# 第 17 章　专题：多例（Multiton）模式与多语言支持

作为对象的创建模式，多例模式中的多例类可有多个实例；而且多例类必须自己创建、管理自己的实例，并向外界提供自己的实例。

请读者在阅读本章之前，先阅读本书的“单例模式”一章。

## 17.1　引　　言

### 一个真实的项目

这是一个真实的、面向全球消费者的华尔街金融网站项目的一部分。按照项目计划书，这个网站系统是要由数据库驱动的，并且要支持十九种不同的语言；而且在将来支持更多的语言。消费者在登录到系统上时可以选择自己所需要的语言，系统则根据用户的选择将网站的静态文字和动态文字全部转换为用户所选择的语言。

经过讨论，设计师们同意对静态文字和动态文字采取不同的解决方案：

（1）把所有的网页交给翻译公司对上面的静态文字进行翻译，

（2）而网页上面的动态内容则需要程序解决。

在进行了研究后，设计师们发现，他们需要解决的动态文字的“翻译”问题，实际上是将数据库中的一些静态或者半静态的数据进行“翻译”。就是一个典型的数据表，如下表所示。

| 货币代码 | 货币名称 | 货币尾数 |
|---|---|---|
| USD | America (United States of America), Dollars | 2 |
| CNY | China, Yuan Renminbi | 2 |
| EUR | France, Euro | 2 |
| JPY | Japan, Yen | 0 |

货币代码永远是上面所看到的英文代码，但是货币名称应当根据用户所选择的语言不同而不同。比如对中文读者就应当翻译成为下面的表，如下表所示。

| 货币代码 | 货币名称 | 货币尾数 |
|---|---|---|
| USD | 美国（美利坚合众国），美元 | 2 |

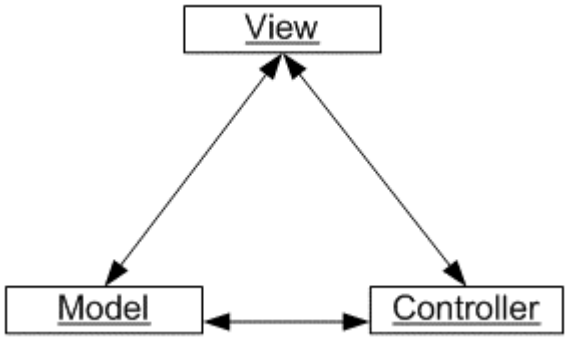
| | | |
|---|---|---|
| CNY | 中国，人民币元 | 2 |
| EUR | 法国，欧元 | 2 |
| JPY | 日本，日元 | 0 |

这样的表会在网页上作为下拉菜单出现，用户看到的是货币名称，而系统内部使用的是货币代码。

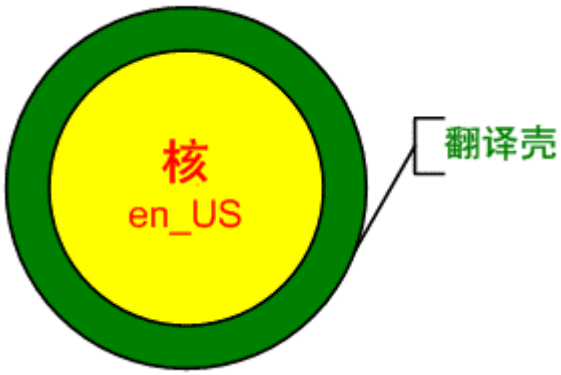
## 国际化解决方案

这样的问题就是国际化的问题，所谓国际化就是 Internationalization，简称 i18n（请参见本章最后的问答题）。

设计师所采取的实际方案是分层方案，也就是 MVC 模式。MVC 模式将系统分为三个层次，即模型（Model）、视图（View）、控制器（Control）。国际化是视图部分的问题，因此，应当在视图部分得到解决。MVC 模式的示意图如下所示。




换言之，系统的内核可以是纯英文的。在内核外部增加一个壳层负责语言翻译工作。请见下面的英文内核和翻译壳层的概念图。




所谓内核就是系统的模型，而翻译壳层便是视图的一部分。对多语言的支持属于视图功能，因此，不应当在内核解决，而应当在视图解决。这就是设计师们达成的总体方案。

# 17.2　多 态 模 式

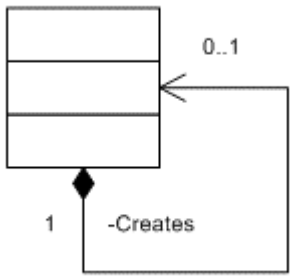
## 多例模式的特点

所谓的多例模式（Multiton Pattern），实际上就是单例模式的自然推广。作为对象的创建模式，多例模式或多例类有以下的特点：

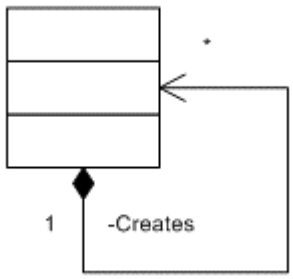
（1）多例类可有多个实例。

（2）多例类必须自己创建、管理自己的实例，并向外界提供自己的实例。

单例类一般情况下最多只可以有一个实例，请见下面的单例类结构图。




但是单例模式的精神是允许有限个实例，并不是仅允许一个实例。这种最多允许有限多个实例，并向整个 JVM 提供自己实例的类叫做多例类（Multiton），这种模式叫做多例模式（Multiton Pattern），请参见下面的多例类结构图。



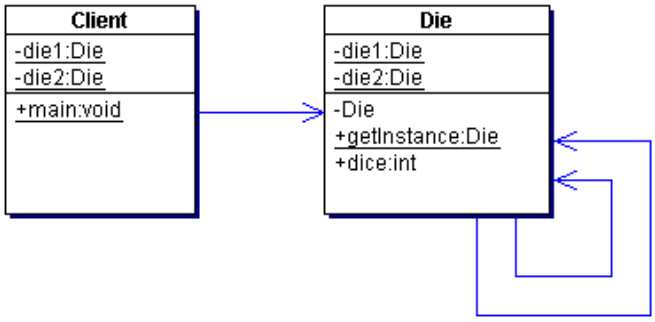

本章就需要用多例模式来实现资源对象，需要构造出能提供有限个实例、每个实例有各不相同的属性（即 Locale 代码）的代码。

## 有上限多例类

一个实例数目有上限的多例类已经把实例的上限当做逻辑的一部分，并建造到了多例

类的内部，这种多例模式叫做有上限多例模式。

比如每一麻将牌局都需要两个色子，因此色子就应当是双态类。这里就以这个系统为例，说明多例模式的结构。色子的类图如下所示。




下面就是多例类 Die（色子）的源代码。

代码清单 3：多例类的源代码

```
package com.javapatterns.multilingual.dice;
import java.util.Random;
import java.util.Date;
public class Die
{
    private static Die die1 = new Die();
    private static Die die2 = new Die();
   /**
    *      私有的构造子保证外界无法
    *      直接将此类实例化
    */
    private Die()
    {
    }
   /**
    *      工厂方法
    */
    public static Die getInstance(int whichOne)
    {
        if (whichOne == 1)
        {
            return die1;
        }
        else
        {
            return die2;
        }
    }
```

```
    /**
     *      掷色子，返还一个在 1～6 之间的
     *      随机数
     */
     public synchronized int dice()
     {
          Date d = new Date();
          Random r = new Random( d.getTime() );
          int value = r.nextInt();
          value = Math.abs(value);
          value = value % 6;
          value += 1;
          return value;
     }
}
```

在多例类 Die 中，使用了饿汉方式创建了两个 Die 的实例。根据静态工厂方法的参数，工厂方法返还两个事例中的一个。Die 对象的 die()方法代表掷色子，这个方法会返还一个在 1～6 之间的随机数，相当于色子的点数。

代码清单 4：客户端的源代码

```
package com.javapatterns.multilingual.dice;
public class Client
{
     private static Die die1, die2;
     public static void main(String[] args)
     {
          die1 = Die.getInstance(1);
          die2 = Die.getInstance(2);
          die1.dice();
          die2.dice();
     }
}
```

由于有上限的多例类对实例的数目有上限，因此有上限的多例类在这个上限等于 1 时，多例类就回到了单例类。因此，多例类是单例类的推广，而单例类是多例类的特殊情况。

一个有上限的多例类可以使用静态变量存储所有的实例，特别是在实例数目不多的时候，可以使用一个个的静态变量存储一个个的实例。在数目较多的时候，就需要使用静态聚集储存这些事例。

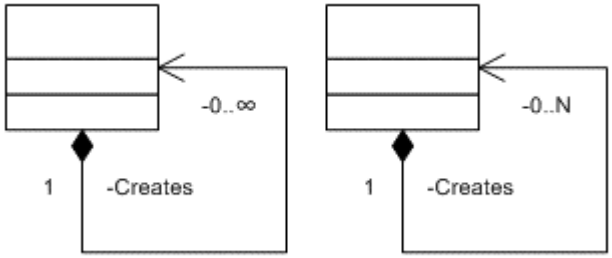
## 无上限多例模式

多例类的实例数目并不需要有上限[CAMP02]，实例数目没有上限的多例模式就叫做无上限多例模式。

由于没有上限的多例类对实例的数目是没有限制的，因此，虽然这种多例模式是单例

模式的推广，但是这种多例类并不一定能够回到单例类。

由于事先不知道要创建多少个实例，因此，必然是使用聚集管理所有的实例。本章要讨论的多语言支持方案就需要应用到多例模式，关于没有上限的多例模式的实现可以参见下面的讨论。没有上限的多例模式（左）和有上限的多例模式（右）的类图如下所示。




其中 N 就是实例数目的上限。

## 有状态的和没有状态的多例类

如同单例类可以分成有状态的和没有状态的两种一样，多例类也可以分成有状态的和没有状态的两种。

多例对象的状态如果是可以在加载后改变的，那么这种多例对象叫做可变多例对象（Mutable Singleton）；如果多例对象的状态在加载后就不可以改变，那么这种多例对象叫做不变多例对象（Immutable Singleton）。显然不变多例类的情形较为简单，而可变单例类的情形较为复杂。

如果一个系统是建立在诸如 EJB 和 RMI 等分散技术之上的，那么多例类有可能会出现数个实例，因此，在这种情况下除非提供有效的协调机制，不然最好不要使用有状态的和可变的单例类，以避免出现状态不自恰的情况。读者可以参考本书的“单例（Singleton）模式”一章中的相关讨论。

# 17.3　多语言项目的设计

由于熟悉了多例模式，系统的设计实际上并不复杂。

## 语言代码

下表就是几个常见的语言代码，如表所示。

| 语 言 代 码 | 说　明 |
| --- | --- |
| de | German |

| | |
|---|---|
| en | English |
| fr | French |
| ja | Japanese |
| jw | Javanese |
| ko | Korean |
| zh | Chinese |

## 地区代码

下表就是几个常见的地区代码，如表所示。

| 地 区 代 码 | 说　明 |
|---|---|
| CN | China |
| DE | Germany |
| FR | France |
| IN | India |
| US | United States |

## Locale 代码

一个 Locale 代码由语言代码和地区代码组合而成，如下表所示。

| 语 言 代 码 | 地 区 代 码 | Locale 代码 | 说　明 |
|---|---|---|---|
| en | US | en_US | 美国英语 |
| en | GB | en_GB | 英国英语 |
| fr | FR | fr_FR | 法国法语 |
| fr | CA | fr_CA | 加拿大法语 |
| de | DE | de_DE | 德国德语 |
| zh | CH | zh_CH | 简体汉语 |

## Resource 文件及其命名规范

一个 Resource 文件是一个简单的文本文件。一个 Resource 文件的名字是由一个短文件名和文件的扩展名 properties 组成的，而 Resource 文件的短文件名则是 Java 程序在调用此文件时使用的文件名。

一个 Resource 文件和一个普通的 properties 文件并无本质区别，但 Java 语言对两者的支持是有区别的。java.util.Properties 类不支持多语言，而 java.util.ResourceBundle 类则支持多语言。

当 Locale 代码是 en_US 时，Resource 文件的文件名应当是短文件名加上 Locale 代码，就是 en_US。当 Locale 代码是 zh_CH 时，Resource 文件的文件名应当是短文件名加上 Locale

代码，就是 zh_CH。

## 怎样使用 Locale 对象和 ResourceBundle 对象

那么怎样使用 ResourceBundle 读取一个 Resource 文件呢？下面就是一个例子。

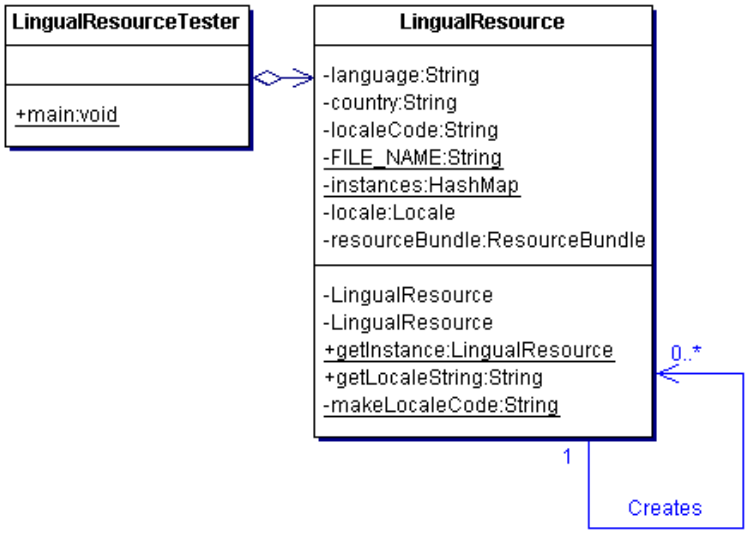
代码清单 4：怎样使用 Locale 对象和 ResourceBundle 对象

```
Locale locale = new Locale("fr","FR");
ResourceBundle res =
    ResourceBundle.getBundle("shortname",locale);
```

在上面的例子里面，res 对象会加载一个名为 shortname_fr_FR.properties 的 Resource 文件。

## 系统的设计

下图所示给出系统的结构图。其中 LingualResourceTester 是一个示意性的客户端类，而 LingualResource 是一个多例类。




下面就是这个多例类的源代码。

代码清单 5：多例类 LingualResource 的源代码

```
package com.javapatterns.multilingual;
import java.util.HashMap;
import java.util.Locale;
import java.util.ResourceBundle;
public class LingualResource
{
```

```
  private String language = "en";
  private String region = "US";
  private String localeCode = "en_US";
  private static final String FILE_NAME = "res";
  private static HashMap instances =
        new HashMap(19);
  private Locale locale = null;
  private ResourceBundle resourceBundle = null;
  private LingualResource lnkLingualResource;
/**
 *      私有的构造子保证外界无法直接将此类实例化
 */
  private LingualResource(
        String language, String region)
  {
        this.localeCode = language;
        this.region = region;
        localeCode =
              makeLocaleCode(language , region);
        locale = new Locale(language, region);
        resourceBundle =
              ResourceBundle.getBundle(FILE_NAME, locale);
        instances.put( makeLocaleCode(language, region) ,
              resourceBundle);
  }
/**
 *      私有的构造子保证外界无法直接将此类实例化
 */
  private LingualResource()
  {
        //do nothing
  }
/**
 *      工厂方法，返还一个具有指定的内部状态的实例
 */
  public synchronized static LingualResource
        getInstance(String language, String region)
  {
        if (instances.containsKey(
              makeLocaleCode(language , region )))
        {
              return (LingualResource) instances.get(
                    makeLocaleCode(language , region ));
        }
        else
        {
```

```
                return new
                    LingualResource(language, region);
            }
        }
        public String getLocaleString(String code)
        {
            return resourceBundle.getString(code);
        }
        private static String makeLocaleCode(
            String language, String region)
        {
            return language + "_" + region;
        }
    }
```

其中的 makeLocaleCode()是一个辅助性的方法，在传入语言代码和地区代码时，此方法可以返回一个 Locale 代码。

这个多例类的构造子是私有的，因此不能用 new 关键字来实例化。所有的实例必须通过调用静态 getInstance()方法来得到。在 getInstance()方法被调用时，程序会首先检查传入的 Locale 代码是否已经在 instances 集合中存在；如果已经存在。即直接返回它所对应的 LingualResource 对象，否则就会首先创建一个这个 Locale 代码所对应的 LingualResource 对象，将之存入 instances 集合，并返回这个实例。

下面给出一个客户端的源代码。

代码清单 6：客户端类 LingualResourceTester 的源代码

```
package com.javapatterns.multilingual;
public class LingualResourceTester
{
    public static void main(String[] args)
    {
        LingualResource ling =
            LingualResource.getInstance("en" , "US");
        String usDollar = ling.getLocaleString("USD");
        System.out.println("USD=" + usDollar);
        LingualResource lingZh =
            LingualResource.getInstance("zh" , "CH");
        String usDollarZh = lingZh.getLocaleString("USD");
        System.out.println("USD=" + usDollarZh);
    }
}
```

如果用户是美国用户，那么在 JSP 网页中可以通过调用 getLocaleString()方法得到相应的英文说明。

代码清单 7：Resource 文件 res_en_US.properties 的内容

```
    LingualResource ling =
        LingualResource.getInstance("en" , "US");
```

```
String usDollar = ling.getLocaleString("USD");
```

就会返还：

```
US Dollar
```

相应地，如果用户是中国大陆的用户，那么在 JSP 网页中可以通过调用 getLocaleString() 方法得到相应的中文说明。比如：

```
LingualResource ling =
    LingualResource.getInstance("zh" , "CH");
String usDollar = ling.getLocaleString("USD");
```

就会返还

```
美元
```

## Resource 文件的内容

为美国英文准备的 Resource 文件 res_en_US.properties 的内容如下：

```
USD=US Dollar
JPY=Japanese Yen
```

为简体中文准备的 Resource 文件 res_zh_CH.properties 的内容如下。

代码清单 8：Resource 文件 res_zh_CH.properties 的内容

```
USD=美元
JPY=日元
```

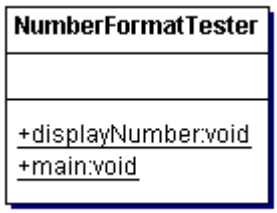
## 问答题

1. 请问为什么 Internationalization 又简称做 i18n？
2. 请给出一个根据语言代码和地区代码将数目字格式化的例子。
3. 请给出一个根据语言代码和地区代码将货币数目字格式化的例子。
4. 请给出一个根据语言代码和地区代码将百分比格式化的例子。

## 问答题答案

1. 在英文字 Internationalization 中，第一个字母 i 和最后一个字母 n 之间有 18 个字母，因此，Internationalization 又简称做 i18n。

2. Java 库 java.text.NumberFormat 类提供了对数目字格式的支持，下面给出的就是对数目字格式支持的解答的类图如下所示。




程序的源代码如下。

代码清单 9：Resource 文件 res_zh_CH.properties 的内容

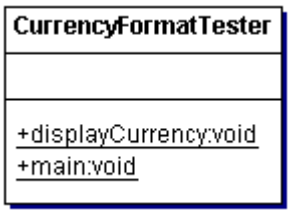
```
package com.javapatterns.multilingual.number;
import java.util.Locale;
import java.text.NumberFormat;
public class NumberFormatTester
{
    static public void displayNumber(
          Double amount, Locale currentLocale)
    {
        NumberFormat formatter;
        String amountOut;
        formatter =
           NumberFormat.getNumberInstance(currentLocale);
        amountOut = formatter.format(amount);
        System.out.println(amountOut + "      "
              + currentLocale.toString());
    }
    static public void main(String[] args)
    {
        displayNumber(new Double(1234567.89),
              new Locale("en", "US"));
        displayNumber(new Double(1234567.89),
              new Locale("de", "DE"));
        displayNumber(new Double(1234567.89),
              new Locale("fr", "FR"));
    }
}
```

在运行时，程序回打印出下面的结果。

代码清单 10：Resource 文件 res_zh_CH.properties 的内容

```
456,789%     en_US
456.789%     de_DE
456 789%     fr_FR
```

3. Java 库 java.text.NumberFormat 类提供了对货币数目格式的支持。下面给出的就是对货币数目格式支持的解答的类图。




程序的源代码如下。

代码清单 11：Resource 文件 res_zh_CH.properties 的内容

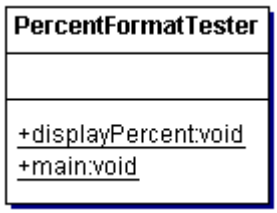
```
package com.javapatterns.multilingual.number;
import java.util.Locale;
import java.text.NumberFormat;
public class CurrencyFormatTester
{
    static public void displayCurrency(Double amount,
        Locale currentLocale)
    {
        NumberFormat formatter;
        String amountOut;
        formatter =
          NumberFormat.getCurrencyInstance(currentLocale);
        amountOut = formatter.format(amount);
        System.out.println(amountOut + "      "
            + currentLocale.toString());
    }
    static public void main(String[] args)
    {
        displayCurrency(new Double(1234567.89),
            new Locale("en", "US"));
        displayCurrency(new Double(1234567.89),
            new Locale("de", "DE"));
        displayCurrency(new Double(1234567.89),
            new Locale("fr", "FR"));
    }
}
```

在运行时，程序回打印出下面的结果。

代码清单 12：Resource 文件 res_zh_CH.properties 的内容

```
$1,234,567.89         en_US
1.234.567,89 DM       de_DE
1 234 567,89 F        fr_FR
```

4. Java 库 java.text.NumberFormat 类提供了对百分比格式的支持，下面给出的就是对百分比式支持的解答的类图。




程序的源代码如下。

代码清单 13：Resource 文件 res_zh_CH.properties 的内容

```
package com.javapatterns.multilingual.number;
import java.util.Locale;
import java.text.NumberFormat;
public class PercentFormatTester
{
    static public void displayPercent(
        Double amount, Locale currentLocale)
    {
        NumberFormat formatter;
        String amountOut;
        formatter =
          NumberFormat.getPercentInstance(currentLocale);
        amountOut = formatter.format(amount);
        System.out.println(amountOut + "    "
            + currentLocale.toString());
    }
    static public void main(String[] args)
    {
        displayPercent(new Double(4567.89),
            new Locale("en", "US"));
        displayPercent(new Double(4567.89),
            new Locale("de", "DE"));
        displayPercent(new Double(4567.89),
            new Locale("fr", "FR"));
    }
}
```

在运行时，程序回打印出下面的结果。

代码清单 14：Resource 文件 res_zh_CH.properties 的内容

```
1,234,567.89    en_US
1.234.567,89    de_DE
1 234 567,89    fr_FR
```

（本章问答题第 2、3、4 题的解答参考了[GREEN]的相关例子，本书做了一些改动。）

# 参考文献

[GREEN] Dale Green, Internationalization, http://java.sun.com/docs/books/tutorial/trailmap.html.

[ISO639] International Organization for Standardization, ISO 639 Language Codes, http://www.iso.org

[ISO3166] International Organization for Standardization, ISO 3166 Country Codes, http://www.iso.org

[ISO4217] International Organization for Standardization, ISO 4217 Currency Codes, http://www.iso.org

[CAMP02] David Van Camp, The Pattern Digest, http://patterndigest.tripod.com/