链滴

从源码角度看枚举

作者：vcjmhg
原文链接：https://ld246.com/article/1588081980196
来源网站：链滴

[TOC]

概述

关于枚举类型，我们学过 C 语言的小伙伴应该都不陌生。所谓枚举类型，是一种特殊的数据结构，它取值范围是有限的，所有取值结果都可以枚举出来，比如说一年四季（春夏秋冬）。对于确定范围的量取值，我们通过枚举类型来表现较之用类表示更加简洁、方便、安全。

下边我们借助一些例子来介绍枚举类型的使用以及其实现原理。

基础

枚举类型的定义和使用都是比较简单的，比如要表示一年的四个季节（春夏秋冬）我们可以定下如下类 ether:

```
public enum Wether{
    SPRING,SUMMER,AUTUMN,WINTER
}
```

Wether 中分别定义了春（Spring）、夏（SUMMER）、秋（AUTUMN），冬（WINTER）这四个。枚举类型使用 enum 这个关键字来进行定义。当然枚举类型可以像类一样单独写到一个文件中，也以写到一个类内部，仅供该类使用。

使用时也非常简单可以直接通过类型命调用

```
public class Test {
    public static void main(String[] args) {
        System.out.println("我是春天:"+Wether.SPRING);
        System.out.println("我是夏天:"+Wether.SUMMER);
        System.out.println("我是秋天:"+Wether.AUTUMN);
        System.out.println("我是冬天:"+Wether.WINTER);
```

```
    }
}
```

输出结果如下

原理

看似简单的枚举类 Wether 其背后有隐藏着什么故事，下边我们拭目以待。

首先我们通过 jad 来对文件进行反编译我们的到如下代码

```
// Decompiled by Jad v1.5.8e2. Copyright 2001 Pavel Kouznetsov.
// Jad home page: http://kpdus.tripod.com/jad.html
// Decompiler options: packimports(3) fieldsfirst ansi space
// Source File Name:   Wether.java

public final class Wether extends Enum
{

    public static final Wether SPRING;
    public static final Wether SUMMER;
    public static final Wether AUTUMN;
    public static final Wether WINTER;
    private static final Wether $VALUES[];

    public static Wether[] values()
    {
        return (Wether[])$VALUES.clone();
    }

    public static Wether valueOf(String name)
    {
        return (
        )Enum.valueOf(testEnum/Wether, name);
    }

    private Wether(String s, int i)
    {
        super(s, i);
    }

    static
    {
        SPRING = new Wether("SPRING", 0);
```

```
        SUMMER = new Wether("SUMMER", 1);
        AUTUMN = new Wether("AUTUMN", 2);
        WINTER = new Wether("WINTER", 3);
        $VALUES = (new Wether[] {
            SPRING, SUMMER, AUTUMN, WINTER
        });
    }
}
```

我们会发现 Java 编译器在实现枚举类型的时候，其实是按照类类型来进行处理的，也就是说：**枚举型其本质上仍然是类，是一个继承自 Enmu 的类。**但是由于编译器帮我们默默的做了一些事情，使我们使用起来更加方便也更加高效。

详细来说针对枚举类型编译器在背后为我们做了一下几件事情。

- 定义一个继承自 Enmu 的一个类命名为 Wether
- 为每个枚举实例对应创建一个类对象，这些类对象是用 public static final 修饰的。同时生成一个组，用于保存全部的类对象
- 生成一个静态代码块，用于初始化类对象和类对象数组
- 生成一个构造函数，构造函数包含自定义参数和两个默认参数（下文会讲解这两个默认参数）
- 生成一个静态的 values() 方法，用于返回所有的类对象
- 生成一个静态的 valueOf() 方法，根据 name 参数返回对应的类实例

官方的对编译器对 Enum 的处理做了如下说明：文档地址

8.9. Enums

An *enum declaration* specifies a new enum type.

```
EnumDeclaration:
        ClassModifiers_opt  enum  Identifier  Interfaces_opt  EnumBody

EnumBody:
        {  EnumConstants_opt  ,_opt  EnumBodyDeclarations_opt  }
```

Enum types (§8.9) must not be declared `abstract`; doing so will result in a compile-time error.

An enum type is implicitly `final` unless it contains at least one enum constant that has a class body.

It is a compile-time error to explicitly declare an enum type to be `final`.

Nested enum types are implicitly `static`. It is permissible to explicitly declare a nested enum type to be `static`.

This implies that it is impossible to define a local (§14.3) enum, or to define an enum in an inner class (§8.1.3).

The direct superclass of an enum type named E is `Enum<E>` (§8.1.4).

An enum type has no instances other than those defined by its enum constants. It is a compile-time error to attempt to explicitly instantiate an enum type (§15.9.1).

The `final clone` method in `Enum` ensures that enum constants can never be cloned, and the special treatment by the serialization mechanism ensures that duplicate instances are never created as a result of deserialization. Reflective instantiation of enum types is prohibited. Together, these four things ensure that no instances of an enum type exist beyond those defined by the enum constants.

为了便于大家阅读我将我对上边的文档尝试进行了翻译

原文链接：从源码角度看枚举

8.9. Enums

An *enum declaration* specifies a new enum type.

声明一个新的枚举类型的说明

```
EnumDeclaration:
        ClassModifiers opt  enum  Identifier  Interfaces opt  EnumBody

EnumBody:

        {  EnumConstants opt  ,opt  EnumBodyDeclarations opt  }
```

Enum types (§8.9) must not be declared abstract; doing so will result in a compile-time error.

新定义的枚举类型必须被定义成抽象类型，负责会造成编译时错误。

An enum type is implicitly final unless it contains at least one enum constant that has a class body. 一个枚举类型除含是final的，除非它有至少一个含有类体的枚举常量。

It is a compile-time error to explicitly declare an enum type to be final.

但明确声明枚举类型为final会出现编译时错误

Nested enum types are implicitly static. It is permissible to explicitly declare a nested enum type to be static.

嵌套的枚举类型是隐式静态的。但我们也可以将嵌套枚举类型显式声明为静态。

This implies that it is impossible to define a local (§14.3) enum, or to define an enum in an inner class (§8.1.3).

这意味着我们不可能定义局部枚举也不可能在一个类内部定义枚举

The direct superclass of an enum type named E is Enum<E> (§8.1.4).

一个名为E的枚举类型的超类是Enum<E>

An enum type has no instances other than those defined by its enum constants. It is a compile-time error to attempt to explicitly instantiate an enum type (§15.9.1).

枚举类型除了由其枚举常量定义的实例外，没有其他实例。尝试显式实例化枚举类型是编译时错误

The final clone method in Enum ensures that enum constants can never be cloned, and the special treatment by the serialization mechanism ensures that duplicate instances are never created as a result of deserialization. Reflective instantiation of enum types is prohibited. Together, these four things ensure that no instances of an enum type exist beyond those defined by the enum constants.

Enum中的最终克隆方法可确保永远不会克隆枚举常量，并且序列化机制的特殊处理可确保不会因反序列化而创建重复的实例。禁止枚举类型的反射实例化。这四件事共同确保了枚举类型的实例不存在超出枚举常量定义的实例的情况。

当让只看官方文档会让人云里雾里的，下边我们结合 Enum 的源码来进行一些分析。

```
package java.lang;

import java.io.Serializable;
import java.io.IOException;
import java.io.InvalidObjectException;
import java.io.ObjectInputStream;
import java.io.ObjectStreamException;


public abstract class Enum<E extends Enum<E>>
        implements Comparable<E>, Serializable {

    private final String name;


    public final String name() {
        return name;
    }


    private final int ;

    public final int ordinal() {
        return ordinal;
    }


    protected Enum(String name, int ordinal) {
        this.name = name;
```

原文链接：从源码角度看枚举

```
    this.ordinal = ordinal;
}


public String toString() {
    return name;
}


public final boolean equals(Object other) {
    return this==other;
}


public final int hashCode() {
    return super.hashCode();
}

protected final Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
}


public final int compareTo(E o) {
    Enum<?> other = (Enum<?>)o;
    Enum<E> self = this;
    if (self.getClass() != other.getClass() && // optimization
        self.getDeclaringClass() != other.getDeclaringClass())
        throw new ClassCastException();
    return self.ordinal - other.ordinal;
}


@SuppressWarnings("unchecked")
public final Class<E> getDeclaringClass() {
    Class<?> clazz = getClass();
    Class<?> zuper = clazz.getSuperclass();
    return (zuper == Enum.class) ? (Class<E>)clazz : (Class<E>)zuper;
}


public static <T extends Enum<T>> T valueOf(Class<T> enumType,
                                            String name) {
    T result = enumType.enumConstantDirectory().get(name);
    if (result != null)
        return result;
    if (name == null)
        throw new NullPointerException("Name is null");
    throw new IllegalArgumentException(
        "No enum constant " + enumType.getCanonicalName() + "." + name);
}


protected final void finalize() { }
```

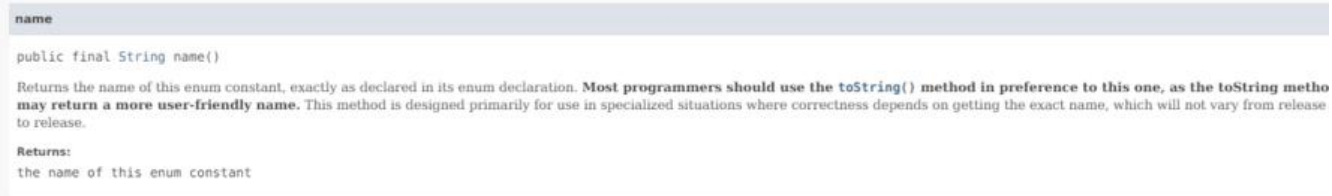
```
    private void readObject(ObjectInputStream in) throws IOException,
        ClassNotFoundException {
        throw new InvalidObjectException("can't deserialize enum");
    }

    private void readObjectNoData() throws ObjectStreamException {
        throw new InvalidObjectException("can't deserialize enum");
    }
}
```

Enum 作为 Java 中所有唯一基类，它在一定程度上定义了枚举类型的公共特征，从上边的源码中我可以看到以下比较重要的几点：

1. Enum 类有两个成员变量，name 和 ordinal 两个成员变量，其中 name 用于枚举常量的名字，如 PRING,SUMMER,AUTUMN 等，oridinal 指的是默认编的序号，一般是从 0 开始。

2. 返回 name 变量的方法有两个：一个是 name()，一个是 toString()，它们都是直接返回 name 量（也就是说实现是一样的），但是它们所代表的含义确是不同的。

关于两者的区别官方的 API 说明如下：



name

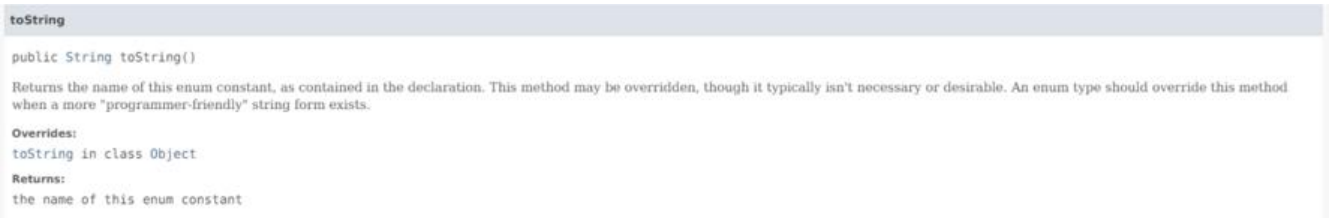
```
public final String name()
```

Returns the name of this enum constant, exactly as declared in its enum declaration. **Most programmers should use the `toString()` method in preference to this one, as the toString metho may return a more user-friendly name.** This method is designed primarily for use in specialized situations where correctness depends on getting the exact name, which will not vary from release to release.

Returns:

the name of this enum constant

从官方 API 中我们可以看到，name() 方法会返回**枚举类型在该枚举类型声明的时所定义的枚举类的精确名称。**官方不太赞同大多数编程者使用该方法获取枚举名称，而应该使用 toString() 类型，为它返回的名称会更加友好。该方法在设计的时候主要是在一些特殊情况下使用（结果的正确性取决给定的精确名称），而且该方法可能会在发布的过程中发生变化。



toString

```
public String toString()
```

Returns the name of this enum constant, as contained in the declaration. This method may be overridden, though it typically isn't necessary or desirable. An enum type should override this method when a more "programmer-friendly" string form exists.

Overrides:

toString in class Object

Returns:

the name of this enum constant

给 name（） 方法类似都是返回枚举常量的名称。官方指出，该方法可被重写（尽管大部分情况下不必须的，也不建议重写）。当一个对编程者更加友好的字符串名称被发现时，应该重写该方法替代原的枚举类型名称。

简而言之，给出 name 方法主要为了便于编程者在某些条件下重写名称的，我们在大部分情况下应该用 toString()。

3. Enum 类不允许克隆，clone() 方法直接抛出异常。（保证枚举永远是单例的）

4. Enum 类实现了 Comparable 接口，因此我们可以直接比较枚举常量的 ordinal 的值。

5. Enum 类不允许反序列化，为了保证枚举永远是单例的。

使用场景

学为所用，讲了那么多主要还是为了能用好枚举，因此我简单总结了一个枚举类型常用的一些场景：

1. 当变量是有穷对象的集合时我们可以尝试使用枚举类型，比如使用到季节(Wether)的时候。

2. 在某些场景下，需要变量是单例的（单例设计模式），我们也可以考虑使用枚举类型，因为枚举类是天生单例的。

原文链接：从源码角度看枚举