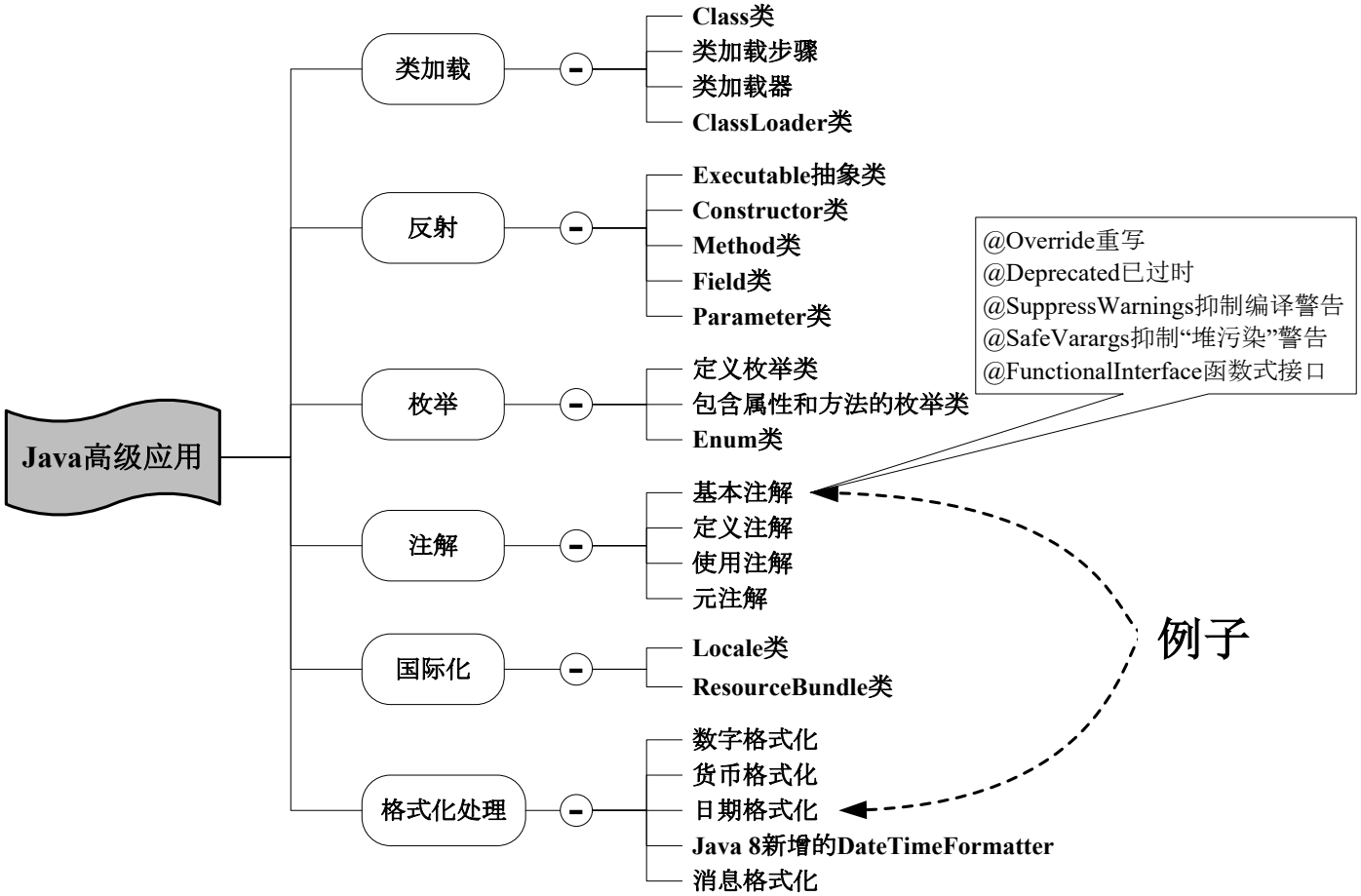
JAVA高级应用

JAVA的类加载机制

JAVA反射机制

JAVA基本注解和自定义注解的使用

JAVA国际化思路


Java高级应用
类加载
Class类
类加载步骤
类加载器
ClassLoader类
反射
Executable抽象类
Constructor类
Method类
Field类
Parameter类
枚举
定义枚举类
包含属性和方法的枚举类
Enum类
注解
基本注解
定义注解
使用注解
元注解
@Override重写
@Deprecated已过时
@SuppressWarnings抑制编译警告
@SafeVarargs抑制“堆污染”警告
@FunctionalInterface函数式接口
国际化
Locale类
ResourceBundle类
格式化处理
数字格式化
货币格式化
日期格式化
Java 8新增的DateTimeFormatter
消息格式化
例子

反射

Java反射机制主要提供了以下功能

- 在运行时判断任意一个对象所属的类
- 在运行时构造任意一个类的对象
- 在运行时获取任意一个类所具有的成员变量和方法
- 在运行时调用任意一个对象的方法
- 生成动态代理

```
Class c = Class.forName("java.lang.String");
//获取当前类对象的所有方法
Method[] mtds = c.getDeclaredMethods();
for (Method m : mtds) {
        System.out.println(m);
}
```

在Java程序中获取Class对象有如下方式

-使用Class类的forName(String className)静态方法

-调用某个类的class属性来获取该类对应的Class对象

-调用某个对象的getClass()方法来获取该类对应的Class对象

【示例】获取Class对象

```
//1.使用Class.forName()方法
Class strClass=Class.forName("java.lang.String");

//2.使用类的calss属性
Class<Float> fclass=Float.class;

// 3.使用实例对象的getClass()方法
Date nowTime = new Date();
Class dateClass = nowTime.getClass();
```

Class类

java.lang.Class类封装一个对象和接口运行时的状态，其常用方法

方法	功能描述
static Class forName(String className)	返回指定类名的Class对象
T newInstance()	调用缺省构造方法，返回该Class对象的一个实例
String getName()	返回Class对象所对应类的名称
Constructor<?>[] getConstructors()	返回Class对象所对应类的所有public构造方法
Method[] getMethods()	返回Class对象所对应类的所有public方法
Constructor<?>[] getDeclaredConstructors()	返回Class对象所对应类的所有构造方法，与访问权限无关
Method[] getDeclaredMethods()	返回Class对象所对应类的所有方法，与访问权限无关
Field[] getFields()	返回Class对象所对应类的public成员变量
Field getField(String name)	返回Class对象所对应类的指定参数的public成员变量
Field[] getDeclaredFields()	返回Class对象所对应类的所有成员变量，与访问权限无关
Field getDeclaredField(String name)	返回Class对象所对应类指定参数的成员变量，与访问权限无关
…	…

```
System.out.println("----String的Class类对象----");
try {
    // 1.使用Class.forName("全限定类名")方法获取String类对象
    Class strClass = Class.forName("java.lang.String");
    System.out.println(strClass);
} catch (ClassNotFoundException e) {
    e.printStackTrace();
}

System.out.println("----Float的Class类对象----");
// 2.使用类的class属性获取Float类对应的Class对象
Class fClass = Float.class;
System.out.println(fClass);

System.out.println("----Date类的Class类对象----");
// 3.使用实例对象的getClass()方法获取Date类对应的Class对象
Date nowTime = new Date();
Class dateClass = nowTime.getClass();
System.out.println(dateClass);

System.out.println("----Date类的父类----");
System.out.println(dateClass.getSuperclass());

System.out.println("----Date类的所有构造方法----");
// 获取所有构造方法
Constructor[] ctors = dateClass.getDeclaredConstructors();
for (Constructor c : ctors) {
    System.out.println(c);
}
```

```
System.out.println("----Date类的所有public方法----");
// 获取所有public方法
Method[] mtds = dateClass.getMethods();
for (Method m : mtds) {
    System.out.println(m);
}

// 构造一个实例对象，构造类中必须提供相应的缺省构造方法实现
try {
    Object obj = dateClass.newInstance();
    System.out.println(obj);
} catch (InstantiationException e) {
    e.printStackTrace();
} catch (IllegalAccessException e) {
    e.printStackTrace();
}
```

Constructor类

Constructor类用于表示类的构造方法，通过调用Class对象的getConstructors()方法可以获取当前类的构造方法的集合

Constructor常用方法

方法	功能描述
String getName()	返回构造方法的名称
Class [] getParameterTypes()	返回当前构造方法的参数类型
int getModifiers()	返回修饰符的整型标识，返回的整数是修饰符public、protected、private、final、static、abstract等关键字所对应的常量，需要使用Modifier工具类的方法解码后才能获得真实的修饰符

```java
public class ConstructorDemo {

    public static void main(String[] args) {
        try {
            // 获取String类对象
            Class clazz = Class.forName("java.lang.String");
            // 返回所有构造方法
            Constructor[] ctors = clazz.getDeclaredConstructors();
            // 遍历构造方法
            for (Constructor c : ctors) {
                // 获取构造方法的修饰符
                int mod = c.getModifiers();
                // 使用Modifier工具类的方法获得真实的修饰符,并输出
                System.out.print(Modifier.toString(mod));

                // 获取构造方法的名称,并输出
                System.out.print(" " + c.getName() + "(");

                // 获取构造方法的参数类型
                Class[] paramTypes = c.getParameterTypes();
                // 循环输出构造方法的参数类型
                for (int i = 0; i < paramTypes.length; i++) {
                    if (i > 0) {
                        System.out.print(", ");
                    }
                    // 输出类型名称
                    System.out.print(paramTypes[i].getName());
                }
                System.out.println(");");
            }
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SecurityException e) {
            e.printStackTrace();
        }
    }
}
```

Method类

Method类用于封装方法的信息，调用Class对象的getMethods()方法或getMethod()可以获取当前类的所有方法或指定方法

Method类的常用方法

方法	功能描述
String getName()	返回方法的名称
Class [] getParameterTypes()	返回当前方法的参数类型
int getModifiers()	返回修饰符的整型标识
Class getReturnType()	返回当前方法的返回类型

```java
// 获取String类对象
Class clazz = Class.forName("java.lang.String");
// 返回所有方法
Method[] mtds = clazz.getMethods();
// 遍历构造方法
for (Method m : mtds) {
    // 获取方法的修饰符
    int mod = m.getModifiers();
    // 使用Modifier工具类的方法获得真实的修饰符,并输出
    System.out.print(Modifier.toString(mod));

    // 获取方法的返回类型,并输出
    Class retType = m.getReturnType();
    System.out.print(" " + retType.getName());

    // 获取方法的名称,并输出
    System.out.print(" " + m.getName() + "(");

    // 获取方法的参数类型
    Class[] paramTypes = m.getParameterTypes();
    // 循环输出方法的参数类型
    for (int i = 0; i < paramTypes.length; i++) {
        if (i > 0) {
            System.out.print(", ");
        }
        // 输出类型名称
        System.out.print(paramTypes[i].getName());
    }
    System.out.println(");");
}
```

Field类

Field类用于封装属性的信息，调用Class对象的getFields()方法或getField()可以获取当前类的所有属性或指定属性

Field类的常用方法

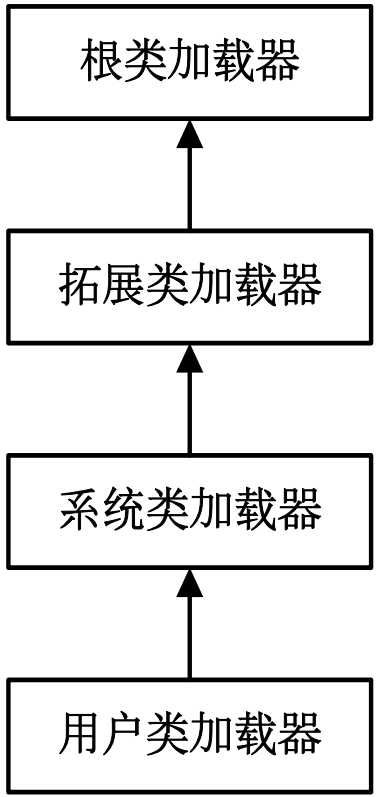
String getName()	返回属性的名称
int getModifiers()	返回修饰符的整型标识
getXxx(Object obj)	获取属性的值，此处的Xxx对应Java8中的基本类型，如果是属性是引用类型，则直接使用get(Object obj)方法
setXxx(Object obj,Xxx val)	设置属性的值，此处的Xxx对应Java8的中基本类型，如果是属性是引用类型，则直接使用set(Object obj,Object val)方法
Class [] getType()	返回当前属性的类型

```java
// 获取Person类对应的Class对象
Class<Person> personClazz = Person.class;
System.out.println("--------Person类的属性--------");
// 返回声明的所有属性包括私有的和受保护的，但不包括超类属性
Field[] fields = personClazz.getDeclaredFields();
for (Field f : fields) {
    // 获取属性的修饰符
    int mod = f.getModifiers();
    // 使用Modifier工具类的方法获得真实的修饰符,并输出
    System.out.print(Modifier.toString(mod));

    // 获取属性的类型,并输出
    Class type = f.getType();
    System.out.print(" " + type.getName());

    // 获取属性的名称,并输出
    System.out.println(" " + f.getName());
}

// 创建一个Person对象
Person p = new Person();

// 使用getDeclaredField()方法表明可获取各种访问控制符的成员变量
// 获取Person类的name属性对象
Field nameField = personClazz.getDeclaredField("name");
// 设置通过反射访问该成员变量时取消访问权限检查
nameField.setAccessible(true);
// 为p对象的name属性设置值，因String是引用类型，所以直接使用set()方法
nameField.set(p, "test");

// 获取Person类的age属性对象
Field ageField = personClazz.getDeclaredField("age");
// 设置通过反射访问该成员变量时取消访问权限检查
ageField.setAccessible(true);
// 调用setInt()方法为p对象的age属性设置值
ageField.setInt(p, 36);
```

类加载器

JVM启动时，会形成由三个类加载器组成的初始类加载器层次结构

- 根类加载器：负责加载支撑JVM运行的位于JRE的lib目录下的核心类库，比如rt.jar、charsets.jar
- 扩展类加载器：负责加载支撑JVM运行的位于JRE的lib目录下的ext扩展目录中的JAR类包
- 系统类/应用程序类加载器：负责加载ClassPath路径下的类包，主要就是加载我们应用中自己写的那些类
- 用户类/自定义加载器：负责加载用户自定义路径下的类包



ClassLoader类

java.lang.ClassLoader是一个抽象类，通过继承ClassLoader类实现自定义的用户类加载器，ClassLoader类常用的方法

方法	功能描述
public Class<?> loadClass(String name)	根据指定的名称加载类
ProtectedClass<?>loadClass(Stringname,booleanresolve)	根据指定的名称加载类
protected Class<?> findClass(String name)	根据指定名称查找类
protected final Class<?> findLoadedClass(String name)	如果JVM已经加载指定的类，则返回该类对应的Class实例，否则返回null
protected final Class<?> defineClass(String name,byte[] b,int off,int len)	将指定的来源于文件或网络上的字节码文件（即.class文件）读入字节数组中，并转换为Class对象
protected final Class<?>findSystemClass(Stringname)	从本地系统文件装入文件
public static ClassLoader getSystemClassLoader()	用于返回系统类加载器的静态方法
public final ClassLoader getParent()	获取当前类加载器的父类加载器
protected final void resolveClass(Class<?> c)	链接指定的类

ClassLoader类

例如，一个应用程序可以创建一个网络类加载器来从服务器下载类文件。 示例代码如下：

```
ClassLoader loader = new NetworkClassLoader(host, port);
  Object main = loader.loadClass("Main", true).newInstance();
```

网络类加载器子类必须定义findClass方法和loadClassData方法来从网络中加载类。 下载组成该类的字节后，应使用defineClass方法创建一个类实例。示例实现如下：

```
class NetworkClassLoader extends ClassLoader {
      String host;
      int port;
public Class findClass(String name) {
      byte[] b = loadClassData(name);
      return defineClass(name, b, 0, b.length);
  }

  private byte[] loadClassData(String name) {
      // 从连接中加载类数据
      ...
  }
}
```

注解

注解（Annotation）是告知编译器要做什么事情的说明，在程序中可以对任何元素进行注解，使用时在其前面增加@符号

基本注解

Java 8在java.lang包中提供了5个基本的注解

@Override：限定重写父类的方法

@Deprecated：标示某个元素已过时

@SuppressWarnings：抑制编译警告的发布

@SafeVarargs：抑制“堆污染”警告

@FunctionalInterface：指定某个接口必须是函数式接口

@Override注解

@Override注解用于指定方法的重写，强制一个子类必须覆盖父类的方法，其语法格式

```
@Override
[访问符] 返回类型 重写方法名(){...}
```

@Deprecated注解

@Deprecated注解标示某个程序元素（接口、类、方法等）已过时

【语法】

```
@Deprecated
//接口、类、方法等程序元素定义
```

@Deprecated注解的使用 DeprecatedDemo.java

```
//定义myMethod()方法已过时
  @Deprecated
  public void myMethod(){
    System.out.println("该方法已过时");
    }
  public static void main(String[] args) {
    //下面使用已过时的方法会被编译警告
    new DeprecatedDemo().myMethod();
  }
```

@SuppressWarnings注解

@SuppressWarnins注解允许开发人员控制编译器警告的发布

【语法】

```
@SuppressWarnins("参数")
//程序元素（包括该元素的所有子元素）
```

【示例】@SuppressWarnins注解忽略unchecked类型警告

```
@SuppressWarnins(“unchecked”) 或 @SuppressWarnins("value=unchecked")
```

在程序中使用没有泛型限制的集合将会引起编译器警告，可以使用

@SuppressWarnins注解抑制此类编译警告的发布

```
@SuppressWarnings(value = "all")
public class SuppressWarningsDemo {
  public static void main(String[] args) {
    List<String> myList = new ArrayList(); // ①
    }
```

@SafeVarargs注解

“堆污染”是将一个不带泛型的变量赋值给一个带泛型的变量，将导致泛型变量污染

【示例】堆污染

```
List list =new ArrayList();//没有使用泛型限制的集合
list.add(10);//未经检查的类型转换，unchecked警告
List<String> ls=list;// ① 发生“堆污染”
System.out.println(ls.get(0));//产生ClassCastException异常
```

@FunctionalInterface注解

@FunctionalInterface注解用于指定某个接口必须是函数式接口

函数式接口：接口里面只能有一个抽象方法

```
@FunctionalInterface
interface GreetingService
{
    void sayMessage(String message);
}
```

```
GreetingService greetService1 = message -> System.out.println("Hello " + message);
```

定义注解

使用@interface定义一个新的注解类型，其语法格式

```
[访问符] @interface 注解名{
        ...
}
```

```java
// 元注解 @Target 指定注解可以应用的程序元素类型
@Target(ElementType.METHOD)
// 元注解 @Retention 指定注解的保留策略，这里是源码中保留，编译后不保
@Retention(RetentionPolicy.SOURCE)
public @interface CustomAnnotation {
    // 注解的属性
    String value() default "default";
    int number() default 42;
}
```

```java
public class Example {

    // 使用自定义注解，并设置属性值
    @CustomAnnotation(value = "ExampleMethod", number = 24)
    public void exampleMethod() {
        // 方法实现
    }
}
```

- 自定义注解是通过@interface声明，注解的成员由未实现的方法组成

【示例】注解中的成员在使用时实现

```
@MyAnno1(comment="功能描述",order =1)
//程序单元（类、接口、方法等）
```

- 在定义注解时，可以使用default语句为注解成员指定缺省值，其语法

```
类型 成员() default 值;
```

【示例】包含缺省值的注解

```
public @interface MyAnno1{
    String comment();
    int order() default 1;
}
```

使用注解

AnnotatedElement接口，用于在反射过程中获取注解信息，并为注解相关操作提供支持，AnnotatedElement接口中的方法有

方法	功能描述
Annotation getAnnotation(Class annotype)	返回调用对象的注解
Annotation getAnnotation(Class annotype)	返回调用对象的所有注解
Annotation getDeclareedAnnotations()	返回调用对象的所有非继承注解
Boolean isAnnotationPresent(Class annotype)	判断与调用对象关联的注解是由annoType指定的

元注解

元注解的作用就是负责注解其他注解

Java 8在java.lang.annotation包下提供了6个元注解

- @Retention
- @Document
- @Target
- @Inherited
- @Repeatable
- 类型注解（Type Annotation）

@Retention注解

@Retention注解用于指定被修饰的注解可以保留多长时间

@Retention使用java.lang.annotation.RetentionPolicy来指定保留策略值。注解保留策略值有三个

策略值	功能描述
RetentionPolicy.SOURCE	注解只在源文件中保留，在编译期间删除
RetentionPolicy.CLASS	注解只在编译期间存在于.class文件中
RetentionPolicy.RUNTIME	运行时JVM可以获取注解信息是最长注解持续期

【示例】运行时JVM可以获取注解信息

```
@Retention(RetentionPolicy.RUNTIME)
//定义注解
```

@Document注解

@Document注解用于指定被修饰的注解可以被javadoc工具提取成文档

【示例】@Document文档注解

```
@Document
//定义注解
```

@Target注解

@Target注解用来限制注解的使用范围，其语法格式

```
@Target({应用类型1,应用类型2,…})
```

应用类型使用ElementType枚举进行指定，其枚举值有

枚举值	功能描述
ElementType.TYPE	可以修饰类、接口、注解或枚举类型
ElementType.FIELD	可以修饰属性（成员变量），包括枚举常量
ElementType.METHOD	可以修饰方法
ElementType.PARAMETER	可以修饰参数
ElementType.CONSTRUCTOR	可以修饰构造方法
ElementType.LOCAL_VARIABLE	可以修饰局部变量
ElementType.ANNOTATION_TYPE	可以修饰注解类
ElementType.PACKAGE	可以修饰包

【示例】指定注解只能修饰字段

```
@Target(ElementType.FIELD)
//定义注解
```

@Inherited注解

@Inherited注解指定注解具有继承性

如果某个注解使用@Inherited进行修饰，则该类使用该注解时，其子类将自动被修饰

国际化

软件的全球化首先就要使程序能支持多国语言，即国际化

在Java中，为解决国际化问题，用到的类大部分由java.util包提供

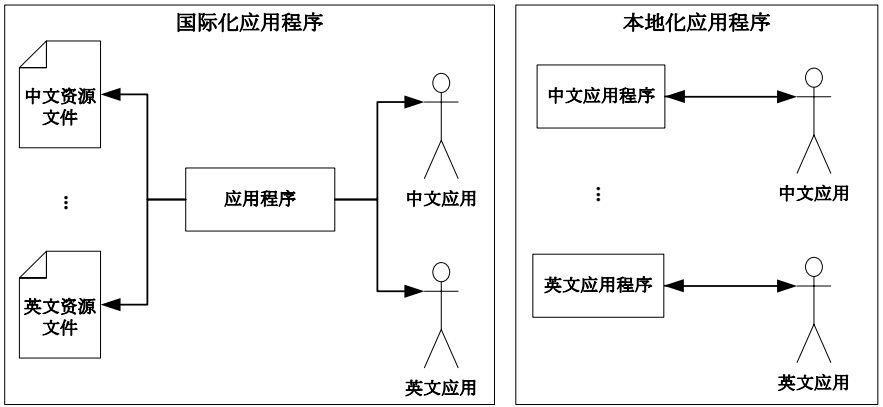
相关的类有

Locale

ResourceBundle

ResourceBundle

PropertyResourceBundle



国际化与本地化区别

Locale类

Locale类是用来标识本地化消息的重要工具类，一个Locale实例代表一种特定的语言和地区，Locale类常用方法

方法	功能描述
Locale(String language)	构造language指定的语言的Locale对象
Locale(String language,String country)	构造language指定的语言和country指定的国家的Locale对象
String getCountry()	返回国家（地区）代码
String getDisplayCountry()	返回国家（地区）名称
String getLanguage()	返回语言代码
String getDisplayLanguage()	返回语言名称
Static Locale getDefault()	获取当前系统信息的对应的Locale对象
Static void setDefault(Locale new)	重新设置缺省的Locale对象

Locale构造方法中需要提供language和country两个参数

常用语言编码：

语言	英语名称	编码
汉语	Chinese	zh
英语	English	en
日本语	Japanese	ja
德语	German	de

常用国家编码：

国家（地区）	英文名称	英文名称
中国	China	CN
英国	Great Britain	GB
日本	Japan	JP
美国	United States	US

【示例】定义中国大陆的Local对象

```
Locale locale = new Locale("zh","CN");
```

ResourceBundle类

ResourceBundle类用于加载国家和语言资源包

资源文件的内容是以“key-value”对组成

资源文件的命名有三种形式：

baseName_language_country.properties

baseName_language.properties

baseName.properties

【示例】资源文件名

```
myres_en_US.properties
myres_zh_CN.properties
myres.properties
```

ResourceBundle类常用的方法

方法	功能描述
public static final ResourceBundle getBundle(String baseName)	使用指定的基本名称、默认的语言环境和调用者的类加载器获取资源包
public abstract Enumeration<String> getKeys()	返回键的枚举
public Locale getLocale()	返回此资源包的语言环境
public final Object getObject(String key)	从此资源包或其某个父包中获取给定键的对象
public final String getString(String key)	从此资源包或其某个父包中获取给定键的字符串
public final String[] getStringArray(String key)	从此资源包或其某个父包中获取给定键的字符串数组
public boolean containsKey(String key)	判断key是否包含在此ResourceBundle及其父包中
public Set<String> keySet()	返回此ResourceBundle及其父包中包含的所有键的Set

数字格式化

在java.text包中提供了一个NumberFormat类，用于完成对数字、百分比进行格式化和对字符串对象进行解析，其常用方法

方法	功能描述
static NumberFormat getNumberInstance()	返回与当前系统信息相关的缺省的数字格式器对象
static NumberFormat getNumberInstance(Locale l)	返回指定Locale为l的数字格式器对象
static NumberFormat getPercentInstance()	返回与当前系统信息相关的缺省的百分比格式器对象
static NumberFormat getPercentInstance(Locale l)	返回指定Locale为l的百分比格式器对象
static NumberFormat getCurrencyInstance()	返回与当前系统信息相关的缺省的货币格式器对象
static NumberFormat getCurrencyInstance (Locale l)	返回指定Locale为l的货币格式器对象
String format(double number)	将数字number格式化为字符串返回
Number parse(String source)	将指定的字符串解析为Number对象

货币格式化

NumberFormat除了能对数字、百分比格式化外，还可以对货币数据格式化

货币格式化通常是在钱数前面加上类似于“￥”、“$”的货币符号来区分货币类型

使用NumberFormat的静态方法getCurrencyInstance()方法来获取格式器

日期格式化

Java中日期和时间的格式化是通过DateFormat类来完成,其方法有

方法	功能描述
static DateFormat getDateInstance()	返回缺省样式的日期格式器
static DateFormat getDateInstance(int style)	返回缺省指定样式的日期格式器
static DateFormat getDateInstance(int style, Locale aLocale)	返回缺省指定样式和Locale信息的日期格式器
static DateFormat getTimeInstance()	返回缺省样式的时间格式器
static DateFormat getTimeInstance(int style)	返回缺省指定样式的时间格式器
static DateFormat getTimeInstance(int style, Locale aLocale)	返回缺省指定样式和Locale信息的时间格式器
static DateFormat getDateTimeInstance()	返回缺省样式的日期时间格式器
static DateFormat getDateTimeInstance(int dateStyle, int timeStyle)	返回缺省指定样式的日期时间格式器
static DateFormat getDateTimeInstance(int dateStyle, int timeStyle, Locale aLocale)	返回缺省指定样式和Locale信息的日期时间格式器

上表中dateStyle日期样式和timeStyle时间样式，用于控制输出日期、时间的显示形式，常用的样式控制有

DateFormat.FULL

DateFormat.LONG

DateFormat.DEFAULT

DateFormat.SHORT

SimpleDateFormat类

Java提供了更加简便的日期格式器SimpleDateFormat类，该类是DateFormat的子类

通过模式字符可以构建控制日期、时间格式的模式串

格式串	输出实例
yyyy.MM.dd G 'at' HH:mm:ss	2010.03.22 公元 at 13:57:47
h:mm a	1:58 下午
yyyy年MM月dd日 HH时mm分ss秒	2010年03月22日 13时50分02秒
EEE, d MMM yyyy HH:mm:ss	星期一, 22 三月 2010 13:58:52
yyyy-MM-dd HH:mm:ss	2010-03-22 13:50:02

Java 8新增的DateTimeFormatter

DateTimeFormatter类功能非常强大，相当于DateFormat和SimpleDateFormat的综合体

获取DateTimeFormatter对象有三种方式

- 直接使用静态常量创建DateTimeFormatter对象
- 使用不同风格的枚举值来创建DateTimeFormatter对象
- 根据模式字符串来创建DateTimeFormatter对象

本章总结

- Class类的实例表示正在运行的Java应用程序中的类和接口
- JVM为每种类型创建一个独一无二的Class对象，可以使用==操作符来比较类对象
- ClassLoader是JVM将类装入内存的中间类
- 反射是Java被视为动态（或准动态）语言的一个关键性质
- 利用Java反射机制可以获取类的相关定义信息：属性、方法和访问修饰符等
- Constructor类用于表示类中的构造方法，Method类提供关于类或接口上某个方法的信息，Field类提供有关类或接口的属性信息
- 注解能将补充的信息补充到源文件中而不会改变程序的操作
- 通过反射机制来获取注解的相关信息
- Java是一个全面支持国际化的语言，使用Unicode处理所有字符串
- Java通过类Locale设定语言及国家
- NumberFormat用于进行数字、货币格式化
- DateFormat、SimpleDateFormat用于格式化日期和时间