研磨设计模式--chjavach的博客文章

作者: chjavach http://chjavach.javaeye.com

研磨设计模式系列,包括:

单例模式、工厂方法模式、策略模式、命令模式和桥接模式

目 录

1. 设计模式

1.1 研磨设计模式之工厂方法模式-1

发表时间: 2010-06-17

做Java一晃就十年了,最近手痒痒,也决定跟随一下潮流,整个博客,写点东西,就算对自己的知识进行一个梳理和总结，也跟朋友们交流交流，希望能坚持下去。

先写写设计模式方面的内容吧，就是GoF的23个模式，先从大家最熟悉的工厂方法模式开始，这个最简单，明白的人多，看看是否能写出点跟别人不一样的东西，欢迎大家来热烈讨论，提出建议或意见，并进行批评指正，一概虚心接受，在此先谢过了！

另外，大家也可以说说最想看到哪个模式，那我就先写它，呵呵，大家感兴趣，我才会有动力写下去！好了，言归正传，Now Go!

工厂方法模式（Factory Method）

1 场景问题

1.1 导出数据的应用框架

考虑这样一个实际应用：实现一个导出数据的应用框架，来让客户选择数据的导出方式，并真正执行数据导出。

在一些实际的企业应用中，一个公司的系统往往分散在很多个不同的地方运行，比如各个分公司或者是门市点，公司没有建立全公司专网的实力，但是又不愿意让业务数据实时的在广域网上传递，一个是考虑数据安全的问题，一个是运行速度的问题。

这种系统通常会有一个折中的方案，那就是各个分公司内运行系统的时候是独立的，是在自己分公司的局域网内运行。然后在每天业务结束的时候，各个分公司会导出自己的业务数据，然后把业务数据打包通过网络传送给总公司，或是专人把数据送到总公司，然后由总公司进行数据导入和核算。

通常这种系统，在导出数据上，会有一些约定的方式，比如导出成：文本格式、数据库备份形式、Excel格式、Xml格式等等。

现在就来考虑实现这样一个应用框架。在继续之前，先来了解一些关于框架的知识。

1.2 框架的基础知识

(1)：框架是什么

简单点说：**框架就是能完成一定功能的半成品软件。**

就其本质而言，框架是一个软件，而且是一个半成品的软件。所谓半成品，就是还不能完全实现用户需要的功能，框架只是实现用户需要的功能的一部分，还需要进一步加工，才能成为一个满足用户需要的、完整的软件。因此框架级的软件，它的主要客户是开发人员，而不是最终用户。

有些朋友会想，既然框架只是个半成品，那何必要去学习和使用框架呢？学习成本也不算小，那就是因为

框架能完成一定的功能，也就是这“框架已经完成的一定的功能”在吸引着开发人员，让大家投入去学习和使用框架。

(2)：框架能干什么

能完成一定功能，加快应用开发进度

由于框架完成了一定的功能，而且通常是一些基础的、有难度的、通用的功能，这就避免我们在应用开发的时候完全从头开始，而是在框架已有的功能之上继续开发，也就是说会复用框架的功能，从而加快应用的开发进度。

给我们一个精良的程序架构

框架定义了应用的整体结构，包括类和对象的分割，各部分的主要责任，类和对象怎么协作，以及控制流程等等。

现在Java界大多数流行的框架，大都出自大师手笔，设计都很精良。基于这样的框架来开发，一般会遵循框架已经规划好的结构来进行开发，从而让我们开发的应用程序的结构也相对变得精良了。

(3)：对框架的理解

基于框架来开发，事情还是那些事情，只是看谁做的问题

对于应用程序和框架的关系，可以用一个图来简单描述一下，如图1所示：

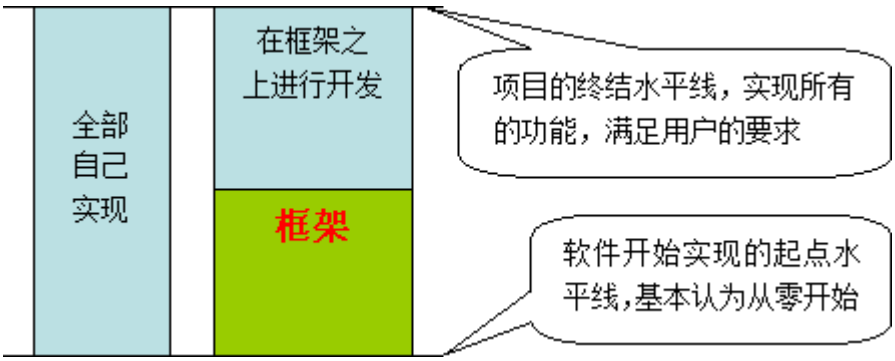


图1 应用程序和框架的简单关系示意图

如果没有框架，那么客户要求的所有功能都由开发人员自己来开发，没问题，同样可以实现用户要求的功能，只是开发人员的工作多点。

如果有了框架，框架本身完成了一定的功能，那么框架已有的功能，开发人员就可以不做了，开发人员只需要完成框架没有的功能，最后同样是完成客户要求的所有功能，但是开发人员的工作就减少了。

也就是说，基于框架来开发，软件要完成的功能并没有变化，还是客户要求的所有功能，也就是“事情还是那些事情”的意思。但是有了框架过后，框架完成了一部分功能，然后开发人员再完成一部分功能，最后由框架和开发人员合起来完成了整个软件的功能，也就是看这些功能“由谁做”的问题。

基于框架开发，可以不去做框架所做的事情，但是应该明白框架在干什么，以及框架是如何实现相应功能的

事实上，在实际开发中，应用程序和框架的关系，通常都不会如上面讲述的那样，分得那么清楚，更为普遍的是相互交互的，也就是应用程序做一部分工作，然后框架做一部分工作，然后应用程序再做一部分工作，然后框架再做一部分工作，如此交错，最后由应用程序和框架组合起来完成用户的功能要求。

也用个图来说明，如图2所示：

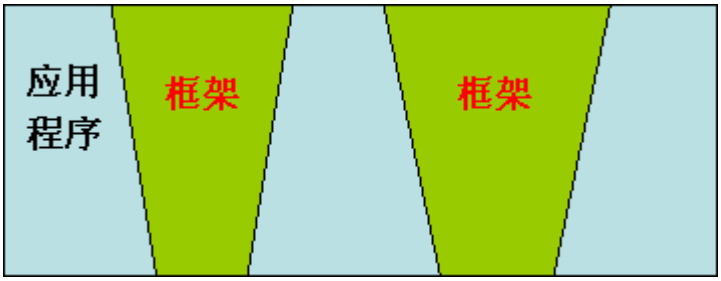


图2 应用程序和框架的关系示意图

如果把这个由应用程序和框架组合在一起构成的矩形，当作最后完成的软件。试想一下，如果你不懂框架在干什么的话，相当于框架对你来讲是个黑盒，也就是相当于在上面图2中，去掉框架的两块，会发现什么？没错，剩下的应用程序是支离破碎的，是相互分隔开来的。

这会导致一个非常致命的问题，整个应用是如何运转起来的，你是不清楚的，也就是说对你而言，项目已经失控了，从项目管理的角度来讲，这是很危险的。

因此，在基于框架开发的时候，虽然我们可以不去做框架所做的事情，但是应该搞明白框架在干什么，如果条件许可的话，还应该搞清楚框架是如何实现相应功能的，至少应该把大致的实现思路和实现步骤搞清楚，这样我们才能整体的掌控整个项目，才能尽量减少出现项目失控的情况。

(4)：框架和设计模式的关系

设计模式比框架更抽象

框架已经是实现出来的软件了，虽然只是个半成品的软件，但毕竟是已经实现出来的了。而设计模式的重心还在于解决问题的方案上，也就是还停留在思想的层面。因此设计模式比框架更为抽象。

设计模式是比框架更小的体系结构元素

如上所述，框架是已经实现出来的软件，并实现了一系列的功能，因此一个框架，通常会包含多个设计模式的应用。

框架比设计模式更加特例化

框架是完成一定功能的半成品软件，也就是说，框架的目的很明确，就是要解决某一个领域的某些问题，那是很具体的功能，不同的领域实现出来的框架是不一样的。

而设计模式还停留在思想的层面，在不同的领域都可以应用，只要相应的问题适合用某个设计模式来解决。因此框架总是针对特定领域的，而设计模式更加注重从思想上，从方法上来解决问题，更加通用化。

1.3 有何问题

分析上面要实现的应用框架，不管用户选择什么样的导出格式，最后导出的都是一个文件，而且系统并不知道究竟要导出成为什么样的文件，因此应该有一个统一的接口，来描述系统最后生成的对象，并操作输出的文件。

先把导出的文件对象的接口定义出来，示例代码如下：

```
/**
 * 导出的文件对象的接口
 */
public interface ExportFileApi {
    /**
     * 导出内容成为文件
     * @param data 示意：需要保存的数据
     * @return 是否导出成功
     */
    public boolean export(String data);
}
```

对于实现导出数据的业务功能对象，它应该根据需要来创建相应的ExportFileApi的实现对象，因为特定的ExportFileApi的实现是与具体的业务相关的。但是对于实现导出数据的业务功能对象而言，它并不知道应该创建哪一个ExportFileApi的实现对象，也不知道如何创建。

也就是说：对于实现导出数据的业务功能对象，它需要创建ExportFileApi的具体实例对象，但是它只知道ExportFileApi接口，而不知道其具体的实现。那该怎么办呢？

未完待续.....

1.2 研磨设计模式之工厂方法模式-2

发表时间: 2010-06-17

2 解决方案

2.1 工厂方法模式来解决

用来解决上述问题的一个合理的解决方案就是工厂方法模式。那么什么是工厂方法模式呢？

（1）工厂方法模式定义

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，Factory Method使一个类的实例化延迟到其子类。

（2）应用工厂方法模式来解决的思路

仔细分析上面的问题，事实上在实现导出数据的业务功能对象里面，根本就不知道究竟要使用哪一种导出文件的格式，因此这个对象本就不应该和具体的导出文件的对象耦合在一起，它只需要面向导出的文件对象的接口就好了。

但是这样一来，又有新的问题产生了：接口是不能直接使用的，需要使用具体的接口实现对象的实例。

这不是自相矛盾吗？要求面向接口，不让和具体的实现耦合，但是又需要创建接口的具体实现对象的实例。怎么解决这个矛盾呢？

工厂方法模式的解决思路很有意思，那就是不解决，采取无为而治的方式：不是需要接口对象吗，那就定义一个方法来创建；可是事实上它自己是不知道如何创建这个接口对象的，没有关系，那就定义成抽象方法就好了，自己实现不了，那就让子类来实现，这样这个对象本身就可以只是面向接口编程，而无需关心到底如何创建接口对象了。

2.2 模式结构和说明

工厂方法模式的结构如图3所示：

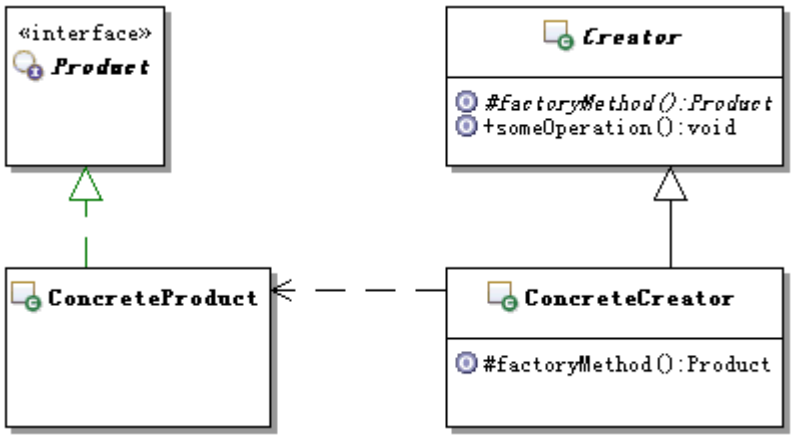


图3 工厂方法模式结构示意图

Product：

定义工厂方法所创建的对象的接口，也就是实际需要使用的对象的接口。

ConcreteProduct：

具体的Product接口的实现对象。

Creator：

创建器，声明工厂方法，工厂方法通常会返回一个Product类型的实例对象，而且多是抽象方法。也可以在Creator里面提供工厂方法的默认实现，让工厂方法返回一个缺省的Product类型的实例对象。

ConcreteCreator：

具体的创建器对象，覆盖实现Creator定义的工厂方法，返回具体的Product实例。

2.3 工厂方法模式示例代码

（1）先看看Product的定义，示例代码如下：

```
/**
 * 工厂方法所创建的对象的接口
 */
public interface Product {
   //可以定义Product的属性和方法
}
```

（2）再看看具体的Product的实现对象，示例代码如下：

```
/**
 * 具体的Product对象
 */
public class ConcreteProduct implements Product {
    //实现Product要求的方法
}
```

（3）接下来看看创建器的定义，示例代码如下：

```
/**
 * 创建器，声明工厂方法
 */
public abstract class Creator {
    /**
     * 创建Product的工厂方法
     * @return Product对象
     */
    protected abstract Product factoryMethod();
    /**
     * 示意方法，实现某些功能的方法
     */
    public void someOperation() {
        //通常在这些方法实现中，需要调用工厂方法来获取Product对象
        Product product = factoryMethod();
    }
}
```

（4）再看看具体的创建器实现对象，示例代码如下：

```
/**
 * 具体的创建器实现对象
 */
public class ConcreteCreator extends Creator {
    protected Product factoryMethod() {
        //重定义工厂方法，返回一个具体的Product对象
        return new ConcreteProduct();
    }
}
```

2.4 使用工厂方法模式来实现示例

要使用工厂方法模式来实现示例，先来按照工厂方法模式的结构，对应出哪些是被创建的Product，哪些是Creator。分析要求实现的功能，导出的文件对象接口ExportFileApi就相当于是Product，而用来实现导出数据的业务功能对象就相当于Creator。把Product和Creator分开过后，就可以分别来实现它们了。

使用工厂模式来实现示例的程序结构如图4所示：

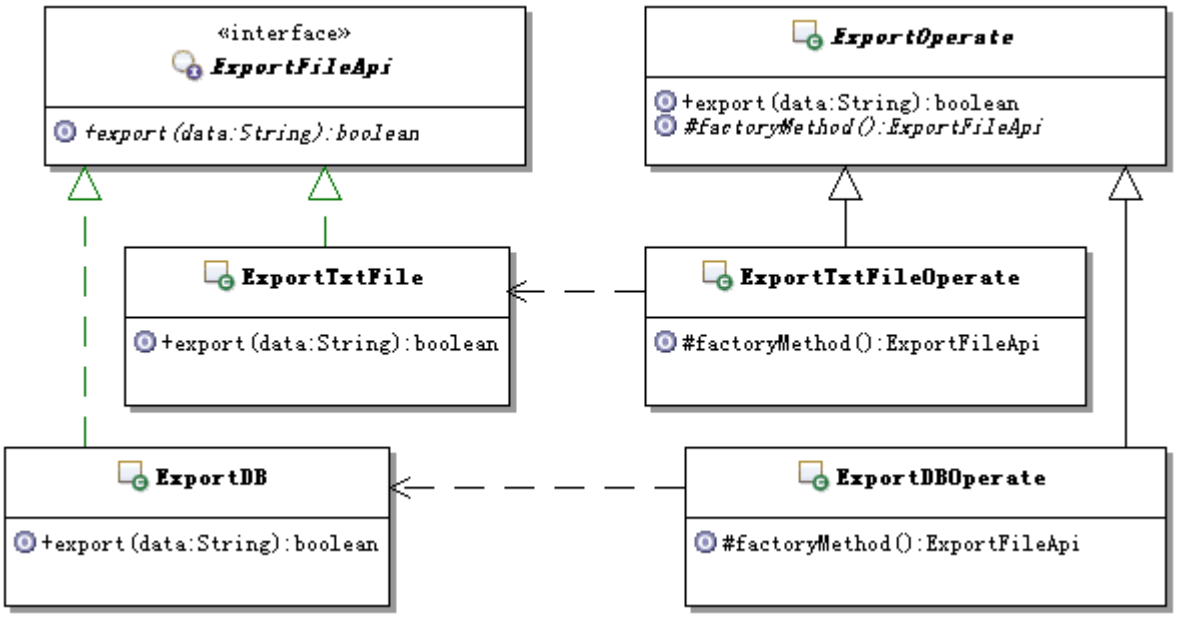


图4 使用工厂模式来实现示例的程序结构示意图

下面一起来看看代码实现。

（1）导出的文件对象接口ExportFileApi的实现没有变化，这里就不去赘述了

（2）接下来看看接口ExportFileApi的实现，为了示例简单，只实现导出文本文件格式和数据库备份文件两种。先看看导出文本文件格式的实现，示例代码如下：

```
/**
 * 导出成文本文件格式的对象
 */
public class ExportTxtFile implements ExportFileApi{
    public boolean export(String data) {
        //简单示意一下，这里需要操作文件
        System.out.println("导出数据"+data+"到文本文件");
        return true;
    }
}
```

再看看导出成数据库备份文件形式的对象的实现，示例代码如下：

```
/**
 * 导出成数据库备份文件形式的对象
 */
public class ExportDB implements ExportFileApi{
    public boolean export(String data) {
      //简单示意一下，这里需要操作数据库和文件
      System.out.println("导出数据"+data+"到数据库备份文件");
      return true;
    }
}
```

（3）Creator这边的实现，首先看看ExportOperate的实现，示例代码如下：

```
/**
 * 实现导出数据的业务功能对象
 */
public abstract class ExportOperate {
    /**
     * 导出文件
     * @param data 需要保存的数据
     * @return 是否成功导出文件
     */
    public boolean export(String data){
      //使用工厂方法
      ExportFileApi api = factoryMethod();
      return api.export(data);
    }
    /**
     * 工厂方法，创建导出的文件对象的接口对象
     * @return 导出的文件对象的接口对象
     */
    protected abstract ExportFileApi factoryMethod();
}
```

（4）加入了两个Creator实现，先看看创建导出成文本文件格式的对象，示例代码如下：

```
/**
 * 具体的创建器实现对象，实现创建导出成文本文件格式的对象
 */
public class ExportTxtFileOperate extends ExportOperate{
    protected ExportFileApi factoryMethod() {
      //创建导出成文本文件格式的对象
```

```
        return new ExportTxtFile();
    }
}
```

再看看创建导出成数据库备份文件形式的对象，示例代码如下：

```
/**
 * 具体的创建器实现对象，实现创建导出成数据库备份文件形式的对象
 */
public class ExportDBOperate extends ExportOperate{
    protected ExportFileApi factoryMethod() {
        //创建导出成数据库备份文件形式的对象
        return new ExportDB();
    }
}
```

（5）客户端直接创建需要使用的Creator对象，然后调用相应的功能方法，示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建需要使用的Creator对象
        ExportOperate operate = new ExportDBOperate();
        //调用输出数据的功能方法
        operate.export("测试数据");
    }
}
```

运行结果如下：

```
导出数据测试数据到数据库备份文件
```

你还可以修改客户端new的对象，切换成其它的实现对象，试试看会发生什么。看来应用工厂方法模式是很简单的，对吧。

未完待续.....

1.3 研磨设计模式之工厂方法模式-3

发表时间: 2010-06-17

3 模式讲解

3.1 认识工厂方法模式

（1）模式的功能

工厂方法的主要功能是让父类在不知道具体实现的情况下，完成自身的功能调用，而具体的实现延迟到子类来实现。

这样在设计的时候，不用去考虑具体的实现，需要某个对象，把它通过工厂方法返回就好了，在使用这些对象实现功能的时候还是通过接口来操作，这非常类似于IoC/DI的思想，这个在后面给大家稍详细点介绍一下。

（2）实现成抽象类

工厂方法的实现中，通常父类会是一个抽象类，里面包含创建所需对象的抽象方法，这些抽象方法就是工厂方法。

这里要注意一个问题，子类在实现这些抽象方法的时候，通常并不是真的由子类来实现具体的功能，而是在子类的方法里面做选择，选择具体的产品实现对象。

父类里面，通常会有使用这些产品对象来实现一定的功能的方法，而且这些方法所实现的功能通常都是公共的功能，不管子类选择了何种具体的产品实现，这些方法的功能总是能正确执行。

（3）实现成具体的类

当然也可以把父类实现成为一个具体的类，这种情况下，通常是在父类中提供获取所需对象的默认实现方法，这样就算没有具体的子类，也能够运行。

通常这种情况还是需要具体的子类来决定具体要如何创建父类所需要的对象。也把这种情况称为工厂方法为子类提供了挂钩，通过工厂方法，可以让子类对象来覆盖父类的实现，从而提供更好的灵活性。

（4）工厂方法的参数和返回

工厂方法的实现中，可能需要参数，以便决定到底选用哪一种具体的实现。也就是说通过在抽象方法里面传递参数，在子类实现的时候根据参数进行选择，看看究竟应该创建哪一个具体的实现对象。

一般工厂方法返回的是被创建对象的接口对象，当然也可以是抽象类或者一个具体的类的实例。

（5）谁来使用工厂方法创建的对象

这里首先要搞明白一件事情，就是谁在使用工厂方法创建的对象？

事实上，在工厂方法模式里面，应该是Creator中的其它方法在使用工厂方法创建的对象，虽然也可以把工厂方法创建的对象直接提供给Creator外部使用，但工厂方法模式的本意，是由Creator对象内部的方法来使用工厂方法创建的对象，也就是说，工厂方法一般不提供给Creator外部使用。

客户端应该是使用Creator对象，或者是使用由Creator创建出来的对象。对于客户端使用Creator对象，这个时候工厂方法创建的对象，是Creator中的某些方法使用。对于使用那些由Creator创建出来的对象，这个时候工厂方法创建的对象，是构成客户端需要的对象的一部分。分别举例来说明。

①客户端使用Creator对象的情况

比如前面的示例，对于“实现导出数据的业务功能对象”的类ExportOperate，它有一个export的方法，在这个方法里面，需要使用具体的“导出的文件对象的接口对象” ExportFileApi，而ExportOperate是不知道具体的ExportFileApi实现的，那么怎么做的呢？就是定义了一个工厂方法，用来返回ExportFileApi的对象，然后export方法会使用这个工厂方法来获取它所需要的对象，然后执行功能。

这个时候的客户端是怎么做的呢？这个时候客户端主要就是使用这个ExportOperate的实例来完成它想要完成的功能，也就是客户端使用Creator对象的情况，简单描述这种情况下的代码结构如下：

```
/**
 * 客户端使用Creator对象的情况下，Creator的基本实现结构
 */
public abstract class Creator {
    /**
     * 工厂方法，一般不对外
     * @return 创建的产品对象
     */
    protected abstract Product factoryMethod();
    /**
     * 提供给外部使用的方法，
     * 客户端一般使用Creator提供的这些方法来完成所需要的功能
     */
    public void someOperation(){
      //在这里使用工厂方法
      Product p = factoryMethod();
    }
}
```

②客户端使用由Creator创建出来的对象

另外一种是由Creator向客户端返回由“工厂方法创建的对象”来构建的对象，这个时候工厂方法创建的对象，是构成客户端需要的对象的一部分。简单描述这种情况下的代码结构如下：

```
/**
 * 客户端使用Creator来创建客户端需要的对象的情况下，Creator的基本实现结构
 */
public abstract class Creator {
    /**
     * 工厂方法，一般不对外，创建一个部件对象
     * @return 创建的产品对象，一般是另一个产品对象的部件
     */
    protected abstract Product1 factoryMethod1();
    /**
     * 工厂方法，一般不对外，创建一个部件对象
     * @return 创建的产品对象，一般是另一个产品对象的部件
     */
    protected abstract Product2 factoryMethod2();
    /**
     * 创建客户端需要的对象，客户端主要使用产品对象来完成所需要的功能
     * @return 客户端需要的对象
     */
    public Product createProduct(){
      //在这里使用工厂方法，得到客户端所需对象的部件对象
      Product1 p1 = factoryMethod1();
      Product2 p2 = factoryMethod2();

      //工厂方法创建的对象是创建客户端对象所需要的
      Product p = new ConcreteProduct();
      p.setProduct1(p1);
      p.setProduct2(p2);

      return p;
    }
}
```

小结一下：在工厂方法模式里面，客户端要么使用Creator对象，要么使用Creator创建的对象，一般客户端不直接使用工厂方法。当然也可以直接把工厂方法暴露给客户端操作，但是一般不这么做。

（6）工厂方法模式的调用顺序示意图

由于客户端使用Creator对象有两种典型的情况，因此调用的顺序示意图也分做两种情况，先看看客户端使用由Creator创建出来的对象情况的调用顺序示意图，如图.5所示：

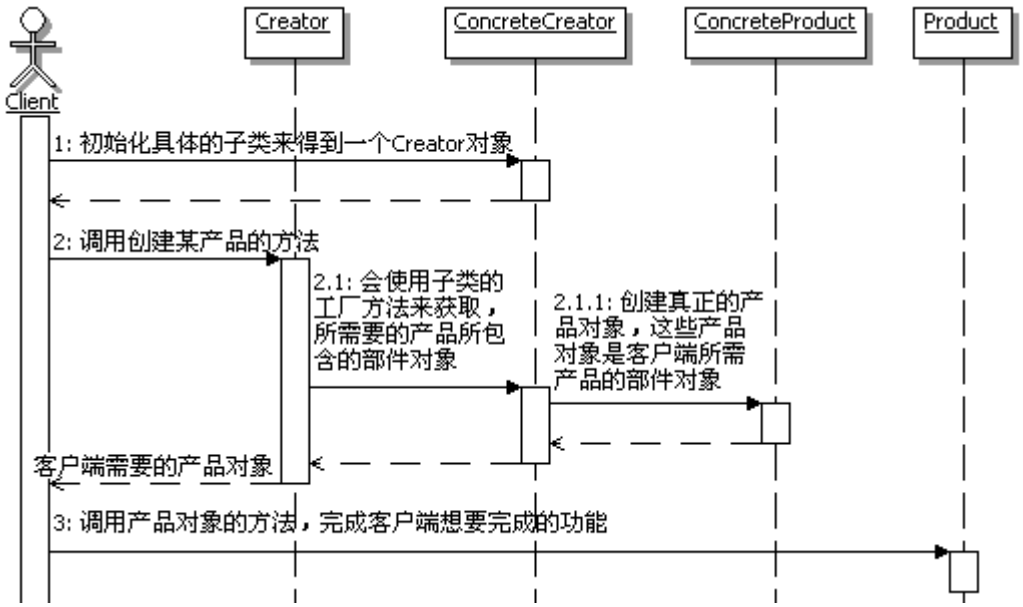


图5 客户端使用由Creator创建出来的对象的调用顺序示意图

接下来看看客户端使用Creator对象时候的调用顺序示意图，如图6所示：

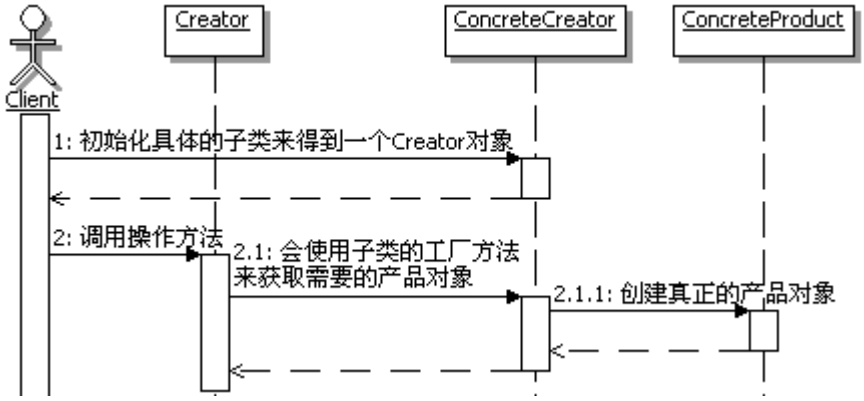


图6 客户端使用Creator对象的调用顺序示意图

未完待续.....

1.4 研磨设计模式之工厂方法模式-4

发表时间: 2010-06-17

3.2 工厂方法模式与IoC/DI

IoC——Inversion of Control 控制反转
DI——Dependency Injection 依赖注入

1：如何理解IoC/DI

要想理解上面两个概念，就必须搞清楚如下的问题：

- 参与者都有谁？
- 依赖：谁依赖于谁？为什么需要依赖？
- 注入：谁注入于谁？到底注入什么？
- 控制反转：谁控制谁？控制什么？为何叫反转（有反转就应该有正转了）？
- 依赖注入和控制反转是同一概念吗？

下面就来简要的回答一下上述问题，把这些问题搞明白了，IoC/DI也就明白了。

（1）参与者都有谁：

一般有三方参与者，一个是某个对象；一个是IoC/DI的容器；另一个是某个对象的外部资源。

又要名词解释一下，某个对象指的就是任意的、普通的Java对象; IoC/DI的容器简单点说就是指用来实现IoC/DI功能的一个框架程序；对象的外部资源指的就是对象需要的，但是是从对象外部获取的，都统称资源，比如：对象需要的其它对象、或者是对象需要的文件资源等等。

（2）谁依赖于谁：

当然是某个对象依赖于IoC/DI的容器

（3）为什么需要依赖：

对象需要IoC/DI的容器来提供对象需要的外部资源

（4）谁注入于谁：

很明显是IoC/DI的容器 注入 某个对象

（5）到底注入什么：

就是注入某个对象所需要的外部资源

（6）谁控制谁：

当然是IoC/DI的容器来控制对象了

（7）控制什么：

主要是控制对象实例的创建

（8）为何叫反转：

反转是相对于正向而言的，那么什么算是正向的呢？考虑一下常规情况下的应用程序，如果要在A里面使用C，你会怎么做呢？当然是直接去创建C的对象，也就是说，是在A类中主动去获取所需要的外部资源C，这种情况被称为正向的。那么什么是反向呢？就是A类不再主动去获取C，而是被动等待，等待IoC/DI的容器获取一个C的实例，然后反向的注入到A类中。

用图例来说明一下，先看没有IoC/DI的时候，常规的A类使用C类的示意图，如图7所示：

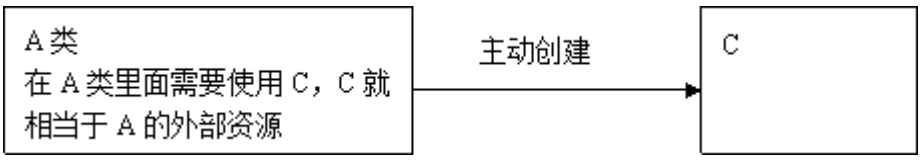


图7 常规A使用C示意图

当有了IoC/DI的容器后，A类不再主动去创建C了，如图8所示：

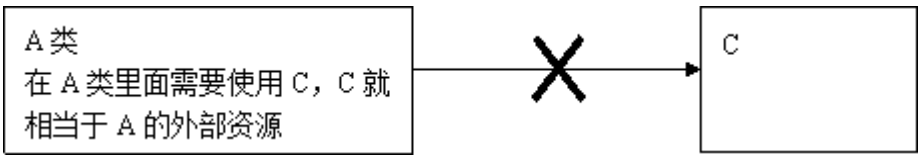


图8 A类不再主动创建C

而是被动等待，等待IoC/DI的容器获取一个C的实例，然后反向的注入到A类中，如图9所示：

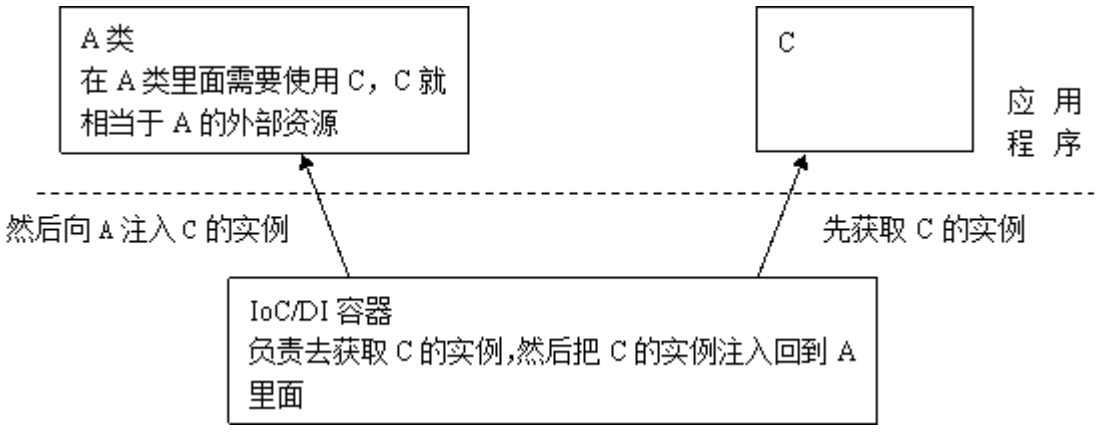


图9 有IoC/DI容器后程序结构示意图

（9）依赖注入和控制反转是同一概念吗？

根据上面的讲述，应该能看出来，依赖注入和控制反转是对同一件事情的不同描述，从某个方面讲，就是它们描述的角度不同。依赖注入是从应用程序的角度在描述，可以把依赖注入描述完整点：应用程序依赖容器创建并注入它所需要的外部资源；而控制反转是从容器的角度在描述，描述完整点：容器控制应用程序，由容器反向的向应用程序注入应用程序所需要的外部资源。

（10）小结一下：

其实IoC/DI对编程带来的最大改变不是从代码上，而是从思想上，发生了“主从换位”的变化。应用程序原本是老大，要获取什么资源都是主动出击，但是在IoC/DI思想中，应用程序就变成被动的了，被动的等待IoC/DI容器来创建并注入它所需要的资源了。

这么小小的一个改变其实是编程思想的一个大进步，这样就有效的分离了对象和它所需要的外部资源，使得它们松散耦合，有利于功能复用，更重要的是使得程序的整个体系结构变得非常灵活。

2：工厂方法模式和IoC/DI有什么关系呢？

从某个角度讲，它们的思想很类似。

上面讲了，有了IoC/DI过后，应用程序就不再主动了，而是被动等待由容器来注入资源，那么在编写代码的时候，一旦要用到外部资源，就会开一个窗口，让容器能注入进来，也就是提供给容器使用的注入的途径，当然这不是我们的重点，就不去细细讲了，用setter注入来示例一下，看看使用IoC/DI的代码是什么样子，示例代码如下：

```
public class A {
    /**
     * 等待被注入进来
     */
    private C c = null;
    /**
     * 注入资源C的方法
     * @param c 被注入的资源
     */
    public void setC(C c){
       this.c = c;
    }
    public void t1(){
       //这里需要使用C，可是又不让主动去创建C了，怎么办？
       //反正就要求从外部注入，这样更省心，
       //自己不用管怎么获取C，直接使用就好了
       c.tc();
```

```
    }
}
```

接口C的示例代码如下：

```
public interface C {
    public void tc();
}
```

从上面的示例代码可以看出，现在在A里面写代码的时候，凡是碰到了需要外部资源，那么就提供注入的途径，要求从外部注入，自己只管使用这些对象。

再来看看工厂方法模式，如何实现上面同样的功能，为了区分，分别取名为A1和C1。这个时候在A1里面要使用C1对象，也不是由A1主动去获取C1对象，而是创建一个工厂方法，就类似于一个注入的途径；然后由子类，假设叫A2吧，由A2来获取C1对象，在调用的时候，替换掉A1的相应方法，相当于反向注入回到A1里面，示例代码如下：

```
public abstract class A1 {
    /**
     * 工厂方法，创建C1，类似于从子类注入进来的途径
     * @return C1的对象实例
     */
    protected abstract C1 createC1();
    public void t1(){
      //这里需要使用C1类，可是不知道究竟是用哪一个
          //也就不主动去创建C1了，怎么办？
      //反正会在子类里面实现，这里不用管怎么获取C1，直接使用就好了
      createC1().tc();
    }
}
```

子类的示例代码如下：

```
public class A2 extends A1 {
    protected C1 createC1() {
      //真正的选择具体实现，并创建对象
      return new C2();
```

```
    }
}
```

C1接口和前面C接口是一样的，C2这个实现类也是空的，只是演示一下，因此就不去展示它们的代码了。

仔细体会上面的示例，对比它们的实现，尤其是从思想层面上，会发现工厂方法模式和IoC/DI的思想是相似的，都是“**主动变被动**”，进行了“**主从换位**”，从而获得了更灵活的程序结构。

未完待续.....

1.5 研磨设计模式之工厂方法模式-5

发表时间: 2010-06-20

3.3 平行的类层次结构

（1）什么是平行的类层次结构呢？

简单点说，假如有两个类层次结构，其中一个类层次中的每个类在另一个类层次中都有一个对应的类的结构，就被称为平行的类层次结构。

举个例子来说，硬盘对象有很多种，如分成台式机硬盘和笔记本硬盘，在台式机硬盘的具体实现上面，又有希捷、西数等不同品牌的实现，同样在笔记本硬盘上，也有希捷、日立、IBM等不同品牌的实现；硬盘对象具有自己的行为，如硬盘能存储数据，也能从硬盘上获取数据，不同的硬盘对象对应的行为对象是不一样的，因为不同的硬盘对象，它的行为的实现方式是不一样的。如果把硬盘对象和硬盘对象的行为分开描述，那么就构成了如图10所示的结构：

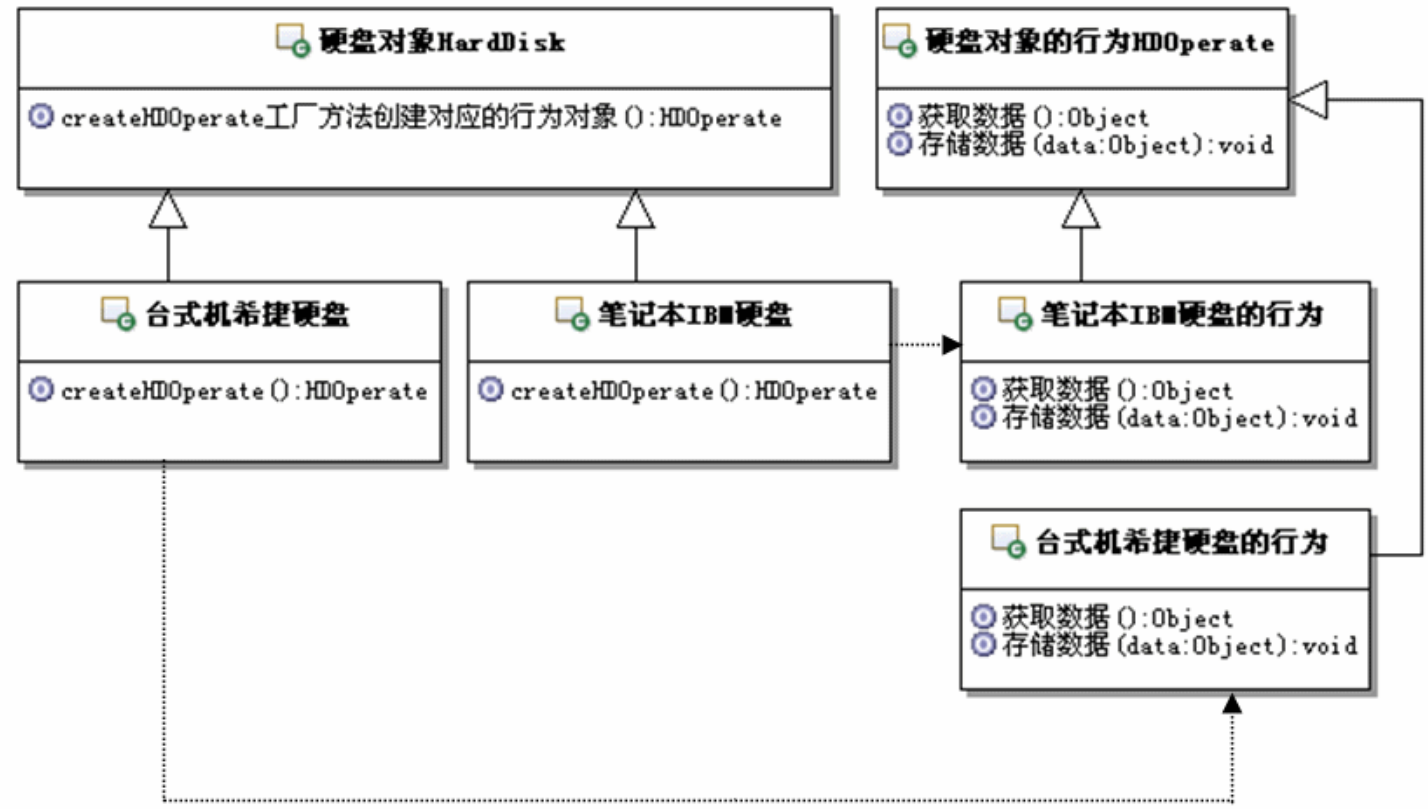


图10 平行的类层次结构示意图

硬盘对象是一个类层次，硬盘的行为这边也是一个类层次，而且两个类层次中的类是对应的。台式机西捷硬盘对象就对应着硬盘行为里面的台式机西捷硬盘的行为；笔记本IBM硬盘就对应着笔记本IBM硬盘的行为，这就是一种典型的平行的类层次结构。

这种平行的类层次结构用来干什么呢？主要用来把一个类层次中的某些行为分离出来，让类层次中的类把

原本属于自己的职责，委托给分离出来的类去实现，从而使得类层次本身变得更简单，更容易扩展和复用。一般来讲，分离出去的这些类的行为，会对应着类层次结构来组织，从而形成一个新的类层次结构，相当于原来对象的行为的这么一个类层次结构，而这个层次结构和原来的类层次结构是存在对应关系的，因此被称为平行的类层次结构。

（2）工厂方法模式跟平行的类层次结构有何关系呢？

可以使用工厂方法模式来连接平行的类层次。

看上面的示例图10，在每个硬盘对象里面，都有一个工厂方法createHDOperate，通过这个工厂方法，客户端就可以获取一个跟硬盘对象相对应的行为对象。在硬盘对象的子类里面，会覆盖父类的工厂方法createHDOperate，以提供跟自身相对应的行为对象，从而自然的把两个平行的类层次连接起来使用。

3.4 参数化工厂方法

所谓参数化工厂方法指的就是：**通过给工厂方法传递参数，让工厂方法根据参数的不同来创建不同的产品对象，这种情况就被称为参数化工厂方法。**当然工厂方法创建的不同的产品必须是同一个Product类型的。

来改造前面的示例，现在有一个工厂方法来创建ExportFileApi这个产品的对象，但是ExportFileApi接口的具体实现很多，为了方便创建的选择，直接从客户端传入一个参数，这样在需要创建ExportFileApi对象的时候，就把这个参数传递给工厂方法，让工厂方法来实例化具体的ExportFileApi实现对象。

还是看看代码示例会比较清楚。

（1）先来看Product的接口，就是ExportFileApi接口，跟前面的示例没有任何变化，为了方便大家查看，这里重复一下，示例代码如下：

```
/**
 * 导出的文件对象的接口
 */
public interface ExportFileApi {
    /**
     * 导出内容成为文件
     * @param data 示意：需要保存的数据
     * @return 是否导出成功
     */
    public boolean export(String data);
}
```

（2）同样提供保存成文本文件和保存成数据库备份文件的实现，跟前面的示例没有任何变化，示例代码如下：

```
public class ExportTxtFile implements ExportFileApi{
    public boolean export(String data) {
      //简单示意一下，这里需要操作文件
      System.out.println("导出数据"+data+"到文本文件");
      return true;
    }
}
public class ExportDB implements ExportFileApi{
    public boolean export(String data) {
      //简单示意一下，这里需要操作数据库和文件
      System.out.println("导出数据"+data+"到数据库备份文件");
      return true;
    }
}
```

（3）接下来该看看ExportOperate类了，这个类的变化大致如下：

- ExportOperate类中的创建产品的工厂方法，通常需要提供默认的实现，不抽象了，也就是变成正常方法
- ExportOperate类也不再定义成抽象类了，因为有了默认的实现，客户端可能需要直接使用这个对象
- 设置一个导出类型的参数，通过export方法从客户端传入

看看代码吧，示例代码如下：

```
/**
 * 实现导出数据的业务功能对象
 */


public class ExportOperate {
    /**
     * 导出文件
     * @param type 用户选择的导出类型


     * @param data 需要保存的数据
     * @return 是否成功导出文件
     */
    public boolean export(int type,String data){
      //使用工厂方法
      ExportFileApi api = factoryMethod(type);
      return api.export(data);
    }
    /**
     * 工厂方法，创建导出的文件对象的接口对象
     * @param type 用户选择的导出类型
     * @return 导出的文件对象的接口对象
```

```
     */
    protected ExportFileApi factoryMethod(int type){


        ExportFileApi api = null;
        //根据类型来选择究竟要创建哪一种导出文件对象
        if(type==1){
            api = new ExportTxtFile();
        }else if(type==2){
            api = new ExportDB();
        }
        return api;
    }
}
```

（4）此时的客户端，非常简单，直接使用ExportOperate类，示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建需要使用的Creator对象
        ExportOperate operate = new ExportOperate();
        //调用输出数据的功能方法，传入选择到处类型的参数
        operate.export(1,"测试数据");
    }
}
```

测试看看，然后修改一下客户端的参数，体会一下通过参数来选择具体的导出实现的过程。这是一种很常见的参数化工厂方法的实现方式，但是也还是有把参数化工厂方法实现成为抽象的，这点要注意，并不是说参数化工厂方法就不能实现成为抽象类了。只是一般情况下，参数化工厂方法，在父类都会提供默认的实现。

（5）扩展新的实现

使用参数化工厂方法，扩展起来会非常容易，已有的代码都不会改变，只要新加入一个子类来提供新的工厂方法实现，然后在客户端使用这个新的子类即可。

这种实现方式还有一个有意思的功能，就是子类可以选择性覆盖，不想覆盖的功能还可以返回去让父类来实现，很有意思。

先扩展一个导出成xml文件的实现，试试看，示例代码如下：

```
/**
 * 导出成xml文件的对象
 */
public class ExportXml implements ExportFileApi{
    public boolean export(String data) {
```

```
        //简单示意一下
        System.out.println("导出数据"+data+"到XML文件");
         return true;
    }
}
```

然后扩展ExportOperate类，来加入新的实现，示例代码如下：

```
/**
 * 扩展ExportOperate对象，加入可以导出XML文件
 */
public class ExportOperate2 extends ExportOperate{
    /**
     * 覆盖父类的工厂方法，创建导出的文件对象的接口对象
     * @param type 用户选择的导出类型
     * @return 导出的文件对象的接口对象
     */
    protected ExportFileApi factoryMethod(int type){
        ExportFileApi api = null;
        //可以全部覆盖，也可以选择自己感兴趣的覆盖，
        //这里只想添加自己新的实现，其它的不管
        if(type==3){
            api = new ExportXml();
        }else{
            //其它的还是让父类来实现
            api = super.factoryMethod(type);
        }
        return api;
    }
}
```

看看此时的客户端，也非常简单，只是在变换传入的参数，示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建需要使用的Creator对象
        ExportOperate operate = new ExportOperate2();
        //下面变换传入的参数来测试参数化工厂方法
        operate.export(1,"Test1");
        operate.export(2,"Test2");
        operate.export(3,"Test3");
    }
}
```

对应的测试结果如下：

```
导出数据Test1到文本文件
导出数据Test2到数据库备份文件
导出数据Test3到XML文件
```

通过上面的示例，好好体会一下参数化工厂方法的实现和带来的好处。

3.5 工厂方法模式的优缺点

- 可以在不知具体实现的情况下编程

  工厂方法模式可以让你在实现功能的时候，如果需要某个产品对象，只需要使用产品的接口即可，而无需关心具体的实现。选择具体实现的任务延迟到子类去完成。
- 更容易扩展对象的新版本

  工厂方法给子类提供了一个挂钩，使得扩展新的对象版本变得非常容易。比如上面示例的参数化工厂方法实现中，扩展一个新的导出Xml文件格式的实现，已有的代码都不会改变，只要新加入一个子类来提供新的工厂方法实现，然后在客户端使用这个新的子类即可。

  另外这里提到的挂钩，就是我们经常说的钩子方法(hook)，这个会在后面讲模板方法模式的时候详细点说明。
- 连接平行的类层次

  工厂方法除了创造产品对象外，在连接平行的类层次上也大显身手。这个在前面已经详细讲述了。
- 具体产品对象和工厂方法的耦合性

  在工厂方法模式里面，工厂方法是需要创建产品对象的，也就是需要选择具体的产品对象，并创建它们的实例，因此具体产品对象和工厂方法是耦合的。

3.6 思考工厂方法模式

1：工厂方法模式的本质

工厂方法模式的本质：**延迟到子类来选择实现。**

仔细体会前面的示例，你会发现，工厂方法模式中的工厂方法，在真正实现的时候，一般是先选择具体使用哪一个具体的产品实现对象，然后创建这个具体产品对象的示例，然后就可以返回去了。也就是说，工厂方法本身并不会去实现产品接口，具体的产品实现是已经写好了的，工厂方法只要去选择实现就好了。

有些朋友可能会说，这不是跟简单工厂一样吗？

确实从本质上讲，它们是非常类似的，具体实现上都是在“选择实现”。但是也存在不同点，简单工厂是直接在工厂类里面进行“选择实现”；而工厂方法会把这个工作延迟到子类来实现，工厂类里面使用工厂方法的地方是依赖于抽象而不是具体的实现，从而使得系统更加灵活，具有更好的可维护性和可扩展性。

其实如果把工厂模式中的Creator退化一下，只提供工厂方法，而且这些工厂方法还都提供默认的实现，那不就变成了简单工厂了吗？比如把刚才示范参数化工厂方法的例子代码拿过来再简化一下，你就能看出来，写得跟简单工厂是差不多的，示例代码如下：

```
public class ExportOperate {
    /**
     * 导出文件
     * @param type 用户选择的导出类型
     * @param data 需要保存的数据
     * @return 是否成功导出文件
     */
    public boolean export(int type,String data){
        //使用工厂方法
        ExportFileApi api = factoryMethod(type);
        return api.export(data);
    }
    /**
     * 工厂方法，创建导出的文件对象的接口对象
     * @param type 用户选择的导出类型
     * @return 导出的文件对象的接口对象
     */
    protected ExportFileApi factoryMethod(int type){
        ExportFileApi api = null;
        //根据类型来选择究竟要创建哪一种导出文件对象
        if(type==1){
            api = new ExportTxtFile();
        }else if(type==2){
            api = new ExportDB();
        }
        return api;
    }
}
```

简化这个Creator，把这些都删除

留下的这个方法，如果把它修改成public static的，是不是就和简单工厂写得一样了

看完上述代码，会体会到简单工厂和工厂方法模式是有很大相似性的了吧，从某个角度来讲，可以认为简单工厂就是工厂方法模式的一种特例，因此它们的本质是类似的，也就不足为奇了。

2：对设计原则的体现

工厂方法模式很好的体现了“依赖倒置原则”。

依赖倒置原则告诉我们“要依赖抽象，不要依赖于具体类”，简单点说就是：不能让高层组件依赖于低层组件，而且不管高层组件还是低层组件，都应该依赖于抽象。

比如前面的示例，实现客户端请求操作的ExportOperate就是高层组件；而具体实现数据导出的对象就是

低层组件，比如ExportTxtFile、ExportDB；而ExportFileApi接口就相当于是那个抽象。

对于ExportOperate来说，它不关心具体的实现方式，它只是“面向接口编程”；对于具体的实现来说，它只关心自己“如何实现接口”所要求的功能。

那么倒置的是什么呢？倒置的是这个接口的“所有权”。事实上，ExportFileApi接口中定义的功能，都是由高层组件ExportOperate来提出的要求，也就是说接口中的功能，是高层组件需要的功能。但是高层组件只是提出要求，并不关心如何实现，而低层组件，就是来真正实现高层组件所要求的接口功能的。因此看起来，低层实现的接口的所有权并不在底层组件手中，而是倒置到高层组件去了。

3：何时选用工厂方法模式

建议在如下情况中，选用工厂方法模式：

- 如果一个类需要创建某个接口的对象，但是又不知道具体的实现，这种情况可以选用工厂方法模式，把创建对象的工作延迟到子类去实现
- 如果一个类本身就希望，由它的子类来创建所需的对象的时候，应该使用工厂方法模式

3.7 相关模式

- 工厂方法模式和抽象工厂模式

  这两个模式可以组合使用，具体的放到抽象工厂模式中去讲。
- 工厂方法模式和模板方法模式

  这两个模式外观类似，都是有一个抽象类，然后由子类来提供一些实现，但是工厂方法模式的子类专注的是创建产品对象，而模板方法模式的子类专注的是为固定的算法骨架提供某些步骤的实现。

  这两个模式可以组合使用，通常在模板方法模式里面，使用工厂方法来创建模板方法需要的对象。

工厂方法模式结束,谢谢观看!

1.6 研磨设计模式之单例模式-1

发表时间: 2010-07-26

看到很多朋友在写单例，也来凑个热闹，虽然很简单，但是也有很多知识点在单例里面，看看是否能写出点不一样来。

单例模式（Singleton）

1 场景问题

1.1 读取配置文件的内容

考虑这样一个应用，读取配置文件的内容。

很多应用项目，都有与应用相关的配置文件，这些配置文件多是由项目开发人员自定义的，在里面定义一些应用需要的参数数据。当然在实际的项目中，这种配置文件多采用xml格式的。也有采用properties格式的，毕竟使用Java来读取properties格式的配置文件比较简单。

现在要读取配置文件的内容，该如何实现呢？

1.2 不用模式的解决方案

有些朋友会想，要读取配置文件的内容，这也不是个什么困难的事情，直接读取文件的内容，然后把文件内容存放在相应的数据对象里面就可以了。真的这么简单吗？先实现看看吧。

为了示例简单，假设系统是采用的properties格式的配置文件。

（1）那么直接使用Java来读取配置文件，示例代码如下：

```
/**
 * 读取应用配置文件
 */
public class AppConfig {
        /**
         * 用来存放配置文件中参数A的值
         */
```

```
        private String parameterA;
        /**
         * 用来存放配置文件中参数B的值
         */
        private String parameterB;

        public String getParameterA() {
                return parameterA;
        }
        public String getParameterB() {
                return parameterB;
        }
        /**
         * 构造方法
         */
        public AppConfig(){
                //调用读取配置文件的方法
                readConfig();
        }
        /**
         * 读取配置文件，把配置文件中的内容读出来设置到属性上
         */
        private void readConfig(){
                Properties p = new Properties();
                InputStream in = null;
                try {
                        in = AppConfig.class.getResourceAsStream(
"AppConfig.properties");
                        p.load(in);
                        //把配置文件中的内容读出来设置到属性上
                        this.parameterA = p.getProperty("paramA");
                        this.parameterB = p.getProperty("paramB");
                } catch (IOException e) {
                        System.out.println("装载配置文件出错了，具体堆栈信息如下：");
                        e.printStackTrace();
                }finally{
                        try {
```

```
                        in.close();
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
    }
}
```

注意：只有访问参数的方法，没有设置参数的方法。

（2）应用的配置文件，名字是AppConfig.properties，放在AppConfig相同的包里面，简单示例如下：

```
paramA=a
paramB=b
```

（3）写个客户端来测试一下，示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //创建读取应用配置的对象
        AppConfig config = new AppConfig();

        String paramA = config.getParameterA();
        String paramB = config.getParameterB();

        System.out.println("paramA="+paramA+",paramB="+paramB);
    }
}
```

运行结果如下：

```
paramA=a,paramB=b
```

1.3 有何问题

上面的实现很简单嘛，很容易的就实现了要求的功能。仔细想想，有没有什么问题呢？

看看客户端使用这个类的地方，是通过new一个AppConfig的实例来得到一个操作配置文件内容的对象。如果在系统运行中，有很多地方都需要使用配置文件的内容，也就是很多地方都需要创建AppConfig这个对象的实例。

换句话说，在系统运行期间，系统中会存在很多个AppConfig的实例对象，这有什么问题吗？

当然有问题了，试想一下，每一个AppConfig实例对象，里面都封装着配置文件的内容，系统中有多个AppConfig实例对象，也就是说系统中会同时存在多份配置文件的内容，这会严重浪费内存资源。如果配置文件内容较少，问题还小一点，如果配置文件内容本来就多的话，对于系统资源的浪费问题就大了。事实上，对于AppConfig这种类，在运行期间，只需要一个实例对象就够了。

把上面的描述进一步抽象一下，问题就出来了：在一个系统运行期间，某个类只需要一个类实例就可以了，那么应该怎么实现呢？

2 解决方案

2.1 单例模式来解决

用来解决上述问题的一个合理的解决方案就是单例模式。那么什么是单例模式呢？

（1）单例模式定义

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。

（2）应用单例模式来解决的思路

仔细分析上面的问题，现在一个类能够被创建多个实例，问题的根源在于类的构造方法是公开的，也就是可以让类的外部来通过构造方法创建多个实例。换句话说，只要类的构造方法能让类的外部访问，就没有办法去控制外部来创建这个类的实例个数。

要想控制一个类只被创建一个实例，那么首要的问题就是要把创建实例的权限收回来，让类自身来负责自己类实例的创建工作，然后由这个类来提供外部可以访问这个类实例的方法，这就是单例模式的实现方式。

2.2 模式结构和说明

单例模式结构见图1所：

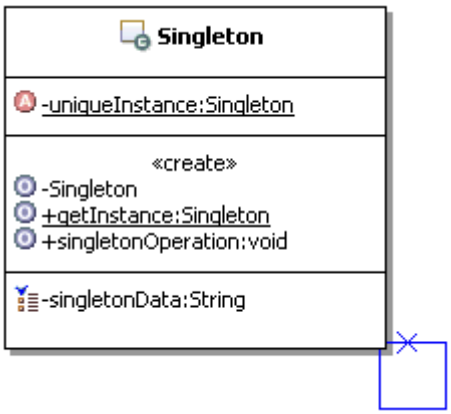


图1 单例模式结构图

Singleton：

负责创建Singleton类自己的唯一实例，并提供一个getInstance的方法，让外部来访问这个类的唯一实例。

2.3 单例模式示例代码

在Java中，单例模式的实现又分为两种，一种称为懒汉式，一种称为饿汉式，其实就是在具体创建对象实例的处理上，有不同的实现方式。下面分别来看这两种实现方式的代码示例。为何这么写，具体的在后面再讲述。

（1）懒汉式实现，示例代码如下：

```
/**
 * 懒汉式单例实现的示例
 */
public class Singleton {
        /**
         * 定义一个变量来存储创建好的类实例
         */
        private static Singleton uniqueInstance = null;
        /**
         * 私有化构造方法，好在内部控制创建实例的数目
         */
        private Singleton(){
                //
        }
        /**
         * 定义一个方法来为客户端提供类实例
         * @return 一个Singleton的实例
```

```
         */
        public static synchronized Singleton getInstance(){
                //判断存储实例的变量是否有值
                if(uniqueInstance == null){
                        //如果没有，就创建一个类实例，并把值赋值给存储类实例的变量
                        uniqueInstance = new Singleton();
                }
                //如果有值，那就直接使用
                return uniqueInstance;
        }
        /**
         * 示意方法，单例可以有自己的操作
         */
        public void singletonOperation(){
                //功能处理
        }
        /**
         * 示意属性，单例可以有自己的属性
         */
        private String singletonData;
        /**
         * 示意方法，让外部通过这些方法来访问属性的值
         * @return 属性的值
         */
        public String getSingletonData(){
                return singletonData;
        }
}
```

（2）饿汉式实现，示例代码如下：

```
/**
 * 饿汉式单例实现的示例
 */
public class Singleton {
        /**
         * 定义一个变量来存储创建好的类实例，直接在这里创建类实例，只会创建一次
```

```
         */
        private static Singleton uniqueInstance = new Singleton();
        /**
         * 私有化构造方法，好在内部控制创建实例的数目
         */
        private Singleton(){
                //
        }
        /**
         * 定义一个方法来为客户端提供类实例
         * @return 一个Singleton的实例
         */
        public static Singleton getInstance(){
                //直接使用已经创建好的实例
                return uniqueInstance;
        }

        /**
         * 示意方法，单例可以有自己的操作
         */
        public void singletonOperation(){
                //功能处理
        }
        /**
         * 示意属性，单例可以有自己的属性
         */
        private String singletonData;
        /**
         * 示意方法，让外部通过这些方法来访问属性的值
         * @return 属性的值
         */
        public String getSingletonData(){
                return singletonData;
        }
}
```

2.4 使用单例模式重写示例

要使用单例模式来重写示例，由于单例模式有两种实现方式，这里选一种来实现就好了，就选择饿汉式的实现方式来重写示例吧。

采用饿汉式的实现方式来重写实例的示例代码如下：

```
/**
 * 读取应用配置文件，单例实现
 */
public class AppConfig {
        /**
         * 定义一个变量来存储创建好的类实例，直接在这里创建类实例，只会创建一次
         */
        private static AppConfig instance = new AppConfig();
        /**
         * 定义一个方法来为客户端提供AppConfig类的实例
         * @return 一个AppConfig的实例
         */
        public static AppConfig getInstance(){
                return instance;
        }

        /**
         * 用来存放配置文件中参数A的值
         */
        private String parameterA;
        /**
         * 用来存放配置文件中参数B的值
         */
        private String parameterB;
        public String getParameterA() {
                return parameterA;
        }
        public String getParameterB() {
                return parameterB;
        }
        /**
```

```
         * 私有化构造方法
         */
        private AppConfig(){
                //调用读取配置文件的方法
                readConfig();
        }
        /**
         * 读取配置文件，把配置文件中的内容读出来设置到属性上
         */
        private void readConfig(){
                Properties p = new Properties();
                InputStream in = null;
                try {
                        in = AppConfig.class.getResourceAsStream(
"AppConfig.properties");
                        p.load(in);
                        //把配置文件中的内容读出来设置到属性上
                        this.parameterA = p.getProperty("paramA");
                        this.parameterB = p.getProperty("paramB");
                } catch (IOException e) {
                        System.out.println("装载配置文件出错了，具体堆栈信息如下：");
                        e.printStackTrace();
                }finally{
                        try {
                                in.close();
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                }
        }
}
```

当然，测试的客户端也需要相应的变化，示例代码如下：

```
public class Client {
        public static void main(String[] args) {
                //创建读取应用配置的对象
```

```
                AppConfig config = AppConfig.getInstance();

                String paramA = config.getParameterA();
                String paramB = config.getParameterB();

                System.out.println("paramA="+paramA+",paramB="+paramB);
        }
}
```

去测试看看，是否能满足要求。

未完待续，精彩稍后继续

1.7 研磨设计模式之单例模式-2

发表时间: 2010-07-29

3 模式讲解

3.1 认识单例模式

（1）单例模式的功能

单例模式的功能是用来保证这个类在运行期间只会被创建一个类实例，另外单例模式还提供了一个全局唯一访问这个类实例的访问点，就是那个getInstance的方法。不管采用懒汉式还是饿汉式的实现方式，这个全局访问点是一样的。

对于单例模式而言，不管采用何种实现方式，它都是只关心类实例的创建问题，并不关心具体的业务功能。

（2）单例模式的范围

也就是在多大范围内是单例呢？

观察上面的实现可以知道，目前Java里面实现的单例是一个ClassLoader及其子ClassLoader的范围。因为一个ClassLoader在装载饿汉式实现的单例类的时候就会创建一个类的实例。

这就意味着如果一个虚拟机里面有很多个ClassLoader，而且这些ClassLoader都装载某个类的话，就算这个类是单例，它也会产生很多个实例。当然，如果一个机器上有多个虚拟机，那么每个虚拟机里面都应该至少有一个这个类的实例，也就是说整个机器上就有很多个实例，更不会是单例了。

另外请注意一点，这里讨论的单例模式并不适用于集群环境，对于集群环境下的单例这里不去讨论，那不属于这里的内容范围。

（3）单例模式的命名

一般建议单例模式的方法命名为：getInstance()，这个方法的返回类型肯定是单例类的类型了。getInstance方法可以有参数，这些参数可能是创建类实例所需要的参数，当然，大多数情况下是不需要的。

单例模式的名称：单例、单件、单体等等，翻译的不同，都是指的同一个模式。

3.2 懒汉式和饿汉式实现

前面提到了单例模式有两种典型的解决方案，一种叫懒汉式，一种叫饿汉式，这两种方式究竟是如何实现的，下面分别来看看。为了看得更清晰一点，只是实现基本的单例控制部分，不再提供示例的属性和方法了；而且暂时也不去考虑线程安全的问题，这个问题在后面会重点分析。

1：第一种方案 懒汉式

（1）私有化构造方法

要想在运行期间控制某一个类的实例只有一个，那首先的任务就是要控制创建实例的地方，也就是不能随随便便就可以创建类实例，否则就无法控制创建的实例个数了。现在是让使用类的地方来创建类实例，也就是

在类外部来创建类实例。

那么怎样才能让类的外部不能创建一个类的实例呢？很简单，私有化构造方法就可以了！示例代码如下：

```
private Singleton(){
}
```

（2）提供获取实例的方法

构造方法被私有化了，外部使用这个类的地方不干了，外部创建不了类实例就没有办法调用这个对象的方法，就实现不了功能处理，这可不行。经过思考，单例模式决定让这个类提供一个方法来返回类的实例，好让外面使用。示例代码如下：

```
public Singleton getInstance(){
}
```

（3）把获取实例的方法变成静态的

又有新的问题了，获取对象实例的这个方法是个实例方法，也就是说客户端要想调用这个方法，需要先得到类实例，然后才可以调用，可是这个方法就是为了得到类实例，这样一来不就形成一个死循环了吗？这不就是典型的“先有鸡还是先有蛋的问题”嘛。

解决方法也很简单，在方法上加上static，这样就可以直接通过类来调用这个方法，而不需要先得到类实例了，示例代码如下：

```
public static Singleton getInstance(){
}
```

（4）定义存储实例的属性

方法定义好了，那么方法内部如何实现呢？如果直接创建实例并返回，这样行不行呢？示例代码如下

```
public static Singleton getInstance(){
        return new Singleton();
}
```

当然不行了，如果每次客户端访问都这样直接new一个实例，那肯定会有多个实例，根本实现不了单例的功能。

怎么办呢？单例模式想到了一个办法，那就是用一个属性来记录自己创建好的类实例，当第一次创建过后，就把这个实例保存下来，以后就可以复用这个实例，而不是重复创建对象实例了。示例代码如下：

```
private Singleton instance = null;
```

（5）把这个属性也定义成静态的

这个属性变量应该在什么地方用呢？肯定是第一次创建类实例的地方，也就是在前面那个返回对象实例的静态方法里面使用。

由于要在一个静态方法里面使用，所以这个属性被迫成为一个类变量，要强制加上static，也就是说，这里并没有使用static的特性。示例代码如下：

```
private static Singleton instance = null;
```

（6）实现控制实例的创建

现在应该到getInstance方法里面实现控制实例创建了，控制的方式很简单，只要先判断一下，是否已经创建过实例了。如何判断？那就看存放实例的属性是否有值，如果有值，说明已经创建过了，如果没有值，那就是应该创建一个，示例代码如下：

```
public static Singleton getInstance(){
        //先判断instance是否有值
        if(instance == null){
        //如果没有值，说明还没有创建过实例，那就创建一个
        //并把这个实例设置给instance
            instance = new Singleton ();
        }
        //如果有值，或者是创建了值，那就直接使用
        return instance;
}
```

（7）完整的实现

至此，成功解决了：在运行期间，控制某个类只被创建一个实例的要求。完整的代码如下，**为了大家好理解，用注释标示了代码的先后顺序**，示例代码如下：

```
public class Singleton {
        //4：定义一个变量来存储创建好的类实例
        //5：因为这个变量要在静态方法中使用，所以需要加上static修饰
        private static Singleton instance = null;
```

```
        //1：私有化构造方法，好在内部控制创建实例的数目
        private Singleton(){
        }
        //2：定义一个方法来为客户端提供类实例
        //3：这个方法需要定义成类方法，也就是要加static
        public static Singleton getInstance(){
                //6：判断存储实例的变量是否有值
                if(instance == null){
                        //6.1：如果没有，就创建一个类实例，并把值赋值给存储类实例的变量
                        instance = new Singleton();
                }
                //6.2：如果有值，那就直接使用
                return instance;
        }
}
```

2：第二种方案 饿汉式

这种方案跟第一种方案相比，前面的私有化构造方法，提供静态的getInstance方法来返回实例等步骤都一样。差别在如何实现getInstance方法，在这个地方，单例模式还想到了另外一种方法来实现getInstance方法。

不就是要控制只创造一个实例吗？那么有没有什么现成的解决办法呢？很快，单例模式回忆起了Java中static的特性：

- static变量在类装载的时候进行初始化
- 多个实例的static变量会共享同一块内存区域。

这就意味着，在Java中，static变量只会被初始化一次，就是在类装载的时候，而且多个实例都会共享这个内存空间，这不就是单例模式要实现的功能吗？真是得来全不费功夫啊。根据这些知识，写出了第二种解决方案的代码，示例代码如下：

```
public class Singleton {
        //4：定义一个静态变量来存储创建好的类实例
        //直接在这里创建类实例，只会创建一次
        private static Singleton instance = new Singleton();
        //1：私有化构造方法，好在内部控制创建实例的数目
        private Singleton(){
        }
```

```
        //2：定义一个方法来为客户端提供类实例
        //3：这个方法需要定义成类方法，也就是要加static
        //这个方法里面就不需要控制代码了
        public static Singleton getInstance(){
                //5：直接使用已经创建好的实例
                return instance;
        }
}
```

注意一下，这个方案是用到了static的特性的，而第一个方案是没有用到的，因此两个方案的步骤会有一些不同，在第一个方案里面，强制加上static也是算作一步的，而在这个方案里面，是主动加上static，就不单独算作一步了。

所以在查看上面两种方案的代码的时候，仔细看看编号，顺着编号的顺序看，可以体会出两种方案的不一样来。

不管是采用哪一种方式，在运行期间，都只会生成一个实例，而访问这些类的一个全局访问点，就是那个静态的getInstance方法。

3：单例模式的调用顺序示意图

由于单例模式有两种实现方式，那么它的调用顺序也分成两种。先看懒汉式的调用顺序，如图2所示：

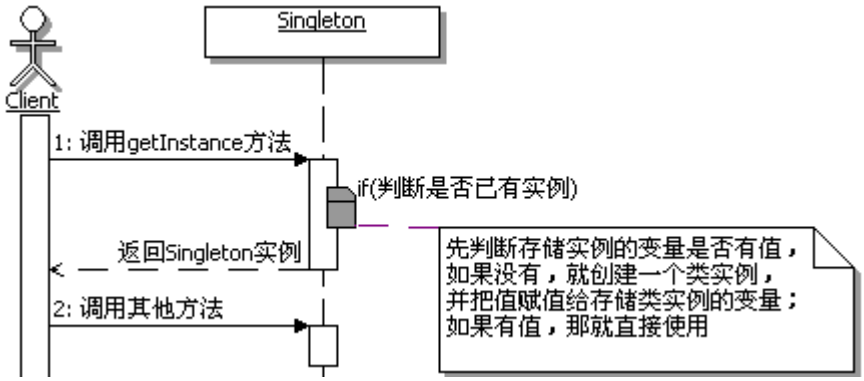


图2 懒汉式调用顺序示意图

饿汉式的调用顺序，如图3所示：

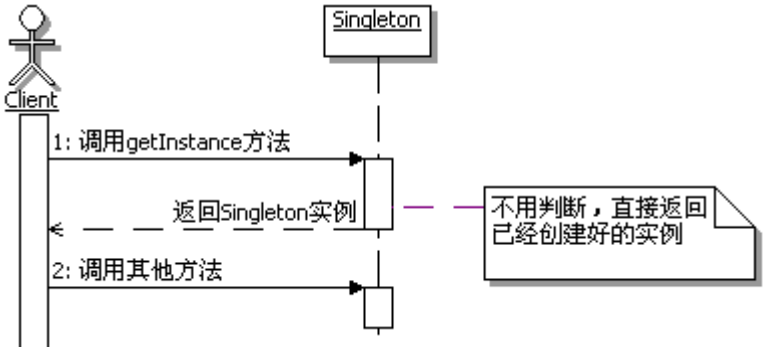


图3 饿汉式调用顺序示意图

未完待续

1.8 研磨设计模式之单例模式-3

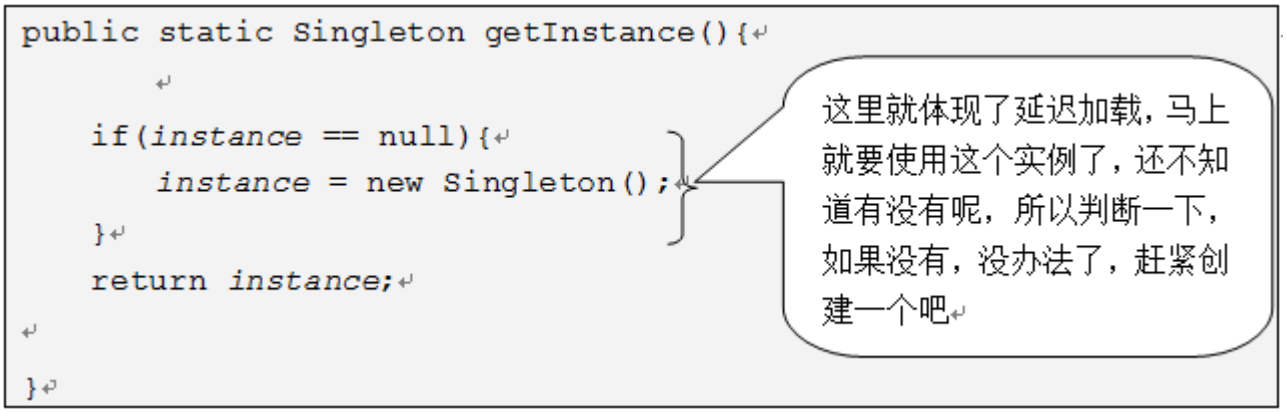
发表时间: 2010-08-02

3.3 延迟加载的思想

单例模式的懒汉式实现方式体现了延迟加载的思想，什么是延迟加载呢？

通俗点说，就是一开始不要加载资源或者数据，一直等，等到马上就要使用这个资源或者数据了，躲不过去了才加载，所以也称Lazy Load，不是懒惰啊，是“延迟加载”，这在实际开发中是一种很常见的思想，尽可能的节约资源。

体现在什么地方呢？看如下代码：



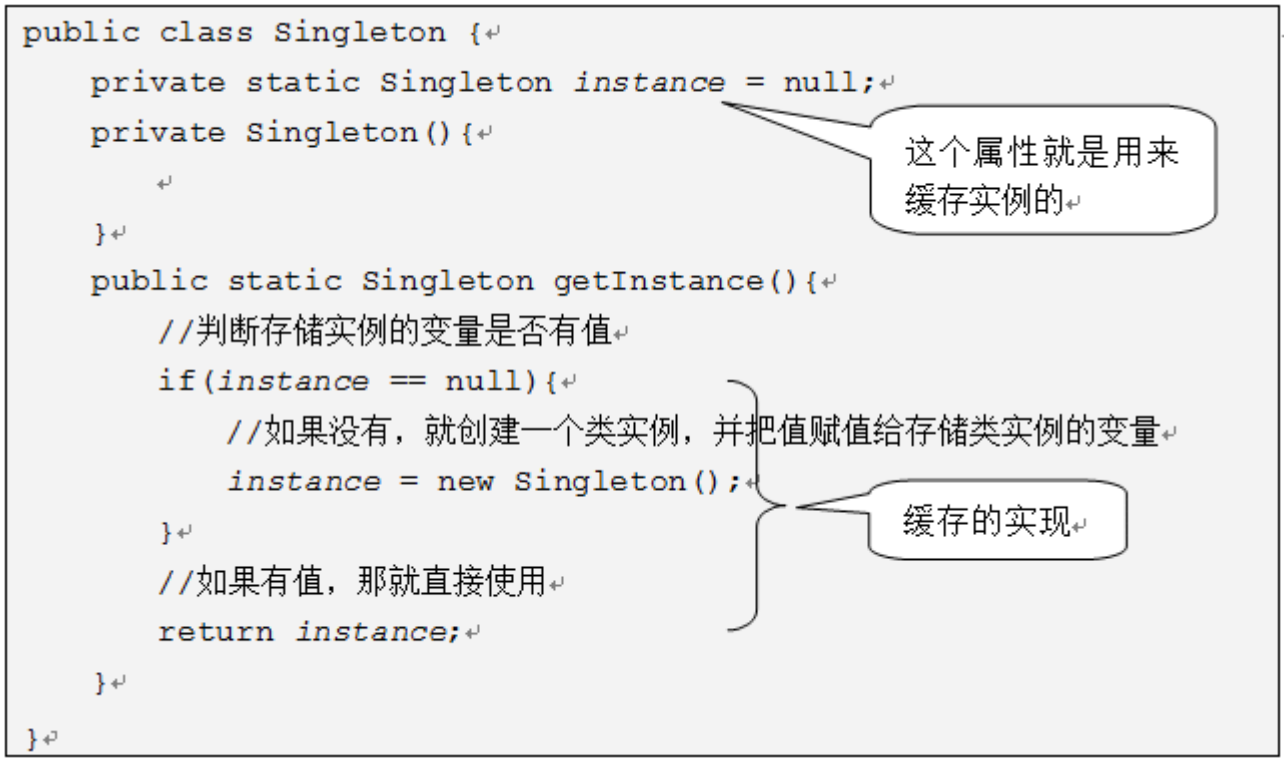
3.4 缓存的思想

单例模式的懒汉式实现还体现了缓存的思想，缓存也是实际开发中非常常见的功能。

简单讲就是，如果某些资源或者数据会被频繁的使用，而这些资源或数据存储在系统外部，比如数据库、硬盘文件等，那么每次操作这些数据的时候都从数据库或者硬盘上去获取，速度会很慢，会造成性能问题。

一个简单的解决方法就是：把这些数据缓存到内存里面，每次操作的时候，先到内存里面找，看有没有这些数据，如果有，那么就直接使用，如果没有那么就获取它，并设置到缓存中，下一次访问的时候就可以直接从内存中获取了。从而节省大量的时间，当然，缓存是一种典型的空间换时间的方案。

缓存在单例模式的实现中怎么体现的呢？




```
public class Singleton {
    private static Singleton instance = null;
    private Singleton(){

    }
    public static Singleton getInstance(){
        //判断存储实例的变量是否有值
        if(instance == null){
            //如果没有，就创建一个类实例，并把值赋值给存储类实例的变量
            instance = new Singleton();
        }
        //如果有值，那就直接使用
        return instance;
    }
}
```

3.5 Java中缓存的基本实现

引申一下，看看在Java开发中的缓存的基本实现，在Java中最常见的一种实现缓存的方式就是使用Map，基本的步骤是：

- 先到缓存里面查找，看看是否存在需要使用的数据
- 如果没有找到，那么就创建一个满足要求的数据，然后把这个数据设置回到缓存中，以备下次使用
- 如果找到了相应的数据，或者是创建了相应的数据，那就直接使用这个数据。

还是看看示例吧，示例代码如下：

```
/**
 * Java中缓存的基本实现示例
 */
public class JavaCache {
        /**
         * 缓存数据的容器，定义成Map是方便访问，直接根据Key就可以获取Value了
         * key选用String是为了简单，方便演示
         */
        private Map<String,Object> map = new HashMap<String,Object>();
        /**
         * 从缓存中获取值
```

```
         * @param key 设置时候的key值
         * @return key对应的Value值
         */
        public Object getValue(String key){
                //先从缓存里面取值
                Object obj = map.get(key);
                //判断缓存里面是否有值
                if(obj == null){
                        //如果没有，那么就去获取相应的数据，比如读取数据库或者文件
                        //这里只是演示，所以直接写个假的值
                        obj = key+",value";
                        //把获取的值设置回到缓存里面
                        map.put(key, obj);
                }
                //如果有值了，就直接返回使用
                return obj;
        }
}
```

这里只是缓存的基本实现，还有很多功能都没有考虑，比如缓存的清除，缓存的同步等等。当然，Java的缓存还有很多实现方式，也是非常复杂的，现在有很多专业的缓存框架，更多缓存的知识，这里就不再去讨论了。

3.6 利用缓存来实现单例模式

其实应用Java缓存的知识，也可以变相实现Singleton模式，算是一个模拟实现吧。每次都先从缓存中取值，只要创建一次对象实例过后，就设置了缓存的值，那么下次就不用再创建了。

虽然不是很标准的做法，但是同样可以实现单例模式的功能，为了简单，先不去考虑多线程的问题，示例代码如下：

```
/**
 * 使用缓存来模拟实现单例
 */
public class Singleton {
```

```
        /**
         * 定义一个缺省的key值，用来标识在缓存中的存放
         */
        private final static String DEFAULT_KEY = "One";
        /**
         * 缓存实例的容器
         */
        private static Map<String,Singleton> map =
new HashMap<String,Singleton>();
        /**
         * 私有化构造方法
         */
        private Singleton(){
                //
        }
        public static Singleton getInstance(){
                //先从缓存中获取
                Singleton instance = (Singleton)map.get(DEFAULT_KEY);
                //如果没有，就新建一个，然后设置回缓存中
                if(instance==null){
                        instance = new Singleton();
                        map.put(DEFAULT_KEY, instance);
                }
                //如果有就直接使用
                return instance;
        }
}
```

是不是也能实现单例所要求的功能呢？其实实现模式的方式有很多种，并不是只有模式的参考实现所实现的方式，上面这种也能实现单例所要求的功能，只不过实现比较麻烦，不是太好而已，但在后面扩展单例模式的时候会有用。

另外，模式是经验的积累，模式的参考实现并不一定是最优的，对于单例模式，后面会给大家一些更好的实现方式。

3.7 单例模式的优缺点

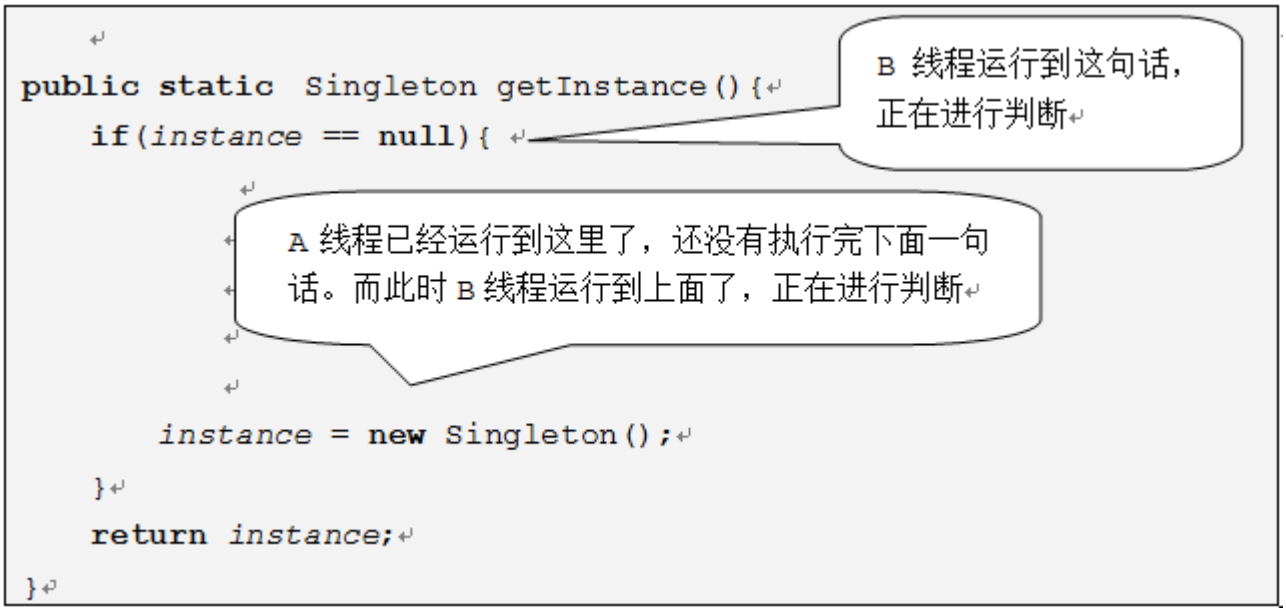
1：时间和空间

比较上面两种写法：**懒汉式是典型的时间换空间**，也就是每次获取实例都会进行判断，看是否需要创建实例，费判断的时间，当然，如果一直没有人使用的话，那就不会创建实例，节约内存空间。

饿汉式是典型的空间换时间，当类装载的时候就会创建类实例，不管你用不用，先创建出来，然后每次调用的时候，就不需要再判断了，节省了运行时间。

2：线程安全

（1）从线程安全性上讲，**不加同步的懒汉式是线程不安全的**，比如说：有两个线程，一个是线程A，一个是线程B，它们同时调用getInstance方法，那就可能导致并发问题。如下示例：



程序继续运行，两个线程都向前走了一步，如下：

```
public static  Singleton getInstance(){
    if(instance == null){
        1：由于 B 线程运行较快，一下就判断 instance==null 为 true
        2：而此时 A 线程正在创建实例，也就是正运行 new Singleton()
        3：但是 B 线程已经判断完了，也进入到这里了
        问题产生了：这样就没有控制住，并发了，会创建两个实例
        instance = new Singleton();
        A 线程正在创建实例
    }
    return instance;
}
```

可能有些朋友会觉得文字描述还是不够直观，再来画个图说明一下，如图4所示：

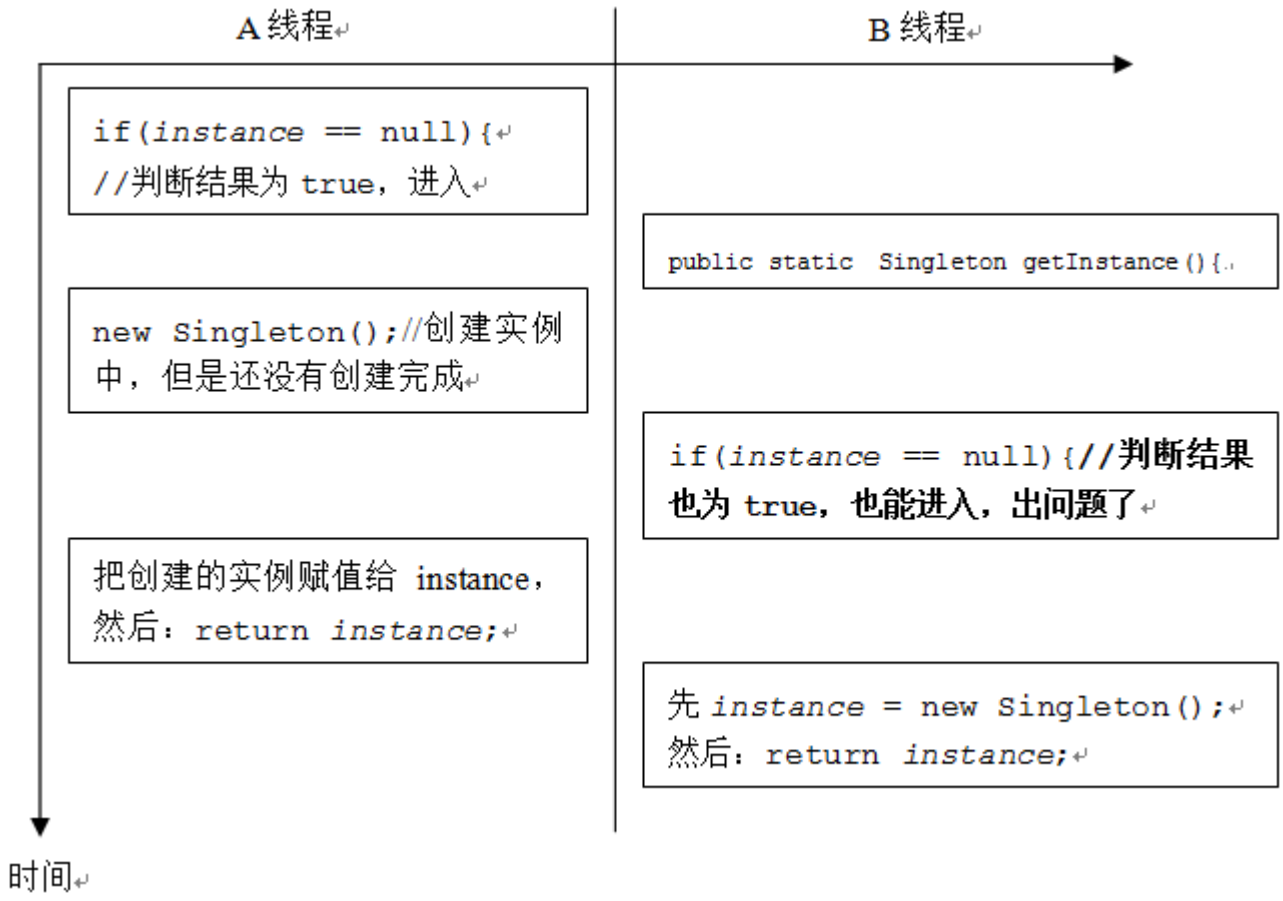


图4 懒汉式单例的线程问题示意图

通过图4的分解描述，明显可以看出，当A、B线程并发的情况下，会创建出两个实例来，也就是单例的控制在并发情况下失效了。

（2）饿汉式是线程安全的，因为虚拟机保证了只会装载一次，在装载类的时候是不会发生并发的。

（3）如何实现懒汉式的线程安全呢？

当然懒汉式也是可以实现线程安全的，只要加上synchronized即可，如下：

```
public static synchronized Singleton getInstance(){}
```

但是这样一来，会降低整个访问的速度，而且每次都要判断，也确实是稍微慢点。那么有没有更好的方式来实现呢？

（4）双重检查加锁

可以使用“双重检查加锁”的方式来实现，就可以既实现线程安全，又能够使性能不受到大的影响。那么什么是“双重检查加锁”机制呢？

所谓双重检查加锁机制，指的是：并不是每次进入getInstance方法都需要同步，而是先不同步，进入方法过后，先检查实例是否存在，如果不存在才进入下面的同步块，这是第一重检查。进入同步块过后，再次检查实例是否存在，如果不存在，就在同步的情况下创建一个实例，这是第二重检查。这样一来，就只需要同步一次了，从而减少了多次在同步情况下进行判断所浪费的时间。

双重检查加锁机制的实现会使用一个关键字volatile，它的意思是：被volatile修饰的变量的值，将不会被本地线程缓存，所有对该变量的读写都是直接操作共享内存,从而确保多个线程能正确的处理该变量。

注意:在Java1.4及以前版本中，很多JVM对于volatile关键字的实现有问题，会导致双重检查加锁的失败，因此双重检查加锁的机制只能用在Java5及以上的版本。

看看代码可能会更清楚些，示例代码如下：

```
public class Singleton {
        /**
         * 对保存实例的变量添加volatile的修饰
         */
        private volatile static Singleton instance = null;
        private Singleton(){
        }
        public static  Singleton getInstance(){
                //先检查实例是否存在，如果不存在才进入下面的同步块
                if(instance == null){
                        //同步块，线程安全的创建实例
```

```
                        synchronized(Singleton.class){
                                //再次检查实例是否存在，如果不存在才真的创建实例
                                if(instance == null){
                                        instance = new Singleton();
                                }
                        }
                }
                return instance;
        }
}
```

这种实现方式既可使实现线程安全的创建实例，又不会对性能造成太大的影响，它只是在第一次创建实例的时候同步，以后就不需要同步了，从而加快运行速度。

提示：由于volatile关键字可能会屏蔽掉虚拟机中一些必要的代码优化，所以运行效率并不是很高，因此一般建议，没有特别的需要，不要使用。也就是说，虽然可以使用双重加锁机制来实现线程安全的单例，但并不建议大量采用，根据情况来选用吧。

未完待续

1.9 研磨设计模式之单例模式-4

发表时间: 2010-08-09

3.8 在Java中一种更好的单例实现方式

根据上面的分析，常见的两种单例实现方式都存在小小的缺陷，那么有没有一种方案，既能够实现延迟加载，又能够实现线程安全呢？

还真有高人想到这样的解决方案了，这个解决方案被称为Lazy initialization holder class模式，这个模式综合使用了Java的类级内部类和多线程缺省同步锁的知识，很巧妙的同时实现了延迟加载和线程安全。

1：先来看点相应的基础知识

先简单的看看类级内部类相关的知识。

- 什么是类级内部类？

  简单点说，类级内部类指的是：有static修饰的成员式内部类。如果没有static修饰的成员式内部类被称为对象级内部类。
- 类级内部类相当于其外部类的static成分，它的对象与外部类对象间不存在依赖关系，因此可直接创建。而对象级内部类的实例，是绑定在外部对象实例中的。
- 类级内部类中，可以定义静态的方法，在静态方法中只能够引用外部类中的静态成员方法或者成员变量。
- 类级内部类相当于其外部类的成员，只有在第一次被使用的时候才会被装载

再来看看多线程缺省同步锁的知识。

大家都知道，在多线程开发中，为了解决并发问题，主要是通过使用synchronized来加互斥锁进行同步控制。但是在某些情况中，JVM已经隐含地为您执行了同步，这些情况下就不用自己再来进行同步控制了。这些情况包括：

- 由静态初始化器（在静态字段上或 static{} 块中的初始化器）初始化数据时
- 访问 final 字段时
- 在创建线程之前创建对象时
- 线程可以看见它将要处理的对象时

2：接下来看看这种解决方案的思路

要想很简单的实现线程安全，可以采用静态初始化器的方式，它可以由JVM来保证线程安全性。比如前面的“饿汉式”实现方式，但是这样一来，不是会浪费一定的空间吗？因为这种实现方式，会在类装载的时候就初始化对象，不管你需不需要。

如果现在有一种方法能够让类装载的时候不去初始化对象，那不就解决问题了？一种可行的方式就是采用类级内部类，在这个类级内部类里面去创建对象实例，这样一来，只要不使用到这个类级内部类，那就不会创

建对象实例。从而同时实现延迟加载和线程安全。

看看代码示例可能会更清晰，示例代码如下：

```
public class Singleton {
        /**
         * 类级的内部类，也就是静态的成员式内部类，该内部类的实例与外部类的实例
         * 没有绑定关系，而且只有被调用到才会装载，从而实现了延迟加载
         */
        private static class SingletonHolder{
                /**
                 * 静态初始化器，由JVM来保证线程安全
                 */
                private static Singleton instance = new Singleton();
        }
        /**
         * 私有化构造方法
         */
        private Singleton(){
        }
        public static  Singleton getInstance(){
                return SingletonHolder.instance;
        }
}
```

仔细想想，是不是很巧妙呢！

当getInstance方法第一次被调用的时候，它第一次读取SingletonHolder.instance，导致SingletonHolder类得到初始化；而这个类在装载并被初始化的时候，会初始化它的静态域，从而创建Singleton的实例，由于是静态的域，因此只会被虚拟机在装载类的时候初始化一次，并由虚拟机来保证它的线程安全性。

这个模式的优势在于，getInstance方法并没有被同步，并且只是执行一个域的访问，因此延迟初始化并没有增加任何访问成本。

3.9 单例和枚举

按照《高效Java 第二版》中的说法：单元素的枚举类型已经成为实现Singleton的最佳方法。

为了理解这个观点，先来了解一点相关的枚举知识，这里只是强化和总结一下枚举的一些重要观点，更多基本的枚举的使用，请参看Java编程入门资料：

- Java的枚举类型实质上是功能齐全的类，因此可以有自己的属性和方法
- Java枚举类型的基本思想：通过公有的静态final域为每个枚举常量导出实例的类
- 从某个角度讲，枚举是单例的泛型化，本质上是单元素的枚举

用枚举来实现单例非常简单，只需要编写一个包含单个元素的枚举类型即可，示例代码如下：

```
/**
 * 使用枚举来实现单例模式的示例
 */
public enum Singleton {
        /**
         * 定义一个枚举的元素,它就代表了Singleton的一个实例
         */
        uniqueInstance;

        /**
         * 示意方法，单例可以有自己的操作
         */
        public void singletonOperation(){
                //功能处理
        }
}
```

使用枚举来实现单实例控制，会更加简洁，而且无偿的提供了序列化的机制，并由JVM从根本上提供保障，绝对防止多次实例化，是更简洁、高效、安全的实现单例的方式。

3.10 思考单例模式

1：单例模式的本质

单例模式的本质：**控制实例数目。**

单例模式是为了控制在运行期间，某些类的实例数目只能有一个。可能有人就会想了，那么我能不能控制实例数目为2个，3个，或者是任意多个呢？目的都是一样的，节省资源啊，有些时候单个实例不能满足实际的需要，会忙不过来，根据测算，3个实例刚刚好，也就是说，现在要控制实例数目为3个，怎么办呢？

其实思路很简单，就是利用上面通过Map来缓存实现单例的示例，进行变形，一个Map可以缓存任意多个实例，新的问题就是，Map中有多个实例，但是客户端调用的时候，到底返回那一个实例呢，也就是实例的调

度问题，我们只是想要来展示设计模式，对于这个调度算法就不去深究了，做个最简单的，循环返回就好了，示例代码如下：

```
/**
* 简单演示如何扩展单例模式，控制实例数目为3个
*/
public class OneExtend {
        /**
        * 定义一个缺省的key值的前缀
        */
        private final static String DEFAULT_PREKEY = "Cache";
        /**
        * 缓存实例的容器
        */
        private static Map<String,OneExtend> map =
new HashMap<String,OneExtend>();
        /**
        * 用来记录当前正在使用第几个实例，到了控制的最大数目，就返回从1开始
        */
        private static int num = 1;
        /**
        * 定义控制实例的最大数目
        */
        private final static int NUM_MAX = 3;
        private OneExtend(){}
        public static OneExtend getInstance(){
                String key = DEFAULT_PREKEY+num;
                //缓存的体现，通过控制缓存的数据多少来控制实例数目
                OneExtend oneExtend = map.get(key);
                if(oneExtend==null){
                        oneExtend = new OneExtend();
                        map.put(key, oneExtend);
                }
                //把当前实例的序号加1
                num++;
                if(num > NUM_MAX){
                        //如果实例的序号已经达到最大数目了，那就重复从1开始获取
```

```
                        num = 1;
                }
                return oneExtend;
        }

        public static void main(String[] args) {
                //测试是否能满足功能要求
                OneExtend t1 = getInstance ();
                OneExtend t2 = getInstance ();
                OneExtend t3 = getInstance ();
                OneExtend t4 = getInstance ();
                OneExtend t5 = getInstance ();
                OneExtend t6 = getInstance ();

                System.out.println("t1=="+t1);
                System.out.println("t2=="+t2);
                System.out.println("t3=="+t3);
                System.out.println("t4=="+t4);
                System.out.println("t5=="+t5);
                System.out.println("t6=="+t6);
        }
}
```

测试一下，看看结果，如下：

```
t1==cn.javass.dp.singleton.example9.OneExtend@6b97fd
t2==cn.javass.dp.singleton.example9.OneExtend@1c78e57
t3==cn.javass.dp.singleton.example9.OneExtend@5224ee
t4==cn.javass.dp.singleton.example9.OneExtend@6b97fd
t5==cn.javass.dp.singleton.example9.OneExtend@1c78e57
t6==cn.javass.dp.singleton.example9.OneExtend@5224ee
```

第一个实例和第四个相同，第二个与第五个相同，第三个与第六个相同，也就是说一共只有三个实例，而且调度算法是从第一个依次取到第三个，然后回来继续从第一个开始取到第三个。

当然这里我们不去考虑复杂的调度情况，也不去考虑何时应该创建新实例的问题。

注意：这种实现方式同样是线程不安全的，需要处理，这里就不再展开去讲了。

2：何时选用单例模式

建议在如下情况中，选用单例模式：

- 当需要控制一个类的实例只能有一个，而且客户只能从一个全局访问点访问它时，可以选用单例模式，这些功能恰好是单例模式要解决的问题

3.11 相关模式

很多模式都可以使用单例模式，只要这些模式中的某个类，需要控制实例为一个的时候，就可以很自然的使用上单例模式。比如抽象工厂方法中的具体工厂类就通常是一个单例。

单例模式结束,谢谢各位的捧场,鞠躬ing

1.10 研磨设计模式之策略模式-1

发表时间: 2010-06-22

首先感谢众多朋友的支持、评论和鼓励，只有多多努力，写点好的博文来回报大家的好意！

接下来想写写另外一个虽然较简单，但是使用很频繁的模式——策略模式

策略模式(Strategy)

1 场景问题

1.1 报价管理

向客户报价，对于销售部门的人来讲，这是一个非常重大、非常复杂的问题，对不同的客户要报不同的价格，比如：

- 对普通客户或者是新客户报的是全价
- 对老客户报的价格，根据客户年限，给予一定的折扣
- 对大客户报的价格，根据大客户的累计消费金额，给予一定的折扣
- 还要考虑客户购买的数量和金额，比如：虽然是新用户，但是一次购买的数量非常大，或者是总金额非常高，也会有一定的折扣
- 还有，报价人员的职务高低，也决定了他是否有权限对价格进行一定的浮动折扣

甚至在不同的阶段，对客户的报价也不同，一般情况是刚开始比较高，越接近成交阶段，报价越趋于合理。

总之，向客户报价是非常复杂的，因此在一些CRM（客户关系管理）的系统中，会有一个单独的报价管理模块，来处理复杂的报价功能。

为了演示的简洁性，假定现在需要实现一个简化的报价管理，实现如下的功能：

（1）对普通客户或者是新客户报全价

（2）对老客户报的价格，统一折扣5%

（3）对大客户报的价格，统一折扣10%

该怎么实现呢？

1.2 不用模式的解决方案

要实现对不同的人员报不同的价格的功能，无外乎就是判断起来麻烦点，也不多难，很快就有朋友能写出如下的实现代码，示例代码如下：

```
/**
 * 价格管理，主要完成计算向客户所报价格的功能
 */
public class Price {
    /**
     * 报价，对不同类型的，计算不同的价格
     * @param goodsPrice 商品销售原价
     * @param customerType 客户类型
     * @return 计算出来的，应该给客户报的价格
     */
    public double quote(double goodsPrice,String customerType){
        if(customerType.equals("普通客户")){
            System.out.println("对于新客户或者是普通客户，没有折扣");
            return goodsPrice;
        }else if(customerType.equals("老客户")){
            System.out.println("对于老客户，统一折扣5%");
            return goodsPrice*(1-0.05);
        }else if(customerType.equals("大客户")){
            System.out.println("对于大客户，统一折扣10%");
            return goodsPrice*(1-0.1);
        }
        //其余人员都是报原价
        return goodsPrice;
    }
}
```

1.3 有何问题

上面的写法是很简单的，也很容易想，但是仔细想想，这样实现，问题可不小，比如：

- 第一个问题：价格类包含了所有计算报价的算法，使得价格类，尤其是报价这个方法比较庞杂，难以维护。

有朋友可能会想，这很简单嘛，把这些算法从报价方法里面拿出去，形成独立的方法不就可以解决这个问题了吗？据此写出如下的实现代码，示例代码如下：

```
/**
 * 价格管理，主要完成计算向客户所报价格的功能
 */
public class Price {
    /**
     * 报价，对不同类型的，计算不同的价格
     * @param goodsPrice 商品销售原价
     * @param customerType 客户类型
     * @return 计算出来的，应该给客户报的价格
     */
    public double quote(double goodsPrice,String customerType){
      if(customerType.equals("普通客户")){
          return this.calcPriceForNormal(goodsPrice);
      }else if(customerType.equals("老客户")){
          return this.calcPriceForOld(goodsPrice);
      }else if(customerType.equals("大客户")){
          return this.calcPriceForLarge(goodsPrice);
      }
      //其余人员都是报原价
      return goodsPrice;
    }
    /**
     * 为新客户或者是普通客户计算应报的价格
     * @param goodsPrice 商品销售原价
     * @return 计算出来的，应该给客户报的价格
     */
    private double calcPriceForNormal(double goodsPrice){
      System.out.println("对于新客户或者是普通客户，没有折扣");
      return goodsPrice;
    }
    /**
     * 为老客户计算应报的价格
     * @param goodsPrice 商品销售原价
     * @return 计算出来的，应该给客户报的价格
     */
    private double calcPriceForOld(double goodsPrice){
      System.out.println("对于老客户，统一折扣5%");
      return goodsPrice*(1-0.05);
    }
    /**
     * 为大客户计算应报的价格
     * @param goodsPrice 商品销售原价
     * @return 计算出来的，应该给客户报的价格
     */
    private double calcPriceForLarge(double goodsPrice){
      System.out.println("对于大客户，统一折扣10%");
      return goodsPrice*(1-0.1);
    }
}
```

这样看起来，比刚开始稍稍好点，计算报价的方法会稍稍简单一点，这样维护起来也稍好一些，某个算法发生了变化，直接修改相应的私有方法就可以了。扩展起来也容易一点，比如要增加一个“战略合作客户”的类型，报价为直接8折，就只需要在价格类里面新增加一个私有的方法来计算新的价格，然后在计算报价的方法里面新添一个else-if即可。看起来似乎很不错了。

真的很不错了吗？

再想想，问题还是存在，只不过从计算报价的方法挪动到价格类里面了，假如有100个或者更多这样的计算方式，这会让这个价格类非常庞大，难以维护。而且，维护和扩展都需要去修改已有的代码，这是很不好的，违反了开-闭原则。

- 第二个问题：经常会有这样的需要，在不同的时候，要使用不同的计算方式。

比如：在公司周年庆的时候，所有的客户额外增加3%的折扣；在换季促销的时候，普通客户是额外增加折扣2%，老客户是额外增加折扣3%，大客户是额外增加折扣5%。这意味着计算报价的方式会经常被修改，或者被切换。

通常情况下应该是被切换，因为过了促销时间，又还回到正常的价格体系上来了。而现在的价格类中计算报价的方法，是固定调用各种计算方式，这使得切换调用不同的计算方式很麻烦，每次都需要修改if-else里面的调用代码。

看到这里，可能有朋友会想，**那么到底应该如何实现，才能够让价格类中的计算报价的算法，能很容易的实现可维护、可扩展，又能动态的切换变化呢？**

未完待续……

1.11 研磨设计模式之策略模式-2

发表时间: 2010-06-23

2 解决方案

2.1 策略模式来解决

用来解决上述问题的一个合理的解决方案就是策略模式。那么什么是策略模式呢？

（1）策略模式定义

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。

（2）应用策略模式来解决的思路

仔细分析上面的问题，先来把它抽象一下，各种计算报价的计算方式就好比是具体的算法，而使用这些计算方式来计算报价的程序，就相当于是使用算法的客户。

再分析上面的实现方式，为什么会造成那些问题，根本原因，就在于算法和使用算法的客户是耦合的，甚至是密不可分的，在上面实现中，具体的算法和使用算法的客户是同一个类里面的不同方法。

现在要解决那些问题，按照策略模式的方式，应该先把所有的计算方式独立出来，每个计算方式做成一个单独的算法类，从而形成一系列的算法，并且为这一系列算法定义一个公共的接口，这些算法实现是同一接口的不同实现，地位是平等的，可以相互替换。这样一来，要扩展新的算法就变成了增加一个新的算法实现类，要维护某个算法，也只是修改某个具体的算法实现即可，不会对其它代码造成影响。也就是说这样就解决了可维护、可扩展的问题。

为了实现让算法能独立于使用它的客户，策略模式引入了一个上下文的对象，这个对象负责持有算法，但是不负责决定具体选用哪个算法，把选择算法的功能交给了客户，由客户选择好具体的算法后，设置到上下文对象里面，让上下文对象持有客户选择的算法，当客户通知上下文对象执行功能的时候，上下文对象会去转调具体的算法。这样一来，具体的算法和直接使用算法的客户是分离的。

具体的算法和使用它的客户分离过后，使得算法可独立于使用它的客户而变化，并且能够动态的切换需要使用的算法，只要客户端动态的选择使用不同的算法，然后设置到上下文对象中去，实际调用的时候，就可以调用到不同的算法。

2.2 模式结构和说明

策略模式的结构示意图如图1所示：

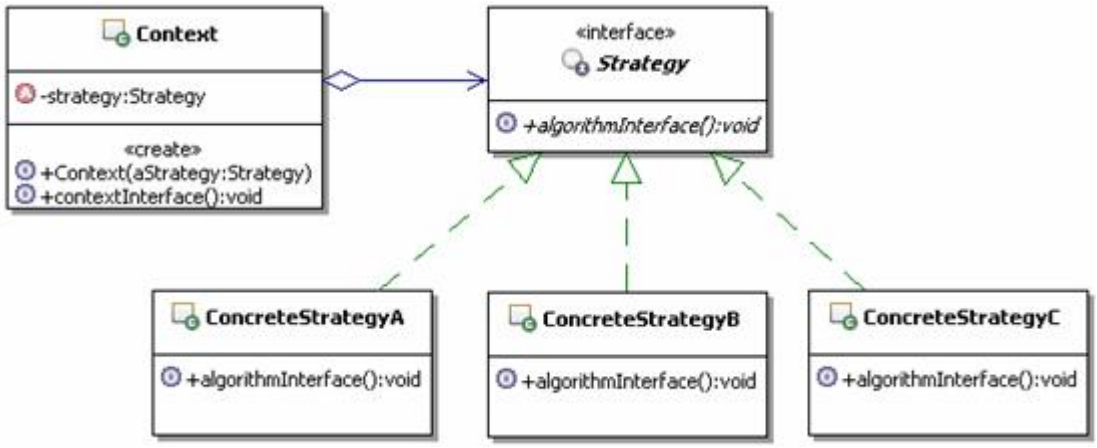


图1 策略模式结构示意图

Strategy：

策略接口，用来约束一系列具体的策略算法。Context使用这个接口来调用具体的策略实现定义的算法。

ConcreteStrategy：

具体的策略实现，也就是具体的算法实现。

Context：

上下文，负责和具体的策略类交互，通常上下文会持有一个真正的策略实现，上下文还可以让具体的策略类来获取上下文的数据，甚至让具体的策略类来回调上下文的方法。

2.3 策略模式示例代码

（1）首先来看策略，也就是定义算法的接口，示例代码如下：

```
/**
 * 策略，定义算法的接口
 */
public interface Strategy {
    /**
     * 某个算法的接口，可以有传入参数，也可以有返回值
     */
    public void algorithmInterface();
}
```

（2）该来看看具体的算法实现了，定义了三个，分别是ConcreteStrategyA、ConcreteStrategyB、ConcreteStrategyC，示例非常简单，由于没有具体算法的实现，三者也就是名称不同，示例代码如下：

```
/**
 * 实现具体的算法
 */
public class ConcreteStrategyA implements Strategy {
    public void algorithmInterface() {
      //具体的算法实现
    }
}
/**
 * 实现具体的算法
 */
public class ConcreteStrategyB implements Strategy {
    public void algorithmInterface() {
      //具体的算法实现
    }
}
/**
 * 实现具体的算法
 */
public class ConcreteStrategyC implements Strategy {
    public void algorithmInterface() {
      //具体的算法实现
    }
}
```

（3）再来看看上下文的实现，示例代码如下：

```
/**
 * 上下文对象，通常会持有一个具体的策略对象
 */
public class Context {
    /**
     * 持有一个具体的策略对象
     */
    private Strategy strategy;
    /**
     * 构造方法，传入一个具体的策略对象
     * @param aStrategy 具体的策略对象
     */
```

```
    public Context(Strategy aStrategy) {
      this.strategy = aStrategy;
    }
    /**
     * 上下文对客户端提供的操作接口，可以有参数和返回值
     */
    public void contextInterface() {
      //通常会转调具体的策略对象进行算法运算
      strategy.algorithmInterface();
    }
}
```

2.4 使用策略模式重写示例

要使用策略模式来重写前面报价的示例，大致有如下改变：

- 首先需要定义出算法的接口。
- 然后把各种报价的计算方式单独出来，形成算法类。
- 对于Price这个类，把它当做上下文，在计算报价的时候，不再需要判断，直接使用持有的具体算法进行运算即可。选择使用哪一个算法的功能挪出去，放到外部使用的客户端去。

这个时候，程序的结构如图2所示：

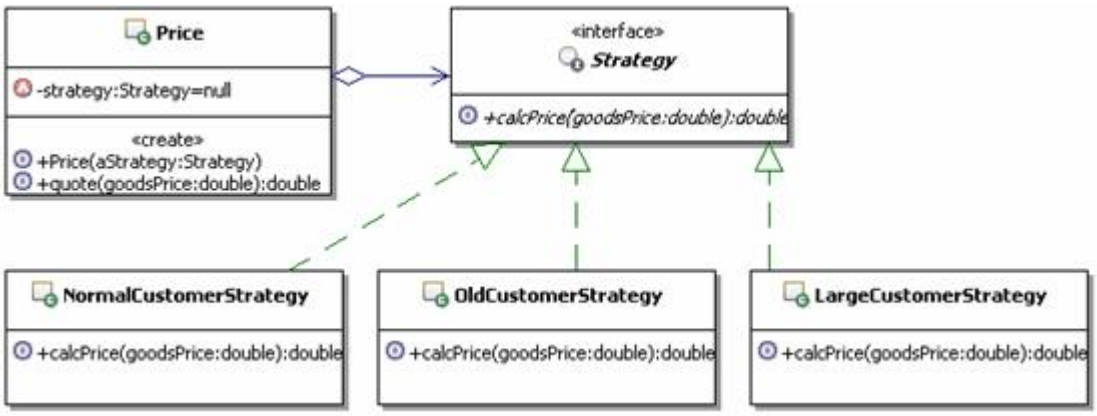


图2 使用策略模式实现示例的结构示意图

（1）先看策略接口，示例代码如下：

```
/**
 * 策略，定义计算报价算法的接口
 */
public interface Strategy {
    /**
```

```
     * 计算应报的价格
     * @param goodsPrice 商品销售原价
     * @return 计算出来的，应该给客户报的价格
     */
    public double calcPrice(double goodsPrice);
}
```

（2）接下来看看具体的算法实现，不同的算法，实现也不一样，先看为新客户或者是普通客户计算应报的价格的实现，示例代码如下：

```
/**
 * 具体算法实现，为新客户或者是普通客户计算应报的价格
 */
public class NormalCustomerStrategy implements Strategy{
    public double calcPrice(double goodsPrice) {
        System.out.println("对于新客户或者是普通客户，没有折扣");
        return goodsPrice;
    }
}
```

再看看为老客户计算应报的价格的实现，示例代码如下：

```
/**
 * 具体算法实现，为老客户计算应报的价格
 */
public class OldCustomerStrategy implements Strategy{
    public double calcPrice(double goodsPrice) {
        System.out.println("对于老客户，统一折扣5%");
        return goodsPrice*(1-0.05);
    }
}
```

再看看为大客户计算应报的价格的实现，示例代码如下：

```
/**
 * 具体算法实现，为大客户计算应报的价格
```

```
 */
public class LargeCustomerStrategy implements Strategy{
    public double calcPrice(double goodsPrice) {
      System.out.println("对于大客户，统一折扣10%");
      return goodsPrice*(1-0.1);
    }
}
```

（3）接下来看看上下文的实现，也就是原来的价格类，它的变化比较大，主要有：

- 原来那些私有的，用来做不同计算的方法，已经去掉了，独立出去做成了算法类
- 原来报价方法里面，对具体计算方式的判断，去掉了，让客户端来完成选择具体算法的功能
- 新添加持有一个具体的算法实现，通过构造方法传入
- 原来报价方法的实现，变化成了转调具体算法来实现

示例代码如下：

```
/**
 * 价格管理，主要完成计算向客户所报价格的功能
 */
public class Price {
    /**
     * 持有一个具体的策略对象
     */
    private Strategy strategy = null;
    /**
     * 构造方法，传入一个具体的策略对象
     * @param aStrategy 具体的策略对象
     */
    public Price(Strategy aStrategy){
      this.strategy = aStrategy;
    }
    /**
     * 报价，计算对客户的报价
     * @param goodsPrice 商品销售原价
     * @return 计算出来的，应该给客户报的价格
     */
    public double quote(double goodsPrice){
      return this.strategy.calcPrice(goodsPrice);
    }
}
```

（4）写个客户端来测试运行一下，好加深体会，示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        //1：选择并创建需要使用的策略对象
        Strategy strategy = new LargeCustomerStrategy ();
        //2：创建上下文
        Price ctx = new Price(strategy);

        //3：计算报价
        double quote = ctx.quote(1000);
        System.out.println("向客户报价："+quote);
    }
}
```

运行一下，看看效果。

你可以修改使用不同的策略算法具体实现，现在用的是LargeCustomerStrategy，你可以尝试修改成其它两种实现，试试看，体会一下切换算法的容易性。

未完待续.....

1.12 研磨设计模式之策略模式-3

发表时间: 2010-06-24

3 模式讲解

3.1 认识策略模式

（1）策略模式的功能

策略模式的功能是把具体的算法实现，从具体的业务处理里面独立出来，把它们实现成为单独的算法类，从而形成一系列的算法，并让这些算法可以相互替换。

策略模式的重心不是如何来实现算法，而是如何组织、调用这些算法，从而让程序结构更灵活、具有更好的维护性和扩展性。

（2）策略模式和if-else语句

看了前面的示例，很多朋友会发现，每个策略算法具体实现的功能，就是原来在if-else结构中的具体实现。

没错，其实多个if-elseif语句表达的就是一个平等的功能结构，你要么执行if，要不你就执行else，或者是elseif，这个时候，if块里面的实现和else块里面的实现从运行地位上来讲就是平等的。

而策略模式就是把各个平等的具体实现封装到单独的策略实现类了，然后通过上下文来与具体的策略类进行交互。

因此多个if-else语句可以考虑使用策略模式。

（3）算法的平等性

策略模式一个很大的特点就是各个策略算法的平等性。对于一系列具体的策略算法，大家的地位是完全一样的，正是因为这个平等性，才能实现算法之间可以相互替换。

所有的策略算法在实现上也是相互独立的，相互之间是没有依赖的。

所以可以这样描述这一系列策略算法：**策略算法是相同行为的不同实现。**

（4）谁来选择具体的策略算法

在策略模式中，可以在两个地方来进行具体策略的选择。

一个是在客户端，在使用上下文的时候，由客户端来选择具体的策略算法，然后把这个策略算法设置给上下文。前面的示例就是这种情况。

还有一个是客户端不管，由上下文来选择具体的策略算法，这个在后面讲容错恢复的时候给大家演示一下。

（5）Strategy的实现方式

在前面的示例中，Strategy都是使用的接口来定义的，这也是常见的实现方式。但是如果多个算法具有公共功能的话，可以把Strategy实现成为抽象类，然后把多个算法的公共功能实现到Strategy里面。

（6）运行时策略的唯一性

运行期间，策略模式在每一个时刻只能使用一个具体的策略实现对象，虽然可以动态的在不同的策略实现中切换，但是同时只能使用一个。

（7）增加新的策略

在前面的示例里面，体会到了策略模式中切换算法的方便，但是增加一个新的算法会怎样呢？比如现在要实现如下的功能：对于公司的“战略合作客户”，统一8折。

其实很简单，策略模式可以让你很灵活的扩展新的算法。具体的做法是：先写一个策略算法类来实现新的要求，然后在客户端使用的时候指定使用新的策略算法类就可以了。

还是通过示例来说明。先添加一个实现要求的策略类，示例代码如下：

```
/**
 * 具体算法实现，为战略合作客户客户计算应报的价格
 */
public class CooperateCustomerStrategy implements Strategy{
    public double calcPrice(double goodsPrice) {
        System.out.println("对于战略合作客户，统一8折");
        return goodsPrice*0.8;
    }
}
```

然后在客户端指定使用策略的时候指定新的策略算法实现，示例如下：

```
public class Client2 {
    public static void main(String[] args) {
        //1：选择并创建需要使用的策略对象
        Strategy strategy = new CooperateCustomerStrategy ();
        //2：创建上下文
            Price ctx = new Price(strategy);

        //3：计算报价
        double quote = ctx.quote(1000);
        System.out.println("向客户报价："+quote);
    }
}
```

除了加粗部分变动外，客户端没有其他的变化。

运行客户端，测试看看，好好体会一下。

除了客户端发生变化外，已有的上下文、策略接口定义和策略的已有实现，都不需要做任何的修改，可见能很方便的扩展新的策略算法。

（8）策略模式调用顺序示意图

策略模式的调用顺序，有两种常见的情况，一种如同前面的示例，具体如下：

- 先是客户端来选择并创建具体的策略对象
- 然后客户端创建上下文
- 接下来客户端就可以调用上下文的方法来执行功能了，在调用的时候，从客户端传入算法需要的参数
- 上下文接到客户的调用请求，会把这个请求转发给它持有的Strategy

这种情况的调用顺序示意图如图3所示：

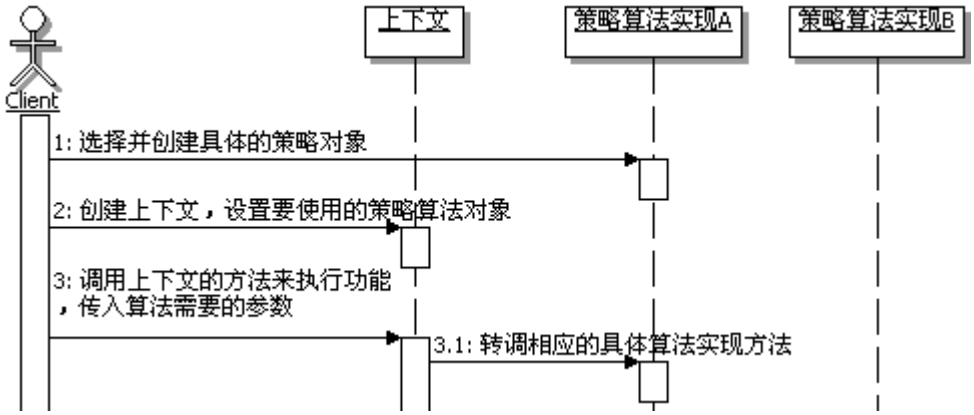


图3 策略模式调用顺序示意图一

策略模式调用还有一种情况，就是把Context当做参数来传递给Strategy，这种方式的调用顺序图，在讲具体的Context和Strategy的关系时再给出。

3.2 容错恢复机制

容错恢复机制是应用程序开发中非常常见的功能。那么什么是容错恢复呢？简单点说就是：程序运行的时候，正常情况下应该按照某种方式来做，如果按照某种方式来做发生错误的话，系统并不会崩溃，也不会就此不能继续向下运行了，而是有容忍出错的能力，不但能容忍程序运行出现错误，还提供出现错误后的备用方案，也就是恢复机制，来代替正常执行的功能，使程序继续向下运行。

举个实际点的例子吧，比如在一个系统中，所有对系统的操作都要有日志记录，而且这个日志还需要有管理界面，这种情况下通常会把日志记录在数据库里面，方便后续的管理，但是在记录日志到数据库的时候，可能会发生错误，比如暂时连不上数据库了，那就先记录在文件里面，然后在合适的时候把文件中的记录再转录到数据库中。

对于这样的功能的设计，就可以采用策略模式，把日志记录到数据库和日志记录到文件当作两种记录日志的策略，然后在运行期间根据需要进行动态的切换。

在这个例子的实现中，要示范由上下文来选择具体的策略算法，前面的例子都是由客户端选择好具体的算法，然后设置到上下文中。

下面还是通过代码来示例一下。

（1）先定义日志策略接口，很简单，就是一个记录日志的方法，示例代码如下：

```
/**
 * 日志记录策略的接口
 */
public interface LogStrategy {
    /**
     * 记录日志
     * @param msg 需记录的日志信息
     */
    public void log(String msg);
}
```

（2）实现日志策略接口，先实现默认的数据库实现，假设如果日志的长度超过长度就出错，制造错误的是一个最常见的运行期错误，示例代码如下：

```
/**
 * 把日志记录到数据库
 */
public class DbLog implements LogStrategy{
    public void log(String msg) {
        //制造错误
        if(msg!=null && msg.trim().length()>5){
            int a = 5/0;
        }
        System.out.println("现在把 '"+msg+"' 记录到数据库中");
```

```
    }
}
```

接下来实现记录日志到文件中去，示例代码如下：

```
/**
 * 把日志记录到文件
 */
public class FileLog implements LogStrategy{
    public void log(String msg) {
        System.out.println("现在把 '"+msg+"' 记录到文件中");
    }
}
```

（3）接下来定义使用这些策略的上下文，注意这次是在上下文里面实现具体策略算法的选择，所以不需要客户端来指定具体的策略算法了，示例代码如下：

```
/**
 * 日志记录的上下文
 */
public class LogContext {
    /**
     * 记录日志的方法，提供给客户端使用
     * @param msg 需记录的日志信息
     */
    public void log(String msg){
        //在上下文里面，自行实现对具体策略的选择
        //优先选用策略：记录到数据库
        LogStrategy strategy = new DbLog();
        try{
            strategy.log(msg);
        }catch(Exception err){
            //出错了，那就记录到文件中
            strategy = new FileLog();
            strategy.log(msg);
        }
    }
}
```

在这里进行具体策略算法的选择，把 try-catch 变相当成了 if-else 来用

（4）看看现在的客户端，没有了选择具体实现策略算法的工作，变得非常简单，故意多调用一次，可以看出不同的效果，示例代码如下：

```
public class Client {
    public static void main(String[] args) {
        LogContext log = new LogContext();
        log.log("记录日志");
        log.log("再次记录日志");
    }
}
```

看起来这两句是没有什么区别的吧，运行一下看看结果，看看会发生什么

运行结果如下：

```
现在把 '记录日志' 记录到数据库中
现在把 '再次记录日志' 记录到文件中
```

为什么运行的结果是一个记录到了数据库中，一个记录到了文件呢？很简单，第二次调用记录日志的日志消息超长了，运行出错，容错恢复，记录日志到文件中去

（5）小结一下，通过上面的示例，会看到策略模式的一种简单应用，也顺便了解一下基本的容错恢复机制的设计和实现。在实际的应用中，需要设计容错恢复的系统一般要求都比较高，应用也会比较复杂，但是基本的思路是差不多的。

未完待续.....

1.13 研磨设计模式之策略模式-4

发表时间: 2010-06-28

3.3 Context和Strategy的关系

在策略模式中，通常是上下文使用具体的策略实现对象，反过来，策略实现对象也可以从上下文获取所需要的数据，因此可以将上下文当参数传递给策略实现对象，这种情况下上下文和策略实现对象是紧密耦合的。

在这种情况下，上下文封装着具体策略对象进行算法运算所需要的数据，具体策略对象通过回调上下文的方法来获取这些数据。

甚至在某些情况下，策略实现对象还可以回调上下文的方法来实现一定的功能，这种使用场景下，上下文变相充当了多个策略算法实现的公共接口，在上下文定义的方法可以当做是所有或者是部分策略算法使用的公共功能。

但是请注意，由于所有的策略实现对象都实现同一个策略接口，传入同一个上下文，可能会造成传入的上下文数据的浪费，因为有的算法会使用这些数据，而有的算法不会使用，但是上下文和策略对象之间交互的开销是存在的了。

还是通过例子来说明。

1：工资支付的实现思路

考虑这样一个功能：工资支付方式的问题，很多企业的工资支付方式是很灵活的，可支付方式是比较多的，比如：人民币现金支付、美元现金支付、银行转账到工资帐户、银行转账到工资卡；一些创业型的企业为了留住骨干员工，还可能有：工资转股权等等方式。总之一句话，工资支付方式很多。

随着公司的发展，会不断有新的工资支付方式出现，这就要求能方便的扩展；另外工资支付方式不是固定的，是由公司和员工协商确定的，也就是说可能不同的员工采用的是不同的支付方式，甚至同一个员工，不同时间采用的支付方式也可能会不同，这就要求能很方便的切换具体的支付方式。

要实现这样的功能，策略模式是一个很好的选择。在实现这个功能的时候，不同的策略算法需要的数据是不一样，比如：现金支付就不需要银行帐号，而银行转账就需要帐号。这就导致在设计策略接口中的方法时，不太好确定参数的个数，而且，就算现在把所有的参数都列上了，今后扩展呢？难道再来修改策略接口吗？如果这样做，那无异于一场灾难，加入一个新策略，就需要修改接口，然后修改所有已有的实现，不疯掉才怪！那么到底如何实现，在今后扩展的时候才最方便呢？

解决方案之一，就是把上下文当做参数传递给策略对象，这样一来，如果要扩展新的策略实现，只需要扩展上下文就可以了，已有的实现不需要做任何的修改。

这样是不是能很好的实现功能，并具有很好的扩展性呢？还是通过代码示例来具体的看。假设先实现人民币现金支付和美元现金支付这两种支付方式，然后就进行使用测试，然后再来添加银行转账到工资卡的支付方式，看看是不是能很容易的与已有的实现结合上。

2：实现代码示例

（1）先定义工资支付的策略接口，就是定义一个支付工资的方法，示例代码如下：

```
/**
 * 支付工资的策略的接口，公司有多种支付工资的算法
 * 比如：现金、银行卡、现金加股票、现金加期权、美元支付等等
 */
public interface PaymentStrategy {
    /**
     * 公司给某人真正支付工资
     * @param ctx 支付工资的上下文，里面包含算法需要的数据
     */
    public void pay(PaymentContext ctx);
}
```

（2）定义好了工资支付的策略接口，该来考虑如何实现这多种支付策略了。

为了演示的简单，这里先简单实现人民币现金支付和美元现金支付方式，当然并不真的去实现跟银行的交互，只是示意一下。

人民币现金支付的策略实现，示例代码如下：

```
/**
 * 人民币现金支付
 */
public class RMBCash implements PaymentStrategy{
    public void pay(PaymentContext ctx) {
        System.out.println("现在给"+ctx.getUserName()
                                    +"人民币现金支付"+ctx.getMoney()+"元");
    }
}
```

同样的实现美元现金支付的策略，示例代码如下：

```
/**
 * 美元现金支付
 */
public class DollarCash implements PaymentStrategy{
    public void pay(PaymentContext ctx) {
        System.out.println("现在给"+ctx.getUserName()
                                    +"美元现金支付"+ctx.getMoney()+"元");
    }
}
```

（3）该来看支付上下文的实现了，当然这个使用支付策略的上下文，是需要知道具体使用哪一个支付策略的，一般由客户端来确定具体使用哪一个具体的策略，然后上下文负责去真正执行。因此，这个上下文需要持有一个支付策略，而且是由客户端来配置它。示例代码如下：

```
/**
 * 支付工资的上下文，每个人的工资不同，支付方式也不同
 */
public class PaymentContext {
    /**
     * 应被支付工资的人员，简单点，用姓名来代替
     */
    private String userName = null;
    /**
     * 应被支付的工资的金额
     */
    private double money = 0.0;
    /**
     * 支付工资的方式策略的接口
     */
    private PaymentStrategy strategy = null;
    /**
     * 构造方法，传入被支付工资的人员，应支付的金额和具体的支付策略
     * @param userName 被支付工资的人员
     * @param money 应支付的金额
     * @param strategy 具体的支付策略
     */
    public PaymentContext(String userName,double money,
                                                    PaymentStrategy strategy){
      this.userName = userName;
      this.money = money;
      this.strategy = strategy;
     }
    public String getUserName() {
      return userName;
    }
    public double getMoney() {
      return money;
    }
      /**
     * 立即支付工资
     */
    public void payNow(){
      //使用客户希望的支付策略来支付工资
      this.strategy.pay(this);
    }
}
```

（4）准备好了支付工资的各种策略，下面看看如何使用这些策略来真正支付工资，很简单，客户端是使用上下文来使用具体的策略的，而且是客户端来确定具体的策略，就是客户端创建哪个策略，最终就运行哪一个策略，各个策略之间是可以动态切换的，示例代码如下：

```
public class Client {
    public static void main(String[] args) {
      //创建相应的支付策略
      PaymentStrategy strategyRMB = new RMBCash();
      PaymentStrategy strategyDollar = new DollarCash();

      //准备小李的支付工资上下文
      PaymentContext ctx1 =
                        new PaymentContext("小李",5000,strategyRMB);
      //向小李支付工资
      ctx1.payNow();

      //切换一个人，给petter支付工资
      PaymentContext ctx2 =
                        new PaymentContext("Petter",8000,strategyDollar);
      ctx2.payNow();
    }
}
```

运行一下，看看效果，运行结果如下：

```
现在给小李人民币现金支付5000.0元
现在给Petter美元现金支付8000.0元
```

请接着看策略模式-5，其实是讲的一个主题，写在一个里面超长了，只好分成了两个，请见谅！

1.14 研磨设计模式之策略模式-5

发表时间: 2010-06-28

接策略模式-4，其实是讲的一个主题，写在一个里面超长了，只好分成了两个，请见谅！

3：扩展示例，实现方式一

经过上面的测试可以看出，通过使用策略模式，已经实现好了两种支付方式了。如果现在要增加一种支付方式，要求能支付到银行卡，该怎么扩展最简单呢？

应该新增加一种支付到银行卡的策略实现，然后通过继承来扩展支付上下文，在里面添加新的支付方式需要的新的数据，比如银行卡账户，然后在客户端使用新的上下文和新的策略实现就可以了，这样已有的实现都不需要改变，完全遵循开-闭原则。

先看看扩展的支付上下文对象的实现，示例代码如下：

```
/**
 * 扩展的支付上下文对象
 */
public class PaymentContext2 extends PaymentContext {
    /**
     * 银行帐号
     */
    private String account = null;
    /**
     * 构造方法，传入被支付工资的人员，应支付的金额和具体的支付策略
     * @param userName 被支付工资的人员
     * @param money 应支付的金额
     * @param account 支付到的银行帐号
     * @param strategy 具体的支付策略
     */
    public PaymentContext2(String userName,double money,
                           String account,PaymentStrategy strategy){
      super(userName,money,strategy);
      this.account = account;
    }
    public String getAccount() {
      return account;
```

```
    }
}
```

然后看看新的策略算法的实现，示例代码如下：

```
/**
 * 支付到银行卡
 */
public class Card implements PaymentStrategy{
    public void pay(PaymentContext ctx) {
      //这个新的算法自己知道要使用扩展的支付上下文，所以强制造型一下
      PaymentContext2 ctx2 = (PaymentContext2)ctx;
      System.out.println("现在给"+ctx2.getUserName()+"的"
                  +ctx2.getAccount()+"帐号支付了"+ctx2.getMoney()+"元");
      //连接银行，进行转帐，就不去管了
    }
}
```

最后看看客户端怎么使用这个新的策略呢？原有的代码不变，直接添加新的测试就可以了，示例代码如下：

```
public class Client {
    public static void main(String[] args) {
      //创建相应的支付策略
      PaymentStrategy strategyRMB = new RMBCash();
      PaymentStrategy strategyDollar = new DollarCash();

      //准备小李的支付工资上下文
      PaymentContext ctx1 =
                  new PaymentContext("小李",5000,strategyRMB);
      //向小李支付工资
      ctx1.payNow();

      //切换一个人，给petter支付工资
      PaymentContext ctx2 =
                  new PaymentContext("Petter",8000,strategyDollar);
      ctx2.payNow();

      //测试新添加的支付方式
      PaymentStrategy strategyCard = new Card();
      PaymentContext ctx3 = new PaymentContext2(
                        "小王",9000,"010998877656",strategyCard);
```

```
        ctx3.payNow();
    }
}
```

再次测试，体会一下，运行结果如下：

```
现在给小李人民币现金支付5000.0元
现在给Petter美元现金支付8000.0元
现在给小王的010998877656帐号支付了9000.0元
```

4：扩展示例，实现方式二

同样还是实现上面这个功能：现在要增加一种支付方式，要求能支付到银行卡。

（1）上面这种实现方式，是通过扩展上下文对象来准备新的算法需要的数据。还有另外一种方式，那就是通过策略的构造方法来传入新算法需要的数据。这样实现的话，就不需要扩展上下文了，直接添加新的策略算法实现就好了。示例代码如下：

```
/**
 * 支付到银行卡
 */
public class Card2 implements PaymentStrategy{
    /**
     * 帐号信息
     */
    private String account = "";
    /**
     * 构造方法，传入帐号信息
     * @param account 帐号信息
     */
    public Card2(String account){
        this.account = account;
    }
    public void pay(PaymentContext ctx) {
        System.out.println("现在给"+ctx.getUserName()+"的"
                           +this.account+"帐号支付
了"+ctx.getMoney()+"元");
        //连接银行，进行转帐，就不去管了
    }
}
```

（2）直接在客户端测试就可以了，测试示例代码如下：

```
public class Client {
    public static void main(String[] args) {
      //测试新添加的支付方式
      PaymentStrategy strategyCard2 = new Card2("010998877656");
      PaymentContext ctx4 =
                        new PaymentContext("小张",9000,strategyCard2);
      ctx4.payNow();
    }
}
```

运行看看，好好体会一下。

（3）现在有这么两种扩展的实现方式，到底使用哪一种呢？或者是哪种实现更好呢？下面来比较一下：

对于扩展上下文的方式：这样实现，所有策略的实现风格更统一，策略需要的数据都统一从上下文来获取，这样在使用方法上也很统一；另外，在上下文中添加新的数据，别的相应算法也可以用得上，可以视为公共的数据。但缺点也很明显，如果这些数据只有一个特定的算法来使用，那么这些数据有些浪费；另外每次添加新的算法都去扩展上下文，容易形成复杂的上下文对象层次，也未见得有必要。

对于在策略算法的实现上添加自己需要的数据的方式：这样实现，比较好想，实现简单。但是缺点也很明显，跟其它策略实现的风格不一致，其它策略都是从上下文中来获取数据，而这个策略的实现一部分数据来自上下文，一部分数据来自自己，有些不统一；另外，这样一来，外部使用这些策略算法的时候也不一样了，不太好以一个统一的方式来动态切换策略算法。

两种实现各有优劣，至于如何选择，那就具体问题，具体的分析了。

5：另一种策略模式调用顺序示意图

策略模式调用还有一种情况，就是把Context当做参数来传递给Strategy，也就是本例示范的这种方式，这个时候策略模式的调用顺序如图4所示：

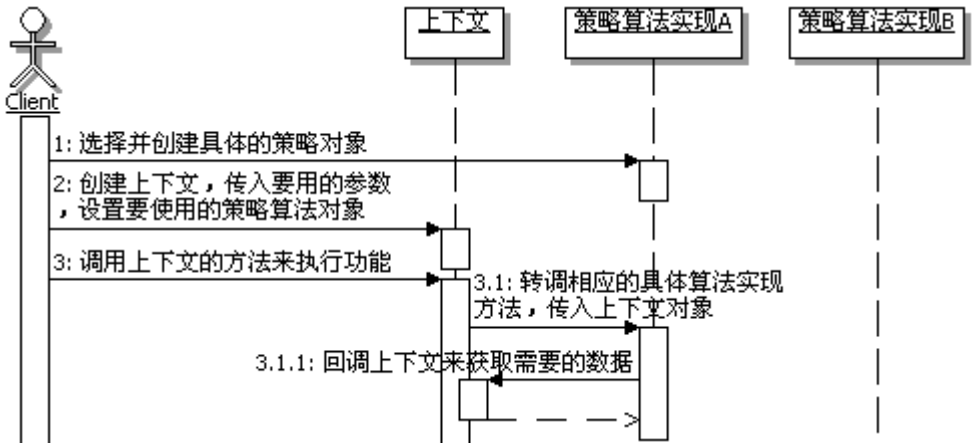


图4 策略模式调用顺序示意图二

未完待续.....

1.15 研磨设计模式之策略模式-6

发表时间: 2010-07-01

3.4 策略模式结合模板方法模式

在实际应用策略模式的过程中，经常会出现这样一种情况，就是发现这一系列算法的实现上存在公共功能，甚至这一系列算法的实现步骤都是一样的，只是在某些局部步骤上有所不同，这个时候，就需要对策略模式进行些许的变化使用了。

对于一系列算法的实现上存在公共功能的情况，策略模式可以有如下三种实现方式：

- 一个是在上下文当中实现公共功能，让所有具体的策略算法回调这些方法。
- 另外一种情况就是把策略的接口改成抽象类，然后在里面实现具体算法的公共功能。
- 还有一种情况是给所有的策略算法定义一个抽象的父类，让这个父类去实现策略的接口，然后在这个父类里面去实现公共的功能。

更进一步，如果这个时候发现“一系列算法的实现步骤都是一样的，只是在某些局部步骤上有所不同”的情况，那就可以在这个抽象类里面定义算法实现的骨架，然后让具体的策略算法去实现变化的部分。这样的一个结构自然就变成了策略模式来结合模板方法模式了，那个抽象类就成了模板方法模式的模板类。

在上一章我们讨论过模板方法模式来结合策略模式的方式，也就是主要的结构是模板方法模式，局部采用策略模式。而这里讨论的是策略模式来结合模板方法模式，也就是主要的结构是策略模式，局部实现上采用模板方法模式。通过这个示例也可以看出来，模式之间的结合是没有定势的，要具体问题具体分析。

此时策略模式结合模板方法模式的系统结构如下图5所示：

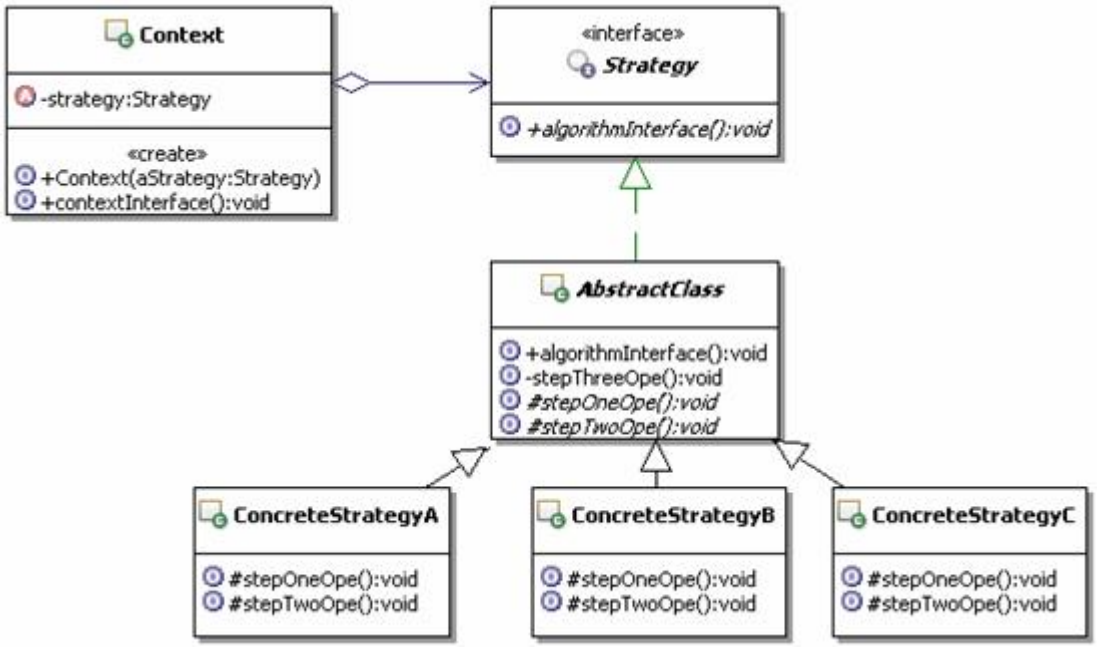


图5 策略模式结合模板方法模式的结构示意图

还是用实际的例子来说吧，比如上面那个记录日志的例子，如果现在需要在所有的消息前面都添加上日志

时间，也就是说现在记录日志的步骤变成了：第一步为日志消息添加日志时间；第二步具体记录日志。

那么该怎么实现呢？

（1）记录日志的策略接口没有变化，为了看起来方便，还是示例一下，示例代码如下：

```
/**
 * 日志记录策略的接口
 */
public interface LogStrategy {
    /**
     * 记录日志
     * @param msg 需记录的日志信息
     */
    public void log(String msg);
}
```

（2）增加一个实现这个策略接口的抽象类，在里面定义记录日志的算法骨架，相当于模板方法模式的模板，示例代码如下：

```
/**
 * 实现日志策略的抽象模板，实现给消息添加时间
 */
public abstract class LogStrategyTemplate implements LogStrategy{
    public final void log(String msg) {
      //第一步：给消息添加记录日志的时间
      DateFormat df = new SimpleDateFormat(
                                                    "yyyy-MM-dd HH:mm:ss SSS");
      msg = df.format(new java.util.Date())+" 内容是："+ msg;
      //第二步：真正执行日志记录
      doLog(msg);
    }
    /**
     * 真正执行日志记录，让子类去具体实现
     * @param msg 需记录的日志信息
     */
    protected abstract void doLog(String msg);
}
```

（3）这个时候那两个具体的日志算法实现也需要做些改变，不再直接实现策略接口了，而是继承模板，实现模板方法了。这个时候记录日志到数据库的类，示例代码如下：

```
/**
 * 把日志记录到数据库
 */
public class DbLog extends LogStrategyTemplate{
    //除了定义上发生了改变外，具体的实现没变
```

```
    public void doLog(String msg) {
      //制造错误
      if(msg!=null && msg.trim().length()>5){
         int a = 5/0;
      }
      System.out.println("现在把 '"+msg+"' 记录到数据库中");
    }
}
```

同理实现记录日志到文件的类如下：

```
/**
 * 把日志记录到数据库
 */
public class FileLog extends LogStrategyTemplate{
    public void doLog(String msg) {
      System.out.println("现在把 '"+msg+"' 记录到文件中");
    }
}
```

（4）算法实现的改变不影响使用算法的上下文，上下文跟前面一样，示例代码如下：

```
/**
 * 日志记录的上下文
 */
public class LogContext {
    /**
     * 记录日志的方法，提供给客户端使用
     * @param msg 需记录的日志信息
     */
    public void log(String msg){
      //在上下文里面，自行实现对具体策略的选择
      //优先选用策略：记录到数据库
      LogStrategy strategy = new DbLog();
      try{
         strategy.log(msg);
      }catch(Exception err){
         //出错了，那就记录到文件中
         strategy = new FileLog();
         strategy.log(msg);
      }
    }
}
```

（5）客户端跟以前也一样，示例代码如下：

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

```
        LogContext log = new LogContext();
        log.log("记录日志");
        log.log("再次记录日志");
    }
}
```

运行一下客户端再次测试看看，体会一下，看看结果是否带上了时间。

通过这个示例，好好体会一下策略模式和模板方法模式的组合使用，在实用开发中是很常见的方式。

3.5 策略模式的优缺点

- 定义一系列算法

  策略模式的功能就是定义一系列算法，实现让这些算法可以相互替换。所以会为这一系列算法定义公共的接口，以约束一系列算法要实现的功能。如果这一系列算法具有公共功能，可以把策略接口实现成为抽象类，把这些公共功能实现到父类里面，对于这个问题，前面讲了三种处理方法，这里就不罗嗦了。

避免多重条件语句

根据前面的示例会发现，策略模式的一系列策略算法是平等的，可以互换的，写在一起就是通过if-else结构来组织，如果此时具体的算法实现里面又有条件语句，就构成了多重条件语句，使用策略模式能避免这样的多重条件语句。

如下示例来演示了不使用策略模式的多重条件语句，示例代码如下：

```
public class OneClass {
    /**
     * 示范多重条件语句
     * @param type 某个用于判断的类型
     */
    public void oneMethod(int type){
      //使用策略模式的时候，这些算法的处理代码就被拿出去，
      //放到单独的算法实现类去了，这里就不再是多重条件了

       if(type==1){
          //算法一示范
          //从某个地方获取这个s的值
          String s = "";
          //然后判断进行相应处理
          if(s.indexOf("a") > 0){
             //处理
          }else{
             //处理
          }
       }else if(type==2){
          //算法二示范
```

```
        //从某个地方获取这个a的值
        int a = 3;
        //然后判断进行相应处理
        if(a > 10){
          //处理
        }else{
          //处理
        }
      }
    }
}
```

- 更好的扩展性

  在策略模式中扩展新的策略实现非常容易，只要增加新的策略实现类，然后在选择使用策略的地方选择使用这个新的策略实现就好了。

- 客户必须了解每种策略的不同

  策略模式也有缺点，比如让客户端来选择具体使用哪一个策略，这就可能会让客户需要了解所有的策略，还要了解各种策略的功能和不同，这样才能做出正确的选择，而且这样也暴露了策略的具体实现。

- 增加了对象数目

  由于策略模式把每个具体的策略实现都单独封装成为类，如果备选的策略很多的话，那么对象的数目就会很可观。

- 只适合扁平的算法结构

  策略模式的一系列算法地位是平等的，是可以相互替换的，事实上构成了一个扁平的算法结构，也就是在一个策略接口下，有多个平等的策略算法，就相当于兄弟算法。而且在运行时刻只有一个算法被使用，这就限制了算法使用的层级，使用的时候不能嵌套使用。

  对于出现需要嵌套使用多个算法的情况，比如折上折、折后返卷等业务的实现，需要组合或者是嵌套使用多个算法的情况，可以考虑使用装饰模式、或是变形的职责链、或是AOP等方式来实现。

3.6 思考策略模式

1：策略模式的本质

策略模式的本质：**分离算法，选择实现。**

仔细思考策略模式的结构和实现的功能，会发现，如果没有上下文，策略模式就回到了最基本的接口和实现了，只要是面向接口编程的，那么就能够享受到接口的封装隔离带来的好处。也就是通过一个统一的策略接口来封装和隔离具体的策略算法，面向接口编程的话，自然不需要关心具体的策略实现，也可以通过使用不同的实现类来实例化接口，从而实现切换具体的策略。

看起来好像没有上下文什么事情，但是如果没有上下文，那么就需要客户端来直接与具体的策略交互，尤其是当需要提供一些公共功能，或者是相关状态存储的时候，会大大增加客户端使用的难度。因此，引入上下文还是很必要的，有了上下文，这些工作就由上下文来完成了，客户端只需要与上下文交互就可以了，这样会让整个设计模式更独立、更有整体性，也让客户端更简单。

但纵观整个策略模式实现的功能和设计，它的本质还是“分离算法，选择实现”，因为分离并封装了算

法，才能够很容易的修改和添加算法；也能很容易的动态切换使用不同的算法，也就是动态选择一个算法来实现需要的功能了。

2：对设计原则的体现

从设计原则上来看，策略模式很好的体现了开-闭原则。策略模式通过把一系列可变的算法进行封装，并定义出合理的使用结构，使得在系统出现新算法的时候，能很容易的把新的算法加入到已有的系统中，而已有的实现不需要做任何修改。这在前面的示例中已经体现出来了，好好体会一下。

从设计原则上来看，策略模式还很好的体现了里氏替换原则。策略模式是一个扁平结构，一系列的实现算法其实是兄弟关系，都是实现同一个接口或者继承的同一个父类。这样只要使用策略的客户保持面向抽象类型编程，就能够使用不同的策略的具体实现对象来配置它，从而实现一系列算法可以相互替换。

3：何时选用策略模式

建议在如下情况中，选用策略模式：

- 出现有许多相关的类，仅仅是行为有差别的情况，可以使用策略模式来使用多个行为中的一个来配置一个类的方法，实现算法动态切换
- 出现同一个算法，有很多不同的实现的情况，可以使用策略模式来把这些“不同的实现”实现成为一个算法的类层次
- 需要封装算法中，与算法相关的数据的情况，可以使用策略模式来避免暴露这些跟算法相关的数据结构
- 出现抽象一个定义了很多行为的类，并且是通过多个if-else语句来选择这些行为的情况，可以使用策略模式来代替这些条件语句

3.7 相关模式

- 策略模式和状态模式

  这两个模式从模式结构上看是一样的，但是实现的功能是不一样的。

  状态模式是根据状态的变化来选择相应的行为，不同的状态对应不同的类，每个状态对应的类实现了该状态对应的功能，在实现功能的同时，还会维护状态数据的变化。这些实现状态对应的功能的类之间是不能相互替换的。

  策略模式是根据需要或者是客户端的要求来选择相应的实现类，各个实现类是平等的，是可以相互替换的。

  另外策略模式可以让客户端来选择需要使用的策略算法，而状态模式一般是由上下文，或者是在状态实现类里面来维护具体的状态数据，通常不由客户端来指定状态。
- 策略模式和模板方法模式

  这两个模式可组合使用，如同前面示例的那样。

  模板方法重在封装算法骨架，而策略模式重在分离并封装算法实现。
- 策略模式和享元模式

  这两个模式可组合使用。

策略模式分离并封装出一系列的策略算法对象，这些对象的功能通常都比较单一，很多时候就是为了实现某个算法的功能而存在，因此，针对这一系列的、多个细粒度的对象，可以应用享元模式来节省资源，但前提是这些算法对象要被频繁的使用，如果偶尔用一次，就没有必要做成享元了。

策略模式结束,谢谢观赏!鞠躬ing

1.16 研磨设计模式之命令模式-1

发表时间: 2010-07-05

命令模式也是开发中常见的一个模式,也不是太难,比较简单,下面来详细的写一下命令模式。

命令模式(Command)

1 场景问题

1.1 如何开机

估计有些朋友看到这个标题会非常奇怪，电脑装配好了，如何开机？不就是按下启动按钮就可以了吗？难道还有什么玄机不成。

对于使用电脑的客户——就是我们来说，开机确实很简单，按下启动按钮，然后耐心等待就可以了。但是当我们按下启动按钮过后呢？谁来处理？如何处理？都经历了怎样的过程，才让电脑真正的启动起来，供我们使用。

先一起来简单的认识一下电脑的启动过程，了解一下即可。

- 当我们按下启动按钮，电源开始向主板和其它设备供电
- 主板的系统BIOS（基本输入输出系统）开始加电后自检
- 主板的BIOS会依次去寻找显卡等其它设备的BIOS，并让它们自检或者初始化
- 开始检测CPU、内存、硬盘、光驱、串口、并口、软驱、即插即用设备等等
- BIOS更新ESCD（扩展系统配置数据），ESCD是BIOS和操作系统交换硬件配置数据的一种手段
- 等前面的事情都完成后，BIOS才按照用户的配置进行系统引导，进入操作系统里面，等到操作系统装载并初始化完毕，就出现我们熟悉的系统登录界面了。

1.2 与我何干

讲了一通电脑启动的过程，有些朋友会想，这与我何干呢？

没错，看起来这些硬件知识跟你没有什么大的关系，但是，如果现在提出一个要求：请你用软件把上面的过程表现出来，你该如何实现？

首先把上面的过程总结一下，主要就这么几个步骤：首先加载电源，然后是设备检查，再然后是装载系统，最后电脑就正常启动了。可是谁来完成这些过程？如何完成？

不能让使用电脑的客户——就是我们来做这些工作吧，真正完成这些工作的是主板，那么客户和主板如何

发生联系呢？现实中，是用连接线把按钮连接到主板上的，这样当客户按下按钮的时候，就相当于发命令给主板，让主板去完成后续的工作。

另外，从客户的角度来看，开机就是按下按钮，不管什么样的主板都是一样的，也就是说，客户只管发出命令，谁接收命令，谁实现命令，如何实现，客户是不关心的。

1.3 有何问题

把上面的问题抽象描述一下：客户端只是想要发出命令或者请求，不关心请求的真正接收者是谁，也不关心具体如何实现，而且同一个请求的动作可以有不同的请求内容，当然具体的处理功能也不一样，请问该怎么实现？

2 解决方案

2.1 命令模式来解决

用来解决上述问题的一个合理的解决方案就是命令模式。那么什么是命令模式呢？

（1）命令模式定义

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；对请求排队或记录请求日志，以及支持可撤销的操作。

（2）应用命令模式来解决的思路

首先来看看实际电脑的解决方案

先画个图来描述一下，看看实际的电脑是如何处理上面描述的这个问题的，如图1所示：

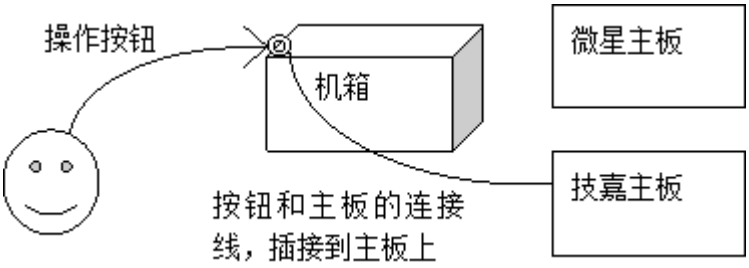


图1 电脑操作示意图

当客户按下按钮的时候，按钮本身并不知道如何处理，于是通过连接线来请求主板，让主板去完成真正启动机器的功能。

这里为了描述它们之间的关系，把主板画到了机箱的外面。如果连接线连接到不同的主板，那么真正执行按钮请求的主板也就不同了，而客户是不知道这些变化的。

通过引入按钮和连接线，来让发出命令的客户和命令的真正实现者——主板完全解耦，客户操作的始终是按钮，按钮后面的事情客户就统统不管了。

要用程序来解决上面提出的问题，一种自然的方案就是来模拟上述解决思路。

在命令模式中，会定义一个命令的接口，用来约束所有的命令对象，然后提供具体的命令实现，每个命令

实现对象是对客户端某个请求的封装，对应于机箱上的按钮，一个机箱上可以有很多按钮，也就相当于会有多个具体的命令实现对象。

在命令模式中，命令对象并不知道如何处理命令，会有相应的接收者对象来真正执行命令。就像电脑的例子，机箱上的按钮并不知道如何处理功能，而是把这个请求转发给主板，由主板来执行真正的功能，这个主板就相当于命令模式的接收者。

在命令模式中，命令对象和接收者对象的关系，并不是与生俱来的，需要有一个装配的过程，命令模式中的Client对象就来实现这样的功能。这就相当于在电脑的例子中，有了机箱上的按钮，也有了主板，还需要有一个连接线把这个按钮连接到主板上才行。

命令模式还会提供一个Invoker对象来持有命令对象，就像电脑的例子，机箱上会有多个按钮，这个机箱就相当于命令模式的Invoker对象。这样一来，命令模式的客户端就可以通过Invoker来触发并要求执行相应的命令了，这也相当于真正的客户是按下机箱上的按钮来操作电脑一样。

2.2 模式结构和说明

命令模式的结构如图2所示：

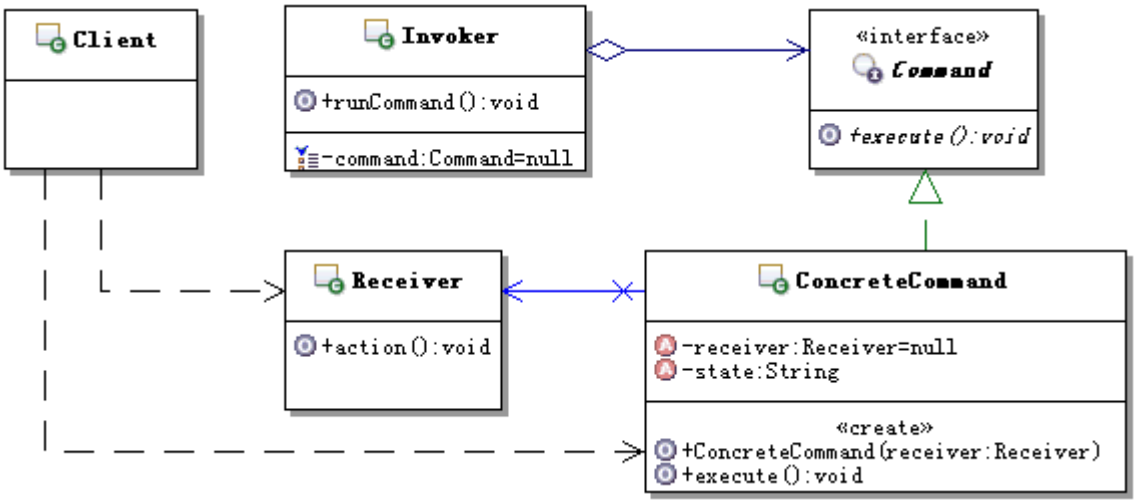


图2 命令模式结构图

Command：

定义命令的接口，声明执行的方法。

ConcreteCommand：

命令接口实现对象，是“虚”的实现；通常会持有接收者，并调用接收者的功能来完成命令要执行的操作。

Receiver：

接收者，真正执行命令的对象。任何类都可能成为一个接收者，只要它能够实现命令要求实现的相应功能。

Invoker：

要求命令对象执行请求，通常会持有命令对象，可以持有很多的命令对象。这个是客户端真正触发命令并

要求命令执行相应操作的地方，也就是说相当于使用命令对象的入口。

Client：

创建具体的命令对象，并且设置命令对象的接收者。注意这个不是我们常规意义上的客户端，而是在组装命令对象和接收者，或许，把这个Client称为装配者会更好理解，因为真正使用命令的客户端是从Invoker来触发执行。

2.3 命令模式示例代码

（1）先来看看命令接口的定义，示例代码如下：

```
/**
 * 命令接口，声明执行的操作
 */
public interface Command {
        /**
         * 执行命令对应的操作
         */
        public void execute();
}
```

（2）再来看看具体的命令实现对象，示例代码如下：

```
/**
 * 具体的命令实现对象
 */
public class ConcreteCommand implements Command {
        /**
         * 持有相应的接收者对象
         */
        private Receiver receiver = null;
        /**
         * 示意，命令对象可以有自己的状态
         */
        private String state;
        /**
         * 构造方法，传入相应的接收者对象
         * @param receiver 相应的接收者对象
         */
```

```
        public ConcreteCommand(Receiver receiver){
                this.receiver = receiver;
        }
        public void execute() {
                //通常会转调接收者对象的相应方法，让接收者来真正执行功能
                receiver.action();
        }
}
```

（3）再来看看接收者对象的实现示意，示例代码如下：

```
/**
 * 接收者对象
 */
public class Receiver {
        /**
         * 示意方法，真正执行命令相应的操作
         */
        public void action(){
                //真正执行命令操作的功能代码
        }
}
```

（4）接下来看看Invoker对象，示例代码如下：

```
/**
 * 调用者
 */
public class Invoker {
        /**
         * 持有命令对象
         */
        private Command command = null;
        /**
```

```
         * 设置调用者持有的命令对象
         * @param command 命令对象
         */
        public void setCommand(Command command) {
                this.command = command;
        }
        /**
         * 示意方法，要求命令执行请求
         */
        public void runCommand() {
                //调用命令对象的执行方法
                command.execute();
        }
}
```

（5）再来看看Client的实现，**注意这个不是我们通常意义上的测试客户端，主要功能是要创建命令对象并设定它的接收者，因此这里并没有调用执行的代码**，示例代码如下：

```
public class Client {
        /**
         * 示意，负责创建命令对象，并设定它的接收者
         */
        public void assemble(){
                //创建接收者
                Receiver receiver = new Receiver();
                //创建命令对象，设定它的接收者
                Command command = new ConcreteCommand(receiver);
                //创建Invoker，把命令对象设置进去
                Invoker invoker = new Invoker();
                invoker.setCommand(command);
        }
}
```

2.4 使用命令模式来实现示例

要使用命令模式来实现示例，需要先把命令模式中所涉及的各个部分，在实际的示例中对应出来，然后才能按照命令模式的结构来设计和实现程序。根据前面描述的解决思路，大致对应如下：

- 机箱上的按钮就相当于是命令对象
- 机箱相当于是Invoker
- 主板相当于接收者对象
- 命令对象持有一个接收者对象，就相当于是给机箱的按钮连上了一根连接线
- 当机箱上的按钮被按下的时候，机箱就把这个命令通过连接线发送出去。

主板类才是真正实现开机功能的地方，是真正执行命令的地方，也就是“接收者”。命令的实现对象，其实是个“虚”的实现，就如同那根连接线，它哪知道如何实现啊，还不就是把命令传递给连接线连到的主板。

使用命令模式来实现示例的结构如图3所示：

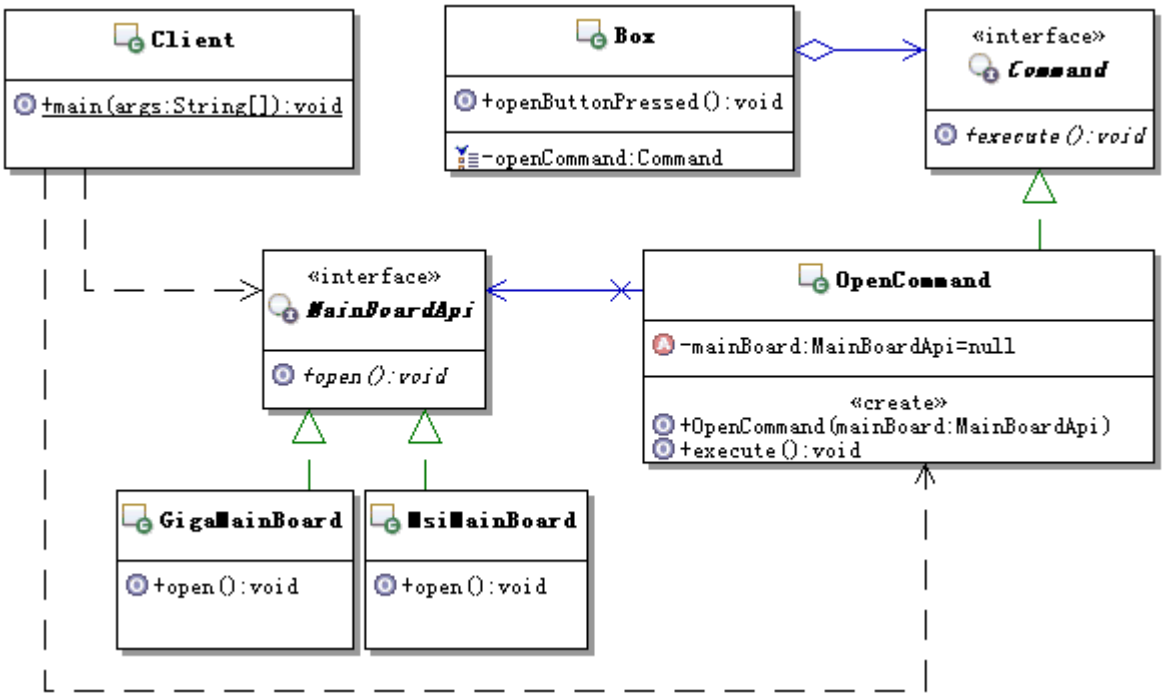


图3 使用命令模式来实现示例的结构示意图

还是来看看示例代码，会比较清楚。

（1）定义主板

根据前面的描述，我们会发现，真正执行客户命令或请求的是主板，也只有主板才知道如何去实现客户的命令，因此先来抽象主板，把它用对象描述出来。

先来定义主板的接口，最起码主板会有一个能开机的方法，示例代码如下：

```
/**
 * 主板的接口
 */
```

```
public interface MainBoardApi {
        /**
         * 主板具有能开机的功能
         */
        public void open();
}
```

定义了接口，那就接着定义实现类吧，定义两个主板的实现类，一个是技嘉主板，一个是微星主板，现在的实现是一样的，但是不同的主板对同一个命令的操作可以是不同的，这点大家要注意。由于两个实现基本一样，就示例一个，示例代码如下：

```
/**
 * 技嘉主板类，开机命令的真正实现者，在Command模式中充当Receiver
 */
public class GigaMainBoard implements MainBoardApi{
        /**
         * 真正的开机命令的实现
         */
        public void open(){
                System.out.println("技嘉主板现在正在开机，请等候");
                System.out.println("接通电源......");
                System.out.println("设备检查......");
                System.out.println("装载系统......");
                System.out.println("机器正常运转起来......");
                System.out.println("机器已经正常打开，请操作");
        }
}
```

微星主板的实现和这个完全一样，只是把技嘉改名成微星了。

（2）定义命令接口和命令的实现

对于客户来说，开机就是按下按钮，别的什么都不想做。把用户的这个动作抽象一下，就相当于客户发出了一个命令或者请求，其它的客户就不关心了。为描述客户的命令，现定义出一个命令的接口，里面只有一个方法，那就是执行，示例代码如下：

```
/**
 * 命令接口，声明执行的操作
 */
public interface Command {
        /**
         * 执行命令对应的操作
         */
        public void execute();
}
```

有了命令的接口，再来定义一个具体的实现，其实就是模拟现实中机箱上按钮的功能，因为我们按下的是按钮，但是按钮本身是不知道如何启动电脑的，它需要把这个命令转给主板，让主板去真正执行开机功能。示例代码如下：

```
/**
 * 开机命令的实现，实现Command接口，
 * 持有开机命令的真正实现，通过调用接收者的方法来实现命令
 */
public class OpenCommand implements Command{
        /**
         * 持有真正实现命令的接收者——主板对象
         */
        private MainBoardApi mainBoard = null;
        /**
         * 构造方法，传入主板对象
         * @param mainBoard 主板对象
         */
        public OpenCommand(MainBoardApi mainBoard) {
                this.mainBoard = mainBoard;
        }

        public void execute() {
                //对于命令对象，根本不知道如何开机，会转调主板对象
                //让主板去完成开机的功能
                this.mainBoard.open();
```

```
        }
}
```

由于客户不想直接和主板打交道，而且客户根本不知道具体的主板是什么，客户只是希望按下启动按钮，电脑就正常启动了，就这么简单。就算换了主板，客户还是一样的按下启动按钮就可以了。

换句话说就是：客户想要和主板完全解耦，怎么办呢？

这就需要在客户和主板之间建立一个中间对象了，客户发出的命令传递给这个中间对象，然后由这个中间对象去找真正的执行者——主板，来完成工作。

很显然，这个中间对象就是上面的命令实现对象，请注意：这个实现其实是个虚的实现，真正的实现是主板完成的，在这个虚的实现里面，是通过转调主板的功能来实现的，主板对象实例，是从外面传进来的。

（3）提供机箱

客户需要操作按钮，按钮是放置在机箱之上的，所以需要把机箱也定义出来，示例代码如下：

```
/**
 * 机箱对象，本身有按钮，持有按钮对应的命令对象
 */
public class Box {
        /**
         * 开机命令对象
         */
        private Command openCommand;
        /**
         * 设置开机命令对象
         * @param command 开机命令对象
         */
        public void setOpenCommand(Command command){
                this.openCommand = command;
        }
        /**
         * 提供给客户使用，接收并响应用户请求，相当于按钮被按下触发的方法
         */
        public void openButtonPressed(){
                //按下按钮，执行命令
                openCommand.execute();
```

```
        }
}
```

（4）客户使用按钮

抽象好了机箱和主板，命令对象也准备好了，客户想要使用按钮来完成开机的功能，在使用之前，客户的第一件事情就应该是把按钮和主板组装起来，形成一个完整的机器。

在实际生活中，是由装机工程师来完成这部分工作，这里为了测试简单，直接写在客户端开头了。机器组装好过后，客户应该把与主板连接好的按钮对象放置到机箱上，等待客户随时操作。把这个过程也用代码描述出来，示例代码如下：

```
public class Client {
        public static void main(String[] args) {
                //1：把命令和真正的实现组合起来，相当于在组装机器，
                //把机箱上按钮的连接线插接到主板上。
                MainBoardApi mainBoard = new GigaMainBoard();
                OpenCommand openCommand = new OpenCommand(mainBoard);
                //2：为机箱上的按钮设置对应的命令，让按钮知道该干什么
                Box box = new Box();
                box.setOpenCommand(openCommand);

                //3：然后模拟按下机箱上的按钮
                box.openButtonPressed();
        }
}
```

运行一下，看看效果，输出如下：

```
技嘉主板现在正在开机，请等候
接通电源......
设备检查......
装载系统......
机器正常运转起来......
机器已经正常打开，请操作
```

你可以给命令对象组装不同的主板实现类，然后再次测试，看看效果。

事实上，你会发现，如果对象结构已经组装好了过后，对于真正的客户端，也就是真实的用户而言，任务就是面对机箱，按下机箱上的按钮，就可以执行开机的命令了，实际生活中也是这样的。

（5）小结

如同前面的示例，把客户的开机请求封装成为一个OpenCommand对象，客户的开机操作就变成了执行OpenCommand对象的方法了？如果还有其它的命令对象，比如让机器重启的ResetCommand对象；那么客户按下按钮的动作，就可以用这不同的命令对象去匹配，也就是对客户进行参数化。

用大白话描述就是：客户按下一个按钮，到底是开机还是重启，那要看参数化配置的是哪一个具体的按钮对象，如果参数化的是开机的命令对象，那就执行开机的功能，如果参数化的是重启的命令对象，那就执行重启的功能。虽然按下的是同一个按钮，但是请求是不同的，对应执行的功能也就不同了。

在模式讲解的时候会给大家一个参数化配置的示例，这里就不多讲了。至于对请求排队或记录请求日志，以及支持可撤销的操作等功能，也放到模式讲解里面。

未完待续.....

1.17 研磨设计模式之命令模式-2

发表时间: 2010-07-10

3 模式讲解

3.1 认识命令模式

（1）命令模式的关键

命令模式的关键之处就是把请求封装成为对象，也就是命令对象，并定义了统一的执行操作的接口，这个命令对象可以被存储、转发、记录、处理、撤销等，整个命令模式都是围绕这个对象在进行。

（2）命令模式的组装和调用

在命令模式中经常会有一个命令的组装者，用它来维护命令的“虚”实现和真实实现之间的关系。如果是超级智能的命令，也就是说命令对象自己完全实现好了，不需要接收者，那就是命令模式的退化，不需要接收者，自然也不需要组装者了。

而真正的用户就是具体化请求的内容，然后提交请求进行触发就好了。真正的用户会通过invoker来触发命令。

在实际开发过程中，Client和Invoker可以融合在一起，由客户在使用命令模式的时候，先进行命令对象和接收者的组装，组装完成后，就可以调用命令执行请求。

（3）命令模式的接收者

接收者可以是任意的类，对它没有什么特殊要求，这个对象知道如何真正执行命令的操作，执行时是从command的实现类里面转调过来。

一个接收者对象可以处理多个命令，接收者和命令之间没有约定的对应关系。接收者提供的方法个数、名称、功能和命令中的可以不一样，只要能够通过调用接收者的方法来实现命令对应的功能就可以了。

（4）智能命令

在标准的命令模式里面，命令的实现类是没有真正实现命令要求的功能的，真正执行命令的功能的是接收者。

如果命令的实现对象比较智能，它自己就能真实地实现命令要求的功能，而不再需要调用接收者，那么这种情况就称为智能命令。

也可以有半智能的命令，命令对象知道部分实现，其它的还是需要调用接收者来完成，也就是说命令的功能由命令对象和接收者共同来完成。

（5）发起请求的对象和真正实现的对象是解耦的

请求究竟由谁处理，如何处理，发起请求的对象是不知道的，也就是发起请求的对象和真正实现的对象是解耦的。发起请求的对象只管发出命令，其它的就不管了。

（6）命令模式的调用顺序示意图

使用命令模式的过程分成两个阶段，一个阶段是组装命令对象和接收者对象的过程，另外一个阶段是触发调用Invoker，来让命令真正执行的过程。

先看看组装过程的调用顺序示意图，如图4所示：

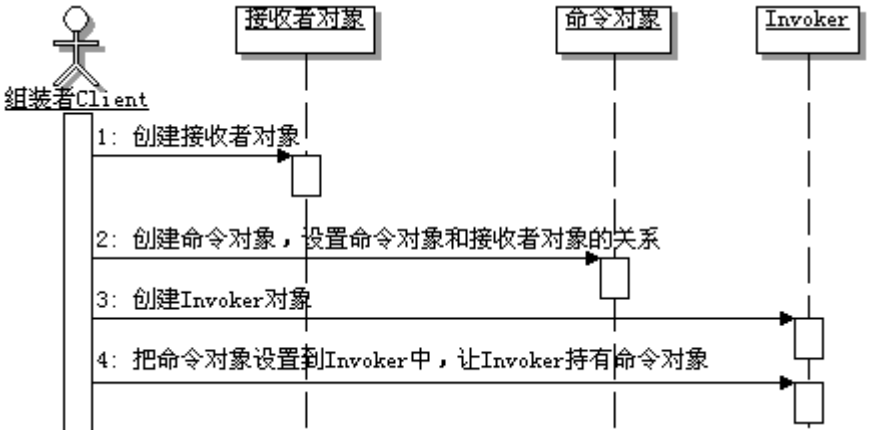


图4 命令模式组装过程的调用顺序示意图

接下来再看看真正执行命令时的调用顺序示意图，如图5所示：

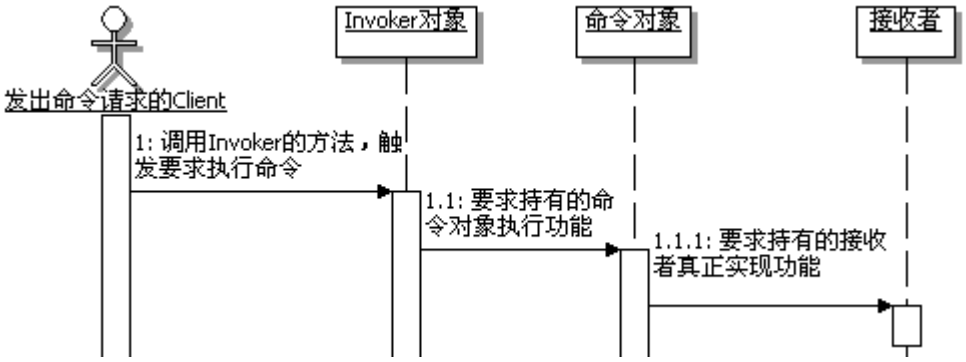


图5 命令模式执行过程的调用顺序示意图

3.2 参数化配置

所谓命令模式的参数化配置，指的是：可以用不同的命令对象，去参数化配置客户的请求。

像前面描述的那样：客户按下一个按钮，到底是开机还是重启，那要看参数化配置的是哪一个具体的按钮对象，如果参数化的是开机的命令对象，那就执行开机的功能，如果参数化的是重启的命令对象，那就执行重启的功能。虽然按下的是同一个按钮，相当于是同一个请求，但是为请求配置不同的按钮对象，那就会执行不同的功能。

把这个功能用代码实现出来，一起来体会一下命令模式的参数化配置。

（1）同样先定义主板接口吧，现在想要添加一个重启的按钮，因此主板需要添加一个方法来实现重启的功能，示例代码如下：

/** * 主板的接口 */ public interface MainBoardApi { /** * 主板具有能开机的功能 */ public void open(); /** * 主板具有实现重启的功能 */ public void reset(); }

接口发生了改变，实现类也得有相应的改变，由于两个主板的实现示意差不多，因此还是只示例一个，示例代码如下：

/** * 技嘉主板类，命令的真正实现者，在Command模式中充当Receiver */ public class GigaMainBoard implements MainBoardApi{ /** * 真正的开机命令的实现 */ public void open(){ System.out.println("技嘉主板现在正在开机，请等候"); System.out.println("接通电源....."); System.out.println("设备检查....."); System.out.println("装载系统....."); System.out.println("机器正常运转起来....."); System.out.println("机器已经正常打开，请操作"); } /** * 真正的重新启动机器命令的实现 */ public void reset(){ System.out.println("技嘉主板现在正在重新启动机器，请等候"); System.out.println("机器已经正常打开，请操作"); } }

（2）该来定义命令和按钮了，命令接口没有任何变化，原有的开机命令的实现也没有任何变化，只是新添加了一个重启命令的实现，示例代码如下：

/** * 重启机器命令的实现，实现Command接口， * 持有重启机器命令的真正实现，通过调用接收者的方法来实现命令 */ public class ResetCommand implements Command{ /** * 持有真正实现命令的接收者——主板对象 */ private MainBoardApi mainBoard = null; /** * 构造方法，传入主板对象 * @param mainBoard 主板对象 */ public ResetCommand(MainBoardApi mainBoard) { this.mainBoard = mainBoard; } public void execute() { //对于命令对象，根本不知道如何重启机器，会转调主板对象 //让主板去完成重启机器的功能 this.mainBoard.reset(); } }

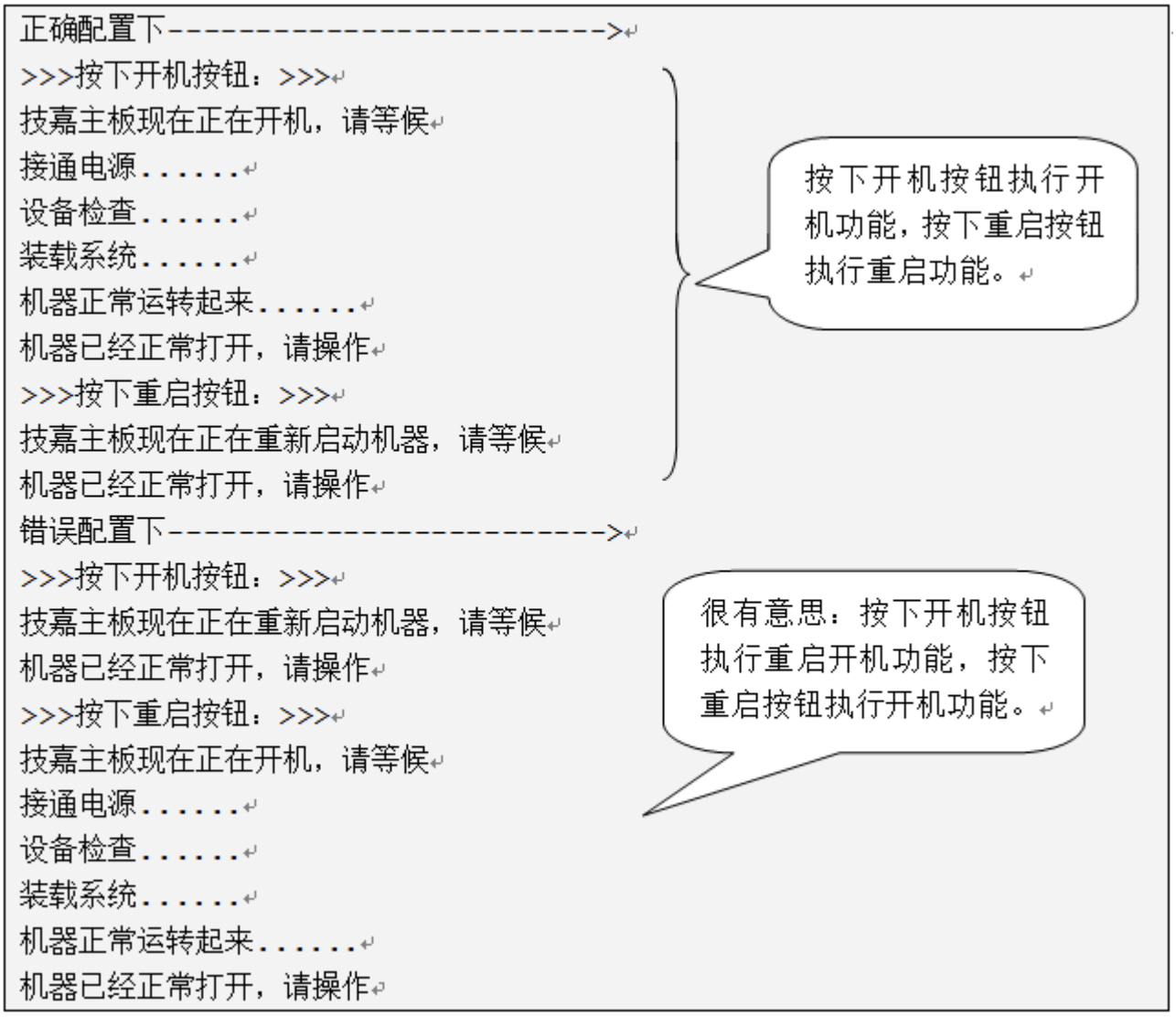
（3）持有命令的机箱也需要修改，现在不只一个命令按钮了，有两个了，所以需要在机箱类里面新添加重启的按钮，为了简单，没有做成集合。示例代码如下：

/** * 机箱对象，本身有按钮，持有按钮对应的命令对象 */ public class Box { private Command openCommand; public void setOpenCommand(Command command){ this.openCommand = command; } public void openButtonPressed(){ //按下按钮，执行命令 openCommand.execute(); } /** * 重启机器命令对象 */ private Command resetCommand; /** * 设置重启机器命令对象 * @param command */ public void setResetCommand(Command command){ this.resetCommand = command; } /** * 提供给客户使用，接收并相应用户请求，相当于重启按钮被按下触发的方法 */ public void resetButtonPressed(){ //按下按钮，执行命令 resetCommand.execute(); } }

（4）看看客户如何使用这两个按钮，示例代码如下

public class Client { public static void main(String[] args) { //1：把命令和真正的实现组合起来，相当于在组装机器， //把机箱上按钮的连接线插接到主板上。 MainBoardApi mainBoard = new GigaMainBoard(); //创建开机命令 OpenCommand openCommand = new OpenCommand(mainBoard); //创建重启机器的命令 ResetCommand resetCommand = new ResetCommand(mainBoard); //2：为机箱上的按钮设置对应的命令，让按钮知道该干什么 Box box = new Box(); **//先正确配置，就是开机按钮对开机命令，重启按钮对重启命令** box.setOpenCommand(openCommand); box.setResetCommand(resetCommand); //3：然后模拟按下机箱上的按钮 System.out.println("正确配置下----------->"); System.out.println(">>>按下开机按钮：>>>"); box.openButtonPressed(); System.out.println(">>>按下重启按钮：>>>"); box.resetButtonPressed(); **//然后来错误配置一回，反正是进行参数化配置 //就是开机按钮对重启命令，重启按钮对开机命令** box.setOpenCommand(resetCommand); box.setResetCommand(openCommand); //4：然后还是来模拟按下机箱上的按钮 System.out.println("错误配置下----------->"); System.out.println(">>>按下开机按钮：>>>"); box.openButtonPressed(); System.out.println(">>>按下重启按钮：>>>"); box.resetButtonPressed(); } }

运行一下看看，很有意思，结果如下：


正确配置下---------->
>>>按下开机按钮：>>>
技嘉主板现在正在开机，请等候
接通电源.....
设备检查.....
装载系统.....
机器正常运转起来.....
机器已经正常打开，请操作
>>>按下重启按钮：>>>
技嘉主板现在正在重新启动机器，请等候
机器已经正常打开，请操作
错误配置下---------->
>>>按下开机按钮：>>>
技嘉主板现在正在重新启动机器，请等候
机器已经正常打开，请操作
>>>按下重启按钮：>>>
技嘉主板现在正在开机，请等候
接通电源.....
设备检查.....
装载系统.....
机器正常运转起来.....
机器已经正常打开，请操作
按下开机按钮执行开机功能，按下重启按钮执行重启功能。
很有意思：按下开机按钮执行重启开机功能，按下重启按钮执行开机功能。

未完待续.....

1.18 研磨设计模式之命令模式-3

发表时间: 2010-07-13

3.3 可撤销的操作

可撤销操作的意思就是：放弃该操作，回到未执行该操作前的状态。这个功能是一个非常重要的功能，几乎所有GUI应用里面都有撤消操作的功能。GUI的菜单是命令模式最典型的应用之一，所以你总是能在菜单上找到撤销这样的菜单项。

既然这么常用，那该如何实现呢？

有两种基本的思路来实现可撤销的操作，**一种是补偿式，又称反操作式**：比如被撤销的操作是加的功能，那撤消的实现就变成减的功能；同理被撤销的操作是打开的功能，那么撤销的实现就变成关闭的功能。

另外一种方式是存储恢复式，意思就是把操作前的状态记录下来，然后要撤销操作的时候就直接恢复回去就可以了。

这里先讲第一种方式，就是补偿式或者反操作式，第二种方式放到备忘录模式中去讲解。为了让大家更好的理解可撤销操作的功能，还是用一个例子来说明会比较清楚。

1：范例需求

考虑一个计算器的功能，最简单的那种，只能实现加减法运算，现在要让这个计算器支持可撤销的操作。

2：补偿式或者反操作式的解决方案

（1）在实现命令接口之前，先来定义真正实现计算的接口，没有它命令什么都做不了，操作运算的接口的示例代码如下：

```
/**
 * 操作运算的接口
 */
public interface OperationApi {
        /**
         * 获取计算完成后的结果
         * @return 计算完成后的结果
         */
        public int getResult();
        /**
         * 设置计算开始的初始值
         * @param result 计算开始的初始值
         */
        public void setResult(int result);
        /**
         * 执行加法
```

```
         * @param num 需要加的数
         */
        public void add(int num);
        /**
         * 执行减法
         * @param num 需要减的数
         */
        public void substract(int num);
}
```

定义了接口，来看看真正执行加减法的实现，示例代码如下：

```
/**
 * 运算类，真正实现加减法运算
 */
public class Operation implements OperationApi{
        /**
         * 记录运算的结果
         */
        private int result;
        public int getResult() {
                return result;
        }
        public void setResult(int result) {
                this.result = result;
        }
        public void add(int num){
                //实现加法功能
                result += num;
        }
        public void substract(int num){
                //实现减法功能
                result -= num;
        }
}
```

（2）接下来，来抽象命令接口，由于要支持可撤销的功能，所以除了跟前面一样定义一个执行方法外，还需要定义一个撤销操作的方法，示例代码如下：

```
/**
 * 命令接口，声明执行的操作，支持可撤销操作
 */
public interface Command {
        /**
         * 执行命令对应的操作
         */
        public void execute();
        /**
         * 执行撤销命令对应的操作
         */
        public void undo();
}
```

（3）应该来实现命令了，具体的命令分成了加法命令和减法命令，先来看看加法命令的实现，示例代码如下：

```
/**
 * 具体的加法命令实现对象
 */
public class AddCommand implements Command{
        /**
         * 持有具体执行计算的对象
         */
        private OperationApi operation = null;
        /**
         * 操作的数据，也就是要加上的数据
         */
        private int opeNum;

        public void execute() {
                //转调接收者去真正执行功能，这个命令是做加法
                this.operation.add(opeNum);
        }
        public void undo() {
```

```
                //转调接收者去真正执行功能
                //命令本身是做加法，那么撤销的时候就是做减法了
                this.operation.substract(opeNum);
        }
              /**
         * 构造方法，传入具体执行计算的对象
         * @param operation 具体执行计算的对象
         * @param opeNum 要加上的数据
         */
        public AddCommand(OperationApi operation,int opeNum){
                this.operation = operation;
                this.opeNum = opeNum;
        }
}
```

减法命令和加法类似，只是在实现的时候和加法反过来了，示例代码如下：

```
/**
 * 具体的减法命令实现对象
 */
public class SubstractCommand implements Command{
        /**
         * 持有具体执行计算的对象
         */
        private OperationApi operation = null;
        /**
         * 操作的数据，也就是要减去的数据
         */
        private int opeNum;
        /**
         * 构造方法，传入具体执行计算的对象
         * @param operation 具体执行计算的对象
         * @param opeNum 要减去的数据
         */
        public SubstractCommand(OperationApi operation,int opeNum){
```

```
                this.operation = operation;
                this.opeNum = opeNum;
        }

        public void execute() {
                //转调接收者去真正执行功能，这个命令是做减法
                this.operation.substract(opeNum);
        }
        public void undo() {
                //转调接收者去真正执行功能
                //命令本身是做减法，那么撤销的时候就是做加法了
                this.operation.add(opeNum);
        }
}
```

（4）接下来应该看看计算器了，计算器就相当于Invoker，持有多个命令对象，计算器是实现可撤销操作的地方。

为了大家更好的理解可撤销的功能，先来看看不加可撤销操作的计算器类什么样子，然后再添加上可撤销的功能示例。示例代码如下：

```
/**
 * 计算器类，计算器上有加法按钮、减法按钮
 */
public class Calculator {
        /**
         * 持有执行加法的命令对象
         */
        private Command addCmd = null;
        /**
         * 持有执行减法的命令对象
         */
        private Command substractCmd = null;
        /**
         * 设置执行加法的命令对象
         * @param addCmd 执行加法的命令对象
         */
```

```
        public void setAddCmd(Command addCmd) {
                this.addCmd = addCmd;
        }
        /**
         * 设置执行减法的命令对象
         * @param substractCmd 执行减法的命令对象
         */
        public void setSubstractCmd(Command substractCmd) {
                this.substractCmd = substractCmd;
        }
        /**
         * 提供给客户使用，执行加法功能
         */
        public void addPressed(){
                this.addCmd.execute();
        }
        /**
         * 提供给客户使用，执行减法功能
         */
        public void substractPressed(){
                this.substractCmd.execute();
        }
}
```

目前看起来跟前面的例子实现得差不多，现在就在这个基本的实现上来添加可撤销操作的功能。要想实现可撤销操作，首先就需要把操作过的命令记录下来，形成命令的历史列表，撤销的时候就从最后一个开始执行撤销。因此我们先在计算器类里面加上命令历史列表，示例代码如下：

```
/**
* 命令的操作的历史记录，在撤销时候用
*/
private List<Command> undoCmds = new ArrayList<Command>();
```

什么时候向命令的历史记录里面加值呢？

很简单，答案是在每个操作按钮被按下的时候，也就是你操作加法按钮或者减法按钮的时候，示例代码如下

```
        public void addPressed(){
                this.addCmd.execute();
                //把操作记录到历史记录里面
                undoCmds.add(this.addCmd);
        }
        public void substractPressed(){
                this.substractCmd.execute();
                //把操作记录到历史记录里面
                undoCmds.add(this.substractCmd);
        }
```

然后在计算器类里面添加上一个撤销的按钮，如果它被按下，那么就从命令历史记录里取出最后一个命令来撤销，撤消完成后要把已经撤销的命令从历史记录里面删除掉，相当于没有执行过该命令了，示例代码如下：

```
public void undoPressed(){
        if(this.undoCmds.size()>0){
                //取出最后一个命令来撤销
                Command cmd = this.undoCmds.get(this.undoCmds.size()-1);
                cmd.undo();
                //然后把最后一个命令删除掉，
                this.undoCmds.remove(cmd);
        }else{
                System.out.println("很抱歉，没有可撤销的命令");
        }
}
```

同样的方式，还可以实现恢复的功能，也为恢复设置一个可恢复的列表，需要恢复的时候从列表里面取最后一个命令进行重新执行就好了，示例代码如下：

```
/**
* 命令被撤销的历史记录，在恢复时候用
*/
private List<Command> redoCmds = new ArrayList<Command>();
```

那么什么时候向这个集合里面赋值呢？大家要注意，恢复的命令数据是来源于撤销的命令，也就是说有撤销才会有恢复，所以在撤销的时候向这个集合里面赋值，注意要在撤销的命令被删除前赋值。示例代码如下:

```
public void undoPressed(){
        if(this.undoCmds.size()>0){
                //取出最后一个命令来撤销
                Command cmd = this.undoCmds.get(this.undoCmds.size()-1);
                cmd.undo();
                //如果还有恢复的功能，那就把这个命令记录到恢复的历史记录里面
                this.redoCmds.add(cmd);
                //然后把最后一个命令删除掉，
                this.undoCmds.remove(cmd);
        }else{
                System.out.println("很抱歉，没有可撤销的命令");
        }
}
```

那么如何实现恢复呢？请看示例代码：

```
public void redoPressed(){
        if(this.redoCmds.size()>0){
                //取出最后一个命令来重做
                Command cmd = this.redoCmds.get(this.redoCmds.size()-1);
                cmd.execute();
                //把这个命令记录到可撤销的历史记录里面
                      this.undoCmds.add(cmd);
                //然后把最后一个命令删除掉
                this.redoCmds.remove(cmd);
        }else{
```

```
                System.out.println("很抱歉，没有可恢复的命令");
        }
}
```

好了，分步讲解了计算器类，一起来看看完整的计算器类的代码：

```
/**
 * 计算器类，计算器上有加法按钮、减法按钮，还有撤销和恢复的按钮
 */
public class Calculator {
        /**
         * 命令的操作的历史记录，在撤销时候用
         */
        private List<Command> undoCmds = new ArrayList<Command>();
        /**
         * 命令被撤销的历史记录，在恢复时候用
         */
        private List<Command> redoCmds = new ArrayList<Command>();

        private Command addCmd = null;
        private Command substractCmd = null;
        public void setAddCmd(Command addCmd) {
                this.addCmd = addCmd;
        }
        public void setSubstractCmd(Command substractCmd) {
                this.substractCmd = substractCmd;
        }
        public void addPressed(){
                this.addCmd.execute();
                //把操作记录到历史记录里面
                undoCmds.add(this.addCmd);
        }
        public void substractPressed(){
                this.substractCmd.execute();
                //把操作记录到历史记录里面
```

```
                undoCmds.add(this.substractCmd);
        }
        public void undoPressed(){
                if(this.undoCmds.size()>0){
                        //取出最后一个命令来撤销
                        Command cmd = this.undoCmds.get(undoCmds.size()-1);
                        cmd.undo();
                        //如果还有恢复的功能，那就把这个命令记录到恢复的历史记录里面
                        this.redoCmds.add(cmd );
                        //然后把最后一个命令删除掉，
                        this.undoCmds.remove(cmd);
                }else{
                        System.out.println("很抱歉，没有可撤销的命令");
                }
        }
        public void redoPressed(){
                if(this.redoCmds.size()>0){
                        //取出最后一个命令来重做
                        Command cmd = this.redoCmds.get(redoCmds.size()-1);
                        cmd.execute();
                        //把这个命令记录到可撤销的历史记录里面
                        this.undoCmds.add(cmd);
                        //然后把最后一个命令删除掉
                        this.redoCmds.remove(cmd);
                }else{
                        System.out.println("很抱歉，没有可恢复的命令");
                }
        }
}
```

（5）终于到可以收获的时候了，写个客户端，组装好命令和接收者，然后操作几次命令，来测试一下撤销和恢复的功能，示例代码如下：

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

```
//1：组装命令和接收者
//创建接收者
OperationApi operation = new Operation();
//创建命令对象，并组装命令和接收者
AddCommand addCmd = new AddCommand   (operation,5);
SubstractCommand substractCmd =
                    new SubstractCommand(operation,3);

//2：把命令设置到持有者，就是计算器里面
Calculator calculator = new Calculator();
calculator.setAddCmd(addCmd);
calculator.setSubstractCmd(substractCmd);

//3:模拟按下按钮，测试一下
calculator.addPressed();
System.out.println("一次加法运算后的结果为："
                    +operation.getResult());
calculator.substractPressed();
System.out.println("一次减法运算后的结果为："
                    +operation.getResult());

//测试撤消
calculator.undoPressed();
System.out.println("撤销一次后的结果为："
                    +operation.getResult());
calculator.undoPressed();
System.out.println("再撤销一次后的结果为："
                    +operation.getResult());

//测试恢复
calculator.redoPressed();
System.out.println("恢复操作一次后的结果为："
                    +operation.getResult());
calculator.redoPressed();
System.out.println("再恢复操作一次后的结果为："
                    +operation.getResult());
```

```
        }
}
```

（6）运行一下，看看结果，享受一下可以撤销和恢复的操作，结果如下：

```
一次加法运算后的结果为：5
一次减法运算后的结果为：2
撤销一次后的结果为：5
再撤销一次后的结果为：0
恢复操作一次后的结果为：5
再恢复操作一次后的结果为：2
```

也就是初始值为0，执行的两次命令操作为先加上5，然后再减去3。看起来也很容易，对不。

未完待续.....

1.19 研磨设计模式之命令模式-4

发表时间: 2010-07-16

3.4 宏命令

什么是宏命令呢？简单点说就是包含多个命令的命令，是一个命令的组合。举个例子来说吧，设想一下你去饭店吃饭的过程：

（1）你走进一家饭店，找到座位坐下

（2）服务员走过来，递给你菜谱

（3）你开始点菜，服务员开始记录菜单，菜单是三联的，点菜完毕，服务员就会把菜单分成三份，一份给后厨，一份给收银台，一份保留备查。

（4）点完菜，你坐在座位上等候，后厨会按照菜单做菜

（5）每做好一份菜，就会由服务员送到你桌子上

（6）然后你就可以大快朵颐了

事实上，到饭店点餐是一个很典型的命令模式应用，作为客户的你，只需要发出命令，就是要吃什么菜，每道菜就相当于一个命令对象，服务员会在菜单上记录你点的菜，然后把菜单传递给后厨，后厨拿到菜单，会按照菜单进行饭菜制作，后厨就相当于接收者，是命令的真正执行者，厨师才知道每道菜具体怎么实现。

在这个过程中，地位比较特殊的是服务员，在不考虑更复杂的管理，比如后厨管理的时候，负责命令和接收者的组装的就是服务员。比如你点了凉菜、热菜，你其实是不知道到底凉菜由谁来完成，热菜由谁来完成的，因此你只管发命令，而组装的工作就由服务员完成了，服务员知道凉菜送到凉菜部，那是已经做好的了，热菜才送到后厨，需要厨师现做，看起来服务员是一个组装者。

同时呢，服务员还持有命令对象，也就是菜单，最后启动命令执行的也是服务员。因此，服务员就相当于标准命令模式中的Client和Invoker的融合。

画个图来描述上述对应关系，如图6所示：

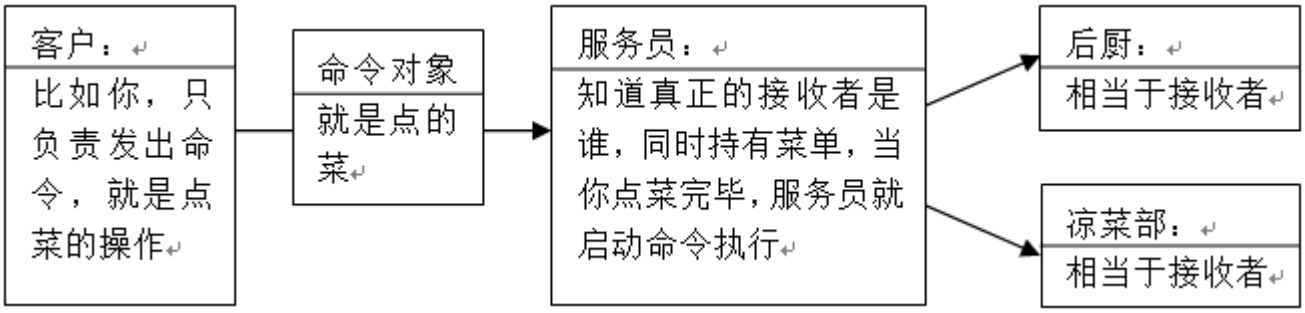


图6 点菜行为与命令模式对应示意图

1：宏命令在哪里？

仔细观察上面的过程，再想想前面的命令模式的实现，看出点什么没有？

前面实现的命令模式，都是客户端发出一个命令，然后马上就执行了这个命令，但是在上面的描述里面呢？是点一个菜，服务员就告诉厨师，然后厨师就开始做吗？很明显不是的，服务员会一直等，等到你点完

菜，当你说“点完了”的时候，服务员才会启动命令的执行，请注意，这个时候执行的就不是一个命令了，而是执行一堆命令。

描述这一堆命令的就是菜单，如果把菜单也抽象成为一个命令，就相当于一个大的命令，当客户说“点完了”的时候，就相当于触发这个大的命令，意思就是执行菜单这个命令就可以了，这个菜单命令包含多个命令对象，一个命令对象就相当于一道菜。

那么这个菜单就相当于我们说的宏命令。

2：如何实现宏命令

宏命令从本质上讲类似于一个命令，基本上把它当命令对象进行处理。但是它跟普通的命令对象又有些不一样，就是宏命令包含有多个普通的命令对象，执行一个宏命令，简单点说，就是执行宏命令里面所包含的所有命令对象，有点打包执行的意味。

（1）先来定义接收者，就是厨师的接口和实现，先看接口，示例代码如下：

```
/**
 * 厨师的接口
 */
public interface CookApi {
        /**
         * 示意，做菜的方法
         * @param name 菜名
         */
        public void cook(String name);
}
```

厨师又分成两类，一类是做热菜的师傅，一类是做凉菜的师傅，先看看做热菜的厨师的实现示意，示例代码如下：

```
/**
 * 厨师对象，做热菜
 */
public class HotCook implements CookApi{
        public void cook(String name) {
                System.out.println("本厨师正在做："+name);
        }
}
```

做凉菜的师傅，示例代码如下：

```
/**
 * 厨师对象，做凉菜
 */
public class CoolCook implements CookApi {
        public void cook(String name) {
                System.out.println("凉菜"+name+"已经做好，本厨师正在装盘。" );
        }
}
```

（2）接下来，来定义命令接口，跟以前一样，示例代码如下：

```
/**
 * 命令接口，声明执行的操作
 */
public interface Command {
        /**
         * 执行命令对应的操作
         */
        public void execute();
}
```

（3）定义好了命令的接口，该来具体实现命令了。

实现方式跟以前一样，持有接收者，当执行命令的时候，转调接收者，让接收者去真正实现功能，这里的接收者就是厨师。

这里实现命令的时候，跟标准的命令模式的命令实现有一点不同，标准的命令模式的命令实现的时候，是通过构造方法传入接收者对象，这里改成了使用setter的方式来设置接收者对象，也就是说可以动态的切换接收者对象，而无须重新构建对象。

示例中定义了三道菜，分别是两道热菜：北京烤鸭、绿豆排骨煲，一道凉菜：蒜泥白肉，三个具体的实现类非常类似，只是菜名不同，为了节省篇幅，这里就只看一个命令对象的具体实现。代码示例如下：

```
/**
 * 命令对象，绿豆排骨煲
```

```
 */
public class ChopCommand implements Command{
        /**
         * 持有具体做菜的厨师的对象
         */
        private CookApi cookApi = null;
        /**
         * 设置具体做菜的厨师的对象
         * @param cookApi 具体做菜的厨师的对象
         */
        public void setCookApi(CookApi cookApi) {
                this.cookApi = cookApi;
        }

        public void execute() {
                this.cookApi.cook("绿豆排骨煲");
        }
}
```

（4）该来组合菜单对象了，也就是宏命令对象。

- 首先宏命令就其本质还是一个命令，所以一样要实现Command接口
- 其次宏命令跟普通命令的不同在于：宏命令是多个命令组合起来的，因此在宏命令对象里面会记录多个组成它的命令对象
- 第三，既然是包含多个命令对象，得有方法让这多个命令对象能被组合进来
- 第四，既然宏命令包含了多个命令对象，执行宏命令对象就相当于依次执行这些命令对象，也就是循环执行这些命令对象

看看代码示例会更清晰些，代码示例如下：

```
/**
 * 菜单对象，是个宏命令对象
 */
public class MenuCommand implements Command {
        /**
         * 用来记录组合本菜单的多道菜品，也就是多个命令对象
```

```
         */
        private Collection<Command> col = new ArrayList<Command>();
        /**
         * 点菜，把菜品加入到菜单中
         * @param cmd 客户点的菜
         */
        public void addCommand(Command cmd){
                col.add(cmd);
        }
        public void execute() {
                //执行菜单其实就是循环执行菜单里面的每个菜
                for(Command cmd : col){
                        cmd.execute();
                }
        }
}
```

（5）该服务员类重磅登场了，它实现的功能，相当于标准命令模式实现中的Client加上Invoker，前面都是文字讲述，看看代码如何实现，示例代码如下：

```
/**
 * 服务员，负责组合菜单，负责组装每个菜和具体的实现者，
 * 还负责执行调用，相当于标准Command模式的Client+Invoker
 */
public class Waiter {
        /**
         * 持有一个宏命令对象——菜单
         */
        private MenuCommand menuCommand = new MenuCommand();
        /**
         * 客户点菜
         * @param cmd 客户点的菜，每道菜是一个命令对象
         */
        public void orderDish(Command cmd){
                //客户传过来的命令对象是没有和接收者组装的
                //在这里组装吧
                CookApi hotCook = new HotCook();
```

```
                CookApi coolCook = new CoolCook();
                //判读到底是组合凉菜师傅还是热菜师傅
                //简单点根据命令的原始对象的类型来判断
                if(cmd instanceof DuckCommand){
                        ((DuckCommand)cmd).setCookApi(hotCook);
                }else if(cmd instanceof ChopCommand){
                        ((ChopCommand)cmd).setCookApi(hotCook);
                }else if(cmd instanceof PorkCommand){
                        //这是个凉菜，所以要组合凉菜的师傅
                        ((PorkCommand)cmd).setCookApi(coolCook);
                }
                //添加到菜单中
                menuCommand.addCommand(cmd);
        }
        /**
         * 客户点菜完毕，表示要执行命令了，这里就是执行菜单这个组合命令
         */
        public void orderOver(){
                this.menuCommand.execute();
        }
}
```

（6）费了这么大力气，终于可以坐下来歇息一下，点菜吃饭吧，一起来看看客户端怎么使用这个宏命令，其实在客户端非常简单，根本看不出宏命令来，代码示例如下：

```
public class Client {
        public static void main(String[] args) {
                //客户只是负责向服务员点菜就好了
                //创建服务员
                Waiter waiter = new Waiter();

                //创建命令对象，就是要点的菜
                Command chop = new ChopCommand();
                Command duck = new DuckCommand();
                Command pork = new PorkCommand();

                //点菜，就是把这些菜让服务员记录下来
```

```
                waiter.orderDish(chop);
                waiter.orderDish(duck);
                waiter.orderDish(pork);

                //点菜完毕
                waiter.orderOver();
        }
}
```

运行一下，享受一下成果，结果如下：

```
本厨师正在做：绿豆排骨煲
本厨师正在做：北京烤鸭
凉菜蒜泥白肉已经做好，本厨师正在装盘。
```

未完待续.....

1.20 研磨设计模式之命令模式-5

发表时间: 2010-07-20

3.5 队列请求

所谓队列请求，就是对命令对象进行排队，组成工作队列，然后依次取出命令对象来执行。多用多线程或者线程池来进行命令队列的处理，当然也可以不用多线程，就是一个线程，一个命令一个命令的循环处理，就是慢点。

继续宏命令的例子，其实在后厨，会收到很多很多的菜单，一般是按照菜单传递到后厨的先后顺序来进行处理，对每张菜单，假定也是按照菜品的先后顺序进行制作，那么在后厨就自然形成了一个菜品的队列，也就是很多个用户的命令对象的队列。

后厨有很多厨师，每个厨师都从这个命令队列里面取出一个命令，然后按照命令做出菜来，就相当于多个线程在同时处理一个队列请求。

因此后厨就是一个很典型的队列请求的例子。

提示一点：后厨的厨师与命令队列之间是没有任何关联的，也就是说是完全解耦的。命令队列是客户发出的命令，厨师只是负责从队列里面取出一个，处理，然后再取下一个，再处理，仅此而已，厨师不知道也不管客户是谁。

下面就一起来看看如何实现队列请求。

1：如何实现命令模式的队列请求

（1）先从命令接口开始，除了execute方法外，新加了一个返回发出命令的桌号，就是点菜的桌号，还有一个是为命令对象设置接收者的方法，也把它添加到接口上，这个是为了后面多线程处理的时候方便使用。示例代码如下：

```
/**
 * 命令接口，声明执行的操作
 */
public interface Command {
        /**
         * 执行命令对应的操作
         */
        public void execute();
        /**
         * 设置命令的接收者
         * @param cookApi 命令的接收者
         */
        public void setCookApi(CookApi cookApi);
        /**
```

```
         * 返回发起请求的桌号，就是点菜的桌号
         * @return 发起请求的桌号
         */
        public int getTableNum();
}
```

（2）厨师的接口也发生了一点变化，在cook的方法上添加了发出命令的桌号，这样在多线程输出信息的时候，才知道到底是在给哪个桌做菜，示例代码如下：

```
/**
 * 厨师的接口
 */
public interface CookApi {
        /**
         * 示意，做菜的方法
         * @param tableNum 点菜的桌号
         * @param name 菜名
         */
        public void cook(int tableNum,String name);
}
```

（3）开始来实现命令接口，为了简单，这次只有热菜，因为要做工作都在后厨的命令队列里面，因此凉菜就不要了，示例代码如下：

```
/**
 * 命令对象，绿豆排骨煲
 */
public class ChopCommand implements Command{
        /**
         * 持有具体做菜的厨师的对象
         */
        private CookApi cookApi = null;
        /**
         * 设置具体做菜的厨师的对象
```

```
         * @param cookApi 具体做菜的厨师的对象
         */
        public void setCookApi(CookApi cookApi) {
                this.cookApi = cookApi;
        }
        /**
         * 点菜的桌号
         */
        private int tableNum;
        /**
         * 构造方法，传入点菜的桌号
         * @param tableNum 点菜的桌号
         */
        public ChopCommand(int tableNum){
                this.tableNum = tableNum;
        }
        public int getTableNum(){
                return this.tableNum;
        }
        public void execute() {
                this.cookApi.cook(tableNum,"绿豆排骨煲");
        }
}
```

还有一个命令对象是“北京烤鸭 “，跟上面实现一样，只是菜名不同而已，所以就不去展示示例代码了。

（4）接下来构建很重要的命令对象的队列，其实也不是有多难，多个命令对象嘛，用个集合来存储就好了，然后按照放入的顺序，先进先出即可。

请注意：为了演示的简单性，这里没有使用java.util.Queue，直接使用List来模拟实现了。

示例代码如下：

```
/**
 * 命令队列类
 */
public class CommandQueue {
        /**
```

```
* 用来存储命令对象的队列
*/
        private static List<Command> cmds = new ArrayList<Command>();
        /**
         * 服务员传过来一个新的菜单，需要同步，
         * 因为同时会有很多的服务员传入菜单，而同时又有很多厨师在从队列里取值
         * @param menu 传入的菜单
         */
        public  synchronized  static void addMenu(MenuCommand menu){
                //一个菜单对象包含很多命令对象
                for(Command cmd : menu.getCommands()){
                        cmds.add(cmd);
                }
        }
        /**
         * 厨师从命令队列里面获取命令对象进行处理，也是需要同步的
         */
        public   synchronized   static Command getOneCommand(){
                Command cmd = null;
                if(cmds.size() > 0 ){
                        //取出队列的第一个，因为是约定的按照加入的先后来处理
                        cmd = cmds.get(0);
                        //同时从队列里面取掉这个命令对象
                        cmds.remove(0);
                }
                return cmd;
        }
}
```

提示：这里并没有考虑一些复杂的情况，比如：如果命令队列里面没有命令，而厨师又来获取命令怎么办？这里只是做一个基本的示范，并不是完整的实现，所以这里就没有去处理这些问题了，当然出现这种问题，就需要使用wait/notify来进行线程调度了。

（5）有了命令队列，谁来向这个队列里面传入命令呢？

很明显是服务员，当客户点菜完成，服务员就会执行菜单，现在执行菜单就相当于把菜单直接传递给后厨，也就是要把菜单里的所有命令对象加入到命令队列里面。因此菜单对象的实现需要改变，示例代码如下：

```
/**
 * 菜单对象，是个宏命令对象
 */
public class MenuCommand implements Command {
        /**
          * 用来记录组合本菜单的多道菜品，也就是多个命令对象
*/
        private Collection<Command> col = new ArrayList<Command>();
        /**
         * 点菜，把菜品加入到菜单中
         * @param cmd 客户点的菜
         */
        public void addCommand(Command cmd){
                col.add(cmd);
        }
        public void setCookApi(CookApi cookApi){
                //什么都不用做
        }
        public int getTableNum(){
                //什么都不用做
                return 0;
        }
        /**
         * 获取菜单中的多个命令对象
         * @return 菜单中的多个命令对象
         */
        public Collection<Command> getCommands(){
                return this.col;
        }

        public void execute() {
                //执行菜单就是把菜单传递给后厨
                CommandQueue.addMenu(this);
        }
}
```

（6）现在有了命令队列，也有人负责向队列里面添加命令了，可是谁来执行命令队列里面的命令呢？

答案是：由厨师从命令队列里面获取命令，并真正处理命令，而且厨师在处理命令前会把自己设置到命令对象里面去当接收者，表示这个菜由我来实际做。

厨师对象的实现，大致有如下的改变：

- 为了更好的体现命令队列的用法，再说实际情况也是多个厨师，这里用多线程来模拟多个厨师，他们自己从命令队列里面获取命令，然后处理命令，然后再获取下一个，如此反复，因此厨师类要实现多线程接口。
- 还有一个改变，为了在多线程中输出信息，让我们知道是哪一个厨师在执行命令，给厨师添加了一个姓名的属性，通过构造方法传入。
- 另外一个改变是为了在多线程中看出效果，在厨师真正做菜的方法里面使用随机数模拟了一个做菜的时间。

好了，介绍完了改变的地方，一起看看代码吧，示例代码如下：

```
/**
 * 厨师对象，做热菜的厨师
 */
public class HotCook implements CookApi,Runnable{
        /**
         * 厨师姓名
         */
        private String name;
        /**
         * 构造方法，传入厨师姓名
         * @param name 厨师姓名
         */
        public HotCook(String name){
                this.name = name;
        }
        public void cook(int tableNum,String name) {
                //每次做菜的时间是不一定的，用个随机数来模拟一下
                int cookTime = (int)(20 * Math.random());
                System.out.println(this.name+"厨师正在为"+tableNum
+"号桌做："+name);
                try {
                        //让线程休息这么长时间，表示正在做菜
                        Thread.sleep(cookTime);
```

```
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
                System.out.println(this.name+"厨师为"+tableNum
+"号桌做好了："+name+",共计耗时="+cookTime+"秒");
        }
        public void run() {
                while(true){
                        //到命令队列里面获取命令对象
                        Command cmd = CommandQueue.getOneCommand();
                        if(cmd != null){
                                //说明取到命令对象了，这个命令对象还没有设置接收者
                                //因为前面都还不知道到底哪一个厨师来真正执行这个命令
                                //现在知道了，就是当前厨师实例，设置到命令对象里面
                                cmd.setCookApi(this);
                                //然后真正执行这个命令
                                cmd.execute();
                        }
                        //休息1秒
                        try {
                                Thread.sleep(1000L);
                        } catch (InterruptedException e) {
                                e.printStackTrace();
                        }
                }
        }
}
```

（7）该来看看服务员类了，由于现在考虑了后厨的管理，因此从实际来看，这次服务员也不知道到底命令的真正接收者是谁了，也就是说服务员也不知道某个菜到底最后由哪一位厨师完成，所以服务员类就简单了。

组装命令对象和接收者的功能后移到厨师类的线程里面了，当某个厨师从命令队列里面获取一个命令对象的时候，这个厨师就是这个命令的真正接收者。

看看服务员类的示例代码如下：

```
/**
 * 服务员，负责组合菜单,还负责执行调用
 */
public class Waiter {
        /**
         * 持有一个宏命令对象——菜单
         */
        private MenuCommand menuCommand = new MenuCommand();
        /**
         * 客户点菜
         * @param cmd 客户点的菜，每道菜是一个命令对象
         */
        public void orderDish(Command cmd){
                //添加到菜单中
                menuCommand.addCommand(cmd);
        }
        /**
         * 客户点菜完毕，表示要执行命令了，这里就是执行菜单这个组合命令
         */
        public void orderOver(){
                this.menuCommand.execute();
        }
}
```

（8）在见到曙光之前，还有一个问题要解决，就是谁来启动多线程的厨师呢？

为了实现后厨的管理，为此专门定义一个后厨管理的类，在这个类里面去启动多个厨师的线程。而且这种启动在运行期间应该只有一次。示例代码如下：

```
/**
 * 后厨的管理类，通过此类让后厨的厨师进行运行状态
 */
public class CookManager {
        /**
         * 用来控制是否需要创建厨师，如果已经创建过了就不要再执行了
         */
        private static boolean runFlag = false;
        /**
```

```
         * 运行厨师管理，创建厨师对象并启动他们相应的线程，
         * 无论运行多少次，创建厨师对象和启动线程的工作就只做一次
         */
        public static void runCookManager(){
                if(!runFlag){
                        runFlag = true;
                        //创建三位厨师
                        HotCook cook1 = new HotCook("张三");
                        HotCook cook2 = new HotCook("李四");
                        HotCook cook3 = new HotCook("王五");

                        //启动他们的线程
                        Thread t1 = new Thread(cook1);
                        t1.start();
                        Thread t2 = new Thread(cook2);
                        t2.start();
                        Thread t3 = new Thread(cook3);
                        t3.start();
                }
        }
}
```

（9）曙光来临了，写个客户端测试测试，示例代码如下：

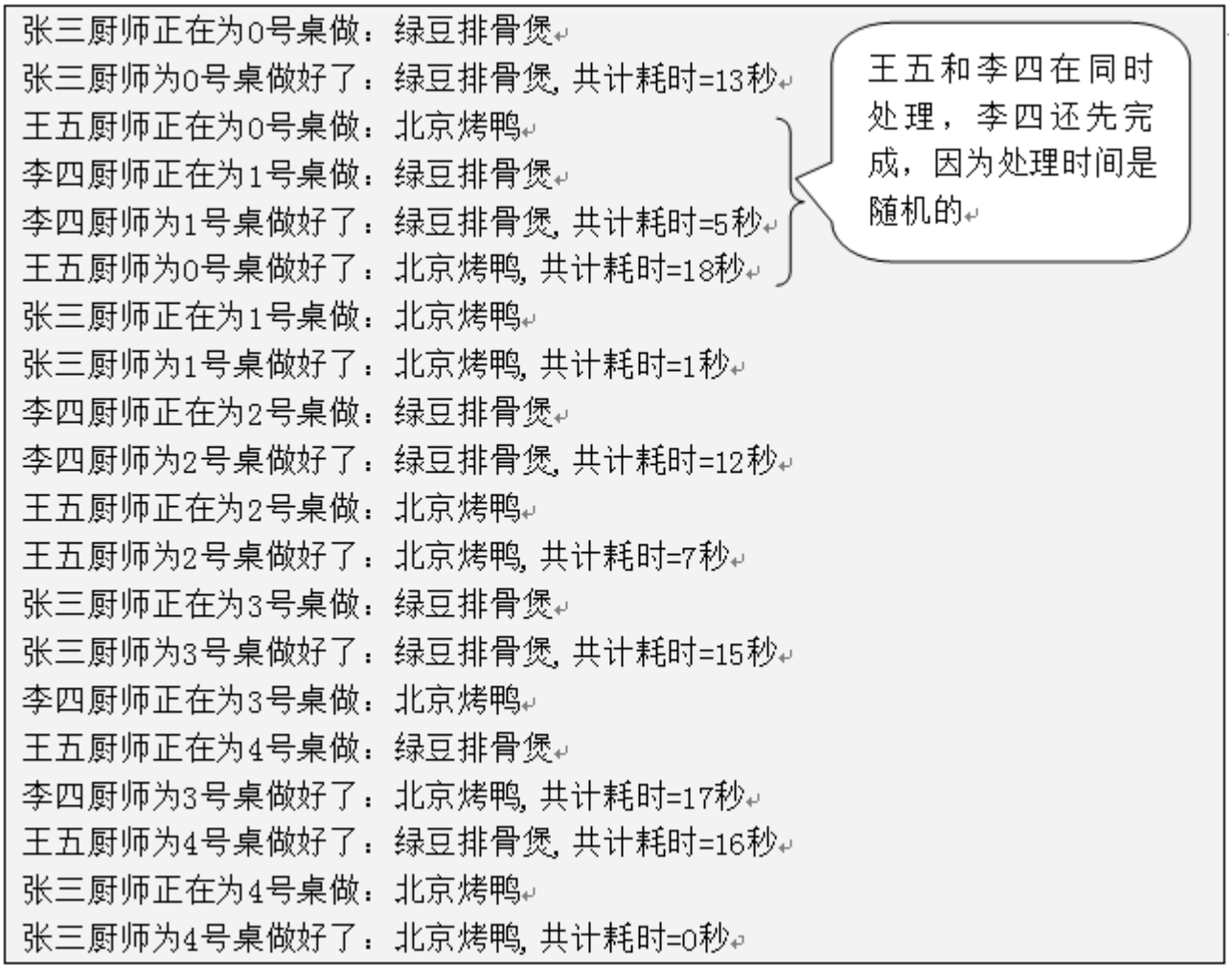
```
public class Client {
        public static void main(String[] args) {
                //先要启动后台，让整个程序运行起来
                CookManager.runCookManager();

                //为了简单，直接用循环模拟多个桌号点菜
                for(int i = 0;i<5;i++){
                        //创建服务员
                        Waiter waiter = new Waiter();
                        //创建命令对象，就是要点的菜
                        Command chop = new ChopCommand(i);
                        Command duck = new DuckCommand(i);
```

```
                    //点菜，就是把这些菜让服务员记录下来
                    waiter.orderDish(chop);
                    waiter.orderDish(duck);

                    //点菜完毕
                    waiter.orderOver();
                }
        }
}
```

（10）运行一下，看看效果，可能每次运行的效果不一样，毕竟是使用多线程在处理请求队列，某次运行的结果如下：



好好观察上面的数据，在多线程环境下，虽然保障了命令对象取出的顺序是先进先出，但是究竟是哪一个厨师来做，还有具体做多长时间都是不定的。

PS: 有朋友发信息要联系方式，好多多交流，这里公布一下我的QQ号，有需要的朋友可以加入：1500562586

未完待续.....

1.21 研磨设计模式之命令模式-6

发表时间: 2010-07-23

3.7 命令模式的优缺点

- 更松散的耦合

  命令模式使得发起命令的对象——客户端，和具体实现命令的对象——接收者对象完全解耦，也就是说发起命令的对象，完全不知道具体实现对象是谁，也不知道如何实现。
- 更动态的控制

  命令模式把请求封装起来，可以动态对它进行参数化、队列化和日志化等操作，从而使得系统更灵活。
- 能很自然的复合命令

  命令模式中的命令对象，能够很容易的组合成为复合命令，就是前面讲的宏命令，从而使系统操作更简单，功能更强大。
- 更好的扩展性

  由于发起命令的对象和具体的实现完全解耦，因此扩展新的命令就很容易，只需要实现新的命令对象，然后在装配的时候，把具体的实现对象设置到命令对象里面，然后就可以使用这个命令对象，已有的实现完全不用变化。

3.8 思考命令模式

1：命令模式的本质

命令模式的本质：**封装请求。**

前面讲了，命令模式的关键就是把请求封装成为命令对象，然后就可以对这个对象进行一系列的处理了，比如上面讲到的参数化配置、可撤销操作、宏命令、队列请求、日志请求等功能处理。

2：何时选用命令模式

建议在如下情况中，选用命令模式：

- 如果需要抽象出需要执行的动作，并参数化这些对象，可以选用命令模式，把这些需要执行的动作抽象成为命令，然后实现命令的参数化配置
- 如果需要在不同的时刻指定、排列和执行请求，可以选用命令模式，把这些请求封装成为命令对象，然后实现把请求队列化
- 如果需要支持取消操作，可以选用命令模式，通过管理命令对象，能很容易的实现命令的恢复和重做的功能

- 如果需要支持当系统崩溃时，能把对系统的操作功能重新执行一遍，可以选用命令模式，把这些操作功能的请求封装成命令对象，然后实现日志命令，就可以在系统恢复回来后，通过日志获取命令列表，从而重新执行一遍功能
- 在需要事务的系统中，可以选用命令模式，命令模式提供了对事务进行建模的方法，命令模式有一个别名就是Transaction。

3.9 退化的命令模式

在领会了命令模式本质后，来思考一个命令模式退化的情况。

前面讲到了智能命令，如果命令的实现对象超级智能，实现了命令所要求的功能，那么就不需要接收者了，既然没有了接收者，那么也就不需要组装者了。

（1）举个最简单的示例来说明

比如现在要实现一个打印服务，由于非常简单，所以基本上就没有什么讲述，依次来看，命令接口定义如下：

```
public interface Command {
        public void execute();
}
```

命令的实现示例代码如下：

```
public class PrintService implements Command{
        /**
         * 要输出的内容
         */
        private String str = "";
        /**
         * 构造方法，传入要输出的内容
         * @param s 要输出的内容
         */
        public PrintService(String s){
                str = s;
        }
        public void execute() {
```

```
                                //智能的体现，自己知道怎么实现命令所要求的功能，并真的实现了相应的功
                System.out.println("打印的内容为="+str);
        }
}
```

此时的Invoker示例代码如下：

```
public class Invoker {
        /**
         * 持有命令对象
         */
        private Command cmd = null;
        /**
         * 设置命令对象
         * @param cmd 命令对象
         */
        public void setCmd(Command cmd){
                this.cmd = cmd;
        }
        /**
         * 开始打印
         */
        public void startPrint(){
                //执行命令的功能
                this.cmd.execute();
        }
}
```

最后看看客户端的代码，示例如下：

```
public class Client {
        public static void main(String[] args) {
                //准备要发出的命令
                Command cmd = new PrintService("退化的命令模式示例");
```

```
                //设置命令给持有者
                Invoker invoker = new Invoker();
                invoker.setCmd(cmd);

                //按下按钮，真正启动执行命令
                invoker.startPrint();
        }
}
```

测试结果如下：

```
打印的内容为=退化的命令模式示例
```

（2）继续变化

如果此时继续变化，Invoker也开始变得智能化，在Invoker的startPrint方法里面，Invoker加入了一些实现，同时Invoker对持有命令也有意见，觉得自己是个傀儡，要求改变一下，直接在调用方法的时候传递命令对象进来，示例代码如下：

```
public class Invoker {
        public void startPrint(Command cmd){
                System.out.println("在Invoker中，输出服务前");
                cmd.execute();
                System.out.println("输出服务结束");
        }
}
```

看起来Invoker退化成一个方法了。

这个时候Invoker很高兴，宣称自己是一个智能的服务，不再是一个傻傻的转调者，而是有自己功能的服务了。这个时候Invoker调用命令对象的执行方法，也不叫转调，改名叫“回调”，意思是在我Invoker需要的时候，会回调你命令对象，命令对象你就乖乖的写好实现，等我“回调”你就可以了。

事实上这个时候的命令模式的实现，基本上就等同于Java回调机制的实现，可能有些朋友看起来感觉还不是很像，那是因为在Java回调机制的常见实现上，经常没有单独的接口实现类，而是采用匿名内部类的方式来实现的。

（3）再进一步

把单独实现命令接口的类改成用匿名内部类实现，这个时候就只剩下命令的接口、Invoker类，还有客户端了。

为了使用匿名内部类，还要设置要输出的值，对命令接口做点小改动，增加一个设置输出值的方法，示例代码如下：

```
public interface Command {
        public void execute();
        /**
         * 设置要输出的内容
         * @param s 要输出的内容
         */
        public void setStr(String s);
}
```

此时Invoker就是上面那个，而客户端会有些改变，客户端的示例代码如下：

```
public class Client {
        public static void main(String[] args) {
                //准备要发出的命令，没有具体实现类了
                    //匿名内部类来实现命令
                Command cmd = new Command(){
                        private String str = "";
                        public void setStr(String s){
                                str = s;
                        }
                        public void execute() {
                                System.out.println("打印的内容为="+str);
                        }
                };
                cmd.setStr("退化的命令模式类似于Java回调的示例");
                //这个时候的Invoker或许该称为服务了
                Invoker invoker = new Invoker();
                //按下按钮，真正启动执行命令
                invoker.startPrint(cmd);
```

```
        }
}
```

运行测试一下，结果如下：

```
在Invoker中，输出服务前
打印的内容为=退化的命令模式类似于Java回调的示例
输出服务结束
```

（4）现在是不是看出来了，这个时候的命令模式的实现，基本上就等同于Java回调机制的实现。这也是很多人大谈特谈命令模式可以实现Java回调的意思。

当然更狠的是连Invoker也不要了，直接把那个方法搬到Client中，那样测试起来就更方便了。在实际开发中，应用命令模式来实现回调机制的时候，Invoker通常还是有的，但可以智能化实现，更准确的说Invoker充当客户调用的服务实现，而回调的方法只是实现服务功能中的一个或者几个步骤。

3.10 相关模式

- 命令模式和组合模式

  这两个模式可以组合使用。

  在命令模式中，实现宏命令的功能，就可以使用组合模式来实现。前面的示例并没有按照组合模式来做，那是为了保持示例的简单，还有突出命令模式的实现，这点请注意。
- 命令模式和备忘录模式

  这两个模式可以组合使用。

  在命令模式中，实现可撤销操作功能时，前面讲了有两种实现方式，其中有一种就是保存命令执行前的状态，撤销的时候就把状态恢复回去。如果采用这种方式实现，就可以考虑使用备忘录模式。

  如果状态存储在命令对象里面，那么还可以使用原型模式，把命令对象当作原型来克隆一个新的对象，然后把克隆出来的对象通过备忘录模式存放。
- 命令模式和模板方法模式

  这两个模式从某种意义上有相似的功能，命令模式可以作为模板方法的一种替代模式，也就是说命令模式可以模仿实现模板方法模式的功能。

  如同前面讲述的退化的命令模式可以实现Java的回调，而Invoker智能化后向服务进化，如果Invoker的方法就是一个算法骨架，其中有两步在这个骨架里面没有具体实现，需要外部来实现，这个时候就可以通过回调命令接口来实现。

  而类似的功能在模板方法里面，一个算法骨架，其中有两步在这个骨架里面没有具体实现，是先调用

抽象方法，然后等待子类来实现。

可以看出虽然实现方式不一样，但是可以实现相同的功能。

命令模式结束，谢谢您的捧场，鞠躬ing！！！

1.22 研磨设计模式之桥接模式-1

发表时间: 2010-08-16

来写一个大家既陌生又熟悉的设计模式，也是非常实用的一个设计模式，那就是桥接模式。

说陌生是很多朋友并不熟悉这个设计模式，说熟悉是很多人经常见到或者是下意识的用到这个设计模式，只是不知道罢了。桥接模式是非常实用的一个模式，下面就来写写它。

桥接模式（Bridge）

1 场景问题

1.1 发送提示消息

考虑这样一个实际的业务功能：发送提示消息。基本上所有带业务流程处理的系统都会有这样的功能，比如某人有新的工作了，需要发送一条消息提示他。

从业务上看，消息又分成普通消息、加急消息和特急消息多种，不同的消息类型，业务功能处理是不一样的，比如加急消息是在消息上添加加急，而特急消息除了添加特急外，还会做一条催促的记录，多久不完成会继续催促。从发送消息的手段上看，又有系统内短消息、手机短消息、邮件等等。

现在要实现这样的发送提示消息的功能，该如何实现呢？

1.2 不用模式的解决方案

1：实现简化版本

先考虑实现一个简单点的版本，比如：消息先只是实现发送普通消息，发送的方式呢，先实现系统内短消息和邮件。其它的功能，等这个版本完成过后，再继续添加，这样先把问题简单化，实现起来会容易一点。

（1）由于发送普通消息会有两种不同的实现方式，为了让外部能统一操作，因此，把消息设计成接口，然后由两个不同的实现类，分别实现系统内短消息方式和邮件发送消息的方式。此时系统结构如图1所示：

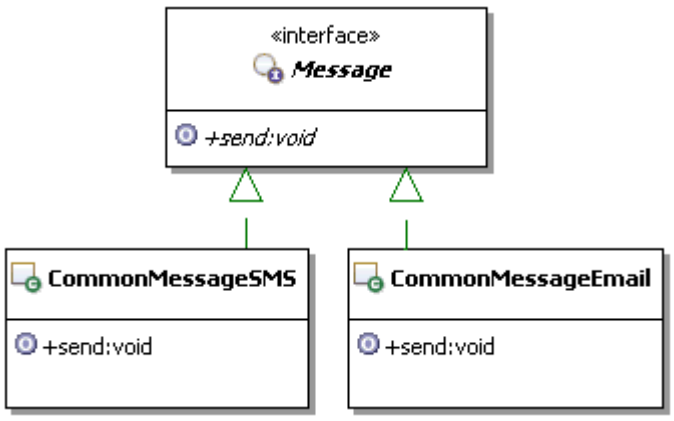


图1 简化版本的系统结构示意图

下面看看大致的实现示意。

（2）先来看看消息的统一接口，示例代码如下：

```
/**
 * 消息的统一接口
 */
public interface Message {
        /**
         * 发送消息
         * @param message 要发送的消息内容
         * @param toUser 消息发送的目的人员
         */
        public void send(String message,String toUser);
}
```

（3）再来分别看看两种实现方式，这里只是为了示意，并不会真的去发送Email和站内短消息，先看站内短消息的方式，示例代码如下：

```
/**
 * 以站内短消息的方式发送普通消息
 */
public  class CommonMessageSMS implements Message{
        public void send(String message, String toUser) {
```

```
                System.out.println("使用站内短消息的方式，发送消息'"
+message+"'给"+toUser);
        }
}
```

同样的，实现以Email的方式发送普通消息，示例代码如下：

```
/**
 * 以Email的方式发送普通消息
 */
public class CommonMessageEmail implements Message{
        public void send(String message, String toUser) {
                System.out.println("使用Email的方式，发送消息'"
+message+"'给"+toUser);
        }
}
```

2：实现发送加急消息

上面的实现，看起来很简单，对不对。接下来，添加发送加急消息的功能，也有两种发送的方式，同样是站内短消息和Email的方式。

加急消息的实现跟普通消息不同，加急消息会自动在消息上添加加急，然后再发送消息；另外加急消息会提供监控的方法，让客户端可以随时通过这个方法来了解对于加急消息处理的进度，比如：相应的人员是否接收到这个信息，相应的工作是否已经开展等等。因此加急消息需要扩展出一个新的接口，除了基本的发送消息的功能，还需要添加监控的功能，这个时候，系统的结构如图2所示：

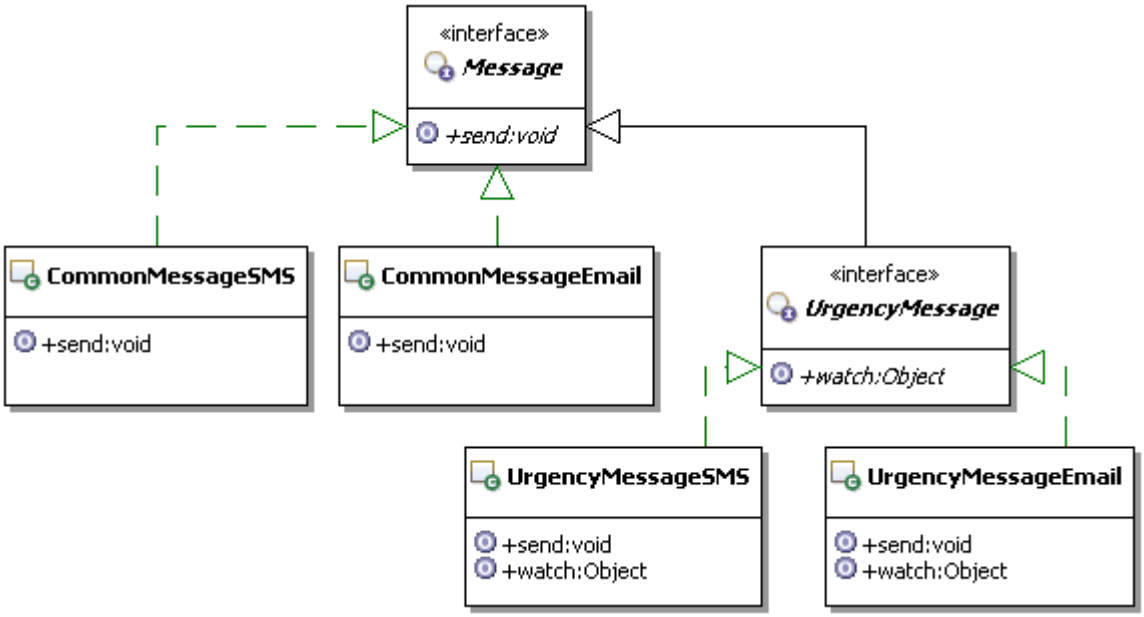


图2 加入发送加急消息后的系统结构示意图

（1）先看看扩展出来的加急消息的接口，示例代码如下：

```
/**
 * 加急消息的抽象接口
 */
public interface UrgencyMessage extends Message{
        /**
         * 监控某消息的处理过程
         * @param messageId 被监控的消息的编号
         * @return 包含监控到的数据对象，这里示意一下，所以用了Object
         */
        public Object watch(String messageId);
}
```

（2）相应的实现方式还是发送站内短消息和Email两种，同样需要两个实现类来分别实现这两种方式，先看站内短消息的方式，示例代码如下：

```
public class UrgencyMessageSMS implements UrgencyMessage{
        public void send(String message, String toUser) {
                message = "加急："+message;
                System.out.println("使用站内短消息的方式，发送消息'"
```

```
+message+"'给"+toUser);
        }

        public Object watch(String messageId) {
                //获取相应的数据，组织成监控的数据对象，然后返回
                return null;
        }
}
```

再看看Emai的方式，示例代码如下：

```
public class UrgencyMessageEmail implements UrgencyMessage{
        public void send(String message, String toUser) {
                message = "加急："+message;
                System.out.println("使用Email的方式，发送消息'"
+message+"'给"+toUser);
        }
        public Object watch(String messageId) {
                //获取相应的数据，组织成监控的数据对象，然后返回
                return null;
        }
}
```

（3）事实上，在实现加急消息发送的功能上，可能会使用前面发送不同消息的功能，也就是让实现加急消息处理的对象继承普通消息的相应实现，这里为了让结构简单一点，清晰一点，所以没有这样做。

1.3 有何问题

上面这样实现，好像也能满足基本的功能要求，可是这么实现好不好呢？有没有什么问题呢？

咱们继续向下来添加功能实现，为了简洁，就不再去进行代码示意了，通过实现的结构示意图就可以看出实现上的问题。

1：继续添加特急消息的处理

特急消息不需要查看处理进程，只要没有完成，就直接催促，也就是说，对于特急消息，在普通消息的处理基础上，需要添加催促的功能。而特急消息、还有催促的发送方式，相应的实现方式还是发送站内短消息和Email两种，此时系统的结构如图3所示：

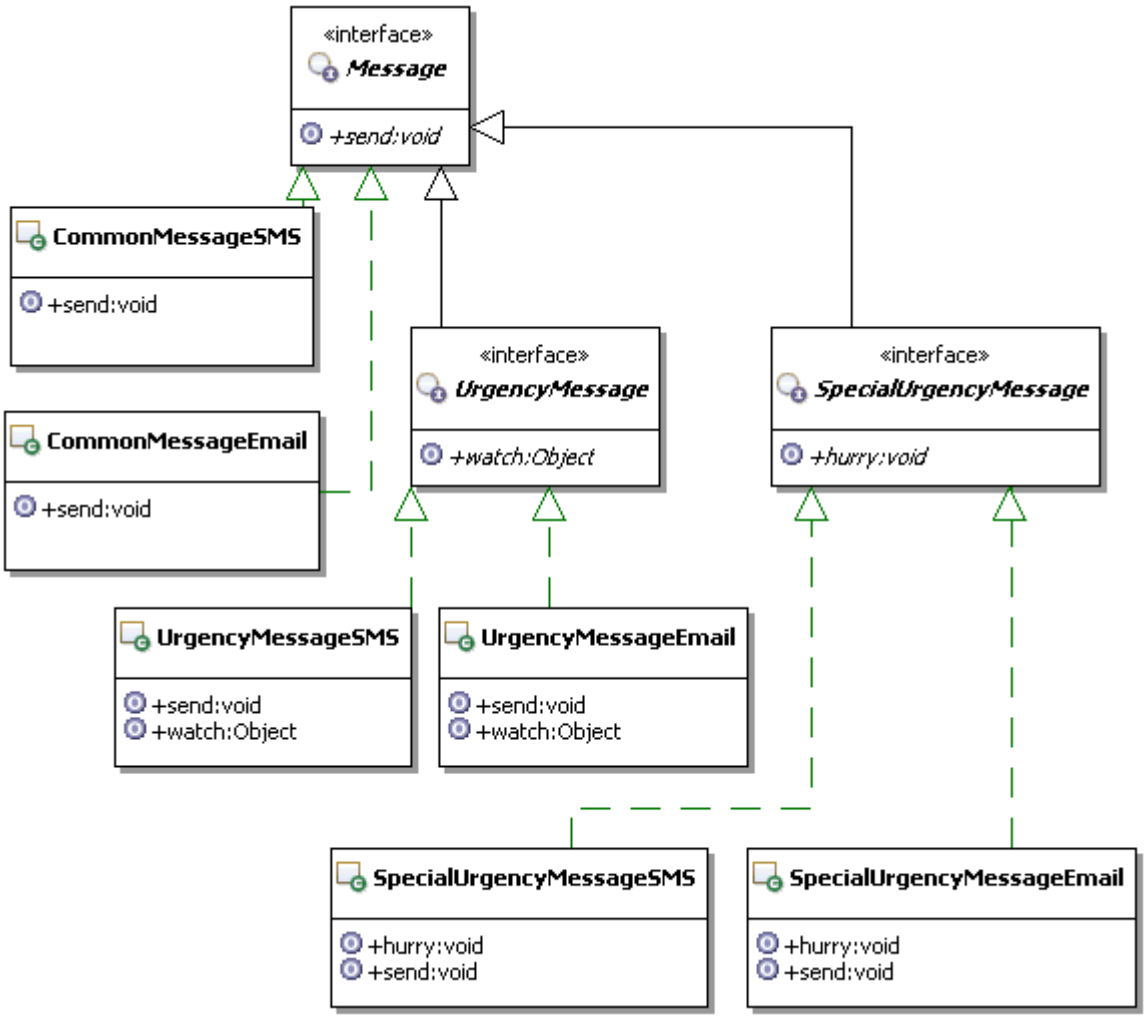


图3 加入发送特急消息后的系统结构示意图

仔细观察上面的系统结构示意图，会发现一个很明显的问题，那就是：通过这种继承的方式来扩展消息处理，会非常不方便。

你看，实现加急消息处理的时候，必须实现站内短消息和Email两种处理方式，因为业务处理可能不同；在实现特急消息处理的时候，又必须实现站内短消息和Email这两种处理方式。

这意味着，以后每次扩展一下消息处理，都必须要实现这两种处理方式，是不是很痛苦，这还不算完，如果要添加新的实现方式呢？继续向下看吧。

2：继续添加发送手机消息的处理方式

如果看到上面的实现，你还感觉问题不是很大的话，继续完成功能，添加发送手机消息的处理方式。

仔细观察现在的实现，如果要添加一种新的发送消息的方式，是需要在每一种抽象的具体实现里面，都要添加发送手机消息的处理的。也就是说：发送普通消息、加急消息和特急消息的处理，都可以通过手机来发送。这就意味着，需要添加三个实现。此时系统结构如图4所示：

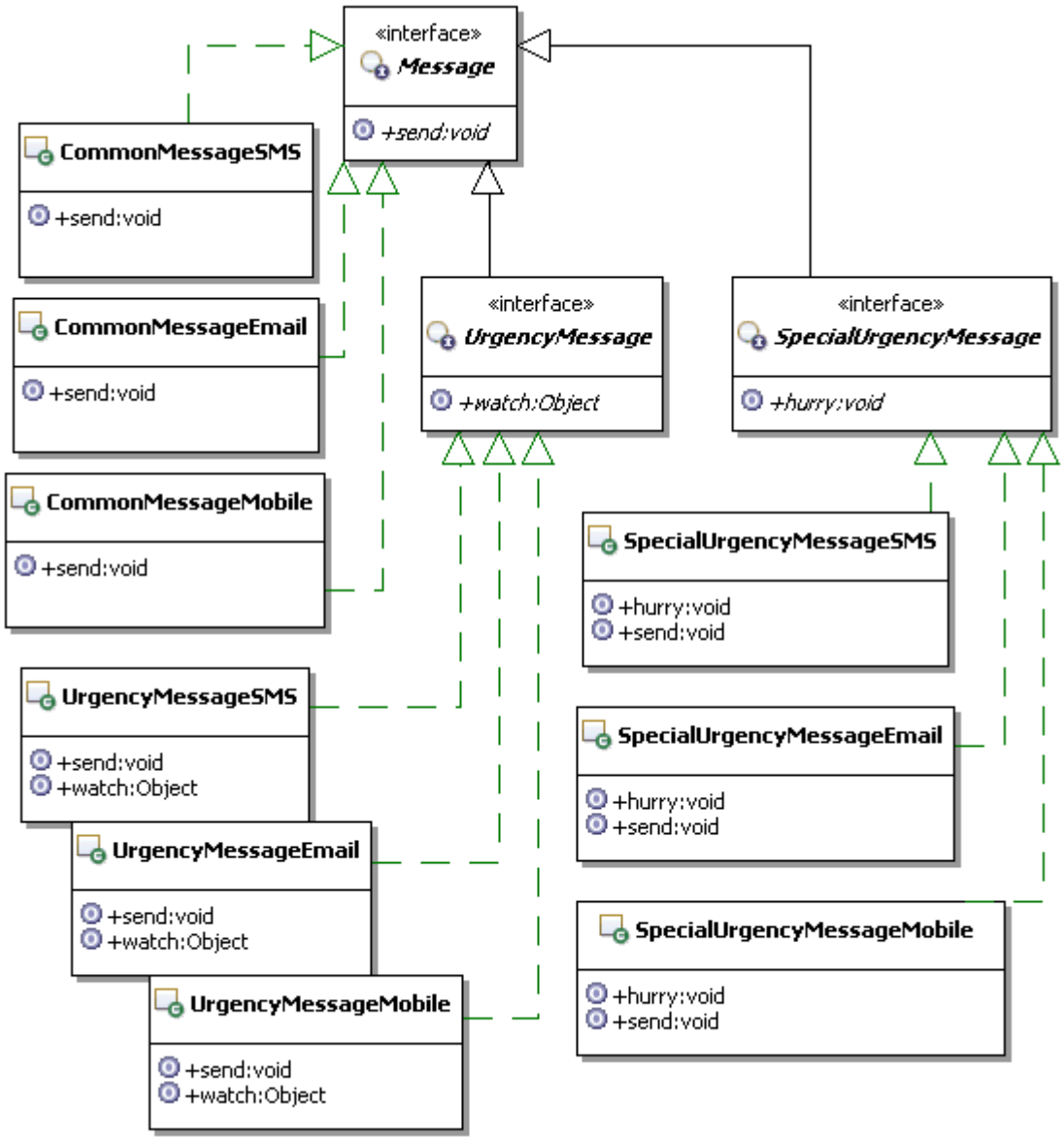


图4 加入发送手机消息后的系统结构示意图

这下能体会到这种实现方式的大问题了吧。

3：小结一下出现的问题

采用通过继承来扩展的实现方式，有个明显的缺点：扩展消息的种类不太容易，不同种类的消息具有不同的业务，也就是有不同的实现，在这种情况下，每个种类的消息，需要实现所有不同的消息发送方式。

更可怕的是，如果要新加入一种消息的发送方式，那么会要求所有的消息种类，都要加入这种新的发送方式的实现。

要是考虑业务功能上再扩展一下呢？比如：要求实现群发消息，也就是一次可以发送多条消息，这就意味着很多地方都得修改，太恐怖了。

那么究竟该如何实现才能既实现功能，又能灵活的扩展呢？

=========未完待续=========

1.23 研磨设计模式之桥接模式-2

发表时间: 2010-08-23

2 解决方案

2.1 桥接模式来解决

用来解决上述问题的一个合理的解决方案，就是使用桥接模式。那么什么是桥接模式呢？

（1）桥接模式定义

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。

（2）应用桥接模式来解决的思路

仔细分析上面的示例，根据示例的功能要求，示例的变化具有两个纬度，一个纬度是抽象的消息这边，包括普通消息、加急消息和特急消息，这几个抽象的消息本身就具有一定的关系，加急消息和特急消息会扩展普通消息；另一个纬度在具体的消息发送方式上，包括站内短消息、Email和手机短信息，这几个方式是平等的，可被切换的方式。这两个纬度一共可以组合出9种不同的可能性来，它们的关系如下图5所示：

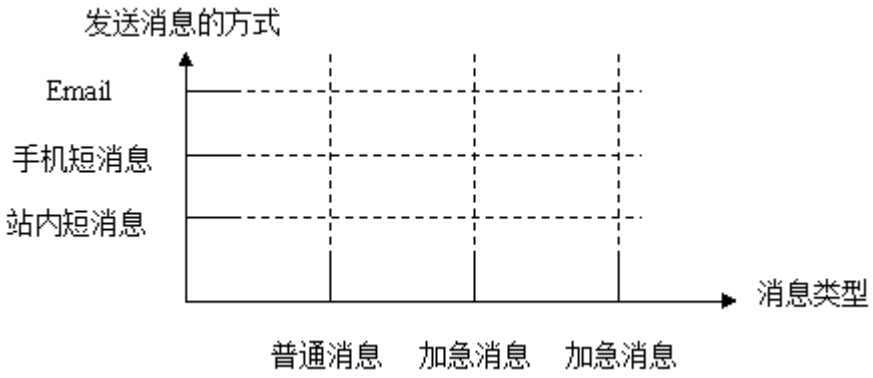


图5 发送消息的可能性的组合示意图

现在出现问题的根本原因，就在于消息的抽象和实现是混杂在一起的，这就导致了，一个纬度的变化，会引起另一个纬度进行相应的变化，从而使得程序扩展起来非常困难。

要想解决这个问题，就必须把这两个纬度分开，也就是将抽象部分和实现部分分开，让它们相互独立，这样就可以实现独立的变化，使扩展变得简单。

桥接模式通过引入实现的接口，把实现部分从系统中分离出去；那么，抽象这边如何使用具体的实现呢？肯定是面向实现的接口来编程了，为了让抽象这边能够很方便的与实现结合起来，把顶层的抽象接口改成抽象类，在里面持有一个具体的实现部分的实例。

这样一来，对于需要发送消息的客户端而言，就只需要创建相应的消息对象，然后调用这个消息对象的方

法就可以了，这个消息对象会调用持有的真正的消息发送方式来把消息发送出去。也就是说客户端只是想要发送消息而已，并不想关心具体如何发送。

2.2 模式结构和说明

桥接模式的结构如图6所示：

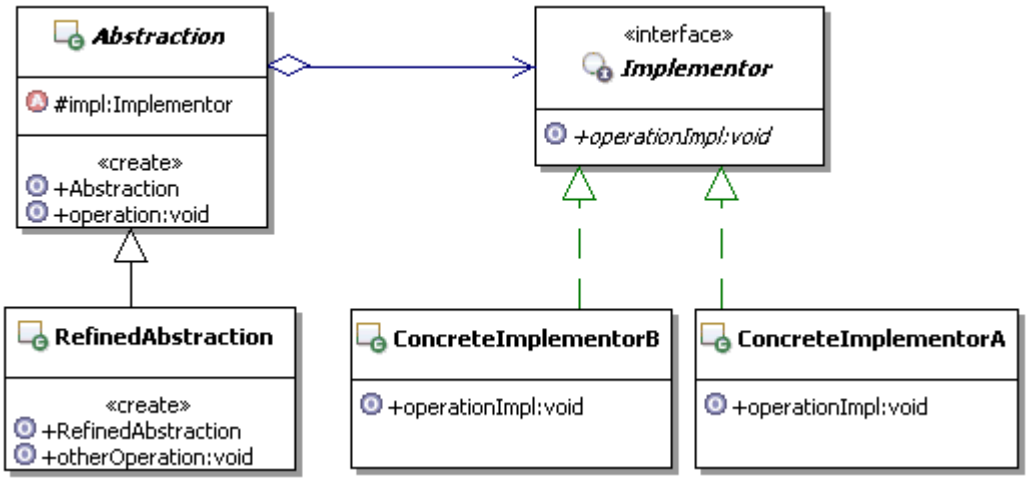


图6 桥接模式的结构示意图

Abstraction：

抽象部分的接口。通常在这个对象里面，要维护一个实现部分的对象引用，在抽象对象里面的方法，需要调用实现部分的对象来完成。这个对象里面的方法，通常都是跟具体的业务相关的方法。

RefinedAbstraction：

扩展抽象部分的接口，通常在这些对象里面，定义跟实际业务相关的方法，这些方法的实现通常会使用Abstraction中定义的方法，也可能需要调用实现部分的对象来完成。

Implementor：

定义实现部分的接口，这个接口不用和Abstraction里面的方法一致，通常是由Implementor接口提供基本的操作，而Abstraction里面定义的是基于这些基本操作的业务方法，也就是说Abstraction定义了基于这些基本操作的较高层次的操作。

ConcreteImplementor：

真正实现Implementor接口的对象。

2.3 桥接模式示例代码

（1）先看看Implementor接口的定义，示例代码如下：

```
/**
 * 定义实现部分的接口，可以与抽象部分接口的方法不一样
 */
public interface Implementor {
        /**
         * 示例方法，实现抽象部分需要的某些具体功能
         */
        public void operationImpl();
}
```

（2）再看看Abstraction接口的定义，注意一点，虽然说是接口定义，但其实是实现成为抽象类。示例代码如下：

```
/**
 * 定义抽象部分的接口
 */
public abstract class Abstraction {
        /**
         * 持有一个实现部分的对象
         */
        protected Implementor impl;
        /**
         * 构造方法，传入实现部分的对象
         * @param impl 实现部分的对象
         */
        public Abstraction(Implementor impl){
                this.impl = impl;
        }
        /**
         * 示例操作，实现一定的功能，可能需要转调实现部分的具体实现方法
         */
        public void operation() {
                impl.operationImpl();
        }
}
```

（3）该来看看具体的实现了，示例代码如下：

```
/**
 * 真正的具体实现对象
 */
public class ConcreteImplementorA implements Implementor {
        public void operationImpl() {
                //真正的实现
        }
}
```

另外一个实现，示例代码如下：

```
/**
 * 真正的具体实现对象
 */
public class ConcreteImplementorB implements Implementor {
        public void operationImpl() {
                //真正的实现
        }
}
```

（4）最后来看看扩展Abstraction接口的对象实现，示例代码如下：

```
/**
 * 扩充由Abstraction定义的接口功能
 */
public class RefinedAbstraction extends Abstraction {
        public RefinedAbstraction(Implementor impl) {
                super(impl);
        }
        /**
         * 示例操作，实现一定的功能
         */
        public void otherOperation(){
                //实现一定的功能，可能会使用具体实现部分的实现方法，
                //但是本方法更大的可能是使用Abstraction中定义的方法，
                //通过组合使用Abstraction中定义的方法来完成更多的功能
```

```
        }
}
```

2.4 使用桥接模式重写示例

学习了桥接模式的基础知识过后，该来使用桥接模式重写前面的示例了。通过示例，来看看使用桥接模式来实现同样的功能，是否能解决“既能方便的实现功能，又能有很好的扩展性”的问题。

要使用桥接模式来重新实现前面的示例，首要任务就是要把抽象部分和实现部分分离出来，分析要实现的功能，抽象部分就是各个消息的类型所对应的功能，而实现部分就是各种发送消息的方式。

其次要按照桥接模式的结构，给抽象部分和实现部分分别定义接口，然后分别实现它们就可以了。

1：从简单功能开始

从相对简单的功能开始，先实现普通消息和加急消息的功能，发送方式先实现站内短消息和Email这两种。

使用桥接模式来实现这些功能的程序结构如图7所示

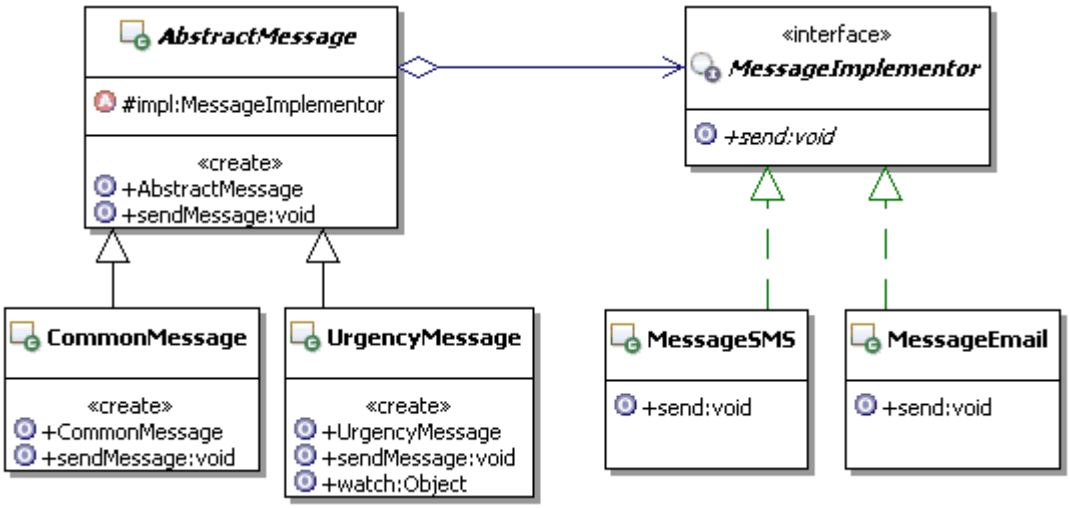


图7 使用桥接模式来实现简单功能示例的程序结构示意图

还是看看代码实现，会更清楚一些。

（1）先看看实现部分定义的接口，示例代码如下：

```
/**
 * 实现发送消息的统一接口
 */
public interface MessageImplementor {
        /**
```

```
         * 发送消息
         * @param message 要发送的消息内容
         * @param toUser 消息发送的目的人员
         */
        public void send(String message,String toUser);
}
```

（2）再看看抽象部分定义的接口，示例代码如下：

```
/**
 * 抽象的消息对象
 */
public abstract class AbstractMessage {
        /**
         * 持有一个实现部分的对象
         */
        protected MessageImplementor impl;
        /**
         * 构造方法，传入实现部分的对象
         * @param impl 实现部分的对象
         */
        public AbstractMessage(MessageImplementor impl){
                this.impl = impl;
        }
        /**
         * 发送消息，转调实现部分的方法
         * @param message 要发送的消息内容
         * @param toUser 消息发送的目的人员
         */
        public void sendMessage(String message,String toUser){
                this.impl.send(message, toUser);
        }
}
```

（3）看看如何具体的实现发送消息，先看站内短消息的实现吧，示例代码如下：

```
/**
 * 以站内短消息的方式发送消息
 */
public  class MessageSMS implements MessageImplementor{
        public void send(String message, String toUser) {
                System.out.println("使用站内短消息的方式，发送消息'"
+message+"'给"+toUser);
        }
}
```

再看看Email方式的实现，示例代码如下：

```
/**
 * 以Email的方式发送消息
 */
public class MessageEmail implements MessageImplementor{
        public void send(String message, String toUser) {
                System.out.println("使用Email的方式，发送消息'"
                                +message+"'给"+toUser);
        }
}
```

（4）接下来该看看如何扩展抽象的消息接口了，先看普通消息的实现，示例代码如下：

```
public class CommonMessage extends AbstractMessage{
        public CommonMessage(MessageImplementor impl) {
                super(impl);
        }
        public void sendMessage(String message, String toUser) {
                //对于普通消息，什么都不干，直接调父类的方法，把消息发送出去就可以了
                super.sendMessage(message, toUser);
        }
}
```

再看看加急消息的实现，示例代码如下：

```
public class UrgencyMessage extends AbstractMessage{
        public UrgencyMessage(MessageImplementor impl) {
```

```
                super(impl);
        }
        public void sendMessage(String message, String toUser) {
                message = "加急："+message;
                super.sendMessage(message, toUser);
        }
        /**
         * 扩展自己的新功能：监控某消息的处理过程
         * @param messageId 被监控的消息的编号
         * @return 包含监控到的数据对象，这里示意一下，所以用了Object
         */
        public Object watch(String messageId) {
                //获取相应的数据，组织成监控的数据对象，然后返回
                return null;
        }
}
```

2：添加功能

看了上面的实现，发现使用桥接模式来实现也不是很困难啊，关键得看是否能解决前面提出的问题，那就来添加还未实现的功能看看，添加对特急消息的处理，同时添加一个使用手机发送消息的方式。该怎么实现呢？

很简单，只需要在抽象部分再添加一个特急消息的类，扩展抽象消息就可以把特急消息的处理功能加入到系统中了；对于添加手机发送消息的方式也很简单，在实现部分新增加一个实现类，实现用手机发送消息的方式，也就可以了。

这么简单？好像看起来完全没有了前面所提到的问题。的确如此，采用桥接模式来实现过后，抽象部分和实现部分分离开了，可以相互独立的变化，而不会相互影响。因此在抽象部分添加新的消息处理，对发送消息的实现部分是没有影响的；反过来增加发送消息的方式，对消息处理部分也是没有影响的。

（1）接着看看代码实现，先看看新的特急消息的处理类，示例代码如下：

```
public class SpecialUrgencyMessage extends AbstractMessage{
        public SpecialUrgencyMessage(MessageImplementor impl) {
                super(impl);
        }
        public void hurry(String messageId) {
```

```
                //执行催促的业务，发出催促的信息
        }
        public void sendMessage(String message, String toUser) {
                message = "特急："+message;
                super.sendMessage(message, toUser);
                //还需要增加一条待催促的信息
        }
}
```

（2）再看看使用手机短消息的方式发送消息的实现，示例代码如下：

```
/**
 * 以手机短消息的方式发送消息
 */
public  class MessageMobile implements MessageImplementor{
        public void send(String message, String toUser) {
                System.out.println("使用手机短消息的方式，发送消息'"
+message+"'给"+toUser);
        }
}
```

3：测试一下功能

看了上面的实现，可能会感觉得到，使用桥接模式来实现前面的示例过后，添加新的消息处理，或者是新的消息发送方式是如此简单，可是这样实现，好用吗？写个客户端来测试和体会一下，示例代码如下：

```
public class Client {
        public static void main(String[] args) {
                //创建具体的实现对象
                MessageImplementor impl = new MessageSMS();
                //创建一个普通消息对象
                AbstractMessage m = new CommonMessage(impl);
                m.sendMessage("请喝一杯茶", "小李");
                //创建一个紧急消息对象
                m = new UrgencyMessage(impl);
                m.sendMessage("请喝一杯茶", "小李");
```

```
                //创建一个特急消息对象
                m = new SpecialUrgencyMessage(impl);
                m.sendMessage("请喝一杯茶", "小李");

                //把实现方式切换成手机短消息，然后再实现一遍
                impl = new MessageMobile();
                m = new CommonMessage(impl);
                m.sendMessage("请喝一杯茶", "小李");
                m = new UrgencyMessage(impl);
                m.sendMessage("请喝一杯茶", "小李");
                m = new SpecialUrgencyMessage(impl);
                m.sendMessage("请喝一杯茶", "小李");
        }
}
```

运行结果如下：

```
使用站内短消息的方式，发送消息'请喝一杯茶'给小李
使用站内短消息的方式，发送消息'加急：请喝一杯茶'给小李
使用站内短消息的方式，发送消息'特急：请喝一杯茶'给小李
使用手机短消息的方式，发送消息'请喝一杯茶'给小李
使用手机短消息的方式，发送消息'加急：请喝一杯茶'给小李
使用手机短消息的方式，发送消息'特急：请喝一杯茶'给小李
```

前面三条是使用的站内短消息，后面三条是使用的手机短消息，正确的实现了预期的功能。看来前面的实现应该是正确的，能够完成功能，且能灵活扩展。

未完待续

1.24 研磨设计模式之桥接模式-3

发表时间: 2010-08-30

3 模式讲解

3.1 认识桥接模式

（1）什么是桥接

在桥接模式里面，不太好理解的就是桥接的概念，什么是桥接？为何需要桥接？如何桥接？把这些问题搞清楚了，也就基本明白桥接的含义了。

一个一个来，先看什么是桥接？所谓桥接，通俗点说就是在不同的东西之间搭一个桥，让他们能够连接起来，可以相互通讯和使用。那么在桥接模式中到底是给什么东西来搭桥呢？就是为被分离了的抽象部分和实现部分来搭桥，比如前面示例中抽象的消息和具体消息发送之间搭个桥。

但是这里要注意一个问题：**在桥接模式中的桥接是单向的**，也就是只能是抽象部分的对象去使用具体实现部分的对象，而不能反过来，也就是个单向桥。

（2）为何需要桥接

为了达到让抽象部分和实现部分都可以独立变化的目的，在桥接模式中，是把抽象部分和实现部分分离开来的，虽然从程序结构上是分开了，但是在抽象部分实现的时候，还是需要使用具体的实现的，这可怎么办呢？抽象部分如何才能调用到具体实现部分的功能呢？很简单，搭个桥不就可以了，搭个桥，让抽象部分通过这个桥就可以调用到实现部分的功能了，因此需要桥接。

（3）如何桥接

这个理解上也很简单，只要让抽象部分拥有实现部分的接口对象，这就桥接上了，在抽象部分就可以通过这个接口来调用具体实现部分的功能。也就是说，桥接在程序上就体现成了在抽象部分拥有实现部分的接口对象，维护桥接就是维护这个关系。

（4）独立变化

桥接模式的意图：使得抽象和实现可以独立变化，都可以分别扩充。也就是说抽象部分和实现部分是一种非常松散的关系，从某个角度来讲，抽象部分和实现部分是可以完全分开的，独立的，抽象部分不过是一个使用实现部分对外接口的程序罢了。

如果这么看桥接模式的话，就类似于策略模式了，抽象部分需要根据某个策略，来选择真实的实现，也就是说桥接模式的抽象部分相当于策略模式的上下文。更原始的就直接类似于面向接口编程，通过接口分离的两个部分而已。但是别忘了，桥接模式的抽象部分，是可以继续扩展和变化的，而策略模式只有上下文，是不存在所谓抽象部分的。

那抽象和实现为何还要组合在一起呢？原因是在抽象部分和实现部分还是存在内部联系的，抽象部分的实现通常是需要调用实现部分的功能来实现的。

（5）动态变换功能

由于桥接模式中的抽象部分和实现部分是完全分离的，因此可以在运行时动态组合具体的真实实现，从而达到动态变换功能的目的。

从另外一个角度看，抽象部分和实现部分没有固定的绑定关系了，因此同一个真实实现可以被不同的抽象对象使用，反过来，同一个抽象也可以有多个不同的实现。就像前面示例的那样，比如：站内短消息的实现功能，可以被普通消息、加急消息或是特急消息等不同的消息对象使用；反过来，某个消息具体的发送方式，可以是站内短消息，或者是Email，也可以是手机短消息等具体的发送方式。

（6）退化的桥接模式

如果Implementor仅有一个实现，那么就没有必要创建Implementor接口了，这是一种桥接模式退化的情况。这个时候Abstraction和Implementor是一对一的关系，虽然如此，也还是要保持它们的分离状态，这样的话，它们才不会相互影响，才可以分别扩展。

也就是说，就算不要Implementor接口了，也要保持Abstraction和Implementor是分离的，模式的分离机制仍然是非常有用的。

（7）桥接模式和继承

继承是扩展对象功能的一种常见手段，通常情况下，继承扩展的功能变化纬度都是一纬的，也就是变化的因素只有一类。

对于出现变化因素有两类的，也就是有两个变化纬度的情况，继承实现就会比较痛苦。比如上面的示例，就有两个变化纬度，一个是消息的类别，不同的消息类别处理不同；另外一个是消息的发送方式。

从理论上来说，如果用继承的方式来实现这种有两个变化纬度的情况，最后实际的实现类应该是两个纬度上可变数量的乘积那么多个。比如上面的示例，在消息类别的纬度上，目前的可变数量是3个，普通消息、加急消息和特急消息；在消息发送方式的纬度上，目前的可变数量也是3个，站内短消息、Email和手机短消息。这种情况下，如果要实现全的话，那么需要的实现类应该是：3 X 3 = 9个。

如果要在任何一个纬度上进行扩展，都需要实现另外一个纬度上的可变数量那么多个实现类，这也是为何会感到扩展起来很困难。而且随着程序规模的加大，会越来越难以扩展和维护。

而桥接模式就是用来解决这种有两个变化纬度的情况下，如何灵活的扩展功能的一个很好的方案。其实，桥接模式主要是把继承改成了使用对象组合，从而把两个纬度分开，让每一个纬度单独去变化，最后通过对象组合的方式，把两个纬度组合起来，每一种组合的方式就相当于原来继承中的一种实现，这样就有效的减少了实际实现的类的个数。

从理论上来说，如果用桥接模式的方式来实现这种有两个变化纬度的情况，最后实际的实现类应该是两个纬度上可变数量的和那么多个。同样是上面那个示例，使用桥接模式来实现，实现全的话，最后需要的实现类的数目应该是：3 + 3 = 6个。

这也从侧面体现了，使用对象组合的方式比继承要来得更灵活。

（8）桥接模式的调用顺序示意图

桥接模式的调用顺序如图8所示：

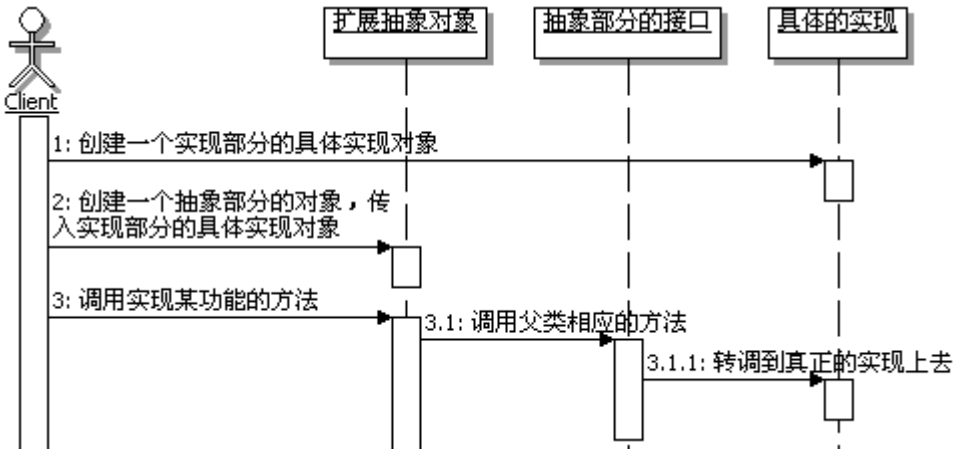


图8 桥接模式的调用顺序示意图

3.2 谁来桥接

所谓谁来桥接，就是谁来负责创建抽象部分和实现部分的关系，说得更直白点，就是谁来负责创建Implementor的对象，并把它设置到抽象部分的对象里面去，这点对于使用桥接模式来说，是十分重要的一点。

大致有如下几种实现方式：

- 由客户端负责创建Implementor的对象，并在创建抽象部分的对象的时候，把它设置到抽象部分的对象里面去，前面的示例采用的就是这个方式
- 可以在抽象部分的对象构建的时候，由抽象部分的对象自己来创建相应的Implementor的对象，当然可以给它传递一些参数，它可以根据参数来选择并创建具体的Implementor的对象
- 可以在Abstraction中选择并创建一个缺省的Implementor的对象，然后子类可以根据需要改变这个实现
- 也可以使用抽象工厂或者简单工厂来选择并创建具体的Implementor的对象，抽象部分的类可以通过调用工厂的方法来获取Implementor的对象
- 如果使用IoC/DI容器的话，还可以通过IoC/DI容器来创建具体的Implementor的对象，并注入回到Abstraction中

下面分别给出后面几种实现方式的示例。

1：由抽象部分的对象自己来创建相应的Implementor的对象

对于这种情况的实现，又分成两种，一种是需要外部传入参数，一种是不需要外部传入参数。

（1）从外面传递参数比较简单，比如前面的示例，如果用一个type来标识具体采用哪种发送消息的方案，然后在Abstraction的构造方法中，根据type进行创建就好了。

还是代码示例一下，主要修改Abstraction的构造方法，示例代码如下：

```
/**
 * 抽象的消息对象
 */
public abstract class AbstractMessage {
        /**
         * 持有一个实现部分的对象
         */
        protected MessageImplementor impl;
        /**
         * 构造方法，传入选择实现部分的类型
         * @param type 传入选择实现部分的类型
         */
        public AbstractMessage(int type){
                if(type==1){
                        this.impl = new MessageSMS();
                }else if(type==2){
                        this.impl = new MessageEmail();
                }else if(type==3){
                        this.impl = new MessageMobile();
                }
        }
        /**
         * 发送消息，转调实现部分的方法
         * @param message 要发送的消息内容
         * @param toUser 把消息发送的目的人员
         */
        public void sendMessage(String message,String toUser){
                this.impl.send(message, toUser);
        }
}
```

（2）对于不需要外部传入参数的情况，那就说明是在Abstraction的实现中，根据具体的参数数据来选择相应的Implementor对象。有可能在Abstraction的构造方法中选，也有可能在具体的方法中选。

比如前面的示例，如果发送的消息长度在100以内采用手机短消息，长度在100-1000采用站内短消息，长度在1000以上采用Email，那么就可以在内部方法中自己判断实现了。

实现中，大致有如下改变：

- 原来protected的MessageImplementor类型的属性，不需要了，去掉
- 提供一个protected的方法来获取要使用的实现部分的对象，在这个方法里面，根据消息的长度来选择合适的实现对象
- 构造方法什么都不用做了，也不需要传入参数
- 在原来使用impl属性的地方，要修改成通过上面那个方法来获取合适的实现对象了，不能直接使用impl属性，否则会没有值

示例代码如下：

```
public abstract class AbstractMessage {
        /**
         * 构造方法
         */
        public AbstractMessage(){
                //现在什么都不做了
        }
        /**
         * 发送消息，转调实现部分的方法
         * @param message 要发送的消息内容
         * @param toUser 把消息发送的目的人员
         */
        public void sendMessage(String message,String toUser){
                this.getImpl(message).send(message, toUser);
        }
/**
         * 根据消息的长度来选择合适的实现
         * @param message 要发送的消息
         * @return 合适的实现对象
         */
        protected MessageImplementor getImpl(String message) {
                MessageImplementor impl = null;
                if(message == null){
```

```
                        //如果没有消息，默认使用站内短消息
                        impl = new MessageSMS();
                }else if(message.length()< 100){
                        //如果消息长度在100以内，使用手机短消息
                        impl = new MessageMobile();
                }else if(message.length()<1000){
                        //如果消息长度在100-1000以内，使用站内短消息
                        impl = new MessageSMS();
                }else{
                        //如果消息长度在1000以上
                        impl = new MessageEmail();
                }
                return impl;
        }
}
```

（3）小结一下

对于由抽象部分的对象自己来创建相应的Implementor的对象的这种情况，不管是否需要外部传入参数，优点是客户使用简单，而且集中控制Implementor对象的创建和切换逻辑；缺点是要求Abstraction知道所有的具体的Implementor实现，并知道如何选择和创建它们，如果今后要扩展Implementor的实现，就要求同时修改Abstraction的实现，这会很不灵活，使扩展不方便。

2：在Abstraction中创建缺省的Implementor对象

对于这种方式，实现比较简单，直接在Abstraction的构造方法中，创建一个缺省的Implementor对象，然后子类根据需要，看是直接使用还是覆盖掉。示例代码如下：

```
public abstract class AbstractMessage {
        protected MessageImplementor impl;
        /**
         * 构造方法
         */
        public AbstractMessage(){
                //创建一个默认的实现
                this.impl = new MessageSMS();
        }
        public void sendMessage(String message,String toUser){
```

```
                this.impl.send(message, toUser);
        }
}
```

这种方式其实还可以使用工厂方法，把创建工作延迟到子类。

3：使用抽象工厂或者是简单工厂

对于这种方式，根据具体的需要来选择，如果是想要创建一系列实现对象，那就使用抽象工厂，如果是创建单个的实现对象，那就使用简单工厂就可以了。

直接在原来创建Implementor对象的地方，直接调用相应的抽象工厂或者是简单工厂，来获取相应的Implementor对象，很简单，这个就不去示例了。

这种方法的优点是Abstraction类不用和任何一个Implementor类直接耦合。

4：使用IoC/DI的方式

对于这种方式，Abstraction的实现就更简单了，只需要实现注入Implementor对象的方法就可以了，其它的Abstraction就不管了。

IoC/DI容器会负责创建Implementor对象，并设置回到Abstraction对象中，使用IoC/DI的方式，并不会改变Abstraction和Implementor的关系，Abstraction同样需要持有相应的Implementor对象，同样会把功能委托给Implementor对象去实现。

3.3 典型例子-JDBC

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。

先简单的回忆一下使用JDBC进行开发的过程，简单的片断代码示例如下：

```
String sql = "具体要操作的sql语句";
                // 1：装载驱动
                Class.forName("驱动的名字");
                // 2：创建连接
                Connection conn = DriverManager.getConnection(
"连接数据库服务的URL", "用户名","密码");

                // 3：创建statement或者是preparedStatement
                PreparedStatement pstmt = conn.prepareStatement(sql);
```

```
            // 4：执行sql，如果是查询，再获取ResultSet
            ResultSet rs = pstmt.executeQuery(sql);

            // 5：循环从ResultSet中把值取出来，封装到数据对象中去
            while (rs.next()) {
                    // 取值示意，按名称取值
                    String uuid = rs.getString("uuid");
                    // 取值示意，按索引取值
                    int age = rs.getInt(2);
            }
            //6：关闭
            rs.close();
            pstmt.close();
            conn.close();
```

从上面的示例可以看出，我们写的应用程序，是面向JDBC的API在开发，这些接口就相当于桥接模式中的抽象部分的接口。那么怎样得到这些API的呢？是通过DriverManager来得到的。此时的系统结构如图9所示：

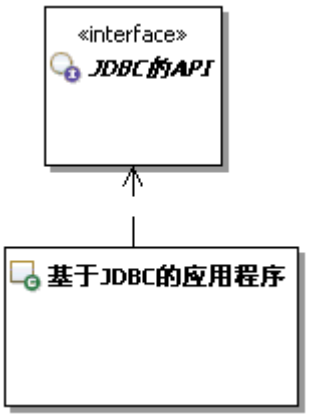


图9 基于JDBC开发的应用程序结构示意图

那么这些JDBC的API，谁去实现呢？光有接口，没有实现也不行啊。

该驱动程序登场了，JDBC的驱动程序实现了JDBC的API，驱动程序就相当于桥接模式中的具体实现部分。而且不同的数据库，由于数据库实现不一样，可执行的Sql也不完全一样，因此对于JDBC驱动的实现也是不一样的，也就是不同的数据库会有不同的驱动实现。此时驱动程序这边的程序结构如图10所示：

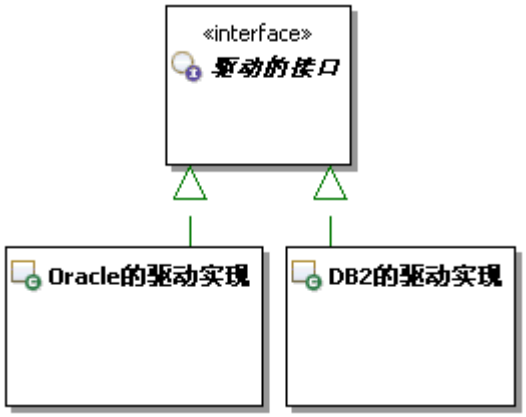


图10 驱动程序实现结构示意图

有了抽象部分——JDBC的API，有了具体实现部分——驱动程序，那么它们如何连接起来呢？就是如何桥接呢？

就是前面提到的DriverManager来把它们桥接起来，从某个侧面来看，DriverManager在这里起到了类似于简单工厂的功能，基于JDBC的应用程序需要使用JDBC的API，如何得到呢？就通过DriverManager来获取相应的对象。

那么此时系统的整体结构如图11所示：

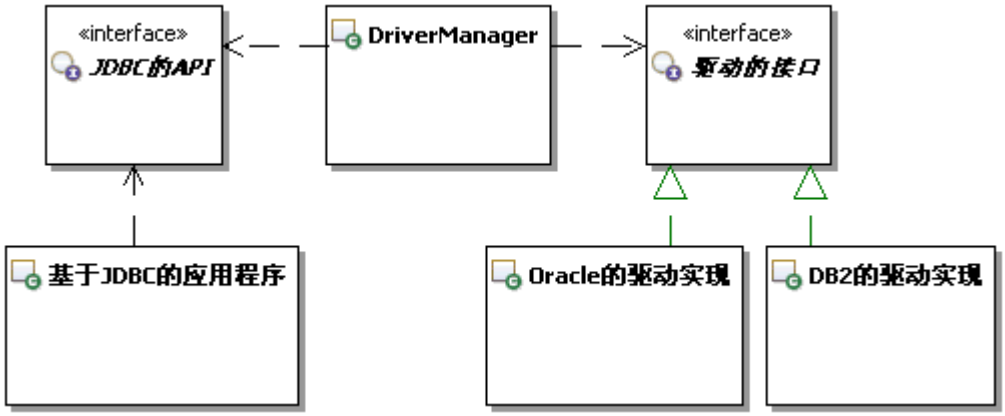


图11 JDBC的结构示意图

通过上图可以看出，基于JDBC的应用程序，使用JDBC的API，相当于是对数据库操作的抽象的扩展，算作桥接模式的抽象部分；而具体的接口实现是由驱动来完成的，驱动这边自然就相当于桥接模式的实现部分了。而桥接的方式，不再是让抽象部分持有实现部分，而是采用了类似于工厂的做法，通过DriverManager来把抽象部分和实现部分对接起来，从而实现抽象部分和实现部分解耦。

JDBC的这种架构，把抽象和具体分离开来，从而使得抽象和具体部分都可以独立扩展。对于应用程序而言，只要选用不同的驱动，就可以让程序操作不同的数据库，而无需更改应用程序，从而实现在不同的数据库上移植；对于驱动程序而言，为数据库实现不同的驱动程序，并不会影响应用程序。而且，JDBC的这种架构，还合理的划分了应用程序开发人员和驱动程序开发人员的边界。

对于有些朋友会认为，从局部来看，体现了策略模式，比如在上面的结构中去掉“JDBC的API和基于JDBC的应用程序”这边，那么剩下的部分，看起来就是一个策略模式的体现。此时的DriverManager就相当于上下文，而各个具体驱动的实现就相当于是具体的策略实现，这个理解也不算错，但是在这里看来，这么理解是比较片面的。

对于这个问题，再次强调一点：对于设计模式，要从整体结构上、从本质目标上、从思想体现上来把握，而不要从局部、从表现、从特例实现上来把握。

未完待续

1.25 研磨设计模式之桥接模式-4

发表时间: 2010-09-06

3.4 广义桥接-Java中无处不桥接

使用Java编写程序，一个很重要的原则就是“面向接口编程”，说得准确点应该是“面向抽象编程”，由于在Java开发中，更多的使用接口而非抽象类，因此通常就说成“面向接口编程”了。

接口把具体的实现和使用接口的客户程序分离开来，从而使得具体的实现和使用接口的客户程序可以分别扩展，而不会相互影响。使用接口的程序结构如图12所示：

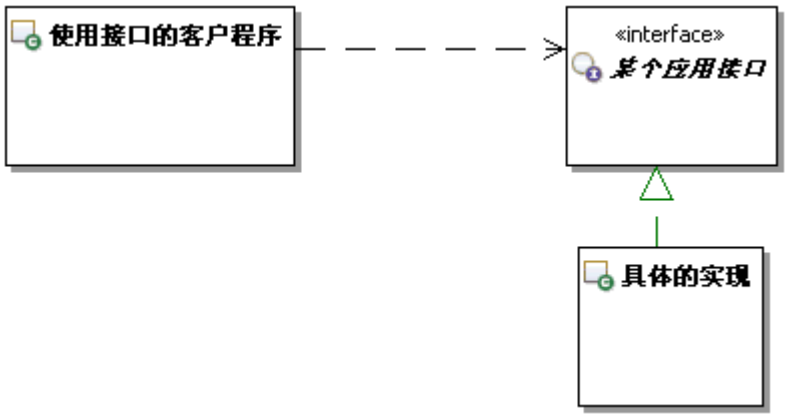


图12 使用接口的程序结构示意图

可能有些朋友会觉得，听起来怎么像是桥接模式的功能呢？没错，如果把桥接模式的抽象部分先稍稍简化一下，暂时不要RefinedAbstraction部分，那么就跟上面的结构图差不多了。去掉RefinedAbstraction后的简化的桥接模式结构示意图如图13所示：

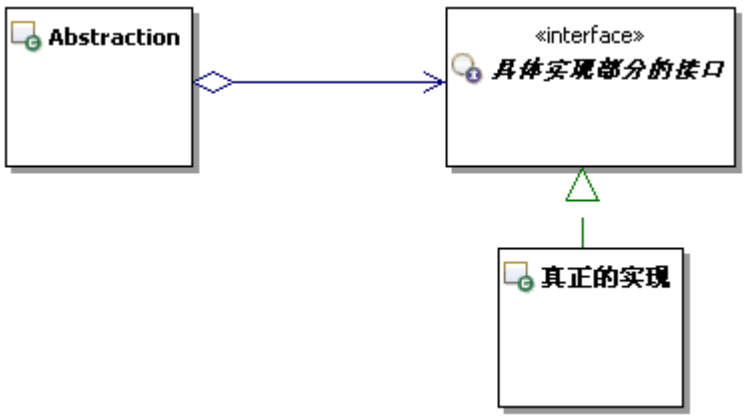


图13 简化的桥接模式结构示意图

是不是差不多呢？有朋友可能会觉得还是有很大差异，差异主要在：前面接口的客户程序是直接使用接口

对象，不像桥接模式的抽象部分那样，是持有具体实现部分的接口，这就导致画出来的结构图，一个是依赖，一个是聚合关联。

请思考它们的本质功能，桥接模式中的抽象部分持有具体实现部分的接口，最终目的是什么，还不是需要通过调用具体实现部分的接口中的方法，来完成一定的功能，这跟直接使用接口没有什么不同，只是表现形式有点不一样。再说，前面那个使用接口的客户程序也可以持有相应的接口对象，这样从形式上就一样了。

也就是说，**从某个角度来讲，桥接模式不过就是对“面向抽象编程”这个设计原则的扩展。**正是通过具体实现的接口，把抽象部分和具体的实现分离开来，抽象部分相当于是使用实现部分接口的客户程序，这样抽象部分和实现部分就松散耦合了，从而可以实现相互独立的变化。

这样一来，几乎可以把所有面向抽象编写的程序，都视作是桥接模式的体现，至少算是简化的桥接模式，就算是广义的桥接吧。而Java编程很强调“面向抽象编程”，因此，广义的桥接，在Java中可以说是无处不在。

再举个大家最熟悉的例子来示例一下。在Java应用开发中，分层实现算是最基本的设计方式了吧，就拿大家最熟的三层架构来说，表现层、逻辑层和数据层，或许有些朋友对它们称呼的名称不同，但都是同一回事情。

三层的基本关系是表现层调用逻辑层，逻辑层调用数据层，通过什么来进行调用呢？当然是接口了，它们的基本结构如图14所示：

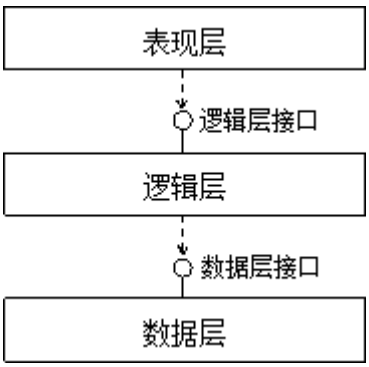


图14 基本的三层架构示意图

通过接口来进行调用，使得表现层和逻辑层分离开来，也就是说表现层的变化，不会影响到逻辑层，同理逻辑层的变化不会影响到表现层。这也是同一套逻辑层和数据层，就能够同时支持不同的表现层实现的原因，比如支持Swing或Web方式的表现层。

在逻辑层和数据层之间也是通过接口来调用，同样使得逻辑层和数据层分离开，使得它们可以独立的扩展。尤其是数据层，可能会有很多的实现方式，比如：数据库实现、文件实现等，就算是数据库实现，又有针对不同数据库的实现，如Oracle、DB2等等。

总之，通过面向抽象编程，三层架构的各层都能够独立的扩展和变化，而不会对其它层次产生影响。这正好是桥接模式的功能，实现抽象和实现的分离，从而使得它们可以独立的变化。当然三层架构不只是在一个地方使用桥接模式，而是至少在两个地方来使用了桥接模式，一个在表现层和逻辑层之间，一个在逻辑层和数据层之间。

下面先分别看看这两个使用桥接模式的地方的程序结构，然后再综合起来看看整体的程序结构。先看看逻

辑层和数据层之间的程序结构，如图15所示：

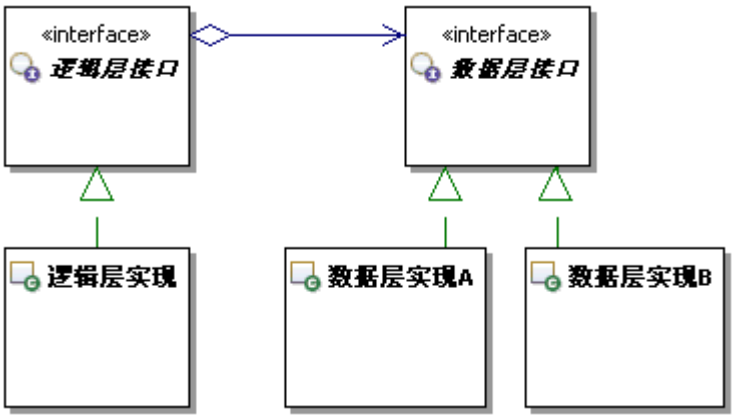


图15 逻辑层和数据层的程序结构示意图

再看看表现层和逻辑层之间的结构示意，如图16所示：

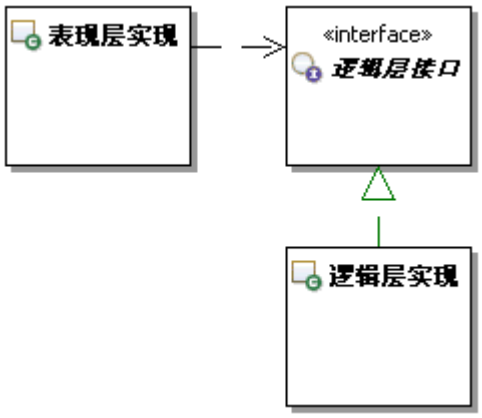


图16 表现层和逻辑层的结构示意图

然后再把它们结合起来，看看结合后的程序结构，如图17所示：

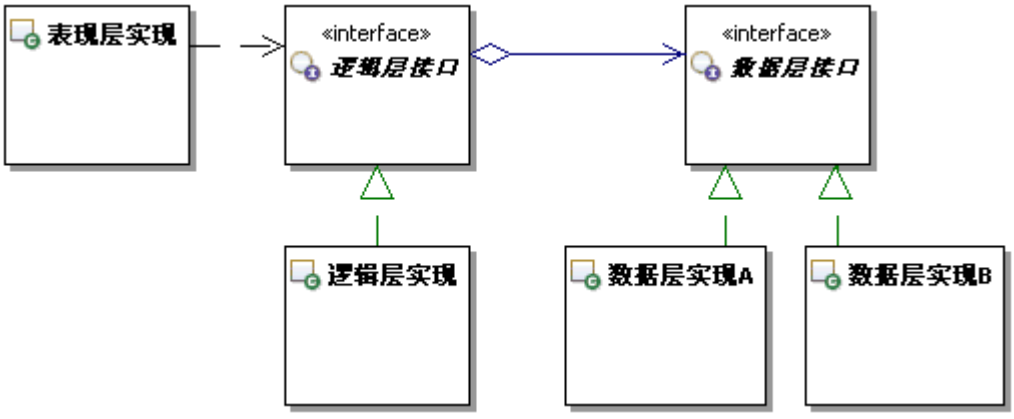


图17 三层结合的结构示意图

从广义桥接模式的角度来看，平日熟悉的三层架构其实就是在组合使用桥接模式。从这个图还可以看出，**桥接模式是可以连续组合使用的，一个桥接模式的实现部分，可以作为下一个桥接模式的抽象部分。**如此类推，可以从三层架构扩展到四层、五层、直到N层架构，都可以使用桥接模式来组合。

如果从更本质的角度来看，基本上只要是面向抽象编写的Java程序，都可以视为是桥接模式的应用，都是让抽象和实现相分离，从而使它们能独立的变化。不过只要分离的目的达到了，叫不叫桥接模式就无所谓了。

3.5 桥接模式的优缺点

- 分离抽象和实现部分

  桥接模式分离了抽象和实现部分，从而极大地提高了系统的灵活性。让抽象部分和实现部分独立开来，分别定义接口，这有助于对系统进行分层，从而产生更好的结构化的系统。对于系统的高层部分，只需要知道抽象部分和实现部分的接口就可以了。
- 更好的扩展性

  由于桥接模式把抽象和实现部分分离开了，而且分别定义接口，这就使得抽象部分和实现部分可以分别独立的扩展，而不会相互影响，从而大大的提高了系统的可扩展性。
- 可动态切换实现

  由于桥接模式把抽象和实现部分分离开了，那么在实现桥接的时候，就可以实现动态的选择和使用具体的实现，也就是说一个实现不再是固定的绑定在一个抽象接口上了，可以实现运行期间动态的切换实现。
- 可减少子类的个数

  根据前面的讲述，对于有两个变化纬度的情况，如果采用继承的实现方式，大约需要两个纬度上的可变化数量的乘积个子类；而采用桥接模式来实现的话，大约需要两个纬度上的可变化数量的和个子类。可以明显地减少子类的个数。

3.6 思考桥接模式

1：桥接模式的本质

桥接模式的本质：**分离抽象和实现。**

桥接模式最重要的工作就是分离抽象部分和实现部分，这是解决问题的关键。只有把抽象和实现分离开了，才能够让它们可以独立的变化；只有抽象和实现可以独立的变化，系统才会有更好的可扩展性、可维护性。

至于其它的好处，比如：可以动态切换实现、可以减少子类个数等。都是把抽象部分和实现部分分离过后，带来的，如果不把抽象部分和实现部分分离开，那就一切免谈了。所以综合来说，桥接模式的本质在于“分离抽象和实现”。

2：对设计原则的体现

（1）桥接模式很好的实现了开闭原则。

通常应用桥接模式的地方，抽象部分和实现部分都是可变化的，也就是应用会有两个变化纬度，桥接模式就是找到这两个变化，并分别封装起来，从而合理的实现OCP。

在使用桥接模式的时候，通常情况下，顶层的Abstraction和Implementor是不变的，而具体的Implementor的实现，是可变的，由于Abstraction是通过接口来操作具体的实现，因此具体的Implementor的实现是可以扩展的，根据需要可以有多个具体的实现。

同样的，RefinedAbstraction也是可变的，它继承并扩展Abstraction，通常在RefinedAbstraction的实现里面，会调用Abstraction中的方法，通过组合使用来完成更多的功能，这些功能常常是与具体业务有关系的功能。

桥接模式还很好的体现了：多用对象组合，少用对象继承。

在前面的示例中，如果使用对象继承来扩展功能，不但让对象之间有很强的耦合性，而且会需要很多的子类才能完成相应的功能，前面已经讲述过了，需要两个纬度上的可变化数量的乘积个子类。

而采用对象的组合，松散了对象之间的耦合性，不但使每个对象变得简单和可维护，还大大减少了子类的个数，根据前面的讲述，大约需要两个纬度上的可变化数量的和这么多个子类。

3：何时选用桥接模式

建议在如下情况中，选用桥接模式：

- 如果你不希望在抽象和实现部分采用固定的绑定关系，可以采用桥接模式，来把抽象和实现部分分开，然后在程序运行期间来动态的设置抽象部分需要用到的具体的实现，还可以动态切换具体的实现。
- 如果出现抽象部分和实现部分都应该可以扩展的情况，可以采用桥接模式，让抽象部分和实现部分可以独立的变化，从而可以灵活的进行单独扩展，而不是搅在一起，扩展一边会影响到另一边。
- 如果希望实现部分的修改，不会对客户产生影响，可以采用桥接模式，客户是面向抽象的接口在运行，实现部分的修改，可以独立于抽象部分，也就不会对客户产生影响了，也可以说对客户是透明的。
- 如果采用继承的实现方案，会导致产生很多子类，对于这种情况，可以考虑采用桥接模式，分析功能变化的原因，看看是否能分离成不同的纬度，然后通过桥接模式来分离它们，从而减少子类的数目。

3.7 相关模式

- 桥接模式和策略模式

  这两个模式有很大的相似之处。

  如果把桥接模式的抽象部分简化来看，如果暂时不去扩展Abstraction，也就是去掉RefinedAbstraction。桥接模式简化过后的结构图参见图13。再看策略模式的结构图参见图17.1。会发现，这个时候它们的结构都类似如图18所示：

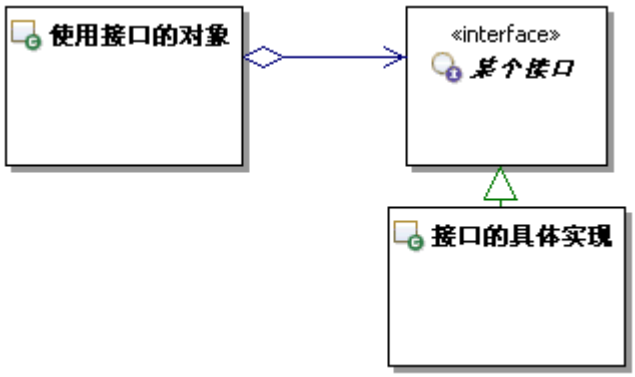


图18 桥接模式和策略模式结构示意图

通过上面的结构图，可以体会到桥接模式和策略模式是如此相似。可以把策略模式的Context视做是使用接口的对象，而Strategy就是某个接口了，具体的策略实现那就相当于接口的具体实现。这样看来的话，某些情况下，可以使用桥接模式来模拟实现策略模式的功能。

这两个模式虽然相似，也还是有区别的。最主要的是模式的目的不一样，策略模式的目的是封装一系列的算法，使得这些算法可以相互替换；而桥接模式的目的是分离抽象和实现部分，使得它们可以独立的变化。

- 桥接模式和状态模式

  由于从模式结构上看，状态模式和策略模式是一样的，这两个模式的关系也基本上类似于桥接模式和策略模式的关系。

  只不过状态模式的目的是封装状态对应的行为，并在内部状态改变的时候改变对象的行为。

- 桥接模式和模板方法模式

  这两个模式有相似之处。

  虽然标准的模板方法模式是采用继承来实现的，但是模板方法也可以通过回调接口的方式来实现，如果把接口的实现独立出去，那就类似于模板方法通过接口去调用具体的实现方法了。这样的结构就和简化的桥接模式类似了。

  可以使用桥接模式来模拟实现模板方法模式的功能。如果在实现Abstraction对象的时候，在里面定义方法，方法里面就是某个固定的算法骨架，也就是说这个方法就相当于模板方法。在模板方法模式里，是把不能确定实现的步骤延迟到子类去实现；现在在桥接模式里面，把不能确定实现的步骤委托给具体实现部分去完成，通过回调实现部分的接口，来完成算法骨架中的某些步骤。这样一来，就可以实现使用桥接模式来模拟实现模板方法模式的功能了。

  使用桥接模式来模拟实现模板方法模式的功能，还有个潜在的好处，就是模板方法也可以很方便的扩展和变化了。在标准的模板方法里面，一个问题就是当模板发生变化的时候，所有的子类都要变化，非常不方便。而使用桥接模式来实现类似的功能，就没有这个问题了。

  另外，这里只是说从实现具体的业务功能上，桥接模式可以模拟实现出模板方法模式能实现的功能，并不是说桥接模式和模板方法模式就变成一样的，或者是桥接模式就可以替换掉模板方法模式了。要注意它们本身的功能、目的、本质思想都是不一样的。

- 桥接模式和抽象工厂模式

  这两个模式可以组合使用。

  桥接模式中，抽象部分需要获取相应的实现部分的接口对象，那么谁来创建实现部分的具体实现对象

呢？这就是抽象工厂模式派上用场的地方。也就是使用抽象工厂模式来创建和配置一个特定的具体实现的对象。

事实上，抽象工厂主要是用来创建一系列对象的，如果创建的对象很少，或者是很简单，还可以采用简单工厂，可以达到一样的效果，但是会比抽象工厂来得简单。

- 桥接模式和适配器模式

  这两个模式可以组合使用。

  这两个模式功能是完全不一样的，适配器模式的功能主要是用来帮助无关的类协同工作，重点在解决原本由于接口不兼容而不能一起工作的那些类，使得它们可以一起工作。而桥接模式则重点在分离抽象和实现部分。

  所以在使用上，通常在系统设计完成过后，才会考虑使用适配器模式；而桥接模式，是在系统开始的时候就要考虑使用。

  虽然功能上不一样，这两个模式还是可以组合使用的，比如：已有实现部分的接口，但是有些不太适应现在新的功能对接口的需要，完全抛弃吧，有些功能还用得上，该怎么办呢？那就使用适配器来进行适配，使得旧的接口能够适应新的功能的需要。

研磨设计模式结束，谢谢捧场!

希望收到您的反馈，个人QQ：1500562586 ，个人邮件：sjms_2010@yahoo.cn

研磨设计模式--chjavach的博客文章

作者: chjavach
http://chjavach.javaeye.com