工厂模式

基本概念

工厂模式(简单工厂与工厂模式)可以认为是日常项目中使用最为频繁的一种设计模式

根据不同编码格式(JSON, XML)选择不同的解析器，根据不同的图片格式(JPEG, PNG)选择不同的图片处理方法，根据不同的客户(个人客户、企业客户)选择查询不同的数据库....

工厂模式的目的就是为了应对现实世界中复杂且多样的分类

工厂模式可分为简单工厂模式、工厂模式以及抽象工厂模式，其中抽象工厂模式实现较为复杂，很少用到

简单工厂模式

简单工厂模式的范围及广，工厂函数不一定必须返回一个实例，返回一个方法也可以称之为简单工厂模式

Java

```java
public interface IParser {}

public class JsonParser implements IParser {}

public class XMLParser implements IParser {}

public class YAMLParser implements IParser {}

public class ParserFacroty {

    public static IParser getParser(String format) {
        IParser parser = null;
        if (format == "JSON") {
            parser = new JsonParser();
        } else if (format == "XML") {
            parser = new XMLParser();
        } else if (format == "YAML") {
            parser = new YAMLParser();
        }

        return parser;
    }
}
```

以数据解析器为例，通过传入工厂函数的参数来判断并选择对应的解析器

```java
public class ParserFacroty {
  private static final Map<String, IParser> parsers = new HashMap<>();

  static {
    parsers.put("json", new JsonParser());
    parsers.put("xml", new XMLParser());
    parsers.put("yaml", new YAMLParser());
  }

  public static IParser getParser(String format) {
    if (format == null || format.isEmpty()) {
      return null;
    }
    IParser parser = parsers.get(format);
    return parser;
  }
}
```

有时候不想用if-else的形式，也可以使用Map来对其进行封装

使用Map的工厂函数同时也是一个单例模式的应用。相较于在条件语句中创建实例，该方法更节省内存空间

当系统需要添加一个新的解析类，例如解析.ini文件格式，则需要修改ParserFactory，从而不满足开放-封闭原则

但是，使用简单工厂模式的前提就是开发人员认为其变动频率非常之低，甚至在整个产品生命周期内不会发生修改。那么，虽然违反开闭原则，但是权衡之下，简单工厂模式仍然值得我们最为首选项

并且使用Map的实现既能使代码更加简洁、清晰，同时能节省一部分的内存空间，添加一个新的解析类也更为简单

工厂模式

工厂模式实际上是对简单工厂模式的进一步封装，其目的在于将复杂类的创建从工厂类中剥离出去，并使得代码的可拓展性更强

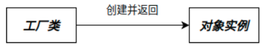
以Map版本的简单工厂为例，假设JsonParfer()使用了依赖注入，需在构造函数中将所有依赖的类注入，那么ParserFactory工厂类将会变得异常复杂

XXXParser依赖同一个类

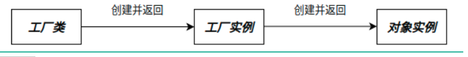
事情就比较简单，ParserFactory在创建对象时注入进去即可

XXXParser依赖不同的类

ParserFactory在创建对象时需要根据不同的类传入不同的类对象，工厂类将会变得相当臃肿和复杂



所以，工厂模式认为，复杂的对象构建应该下沉到底层，工厂类只提供工厂实例，由工厂实例再创建出具体的类对象。而不是工厂类直接创建复杂的对象实例



工厂模式

```java
public interface IParser {}

public class JsonParser implements IParser {}

public interface IParserFacroty {
    IParser getParser();  /* 创建工厂接口 */
}

public class JsonParserFactory implements IParserFacroty {
    @Override
    public IParser getParser() {
        /* 初始化JsonParser所依赖的类对象 */
        return new JsonParser();
    }
}

public class ParserFactoryMap {
    private static final Map<String, IParserFacroty> cachedFactories = new HashMap<>();

    static {
        cachedFactories.put("json", new JsonParserFactory());
        cachedFactories.put("json", new XMLParserFactory());
        cachedFactories.put("json", new YAMLParserFactory());
    }

    public static IParserFacroty getParserFactory(String format) {
        /* 不直接获取具体的Parser实例，而是获取Parser对应的工厂 */
        if (format == null || format.isEmpty()) {
            return null;
        }

        IParserFacroty parserFactory = cachedFactories.get(format);
        return parserFactory;
    }
}
```

工厂方法其实就是在简单工厂方法之上，又添加了一层抽象而已，原因可能是因为具体的类对象在创建实例时较为复杂，且包含不同的具体依赖

当每个对象的创建比较简单的时候，使用简单工厂模式即可。当对象的创建比较复杂，相关依赖较多时，为了避免设计出一个过于庞大且复杂的工厂类，应使用工厂模式，将复杂的创建逻辑下沉至底部

什么是复杂对象

在应用工厂模式时，判断一个对象的复杂性并不是看其内部逻辑的复杂性，而是判断其创建实例的复杂性

判断标准通常有2个方面

在构造函数中包含大量的逻辑判断

例如，构造函数需要对用户传入的参数进行转义操作，并根据不同的参数采取不同的转义动作

对象的创建包含较多外部依赖类

类和类的依赖关系非常常见，为了不将依赖的类硬编码进对象中，通常采用依赖注入的方式将依赖的对象注入，以此降低耦合性