



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。

```

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";
    }
}

```

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：

```

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？

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。

### 精选留言



逍遥思

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

2020-02-12 17:46



麦可

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

2020-02-12 09:53



失火的夏天



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。

2020-02-12 02:32



Brian

## 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::getCheckoutType, 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 OrderOperateStrategyFactory();
}

/**
 * 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(checkoutType);
strategy.complete(orderId);

```

2020-02-13 20:53



Jxin

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”。

2020-02-13 00:21



勤劳的明酱

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。

2020-02-12 14:03



唐龙

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。

2020-02-12 15:08



辣么大

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。

2020-02-12 21:03



小晏子

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创建好的对象,最后使用静态工厂方法可以返回原返回类型的任何子类对象,更灵活。使用Calendar.getInstance, NumberFormat这种工具类的静态工厂方法一般是不会影响到可测试性的,另外使用文中Parser的例子静态方法也不会有影响,暂时没想到会有影响的例子。

2020-02-12 11:36



李小四

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。

2. 查了一下资料,意识到这个问题的核心在于使用\*静态工厂方法\*替代的是使用构造函数,之所以用\*静态方法\*,是因为它比构造函数具有以下优势:

- (1) 构造函数的名字无意义,方法的名字包含更多有用信息
- (2) 构造函数只能返回当前Class类型对象,而方法可以返回当前类型对象、当前类型的子类对象,也可以返回基础数据类型
- (3) 如果创建过程很复杂,那么方法可以把很多不应该由构造函数处理的过程放在方法中,让构造函数只处理初始化成员的工作,职责更单一。
- (4) 方法可以控制生成对象的个数(单例,多例等)

# 感想

看了今天的内容,突然有个疑问:

\*static\*方法可以是抽象方法吗?可以被继承吗?

验证了一下,发现\*static\*方法可以被重写,\*static\*与\*abstract\*是冲突的,不能同时修饰一个方法;而且,如果用子类重写了父类的static方法,这时候让父类的引用指向子类对象,然后调用该\*static\*方法,这时调用的是父类的\*static\*方法,也就是不支持“多态”,这也解释了为什么\*static\*与\*abstract\*冲突。

关于第二题,直觉上来讲,如果不用静态方法就只能对对象方法,但使用对象方法的前提是有一个对象,但这个方法就是用来创建对象的,这时一个死锁。。。但显然问题的用意不是这个,于是查了资料。。。

2020-02-23 11:01



乾坤瞬间

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

习题2,个人认为这样的静态方法,第一与单例模式的思想不可分离,因为创建对象的抽象不需要通过创建一个新的类来实现,或者根据dry选择,用静态方法复用代码块的方式更加直接粗糙,简单美。我觉得在可测试方面是有影响的,不过因为这种简单的抽象是基于原有逻辑不存在未决行为的基础上的,而且对新增的代码有足够的信心

同时总结一下今天的三种工厂方法的演进

利用数学公式 $y=f(x,x_2)$ 的角度,  $y$ 是关于 $x, x_2$ 的一个系统描述。

简单工厂只基于在系统 $y$ 在不断加上 $x_3$ 的情况下, 直接引入一个新的变量来简单替换 $f$ 函数

工厂函数是在替换变量的基础上对 $x$ 进行了再替换, 使得系统更容易理解,  $y=f(\theta(x), \theta(x_2) \dots)$ 形式

抽象方法是把 $x$ 变量替换为 $\delta(x, m)$ 即,  $y=f(\delta(x, m), \delta(x_2, m))$ 形式

2020-02-21 22:01



whistleman

学习了~

2020-02-21 07:41



守拙

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?

在android开发网络请求类库Retrofit中, 可以通过添加Converter将json格式的数据转化为实体类. 例如,GsonConverterFactory通过继承自Converter.Factory的方式来实现将json通过Gson转化为Bean.

这里ConverterFactory设计为工厂模式主要是为了衍生类自己实现Factory接口的方法,并且可以为Retrofit实例添加多个ConverterFactory.

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, 在使用的時候, 是否会影响到代码的可测试性呢?

若创建对象的方法不设置为静态的, 需要new工厂对象, 才能使用工厂对象创建产品对象. 还不如直接new产品对象. 考虑到简单工厂的业务比较简单, 对测试性的影响较小.

2020-02-20 23:06



Liam

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

2020-02-20 16:09



majaja

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。

此處工厂方法厂是錯字嗎?

2020-02-20 12:58



岁月

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。

2020-02-20 11:16



传说中的成大大

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

2020-02-17 11:42



桂城老托尼

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。

2020-02-15 17:11



相逢是缘

打卡

三种工厂模式

一、简单工厂模式

使用场景：

1) 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。同时，对象的创建也不复杂。

实现方式：

1) 使用if else逻辑进行创建

2) 当对象可以复用的时候，可以使用单例模式+工厂模式组合

二、工厂方法模式

使用场景：

1) 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。

2) 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。又想去除过多的if-else分支

实现方式：

1) 定义相应的Factory接口，为每个工厂定义一个实现类。

2) 之后定义工厂的工厂，把这些Factory实现类放到一个简单工厂类中

三、抽象工厂（Abstract Factory） - 不常用

判断要不要使用工厂模式的标准

一、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。

二、代码复用：创建代码抽离到独立的工厂类中可以复用

三、隔离复杂性：可以封装复杂的创建逻辑

四、控制复杂度：将创建代码抽离出来，让原本的函数或是类职责更单一，代码更简洁

2020-02-15 15:50



L

课堂讨论：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？

答：设置成静态的是方便调用者调用，同时，静态方法在编译时就绑定了，它是全局的，有利于资源利用，节省开销

2020-02-15 10:59