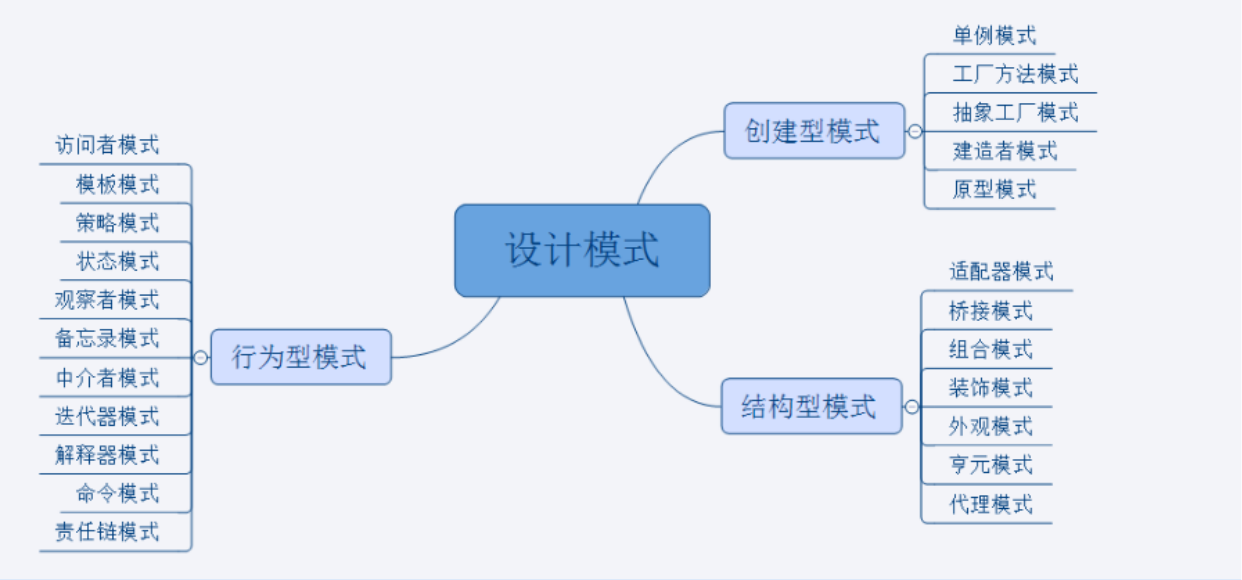
设计模式

基本概念

设计模式是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更为被别人理解，保证代码可靠性。毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理的运用设计模式可以完美的解决很多问题，每种模式在现在中都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因。

设计模式的分类

- 创建型模式：对象实例化的模式，创建型模式用于解耦对象的实例化过程。
- 结构型模式：把类或对象结合在一起形成一个更大的结构
- 行为型模式：类和对象如何交互，及划分责任和算法



23种设计模式的理解

link

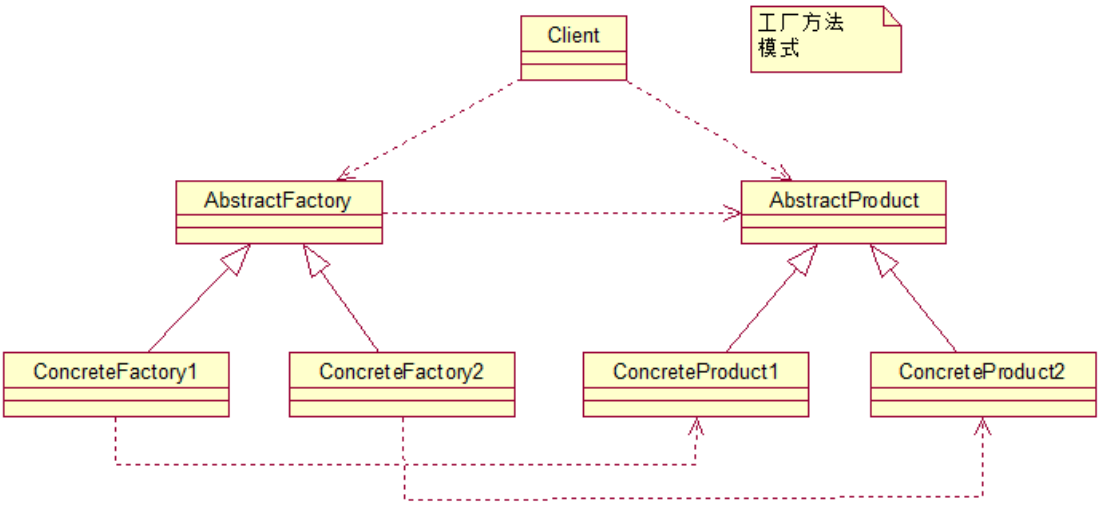
单例模式

它的定义就是确保某一个类只有一个实例，并且提供一个全局访问点。

单例模式的3个特点：1、只有一个实例。2、自我实例化。 3、提供全局访问点

工厂方法模式

模式的问题：如何能轻松方便地构造对象实例，而不关心构造对象的实例的细节和复杂过程。 **解决方案**：建立一个工厂来创建对象



抽象工厂模式

抽象工厂模式就是提供一个接口，用于创建相关或者依赖对象的家族，而不用明确指定具体类。

建造者模式

主要是将一个复杂对象的构建与表示分离。使得同样的构建过程可以创建不同的表示。

原型模式

在我们应用程序可能有某些对象的结构比较复杂，但是我们又需要频繁的使用它们，如果这个时候我们来不断的新建这个对象势必会大大损耗系统内存的，这个时候我们需要使用原型模式来对这个结构复杂又要频繁使用的对象进行克隆。所以原型模式就是用原型实例指定创建对象的种类，并且通过复制这些原型创建新的对象。

与创建新对象的过大成本相比，它的优点是简化了新对象的创建过程，提高了效率，同时原型模式提供了简化的创建结构

适配器模式

在我们的应用程序中我们可能需要将两个不同接口的类来进行通信，在不修改这两个的前提下我们可能会需要某个中间件来完成这个衔接的过程。这个中间件就是适配器。所谓适配器模式就是将一个类的接口，转换成客户期望的另一个接口。它可以让原本两个不兼容的接口能够无缝完成对接。

作为中间件的适配器将目标类和适配者解耦，增加了类的透明性和可复用性。

桥接模式

将抽象部分和实现部分隔离开来，使得他们能够独立变化。桥接模式将继承关系转化成关联关系，封装了变化，完成了解耦，减少了系统中类的数量，也减少了代码量。

组合模式

组合模式组合多个对象形成树形结构以表示“整体-部分”的结构层次

装饰模式

外观模式

哼元模式

代理模式

访问者模式

模板模式

策略模式

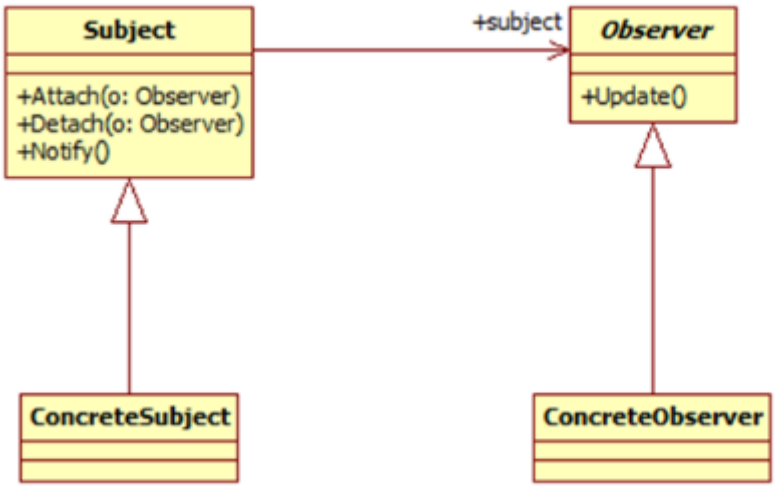
状态模式

观察者模式

观察者模式定义了对象之间的一对多依赖关系，这样一来，当一个对象改变状态时，它的所有依赖者都会收到通知并且自动更新。

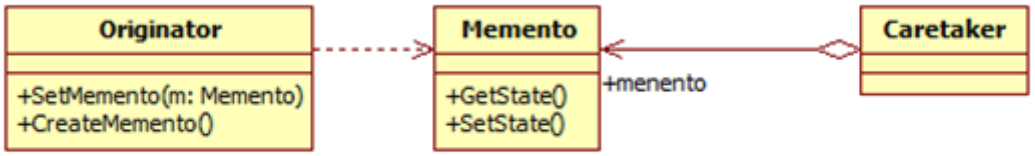
在这里，发生改变的对象称之为观察目标，而被通知的对象称之为观察者。一个观察目标可以对应多个观察者，而且这些观察者之间没有相互联系，所以么可以根据需要增加和删除观察者，使得系统更

易于扩展。所以观察者提供了一种对象设计，让主题和观察者之间以松耦合的方式结合。



备忘录模式

备忘录模式就是在不破坏封装的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，这样可以在以后将对象恢复到原先保存的状态。它实现了对信息的封装，使得客户不需要关心状态保存的细节。保存就要消耗资源，所以备忘录模式的缺点就在于消耗资源。如果类的成员变量过多，势必会占用比较大的资源，而且每一次保存都会消耗一定的内存。



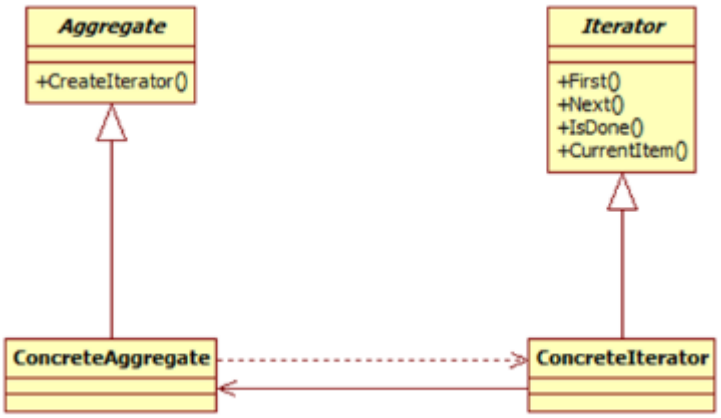
中介者模式

中介者模式就是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。在中介者模式中，中介对象用来封装对象之间的关系，各个对象可以不需要知道具体的信息通过中介者对象就可以实现相互通信。它减少了对象之间的互相关系，提供了系统可复用性，简化了系统的结构。

迭代器模式

所谓迭代器模式就是提供一种方法顺序访问一个聚合对象中的各个元素，而不是暴露其内部的表示。迭代器模式是将迭代元素的责任交给迭代器，而不是聚合对象，我们甚至在不需要知道该聚合对象的

内部结构就可以实现该聚合对象的迭代。

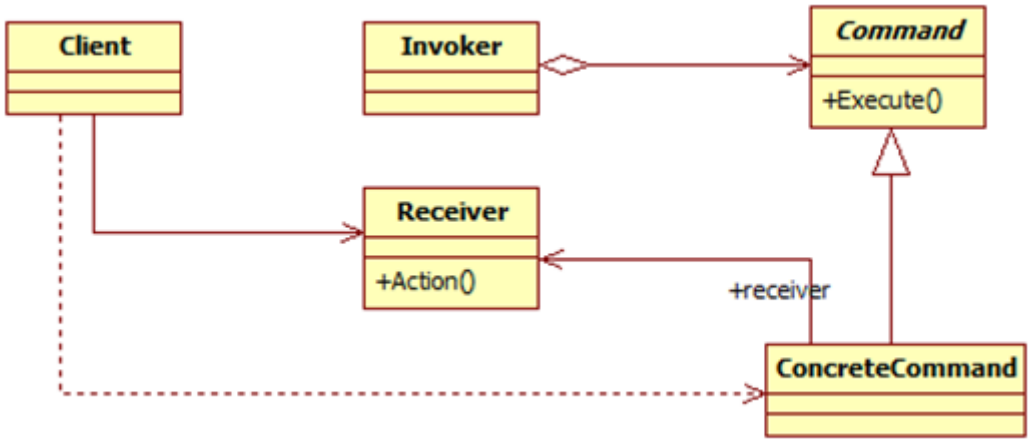


解释器模式

所谓解释器模式就是定义语言的文法，并且建立一个解释器来解释该语言中的句子。解释器模式描述了如何构成一个简单的语言解释器，主要应用在使用面向对象语言开发的编译器中。它描述了如何为简单的语言定义一个文法，如何在该语言中表示一个句子，以及如何解释这些句子。

命令模式

命令模式就是将同一个请求的发送者和接受者之间实现完全的解耦。Struts就是将请求和呈现分离的



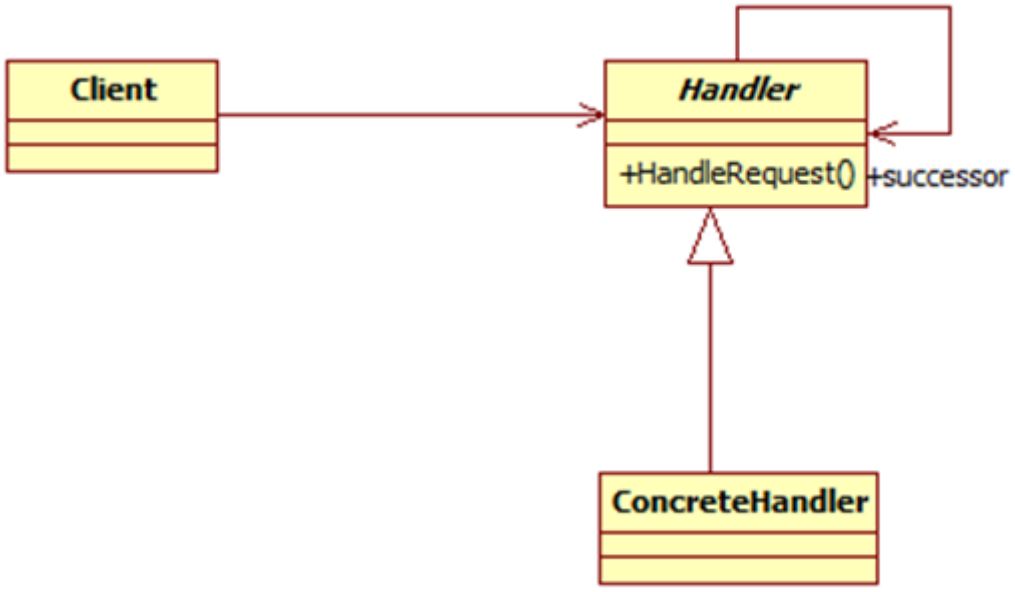
技术。

责任链模式

它描述请求如何沿着对象所组成的链来传递的。它将对象组成一条链，发送者将请求发给链的第一个接收者，并且沿着这条链传递，直到有一个对象来处理它或者直到最后也没有对象处理而留在链末尾端。

避免请求发送者与接收者耦合在一起，让多个对象都有可能接收请求，将这些对象连接成一条链，并且沿着这条链传递请求，直到有对象处理它为止，这就是职责链模式。在职责链模式中，使得每一个对象都有可能来处理请求，从而实现了请求的发送者和接收者之间的解耦。同时职责链模式简化了对象的结构，它使得每个对象都只需要引用它的后继者即可，而不必了解整条链，这样既提高了系统的灵活性也使得增加新的请求处理类也比较方便。但是在职责链中我们不能保证所有的请求都能够被处

理，而且不利于观察运行时特征



handler：抽象处理者 ConcreteHandler：具体处理者 Clien：客户类

单例模式

基本概念

特点

- 单例类只有一个类
- 单例类必须自己创建自己的唯一实例
- 单例类必须给所有其他对象提供这一实例

应用场景 单例模式确保某个类只有一个实例，而且自行实例化并向整个系统提供这个实例。在计算机系统中，线程池、缓存、日志对象、对话框、打印机、显卡的驱动程序对象常被设计成单例

简单单例模式（懒汉式单例模式）

Singleton的唯一实例，只能通过getInstance的方法区访问。当然可以通过java反射机制来实例化构造方法为private的类，此处不做讨论。

```
public class Singleton{
    //持有私有的静态实例，防止被引用，此处赋值null，目的在于实现延迟加载
    private static Singleton instance = null;
    //私有构造方法，防止被外部实例化
    private Singleton(){

    }
    //静态方法创建实例
    public static Singleton getInstance(){
```

```
        if(instance == null){
            instance = new Singleton;
        }
        return isntance;
    }

}
```

解决简单单例模式的线程安全问题

在getInstance方法上加上同步

```
public static synchronized Singleton getInstance(){
        if(instance == null){
            instance = new Singleton;
        }
        return isntance;
    }
```

双重检查锁定 但是，synchronized关键字锁住的是这个对象，这样的用法，在性能上会有所下降，因为每次调用getInstance()，都要对对象上锁。 事实上，只有在第一次创建对象的时候需要加锁，之后就不需要了，所以，这个地方需要改进。

```
      public static Singleton getInstance() {
          if (instance== null) {
              synchronized (Singleton.class) {
              if (instance== null) {
                  instance= new Singleton();
                }
              }
          }
             return instance;
       }
```

静态内部类 第一次加载Singleton类时并不会初始化Instance，只有第一次调用getInstance方法时虚拟机加载LazyHolder 并初始化Instance ，这样不仅能确保线程安全也能保证Singleton类的唯一性，所以推荐使用静态内部类单例模式

```
public class Singleton {
    private static class LazyHolder {
       private static final Singleton INSTANCE = new Singleton();
    }
    private Singleton (){}
    public static final Singleton getInstance() {
       return LazyHolder.INSTANCE;
    }
}
```

饿汉式单例

在创建类的同事就已经创建好一个静态的对象供系统使用，以后不再改变，所以天生是线程安全的。

```
public class Singleton {
    private Singleton() {}
    private static final Singleton instance= new Singleton1();
    //静态工厂方法
    public static Singleton getInstance() {
        return instance;
    }
```

总结

1、线程安全：

饿汉式天生就是线程安全的，可以直接用于多线程而不会出现问题，

懒汉式本身是非线程安全的，为了实现线程安全有几种写法，分别是上面的1、2、3，这三种实现在资源加载和性能方面有些区别。

2、资源加载和性能：

饿汉式在类创建的同时就实例化一个静态对象出来，不管之后会不会使用这个单例，都会占据一定的内存，但是相应的，在第一次调用时速度也会更快，因为其资源已经初始化完成，

而懒汉式顾名思义，会延迟加载，在第一次使用该单例的时候才会实例化对象出来，第一次调用时要做初始化，如果要做的工作比较多，性能上会有些延迟，之后就和饿汉式一样了。

至于1、2、3这三种实现又有些区别，

第1种，在方法调用上加了同步，虽然线程安全了，但是每次都要同步，会影响性能，毕竟99%的情况下是不需要同步的，

第2种，在getInstance中做了两次null检查，确保了只有第一次调用单例的时候才会做同步，这样也是线程安全的，同时避免了每次都同步的性能损耗

第3种，利用了classloader的机制来保证初始化instance时只有一个线程，所以也是线程安全的，同时没有性能损耗，所以一般我倾向于使用这一种。

工厂模式

简单工厂模式

工厂类 将创建产品的方法都放在工厂当中，客户只需要直接调用即可。即对实现了同一接口的类进行实例的创建。

```
public class SendFactory {


    public Sender produce(String type) {
        if ("mail".equals(type)) {
            return new MailSender();
        } else if ("sms".equals(type)) {
            return new SmsSender();
        } else {
            System.out.println("请输入正确的类型!");
            return null;
        }
    }

}
```

产品类 基于以共同的接口，或者抽象基类，不同的产品，实现不同的方法。

```
public interface Sender{
    public void Send();
}

public class MailSender implements Sender {
    @Override
    public void Send() {
        System.out.println("this is mailsender!");
    }
}
public class SmsSender implements Sender {


    @Override
    public void Send() {
        System.out.println("this is sms sender!");
    }
}
```

客户/测试类 新建一个工厂类，直接通过一个工厂进行不同产品的生产。不同方法的调用。

```
public class FactoryTest {


    public static void main(String[] args) {
        SendFactory factory = new SendFactory();
        Sender sender = factory.produce("sms");
        sender.Send();
```

```
        Sender sender1 = factory.produce("mail");
        sender1.Send();
    }
}
```

在这个简单工厂的基础上，还有以下修改

- 避免传参发生错误，不进行传参，而是直接通过方法名来return不同的类。也就是在工厂中创建多个方法，直接调用。

```
public class SendFactory {

    public Sender produceMail(){
        return new MailSender();
    }

    public Sender produceSms(){
        return new SmsSender();
    }
}

//测试

public class FactoryTest {


    public static void main(String[] args) {
        SendFactory factory = new SendFactory();
        Sender sender = factory.produceMail();
        sender.Send();
    }
}
```

- 静态工厂方法模式：将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可。

```
public class SendFactory {

    public static Sender produceMail(){
        return new MailSender();
    }

    public static Sender produceSms(){
        return new SmsSender();
    }
}

//测试
public class FactoryTest {


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

```
        Sender sender = SendFactory.produceMail();
        sender.Send();
    }
}
```

工厂方法模式

因为当我们需要增加或者减少方法的时候，除开新增一个产品的return方法后，还需要在facotry内部进行代码地修改。于是我们采用工厂方法类模式来解决问题。 将实例化对象的操作推迟到子类。而不是工厂中就进行new操作。

其实就是把工厂类从原先的一个类，变成一个接口，然后工厂以123分别是实现工厂接口，每个工厂123去对应具体的产品方法123。如果需要新增和减少方法，只需要新加两个类就好了，不用对原先的代码进行修改。

抽象工厂模式

抽象工厂模式是工厂方法模式的升级版本，他用来创建一组相关或者相互依赖的对象。比如宝马320系列使用空调型号A和发动机型号A，而宝马230系列使用空调型号B和发动机型号B，那么使用抽象工厂模式，在为320系列生产相关配件时，就无需制定配件的型号，它会自动根据车型生产对应的配件型号A。 **产品类**

```
//发动机以及型号
public interface Engine {

}
public class EngineA extends Engine{
    public EngineA(){
        System.out.println("制造-->EngineA");
    }
}
public class EngineBextends Engine{
    public EngineB(){
        System.out.println("制造-->EngineB");
    }
}

//空调以及型号
public interface Aircondition {

}
public class AirconditionA extends Aircondition{
    public AirconditionA(){
        System.out.println("制造-->AirconditionA");
    }
}
public class AirconditionB extends Aircondition{
    public AirconditionB(){
        System.out.println("制造-->AirconditionB");
    }
}
```

工厂类

```
//创建工厂的接口
public interface AbstractFactory {
    //制造发动机
    public Engine createEngine();
    //制造空调
    public Aircondition createAircondition();
}


//为宝马320系列生产配件
public class FactoryBMW320 implements AbstractFactory{

    @Override
    public Engine createEngine() {
        return new EngineA();
    }
    @Override
    public Aircondition createAircondition() {
        return new AirconditionA();
    }
}
//宝马523系列
public class FactoryBMW523 implements AbstractFactory {

     @Override
    public Engine createEngine() {
        return new EngineB();
    }
    @Override
    public Aircondition createAircondition() {
        return new AirconditionB();
    }


}
```

测试类

```
public class Customer {
    public static void main(String[] args){
        //生产宝马320系列配件
        FactoryBMW320 factoryBMW320 = new FactoryBMW320();
        factoryBMW320.createEngine();
        factoryBMW320.createAircondition();

        //生产宝马523系列配件
        FactoryBMW523 factoryBMW523 = new FactoryBMW523();
        factoryBMW320.createEngine();
        factoryBMW320.createAircondition();
    }
}
```

代理模式

基本概念

代理的定义是：为其他对象提供一个代理以控制对某个对象的访问，即通过代理对象来访问目标对象，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，扩展目标对象的功能。

注意事项： 1、和适配器模式的区别：适配器模式主要改变所考虑对象的接口，而代理模式不能改变所代理类的接口。 2、和装饰器模式的区别：装饰器模式为了增强功能，而代理模式是为了加以控制。

基本实现

主要实现的话有4个，公共接口，实现接口的实体类，我们基于实体类实现的代理类，以及测试类。

接口

```
public interface Image {
   void display();
}
```

实现接口的具体实现类

```
public class RealImage implements Image{
       private String fileName;

   public RealImage(String fileName){
      this.fileName = fileName;
      loadFromDisk(fileName);
   }

   @Override
   public void display() {
      System.out.println("Displaying " + fileName);
   }

   private void loadFromDisk(String fileName){
      System.out.println("Loading " + fileName);
   }
}
```

对实际实现类进行了修饰的代理类(也要实现相同接口)

```
public class ProxyImage implements Image{
    //私有具体实现类对象
    private RealImage realImage;
```

```
    private String fileName;

    public ProxyImage(String fileName){
       this.fileName = fileName;
    }

    //代理类的display方法
    @Override
    public void display() {
        if(realImage == null){
            realImage = new RealImage(fileName);
        }
        realImage.display();
    }
}
```

实际的调用

```
public class ProxyPatternDemo {

   public static void main(String[] args) {
      Image image = new ProxyImage("test_10mb.jpg");

      // 图像将从磁盘加载
      image.display();
      System.out.println("");
      // 图像不需要从磁盘加载
      image.display();
   }
}
```

jdk自带代理

可以结合jdk实现spring底层的aop代理来看，也是通过对具体实现类进行代理，但是在invoke方法的重写中，除开业务逻辑本身外，还可以在前后加上其他切面方法，如日志记录等。然后他是在代理类中，就新建了新的代理对象，所以在测试类中直接调用对应方法即可。

接口

```
public interface Subject {

    public int sellBooks();

    public String speak();
}
```

具体实现类

```
public class RealSubject implements Subject{
    @Override
```

```
    public int sellBooks() {
        System.out.println("卖书");
        return 1 ;
    }

    @Override
    public String speak() {
        System.out.println("说话");
        return "张三";
    }
}
```

代理类

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


public class MyInvocationHandler implements InvocationHandler {
    /**
     * 因为需要处理真实角色，所以要把真实角色传进来
     */
    Subject realSubject ;

    public MyInvocationHandler(Subject realSubject) {
        this.realSubject = realSubject;
    }

    /**
     *
     * @param proxy    代理类
     * @param method    正在调用的方法
     * @param args      方法的参数
     * @return
     * @throws Throwable
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("调用代理类");
        if(method.getName().equals("sellBooks")){
            int invoke = (int)method.invoke(realSubject, args);
            //可以对实际的方法进行其他方法的控制，比如日志记录等
            System.out.println("调用的是卖书的方法");
            return invoke ;
        }else {
            String string = (String) method.invoke(realSubject,args) ;
            System.out.println("调用的是说话的方法");
            return  string ;
        }
    }
}
```

测试类

```
import java.lang.reflect.Proxy;

public class Client {
    public static void main(String[] args) {
        //真实对象
        Subject realSubject =  new RealSubject();

        MyInvocationHandler myInvocationHandler = new MyInvocationHandler(realSubject);
        //代理类
         <!-- newProcyInstance(Classloader loader,Class<?>[],InvocationHandler h)
         参数1：当前类的类加载器
        参数2：代理类所需要实现的接口，假如是一个类，就是该类下所有方法
        参数3：处理类，一般写匿名类--!>

        Subject proxyClass = (Subject)
Proxy.newProxyInstance(ClassLoader.getSystemClassLoader(), new Class[]{Subject.class},
myInvocationHandler);

        proxyClass.sellBooks();

        proxyClass.speak();
    }
}
```

cglib代理

- cglib是通过将我们的目标类创建子类，然后每次执行的时候都会进行拦截，然后将子类方法注入运行并返回。 原理也是通通过实现代理类来进行拦截，加入增强方法后将原方法释放。但它创建代理类的方法和jdk不同，jdk生成的代理类只有一个，因为其被代理的目标是动态传入。而cglib是为每个目标类生成相应的子类，编译较慢，但在运行中因为已经静态编译生成，所以执行效率快。
- 还有在运行方法时，jdk的方法直接就是代理类属性了，但cglib的话还是原来的方法类
- jdk是针对有接口和实现类的，cglib是都可以

具体实现类

```
public class Engineer {
    // 可以被代理
    public void eat() {
        System.out.println("工程师正在吃饭");
    }

    // final 方法不会被生成的字类覆盖
    public final void work() {
        System.out.println("工程师正在工作");
    }

    // private 方法不会被生成的字类覆盖
    private void play() {
```

```
        System.out.println("this engineer is playing game");
    }
}
```

代理类

```
import net.sf.cglib.proxy.Enhancer;
import net.sf.cglib.proxy.MethodInterceptor;
import net.sf.cglib.proxy.MethodProxy;
import java.lang.reflect.Method;

public class CglibProxy implements MethodInterceptor {
    private Object target;

    public CglibProxy(Object target) {
        this.target = target;
    }

    @Override
    public Object intercept(Object o, Method method, Object[] objects, MethodProxy
methodProxy) throws Throwable {
        System.out.println("###   before invocation");
        Object result = method.invoke(target, objects);
        System.out.println("###   end invocation");
        return result;
    }

    public static Object getProxy(Object target) {
        Enhancer enhancer = new Enhancer();
        // 设置需要代理的对象
        enhancer.setSuperclass(target.getClass());
        // 设置代理人
        enhancer.setCallback(new CglibProxy(target));
        return enhancer.create();
    }
}
```

测试类

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

public class CglibMainTest {
    public static void main(String[] args) {
        // 生成 Cglib 代理类
        Engineer engineerProxy = (Engineer) CglibProxy.getProxy(new Engineer());
        // 调用相关方法
        engineerProxy.eat();
    }
}
```

设计模式在jdk中应用的相关包

Birdge 桥接模式： 这个模式将抽象和抽象操作的实现进行了解耦，这样使得抽象和实现可以独立地变化。 GOF在提出桥梁模式的时候指出，桥梁模式的用意是"将抽象化(Abstraction)与实现化(Implementation)脱耦，使得二者可以独立地变化"。这句话有三个关键词，也就是抽象化、实现化和脱耦。 在Java应用中，对于桥接模式有一个非常典型的例子，就是应用程序使用JDBC驱动程序进行开发的方式。所谓驱动程序，指的是按照预先约定好的接口来操作计算机系统或者是外围设备的程序。

Adapter 适配器模式： 用来把一个接口转化成另一个接口。使得原本由于接口不兼容而不能一起工作的那些类可以在一起工作。
java.util.Arrays#asList()
java.io.InputStreamReader(InputStream)
java.io.OutputStreamWriter(OutputStream)

Composite 组合模式： 又叫做部分-整体模式，使得客户端看来单个对象和对象的组合是同等的。换句话说，某个类型的方法同时也接受自身类型作为参数。
avax.swing.JComponent#add(Component)
java.util.Map#putAll(Map)
java.util.List#addAll(Collection)
java.util.Set#addAll(Collection)

装饰者模式：
动态的给一个对象附加额外的功能，这也是子类的一种替代方式。可以看到，在创建一个类型的时候，同时也传入同一类型的对象。这在JDK里随处可见，你会发现它无处不在，所以下面这个列表只是一小部分。
java.io.BufferedInputStream(InputStream)
java.io.DataInputStream(InputStream)
java.io.BufferedOutputStream(OutputStream
) java.util.zip.ZipOutputStream(OutputStream)
java.util.Collections#checkedList|Map|Set|SortedSet|SortedMap

Facade 门面模式，即外观模式： 给一组组件，接口，抽象，或者子系统提供一个简单的接口。
java.lang.Class
javax.faces.webapp.FacesServlet

Flyweight 享元模式：
使用缓存来加速大量小对象的访问时间。
java.lang.Integer#valueOf(int)
java.lang.Boolean#valueOf(boolean)
java.lang.Byte#valueOf(byte)
java.lang.Character#valueOf(char)

Proxy 代理模式： 代理模式是用一个简单的对象来代替一个复杂的或者创建耗时的对象。
java.lang.reflect.Proxy
RMI

Abstract Factory 抽象工厂模式:
抽象工厂模式提供了一个协议来生成一系列的相关或者独立的对象，而不用指定具体对象的类型。它使得应用程序能够和使用的框架的具体实现进行解耦。这在JDK或者许多框架比如Spring中都随处可见。它们也很容易识别，一个创建新对象的方法，返回的却是接口或者抽象类的，就是抽象工厂模式了。 java.util.Calendar#getInstance()
java.util.Arrays#asList()
java.util.ResourceBundle#getBundle()
java.sql.DriverManager#getConnection()
java.sql.Connection#createStatement()
java.sql.Statement#executeQuery()
java.text.NumberFormat#getInstance()
javax.xml.transform.TransformerFactory#newInstance()

Builder 建造者模式: 定义了一个新的类来构建另一个类的实例，以简化复杂对象的创建。建造模式通常也使用方法链接来实现。
java.lang.StringBuilder#append()
java.lang.StringBuffer#append()
java.sql.PreparedStatement
javax.swing.GroupLayout.Group#addComponent()

工厂方法： 就是一个返回具体对象的方法。
java.lang.Proxy#newProxyInstance()
java.lang.Object#toString()
java.lang.Class#newInstance()
java.lang.reflect.Array#newInstance()
java.lang.reflect.Constructor#newInstance() java.lang.Boolean#valueOf(String)
java.lang.Class#forName()

原型模式 使得类的实例能够生成自身的拷贝。如果创建一个对象的实例非常复杂且耗时时，就可以使用这种模式，而不重新创建一个新的实例，你可以拷贝一个对象并直接修改它。
java.lang.Object#clone()
java.lang.Cloneable

单例模式 用来确保类只有一个实例。Joshua Bloch在Effetive Java中建议到，还有一种方法就是使用枚举。 java.lang.Runtime#getRuntime()
java.awt.Toolkit#getDefaultToolkit()
java.awt.GraphicsEnvironment#getLocalGraphicsEnvironment()
java.awt.Desktop#getDesktop()

责任链模式 通过把请求从一个对象传递到链条中下一个对象的方式，直到请求被处理完毕，以实现对象间的解耦。

java.util.logging.Logger#log()
javax.servlet.Filter#doFilter()

命令模式 将操作封装到对象内，以便存储，传递和返回。

java.lang.Runnable
javax.swing.Action

解释器模式 这个模式通常定义了一个语言的语法，然后解析相应语法的语句。

java.util.Pattern
java.text.Normalizer
java.text.Format

迭代器模式 提供一个一致的方法来顺序访问集合中的对象，这个方法与底层的集合的具体实现无关。

java.util.Iterator
java.util.Enumeration

中介者模式 通过使用一个中间对象来进行消息分发以及减少类之间的直接依赖。

java.util.Timer
java.util.concurrent.Executor#execute()
java.util.concurrent.ExecutorService#submit()
java.lang.reflect.Method#invoke()

备忘录模式 生成对象状态的一个快照，以便对象可以恢复原始状态而不用暴露自身的内容。Date对象通过自身内部的一个long值来实现备忘录模式。

java.util.Date
java.io.Serializable

空对象模式 这个模式通过一个无意义的对象来代替没有对象这个状态。它使得你不用额外对空对象进行处理。

java.util.Collections#emptyList()
java.util.Collections#emptyMap()
java.util.Collections#emptySet()

观察者模式 它使得一个对象可以灵活的将消息发送给感兴趣的对象。

java.util.EventListener
javax.servlet.http.HttpSessionBindingListener
javax.servlet.http.HttpSessionAttributeListener
javax.faces.event.PhaseListener

状态模式 通过改变对象内部的状态，使得你可以在运行时动态改变一个对象的行为。

java.util.Iterator
javax.faces.lifecycle.LifeCycle#execute()

策略模式 使用这个模式来将一组算法封装成一系列对象。通过传递这些对象可以灵活的改变程序的功能。

java.util.Comparator#compare()
javax.servlet.http.HttpServlet
javax.servlet.Filter#doFilter()

模板方法模式 让子类可以重写方法的一部分，而不是整个重写，你可以控制子类需要重写那些操作。

java.util.Collections#sort()
java.io.InputStream#skip()
java.io.InputStream#read()
java.util.AbstractList#indexOf()

访问者模式 提供一个方便的可维护的方式来操作一组对象。它使得你在不改变操作的对象前提下，可以修改或者扩展对象的行为。

javax.lang.model.element.Element and javax.lang.model.element.ElementVisitor
javax.lang.model.type.TypeMirror and javax.lang.model.type.TypeVisitor