第5章

多态工厂的实现——工厂方法模式

简单工厂模式虽然简单，但存在一个很严重的问题：当系统中需要引入新产品时，由于静态工厂方法通过所传入参数的不同来创建不同的产品，这必定要修改工厂类的源代码，将违背开闭原则。如何实现增加新产品而不影响已有代码？工厂方法模式为此应运而生。本章将介绍第 2 种工厂模式——工厂方法模式。

5.1 日志记录器的设计

> Sunny 软件公司欲开发一个系统运行日志记录器(Logger)，该记录器可以通过多种途径保存系统的运行日志，例如通过文件记录或数据库记录，用户可以通过修改配置文件灵活地更换日志记录方式。在设计各类日志记录器时，Sunny 公司的开发人员发现需要对日志记录器进行一些初始化工作，初始化参数的设置过程较为复杂，而且某些参数的设置有严格的先后次序，否则可能会发生记录失败。如何封装记录器的初始化过程并保证多种记录器切换的灵活性是 Sunny 公司开发人员面临的一个难题。

Sunny 公司的开发人员通过对该需求进行分析，发现该日志记录器有如下两个设计要点：

(1) 需要封装日志记录器的初始化过程，这些初始化工作较为复杂。例如需要初始化其他相关的类，还有可能需要配置工作环境(例如连接数据库或创建文件)，导致代码较长。如果将它们都写在构造函数中，会导致构造函数庞大，不利于代码的修改和维护。

(2) 用户可能需要更换日志记录方式，在客户端代码中需要提供一种灵活的方式来选择日志记录器，尽量在不修改源代码的基础上更换或者增加日志记录方式。

Sunny 公司开发人员最初使用简单工厂模式对日志记录器进行了设计，初始结构如图 5-1 所示。

在图 5-1 中，LoggerFactory 充当创建日志记录器的工厂，提供了工厂方法 createLogger()用于创建日志记录器，Logger 是抽象日志记录器接口，其子类为具体日志记录器。其中，工厂类 LoggerFactory 代码片段如下：

```
//日志记录器工厂
class LoggerFactory {
```

```
        //静态工厂方法
        public static Logger createLogger(String args) {
            if(args.equalsIgnoreCase("db")) {
                //连接数据库,代码省略
                //创建数据库日志记录器对象
                Logger logger = new DatabaseLogger();
                //初始化数据库日志记录器,代码省略
                return logger;
            }
            else if(args.equalsIgnoreCase("file")) {
                //创建日志文件,代码省略
                //创建文件日志记录器对象
                Logger logger = new FileLogger();
                //初始化文件日志记录器,代码省略
                return logger;
            }
            else {
                return null;
            }
        }
    }
```

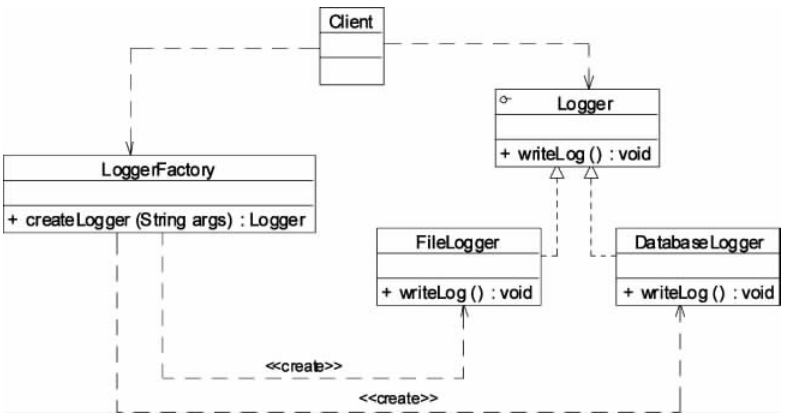

图 5-1　基于简单工厂模式设计的日志记录器结构图

为了突出设计重点，上述代码进行了简化，省略了具体日志记录器类的初始化代码。在LoggerFactory 类中提供了静态工厂方法 createLogger()，用于根据所传入的参数创建各种不同类型的日志记录器。通过使用简单工厂模式，将日志记录器对象的创建和使用分离，客户端只需使用由工厂类创建的日志记录器对象即可，无须关心对象的创建过程。

但是，虽然简单工厂模式实现了对象的创建和使用分离，仍然存在以下两个问题：

（1）工厂类过于庞大，包含了大量的 if…else…代码，导致维护和测试难度增大。

（2）系统扩展不灵活，如果增加新类型的日志记录器，必须修改静态工厂方法的业务逻辑，违反了开闭原则。

如何解决这两个问题并提供一种简单工厂模式的改进方案呢？这就是本章所要介绍的

工厂方法模式的动机之一。

5.2　工厂方法模式概述

在简单工厂模式中只提供一个工厂类，该工厂类处于对产品类进行实例化的中心位置，它需要知道每个产品对象的创建细节，并决定何时实例化哪一个产品类。简单工厂模式最大的缺点是当有新产品要加入系统中时，必须修改工厂类，需要在其中加入必要的业务逻辑，这违背了开闭原则。此外，在简单工厂模式中，所有的产品都由同一个工厂创建，工厂类职责较重，业务逻辑较为复杂，具体产品与工厂类之间的耦合度高，严重影响了系统的灵活性和扩展性，而工厂方法模式则可以很好地解决这一问题。

在工厂方法模式中，不再提供一个统一的工厂类来创建所有的产品对象，而是针对不同的产品提供不同的工厂，系统提供一个与产品等级结构对应的工厂等级结构。工厂方法模式定义如下：

> **工厂方法模式(Factory Method Pattern)**：定义一个用于创建对象的接口，让子类决定将哪一个类实例化。工厂方法模式让一个类的实例化延迟到其子类。工厂方法模式又简称为**工厂模式(Factory Pattern)**，又可称作**虚拟构造器模式(Virtual Constructor Pattern)**或**多态工厂模式(Polymorphic Factory Pattern)**。工厂方法模式是一种类创建型模式。

工厂方法模式提供一个抽象工厂接口来声明抽象工厂方法，而由其子类来具体实现工厂方法，创建具体的产品对象。工厂方法模式结构如图 5-2 所示。

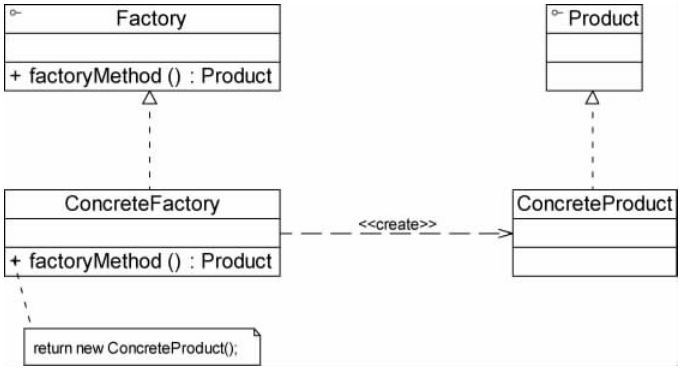

图 5-2　工厂方法模式结构图

从图 5-2 可以看出，在工厂方法模式结构图中包含以下 4 个角色。

(1) Product(抽象产品)：它是定义产品的接口，是工厂方法模式所创建对象的超类型，也就是产品对象的公共父类。

(2) ConcreteProduct(具体产品)：它实现了抽象产品接口，某种类型的具体产品由专门的具体工厂创建，具体工厂和具体产品之间一一对应。

(3) Factory(抽象工厂)：在抽象工厂类中，声明了工厂方法(Factory Method)，用于返

回一个产品。抽象工厂是工厂方法模式的核心，所有创建对象的工厂类都必须实现该接口。

(4) ConcreteFactory(具体工厂)：它是抽象工厂类的子类，实现了抽象工厂中定义的工厂方法，并可由客户端调用，返回一个具体产品类的实例。

与简单工厂模式相比，工厂方法模式最重要的区别是引入了抽象工厂角色。抽象工厂可以是接口，也可以是抽象类或者具体类，其典型代码如下：

```
interface Factory {
    public Product factoryMethod();
}
```

在抽象工厂中声明了工厂方法但并未实现工厂方法，具体产品对象的创建由其子类负责。客户端针对抽象工厂编程，可在运行时再指定具体工厂类。具体工厂类实现了工厂方法，不同的具体工厂可以创建不同的具体产品，其典型代码如下：

```
class ConcreteFactory implements Factory {
    public Product factoryMethod() {
        return new ConcreteProduct();
    }
}
```

在实际使用时，具体工厂类在实现工厂方法时除了创建具体产品对象之外，还可以负责产品对象的初始化工作以及一些资源和环境配置工作，例如连接数据库、创建文件等。

在客户端代码中，只需关心工厂类即可。不同的具体工厂可以创建不同的产品，典型的客户端类代码片段如下：

```
...
Factory factory;
factory = new ConcreteFactory();          //可通过配置文件实现
Product product;
product = factory.factoryMethod();
...
```

可以通过配置文件来存储具体工厂类 ConcreteFactory 的类名，更换新的具体工厂时无须修改源代码，系统扩展更为方便。

思考

工厂方法模式中的工厂方法能否为静态方法？为什么？

5.3 完整解决方案

Sunny 公司开发人员决定使用工厂方法模式来设计日志记录器，其基本结构如图 5-3 所示。

在图 5-3 中，Logger 接口充当抽象产品，其子类 FileLogger 和 DatabaseLogger 充当具体产

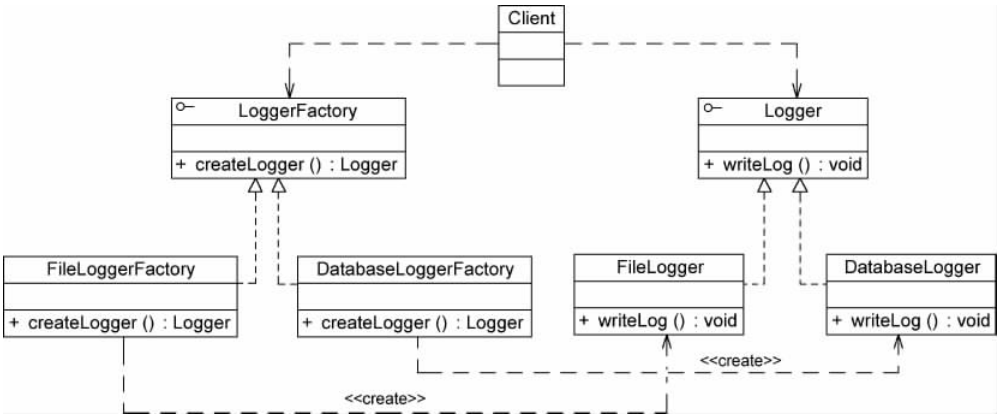

图 5-3　使用工厂方法模式设计的日志记录器结构图

品。LoggerFactory 接口充当抽象工厂，其子类 FileLoggerFactory 和 DatabaseLoggerFactory 充当具体工厂。完整代码如下：

```
//日志记录器接口：抽象产品
interface Logger {
    public void writeLog();
}

//数据库日志记录器：具体产品
class DatabaseLogger implements Logger {
    public void writeLog() {
        System.out.println("数据库日志记录。");
    }
}

//文件日志记录器：具体产品
class FileLogger implements Logger {
    public void writeLog() {
        System.out.println("文件日志记录。");
    }
}

//日志记录器工厂接口：抽象工厂
interface LoggerFactory {
    public Logger createLogger();
}

//数据库日志记录器工厂类：具体工厂
class DatabaseLoggerFactory implements LoggerFactory {
    public Logger createLogger() {
            //连接数据库，代码省略
            //创建数据库日志记录器对象
            Logger logger = new DatabaseLogger();
            //初始化数据库日志记录器，代码省略
            return logger;
```

```
    }
}

//文件日志记录器工厂类：具体工厂
class FileLoggerFactory implements LoggerFactory {
    public Logger createLogger() {
            //创建文件日志记录器对象
            Logger logger = new FileLogger();
            //创建文件,代码省略
            return logger;
    }
}
```

编写如下客户端测试代码：

```
class Client {
    public static void main(String args[]) {
        LoggerFactory factory;
        Logger logger;
        factory = new FileLoggerFactory();      //可引入配置文件实现
        logger = factory.createLogger();
        logger.writeLog();
    }
}
```

编译并运行程序，输出结果如下：

```
文件日志记录。
```

5.4 反射与配置文件

为了让系统具有更好的灵活性和可扩展性，Sunny 公司开发人员决定对日志记录器客户端代码进行重构，使得可以在不修改任何客户端代码的基础上更换或增加新的日志记录方式。

在客户端代码中将不再使用 new 关键字来创建工厂对象，而是将具体工厂类的类名存储在配置文件（例如 XML 文件）中，通过读取配置文件获取类名字符串，再使用 Java 的反射机制，根据类名字符串生成对象。在整个实现过程中需要用到两个技术：Java 反射机制与配置文件读取。软件系统的配置文件通常为 XML 文件，可以使用 DOM（Document Object Model）、SAX（Simple API for XML）、StAX（Streaming API for XML）等技术来处理 XML 文件。关于 DOM、SAX、StAX 等技术的详细学习，大家可以参考其他相关资料，在此不予扩展。

扩展

关于 Java 与 XML 的相关资料，大家可以阅读 Tom Myers 和 Alexander Nakhimovsky 所著的《Java XML 编程指南》一书。

Java 反射(Java Reflection)是指在程序运行时获取已知名称的类或已有对象的相关信息的一种机制,包括类的方法、属性、父类等信息,还包括实例的创建和实例类型的判断等。在反射中使用最多的类是 Class。Class 类的实例表示正在运行的 Java 应用程序中的类和接口,其 forName(String className)方法可以返回与带有给定字符串名的类或接口相关联的 Class 对象,再通过 Class 对象的 newInstance()方法创建此对象所表示的类的一个新实例,即通过一个类名字符串得到类的实例。例如创建一个字符串类型的对象,其代码如下:

```
//通过类名生成实例对象并将其返回
Class c = Class.forName("String");
Object obj = c.newInstance();
return obj;
```

此外,在 JDK 中还提供了 java.lang.reflect 包,封装了其他与反射相关的类,在本书中只用到上述简单的反射代码,在此不予扩展。

Sunny 公司开发人员创建了如下 XML 格式的配置文件 config.xml 用于存储具体日志记录器工厂类类名:

```
<!— config.xml —->
<?xml version = "1.0"?>
<config>
    <className>FileLoggerFactory</className>
</config>
```

为了读取该配置文件并通过存储在其中的类名字符串反射生成对象,Sunny 公司开发人员开发了一个名为 XMLUtil 的工具类,其详细代码如下:

```
//工具类 XMLUtil.java
import javax.xml.parsers.*;
import org.w3c.dom.*;
import org.xml.sax.SAXException;
import java.io.*;

public class XMLUtil {
//该方法用于从 XML 配置文件中提取具体类类名,并返回一个实例对象
    public static Object getBean() {
        try {
            //创建 DOM 文档对象
            DocumentBuilderFactory dFactory = DocumentBuilderFactory.newInstance();
            DocumentBuilder builder = dFactory.newDocumentBuilder();
            Document doc;
            doc = builder.parse(new File("config.xml"));

            //获取包含类名的文本节点
```

```
            NodeList nl = doc.getElementsByTagName("className");
            Node classNode = nl.item(0).getFirstChild();
            String cName = classNode.getNodeValue();

            //通过类名生成实例对象并将其返回
            Class c = Class.forName(cName);
            Object obj = c.newInstance();
            return obj;
        }
        catch(Exception e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

注：在后续的设计模式学习过程中将多次重用该类。

有了 XMLUtil 类后，可以对日志记录器的客户端代码进行修改，不再直接使用 new 关键字来创建具体的工厂类，而是将具体工厂类的类名存储在 XML 文件中，再通过 XMLUtil 类的静态工厂方法 getBean()方法进行对象的实例化。客户端代码修改如下：

```
class Client {
    public static void main(String args[]) {
        LoggerFactory factory;
        Logger logger;
        factory = (LoggerFactory)XMLUtil.getBean();      //getBean()的返回类型为 Object,
                                                         //需要进行强制类型转换
        logger = factory.createLogger();
        logger.writeLog();
    }
}
```

引入 XMLUtil 类和 XML 配置文件后，如果要增加新的日志记录方式，只需要执行如下几个步骤：

(1) 新的日志记录器需要继承抽象日志记录器 Logger。

(2) 对应增加一个新的具体日志记录器工厂，继承抽象日志记录器工厂 LoggerFactory，并实现其中的工厂方法 createLogger()，设置好初始化参数和环境变量，返回具体日志记录器对象。

(3) 修改配置文件 config.xml，用新增的具体日志记录器工厂类的类名字符串替换原有工厂类类名字符串。

(4) 编译新增的具体日志记录器类和具体日志记录器工厂类，运行客户端测试类即可使用新的日志记录方式，而原有类库代码无须做任何修改，完全符合开闭原则。

通过上述重构可以使得系统更加灵活，由于很多设计模式都关注系统的可扩展性和灵活性，因此都定义了抽象层，在抽象层中声明业务方法，而将业务方法的实现放在实现层中。为了更好地体现这些设计模式的特点，本书在讲解很多设计模式时都使用 XML 配置文件

和 Java 反射机制来创建对象。

思考

有人说：可以在客户端代码中直接通过反射机制来生成产品对象。在定义产品对象时使用抽象类型，同样可以确保系统的灵活性和可扩展性。增加新的具体产品类无须修改源代码，只需要将其作为抽象产品类的子类再修改配置文件即可，根本不需要抽象工厂类和具体工厂类。

试思考：这种做法是否可行？如果可行，这种做法是否存在问题？为什么？

5.5　重载的工厂方法

Sunny 公司开发人员通过进一步分析，发现可以通过多种方式来初始化日志记录器。例如可以为各种日志记录器提供默认实现；还可以为数据库日志记录器提供数据库连接字符串，为文件日志记录器提供文件路径；也可以将参数封装在一个 Object 类型的对象中，通过 Object 对象将配置参数传入工厂类。此时，可以提供一组重载的工厂方法，以不同的方式对产品对象进行创建，如图 5-4 所示。当然，对于同一个具体工厂而言，无论使用哪个工厂方法，创建的产品类型均要相同。

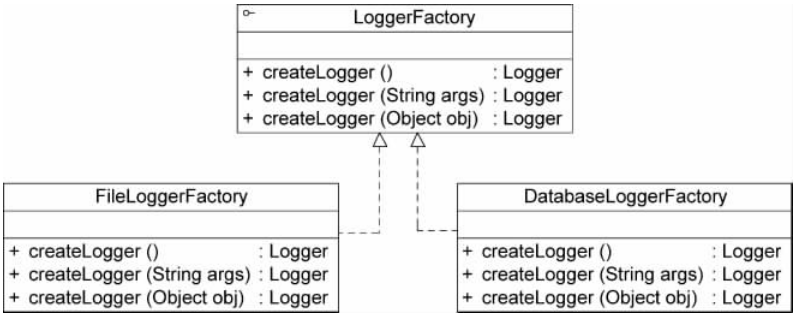

图 5-4　重载的工厂方法结构图

引入重载方法后，抽象工厂 LoggerFactory 的代码修改如下：

```
interface LoggerFactory {
    public Logger createLogger();
    public Logger createLogger(String args);
    public Logger createLogger(Object obj);
}
```

具体工厂类 DatabaseLoggerFactory 代码修改如下：

```
class DatabaseLoggerFactory implements LoggerFactory {
    public Logger createLogger() {
            //使用默认方式连接数据库，代码省略
            Logger logger = new DatabaseLogger();
```

```
            //初始化数据库日志记录器,代码省略
            return logger;
        }

        public Logger createLogger(String args) {
            //使用参数 args 作为连接字符串来连接数据库,代码省略
            Logger logger = new DatabaseLogger();
            //初始化数据库日志记录器,代码省略
            return logger;
        }

        public Logger createLogger(Object obj) {
            //使用封装在参数 obj 中的连接字符串来连接数据库,代码省略
            Logger logger = new DatabaseLogger();
            //使用封装在参数 obj 中的数据来初始化数据库日志记录器,代码省略
            return logger;
        }
    }

    //其他具体工厂类代码省略
```

在抽象工厂中定义多个重载的工厂方法,在具体工厂中实现了这些工厂方法,这些方法可以包含不同的业务逻辑,以满足对不同产品对象的需求。

5.6 工厂方法的隐藏

有时候,为了进一步简化客户端的使用,还可以对客户端隐藏工厂方法。此时,在工厂类中将直接调用产品类的业务方法,客户端无须调用工厂方法创建产品,直接通过工厂即可使用所创建的对象中的业务方法。

如果对客户端隐藏工厂方法,日志记录器的结构图将修改为如图 5-5 所示。

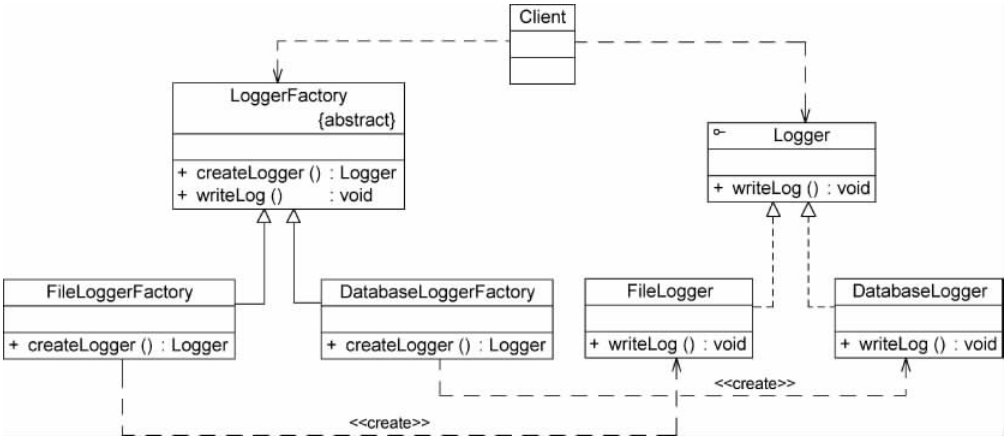

图 5-5 隐藏工厂方法后的日志记录器结构图

在图 5-5 中，抽象工厂类 LoggerFactory 的代码修改如下：

```
//改为抽象类
abstract class LoggerFactory {
    //在工厂类中直接调用日志记录器类的业务方法 writeLog()
    public void writeLog() {
        Logger logger = this.createLogger();
        logger.writeLog();
    }

    public abstract Logger createLogger();
}
```

客户端代码修改如下：

```
class Client {
    public static void main(String args[]) {
        LoggerFactory factory;
        factory = (LoggerFactory)XMLUtil.getBean();
        factory.writeLog();     //直接使用工厂对象来调用产品对象的业务方法
    }
}
```

通过将业务方法的调用移入工厂类，可以直接使用工厂对象来调用产品对象的业务方法，客户端无须直接使用工厂方法，在某些情况下大家也可以使用这种设计方案。

5.7　工厂方法模式总结

工厂方法模式是简单工厂模式的延伸，它继承了简单工厂模式的优点，同时还弥补了简单工厂模式的不足。工厂方法模式是使用频率最高的设计模式之一，是很多开源框架和 API 类库的核心模式。

1. 主要优点

工厂方法模式的主要优点如下：

（1）在工厂方法模式中，工厂方法用来创建客户所需要的产品，同时还向客户隐藏了哪种具体产品类将被实例化这一细节。用户只需要关心所需产品对应的工厂，无须关心创建细节，甚至无须知道具体产品类的类名。

（2）基于工厂角色和产品角色的多态性设计是工厂方法模式的关键。它能够让工厂可以自主确定创建何种产品对象，而如何创建这个对象的细节则完全封装在具体工厂内部。工厂方法模式之所以又被称为多态工厂模式，正是因为所有的具体工厂类都具有同一抽象父类。

（3）使用工厂方法模式的另一个优点是在系统中加入新产品时，无须修改抽象工厂和抽象产品提供的接口，无须修改客户端，也无须修改其他的具体工厂和具体产品，而只要添加一个具体工厂和具体产品就可以了。这样，系统的可扩展性也就变得非常好，完全符合开

闭原则。

2. 主要缺点

工厂方法模式的主要缺点如下：

(1) 在添加新产品时，需要编写新的具体产品类，而且还要提供与之对应的具体工厂类，系统中类的个数将成对增加，在一定程度上增加了系统的复杂度，有更多的类需要编译和运行，会给系统带来一些额外的开销。

(2) 由于考虑到系统的可扩展性，需要引入抽象层，在客户端代码中均使用抽象层进行定义，增加了系统的抽象性和理解难度，且在实现时可能需要用到 DOM、反射等技术，增加了系统的实现难度。

3. 适用场景

在以下情况下可以考虑使用工厂方法模式：

(1) 客户端不知道其所需要的对象的类。在工厂方法模式中，客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可，具体的产品对象由具体工厂类创建，可将具体工厂类的类名存储在配置文件或数据库中。

(2) 抽象工厂类通过其子类来指定创建哪个对象。在工厂方法模式中，抽象工厂类只需要提供一个创建产品的接口，而由其子类来确定具体要创建的对象，利用面向对象的多态性和里氏代换原则，在程序运行时，子类对象将覆盖父类对象，从而使得系统更容易扩展。

练习

使用工厂方法模式设计一个程序来读取各种不同类型的图片格式，针对每种图片格式都设计一个图片读取器。例如，GIF 图片读取器用于读取 GIF 格式的图片，JPG 图片读取器用于读取 JPG 格式的图片。需充分考虑系统的灵活性和可扩展性。