第3章

简单工厂模式

本章导学

创建型模式关注对象的创建过程，是一类最常见的设计模式，在软件开发中的应用非常广泛。创建型模式描述如何将对象的创建和使用分离，让用户在使用对象时无须关心对象的创建细节，从而降低系统的耦合度，让设计方案更易于修改和扩展。

简单工厂模式是最简单的设计模式之一，它虽然不属于 GoF 的 23 种设计模式，但是应用也较为频繁，同时它是学习其他创建型模式的基础。在简单工厂模式中只需要记住一个简单的参数即可获得所需的对象实例，它提供专门的核心工厂类来负责对象的创建，实现对象创建和使用的分离。

本章将对 6 种创建型模式进行简要的介绍，并通过实例来学习简单工厂模式，理解简单工厂模式的结构及特点，学会如何在实际软件项目开发中合理地使用简单工厂模式。

本章知识点

- 创建型模式
- 简单工厂模式的定义
- 简单工厂模式的结构
- 简单工厂模式的实现
- 简单工厂模式的应用
- 简单工厂模式的优/缺点
- 简单工厂模式的适用环境

3.1 创建型模式

软件系统在运行时类将实例化成对象，并由这些对象来协作完成各项业务功能。创建型模式(Creational Pattern)关注对象的创建过程，是一类最常用的设计模式，在软件开发中

的应用非常广泛。创建型模式对类的实例化过程进行了抽象，能够将软件模块中对象的创建和对象的使用分离，对用户隐藏了类的实例的创建细节。

创建型模式描述如何将对象的创建和使用分离，让用户在使用对象时无须关心对象的创建细节，从而降低系统的耦合度，让设计方案更易于修改和扩展。每一个创建型模式都通过采用不同的解决方案来回答 3 个问题，即创建什么(What)、由谁创建(Who)和何时创建(When)。

在 GoF 设计模式中包含 5 种创建型模式，通常将一种非 GoF 设计模式——简单工厂模式作为学习其他工厂模式的基础，这 6 种设计模式的名称、定义、学习难度和使用频率如表 3-1 所示。

表 3-1　创建型模式一览表

模 式 名 称	定　　义	学习难度	使用频率
简单工厂模式 (Simple Factory Pattern)	定义一个工厂类，它可以根据参数的不同返回不同类的实例，被创建的实例通常都具有共同的父类	★★☆☆☆	★★★☆☆
工厂方法模式 (Factory Method Pattern)	定义一个用于创建对象的接口，但是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	★★☆☆☆	★★★★★
抽象工厂模式 (Abstract Factory Pattern)	提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类	★★★★☆	★★★★★
建造者模式 (Builder Pattern)	将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示	★★★★☆	★★☆☆☆
原型模式 (Prototype Pattern)	使用原型实例指定待创建对象的类型，并且通过复制这个原型来创建新的对象	★★★☆☆	★★★☆☆
单例模式 (Singleton Pattern)	确保一个类只有一个实例，并提供一个全局访问点来访问这个唯一实例	★☆☆☆☆	★★★★☆

3.2　简单工厂模式概述

简单工厂模式并不属于 GoF 的 23 种经典设计模式，但通常将它作为学习其他工厂模式的基础，下面通过一个简单实例来引出简单工厂模式。

考虑一个水果农场，当用户需要某一种水果时该农场能够根据用户所提供的水果名称返回该水果。在此，水果农场被称为工厂(Factory)，而生成的水果被称为产品(Product)，水果的名称则被称为参数，工厂可以根据参数的不同返回不同的产品，这就是简单工厂模式的动机。该过程的示意图如图 3-1 所示，用户无须知道苹果(Apple)、橙(Orange)、香蕉(Banana)如何创建，只需要知道水果的名字即可得到对应的水果。

作为最简单的设计模式之一，简单工厂模式的设计思想和实现过程都比较简单，其基本实现流程如下：

首先将需要创建的各种不同产品对象的相关代码封装到不同的类中，这些类称为具体

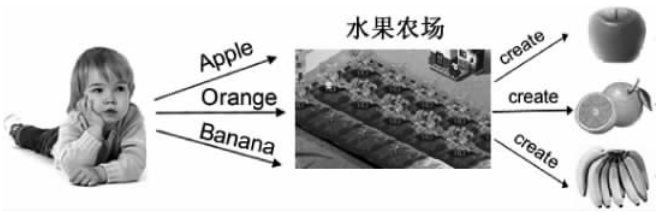

图 3-1 简单工厂模式示意图

产品类，而将它们公共的代码进行抽象和提取后封装在一个抽象产品类中，每一个具体产品类都是抽象产品类的子类；然后提供一个工厂类用于创建各种产品，在工厂类中提供一个创建产品的工厂方法，该方法可以根据所传入的参数不同创建不同的具体产品对象；客户端只需调用工厂类的工厂方法并传入相应的参数即可得到一个产品对象。

简单工厂模式的定义如下：

> **简单工厂模式(Simple Factory Pattern)**：定义一个工厂类，它可以根据参数的不同返回不同类的实例，被创建的实例通常都具有共同的父类。

由于在简单工厂模式中用于创建实例的方法通常是静态(static)方法，因此简单工厂模式又被称为静态工厂方法(Static Factory Method)模式，它是一种类创建型模式。简单工厂模式的要点在于当用户需要什么时，只需要传入一个正确的参数就可以获取所需要的对象，而无须知道其创建细节。

3.3 简单工厂模式结构与实现

3.3.1 简单工厂模式结构

简单工厂模式的结构比较简单，其核心是工厂类的设计，其结构如图 3-2 所示。

由图 3-2 可知，简单工厂模式包含以下 3 个角色。

(1) Factory(工厂角色)：工厂角色即工厂类，它是简单工厂模式的核心，负责实现创建所有产品实例的内部逻辑；工厂类可以被外界直接调用，创建所需的产品对象；在工厂类中提供了静态的工厂方法 factoryMethod()，它的返回类型为抽象产品类型 Product。

(2) Product(抽象产品角色)：它是工厂类创建的所有对象的父类，封装了各种产品对象的公有方法，它的引入将提高系统的灵活性，使得在工厂类中只需定义一个通用的工厂方法，因为所有创建的具体产品对象都是其子类对象。

(3) ConcreteProduct(具体产品角色)：它是简单工厂模式的创建目标，所有被创建的对象都充当这个角色的某个具体类的实例。每一个具体产品角色都继承了抽象产品角色，需要实现在抽象产品中声明的抽象方法。

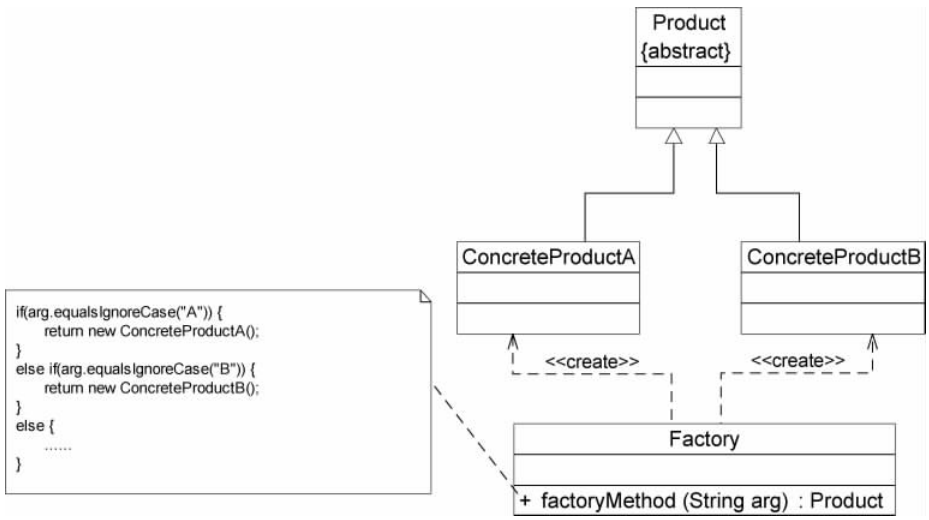

图 3-2 简单工厂模式结构图

3.3.2 简单工厂模式实现

在简单工厂模式中客户端通过工厂类来创建一个产品类的实例，而无须直接使用 new 关键字来创建对象，它是工厂模式家族中最简单的一员。

在使用简单工厂模式时首先需要对产品类进行重构，不能设计一个包罗万象的产品类，而需根据实际情况设计一个产品层次结构，将所有产品类公共的代码移至抽象产品类，并在抽象产品类中声明一些抽象方法，以供不同的具体产品类来实现。典型的抽象产品类代码如下：

```
public abstract class Product {
    //所有产品类的公共业务方法
    public void methodSame() {
        //公共方法的实现
    }

    //声明抽象业务方法
    public abstract void methodDiff();
}
```

在具体产品类中实现了抽象产品类中声明的抽象业务方法，不同的具体产品类可以提供不同的实现。典型的具体产品类的代码如下：

```
public class ConcreteProduct extends Product{
    //实现业务方法
    public void methodDiff() {
        //业务方法的实现
    }
}
```

简单工厂模式的核心是工厂类，在没有工厂类之前客户端一般会使用 new 关键字来直接创建产品对象，而在引入工厂类之后客户端可以通过工厂类来创建产品，在简单工厂模式中工厂类提供了一个静态工厂方法供客户端使用，根据所传入的参数不同可以创建不同的产品对象。典型的工厂类的代码如下：

```
public class Factory {
    //静态工厂方法
    public static Product getProduct(String arg) {
        Product product = null;
        if (arg.equalsIgnoreCase("A")) {
            product = new ConcreteProductA();
            //初始化设置 product
        }
        else if (arg.equalsIgnoreCase("B")) {
            product = new ConcreteProductB();
            //初始化设置 product
        }
        return product;
    }
}
```

在客户端代码中，通过调用工厂类的工厂方法即可得到产品对象。其典型代码如下：

```
public class Client {
    public static void main(String args[]) {
        Product product;
        product = Factory.getProduct("A"); //通过工厂类创建产品对象
        product.methodSame();
        product.methodDiff();
    }
}
```

3.4 简单工厂模式应用实例

下面通过一个应用实例来进一步学习和理解简单工厂模式。

1. 实例说明

> 某软件公司要基于 Java 语言开发一套图表库，该图表库可以为应用系统提供多种不同外观的图表，例如柱状图(HistogramChart)、饼状图(PieChart)、折线图(LineChart)等。该软件公司图表库设计人员希望为应用系统开发人员提供一套灵活易用的图表库，通过设置不同的参数即可得到不同类型的图表，而且可以较为方便地对图表库进行扩展，以便能够在将来增加一些新类型的图表。
>
> 现使用简单工厂模式来设计该图表库。

2. 实例类图

通过分析，本实例的结构图如图 3-3 所示。

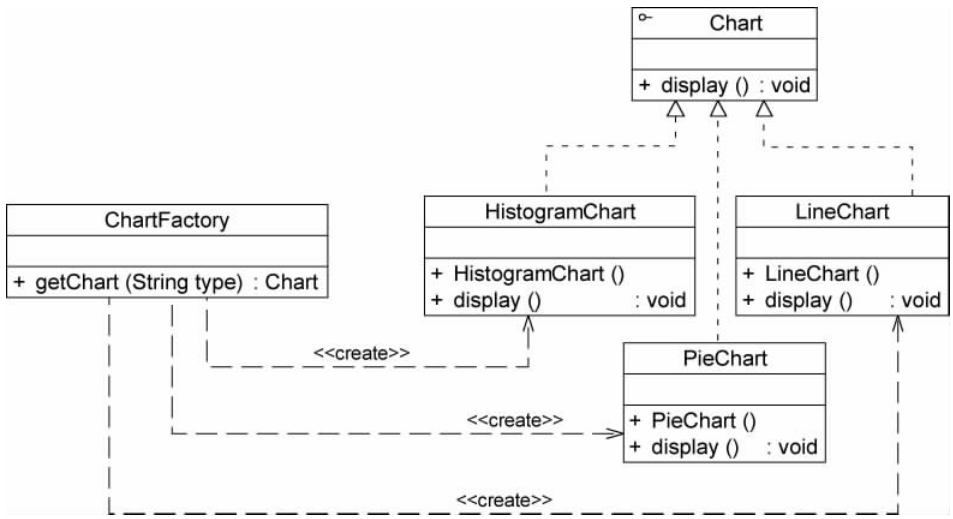

图 3-3 图表库结构图

在图 3-3 中，Chart 接口充当抽象产品类，其子类 HistogramChart、PieChart 和 LineChart 充当具体产品类，ChartFactory 充当工厂类。

3. 实例代码

（1）Chart：抽象图表接口，充当抽象产品类。

```
//designpatterns.simplefactory.Chart.java
package designpatterns.simplefactory;

public interface Chart {
    public void display();
}
```

（2）HistogramChart：柱状图类，充当具体产品类。

```
//designpatterns.simplefactory.HistogramChart.java
package designpatterns.simplefactory;

public class HistogramChart implements Chart {
    public HistogramChart() {
        System.out.println("创建柱状图!");
    }

    public void display() {
        System.out.println("显示柱状图!");
    }
}
```

（3）PieChart：饼状图类，充当具体产品类。

```
//designpatterns.simplefactory.PieChart.java
package designpatterns.simplefactory;

public class PieChart implements Chart {
    public PieChart() {
        System.out.println("创建饼状图!");
    }

    public void display() {
        System.out.println("显示饼状图!");
    }
}
```

（4）LineChart：折线图类，充当具体产品类。

```
//designpatterns.simplefactory.LineChart.java
package designpatterns.simplefactory;

public class LineChart implements Chart {
    public LineChart() {
        System.out.println("创建折线图!");
    }

    public void display() {
        System.out.println("显示折线图!");
    }
}
```

（5）ChartFactory：图表工厂类，充当工厂类。

```
//designpatterns.simplefactory.ChartFactory.java
package designpatterns.simplefactory;

public class ChartFactory {
    //静态工厂方法
    public static Chart getChart(String type) {
        Chart chart = null;
        if (type.equalsIgnoreCase("histogram")) {
            chart = new HistogramChart();
            System.out.println("初始化设置柱状图!");
        }
        else if (type.equalsIgnoreCase("pie")) {
            chart = new PieChart();
            System.out.println("初始化设置饼状图!");
        }
        else if (type.equalsIgnoreCase("line")) {
```

```
            chart = new LineChart();
            System.out.println("初始化设置折线图!");
        }
        return chart;
    }
}
```

(6) Client：客户端测试类。

```
//designpatterns.simplefactory.Client.java
package designpatterns.simplefactory;

public class Client {
    public static void main(String args[]) {
        Chart chart;
        chart = ChartFactory.getChart("histogram"); //通过静态工厂方法创建产品
        chart.display();
    }
}
```

4. 结果及分析

编译并运行程序，输出结果如下：

```
创建柱状图!
初始化设置柱状图!
显示柱状图!
```

在客户端测试类中使用工厂类 ChartFactory 的静态工厂方法创建产品对象，如果需要更换产品，只需修改静态工厂方法中的参数即可。例如将柱状图改为饼状图，只需将代码

```
chart = ChartFactory.getChart("histogram");
```

改为：

```
chart = ChartFactory.getChart("pie");
```

编译并运行程序，输出结果如下：

```
创建饼状图!
初始化设置饼状图!
显示饼状图!
```

不难发现，本实例在创建具体 Chart 对象时必须通过修改客户端代码中静态工厂方法的参数来更换具体产品对象，客户端代码需要重新编译，这对于客户端而言违反了开闭

原则。

下面介绍一种常用的解决方案，可以实现在不修改客户端代码的前提下让客户端能够更换具体产品对象。

首先可以将静态工厂方法的参数存储在 XML 格式的配置文件中，例如：

```
<?xml version = "1.0"?>
<config>
    <chartType>histogram</chartType>
</config>
```

再通过一个工具类 XMLUtil 来读取配置文件中的字符串参数。XMLUtil 类的代码如下：

```
package designpatterns.simplefactory;
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 配置文件中提取图表类型，并返回类型名
    public static String getChartType() {
        try {
            //创建文档对象
            DocumentBuilderFactory dFactory = DocumentBuilderFactory.newInstance();
            DocumentBuilder builder = dFactory.newDocumentBuilder();
            Document doc;
            doc = builder.parse(new File("src//designpatterns//simplefactory//config.xml"));

            //获取包含图表类型的文本结点
            NodeList nl = doc.getElementsByTagName("chartType");
            Node classNode = nl.item(0).getFirstChild();
            String chartType = classNode.getNodeValue().trim();
            return chartType;
        }
        catch(Exception e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

在引入了配置文件和工具类 XMLUtil 之后，客户端代码修改如下：

```
//designpatterns.simplefactory.Client.java
package designpatterns.simplefactory;
```

```
public class Client {
    public static void main(String args[]) {
        Chart chart;
        String type = XMLUtil.getChartType(); //读取配置文件中的参数
        chart = ChartFactory.getChart(type); //创建产品对象
        chart.display();
    }
}
```

编译并运行程序,输出结果如下:

```
创建柱状图!
初始化设置柱状图!
显示柱状图!
```

不难发现,在上述客户端代码中不包含任何与具体图表对象相关的信息,如果需要更换具体图表对象,只需修改配置文件 config.xml,无须修改任何源代码,符合开闭原则。

3.5 关于创建对象与使用对象

本节将讨论工厂类的作用以及如何通过工厂类来创建对象。在一个面向对象软件系统中与一个对象相关的职责通常有 3 类,即对象本身所具有的职责、创建对象的职责和使用对象的职责。对象本身的职责比较容易理解,就是对象自身所具有的一些数据和行为,可通过一些公开的(public)方法来实现它的职责。

在 Java 语言中通常有以下几种创建对象的方式:

(1) 使用 new 关键字直接创建对象。

(2) 通过反射机制创建对象(第 4 章将学习此方式)。

(3) 通过克隆方法创建对象(第 7 章将学习此方式)。

(4) 通过工厂类创建对象。

毫无疑问,在客户端代码中直接使用 new 关键字是最简单的一种创建对象的方式,但是它的灵活性较差,下面通过一个简单的示例来加以说明:

```
public class LoginAction {
    private UserDAO udao;

    public LoginAction() {
        udao = new JDBCUserDAO(); //创建对象
    }

    public void execute() {
        //其他代码
        udao.findUserById();     //使用对象
        //其他代码
    }
}
```

在以上代码中，在 LoginAction 类中定义了一个 UserDAO 类型的对象 udao，在 LoginAction 的构造函数中创建了 JDBCUserDAO 类型的 udao 对象，并在 execute()方法中调用了 udao 对象的 findUserById()方法。LoginAction 类负责创建了一个 UserDAO 子类的对象并使用 UserDAO 的方法来完成相应的业务处理，也就是说，LoginAction 既负责 udao 的创建又负责 udao 的使用，创建对象和使用对象的职责耦合在一起，这样的设计会导致一个很严重的问题，即如果在 LoginAction 中希望能够使用 UserDAO 的另一个子类，例如 HibernateUserDAO 类型的对象，必须修改 LoginAction 类的源代码，这将违背开闭原则。

当遇到这种情况时，最常用的一种解决方法是将 udao 对象的创建职责从 LoginAction 类中移除，在 LoginAction 类之外创建对象，由专门的工厂类来负责 udao 对象的创建。通过引入工厂类，让客户类(例如 LoginAction)不涉及对象的创建，对象的创建者也不会涉及对象的使用。引入工厂类 UserDAOFactory 之后的结构如图 3-4 所示。

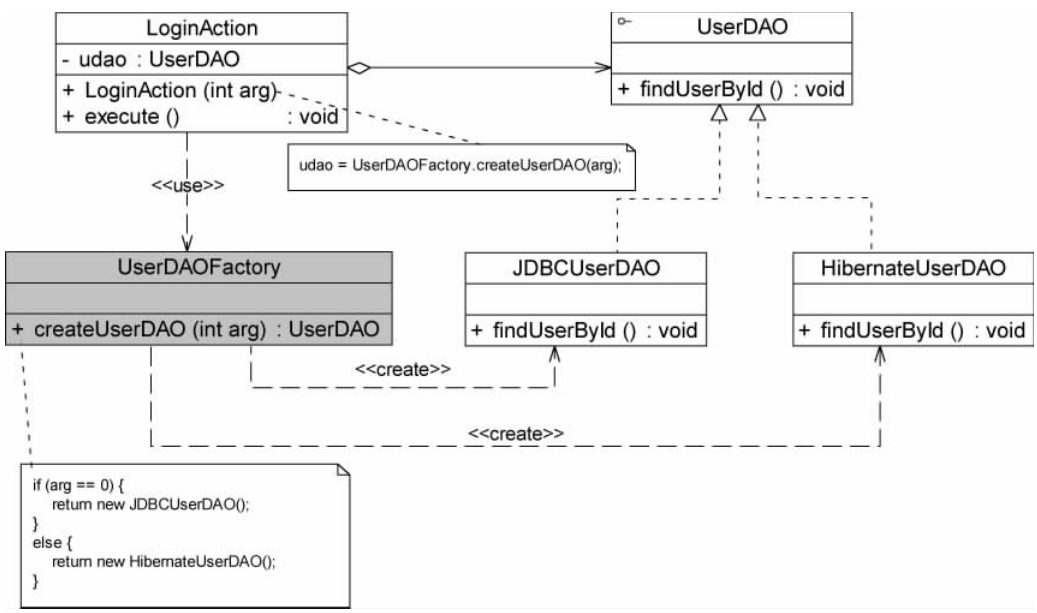

图 3-4 引入工厂类之后的结构图

工厂类的引入将降低因为产品或工厂类改变所造成的维护工作量。如果 UserDAO 的某个子类的构造函数发生改变或者需要添加或移除不同的子类，只需要维护 UserDAOFactory 的代码，而不会影响到 LoginAction；如果 UserDAO 的接口发生改变，例如添加、移除方法或改变方法名，只需要修改 LoginAction，不会给 UserDAOFactory 带来任何影响。

所有的工厂模式都强调一点：两个类 A 和 B 之间的关系应该仅仅是 A 创建 B 或是 A 使用 B，而不能两种关系都有。将对象的创建和使用分离，使得系统更加符合单一职责原则，有利于对功能的复用和系统的维护。

此外，将对象的创建和使用分离还有一个好处：防止用来实例化一个类的数据和代码在多个类中到处都是，可以将有关创建的知识搬移到一个工厂类中。因为有时候创建一个对象不只是简单地调用其构造函数，还需要设置一些参数，可能还需要配置环境，如果将这

些代码散落在每一个创建对象的客户类中，势必会出现代码重复、创建蔓延的问题，而这些客户类其实无须承担对象的创建工作，它们只需使用已创建好的对象就可以了，此时可以引入工厂类来封装对象的创建逻辑和客户代码的实例化配置选项。

使用工厂类还有一个优点，一个类可能拥有多个构造函数，而在 Java、C＃等语言中构造函数的名字都与类名相同，客户端只能通过传入不同的参数来调用不同的构造函数创建对象，单从构造函数和参数列表很难了解不同构造函数所构造的产品之间的差异。但如果将对象的创建过程封装在工厂类中，可以提供一系列名字完全不同的工厂方法，每一个工厂方法对应一个构造函数，客户端可以用一种更加可读、易懂的方式来创建对象，而且从一组工厂方法中选择一个意义明确的工厂方法比从一组名称相同、参数不同的构造函数中选择一个构造函数要方便很多，如图 3-5 所示。

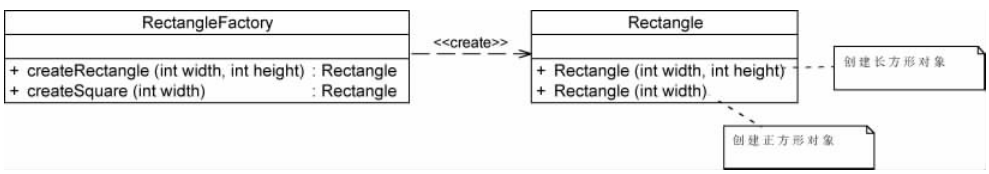

图 3-5　矩形工厂与矩形类

在图 3-5 中，矩形工厂类 RectangleFactory 提供了两个工厂方法 createRectangle()和 createSquare()，一个用于创建长方形，一个用于创建正方形，这两个方法比直接通过构造函数来创建长方形或正方形对象的意义更加明确，也在一定程度上降低了客户端调用时出错的概率。

但是并不需要为系统中的每一个类都配备一个工厂类，如果一个类很简单，而且不存在太多变化，其构造过程也很简单，此时就无须为其提供工厂类，直接在使用之前实例化即可。例如 Java 语言中的 String 类，就无须为它专门提供一个 StringFactory，这样做反而会导致工厂泛滥，增加系统的复杂度。

以上关于创建对象和使用对象的讨论适用于各种工厂模式，包括第 4 章将要介绍的工厂方法模式和第 5 章将要介绍的抽象工厂模式。

3.6　简单工厂模式的简化

有时候，为了简化简单工厂模式，可以将抽象产品类和工厂类合并，将静态工厂方法移至抽象产品类中，如图 3-6 所示。

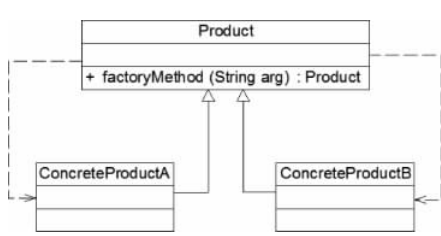

图 3-6　简化的简单工厂模式

在图 3-6 中，客户端可以通过调用产品父类的静态工厂方法，根据参数的不同创建不同类型的产品子类对象，这种做法在很多类库和框架中也广泛存在。

3.7 简单工厂模式优/缺点与适用环境

简单工厂模式提供了专门的工厂类用于创建对象，将对象的创建和对象的使用分离开，它作为一种最简单的工厂模式在软件开发中得到了较为广泛的应用。

3.7.1 简单工厂模式优点

简单工厂模式的优点主要如下：

(1) 工厂类包含必要的判断逻辑，可以决定在什么时候创建哪一个产品类的实例，客户端可以免除直接创建产品对象的职责，而仅仅“消费”产品，简单工厂模式实现了对象创建和使用的分离。

(2) 客户端无须知道所创建的具体产品类的类名，只需要知道具体产品类所对应的参数即可，对于一些复杂的类名，通过简单工厂模式可以在一定程度上减少使用者的记忆量。

(3) 通过引入配置文件，可以在不修改任何客户端代码的情况下更换和增加新的具体产品类，在一定程度上提高了系统的灵活性。

3.7.2 简单工厂模式缺点

简单工厂模式的缺点主要如下：

(1) 由于工厂类集中了所有产品的创建逻辑，职责过重，一旦不能正常工作，整个系统都要受到影响。

(2) 使用简单工厂模式势必会增加系统中类的个数(引入了新的工厂类)，增加了系统的复杂度和理解难度。

(3) 系统扩展困难，一旦添加新产品就不得不修改工厂逻辑，在产品类型较多时有可能造成工厂逻辑过于复杂，不利于系统的扩展和维护。

(4) 简单工厂模式由于使用了静态工厂方法，造成工厂角色无法形成基于继承的等级结构。

3.7.3 简单工厂模式适用环境

在以下情况下可以考虑使用简单工厂模式：

(1) 工厂类负责创建的对象比较少，由于创建的对象较少，不会造成工厂方法中的业务逻辑太过复杂。

(2) 客户端只知道传入工厂类的参数，对于如何创建对象并不关心。

3.8 本章小结

1. 创建型模式关注对象的创建过程，它对类的实例化过程进行了抽象，能够将软件模块中对象的创建和对象的使用分离，对用户隐藏了类的实例的创建细节。在 GoF 设计模式

中一共包含 5 种创建型模式，通常将简单工厂模式作为学习其他工厂模式的基础，简单工厂模式不是 GoF 设计模式。

2. 在简单工厂模式中定义一个工厂类，它可以根据参数的不同返回不同类的实例，被创建的实例通常都具有共同的父类。简单工厂模式是一种类创建型模式。

3. 简单工厂模式包含工厂角色、抽象产品角色和具体产品角色 3 个角色。其中，工厂角色是简单工厂模式的核心，负责实现创建所有产品实例的内部逻辑；抽象产品角色是工厂类创建的所有对象的父类，封装了各种产品对象的公有方法；具体产品角色是简单工厂模式的创建目标，所有被创建的对象都充当这个角色的某个具体类的实例。

4. 简单工厂模式的优点主要在于实现了对象创建和使用的分离；客户端无须知道所创建的具体产品类的类名，只需要知道具体产品类所对应的参数即可；通过引入配置文件，可以在不修改任何客户端代码的情况下更换和增加新的具体产品类，在一定程度上提高了系统的灵活性。其缺点主要在于工厂类集中了所有产品的创建逻辑，职责过重，一旦不能正常工作，整个系统都要受到影响；增加了系统中类的个数且增加了系统的复杂度和理解难度；系统扩展困难，一旦添加新产品就不得不修改工厂逻辑且工厂角色无法形成基于继承的等级结构。

5. 简单工厂模式适用于以下环境：工厂类负责创建的对象比较少，由于创建的对象较少，不会造成工厂方法中的业务逻辑太过复杂；客户端只知道传入工厂类的参数，对于如何创建对象并不关心。

6. 将对象的创建和使用分离，使得系统更加符合单一职责原则，有利于对功能的复用和系统的维护。

3.9 习题

1. 在简单工厂模式中，如果需要增加新的具体产品，通常需要修改(　　)的源代码。

   A. 抽象产品类　　B. 其他具体产品类

   C. 工厂类　　D. 客户类

2. 以下关于简单工厂模式的叙述错误的是(　　)。

   A. 简单工厂模式可以根据参数的不同返回不同的产品类的实例

   B. 简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类

   C. 简单工厂模式可以减少系统中类的个数，简化系统的设计，使得系统更易于理解

   D. 系统的扩展困难，在添加新的产品时需要修改工厂的业务逻辑，违背了开闭原则

3. 以下代码使用了(　　)模式。

```
public abstract class Product {
    public abstract void process();
}

public class ConcreteProductA extends Product {
```

```
    public void process() {...}
}

public class ConcreteProductB extends Product{
    public void process() {...}
}

public class Factory {
    public static Product createProduct (char type) {
        switch(type) {
            case 'A':
            return new ConcreteProductA(); break;
            case 'B':
            return new ConcreteProductB(); break;
            ...
            }
        }
}
```

A. Simple Factory　　B. Factory Method

C. Abstract Factory　　D. 未用任何设计模式

4. 使用简单工厂模式模拟女娲(Nvwa)造人(Person),如果向造人的工厂方法传入参数“M”,则返回一个男人(Man)对象,如果传入参数“W”,则返回一个女人(Woman)对象,绘制相应的类图并使用 Java 语言模拟实现该场景。现需要增加一个新的机器人(Robot)类,如果传入参数“R”,则返回一个机器人对象,对代码进行修改并注意“女娲”类的变化。

5. 使用简单工厂模式设计一个可以创建不同几何形状(Shape)的绘图工具类,例如圆形(Circle)、矩形(Rectangle)和三角形(Triangle)等,每个几何图形均具有绘制 draw()和擦除 erase()两个方法,要求在绘制不支持的几何图形时抛出一个 UnsupportedShapeException 异常,绘制类图并使用 Java 语言编程模拟实现。