**Design pattern in action**

切勿要求胜利，只应要求一往无前的勇气。

因为从坚忍不拔的奋斗中，你将为自己迎来荣誉。

# Chapter 1 Introduction(Strategy)

所谓的strategy，就是把某几个算法封装起来，如果客户端调用这些算法的时候，可以随意替换这些算法。实现strategy这个设计模式的方式，就是java中的接口。

具体用法参考代码。

# **Chapter2 Observer Pattern**

观察者模式的定义如下：

|  |
| --- |
| **The Observer Pattern**  Defines a one-to-many dependency between objects so that when one object changes state,all of its depentends are notified and updated automatically. |

其实就是发行者-订阅者模式。发行方一旦有新的内容，就能马上更新到订阅者那里。这个更新动作被抽象出来，放到一个接口中。这样，发行者、订阅者就能解耦了。类图如下：

OLE-object

图2-1

# **Chapter3 Decorator Pattern**

装饰模式定义如下

|  |
| --- |
| **The Decorator pattern**  Attaches additional responsibilities to an object dynamically.Decorators provides a flexible altinative to subclassing for extending functionality. |

    Class diagram如下图所示。我们可以看到，通过Decorator，我们不断地在原有的类上添加新的方法或者特性。

OLE-object

图3-1

# **Chapter4 Factory Pattern**

    工厂模式定义如下：

|  |
| --- |
| **The Factory Method Pattern**  Defines an interface for creating an object ,but lets subclasses decide which class to instantiate.Factory Methods lets a class defer instantiation to subclasses. |

|  |
| --- |
| **The abstract Factory Pattern**  Provides an interface for creating families of related or dependent objects without specifying their contrete classes. |

# **Chapter5 Singleton Pattern**

单例模式定义如下：

|  |
| --- |
| **The Singleton Pattern**  Ensures a class has only one instance,and provides a global point of access to it. |

类图如下：

OLE-object

类图已经很清楚地说明了Singleton这个设计模式的意思了。用实际代码展示就是：

|  |
| --- |
| **private static** Singleton *mySingleton*;  **public static** Singleton getInstance(){  **if**(*mySingleton* == **null**){  *mySingleton* = **new** Singleton();      }  **return** *mySingleton*;  } |

代码1 Singleton设计模式 v1.0

v1.0存在的问题是，在多线程模式下会有问题。多个线程同时调用getInstance()方法的时候，会导致mySingleton被多次实例化的，导致Singleton模式失败。

如何验证这个现象呢，我们通过ThreadTest.java就能验证。

解决方案1：

|  |
| --- |
| **private static** Singleton *mySingleton*;  **public static synchronized** Singleton getInstance(){  **if**(*mySingleton* == **null**){  *mySingleton* = **new** Singleton();      }  **return** *mySingleton*;  } |

代码2 Singleton设计模式 v1.1

v1.1通过给getInstance()加上synchronized关键字，防止并发线程同时调用getInstance()方法，规避了以上的问题。

v1.1的问题是效率比较低。为啥效率低呢？因为我们看到getInstance()方法被加上synchronized关键字后，每次获取实例都加上了synchronized关键字，这样其实是不合理的。因为需要new Singleton对象的场景非常少。

后续针对效率低的问题，提出了2个解决方案：

|  |
| --- |
| **private static** ChocolateBoiler *chocolateBoiler* =  **new** ChocolateBoiler();  **public static synchronized** ChocolateBoiler getInstance(){  **return** *chocolateBoiler*;  } |

代码3 Singleton设计模式 v1.2 解决方案1

|  |
| --- |
| **private volatile static** ChocolateBoiler *chocolateBoiler*;  *// static getInstance()*  **public static** ChocolateBoiler getInstance(){  **if**(*chocolateBoiler* == **null**){  **synchronized** (ChocolateBoiler.**class**){  **if**(*chocolateBoiler* == **null**){  *chocolateBoiler* = **new** ChocolateBoiler();              }          }      }  **return** *chocolateBoiler*;  } |

代码4 Singleton设计模式 v1.3 解决方案2

# **Chapter6 Command Pattern**

|  |
| --- |
| **The Command Pattern**  Encapsulates a request as an object, thereby letting you parameterize other objects with different requests,queue or log requests, and support undoable operations. |

# **Chapter7 Adpter Pattern**

|  |
| --- |
| **The Adapter Pattern**  Converts the interface of a class into another interface the clients expect.Adapter lets classes work together that couldn’t otherwise because of incompatible interface. |

|  |
| --- |
| **The Façade Pattern**  Provide a unified interface to a set of interfaces in a subsystem.Facede defines a higher-level interface that makes the subsystem easier to use. |

# **Chapter8 Template Method Pattern**

|  |
| --- |
| **The Template Method Pattern**  Defines the skeleton of an algorithm in a method ,deffering some steps to subclasses.Template Method lets subclasses fedefine certain steps of an algorithm without changing the algorithm’s structure. |

有一个父类，这个父类中部分方法是定义在父类中，另外一部分是抽象类。需要在子类中定义。

一个典型的应用场景是Comparable，那些需要通过Arrays.sort()进行排序的类必集成Comparable类。排序的时候会调用Comparable类，其中CompareTo()方法需要在Comparable子类中定义。

例子1

Tea and Coffee

|  |
| --- |
| 父类：  Coffeine.java  子类：  Tea.java  Coffee.java  抽象方法放在父类中，特色的方法定义在子类中 |

例子2 CompareTo方法

|  |
| --- |
| Duck.java  DuckSortDrive.java  Duck.java实现了Comarable接口，实现了compareTo()方法  Comparable接口中只是声明了compareTo()的抽象方法。  具体的比较逻辑定义在Duck.java类中。 |

例子3

|  |
| --- |
| MyFrame.java  MyFrame.java继承了JFrame，重载了paint方法。 |

# **Chapter9 The Iterator and Composite Patterns**

**Iterator**

|  |
| --- |
| **The Iterator Pattern**  Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation. |

       我们来总结一下Iterator这个设计模式：有一个集合，我们通过Iterator的形式，将遍历这个集合的动作抽象出来。

       这样做有什么好处呢？我们遍历这个集合对象的时候，不用考虑这个对象的内部实现，即便集合对象的内部实现有了变化，遍历的动作也无须改变。

       示例代码如下：

|  |
| --- |
| MenuItem.java  DinerMenu.java  PancakeHouseMenu.java  DinerMenuIterator.java  PancakeHouseMenuIterator.java  Waitress.java  Iterator.java   // interface  Menu.java       // interface  MenuTestDrive.java // 测试程序入口 |

**Composite**

|  |
| --- |
| **The Composite Pattern**  Allows you to compose objects into tree structures to represent part-whole hierarchies.Composite lets clients treat individual objects and compositions of objects uniformly. |

       怎么解释composite pattern呢？例子还是之前的menu，之前我们在menu中只能有menuitem，如下图所示：

Menu

MenuItem

MenuItem

MenuItem

    现在我们要把Menu也作为tree hiarachy中的一个node，如下图：

Menu

MenuItem

MenuItem

Menu

MenuItem

MenuItem

MenuItem

   说白了，就是能够将Menu和MenuItem融合起来，无差别地进行访问(当然细微的差别还是有的)。

   类比一下，就是Menu相当于Windows中的目录(dir)，MenuItem相当于文件(file)，之前我们只有一层目录，现在我们可以通过Composite，实现目录嵌套。

    怎么实现呢？实现代码汇总：

|  |
| --- |
| MenuComponent.java  // 抽象类  MenuNode.java // menu节点 是MenuComponent的子类  MenuItem.java // menu中每个具体的菜单项 是MenuComponent的子类  WaitressNew.java  // 打印MenuComponent组成的tree hierachy  MenuTestDriveNew.java // 新的驱动类 |

    看看代码当然是看不出什么意思来的，还是总结一下吧。

    无差别地访问Menu和MenuItem？

    听起来是不是有点熟悉？对了，之前我们通过Iterator实现对象遍历的时候也是这个意思：

    无差别地访问集合对象，无论集合对象是什么。

    Composite和Iterator其实有异曲同工之妙，都是不管你集合或者tree hierachy中的对象是什么，都能无差别地访问。

    之前Iterator的实现方式是，集合对象实现Iterator接口(hasNex() and next())

    Composite的实现方式是，tree hierachy中的node只要继承了MenuComponent抽象类(add()/remove()/getChild()/print()等方法)，就能无缝集成到我们tree hierachy体系中，进行访问。

我们的例子：

allMenus

cafeMenu

pancakeMenu

dinerMenu

MenuItem1

MenuItem2

dessertMenu

MenuItem3

MenuItem1

MenuItem1

MenuNode

MenuItem

**Composite和Iterator结合**

    前面两结我们通过Iterator访问MenuItem，通过Comonent实现了MenuNode和MenuItem融合在Tree hierachy架构下统一访问。现在我们将Comosite和Iterator结合：通过Iterator的方式嵌套访问MenuNode下的MenuItem信息。

**总结**

    我们总结一下，这一章的代码。最关键的是2个代码文件：

|  |
| --- |
| MyIterator.java  MenuComponent.java  CompositeIterator.java |

    其中MyIterator.java是一个interface，类似java.util.Iterator。我们通过实现这个自定义的Iterator，实现对Menu下MenuItem的遍历。

  其中MenuComponent.java是抽象类，通过集成这个抽象类，MenuItem和MenuNode可以无差别地进行访问。

  那么如何实现以Iterator的方式访问MenuComponent通过对象呢？那就要用到CompositeIterator.java了。

**注意点**

    让人大跌眼镜的是，这一章居然还有一个代码bug:

|  |
| --- |
| CompositeIterator.java  next()方法：  *stack.push(component.createIterator());*  ------>  **stack**.push(currentNode.**menucomponents**.iterator()); |

    仔细想想，也不难理解，stack.push()的意思是，将需要遍历的MenuItem或者MenuNode下的MenuItem节点以Iterator的形式，放到stack中。

    而*stack.push(component.createIterator())*中，*component.createIterator()*虽然是返回*Iterator*，但是我们看对于*MenuNode*，*createIterator()*方法是这样的：

|  |
| --- |
| **public** Iterator createIterator(){  **return new** CompositeIterator(**menucomponents**.iterator());  } |

看出问题来了吗：这里是创建了一个新的*CompositeIterator*实例，那么在 *CompositeIterator*的*constructor*中，

|  |
| --- |
| **public** CompositeIterator(Iterator iterator){  **stack**.push(iterator);  } |

又执行了一次*stack.push()*，这样不是*stack*就重复*push*了吗？其实正确的做法是将当前*MenuNode*下的*MenuItem*节点以*Iterator*的形式，直接*push()*到*stack*中：

|  |
| --- |
| **stack**.push(currentNode.**menucomponents**.iterator()); |

# **Chapter10 The State of Things**

|  |
| --- |
| **The State Pattern**  Allows an object to alter its behavior when its internal state changes. The object will appear to change its class. |

     这章的例子是gumball machine，gumball machine的各个状态如下：

Has Quarter

Out of

ball

投币

按上按钮，放球

No Quarter

退币

Gumball = 0

球全部卖光了

Gumball sold

Gumball >0

放球后，回到硬币清空的状态

    作为对比，这章主要有两个代码：

1.GumbalMachine.java

2.GumbalMachineState.java

       其中GumbalMachine.java是老的思路：面向动作：

   在每个动作中，都要先判断当前*gumball machine*的状态，然后选择相应的动作，类似*: if(state == SOLD\_OUT){ ... } else if(){ ... }*

    GumbalMachineState.java给我们带来了新的思路：

和之前的*GumbalMachine.java*不同的是，这次我们每次要做动作的时候，不需要判断当前状态了。为什么呢？因为我们有一个变量：*state*。这个变量保存的就是当前的状态。这个类实现的思路是：在当前状态下，如果要做某个动作，会发生什么。至于不同状态下*,*做某个动作，需要做什么是，逻辑都放在各个状态的实现类中。

    新的思路和状态机有点类似：在某给状态下，如果施加某个动作，会做什么事情，事情做完后，会转到下一个动作。

# **Chapter11 The Proxy Pattern**

|  |
| --- |
| **The Proxy Pattern**  Provides a surrogate or placeholder for another object to control access to it |

## 1.Gumball例子

Gumball是一个普通的java代码示例，为后续说明如何使用动态代理进行铺垫。大致代码如下：

|  |
| --- |
| 服务端：  GumballMachine.java  客户端：  GumballMonitor.java  客户端GumballMonitor.report()，获取服务端GumballMachine对象的状态。 |

具体代码参考如下package：

|  |
| --- |
| com.nbcb.design.proxy.gumballmachine |

## 2.RMI例子

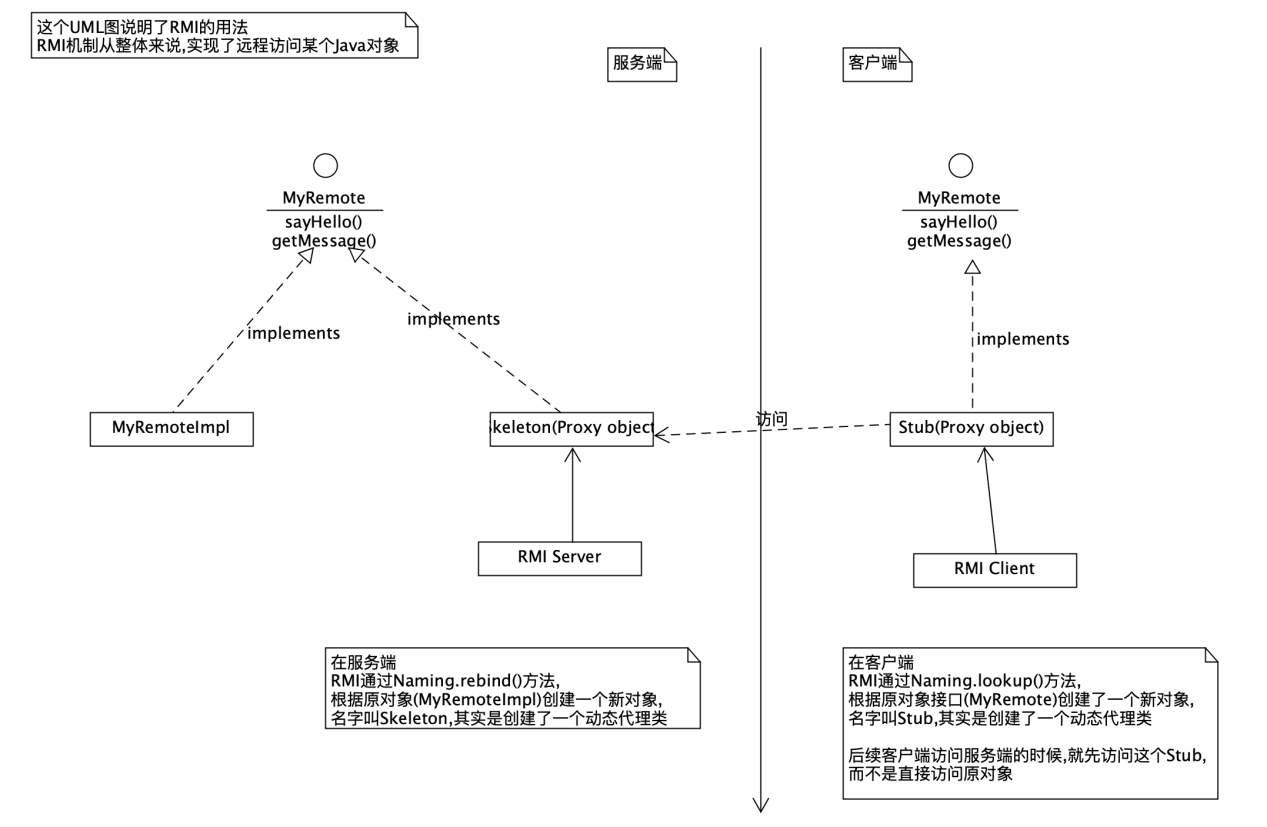
我们通过这个小节的例子，说明一下java RMI的用法。

### RMI功能说明

所谓的RMI，其实就是java提供的一种远程方法调用的方式。通过RMI，能够跨JVM，甚至跨服务器，调用远程>的java方法。

### RMI整体架构

我们先梳理一下RMI整体架构：



接下来，我们整理一下java RMI通讯的详细步骤：

### 服务端：

1. 定义一个RMI接口：

|  |
| --- |
| interface MyRemote extends Remote { ... } |

其中定义的接口方法，供客户端调用

1. 定义RMI实现类：

|  |
| --- |
| public class MyRemoteImpl  extends UnicastRemoteObject         implements MyRemote { ... } |

1. 定义RMI Server

比如:

|  |
| --- |
| LocateRegistry.createRegistry(1900); Naming.rebind("rmi://localhost:1900/remoteSayHello",myRemote); |

### 客户端：

|  |
| --- |
| MyRemote myRemote  = (MyRemote)Naming.lookup("rmi://localhost:1900/remoteSayHello"); myRemote.sayHello() |

### 特别备注：

1. RMI接口中定义的接口方法，必须throws RemoteException 否则启动RmiServer的时候会抛错：

2.如果RMI Server和client之间需要传递对象，这个对象

必须实现Serializable接口 用于序列化、反序列化 参考：Message.java

### 具体代码

具体代码参考：

|  |
| --- |
| com.nbcb.design.proxy.rmi |

### RMI总结

我们总结一下RMI的用法，其实要实现RMI通讯，核心是定义一个客户端、服务端都能够接受的Remote接口。然后RMI会通过动态代理的方式，分别在服务端创建Skeleton、客户端创建Stub(Skeleton/Stub都实现了Remote接口，其实Skeleton/Stub都是动态代理类)。

后续client/Server通过Skeleton/Stub进行通讯。

## 3.Gumball结合RMI

Gumball和RMI结合，就能够实现远程调用

为了实现跨JVM访问GumballMachine信息的目标，我们要做哪些事情呢？  
我们记录一下步骤：

### **1.定义RMI Remote接口类**

GumballMachineRemote  
  
2.改造GumballMachine

我们原来的GumballMachine类要改造一下，实现  
新定义的Remote接口：GumballMachineRemote  
  
GumballMachine  
 extends UnicastRemoteObject  
 implements GumballMachineRemote {  
 ...  
 }  
  
3.创建RMI Server  
参考：GumballMachineServer.java  
这个Server的功能就是把我们新创建的RMI Remote接口类发布出去

4.创建RMI Client  
参考：  
GumballMachineClient.java  
GumballMonitor.java  
  
整理总结一下  
我们看到，要实现某个对象的远程访问，  
必须通过一个定义好的RMI Remote接口  
无论是RMI服务端还是RMI客户端，都通过这个RMI Remote接口进行交互

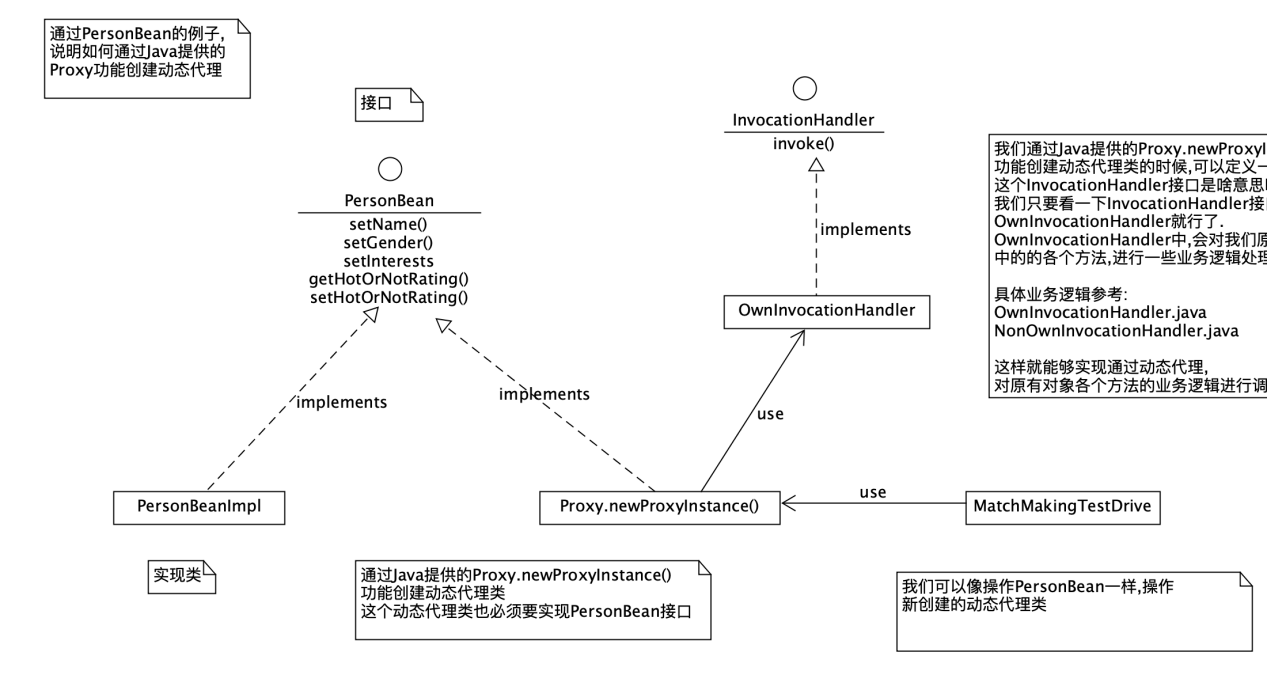
具体代码参考：

|  |
| --- |
| com.nbcb.design.proxy.remotegumball |

## 4.CDViewer

## 5.PersonBean

通过PersonBean的例子,说明如何通过Java提供的Proxy功能创建动态代理

大致架构如下：

## 总结

我们通过各种动态代理的案例，来说明动态代理的功能。这章更多的是介绍动态代理是什么，并没有解释动态代理的原理。

我们从Java自带的Proxy.newProxyInstance()的用法，也能对动态代理的实现方式窥探一二：

|  |
| --- |
| return (PersonBean) Proxy.newProxyInstance(  personBean.getClass().getClassLoader(),  personBean.getClass().getInterfaces(),  new OwnInvocationHandler(personBean)); |

其实就是基于反射的原理，分析personBean对象的各个方法，然后根据InvocationHandler中定义的业务逻辑，对personBean对象的各个方法做一定的处理。

动态代理的设计模式涉及反射，是后续AOP功能的基石，还是比较有意思的。

# **Chapter12 Comound Patterns:Patterns of Patterns**

# **Chapter13 Patterns in the Real World**

# Chapter14 Appendix: Leftover Patterns

其它设计模式还有很多。