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，随便传一个整数都可以编译通过，所以不安全。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、特性、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;
  //每个元素都是public static final修饰的
}
```

默认继承 java.lang.Enum 类，看字节码文件就可以。所以不能继承其他父类；其中 java.lang.Enum 类实现了 java.lang.Serializable 和 java.lang.Comparable 接口；
使用 enum 定义，默认使用 final 修饰，因此不能派生子类；看字节码final enum修饰类
构造器默认使用 private 修饰，且只能使用 private 修饰；看字节码private修饰构造器
枚举类所有实例必须在第一行给出，默认添加 public static final 修饰，否则无法产生实例；枚举类中每一个元素都是用public static final修饰，看字节码得到的

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 :  season.name() 获取常量的名称，可读性高。
            result.append("[中文：春天，枚举常量:" + season.name() + "，数据:" +
season.getCode() + "]"); season.getCode() 获取枚举表示的具体数值
            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();
   }
}
```

这里其实是四个内部类Spring extends Season,并且有包含有参构造方法

[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) {
```

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
    }
}
```

使用枚举类的注意事项

14.【参考】枚举类名建议带上 Enum 后缀，枚举成员名称需要全大写，单词间用下划线隔开。
说明：枚举其实就是特殊的常量类，且构造方法被默认强制是私有。
正例：枚举名字：DealStatusEnum；成员名称：SUCCESS / UNKOWN_REASON。

枚举类型对象之间的值比较，是可以使用==，直接来比较值，是否相等的，不是必须使用equals方法的哟。

因为枚举类Enum已经重写了equals方法

```
/**
 * Returns true if the specified object is equal to this
 * enum constant.
 *
 * @param other the object to be compared for equality with this object.
```

```
     * @return  true if the specified object is equal to this
     *          enum constant.
     */
    public final boolean equals(Object other) {
        return this==other;
    }
```

枚举类的实现原理（内部类）

这部分参考https://blog.csdn.net/mhmyqn/article/details/48087247

> Java从JDK1.5开始支持枚举，也就是说，Java一开始是不支持枚举的，就像泛型一样，都是JDK1.5才加入的新特性。通常一个特性如果在一开始没有提供，在语言发展后期才添加，会遇到一个问题，就是向后兼容性的问题。底层通过内部类兼容枚举
>
> 像Java在1.5中引入的很多特性，为了向后兼容，编译器会帮我们写的源代码做很多事情，比如泛型为什么会擦除类型，为什么会生成桥接方法，foreach迭代，自动装箱/拆箱等，这有个术语叫“语法糖”，而编译器的特殊处理叫“解语法糖”。那么像枚举也是在JDK1.5中才引入的，又是怎么实现的呢？

> Java在1.5中添加了java.lang.Enum抽象类，它是所有枚举类型基类。提供了一些基础属性和基础方法。同时，对把枚举用作Set和Map也提供了支持，即java.util.EnumSet和java.util.EnumMap。

接下来定义一个简单的枚举类

```
public enum Day {
    MONDAY {
        @Override
        void say() {
            System.out.println("MONDAY");
        }
    }
    , TUESDAY {
        @Override
        void say() {
            System.out.println("TUESDAY");
        }
    }, FRIDAY("work"){
        @Override
        void say() {
            System.out.println("FRIDAY");
        }
    }, SUNDAY("free"){
        @Override
        void say() {
            System.out.println("SUNDAY");
        }
    };
```

内部类“MONDAY”=Day$1字节码：构造器(name,original数值)

```
final class se基础.demo.枚举.Day$1 extends se基础.demo.枚举.Day {
  se基础.demo.枚举.Day$1(java.lang.String, int);
    Code:
      0: aload_0
      1: aload_1
      2: iload_2
      3: aconst_null
      4: invokespecial #1                  // Method se基础/demo/枚举/Day."<init>":(Ljava/lang/String;ILse基础/demo/枚举/Day$1;)V
      7: return
```

```
    String work;
    //没有构造参数时，每个实例可以看做常量。
    //使用构造参数时，每个实例都会变得不一样，可以看做不同的类型，所以编译后会生成
实例个数对应的class。
    private Day(String work) {
        this.work = work;
    }
    private Day() {

    }
    //枚举实例必须实现枚举类中的抽象方法
    abstract void say ();

}
```

进一步证明枚举常量是内部类继承自外部枚举类

反编译结果

```
D:\MyTech\out\production\MyTech\com\javase\枚举类>javap Day.class
Compiled from "Day.java"

public abstract class com.javase.枚举类.Day extends java.lang.Enum<com.javase.
枚举类.Day> {
  public static final com.javase.枚举类.Day MONDAY;
  public static final com.javase.枚举类.Day TUESDAY;
  public static final com.javase.枚举类.Day FRIDAY;
  public static final com.javase.枚举类.Day SUNDAY;
  java.lang.String work;
  public static com.javase.枚举类.Day[] values();
  public static com.javase.枚举类.Day valueOf(java.lang.String);
  abstract void say();
  com.javase.枚举类.Day(java.lang.String, int, com.javase.枚举类.Day$1);
  com.javase.枚举类.Day(java.lang.String, int, java.lang.String, com.javase.枚
举类.Day$1);
  static {};
}
```

> 可以看到，一个枚举在经过编译器编译过后，变成了一个抽象类，它继承了java.lang.Enum；而枚举中定义的枚举常量，变成了相应的public static final属性，而且其类型就抽象类的类型，名字就是枚举常量的名字.
>
> 同时我们可以在Operator.class的相同路径下看到四个内部类的.class文件com/mikan/Day$1.class、com/mikan/Day$2.class、com/mikan/Day$3.class、com/mikan/Day$4.class，也就是说这四个命名字段分别使用了内部类来实现的；同时添加了两个方法values()和valueOf(String)；我们定义的构造方法本来只有一个参数，但却变成了三个参数；同时还生成了一个静态代码块。这些具体的内容接下来仔细看看。

下面分析一下字节码中的各部分，其中：

```
InnerClasses:
     static #23; //class com/javase/枚举类/Day$4
     static #18; //class com/javase/枚举类/Day$3
     static #14; //class com/javase/枚举类/Day$2
     static #10; //class com/javase/枚举类/Day$1
```

从中可以看到它有4个内部类，这四个内部类的详细信息后面会分析。静态代码块初始化枚举类常量public static final enmu

一、分别设置生成的四个公共静态常量字段的值，（new对象设置）

```
0 new #10 <se基础/demo/枚举/Day$1> 新建内部类对象Day1,Monday
3 dup                              构造器第3个参数：操作数栈中复制一份
4 ldc #11 <MONDAY>                 构造器第1个参数：操作数栈放入字符串Monday
6 iconst_0                         构造器第2个参数：操作数栈放入0
7 invokespecial #12 <se基础/demo/枚举/Day$1.<init>> 执行对象的init方法
10 putstatic #13 <se基础/demo/枚举/Day.MONDAY>
```

编译器添加的构造器 Day(java.lang.String, int, com.javase.枚举类.Day$1);

因此第一个内部类对象，在类加载阶段执行clinit方法时候，初始化了内部的枚举类变量，通过上面的三个此参数的构造器，(枚举类名字、代表数值、枚举类型)，剩下的三个枚举变量也是类似。

二、创建包含所有内部类对象的，枚举对象的数组Day[]

反序列化时候使用，根据名字返回values数组中枚举常量，保证枚举对象不被反序列化破坏唯一性

```
56 iconst_4
57 anewarray #6 <se基础/demo/枚举/Day>
60 dup
```

三、把四个静态常量的枚举对象放入枚举数组中。

0放入操作数栈

```
61 iconst_0
```

获取public static final Day MONDAY对象，加入到枚举数组中。

```
62 getstatic #13 <se基础/demo/枚举/Day.MONDAY>
65 aastore

84 putstatic #3 <se基础/demo/枚举/Day.$VALUES>
87 return
```

其实编译器生成的这个静态代码块做了如下工作：分别设置生成的四个公共静态常量字段的值，同时编译器还生成了一个静态字段$VALUES，保存的是枚举类型定义的所有枚举常量 编译器添加的values方法：

```
public static com.javase.Day[] values();

   0 getstatic #3 <se基础/demo/枚举/Day.$VALUES>
   3 invokevirtual #4 <[Lse基础/demo/枚举/Day;.clone>
   6 checkcast #5 <[Lse基础/demo/枚举/Day;>
   9 areturn
```

获取枚举类在类加载阶段执行clinti初始化，赋值的枚举常量对象数组对象[]，并且通过使用Object.clone方法克隆VALUES数组字段，强制转换成枚举类数组。

这个方法是一个公共的静态方法，所以我们可以直接调用该方法（Day.values()）,返回这个枚举值的数组，另外，这个方法的实现是，克隆在静态代码块中初始化的$VALUES字段的值，并把类型强转成Day[]类型返回。

造方法为什么增加了两个参数?

有一个问题，构造方法我们明明只定义了一个参数，为什么生成的构造方法是三个参数呢？

从Enum类中我们可以看到，为每个枚举都定义了两个属性，name和ordinal，name表示我们定义的枚举常量的名称，如FRIDAY、TUESDAY，而ordinal是一个顺序号，根据定义的顺序分别赋予一个整形值，从0开始。在枚举常量初始化时，会自动为初始化这两个字段，设置相应的值，所以才在构造方法中添加了两个参数。即：

另外三个枚举常量生成的内部类基本上差不多，这里就不重复说明了。

> 我们可以从Enum类的代码中看到，定义的name和ordinal属性都是final的，而且大部分方法也都是final的，特别是clone、readObject、writeObject（这三个方法都是直接抛出异常，不允许克隆序列化）这三个方法，这三个方法和枚举通过静态代码块来进行初始化一起。

> 它保证了枚举类型的不可变性，不能通过克隆（Enum枚举类的final clone()方法体直接抛出异常，不允许克隆），不能通过序列化和反序列化来复制枚举，这能保证一个枚举常量只是一个实例，即是单例的，所以在effective java中推荐使用枚举来实现单例。

枚举类实战

实战一无参

（1）定义一个无参枚举类

```
enum SeasonType {
    SPRING, SUMMER, AUTUMN, WINTER
}
```

（2）实战中的使用

```
// 根据实际情况选择下面的用法即可
SeasonType springType = SeasonType.SPRING;    // 输出 SPRING
String springString = SeasonType.SPRING.toString();    // 输出 SPRING
```

实战二有一参

（1）定义只有一个参数的枚举类

```
enum SeasonType {
    // 通过构造函数传递参数并创建实例
    SPRING("spring"),
    SUMMER("summer"),
    AUTUMN("autumn"),
    WINTER("winter");
```

```
    // 定义实例对应的参数
    private String msg;

    // 必写：通过此构造器给枚举值创建实例
    SeasonType(String msg) {
        this.msg = msg;
    }

    // 通过此方法可以获取到对应实例的参数值
    public String getMsg() {
        return msg;
    }
}
```

(2) 实战中的使用

```
// 当我们为某个实例类赋值的时候可使用如下方式
String msg = SeasonType.SPRING.getMsg();    // 输出 spring
```

实战三有两参

(1) 定义有两个参数的枚举类

```
public enum Season {
    // 通过构造函数传递参数并创建实例
    SPRING(1, "spring"),
    SUMMER(2, "summer"),
    AUTUMN(3, "autumn"),
    WINTER(4, "winter");

    // 定义实例对应的参数
    private Integer key;
    private String msg;

    // 必写：通过此构造器给枚举值创建实例
    Season(Integer key, String msg) {
        this.key = key;
        this.msg = msg;
    }

    // 很多情况，我们可能从前端拿到的值是枚举类的 key ，然后就可以通过以下静态方法获取到对应枚举值
    public static Season valueofKey(Integer key) {
        for (Season season : Season.values()) {
            if (season.key.equals(key)) {
                return season;
            }
        }
        throw new IllegalArgumentException("No element matches " + key);
    }
```

```
    // 通过此方法可以获取到对应实例的 key 值
    public Integer getKey() {
        return key;
    }

    // 通过此方法可以获取到对应实例的 msg 值
    public String getMsg() {
        return msg;
    }
}
```

（2）实战中的使用

```
// 输出 key 为 1 的枚举值实例
Season season = Season.valueofKey(1);
// 输出 SPRING 实例对应的 key
Integer key = Season.SPRING.getKey();
// 输出 SPRING 实例对应的 msg
String msg = Season.SPRING.getMsg();
```

枚举类总结

其实枚举类懂了其概念后，枚举就变得相当简单了，随手就可以写一个枚举类出来。所以如上几个实战小例子一定要先搞清楚概念，然后在练习几遍就 ok 了。

重要的概念，我在这里在赘述一遍，帮助老铁们快速掌握这块知识，首先记住，枚举类中的枚举值可以没有参数，也可以有多个参数，每一个枚举值都是一个实例；

并且还有一点很重要，就是如果枚举值有 n 个参数，那么构造函数中的参数值肯定有 n 个，因为声明的每一个枚举值都会调用构造函数去创建实例，所以参数一定是一一对应的；既然明白了这一点，那么我们只需要在枚举类中把这 n 个参数定义为 n 个成员变量，然后提供对应的 get() 方法，之后通过实例就可以随意的获取实例中的任意参数值了。

如果想让枚举类更加的好用，就可以模仿我在实战三中的写法那样，通过某一个参数值，比如 key 参数值，就能获取到其对应的枚举值，然后想要什么值，就 get 什么值就好了。

枚举 API

我们使用 enum 定义的枚举类都是继承 java.lang.Enum 类的，那么就会继承其 API ，常用的 API 如下：

- String name()

获取枚举名称

- int ordinal()

获取枚举的位置（下标，初始值为 0 ）

- valueof(String msg)

通过 msg 获取其对应的枚举类型。（比如实战二中的枚举类或其它枚举类都行，只要使用得当都可以使用此方法）

- values()

获取枚举类中的所有枚举值（比如在实战三中就使用到了）

总结

枚举本质上是通过普通的类来实现的，只是编译器为我们进行了处理。**每个枚举类型都继承自java.lang.Enum，并自动添加了values和valueOf方法。**

而每个枚举常量是一个静态常量字段，**使用内部类实现**，该内部类继承了枚举类。**所有枚举常量都通过静态代码块来进行初始化，即在类加载期间就初始化。**

另外通过把clone、readObject、writeObject这三个方法定义为final的，同时实现是抛出相应的异常。这样保证了每个枚举类型及枚举常量都是不可变的。**可以利用枚举的这两个特性来实现线程安全的单例。**

参考文章

https://blog.csdn.net/qq_34988624/article/details/86592229
https://www.meiwen.com.cn/subject/slhvhqtx.html
https://blog.csdn.net/qq_34988624/article/details/86592229
https://segmentfault.com/a/1190000012220863
https://my.oschina.net/wuxinshui/blog/1511484
https://blog.csdn.net/hukailee/article/details/81107412

微信公众号

> Enum类是java.lang包中一个类，他是Java语言中所有枚举类型的公共基类。

一、定义

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

1.抽象类。

首先，**抽象类不能被实例化**，所以我们**在java程序中不能使用new关键字来声明一个Enum**，如果想要定义可以使用这样的语法：

```
enum enumName{
    value1,value2
    method1(){}
    method2(){}
}
```

其次，看到抽象类，第一印象是肯定有类继承他。至少我们应该是可以继承他的，所以：

```
/**
 * @author hollis
 */
public class testEnum extends Enum{
}
public class testEnum extends Enum<Enum<E>>{
}
public class testEnum<E> extends Enum<Enum<E>>{
}
```

尝试了以上三种方式之后，得出以下结论：**Enum类无法被继承。**

为什么一个抽象类不让继承？enum定义的枚举是怎么来的？难道不是对Enum的一种继承吗？带着这些疑问我们来反编译以下代码：

```
enum Color {RED, BLUE, GREEN}
```

编译器将会把他转成如下内容：

```
/**
 * @author hollis
 */
public final class Color extends Enum<Color> {
  public static final Color[] values() { return (Color[])$VALUES.clone(); }   public static
Color valueOf(String name) { ... }

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

  public static final Color RED;
  public static final Color BLUE;
  public static final Color GREEN;

  private static final Color $VALUES[];

  static { //类加载阶段是单线程安全的，由JVM控制。因此枚举变量初始化在类加载阶段赋值
    RED = new Color("RED", 0);
    BLUE = new Color("BLUE", 1);
    GREEN = new Color("GREEN", 2);
    $VALUES = (new Color[] { RED, BLUE, GREEN });
  }
}
```

短短的一行代码，被编译器处理过之后竟然变得这么多，看来，enmu关键字是java提供给我们的一个语法糖啊。。。从反编译之后的代码中，我们发现，编译器不让我们继承Enum，但是当我们使用enum关键字定义一个枚举的时候，他会帮我们在编译后默认继承java.lang.Enum类，而不像其他的类一样默认继承Object类。且采用enum声明后，该类会被编译器加上final声明，故该类是无法继承的。 PS：由于JVM类初始化是线程安全的，所以可以采用枚举类实现一个线程安全的单例模式。

2.实现 Comparable 和 Serializable 接口。

Enum实现了Serializable接口，可以序列化。 Enum实现了Comparable接口，可以进行比较，默认情况下，只有同类型的enum才进行比较（原因见后文），要实现不同类型的enum之间的比较，只能复写compareTo方法。

3.泛型： **<E extends Enum<E>>**

怎么理解 <E extends Enum<E>> ?

> 首先，这样写只是为了让Java的API更有弹性，他主要是限定形态参数实例化的对象，故要求只能是Enum，这样才能对 compareTo 之类的方法所传入的参数进行形态检查。所以，**我们完全可以不必去关心他为什么这么设计。**

这里倒是可以关注一下泛型中extends的用法 (/archives/255)，以及 K、V、O、I、E、?、object 这几个符号之间的区别 (/archives/252)。

好啦，我们回到这个令人实在是无法理解的 <E extends Enum<E>>

首先我们先来“翻译”一下这个 Enum<E extends Enum<E>> 到底什么意思，然后再来解释为什么Java要这么用。 我们先看一个比较常见的泛型： List<String> 。这个泛型的意思是，List中存的都是String类型，告诉编译器要接受String类型，并且从List中取出内容的时候也自动帮我们转成String类型。 所以 Enum<E extends Enum<E>> 可以暂时理解为Enum里面的内容都是 E extends Enum<E> 类型。 这里的 E 我们就理解为枚举，extends表示上界，比如： List<? extends Object> ，List中的内容可以是Object或者扩展自Object的类。这就是extends的含义。 所以， E extends Enum<E> 表示为一个继承了 Enum<E> 类型的枚举类型。 那么， Enum<E extends Enum<E>> 就不难理解了，就是一个Enum只接受一个Enum或者他的子类作为参数。相当于把一个子类或者自己当成参数，传入到自身，引起一些特别的语法效果。

为什么Java要这样定义Enum

首先我们来科普一下enum，

```
/**
 * @author hollis
 */
enum Color{
    RED,GREEN,YELLOW
}
enum Season{
    SPRING,SUMMER,WINTER
}
public class EnumTest{
    public static void main(String[] args) {
        System.out.println(Color.RED.ordinal());
        System.out.println(Season.SPRING.ordinal());
    }
}
```

代码中两处输出内容都是 0 ，因为枚举类型的默认的序号都是从零开始的。

要理解这个问题，首先我们来看一个Enum类中的方法（暂时忽略其他成员变量和方法）：

```
/**
 * @author hollis
 */
public abstract class Enum<E extends Enum<E>> implements Comparable<E>, Serializable {
        private final int ordinal;

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

首先我们认为Enum的定义中没有使用 `Enum<E extends Enum<E>>` ，那么compareTo方法就要这样定义（因为没有使用泛型，所以就要使用Object，这也是Java中很多方法常用的方式）：

```
public final int compareTo(Object o)
```

当我们调用compareTo方法的时候依然传入两个枚举类型，在compareTo方法的实现中，比较两个枚举的过程是先将参数转化成Enum类型，然后再比较他们的序号是否相等。那么我们这样比较：

```
Color.RED.compareTo(Color.RED);
Color.RED.compareTo(Season.SPRING);
```

如果在 compareTo 方法中不做任何处理的话，那么以上这段代码返回内容将都是true（因为Season.SPRING的序号和Color.RED的序号都是 0 ）。但是，很明显， Color.RED 和 Season.SPRING 并不相等。

但是Java使用 Enum<E extends Enum<E>> 声明Enum，并且在 compareTo 的中使用 E 作为参数来避免了这种问题。 以上两个条件限制Color.RED只能和Color定义出来的枚举进行比较，当我们试图使用 Color.RED.compareTo(Season.SPRING); 这样的代码是，会报出这样的错误：

```
The method compareTo(Color) in the type Enum<Color> is not applicable for the arguments (Season)
```

他说明，compareTo方法只接受 Enum<Color> 类型。

Java为了限定形态参数实例化的对象，故要求只能是Enum，这样才能对 compareTo之类的方法所传入的参数进行形态检查。 因为“红色”只有和“绿色”比较才有意义，用“红色”和“春天”比较毫无意义，所以，Java用这种方式一劳永逸的保证像compareTo这样的方法可以正常的使用而不用考虑类型。

> PS：在Java中，其实也可以实现“红色”和“春天”比较，因为Enum实现了 Comparable 接口，可以重写compareTo方法来实现不同的enum之间的比较。

二、成员变量

在Enum中，有两个成员变量，一个是名字(name)，一个是序号(ordinal)。 序号是一个枚举常量，表示在枚举中的位置，从0开始，依次递增。

```
/**
 * @author hollis
 */
private final String name;
public final String name() {
    return name;
}
private final int ordinal;
public final int ordinal() {
    return ordinal;
}
```

三、构造函数

前面我们说过，Enum是一个抽象类，不能被实例化，但是他也有构造函数，从前面我们反编译出来的代码中，我们也发现了Enum的构造函数，在Enum中只有一个保护类型的构造函数：

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

文章开头反编译的代码中 `private Color(String s, int i) { super(s, i); }` 中的 `super(s, i);` 就是调用Enum中的这个保护类型的构造函数来初始化name和ordinal。

四、其他方法

Enum当中有以下这么几个常用方法，调用方式就是使用 `Color.RED.methodName（params...）` 的方式调用

```
public String toString() {
    return name;
}

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

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

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

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

public static <T extends Enum<T>> T valueOf(Class<T> enumType,String name) {
    T result = enumType.enumConstantDirectory().get(name);
    if (result != null)
```

方法内容都比较简单，平时能使用的就会也不是很多，这里就不详细介绍了。

参考资料：

java.lang.Enum (http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/7u40-b43/java/lang/Enum.java#Enum)

Java Generics FAQs (http://www.angelikalanger.com/GenericsFAQ/FAQSections/TypeParameters.html)

关于单例模式，我的博客中有很多文章介绍过。作为23种设计模式中最为常用的设计模式，单例模式并没有想象的那么简单。因为在设计单例的时候要考虑很多问题，比如线程安全问题、序列化对单例的破坏等。

单例相关文章一览：

如果你对单例不是很了解，或者对于单例的线程安全问题以及序列化会破坏单例等问题不是很清楚，可以先阅读以上文章。上面六篇文章看完之后，相信你一定会对单例模式有更多，更深入的理解。

我们知道，单例模式，一般有七种写法，那么这七种写法中，最好的是哪一种呢？为什么呢？本文就来抽丝剥茧一下。

哪种写单例的方式最好

在StakcOverflow中，有一个关于What is an efficient way to implement a singleton pattern in Java? (https://stackoverflow.com/questions/70689/what-is-an-efficient-way-to-

如上图，得票率最高的回答是：使用枚举。

回答者引用了Joshua Bloch大神在《Effective Java》中明确表达过的观点：

> 使用枚举实现单例的方法虽然还没有广泛采用，但是单元素的枚举类型已经成为实现Singleton的最佳方法。

如果你真的深入理解了单例的用法以及一些可能存在的坑的话，那么你也许也能得到相同的结论，那就是：使用枚举实现单例是一种很好的方法。

枚举单例写法简单

如果你看过《单例模式的七种写法 (http://www.hollischuang.com/archives/205)》中的实现单例的所有方式的代码，那就会发现，各种方式实现单例的代码都比较复杂。主要原因是在考虑线程安全问题。

我们简单对比下“双重校验锁”方式和枚举方式实现单例的代码。

“双重校验锁”实现单例：

```
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;
    }
}
```

枚举实现单例：

```
public enum Singleton {
    INSTANCE;
    public void whateverMethod() {
    }
}
```

相比之下，你就会发现，枚举实现单例的代码会精简很多。

上面的双重锁校验的代码之所以很臃肿，是因为大部分代码都是在保证线程安全。为了在保证线程安全和锁粒度之间做权衡，代码难免会写的复杂些。但是，这段代码还是有问题的，因为他无法解决反序列化会破坏单例的问题。

枚举可解决线程安全问题

上面提到过。使用非枚举的方式实现单例，都要自己来保证线程安全，所以，这就导致其他方法必然是比较臃肿的。那么，为什么使用枚举就不需要解决线程安全问题呢？

其实，并不是使用枚举就不需要保证线程安全，只不过线程安全的保证不需要我们关心而已。也就是说，其实在“底层”还是做了线程安全方面的保证的。

那么，“底层”到底指的是什么？

这就要说到关于枚举的实现了。这部分内容可以参考我的另外一篇博文深度分析Java的枚举类型—-枚举的线程安全性及序列化问题 (http://www.hollischuang.com/archives/197)，这里我简单说明一下：

定义枚举时使用enum和class一样，是Java中的一个关键字。就像class对应用一个Class类一样，enum也对应有一个Enum类。

通过将定义好的枚举反编译 (http://www.hollischuang.com/archives/58)，我们就能发现，其实枚举在经过 `javac` 的编译之后，会被转换成形如 `public final class T extends Enum` 的定义。

而且，枚举中的各个枚举项同事通过 `static` 来定义的。如：

```
public enum T {
    SPRING,SUMMER,AUTUMN,WINTER;
}
```

反编译后代码为：

```
public final class T extends Enum
{
    //省略部分内容
    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
        });
    }
}
```

了解JVM的类加载机制的朋友应该对这部分比较清楚。 static 类型的属性会在类被加载之后被初始化，我们在深度分析Java的ClassLoader机制（源码级别）(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（因为虚拟机在加载枚举的类的时候，会使用ClassLoader的loadClass方法，而这个方法使用同步代码块保证了线程安全）。所以，创建一个enum类型是线程安全的。

也就是说，我们定义的一个枚举，在第一次被真正用到的时候，会被虚拟机加载并初始化，而这个初始化过程是线程安全的。而我们知道，解决单例的并发问题，主要解决的就是初始化过程中的线程安全问题。

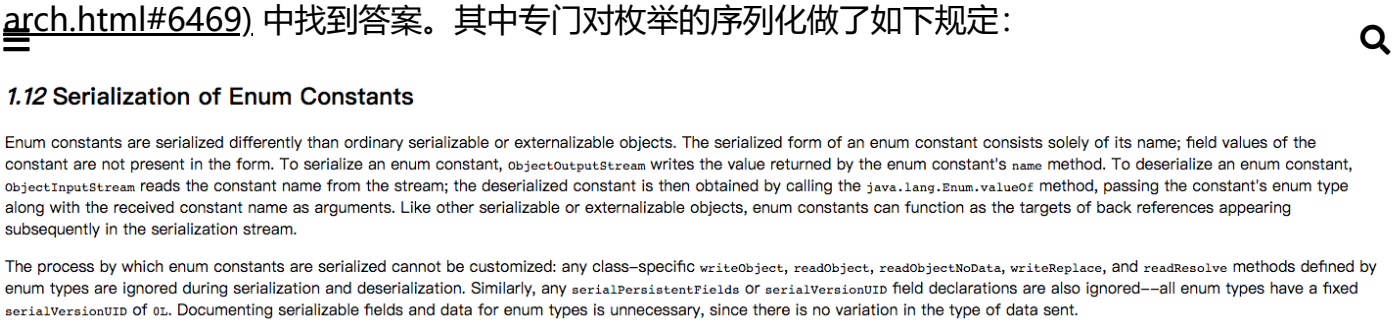
所以，由于枚举的以上特性，枚举实现的单例是天生线程安全的。

枚举可解决反序列化会破坏单例的问题

前面我们提到过，就是使用双重校验锁实现的单例其实是存在一定问题的，就是这种单例有可能被序列化锁破坏，关于这种破坏及解决办法，参看单例与序列化的那些事儿(http://www.hollischuang.com/archives/1144)，这里不做更加详细的说明了。

那么，对于序列化这件事情，为什么枚举又有先天的优势了呢？答案可以在Java Object Serialization Specification (https://docs.oracle.com/javase/7/docs/platform/serialization/spec/serial-

arch.html#6469) 中找到答案。其中专门对枚举的序列化做了如下规定：



1.12 Serialization of Enum Constants

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 fixed `serialVersionUID` 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：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。 枚举对象序列化之后在反序列化，和之前没序列化的对象==判断是true完全相等

普通的Java类的反序列化过程中，会通过反射调用类的默认构造函数来初始化对象。所以，即使单例中构造函数是私有的，也会被反射给破坏掉。由于反序列化后的对象是重新new出来的，所以这就破坏了单例。

规定反序列化通过valueof方法反序列化，禁止自定义序列化规则，保证枚举对象唯一性

但是，枚举的反序列化并不是通过反射实现的。所以，也就不会发生由于反序列化导致的单例破坏问题。这部分内容在深度分析Java的枚举类型—-枚举的线程安全性及序列化问题(http://www.hollischuang.com/archives/197)中也有更加详细的介绍，还展示了部分代码，感兴趣的朋友可以前往阅读。

总结

在所有的单例实现方式中，枚举是一种在代码写法上最简单的方式，之所以代码十分简洁，是因为Java给我们提供了 `enum` 关键字，我们便可以很方便的声明一个枚举类型，而不需要关心其初始化过程中的线程安全问题，因为枚举类在被虚拟机加载的时候会保证线程安全的被初始化。

除此之外，在序列化方面，Java中有明确规定，枚举的序列化和反序列化是有特殊定制的。这就可以避免反序列化过程中由于反射而导致的单例被破坏问题。

> 写在前面：Java SE5提供了一种新的类型-Java的枚举类型 (/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 (/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，我们在深度分析Java的ClassLoader机制（源码级别） (/archives/199)和Java类的加载、链接和初始化 (/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。**

为什么用枚举实现的单例是最好的方式

在[转+注]单例模式的七种写法 (/archives/205)中，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（反射调用空参构造），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：

枚举对象序列化之前和反序列化之后得到的两个对象==判断，true

> 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) {//枚举反序列化使用这个方法
            T result = enumType.enumConstantDirectory().get(name);从编译器添加的final values数组中获取，Values在累加载阶段赋值，保证枚举唯一性
            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（源码级别）(/archives/199)和Java类的加载、链接和初始化(/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。**