



Scala核心编程

-设计模式

尚硅谷研究院

● 学习设计模式的必要性

- 1) 面试会被问，所以必须学
- 2) 读源码时看到别人在用，尤其是一些框架大量使用到设计模式，不学看不懂源码为什么这样写，比如Runtime的单例模式。

```
public class Runtime {  
    private static Runtime currentRuntime = new Runtime();  
}
```



图片1.z

- 3) 设计模式能让专业人之间交流方便
- 4) 提高代码的易维护
- 5) 设计模式是编程经验的总结，我的理解：即通用的编程应用场景的模式化，套路化（站在**软件设计层面**思考）。

以单例模式说明

● 掌握设计模式的层次

- 1) 第1层：刚开始学编程不久，听说过什么是设计模式
- 2) 第2层：有很长时间的编程经验，自己写了很多代码，其中用到了设计模式，但是自己却不知道
- 3) 第3层：学习过了设计模式，发现自己已经在使用了，并且发现了一些新的模式挺好用的
- 4) 第4层：阅读了很多别人写的源码和框架，在其中看到别人设计模式，并且能够领会**设计模式的精妙和带来的好处**。
- 5) 第5层：代码写着写着，自己都没有意识到使用了设计模式，并且熟练的写了出来。



● 设计模式介绍

- 1) 设计模式是程序员在面对同类软件工程设计问题所总结出来的有用的经验，模式【设计，思想】不是代码，而是某类问题的通用解决方案，设计模式（Design pattern）代表了**最佳的实践**。这些解决方案是众多软件开发人员经过相当长的一段时间的试验和错误总结出来的。
- 2) 设计模式的本质提高 软件的维护性，通用性和扩展性，并降低软件的复杂度【软件巨兽=》软件工程】。
- 3) <<设计模式>> 是经典的书，作者是 Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides Design（俗称“四人组 GOF”）
- 4) 设计模式并不局限于某种语言，java，php，c++ 都有设计模式。

● 设计模式类型

设计模式分为三种类型，共23种

- 1) 创建型模式：单例模式、抽象工厂模式、建造者模式、工厂模式、原型模式。
- 2) 结构型模式：适配器模式、桥接模式、装饰模式、组合模式、外观模式、享元模式、代理模式。
- 3) 行为型模式：模版方法模式、命令模式、迭代器模式、观察者模式、中介者模式、备忘录模式、解释器模式（Interpreter模式）、状态模式、策略模式、职责链模式(责任链模式)、访问者模式。

注意：不同的书籍上对分类和名称略有差别

● 简单工厂模式

基本介绍

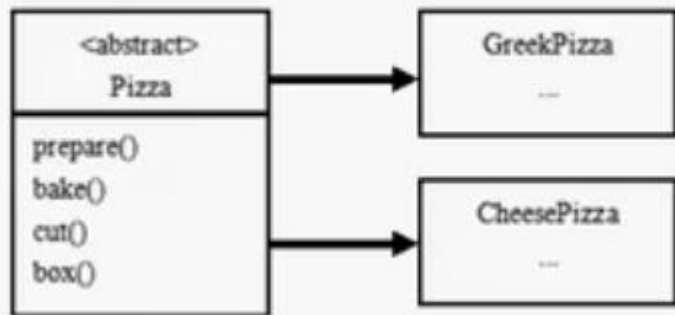
- 1) 简单工厂模式是属于创建型模式，但不属于23种GOF设计模式之一。**简单工厂模式是由一个工厂对象决定创建出哪一种产品类的实例**。简单工厂模式是工厂模式家族中最简单实用的模式
- 2) 简单工厂模式：定义了一个创建对象的类，由这个类来封装实例化对象的行为(代码)
- 3) 在软件开发中，当我们会用到大量的创建某种、某类或者某批对象时，就会使用到工厂模式。

- 简单工厂模式

看一个具体的需求

看一个披萨的项目：要便于披萨种类的扩展，要便于维护，完成披萨订购功能。

披萨簇的设计，如下：



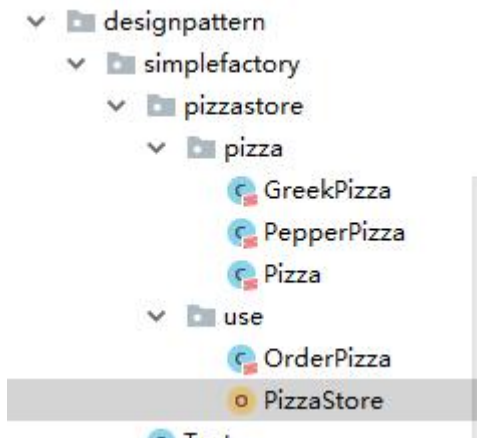
图片21.z



● 简单工厂模式

使用传统的方式来完成

具体看老师代码的演示



simplefactory.zip

```
abstract class Pizza { //写
    var name:String = _
    //假定，每种pizza 的准备原材料不同，因此做成抽象的..
    def prepare()
    def cut(): Unit = {
        println(this.name + " cutting ..")
    }
    def bake(): Unit = {
        println(this.name + " baking ..")
    }
    def box(): Unit = {
        println(this.name + " boxing ..")
    }
}
```

```
class GreekPizza extends Pizza{ //写
    override def prepare(): Unit = {
        this.name = "希腊pizza"
        println(this.name + " preparing..")
    }
}
```

```
class PepperPizza extends Pizza { //写
    override def prepare(): Unit = {
        this.name = "胡椒pizza"
        println(this.name + " preparing..")
    }
}
```

```
object PizzaStore extends App { //写测试程序
    val orderPizza = new OrderPizza
    println("退出程序....")
}
```

```
import scala.io.StdIn
import scala.util.control.Breaks._
import scala.io.StdIn
class OrderPizza {
    var orderType:String = _
    var pizza:Pizza = _
    breakable {
        do {
            println("请输入pizza的类型")
            orderType = StdIn.readLine()
            if (orderType.equals("greek"))
                this.pizza = new GreekPizza
            } else if (orderType.equals("p"))
                this.pizza = new PepperPizza
            } else {
                break()
            }
        }
        this.pizza.prepare()
        this.pizza.bake()
        this.pizza.cut()
        this.pizza.box()
    } while (true)
}
```




● 简单工厂模式

传统的方式的优缺点

- 1) 优点是比较好理解，简单易操作。
- 2) 缺点是违反了设计模式的ocp原则，即**对扩展开放，对修改关闭**。即当我们给类增加新功能的时候，尽量不修改代码，或者尽可能少修改代码。
- 3) 比如我们这时要新增加一个**Pizza**的种类(**Cheese**披萨)，我们需要做如下修改。

//新增 写

```
class CheesePizza extends Pizza{  
  override def prepare(): Unit = {  
    this.name = "奶酪pizza"  
    println(this.name + " preparing..")  
  }  
}
```

//增加一段代码 OrderPizza.scala //写

```
do {  
  println("请输入pizza的类型")  
  orderType = StdIn.readLine()  
  if (orderType.equals("greek")) {  
    this.pizza = new GreekPizza  
  } else if (orderType.equals("cheese")) {  
    this.pizza = new CheesePizza  
  } else {  
    break()  
  }  
}
```

- 简单工厂模式

传统的方式的优缺点

4) 改进的思路分析

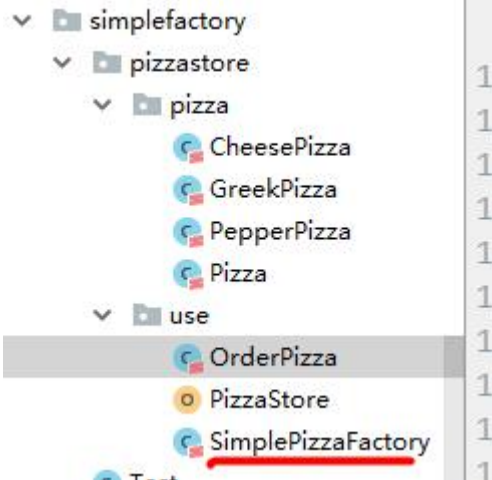
分析：修改代码可以接受，但是如果我们在其它的地方也有创建Pizza的代码，就意味着，也需要修改，而创建Pizza的代码，**往往有多处**。

思路：把创建Pizza对象封装到一个类中，这样我们有新的Pizza种类时，只需要修改该类就可，**其它有创建到Pizza对象的代码就不需要修改了**。→ 简单工厂模式

● 简单工厂模式

使用简单工厂模式

- 1) 简单工厂模式的设计方案: 定义一个实例化**Pizaa**对象的类, 封装创建对象的代码。
- 2) 看代码示例



```
class SimplePizzaFactory { //写
//说明。
def createPizza(t : String ): Pizza ={
    var pizza:Pizza = null
    if (t.equals("greek")) {
        pizza = new GreekPizza
    } else if (t.equals("pepper")) {
        pizza = new PepperPizza
    } else if (t.equals("cheese")) {
        pizza = new CheesePizza
    }
    return pizza
}}
```

```
orderType = StdIn.readLine()
//使用简单工厂模式来创建对象。
pizza = simplePizzaFactory.
createPizza(orderType)
if (pizza == null) {
    break()
}
```

● 工厂方法模式

看一个新的需求

披萨项目新的需求：客户在点披萨时，可以点不同口味的披萨，比如 北京的奶酪pizza、北京的胡椒pizza 或者是伦敦的奶酪pizza、伦敦的胡椒pizza。

思路1

使用简单工厂模式，创建不同的简单工厂类，比如BJPizzaSimpleFactory、LDPizzaSimpleFactory 等等.从当前这个案例来说，也是可以的，但是考虑到项目的规模，以及软件的可维护性、可扩展性并不是特别好的方

思路2

使用工厂方法模式

- 工厂方法模式

工厂方法模式介绍

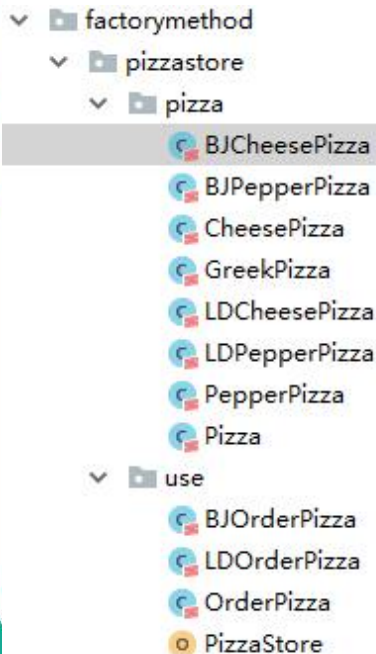
工厂方法模式设计方案：将披萨项目的实例化功能抽象成抽象方法，在不同的口味点餐子类中具体实现。

工厂方法模式：定义了一个创建对象的抽象方法，由子类决定要实例化的类。工厂方法模式将**对象的实例化推迟到子类**。

● 工厂方法模式

工厂方法模式应用案例

披萨项目新的需求：客户在点披萨时，可以点不同口味的披萨，比如 北京的奶酪pizza、北京的胡椒pizza 或者是伦敦的奶酪pizza、伦敦的胡椒pizza



```
class BJCheesePizza extends Pizza { //写
    override def prepare(): Unit = {
        this.name = "北京的奶酪pizza"
        println(this.name + " preparing..")
    }
}
```

```
class BJPepperPizza extends Pizza { //写
    override def prepare(): Unit = {
        this.name = "北京的胡椒pizza"
        println(this.name + " preparing..")
    }
}
```



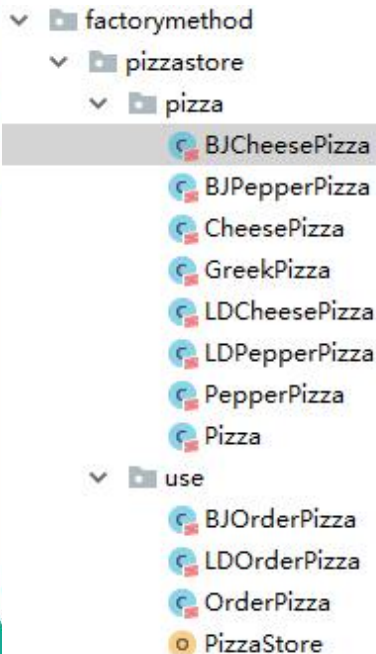
图片23.z



factorymethod.zip

● 工厂方法模式

工厂方法模式应用案例



```
abstract class OrderPizza { //改写
    breakable {
        var orderType: String = null
        var pizza: Pizza = null
    }
    do {
        println("请输入pizza的类型 ,使用工厂方法模式...")
        orderType = StdIn.readLine()
        //使用简单工厂模式来创建对象.
        pizza = createPizza(orderType)
        if (pizza == null) {
            break()
        }
        pizza.prepare()
        pizza.bake()
        pizza.cut()
        pizza.box()
    } while (true)
}
//抽象方法
def createPizza(t: String): Pizza
}
```

```
class BJOrderPizza extends OrderPizza {
    override def createPizza(t: String): Pizza {
        var pizza: Pizza = null
        if (t.equals("cheese")) {
            pizza = new BJCheesePizza
        } else if (t.equals("pepper")) {
            pizza = new BJPepperPizza
        }
        pizza
    }
}
```



● 抽象工厂模式

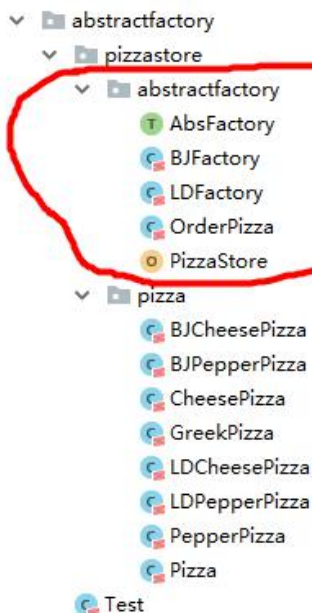
基本介绍

- 1) 抽象工厂模式：定义了一个**trait**用于创建相关或有依赖关系的对象簇，而无需指明具体的类
- 2) 抽象工厂模式可以将**简单工厂模式**和**工厂方法模式**进行整合。
- 3) 从设计层面看，抽象工厂模式就是对简单工厂模式的改进(或者称为进一步的抽象)。
- 4) 将工厂抽象成两层，**AbsFactory**(抽象工厂) 和 具体实现的工厂子类。程序员可以根据创建对象类型使用对应的工厂子类。这样将单个的简单工厂类变成了**工厂簇**，更利于代码的维护和扩展。

● 抽象工厂模式

抽象工厂模式应用实例

使用抽象工厂模式来完成披萨项目。



```
class BJFactory extends AbsFactory {  
  override def createPizza(t: String):  
    pizza.Pizza = {  
      var pizza: Pizza = null  
      if (t.equals("cheese")) {  
        pizza = new BJCheesePizza  
      } else if (t.equals("pepper")) {  
        pizza = new BJPepperPizza  
      }  
      return pizza  
    }  
}
```

//还有一个**LDFactory**开发和BJFactory类似.



abstractfactory.zip

```
//声明一个特质，类似java的接口  
trait AbsFactory {  
  //一个抽象方法  
  def createPizza(t : String ): Pizza  
}
```



● 抽象工厂模式

抽象工厂模式应用实例

```
object PizzaStore extends App {  
  val orderPizza =  
    new OrderPizza(new BJFactory)  
  //val orderPizza =  
  //  new OrderPizza(new LDFactory)  
  println("退出程序....")  
}
```

```
class OrderPizza { //  
  var absFactory :AbsFactory = _  
  def this(absFactory: AbsFactory) {//多态  
    this  
    breakable {  
      var orderType: String = null  
      var pizza: Pizza = null  
      do {  
        println("请输入pizza的类型 ,使用抽象工厂模式...")  
        orderType = StdIn.readLine()  
        //使用简单工厂模式来创建对象.  
        pizza = absFactory.createPizza(orderType)  
        if (pizza == null) {  
          break()  
        }  
        pizza.prepare()  
        pizza.bake()  
        pizza.cut()  
        pizza.box()  
      } while (true)  
    }  
  }  
}
```

● 工厂模式小结

1) 工厂模式的意义

将实例化对象的代码提取出来，放到一个类中统一管理和维护，达到和主项目的依赖关系的解耦。从而提高项目的扩展和维护性。

2) 三种工厂模式

3) 设计模式的**依赖抽象**原则

- 创建对象实例时，不要直接 **new** 类，而是把这个**new** 类的动作放在一个工厂的方法中，并返回。也有的书上说，变量不要直接持有具体类的引用。
- 不要让类继承具体类，而是继承抽象类或者是**trait**（接口）
- 不要覆盖基类中已经实现的方法。



● 单例模式

什么是单例模式

单例模式是指：保证在整个的软件系统中，某个类只能存在一个对象实例。

单例模式的应用场景

比如Hibernate的SessionFactory，它充当数据存储源的代理，并负责创建Session对象。SessionFactory并不是轻量级的，一般情况下，一个项目通常只需要一个SessionFactory就够，这是就会使用到单例模式。

Akka [ActorySystem 单例]



- 单例模式

单例模式的应用案例

Scala中没有静态的概念，所以为了实现Java中单例模式的功能，可以直接采用类对象(即伴生对象)方式构建单例对象



● 单例模式

单例模式的应用案例

1) 方式1-懒汉式

```
object TestSingleTon extends App{  
    val singleTon = SingleTon.getInstance  
    val singleTon2 = SingleTon.getInstance  
    println(singleTon.hashCode() + " " + singleTon2.hashCode())  
}
```

//将SingleTon的构造方法私有化

```
class SingleTon private() {}
```

```
object SingleTon {  
    private var s:SingleTon = null  
    def getInstance = {  
        if(s == null) {  
            s = new SingleTon  
        }  
        s}}  
s}}
```



● 单例模式

单例模式的应用案例

2) 方式2-饿汉式

```
object TestSingleTon extends App {  
  val singleTon = SingleTon.getInstance  
  val singleTon2 = SingleTon.getInstance  
  println(singleTon.hashCode() + " ~ " + singleTon2.hashCode())  
  println(singleTon == singleTon2)  
}
```

//将SingleTon的构造方法私有化

```
class SingleTon private() {  
  println("~~~")  
}
```

```
object SingleTon {  
  private val s: SingleTon = new SingleTon  
  def getInstance = {  
    s}}  
//说明： 饿汉式
```

● 装饰者模式(Decorator)

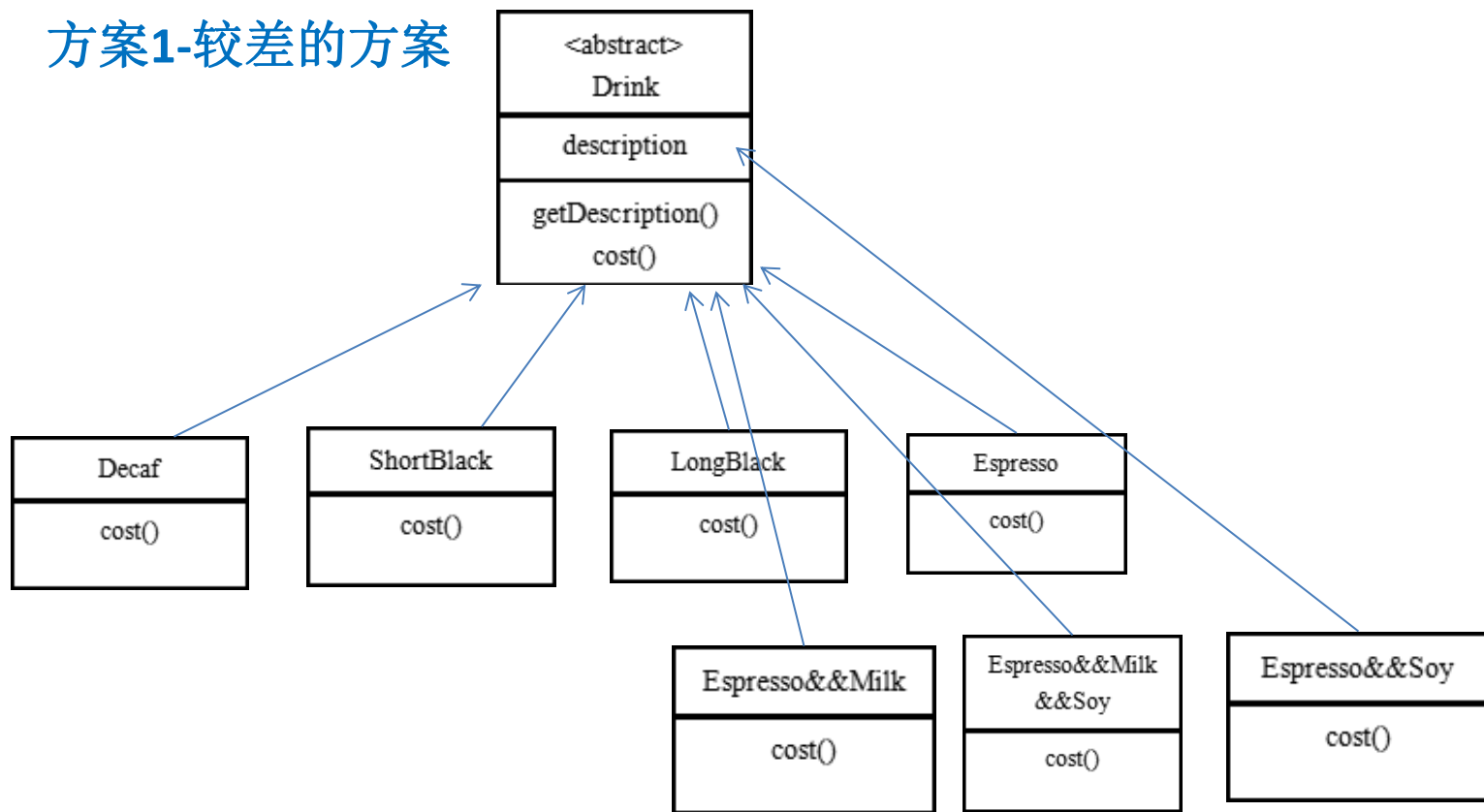
看一个项目需求

咖啡馆订单系统项目（咖啡馆）：

- 1) 咖啡种类/单品咖啡：Espresso(意大利浓咖啡)、ShortBlack、LongBlack(美式咖啡)、Decaf(无因咖啡)
- 2) 调料：Milk、Soy(豆浆)、Chocolate
- 3) 要求在扩展**新的咖啡种类**时，具有良好的扩展性、改动方便、维护方便
- 4) 使用OO的来计算不同种类咖啡的**费用**：客户可以点**单品咖啡**，也可以**单品咖啡+调料组合**。

● 装饰者模式(Decorator)

方案1-较差的方案



● 装饰者模式(Decorator)

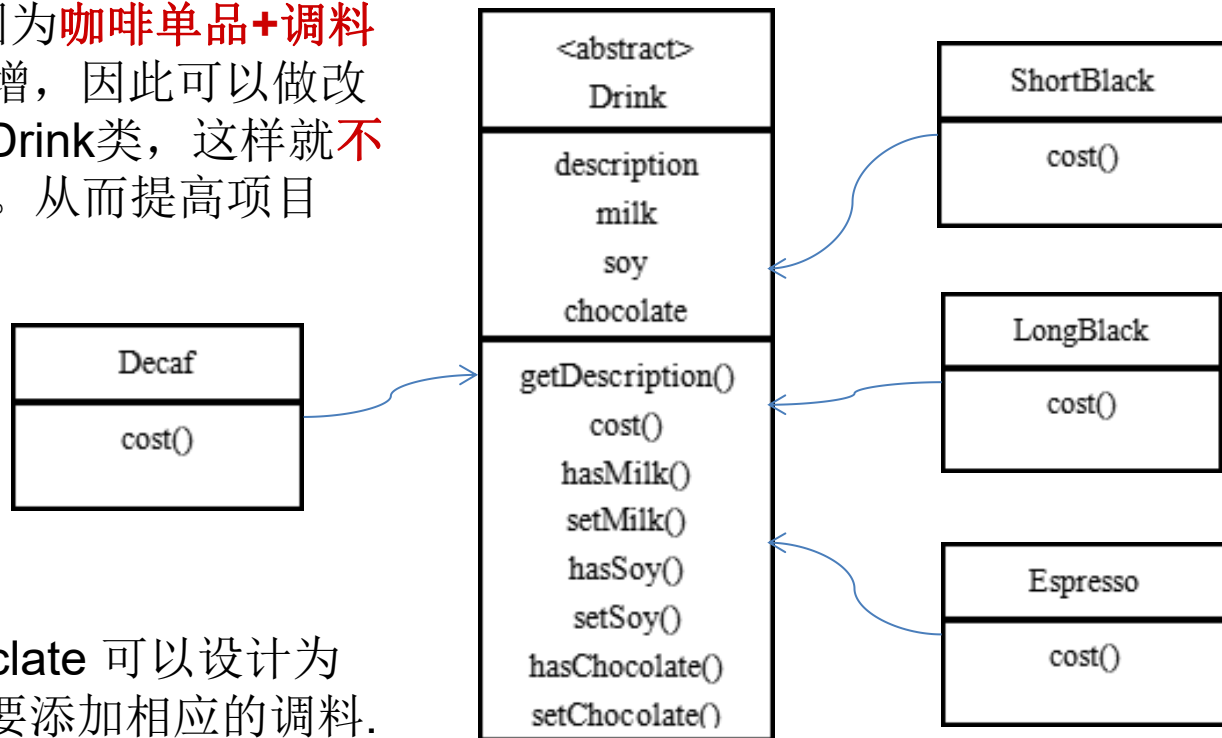
方案1-小结和分析

- 1) Drink 是一个抽象类，表示饮料
- 2) description就是描述，比如咖啡的名字等
- 3) cost就是计算费用，是一个抽象方法
- 4) Decaf 等等就是具体的单品咖啡，继承Drink,并实现cost方法
- 5) Espresso&&Milk 等等就是单品咖啡+各种调料的组合,这个会很多..
- 6) 这种设计方式时，会有很多的类，并且当增加一个新的单品咖啡或者调料时，类的数量就会倍增(类爆炸)

● 装饰者模式(Decorator)

方案2-好点的方案

前面分析到方案1因为**咖啡单品+调料**组合会造成类的倍增，因此可以做改进，将调料内置到Drink类，这样就**不会造成类数量过多**。从而提高项目的维护性(如图)



说明: milk,soy,chocolate 可以设计为 Boolean,表示是否要添加相应的调料.



图27.zi



- 装饰者模式(Decorator)

方案2-的问题分析

- 1) 方案2可以控制类的数量，不至于造成过多的类。
- 2) 在增删调料种类时，代码维护量仍然很大。
- 3) 考虑到添加多份调料时，可以将Boolean 改成 Int

● 装饰者模式(Decorator)

装饰者模式原理

1) 装饰者模式就像**打包一个快递**

➢ 主体：比如：陶瓷、衣服 (Component)

➢ 包装：比如：报纸填充、塑料泡沫、纸板、木板(Decorator)

2) Component

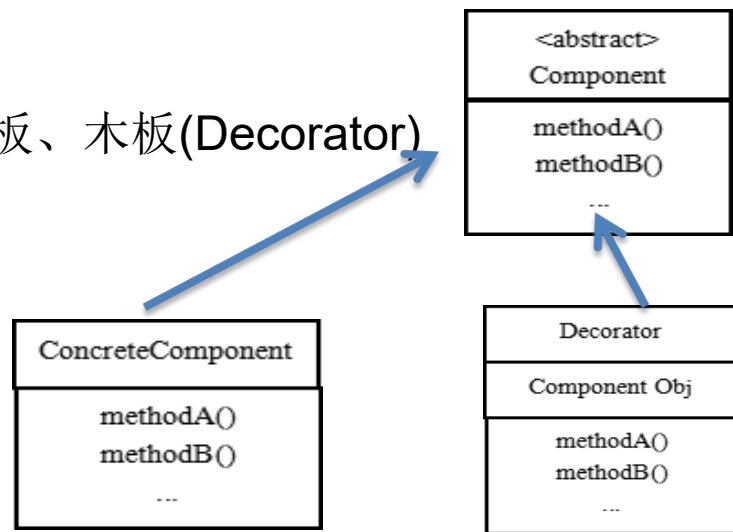
主体：比如类似前面的Drink

3) ConcreteComponent和Decorator

ConcreteComponent: 具体的主体,
比如前面的各个单品咖啡

Decorator: 装饰者, 比如各调料.

4) 在如图的**Component与ConcreteComponent之间**, 如果
ConcreteComponent类很多,还可以设计一个缓冲层, 将共有的部分提取出来, 抽象层一个类。





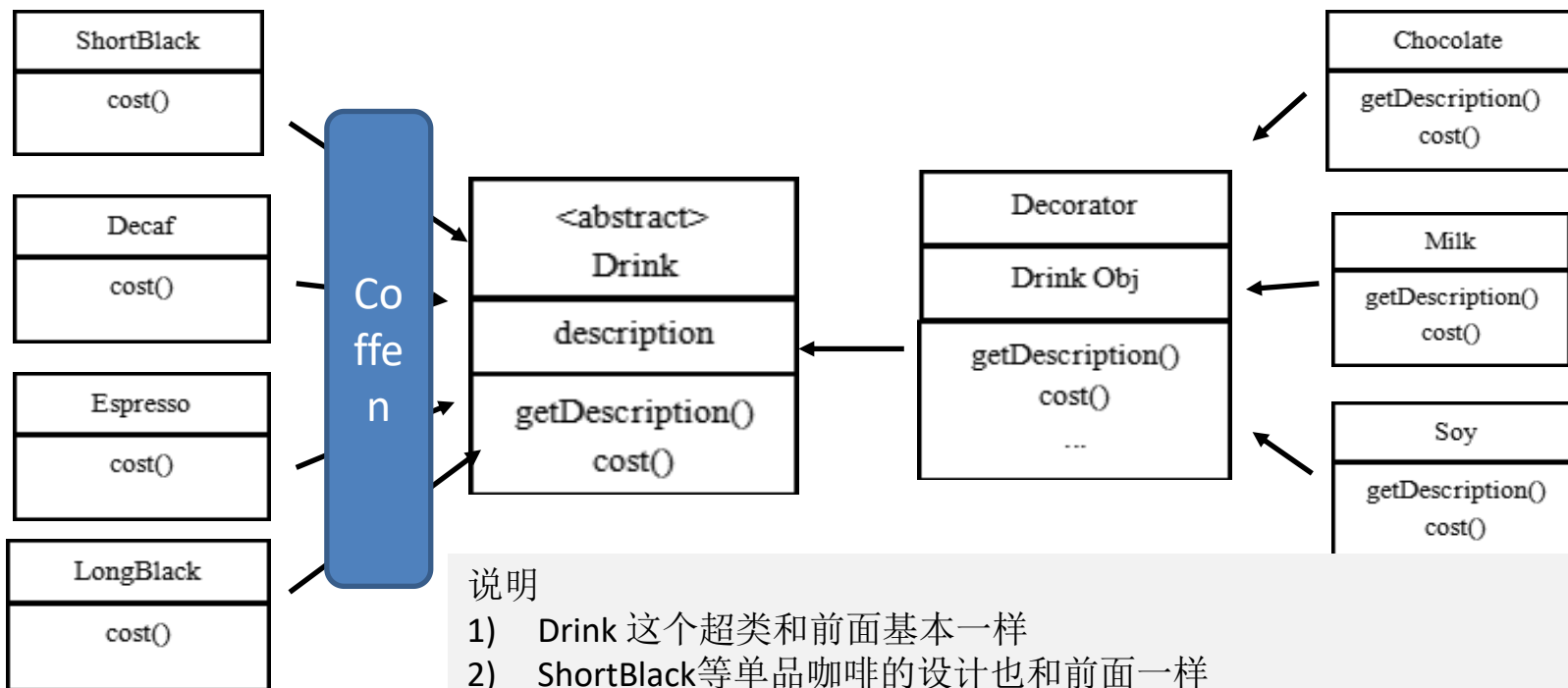
- 装饰者模式(Decorator)

装饰者模式定义

- 1) 装饰者模式：**动态的**将新功能**附加到对象上**。在对象功能扩展方面，它比继承更有弹性，装饰者模式也体现了开闭原则(ocp)
- 2) 这里提到的**动态的将新功能附加到对象**和**ocp原则**，在后面的应用实例上会以代码的形式体现，请同学们注意体会。

● 装饰者模式(Decorator)

用装饰者模式设计重新设计的方案

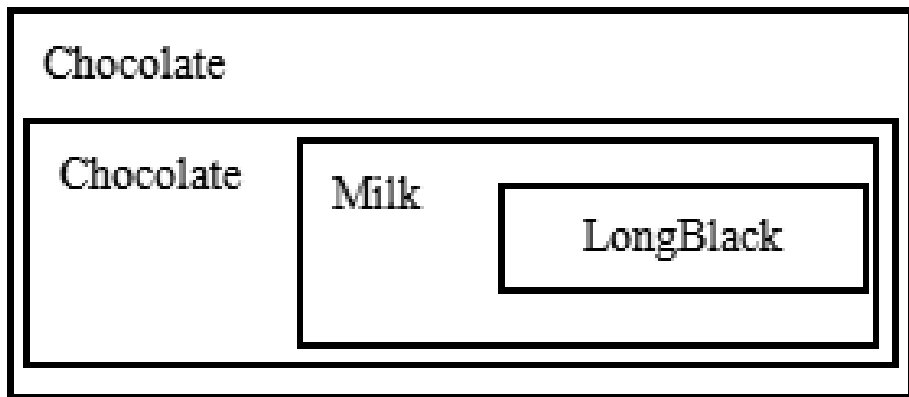


说明

- 1) Drink 这个超类和前面基本一样
- 2) ShortBlack等单品咖啡的设计也和前面一样
- 3) Decorator 是一个装饰类，含义一个被装饰的对象(Drink obj)
- 4) Decorator 的cost 进行费用的叠加，递归计算出价格.

• 装饰者模式(Decorator)

装饰者模式下的订单：2份巧克力+一份牛奶的LongBlack



- 1) 有一份 Milk + LongBlack 【milk装饰LongBlack】
- 2) 使用一份Chocolate 装饰 [份 Milk + LongBlack]
- 3) 使用一份chocolate 装饰[chocolate+Milk+LongBlack]
- 4) 当计算cost要递归的计算



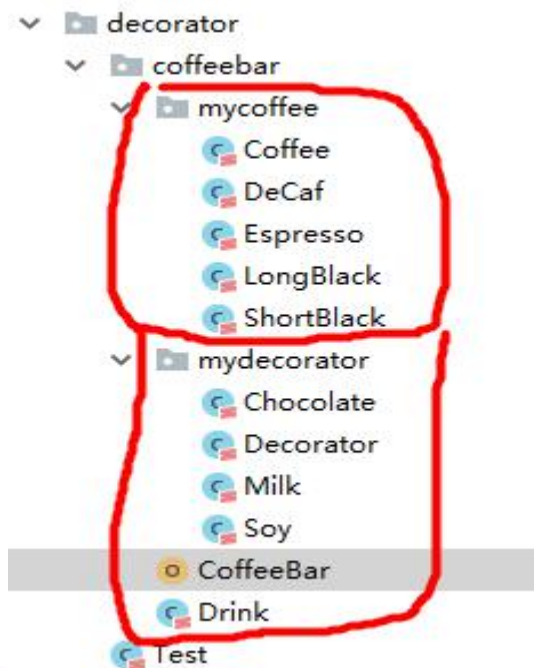
图片37.z

说明

● 装饰者模式(Decorator)

装饰者模式咖啡订单项目应用实例

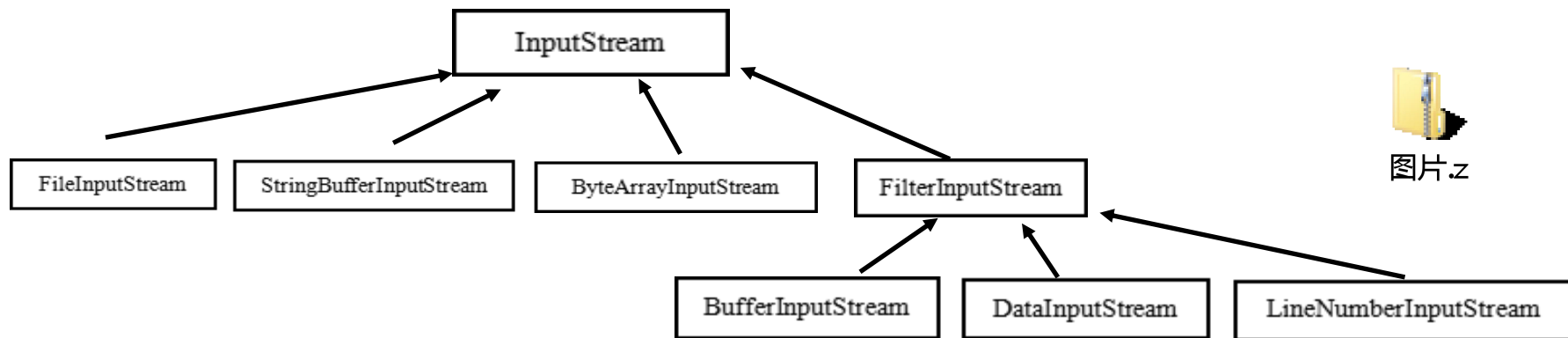
咖啡订单项目包结构



- 装饰者模式(Decorator)

Java中装饰者模式的经典使用

Java的IO结构，FilterInputStream就是一个装饰者



```
public abstract class InputStream implements Closeable //是一个抽象类，即Component
public class FilterInputStream extends InputStream { //是一个装饰者类Decorator
    protected volatile InputStream in //被装饰的对象
```



● 观察者模式(Observer)

看一个项目需求

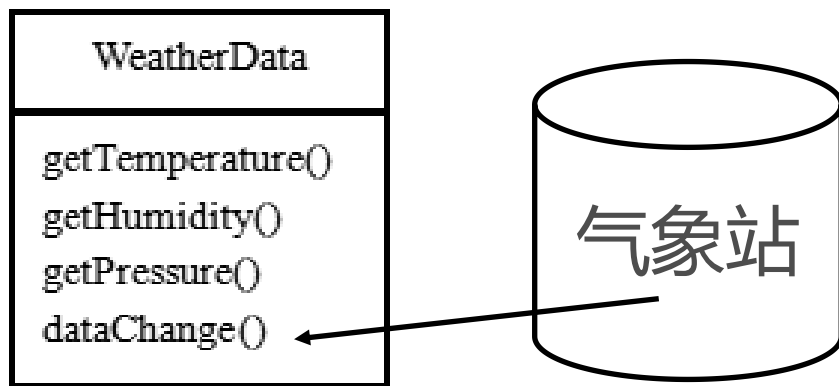
气象站项目，具体要求如下：

- 1) 气象站可以将每天测量到的温度，湿度，气压等等以公告的形式发布出去(比如发布到自己的网站)。
- 2) 需要设计开放型**API**，便于其他第三方公司也能接入气象站获取数据。
- 3) 提供温度、气压和湿度的接口
- 4) 测量数据更新时，要能实时的通知给第三方

● 观察者模式(Observer)

WeatherData类

通过对气象站项目的分析，我们可以初步设计出一个WeatherData类



图片38.z

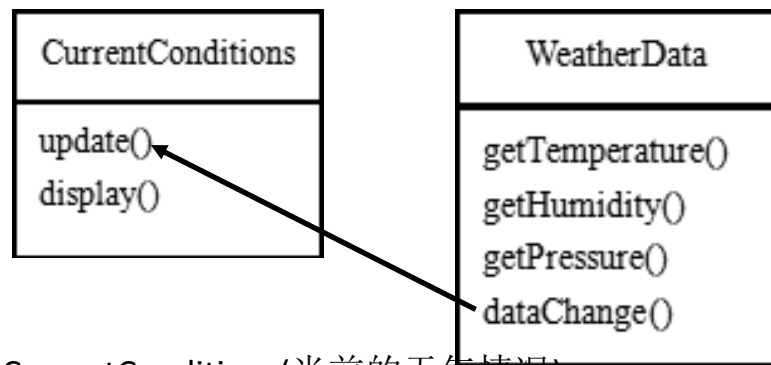
说明:

- 1) 通过getXxx方法，可以让第三方公司接入，并得到相关信息.
- 2) 当数据有更新时，气象站通过调用dataChange() 去更新数据，当第三方再次获取时，就能得到最新数据，当然也可以**推送**。

- 观察者模式(Observer)

气象站设计方案1-普通方案

➤ 示意图



CurrentConditions(当前的天气情况)
可以理解成是我们气象局的
网站 //推送





● 观察者模式(Observer)

气象站设计方案1-普通方案

➤ 代码实现



```
object InternetWeather {  
  def main(args: Array[String]): Unit = {  
    val mCurrentConditions = new CurrentConditions()  
    val mWeatherData = new WeatherData(mCurrentConditions)  
    mWeatherData.setData(30, 150, 40)  
  }  
}
```

```
class CurrentConditions {  
  private var mTemperature: Float = _  
  private var mPressure: Float = _  
  private var mHumidity: Float = _  
  def display() = {  
    println("***Today mTemperature: " + mTemperature + "***")  
    println("***Today mPressure: " + mPressure + "***")  
    println("***Today mHumidity: " + mHumidity + "***")  
  }  
  def update(mTemperature: Float, mPressure: Float, mHumidity: Float) = {  
    this.mTemperature = mTemperature  
    this.mPressure = mPressure  
    this.mHumidity = mHumidity  
    display()}}  
}
```

```
class WeatherData {  
  private var mTemperature: Float = _  
  private var mPressure: Float = _  
  private var mHumidity: Float = _  
  private var mCurrentConditions: CurrentConditions = _  
  def this(mCurrentConditions: CurrentConditions) {  
    this  
    this.mCurrentConditions = mCurrentConditions  
  }  
  def getTemperature() = {  
    mTemperature}  
  def getPressure() = {  
    mPressure}  
  def getHumidity() = {  
    mHumidity}  
  def dataChange() = {  
    mCurrentConditions.update(getTemperature(), getPressure(), getHumidity())}  
  def setData(mTemperature: Float, mPressure: Float, mHumidity: Float) = {  
    this.mTemperature = mTemperature  
    this.mPressure = mPressure  
    this.mHumidity = mHumidity  
    dataChange()  
  }  
}
```

● 观察者模式(Observer)

气象站设计方案1-普通方案

➤ 问题分析

- 1) 其他第三方公司接入气象站获取数据的问题
- 2) 无法在运行时动态的添加第三方

//在WeatherData 中, 当增加一个第三方, 都需要创建一个对应的第三方的公告板对象, 并加入到dataChange, **不利于维护, 也不是动态加入**

```
def dataChange() = {  
    mCurrentConditions.update(getTemperature(), getPressure(), getHumidity())  
}
```

● 观察者模式(Observer)

观察者模式原理

➤ 观察者模式类似订牛奶业务

1) 奶站/气象局: Subject

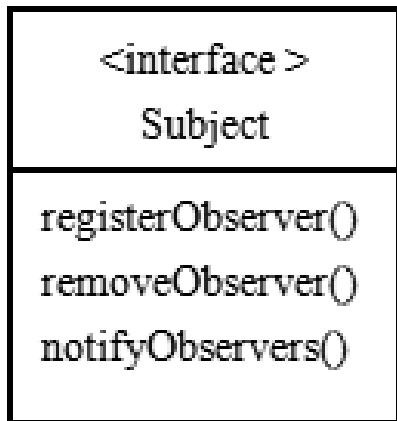
2) 用户/第三方网站: Observer

➤ Subject: 登记注册、移除和通知

1) registerObserver 注册

2) removeObserver 移除

3) notifyObservers() 通知所有的注册的用户, 根据不同需求, 可以是更新数据, 让用户来取, 也可能是实施推送, 看具体需求定

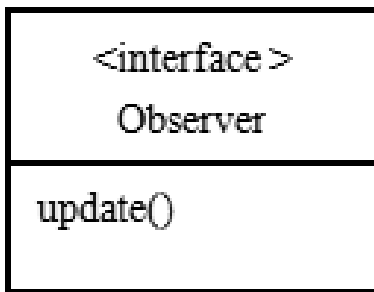


图片40.z

● 观察者模式(Observer)

观察者模式原理

➤ Observer: 接收输入



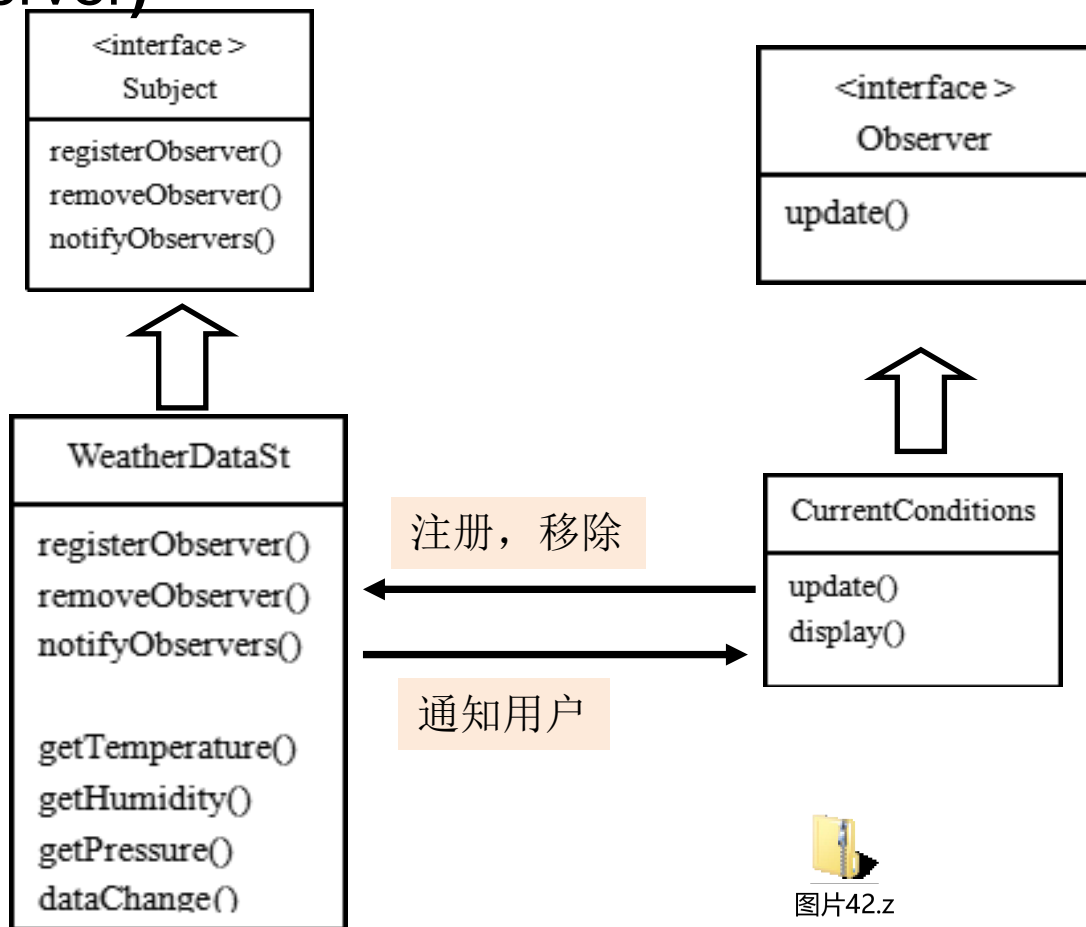
图片41.z

➤ 观察者模式：对象之间**多对一依赖**的一种设计方案，被依赖的对象为**Subject**，依赖的对象为**Observer**，**Subject**通知**Observer**变化,比如这里的奶站是**Subject**，是**1**的一方。用户时**Observer**，是多的一方。

● 观察者模式(Observer)

气象站设计方案2- 观察者模式

➤ 设计类图：



图片42.z



- 观察者模式(Observer)

气象站设计方案2-观察者模式

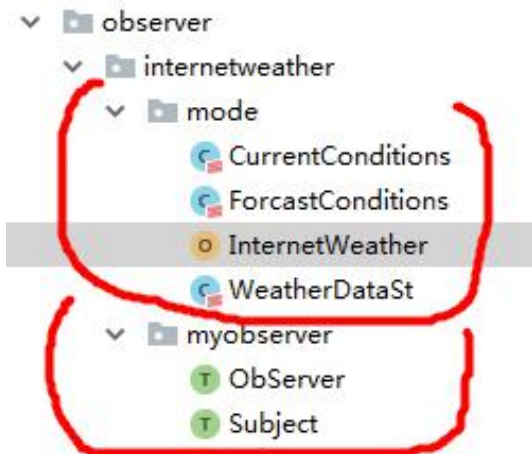
➤ 观察者模式的好处

- 1) 观察者模式设计后，会以集合的方式来管理用户(Observer)，包括注册，移除和通知。
- 2) 这样，我们增加观察者(这里可以理解成一个新的公告板)，就不需要去修改核心类WeatherDataSt/不会修改代码。它可以作为一个独立的进程保持运行，无需重新加载。

● 观察者模式(Observer)

气象站设计方案2-观察者模式

➤ 代码实现



图片3.z



observer.zip

```
trait Subject {
```

```
  def registerObserver(o: Observer)
```

```
  def removeObserver(o: Observer)
```

```
  def notifyObservers()
```

```
}
```

```
trait Observer {
```

```
  def update(mTemperature: Float, mPressure: Float,  
            mHumidity: Float)
```

```
}
```



● 观察者模式(Observer)

气象站设计方案2-观察者模式

➤ 代码实现

```
class CurrentConditions extends Observer { // 观察者
    private var mTemperature: Float = _
    private var mPressure: Float = _
    private var mHumidity: Float = _

    override def update(mTemperature: Float, mPressure: Float,
        mHumidity: Float): Unit = {
        this.mTemperature = mTemperature
        this.mPressure = mPressure
        this.mHumidity = mHumidity
        display()
    }
    def display() = {
        println("***Today mTemperature: " + mTemperature + "****")
        println("***Today mPressure: " + mPressure + "****")
        println("***Today mHumidity: " + mHumidity + "****")
    }
}
```

```
class ForcastConditions extends Observer { // 观察者
    private var mTemperature: Float = _
    private var mPressure: Float = _
    private var mHumidity: Float = _
    override def update(mTemperature: Float, mPressure: Float, mHumidity: Float): Unit = {
        this.mTemperature = mTemperature
        this.mPressure = mPressure
        this.mHumidity = mHumidity
        display()
    }
    def display() = {
        println("***明天温度:" + (mTemperature + Math.random()) + "****")
        println("***明天气压:" + (mPressure + 10 * Math.random()) + "****")
        println("***明天湿度:" + (mHumidity + Math.random()) + "****")
    }
}
```



● 观察者模式(Observer)

气象站设计方案2-观察者模式

➤ 代码实现

```
object InternetWeather { //测试
  def main(args: Array[String]): Unit = {
    val mWeatherDataSt = new WeatherDataSt()
    val mCurrentConditions = new CurrentConditions()
    val mForcastConditions = new ForcastConditions()
    mWeatherDataSt.registerObserver(mCurrentConditions)
    mWeatherDataSt.registerObserver(mForcastConditions)
    mWeatherDataSt.setData(30, 150, 40)

    mWeatherDataSt.removeObserver(mCurrentConditions)
    mWeatherDataSt.setData(40, 250, 50)
  }
}
```

```
class WeatherDataSt extends Subject { //核心Subject实现类 //写
  private var mTemperature: Float = _
  private var mPressure: Float = _
  private var mHumidity: Float = _
  private val mObservers: ListBuffer[Observer] = ListBuffer()
  def getTemperature() = {
    mTemperature
  }
  def getPressure() = {
    mPressure
  }
  def getHumidity() = {
    mHumidity
  }
  def dataChange() = {
    notifyObservers();
  }
  def setData(mTemperature: Float, mPressure: Float, mHumidity: Float) = {
    this.mTemperature = mTemperature
    this.mPressure = mPressure
    this.mHumidity = mHumidity
    dataChange()
  }
  override def registerObserver(o: Observer): Unit = {
    mObservers.append(o)
  }
  override def removeObserver(o: Observer): Unit = {
    if (mObservers.contains(o)) {
      mObservers -= o //移除
    }
  }
  override def notifyObservers(): Unit = {
    for (item <- mObservers) {
      item.update(getTemperature(), getPressure(), getHumidity())
    }
  }
}
```


- 观察者模式(Observer)

Java内置观察者模式

➤ java.util.Observable

- 1) Observable 的作用和地位等价于，我们讲的Subject
- 2) Observable 是类，不是接口，已经实现了核心的方法 注册，移除和通知。

```
//  
public class Observable {  
    private boolean changed = false;  
    private Vector<Observer> obs;  
  
    /** Construct an Observable with zero Observers. */  
  
    public Observable() { obs = new Vector<>(); }
```



图片11.z



- 观察者模式(Observer)

Java内置观察者模式

➤ java.util.Observer

- 1) Observer 的作用和地位等价于我们讲的Observer
- 2) java.util.Observer的源码如下:

```
public interface Observer {  
    void update(Observable o, Object arg);  
}
```

- 3) Observable和Observer 的使用方法和前面讲的案例基本一样，只是Observable是类，通过继承来实现观察者模式。



● 代理模式(Proxy)

代码模式的基本介绍

- 1) 代理模式：为一个对象**提供一个替身**，以控制对这个对象的访问
- 2) 被代理的对象可以是**远程对象**、**创建开销大的对象**或**需要安全控制的对象**
- 3) 代理模式有不同的形式(比如 远程代理，静态代理，动态代理)，都是为了控制与管理对象访问。

● 代理模式(Proxy)

看一个项目需求

糖果机项目，具体要求如下：

- 1) 某公司需要将销售糖果的糖果机放置到**本地(本地监控)**和**外地(远程监控)**，进行糖果销售。
- 2) 给糖果机插入硬币，**转动手柄**，这样就可以购买糖果。
- 3) 可以监控糖果机的状态和销售情况。

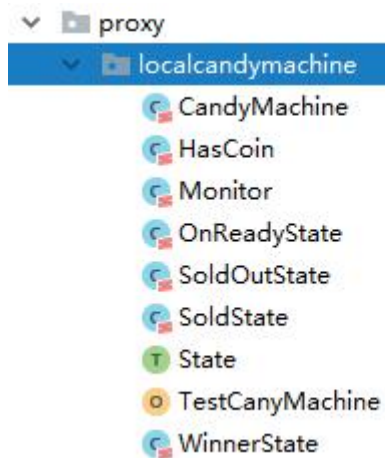




● 代理模式(Proxy)

完成监控本地糖果机

对**本地糖果机**的状态和销售情况进行监控，相对比较简单，完成该功能



图片1.z



localcandymachine.zip



● 代理模式(Proxy)

完成监控远程糖果机

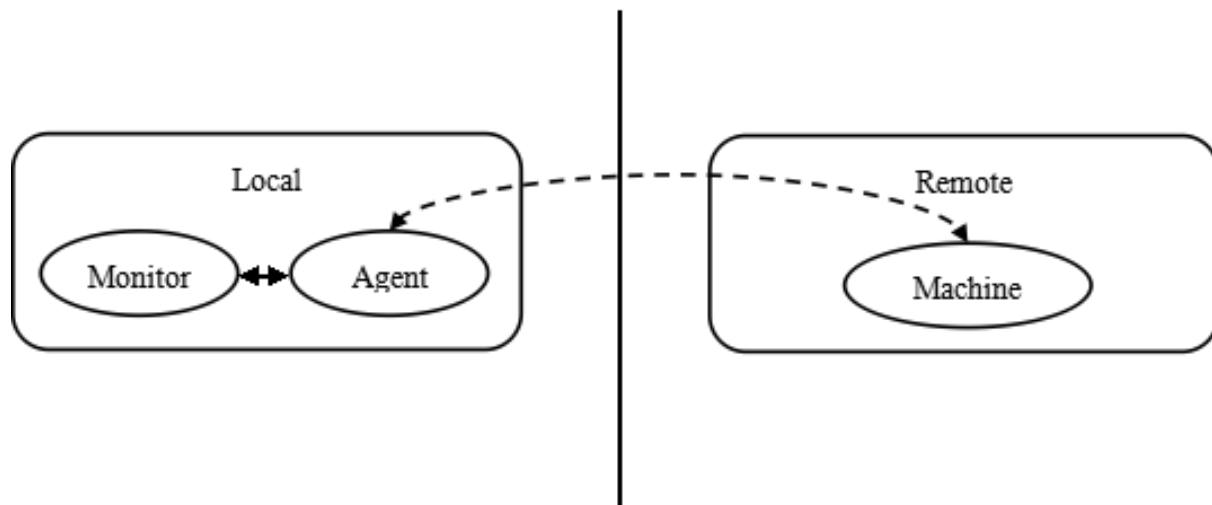
对**远程糖果机**的状态和销售情况进行监控，相对麻烦些，我们先分析一下

- 1) 方式1：因为远程糖果机不在本地，比如在另外的城市，国家，这时可以使用**socket**编程来进行网络编程控制(缺点：麻烦)
- 2) 方案2：在远程放置**web**服务器，通过**web**编程来实现远程监控。
- 3) 方案3：使用**RMI**(Remote Method Invocation)远程方法调用来完成对远程糖果机的监控，因为**RMI**将**socket**的底层封装起来，对外提供调用方法接口即可，这样比较简单，这样我们就可以实现**远程代理模式**开发。

- 代理模式(Proxy)

远程代理模式监控方案

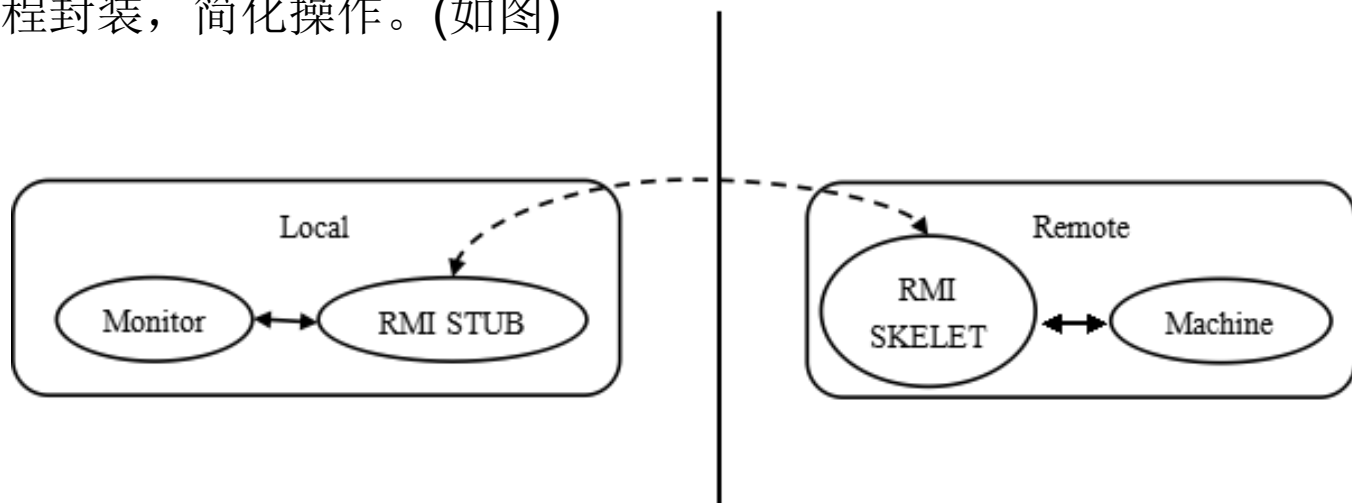
远程代理：远程对象的本地代表，通过它可以**把远程对象当本地对象**来调用。
远程代理通过网络和真正的远程对象沟通信息。



- 代理模式(Proxy)

Java RMI实现远程代理

RMI 指的是远程方法调用 (Remote Method Invocation)。它是一种机制，能够让在某个 Java虚拟机上的对象调用另一个 Java 虚拟机中的对象上的方法。可以用此方法调用的任何对象必须实现该远程接口，RMI可以将底层的socket编程封装，简化操作。(如图)



图片3.z



● 代理模式(Proxy)

Java RMI的介绍

- 1) RMI远程方法调用是计算机之间通过网络实现对象调用的一种通讯机制。
- 2) 使用RMI机制，一台计算机上的对象可以调用另外 一台计算机上的对象来获取远程数据。
- 3) RMI被设计成一种面向对象开发方式，允许程序员使用远程对象来实现通信

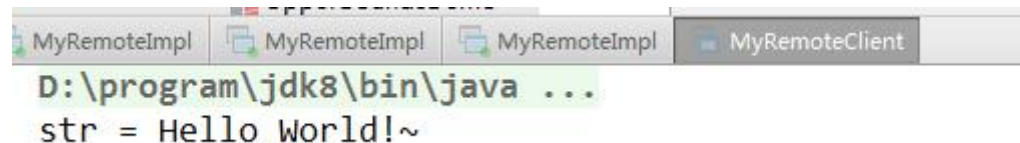
- 代理模式(Proxy)

Java RMI的开发应用案例-说明

请编写一个JavaRMI的案例，代理端(客户端)可以通过rmi远程调用 远程端注册的一个服务的sayHello的方法，并且返回结果。



```
D:\program\jdk8\bin\java ...  
远程服务开启, 在127.0.0.1 的 9999端口监听, 服务名 RemoteHello
```



```
D:\program\jdk8\bin\java ...  
str = Hello World!~
```



2.zip



- 代理模式(Proxy)

Java RMI的开发应用案例-开发步骤

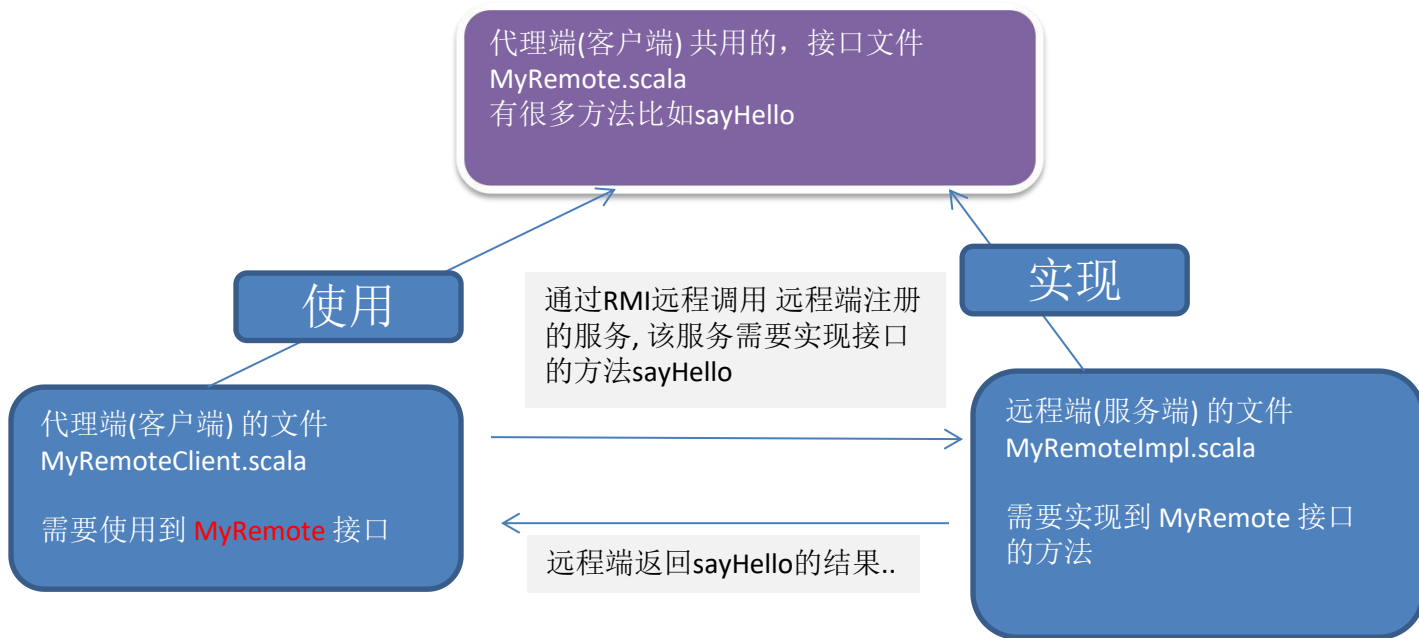
- 1) 制作远程接口：接口文件
- 2) 远程接口的实现：**Service**文件
- 3) RMI服务端注册，开启服务
- 4) RMI代理端通过RMI查询到服务端，建立联系，通过接口调用远程方法

● 代理模式(Proxy)

Java RMI的开发应用案例-程序框架

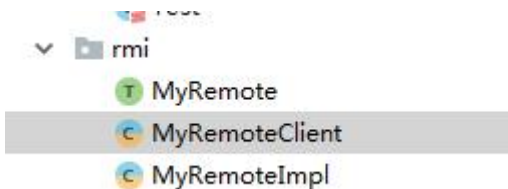
如图(调整+说明)

RMI开发示意图



● 代理模式(Proxy)

Java RMI的开发 应用案例-代码实现



图片4.z

```
trait MyRemote extends Remote {  
  //一个抽象方法  
  @throws(classOf[RemoteException])  
  def sayHello(): String //throws RemoteException  
}
```

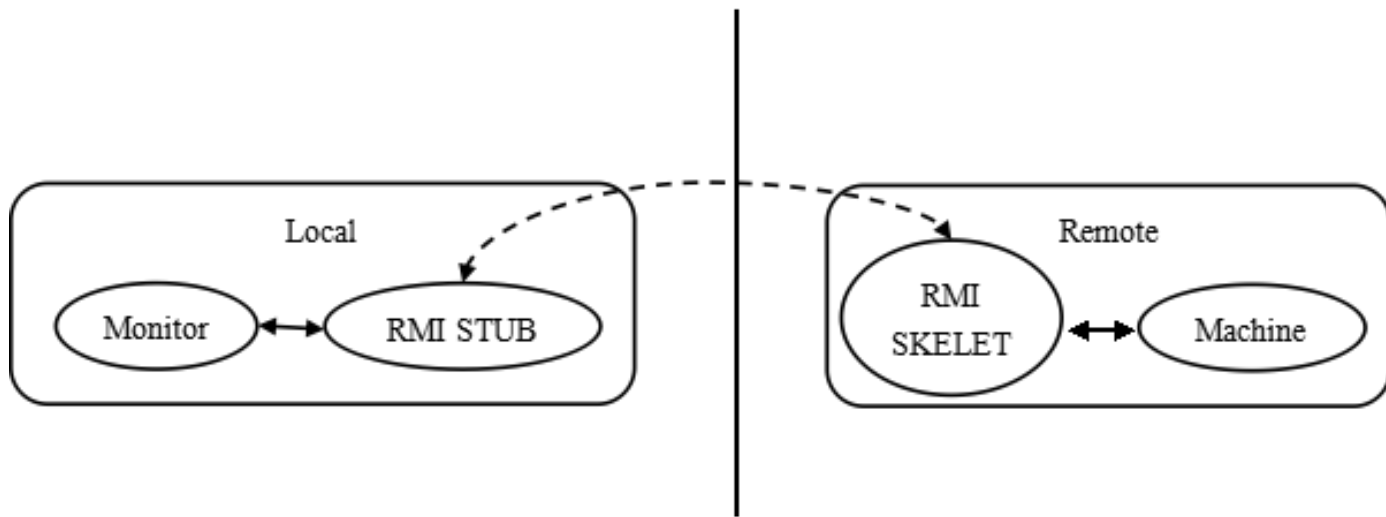
```
class MyRemoteClient {  
  def go() = {  
    val service = Naming.lookup("rmi://127.0.0.1:9999/RemoteHello").asInstanceOf[MyRemote]  
    val str = service.sayHello()  
    println("str = " + str)  
  }  
}  
object MyRemoteClient {  
  def main(args: Array[String]): Unit = {  
    new MyRemoteClient().go()  
  }  
}
```

```
class MyRemoteImpl extends UnicastRemoteObject with MyRemote {  
  @throws(classOf[RemoteException])  
  override def sayHello(): String = {  
    "Hello World!~"  
  }  
}  
object MyRemoteImpl {  
  def main(args: Array[String]): Unit = {  
    val service: MyRemote = new MyRemoteImpl()  
    //准备把服务绑定到9999端口  
    //LocateRegistry.createRegistry(9999)  
    //Naming.rebind("RemoteHello", service)  
    Naming.rebind("rmi://127.0.0.1:9999/RemoteHello", service)  
    println("远程服务开启, 在127.0.0.1的 9999端口监听, 服务名 RemoteHello")  
  }  
}
```

- 代理模式(Proxy)

使用远程代理模式完成远程糖果机监控

➤ 示例项目类结构图



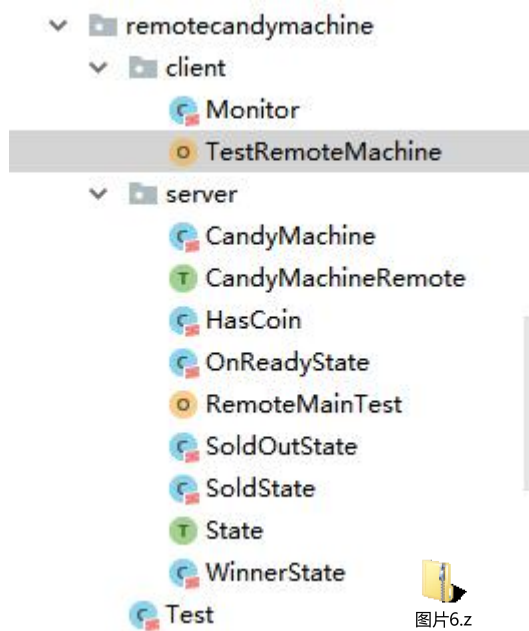
图片5.z



● 代理模式(Proxy)

使用远程代理模式完成远程糖果机监控

➤ 代码实现



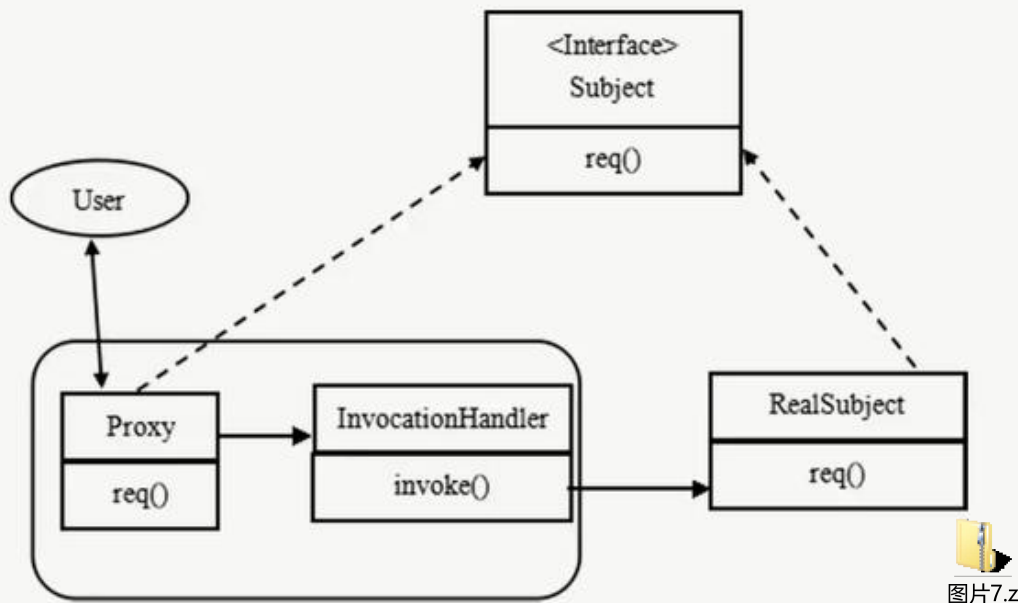
说明：演示时，我的客户端直接使用了server的类和接口,在实际开发中，需要给客户端/代理端拷贝一份



● 代理模式(Proxy)

动态代理

动态代理：运行时动态的创建代理类(对象)，并将方法调用转发到指定类(对象)



图片7.z

- 1) Proxy 和 InvocationHandler 组合充当代理的角色.
- 2) RealSubject 是一个实际对象，它实现接口 Subject
- 3) 在使用时，我们不希望直接访问 RealSubject 的对象，比如：我们对这个对象的访问是有控制的
- 4) 我们使用动态代理，在程序中通过动态代理创建 RealSubject，并完成调用.
- 5) 动态代理可以根据需要，创建多种组合
- 6) Proxy 也会实现 Subject 接口的方法，因此，使用 Proxy+Invocation 可以完成对 RealSubject 的动态调用。
- 7) 但是通过 Proxy 调用 RealSubject 方法是否成功，是由 InvocationHandler 来控制的。(这里其实就是 **保护代理**)
- 8) 理解：创建一个代理对象替代被调用的真实对象，使用反射实现控制



- 代理模式(Proxy)

保护代理

通过前面的分析：大家可以看出动态代理其实就体现出保护代理，即代理时，对**被代理的对象（类）**的哪些方法可以调用，哪些方法不能调用在InvocationHandler可以控制。因此动态代理就体现(实现)了保护代理的效果。



● 代理模式(Proxy)

动态代理的应用案例

➤ 应用案例说明

有一个婚恋网项目，女友/男友有个人信息、兴趣爱好和总体评分, 要求：

- 1) 不能自己给自己评分
- 2) 其它用户可以评分，但是不能设置信息，兴趣爱好。
- 3) 请使用动态代理实现保护代理的效果。
- 4) 分析这里我们需要写两个代理。一个是自己使用，一个是提供给其它用户使用。
- 5) 示意图画出：



图片.z



- 代理模式(Proxy)

动态代理的应用案例

- 代码实现



dyn.zip



● 代理模式(Proxy)

几种常见的代理模式介绍— 几种变体

1) 防火墙代理

内网通过代理穿透防火墙，实现对公网的访问。

2) 缓存代理

比如：当请求图片文件等资源时，先到缓存代理取，如果取到资源则ok,如果取不到资源，再到公网或者数据库取，然后缓存。

3) 静态代理

静态代理通常用于对原有业务逻辑的扩充。

比如持有第三方包的某个类，并调用了其中的某些方法。比如记录日志、打印工作等。可以创建一个代理类实现和第三方方法相同的方法，通过让代理类持有真实对象，调用代理类方法，来达到增加业务逻辑的目的。



- 代理模式(Proxy)

几种常见的代理模式介绍— 几种变体

4) Cglib代理

使用cglib[Code Generation Library]实现动态代理，并不要求委托类必须实现接口，底层采用asm字节码生成框架生成代理类的字节码。

5) 同步代理

主要使用在多线程编程中，完成多线程间同步工作



谢谢！ 欢迎收看！