

44 | 工厂模式（上）：我为什么说没事不要随使用工厂模式创建对象？

王争 · 设计模式之美



上几节课我们讲了单例模式，今天我们再来讲另外一个比较常用的创建型模式：工厂模式（Factory Design Pattern）。

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？

话不多说，让我们正式开始今天的学习吧！

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。

 复制代码

```
1 public class RuleConfigSource {
2     public RuleConfig load(String ruleConfigFilePath) {
3         String ruleConfigFileExtension = getFileExtension(ruleConfigFilePath);
4         IRuleConfigParser parser = null;
5         if ("json".equalsIgnoreCase(ruleConfigFileExtension)) {
6             parser = new JsonRuleConfigParser();
7         } else if ("xml".equalsIgnoreCase(ruleConfigFileExtension)) {
8             parser = new XmlRuleConfigParser();
9         } else if ("yaml".equalsIgnoreCase(ruleConfigFileExtension)) {
10            parser = new YamlRuleConfigParser();
11        } else if ("properties".equalsIgnoreCase(ruleConfigFileExtension)) {
12            parser = new PropertiesRuleConfigParser();
13        } else {
14            throw new InvalidRuleConfigException(
15                "Rule config file format is not supported: " + ruleConfigFilePath);
16        }
17
18        String configText = "";
19        //从ruleConfigFilePath文件中读取配置文本到configText中
20        RuleConfig ruleConfig = parser.parse(configText);
21        return ruleConfig;
22    }
23
24    private String getFileExtension(String filePath) {
25        //...解析文件名获取扩展名，比如rule.json，返回json
26        return "json";
27    }
28 }
```

在“规范和重构”那一部分中，我们有讲到，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：

 复制代码

```
1  public RuleConfig load(String ruleConfigFilePath) {
2      String ruleConfigFileExtension = getFileExtension(ruleConfigFilePath);
3      IRuleConfigParser parser = createParser(ruleConfigFileExtension);
4      if (parser == null) {
5          throw new InvalidRuleConfigException(
6              "Rule config file format is not supported: " + ruleConfigFilePath);
7      }
8
9      String configText = "";
10     //从ruleConfigFilePath文件中读取配置文本到configText中
11     RuleConfig ruleConfig = parser.parse(configText);
12     return ruleConfig;
13 }
14
15 private String getFileExtension(String filePath) {
16     //...解析文件名获取扩展名，比如rule.json，返回json
17     return "json";
18 }
19
20 private IRuleConfigParser createParser(String configFormat) {
21     IRuleConfigParser parser = null;
22     if ("json".equalsIgnoreCase(configFormat)) {
23         parser = new JsonRuleConfigParser();
24     } else if ("xml".equalsIgnoreCase(configFormat)) {
25         parser = new XmlRuleConfigParser();
26     } else if ("yaml".equalsIgnoreCase(configFormat)) {
27         parser = new YamlRuleConfigParser();
28     } else if ("properties".equalsIgnoreCase(configFormat)) {
29         parser = new PropertiesRuleConfigParser();
30     }
31     return parser;
32 }
33 }
```

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：

```
1 public class RuleConfigSource {
2     public RuleConfig load(String ruleConfigFilePath) {
3         String ruleConfigFileExtension = getFileExtension(ruleConfigFilePath);
4         IRuleConfigParser parser = RuleConfigParserFactory.createParser(ruleConfigFil
5         if (parser == null) {
6             throw new InvalidRuleConfigException(
7                 "Rule config file format is not supported: " + ruleConfigFilePath);
8         }
9
10        String configText = "";
11        //从ruleConfigFilePath文件中读取配置文本到configText中
12        RuleConfig ruleConfig = parser.parse(configText);
13        return ruleConfig;
14    }
15
16    private String getFileExtension(String filePath) {
17        //...解析文件名获取扩展名, 比如rule.json, 返回json
18        return "json";
19    }
20 }
21
22 public class RuleConfigParserFactory {
23     public static IRuleConfigParser createParser(String configFormat) {
24         IRuleConfigParser parser = null;
25         if ("json".equalsIgnoreCase(configFormat)) {
26             parser = new JsonRuleConfigParser();
27         } else if ("xml".equalsIgnoreCase(configFormat)) {
28             parser = new XmlRuleConfigParser();
29         } else if ("yaml".equalsIgnoreCase(configFormat)) {
30             parser = new YamlRuleConfigParser();
31         } else if ("properties".equalsIgnoreCase(configFormat)) {
32             parser = new PropertiesRuleConfigParser();
33         }
34         return parser;
35     }
36 }
```

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。

 复制代码

```
1 public class RuleConfigParserFactory {
2     private static final Map<String, RuleConfigParser> cachedParsers = new HashMap<
3
4     static {
5         cachedParsers.put("json", new JsonRuleConfigParser());
6         cachedParsers.put("xml", new XmlRuleConfigParser());
7         cachedParsers.put("yaml", new YamlRuleConfigParser());
8         cachedParsers.put("properties", new PropertiesRuleConfigParser());
9     }
10
11     public static IRuleConfigParser createParser(String configFormat) {
12         if (configFormat == null || configFormat.isEmpty()) {
13             return null; // 返回null还是IllegalArgumentException全凭你自己说了算
14         }
15         IRuleConfigParser parser = cachedParsers.get(configFormat.toLowerCase());
16         return parser;
17     }
18 }
```

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：

 复制代码

```
1 public interface IRuleConfigParserFactory {
2     IRuleConfigParser createParser();
3 }
4
5 public class JsonRuleConfigParserFactory implements IRuleConfigParserFactory {
6     @Override
7     public IRuleConfigParser createParser() {
8         return new JsonRuleConfigParser();
9     }
10 }
11
12 public class XmlRuleConfigParserFactory implements IRuleConfigParserFactory {
13     @Override
14     public IRuleConfigParser createParser() {
15         return new XmlRuleConfigParser();
16     }
17 }
18
19 public class YamlRuleConfigParserFactory implements IRuleConfigParserFactory {
20     @Override
21     public IRuleConfigParser createParser() {
22         return new YamlRuleConfigParser();
23     }
24 }
25
26 public class PropertiesRuleConfigParserFactory implements IRuleConfigParserFactor
27     @Override
28     public IRuleConfigParser createParser() {
29         return new PropertiesRuleConfigParser();
30     }
```

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：

[复制代码](#)

```
1 public class RuleConfigSource {
2     public RuleConfig load(String ruleConfigFilePath) {
3         String ruleConfigFileExtension = getFileExtension(ruleConfigFilePath);
4
5         IRuleConfigParserFactory parserFactory = null;
6         if ("json".equalsIgnoreCase(ruleConfigFileExtension)) {
7             parserFactory = new JsonRuleConfigParserFactory();
8         } else if ("xml".equalsIgnoreCase(ruleConfigFileExtension)) {
9             parserFactory = new XmlRuleConfigParserFactory();
10        } else if ("yaml".equalsIgnoreCase(ruleConfigFileExtension)) {
11            parserFactory = new YamlRuleConfigParserFactory();
12        } else if ("properties".equalsIgnoreCase(ruleConfigFileExtension)) {
13            parserFactory = new PropertiesRuleConfigParserFactory();
14        } else {
15            throw new InvalidRuleConfigException("Rule config file format is not support");
16        }
17        IRuleConfigParser parser = parserFactory.createParser();
18
19        String configText = "";
20        //从ruleConfigFilePath文件中读取配置文本到configText中
21        RuleConfig ruleConfig = parser.parse(configText);
22        return ruleConfig;
23    }
24
25    private String getFileExtension(String filePath) {
26        //...解析文件名获取扩展名，比如rule.json，返回json
27        return "json";
28    }
29 }
```

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。

 复制代码

```
1 public class RuleConfigSource {
2     public RuleConfig load(String ruleConfigFilePath) {
3         String ruleConfigFileExtension = getFileExtension(ruleConfigFilePath);
4
5         IRuleConfigParserFactory parserFactory = RuleConfigParserFactoryMap.getParser
6         if (parserFactory == null) {
7             throw new InvalidRuleConfigException("Rule config file format is not suppor
8         }
9         IRuleConfigParser parser = parserFactory.createParser();
10
11         String configText = "";
12         //从ruleConfigFilePath文件中读取配置文本到configText中
13         RuleConfig ruleConfig = parser.parse(configText);
14         return ruleConfig;
15     }
16
17     private String getFileExtension(String filePath) {
18         //...解析文件名获取扩展名，比如rule.json，返回json
19         return "json";
20     }
21 }
22
23 //因为工厂类只包含方法，不包含成员变量，完全可以复用，
24 //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。
25 public class RuleConfigParserFactoryMap { //工厂的工厂
26     private static final Map<String, IRuleConfigParserFactory> cachedFactories = ne
27
28     static {
29         cachedFactories.put("json", new JsonRuleConfigParserFactory());
30         cachedFactories.put("xml", new XmlRuleConfigParserFactory());
31         cachedFactories.put("yaml", new YamlRuleConfigParserFactory());
32         cachedFactories.put("properties", new PropertiesRuleConfigParserFactory());
33     }
34 }
```



```
35     public static IRuleConfigParserFactory getParserFactory(String type) {
36         if (type == null || type.isEmpty()) {
37             return null;
38         }
39         IRuleConfigParserFactory parserFactory = cachedFactories.get(type.toLowerCase());
40         return parserFactory;
41     }
42 }
```

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。

 复制代码

```
1  针对规则配置的解析器：基于接口IRuleConfigParser
2  JsonRuleConfigParser
3  XmlRuleConfigParser
4  YamlRuleConfigParser
5  PropertiesRuleConfigParser
6
7  针对系统配置的解析器：基于接口ISystemConfigParser
8  JsonSystemConfigParser
9  XmlSystemConfigParser
10 YamlSystemConfigParser
11 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：

 复制代码

```
1  public interface IConfigParserFactory {
2      IRuleConfigParser createRuleParser();
3      ISystemConfigParser createSystemParser();
4      //此处可以扩展新的parser类型，比如IBizConfigParser
5  }
6
```

```
7 public class JsonConfigParserFactory implements IConfigParserFactory {
8     @Override
9     public IRuleConfigParser createRuleParser() {
10         return new JsonRuleConfigParser();
11     }
12
13     @Override
14     public ISystemConfigParser createSystemParser() {
15         return new JsonSystemConfigParser();
16     }
17 }
18
19 public class XmlConfigParserFactory implements IConfigParserFactory {
20     @Override
21     public IRuleConfigParser createRuleParser() {
22         return new XmlRuleConfigParser();
23     }
24
25     @Override
26     public ISystemConfigParser createSystemParser() {
27         return new XmlSystemConfigParser();
28     }
29 }
30
31 // 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？

欢迎在留言区写下你的答案，和同学一起交流和分享。如果有收获，也欢迎你把这篇文章分享给你的朋友。

AI智能总结

工厂模式是一种常用的创建型设计模式，包括简单工厂、工厂方法和抽象工厂。本文重点介绍了简单工厂和工厂方法。简单工厂通过创建一个工厂类来根据条件创建不同的对象，提高了代码的可读性和灵活性。工厂方法则将对象的创建延迟到子类中，更符合开闭原则。文章还介绍了简单工厂模式的两种实现方法，以及对于开闭原则和多态的讨论。工厂方法模式比起简单工厂模式更加符合开闭原则。但在某些场景下，简单工厂模式更加合适。抽象工厂模式则适用于类有多种分类方式的特殊场景。通过让一个工厂负责创建多个不同类型的对象，可以有效地减少工厂类的个数。文章通过具体的代码实现和对比分析，帮助读者理解工厂模式的应用场景和选择原则。在实际应用中，工厂模式的作用主要体现在封装变化、代码复用、隔离复杂性和控制复杂度上。读者需要重点掌握简单工厂和工厂方法的应用场景，以及在何种情况下选择哪种模式。文章还提出了两种情况下的建议使用方式，并举例说明了工厂模式的实际应用。同时，读者还可以参与课堂讨论，深入交流和分享工厂模式的实际应用和设计原则。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

全部留言 (117)

最新 精选



zhengyu.nie

2020-04-24

个人意见，传统的工厂模式太麻烦了，除非业务真的很复杂，通常我会选择以下方案。
还是举文中的例子

1.将不同的RuleConfigParser实现按照约定格式指定beanName注入，比方说@Component (“XmlRuleConfigParser”)，取的时候applicationContext.getBean(typeSuffix+RuleConfigParser)即可，拓展的话，自己写一个xxRuleConfigParser，就注入进去了，也不需要再map容器新增。

整个工厂方法就是

```
public RuleConfigParser getInstance(suffix){  
    return InstanceLocator.getBean(suffix+"RuleConfigParser");  
}
```

2.直接用java.util.functional实现现代函数式编程范式的设计模式
像文中的例子,可以看作工厂,也可以看作获取一种parse策略。

可以有一个FunctionFactory内部维护一组Function<String,String>函数，再有一个Map容器mapping type和Function的关系。这样是简化了类的数量，如果业务简单没必要整太多类，function铺在一个factory里可读性不会有什么问题。如果是没有返回值的操作，也可以用Consumer函数。打个比方

```
public BiConsumer<AbstractProductServiceRequest, Function<ProductServiceQuery
Request,
    ProductServiceQueryResponse>> operateConsumer() {
    switch (serviceOperationEnum) {
        case OPEN:
            return openConsumer();
        case CLOSE:
            return closeConsumer();
        default:
            throw new RuntimeException("not support OperationType");
    }
}
```

如果是对象，那更简单，Map<Supply>函数即可。

```
public class ShapeFactory {
    final static Map<String, Supplier<Shape>> map = new HashMap<>();
    static {
        map.put("CIRCLE", Circle::new);
        map.put("RECTANGLE", Rectangle::new);
    }
    public Shape getShape(String shapeType){
        Supplier<Shape> shape = map.get(shapeType.toUpperCase());
        if(shape != null) {
            return shape.get();
        }
        throw new IllegalArgumentException("No such shape " + shapeType.toUpperCase());
    }
}
```

以上个人意见，对于比较简单的场景，lambda function等方式代替类，会显得不那么臃肿，

具体还是要看需求。至于OOP等原则，也不是完全要遵守的，就像争哥说的少量if可以不管，一样的道理，灵活运用。

作者回复: 🍊

共 35 条评论 >

👍 195



Robin

2020-07-25

原文：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。

原文：工厂方法：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。

感觉说法有点牵强，添加一个类，简单工厂模式修改RuleConfigParserFactory，工厂方法也要修改RuleConfigParserFactoryMap，也是会违背开闭原则。关键简单工厂模式(第二种方式)下添加的代码量一个是map.put,工厂方法也是一个map.put,然后说明工厂方法代码的改动非常少，基本上符合开闭原则？

作者回复: 改动是不多呀💎💎💎💎💎 您有更好的设计思路建议吗？

共 9 条评论 >

👍 11



郑大钱

2020-11-17

传统的工厂模式确实很传统。

简单工厂是在一个工厂方法里通过流程控制语句创建不同的对象，适合创建简单的对象。

工厂方法和简单方法没有什么区别，只是用工厂对象再此封装了复杂对象的创建。工厂的工厂负责调用工厂的创建方法，每个工厂只创建一个对象，适合创建复杂的对象。

工厂模式是对创建方法的封装和抽象，创建的复杂度无法被抵消，只能被转移到工厂内部消化。

作者回复: 💎💎💎💎💎



👍 5



御风

2020-08-08

掌握了使用工厂模式的本质：封装变化（创建逻辑可能变化）、隔离复杂性、控制复杂度（让类职责更加单一）、代码复用。

如果创建的对象不能复用，又不想用if-else，就不能使用简单工厂模式。

这个可以在static代码块中使用反射？

作者回复：反射可以，但不能在static静态代码块中创建对象吧



逍遥思

2020-02-12

复杂度无法被消除，只能被转移：

- 不用工厂模式，if-else 逻辑、创建逻辑和业务代码耦合在一起
- 简单工厂是将不同创建逻辑放到一个工厂类中，if-else 逻辑在这个工厂类中
- 工厂方法是将不同创建逻辑放到不同工厂类中，先用一个工厂类的工厂来得到某个工厂，再用这个工厂来创建，if-else 逻辑在工厂类的工厂中

共 20 条评论 >

414



跳跳

2020-08-10

我觉得很多人被带跑偏了 工厂本身的重点不是解决if else 而是解决简单工厂的开闭原则，大家都在重点讨论if else 即使被省略了 也是map的功劳啊

共 3 条评论 >

61



麦可

2020-02-12

我把Head First的定义贴过来，方便大家理解总结

工厂方法模式：定义了一个创建对象的接口，但由子类决定要实例化的类是哪一个。工厂方法让类把实例化推迟到子类

抽象工厂模式：提供一个接口，用于创建相关或依赖对象的家族，而不需要明确指定具体类

共 4 条评论 >

56



辣么大

2020-02-12

在JDK中工厂方法的命名有些规范：

1. `valueOf()` 返回与入参相等的对象

例如 `Integer.valueOf()`

2. `getInstance()` 返回单例对象

例如 `Calendar.getInstance()`

3. `newInstance()` 每次调用时返回新的对象

例如 `HelloWorld.class.getConstructor().newInstance()`

4 在反射中的工厂方法

例如 `XXX.class.getField(String name)` 返回成员

静态工厂方法的优点：

1. 静态工厂方法子类可以继承，但不能重写，这样返回类型就是确定的。可以返回对象类型或者primitive 类型。

2. 静态工厂方法的名字更有意义，例如`Collections.synchronizedMap()`

3. 静态工厂方法可以封装创建对象的逻辑，还可以做其他事情，让构造方法只初始化成员变量。

4. 静态工厂方法可以控制创建实例的个数。例如单例模式，或者多例模式，使用本质上是可以静态工厂方法实现。

共 6 条评论 >

👍 44



Jxin

2020-02-13

分歧：

1.文中说，创建对象不复杂的情况下用new，复杂的情况用工厂方法。这描述没问题，但工厂方法除了处理复杂对象创建这一职责，还有增加扩展点这优点。工厂方法，在可能有扩展需求，比如要加对象池，缓存，或其他业务需求时，可以提供扩展的地方。所以，除非明确确定该类只会有简单数据载体的职责（值对象），不然建议还是用工厂方法好点。new这种操作是没有扩展性的。

回答问题：

2.工厂方法要么归于类，要么归于实例。如果归于实例，那么第一个实例怎么来？而且实例创建出另一个实例，这种行为应该称为拷贝，或则拆分。是一个平级的复制或分裂的行为。而归于类，创建出实例，是一个父子关系，其创建的语义更强些。

我认为不影响测试。因为工厂方法不该包含业务，它只是new的一种更好的写法。所以你只需要用它，而并不该需要测它。如果你的静态工厂方法都需要测试，那么说明你这个方法不够“干净”。

**Brian**

2020-02-13

一、三种工厂模式

1. 简单工厂 (Simple Factory)

使用场景：

a. 当每个对象的创建逻辑都比较简单的时候，将多个对象的创建逻辑放到一个工厂类中。

实现：

a. if else 创建不同的对象。

b. 用单例模式 + 简单工厂模式结合来实现。

2. 工厂方法 (Factory Method)

使用场景：

a. 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时，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。

b. 避免很多 if-else 分支逻辑时。

实现：

a. 定义相应的ParserFactory接口，每个工厂定义一个实现类。这种方式使用会有多个if else 让使用更加复杂。

b. 创建工厂的工厂来，此方案可以解决上面的问题。

3. 抽象工厂 (Abstract Factory) – 不常用

使用场景：

a. 有多种分类方式，如方式要用一套工厂方法，方式二要用一套工厂方法，详见原文例子。

实现：

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。

二、例子

刚好最近有这方面的应用场景，主要使用了 单例模式 + 工厂模式 + 策略模式，用于解化多过的if else的复杂性。

```
public class OrderOperateStrategyFactory {  
    /**  
     * 消费类型和策略对象映射。  
     */  
    private Map<CheckoutType, OrderOperateStrategy> map;
```

```

/**
 * 构造策略列表。
 */
private OrderOperateStrategyFactory() {
    List<OrderOperateStrategy> list = new ArrayList<>();
    list.add(SpringContextHolder.getBean(ConsumptionOrderOperateStrategy.class));
    list.add(SpringContextHolder.getBean(GroupServiceOrderOperateStrategy.class));
    //...
    map = list.stream().collect(Collectors.toMap(OrderOperateStrategy::getCheckoutT
ype, v -> v));
}

/**
 * 通过消费类型获取订单操作策略。
 *
 * @param checkoutType 消费类型
 * @return 订单操作策略对象
 */
public OrderOperateStrategy get(CheckoutType checkoutType) {
    return map.get(checkoutType);
}

/**
 * 静态内部类单例对象。
 */
private static class Holder {
    private static OrderOperateStrategyFactory INSTANCE = new OrderOperateStrate
gyFactory();
}

/**
 * 获取订单操作策略工厂类实例。
 *
 * @return 单例实例。
 */
public static OrderOperateStrategyFactory getInstance() {
    return Holder.INSTANCE;
}

```

```
}  
}
```

使用：

```
OrderOperateStrategy strategy = OrderOperateStrategyFactory.getInstance().get(check  
outType);  
strategy.complete(orderId);
```

共 3 条评论 >

 27