链滴

深入理解 Java 枚举类型 (enum)

作者：zhaozhizheng
原文链接：https://ld246.com/article/1608459287060
来源网站：链滴

<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20200113.jpg?imageView2/1/w/960/h/540/interlace/1/q/100">
<img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20200821.jpg?imageView2/1/w/960/h/540/interlace/1/q/100">
理解枚举类型
枚举类型是 Java 5 中新增特性的一部分，它是一种特殊的数据类型，之所以特殊是因为它既是种类(class)类型却又比类类型多了些特殊的约束，但是这些约束的存在也造就了枚举类型的简洁性、全性以及便捷性。下面先来看看什么是枚举？如何定义枚举？
枚举的定义
<pre><code class="language-mysql highlight-chroma">public`<span><span class="highlight-w"> </span><span class="highlight-o">`class`</span><span class="highlight-w">/span><span class="highlight-o">`DayDemo<san class="highlight-w"> {`</span><span class="highlight-w"> </span><span class="highlight-o">`<spn class="highlight-w"> ``public`</span><span class="highlight-w"> </span>span class="highlight-o">`static`</span><span class="highlight-w"> </span><span class="highlight-o">`</spanfinal`</span><span class="hghlight-w"> </span><span class="highlight-o">`int</san>`</span><span class="highlight-w"> </span><span class="hghlight-o">`MONDAY =``1<spn class="highlight-o">``;`</span><span class="highlight-w"> </span><span class="highlight-o">` ``public`</span><span class="highlight-w"> </span><pan class="highlight-o">`static`</span><span class="highlight-w"> </span><span class="highlight-o">`span class="highlight-n">final`</span><span class="higlight-w"> </span><span class="highlight-o">`int</spa>`</span><span class="highlight-w"> </span><span class="higlight-o">`TUESDAY=``</span><span class="highlight-mi">2</span><span class="highlight-o">``;`</span><span class="highlight-w"></span><span class="highlight-o">` ``public`</span><span class="highlight-w"> </span><span class="highlight-o">`<spanclass="highlight-n">static`</span><span class="highlight-w"> </span><span class="highlight-o">`finalspan class="highlight-o">`</span><span class="highlight-w"> </span><span class="highlight-o">`int`</span>span class="highlight-w"> </span><span class="highlight-o">`WEDNESDAY=``3``;<spanclass="highlight-o">`</span><span class="highlight-w"> </span><span class="highlight-o>` ``<spanlass="highlight-n">public`</span><span class="highligh-w"> </span><span class="highlight-o">`staticspan class="highlight-o">`</span><span class="highlight-w"> </span><span class="highlight-o">`final`</span><span class="highlight-w"> </span><span class="highlight-o">`<span class="highl

ght-kt">int`</span><span class="highlight-w"> </span><span class="highlight-o">`THURSDAY=``</span><span class="highlight-mi">4</span><span class="highlight-o">``;`</span><span class="highlight-w"> </span><span class="highlight-o">` ``public`</span><span class="highlight-w"> </span><span class="highlight-o">`static`</span><span class="highlight-w"> </span><span class="highlight-o">`final`</span><span class="highlight-w"> </span><span class="highlight-o">`int`</span><span class="highlight-w"> </span><span class="highlight-o">`FRIDAY=``5``;`</span><span class="highlight-w"> </span><span class="highlight-o">` ``public`</span><span class="highlight-w"> </span><span class="highlight-o">`static`</span><span class="highlight-w"> </span><span class="highlight-o">`final`</span><span class="highlight-w"> </span><span class="highlight-o">`int`</span><span class="highlight-w"> </span><span class="highlight-o">`SATURDAY=``</span><span class="highlight-mi">6</span><span class="highlight-o">``;`</span><span class="highlight-w"> </span><span class="highlight-o">` ``public`</span><span class="highlight-w"> </span><span class="highlight-o">`static`</span><span class="highlight-w"> </span><span class="highlight-o">`final`</span><span class="highlight-w"> </span><span class="highlight-o">`int`</span><span class="highlight-w"> </span><span class="highlight-o">`SUNDAY=``7``;`</span><span class="highlight-w"> </span><span class="highlight-o">`}
</code></pre>
　　上述的常量定义常量的方式称为 int 枚举模式，这样的定义方式并没有什么错，但它存在许不足，如在类型安全和使用方便性上并没有多少好处，如果存在定义 int 值相同的变量，混淆的几率是很大的，编译器也不会提出任何警告，因此这种方式在枚举出现后并不提倡，现在我们利用枚举类来重新定义上述的常量，同时也感受一把枚举定义的方式，如下定义周一到周日的常量
<pre><code class="highlight-chroma">//枚举类型，使用关键字enum
enum Day {
    MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY, SUNDAY
}
</code></pre>
相当简洁，在定义枚举类型时我们使用的关键字是 enum，与 class 关键字类似，只不过前者是义枚举类型，后者是定义类类型。枚举类型 Day 中分别定义了从周一到周日的值，这里要注意，值

般是大写的字母，多个值之间以逗号分隔。同时我们应该知道的是枚举类型可以像类(class)类型一样定义为一个单独的文件，当然也可以定义在其他类内部，更重要的是枚举常量在类型安全性和便捷性很有保证，如果出现类型问题编译器也会提示我们改进，但务必记住枚举表示的类型其取值是必须有的，也就是说每个值都是可以枚举出来的，比如上述描述的一周共有七天。那么该如何使用呢？如下
</p>
<pre><code class="highlight-chroma"><span class="highlight
cl">public class EnumDemo {

   public static vo
d main(String[] args){
   //直接引用
   Day day =Da
.MONDAY;
   }

}
//定义枚举类型
enum Day {
   MONDAY, TUE
DAY, WEDNESDAY,
   THURSDAY, FR
DAY, SATURDAY, SUNDAY
}
</code></pre>
<p>就像上述代码那样，直接引用枚举的值即可，这便是枚举类型的最简单模型。</p>
<p>枚举实现的原理</p>
<p>我们大概了解了枚举类型的定义与简单使用后，现在有必要来了解一下枚举类型的基本实现原理实际上在使用关键字 enum 创建枚举类型并编译后，编译器会为我们生成一个相关的类，这个类继承 Java API 中的 java.lang.Enum 类，也就是说通过关键字 enum 创建枚举类型在编译后事实上也是个类类型而且该类继承自 java.lang.Enum 类。下面我们编译前面定义的 EnumDemo.java 并查看生的 class 文件来验证这个结论：</p>
<pre><code class="highlight-chroma"><span class="highlight
cl">javac EnumDemo.java
Day.class EnumD
mo.class EnumDemo.java
</code></pre>
<p>利用 javac 编译前面定义的 EnumDemo.java 文件后分别生成了 Day.class 和 EnumDemo.class 文件，而 Day.class 就是枚举类型，这也就验证前面所说的使用关键字 enum 定义枚举类型并编译后编译器会自动帮助我们生成一个与枚举相关的类。我们再来看看反编译 Day.class 文件：</p>
<pre><code class="highlight-chroma"><span class="highlight
cl">//反编译Day.class
final class Day ext
nds Enum
{
   //编译器为我们
加的静态的values()方法
   public static Da
[] values()
   {
   return (Day[]
$VALUES.clone();
   }
<span class="highlight-line"><span class="highlight-cl">   //编译器为我们
加的静态的valueOf()方法，注意间接调用了Enum也类的valueOf方法

```
    public static Day valueOf(String s)
    {
        return (Day)Enum.valueOf(com/zejian/enumdemo/Day, s);
    }
    //私有构造函数
    private Day(String s, int i)
    {
        super(s, i);
    }
    //前面定义的7枚举实例
    public static final Day MONDAY;
    public static final Day TUESDAY;
    public static final Day WEDNESDAY;
    public static final Day THURSDAY;
    public static final Day FRIDAY;
    public static final Day SATURDAY;
    public static final Day SUNDAY;
    private static final Day $VALUES[];

    static
    {
        //实例化枚举例
        MONDAY = new Day("MONDAY", 0);
        TUESDAY = new Day("TUESDAY", 1);
        WEDNESDAY = new Day("WEDNESDAY", 2);
        THURSDAY = new Day("THURSDAY", 3);
        FRIDAY = new Day("FRIDAY", 4);
        SATURDAY = new Day("SATURDAY", 5);
        SUNDAY = new Day("SUNDAY", 6);
        $VALUES = (new Day[] {
            MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
        });
```

    }
}
</code></pre>
从反编译的代码可以看出编译器确实帮助我们生成了一个 Day 类(注意该类是 final 类型的，将法被继承)而且该类继承自 java.lang.Enum 类，该类是一个抽象类(稍后我们会分析该类中的主要方法，除此之外，编译器还帮助我们生成了 7 个 Day 类型的实例对象分别对应枚举中定义的 7 个日期，也充分说明了我们前面使用关键字 enum 定义的 Day 类型中的每种日期枚举常量也是实实在在的 Day 实例对象，只不过代表的内容不一样而已。注意编译器还为我们生成了两个静态方法，分别是 values(和 valueOf()，稍后会分析它们的用法，到此我们也就明白了，使用关键字 enum 定义的枚举类型，编译期后，也将转换成为一个实实在在的类，而在该类中，会存在每个在枚举类型中定义好变量的对实例对象，如上述的 MONDAY 枚举类型对应 <code>public static final Day MONDAY;</code> 同时编译器会为该类创建两个方法，分别是 values()和 valueOf()。ok~，到此相信我们对枚举的实原理也比较清晰，下面我们深入了解一下 java.lang.Enum 类以及 values()和 valueOf()的用途。
枚举的常用方法
Enum 抽象类常用方法
Enum 是所有 Java 语言枚举类型的公共基本类（注意 Enum 是抽象类），以下是它常见方法：
<table>
<thead>
<tr>
<th>返回类型</th>
<th>方法名称</th>
<th>方法说明</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int</code></td>
<td><code>compareTo(E o)</code></td>
<td>比较此枚举与指定对象的顺序</td>
</tr>
<tr>
<td><code>boolean</code></td>
<td><code>equals(Object other)</code></td>
<td>当指定对象等于此枚举常量时，返回 true。</td>
</tr>
<tr>
<td><code>Class</code></td>
<td><code>getDeclaringClass()</code></td>
<td>返回与此枚举常量的枚举类型相对应的 Class 对象</td>
</tr>
<tr>
<td><code>String</code></td>
<td><code>name()</code></td>
<td>返回此枚举常量的名称，在其枚举声明中对其进行声明</td>
</tr>
<tr>
<td><code>int</code></td>
<td><code>ordinal()</code></td>
<td>返回枚举常量的序数（它在枚举声明中的位置，其中初始常量序数为零）</td>
</tr>
<tr>
<td><code>String</code></td>
<td><code>toString()</code></td>

<td>返回枚举常量的名称，它包含在声明中</td>
</tr>
<tr>
<td><code>static> T</code></td>
<td><code>static valueOf(Class enumType, String name)</code></td>
<td><code>返回带指定名称的指定枚举类型的枚举常量。</code></td>
</tr>
</tbody>
</table>
这里主要说明一下 <code>ordinal()</code> 方法，该方法获取的是枚举变量在枚举类中声明顺序，下标从 0 开始，如日期中的 MONDAY 在第一个位置，那么 MONDAY 的 ordinal 值就是 0 如果 MONDAY 的声明位置发生变化，那么 ordinal 方法获取到的值也随之变化，注意在大多数情况我们都不应该首先使用该方法，毕竟它总是变幻莫测的。<code>compareTo(E o)</code> 方法则比较枚举的大小，注意其内部实现是根据每个枚举的 ordinal 值大小进行比较的。<code>name()</ode> 方法与 <code>toString()</code> 几乎是等同的，都是输出变量的字符串形式。至于 <code valueOf(Class<T> enumType, String name)</code> 方法则是根据枚举类的 Class 对象和举名称获取枚举常量，注意该方法是静态的，后面在枚举单例时，我们还会详细分析该方法，下面的码演示了上述方法：
<pre><code class="highlight-chroma">public class EnumDemo {

    public static void main(String[] args){

        //创建枚举数

        Day[] days=new Day[]{Day.MONDAY, Day.TUESDAY, Day.WEDNESDAY,
                Day.THRSDAY, Day.FRIDAY, Day.SATURDAY, Day.SUNDAY};

        for (int i = 0; i <days.length ; i++) {
            System.out.println("day["+i+"].ordinal():"+days[i].ordinal());
        }

        System.out.println("-----------------------------------");
        //通过compareTo方法比较,实际上其内部是通过ordinal()值比较的
        System.out.println("days[0].compareTo(days[1]):"+days[0].compareTo(days[1]));
        System.out.println("days[0].compareTo(days[1]):"+days[0].compareTo(days[2]));

        //获取该枚举象的Class对象引用,当然也可以通过getClass方法
        Class<?&gt clazz = days[0].getDeclaringClass();
        System.out.println("clazz:"+clazz);

        System.out.pr

```
ntln("-----------------------------------");
        //name()
        System.out.println("days[0].name():"+days[0].name());
        System.out.println("days[1].name():"+days[1].name());
        System.out.println("days[2].name():"+days[2].name());
        System.out.println("days[3].name():"+days[3].name());

        System.out.println("-----------------------------------");

        System.out.println("days[0].toString():"+days[0].toString());
        System.out.println("days[1].toString():"+days[1].toString());
        System.out.println("days[2].toString():"+days[2].toString());
        System.out.println("days[3].toString():"+days[3].toString());

        System.out.println("-----------------------------------");

        Day d=Enum.valueOf(Day.class,days[0].name());
        Day d2=Day.alueOf(Day.class,days[0].name());
        System.out.println("d:"+d);
        System.out.println("d2:"+d2);
    }
/**
执行结果:
   day[0].ordinal():

   day[1].ordinal():

   day[2].ordinal():

   day[3].ordinal():

   day[4].ordinal():

   day[5].ordinal():

   day[6].ordinal():

   ----------
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">    days[0].compar
To(days[1]):-1
</span></span><span class="highlight-line"><span class="highlight-cl">    days[0].compar
To(days[1]):-2
</span></span><span class="highlight-line"><span class="highlight-cl">    clazz:class com.
ejian.enumdemo.Day
</span></span><span class="highlight-line"><span class="highlight-cl">    ----------------
----------------
</span></span><span class="highlight-line"><span class="highlight-cl">    days[0].name():
ONDAY
</span></span><span class="highlight-line"><span class="highlight-cl">    days[1].name():
UESDAY
</span></span><span class="highlight-line"><span class="highlight-cl">    days[2].name():
EDNESDAY
</span></span><span class="highlight-line"><span class="highlight-cl">    days[3].name():
HURSDAY
</span></span><span class="highlight-line"><span class="highlight-cl">    ----------------
----------------
</span></span><span class="highlight-line"><span class="highlight-cl">    days[0].toString(
:MONDAY
</span></span><span class="highlight-line"><span class="highlight-cl">    days[1].toString(
:TUESDAY
</span></span><span class="highlight-line"><span class="highlight-cl">    days[2].toString(
:WEDNESDAY
</span></span><span class="highlight-line"><span class="highlight-cl">    days[3].toString(
:THURSDAY
</span></span><span class="highlight-line"><span class="highlight-cl">    ----------------
----------------
</span></span><span class="highlight-line"><span class="highlight-cl">    d:MONDAY
</span></span><span class="highlight-line"><span class="highlight-cl">    d2:MONDAY
</span></span><span class="highlight-line"><span class="highlight-cl">    */
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">enum Day {
</span></span><span class="highlight-line"><span class="highlight-cl">    MONDAY, TUE
DAY, WEDNESDAY,
</span></span><span class="highlight-line"><span class="highlight-cl">    THURSDAY, FR
DAY, SATURDAY, SUNDAY
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p>到此对于抽象类 Enum 类的基本内容就介绍完了，这里提醒大家一点，Enum 类内部会有一个造函数，该构造函数只能有编译器调用，我们是无法手动操作的，不妨看看 Enum 类的主要源码：</
>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">//实现了Comparable
</span></span><span class="highlight-line"><span class="highlight-cl">public abstract cla
s Enum&amp;lt;E extends Enum&amp;lt;E&amp;gt;&amp;gt;
</span></span><span class="highlight-line"><span class="highlight-cl">        implements
omparable&amp;lt;E&amp;gt;, Serializable {
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    private final Str
ng name; //枚举字符串名称
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    public final Stri
```

```
g name() {
        return name;
    }

    private final int ordinal;//枚举顺序值

    public final int rdinal() {
        return ordinal

    }

    //枚举的构造方，只能由编译器调用
    protected Enum String name, int ordinal) {
        this.name = ame;
        this.ordinal = ordinal;
    }

    public String to tring() {
        return name;
    }

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

    //比较的是ordin l值
    public final int ompareTo(E o) {
        Enum&lt;?&g ; other = (Enum&lt;?&gt;)o;
        Enum&lt;E&g ; self = this;
        if (self.getCla s() != other.getClass() &amp;&amp; // optimization
            self.getDec aringClass() != other.getDeclaringClass())
            throw new ClassCastException();
        return self.or inal - other.ordinal;//根据ordinal值比较大小
    }

    @SuppressWar ings("unchecked")
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">    public final Clas
&amp;lt;E&amp;gt; getDeclaringClass() {
</span></span><span class="highlight-line"><span class="highlight-cl">        //获取class
象引用，getClass()是Object的方法
</span></span><span class="highlight-line"><span class="highlight-cl">        Class&amp;lt;?&amp;gt
 clazz = getClass();
</span></span><span class="highlight-line"><span class="highlight-cl">        //获取父类Cla
s对象引用
</span></span><span class="highlight-line"><span class="highlight-cl">        Class&amp;lt;?&amp;gt
 zuper = clazz.getSuperclass();
</span></span><span class="highlight-line"><span class="highlight-cl">        return (zuper
== Enum.class) ? (Class&amp;lt;E&amp;gt;)clazz : (Class&amp;lt;E&amp;gt;)zuper;
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    public static &amp;lt
T extends Enum&amp;lt;T&amp;gt;&amp;gt; T valueOf(Class&amp;lt;T&amp;gt; enumType,
</span></span><span class="highlight-line"><span class="highlight-cl">
            String name) {
</span></span><span class="highlight-line"><span class="highlight-cl">        //enumType.
numConstantDirectory()获取到的是一个map集合，key值就是name值，value则是枚举变量值
</span></span><span class="highlight-line"><span class="highlight-cl">        //enumConst
ntDirectory是class对象内部的方法，根据class对象获取一个map集合的值
</span></span><span class="highlight-line"><span class="highlight-cl">        T result = en
mType.enumConstantDirectory().get(name);
</span></span><span class="highlight-line"><span class="highlight-cl">        if (result != n
ll)
</span></span><span class="highlight-line"><span class="highlight-cl">            return resu
t;
</span></span><span class="highlight-line"><span class="highlight-cl">        if (name ==
ull)
</span></span><span class="highlight-line"><span class="highlight-cl">            throw new
NullPointerException("Name is null");
</span></span><span class="highlight-line"><span class="highlight-cl">        throw new Ill
galArgumentException(
</span></span><span class="highlight-line"><span class="highlight-cl">            "No enum
onstant " + enumType.getCanonicalName() + "." + name);
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    //.....省略其他没
的方法
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p>通过 Enum 源码，可以知道，Enum 实现了 Comparable 接口，这也是可以使用 compareTo
较的原因，当然 Enum 构造函数也是存在的，该函数只能由编译器调用，毕竟我们只能使用 enum
键字定义枚举，其他事情就放心交给编译器吧。</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">//由编译器调用
</span></span><span class="highlight-line"><span class="highlight-cl">protected Enum(S
ring name, int ordinal) {
</span></span><span class="highlight-line"><span class="highlight-cl">        this.name =
ame;
</span></span><span class="highlight-line"><span class="highlight-cl">        this.ordinal =
ordinal;
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span></code></pre>
```

编译器生成的 Values 方法与 ValueOf 方法

values()方法和 valueOf(String name)方法是编译器生成的 static 方法，因此从前面的分析中，Enum 类中并没出现 values()方法，但 valueOf()方法还是有出现的，只不过编译器生成的 valueOf()方法需传递一个 name 参数，而 Enum 自带的静态方法 valueOf()则需要传递两个方法，从前面反编后的代码可以看出，编译器生成的 valueOf 方法最终还是调用了 Enum 类的 valueOf 方法，下面通代码来演示这两个方法的作用：

```
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">Day[] days2 = Day.values();
</span></span><span class="highlight-line"><span class="highlight-cl">System.out.println
"day2:"+Arrays.toString(days2));
</span></span><span class="highlight-line"><span class="highlight-cl">Day day = Day.val
eOf("MONDAY");
</span></span><span class="highlight-line"><span class="highlight-cl">System.out.println
"day:"+day);
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">/**
</span></span><span class="highlight-line"><span class="highlight-cl"> 输出结果:
</span></span><span class="highlight-line"><span class="highlight-cl"> day2:[MONDAY,
UESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY]
</span></span><span class="highlight-line"><span class="highlight-cl"> day:MONDAY
</span></span><span class="highlight-line"><span class="highlight-cl"> */
</span></span></code></pre>
```

从结果可知道，values()方法的作用就是获取枚举类中的所有变量，并作为数组返回，而 valueOf String name)方法与 Enum 类中的 valueOf 方法的作用类似根据名称获取枚举变量，只不过编译器成的 valueOf 方法更简洁些只需传递一个参数。这里我们还必须注意到，由于 values()方法是由编译插入到枚举类中的 static 方法，所以如果我们将枚举实例向上转型为 Enum，那么 values()方法将无被调用，因为 Enum 类中并没有 values()方法，valueOf()方法也是同样的道理，注意是一个参数的。/p>

```
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">//正常使用
</span></span><span class="highlight-line"><span class="highlight-cl">Day[] ds=Day.valu
s();
</span></span><span class="highlight-line"><span class="highlight-cl">//向上转型Enum
</span></span><span class="highlight-line"><span class="highlight-cl">Enum e = Day.M
NDAY;
</span></span><span class="highlight-line"><span class="highlight-cl">//无法调用,没有此
法
</span></span><span class="highlight-line"><span class="highlight-cl">//e.values();
</span></span></code></pre>
```

枚举与 Class 对象

上述我们提到当枚举实例向上转型为 Enum 类型后，values()方法将会失效，也就无法一次性获所有枚举实例变量，但是由于 Class 对象的存在，即使不使用 values()方法，还是有可能一次获取到有枚举实例变量的，在 Class 对象中存在如下方法：

```
<table>
<thead>
<tr>
<th>返回类型</th>
<th>方法名称</th>
<th>方法说明</th>
</tr>
</thead>
```

<tbody>
<tr>
<td><code>T[]</code></td>
<td><code>getEnumConstants()</code></td>
<td>返回该枚举类型的所有元素，如果 Class 对象不是枚举类型，则返回 null。</td>
</tr>
<tr>
<td><code>boolean</code></td>
<td><code>isEnum()</code></td>
<td>当且仅当该类声明为源代码中的枚举时返回 true</td>
</tr>
</tbody>
</table>
因此通过 getEnumConstants()方法，同样可以轻而易举地获取所有枚举实例变量下面通过代码演示这个功能：
<pre><code class="highlight-chroma">//正常使用
Day[] ds=Day.valu s();
//向上转型Enum
Enum e = Day.M NDAY;
//无法调用,没有此法
//e.values();
//获取class对象引

Class<?> cla z = e.getDeclaringClass();
if(clasz.isEnum()) {
    Day[] dsz = (Da []) clasz.getEnumConstants();
    System.out.print n("dsz:"+Arrays.toString(dsz));
}

/**
   输出结果:
   dsz:[MONDAY, UESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY]
 */
</code></pre>
正如上述代码所展示，通过 Enum 的 class 对象的 getEnumConstants 方法，我们仍能一次性取所有的枚举实例常量。
<h2 id="枚举的进阶用法">枚举的进阶用法</h2>
在前面的分析中，我们都是基于简单枚举类型的定义，也就是在定义枚举时只定义了枚举实例类，并没定义方法或者成员变量，实际上使用关键字 enum 定义的枚举类，除了不能使用继承(因为编器会自动为我们继承 Enum 抽象类而 Java 只支持单继承，因此枚举类是无法手动实现继承的)，可以 enum 类当成常规类，也就是说我们可以向 enum 类中添加方法和变量，甚至是 mian 方法，下面来感受一把。
<h2 id="向enum类添加方法与自定义构造函数">向 enum 类添加方法与自定义构造函数</h2>
重新定义一个日期枚举类，带有 desc 成员变量描述该日期的对于中文描述，同时定义一个 getDsc 方法，返回中文描述内容，自定义私有构造函数，在声明枚举实例时传入对应的中文描述，代码如：

public enum Day2 {
MONDAY("星期一"),
TUESDAY("星期二"),
WEDNESDAY("星期三"),
THURSDAY("星期四"),
FRIDAY("星期五"),
SATURDAY("星期六"),
SUNDAY("星期日");//记住要用分号结束
private String desc;//中文描述
/**
* 私有构造,防止被外部调用
* @param desc
*/
private Day2(String desc){
this.desc=desc;
}
/**
* 定义方法,返回描述,跟常规类的定义没区别
* @return
*/
public String getDesc(){
return desc;
}
public static void main(String[] args){
for (Day2 day:Day2.values()) {
System.out.println("name:"+day.name()+
",desc:"+day.getDesc());
}
}
/**
输出结果:
name:MONDAY,desc:星期一
name:TUESDAY,desc:星期二
name:WEDNESDAY,desc:星期三
name:THURSDAY,desc:星期四
name:FRIDAY,desc:星期五
name:SATURDAY,desc:星期六
name:SUNDAY,desc:星期日
*/
}
从上述代码可知，在 enum 类中确实可以像定义常规类一样声明变量或者成员方法。但是我们须注意到，如果打算在 enum 类中定义方法，务必在声明完枚举实例后使用分号分开，倘若在枚举实前定义任何方法，编译器都将会报错，无法编译通过，同时即使自定义了构造函数且 enum 的定义结，我们也永远无法手动调用构造函数创建枚举实例，毕竟这事只能由编译器执行。
<h2 id="关于覆盖enum类方法">关于覆盖 enum 类方法</h2>
既然 enum 类跟常规类的定义没什么区别（实际上 enum 还是有些约束的），那么覆盖父类的法也不会是什么难说，可惜的是父类 Enum 中的定义的方法只有 toString 方法没有使用 final 修饰因此只能覆盖 toString 方法，如下通过覆盖 toString 省去了 getDesc 方法：
<pre><code class="highlight-chroma">public enum Day2 {
    MONDAY("星一"),
    TUESDAY("星二"),

```
    WEDNESDAY(星期三"),
    THURSDAY("期四"),
    FRIDAY("星期五",
    SATURDAY("星六"),
    SUNDAY("星期");//记住要用分号结束

    private String desc;//中文描述

    /**
     * 私有构造,防被外部调用
     * @param desc
     */
    private Day2(String desc){
        this.desc=desc;
    }

    /**
     * 覆盖
     * @return
     */
    @Override
    public String toString() {
        return desc;
    }


    public static void main(String[] args){
        for (Day2 day:Day2.values()) {
            System.out.println("name:"+day.name()+
                    ",desc:"+day.toString());
        }
    }

    /**
     输出结果:
     name:MONDAY,desc:星期一
     name:TUESDAY,desc:星期二
     name:WEDNE
```

```
DAY,desc:星期三
</span></span><span class="highlight-line"><span class="highlight-cl">    name:THURSD
Y,desc:星期四
</span></span><span class="highlight-line"><span class="highlight-cl">    name:FRIDAY,
esc:星期五
</span></span><span class="highlight-line"><span class="highlight-cl">    name:SATURD
Y,desc:星期六
</span></span><span class="highlight-line"><span class="highlight-cl">    name:SUNDAY
desc:星期日
</span></span><span class="highlight-line"><span class="highlight-cl">    */
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<h2 id="enum类中定义抽象方法">enum 类中定义抽象方法</h2>
<p>与常规抽象类一样，enum 类允许我们为其定义抽象方法，然后使每个枚举实例都实现该方法，便产生不同的行为方式，注意 abstract 关键字对于枚举类来说并不是必须的如下：</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public enum EnumDemo3 {
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    FIRST{
</span></span><span class="highlight-line"><span class="highlight-cl">        @Override
</span></span><span class="highlight-line"><span class="highlight-cl">        public String
getInfo() {
</span></span><span class="highlight-line"><span class="highlight-cl">            return "FIR
T TIME";
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    },
</span></span><span class="highlight-line"><span class="highlight-cl">    SECOND{
</span></span><span class="highlight-line"><span class="highlight-cl">        @Override
</span></span><span class="highlight-line"><span class="highlight-cl">        public String
getInfo() {
</span></span><span class="highlight-line"><span class="highlight-cl">            return "SE
OND TIME";
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    ;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    /**
</span></span><span class="highlight-line"><span class="highlight-cl">     * 定义抽象方法
</span></span><span class="highlight-line"><span class="highlight-cl">     * @return
</span></span><span class="highlight-line"><span class="highlight-cl">     */
</span></span><span class="highlight-line"><span class="highlight-cl">    public abstract
tring getInfo();
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    //测试
</span></span><span class="highlight-line"><span class="highlight-cl">    public static vo
d main(String[] args){
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("F:"+EnumDemo3.FIRST.getInfo());
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("S:"+EnumDemo3.SECOND.getInfo());
</span></span><span class="highlight-line"><span class="highlight-cl">        /**
</span></span><span class="highlight-line"><span class="highlight-cl">         输出结果:
</span></span><span class="highlight-line"><span class="highlight-cl">         F:FIRST TIME
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">        S:SECOND TME
</span></span><span class="highlight-line"><span class="highlight-cl">        */
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p>通过这种方式就可以轻而易举地定义每个枚举实例的不同行为方式。我们可能注意到，enum 类实例似乎表现出了多态的特性，可惜的是枚举类型的实例终究不能作为类型传递使用，就像下面的使方式，编译器是不可能答应的：</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl">//无法通过编译,毕竟EnumDemo3.FIRST是个实例对象
</span></span><span class="highlight-line"><span class="highlight-cl"> public void text(EumDemo3.FIRST instance){ }
</span></span></code></pre>
<p>在枚举实例常量中定义抽象方法</p>
<h2 id="enum类与接口">enum 类与接口</h2>
<p>由于 Java 单继承的原因，enum 类并不能再继承其它类，但并不妨碍它实现接口，因此 enum 同样是可以实现多接口的，如下：</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl">interface food{
</span></span><span class="highlight-line"><span class="highlight-cl">    void eat();
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">interface sport{
</span></span><span class="highlight-line"><span class="highlight-cl">    void run();
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">public enum EnuDemo2 implements food ,sport{
</span></span><span class="highlight-line"><span class="highlight-cl">    FOOD,
</span></span><span class="highlight-line"><span class="highlight-cl">    SPORT,
</span></span><span class="highlight-line"><span class="highlight-cl">    ; //分号分隔
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    @Override
</span></span><span class="highlight-line"><span class="highlight-cl">    public void eat(){
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.println("eat.....");
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    @Override
</span></span><span class="highlight-line"><span class="highlight-cl">    public void run(){
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.println("run.....");
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p>有时候，我们可能需要对一组数据进行分类，比如进行食物菜单分类而且希望这些菜单都属于 food 类型，appetizer(开胃菜)、mainCourse(主菜)、dessert(点心)、Coffee 等，每种分类下有多种具的菜式或食品，此时可以利用接口来组织，如下(代码引用自 Thinking in Java)：</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl">public interface Food {
</span></span><span class="highlight-line"><span class="highlight-cl">  enum Appetizer
```

```
mplements Food {
    SALAD, SOUP,
PRING_ROLLS;
  }
  enum MainCour
e implements Food {
    LASAGNE, BUR
ITO, PAD_THAI,
    LENTILS, HUM
OUS, VINDALOO;
  }
  enum Dessert i
plements Food {
    TIRAMISU, GEL
TO, BLACK_FOREST_CAKE,
    FRUIT, CREME_
ARAMEL;
  }
  enum Coffee imp
ements Food {
    BLACK_COFFEE,
DECAF_COFFEE, ESPRESSO,
    LATTE, CAPPU
CINO, TEA, HERB_TEA;
  }
}

public class TypeO
Food {
  public static void
main(String[] args) {
    Food food = A
petizer.SALAD;
    food = MainCo
rse.LASAGNE;
    food = Dessert
GELATO;
    food = Coffee.
APPUCCINO;
  }
}
```

通过这种方式可以很方便组织上述的情景，同时确保每种具体类型的食物也属于 Food，现在我利用一个枚举嵌套枚举的方式，把前面定义的菜谱存放到一个 Meal 菜单中，通过这种方式就可以统管理菜单的数据了。

```
public enum Meal{
  APPETIZER(Food
Appetizer.class),
  MAINCOURSE(F
od.MainCourse.class),
  DESSERT(Food.D
ssert.class),
  COFFEE(Food.Cof
```

```
ee.class);
  private Food[] values;
  private Meal(Class<? extends Food> kind) {
    //通过class对象取枚举实例
    values = kind.getEnumConstants();
  }
  public interface Food {
    enum Appetizer implements Food {
      SALAD, SOUP, SPRING_ROLLS;
    }
    enum MainCourse implements Food {
      LASAGNE, BURRITO, PAD_THAI,
      LENTILS, HUMOUS, VINDALOO;
    }
    enum Dessert implements Food {
      TIRAMISU, GELATO, BLACK_FOREST_CAKE,
      FRUIT, CREME_CARAMEL;
    }
    enum Coffee implements Food {
      BLACK_COFFEE, DECAF_COFFEE, ESPRESSO,
      LATTE, CAPPUCINO, TEA, HERB_TEA;
    }
  }
}
```

枚举与switch

关于枚举与 switch 是个比较简单的话题，使用 switch 进行条件判断时，条件参数一般只能是整型，字符型。而枚举型确实也被 switch 所支持，在 java 1.7 后 switch 也对字符串进行了支持。这里我们简单看一下 switch 与枚举类型的使用：

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

public class Enumemo4 {

    public static void printName(Color color){
        switch (color)
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">            case BLUE:
//无需使用Color进行引用
</span></span><span class="highlight-line"><span class="highlight-cl">                System.
ut.println("蓝色");
</span></span><span class="highlight-line"><span class="highlight-cl">                break;
</span></span><span class="highlight-line"><span class="highlight-cl">            case RED:
</span></span><span class="highlight-line"><span class="highlight-cl">                System.
ut.println("红色");
</span></span><span class="highlight-line"><span class="highlight-cl">                break;
</span></span><span class="highlight-line"><span class="highlight-cl">            case GREE
:
</span></span><span class="highlight-line"><span class="highlight-cl">                System.
ut.println("绿色");
</span></span><span class="highlight-line"><span class="highlight-cl">                break;
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    public static vo
d main(String[] args){
</span></span><span class="highlight-line"><span class="highlight-cl">        printName(C
lor.BLUE);
</span></span><span class="highlight-line"><span class="highlight-cl">        printName(C
lor.RED);
</span></span><span class="highlight-line"><span class="highlight-cl">        printName(C
lor.GREEN);
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        //蓝色
</span></span><span class="highlight-line"><span class="highlight-cl">        //红色
</span></span><span class="highlight-line"><span class="highlight-cl">        //绿色
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p>需要注意的是使用在于 switch 条件进行结合使用时，无需使用 Color 引用。</p>
<h2 id="枚举与单例模式">枚举与单例模式</h2>
<p>单例模式可以说是最常使用的设计模式了，它的作用是确保某个类只有一个实例，自行实例化并整个系统提供这个实例。在实际应用中，线程池、缓存、日志对象、对话框对象常被设计成单例，总，选择单例模式就是为了避免不一致状态，下面我们将会简单说明单例模式的几种主要编写方式，从对比出使用枚举实现单例模式的优点。首先看看饿汉式的单例模式：</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">/**
</span></span><span class="highlight-line"><span class="highlight-cl"> * 饿汉式（基于cla
sloder机制避免了多线程的同步问题）
</span></span><span class="highlight-line"><span class="highlight-cl"> */
</span></span><span class="highlight-line"><span class="highlight-cl">public class Single
onHungry {
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    private static Si
gletonHungry instance = new SingletonHungry();
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    private Singlet
nHungry() {
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">    public static SingletonHungry getInstance() {
</span></span><span class="highlight-line"><span class="highlight-cl">        return instance;
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p>显然这种写法比较简单，但问题是无法做到延迟创建对象，事实上如果该单例类涉及资源较多，建比较耗时间时，我们更希望它可以尽可能地延迟加载，从而减小初始化的负载，于是便有了如下的汉式单例：</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl">/**
</span></span><span class="highlight-line"><span class="highlight-cl"> * Created by wuzexian on 2017/5/9..
</span></span><span class="highlight-line"><span class="highlight-cl"> * 懒汉式单例模式适合多线程安全)
</span></span><span class="highlight-line"><span class="highlight-cl"> */
</span></span><span class="highlight-line"><span class="highlight-cl">public class SingleonLazy {
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    private static volatile SingletonLazy instance;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    private SingletonLazy() {
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    public static synchronized SingletonLazy getInstance() {
</span></span><span class="highlight-line"><span class="highlight-cl">        if (instance = null) {
</span></span><span class="highlight-line"><span class="highlight-cl">            instance = new SingletonLazy();
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">        return instance;
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p>这种写法能够在多线程中很好的工作避免同步问题，同时也具备 lazy loading 机制，遗憾的是由于 synchronized 的存在，效率很低，在单线程的情景下，完全可以去掉 synchronized，为了兼效率与性能问题，改进后代码如下：</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight-cl">public class Singleton {
</span></span><span class="highlight-line"><span class="highlight-cl">    private static volatile Singleton singleton = null;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    private Singleton(){}
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    public static Singleton getSingleton(){
</span></span><span class="highlight-line"><span class="highlight-cl">        if(singleton = null){
```

            synchronized (Singleton.class){
                if(singleton == null){
                    singleton = new Singleton();
                }
            }
        }
        return singleton;
    }
}
</code></pre>
这种编写方式被称为“双重检查锁”，主要在 getSingleton()方法中，进行两次 null 查。这样可以极大提升并发度，进而提升性能。毕竟在单例中 new 的情况非常少，绝大多数都是可并行的读操作，因此在加锁前多进行一次 null 检查就可以减少绝大多数的加锁操作，也就提高了执行率。但是必须注意的是 volatile 关键字，该关键字有两层语义。第一层语义是可见性，可见性是指在个线程中对该变量的修改会马上由工作内存（Work Memory）写回主内存（Main Memory），所其它线程会马上读取到已修改的值，关于工作内存和主内存可简单理解为高速缓存（直接与 CPU 打道）和主存（日常所说的内存条），注意工作内存是线程独享的，主存是线程共享的。volatile 的第层语义是禁止指令重排序优化，我们写的代码（特别是多线程代码），由于编译器优化，在实际执行时候可能与我们编写的顺序不同。编译器只保证程序执行结果与源代码相同，却不保证实际指令的顺与源代码相同，这在单线程并没什么问题，然而一旦引入多线程环境，这种乱序就可能导致严重问题volatile 关键字就可以从语义上解决这个问题，值得关注的是 volatile 的禁止指令重排序优化功能在 Java 1.5 后才得以实现，因此 1.5 前的版本仍然是不安全的，即使使用了 volatile 关键字。或许我们可利用静态内部类来实现更安全的机制，静态内部类单例模式如下：
<pre><code class="highlight-chroma">/**
 * Created by wuzejian on 2017/5/9.
 * 静态内部类
 */
public class SingletonInner {
    private static class Holder {
        private static SingletonInner singleton = new SingletonInner();
    }

    private SingletonInner(){}

    public static SingletonInner getSingleton(){
        return Holder.singleton;
    }
}
</code></pre>
正如上述代码所展示的，我们把 Singleton 实例放到一个静态内部类中，这样可以避免了静态实在 Singleton 类的加载阶段（类加载过程的其中一个阶段的，此时只创建了 Class 对象，关于 Class 象可以看博主另外一篇博文，<a href="https://ld246.com/forward?goto=http%3A%2F%2Fblog.

sdn.net%2Fjavazejian%2Farticle%2Fdetails%2F70768369" target="_blank" rel="nofollow ugc"
 深入理解 Java 类型信息(Class 对象)与反射机制）就创建对象，毕竟静态变量初始化是在 Single
onInner 类初始化时触发的，并且由于静态内部类只会被加载一次，所以这种写法也是线程安全的。
上述 4 种单例模式的写法中，似乎也解决了效率与懒加载的问题，但是它们都有两个共同的缺点：</
>
<p>序列化可能会破坏单例模式，比较每次反序列化一个序列化的对象实例时都会创建一个新的实例
解决方案如下：</p>
<pre><code class="highlight-chroma"><span class="highlight
cl">//测试例子(四种写解决方式雷同)
public class Single
on implements java.io.Serializable {
   public static Sin
leton INSTANCE = new Singleton();

   protected Single
on() {
   }

   //反序列时直接
回当前INSTANCE
   private Object r
adResolve() {
   return INS
ANCE;
   }
}
</code></pre>
<p>使用反射强行调用私有构造器，解决方式可以修改构造器，让它在创建第二个实例的时候抛异常
如下：</p>
<pre><code class="highlight-chroma"><span class="highlight
cl">public static Singleton INSTANCE = new Singleton();
private static volat
le boolean flag = true;
private Singleton()

   if(flag){
   flag = false;
   }else{
   throw new R
ntimeException("The instance already exists ！ ");
   }
}
</code></pre>
<p>如上所述，问题确实也得到了解决，但问题是我们为此付出了不少努力，即添加了不少代码，还
该注意到如果单例类维持了其他对象的状态时还需要使他们成为 transient 的对象，这种就更复杂了
那有没有更简单更高效的呢？当然是有的，那就是枚举单例了，先来看看如何实现：</p>
<pre><code class="highlight-chroma"><span class="highlight
cl">/**
 * 枚举单利
 */
public enum Sing
etonEnum {
   INSTANCE;
<span class="highlight-line"><span class="highlight-cl">   private String n

me;
    public String ge
Name(){
        return name;
    }
    public void set
ame(String name){
        this.name =
ame;
    }
}
</code></pre>
代码相当简洁，我们也可以像常规类一样编写 enum 类，为其添加变量和方法，访问方式也更单，使用 <code>SingletonEnum.INSTANCE</code> 进行访问，这样也就避免调用 getInstance 法，更重要的是使用枚举单例的写法，我们完全不用考虑序列化和反射的问题。枚举序列化是由 jvm 证的，每一个枚举类型和定义的枚举变量在 JVM 中都是唯一的，在枚举类型的序列化和反序列化上，ava 做了特殊的规定：在序列化时 Java 仅仅是将枚举对象的 name 属性输出到结果中，反序列化的候则是通过 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这种序列化机制的定制的并禁用了 writeObject、readObject、readObjectNoData、writeReplace 和 readResolve 等方法，从而保证了枚举实例的唯一性，这里我们不妨再次看看 Enum 类的 valueOf 方法：
<pre><code class="highlight-chroma"><span class="highlight
cl">public static <T extends Enum<T>> T valueOf(Class<T> enumType,

            String name) {
        T result = en
mType.enumConstantDirectory().get(name);
        if (result != nul
)
            return result

        if (name == nu
l)
            throw new
ullPointerException("Name is null");
        throw new Ille
alArgumentException(
            "No enum
onstant " + enumType.getCanonicalName() + "." + name);
    }
</code></pre>
实际上通过调用 enumType(Class 对象的引用)的 enumConstantDirectory 方法获取到的是一个 Map 集合，在该集合中存放了以枚举 name 为 key 和以枚举实例变量为 value 的 Key&Value 据，因此通过 name 的值就可以获取到枚举实例，看看 enumConstantDirectory 方法源码：
<pre><code class="highlight-chroma"><span class="highlight
cl">Map<String, T> enumConstantDirectory() {
        if (enumCons
antDirectory == null) {
            //getEnu
ConstantsShared最终通过反射调用枚举类的values方法
            T[] univers
 = getEnumConstantsShared();
            if (universe
== null)

```
</span></span><span class="highlight-line"><span class="highlight-cl">            throw n
w IllegalArgumentException(
</span></span><span class="highlight-line"><span class="highlight-cl">                getN
me() + " is not an enum type");
</span></span><span class="highlight-line"><span class="highlight-cl">        Map&amp;lt;Str
ng, T&amp;gt; m = new HashMap&amp;lt;&amp;gt;(2 * universe.length);
</span></span><span class="highlight-line"><span class="highlight-cl">        //map存
了当前enum类的所有枚举实例变量，以name为key值
</span></span><span class="highlight-line"><span class="highlight-cl">        for (T cons
ant : universe)
</span></span><span class="highlight-line"><span class="highlight-cl">            m.put(((
num&amp;lt;?&amp;gt;)constant).name(), constant);
</span></span><span class="highlight-line"><span class="highlight-cl">        enumCons
antDirectory = m;
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">    return enum
onstantDirectory;
</span></span><span class="highlight-line"><span class="highlight-cl">  }
</span></span><span class="highlight-line"><span class="highlight-cl">  private volatile t
ansient Map&amp;lt;String, T&amp;gt; enumConstantDirectory = null;
</span></span></code></pre>
<p>到这里我们也就可以看出枚举序列化确实不会重新创建新实例，jvm 保证了每个枚举实例变量的一性。再来看看反射到底能不能创建枚举，下面试图通过反射获取构造器并创建枚举</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public static void main(String[] args) throws IllegalAccessException, InvocationTargetExce
tion, InstantiationException, NoSuchMethodException {
</span></span><span class="highlight-line"><span class="highlight-cl">  //获取枚举类的
造函数(前面的源码已分析过)
</span></span><span class="highlight-line"><span class="highlight-cl">  Constructor&amp;lt;S
ngletonEnum&amp;gt; constructor=SingletonEnum.class.getDeclaredConstructor(String.class,int.cl
ss);
</span></span><span class="highlight-line"><span class="highlight-cl">  constructor.setA
cessible(true);
</span></span><span class="highlight-line"><span class="highlight-cl">  //创建枚举
</span></span><span class="highlight-line"><span class="highlight-cl">  SingletonEnum s
ngleton=constructor.newInstance("otherInstance",9);
</span></span><span class="highlight-line"><span class="highlight-cl">  }
</span></span></code></pre>
<p>执行报错</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">Exception in thread "main" java.lang.IllegalArgumentException: Cannot reflectively create
num objects
</span></span><span class="highlight-line"><span class="highlight-cl">  at java.lang.refl
ct.Constructor.newInstance(Constructor.java:417)
</span></span><span class="highlight-line"><span class="highlight-cl">  at zejian.Single
onEnum.main(SingletonEnum.java:38)
</span></span><span class="highlight-line"><span class="highlight-cl">  at sun.reflect.N
tiveMethodAccessorImpl.invoke0(Native Method)
</span></span><span class="highlight-line"><span class="highlight-cl">  at sun.reflect.N
tiveMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
</span></span><span class="highlight-line"><span class="highlight-cl">  at sun.reflect.De
egatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
</span></span><span class="highlight-line"><span class="highlight-cl">  at java.lang.refl
ct.Method.invoke(Method.java:498)
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">    at com.intellij.rt
execution.application.AppMain.main(AppMain.java:144)
</span></span></code></pre>
<p>显然告诉我们不能使用反射创建枚举类，这是为什么呢？不妨看看 newInstance 方法源码：</p

<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public T newInstance(Object ... initargs)
</span></span><span class="highlight-line"><span class="highlight-cl">        throws Instan
iationException, IllegalAccessException,
</span></span><span class="highlight-line"><span class="highlight-cl">               IllegalAr
umentException, InvocationTargetException
</span></span><span class="highlight-line"><span class="highlight-cl">    {
</span></span><span class="highlight-line"><span class="highlight-cl">        if (!override) {
</span></span><span class="highlight-line"><span class="highlight-cl">            if (!Reflecti
n.quickCheckMemberAccess(clazz, modifiers)) {
</span></span><span class="highlight-line"><span class="highlight-cl">                Class&lt
?&gt; caller = Reflection.getCallerClass();
</span></span><span class="highlight-line"><span class="highlight-cl">                checkAc
ess(caller, clazz, null, modifiers);
</span></span><span class="highlight-line"><span class="highlight-cl">            }
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">        //这里判断M
difier.ENUM是不是枚举修饰符，如果是就抛异常
</span></span><span class="highlight-line"><span class="highlight-cl">        if ((clazz.get
odifiers() &amp;amp; Modifier.ENUM) != 0)
</span></span><span class="highlight-line"><span class="highlight-cl">            throw new
llegalArgumentException("Cannot reflectively create enum objects");
</span></span><span class="highlight-line"><span class="highlight-cl">        ConstructorA
cessor ca = constructorAccessor;   // read volatile
</span></span><span class="highlight-line"><span class="highlight-cl">        if (ca == null)
{
</span></span><span class="highlight-line"><span class="highlight-cl">            ca = acqui
eConstructorAccessor();
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">        @Suppress
arnings("unchecked")
</span></span><span class="highlight-line"><span class="highlight-cl">        T inst = (T) ca
newInstance(initargs);
</span></span><span class="highlight-line"><span class="highlight-cl">        return inst;
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span></code></pre>
<p>源码很了然，确实无法使用反射创建枚举实例，也就是说明了创建枚举实例只有编译器能够做到已。显然枚举单例模式确实是很不错的选择，因此我们推荐使用它。但是这总不是万能的，对于 android 平台这个可能未必是最好的选择，在 android 开发中，内存优化是个大块头，而使用枚举时占用内存常常是静态变量的两倍还多，因此 android 官方在内存优化方面给出的建议是尽量避免在 android 中使用 enum。但是不管如何，关于单例，我们总是应该记住：线程安全，延迟加载，序列化与反序化安全，反射安全是很重重要的。</p>
<h2 id="EnumMap">EnumMap</h2>
<h2 id="EnumMap基本用法">EnumMap 基本用法</h2>
<p>先思考这样一个问题，现在我们有一堆 size 大小相同而颜色不同的数据，需要统计出每种颜色数量是多少以便将数据录入仓库，定义如下枚举用于表示颜色 Color:</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">enum Color {
</span></span><span class="highlight-line"><span class="highlight-cl">    GREEN,RED,BL
```

```
E,YELLOW
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">import java.util.*;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">public class Enu
MapDemo {
</span></span><span class="highlight-line"><span class="highlight-cl">    public static vo
d main(String[] args){
</span></span><span class="highlight-line"><span class="highlight-cl">        List&lt;Cloth
s&gt; list = new ArrayList&lt;&gt;();
</span></span><span class="highlight-line"><span class="highlight-cl">        list.add(new
lothes("C001",Color.BLUE));
</span></span><span class="highlight-line"><span class="highlight-cl">        list.add(new
lothes("C002",Color.YELLOW));
</span></span><span class="highlight-line"><span class="highlight-cl">        list.add(new
lothes("C003",Color.RED));
</span></span><span class="highlight-line"><span class="highlight-cl">        list.add(new
lothes("C004",Color.GREEN));
</span></span><span class="highlight-line"><span class="highlight-cl">        list.add(new
lothes("C005",Color.BLUE));
</span></span><span class="highlight-line"><span class="highlight-cl">        list.add(new
lothes("C006",Color.BLUE));
</span></span><span class="highlight-line"><span class="highlight-cl">        list.add(new
lothes("C007",Color.RED));
</span></span><span class="highlight-line"><span class="highlight-cl">        list.add(new
lothes("C008",Color.YELLOW));
</span></span><span class="highlight-line"><span class="highlight-cl">        list.add(new
lothes("C009",Color.YELLOW));
</span></span><span class="highlight-line"><span class="highlight-cl">        list.add(new
lothes("C010",Color.GREEN));
</span></span><span class="highlight-line"><span class="highlight-cl">        //方案1:使用
ashMap
</span></span><span class="highlight-line"><span class="highlight-cl">        Map&lt;Strin
,Integer&gt; map = new HashMap&lt;&gt;();
</span></span><span class="highlight-line"><span class="highlight-cl">        for (Clothes c
othes:list){
</span></span><span class="highlight-line"><span class="highlight-cl">            String colo
Name=clothes.getColor().name();
</span></span><span class="highlight-line"><span class="highlight-cl">            Integer co
nt = map.get(colorName);
</span></span><span class="highlight-line"><span class="highlight-cl">            if(count!=
ull){
</span></span><span class="highlight-line"><span class="highlight-cl">                map.put
colorName,count+1);
</span></span><span class="highlight-line"><span class="highlight-cl">            }else {
</span></span><span class="highlight-line"><span class="highlight-cl">                map.put
colorName,1);
</span></span><span class="highlight-line"><span class="highlight-cl">            }
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
```

```
ntln(map.toString());
        System.out.println("---------------");
        //方案2:使用EnumMap
        Map<Integer, Integer> enumMap=new EnumMap<>(Color.class);
        for (Clothes clothes:list){
            Color color=clothes.getColor();
            Integer count = enumMap.get(color);
            if(count!=null){
                enumMap.put(color,count+1);
            }else {
                enumMap.put(color,1);
            }
        }
        System.out.println(enumMap.toString());
    }

    /**
     输出结果:
     {RED=2, BLUE=3, YELLOW=3, GREEN=2}
     ---------------
     {GREEN=2, RED=2, BLUE=3, YELLOW=3}
     */
}
```

代码比较简单，我们使用两种解决方案，一种是 HashMap，一种 EnumMap，虽然都统计出了确的结果，但是 EnumMap 作为枚举的专属的集合，我们没有理由再去使用 HashMap，毕竟 EnumMap 要求其 Key 必须为 Enum 类型，因而使用 Color 枚举实例作为 key 是最恰当不过了，也避免了取 name 的步骤，更重要的是 EnumMap 效率更高，因为其内部是通过数组实现的（稍后分析），意 EnumMap 的 key 值不能为 null，虽说是枚举专属集合，但其操作与一般的 Map 差不多，概括来说 EnumMap 是专门为枚举类型量身定做的 Map 实现，虽然使用其它的 Map（如 HashMap）能完成相同的功能，但是使用 EnumMap 会更加高效，它只能接收同一枚举类型的实例作为键值且能为 null，由于枚举类型实例的数量相对固定并且有限，所以 EnumMap 使用数组来存放与枚举类对应的值，毕竟数组是一段连续的内存空间，根据程序局部性原理，效率会相当高。下面我们来进一了解 EnumMap 的用法，先看构造函数：

```
//创建一个具有指定键类型的空枚举映射。
EnumMap(Class<K> keyType)
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">//创建一个其键类
与指定枚举映射相同的枚举映射，最初包含相同的映射关系（如果有的话）。
</span></span><span class="highlight-line"><span class="highlight-cl">EnumMap(Enum
ap&lt;K,? extends V&gt; m)
</span></span><span class="highlight-line"><span class="highlight-cl">//创建一个枚举映
，从指定映射对其初始化。
</span></span><span class="highlight-line"><span class="highlight-cl">EnumMap(Map&lt
K,? extends V&gt; m)
</span></span></code></pre>
<p>与 HashMap 不同，它需要传递一个类型信息，即 Class 对象，通过这个参数 EnumMap 就可
根据类型信息初始化其内部数据结构，另外两只是初始化时传入一个 Map 集合，代码演示如下：</p

<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">//使用第一种构造
</span></span><span class="highlight-line"><span class="highlight-cl">Map&lt;Color,Inte
er&gt; enumMap=new EnumMap&lt;&gt;(Color.class);
</span></span><span class="highlight-line"><span class="highlight-cl">//使用第二种构造
</span></span><span class="highlight-line"><span class="highlight-cl">Map&lt;Color,Inte
er&gt; enumMap2=new EnumMap&lt;&gt;(enumMap);
</span></span><span class="highlight-line"><span class="highlight-cl">//使用第三种构造
</span></span><span class="highlight-line"><span class="highlight-cl">Map&lt;Color,Inte
er&gt; hashMap = new HashMap&lt;&gt;();
</span></span><span class="highlight-line"><span class="highlight-cl">hashMap.put(Colo
.GREEN, 2);
</span></span><span class="highlight-line"><span class="highlight-cl">hashMap.put(Colo
.BLUE, 3);
</span></span><span class="highlight-line"><span class="highlight-cl">Map&lt;Color, Int
ger&gt; enumMap = new EnumMap&lt;&gt;(hashMap);
</span></span></code></pre>
<p>至于 EnumMap 的方法，跟普通的 map 几乎没有区别，注意与 HashMap 的主要不同在于构
方法需要传递类型参数和 EnumMap 保证 Key 顺序与枚举中的顺序一致，但请记住 Key 不能为 null
</p>
<h2 id="EnumMap实现原理剖析">EnumMap 实现原理剖析</h2>
<p>EnumMap 的源码有 700 多行，这里我们主要分析其内部存储结构，添加查找的实现，了解这
点，对应 EnumMap 内部实现原理也就比较清晰了，先看数据结构和构造函数</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">public class EnumMap&lt;K extends Enum&lt;K&gt;, V&gt; extends AbstractMap&lt;K, V
gt;
</span></span><span class="highlight-line"><span class="highlight-cl">    implements java
io.Serializable, Cloneable
</span></span><span class="highlight-line"><span class="highlight-cl">{
</span></span><span class="highlight-line"><span class="highlight-cl">    //Class对象引用
</span></span><span class="highlight-line"><span class="highlight-cl">    private final Cla
s&lt;K&gt; keyType;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    //存储Key值的
组
</span></span><span class="highlight-line"><span class="highlight-cl">    private transient
K[] keyUniverse;
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">    //存储Value值
数组
</span></span><span class="highlight-line"><span class="highlight-cl">    private transient
Object[] vals;
```


   //map的size
   private transient
int size = 0;

   //空map
   private static fi
al Enum<?>[] ZERO_LENGTH_ENUM_ARRAY = new Enum<?>[0];

   //构造函数
   public EnumM
p(Class<K> keyType) {
   this.keyType
= keyType;
   keyUniverse
 getKeyUniverse(keyType);
   vals = new O
ject[keyUniverse.length];
   }

}
</code></pre>
<p>EnumMap 继承了 AbstractMap 类，因此 EnumMap 具备一般 map 的使用方法，keyType
示类型信息，keyUniverse 表示键数组，存储的是所有可能的枚举值，vals 数组表示键对应的值，size
表示键值对个数。在构造函数中通过 <code>keyUniverse = getKeyUniverse(keyType);</code>
始化了 keyUniverse 数组的值，内部存储的是所有可能的枚举值，接着初始化了存在 Value 值得数组
vals，其大小与枚举实例的个数相同，getKeyUniverse 方法实现如下</p>
<pre><code class="highlight-chroma"><span class="highlight
cl">//返回枚举数组
private static <K
extends Enum<K>> K[] getKeyUniverse(Class<K> keyType) {
   //最终调用到
举类型的values方法，values方法返回所有可能的枚举值
   return Share
Secrets.getJavaLangAccess()

        .getEnumConstantsShared(keyType);
   }
</code></pre>
<p>从方法的返回值来看，返回类型是枚举数组，事实也是如此，最终返回值正是枚举类型的 values
方法的返回值，前面我们分析过 values 方法返回所有可能的枚举值，因此 keyUniverse 数组存储就
枚举类型的所有可能的枚举值。接着看 put 方法的实现</p>
<pre><code class="highlight-chroma"><span class="highlight
cl">public V put(K key, V value) {
   typeCheck(k
y);//检测key的类型
   //获取存放val
e值得数组下标
   int index = k
y.ordinal();
   //获取旧值
   Object oldVa
ue = vals[index];
<span class="highlight-line"><span class="highlight-cl">   //设置value值

```
        vals[index] = maskNull(value);
        if (oldValue = null)
            size++;
        return unmaskNull(oldValue);//返回旧值
    }
```

这里通过 typeCheck 方法进行了 key 类型检测，判断是否为枚举类型，如果类型不对，会抛出异常

```
private void typeCheck(K key) {
    Class&lt;?&gt; keyClass = key.getClass();//获取类型信息
    if (keyClass != keyType &amp;&amp; keyClass.getSuperclass() != keyType)
        throw new ClassCastException(keyClass + " != " + keyType);
}
```

接着通过 `int index = key.ordinal()` 的方式获取到该枚举实例的顺序值，利用值作为下标，把值存储在 vals 数组对应下标的元素中即 `vals[index]`，这也是为什么 EnumMap 能维持与枚举实例相同存储顺序的原因，我们发现在对 vals[]中元素进行赋值和返回旧值分别调用了 maskNull 方法和 unmaskNull 方法

```
//代表NULL值得空对象实例
private static final Object NULL = new Object() {
    public int hashCode() {
        return 0;
    }

    public String toString() {
        return "java.util.EnumMap.NULL";
    }
};

private Object maskNull(Object value) {
    //如果值为空返回NULL对象，否则返回value
    return (value == null ? NULL : value);
}

@SuppressWarnings("unchecked")
private V unmaskNull(Object value) {
    //将NULL对
```

转换为null值
        return (V)(val
e == NULL ? null : value);
    }
</code></pre>
由此看来 EnumMap 还是允许存放 null 值的，但 key 绝对不能为 null，对于 null 值，EnumM
p 进行了特殊处理,将其包装为 NULL 对象，毕竟 vals[]存的是 Object，maskNull 方法和 unmaskNul
 方法正是用于 null 的包装和解包装的。这就是 EnumMap 集合的添加过程。下面接着看获取方法</
>
<pre><code class="highlight-chroma"><span class="highlight
cl">public V get(Object key) {
        return (isVali
Key(key) ?
                unmask
ull(vals[((Enum<?>)key).ordinal()]) : null);
    }

 //对Key值的有效
和类型信息进行判断
 private boolean i
ValidKey(Object key) {
      if (key == null)
          return false;

      // Cheaper th
n instanceof Enum followed by getDeclaringClass
      Class<?>
keyClass = key.getClass();
      return keyClas
 == keyType || keyClass.getSuperclass() == keyType;
  }
</code></pre>
相对应 put 方法，get 方法显示相当简洁，key 有效的话，直接通过 ordinal 方法取索引，然后
值数组 vals 里通过索引获取值返回。remove 方法如下：
<pre><code class="highlight-chroma"><span class="highlight
cl">public V remove(Object key) {
      //判断key值
否有效
      if (!isValidKey
key))
          return null;
      //直接获取索

      int index = ((
num<?>)key).ordinal();

      Object oldVa
ue = vals[index];
      //对应下标元
值设置为null
      vals[index] =
null;
      if (oldValue !
 null)

   size--;//减s
ze
   return unma
kNull(oldValue);
   }
</code></pre>
<p>非常简单，key 值有效，通过 key 获取下标索引值，把 vals[]对应下标值设置为 null，size 减一
查看是否包含某个值，</p>
<pre><code class="highlight-chroma"><span class="highlight
cl">判断是否包含某value
public boolean co
tainsValue(Object value) {
   value = maskNu
l(value);
   //遍历数组实现
   for (Object val :
vals)
   if (value.equa
s(val))
   return true

   return false;
}
//判断是否包含key
public boolean co
tainsKey(Object key) {
   return isValidKe
(key) && vals[((Enum<?>)key).ordinal()] != null;
}
</code></pre>
<p>判断 value 直接通过遍历数组实现，而判断 key 就更简单了，判断 key 是否有效和对应 vals[]
是否存在该值。ok~，这就是 EnumMap 的主要实现原理，即内部有两个数组，长度相同，一个表
所有可能的键(枚举值)，一个表示对应的值，不允许 keynull，但允许 value 为 null，键都有一个对
的索引，根据索引直接访问和操作其键数组和值数组，由于操作都是数组，因此效率很高。</p>
<h2 id="EnumSet">EnumSet</h2>
<p>EnumSet 是与枚举类型一起使用的专用 Set 集合，EnumSet 中所有元素都必须是枚举类型。与
他 Set 接口的实现类 HashSet/TreeSet(内部都是用对应的 HashMap/TreeMap 实现的)不同的是，E
umSet 在内部实现是位向量(稍后分析)，它是一种极为高效的位运算操作，由于直接存储和操作都是 b
t，因此 EnumSet 空间和时间性能都十分可观，足以媲美传统上基于 int 的“位标志”的运算，重要
是我们可像操作 set 集合一般来操作位运算，这样使用代码更简单易懂同时又具备类型安全的优势。
意 EnumSet 不允许使用 null 元素。试图插入 null 元素将抛出 NullPointerException，但试图测试
断是否存在 null 元素或移除 null 元素则不会抛出异常，与大多数 collection 实现一样，EnumSet
是线程安全的，因此在多线程环境下应该注意数据同步问题，ok~，下面先来简单看看 EnumSet 的
用方式。</p>
<h2 id="EnumSet用法">EnumSet 用法</h2>
<p>创建 EnumSet 并不能使用 new 关键字，因为它是个抽象类，而应该使用其提供的静态工厂方
，EnumSet 的静态工厂方法比较多，如下：</p>
<pre><code class="highlight-chroma"><span class="highlight
cl">创建一个具有指定元素类型的空EnumSet。
EnumSet<E>
noneOf(Class<E> elementType)
//创建一个指定元
类型并包含所有枚举值的EnumSet

<E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType)
// 创建一个包括枚值中指定范围元素的EnumSet
<E extends Enum<E>> EnumSet<E> range(E from, E to)
// 初始集合包括指集合的补集
<E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> s)
// 创建一个包括参中所有元素的EnumSet
<E extends Enum<E>> EnumSet<E> of(E e)
<E extends Enum<E>> EnumSet<E> of(E e1, E e2)
<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3)
<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4)
<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4, E e5)
<E extends Enum<E>> EnumSet<E> of(E first, E... rest)
//创建一个包含参容器中的所有元素的EnumSet
<E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> s)
<E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)
</code></pre>
<p>代码演示如下：</p>
<pre><code class="highlight-chroma">import java.util.ArrayList;
import java.util.EnumSet;
import java.util.List

/**
 * Created by wuzeian on 2017/5/12.
 *
 */
enum Color {
   GREEN , RED , BUE , BLACK , YELLOW
}

public class EnumSetDemo {

<span class="highlight-line"><span class="highlight-cl">   public static vo

```
d main(String[] args){
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        //空集合
</span></span><span class="highlight-line"><span class="highlight-cl">        EnumSet&lt;
olor&gt; enumSet= EnumSet.noneOf(Color.class);
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("添加前："+enumSet.toString());
</span></span><span class="highlight-line"><span class="highlight-cl">        enumSet.add
Color.GREEN);
</span></span><span class="highlight-line"><span class="highlight-cl">        enumSet.add
Color.RED);
</span></span><span class="highlight-line"><span class="highlight-cl">        enumSet.add
Color.BLACK);
</span></span><span class="highlight-line"><span class="highlight-cl">        enumSet.add
Color.BLUE);
</span></span><span class="highlight-line"><span class="highlight-cl">        enumSet.add
Color.YELLOW);
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("添加后："+enumSet.toString());
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("----------------------------------");
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        //使用allOf
建包含所有枚举类型的enumSet，其内部根据Class对象初始化了所有枚举实例
</span></span><span class="highlight-line"><span class="highlight-cl">        EnumSet&lt;
olor&gt; enumSet1= EnumSet.allOf(Color.class);
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("allOf直接填充："+enumSet1.toString());
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("----------------------------------");
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        //初始集合包
枚举值中指定范围的元素
</span></span><span class="highlight-line"><span class="highlight-cl">        EnumSet&lt;
olor&gt; enumSet2= EnumSet.range(Color.BLACK,Color.YELLOW);
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("指定初始化范围："+enumSet2.toString());
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("----------------------------------");
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        //指定补集，
就是从全部枚举类型中去除参数集合中的元素，如下去掉上述enumSet2的元素
</span></span><span class="highlight-line"><span class="highlight-cl">        EnumSet&lt;
olor&gt; enumSet3= EnumSet.complementOf(enumSet2);
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("指定补集："+enumSet3.toString());
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("----------------------------------");
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        //初始化时直
```

```
指定元素
</span></span><span class="highlight-line"><span class="highlight-cl">        EnumSet&lt;
olor&gt; enumSet4= EnumSet.of(Color.BLACK);
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("指定Color.BLACK元素："+enumSet4.toString());
</span></span><span class="highlight-line"><span class="highlight-cl">        EnumSet&lt;
olor&gt; enumSet5= EnumSet.of(Color.BLACK,Color.GREEN);
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("指定Color.BLACK和Color.GREEN元素："+enumSet5.toString());
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("---------------------------------");
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        //复制enumS
t5容器的数据作为初始化数据
</span></span><span class="highlight-line"><span class="highlight-cl">        EnumSet&lt;
olor&gt; enumSet6= EnumSet.copyOf(enumSet5);
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("enumSet6："+enumSet6.toString());
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("---------------------------------");
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        List&lt;Color
gt; list = new ArrayList&lt;Color&gt;();
</span></span><span class="highlight-line"><span class="highlight-cl">        list.add(Color
BLACK);
</span></span><span class="highlight-line"><span class="highlight-cl">        list.add(Color
BLACK);//重复元素
</span></span><span class="highlight-line"><span class="highlight-cl">        list.add(Color
RED);
</span></span><span class="highlight-line"><span class="highlight-cl">        list.add(Color
BLUE);
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("list:"+list.toString());
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        //使用copyOf
Collection&lt;E&gt; c)
</span></span><span class="highlight-line"><span class="highlight-cl">        EnumSet en
mSet7=EnumSet.copyOf(list);
</span></span><span class="highlight-line"><span class="highlight-cl">        System.out.pr
ntln("enumSet7:"+enumSet7.toString());
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        /**
</span></span><span class="highlight-line"><span class="highlight-cl">         输出结果：
</span></span><span class="highlight-line"><span class="highlight-cl">         添加前：[]
</span></span><span class="highlight-line"><span class="highlight-cl">         添加后：[GR
EN, RED, BLUE, BLACK, YELLOW]
</span></span><span class="highlight-line"><span class="highlight-cl">         ----------
----------
</span></span><span class="highlight-line"><span class="highlight-cl">         allOf直接填
：[GREEN, RED, BLUE, BLACK, YELLOW]
</span></span><span class="highlight-line"><span class="highlight-cl">         ----------
----------
```

```
</span></span><span class="highlight-line"><span class="highlight-cl">        指定初始化
围：[BLACK, YELLOW]
</span></span><span class="highlight-line"><span class="highlight-cl">        ----------
----------
</span></span><span class="highlight-line"><span class="highlight-cl">        指定补集：[
REEN, RED, BLUE]
</span></span><span class="highlight-line"><span class="highlight-cl">        ----------
----------
</span></span><span class="highlight-line"><span class="highlight-cl">        指定Color.B
ACK元素：[BLACK]
</span></span><span class="highlight-line"><span class="highlight-cl">        指定Color.B
ACK和Color.GREEN元素：[GREEN, BLACK]
</span></span><span class="highlight-line"><span class="highlight-cl">        ----------
----------
</span></span><span class="highlight-line"><span class="highlight-cl">        enumSet6:
GREEN, BLACK]
</span></span><span class="highlight-line"><span class="highlight-cl">        ----------
----------
</span></span><span class="highlight-line"><span class="highlight-cl">        list:[BLACK,
LACK, RED, BLUE]
</span></span><span class="highlight-line"><span class="highlight-cl">        enumSet7:[R
D, BLUE, BLACK]
</span></span><span class="highlight-line"><span class="highlight-cl">        */
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span></code></pre>
<p>转载：<a href="https://ld246.com/forward?goto=https%3A%2F%2Fblog.csdn.net%2Fjav
zejian%2Farticle%2Fdetails%2F71333103" target="_blank" rel="nofollow ugc">https://blog.c
dn.net/javazejian/article/details/71333103</a></p>
```