

程序开发中常用的设计模式

一、概况

总体来说设计模式分为三大类：

- （1）创建型模式，共五种：**工厂方法模式**、**抽象工厂模式**、**单例模式**、**建造者模式**、**原型模式**。
- （2）结构型模式，共七种：**适配器模式**、**装饰器模式**、**代理模式**、**外观模式**、**桥接模式**、**组合模式**、**享元模式**。
- （3）行为型模式，共十一种：**策略模式**、**模板方法模式**、**观察者模式**、**迭代子模式**、**责任链模式**、**命令模式**、**备忘录模式**、**状态模式**、**访问者模式**、**中介者模式**、**解释器模式**。

二、设计模式的六大原则

1、单一职责原则 (Single Responsibility Principle, SRP)

一个类只负责一个功能领域中的相应职责，或者可以定义为：就一个类而言，应该只有一个引起它变化的原因。（高内聚）

2、开放关闭原则 (Open Close Principle)

开闭原则就是说**对扩展开放，对修改关闭**。在程序需要进行拓展的时候，不能去修改原有的代码，实现一个热插拔的效果。

3、里氏替换原则 (Liskov Substitution Principle)

其官方描述比较抽象，可自行百度。实质上就是说任何使用基类的地方，都能够使用子类替换，而且在替换子类后，系统能够正常工作。实际上可以这样理解：（1）子类的能力必须大于等于父类，即父类可以使用的方法，子类都可以使用。（2）返回值也是同样的道理。假设一个父类方法返回一个 `List`，子类返回一个 `ArrayList`，这当然可以。如果父类方法返回一个 `ArrayList`，子类返回一个 `List`，就说不通了。这里子类返回值的能力是比父类小的。（3）还有抛出异常的情况。任何子类方法可以声明抛出父类方法声明异常的子类，而不能声明抛出父类没有声明的异常。

4、依赖倒转原则 (Dependence Inversion Principle)

这个是开闭原则的基础，具体内容：面向接口编程，依赖于抽象而不依赖于具体。

5、接口隔离原则 (Interface Segregation Principle)

这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度的意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和维护方便。所以上文中多次出现：降低依赖，降低耦合。

6、迪米特法则（最少知道原则）(Demeter Principle)

为什么叫最少知道原则，就是说：一个实体应当尽量少的与其他实体之间发生相互作用，使得系统功能模块相对独立。

7、合成复用原则 (Composite Reuse Principle)

原则是尽量使用合成/聚合的方式，而不是使用继承。

三、创建型模式

3.1、单例模式

Singleton 模式主要作用是保证在 Java 应用程序中，一个类 **Class** 只有一个实例存在。让类自身负责保存它的唯一实例。这个类可以保证没有其他实例可以被创建，并且它可以提供一个访问该实例的方法。

1、私有化类的构造函数，防止类外部的代码创建该类的实例。

所有类都有构造方法，不编码则系统默认生成空的构造方法，若有显示定义的构造方法，默认的构造方法机会失效。

2、有一个可以获取该类实例的公共方法，该方法要是静态的。

单例模式也有常见的两种模式：

3.1.1.1、饿汉式单例

```
/**
 * @Description: 单例模式，饿汉式
 * @author: zxt
 * @time: 2018年3月13日 下午3:31:29
 */
public class Singleton2 {
    // 本类类型的私有成员变量，在初始化时即创建类的实例，会提前占用系统资源，不过更加安全
    private static Singleton2 instance = new Singleton2();

    // 构造方法，设计成私有的，防止外部代码的访问，只允许内部构造
    private Singleton2() {
        // 构造逻辑
    }

    // 提供了一个供外部访问本class的静态方法，返回本类的一个实例
    public static Singleton2 getInstance() {
        return instance;
    }
}
```

这种方式静态初始化自己的唯一实例，需要提前占用系统资源，但是多线程访问时较安全。

3.1.2、懒汉式单例

```
/**
 * @Description: 单例模式，懒汉式
 * @author: zxt
 * @time: 2018年3月13日 下午3:13:51
 */
public class Singleton {
    // 本类类型的私有成员变量
    private static Singleton instance;

    // 构造方法，设计成私有的，防止外部代码的访问，只允许内部构造
    private Singleton() {
        // 构造逻辑
    }

    // 提供了一个供外部访问本class的静态方法，返回本类的一个实例
    // 由于本类的构造方法私有化，外部无法调用，所以访问接口必须设计成静态的，通过类名即可访问
    public static Singleton getInstance() {
        // 在第一次使用时才生成实例称为懒汉式单例，效率更好，但是存在多线程访问不安全的问题
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    // 多线程同时访问时，可能会创建多个实例，此时需要使用锁来保证实例唯一
    public synchronized static Singleton getInstance2() {
        // 对方法进行同步，会影响性能，所以需要考虑同步代码块，来改良
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }

    // 双重锁定
    public static Singleton getInstance3() {
        if (instance == null) {

            synchronized (Singleton.class) {
                // 二次检查保证同时判断instance为空的进程只有一个进入同步代码区
                if (instance == null) {
                    instance = new Singleton();
                }
            }
        }
    }
}
```

```
    }  
    }  
    return instance;  
}  
}
```

懒汉式单例是在类实例第一次访问的时候创建，这样效率更高，但是当多线程同时访问时，可能造成创建多个实例。所以需要使用锁，设置临界区代码，当有一个线程位于代码的临界区时，另一个线程不进入临界区。如果其他线程试图进入锁定的代码，则它将一直等待（即被阻止），直至该对象被释放。

其他实现单例的方法还有：静态内置类，`static` 代码块，枚举等。

3.2、简单工厂模式

简单工厂模式又叫静态工厂方法模式（Static Factory Method Pattern），是通过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。

一个简单的实例：要求实现一个计算机控制台程序，要求输入数的运算结果。最原始的解决方法如下：

```
/**
 * @Description:这里使用的是最基本的实现，并没有体现出面向对象的编程思想，代码的扩展性差，
 甚至连除数可能为0的情况也没有考虑
 */
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    System.out.print("请输入第一个数字: ");
    int firstNum = scanner.nextInt();

    System.out.print("请输入第二个数字: ");
    int secondNum = scanner.nextInt();

    System.out.print("请输入运算符: ");
    String operation = scanner.next();

    if(operation.equals("+")) {
        System.out.println("result: " + (firstNum + secondNum));
    } else if(operation.equals("-")) {
        System.out.println("result: " + (firstNum - secondNum));
    } else if(operation.equals("*")) {
        System.out.println("result: " + (firstNum * secondNum));
    } else if(operation.equals("/")){
        System.out.println("result: " + (firstNum / secondNum));
    }
}
```

上面的写法实现虽然简单，但是却没有面向对象的特性，代码拓展性差，甚至没有考虑除数可能为 0 的特殊情况。

在面向对象编程语言中，一切都是对象，所以上面运算符号也应当作对象来处理。因此我们首先建立一个运算接口，所有其他的运算都封装成类，并实现该运算接口。

```
/**
 * @Description: 定义一个运算接口，将所有的运算符号都封装成类，并实现本接口
 * @author: zxt
```

```

* @time: 2018年7月6日 上午10:24:13
*/
public interface Operation {
    public double getResult(double firstNum, double secondNum);
}

public class AddOperation implements Operation {
    @Override
    public double getResult(double firstNum, double secondNum) {
        return firstNum + secondNum;
    }
}

public class SubOperation implements Operation {
    @Override
    public double getResult(double firstNum, double secondNum) {
        return firstNum - secondNum;
    }
}

public class MulOperation implements Operation {
    @Override
    public double getResult(double firstNum, double secondNum) {
        return firstNum * secondNum;
    }
}

public class DivOperation implements Operation {
    @Override
    public double getResult(double firstNum, double secondNum) {
        if(secondNum == 0) {
            try {
                throw new Exception("除数不能为0! ");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        return firstNum / secondNum;
    }
}

```

现在的问题是，如何根据不同的情况创建不同的对象，这里就可以使用简单工厂模式来实现了，客户端只需要提供运算符，工厂类会判断并生成相应的运算类：

```

/**
 * @Description: 简单工厂模式: 通过一个工厂类, 根据情况创建不同的对象
 * @author: zxt
 * @time: 2018年7月6日 上午10:50:15
 */
public class OperationFactory {
    /**
     * @Description: 根据运算符得到具体的运算类
     * @param operationStr
     */
    public static Operation getOperation(String operationStr) {
        Operation result = null;

        switch(operationStr) {
            case "+":
                result = new AddOperation();
                break;
            case "-":
                result = new SubOperation();
                break;
            case "*":
                result = new MulOperation();
                break;
            case "/":
                result = new DivOperation();
                break;
        }

        return result;
    }
}

// 客户端调用
Operation oper = OperationFactory.getOperation(operation);
double result = oper.getResult(firstNum, secondNum);
System.out.println(result);

```

简单工厂将对象的创建过程进行了封装, 用户不需要知道具体的创建过程, 只需要调用工厂类获取对象即可。

这种简单工厂的写法是通过 **switch-case** 来判断对象创建过程的。在实际使用过程中, 违背了**开放-关闭原则**(例如, 当需要扩展一个新的运算符之后, 简单工厂创建的对象也必须多一种, 这就需要修改原来的代码了, 违背了对修改关闭的原则), 当然有些情况下可以通过反射调用来弥补这种不足。

3.3、工厂方法模式

简单工厂模式的最大优点在于工厂类中包含了必要的逻辑判断，根据客户端的选择条件动态实例化相关的类，对于客户端来说，去除了与具体产品的依赖。但是每扩展一个类时，都需要改变工厂类里的方法，这就违背了开放-封闭原则。于是工厂方法模式来了：

工厂方法模式（**Factory Method**），定义一个用于创建对象的接口，让子类决定实例化哪一个类，工厂方法使一个类的实例化延迟到其子类。继续上一个计算器的例子，简单工厂模式由工厂类直接生成相应的运算类对象，判断的逻辑在工厂类中，而工厂方法模式的实现则是定义一个工厂接口，然后每个运算类都对应一个工厂类来创建，然后在客户端判断使用哪个工厂类来创建运算类。

```
/**
 * @Description: 工厂的接口
 * @author: zxt
 * @time: 2019年2月21日 下午2:49:43
 */
public interface IFactory {

    public Operation createOperation();
}

/**
 * @Description: 加法类工厂
 */
public class AddFactory implements IFactory {
    @Override
    public AddOperation createOperation() {
        return new AddOperation();
    }
}

/**
 * @Description: 减法类工厂
 */
public class SubFactory implements IFactory {
    @Override
    public SubOperation createOperation() {
        return new SubOperation();
    }
}
```

```

/**
 * @Description: 乘法类工厂
 */
public class MulFactory implements IFactory {
    @Override
    public MulOperation createOperation() {
        return new MulOperation();
    }
}

/**
 * @Description: 除法类工厂
 */
public class DivFactory implements IFactory {

    @Override
    public DivOperation createOperation() {
        return new DivOperation();
    }
}

```

工厂方法模式实现时，客户端需要决定实例化哪一个工厂来实现运算类，选择判断的问题还是存在的，也就是说，工厂方法把简单工厂的内部逻辑判断移到了客户端代码来进行。你想要加功能，本来是改工厂类的，而现在是修改客户端。

```

/**
 * @Description: 实现一个简单的计算器功能，使用工厂方法模式
 * @author: zxt
 * @time: 2018年7月6日 上午10:11:50
 */
public class Computer {

    private static Scanner scanner;

    public static void main(String[] args) {
        scanner = new Scanner(System.in);

        System.out.print("请输入第一个数字: ");
        int firstNum = scanner.nextInt();

        System.out.print("请输入第二个数字: ");
        int secondNum = scanner.nextInt();

        System.out.print("请输入运算符: ");
        String operation = scanner.next();
    }
}

```

```
IFactory operFactory = null;
if(operation.equals("+")) {
    operFactory = new AddFactory();
} else if(operation.equals("-")) {
    operFactory = new SubFactory();
} else if(operation.equals("*")) {
    operFactory = new MulFactory();
} else if(operation.equals("/")){
    operFactory = new DivFactory();
}
Operation oper = operFactory.createOperation();
double result = oper.getResult(firstNum, secondNum);
System.out.println("result = " + result);
}
}
```

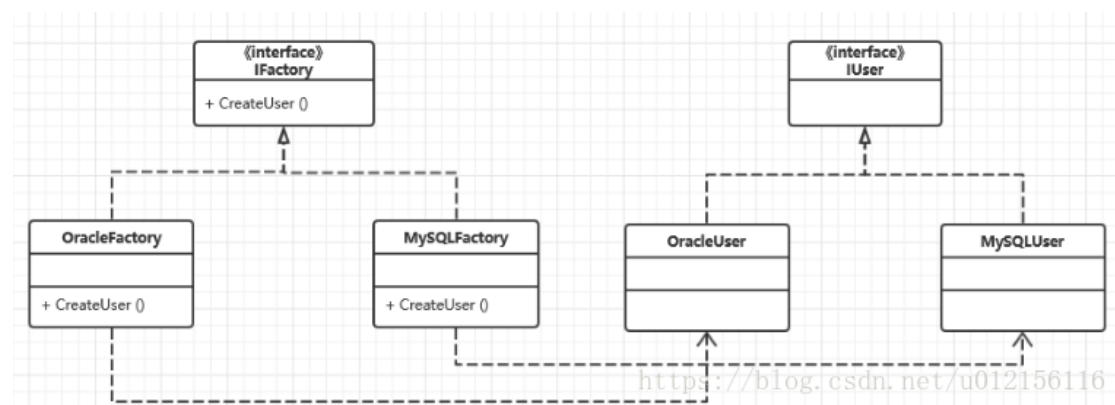
增加新功能时，工厂方法模式比简单工厂模式修改的代码量更小，工厂方法克服了简单工厂违背开放封闭原则的缺点，又保持了封装对象创建过程的优点。但是工厂方法的缺点就是每加一个产品，就需要加一个产品工厂的类，增加了额外的开发量。当然这两种模式都还不是最佳的做法。

3.4、抽象工厂模式

抽象工厂模式是所有形态的工厂模式中最为抽象和最具一般性的一种形态。抽象工厂模式是指当有多个抽象角色时，使用的一种工厂模式。抽象工厂模式可以向客户端提供一个接口，使客户端在不必指定产品的具体的情况下，创建多个产品族中的产品对象。根据里氏替换原则，任何接受父类型的地方，都应当能够接受子类型。因此，实际上系统所需要的，仅仅是类型与这些抽象产品角色相同的一些实例，而不是这些抽象产品的实例。换言之，也就是这些抽象产品的具体子类的实例。工厂类负责创建抽象产品的具体子类的实例。

抽象工厂模式（Abstract Factory）：提供一个创建一系列相关或相互依赖对象的接口，而无需指定他们具体的类。

实例场景：对数据库（各种不同的数据库）中的表进行修改，此时，使用工厂模式结构图如下：



1、User 表的定义：

```
public class User {
    private int id;
    private String name;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }
}
```

```

    public void setName(String name) {
        this.name = name;
    }
}

```

2、定义一个对 User 表进行操作的接口：

```

/**
 * @Description: 对User类操作的接口
 * @author: zxt
 * @time: 2019年2月24日 下午7:00:20
 */
public interface IUser {

    void insert(User user);

    User getUser(int id);
}

```

3、实现 Sql Server 数据库对 User 表的操作：

```

/**
 * @Description: SQL Server数据库中对User表的操作
 * @author: zxt
 * @time: 2019年2月24日 下午7:04:39
 */
public class SqlServerUser implements IUser {
    @Override
    public void insert(User user) {
        System.out.println("在 SQL Server 中给 User 表增加一条记录!");
    }

    @Override
    public User getUser(int id) {
        System.out.println("在 SQL Server 中根据ID得到 User 表的一条记录!");
        return null;
    }
}

```

实现 Oracle 数据库对 User 表的操作：

```

/**
 * @Description: Oracle数据库中对User表的操作
 * @author: zxt
 * @time: 2019年2月24日 下午7:05:07
 */
public class OracleUser implements IUser {

```

```

@Override
public void insert(User user) {
    System.out.println("在 Oracle 中给 User 表增加一条记录!");
}

@Override
public User getUser(int id) {
    System.out.println("在 Oracle 中根据ID得到 User 表的一条记录!");
    return null;
}
}

```

4、定义一个抽象工厂接口，用于生成对 User 表的操作的对象：

```

/**
 * @Description: 得到对User表操作的IUser对象的抽象工厂接口
 * @author: zxt
 * @time: 2019年2月24日 下午7:06:37
 */
public interface IFactory {

    public IUser createUser();
}

```

5、SQLServerFactory 工厂用于生成操作 Sql Server 数据库的 SqlServerUser 对象：

```

public class SQLServerFactory implements IFactory {

    @Override
    public IUser createUser() {
        return new SqlServerUser();
    }
}

```

OracleFactory 工厂用于生成操作 Oracle 数据库的 OracleUser 对象：

```

public class OracleFactory implements IFactory {

    @Override
    public IUser createUser() {
        return new OracleUser();
    }
}

```

6、客户端的使用：

```

public class FactoryMethodTest {

    public static void main(String[] args) {
        User user = new User();
    }
}

```

```

// 若要改成Oracle数据库，只需要将这句改成OracleFactory即可
IFactory ifactory = new SQLServerFactory();

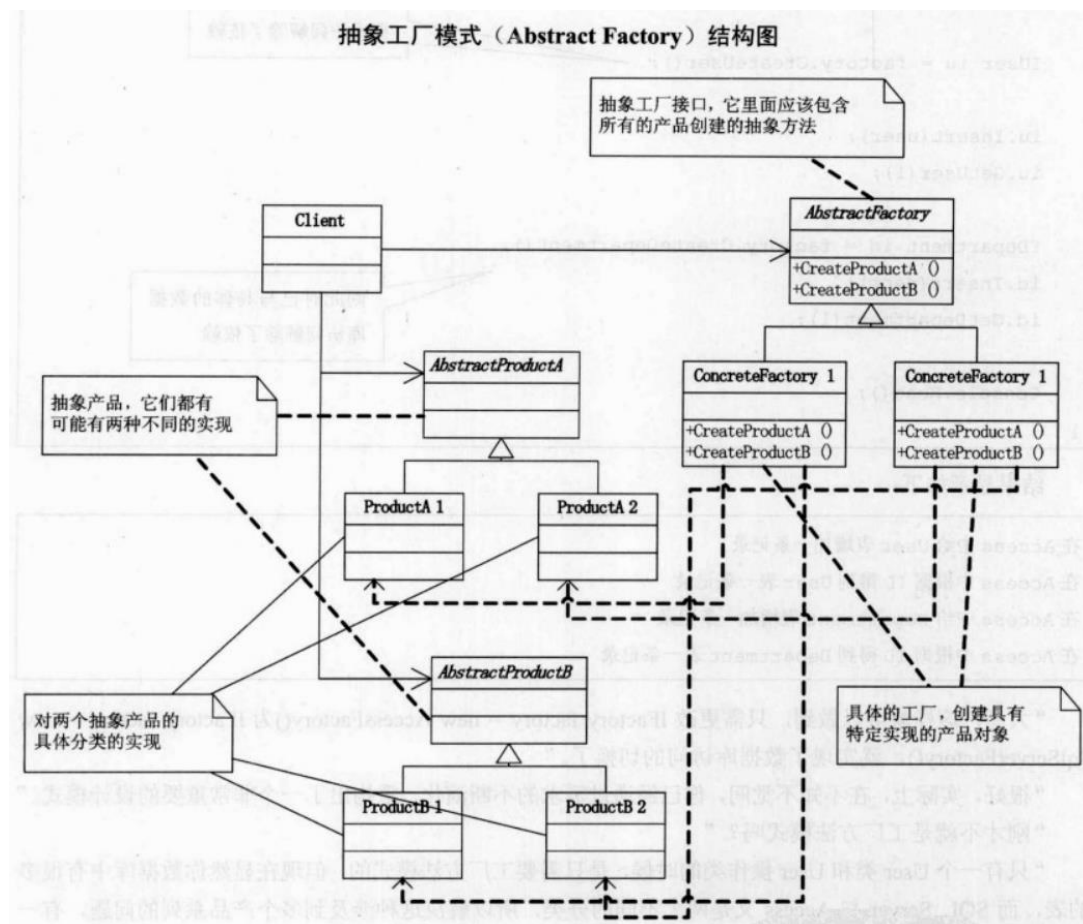
IUser iu = ifactory.createUser();

iu.insert(user);
iu.getUser(1);
}
}

```

到此为止，工厂模式都可以很好的解决，由于多态的关系，**IFactory** 在声明对象之前都不知道在访问哪个数据库，却可以在运行时很好的完成任务，这就是业务逻辑与数据访问的解耦。

但是，当数据库中不止一个表的时候该怎么解决问题呢，此时就可以引入抽象工厂模式了，结构图如下：



例如增加了部门表 Department:

```
public class Department {
    private int id;
    private String deptName;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getDeptName() {
        return deptName;
    }
    public void setDeptName(String deptName) {
        this.deptName = deptName;
    }
}
```

则增加相应的对 Department 表操作的接口:

```
public interface IDepartment {

    public void insert(Department department);

    public Department getDepartment(int id);
}
```

实现 Sql Server 数据库对 Department 表的操作:

```
/**
 * @Description: SQL Server数据库中对Department表的操作
 * @author: zxt
 * @time: 2019年2月24日 下午7:04:39
 */
public class SqlServerDepartment implements IDepartment {
    @Override
    public void insert(Department user) {
        System.out.println("在 SQL Server 中给 Department 表增加一条记录!");
    }

    @Override
    public Department getDepartment(int id) {
        System.out.println("在 SQL Server 中根据ID得到 Department 表的一条记录!");
        return null;
    }
}
```


实现 Oracle 数据库对 Department 表的操作：

```
/**
 * @Description: Oracle数据库中Department表的操作
 * @author: zxt
 * @time: 2019年2月24日 下午7:05:07
 */
public class OracleDepartment implements IDepartment {
    @Override
    public void insert(Department user) {
        System.out.println("在 Oracle 中给 Department 表增加一条记录!");
    }

    @Override
    public Department getDepartment(int id) {
        System.out.println("在 Oracle 中根据ID得到 Department 表的一条记录!");
        return null;
    }
}
```

IFactory 抽象工厂中增加生成对 Department 表操作的对象：

```
/**
 * @Description: 得到对User表操作的IUser对象的抽象工厂接口
 * @author: zxt
 * @time: 2019年2月24日 下午7:06:37
 */
public interface IFactory {

    public IUser createUser();

    public IDepartment createDepartment();
}
```

SQLServerFactory 工厂增加生成操作 Sql Server 数据库的 SqlServerDepartment 对象：

```
public class SQLServerFactory implements IFactory {
    @Override
    public IUser createUser() {
        return new SqlServerUser();
    }

    @Override
    public IDepartment createDepartment() {
        return new SqlServerDepartment();
    }
}
```

```
}
```

OracleFactory 工厂增加生成操作 Oracle 数据库的 OracleDepartment 对象:

```
public class OracleFactory implements IFactory {  
    @Override  
    public IUser createUser() {  
        return new OracleUser();  
    }  
  
    @Override  
    public IDepartment createDepartment() {  
        return new OracleDepartment();  
    }  
}
```

客户端的使用:

```
public class AbstractFactoryTest {  
  
    public static void main(String[] args) {  
        User user = new User();  
        Department department = new Department();  
  
        // 若要改成SQL Server数据库, 只需要将这句改成SqlServerFactory即可  
        IFactory ifactory = new OracleFactory();  
  
        IUser iu = ifactory.createUser();  
        iu.insert(user);  
        iu.getUser(1);  
  
        IDepartment id = ifactory.createDepartment();  
        id.insert(department);  
        id.getDepartment(1);  
    }  
}
```

Problems Console @ Javadoc JUnit
<terminated> AbstractFactoryTest [Java Application] D:\Program Java\jre1.8.0_6

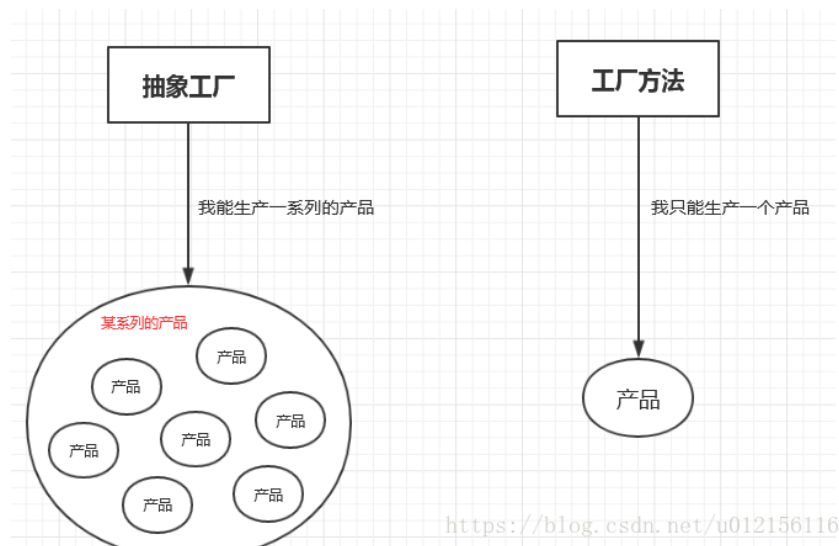
在 Oracle 中给 User 表增加一条记录!

在 Oracle 中根据ID得到 User 表的一条记录!

在 Oracle 中给 Department 表增加一条记录!

在 Oracle 中根据ID得到 Department 表的一条记录!

所以抽象工厂与工厂方法模式的区别在于：抽象工厂是可以生产多个产品的，例如 `OracleFactory` 里可以生产 `OracleUser` 以及 `OracleDepartment` 两个产品，而这两个产品又是属于一个系列的，因为它们都是属于 `Oracle` 数据库的表。而工厂方法模式则只能生产一个产品，例如之前的 `OracleFactory` 里就只可以生产一个 `OracleUser` 产品。



抽象工厂模式的优缺点：

优点：

1、抽象工厂模式最大的好处是易于交换产品系列，由于具体工厂类，例如 `IFactory factory = new OracleFactory();` 在一个应用中只需要在初始化的时候出现一次，这就使得改变一个应用的具体工厂变得非常容易，它只需要改变具体工厂即可使用不同的产品配置。不管是任何人的设计都无法去完全防止需求的更改，或者项目的维护，那么我们的理想便是让改动变得最小、最容易。

2、抽象工厂模式的另一个好处就是它让具体的创建实例过程与客户端分离，客户端是通过它们的抽象接口操作实例，产品实现类的具体类名也被具体的工厂实现类分离，不会出现在客户端代码中。就像我们上面的例子，客户端只认识 `IUser` 和 `IDepartment`，至于它是 `Sql Server` 里的表还是 `Oracle` 里的表就不知道了。

缺点：

1、如果你的需求来自增加功能，比如增加 `Department` 表，就有点太烦了。首先需要增加 `IDepartment`，`SQLServerDepartment`，`OracleDepartment`。然后我们还要去修改工厂类：`IFactory`，`SQLServerFactory`，`OracleFactory` 才可以实现，需要修改三个类，实在是有点麻烦。

2、还有就是，客户端程序肯定不止一个，每次都需要声明 `IFactory factory = new`

OracleFactory(), 如果有 100 个调用数据库的类, 就需要更改 100 次 IFactory factory = new OracleFactory()。

3.4.1、抽象工厂模式的改进 1 (简单工厂+抽象工厂)

我们将 IFactory, SQLServerFactory, OracleFactory 三个工厂类都抛弃掉, 取而代之的是一个简单工厂类 EasyFactory, 如下:

```
public class EasyFactory {

    private static String db = "SqlServer";
    // private static String db = "Oracle";

    public static IUser createUser() {
        IUser result = null;
        switch (db) {
            case "SqlServer":
                result = new SqlServerUser();
                break;
            case "Oracle":
                result = new OracleUser();
                break;
        }

        return result;
    }

    public static IDepartment createDepartment() {
        IDepartment result = null;
        switch (db) {
            case "SqlServer":
                result = new SqlServerDepartment();
                break;
            case "Oracle":
                result = new OracleDepartment();
                break;
        }

        return result;
    }
}
```

客户端:

```
public class EasyClient {

    public static void main(String[] args) {
        User user = new User();
        Department department = new Department();

        // 直接得到实际的数据库访问实例，而不存在任何依赖
        IUser userOperation = EasyFactory.createUser();

        userOperation.getUser(1);
        userOperation.insert(user);

        // 直接得到实际的数据库访问实例，而不存在任何依赖
        IDepartment departmentOperation = EasyFactory.createDepartment();

        departmentOperation.insert(department);
        departmentOperation.getDepartment(1);
    }
}
```

由于事先在简单工厂类里设置好了 db 的值，所以简单工厂的方法都不需要由客户端来输入参数，这样在客户端就只需要使用 EasyFactory.createUser(); 和 EasyFactory.createDepartment(); 方法来获得具体的数据库访问类的实例，客户端代码上没有出现任何一个 SqlServer 或 Oracle 的字样，达到了解耦的目的，客户端已经不再受改动数据库访问的影响了。

3.4.2、抽象工厂的改进 2（反射+简单工厂）

使用反射的话，我们就可以不需要使用 switch，因为使用 switch 的话，我添加一个 Mysql 数据库的话，又要 switch 的话又需要添加 case 条件。

我们可以根据选择的数据库名称，如“mysql”，利用反射技术自动的获得所需要的实例：

```
public class EasyFactoryReflect {

    private static String packName = "com.zxt.abstractfactory";
    private static String sqlName = "Oracle";

    public static IUser createUser() throws Exception {
```

```

        String className = packName + "." + sqlName + "User";
        return (IUser) Class.forName(className).newInstance();
    }

    public static IDepartment createLogin() throws Exception {
        String className = packName + "." + sqlName + "Department";
        return (IDepartment) Class.forName(className).newInstance();
    }
}

```

以上我们使用简单工厂模式设计的代码中，是用一个字符串类型的 `db` 变量来存储数据库名称的，所以变量的值到底是 `SqlServer` 还是 `Oracle`，完全可以由事先设置的那个 `db` 变量来决定，而我们又可以通过反射来去获取实例，这样就可以去除 `switch` 语句了。

3.4.3、抽象工厂的改进3（反射+配置文件+简单工厂）

在使用反射之后，我们还是需要进 `EasyFactory` 中修改数据库类型，还不是完全符合开-闭原则。我们可以通过配置文件来达到目的，每次通过读取配置文件来知道我们应该使用哪种数据库。

如下是一个 `json` 类型的配置文件，也可以使用 `xml` 类型的配置文件：

```

{
    "packName": " com.zxt.abstractfactory",
    "DB": "Oracle"
}

```

之后就可以通过这个配置文件去找需要加载的类是哪一个。我们通过反射机制+配置文件+简单工厂模式解决了数据库访问时的可维护、可扩展的问题。

四、结构型模式

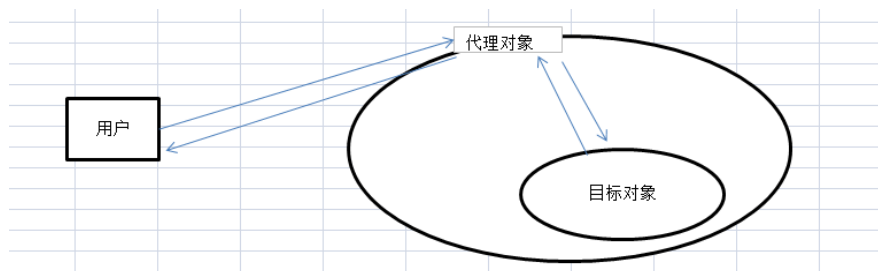
4.1、代理模式

代理模式的定义：给某一个对象提供一个代理，并由代理对象控制对原对象的引用。

代理(Proxy)是一种设计模式，提供了对目标对象另外的访问方式；即通过代理对象访问目标对象。这样做的好处是：可以在目标对象实现的基础上，增强额外的功能操作，即扩展目标对象的功能。

这里使用到编程中的一个思想：**不要随意去修改别人已经写好的代码或者方法**，如果需要修改，可以通过代理的方式来扩展该方法。

举个例子来说明代理的作用：假设我们想邀请一位明星，那么并不是直接联系明星，而是联系明星的经纪人，来达到同样的目的。明星就是一个目标对象，他只要负责活动中的节目，而其他琐碎的事情就交给他的代理人(经纪人)来解决。这就是代理思想在现实中的一个例子。



代理模式的关键点是：代理对象与目标对象。**代理对象是对目标对象的扩展**，并会调用目标对象。

代理模式包含如下角色：

ISubject：抽象主题角色，是一个接口。该接口是对象和它的代理共用的接口。

RealSubject：真实主题角色，是实现抽象主题接口的类。

Proxy：代理角色，内部含有对真实对象 **RealSubject** 的引用，从而可以操作真实对象。代理对象提供与真实对象相同的接口，以便在任何时刻都能代替真实对象。同时，代理对象可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。

代理模式的应用：

远程代理：也就是为一个对象在不同的地址空间提供局部代表。这样可以隐藏一个对象存在于不同地址空间的事实。

虚拟代理：是根据需要创建开销很大的对象。通过它来存放实例化需要很长时间的真实对象。

安全代理：用来控制真实对象访问时的权限。

智能代理：是指当调用真实的对象时，代理处理一些另外的事情。

一般将代理分类的话，可分为静态代理和动态代理两种。

4.1.1.1、静态代理

静态代理比较简单，是由程序员编写的代理类，并在程序运行前就编译好的，而不是由程序动态产生代理类，这就是所谓的静态。

考虑这样的场景，管理员在网站上执行操作，在生成操作结果的同时需要记录操作日志，这是很常见的。此时就可以使用代理模式，代理模式可以通过聚合和继承两种方式实现：

```
/**
 * @Description: 抽象主题接口
 * @author: zxt
 * @time: 2018年7月7日 下午2:29:46
 */
public interface Manager {
    public void doSomething();
}

/**
 * @Description: 真实的主题类
 * @author: zxt
 * @time: 2018年7月7日 下午2:31:21
 */
public class Admin implements Manager {
    @Override
    public void doSomething() {
        System.out.println("这是真实的主题类: Admin doSomething!!!");
    }
}

/**
 * @Description: 以聚合的方式实现代理主题
 * @author: zxt
 * @time: 2018年7月7日 下午2:37:08
 */
public class AdminPoly implements Manager {
    // 真实主题类的引用
```



```

private Admin admin;

public AdminPoly(Admin admin) {
    this.admin = admin;
}

@Override
public void doSomething() {
    System.out.println("聚合方式实现代理: Admin操作开始!!");
    admin.doSomething();
    System.out.println("聚合方式实现代理: Admin操作结束!!");
}
}

/**
 * @Description: 继承方式实现代理
 * @author: zxt
 * @time: 2018年7月7日 下午2:40:39
 */
public class AdminProxy extends Admin {
    @Override
    public void doSomething() {
        System.out.println("继承方式实现代理: Admin操作开始!!");
        super.doSomething();
        System.out.println("继承方式实现代理: Admin操作结束!!");
    }
}

public static void main(String[] args) {
    // 1、聚合方式的测试
    Admin admin = new Admin();
    Manager manager = new AdminPoly(admin);
    manager.doSomething();

    System.out.println("=====");
    // 2、继承方式的测试
    AdminProxy proxy = new AdminProxy();
    proxy.doSomething();
}

```

聚合实现方式中代理类聚合了被代理类，且代理类及被代理类都实现了同一个接口，可实现灵活多变。继承式的实现方式则不够灵活。

比如，在管理员操作的同时需要进行权限的处理，操作内容的日志记录，操作后数据的变化三个功能。三个功能的排列组合有 6 种，也就是说使用继承要编写 6 个继承了 Admin

的代理类，而使用聚合，仅需要针对权限的处理、日志记录和数据变化三个功能编写代理类，在业务逻辑中根据具体需求改变代码顺序即可。

缺点：

1)、代理类和委托类实现了相同的接口，代理类通过委托类实现了相同的方法。这样就出现了大量的代码重复。如果接口增加一个方法，除了所有实现类需要实现这个方法外，所有代理类也需要实现此方法。增加了代码维护的复杂度。

2)、代理对象只服务于一种类型的对象，如果要服务多类型的对象。势必要为每一种对象都进行代理，静态代理在程序规模稍大时就无法胜任了。

4.1.2、动态代理

实现动态代理的关键技术是反射。

一般来说，对代理模式而言，一个主题类与一个代理类一一对应，这也是静态代理模式的特点。

但是，也存在这样的情况，有 n 各主题类，但是代理类中的“前处理、后处理”都是一样的，仅调用主题不同。也就是说，多个主题类对应一个代理类，共享“前处理，后处理”功能，动态调用所需主题，大大减小了程序规模，这就是动态代理模式的特点。动态代理主要有两种：JDK 自带的动态代理和 CGLIB 动态代理。

首先是另一个静态代理的实例：

1、一个可移动接口

```
public interface Moveable {  
  
    public void move();  
}
```

2、一个实现了该接口的 Car 类

```
public class Car implements Moveable {  
  
    @Override  
    public void move() {  
        try {  
            Thread.sleep(new Random().nextInt(1000));  
            System.out.println("汽车行驶中----");  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```

    }
}
}

```

3、现在需要有一个代理类来记录 Car 的运行时间：

```

public class CarTimeProxy implements Moveable {
    private Moveable m;

    public CarTimeProxy(Moveable m) {
        super();
        this.m = m;
    }

    @Override
    public void move() {
        long startTime = System.currentTimeMillis();
        System.out.println("汽车行驶前----");
        m.move();
        long endTime = System.currentTimeMillis();
        System.out.println("汽车行驶结束----行驶时间为: " + (endTime - startTime) + "
毫秒!");
    }
}

```

4、另一个代理类记录 Car 的日志：

```

public class CarLogProxy implements Moveable {
    private Moveable m;

    public CarLogProxy(Moveable m) {
        super();
        this.m = m;
    }

    @Override
    public void move() {
        System.out.println("日志开始");
        m.move();
        System.out.println("日志结束");
    }
}

```

5、客户端的调用：

```

public class CarTest {

    public static void main(String[] args) {
        Car car = new Car();
    }
}

```

```

// 先写日志，再计时
CarTimeProxy ctp = new CarTimeProxy(car);
CarLogProxy clp = new CarLogProxy(ctp);
clp.move();

System.out.println();
// 先计时，再写日志
CarLogProxy clp1 = new CarLogProxy(car);
CarTimeProxy ctp1 = new CarTimeProxy(clp1);
ctp1.move();
}
}

```

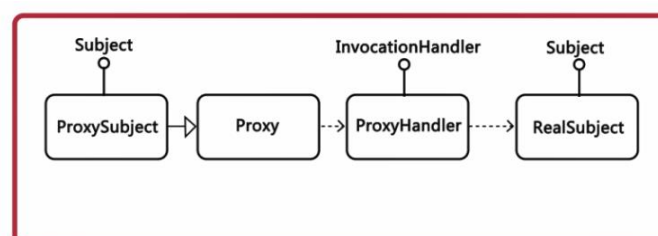
Problems Console @ Javadoc JUnit
 <terminated> CarTest [Java Application] D:\Program Java\jre1.8.0_60\

日志开始
 汽车行驶前 ----
 汽车行驶中 ----
 汽车行驶结束 ---- 行驶时间为：420毫秒!
 日志结束

汽车行驶前 ----
 日志开始
 汽车行驶中 ----
 日志结束
 汽车行驶结束 ---- 行驶时间为：977毫秒!

4.1.2.1、JDK 的动态代理

在 java 的动态代理机制中，有两个重要的类或接口，一个是 `InvocationHandler(Interface)`、另一个则是 `Proxy(Class)`，这一个类和接口是实现我们动态代理所必须用到的。

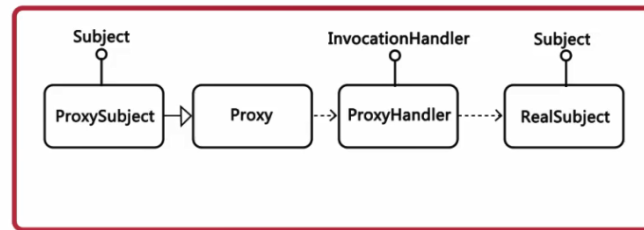


Java动态代理类位于`java.lang.reflect`包下，一般主要涉及到以下两个类：

(1)Interface `InvocationHandler`：该接口中仅定义了一个方法

`public Object invoke(Object obj,Method method, Object[] args)`

在实际使用时，第一个参数obj一般是指代理类，method是被代理的方法，args为该方法的参数数组。这个抽象方法在代理类中动态实现。



(2)Proxy : 该类即为动态代理类

`static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h)` : 返回代理类的一个实例, 返回后的代理类可以当作被代理类使用(可使用被代理类的在接口中声明过的方法)

JDK 动态代理的实现

1、创建一个实现接口 `InvocationHandler` 的类, 它必须实现 `invoke` 方法。

使用 JDK 动态代理类时, 需要实现 `InvocationHandler` 接口, 所有动态代理类的方法调用, 都会交由 `InvocationHandler` 接口实现类里的 `invoke()` 方法去处理。这是动态代理的关键所在。

2、创建被代理的类以及接口。

3、调用 `Proxy` 的静态方法, 创建代理类。

`newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler h);`

4、通过代理调用方法。

使用 JDK 动态代理的方式实现上面 `Car` 的时间代理:

1、首先是 `InvocationHandler` 接口的实现类:

```

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class TimeHandler implements InvocationHandler {
    // 被传递过来的要被代理的对象
    private Object object;

    public TimeHandler(Object object) {
        super();
        this.object = object;
    }

    /**
     * proxy: 被代理的对象
     * method: 被代理的方法
  
```

```

    * args: 被代理方法的参数
    *
    * 函数返回: method的返回
    *
    */
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
{
    long startTime = System.currentTimeMillis();
    System.out.println("汽车行驶前----");
    method.invoke(object, args);
    long endTime = System.currentTimeMillis();
    System.out.println("汽车行驶结束----行驶时间为: " + (endTime - startTime) + "
毫秒!");
    return null;
}
}

```

2、创建动态代理类:

```

/**
 * @Description: JDK动态代理的测试类
 * @author: zxt
 * @time: 2019年3月1日 下午7:59:29
 */
public class TimeHandlerTest {

    public static void main(String[] args) {
        // 需要被代理的对象
        Car car = new Car();
        InvocationHandler h = new TimeHandler(car);

        Class<?> clazz = car.getClass();

        /**
         * 参数一: 类加载器
         * 参数二: 被代理类实现的接口
         * 参数三: InvocationHandler实例
         *
         * 函数返回: 返回由InvocationHandler接口接收的被代理类的一个动态代理类对象
         */
        Moveable m = (Moveable) Proxy.newProxyInstance(clazz.getClassLoader(),
clazz.getInterfaces(), h);
        m.move();
    }
}

```

4.1.2.2、cglib 动态代理

JDK 动态代理可以在运行时动态生成字节码，主要使用到了一个接口 `InvocationHandler` 与 `Proxy.newProxyInstance` 静态方法。使用内置的 `Proxy` 实现动态代理有一个问题：被代理的类必须要实现某接口，未实现接口则没办法完成动态代理。

如果项目中有些类没有实现接口，则不应该为了实现动态代理而刻意去抽象出一些没有实际意义的接口，通过 `cglib` 可以解决该问题。

`CGLIB`(Code Generation Library)是一个开源项目，是一个强大的，高性能，高质量的 Code 生成类库，它可以在运行期扩展 Java 类与实现 Java 接口，通俗地说 `cglib` 可以在运行时动态生成字节码。

使用 `cglib` 完成动态代理，大概的原理是：`cglib` 继承被代理的类，重写方法，织入通知，动态生成字节码并运行。对指定目标类产生一个子类，通过方法拦截技术拦截所有父类的方法调用，因为是继承实现所以 `final` 类是没有办法动态代理的。

CGLIB 动态代理实例：

```
import java.util.Random;

// 不实现接口的被代理类
public class Train {

    public void move() {
        try {
            Thread.sleep(new Random().nextInt(1000));
            System.out.println("火车行驶中----");
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
import java.lang.reflect.Method;

import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;

public class CglibProxy implements MethodInterceptor {
    private Enhancer enhancer = new Enhancer();
```

```

// 得到代理类的方法
public Object getProxy(Class<?> clazz) {
    // 设置创建子类的类 （即我们需要为哪个类产生代理类）
    enhancer.setSuperclass(clazz);
    enhancer.setCallback(this);

    return enhancer.create();
}

/**
 * 拦截所有目标类方法的调用
 *
 * object: 目标类的实例
 * method: 目标类的目标方法的反射实例
 * args: 目标方法的参数
 * proxy: 代理类的实例
 */
@Override
public Object intercept(Object object, Method method, Object[] args, MethodProxy proxy) throws Throwable {
    long startTime = System.currentTimeMillis();
    System.out.println("火车行驶前----");

    // 代理类调用父类的方法（由于Cglib动态代理的实现是通过继承被代理类，因此代理类这里需要调用父类的方法）
    proxy.invokeSuper(object, args);

    long endTime = System.currentTimeMillis();
    System.out.println("火车行驶结束----行驶时间为: " + (endTime - startTime) + "毫秒!");
    return null;
}
}

```

```

public class CglibProxyTest {

    public static void main(String[] args) {
        CglibProxy cglibProxy = new CglibProxy();
        Train train = (Train) cglibProxy.getProxy(Train.class);
        train.move();
    }
}

```


4.1.3、JDK 动态代理的模拟实现

模拟 JDK 动态代理的实现，根据 Java 源代码动态生成代理类。

```
package com.zxt.jdkproxy;

import java.io.File;
import java.lang.reflect.Constructor;
import java.lang.reflect.Method;

import javax.tools.JavaCompiler;
import javax.tools.JavaCompiler.CompilationTask;
import javax.tools.StandardJavaFileManager;
import javax.tools.ToolProvider;

import org.apache.commons.io.FileUtils;

import com.zxt.staticproxy.Car;

/**
 *
 * @Description: 模拟JDK动态代理的实现
 * 动态代理的实现思路：
 * 实现功能：通过自定义的Proxy的newProxyInstance方法返回代理对象
 * 1、声明一段源码（动态产生代理）
 * 2、编译源码（JDK Compiler API），产生新的类（代理类）
 * 3、将这个类load到内存当中，产生一个新的对象（代理对象）
 * 4、return 代理对象
 *
 * @author: zxt
 *
 * @time: 2019年4月18日 下午3:44:58
 */
public class MyProxy {

    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static Object newProxyInstance(Class<?> inteface) throws Exception {
        // 1、声明一段源码（动态产生代理）
        String rt = "\r\n";
        String methodStr = "";
        for(Method m : inteface.getMethods()) {
            methodStr += "    @Override" + rt
                + "    public void " + m.getName() + "() {" + rt
                + "        System.out.println(\"日志开始\");" + rt

```

```

        + "        m." + m.getName() + "();" + rt
        + "        System.out.println(\"日志结束\");" + rt
        + "    }";
    }
    String code =
        "package com.zxt.jdkproxy;" + rt + "\n"
        + "import com.zxt.staticproxy.Moveable;" + rt + "\n"
        + "public class $MyProxy0 implements " + interface.getSimpleName() + " {"
+ rt + "\n"
        + "    private " + interface.getSimpleName() + " m;" + rt + "\n"
        + "    public $MyProxy0(" + interface.getSimpleName() + " m) {" + rt
        + "        super();" + rt
        + "        this.m = m;" + rt
        + "    }" + rt + "\n"
        + methodStr + rt + "\n"
        + "}";

    // 由源代码生成java类文件
    String filename = System.getProperty("user.dir") +
"/bin/com/zxt/jdkproxy/$MyProxy0.java";
    File file = new File(filename);
    // 使用commons-io里面的简便的工具类来写文件
    FileUtils.writeStringToFile(file, code, "UTF-8");

    // 2、编译源码（JDK Compiler API），产生新的类（代理类）
    // 拿到编译器
    JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
    // 文件管理者
    StandardJavaFileManager fileManager = compiler.getStandardFileManager(null,
null, null);
    // 获取文件
    Iterable units = fileManager.getJavaFileObjects(filename);
    // 获取编译任务
    CompilationTask task = compiler.getTask(null, fileManager, null, null, null,
units);
    // 编译
    task.call();
    fileManager.close();

    // 3、加载到内存
    ClassLoader cl = ClassLoader.getSystemClassLoader();
    Class c = cl.loadClass("com.zxt.jdkproxy.$MyProxy0");

```

```

        // 4、返回代理类
        Constructor ctr = c.getConstructor(inteface);
        return ctr.newInstance(new Car());
    }

    public static void main(String[] args) {

    }
}

```

```

public class MyProxyTest {
    public static void main(String[] args) throws Exception {
        Moveable m = (Moveable) MyProxy.newProxyInstance(Moveable.class);
        m.move();
    }
}

```

可以发现上述实现中的源代码是写死在类中的，因此无法对任意类进行动态代理，所以仿照 `InvocationHandler` 接口，定义自己的 `InvocationHandler` 接口从而实现对不同的类进行动态代理。

```

import java.lang.reflect.Method;

public interface MyInvocationHandler {

    public void invoke(Object o, Method m);
}

```

实现该接口的类

```

import java.lang.reflect.Method;

public class MyLogHandler implements MyInvocationHandler {
    // 需要被代理的对象
    private Object target;

    public MyLogHandler(Object target) {
        super();
        this.target = target;
    }

    @Override
    public void invoke(Object o, Method m) {
        try {

```

```

        System.out.println("日志开始");
        m.invoke(target);
        System.out.println("日志结束");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
}

```

对动态代理 MyProxy 类进行改进

```

/**
 *
 * @Description: 模拟JDK动态代理的实现
 * 动态代理的实现思路:
 * 实现功能: 通过自定义的Proxy的newProxyInstance方法返回代理对象
 * 1、声明一段源码（动态产生代理）
 * 2、编译源码（JDK Compiler API），产生新的类（代理类）
 * 3、将这个类load到内存当中，产生一个新的对象（代理对象）
 * 4、return 代理对象
 *
 * @author: zxt
 *
 * @time: 2019年4月18日 下午3:44:58
 */
public class MyProxy {

    @SuppressWarnings({ "rawtypes", "unchecked" })
    public static Object newProxyInstance(Class<?> interface, MyInvocationHandler h)
    throws Exception {
        // 1、声明一段源码（动态产生代理）
        String rt = "\r\n";
        String methodStr = "";
        for(Method m : interface.getMethods()) {
            methodStr += "    @Override" + rt
                + "    public void " + m.getName() + "() {" + rt
                + "        try { " + rt
                + "            Method md = " + interface.getSimpleName() +
                ".class.getMethod(\""
                    + m.getName() + "\");" + rt
                + "            h.invoke(this, md);" + rt
                + "        } catch (Exception e) { " + rt
                + "            e.printStackTrace();" + rt
                + "        }" + rt
                + "    }";
        }
    }
}

```

```

    }
    String code =
        "package com.zxt.jdkproxy;" + rt + "\n"
        + "import java.lang.reflect.Method;" + rt
        + "import com.zxt.staticproxy.Moveable;" + rt + "\n"
        + "public class $MyProxy0 implements " + interface.getSimpleName() + " {"
+ rt + "\n"
        + "    private MyInvocationHandler h;" + rt + "\n"
        + "    public $MyProxy0( MyInvocationHandler h ) {" + rt
        + "        this.h = h;" + rt
        + "    }" + rt + "\n"
        + methodStr + rt + "\n"
        + "}";

    // 由源代码生成java类文件
    String filename = System.getProperty("user.dir") +
"/bin/com/zxt/jdkproxy/$MyProxy0.java";
    File file = new File(filename);
    // 使用commons-io里面的简便的工具类来写文件
    FileUtils.writeStringToFile(file, code, "UTF-8");

    // 2、编译源码（JDK Compiler API），产生新的类（代理类）
    // 拿到编译器
    JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();
    // 文件管理者
    StandardJavaFileManager fileManager = compiler.getStandardFileManager(null,
null, null);
    // 获取文件
    Iterable units = fileManager.getJavaFileObjects(filename);
    // 获取编译任务
    CompilationTask task = compiler.getTask(null, fileManager, null, null, null,
units);
    // 编译
    task.call();
    fileManager.close();

    // 3、加载到内存
    ClassLoader cl = ClassLoader.getSystemClassLoader();
    Class c = cl.loadClass("com.zxt.jdkproxy.$MyProxy0");

    // 4、返回代理类
    Constructor ctr = c.getConstructor(MyInvocationHandler.class);

```

```
        return ctr.newInstance(h);  
    }  
}
```

测试类:

```
public class MyProxyTest {  
  
    public static void main(String[] args) throws Exception {  
        // 需要被代理的对象  
        Car car = new Car();  
        MyInvocationHandler h = new MyLogHandler(car);  
  
        Moveable m = (Moveable) MyProxy.newProxyInstance(Moveable.class, h);  
        m.move();  
    }  
}
```

4.2、享元模式

所谓享元模式就是运用共享技术有效地支持大量细粒度对象的复用。系统使用少量对象，而且这些都比较相似，状态变化小，可以实现对象的多次复用。

共享模式是支持大量细粒度对象的复用，所以享元模式要求能够共享的对象必须是细粒度对象。

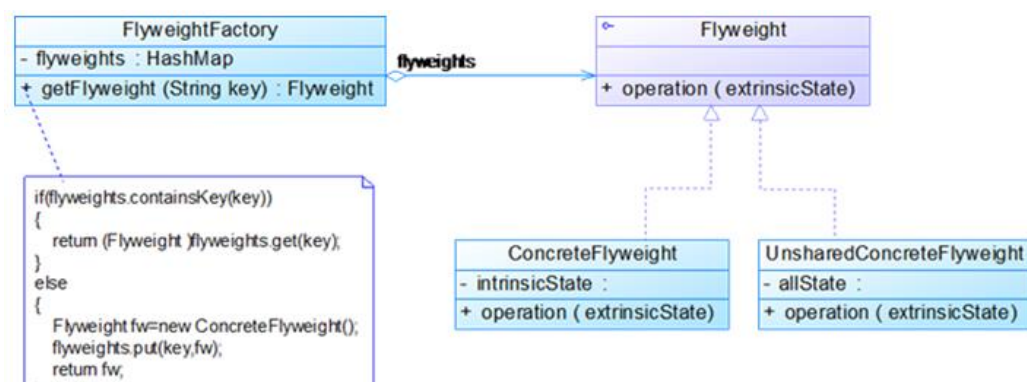
在了解享元模式之前我们先要了解两个概念：内部状态、外部状态。

内部状态：在享元对象内部不随外界环境改变而改变的共享部分。

外部状态：随着环境的改变而改变，不能够共享的状态就是外部状态。

由于享元模式区分了内部状态和外部状态，所以我们可以设置不同的外部状态使得相同的对象可以具备一些不同的特性，而内部状态设置为相同部分。在我们的程序设计过程中，我们可能会需要大量的细粒度对象来表示对象，如果这些对象除了几个参数不同外其他部分都相同，这个时候我们就可以利用享元模式来大大减少应用程序当中的对象。如何利用享元模式呢？这里我们只需要将他们少部分的不同的部分当做参数移动到类实例的外部去，然后在方法调用的时候将他们传递过来就可以了。这里也就说明了一点：内部状态存储于享元对象内部，而外部状态则应该由客户端来考虑。

享元模式结构：



享元模式存在如下几个角色：

Flyweight: 抽象享元类。所有具体享元类的超类或者接口，通过这个接口，Flyweight可以接受并作用于外部专题；

ConcreteFlyweight: 具体享元类。指定内部状态，为内部状态增加存储空间。

UnsharedConcreteFlyweight: 非共享具体享元类。指出那些不需要共享的Flyweight子类。

FlyweightFactory: 享元工厂类。用来创建并管理 **Flyweight** 对象，它主要用来确保合理地共享 **Flyweight**，当用户请求一个 **Flyweight** 时，**FlyweightFactory** 就会提供一个已经创建的 **Flyweight** 对象或者新建一个（如果不存在）。

享元模式的核心在于享元工厂类，享元工厂类的作用在于提供一个用于存储享元对象的享元池，用户需要对象时，首先从享元池中获取，如果享元池中不存在，则创建一个新的享元对象返回给用户，并在享元池中保存该新增对象。

```
public class FlyweightFactory {  
    // 享元对象池  
    private HashMap<String, FlyWeight> flyWeights = new HashMap<String, FlyWeight>();  
  
    public FlyWeight getFlyWeight(String key) {  
        // 享元池中存在享元对象，则直接返回  
        if(flyWeights.containsKey(key)) {  
            return (FlyWeight) flyWeights.get(key);  
        } else {  
            // 享元池中不存在享元对象，并将它放到享元池中  
            FlyWeight fw = new ConcreteFlyweight();  
            flyWeights.put(key, fw);  
            return fw;  
        }  
    }  
}
```

实例场景：假如我们有一个绘图的应用程序，通过它我们可以绘制各种各样的形状、颜色的图形，那么这里形状和颜色就是内部状态了，通过享元模式我们就可以实现该属性的共享了。另外在抽象出一个外部状态即绘图的用户，也就是说绘图的用户对象是不可以共享的，但是当大量用户绘制了相同属性（享元对象内部状态）的对象时，可以返回同一个引用，从而减少系统的对象数量。代码实现如下：

```
public class User {  
    private String name;  
  
    public User(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
}
```



```

        public void setName(String name) {
            this.name = name;
        }
    }

    /**
     * @Description: 抽象形状类，只有一个绘制图型的抽象方法，使用该方法需要传递一个用户对象
     * @author: zxt
     * @time: 2018年7月9日 上午9:43:41
     */
    public abstract class Shape {
        public abstract void draw(User user);
    }

    /**
     * @Description: 绘制图形的具体类
     * @author: zxt
     * @time: 2018年7月9日 上午9:46:16
     */
    public class Circle extends Shape {
        // 享元类的内部状态
        private String color;

        public Circle(String color) {
            this.color = color;
        }

        @Override
        public void draw(User user) {
            System.out.println("用户: " + user.getName() + ", 画了一个 '" + color + "' 的圆形!");
        }
    }

    public class CircleFactory {
        // 享元对象池
        private static HashMap<String, Shape> shapes = new HashMap<String, Shape>();

        public static Shape getShape(String key) {
            // 享元池中存在享元对象，则直接返回
            if(shapes.containsKey(key)) {
                return (Shape) shapes.get(key);
            } else {

```

```

        // 享元池中不存在享元对象，并将它放到享元池中
        Shape shape = new Circle(key);
        shapes.put(key, shape);
        return shape;
    }
}

public static int getShapesNum() {
    return shapes.size();
}
}

public class Client {

    public static void main(String[] args) {
        Shape shape1 = CircleFactory.getShape("红色");
        shape1.draw(new User("张三"));

        Shape shape2 = CircleFactory.getShape("灰色");
        shape2.draw(new User("李四"));

        Shape shape3 = CircleFactory.getShape("绿色");
        shape3.draw(new User("王五"));

        Shape shape4 = CircleFactory.getShape("红色");
        shape4.draw(new User("赵四"));

        Shape shape5 = CircleFactory.getShape("灰色");
        shape5.draw(new User("前乾"));

        Shape shape6 = CircleFactory.getShape("灰色");
        shape6.draw(new User("孙李"));

        System.out.println("一共创建了: " + CircleFactory.getShapesNum() + " 个图形对象!");
    }
}

```

terminated: client (2) Java Application

```

用户: 张三, 画了一个 '红色' 的圆形!
用户: 李四, 画了一个 '灰色' 的圆形!
用户: 王五, 画了一个 '绿色' 的圆形!
用户: 赵四, 画了一个 '红色' 的圆形!
用户: 前乾, 画了一个 '灰色' 的圆形!
用户: 孙李, 画了一个 '灰色' 的圆形!
一共创建了: 3 个图形对象

```

模式优缺点

优点

- 1、享元模式的优点在于它能够极大的减少系统中对象的个数。
- 2、享元模式由于使用了外部状态，外部状态相对独立，不会影响到内部状态，所以享元模式使得享元对象能够在不同的环境被共享。

缺点

- 1、由于享元模式需要区分外部状态和内部状态，使得应用程序在某种程度上来说更加复杂化了。
- 2、为了使对象可以共享，享元模式需要将享元对象的状态外部化，而读取外部状态使得运行时间变长。

模式适用场景

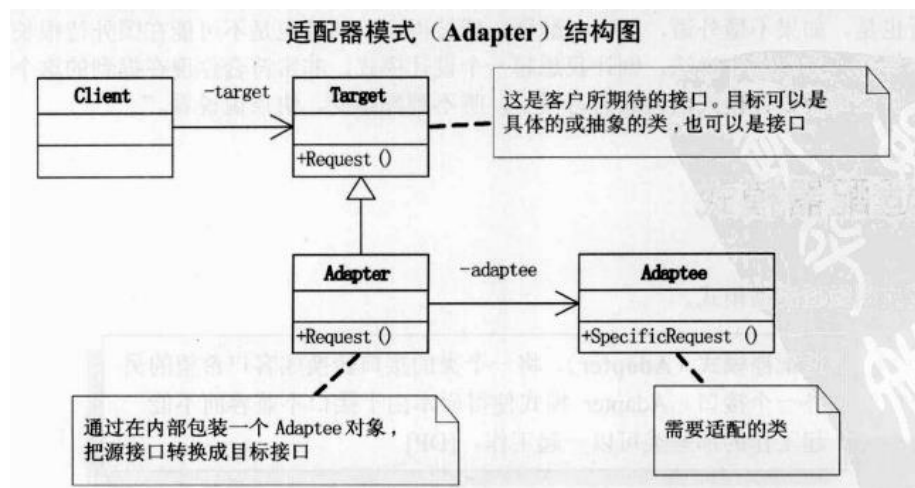
- 1、如果一个系统中存在大量的相同或者相似的对象，由于这类对象的大量使用，会造成系统内存的耗费，可以使用享元模式来减少系统中对象的数量。
- 2、对象的大部分状态都可以外部化，可以将这些外部状态传入对象中。
- 3、`String` 对象的常量池，以及 `Integer` 等包装类的缓存策略：`Integer.valueOf(int i)`等都使用了享元模式。

模式总结

- 1、享元模式可以极大地减少系统中对象的数量。但是它可能会引起系统的逻辑更加复杂化。
- 2、享元模式的核心在于享元工厂，它主要用来确保合理地共享享元对象。
- 3、内部状态为不变共享部分，存储于享元对象内部，而外部状态是可变部分，它应当由客户端来负责。

4.3、适配器模式

适配器模式 (Adapter): 将一个类的接口转换成客户希望的另外一个接口。Adapter 模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。



有两种适配器模式的实现方法，第一种是组合的方式（对象适配器模式）：适配器类将被适配者作为对象组合到该类中以修改目标接口包装被适配者。

第二种是继承的方式（类适配器模式）：实现目标对象类而继承被适配类，再重写被适配类从而适配目标类，此法的缺点是只能继承某个类（java 是单继承的）那么就只能适配一个类，多个类的适配要重写多次。

4.3.1、对象适配器模式

1、一个目标类（接口）。

```
/**
 * @Description: 三接口的插座，假如要使用插座的是笔记本电脑，也就是客户端所需要的接口
 * @author: zxt
 * @time: 2019年5月8日 下午8:47:36
 */
public interface ThreePlugInterface {

    // 使用三相插座充电
    public void powerWithThree();
}
```

2、一个已有的类（需要被适配的类）。

```
/**
 * @Description: 这是一个二接口的电源，无法直接给笔记本电脑供电。即这是已经存在的类（接口），
```

但是无法被客户端直接使用，需要被适配

```
* @author: zxt
* @time: 2019年5月8日 下午9:05:37
*/
public class GBTwoPlug {

    public void powerWithTwo() {
        System.out.println("使用二相电流供电!!!");
    }
}
```

3、一个适配器类，将被适配者作为对象组合到该类中以修改目标接口。

```
/**
 * @Description: 二相插座的适配器类，使得二相插座能够满足客户端的需求为三相电源供电
 * @author: zxt
 * @time: 2019年5月8日 下午9:12:50
 */
public class TwoPlugAdapter implements ThreePlugInterface {
    // 使用组合的方式实现适配器类
    private GBTwoPlug twoPlug;

    public TwoPlugAdapter(GBTwoPlug twoPlug) {
        this.twoPlug = twoPlug;
    }

    @Override
    public void powerWithThree() {
        System.out.print("组合的方式实现适配器类: ");
        twoPlug.powerWithTwo();
    }
}
```

4、客户端的使用

```
/**
 * @Description: 客户端的需求
 * @author: zxt
 * @time: 2019年5月8日 下午9:08:17
 */
public class Computer {

    // 笔记本电脑需要一个三相的插座供电
    private ThreePlugInterface plug;

    public Computer(ThreePlugInterface plug) {
        this.plug = plug;
    }
}
```

```

    }

    // 使用插座充电
    public void charge() {
        plug.powerWithThree();
    }

    public static void main(String[] args) {
        GBTwoPlug two = new GBTwoPlug();
        ThreePlugInterface three = new TwoPlugAdapter(two);
        Computer cp = new Computer(three);
        cp.charge();
    }
}

```

对于对象适配器的总结：

- 1、适配器的类中需要包含被适配器的对象作为成员变量，同时适配器类要实现目标接口。
- 2、被适配者产生对象作为参数传到适配器类中，因为适配器类是目标接口的实现类，所以目标接口通过适配器类产生对象，最后通过使用者类接收目标接口通过适配器类产生的对象作为参数来产生使用者类的对象。
- 3、因为使用者类中有目标接口的对象作为成员变量，从而通过使用者类的对象来调用使用者类中被适配好的方法（目标接口被适配器修改内容的方法）。

4.3.2、类适配器模式

1、适配器类

```

/**
 * @Description: 二使用继承的方式实现适配器
 * @author: zxt
 * @time: 2019年5月8日 下午9:12:50
 */
public class TwoPlugAdapterExtends extends GBTwoPlug implements ThreePlugInterface {

    @Override
    public void powerWithThree() {
        System.out.print("继承的方式实现适配器类: ");
        this.powerWithTwo();
    }
}

```

2、客户端使用

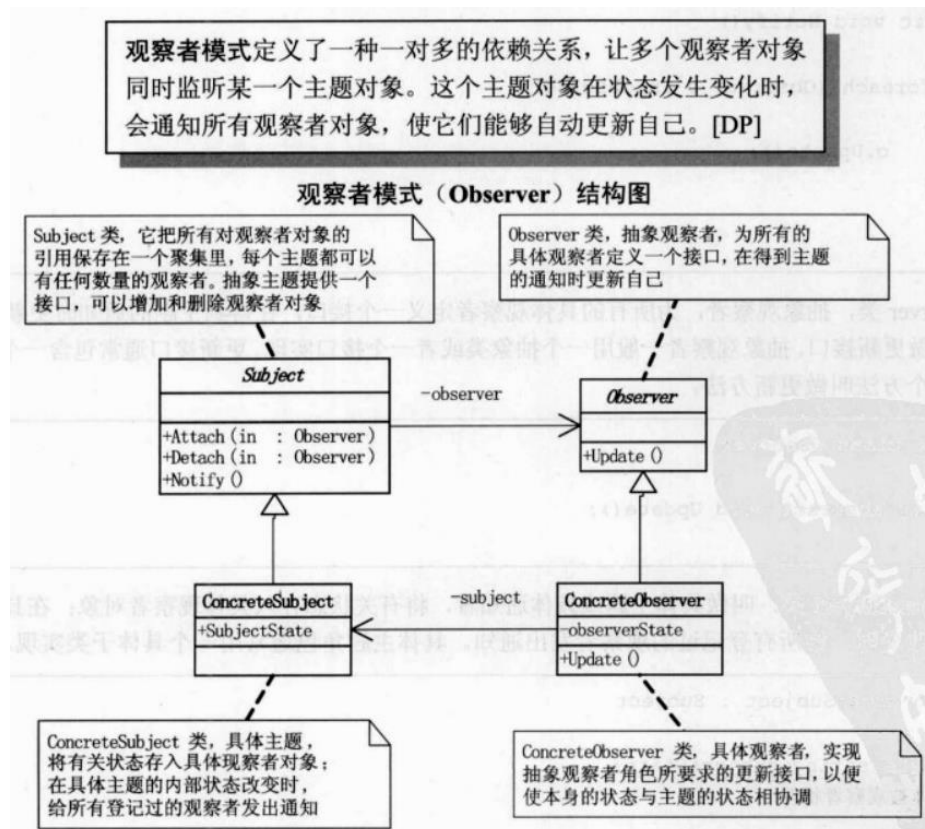
```
public class Computer {  
    // 笔记本电脑需要一个三相的插座供电  
    private ThreePlugInterface plug;  
    public Computer(ThreePlugInterface plug) {  
        this.plug = plug;  
    }  
  
    // 使用插座充电  
    public void charge() {  
        plug.powerWithThree();  
    }  
  
    public static void main(String[] args) {  
        GBTwoPlug two = new GBTwoPlug();  
        ThreePlugInterface three = new TwoPlugAdapter(two);  
        Computer cp = new Computer(three);  
        cp.charge();  
  
        three = new TwoPlugAdapterExtends();  
        cp = new Computer(three);  
        cp.charge();  
    }  
}
```

通过继承的方式成为类适配器。特点：通过多重继承不兼容接口，实现对目标接口的匹配，单一的为某各类而实现适配。

五、行为型模式

5.1、观察者模式

观察者模式又叫做发布-订阅（Publish/Subscribe）模式。



1、Subject 类，可翻译为主题或者抽象通知者，一般用一个抽象类或者一个接口实现。

它把所有对观察者对象的引用保存在一个集合里，每个主题都可以有任何数量的观察者。抽象主题提供一个接口，可以增加和删除观察者对象。

```
import java.util.ArrayList;
import java.util.List;

/**
 * @Description: Subject类，可翻译为主题或者抽象通知者，一般用一个抽象类或者一个接口实现。
 * 它把所有对观察者对象的引用保存在一个集合里，每个主题都可以有任何数量的观察者。抽象主题提供一个
 * 接口，可以增加和删除观察者对象。
 * @author: zxt
 * @time: 2019年4月25日 上午10:54:34
 */
public class Subject {

    private List<Observer> observers = new ArrayList<Observer>();
```



```

// 增加观察者
public void attach(Observer observer) {
    observers.add(observer);
}

// 移除观察者
public void detach(Observer observer) {
    observers.remove(observer);
}

// 通知
public void notifyUpdate() {
    for(Observer o : observers) {
        o.update();
    }
}
}

```

2、抽象观察者，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。这个接口叫做更新接口。抽象观察者一般用一个抽象类或者一个接口实现。更新接口通常包含一个 `update()` 方法，这个方法叫做更新方法。

```

/**
 * @Description: 抽象观察者，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。这个接口叫做更新接口。抽象观察者一般用一个抽象类或者一个接口实现。更新接口通常包含一个update()方法，这个方法叫做更新方法
 * @author: zxt
 * @time: 2019年4月25日 上午11:01:12
 */
public abstract class Observer {

    public abstract void update();
}

```

3、ConcreteSubject 类，叫做具体主题或者具体通知者，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。

```

/**
 * @Description: ConcreteSubject类，叫做具体主题或者具体通知者，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。
 * @author: zxt
 * @time: 2019年4月25日 上午11:09:24
 */
public class ConcreteSubject extends Subject {

```

```

// 具体被观察者状态
private String subjectState;

public String getSubjectState() {
    return subjectState;
}

public void setSubjectState(String subjectState) {
    this.subjectState = subjectState;
}
}

```

4、ConcreteObserver 类，具体观察者，实现抽象观察者角色所要求的更新接口，以便使本身的状态与主题的状态相协调。具体观察者可以保存一个指向具体主题对象的引用。

```

/**
 * @Description: ConcreteObserver类，具体观察者，实现抽象观察者角色所要求的更新接口，以便
 * 使本身的状态与主题的状态相协调。具体观察者角色可以保存一个指向具体主题对象的引用。
 * @author: zxt
 * @time: 2019年4月25日 上午11:22:18
 */
public class ConcreteObserver extends Observer {
    private String name;
    private String observerState;
    private ConcreteSubject subject;

    public ConcreteObserver(ConcreteSubject subject, String name) {
        this.name = name;
        this.subject = subject;
    }

    @Override
    public void update() {
        observerState = subject.getSubjectState();
        System.out.println("观察者 " + name + " 的新状态是" + observerState);
    }

    public ConcreteSubject getSubject() {
        return subject;
    }

    public void setSubject(ConcreteSubject subject) {
        this.subject = subject;
    }
}

```

5.1.1、观察者模式的特点

将一个系统分割成一系列相互协作的类有一个很不好的副作用，那就是需要维护相关对象间的一致性。我们不希望为了维护一致性而使各类紧密耦合，这样会给维护、扩展和重用都带来不便。

而观察者模式的关键对象是主题 **Subject** 和观察者 **Observer**，一个 **Subject** 可以有任意数目的依赖它的 **Observer**，一旦 **Subject** 的状态发生了改变，所有的 **Observer** 都可以得到通知。**Subject** 发出通知时并不需要知道谁是它的观察者，也就是说，具体观察者是谁，它根本不需要知道。而任何一个具体观测者不知道也不需要知道其他观察者的存在。

当一个对象的改变需要同时改变其他对象的时候，而且它不知道具体有多少对象有待改变时，应该使用观察者模式。一个抽象模型有两个方面，其中一方面依赖于另一方面，这时用观察者模式可以将这两者封装在独立的对象中使它们各自独立地改变和复用。总的来说，观察者模式所做的工作其实就是在解除耦合，让耦合的双方都依赖于抽象，而不是依赖于具体，从而使得各自的变化都不会影响另一边的变化。

而在具体的应用中观察者完全可能是风马牛不相及的类，但它们都需要根据通知者的通知来做出 **Update()** 的操作，所以抽象观察者为接口比类更好。

```
public interface Observer {  
  
    public abstract void update();  
}
```

5.1.2、观察者模式的不足

1、观察者所要做的状态更新不一定都在 **Update()** 方法中，有可能是其他方法（或者说没有抽象观察者这样的接口呢），通知者状态改变时如何做到通知不同类的不同方法做出改变。

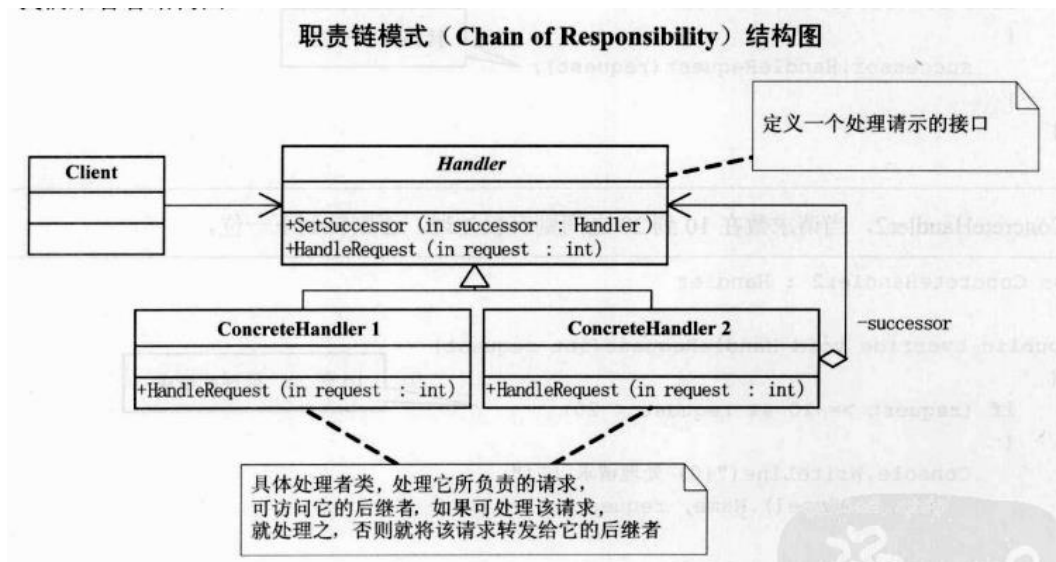
2、通知者对所有的观测者的通知需要循环依次唤醒。

3、解决上述问题的方法：事件委托机制。委托就是一种引用方法的类型。一旦为委托分配了方法，委托与该方法具有完全相同的行为。委托方法的使用可以像其他任何方法一样，具有参数和返回值。委托可以看做是对函数的抽象，是函数的“类”，委托的实例将代表一个具体的函数。而且一个委托可以搭载多个方法，所有方法被一次唤醒。更重要的是，它可以使得委托对象所搭载的方法并不需要同属于一个类。

委托机制使得抽象主题类中的增加和减少的抽象观察者集合以及通知时遍历的抽象观察者都不必要了。转到客户端来让委托搭载多个方法，这也就解决了本来与抽象观察者的耦合问题。

5.2、职责链模式

职责链模式（Chain of Responsibility）：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。



1、Handler 类，定义一个处理请求的接口。

```
/**
 * @Description: 定义一个处理请求的接口
 * @author: zxt
 * @time: 2019年5月13日 下午9:31:56
 */
public abstract class Handler {

    // 设置继任者
    protected Handler successor;

    public void setSuccessor(Handler successor) {
        this.successor = successor;
    }

    // 处理请求的抽象方法
    public abstract void HandleRequest(int request);
}
```

2、ConcreteHandler 类，具体处理者类，处理它所负责的请求，可访问它的后继者，如果可处理该请求，就处理之，否则就将该请求转发给它的后继者。

```

/**
 * @Description: 当请求数在0到10之间则有权处理，否则转到下一位
 * @author: zxt
 * @time: 2019年5月13日 下午9:49:34
 */
public class ConcreteHandler1 extends Handler {

    @Override
    public void HandleRequest(int request) {
        if (request >= 0 && request < 10) {
            System.out.println(this.getClass().getName() + " 处理请求 " + request);

        } else if (successor != null) {
            // 转移到下一位
            successor.HandleRequest(request);
        }
    }
}

```

```

/**
 * @Description: 当请求数在10到20之间则有权处理，否则转到下一位
 * @author: zxt
 * @time: 2019年5月13日 下午9:49:34
 */
public class ConcreteHandler2 extends Handler {

    @Override
    public void HandleRequest(int request) {
        if (request >= 10 && request < 20) {
            System.out.println(this.getClass().getName() + " 处理请求 " + request);

        } else if (successor != null) {
            // 转移到下一位
            successor.HandleRequest(request);
        }
    }
}

```

```

/**
 * @Description: 当请求数在20到30之间则有权处理，否则转到下一位
 * @author: zxt
 * @time: 2019年5月13日 下午9:49:34
 */

```

```

public class ConcreteHandler3 extends Handler {

    @Override
    public void HandleRequest(int request) {
        if (request >= 20 && request < 30) {
            System.out.println(this.getClass().getName() + " 处理请求 " + request);

        } else if (successor != null) {
            // 转移到下一位
            successor.HandleRequest(request);
        }
    }
}

```

```

/**
 * @Description: 末端处理器
 * @author: zxt
 * @time: 2019年5月13日 下午9:49:34
 */
public class ConcreteHandler extends Handler {

    @Override
    public void HandleRequest(int request) {
        System.out.println(this.getClass().getName() + " 处理请求 " + request);
    }
}

```

3、测试

```

public class Test {

    public static void main(String[] args) {
        // 设置职责链的上下家
        Handler h1 = new ConcreteHandler1();
        Handler h2 = new ConcreteHandler2();
        Handler h3 = new ConcreteHandler3();
        Handler h = new ConcreteHandler();
        h1.setSuccessor(h2);
        h2.setSuccessor(h3);
        h3.setSuccessor(h);

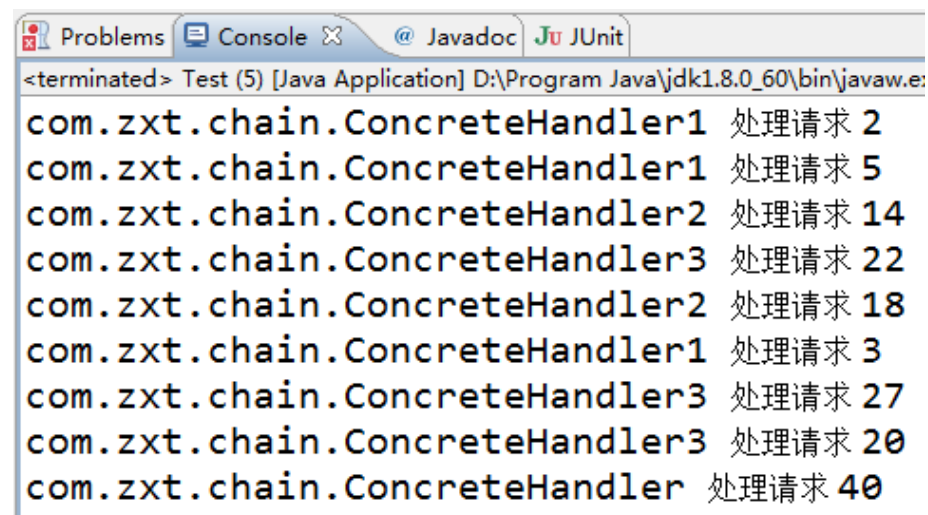
        int[] requests = {2, 5, 14, 22, 18, 3, 27, 20, 40};
    }
}

```

```

// 循环给最小处理者提交请求，不同的数额，由不同权限处理者处理
for(int i = 0; i < requests.length; i++) {
    h1.HandleRequest(requests[i]);
}
}
}

```



```

<terminated> Test (5) [Java Application] D:\Program Java\jdk1.8.0_60\bin\javaw.e:
com.zxt.chain.ConcreteHandler1 处理请求 2
com.zxt.chain.ConcreteHandler1 处理请求 5
com.zxt.chain.ConcreteHandler2 处理请求 14
com.zxt.chain.ConcreteHandler3 处理请求 22
com.zxt.chain.ConcreteHandler2 处理请求 18
com.zxt.chain.ConcreteHandler1 处理请求 3
com.zxt.chain.ConcreteHandler3 处理请求 27
com.zxt.chain.ConcreteHandler3 处理请求 20
com.zxt.chain.ConcreteHandler 处理请求 40

```

5.2.1、职责链模式的好处

职责链当中最关键的是当客户提交一个请求时，请求是沿链传递直至有一个 **ConcreteHandler** 对象负责处理它。这就使得接收者和发送者都没有对方的明确信息，且链中的对象自己也不知道链的结构。结果是职责链可简化对象的相互连接，它们仅需保持一个指向其后继者的引用，而不需要保持它所有的候选者的引用。这也就大大降低了耦合度。

在客户端可以随时地增加或修改处理一个请求的结构，增强了给对象指派职责的灵活性。不过也需要当心，一个请求极有可能到了链的末端都得不到处理，或者因为没有正确配置而得不到处理。

六、总结

内聚性描述的是一个例程内部组成部分之间相互联系的紧密程度。而耦合性描述的是一个例程与其他例程之间联系的紧密程度。软件开发的目标应该是创建这样的例程：内部完整，也就是高内聚，而与其他例程之间的联系是小巧、直接、可见、灵活的，这就是松耦合。

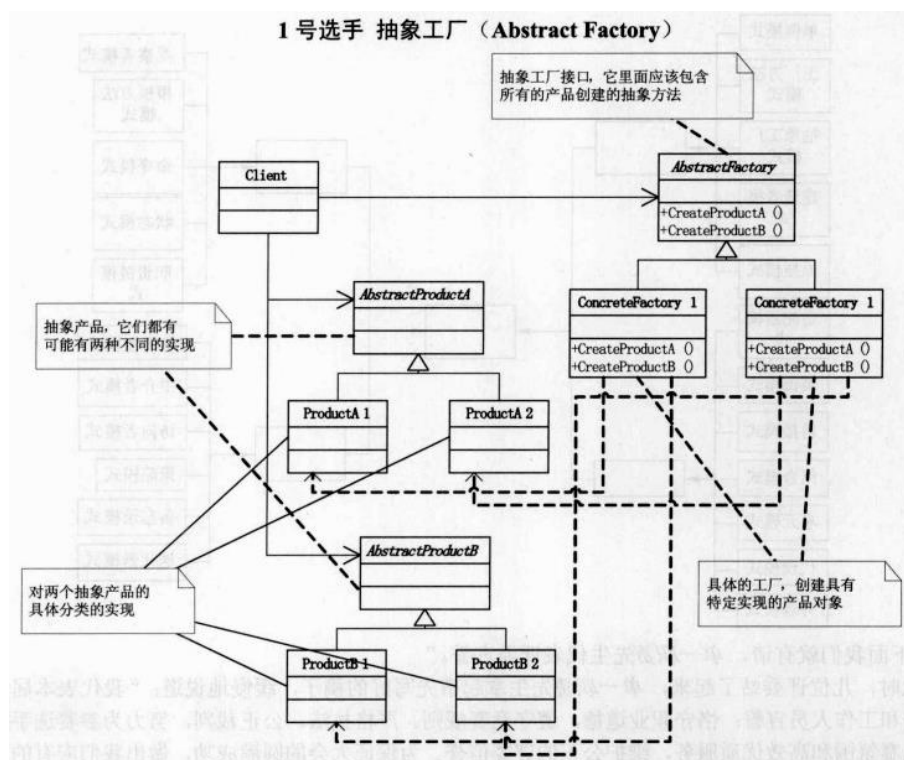
6.1、创建型模式

创建型模式隐藏了这些类的实例是如何被创建和放在一起，整个系统关于这些对象所知道的是由抽象类所定义的接口。这样，创建型模式在创建了什么、谁创建它、它是怎么被创建的，以及何时创建这些方面提供了很大的灵活性。

创建型模式抽象了实例化的过程。它们帮助一个系统独立于如何创建、组合和表示它的那些对象。创建型模式都会将关于该系统使用哪些具体的类的信息封装起来。允许客户用结构和功能差别很大的‘产品’对象配置一个系统。配置可以是静态的，即在编译时指定，也可以是动态的，就是运行时再指定。

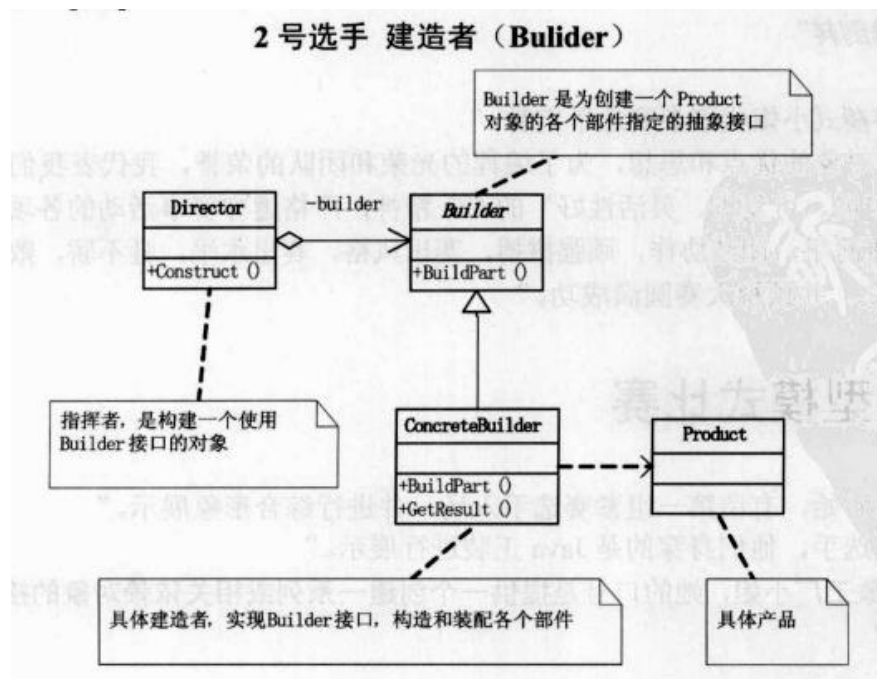
6.1.1、抽象工厂模式（Abstract Factory）

抽象工厂：提供一个创建一系列或相关依赖对象的接口，而无需指定它们具体的类。



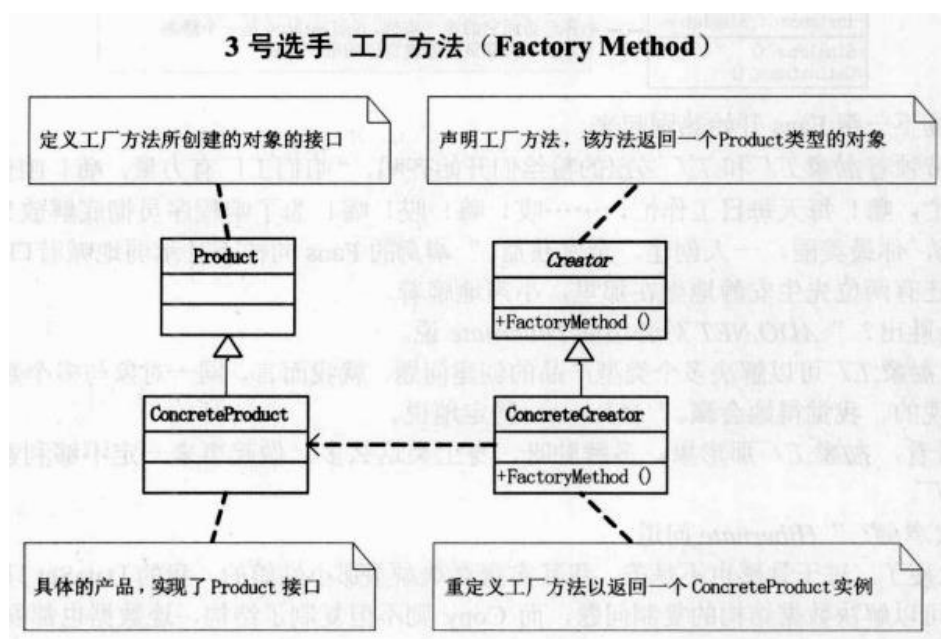
6.1.2、建造者模式 (Builder)

建造者模式：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。



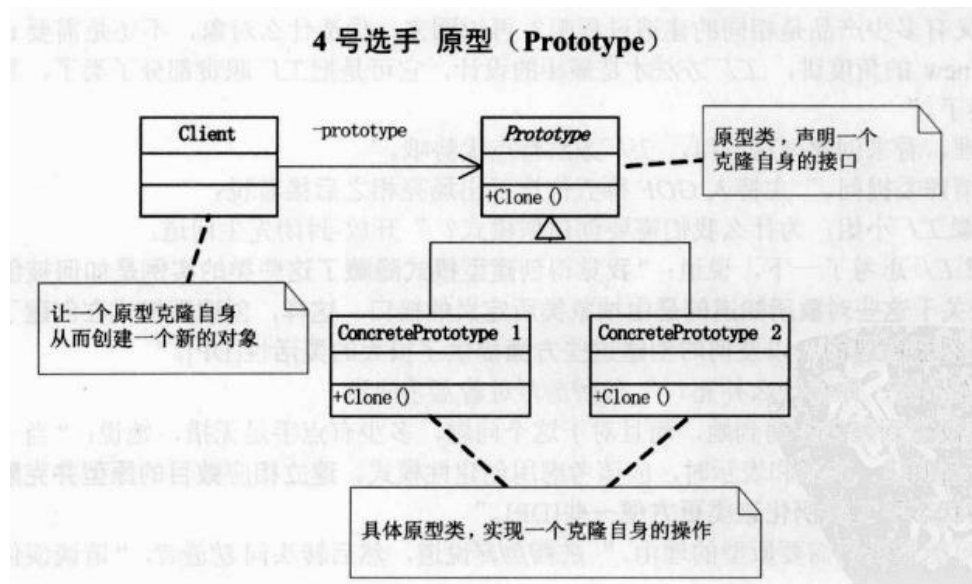
6.1.3、工厂方法模式 (Factory Method)

工厂方法模式：定义一个用于创建对象的接口，让子类决定实例化哪一个类，工厂方法使一个类的实例化延迟到其子类。



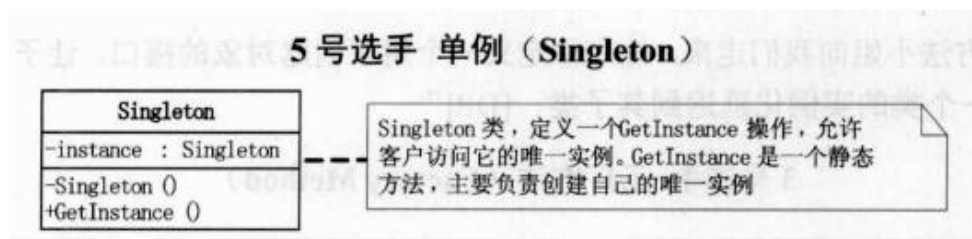
6.1.4、原型模式 (Prototype)

原型模式：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。



6.1.5、单例模式 (Singleton)

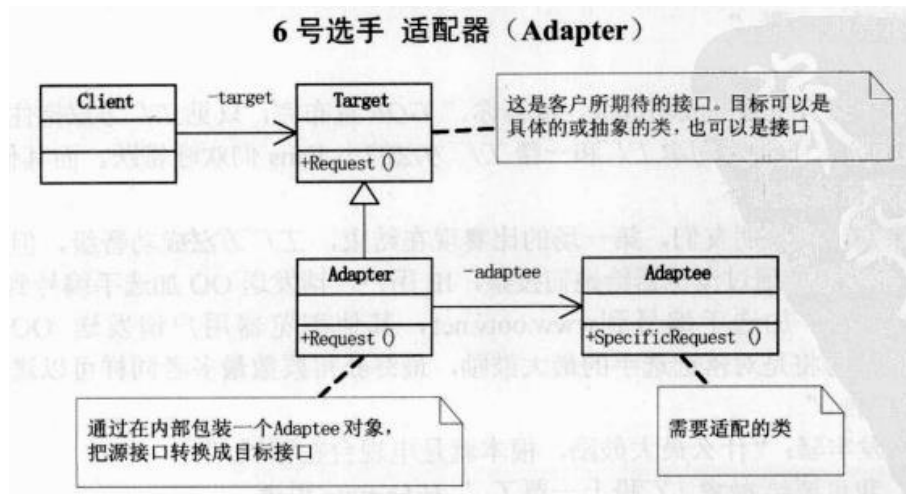
单例模式：保证一个类仅有一个实例，并提供一个访问它的全局访问点。



6.2、结构型模式

6.2.1、适配器模式 (Adapter)

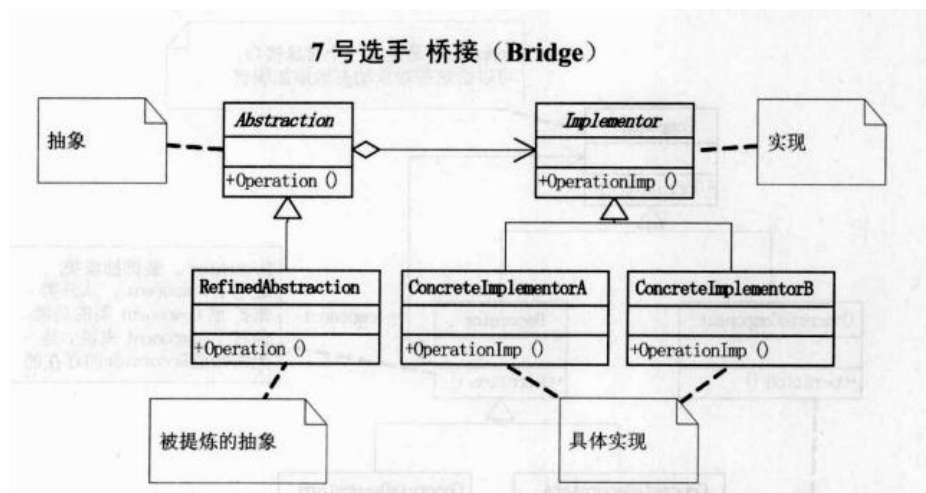
适配器模式：是将一个类的接口转换成客户希望的另外一个接口。适配器模式使得原本由于接口**不兼容**而不能一起工作的那些类可以一起工作。



面向对象的精神就是更好地应对需求的变化，而现实中往往会有下面这些情况，想使用一个已经存在的类，而它的接口不符合要求，或者希望创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类协同工作。适配器模式主要就是为了解决两个已有接口之间不匹配的问题，它不需要考虑这些接口是怎样实现的，也不考虑它们各自可能会如何演化。适配器模式不需要对两个独立设计的类中任一个进行重新设计，通过适配使它们协同工作。

6.2.2、桥接模式 (Bridge)

桥接模式：是将抽象部分与它的实现部分分离，使它们都可以独立地变化。

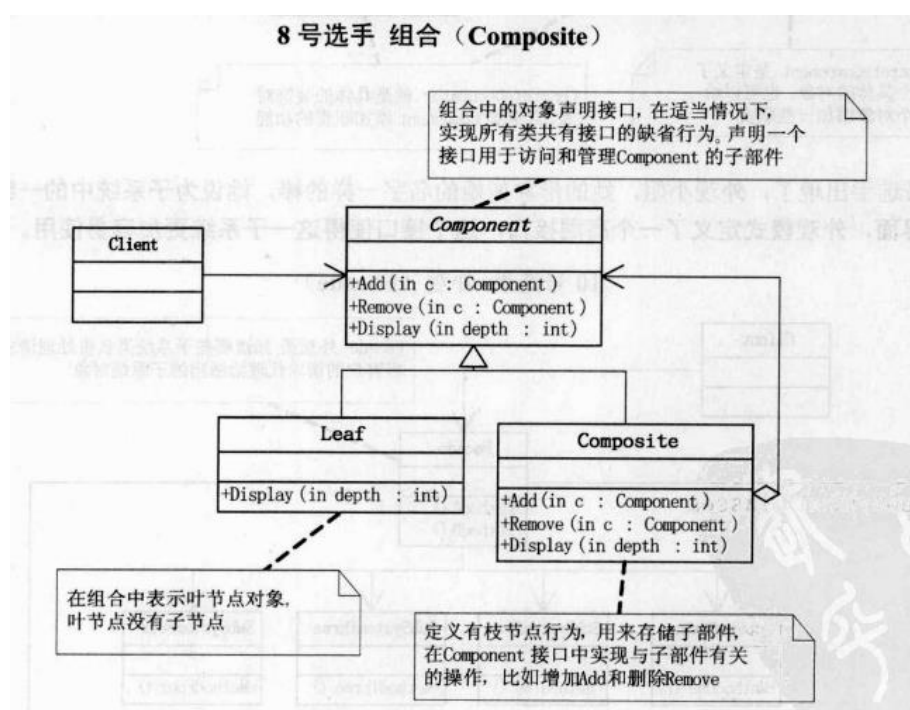


继承是好的东西，但往往会过度地使用，继承会导致类的结构过于复杂，关系太多，难以维护，而更糟糕的是扩展性非常差。而仔细研究如果能发现继承体系中，有两个甚至多个方向的变化，那么就解耦这些不同方向的变化，通过对象组合的方式，把两个角色之间的继承关系改为了组合的关系，从而使这两者可以应对各自独立的变化。总之，面对变化，主张‘找出变化并封装之’。

桥接模式和适配器模式具有一些共同的特征，就是给另一对象提供一定程度的间接性，这样可以有利于系统的灵活性。但正所谓未雨绸缪，我们不能等到问题发生了，再去考虑解决问题，而是更应该在设计之初就想好应该如何做来避免问题的发生，桥接模式通常是在设计之初，就对抽象接口与它的实现部分进行桥接，让抽象与实现两者可以独立演化。

6.2.3、组合模式（Composite）

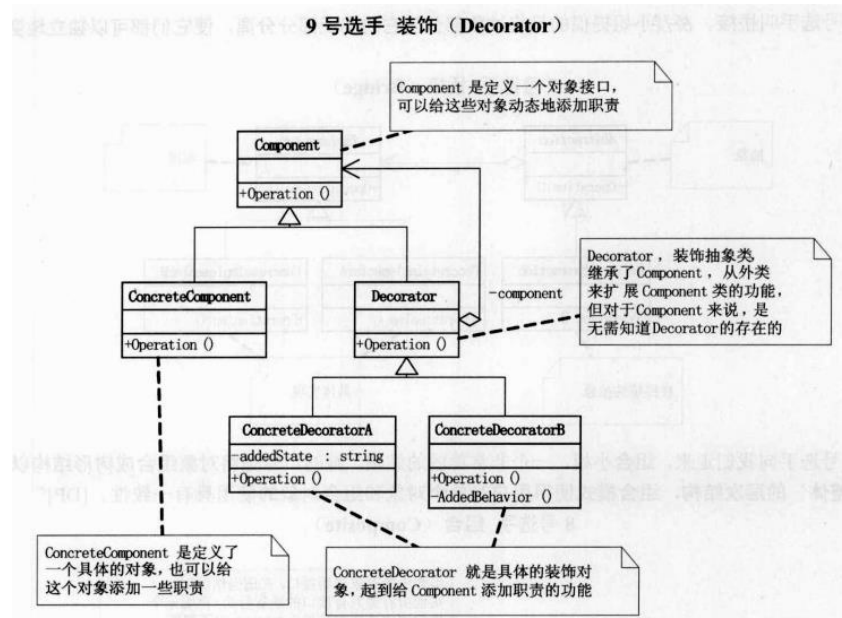
组合模式：是将对象组合成树形结构以表示‘部分-整体’的层次结构，组合模式使得用户对单个对象和组合对象的使用具有一致性。



组合模式希望用户忽略组合对象与单个对象的不同，用户将可以统一地使用组合结构中的所有对象。用户使用组合类接口与组合结构中的对象进行交互，如果接收者是一个叶节点，则直接处理请求，如果接收者是组合对象，通常将请求发送给它的子部件，并在转发请求之前或之后可能执行一些辅助操作。组合模式的效果是客户可以一致地使用组合结构和单个对象。任何用到基本对象的地方都可以使用组合对象。

6.2.4、装饰模式（Decorator）

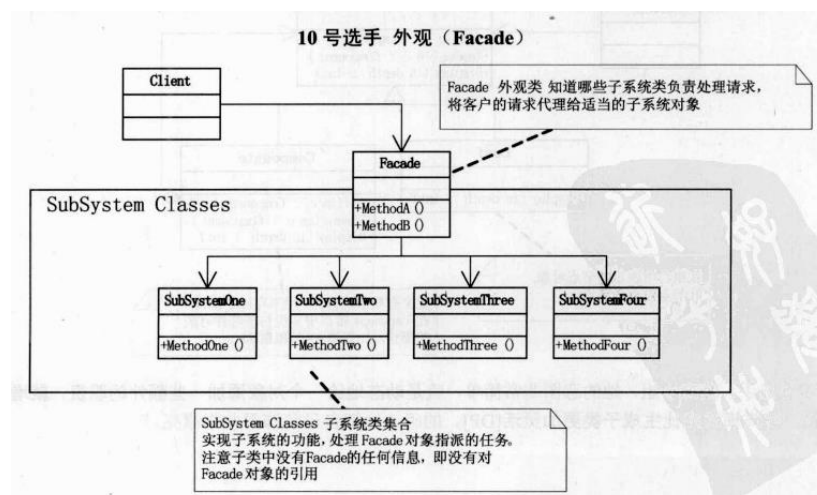
装饰模式：就是动态地给一个对象添加一些额外的职责。就增加功能来说，装饰模式相比生成子类更加灵活。



面对变化，如果采用生成子类的方法进行扩充，为支持每一种扩展的组合，会产生大量的子类，使得子类数目呈爆炸性增长。而事实上，这些子类多半只是为某个对象增加一些职责，此时通过装饰的方式，可以更加灵活、以动态、透明的方式给单个对象添加职责，并在不需要时，撤销相应的职责。

6.2.5、外观模式（Facade）

外观模式：为子系统的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。



信息的隐藏促进了软件的复用。

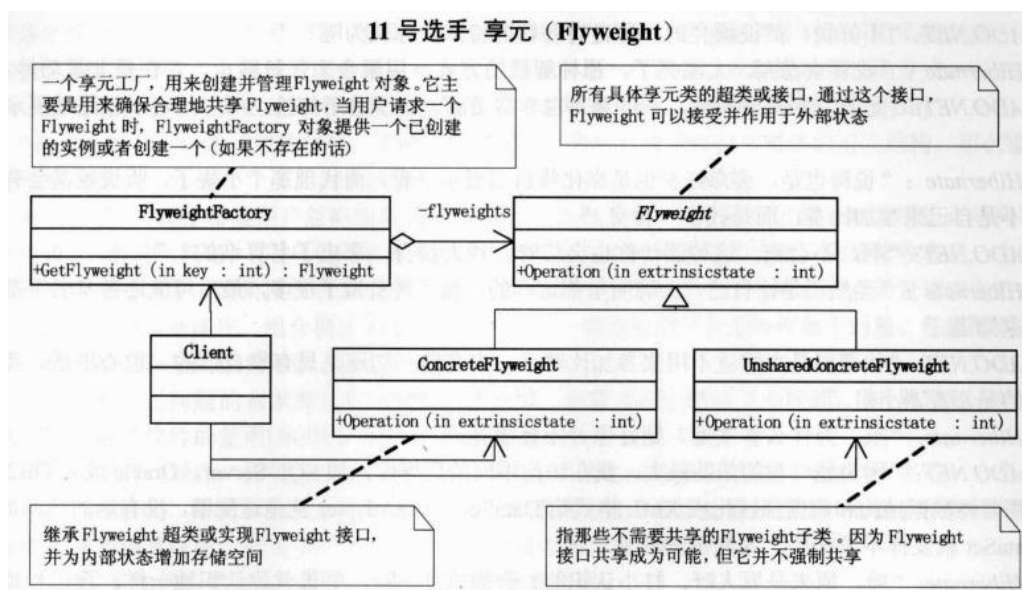
类之间的耦合越弱，越有利于复用，一个处在弱耦合的类被修改，不会对有关系的类造成波及。如果两个类不必彼此直接通信，那么就不要让这两个类发生直接的相互作用。如果实在需要调用，可以通过第三者来转发调用。

外观模式认为应该让一个软件中的子系统间的通信和相互依赖关系达到最小，而具体办法就是引入一个外观对象，它为子系统间提供了一个单一而简单的屏障。通常企业软件的三层或 N 层架构，层与层之间地分离其实就是外观模式的体现。

并不能说设计之初就一定比设计之后的弥补要好，事实上，在现实中，早已设计好的两个类，过后需要它们统一接口，整合为一的事例也比皆是。因此桥接和适配器是被用于软件生命周期的不同阶段，针对的是不同的问题，谈不上孰优孰劣。外观模式和适配器模式还有些近似，都是对现存系统的封装，有人说外观模式其实就是另外一组对象的适配器，这种说法是不准确的，因为外观定义的是一个新的接口，而适配器则是复用一个原有的接口，适配器是使两个已有的接口协同工作，而外观则是为现存系统提供一个更为方便的访问接口。如果硬要说外观模式是适配，那么适配器是用来适配对象的，而外观模式则是用来适配整个子系统的。也就是说，外观模式所针对的对象的粒度更大。

6.2.6、享元模式 (Flyweight)

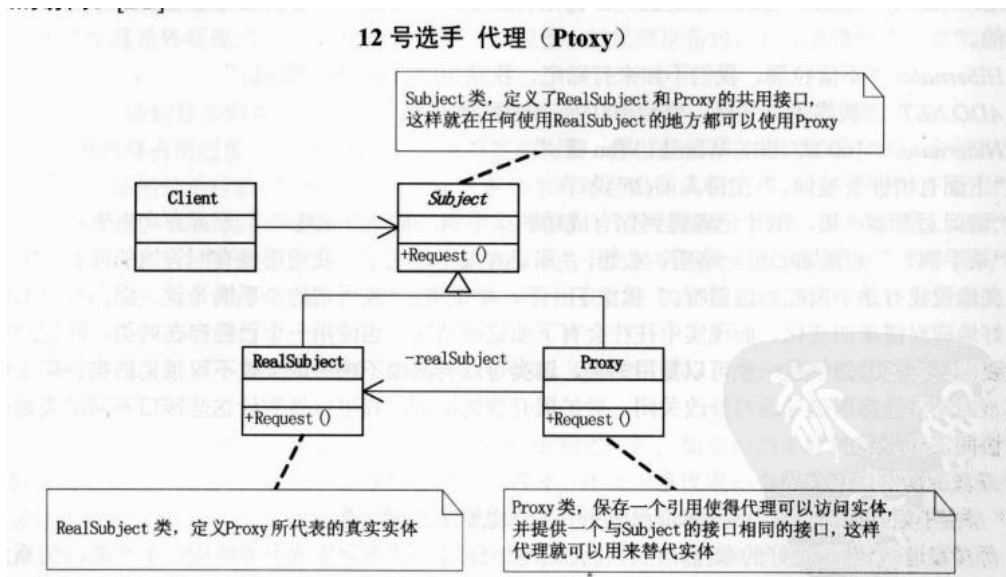
享元模式：运用共享技术有效地支持大量细粒度的对象。



对象使得内存占用过多，而且如果都是大量重复的对象，那就是资源的极大浪费。

6.2.7、代理模式（Proxy）

代理模式：为其他对象提供一种代理以控制对这个对象的访问。



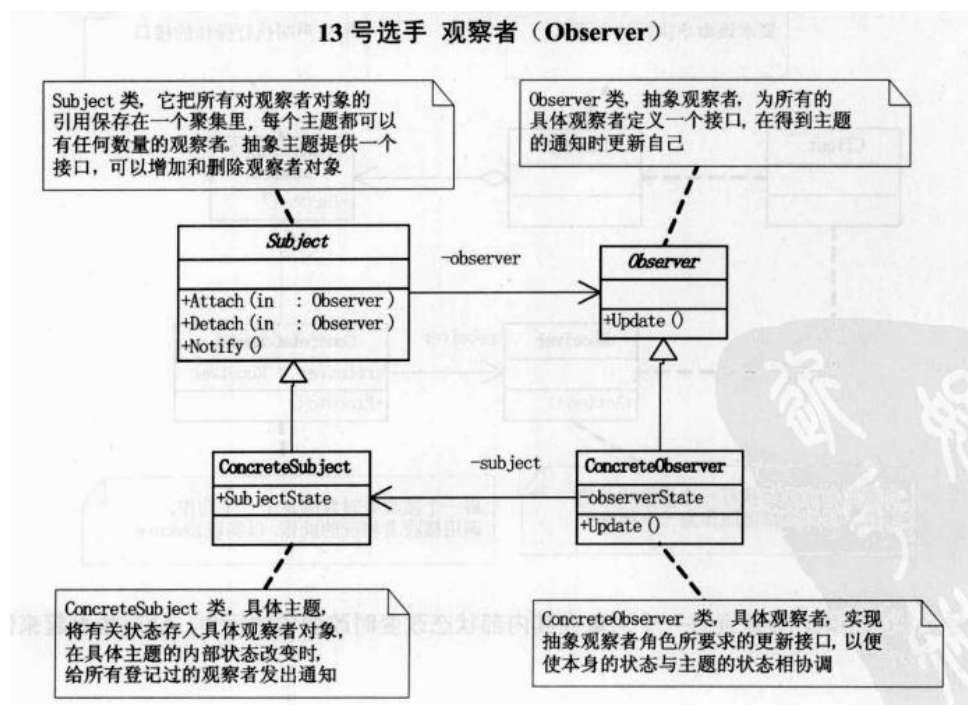
代理与外观的主要区别在于，代理对象代表一个单一对象而外观对象代表一个子系统；代理的客户对象无法直接访问目标对象，由代理提供对单独的目标对象的访问控制，而外观的客户对象可以直接访问子系统各个对象，但通常由外观对象提供对子系统各元件功能的简化的共同层次的调用接口。

至于代理与适配器，其实都是属于一种衔接性质的功能。代理是一种原来对象的代表，其他需要与这个对象打交道的操作都是和这个代表交涉。而适配器则不需要虚构出一个代表者，只需要为应付特定使用目的，将原来的类进行一些组合。

6.3、行为型模式

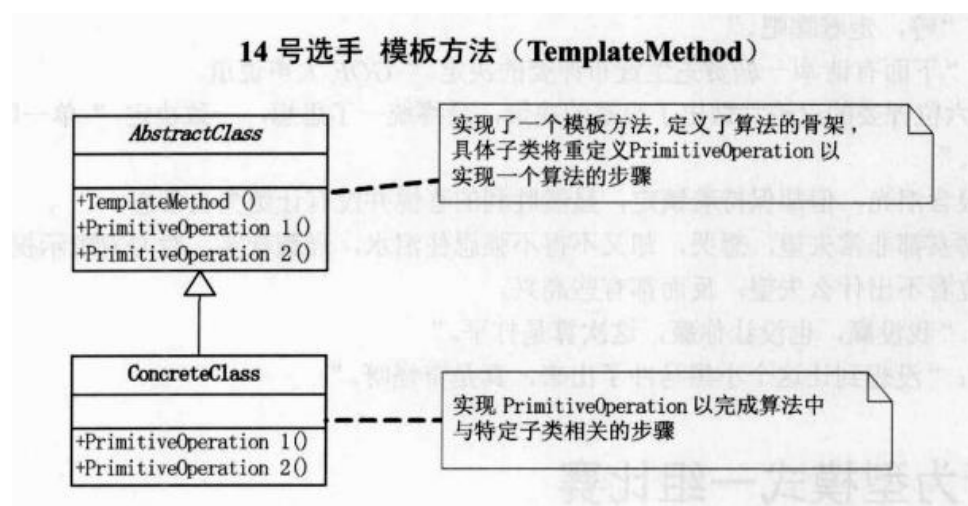
6.3.1、观察者模式 (Observer)

观察者模式：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。



6.3.2、模板方法模式 (Template Method)

模板方法模式：定义一个操作的算法骨架，而将一些步骤延迟到子类中，模板方法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。

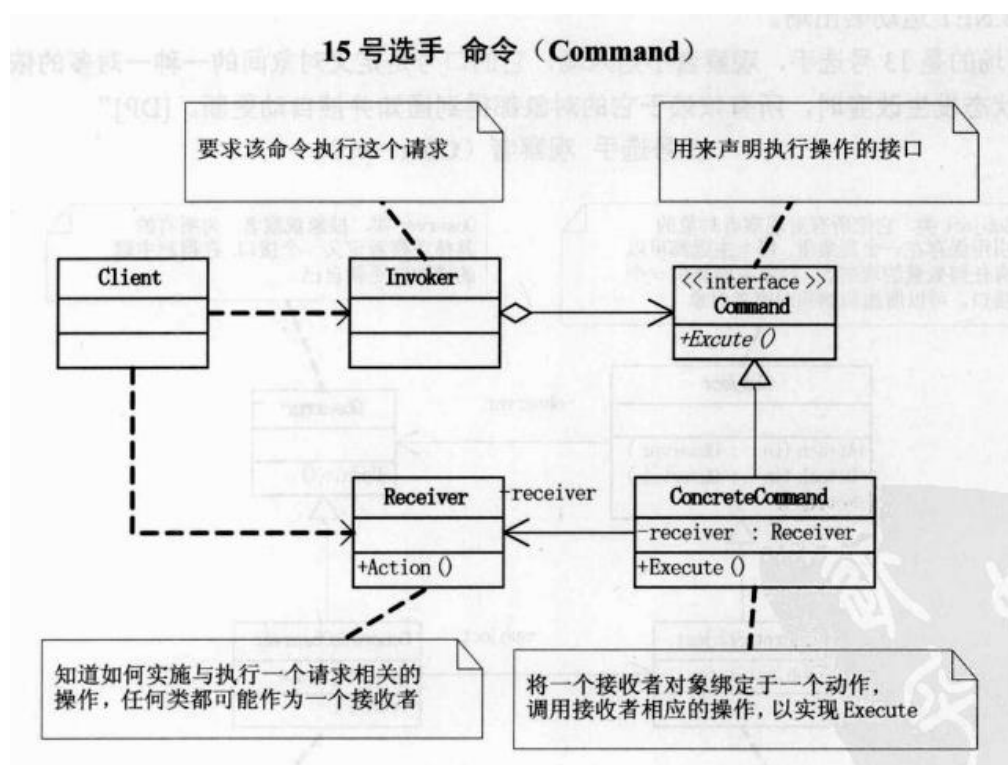


代码重复是编程中最常见、最糟糕的‘坏味道’，如果我们在一个以上的地方看到相同的程序结构，那么可以肯定，设法将它们合而为一，程序会变得更好。但是完全相同的代码当然存在明显的重复，而微妙的重复会出现在表面不同但是本质相同的结构或处理步骤中，这使得我们一定要小心处理。

继承的一个非常大的好处就是你能免费地从基类获取一些东西，当你继承一个类时，派生类马上就可以获得基类中所有的功能，你还可以在它的基础上任意增加新的功能。模板方法模式由一个抽象类组成，这个抽象类定义了需要覆盖的可能有不同实现的模板方法，每个从这个抽象类派生的具体类将为此模板实现新方法。这样就使得，所有可重复的代码都提炼到抽象类中了，这就实现了代码的重用。

6.3.3、命令模式（Command）

命令模式：将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化；可以对请求排队或记录请求日志，以及支持可撤销的操作。

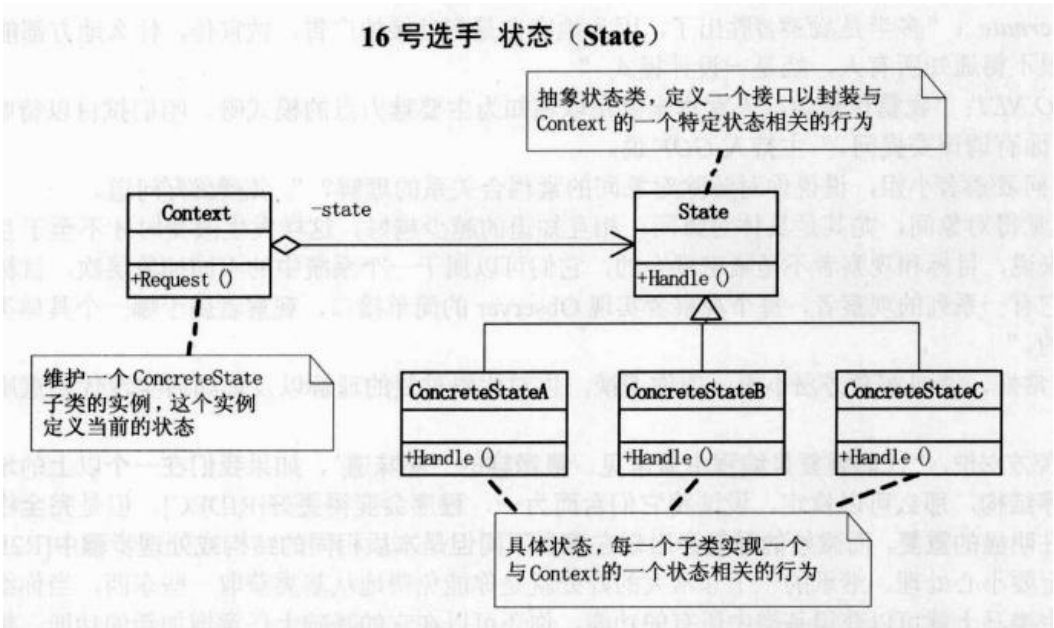


将调用操作的对象与知道如何实现该操作的对象解耦，而这就意味着命令模式可以在这两者之间处理很多事，比如完全可以发送者发送完请求就完事了，具体怎么做是命令模式的事，命令模式可以在不同的时刻指定、排列和执行请求。再比如命令模式可以在实施操作前将状态存储起来，以便支持取消/重做的操作。还可以记录整个操作的日志，以便以后可以

在系统出问题查找原因或恢复重做。当然，这也就意味着可以支持事务，要么所有的命令全部执行成功，要么恢复到什么也没执行的状态。总之，如果有类似的需求时，利用命令模式分离请求者与实现者，是最明智的选择。

6.3.4、状态模式 (State)

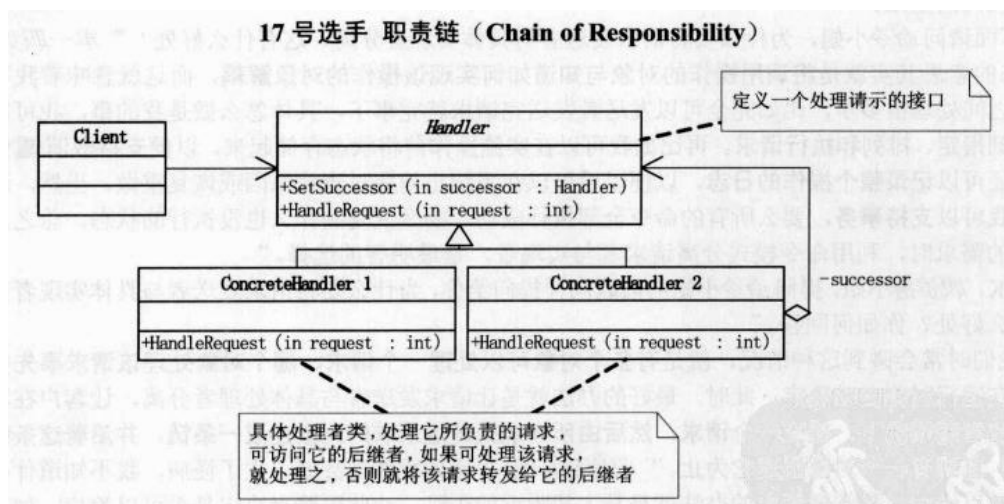
状态模式：允许一个对象在其内部状态改变时改变它的行为，让对象看起来似乎修改了它的类。



状态模式提供了一个更好的办法来组织与特定状态相关的代码，决定状态转移的逻辑不在单块的 if 或 switch 中，而是分布在各个状态子类之间，由于所有与状态相关的代码都存在于某个状态子类中，所以通过定义新的子类可以很容易地增加新的状态和转换。

6.3.5、职责链模式 (Chain of Responsibility)

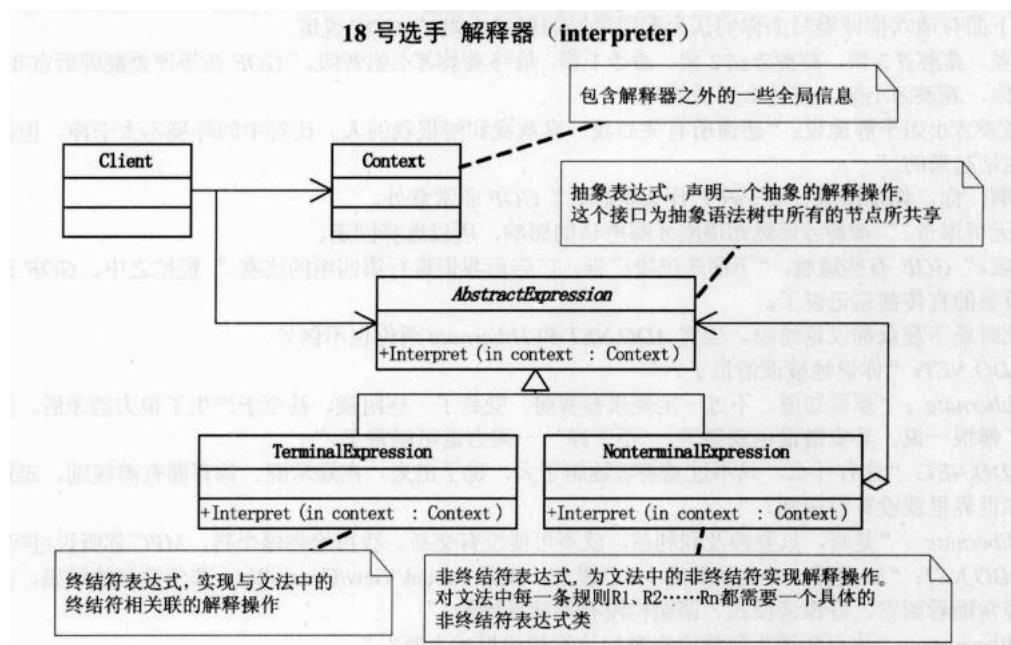
职责链模式：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。



我们时常会碰到这种情况,就是有多个对象可以处理一个请求,哪个对象处理该请求事先并不知道,要在运行时刻自动确定,此时,最好的办法就是让请求发送者与具体处理者分离,让客户在不明确指定接收者的情况下,提交一个请求,然后由所有能处理这请求的对象连成一条链,并沿着这条链传递该请求,直到有一个对象处理它为止。

6.3.6、解释器模式 (Interpreter)

解释器模式: 它声称给定一个语言,定义它的文法的一种表示,并定义一个解释器,这个解释器使用该表示来解释语言中的句子。

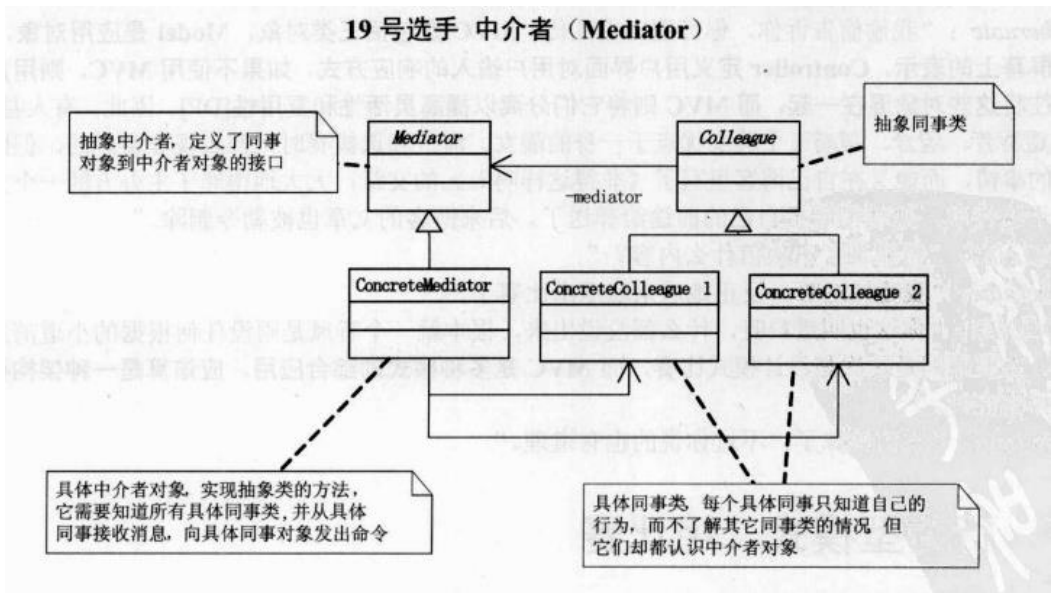


如果一种特定类型的问题发生的频率足够高,那么就可以考虑将该问题的各个实例表述为一个简单语言中的句子。也就是说,通过构建一个解释器,该解释器解释这些句子来解决

该问题。比如正则表达式就是描述字符串模式的一种标准语言，与其为每一个字符串模式都构造一个特定的算法，不如使用一种通用的搜索算法来解释执行一个正则表达式，该正则表达式定义了待匹配字符器的集合。

6.3.7、中介模式 (Mediator)

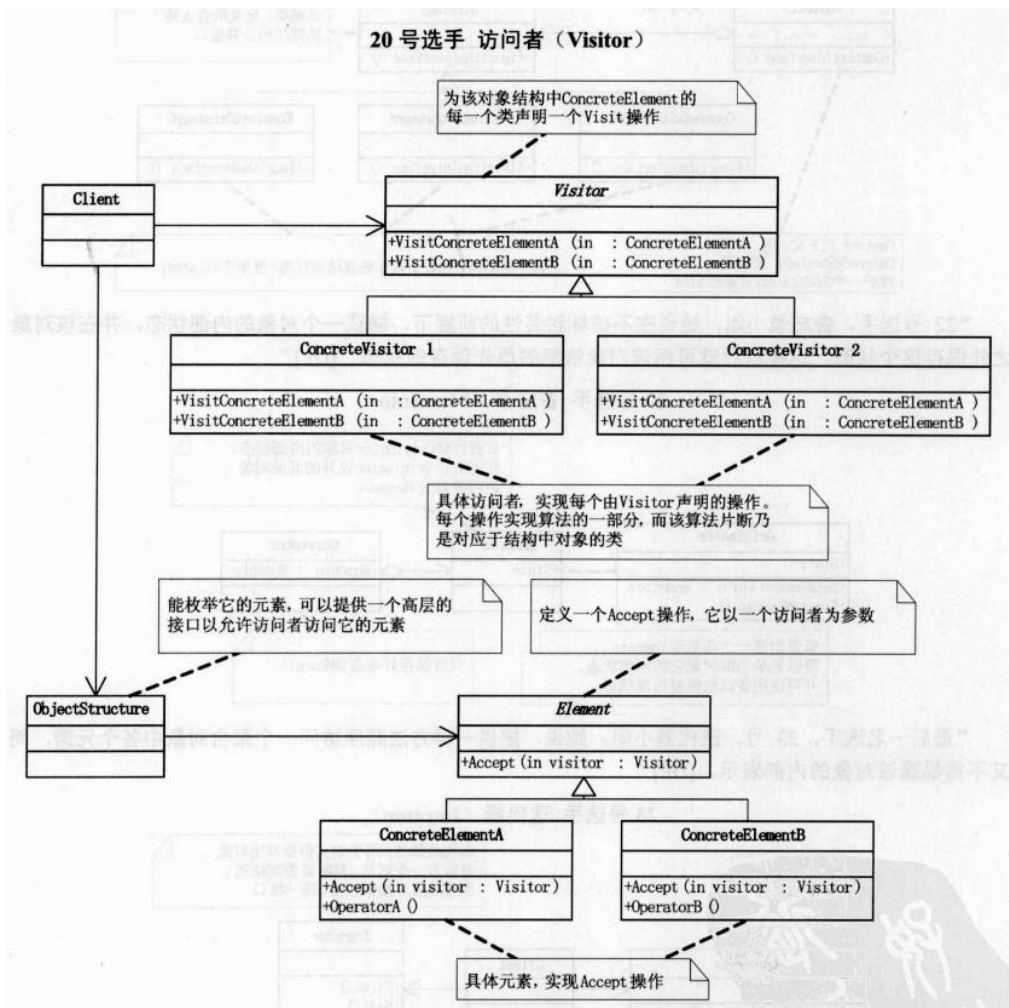
中介者模式：用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。



面向对象设计鼓励将行为分布到各个对象中，这种分布可能会导致对象间有许多连接。也就是说，有可能每一个对象都需要知道其他许多对象。对象间的大量相互连接使得一个对象似乎不太可能在没有其他对象的支持下工作，这对于应对变化是不利的，任何较大的改动都很困难。中介模式提倡将集体行为封装一个单独的中介者对象来避免这个问题，中介者负责控制和协调一组对象间的交互。中介者充当一个中介以使组中的对象不再相互显式引用。这些对象仅知道中介者，从而减少了相互连接的数目。

6.3.8、访问者模式 (Visitor)

访问者模式：表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。



访问者增加具体的 **Element** 是困难的，但增加依赖于复杂对象结构的构件的操作就变得容易。仅需增加一个新的访问者即可在一个对象结构上定义一个新的操作。

6.3.9、策略模式 (Strategy)

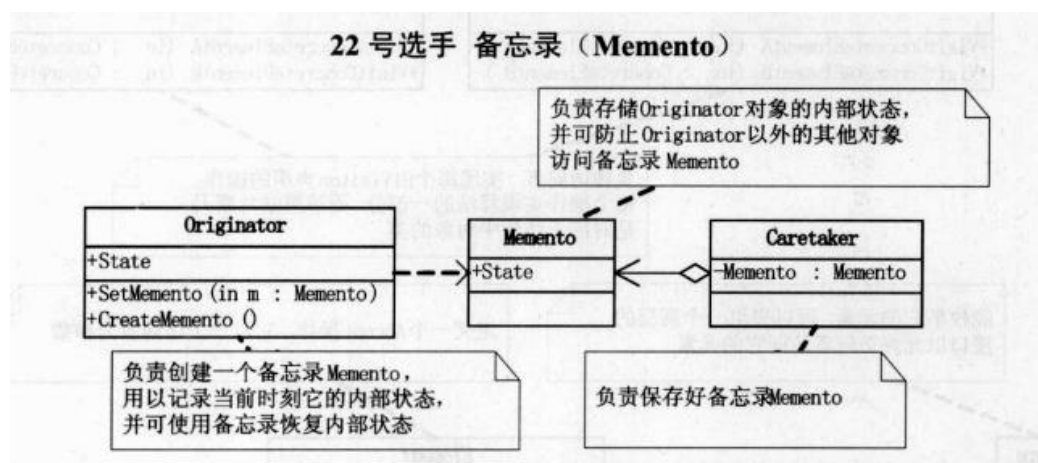
策略模式：定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。



继承提供了一种支持多种算法或行为的方法，我们可以直接生成一个类 A 的子类 B、C、D，从而给它以不同的行为。但这样会将行为强行编制到父类 A 当中，而将算法的实现与类 A 的实现混合起来，从而使得类 A 难以理解、难以维护和难以扩展，而且还不能动态地改变算法。仔细分析会发现，它们之间的唯一差别是它们所使用的算法或行为，将算法封装在独立的策略 **Strategy** 类中使得你可以独立于其类 A 改变它，使它易于切换、易于理解、易于扩展。

6.3.10、备忘录模式 (Memento)

备忘录模式：在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。这样以后就可将该对象恢复到原先保存的状态。

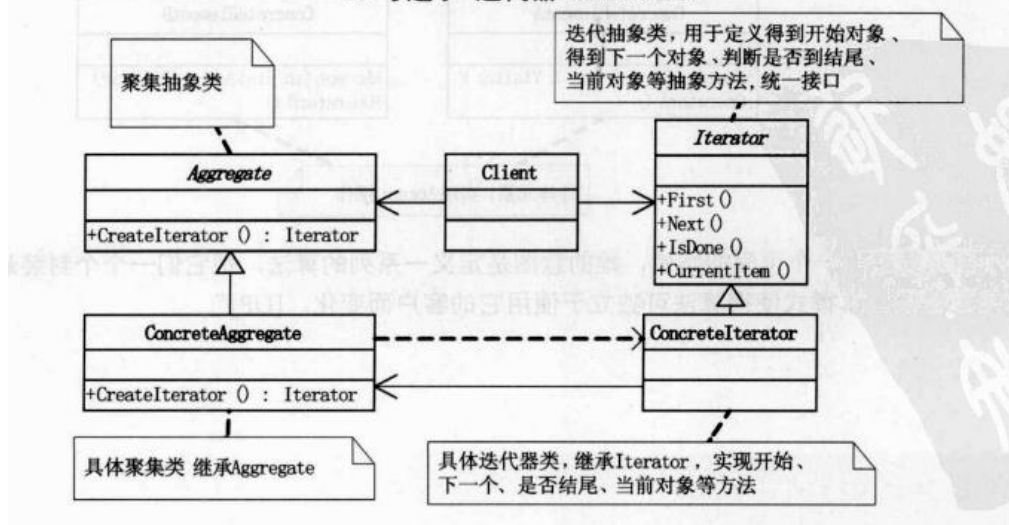


备忘录可以避免暴露一些只应由对象 A 管理却又必须存储在对象 A 之外的信息。备忘录模式把可能很复杂的对象 A 的内部信息对其他对象屏蔽起来，从而保持了封装边界。

6.3.11、迭代器模式 (Iterator)

迭代器模式：提供一种方法顺序访问一个聚合对象中各个元素，而又不需暴露该对象的内部表示。

23 号选手 迭代器 (Iterator)



迭代器模式的关键思想是将对列表的访问和遍历从列表对象中分离出来并放入一个迭代器对象中, 迭代器类定义了一个访问该列表元素的接口。迭代器对象负责跟踪当前的元素, 并且知道哪些元素已经遍历过了。