# **工厂方法模式（Java）**

·简单工厂模式的不足

在简单工厂模式中，只提供了一个工厂类，该工厂类处于对产品类进行实例化的中心位置，它知道每一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到系统中时，必须修改工厂类，加入必要的处理逻辑，这违背了“开闭原则”。在简单工厂模式中，所有的产品都是由同一个工厂创建，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。

工厂方法模式

##### **1、模式动机**

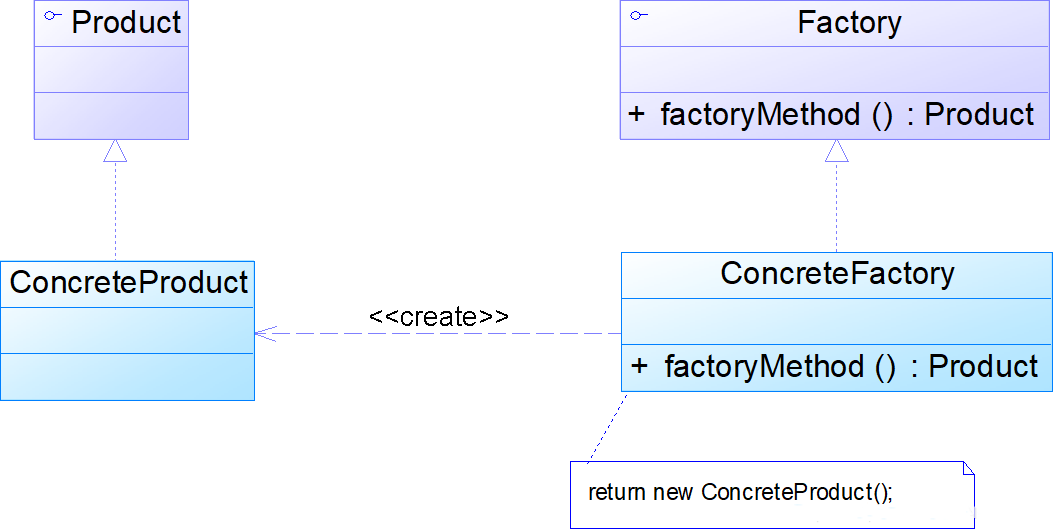
·考虑这样一个系统，按钮工厂类可以返回一个具体的按钮实例，如圆形按钮、矩形按钮、菱形按钮等。在这个系统中，如果需要增加一种新类型的按钮，如椭圆形按钮，那么除了增加一个新的具体产品类之外，还需要修改工厂类的代码，这就使得整个设计在一定程度上违反了“开闭原则”。

·现在对该系统进行修改，不再设计一个按钮工厂类来统一负责所有产品的创建，而是将具体按钮的创建过程交给专门的工厂子类去完成，我们先定义一个抽象的按钮工厂类，再定义具体的工厂类来生成圆形按钮、矩形按钮、菱形按钮等，它们实现在抽象按钮工厂类中定义的方法。这种抽象化的结果使这种结构可以在不修改具体工厂类的情况下引进新的产品，如果出现新的按钮类型，只需要为这种新类型的按钮创建一个具体的工厂类就可以获得该新按钮的实例，这一特点无疑使得工厂方法模式具有超越简单工厂模式的优越性，更加符合“开闭原则”。

##### **2、模式定义**

工厂方法模式(Factory Method Pattern)又称为工厂模式，也叫虚拟构造器(Virtual Constructor)模式或者多态工厂(Polymorphic Factory)模式，它属于类创建型模式。在工厂方法模式中，工厂父类负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象，这样做的目的是将产品类的实例化操作延迟到工厂子类中完成，即通过工厂子类来确定究竟应该实例化哪一个具体产品类。

##### **3、模式结构**



##### **模式角色**

1 Product：抽象产品

2 ConcreteProduct：具体产品

3 Factory：抽象工厂

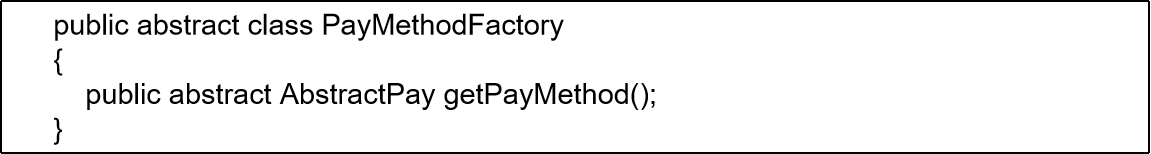
4 ConcreteFactory：具体工厂

##### **5、模式分析**

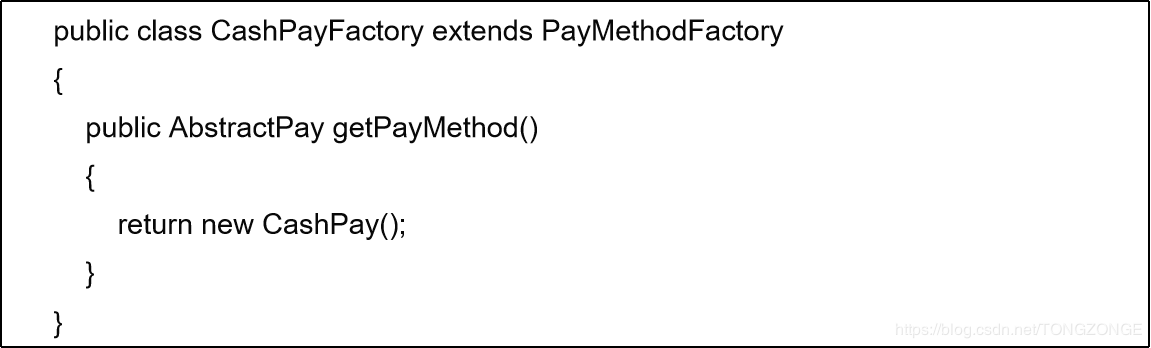
·工厂方法模式是简单工厂模式的进一步抽象和推广。由于使用了面向对象的多态性，工厂方法模式保持了简单工厂模式的优点，而且克服了它的缺点。在工厂方法模式中，核心的工厂类不再负责所有产品的创建，而是将具体创建工作交给子类去做。这个核心类仅仅负责给出具体工厂必须实现的接口，而不负责哪一个产品类被实例化这种细节，这使得工厂方法模式可以允许系统在不修改工厂角色的情况下引进新产品。

·当系统扩展需要添加新的产品对象时，仅仅需要添加一个具体产品对象以及一个具体工厂对象，原有工厂对象不需要进行任何修改，也不需要修改客户端，很好地符合了“开闭原则”。而简单工厂模式在添加新产品对象后不得不修改工厂方法，扩展性不好。工厂方法模式退化后可以演变成简单工厂模式。

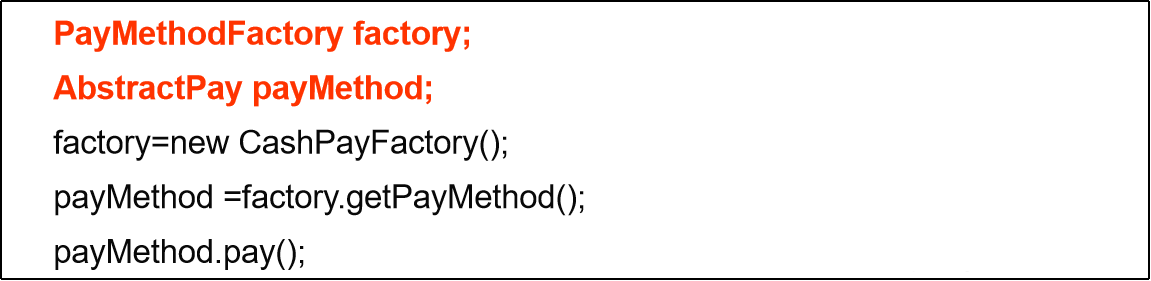
·抽象工厂类代码



·具体工厂类代码



·客户类代码



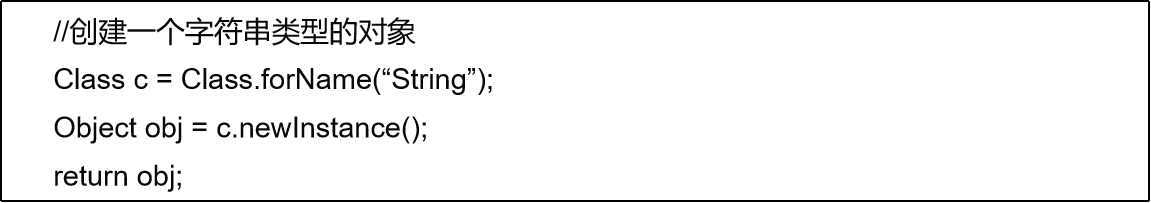
为了提高系统的可扩展性和灵活性，在定义工厂和产品时都必须使用抽象层，如果需要更换产品类，只需要更换对应的工厂即可，其他代码不需要进行任何修改。

###### **配置文件代码**

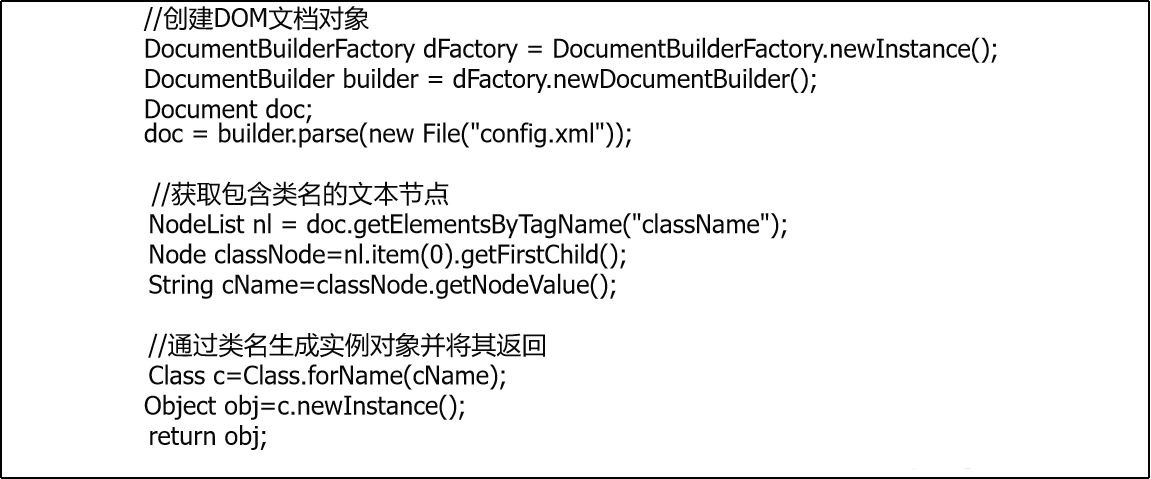
###### **·**在实际的应用开发中，一般将具体工厂类的实例化过程进行改进，不直接使用new关键字来创建对象，而是将具体类的类名写入配置文件中，再通过Java的反射机制，读取XML格式的配置文件，根据存储在XML文件中的类名字符串生成对象。微信图片_20200610115146

·Java反射

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、超类等信息，还包括实例的创建和实例类型的判断等。可通过Class类的forName()方法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。



·工具类XMLUtil代码片段



·修改后返回客户端的代码



6、模式实例与分析

（1）电视机工厂

将原有的工厂进行分割，为每种品牌的电视机提供一个子工厂，海尔工厂专门负责生产海尔电视机，海信工厂专门负责生产海信电视机，如果需要生产TCL电视机或创维电视机，只需要对应增加一个新的TCL工厂或创维工厂即可，原有的工厂无须做任何修改，使得整个系统具有更加的灵活性和可扩展性。

·代码实现

package 工厂方法模式3;

//抽象产品

public interface TV {

public void play();

}

package 工厂方法模式3;

//具体产品

public class HaierTV implements TV {

@Override

public void play() {

System.out.println("海尔电视机播放中。。。。");

}

}

package 工厂方法模式3;

//具体产品

public class HisenseTV implements TV {

@Override

public void play() {

System.out.println("海信电视机播放中。。。。");

}

}

package 工厂方法模式3;

//抽象工厂

public interface TVfactory {

public TV produceTV();

}

package 工厂方法模式3;

//具体工厂

public class HaierTVFactory implements TVfactory {

@Override

public TV produceTV() {

System.out.println("海尔电视机工厂生产海尔电视机");

return new HaierTV();

}

}

package 工厂方法模式3;

//具体工厂

public class HisenseTVFactory implements TVfactory {

@Override

public TV produceTV() {

System.out.println("海信电视机工厂生产海信电视机");

return new HisenseTV();

}

}

//客户端代码

package 工厂方法模式3;

import 工厂方法模式2.XMLUtil;

public class 电视机工厂 {

public static class client{

public static void main(String[] args) {

TV tv;

TVfactory factory;

try {

factory = (TVfactory) XMLUtil.getBean();

tv = factory.produceTV();

tv.play();

} catch (Exception e) {

System.out.println(e.getMessage());

}

}

}

}

* Config.xml文件配置：  
  通过XMLUtil.getBean()方法直接创建并实例化相应所需对象。减少代码冗余。

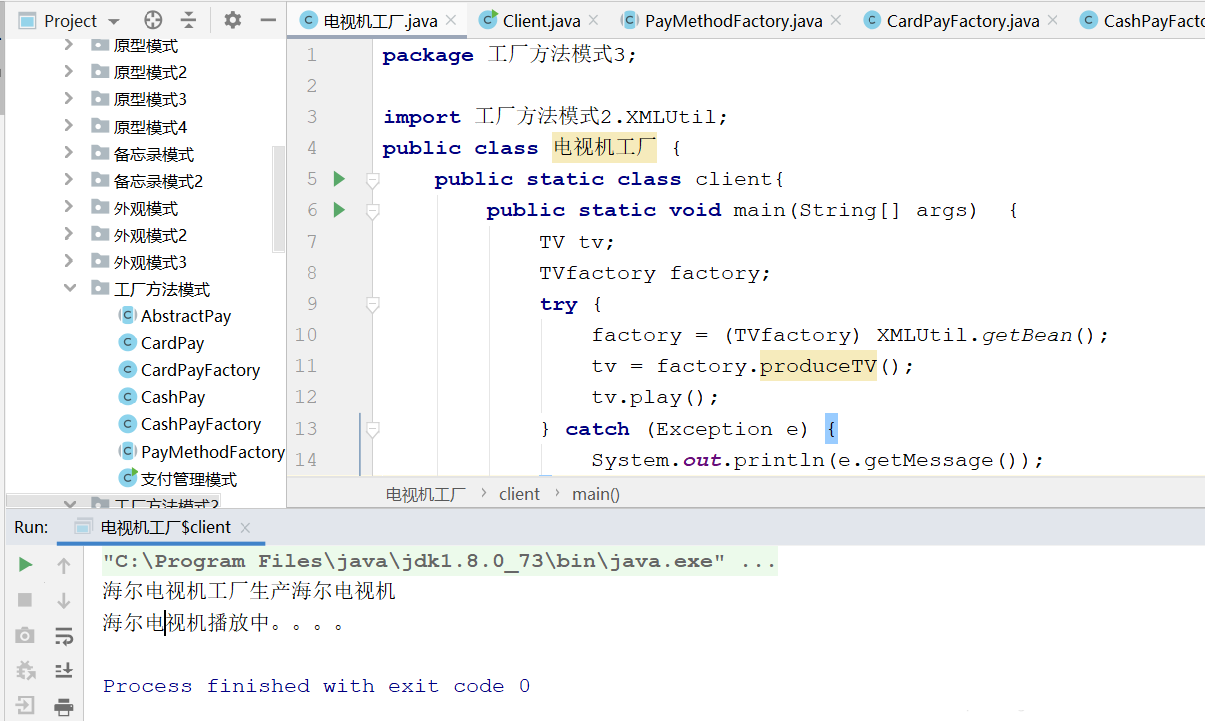
<?xml version="1.0"?>

<config>

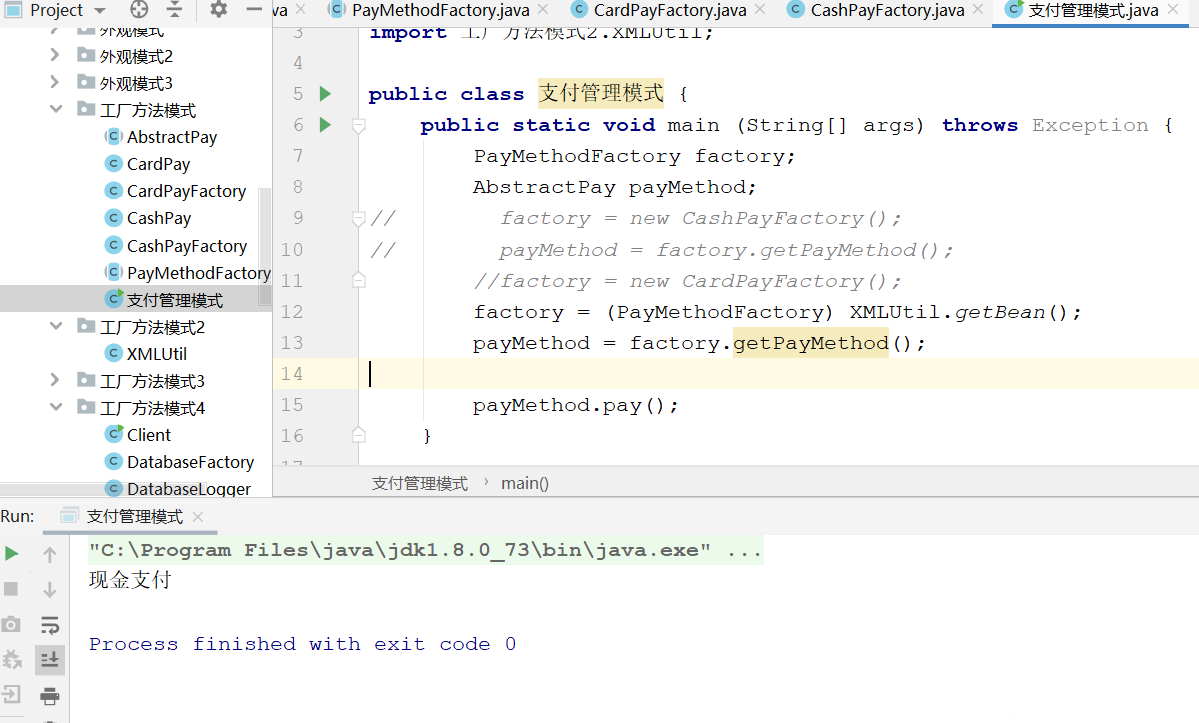
<className>工厂方法模式3.HaierTVFactory</className>

</config>

* 运行结果展示：



###### **（2）支付管理模式**

* ·代码实现
* package 工厂方法模式;
* *//抽象产品* public abstract class AbstractPay { public abstract void pay()；}
* package 工厂方法模式;
* //具体产品
* public class CardPay extends AbstractPay {
* @Override
* public void pay() {
* System.out.println("银行卡支付");
* }
* }
* package 工厂方法模式;
* //具体产品
* public class CashPay extends AbstractPay {
* @Override
* public void pay() {
* System.out.println("现金支付");
* }
* }
* package 工厂方法模式;
* //抽象工厂
* public abstract class PayMethodFactory {
* public abstract AbstractPay getPayMethod();
* }
* package 工厂方法模式;
* //具体工厂
* public class CardPayFactory extends PayMethodFactory {
* @Override
* public AbstractPay getPayMethod() {
* return new CardPay();
* }
* }
* package 工厂方法模式;
* //具体工厂
* public class CashPayFactory extends PayMethodFactory {
* @Override
* public AbstractPay getPayMethod() {
* return new CashPay();
* }
* }
* package 工厂方法模式;
* import 工厂方法模式2.XMLUtil;
* //客户端代码
* public class 支付管理模式 {
* public static void main (String[] args) throws Exception {
* PayMethodFactory factory;
* AbstractPay payMethod;
* // factory = new CashPayFactory();
* // payMethod = factory.getPayMethod();
* //factory = new CardPayFactory();
* factory = (PayMethodFactory) XMLUtil.getBean();
* payMethod = factory.getPayMethod();
* payMethod.pay();
* }
* }
* config.xml配置
* <?xml version="1.0"?>
* <config>
* <className>工厂方法模式.CashPayFactory</className>
* </config>
* · 结果展示：
* 

###### **（3）日志记录器**

* 代码实现
* package 工厂方法模式4;
* //抽象产品
* public interface Logger {
* public void writeLog();
* }
* package 工厂方法模式4;
* //具体产品
* public class DatabaseLogger implements Logger {
* @Override
* public void writeLog() {
* System.out.println("数据库日志记录");
* }
* }
* package 工厂方法模式4;
* //具体产品
* public class FileLogger implements Logger {
* @Override
* public void writeLog() {
* System.out.println("文件日志记录");
* }
* }

package 工厂方法模式4;

// 抽象工厂

public interface LogFactory {

//抽象工厂方法

public Logger createLogger();

}

package 工厂方法模式4;

// 具体工厂

public class DatabaseFactory implements LogFactory {

@Override

public Logger createLogger() {

//连接数据库,代码省略

System.out.println("数据库日志工厂记录数据库日志");

//创建数据库日志记录器对象

//初始化数据库日志记录器,代码省略

return new DatabaseLogger();

}

}

package 工厂方法模式4;

//具体工厂

public class FileLoggerFactory implements LogFactory {

@Override

public Logger createLogger() {

System.out.println("文件日志工厂记录文件日志");

return new FileLogger();

}

}

package 工厂方法模式4;

// 客户端代码

//public class 日志记录器 {

public class Client{

public static void main(String[] args) {

LogFactory factory;

Logger logger;

factory = new FileLoggerFactory();

logger = factory.createLogger();

// factory = (LogFactory) XMLUtil.getBean();

// logger =factory.createLogger();

logger.writeLog();

}

}

* Config.xml文件配置：  
  通过XMLUtil.getBean()方法直接创建并实例化相应所需对象。减少代码冗余。

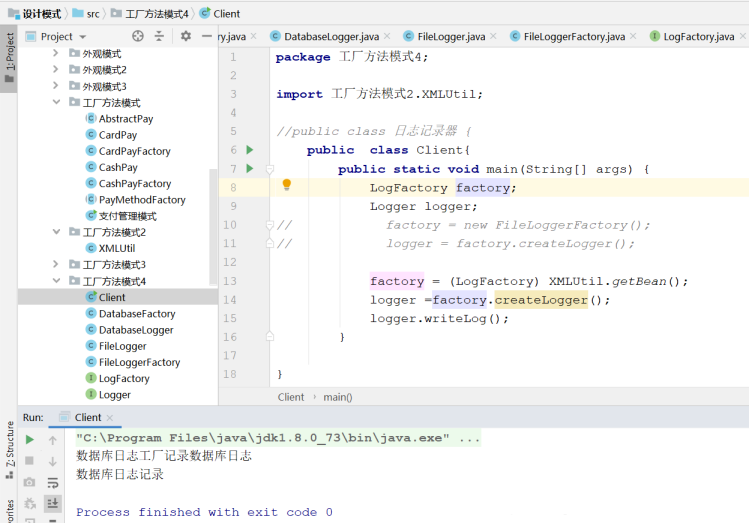
<?xml version="1.0"?>

<config>

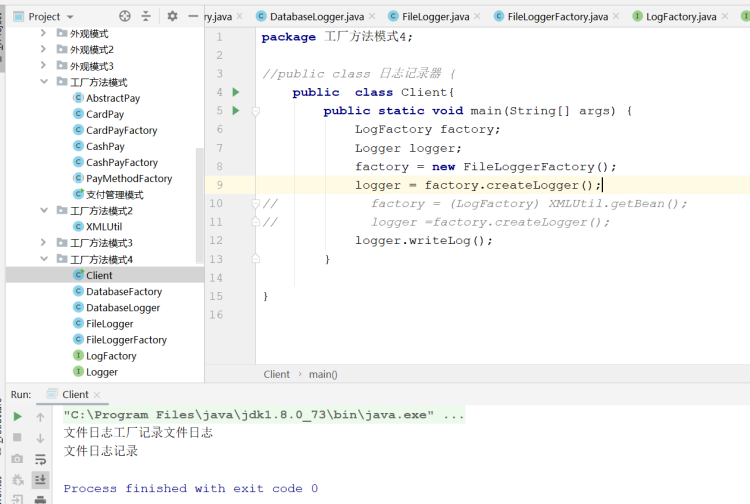
<className>工厂方法模式4.DatabaseFactory</className>

</config>

运行结果截图：



同样，也可以直接通过new FileLoggerFactory()，再调用其createLogger()方法创建对象并实例化。也可以配置在config.xml文件中，结果相同，展示结果如下所示：



7、模式的优缺点

优点

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。它能够使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，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”

缺点

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。

8、模式使用环境

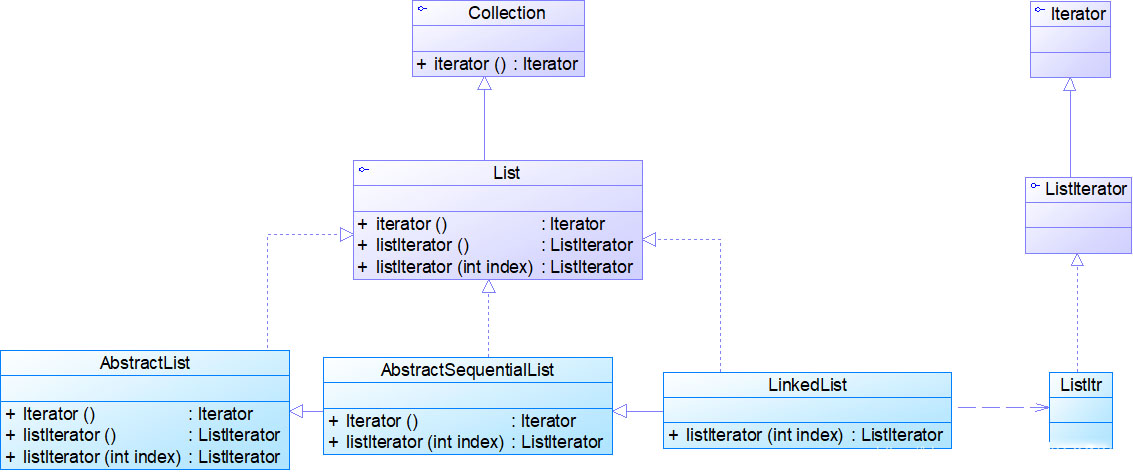
1、一个类不知道它所需要的对象的类：在工厂方法模式中，客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可，具体的产品对象由具体工厂类创建；客户端需要知道创建具体产品的工厂类。

2、一个类通过其子类来指定创建哪个对象：在工厂方法模式中，对于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。

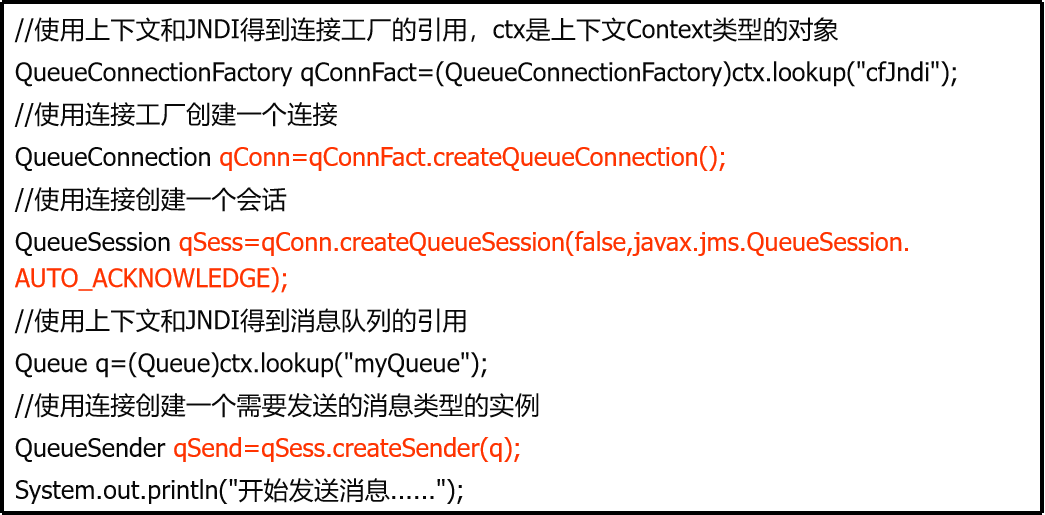
3、将创建对象的任务委托给多个工厂子类中的某一个，客户端在使用时可以无须关心是哪一个工厂子类创建产品子类，需要时再动态指定，可将具体工厂类的类名存储在配置文件或数据库中。

9、模式应用

java.util.Collection接口的iterator()方法：



·Java消息服务JMS(Java Messaging Service)



JDBC中的工厂方法：

