**Java的枚举类型用法介绍**

**1 背景**

在java语言中还没有引入枚举类型之前，表示枚举类型的常用模式是声明一组具有int常量。之前我们通常利用public final static 方法定义的代码如下，分别用1 表示春天，2表示夏天，3表示秋天，4表示冬天。

public class Season {

public static final int SPRING = 1;

public static final int SUMMER = 2;

public static final int AUTUMN = 3;

public static final int WINTER = 4;

}

这种方法称作int枚举模式。可这种模式有什么问题呢，我们都用了那么久了，应该没问题的。通常我们写出来的代码都会考虑它的**安全性**、**易用性**和**可读性**。 首先我们来考虑一下它的类型**安全性**。当然**这种模式不是类型安全的**。比如说我们设计一个函数，要求传入春夏秋冬的某个值。但是使用int类型，我们无法保证传入的值为合法。代码如下所示：

private String getChineseSeason(int season){

StringBuffer result = new StringBuffer();

switch(season){

case Season.SPRING :

result.append("春天");

break;

case Season.SUMMER :

result.append("夏天");

break;

case Season.AUTUMN :

result.append("秋天");

break;

case Season.WINTER :

result.append("冬天");

break;

default :

result.append("地球没有的季节");

break;

}

return result.toString();

}

public void doSomething(){

System.out.println(this.getChineseSeason(Season.SPRING));//这是正常的场景

System.out.println(this.getChineseSeason(5));//这个却是不正常的场景，这就导致了类型不安全问题

}

程序getChineseSeason(Season.SPRING)是我们预期的使用方法。可getChineseSeason(5)显然就不是了，而且编译很通过，在运行时会出现什么情况，我们就不得而知了。这显然就不符合Java程序的类型安全。

接下来我们来考虑一下这种模式的**可读性**。使用枚举的大多数场合，我都需要方便得到枚举类型的字符串表达式。如果将int枚举常量打印出来，我们所见到的就是一组数字，这是没什么太大的用处。我们可能会想到使用String常量代替int常量。虽然它为这些常量提供了可打印的字符串，但是它会导致性能问题，因为它依赖于字符串的比较操作，所以这种模式也是我们不期望的。 从**类型安全性**和**程序可读性**两方面考虑，int和String枚举模式的缺点就显露出来了。幸运的是，从Java1.5发行版本开始，就提出了另一种可以替代的解决方案，可以避免int和String枚举模式的缺点，并提供了许多额外的好处。那就是枚举类型（enum type）。接下来的章节将介绍枚举类型的定义、特征、应用场景和优缺点。

**2 定义**

枚举类型（enum type）是指由一组固定的常量组成合法的类型。Java中由关键字enum来定义一个枚举类型。下面就是java枚举类型的定义。

public enum Season {

SPRING, SUMMER, AUTUMN, WINER;

}

**3 特点**

Java定义枚举类型的语句很简约。它有以下特点：

1) 使用关键字enum 2) 类型名称，比如这里的Season 3) 一串允许的值，比如上面定义的春夏秋冬四季 4) 枚举可以单独定义在一个文件中，也可以嵌在其它Java类中

除了这样的基本要求外，用户还有一些其他选择

5) 枚举可以实现一个或多个接口（Interface） 6) 可以定义新的变量 7) 可以定义新的方法 8) 可以定义根据具体枚举值而相异的类

**4 应用场景**

以在背景中提到的类型安全为例，用枚举类型重写那段代码。代码如下：

public enum Season {

SPRING(1), SUMMER(2), AUTUMN(3), WINTER(4);

private int code;

private Season(int code){

this.code = code;

}

public int getCode(){

return code;

}

}

public class UseSeason {

/\*\*

\* 将英文的季节转换成中文季节

\* @param season

\* @return

\*/

public String getChineseSeason(Season season){

StringBuffer result = new StringBuffer();

switch(season){

case SPRING :

result.append("[中文：春天，枚举常量:" + season.name() + "，数据:" + season.getCode() + "]");

break;

case AUTUMN :

result.append("[中文：秋天，枚举常量:" + season.name() + "，数据:" + season.getCode() + "]");

break;

case SUMMER :

result.append("[中文：夏天，枚举常量:" + season.name() + "，数据:" + season.getCode() + "]");

break;

case WINTER :

result.append("[中文：冬天，枚举常量:" + season.name() + "，数据:" + season.getCode() + "]");

break;

default :

result.append("地球没有的季节 " + season.name());

break;

}

return result.toString();

}

public void doSomething(){

for(Season s : Season.values()){

System.out.println(getChineseSeason(s));//这是正常的场景

}

//System.out.println(getChineseSeason(5));

//此处已经是编译不通过了，这就保证了类型安全

}

public static void main(String[] arg){

UseSeason useSeason = new UseSeason();

useSeason.doSomething();

}

}

[中文：春天，枚举常量:SPRING，数据:1] [中文：夏天，枚举常量:SUMMER，数据:2] [中文：秋天，枚举常量:AUTUMN，数据:3] [中文：冬天，枚举常量:WINTER，数据:4]

这里有一个问题，为什么我要将域添加到枚举类型中呢？目的是想将数据与它的常量关联起来。如1代表春天，2代表夏天。

**5 总结**

那么什么时候应该使用枚举呢？每当需要一组固定的常量的时候，如一周的天数、一年四季等。或者是在我们编译前就知道其包含的所有值的集合。Java 1.5的枚举能满足绝大部分程序员的要求的，它的简明，易用的特点是很突出的。

**6 用法**

**用法一：常量**

public enum Color {

RED, GREEN, BLANK, YELLOW

}

**用法二：switch**

enum Signal {

GREEN, YELLOW, RED

}

public class TrafficLight {

Signal color = Signal.RED;

public void change() {

switch (color) {

case RED:

color = Signal.GREEN;

break;

case YELLOW:

color = Signal.RED;

break;

case GREEN:

color = Signal.YELLOW;

break;

}

}

}

**用法三：向枚举中添加新方法**

public enum Color {

RED("红色", 1), GREEN("绿色", 2), BLANK("白色", 3), YELLO("黄色", 4);

// 成员变量

private String name;

private int index;

// 构造方法

private Color(String name, int index) {

this.name = name;

this.index = index;

}

// 普通方法

public static String getName(int index) {

for (Color c : Color.values()) {

if (c.getIndex() == index) {

return c.name;

}

}

return null;

}

// get set 方法

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public int getIndex() {

return index;

}

public void setIndex(int index) {

this.index = index;

}

}

**用法四：覆盖枚举的方法**

public enum Color {

RED("红色", 1), GREEN("绿色", 2), BLANK("白色", 3), YELLO("黄色", 4);

// 成员变量

private String name;

private int index;

// 构造方法

private Color(String name, int index) {

this.name = name;

this.index = index;

}

//覆盖方法

@Override

public String toString() {

return this.index+"\_"+this.name;

}

}

**用法五：实现接口**

public interface Behaviour {

void print();

String getInfo();

}

public enum Color implements Behaviour{

RED("红色", 1), GREEN("绿色", 2), BLANK("白色", 3), YELLO("黄色", 4);

// 成员变量

private String name;

private int index;

// 构造方法

private Color(String name, int index) {

this.name = name;

this.index = index;

}

//接口方法

@Override

public String getInfo() {

return this.name;

}

//接口方法

@Override

public void print() {

System.out.println(this.index+":"+this.name);

}

}

**用法六：使用接口组织枚举**

public interface Food {

enum Coffee implements Food{

BLACK\_COFFEE,DECAF\_COFFEE,LATTE,CAPPUCCINO

}

enum Dessert implements Food{

FRUIT, CAKE, GELATO

}

}

**枚举的线程安全性及序列化问题**

写在前面：Java SE5提供了一种新的类型-[Java的枚举类型](http://www.hollischuang.com/archives/195)，关键字enum可以将一组具名的值的有限集合创建为一种新的类型，而这些具名的值可以作为常规的程序组件使用，这是一种非常有用的功能。本文将深入分析枚举的源码，看一看枚举是怎么实现的，他是如何保证线程安全的，以及为什么用枚举实现的单例是最好的方式。

**枚举是如何保证线程安全的**

要想看源码，首先得有一个类吧，那么枚举类型到底是什么类呢？是enum吗？答案很明显不是，enum就和class一样，只是一个关键字，他并不是一个类，那么枚举是由什么类维护的呢，我们简单的写一个枚举：

public enum t {

SPRING,SUMMER,AUTUMN,WINTER;

}

然后我们使用反编译，看看这段代码到底是怎么实现的，反编译（[Java的反编译](http://www.hollischuang.com/archives/58)）后代码内容如下：

public final class T extends Enum

{

private T(String s, int i)

{

super(s, i);

}

public static T[] values()

{

T at[];

int i;

T at1[];

System.arraycopy(at = ENUM$VALUES, 0, at1 = new T[i = at.length], 0, i);

return at1;

}

public static T valueOf(String s)

{

return (T)Enum.valueOf(demo/T, s);

}

public static final T SPRING;

public static final T SUMMER;

public static final T AUTUMN;

public static final T WINTER;

private static final T ENUM$VALUES[];

static

{

SPRING = new T("SPRING", 0);

SUMMER = new T("SUMMER", 1);

AUTUMN = new T("AUTUMN", 2);

WINTER = new T("WINTER", 3);

ENUM$VALUES = (new T[] {

SPRING, SUMMER, AUTUMN, WINTER

});

}

}

通过反编译后代码我们可以看到，public final class T extends Enum，说明，该类是继承了Enum类的，同时final关键字告诉我们，这个类也是不能被继承的。当我们使用enmu来定义一个枚举类型的时候，编译器会自动帮我们创建一个final类型的类继承Enum类,所以枚举类型不能被继承，我们看到这个类中有几个属性和方法。

我们可以看到：

public static final T SPRING;

public static final T SUMMER;

public static final T AUTUMN;

public static final T WINTER;

private static final T ENUM$VALUES[];

static

{

SPRING = new T("SPRING", 0);

SUMMER = new T("SUMMER", 1);

AUTUMN = new T("AUTUMN", 2);

WINTER = new T("WINTER", 3);

ENUM$VALUES = (new T[] {

SPRING, SUMMER, AUTUMN, WINTER

});

}

都是static类型的，因为static类型的属性会在类被加载之后被初始化，当一个Java类第一次被真正使用到的时候静态资源被初始化、Java类的加载和初始化过程都是线程安全的。所以，**创建一个enum类型是线程安全的**。

**为什么用枚举实现的单例是最好的方式**

原文地址：<http://cantellow.iteye.com/blog/838473>

**第一种（懒汉，线程不安全）：**

public class Singleton {

private static Singleton instance;

private Singleton (){}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

**第二种（懒汉，线程安全）：**

public class Singleton {

private static Singleton instance;

private Singleton (){}

public static synchronized Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

}

这种写法能够在多线程中很好的工作，而且看起来它也具备很好的lazy loading，但是，遗憾的是，效率很低，99%情况下不需要同步。

**第三种（饿汉）：**

public class Singleton {

private static Singleton instance = new Singleton();

private Singleton (){}

public static Singleton getInstance() {

return instance;

}

}

这种方式基于classloder机制，在[深度分析Java的ClassLoader机制（源码级别）](http://www.hollischuang.com/archives/197)和[Java类的加载、链接和初始化](http://www.hollischuang.com/archives/201)两个文章中有关于CLassload而机制的线程安全问题的介绍，避免了多线程的同步问题，不过，instance在类装载时就实例化，虽然导致类装载的原因有很多种，在单例模式中大多数都是调用getInstance方法， 但是也不能确定有其他的方式（或者其他的静态方法）导致类装载，这时候初始化instance显然没有达到lazy loading的效果。

**第四种（饿汉，变种）：**

public class Singleton {

private Singleton instance = null;

static {

instance = new Singleton();

}

private Singleton (){}

public static Singleton getInstance() {

return this.instance;

}

}

表面上看起来差别挺大，其实更第三种方式差不多，都是在类初始化即实例化instance。

**第五种（静态内部类）：**

public class Singleton {

private static class SingletonHolder {

private static final Singleton INSTANCE = new Singleton();

}

private Singleton (){}

public static final Singleton getInstance() {

return SingletonHolder.INSTANCE;

}

}

这种方式同样利用了classloder的机制来保证初始化instance时只有一个线程，它跟第三种和第四种方式不同的是（很细微的差别）：第三种和第四种方式是只要Singleton类被装载了，那么instance就会被实例化（没有达到lazy loading效果），而这种方式是Singleton类被装载了，instance不一定被初始化。因为SingletonHolder类没有被主动使用，只有显示通过调用getInstance方法时，才会显示装载SingletonHolder类，从而实例化instance。想象一下，如果实例化instance很消耗资源，我想让他延迟加载，另外一方面，我不希望在Singleton类加载时就实例化，因为我不能确保Singleton类还可能在其他的地方被主动使用从而被加载，那么这个时候实例化instance显然是不合适的。这个时候，这种方式相比第三和第四种方式就显得很合理。

**第六种（**[**枚举**](http://www.hollischuang.com/index.php/archives/345/)**）：**

public enum Singleton {

INSTANCE;

public void whateverMethod() {

}

}

这种方式是Effective Java作者Josh Bloch 提倡的方式，它不仅能避免多线程同步问题，而且还能防止反序列化重新创建新的对象，可谓是很坚强的壁垒啊，在[深度分析Java的枚举类型—-枚举的线程安全性及序列化问题](http://www.hollischuang.com/index.php/archives/349/)中有详细介绍枚举的线程安全问题和序列化问题，不过，个人认为由于1.5中才加入enum特性，用这种方式写不免让人感觉生疏，在实际工作中，我也很少看见有人这么写过。

**第七种（双重校验锁）：**

public class Singleton {

private volatile static Singleton singleton;

private Singleton (){}

public static Singleton getSingleton() {

if (singleton == null) {

synchronized (Singleton.class) {

if (singleton == null) {

singleton = new Singleton();

}

}

}

return singleton;

}

}

**总结**

有两个问题需要注意：

1.如果单例由不同的类装载器装入，那便有可能存在多个单例类的实例。假定不是远端存取，例如一些servlet容器对每个servlet使用完全不同的类装载器，这样的话如果有两个servlet访问一个单例类，它们就都会有各自的实例。

2.如果Singleton实现了java.io.Serializable接口，那么这个类的实例就可能被序列化和复原。不管怎样，如果你序列化一个单例类的对象，接下来复原多个那个对象，那你就会有多个单例类的实例。[单例与序列化的那些事儿](http://www.hollischuang.com/archives/1144)

对第一个问题修复的办法是：

private static Class getClass(String classname)

throws ClassNotFoundException {

ClassLoader classLoader = Thread.currentThread().getContextClassLoader();

if(classLoader == null)

classLoader = Singleton.class.getClassLoader();

return (classLoader.loadClass(classname));

}

}

对第二个问题修复的办法是：

public class Singleton implements java.io.Serializable {

public static Singleton INSTANCE = new Singleton();

protected Singleton() {

}

private Object readResolve() {

return INSTANCE;

}

}

对我来说，我比较喜欢第三种和第五种方式，简单易懂，而且在JVM层实现了线程安全（如果不是多个类加载器环境），一般的情况下，我会使用第三种方式，只有在要明确实现lazy loading效果时才会使用第五种方式，另外，如果涉及到反序列化创建对象时我会试着使用枚举的方式来实现单例，不过，我一直会保证我的程序是线程安全的，而且我永远不会使用第一种和第二种方式，如果有其他特殊的需求，我可能会使用第七种方式，毕竟，JDK1.5已经没有双重检查锁定的问题了。

不过一般来说，第一种不算单例，第四种和第三种就是一种，如果算的话，第五种也可以分开写了。所以说，一般单例都是五种写法。懒汉，恶汉，双重校验锁，枚举和静态内部类。

在上面的引用部分，我们看到一共有七种实现单例的方式，其中，**Effective Java**作者Josh Bloch 提倡使用枚举的方式，既然大神说这种方式好，那我们就要知道它为什么好？

**1. 枚举写法简单**

写法简单这个大家看看[转+注]单例模式的七种写法里面的实现就知道区别了。

public enum EasySingleton{

INSTANCE;

}

你可以通过EasySingleton.INSTANCE来访问。

**2. 枚举自己处理序列化**

我们知道，以前的所有的单例模式都有一个比较大的问题，就是一旦实现了Serializable接口之后，就不再是单例得了，因为，每次调用 readObject()方法返回的都是一个新创建出来的对象，有一种解决办法就是使用readResolve()方法来避免此事发生。但是，**为了保证枚举类型像Java规范中所说的那样，每一个枚举类型极其定义的枚举变量在JVM中都是唯一的，在枚举类型的序列化和反序列化上，Java做了特殊的规定。**原文如下：

Enum constants are serialized differently than ordinary serializable or externalizable objects. The serialized form of an enum constant consists solely of its name; field values of the constant are not present in the form. To serialize an enum constant, ObjectOutputStream writes the value returned by the enum constant’s name method. To deserialize an enum constant, ObjectInputStream reads the constant name from the stream; the deserialized constant is then obtained by calling the java.lang.Enum.valueOf method, passing the constant’s enum type along with the received constant name as arguments. Like other serializable or externalizable objects, enum constants can function as the targets of back references appearing subsequently in the serialization stream. The process by which enum constants are serialized cannot be customized: any class-specific writeObject, readObject, readObjectNoData, writeReplace, and readResolve methods defined by enum types are ignored during serialization and deserialization. Similarly, any serialPersistentFields or serialVersionUID field declarations are also ignored–all enum types have a fixedserialVersionUID of 0L. Documenting serializable fields and data for enum types is unnecessary, since there is no variation in the type of data sent.

大概意思就是说，在序列化的时候Java仅仅是将枚举对象的name属性输出到结果中，反序列化的时候则是通过java.lang.Enum的valueOf方法来根据名字查找枚举对象。同时，编译器是不允许任何对这种序列化机制的定制的，因此禁用了writeObject、readObject、readObjectNoData、writeReplace和readResolve等方法。 我们看一下这个valueOf方法：

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 const " + enumType +"." + name);

}

从代码中可以看到，代码会尝试从调用enumType这个Class对象的enumConstantDirectory()方法返回的map中获取名字为name的枚举对象，如果不存在就会抛出异常。再进一步跟到enumConstantDirectory()方法，就会发现到最后会以反射的方式调用enumType这个类型的values()静态方法，也就是上面我们看到的编译器为我们创建的那个方法，然后用返回结果填充enumType这个Class对象中的enumConstantDirectory属性。

所以，**JVM对序列化有保证。**

**3.枚举实例创建是thread-safe(线程安全的)**

我们在[深度分析Java的ClassLoader机制](http://www.hollischuang.com/archives/199)和[Java类的加载、链接和初始化](http://www.hollischuang.com/archives/201)两个文章中分别介绍过，当一个Java类第一次被真正使用到的时候静态资源被初始化、Java类的加载和初始化过程都是线程安全的。所以，**创建一个enum类型是线程安全的**。