模板（Abstract Template）角色：该角色定义一个或多个抽象操作，以便让子类实现；这些抽象操作是基本操作，是一个顶级逻辑的组成步骤。还需要定义并实现一个或几个模板方法，这些模板方法一般是具体方法，即一个框架，实现对基本方法的调度，完成固定的逻辑。
■ 具体模板（Concrete Template）角色：该角色实现抽象模板中定义的一个或多个抽象方法，每一个抽象模板角色都可以有任意多个具体模板角色与之对应，而每一个具体模板角色都可以给出这些抽象方法的不同实现，从而使得顶级逻辑的实现各不相同。

优点

■ 封装不变的部分，扩展可变部分。不变的部分封装到父类中实现，而可变的部分则可以通过继承进行扩展。

■ 提取公共部分代码，便于维护。将公共部分的代码抽取出来放在父类中，维护时只需要修改父类中的代码。

■ 行为由父类控制，子类实现。模板方法模式中的基本方法是由子类实现的，因此子类可以通过扩展增加相应的功能，符合开闭原则。

应用场景

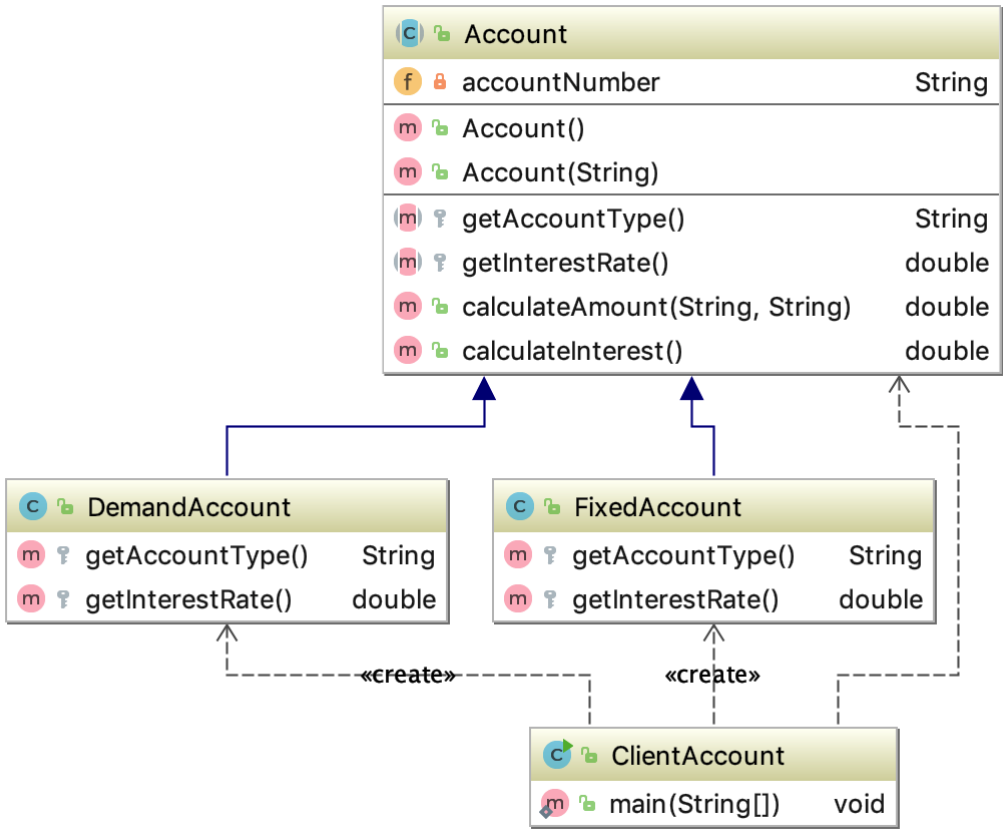
■ 多个子类有公共方法，并且逻辑基本相同时。

■ 可以把重要的、复杂的、核心算法设计为模板方法，周边的相关细节功能则由各个子类实现。

■ 重构时，模板方法模式是一个经常使用的模式，将相同的代码抽取到父类中。

实践

模型案例




```
package com.naixue.vip.p6.template.example;

/**
```

```
 * @Description 抽象模板，抽象账户类
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public abstract class Account {

    //账号
    private String accountNumber;

    //构造函数
    public Account() {
        accountNumber = null;
    }

    public Account(String number) {
        accountNumber = number;
    }

    //基本方法，确定账户类型，留给子类实现
    protected abstract String getAccountType();

    //基本方法，确定利息，留给子类实现
    protected abstract double getInterestRate();

    //基本方法，根据账户类型和账号确定账户金额
    public double calculateAmount(String accountType, String accountNumber) {
        //访问数据库.....(此处仅示意性的返回一个数值)
        return 4567.00D;
    }

    //模板方法，计算账户利息
    public double calculateInterest() {
        String accountType = getAccountType();
        double interestRate = getInterestRate();
        double amount = calculateAmount(accountType, accountNumber);
        return amount * interestRate;
    }
}
```

```
package com.naixue.vip.p6.template.example;

/**
 * @Description 具体模板类，活期账户
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public class DemandAccount extends Account {

    @Override
    protected String getAccountType() {
        return "活期";
    }

    @Override
    protected double getInterestRate() {
        return 0.005D;
    }
```

```
}
```

```
package com.naixue.vip.p6.template.example;

/**
 * @Description 具体模板类，定期账户
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public class FixedAccount extends Account {

    @Override
    protected String getAccountType() {
        return "一年定期";
    }

    @Override
    protected double getInterestRate() {
        return 0.0325D;
    }
}
```

```
package com.naixue.vip.p6.template.example;

/**
 * @Description 调用客户端
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public class ClientAccount {

    public static void main(String args[]) {
        Account account = new DemandAccount();
        System.out.println("活期利息: " + account.calculateInterest());
        account = new FixedAccount();
        System.out.println("定期利息: " + account.calculateInterest());
    }
}
```

Spring中的实践

Spring模板方法模式实质:

是模板方法模式和回调模式的结合，是Template Method不需要继承的另一种实现方式。Spring几乎所有的外接扩展都采用这种模式。

具体实现:

JDBC的抽象和对Hibernate的集成，都采用了一种理念或者处理方式，那就是模板方法模式与相应的Callback接口相结合。采用模板方法模式是为了以一种统一而集中的方式来处理资源的获取和释放，以JdbcTempalte为例:

```
public abstract class JdbcTemplate {
     public final Object execute（String sql）{
        Connection con=null;
        Statement stmt=null;
```

```
        try{
            con=getConnection () ;
            stmt=con.createStatement () ;
            Object retValue=executeWithStatement (stmt,sql) ;
            return retValue;
        }catch (SQLException e) {
             ...
        }finally{
            closeStatement (stmt) ;
            releaseConnection (con) ;
        }
    }
    protected abstract Object executeWithStatement (Statement   stmt, String
sql) ;
}
```

引入回调原因：

JdbcTemplate是抽象类，不能够独立使用，我们每次进行数据访问的时候都要给出一个相应的子类实现,这样肯定不方便，所以就引入了回调。

回调代码

```
public interface StatementCallback{
    Object doWithStatement (Statement stmt) ;
}
```

利用回调方法重写JdbcTemplate方法

```
public class JdbcTemplate {
    public final Object execute (StatementCallback callback) {
        Connection con=null;
        Statement stmt=null;
        try{
            con=getConnection () ;
            stmt=con.createStatement () ;
            Object retValue=callback.doWithStatement (stmt) ;
            return retValue;
        }catch (SQLException e) {
            ...
        }finally{
            closeStatement (stmt) ;
            releaseConnection (con) ;
        }
    }

    ...//其它方法定义
}
```

Jdbc使用方法如下：

```
JdbcTemplate jdbcTemplate=...;
    final String sql=...;
    StatementCallback callback=new StatementCallback(){
    public Object=doWithStatement(Statement stmt){
        return ...;
    }
}
jdbcTemplate.execute(callback);
```

为什么JdbcTemplate没有使用继承?

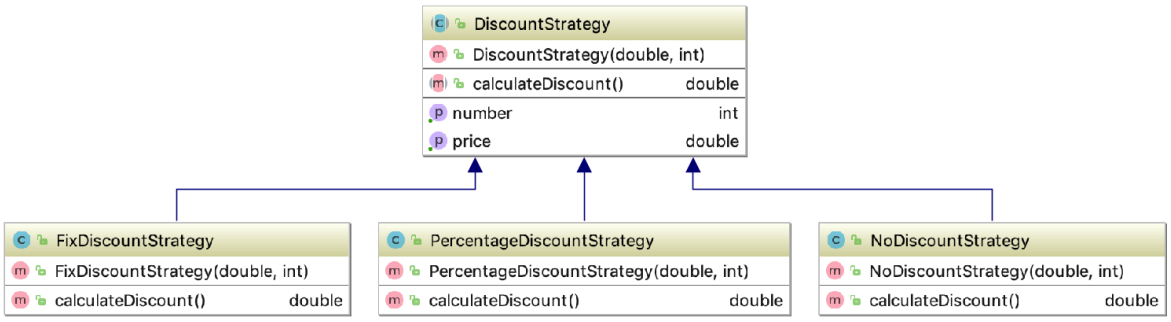
因为这个类的方法太多，但是我们还是想用到JdbcTemplate已有的稳定的、公用的数据库连接，那么我们怎么办呢？我们可以把变化的东西抽出来作为一个参数传入JdbcTemplate的方法中。但是变化的东西是一段代码，而且这段代码会用到JdbcTemplate中的变量。怎么办？那我们就用回调对象吧。在这个回调对象中定义一个操纵JdbcTemplate中变量的方法，我们去实现这个方法，就把变化的东西集中到这里了。然后我们再传入这个回调对象到JdbcTemplate，从而完成了调用。

策略模式

论述

策略Strategy模式
定义一组算法，将每个算法都封装起来，并且使它们之间可以互换。其用意是针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换，使得算法可以在不影响到客户端的情况下发生变化。

理论



■ 环境（Context）角色：该角色也叫上下文角色，起到承上启下的作用，屏蔽高层模块对策略、算法的直接访问，它持有一个Strategy类的引用。
■ 抽象策略（Strategy）角色：该角色对策略、算法进行抽象，通常定义每个策略或算法必须具有的方法和属性。
■ 具体策略（Concrete Strategy）角色：该角色实现抽象策略中的具体操作，含有具体的算法。

策略模式的优点策略模式的优点

■ 策略模式提供了管理相关的算法族的办法。策略类的等级结构定义了一个算法或行为族，恰当地使用继承可以把公共的代码移到父类中，从而避免代码重复。

■ 策略模式提供了可以替换继承关系的办法。继承可以处理多种算法或行为，如果不用策略模式，那么使用算法或行为的环境类就可能会有一些子类，每一个子类提供一个不同的算法或行为。但是，这样算法或行为的使用者就和算法本身混在一起，从而不可能再独立演化。

■ 使用策略模式可以避免使用多重条件转移语句。多重转移语句不易维护，它把采取哪一种算法或采取哪一种行为的逻辑与算法或行为的逻辑混合在一起，统统列在一个多重转移语句里面，这比使用继承的办法还要原始和落后。

策略模式的缺点策略模式的缺点

■ 客户端必须知道所有的策略类，并自行决定使用哪一个策略类。这就意味着客户端必须理解这些算法的区别，以便适时选择恰当的算法类，即策略模式只适用于客户端知道所有的算法或行为的情况。

■ 策略模式造成很多的策略类。有时候可以通过把依赖于环境的状态保持到客户端里面，而将策略类设计成可共享的，这样策略类实例可以被不同客户端使用。可以使用享元模式来减少对象的数量。

策略模式的应用场景使用策略模式的典型场景如下。

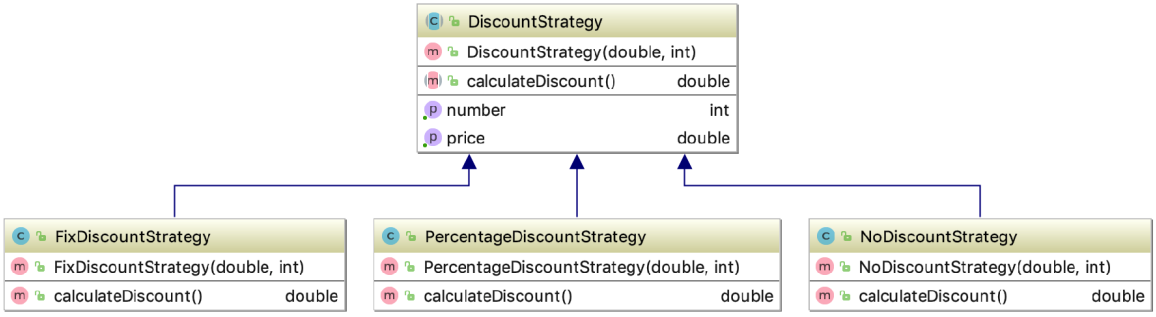
■ 多个类只是在算法或行为上稍有不同的场景。

■ 算法需要自由切换的场景。

■ 需要屏蔽算法规则的场景。

实践

模型案例




```
package com.naixue.vip.p6.strategy.example;

/**
 * @Description 抽象折扣算法类
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public abstract class DiscountStrategy {

    //书的价格
    private double price = 0;

    //书的数量
    private int number = 0;

    //构造函数
    public DiscountStrategy(double price, int number) {
        this.price = price;
        this.number = number;
    }

    //getter方法
    public double getPrice() {
        return price;
```

```
    }

    public int getNumber() {
        return number;
    }

    //策略方法，计算折扣额
    public abstract double calculateDiscount();
}
```

```
package com.naixue.vip.p6.strategy.example;

/**
 * @Description 具体折扣，固定折扣值为1的算法
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public class FixDiscountStrategy extends DiscountStrategy {

    // 构造函数
    public FixDiscountStrategy(double price, int number) {
        super(price, number);
    }

    //实现策略方法，固定折扣额
    @Override
    public double calculateDiscount() {
        return getNumber() * 1;
    }
}
```

```
package com.naixue.vip.p6.strategy.example;

/**
 * @Description 具体折扣，没有折扣算法
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public class NoDiscountStrategy extends DiscountStrategy {

    // 构造函数
    public NoDiscountStrategy(double price, int number) {
        super(price, number);
    }

    //实现策略方法，0折扣额
    @Override
    public double calculateDiscount() {
        return 0;
    }
}
```

```
package com.naixue.vip.p6.strategy.example;

/**
```

```
 * @Description 具体折扣，折扣百分比为15%的算法
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public class PercentageDiscountStrategy extends DiscountStrategy {

    // 构造函数
    public PercentageDiscountStrategy(double price, int number) {
        super(price, number);
    }

    //实现策略方法，百分比为15%的折扣额
    @Override
    public double calculateDiscount() {
        return getNumber() * getPrice() * 0.15;
    }
}
```

```
package com.naixue.vip.p6.strategy.example;

/**
 * @Description 调用客户端
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public class ContextClient {

    private DiscountStrategy ds;

    //构造函数
    public ContextClient(DiscountStrategy ds) {
        this.ds = ds;
    }

    //调用策略模式，计算折扣额
    public double contextCalDisc() {
        return ds.calculateDiscount();
    }

    public static void main(String args[]) {
        ContextClient context0 =
            new ContextClient(new NoDiscountStrategy(48.5, 20));
        System.out.println("0折扣：" + context0.contextCalDisc());
        ContextClient contextFix =
            new ContextClient(new FixDiscountStrategy(46, 60));
        System.out.println("固定折扣：" + contextFix.contextCalDisc());
        ContextClient contextPer =
            new ContextClient(new PercentageDiscountStrategy(38, 40));
        System.out.println("15%的折扣：" + contextPer.contextCalDisc());
    }
}
```

spring中的实践

实现方式：

Spring框架的资源访问Resource接口。该接口提供了更强的资源访问能力，Spring 框架本身大量使用了 Resource 接口来访问底层资源。

Resource 接口介绍

source 接口是具体资源访问策略的抽象，也是所有资源访问类所实现的接口。Resource 接口主要提供了如下几个方法:

- **getInputStream()**：定位并打开资源，返回资源对应的输入流。每次调用都返回新的输入流。调用者必须负责关闭输入流。
- **exists()**：返回 Resource 所指向的资源是否存在。
- **isOpen()**：返回资源文件是否打开，如果资源文件不能多次读取，每次读取结束应该显式关闭，以防止资源泄漏。
- **getDescription()**：返回资源的描述信息，通常用于资源处理出错时输出该信息，通常是全限定文件名或实际 URL。
- **getFile**：返回资源对应的 File 对象。
- **getURL**：返回资源对应的 URL 对象。

最后两个方法通常无须使用，仅在通过简单方式访问无法实现时，Resource 提供传统的资源访问的功能。Resource 接口本身没有提供访问任何底层资源的实现逻辑，**针对不同的底层资源，Spring 将会提供不同的 Resource 实现类，不同的实现类负责不同的资源访问逻辑。**Spring 为 Resource 接口提供了如下实现类：

- **UrlResource**：访问网络资源的实现类。
- **ClassPathResource**：访问类加载路径里资源的实现类。
- **FileSystemResource**：访问文件系统里资源的实现类。
- **ServletContextResource**：访问相对于 ServletContext 路径里的资源的实现类.
- **InputStreamResource**：访问输入流资源的实现类。
- **ByteArrayResource**：访问字节数组资源的实现类。

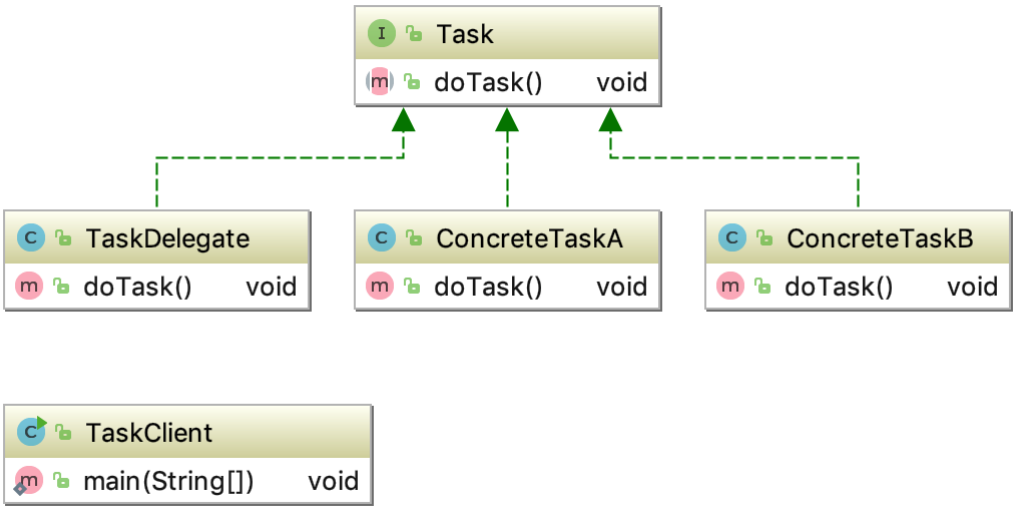
这些 Resource 实现类，针对不同的的底层资源，提供了相应的资源访问逻辑，并提供便捷的包装，以利于客户端程序的资源访问。

委派模式

论述

委派Delegate模式
实现层面上, 定义一个抽象接口, 它有若干实现类, 他们真正执行业务方法, 这些子类是具体任务角色; 定义委派者角色也实现该接口, 但它负责在各个具体角色实例之间做出决策, 由它判断并调用具体实现的方法。

理论



■ 抽象任务角色：抽象出需要执行任务的方法。
■ 委派者角色：在执行任务中调用具体任务角色的任务，达到委派的作用。
■ 具体任务角色：角色的任务交由委派者角色去执行。

实践

模型案例

```
package com.naixue.vip.p6.delegate;

/**
 * @Description 抽象任务角色
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public interface Task {

    void doTask();
}
```

```
package com.naixue.vip.p6.delegate;

/**
 * @Description 任务执行A
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public class ConcreteTaskA implements Task {

    @Override
    public void doTask() {
        System.out.println("执行 , 由A实现");
    }
}
```

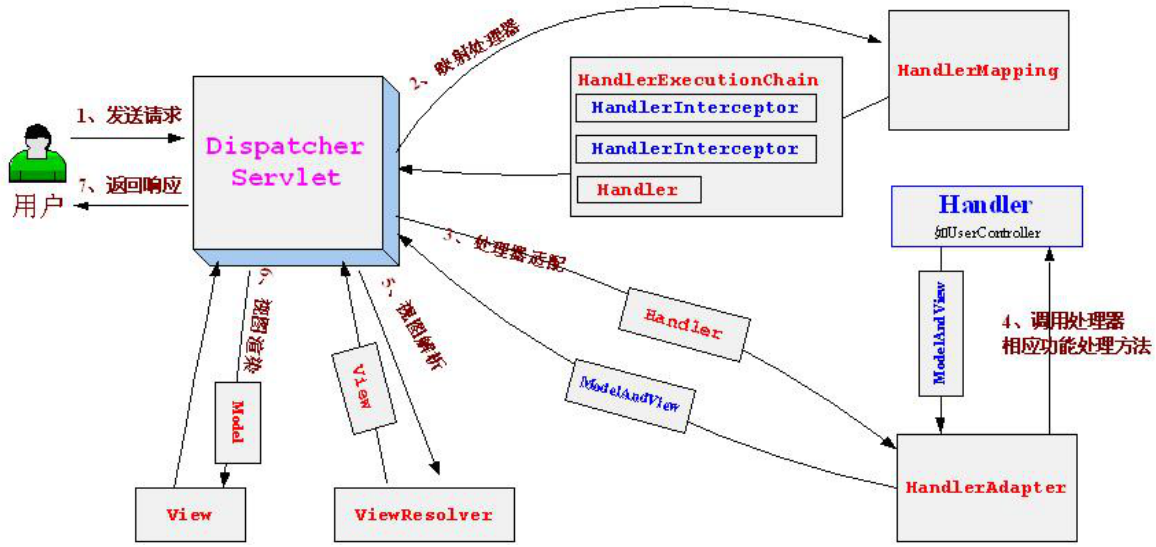
```
package com.naixue.vip.p6.delegate;

/**
 * @Description 任务执行B
```

```
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public class ConcreteTaskB implements Task {

    @Override
    public void doTask() {
        System.out.println("执行 , 由B实现");
    }
}
```

```
package com.naixue.vip.p6.delegate;

import java.util.Random;

/**
 * @Description 委派执行
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public class TaskDelegate implements Task {

    @Override
    public void doTask() {
        System.out.println("代理执行开始....");

        Task task = null;
        if (new Random().nextBoolean()) {
            task = new ConcreteTaskA();
            task.doTask();
        } else {
            task = new ConcreteTaskB();
            task.doTask();
        }

        System.out.println("代理执行完毕....");
    }
}
```

```
package com.naixue.vip.p6.delegate;

/**
 * @Description 调用客户端
 * @Author 向寒 奈学教育
 * @Date 2020/7/21 14:35
 **/
public class TaskClient {

    public static void main(String[] args) {
        new TaskDelegate().doTask();
    }
}
```

spring实践



Spring MVC框架中的DispatcherServlet相当于委派者，其他的解析器相当执行者。

- DispatcherServlet的委托流程

用户发送请求——>DispatcherServlet，前端控制器收到请求后自己不进行处理，而是委托给其他的解析器进行处理，作为统一访问点，进行全局的流程控制。

DispatcherServlet——>HandlerMapping，映射处理器将会把请求映射为HandlerExecutionChain对象（包含一个Handler处理器（页面控制器）对象、多个HandlerInterceptor拦截器）对象。

DispatcherServlet——>HandlerAdapter，处理器适配器将会把处理器包装为适配器，从而支持多种类型的处理器，即适配器设计模式的应用，从而很容易支持很多类型的处理器。

DispatcherServlet——> ViewResolver， 视图解析器将把ModelAndView对象（包含模型数据、逻辑视图名）解析为具体的View。

DispatcherServlet——>View，View会根据传进来的Model模型数据进行渲染，此处的Model实际是一个Map数据结构。

返回控制权给DispatcherServlet，由DispatcherServlet返回响应给用户，到此一个流程结束。