

重要声明

1: 本文并非本人原创, 原创作者为 JavaEye 上的一位大牛人, 网名为 chjavach, 博客地址为: <http://chjavach.javaeye.com/>

2: 这位牛人在 JavaEye 发表一系列关于设计模式的文章, 广受欢迎, 是 JavaEye 网站上设计模式类的第一牛人, 几乎每篇博客, 都在 JavaEye 的博客首页长时间排第一, 超受欢迎。

3: 个人觉得是看过的设计模式类最好的了, 特为它制作电子书, 跟大家分享。其实就是直接从网页上把内容 copy 下来, 再稍稍排了一下版, 原有的文字, 一个字都没敢动, 保证原汁原味。

4: 如果大家喜欢, 我会陆续把这位大牛的其他关于设计模式的文章都制作成电子书, 跟大家分享, 这次跟大家分享工厂方法模式。

部分 JavaEye 网友对工厂方法模式的评论

直接从该牛人博客工厂方法模式部分的评论里面摘录几个, 看看 JavaEye 网友的评价:

[qianfu_zhi](#) 2010-06-25 [引用](#)

框架的知识写得很深入透彻, 好文章

[tzm1984](#) 2010-07-02 [引用](#)

本来还想买本《java 设计模式》看看
看样子现在不需要了 哈哈。定 LZ。

[jintui](#) 2010-08-01 [引用](#)

superheizai 写道

呵呵, 下一本设计模式的书将在这里诞生。

强烈同意楼上兄弟。

追着楼主的文章看, 写得太好了, 绝对是我看过的 No.1, 这么有质量的内容要是能够出书的话, 我一定鼎力支持, 第一时间去买一本来细细研究!

现在很多中文作者, 真能写出点东西的不多, 尤其是写出些自己的东西, 大多数都是东拼西凑, 根本没货, 就是吹牛、骗人, 比如那个有名的李 X。

楼主水平比他们高出太多了, 好好写本书, 也让我们知道, 中文作者的书也是可看的,

虽然少了点, 呵呵。支持楼主出书, 出书我必买! 🍌

[zhmiao](#) 2010-08-03 [引用](#)

一个工作十年的总结, 对于我们这后人来说是精华中的果实, 因为之中酝酿了实实在在走过来的理解, 而楼主这种乐于奉献的精神, 也让我们深深的折服, 而现实中碰到的大多工作十年的人, 很少还在谈论技术、或者从来不屑于给后来者讲解下基础的知识, 而往往基础的知识百看不厌, 每一次看设计模式都有新的认识、新的理解。

thanks!

[ji0000](#) 2010-08-17 [引用](#)

文章很生动，形象。 文章结构也很好。👍 牛 b

[matychen](#) 2010-07-28 [引用](#)

真是 javaeye 上最火的博客啊～～每篇都看了。

[dakaiopen](#) 2010-06-21 [引用](#)

对工厂方法模式的理论讲解很好,很少看到这么有质量的文章了

[weijiezhimi](#) 2010-06-20 [引用](#)

真是好,这个讲解很是透彻,而且切中要点.以前有些地方一直迷惑,都被解惑了.

尤其是第 5 点"谁来使用工厂方法创建的对象",以前都没有思考过.

PS:这是我看过的,对工厂方法模式分析得最到位的文章了

[fansfirst2008](#) 2010-06-17 [引用](#)

一直未对工厂模式的精髓没用捉摸透

非常感谢楼主精彩的这么一课!

[liuyupy](#) 2010-07-12 [引用](#)

该系列的讲解精彩细致,唯一不足之处就是示例不具备应用场景的代表性(或是没体现出来,当然可以通过想象创建需求),若是在最后部分其它框架相关应用 浅示 ,会让绕梁之味更浓.

论坛中讲模式的文章能 深广兼顾 新老皆懂 的,无出其右.再赞.

[clctcx](#) 2010-06-30 [引用](#)

看完工厂方法模式了,写得太棒了,鼓掌👏

[javaisgod](#) 2010-07-09 [引用](#)

博主, 小弟做了一年 java 开发, 设计模式就是听说过的水平。认真拜读博主关于模式设计的文章, 甚是喜欢: 一是喜欢文章的结构, 提出问题, 分析问题, 引入方法, 解决问题, 最

后是心得体会; 二是喜欢语言朴实, 内容丰富。这里期待续集👍

[joknm](#) 2010-07-20 [引用](#)

意犹未尽, 期待楼主的下回分解。

以下是原始博文

研磨设计模式之工厂方法模式-1

原文地址：<http://chjavach.javaeye.com/blog/692790>

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：

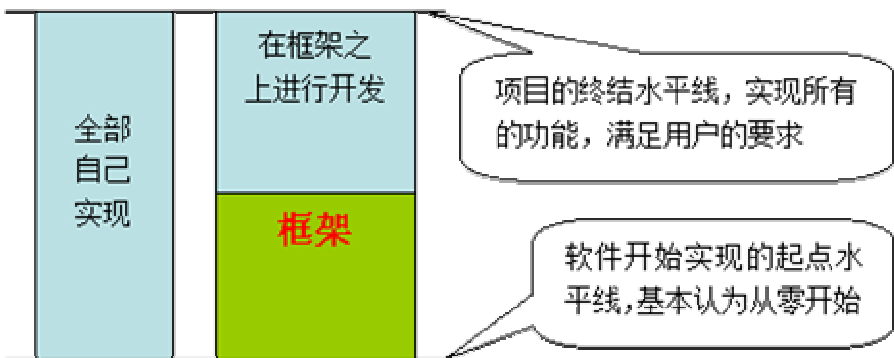


图 1 应用程序和框架的简单关系示意图

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：

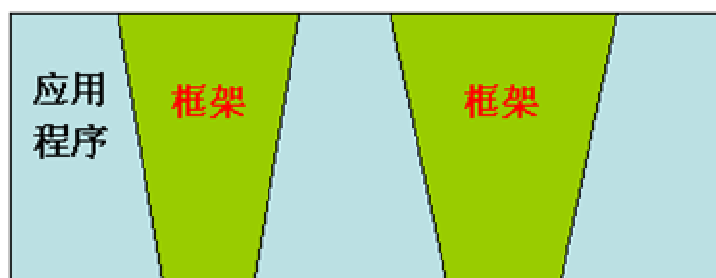


图 2 应用程序和框架的关系示意图

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，精彩稍后继续

研磨设计模式之工厂方法模式-2

原文地址：<http://chjavach.javaeye.com/blog/692791>

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：

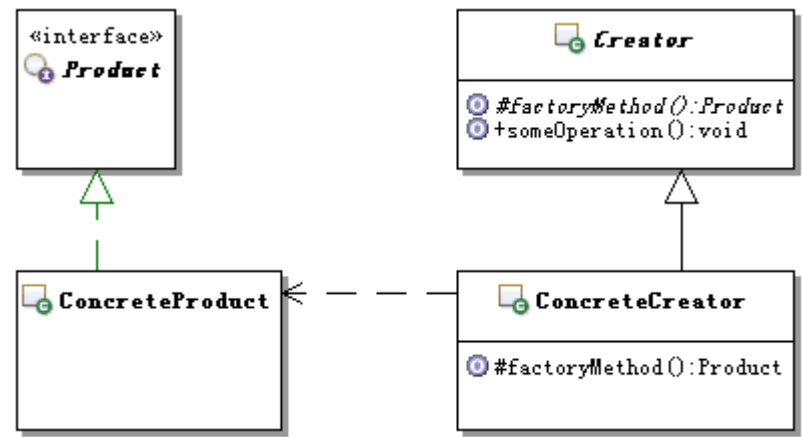


图 3 工厂方法模式结构示意图

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 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：

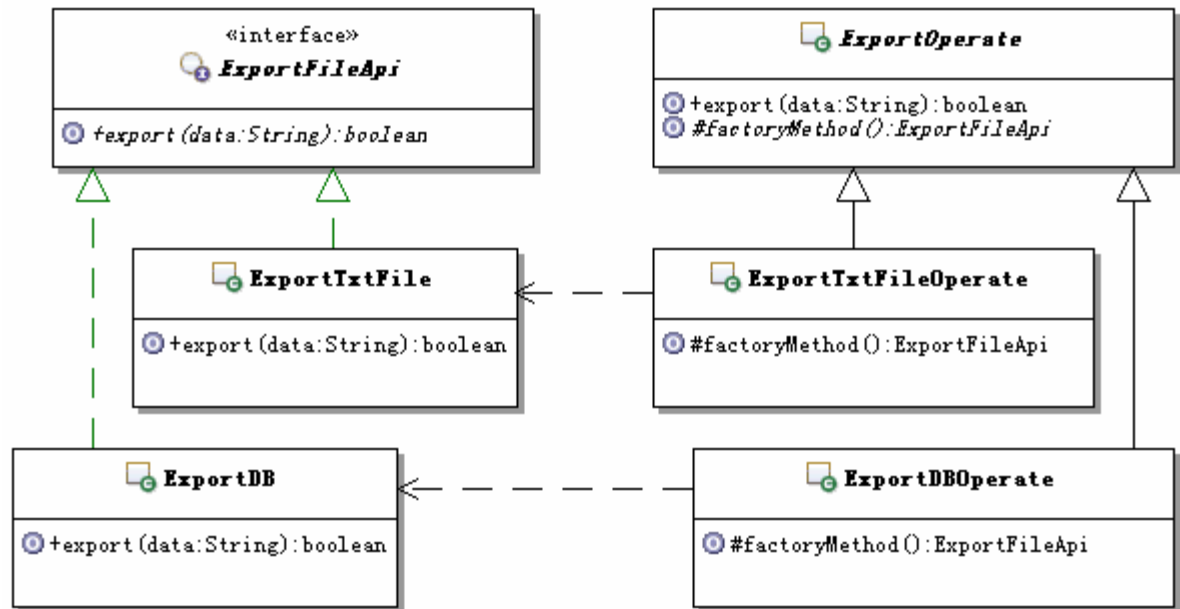


图 4 使用工厂模式来实现示例的程序结构示意图

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

研磨设计模式之工厂方法模式-3

原文地址：<http://chjavach.javaeye.com/blog/692792>

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
     * 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();

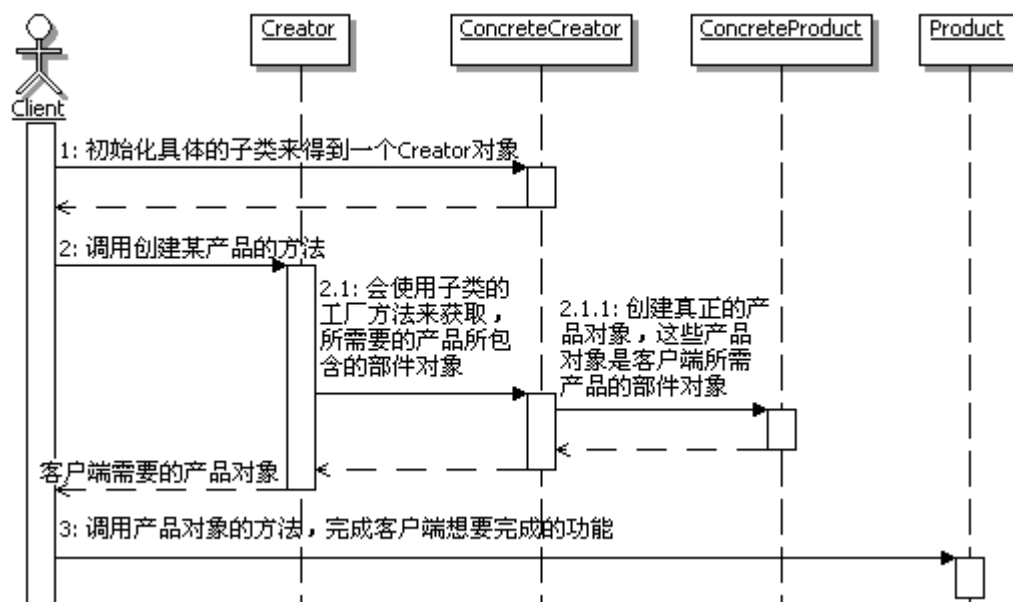
        //工厂方法创建的对象是创建客户端对象所需要的
        Product p = new ConcreteProduct();
        p.setProduct1(p1);
        p.setProduct2(p2);

        return p;
    }
}
```

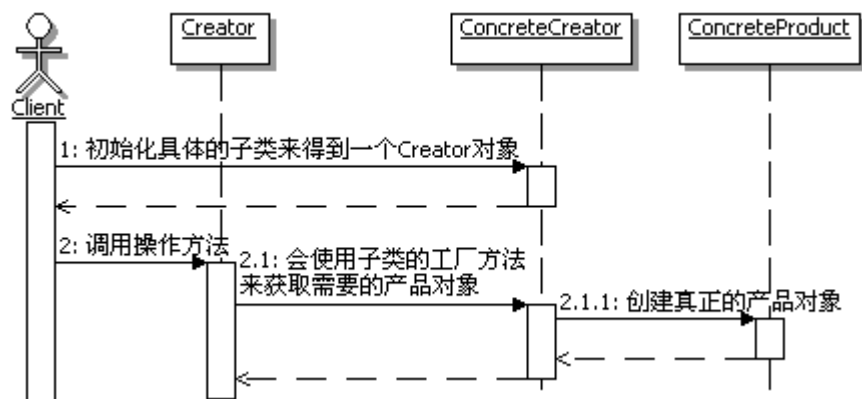
小结一下：在工厂方法模式里面，客户端要么使用 Creator 对象，要么使用 Creator 创建的对象，一般客户端不直接使用工厂方法。当然也可以直接把工厂方法暴露给客户端操作，但是一般不这么做。

(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：



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：



未完待续

研磨设计模式之工厂方法模式-4

原文地址：<http://chjavach.javaeye.com/blog/692793>

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:

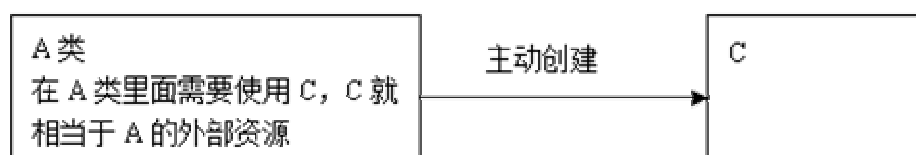


图 7 常规 A 使用 C 示意图

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:

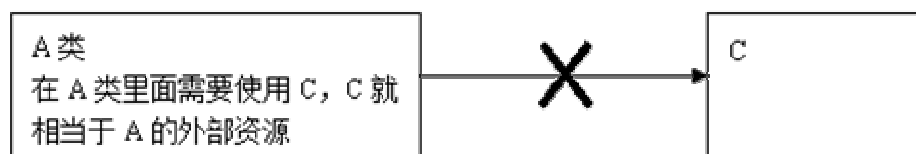


图 8 A 类不再主动创建 C

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:

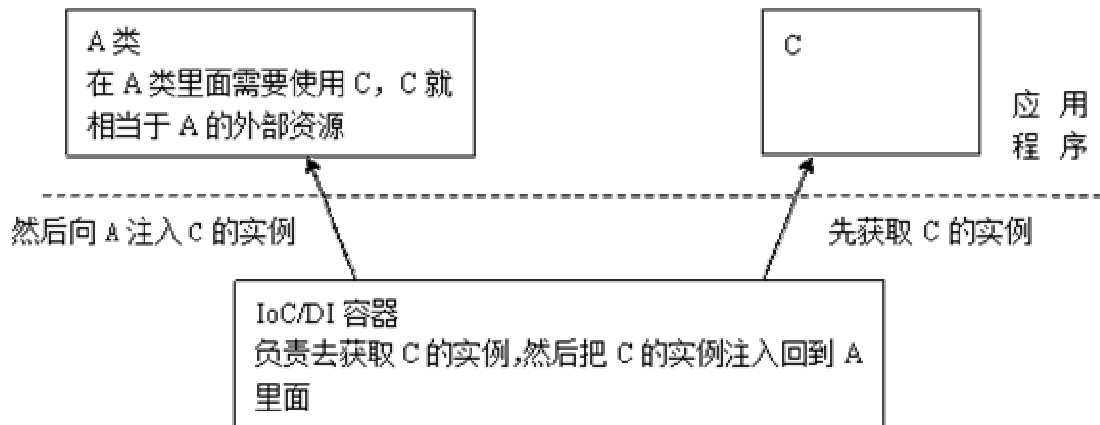


图9 有 IoC/DI 容器后程序结构示意图

(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

研磨设计模式之工厂方法模式-5

原文地址：<http://chjavach.javaeye.com/blog/694864>

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：

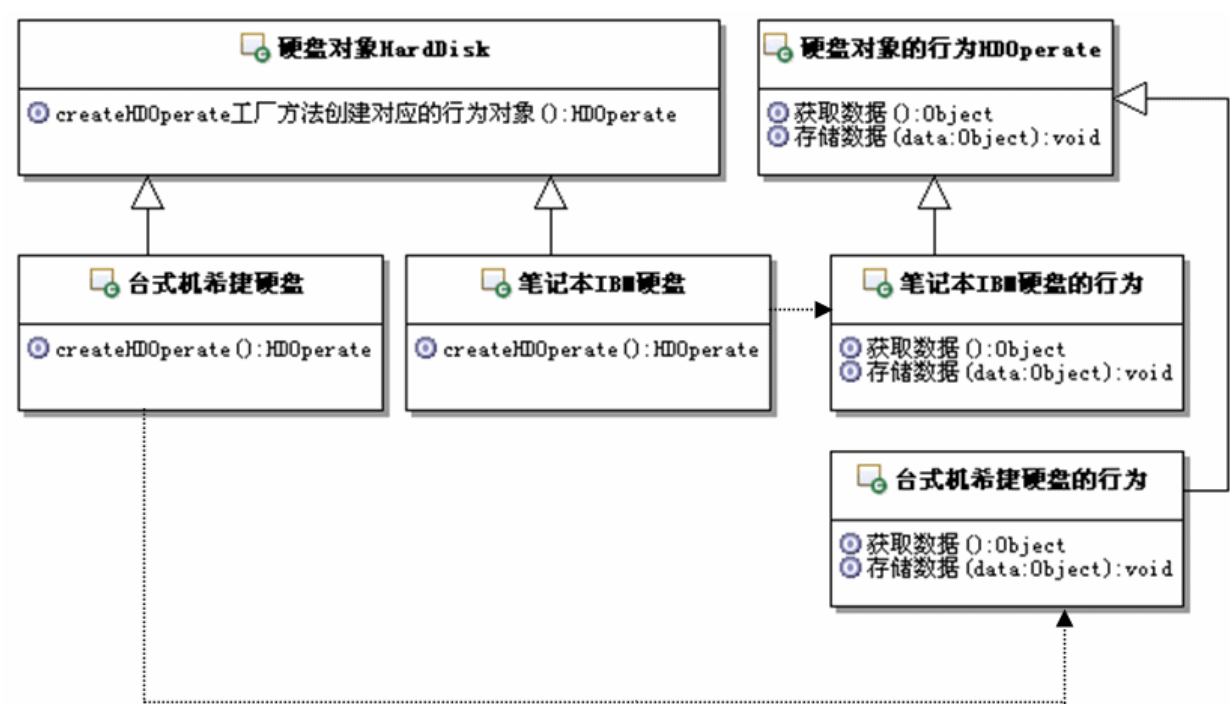


图 10 平行的类层次结构示意图

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!