研磨设计模式 博文集

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

包揽JavaEye设计模式类前十名。
包含前言、简单工厂、工厂方法模式、单例模式、桥接模式

目 录

1. 设计模式

1.1 研磨设计模式终于上市了

发表时间: 2010-12-17

首先感谢大家长久以来的支持和鼓励，没有你们的大力帮助，是很难坚持走到今天的，因此，向你们致以诚挚的感谢！

另外有很多朋友在博客里面、QQ里面询问，到底什么时候能够看到书，这里集中公告一下：

目前书已经上市，网店和书店应该都有了，当然，远一点的地方，可能书店要上得慢一点，过几天应该就会有的。

这里推荐大家去当当网购买，因为：

1：当当网 是 75折，与其他家网店同价，都是最低的

2：当当网现在在搞活动，全场满119 返 30，这是很合适的

当当网的销售链接：http://product.dangdang.com/product.aspx?product_id=20994349&ref=search-0-mix

希望朋友们能给本书评评分，写点评论，多谢大家的支持！

1.2 研磨设计模式 封面show

发表时间: 2010-10-29

今天收到编辑发过来的最终图书封面，很开心，拿出来show一下！

顺带做个广告，《研磨设计模式》 一书由 清华大学出版社 出版并发行，即将上市，敬请关注！

1.3 研磨设计模式 的 前言

发表时间: 2010-12-20

首先向各位长期支持我的朋友致以诚挚的感谢！《研磨设计模式》一书已经上市，具体的信息请参见上一篇博文，这里就不多说了，免得JE做出他们认为合理的举动。

研磨设计模式的前言，里面也有很多对大家有用的信息。

前言

创作背景

软件开发越来越复杂，对软件设计的要求也越来越高，而软件设计和架构的入门功夫就是深入理解和掌握设计模式，因此，设计模式的重要性不言而喻。

很多朋友也认识到了设计模式的重要性，也看了很多的书籍和资料，但是，常听到他们这样的抱怨：“设计模式的书我看了不少，觉得都看懂了，就是不知道在实际开发中怎么用这些设计模式”，从而认为设计模式是“看上去很美的花拳绣腿”。

其实不然，造成这种情况的原因就在于：这些朋友对设计模式的理解不到位，自己感觉懂了，其实还差很远，并不是“真正”理解和掌握了设计模式。

市面上也有不少的设计模式方面的书籍，但对一般的学习者而言，要么就是太深，看得云里雾里的，比如GoF的著作《设计模式——可复用面向对象软件的基础》，很经典，但是能吃透的人少；要么就是太浅，看了跟没看差不多，也就是介绍一下每个设计模式，告诉你这就是某某设计模式，虽然语言很生动但是实在没货，看完也不知道怎么用，就像是带领大家摸到了设计模式的大门口，却不告诉你怎么进去一样，其根本原因还是讲得太浅，跟实际的应用有太大的差距。

对于所有想要真正理解和掌握设计模式的朋友，其实需要这样的书籍：

- 理论全面、准确，难度适中
- 讲解深入浅出、浅显易懂
- 理论联系实际应用，对于晦涩的理论，应有相应的示例
- 示例最好来自实际应用，而不是来自虚拟的场景
- 示例最好相对完整，而不是片断代码，以利于学习和应用

这也是本书写作的目的，希望能够帮助更多的朋友早日修成设计模式的正果。

经过多年的准备和一年的写作，以及各层次读者的多轮试读意见和建议汇总，最终成书，我们可以这样说：**这是一本诚意十足的书，敬请您的评鉴！**

本书的试读人员包括：从还没有参加工作的学生，一直到工作7年的人员；职务覆盖普通的程序员、项目经理、高级系统架构师、技术部的经理；两位作者本身从事开发工作的年限，一位超过10年，一位超过5年。

试读的结果：工作经验在1年以下的朋友，能正常理解和掌握初级部分的内容，能部分理解中高级部分的内容；工作经验在1-2年的朋友，基本上能全面理解，但是领悟尚有不足；工作经验在2-5年的朋友，能够正常理解和掌握，基本达到本书写作的意图；工作经验在5年以上的朋友，主要是弥补以前较少用到的部分，使知识更加系统化和全面化，另外把本书当作一本工具参考书，案头必备。

本书内容

本书完整覆盖GoF的著作《设计模式——可复用面向对象软件的基础》一书所讲述的23个设计模式。

- 初级内容：从基本讲起，包括每个模式的定义、功能、思路、结构、基本实现、运行调用顺序、基本应用示例等等，让读者能系统、完整、准确的掌握每个模式，培养正确的“设计观”
- 中高级内容：深入探讨如何理解这些模式，模式中蕴含什么样的设计思想，模式的本质是什么，模式如何结合实际应用，模式的优缺点，和其它模式的关系等等，以期让读者尽量去理解和掌握每个设计模式的精髓所在

本书在内容上深入、技术上实用、和实际开发结合程度很高，书中大部分的示例程序都是从实际项目中简化而来，因此很多例子都可以直接拿到实际项目中使用。如果你想要深入透彻的理解和掌握设计模式，并期望能真正把设计模式应用到项目中去，那么这是你不可错过的一本书。

本书特色

1. **本书有很多独到的见解和精辟的总结，能写出一些人所不敢写、人所不能写的内容，是一本“真正有货”的书**
2. **本书大部分示例程序都来自真实的项目应用，让你真正理解和掌握设计模式，尽量做到“从实际项目中来，再应用回到实际项目中去”**

本书涉及的实际应用，包含但不限于：

- 代码生成的应用工具（独立应用）
- 日志管理平台（来自于基础平台）
- 缓存管理（来自于基础平台）
- 订单处理（来自于CRM系统）
- 导出数据的应用框架（来自于SCM）

- 组织机构管理（来自于基础平台）
- 大数据量访问（很多系统都有）
- 水质监测系统（独立应用）
- 工资管理（来自于HRM系统）
- 商品管理（来自于电子商务系统）
- 登录控制（来自于OA系统）
- 报价管理（来自于CRM系统）
- 在线投票系统（来自于OA系统）
- 仿真系统（来自于WorkFlow系统）
- 权限管理（来自于基础平台）
- 配置文件管理（来自于基础平台）
- 奖金核算系统（来自于HRM系统）
- 费用报销管理（来自于OA系统）
- 客户管理（来自于CRM系统）

说明：OA：办公自动化（Office Automation）
CRM：客户关系管理（Customer Relationship Management）
HRM：人力资源管理（Human Resource Management）
SCM：供应链管理（Supply Chain Management)
WorkFlow：工作流

本书探讨了很多应用设计模式来解决的实际项目中的问题

本书涉及的实际问题，包含但不限于：

- 如何实现可配置
- 如何实现同时支持数据库和文件存储的日志管理
- 如何实现缓存以及缓存的管理
- 如何实现用缓存来控制多实例的创建
- 如何处理平行功能
- 如何实现参数化工厂
- 如何应用工厂实现DAO
- 如何实现可扩展工厂
- 如何实现原型管理器
- 如何实现Java的静态代理和动态代理
- 如何实现多线程处理队列请求
- 如何实现命令的参数化配置、可撤销的操作、宏命令、队列请求和日志请求
- 如何实现双向迭代

- 如何实现带策略的迭代器
- 如何实现翻页迭代
- 如何实现树状结构和父组件引用
- 如何检测环状结构
- 如何实现通用的增删改查
- 如何实现容错恢复机制
- 如何模拟工作流来处理流程
- 如何实现对象实例池
- 如何实现自定义语言的解析
- 如何实现简单又通用的XML读取
- 如何实现功能链，实现类似于Web开发中Filter的功能
- 如何实现模拟AOP的功能
- 如何为系统加入权限控制
- 如何自定义I/O装饰器
- 如何实现通用请求处理框架

3：本书的示例程序基本上都是带着客户端测试代码的，可直接运行，不是片断代码，更有利于大家整体学习和理解

读者定位

本书难度为初级到中级，适合于所有开发人员、设计人员或者即将成为开发人员的朋友。也可以作为高校学生深入学习设计模式的参考读物。

我们强烈建议您认真阅读和学习本书的内容，全面、准确、深入、实用的内容定会有助于您凤凰涅磐般的实现技术升华，请相信。

阅读指南

本书假定您懂一些基本的Java知识，并具备一定的开发经验。

1：对于初学设计模式的朋友

如果对常见面向对象的设计原则不太熟悉的话，请先参看附录一。

如果对UML不太熟悉的话，请先参看附录二。

然后开始看第一章，学习设计模式的一些基础知识，了解本书的整体大纲。

接下来就可以从前到后，循序渐进的学习每个设计模式。对每个模式建议先认真学习场景问题和解决方案两个部分，切实掌握每个模式标准的结构、实现和基本的应用。对于模式讲解中简单的内容也可以先看，但是对于后面较为复杂的内容，可以先不看，等到技术和经验积累到一定程度的时候，再循序渐进向后学习。

2：对于已有一定的开发经验和设计经验的朋友

还是应该从场景问题和解决方案看起，对于其中已会的内容权当复习，对于不会的内容，相当于是在查漏补缺，先把基础部分整得全面、扎实。

然后再认真学习模式讲解部分，并结合实际的开发经验来思考，看看如何应用模式来解决实际问题，看看如何把模式应用到实际的项目中去，再深入的思考模式的本质和设计思想，掌握模式的精髓，这样才能真正做到在实际开发中自如应用设计模式。

3：对所有的朋友

这不是一本随便看看，读完一遍就可以扔掉的书籍，需要反复研读。因此，第一次阅读本书时，如果发现有些不理解的内容也不要紧，可以在今后的学习和工作中，反复参阅本书，以加深对设计模式的理解，获取设计灵感，并把设计模式切实应用到实际项目中去。

4：善意提醒

在实际开发和设计中，要遵循简单设计的原则，不要为了设计模式而模式，不要过度设计，要在合适的地方应用合适的设计模式来解决问题。

这对于初学者尤其要注意，因为刚学会一个东西，总是跃跃欲试，急于一显身手，往往容易造成设计模式的误用。

本书约定

1：本书的知识边界

由于关于设计的知识过于博大精深，因此本书“集中火力”，重点在讲述GoF著作中涉及的23个设计模式本身、以及和这些设计模式相关的应用内容。

没有过多涉及：面向对象设计原则、重构、系统架构设计、JavaEE（原J2EE，也有简写成JEE）设计模式或是其它分类的设计模式（如EJB设计模式）等内容，原因可以参见附录一。也没有过多讲述UML，有需要的朋友请参看附录二。

对于每章涉及的实际应用，描述也非常简略，只抽取讲述模式需要的一点东西。因为这些实际应用的东西，对于有相应开发经验的朋友多说无益，一提就明白，对于没有相应经验的朋友，多讲一点也未见得能多明白多少，反而冲淡了设计模式这个主题。

2：本书的示例和代码

本书的示例虽然大都来自实际应用，但是经过相当的删除简化和重新组合；另外一点，为了突出设计模式这个主题，因此代码并不是按照实际应用那样来严格要求，很多例外处理、数据检测等都没有做，逻辑也未见得那么严密；还有一点，在实际的开发中，很可能是多个模式组合来实现某个功能，但是本书为了示例某个模式，让重点突出而避免读者迷惑，会选择重点示例某个模式的用法，而简化或去掉其它模式。

如果要把这些示例代码在实际应用中使用，还需要对这些代码进行加工，使其更加严谨，才能达到工业级的要求。

真诚致谢

首先要感谢本书的编辑栾大成先生，他给予本书很多中肯的意见和建议，对本书从选题到出版的各个环节，都给予了大量的指导和帮助。

其次要感谢张开涛先生，他对本书的内容提出了很多有用的意见和建议。

然后要感谢魏源先生和蔡抒杨先生，他们对本书内容的完善也给出了很好的建议。

接下来，按照惯例，应该感谢家人、感谢朋友、感谢北京的漫天风沙和明媚阳光，以及那可爱的阳台和小巧的书桌，总之，感谢一切。

最后，提前感谢购买本书的朋友们，你们的信任和赏识是我们继续前进的动力，对于本书有任何意见或建议，可以直接与我们联系，联系邮件：sjms_2010@yahoo.cn ，我们也很乐意与各位朋友交流设计模式或是其它相关的技术内容。

前言结束，供大家参考，谢谢

1.4 研磨设计模式之简单工厂模式-1

发表时间: 2010-11-02

继续研磨设计模式，来个简单的

简单工厂

简单工厂不是一个标准的设计模式，但是它实在是太常用了，简单而又神奇，所以还是需要好好掌握的，就当是对学习设计模式的热身运动吧。

为了保持一致性，我们尽量按照学习其它模式的步骤来进行学习。

1 场景问题

大家都知道，在Java应用开发中，要“面向接口编程”。

那么什么是接口？接口有什么作用？接口如何使用？一起来回顾一下：

1.1 接口回顾

（1）Java中接口的概念

在Java中接口是一种特殊的抽象类，跟一般的抽象类相比，接口里面的所有方法都是抽象方法，接口里面的所有属性都是常量。也就是说，接口里面是只有方法定义而不会有任何方法实现。

（2）接口用来干什么

通常用接口来定义实现类的外观，也就是实现类的行为定义，用来约束实现类的行为。接口就相当于一份契约，根据外部应用需要的功能，约定了实现类应该要实现的功能，但是具体的实现类除了实现接口约定的功能外，还可以根据需要实现一些其它的功能，这是允许的，也就是说实现类的功能包含但不仅限于接口约束的功能。

通过使用接口，可以实现不相关类的相同行为，而不需考虑这些类之间的层次关系，接口就是实现类对外的外观。

（3）接口的思想

根据接口的作用和用途，浓缩下来，**接口的思想就是“封装隔离”。**

通常提到封装是指对数据的封装，但是这里的封装是指“对被隔离体的行为的封装”，或者是“对被隔离体的职责的封装”；而隔离指的是外部调用和内部实现，外部调用只能通过接口进行调用，而外部调用是不知

道内部具体实现的，也就是说外部调用和内部实现是被接口隔离开的。

（4）使用接口的好处

由于外部调用和内部实现被接口隔离开了，那么只要接口不变，内部实现的变化就不会影响到外部应用，从而使得系统更灵活，具有更好的扩展性和可维护性，这也就是所谓“接口是系统可插拔性的保证”这句话的意思。

（5）接口和抽象类的选择

既然接口是一种特殊的抽象类，那么在开发中，何时选用接口，何时选用抽象类呢？

对于它们的选择，在开发中是一个很重要的问题，特别总结两句话给大家：

- 优先选用接口
- **在如下情况应选择抽象类：既要定义子类的行为，又要为子类提供公共的功能**

1.2 面向接口编程

面向接口编程是Java编程中的一个重要原则。

在Java 程序设计里面，非常讲究层的划分和模块的划分。通常按照三层来划分Java程序，分别是表现层、逻辑层、数据层，它们之间都要通过接口来通讯。

在每一个层里面，又有很多个小模块，一个小模块对外也应该是一个整体，那么一个模块对外也应该提供接口，其它地方需要使用到这个模块的功能，都应该通过此接口来进行调用。这也就是常说的“接口是被其隔离部分的外观”。基本的三层结构如图1所示：

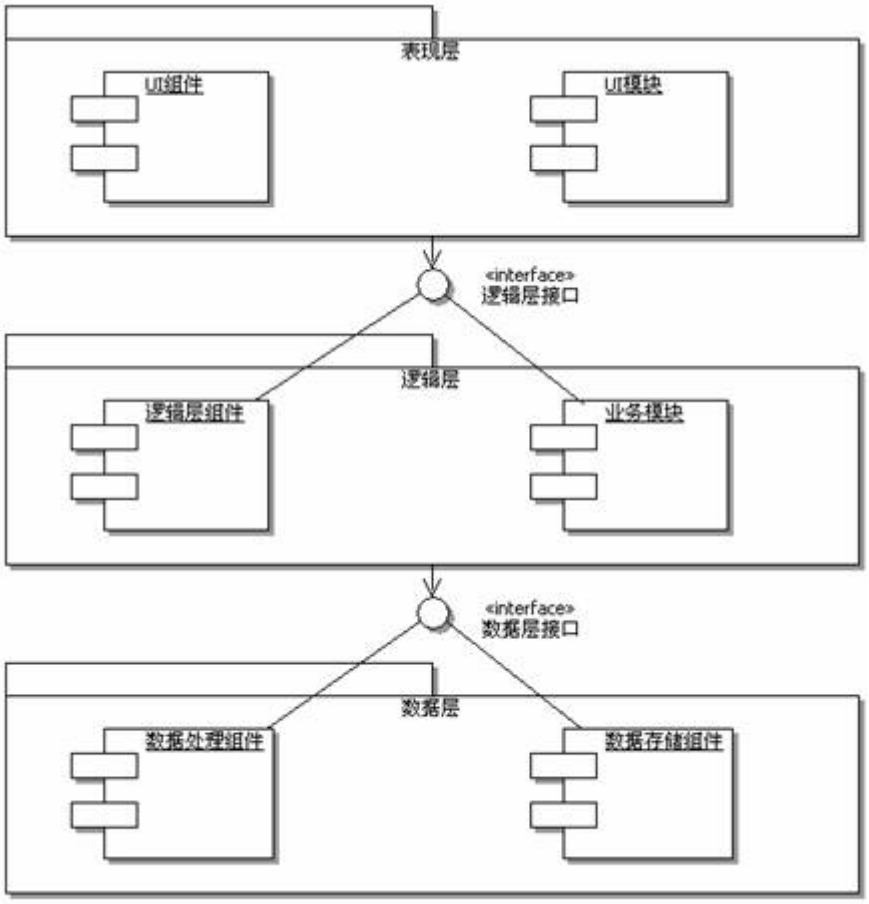


图1 基本的三层结构示意图

在一个层内部的各个模块交互也要通过接口，如图2所示：

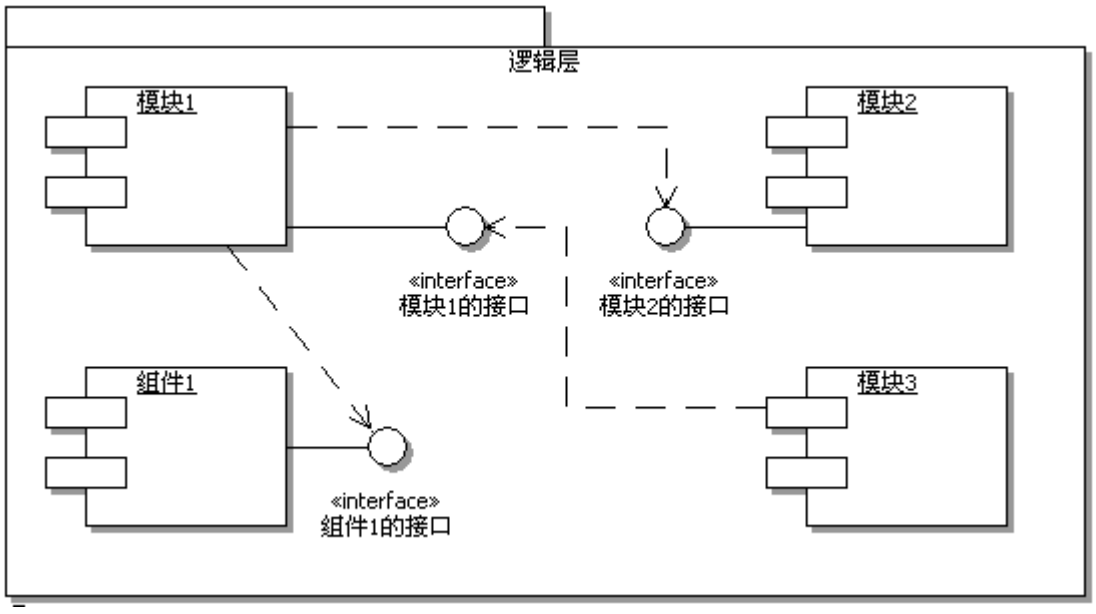


图2 一个层内部的各个模块交互示意图

各个部分的接口具体应该如何去定义，具体的内容是什么，不去深究，那是需要具体问题具体分析的，这里只是来学习设计的方法。

上面频频提到“组件”，那么什么是组件呢？先简单的名词解释一下：

所谓组件：**从设计上讲，组件就是能完成一定功能的封装体。**小到一个类，大到一个系统，都可以称为组件，因为一个小系统放到更大的系统里面去，也就当个组件而已。事实上，从设计的角度看，系统、子系统、模块、组件等说的其实是同一回事情，都是完成一定功能的封装体，只不过功能多少不同而已。

继续刚才的思路，大家会发现，不管是一层还是一个模块或者一个组件，都是一个被接口隔离的整体，那么下面我们就不去区分它们，统一认为都是接口隔离体即可，如图3所示：

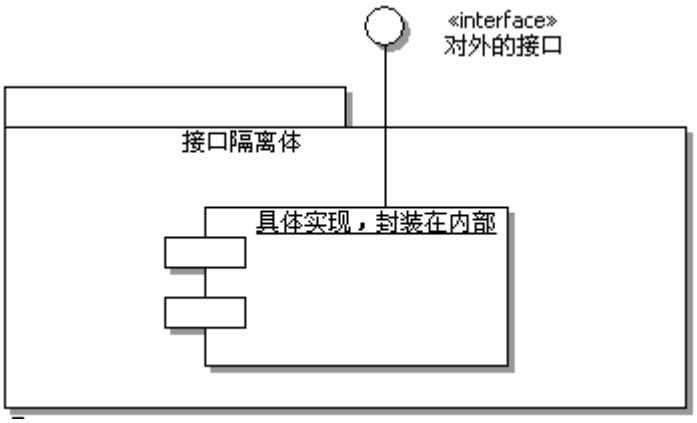


图3 接口隔离体示意图

既然在Java中需要面向接口编程，那么在程序中到底如何使用接口，来做到真正的面向接口编程呢？

1.3 不用模式的解决方案

回忆一下，以前是如何使用接口的呢，假设有一个接口叫Api，然后有一个实现类Impl实现了它，在客户端怎么用这个接口呢？

通常都是在客户端创建一个Impl的实例，把它赋值给一个Api接口类型的变量，然后客户端就可以通过这个变量来操作接口的功能了，此时具体的结构图如图4：

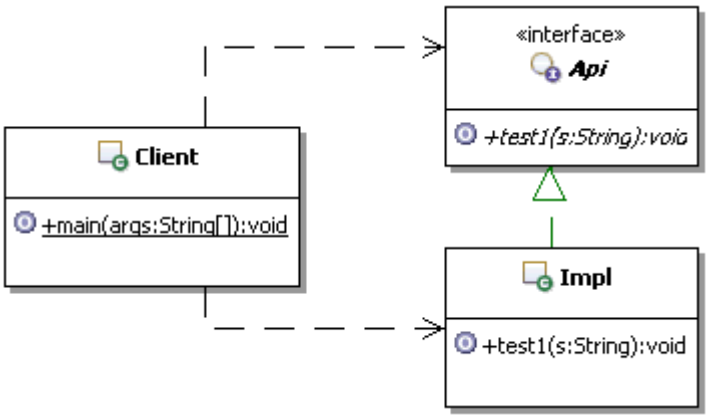


图4 基本的接口和实现

还是用代码来说明，会更清楚一些。

（1）先定义接口Api，示例代码如下：

```
/**
* 某个接口(通用的、抽象的、非具体的功能)
*/
public interface Api {
        /**
         * 某个具体的功能方法的定义，用test1来演示一下。
         * 这里的功能很简单，把传入的s打印输出即可
         * @param s 任意想要打印输出的字符串
         */
        public void test1(String s);
}
```

（2）既然有了接口，自然就要有实现，定义实现Impl，示例代码如下：

```
/**
* 对接口的实现
*/
public class Impl implements Api{
        public void test1(String s) {
                System.out.println("Now In Impl. The input s=="+s);
        }
}
```

（3）那么此时的客户端怎么写呢？

按照Java的知识，接口不能直接使用，需要使用接口的实现类，示例代码如下：

```
/**
* 客户端：测试使用Api接口
*/
public class Client {
        public static void main(String[] args) {
                Api api = new Impl();
                api.test1("哈哈，不要紧张，只是个测试而已！");
        }
}
```

1.4 有何问题

上面写得没错吧，在Java的基础知识里面就是这么学的，难道这有什么问题吗？

请仔细看位于客户端的下面这句话：

```
  Api api = new Impl();
```

然后再想想接口的功能和思想，发现什么了？仔细再想想？

你会发现在客户端调用的时候，客户端不但知道了接口，同时还知道了具体的实现就是Impl。而接口的思想是“封装隔离”，而Impl这个实现类，应该是被接口Api封装并同客户端隔离开的，也就是说，客户端根本就不应该知道具体的实现类是Impl。

有朋友说，那好，我就把Impl从客户端拿掉，让Api真正的对实现进行“封装隔离”，然后我们还是面向接口来编程。可是，新的问题出现了，当他把“new Impl()”去掉过后，发现他无法得到Api接口对象了，怎么办呢？

把这个问题描述一下：**在Java编程中，出现只知接口而不知实现，该怎么办？**

就像现在的Client，它知道要使用Api接口，但是不知由谁实现，也不知道如何实现，从而得不到接口对象，就无法使用接口，该怎么办呢？

未完待续

1.5 研磨设计模式之简单工厂模式-2

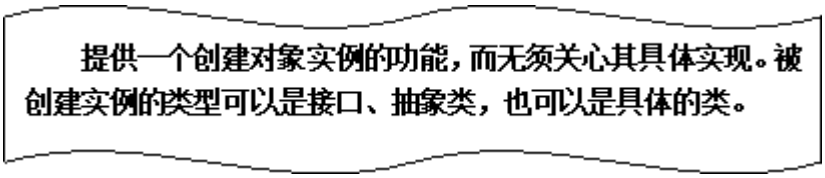
发表时间: 2010-11-18

2 解决方案

1 简单工厂来解决

用来解决上述问题的一个合理的解决方案就是简单工厂，那么什么是简单工厂呢？

1：简单工厂定义



2：应用简单工厂来解决的思路

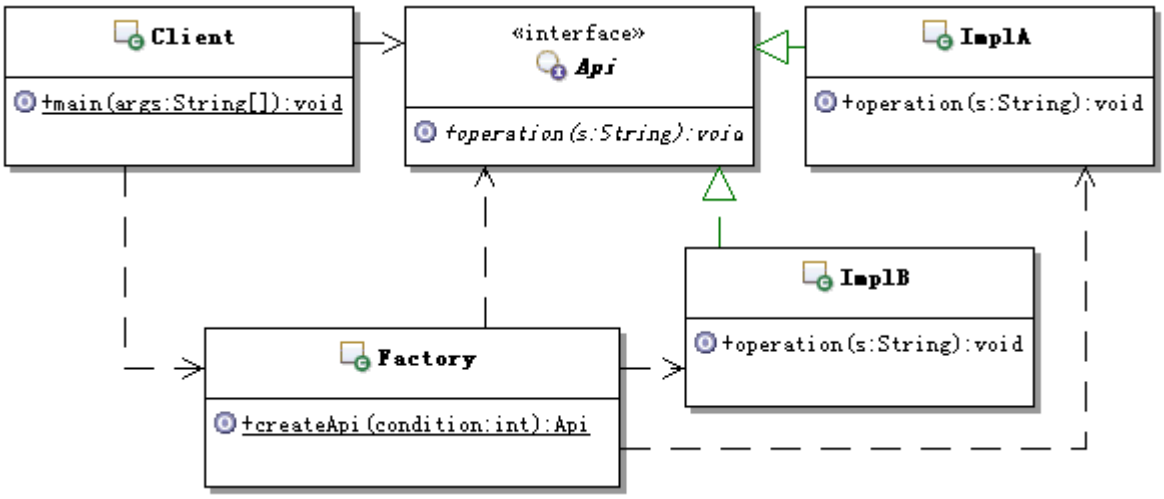
分析上面的问题，虽然不能让模块外部知道模块内的具体实现，但是模块内部是可以知道实现类的，而且创建接口是需要具体实现类的。

那么干脆在模块内部新建一个类，在这个类里面来创建接口，然后把创建好的接口返回给客户端，这样外部应用就只需要根据这个类来获取相应的接口对象，然后就可以操作接口定义的方法了。把这样的对象称为简单工厂，就叫Factory吧。

这样一来，客户端就可以通过这个Factory来获取需要的接口对象，然后调用接口的方法来实现需要的功能，而且客户端也不用再关心具体实现了。

2.2 简单工厂结构和说明

简单工厂的结构如图5所示：



Api：

定义客户所需要的功能接口

Impl：

具体实现Api的实现类，可能会有多个

Factory：

工厂，选择合适的实现类来创建Api接口对象

Client：

客户端，通过Factory去获取Api接口对象，然后面向Api接口编程

2.3 简单工厂示例代码

（1）先看看Api的定义，示例代码如下：

```
/**
 * 接口的定义，该接口可以通过简单工厂来创建
 */
public interface Api {
        /**
         * 示意，具体的功能方法的定义
         * @param s 示意，需要的参数
         */
        public void operation(String s);
}
```

（2）定义了接口，该来实现它了，ImplA的示例代码如下：

```
/**
 * 接口的具体实现对象A
 */
public class ImplA implements Api{
        public void operation(String s) {
                //实现功能的代码，示意一下
                System.out.println("ImplA s=="+s);
        }
}
```

ImplB的示意实现和ImplA基本一样，示例代码如下：

```
/**
 * 接口的具体实现对象B
 */
public class ImplB implements Api{
        public void operation(String s) {
                //实现功能的代码，示意一下
                System.out.println("ImplB s=="+s);
        }
}
```

（3）该来看看简单工厂的实现，示例代码如下：

```
/**
 * 工厂类，用来创造Api对象
 */
public class Factory {
        /**
         * 具体的创造Api对象的方法
         * @param condition 示意，从外部传入的选择条件
         * @return 创造好的Api对象
         */
        public static Api createApi(int condition){
                //应该根据某些条件去选择究竟创建哪一个具体的实现对象，
                //这些条件可以从外部传入，也可以从其它途径获取。
                //如果只有一个实现，可以省略条件，因为没有选择的必要。
```

```
                //示意使用条件
                Api api = null;
                if(condition == 1){
                        api = new ImplA();
                }else if(condition == 2){
                        api = new ImplB();
                }
                return api;
        }
}
```

（4）再来看看客户端的示意，示例代码如下：

```
/**
 * 客户端，使用Api接口
 */
public class Client {
        public static void main(String[] args) {
                //通过简单工厂来获取接口对象
                Api api = Factory.createApi(1);
                api.operation("正在使用简单工厂");
        }
}
```

2.4 使用简单工厂重写示例

要使用简单工厂来重写前面的示例，主要就是要创建一个简单工厂对象，让简单工厂来负责创建接口对象。然后让客户端通过工厂来获取接口对象，而不再由客户端自己去创建接口的对象了。

此时系统的结构如图6所示。

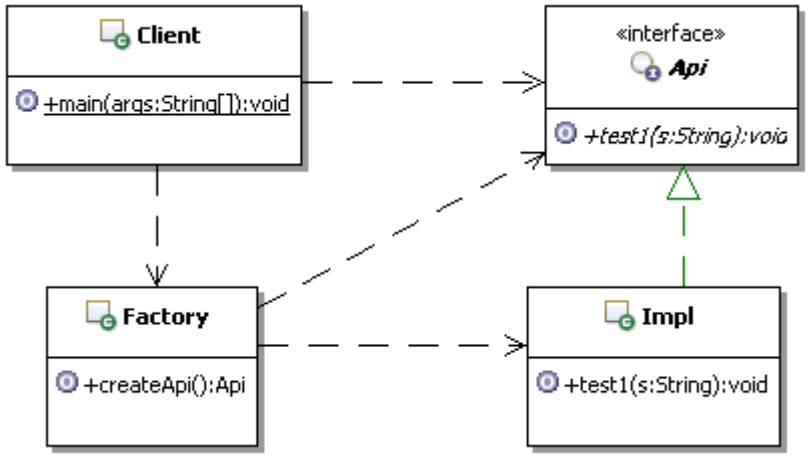


图6 使用简单工厂重写示例的结构示意图

（1）接口Api和实现类Impl都和前面的示例一样，就不去赘述了。

（2）新创建一个简单工厂的对象，示例代码如下：

```
/**
 * 工厂类，用来创造Api对象
 */
public class Factory {
        /**
         * 具体的创造Api对象的方法
         * @return 创造好的Api对象
         */
        public static Api createApi(){
                //由于只有一个实现，就不用条件判断了
                return new Impl();
        }
}
```

（3）使用简单工厂

客户端如何使用简单工厂提供的功能呢？这个时候，客户端就不用再自己去创建接口的对象了，应该使用工厂来获取，经过改造，客户端代码如下：

```
/**
* 客户端：测试使用Api接口
*/
```

```
public class Client {
        public static void main(String[] args) {
                //重要改变，没有new Impl()了，取而代之Factory.createApi()
Api api = Factory.createApi();
                api.test1("哈哈，不要紧张，只是个测试而已！");
        }
}
```

就如同上面的示例，客户端通过简单工厂创建了一个实现接口的对象，然后面向接口编程，从客户端来看，它根本就不知道具体的实现是什么，也不知道是如何实现的，它只知道通过工厂获得了一个接口对象，然后就能通过这个接口来获取想要的功能。

事实上，简单工厂能帮助我们真正开始面向接口编程，像以前的做法，其实只是用到了接口的多态那部分的功能，最重要的“封装隔离性”并没有体现出来。

未完待续

1.6 研磨设计模式之简单工厂模式-3

发表时间: 2010-12-06

3 模式讲解

3.1 典型疑问

首先来解决一个常见的疑问：可能有朋友会认为，上面示例中的简单工厂看起来不就是把客户端里面的“new Impl()”移动到简单工厂里面吗？不还是一样通过new一个实现类来得到接口吗？把“new Impl()”这句话放到客户端和放到简单工厂里面有什么不同吗？

理解这个问题的重点就在于理解简单工厂所处的位置。

根据前面的学习，我们知道接口是用来封装隔离具体的实现的，目标就是不要让客户端知道封装体内部的具体实现。简单工厂的位置是位于封装体内的，也就是简单工厂是跟接口和具体的实现在一起的，算是封装体内部的一个类，所以简单工厂知道具体的实现类是没有关系的。整理一下简单工厂的结构图，新的图如图7所示：

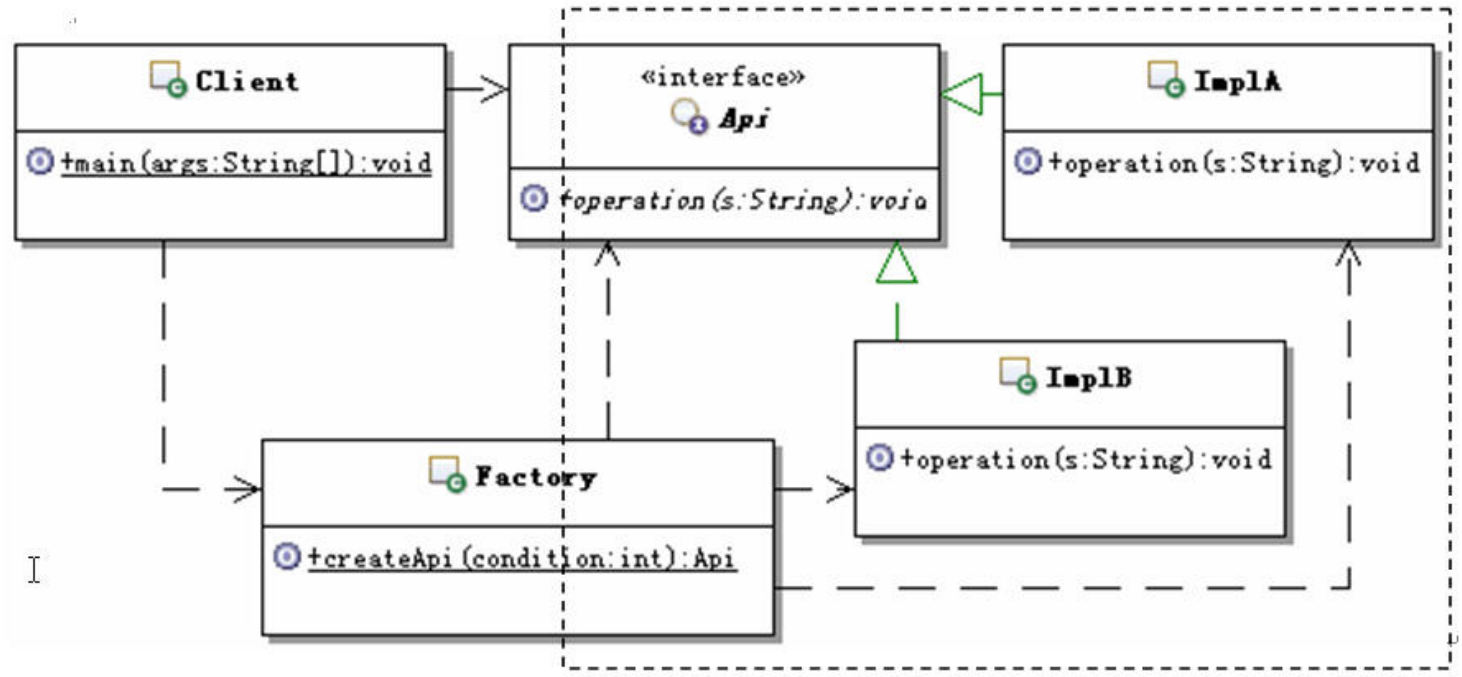


图7 整理后的简单工厂结构

图7中虚线框，就好比是一个组件的包装边界，表示接口、实现类和工厂类组合成了一个组件，在这个封装体里面，只有接口和工厂是对外的，也就是让外部知道并使用的，所以故意漏了一些在虚线框外，而具体的实现类是不对外的，被完全包含在虚线框内。

对于客户端而言，只是知道了接口Api和简单工厂Factory，通过Factory就可以获得Api了，这样就达到了

让Client在不知道具体实现类的情况下获取接口Api。

所以看似简单的把“new Impl()”这句话从客户端里面移动到了简单工厂里面，其实是有了质的变化的。

3.2 认识简单工厂

（1）简单工厂的功能

工厂嘛，就是用来造东西的。在Java里面，通常情况下是用来造接口的，但是也可以造抽象类，甚至是一个具体的类实例。

一定要注意，虽然前面的示例是利用简单工厂来创建的接口，但是也是可以用简单工厂来创建抽象类或者是普通类的实例的。

（2）静态工厂

使用简单工厂的时候，通常不用创建简单工厂类的类实例，没有创建实例的必要。因此可以把简单工厂类实现成一个工具类，直接使用静态方法就可以了，也就是说简单工厂的方法通常都是静态的，所以也被称为静态工厂。如果要防止客户端无谓的创造简单工厂实例，还可以把简单工厂的构造方法私有化了。

（3）万能工厂

一个简单工厂可以包含很多用来构造东西的方法，这些方法可以创造不同的接口、抽象类或者是类实例，一个简单工厂理论上可以构造任何东西，所以又称之为“万能工厂”。

虽然上面的实例中，在简单工厂里面只有一个方法，但事实上，是可以有很多这样创建方法的，这点要注意。

（4）简单工厂创建对象的范围

虽然从理论上讲，简单工厂什么都能造，但对于简单工厂可创建对象的范围，通常不要太大，建议控制在一个独立的组件级别或者一个模块级别，也就是一个组件或模块一个简单工厂。否则这个简单工厂类会职责不明，有点大杂烩的感觉。

（5）简单工厂的调用顺序示意图

简单工厂的调用顺序如图8所示：

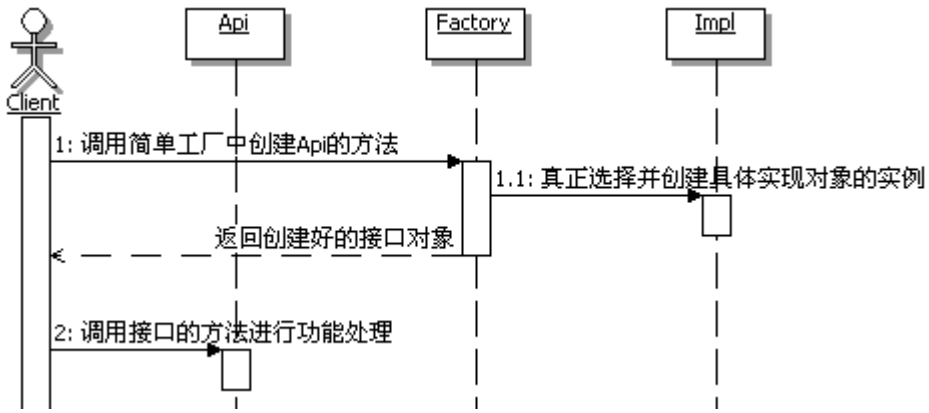


图8 简单工厂的调用顺序示意图

（6）简单工厂命名的建议

- 类名建议为“模块名称+Factory”，比如：用户模块的工厂就称为：UserFactory
- 方法名称通常为“get+接口名称”或者是“create+接口名称”，比如：有一个接口名称为UserEbi，那么方法名称通常为：getUserEbi 或者是 createUserEbi。
- 当然，也有一些朋友习惯于把方法名称命名为“new+接口名称”，比如：newUserEbi,我们不是很建议。因为new在Java中代表特定的含义，而且通过简单工厂的方法来获取对象实例，并不一定每次都是要new一个新的实例。如果使用newUserEbi，这会给人错觉，好像每次都是new一个新的实例一样。

3.3 简单工厂中方法的写法

虽然说简单工厂的方法多是用来造接口的，但是仔细分析就会发现，真正能实现功能的是具体的实现类，这些实现类是已经做好的，并不是真的靠简单工厂来创造出来的，简单工厂的方法无外乎就是：实现了选择一个合适的实现类来使用。

所以简单工厂方法的内部主要实现的功能是“选择合适的实现类”来创建实例对象。既然要实现选择，那么就需要选择的条件或者是选择的参数，选择条件或者是参数的来源通常又有几种：

- 来源于客户端，由Client来传入参数
- 来源于配置文件，从配置文件获取用于判断的值
- 来源于程序运行期的某个值，比如从缓存中获取某个运行期的值

下面来看个示例，看看由客户端来传入参数，如何写简单工厂中的方法。

（1）在刚才的示例上再添加一个实现，称为Impl2，示例代码如下：

```
/**
 * 对接口的一种实现
 */
public class Impl2 implements Api{
        public void test1(String s) {
                System.out.println("Now In Impl The input s=="+s);
        }
}
```

（2）现在对Api这个接口，有了两种实现，那么工厂类该怎么办呢？到底如何选择呢？不可能两个同时使用吧，看看新的工厂类，示例代码如下：

```
/**
 * 工厂类，用来创造Api的
 */
public class Factory {
        /**
         * 具体的创造Api的方法，根据客户端的参数来创建接口
         * @param type 客户端传入的选择创造接口的条件
         * @return 创造好的Api对象
         */
        public static Api createApi(int type){
                //这里的type也可以不由外部传入，而是直接读取配置文件来获取
                //为了把注意力放在模式本身上，这里就不去写读取配置文件的代码了

                //根据type来进行选择，当然这里的1和2应该做成常量
                Api api = null;
                if(type==1){
                        api = new Impl();
                }else if(type==2){
                        api = new Impl2();
                }
                return api;
        }
}
```

（3）客户端没有什么变化，只是在调用Factory的createApi方法的时候需要传入参数，示例代码如下：

```
public class Client {
        public static void main(String[] args) {
                //注意这里传递的参数，修改参数就可以修改行为，试试看吧
                Api api = Factory.createApi(2);
                api.test1("哈哈，不要紧张，只是个测试而已！");
        }
}
```

（4）要注意这种方法有一个缺点

由于是从客户端在调用工厂的时候，传入选择的参数，这就说明客户端必须知道每个参数的含义，也需要理解每个参数对应的功能处理。这就要求必须在一定程度上，向客户暴露一定的内部实现细节。

3.4 可配置的简单工厂

现在已经学会通过简单工厂来选择具体的实现类了，可是还有问题。比如：在现在的实现中，再新增加一种实现，会怎样呢？

那就需要修改工厂类，才能把新的实现添加到现有系统中。比如现在新加了一个实现Impl3，那么需要类似下面这样来修改工厂类：

```
public class Factory {
        public static Api createApi(int type){
                Api api = null;
                if(type==1){
                        api = new Impl();
                }else if(type==2){
                        api = new Impl2();
                }

                else if(type==3){
                        api = new Impl3();
                }
                return api;
        }
}
```

每次新增加一个实现类都来修改工厂类的实现，肯定不是一个好的实现方式。那么现在希望新增加了实现类过后不修改工厂类，该怎么办呢？

一个解决的方法就是使用配置文件，当有了新的实现类过后，只要在配置文件里面配置上新的实现类就好了，在简单工厂的方法里面可以使用反射，当然也可以使用IoC/DI（控制反转/依赖注入，这个不在这里讨论）来实现。

看看如何使用反射加上配置文件，来实现添加新的实现类过后，无须修改代码，就能把这个新的实现类加入应用中。

（1）配置文件用最简单的properties文件，实际开发中多是xml配置。定义一个名称为“FactoryTest.properties”的配置文件，放置到Factory同一个包下面，内容如下：

```
ImplClass=cn.javass.dp.simplefactory.example5.Impl
```

如果新添加了实现类，修改这里的配置就可以了，就不需要修改程序了。

（2）此时的工厂类实现如下：

```
/**
* 工厂类，用来创造Api对象
*/
public class Factory {
        /**
         * 具体的创造Api的方法，根据配置文件的参数来创建接口
         * @return 创造好的Api对象
         */
        public static Api createApi(){
                //直接读取配置文件来获取需要创建实例的类
                //至于如何读取Properties，还有如何反射这里就不解释了
                Properties p = new Properties();
                InputStream in = null;
                try {
                        in = Factory.class.getResourceAsStream(
"FactoryTest.properties");
                        p.load(in);
                } catch (IOException e) {
                        System.out.println(
"装载工厂配置文件出错了，具体的堆栈信息如下：");
                        e.printStackTrace();
                }finally{
                        try {
                                in.close();
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                }
                //用反射去创建，那些例外处理等完善的工作这里就不做了
                Api api = null;
                try {
                        api = (Api)Class.forName(p.getProperty("ImplClass"))
```

```
.newInstance();
                } catch (InstantiationException e) {
                        e.printStackTrace();
                } catch (IllegalAccessException e) {
                        e.printStackTrace();
                } catch (ClassNotFoundException e) {
                        e.printStackTrace();
                }
                return api;
        }
}
```

（3）此时的客户端就变得很简单了，不再需要传入参数，代码示例如下：

```
public class Client {
        public static void main(String[] args) {
                Api api = Factory.createApi();
                api.test1("哈哈，不要紧张，只是个测试而已！");
        }
}
```

把上面的示例代码敲到电脑里面，测试一下，体会体会。

3.5 简单工厂的优缺点

- 帮助封装

  简单工厂虽然很简单，但是非常友好的帮助我们实现了组件的封装，然后让组件外部能真正面向接口编程。

- 解耦

  通过简单工厂，实现了客户端和具体实现类的解耦。

  如同上面的例子，客户端根本就不知道具体是由谁来实现，也不知道具体是如何实现的，客户端只是通过工厂获取它需要的接口对象。

- 可能增加客户端的复杂度

  如果通过客户端的参数来选择具体的实现类，那么就必须让客户端能理解各个参数所代表的具体功能和含义，这会增加客户端使用的难度，也部分暴露了内部实现，这种情况可以选用可配置的方式来实现。

- 不方便扩展子工厂

  私有化简单工厂的构造方法，使用静态方法来创建接口，也就不能通过写简单工厂类的子类来改变创建接口的方法的行为了。不过，通常情况下是不需要为简单工厂创建子类的。

3.6 思考简单工厂

1：简单工厂的本质

简单工厂的本质是：**选择实现。**

注意简单工厂的重点在选择，实现是已经做好了的。就算实现再简单，也要由具体的实现类来实现，而不是在简单工厂里面来实现。简单工厂的目的在于为客户端来选择相应的实现，从而使得客户端和实现之间解耦，这样一来，具体实现发生了变化，就不用变动客户端了，这个变化会被简单工厂吸收和屏蔽掉。

实现简单工厂的难点就在于 “如何选择” 实现，前面讲到了几种传递参数的方法，那都是静态的参数，还可以实现成为动态的参数。比如：在运行期间，由工厂去读取某个内存的值，或者是去读取数据库中的值，然后根据这个值来选择具体的实现等等。

2：何时选用简单工厂

建议在如下情况中，选用简单工厂：

- 如果想要完全封装隔离具体实现，让外部只能通过接口来操作封装体，那么可以选用简单工厂，让客户端通过工厂来获取相应的接口，而无需关心具体实现
- 如果想要把对外创建对象的职责集中管理和控制，可以选用简单工厂，一个简单工厂可以创建很多的、不相关的对象，可以把对外创建对象的职责集中到一个简单工厂来，从而实现集中管理和控制

3.7 相关模式

- 简单工厂和抽象工厂模式

  简单工厂是用来选择实现的，可以选择任意接口的实现，一个简单工厂可以有多个用于选择并创建对象的方法，多个方法创建的对象可以有关系也可以没有关系。

  抽象工厂模式是用来选择产品簇的实现的，也就是说一般抽象工厂里面有多个用于选择并创建对象的方法，但是这些方法所创建的对象之间通常是有关系的，这些被创建的对象通常是构成一个产品簇所需要的部件对象。

  所以从某种意义上来说，简单工厂和抽象工厂是类似的，如果抽象工厂退化成为只有一个实现，不分层次，那么就相当于简单工厂了。
- 简单工厂和工厂方法模式

  简单工厂和工厂方法模式也是非常类似的。

  工厂方法的本质也是用来选择实现的，跟简单工厂的区别在于工厂方法是把选择具体实现的功能延迟

到子类去实现。

如果把工厂方法中选择的实现放到父类直接实现，那就等同于简单工厂。

- 简单工厂和能创建对象实例的模式

  简单工厂的本质是选择实现，所以它可以跟其它任何能够具体的创建对象实例的模式配合使用，比如：单例模式、原型模式、生成器模式等等。

简单工厂模式结束，谢谢观赏

1.7 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：

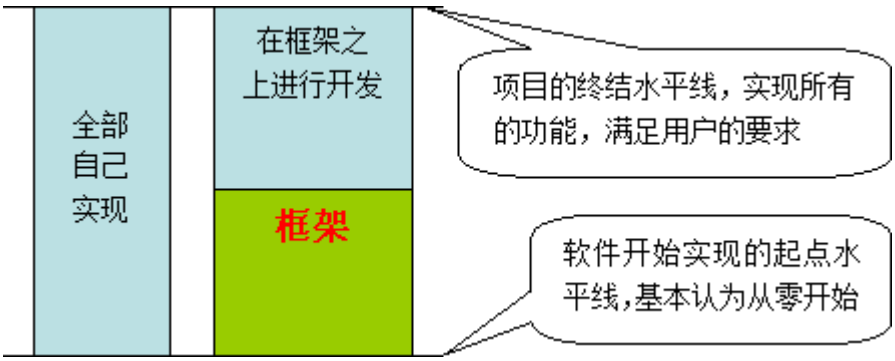


图1 应用程序和框架的简单关系示意图

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：

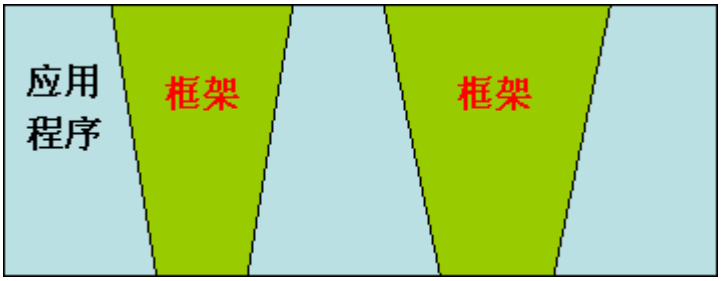


图2 应用程序和框架的关系示意图

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.....

1.8 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：

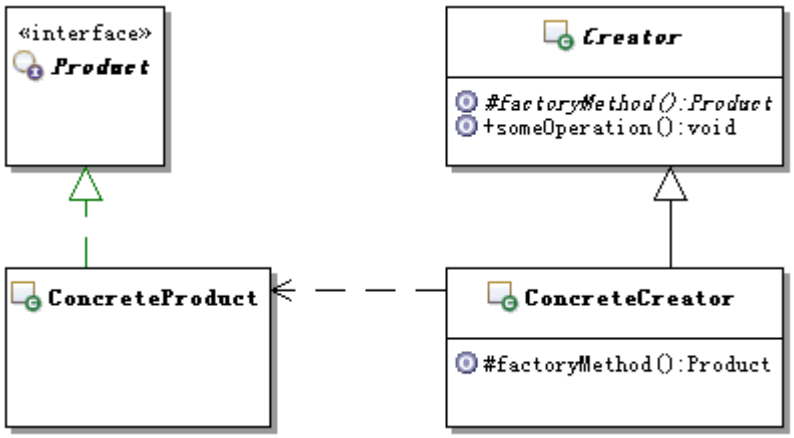


图3 工厂方法模式结构示意图

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：

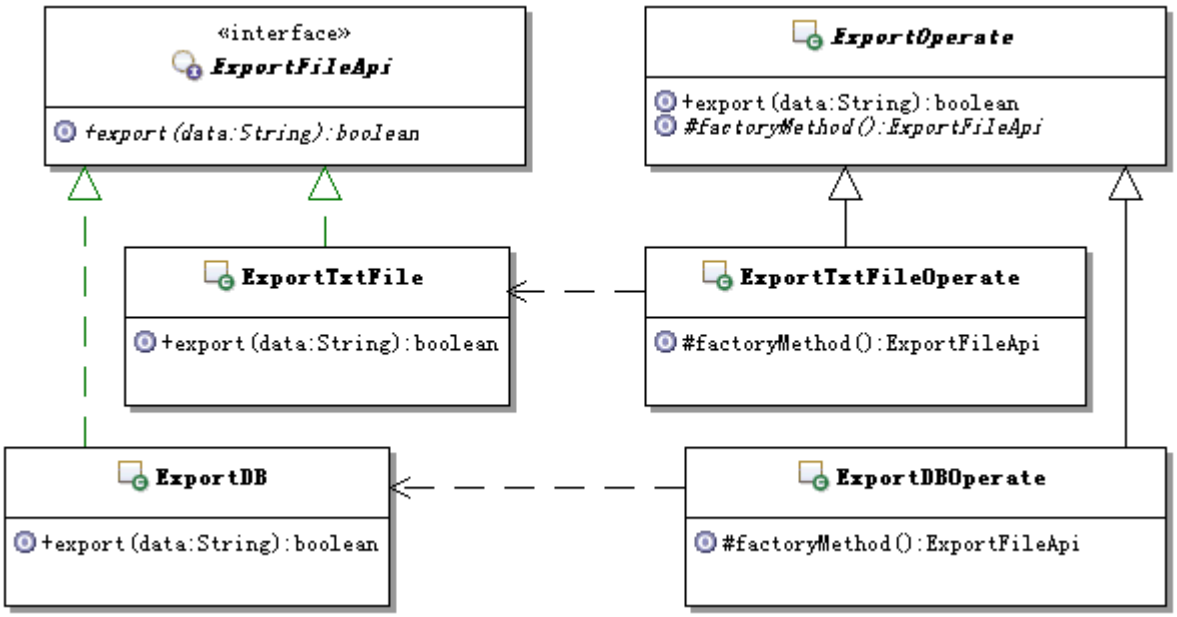


图4 使用工厂模式来实现示例的程序结构示意图

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……

1.9 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：

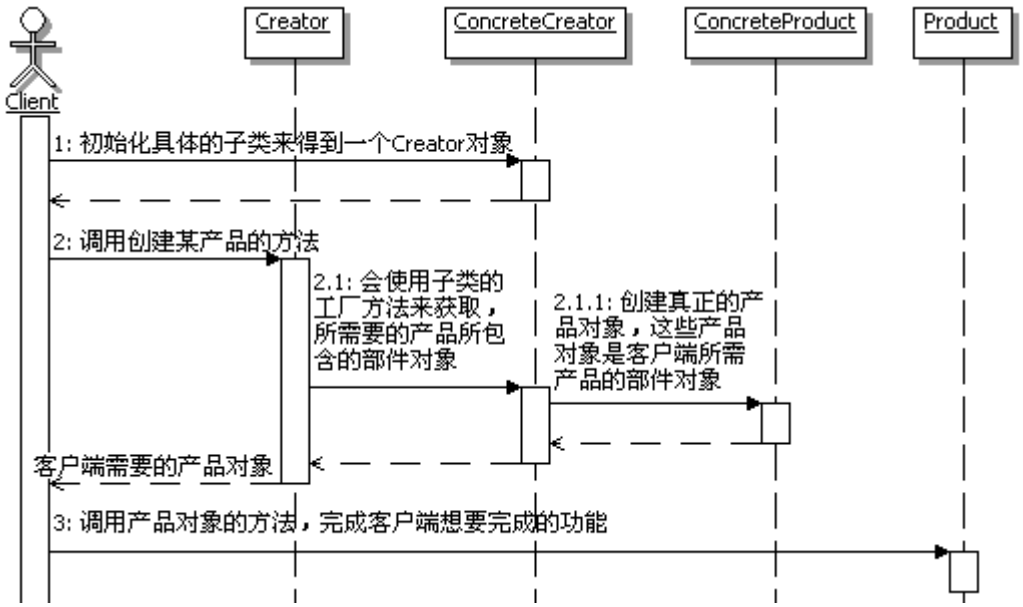


图5 客户端使用由Creator创建出来的对象的调用顺序示意图

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：

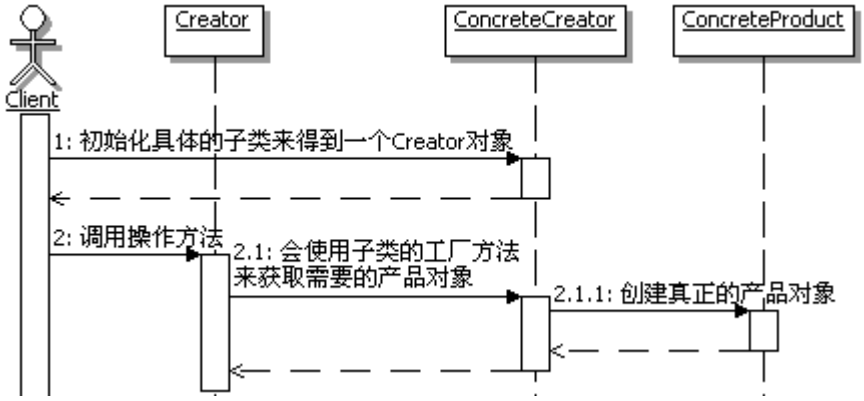


图6 客户端使用Creator对象的调用顺序示意图

未完待续.....

1.10 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：

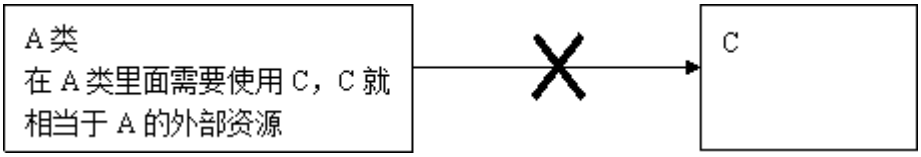


图8 A类不再主动创建C

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：

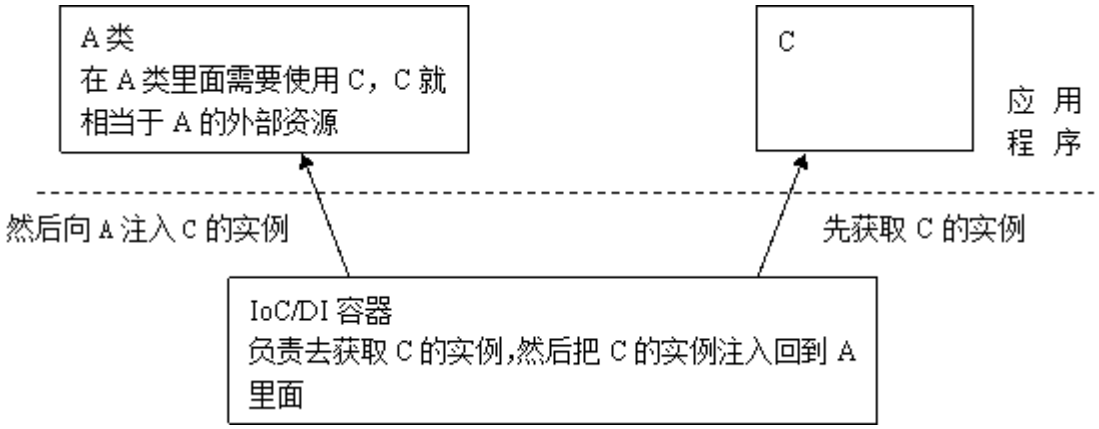


图9 有IoC/DI容器后程序结构示意图

（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.....

1.11 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：

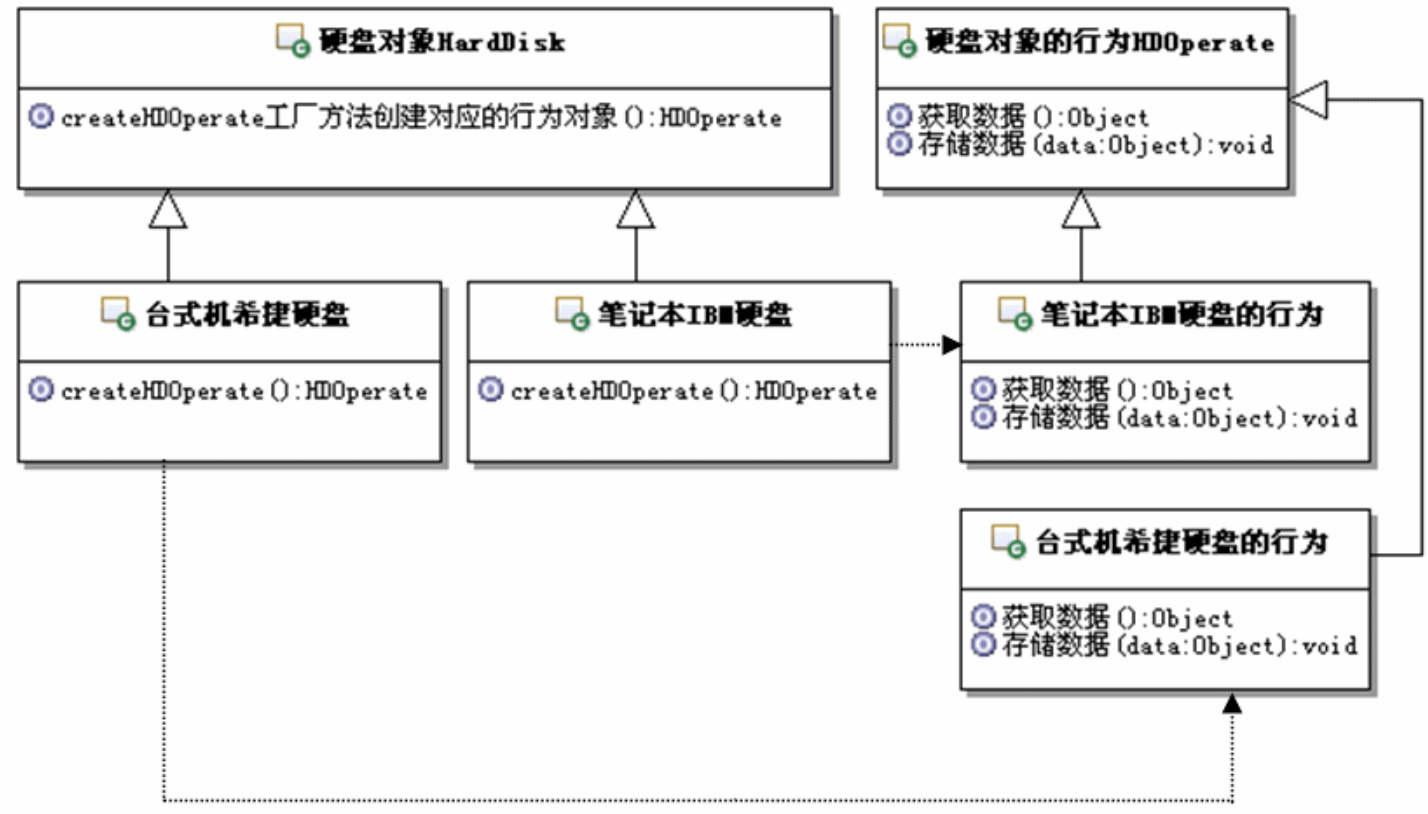


图10 平行的类层次结构示意图

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!

1.12 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：

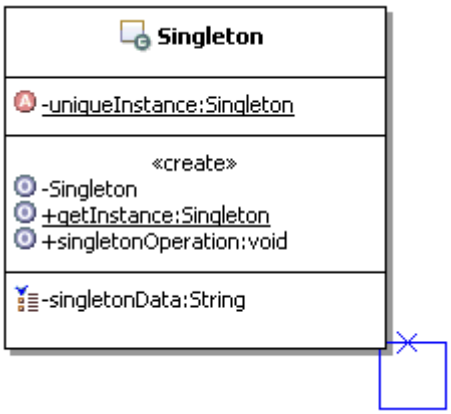


图1 单例模式结构图

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

1.13 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：

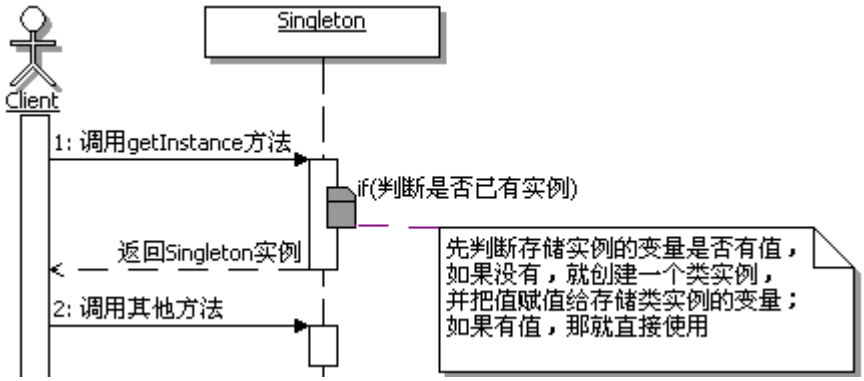


图2 懒汉式调用顺序示意图

饿汉式的调用顺序，如图3所示：

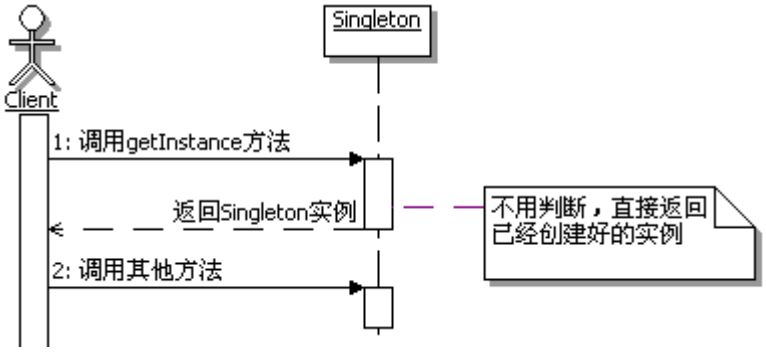


图3 饿汉式调用顺序示意图

未完待续

1.14 研磨设计模式之单例模式-3

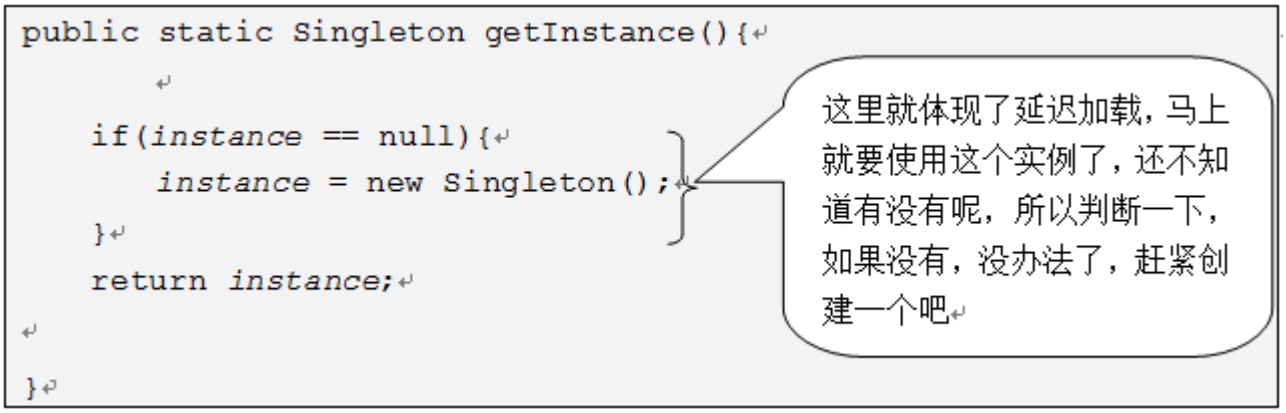
发表时间: 2010-08-02

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：



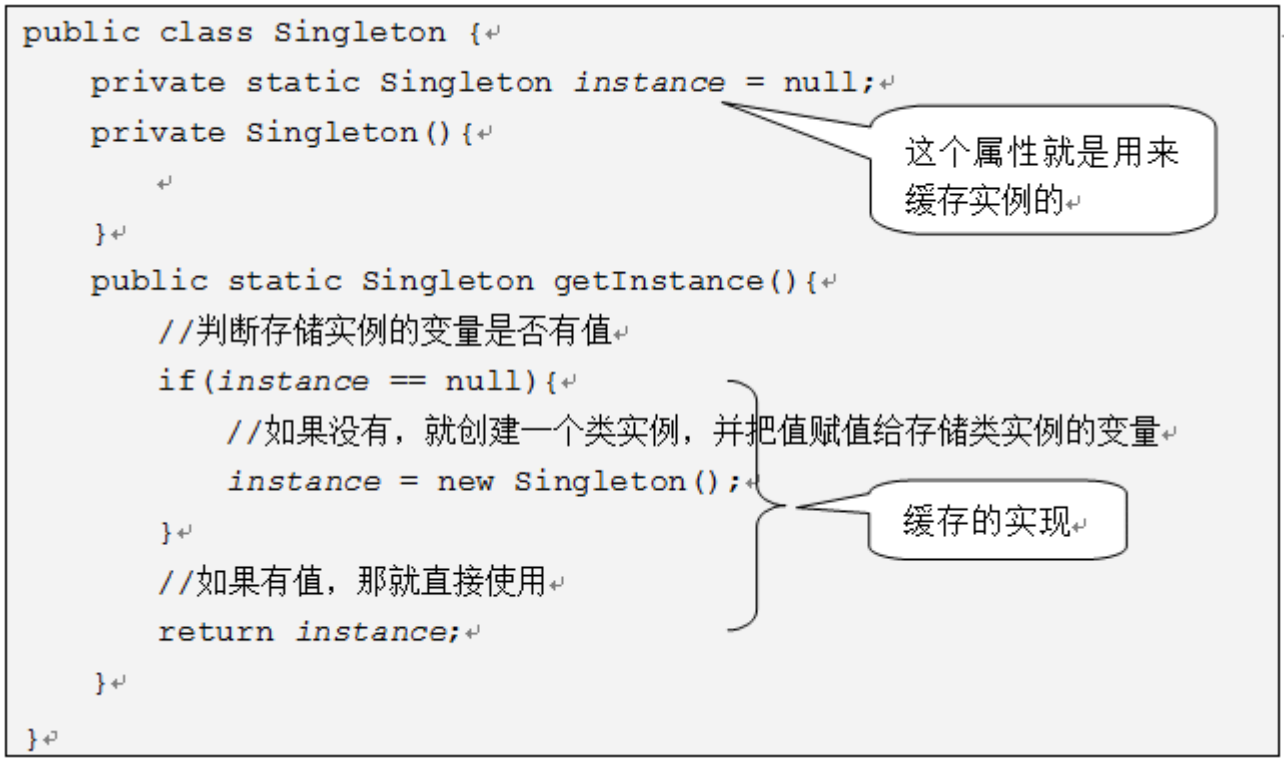
3.4 缓存的思想

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

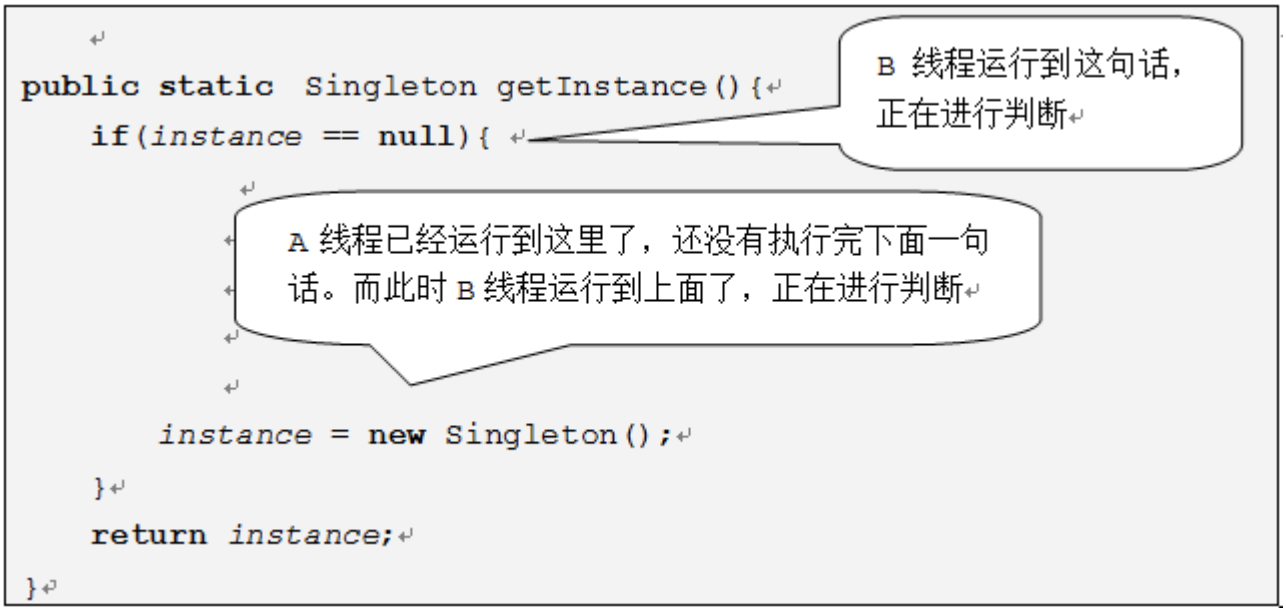
1：时间和空间

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){
        instance = new Singleton();
    }
    return instance;
}
```

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

A 线程正在创建实例

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：

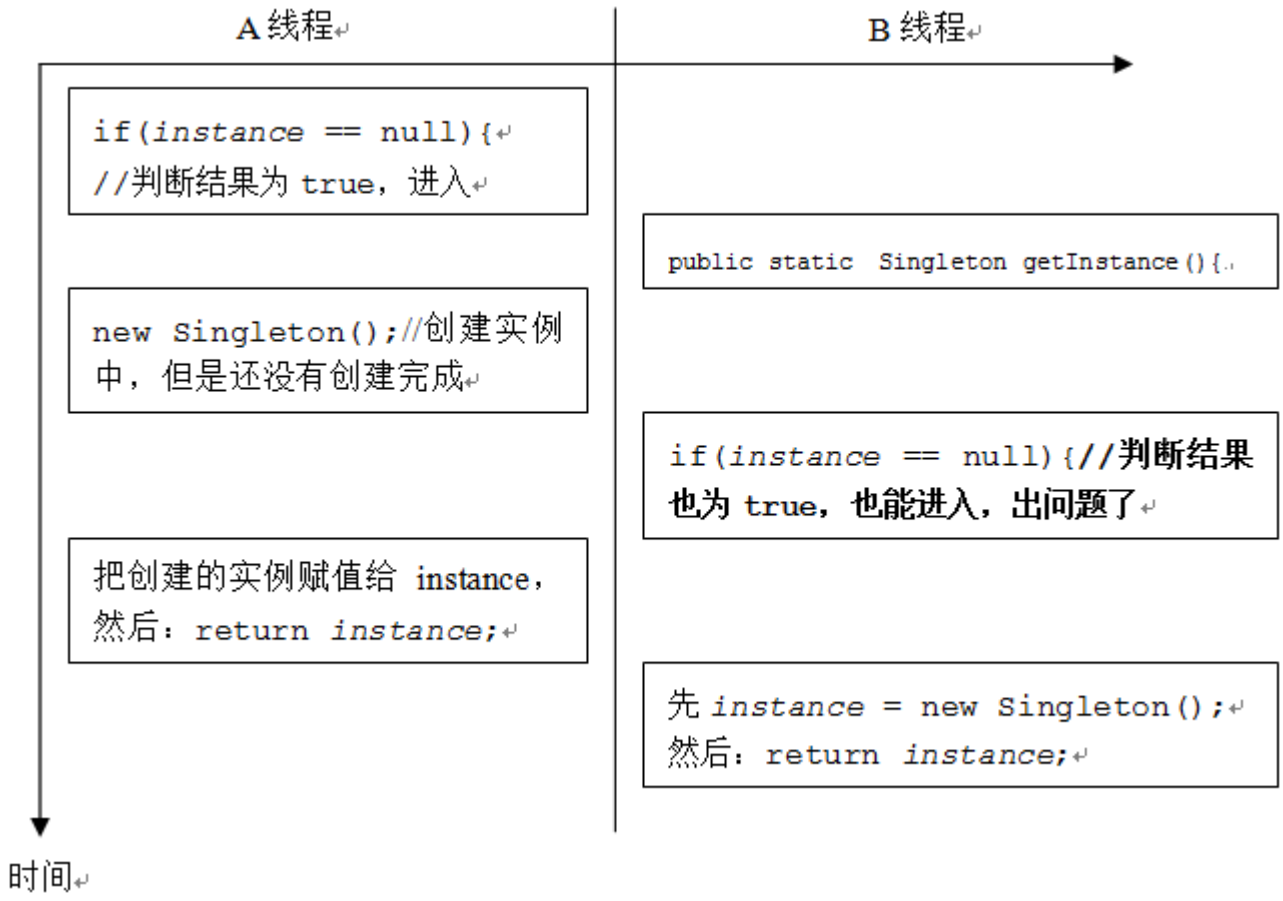


图4 懒汉式单例的线程问题示意图

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

1.15 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

1.16 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：

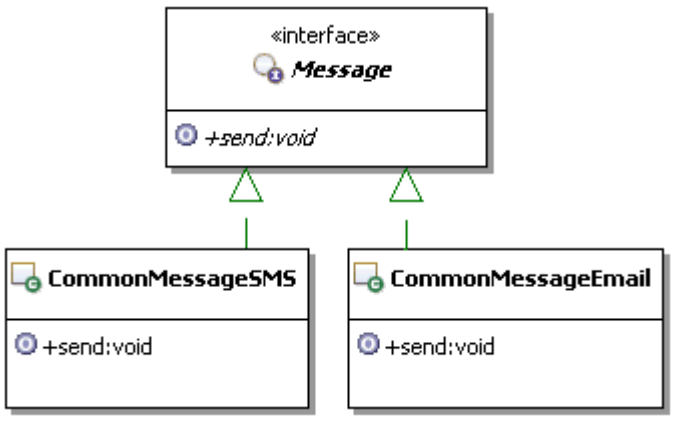


图1 简化版本的系统结构示意图

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：

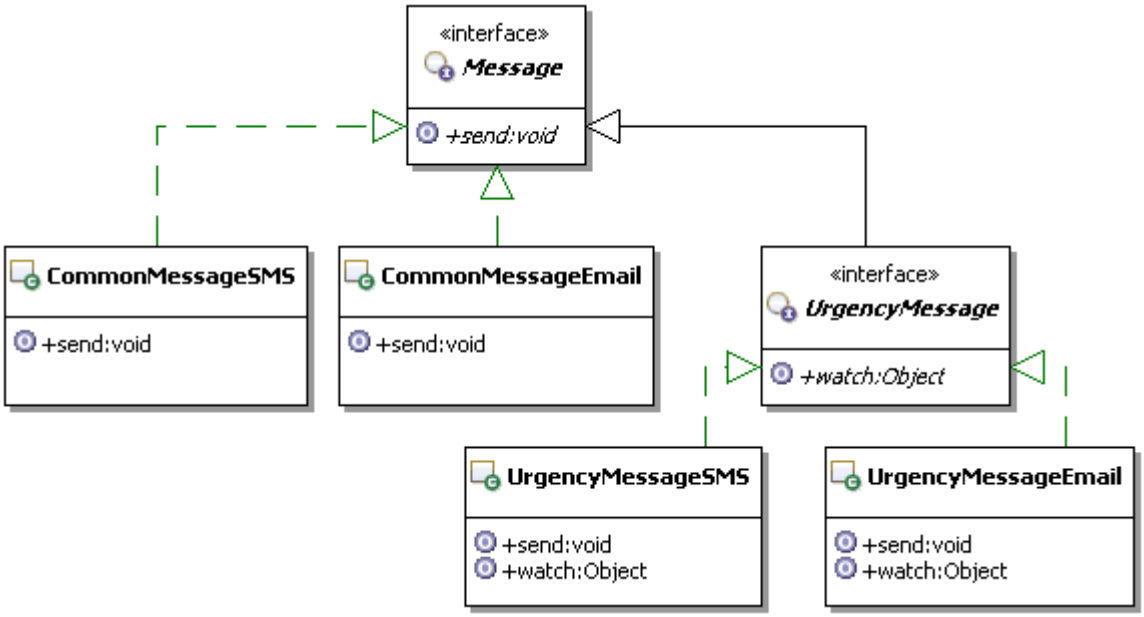


图2 加入发送加急消息后的系统结构示意图

（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：

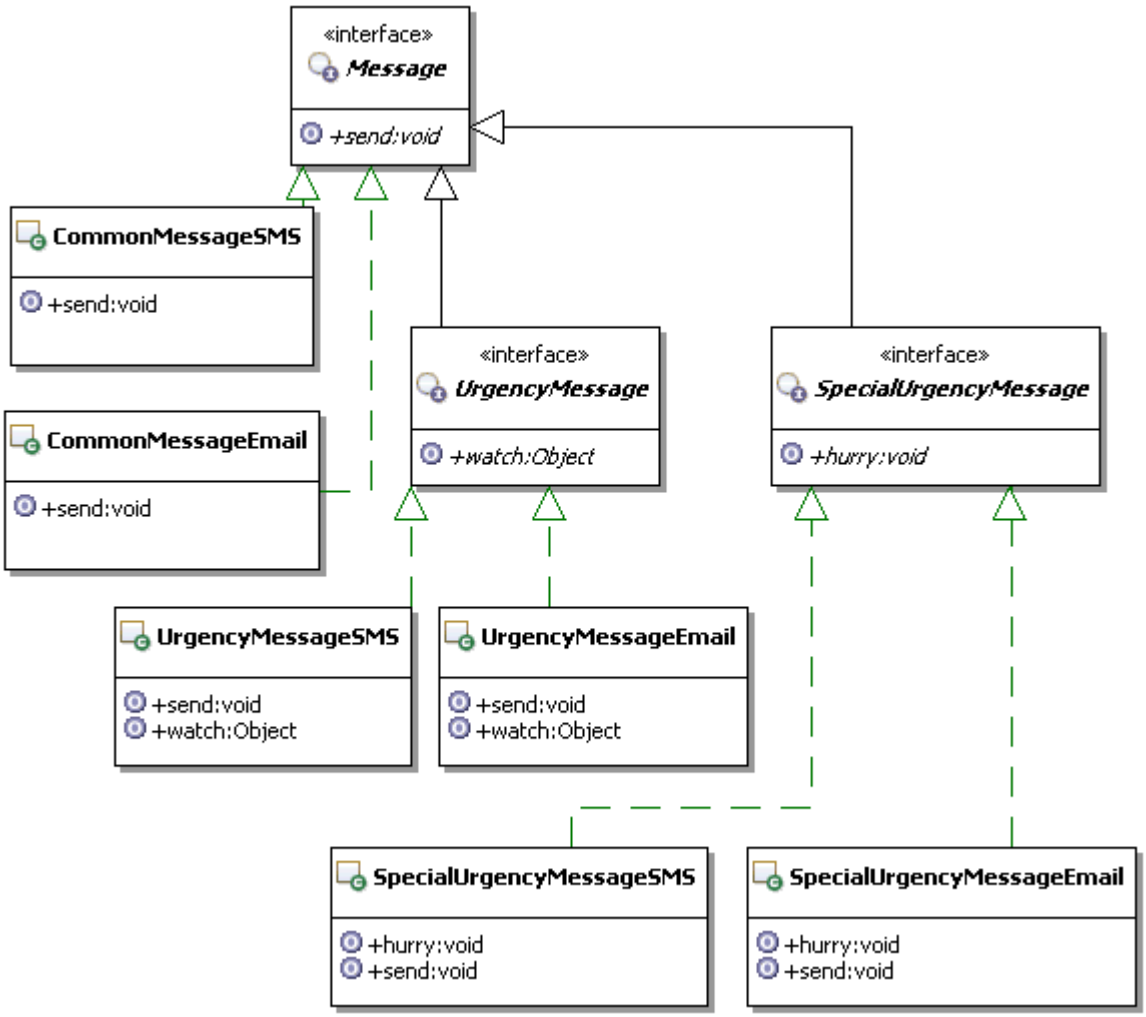


图3 加入发送特急消息后的系统结构示意图

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：

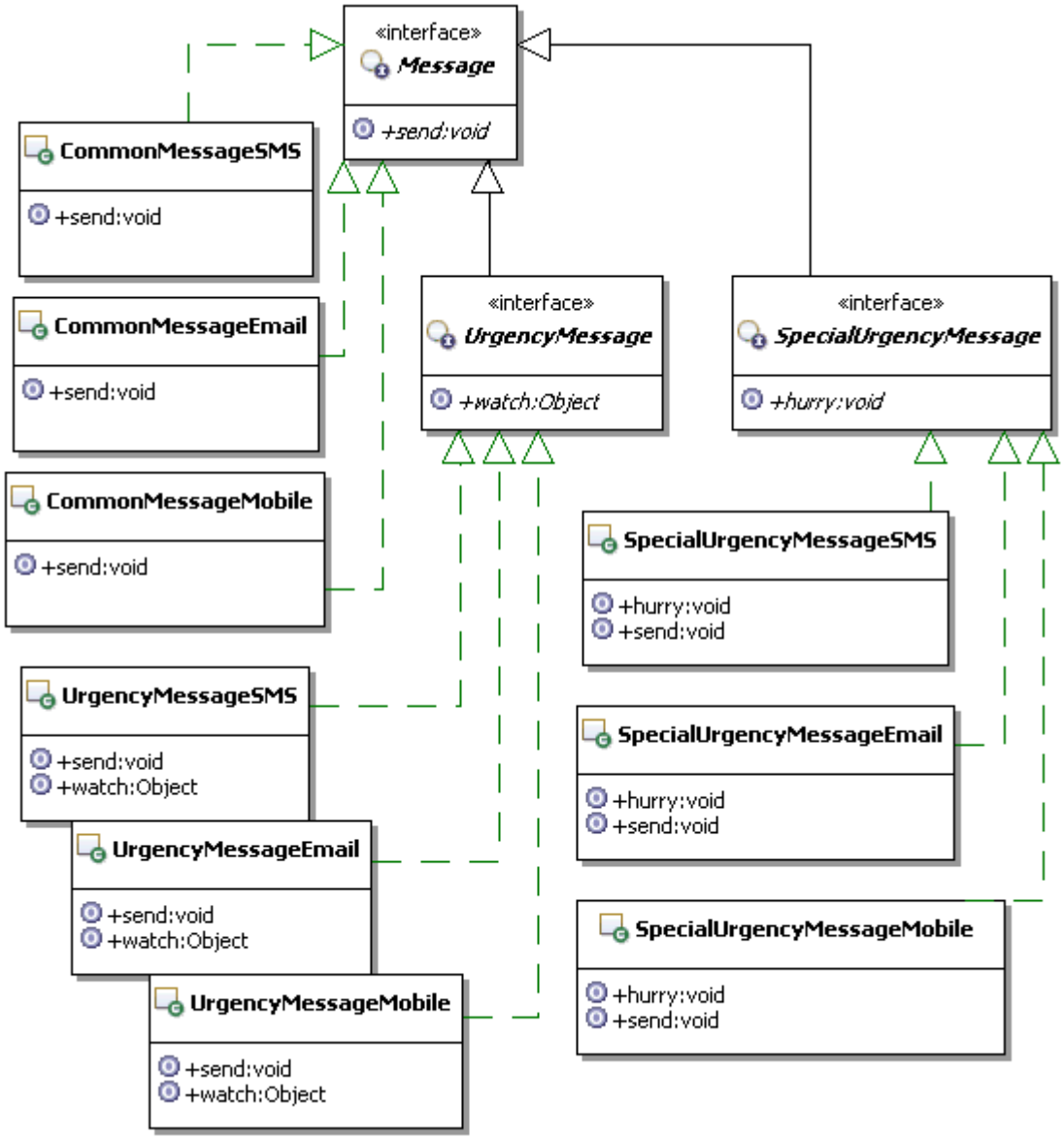


图4 加入发送手机消息后的系统结构示意图

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=========

1.17 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：

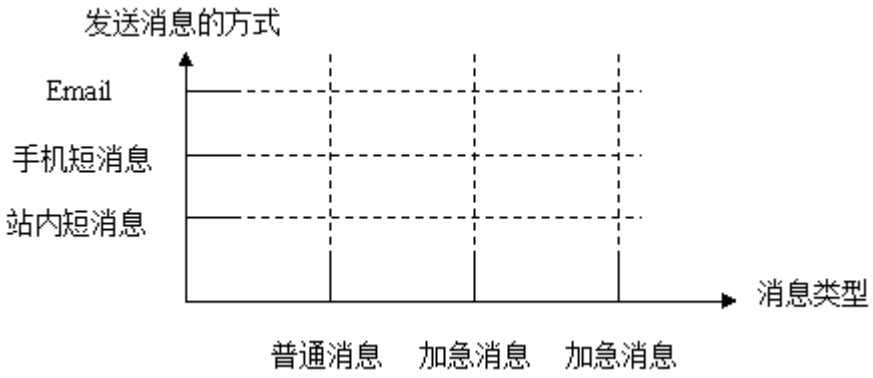


图5 发送消息的可能性的组合示意图

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：

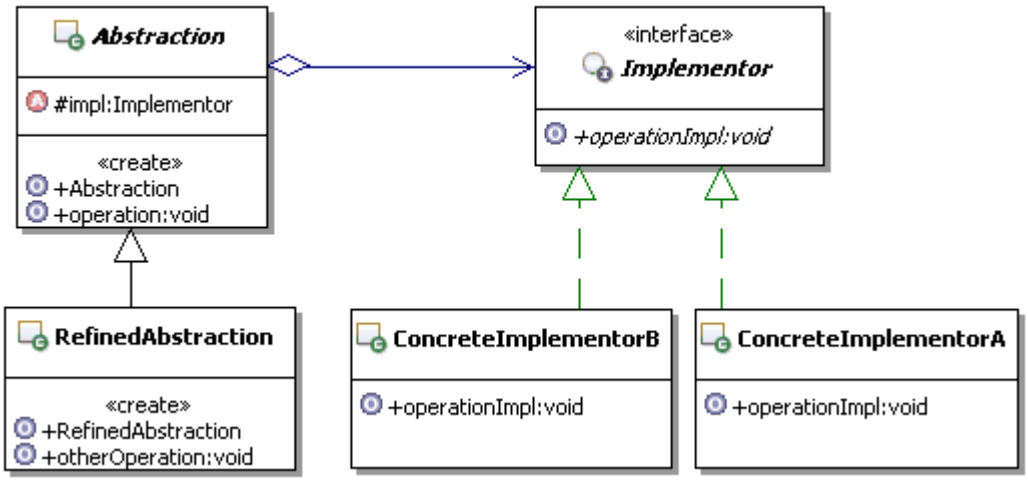


图6 桥接模式的结构示意图

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

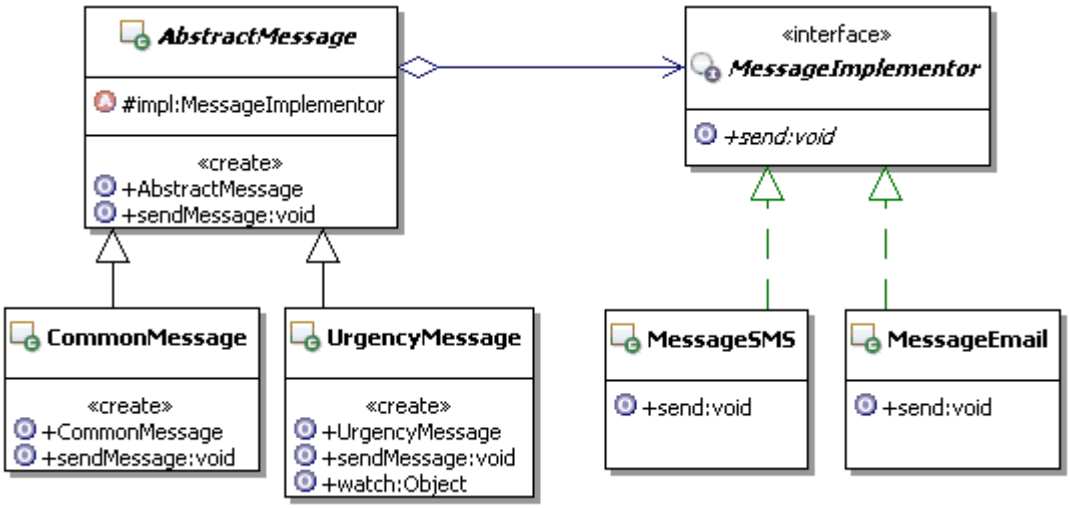


图7 使用桥接模式来实现简单功能示例的程序结构示意图

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

1.18 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：

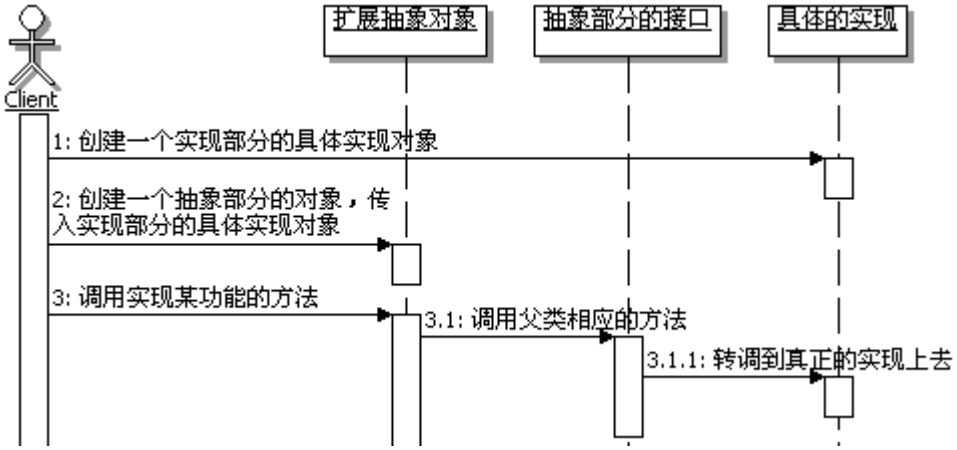


图8 桥接模式的调用顺序示意图

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：

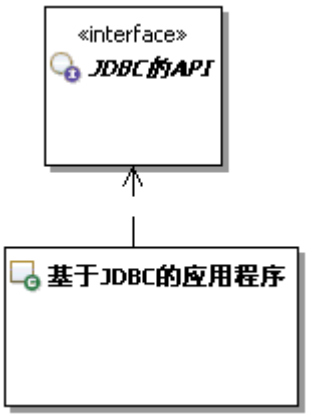


图9 基于JDBC开发的应用程序结构示意图

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：

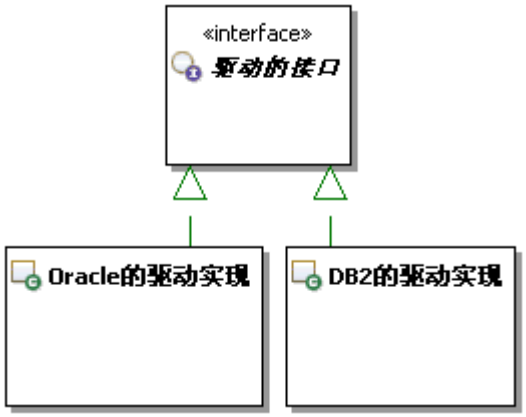


图10 驱动程序实现结构示意图

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：

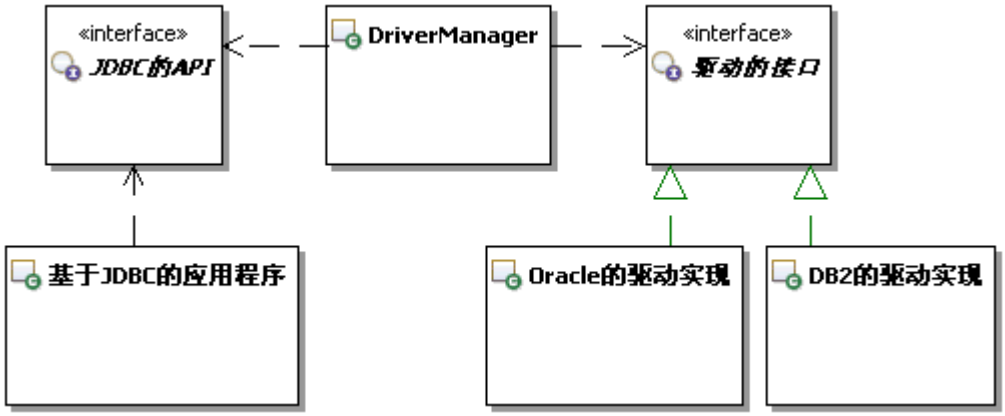


图11 JDBC的结构示意图

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

1.19 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：

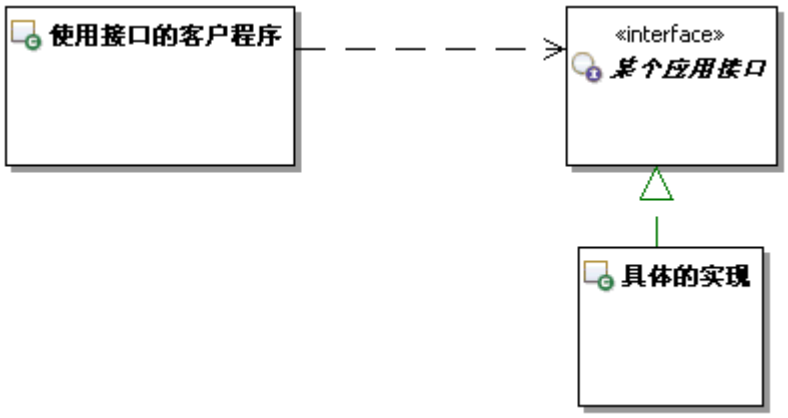


图12 使用接口的程序结构示意图

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：

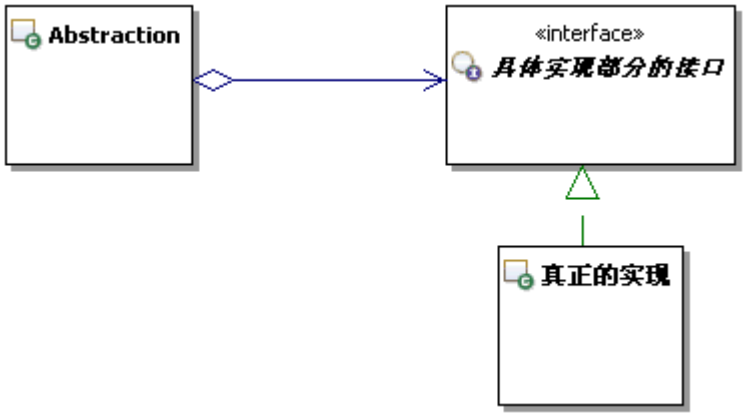


图13 简化的桥接模式结构示意图

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：

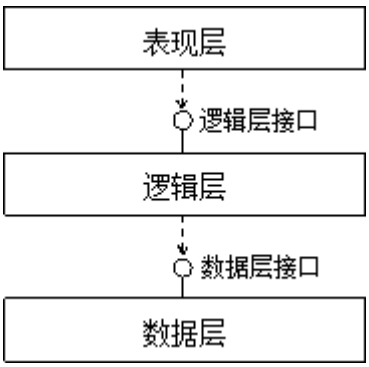


图14 基本的三层架构示意图

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：

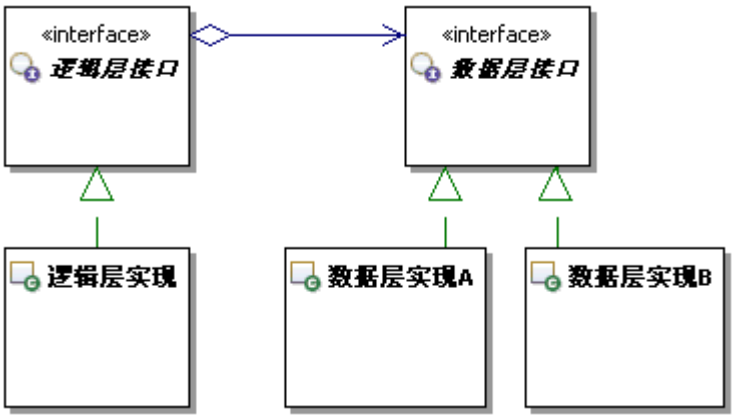


图15 逻辑层和数据层的程序结构示意图

再看看表现层和逻辑层之间的结构示意，如图16所示：

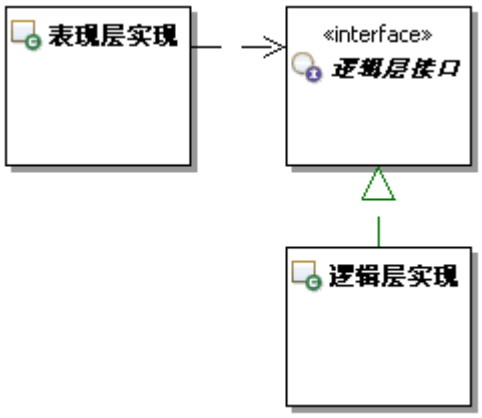


图16 表现层和逻辑层的结构示意图

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：

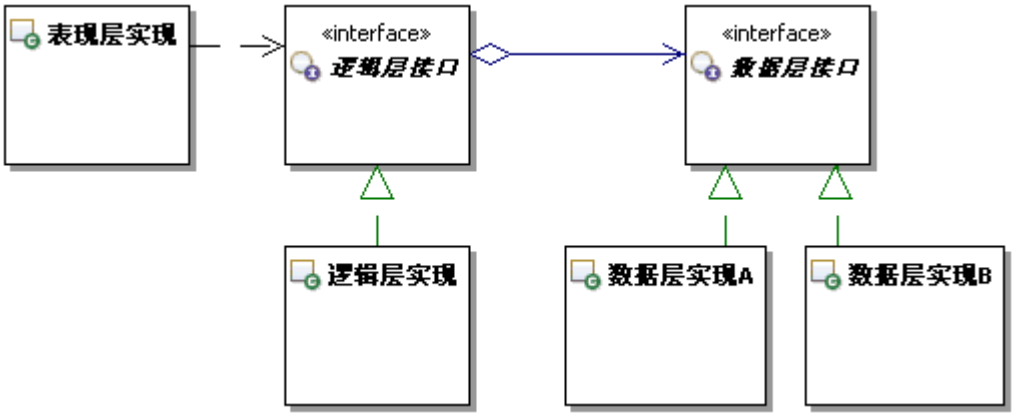


图17 三层结合的结构示意图

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：

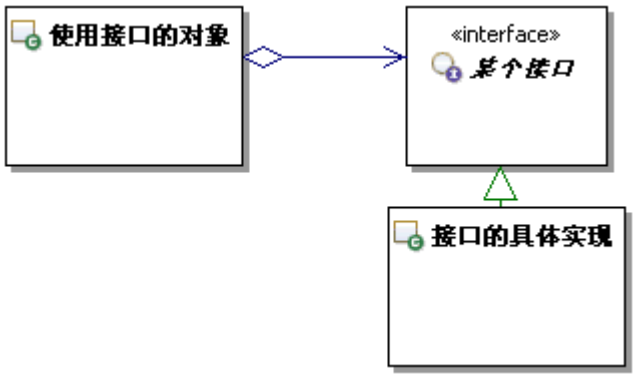


图18 桥接模式和策略模式结构示意图

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!