## 我为什么说没事不要随便用工厂模式创建对象？

一般情况下，工厂模式分为三种更加细分的类型：简单工厂、工厂方法和抽象工厂。不过，在 GoF 的《设计模式》一书中，它将简单工厂模式看作是工厂方法模式的一种特例，所以工厂模式只被分成了工厂方法和抽象工厂两类。实际上，前面一种分类方法更加常见，所以，在今天的讲解中，我们沿用第一种分类方法。

在这三种细分的工厂模式中，简单工厂、工厂方法原理比较简单，在实际的项目中也比较常用。而抽象工厂的原理稍微复杂点，在实际的项目中相对也不常用。所以，我们今天讲解的重点是前两种工厂模式。对于抽象工厂，你稍微了解一下即可。

除此之外，我们讲解的重点也不是原理和实现，因为这些都很简单，重点还是带你搞清楚应用场景：什么时候该用工厂模式？相对于直接 new 来创建对象，用工厂模式来创建究竟有什么好处呢？

### 简单工厂（Simple Factory）

首先，我们来看，什么是简单工厂模式。我们通过一个例子来解释一下。

在下面这段代码中，我们根据配置文件的后缀（json、xml、yaml、properties），选择不同的解析器（JsonRuleConfigParser、XmlRuleConfigParser……），将存储在文件中的配置解析成内存对象 RuleConfig。

public class RuleConfigSource {

public RuleConfig load(String ruleConfigFilePath) {

String ruleConfigFileExtension = getFileExtension(ruleConfigFilePath);

IRuleConfigParser parser = null;

if ("json".equalsIgnoreCase(ruleConfigFileExtension)) {

parser = new JsonRuleConfigParser();

} else if ("xml".equalsIgnoreCase(ruleConfigFileExtension)) {

parser = new XmlRuleConfigParser();

} else if ("yaml".equalsIgnoreCase(ruleConfigFileExtension)) {

parser = new YamlRuleConfigParser();

} else if ("properties".equalsIgnoreCase(ruleConfigFileExtension)) {

parser = new PropertiesRuleConfigParser();

} else {

throw new InvalidRuleConfigException(

"Rule config file format is not supported: " + ruleConfigFilePath);

}

String configText = "";

//从ruleConfigFilePath文件中读取配置文本到configText中

RuleConfig ruleConfig = parser.parse(configText);

return ruleConfig;

}

private String getFileExtension(String filePath) {

//...解析文件名获取扩展名，比如rule.json，返回json

return "json";

}

}

为了让代码逻辑更加清晰，可读性更好，我们要善于将功能独立的代码块封装成函数。按照这个设计思路，我们可以将代码中涉及 parser 创建的部分逻辑剥离出来，抽象成 createParser() 函数。重构之后的代码如下所示：

public RuleConfig load(String ruleConfigFilePath) {

String ruleConfigFileExtension = getFileExtension(ruleConfigFilePath);

IRuleConfigParser parser = createParser(ruleConfigFileExtension);

if (parser == null) {

throw new InvalidRuleConfigException(

"Rule config file format is not supported: " + ruleConfigFilePath);

}

String configText = "";

//从ruleConfigFilePath文件中读取配置文本到configText中

RuleConfig ruleConfig = parser.parse(configText);

return ruleConfig;

}

private String getFileExtension(String filePath) {

//...解析文件名获取扩展名，比如rule.json，返回json

return "json";

}

private IRuleConfigParser createParser(String configFormat) {

IRuleConfigParser parser = null;

if ("json".equalsIgnoreCase(configFormat)) {

parser = new JsonRuleConfigParser();

} else if ("xml".equalsIgnoreCase(configFormat)) {

parser = new XmlRuleConfigParser();

} else if ("yaml".equalsIgnoreCase(configFormat)) {

parser = new YamlRuleConfigParser();

} else if ("properties".equalsIgnoreCase(configFormat)) {

parser = new PropertiesRuleConfigParser();

}

return parser;

}

}

为了让类的职责更加单一、代码更加清晰，我们还可以进一步将 createParser() 函数剥离到一个独立的类中，让这个类只负责对象的创建。而这个类就是我们现在要讲的简单工厂模式类。具体的代码如下所示：

public class RuleConfigSource {

public RuleConfig load(String ruleConfigFilePath) {

String ruleConfigFileExtension = getFileExtension(ruleConfigFilePath);

IRuleConfigParser parser = RuleConfigParserFactory.createParser(ruleConfigFileExtension);

if (parser == null) {

throw new InvalidRuleConfigException(

"Rule config file format is not supported: " + ruleConfigFilePath);

}

String configText = "";

//从ruleConfigFilePath文件中读取配置文本到configText中

RuleConfig ruleConfig = parser.parse(configText);

return ruleConfig;

}

private String getFileExtension(String filePath) {

//...解析文件名获取扩展名，比如rule.json，返回json

return "json";

}

}

public class RuleConfigParserFactory {

public static IRuleConfigParser createParser(String configFormat) {

IRuleConfigParser parser = null;

if ("json".equalsIgnoreCase(configFormat)) {

parser = new JsonRuleConfigParser();

} else if ("xml".equalsIgnoreCase(configFormat)) {

parser = new XmlRuleConfigParser();

} else if ("yaml".equalsIgnoreCase(configFormat)) {

parser = new YamlRuleConfigParser();

} else if ("properties".equalsIgnoreCase(configFormat)) {

parser = new PropertiesRuleConfigParser();

}

return parser;

}

}

大部分工厂类都是以“Factory”这个单词结尾的，但也不是必须的，比如 Java 中的 DateFormat、Calender。除此之外，工厂类中创建对象的方法一般都是 create 开头，比如代码中的 createParser()，但有的也命名为 getInstance()、createInstance()、newInstance()，有的甚至命名为 valueOf()（比如 Java String 类的 valueOf() 函数）等等，这个我们根据具体的场景和习惯来命名就好。

在上面的代码实现中，我们每次调用 RuleConfigParserFactory 的 createParser() 的时候，都要创建一个新的 parser。实际上，如果 parser 可以复用，为了节省内存和对象创建的时间，我们可以将 parser 事先创建好缓存起来。当调用 createParser() 函数的时候，我们从缓存中取出 parser 对象直接使用。

这有点类似单例模式和简单工厂模式的结合，具体的代码实现如下所示。在接下来的讲解中，我们把上一种实现方法叫作简单工厂模式的第一种实现方法，把下面这种实现方法叫作简单工厂模式的第二种实现方法。

public class RuleConfigParserFactory {

private static final Map<String, RuleConfigParser> cachedParsers = new HashMap<>();

**static** {

cachedParsers.put("json", new JsonRuleConfigParser());

cachedParsers.put("xml", new XmlRuleConfigParser());

cachedParsers.put("yaml", new YamlRuleConfigParser());

cachedParsers.put("properties", new PropertiesRuleConfigParser());

}

public static IRuleConfigParser createParser(String configFormat) {

if (configFormat == null || configFormat.isEmpty()) {

return null;//返回null还是IllegalArgumentException全凭你自己说了算

}

IRuleConfigParser parser = cachedParsers.get(configFormat.toLowerCase());

return parser;

}

}

对于上面两种简单工厂模式的实现方法，如果我们要添加新的 parser，那势必要改动到 RuleConfigParserFactory 的代码，那这是不是违反开闭原则呢？实际上，如果不是需要频繁地添加新的 parser，只是偶尔修改一下 RuleConfigParserFactory 代码，稍微不符合开闭原则，也是完全可以接受的。

除此之外，在 RuleConfigParserFactory 的第一种代码实现中，有一组 if 分支判断逻辑，是不是应该用多态或其他设计模式来替代呢？实际上，如果 if 分支并不是很多，代码中有 if 分支也是完全可以接受的。应用多态或设计模式来替代 if 分支判断逻辑，也并不是没有任何缺点的，它虽然**提高了代码的扩展性，更加符合开闭原则**，**但也增加了类的个数，牺牲了代码的可读性**。关于这一点，我们在后面章节中会详细讲到。

总结一下，尽管简单工厂模式的代码实现中，有多处 if 分支判断逻辑，违背开闭原则，但权衡扩展性和可读性，这样的代码实现在大多数情况下（比如，不需要频繁地添加 parser，也没有太多的 parser）是没有问题的。

### 工厂方法（Factory Method）

如果我们非得要将 if 分支逻辑去掉，那该怎么办呢？比较经典处理方法就是利用多态。按照多态的实现思路，对上面的代码进行重构。重构之后的代码如下所示：

public interface IRuleConfigParserFactory {

IRuleConfigParser createParser();

}

public class JsonRuleConfigParserFactory implements IRuleConfigParserFactory {

@Override

public IRuleConfigParser createParser() {

return new JsonRuleConfigParser();

}

}

public class XmlRuleConfigParserFactory implements IRuleConfigParserFactory {

@Override

public IRuleConfigParser createParser() {

return new XmlRuleConfigParser();

}

}

public class YamlRuleConfigParserFactory implements IRuleConfigParserFactory {

@Override

public IRuleConfigParser createParser() {

return new YamlRuleConfigParser();

}

}

public class PropertiesRuleConfigParserFactory implements IRuleConfigParserFactory {

@Override

public IRuleConfigParser createParser() {

return new PropertiesRuleConfigParser();

}

}

实际上，这就是工厂方法模式的典型代码实现。这样当我们新增一种 parser 的时候，只需要新增一个实现了 IRuleConfigParserFactory 接口的 Factory 类即可。所以，工厂方法模式比起简单工厂模式更加符合开闭原则。

从上面的工厂方法的实现来看，一切都很完美，但是实际上存在挺大的问题。问题存在于这些工厂类的使用上。接下来，我们看一下，如何用这些工厂类来实现 RuleConfigSource 的 load() 函数。具体的代码如下所示：

public class RuleConfigSource {

public RuleConfig load(String ruleConfigFilePath) {

String ruleConfigFileExtension = getFileExtension(ruleConfigFilePath);

IRuleConfigParserFactory parserFactory = null;

if ("json".equalsIgnoreCase(ruleConfigFileExtension)) {

parserFactory = new JsonRuleConfigParserFactory();

} else if ("xml".equalsIgnoreCase(ruleConfigFileExtension)) {

parserFactory = new XmlRuleConfigParserFactory();

} else if ("yaml".equalsIgnoreCase(ruleConfigFileExtension)) {

parserFactory = new YamlRuleConfigParserFactory();

} else if ("properties".equalsIgnoreCase(ruleConfigFileExtension)) {

parserFactory = new PropertiesRuleConfigParserFactory();

} else {

throw new InvalidRuleConfigException("Rule config file format is not supported: " + ruleConfigFilePath);

}

IRuleConfigParser parser = parserFactory.createParser();

String configText = "";

//从ruleConfigFilePath文件中读取配置文本到configText中

RuleConfig ruleConfig = parser.parse(configText);

return ruleConfig;

}

private String getFileExtension(String filePath) {

//...解析文件名获取扩展名，比如rule.json，返回json

return "json";

}

}

从上面的代码实现来看，工厂类对象的创建逻辑又耦合进了 load() 函数中，跟我们最初的代码版本非常相似，引入工厂方法非但没有解决问题，反倒让设计变得更加复杂了。那怎么来解决这个问题呢？

**我们可以为工厂类再创建一个简单工厂，也就是工厂的工厂，用来创建工厂类对象**。这段话听起来有点绕，我把代码实现出来了，你一看就能明白了。其中，RuleConfigParserFactoryMap 类是创建工厂对象的工厂类，getParserFactory() 返回的是缓存好的单例工厂对象。

public class RuleConfigSource {

public RuleConfig load(String ruleConfigFilePath) {

String ruleConfigFileExtension = getFileExtension(ruleConfigFilePath);

IRuleConfigParserFactory parserFactory = RuleConfigParserFactoryMap.getParserFactory(ruleConfigFileExtension);

if (parserFactory == null) {

throw new InvalidRuleConfigException("Rule config file format is not supported: " + ruleConfigFilePath);

}

IRuleConfigParser parser = parserFactory.createParser();

String configText = "";

//从ruleConfigFilePath文件中读取配置文本到configText中

RuleConfig ruleConfig = parser.parse(configText);

return ruleConfig;

}

private String getFileExtension(String filePath) {

//...解析文件名获取扩展名，比如rule.json，返回json

return "json";

}

}

//因为工厂类只包含方法，不包含成员变量，完全可以复用，

//不需要每次都创建新的工厂类对象，所以，简单工厂模式的第二种实现思路更加合适。

public class RuleConfigParserFactoryMap { //工厂的工厂

private static final Map<String, IRuleConfigParserFactory> cachedFactories = new HashMap<>();

static {

cachedFactories.put("json", new JsonRuleConfigParserFactory());

cachedFactories.put("xml", new XmlRuleConfigParserFactory());

cachedFactories.put("yaml", new YamlRuleConfigParserFactory());

cachedFactories.put("properties", new PropertiesRuleConfigParserFactory());

}

public static IRuleConfigParserFactory getParserFactory(String type) {

if (type == null || type.isEmpty()) {

return null;

}

IRuleConfigParserFactory parserFactory = cachedFactories.get(type.toLowerCase());

return parserFactory;

}

}

当我们需要添加新的规则配置解析器的时候，我们只需要创建新的 parser 类和 parser factory 类，并且在 RuleConfigParserFactoryMap 类中，将新的 parser factory 对象添加到 cachedFactories 中即可。代码的改动非常少，基本上符合开闭原则。

实际上，对于规则配置文件解析这个应用场景来说，工厂模式需要额外创建诸多 Factory 类，也会增加代码的复杂性，而且，每个 Factory 类只是做简单的 new 操作，功能非常单薄（只有一行代码），也没必要设计成独立的类，所以，在这个应用场景下，简单工厂模式简单好用，比工厂方法模式更加合适。

### 那什么时候该用工厂方法模式，而非简单工厂模式呢？

我们前面提到，之所以将某个代码块剥离出来，独立为函数或者类，原因是这个代码块的逻辑过于复杂，剥离之后能让代码更加清晰，更加可读、可维护。但是，如果代码块本身并不复杂，就几行代码而已，我们完全没必要将它拆分成单独的函数或者类。

基于这个设计思想，当对象的创建逻辑比较复杂，不只是简单的 new 一下就可以，而是要组合其他类对象，做各种初始化操作的时候，我们推荐使用工厂方法模式，将复杂的创建逻辑拆分到多个工厂类中，让每个工厂类都不至于过于复杂。而使用简单工厂模式，将所有的创建逻辑都放到一个工厂类中，会导致这个工厂类变得很复杂。

除此之外，在某些场景下，如果对象不可复用，那工厂类每次都要返回不同的对象。如果我们使用简单工厂模式来实现，就只能选择第一种包含 if 分支逻辑的实现方式。如果我们还想避免烦人的 if-else 分支逻辑，这个时候，我们就推荐使用工厂方法模式。

### 抽象工厂（Abstract Factory）

讲完了简单工厂、工厂方法，我们再来看抽象工厂模式。抽象工厂模式的应用场景比较特殊，没有前两种常用，所以不是我们本节课学习的重点，你简单了解一下就可以了。

在简单工厂和工厂方法中，类只有一种分类方式。比如，在规则配置解析那个例子中，解析器类只会根据配置文件格式（Json、Xml、Yaml……）来分类。但是，如果类有两种分类方式，比如，我们既可以按照配置文件格式来分类，也可以按照解析的对象（Rule 规则配置还是 System 系统配置）来分类，那就会对应下面这 8 个 parser 类。

针对规则配置的解析器：基于接口IRuleConfigParser

JsonRuleConfigParser

XmlRuleConfigParser

YamlRuleConfigParser

PropertiesRuleConfigParser

针对系统配置的解析器：基于接口ISystemConfigParser

JsonSystemConfigParser

XmlSystemConfigParser

YamlSystemConfigParser

PropertiesSystemConfigParser

针对这种特殊的场景，如果还是继续用工厂方法来实现的话，我们要针对每个 parser 都编写一个工厂类，也就是要编写 8 个工厂类。如果我们未来还需要增加针对业务配置的解析器（比如 IBizConfigParser），那就要再对应地增加 4 个工厂类。而我们知道，过多的类也会让系统难维护。这个问题该怎么解决呢？

抽象工厂就是针对这种非常特殊的场景而诞生的。我们可以让一个工厂负责创建多个不同类型的对象（IRuleConfigParser、ISystemConfigParser 等），而不是只创建一种 parser 对象。这样就可以有效地减少工厂类的个数。具体的代码实现如下所示：

public interface IConfigParserFactory {

IRuleConfigParser createRuleParser();

ISystemConfigParser createSystemParser();

//此处可以扩展新的parser类型，比如IBizConfigParser

}

public class JsonConfigParserFactory implements IConfigParserFactory {

@Override

public IRuleConfigParser createRuleParser() {

return new JsonRuleConfigParser();

}

@Override

public ISystemConfigParser createSystemParser() {

return new JsonSystemConfigParser();

}

}

public class XmlConfigParserFactory implements IConfigParserFactory {

@Override

public IRuleConfigParser createRuleParser() {

return new XmlRuleConfigParser();

}

@Override

public ISystemConfigParser createSystemParser() {

return new XmlSystemConfigParser();

}

}

// 省略YamlConfigParserFactory和PropertiesConfigParserFactory代码

### 重点回顾

好了，今天的内容到此就讲完了。我们来一块总结回顾一下，你需要重点掌握的内容。

在今天讲的三种工厂模式中，简单工厂和工厂方法比较常用，抽象工厂的应用场景比较特殊，所以很少用到，不是我们学习的重点。所以，下面我重点对前两种工厂模式的应用场景进行总结。

当创建逻辑比较复杂，是一个“大工程”的时候，我们就考虑使用工厂模式，封装对象的创建过程，将对象的创建和使用相分离。何为创建逻辑比较复杂呢？我总结了下面两种情况。

* 第一种情况：类似规则配置解析的例子，代码中存在 if-else 分支判断，动态地根据不同的类型创建不同的对象。针对这种情况，我们就考虑使用工厂模式，将这一大坨 if-else 创建对象的代码抽离出来，放到工厂类中。
* 还有一种情况，尽管我们不需要根据不同的类型创建不同的对象，但是，单个对象本身的创建过程比较复杂，比如前面提到的要组合其他类对象，做各种初始化操作。在这种情况下，我们也可以考虑使用工厂模式，将对象的创建过程封装到工厂类中。

对于第一种情况，当每个对象的创建逻辑都比较简单的时候，我推荐使用简单工厂模式，将多个对象的创建逻辑放到一个工厂类中。当每个对象的创建逻辑都比较复杂的时候，为了避免设计一个过于庞大的简单工厂类，我推荐使用工厂方法模式，将创建逻辑拆分得更细，每个对象的创建逻辑独立到各自的工厂类中。同理，对于第二种情况，因为单个对象本身的创建逻辑就比较复杂，所以，我建议使用工厂方法模式。

除了刚刚提到的这几种情况之外，如果创建对象的逻辑并不复杂，那我们就直接通过 new 来创建对象就可以了，不需要使用工厂模式。

现在，我们上升一个思维层面来看工厂模式，它的作用无外乎下面这四个。这也是判断要不要使用工厂模式的最本质的参考标准。

* 封装变化：创建逻辑有可能变化，封装成工厂类之后，创建逻辑的变更对调用者透明。
* 代码复用：创建代码抽离到独立的工厂类之后可以复用。
* 隔离复杂性：封装复杂的创建逻辑，调用者无需了解如何创建对象。
* 控制复杂度：将创建代码抽离出来，让原本的函数或类职责更单一，代码更简洁。

### 课堂讨论

1. 工厂模式是一种非常常用的设计模式，在很多开源项目、工具类中到处可见，比如 Java 中的 Calendar、DateFormat 类。除此之外，你还知道哪些用工厂模式实现类？可以留言说一说它们为什么要设计成工厂模式类？
2. 实际上，简单工厂模式还叫作静态工厂方法模式（Static Factory Method Pattern）。之所以叫静态工厂方法模式，是因为其中创建对象的方法是静态的。那为什么要设置成静态的呢？设置成静态的，在使用的时候，是否会影响到代码的可测试性呢？