

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？

2020-02-12 王争

设计模式之美

[进入课程 >](#)



工厂模式（上）

讲述：冯永吉

时长 12:57 大小 10.38M



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，☆
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: " + ruleConfigFilePatl
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：

 复制代码

```
1 public class RuleConfigSource {
2     public RuleConfig load(String ruleConfigFilePath) {
3         String ruleConfigFileExtension = getFileExtension(ruleConfigFilePath);
4         IRuleConfigParser parser = RuleConfigParserFactory.createParser(ruleConfig
5         if (parser == null) {
6             throw new InvalidRuleConfigException(
7                 "Rule config file format is not supported: " + ruleConfigFilePatl
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 HashM;
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 IRuleConfigParserFactory {
27     @Override
28     public IRuleConfigParser createParser() {
29         return new PropertiesRuleConfigParser();
30     }
31 }
```

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 supported");
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.getPar:
6         if (parserFactory == null) {
7             throw new InvalidRuleConfigException("Rule config file format is not sup
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 =
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.toLowerC:
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。

```
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：

```
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。

课程学习计划

关注极客时间服务号 每日学习签到

月领 25+ 极客币

【点击】保存图片，打开【微信】扫码>>>



上一篇 43 | 单例模式（下）：如何设计实现一个集群环境下的分布式单例模式？

下一篇 45 | 工厂模式（下）：如何设计实现一个Dependency Injection框架？

精选留言 (29)

写留言



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...

展开 ∨



19



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

...

展开 ∨



11



失火的夏天

2020-02-12

对象每次都要重用，也可以用map缓存，不过value要改成全类名，通过反射来创建对象，这样每次都是一个新的类了，除非那个类被设计成禁止反射调用。

展开 ∨



4



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。...

展开 ▾



3



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。n...

展开 ▾



3



勤劳的明酱

2020-02-12

那Spring的BeanFactory实际上使用的是简单工厂模式 + 单例模式对吧，如果是工厂模式那就是使用ObjectFactory和FactoryBean来实现。第三方的复杂bean的初始化使用工厂模式，对于普通的bean统一处理，虽然复杂但没必要使用工厂。

展开 ▾



3



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()...

展开 ▾



2



唐龙

2020-02-12

试着把代码翻译成了C++语言，应该算是搞懂了(以前只会单例)。目前没写过特别复杂的项目，简单工厂对我个人来说够用了。

展开 ▾



2



李小四

2020-02-23

设计模式_44:

作业

1. Android开发中工厂模式也很常用，比如`BitmapFactory`类；用工厂模式的原因是`Bitmap`对象的创建过程比较复杂，并且可以通过不同的方式来创建。

...

展开 ▾



1



小晏子

2020-02-12

java.text.NumberFormat是使用工厂模式实现的，它可以根据特定的区域设置格式化数字，这个类设置成工厂模式是因为全世界有很多不同的区域，有很多不同的数字表示法，所有从开闭原则角度，用工厂模式实现可以方便的增加对不同区域数字转换的支持。

使用静态方法创建对象首先可以使得创建对象的方法名字更有意义，使用者看到方法名就知道什么意思了，提高了代码的可读性。其次使用静态方法创建对象可以重复使用事先...

展开 ▾



1



高源

2020-02-12

老师最好提供你讲课例子代码完整的版本，结合你讲的内容消化理解🤔



2



DullBird

2020-02-24

1. Executors利用静态方法做工厂方法，dubbo的spi也是工厂的思路
2. 首先包含简单工厂方法的对象，没有必要有构造函数。因为他只是创建对象的一个类。不用实例化，所以如果方法不是静态的，就没法调用了。
3. 我认为不影响可测试性，因为静态方法仅仅返回了一个对象，而需要测试的应该是这个对象的被你调用的方法。并不是拿到这个对象的过程。

展开 ▾



乾坤瞬间

2020-02-21

课后习题1，在spark livy框架中，有一个ClientFactory类，这个类根据用户的开发环境会设置成不同的客户端，一种是用来生产rpcClient客户端，一种是用来生产httpClient，每一种创建的逻辑和方式都非常复杂，会根据不同的参数生成Client,有些客户端会内置看门

狗，以提高可用性，有些没有.所以应对这种创建的复杂性，使用了工厂模式，使用了工厂的工厂...

展开 ▾



whistleman

2020-02-21

学习了~

展开 ▾



守拙

2020-02-20

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？ ...

展开 ▾



Liam

2020-02-20

会影响可测试性，因为它无法被mock（通常是mock实例的实例方法），导致其他依赖工厂类的类难以测试



majaja

2020-02-20

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。

此處工厂方法是錯字嗎？

展开 ▾



岁月

2020-02-20

课堂讨论

2. 简单工厂创建对象的方法之所以用静态的，那是因为他的职责就是创建对象，本身不需要包含任何和对象相关的任何成员变量，换句话说，就是任何人使用这个简单工厂的时候，工厂都是无状态的，所以直接用静态方法即可。

展开 ▾



传说中的成大大

2020-02-17

第一问 很少遇到

第二问 我觉得应该要考虑创建出来的对象是否存在一些成员变量吧 如果有数据的话就会有影响或者没啥影响吧

展开 ▾



桂城老托尼

2020-02-15

很喜欢最后的这几个总结。 感谢分享！

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。

展开 ▾

