

《JAVA与模式》之工厂方法模式

在闫宏博士的《JAVA与模式》一书中开头是这样描述工厂方法模式的：

**工厂方法模式是类的创建模式，又叫做虚拟构造子(Virtual Constructor)模式或者多态性工厂 ( Polymorphic Factory ) 模式。**

**工厂方法模式的用意是定义一个创建产品对象的工厂接口，将实际创建工作推迟到子类中。**

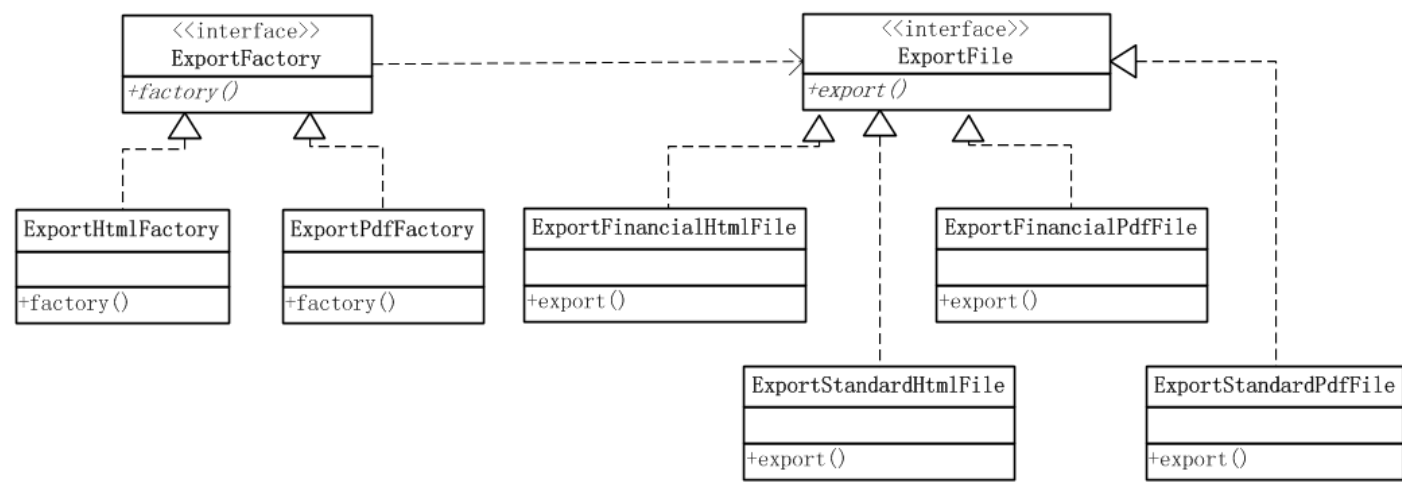
那么**工厂方法模式**是在什么场景下使用呢，下面就以本人的理解举例说明：

相信很多人都做过导入导出功能，就拿导出功能来说。有这么一个需求：XX系统需要支持对数据库中的员工薪资进行导出，并且支持多种格式如：HTML、CSV、PDF等，每种格式导出的结构有所不同，比如：财务跟其他人对导出薪资的HTML格式要求可能会不一样，因为财务可能需要特定的格式方便核算或其他用途。

如果使用简单工厂模式，则工厂类必定过于臃肿。因为简单工厂模式只有一个工厂类，它需要处理所有的创建的逻辑。假如以上需求暂时只支持3种导出的格式以及2种导出的结构，那工厂类则需要6个if else来创建6种不同的类型。如果日后需求不断增加，则后果不堪设想。

这时候就需要工厂方法模式来处理以上需求。在工厂方法模式中，核心的工厂类不再负责所有的对象的创建，而是将具体创建的工作交给子类去做。这个核心类则摇身一变，成为了一个抽象工厂角色，仅负责给出具体工厂子类必须实现的接口，而不接触哪一个类应当被实例化这种细节。

这种进一步抽象化的结果，使这种工厂方法模式可以用来允许系统在不修改具体工厂角色的情况下引进新的产品，这一特点无疑使得工厂方法模式具有超过简单工厂模式的优越性。下面就针对以上需求设计UML图：



从上图可以看出，这个使用的工厂方法模式的系统涉及到以下角色：

**抽象工厂 ( ExportFactory ) 角色：**担任这个角色的是工厂方法模式的核心，任何在模式中创建对象的工厂类必须实现这个接口。在实际的系统中，这个角色也常常使用抽象类实现。

**具体工厂 ( ExportHtmlFactory、ExportPdfFactory ) 角色：**担任这个角色的是实现了抽象工厂接口的具体JAVA类。具体工厂角色含有与业务密切相关的逻辑，并且受到使用者的调用以创建导出类（如：ExportStandardHtmlFile）。

**抽象导出 ( ExportFile ) 角色：**工厂方法模式所创建的对象超类，也就是所有导出类的共同父类或共同拥有的接口。在实际的系统中，这个角色也常常使用抽象类实现。

**具体导出 ( ExportStandardHtmlFile等 ) 角色：**这个角色实现了抽象导出 ( ExportFile ) 角色所声明的接口，工厂方法模式所创建的每一个对象都是某个具体导出角色的实例。

源代码

首先是抽象工厂角色源代码。它声明了一个工厂方法，要求所有的具体工厂角色都实现这个工厂方法。参数type表示导出的格式是哪一种结构，如：导出HTML格式有两种结构，一种是标准结构，一种是财务需要的结构。

```
public interface ExportFactory {
    public ExportFile factory(String type);
}
```

具体工厂角色类源代码：

```
public class ExportHtmlFactory implements ExportFactory{

    @Override
    public ExportFile factory(String type) {
        // TODO Auto-generated method stub
        if("standard".equals(type)){

            return new ExportStandardHtmlFile();
        }
    }
}
```

```

        }else if("financial".equals(type)){

            return new ExportFinancialHtmlFile();

        }else{
            throw new RuntimeException("没有找到对象");
        }
    }
}

```



```

public class ExportPdfFactory implements ExportFactory {

    @Override
    public ExportFile factory(String type) {
        // TODO Auto-generated method stub
        if("standard".equals(type)){

            return new ExportStandardPdfFile();

        }else if("financial".equals(type)){

            return new ExportFinancialPdfFile();

        }else{
            throw new RuntimeException("没有找到对象");
        }
    }

}

```



抽象导出角色类源代码：

```

public interface ExportFile {
    public boolean export(String data);
}

```

具体导出角色类源代码，通常情况下这个类会有复杂的业务逻辑。



```

public class ExportFinancialHtmlFile implements ExportFile{

    @Override
    public boolean export(String data) {
        // TODO Auto-generated method stub
        /**
         * 业务逻辑
         */
        System.out.println("导出财务版HTML文件");
        return true;
    }

}

```



```

public class ExportFinancialPdfFile implements ExportFile{

    @Override
    public boolean export(String data) {
        // TODO Auto-generated method stub
        /**
         * 业务逻辑
         */
        System.out.println("导出财务版PDF文件");
        return true;
    }

}

```





```
public class ExportStandardHtmlFile implements ExportFile{

    @Override
    public boolean export(String data) {
        // TODO Auto-generated method stub
        /**
         * 业务逻辑
         */
        System.out.println("导出标准HTML文件");
        return true;
    }

}
```




```
public class ExportStandardPdfFile implements ExportFile {

    @Override
    public boolean export(String data) {
        // TODO Auto-generated method stub
        /**
         * 业务逻辑
         */
        System.out.println("导出标准PDF文件");
        return true;
    }

}
```




客户端角色类源代码：



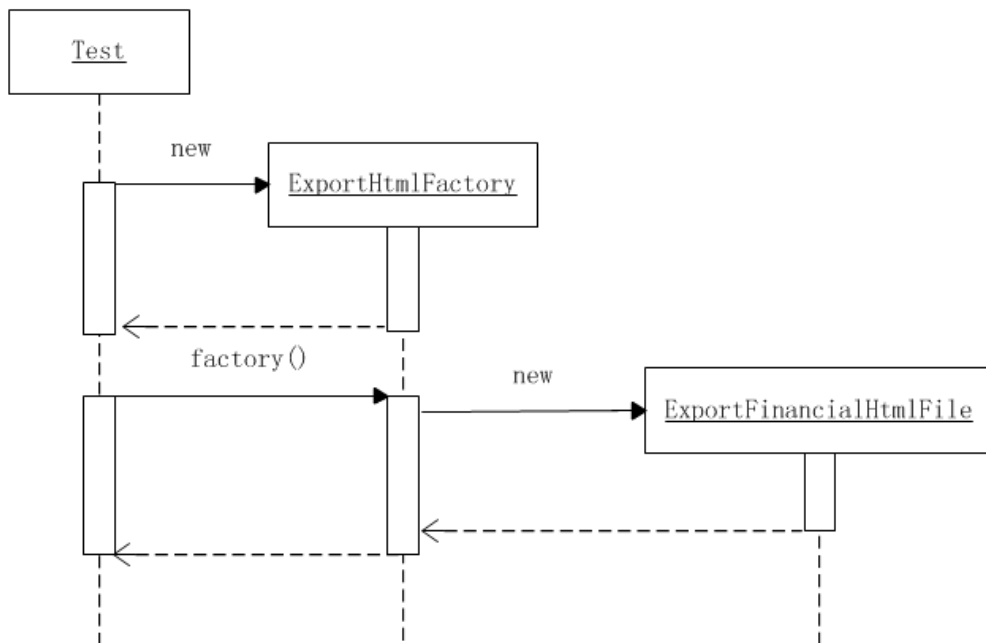
```
public class Test {

    /**
     * @param args
     */
    public static void main(String[] args) {
        // TODO Auto-generated method stub
        String data = "";
        ExportFactory exportFactory = new ExportHtmlFactory();
        ExportFile ef = exportFactory.factory("financial");
        ef.export(data);
    }

}
```



**工厂方法模式的活动序列图**



客户端创建ExportHtmlFactory对象，这时客户端所持有变量的静态类型为ExportFactory，而实际类型为ExportHtmlFactory。然后客户端调用ExportHtmlFactory对象的工厂方法factory()，接着后者调用ExportFinancialHtmlFile的构造子创建出导出对象。

## 工厂方法模式和简单工厂模式

工厂方法模式和简单工厂模式在结构上的不同很明显。工厂方法模式的核心是一个抽象工厂类，而简单工厂模式把核心放在一个具体类上。

工厂方法模式退化后可以变得很像简单工厂模式。设想如果非常确定一个系统只需要一个具体工厂类，那么不妨把抽象工厂类合并到具体工厂类中去。由于只有一个具体工厂类，所以不妨将工厂方法改为静态方法，这时候就得到了简单工厂模式。

如果系统需要加入一个新的导出类型，那么所需要的就是向系统中加入一个这个导出类以及所对应的工厂类。没有必要修改客户端，也没有必要修改抽象工厂角色或者其他已有的具体工厂角色。对于增加新的导出类型而言，这个系统完全支持“开-闭原则”。

## 完结

一个应用系统是由多人开发的，导出的功能是你实现的，但是使用者（调用这个方法的人）可能却是其他人。这时候你应该设计的足够灵活并尽可能降低两者之间的耦合度，当你修改或增加一个新的功能时，使用者不需要修改任何地方。假如你的设计不够灵活，那么将无法面对客户多变的需求。可能一个极小的需求变更，都会使你的代码结构发生改变，并导致其他使用你所提供的接口的人都要修改他们的代码。牵一处而动全身，这就使得日后这个系统将难以维护。