> 本文由 简悦 SimpRead 转码， 原文地址 www.imooc.com

《手册》第 3 、4 、39 页中有几段关于枚举类型的描述：

> 【参考】枚举类名带上 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类，域成员均为常量，且构造方法被默认强制是私有。
>
> 【推荐】如果变量值仅在一个固定范围内变化用 enum 类型来定义。
>
> 【强制】二方库里可以定义枚举类型，参数可以使用枚举类型，但是接口返回值不允许使用 枚举类型或者包含枚举类型的 POJO 对象。

大多数 Java 程序员对枚举类型一知半解，大多数程序员对枚举的用法都非常简单。

本小节主要解决以下几个问题：

- 那么枚举类究竟是怎样的?
- 默认的构造方法为何是私有的?
- 为什么接口不要返回枚举类型。
- 枚举类还有哪些高级用法?

2.1 勿忘初心

我们学习一个框架，学习一个语言特性时，可以思考一下这个框架和语言特性出现的原因。

枚举一般用来表示一组相同类型的常量，比如月份、星期、颜色等。

枚举的主要使用场景是，当需要一组固定的常量，并且编译时成员就已能确定时就应该使用枚举。

因此枚举类型没必要多例，如果能够保证单例，则可以减少内存开销。

另外枚举为数值提供了命名，更容易理解，而且枚举更加安全，功能更加强大。

2.2 官方文档法

前面介绍过，优先通过官方文档来学习 Java 的语言特性。

> 如果枚举类如果被 abstract 或 final 修饰，枚举如果常量重复，如果尝试实例化枚举类型都会有编译错误。
>
> 枚举类除声明的枚举常量没有其他实例。
>
> 枚举类型的 E 是 Enum 的直接子类。

那么 Java 是如何保证除了定义的枚举常量外没有其他实例呢?

从手册中我们可以找到原因：

- Enum 的 clone 方法被 final 修饰，保证 enum 常量不会被克隆。
- 禁止对枚举类型的反射。
- 序列化机制保证反序列化时枚举类型不允许构造多个相同实例。

通过这些提示，我们就明白为何枚举类的构造函数是私有的，

文档中还介绍了枚举的成员，枚举的迭代，枚举类型作为 switch 的条件，带抽象函数的枚举常量等。

2.3 Java 反汇编

我们选取 JLS 中的一个代码片段：

```
public enum CoinEnum {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    CoinEnum(int value) {
        this.value = value;
    }

    private final int value;
    public int value() { return value; }
}
```

先编译：`javac CoinEnum.java`

然后再反汇编：`javap -c CoinEnum`

得到下面的反汇编后的代码：

```
public final class com.imooc.basic.learn_enum.CoinEnum extends
java.lang.Enum<com.imooc.basic.learn_enum.CoinEnum> {
  public static final com.imooc.basic.learn_enum.CoinEnum PENNY;

  public static final com.imooc.basic.learn_enum.CoinEnum NICKEL;

  public static final com.imooc.basic.learn_enum.CoinEnum DIME;

  public static final com.imooc.basic.learn_enum.CoinEnum QUARTER;


  public static com.imooc.basic.learn_enum.CoinEnum[] values();
    Code:
       0: getstatic     #1
       3: invokevirtual #2
       6: checkcast     #3
       9: areturn


  public static com.imooc.basic.learn_enum.CoinEnum valueOf(java.lang.String);
    Code:
       0: ldc           #4
       2: aload_0
       3: invokestatic  #5
       6: checkcast     #4
       9: areturn

  public int value();
    Code:
       0: aload_0
       1: getfield      #7
       4: ireturn

  static {};
    Code:
       0: new           #4
       3: dup
       4: ldc           #8
```

```
        6: iconst_0
        7: iconst_1
        8: invokespecial #9
       11: putstatic     #10
       14: new           #4
       17: dup
       18: ldc           #11
       20: iconst_1
       21: iconst_5
       22: invokespecial #9
       25: putstatic     #12
       28: new           #4
       31: dup
       32: ldc           #13
       34: iconst_2
       35: bipush        10
       37: invokespecial #9
       40: putstatic     #14
       43: new           #4
       46: dup
       47: ldc           #15
       49: iconst_3
       50: bipush        25
       52: invokespecial #9
       55: putstatic     #16
       58: iconst_4
       59: anewarray     #4
       62: dup
       63: iconst_0
       64: getstatic     #10
       67: aastore
       68: dup
       69: iconst_1
       70: getstatic     #12
       73: aastore
       74: dup
       75: iconst_2
       76: getstatic     #14
       79: aastore
       80: dup
       81: iconst_3
       82: getstatic     #16
       85: aastore
       86: putstatic     #1
       89: return
}
```


通过开头位置的继承关系 `com.imooc.basic.learn_enum.Coin extends java.lang.Enum<com.imooc.basic.learn_enum.Coin>`，验证了官方手册描述的 “枚举类型的 E 是 Enum 的直接子类。” 的说法。

我们还看到枚举类编译后被被自动加上 `final` 关键字。

枚举常量也会被加上 `public static final` 修饰。

另外我们还注意到和源码相比多了两个函数：

其中一个为：`public static com.imooc.basic.learn_enum.CoinEnum valueOf(java.lang.String);`（见“第 2 处代码”）

```
  public static com.imooc.basic.learn_enum.CoinEnum valueOf(java.lang.String);
    Code:
       0: ldc           #4
       2: aload_0
       3: invokestatic  #5
       6: checkcast     #4
       9: areturn
```

这是怎么回事？干嘛用的呢？

通过第 2 处代码的 code 偏移为 3 处的代码，我们可以看出调用了 `java.lang.Enum#valueOf` 函数。

我们直接找到该函数的源码：

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


根据注释我们可以知道：

- 该函数的功能时根据枚举名称和枚举类型找到对应的枚举常量。
- 所有的枚举类型有一个隐式的函数 `public static T valueOf(String)` 用来根据枚举名称来获取枚举常量。
- 如果想获取当前枚举的所有枚举常量可以通过调用隐式的 `public static T[] values()` 函数来实现。

另外一个就是上面提到的 `public static com.imooc.basic.learn_enum.CoinEnum[] values();` 函数。

我们回到上面反汇编的代码，偏移为 58 到 86 的指令转为 Java **代码效果**和下面很类似：

```
private static CoinEnum[] $VALUES;
static {
    $VALUES = new CoinEnum[4];
    $VALUES[0] = PENNY;
    $VALUES[1] = NICKEL;
    $VALUES[2] = DIME;
    $VALUES[3] = QUARTER;
}
```

根据第 1 处代码

```
public static com.imooc.basic.learn_enum.CoinEnum[] values();
  Code:
     0: getstatic     #1
     3: invokevirtual #2
     6: checkcast     #3
     9: areturn
```

我们可以大致还原成下面的代码：

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


因此整体的逻辑就很清楚了。

结合前面拷贝章节讲到的内容，接下来大家思考下一个新问题：**为什么返回克隆对象而不是属性里的枚举数组呢?**

其实这样设计的主要原因是：避免枚举数组在外部进行修改，影响到下一次调用：`CoinEnum.values()` 的结果。如：

```
@Test
public void testValues(){
     CoinEnum[] values1 = CoinEnum.values();
     values1[0] = CoinEnum.QUARTER;

     CoinEnum[] values2 = CoinEnum.values();
     Assert.assertEquals(values2[0],CoinEnum.PENNY);
}
```

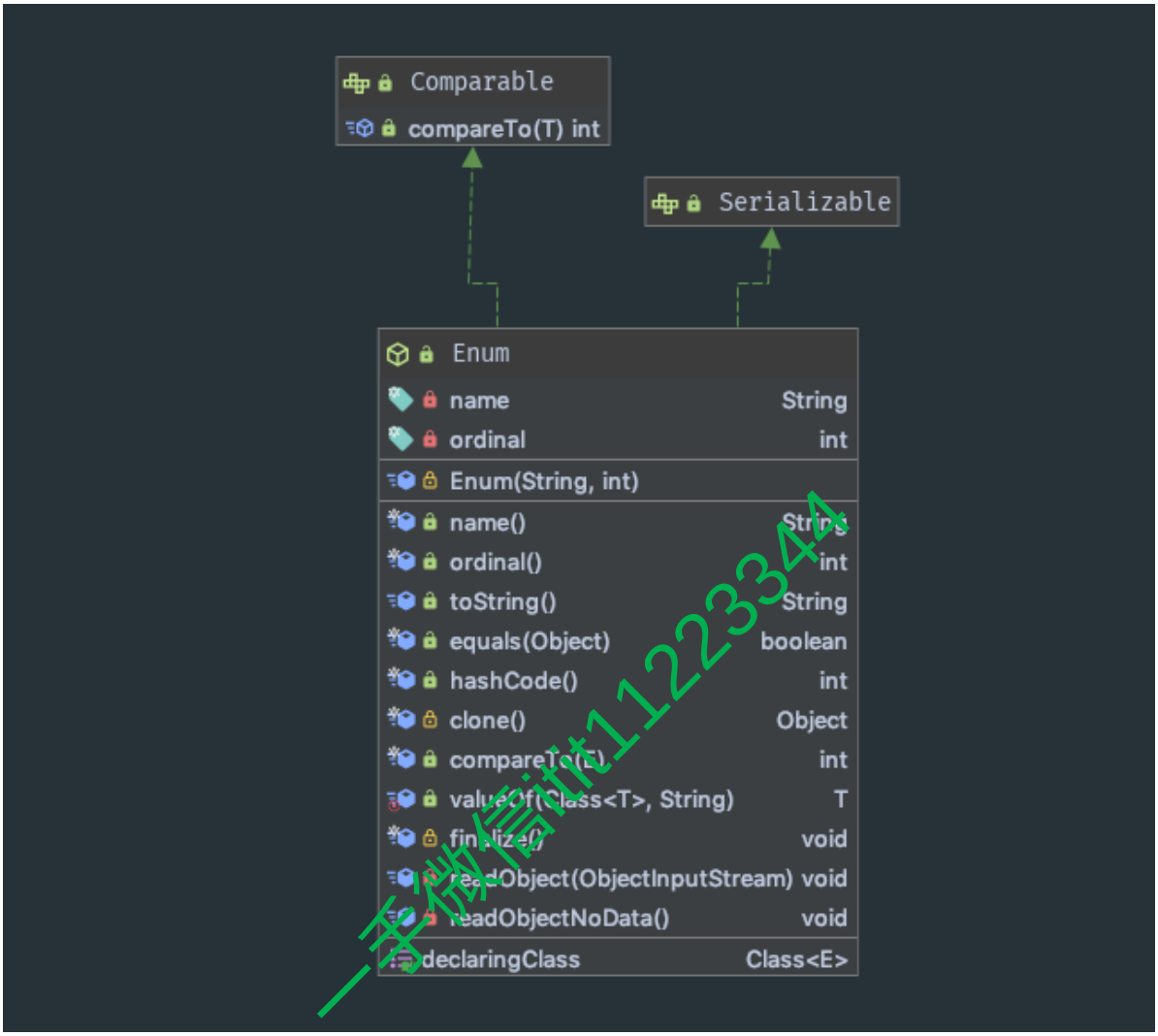
通过上面代码片段可以看出：对通过 clone 函数构造的新的数组对象（values1）的某个元素重新赋值并不会影响到原数组。

因此再次调用 `CoinEnum.values()` 仍然会返回基于原始枚举数组创建的新的拷贝对象（values2）。

2.4 源码大法

通过官方文档和反汇编，我们知道：枚举类都是 `java.lang.Enum` 的子类型。正因如此，我们可以通过查看 `Enum` 类的源码来学习枚举的一些知识。

我们通过 IDEA 自带的 Diagrams -> Show Diagrams -> Java Class Diagram 可以看到 Enum 类的继承关系，以及属性和函数等信息。



可以看到实现了 `Comparable<E>` 和 `Serializable` 接口。

那么为什么要实现这两个接口?

- 实现 `Comparable<E>` 接口很好理解，是为了排序。
- 实现 `Serializable` 接口是为了序列化。

前面序列化的小节中讲到：“一个类实现序列化接口，那么其子类也具备序列化的能力。”

从这里大家就会明白，正是因为其父类 `Enum` 实现了序列化接口，我们的枚举类没有显式实现序列化接口，使用 Java 原生序列化也并不会报错。

其中 `Enum` 类有两个属性 **:

`name` 表示枚举的名称。

`ordinal` 表示枚举的顺序，其主要用在 `java.util.EnumSet` 和 `java.util.EnumMap` 这两种基于枚举的数据结构中。

感兴趣的同学可以继续研究这两个数据结构的用法。

接下来我带大家重点看两个函数的源码：`java.lang.Enum#clone`函数和`java.lang.Enum#compareTo`函数。

我们查看`Enum`类的`clone`函数：

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

通过注释和源码我们可以明确地学习到，枚举类不支持`clone`，如果调用会报`CloneNotSupportedException`异常。

目的是为了保证枚举不能被克隆，维持单例的状态。

我们知道即使将构造方法设置为私有，也可以通过反射机制`setAccessible`为`true`后调用。普通的类可以通过`java.lang.reflect.Constructor#newInstance`来构造实例，这样就破坏了单例。

然而在该函数源码中对枚举类型会作判断并报`IllegalArgumentException`。

```
public T newInstance(Object ... initargs)
    throws InstantiationException, IllegalAccessException,
           IllegalArgumentException, InvocationTargetException
{

    if ((clazz.getModifiers() & Modifier.ENUM) != 0)
        throw new IllegalArgumentException("Cannot reflectively create enum
objects");


    return inst;
}
```


这样就防止了通过反射来构造枚举实例的可能性。

接下来我们看`compareTo`函数源码：

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

根据注释和源码，我们可以看到：其排序的依据是 枚举常量在枚举类的声明顺序。

2.5 断点大法

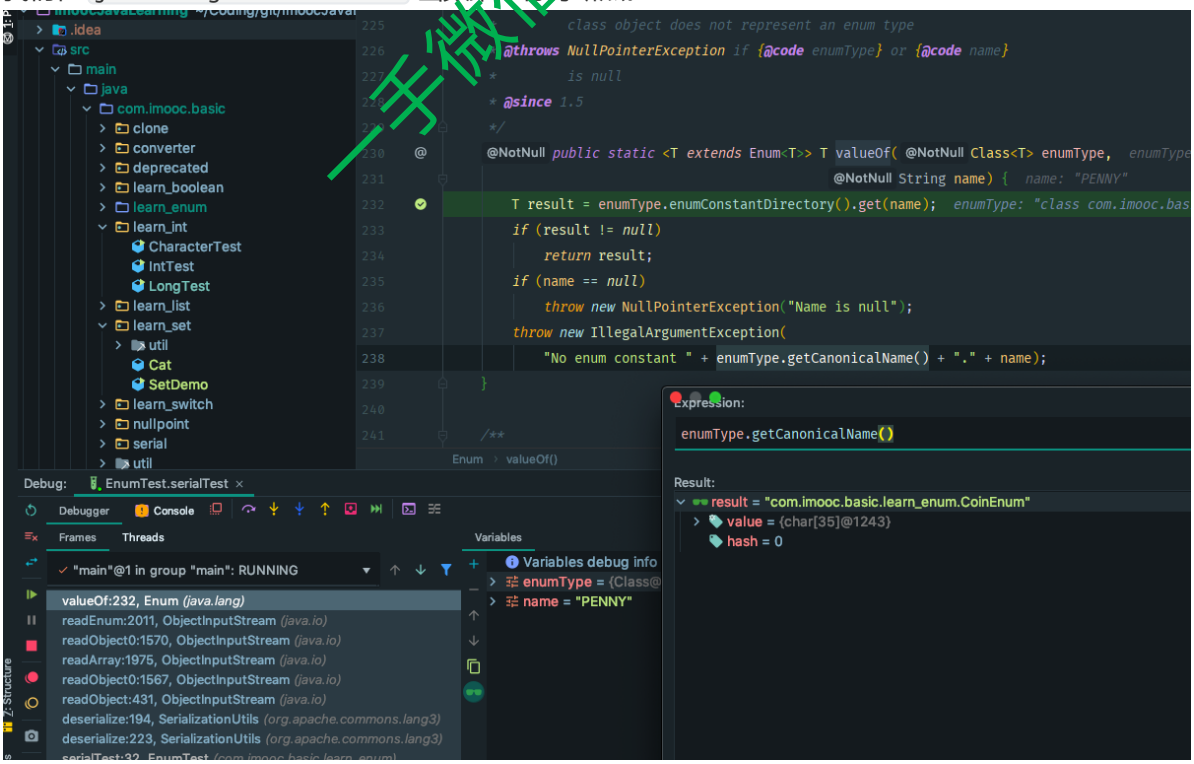
那么我们想想为啥《手册》中会有下面的这个规定呢?

> 【强制】二方库里可以定义枚举类型，参数可以使用枚举类型，但是接口返回值不允许使用枚举类型或者包含枚举类型的 POJO 对象。
>
> 注:
>
> 二方是指公司内部的其他部门;
>
> 二方库是指公司内部发布到中央仓库，可供公司内部其他应用依赖的库（jar 包）。

我们写一个测试函数来研究这个问题:

```
@Test
public void serialTest() {
    CoinEnum[] values = CoinEnum.values();

    byte[] serialize = SerializationUtils.serialize(values);

    log.info("序列化后的字符：{}",new String(serialize));

    CoinEnum[] values2 = SerializationUtils.deserialize(serialize);

    Assert.assertTrue(Objects.deepEquals(values, values2));
}
```


我们在 `java.lang.Enum#valueOf` 函数第一行打断点。



大家一定要自己尝试双击左下角的调用栈部分，查看从顶层调用

`org.apache.commons.lang3.SerializationUtils#deserialize(byte[])` 到

`java.lang.Enum#valueOf` 的整个调用过程。大家还可以通过表达式来查看参数的各种属性。

可以看到枚举的反序列化是通过调用 `java.lang.Enum#valueOf` 来实现的 **。

另外我们可以查看序列化后的字节流的字符表示形式：

序列化后的字符：

> ��□ur&
> [Lcom.imooc.basic.learn_enum.CoinEnum;ċ���>��□xp□r#com.imooc.basic.learn_enum.CoinEnum□xr□java.lang.Enum□xpt□PENNYq□t□NICKELq□t□DIMEq~□t□QUARTER

大致可以看出，序列化后的数据中主要包含**枚举的类型**和**枚举名称**。

我们了解了枚举的序列化和反序列化的原理后我们再思考：**为什么接口返回值不允许使用枚举类型或者包含枚举类型的 POJO 对象?**

上面讲到反序列化枚举类会调用 `java.lang.Enum#valueOf` ：

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

大家可以设想一下，如果将枚举当做 RPC 接口的返回值或者返回值对象的属性。如果己方接口新增枚举常量，而二方（公司的其他部门）没有及时升级 JAR 包，会出现什么情况?

此时，如果己方调用此接口时传入新的枚举常量，进行序列化。

反序列化时会调用到 `java.lang.Enum#valueOf` 函数， 此时参数 `name` 值为新的枚举名称。

```
T result = enumType.enumConstantDirectory().get(name);
```

此时 `result = null` ，从源码可以看出，将会抛出 `IllegalArgumentException` 。

通过查看该函数顶部的 `@throws IllegalArgumentException` 注释，我们也可以得知：

> 如果枚举类没有该常量，或者该反序列化的类对象并不是枚举类型则会抛出该异常。

因此，二方的枚举类添加新的常量后，如果使用方没有及时更新 JAR 包，使用 Java 反序列化时可能会抛出 `IllegalArgumentException` 。

除了 Java 序列化、反序列化外，其他的序列化框架对于枚举类处理也容易出现各种错误，因此请严格遵守这一条。

大家可以通过为 `CoinEnum` 枚举类新增一个枚举常量，并将新增的枚举常量通过 Java 序列化到文件中，然后在源码中注释掉新增的枚举常量，再反序列化，来复现这个 BUG。

有没有好的解决办法?

最常见的做法就是返回枚举的数值，并在返回的包中给出枚举类，在枚举类中提供通过根据值去获取枚举常量的方法（具体做法见下文）。

并通过使用 `@see` 或 `{@link}` 在该返回的枚举的数值注释中给出指向枚举类的快捷方式，如：

```
private Integer coinValue;
```

偶尔会遇到有些团队实现通过枚举中的值获取枚举常量时，居然用 switch ，非常让人吃惊。

如上面的 `CoinEnum` 的根据值获取枚举的函数，有些人会这么写：

```
public static CoinEnum getEnum(int value) {
        switch (value) {
            case 1:
                return PENNY;
            case 5:
                return NICKEL;
            case 10:
                return DIME;
            case 25:
                return QUARTER;
            default:
                return null;
        }
    }
```


这样做不符合设计模式的六大原则之一的“开闭原则”，因为如果删除、新增一个枚举常量等，也需要修改该函数。

另外如果枚举常量较多，很容易映射错误，后期很难维护。

可以利用前面讲到的枚举的 values 函数实现该功能，参考写法如下：

```
public static CoinEnum getEnum(int value) {
    for (CoinEnum coinEnum : CoinEnum.values()) {
        if (coinEnum.value == value) {
            return coinEnum;
        }
    }
    return null;
}
```

使用上面的写法，如果后面需要对枚举常量进行修改，该函数不需要改动，显然比之前好了很多。

实际工作中这种写法也很常见。

那么还有改进空间吗？

这种写法虽然挺不错，但是每次获取枚举对象都要遍历一次枚举数组，时间复杂度是 O(n)。

降低时间复杂度该怎么做？一个常见的思路就是**空间换时间。**

因此我们可以事先通过 Map 将映射关系存起来，使用时直接从 Map 中获取，参考代码如下：

```
@Getter
public enum CoinEnum {
    PENNY(1), NICKEL(5), DIME(10), QUARTER(25);

    CoinEnum(int value) {
        this.value = value;
    }

    private final int value;

    public int value() {
        return value;
    }

    private static final Map<Integer, CoinEnum> cache = new HashMap<>();

    static {
        for (CoinEnum coinEnum : CoinEnum.values()) {
            cache.put(coinEnum.getValue(), coinEnum);
        }
    }

    public static CoinEnum getEnum(int value) {
        return cache.getOrDefault(value, null);
    }
}
```

通过上面的优化，使用时时间复杂度为 O(1)，性能有所提升。

那么还有改进的空间吗？

上面的代码还存在以下几个问题：

- 每个枚举类中都需要编写类似的代码，很繁琐。
- 引入提供上述工具的很多枚举类，如果仅使用枚举常量，也会触发静态代码块的执行。

可不可以不修改枚举就能具备这种功能？是不是可以抽取公共部分代码封装成工具类？

我们来试一试。

首先大家可以想想，如果我们要将这部分封装成工具函数，需要哪些参数？

显然需要枚举的类型，还需要知道枚举中哪个属性作为缓存的 key，还需要传入匹配的参数。

因此可以编写如下工具类封装获取枚举对象的方法：

```
mport java.util.Map;
import java.util.Optional;
import java.util.Set;
import java.util.concurrent.ConcurrentHashMap;
import java.util.function.Function;

public class EnumUtils {
```

```
    private static final Map<Object, Object> key2EnumMap = new
ConcurrentHashMap<>();

    private static final Set<Class> enumSet = ConcurrentHashMap.newKeySet();



    public static <T extends java.lang.Enum<T>> Optional<T>
getEnumWithCache(Class<T> enumType, Function<T, Object> keyFunction, Object key)
{

        if (!enumSet.contains(enumType)) {

            synchronized (enumType) {
                if (!enumSet.contains(enumType)) {

                    enumSet.add(enumType);

                    for (T enumThis : enumType.getEnumConstants()) {

                        String mapKey = getKey(enumType,
keyFunction.apply(enumThis));

                        key2EnumMap.put(mapKey, enumThis);
                    }

                }
            }
        }
         return Optional.ofNullable((T) key2EnumMap.get(getKey(enumType, key)));
    }


    public static <T extends java.lang.Enum<T>> String getKey(Class<T> enumType,
Object key) {

        return enumType.getName().concat(key.toString());
    }


    public static <T extends java.lang.Enum<T>> Optional<T> getEnum(Class<T>
enumType, Function<T, Object> keyFunction, Object key) {
        for (T enumThis : enumType.getEnumConstants()) {
            if (keyFunction.apply(enumThis).equals(key)) {
                return Optional.of(enumThis);
            }
        }
        return Optional.empty();
    }
}
```


> 注：上述的几种写法，仅适合枚举常量和对应的属性一对一的情况，其他场景可能要换一种写法。
>
> 另外建议大家再思考下此方案还有没有优化的空间？是否还有其他优雅解决方案？

使用也非常简单:

```
@Test
public void test() {
    int key = 5;

    CoinEnum targetEnum = CoinEnum.NICKEL;

    CoinEnum anEnum = CoinEnum.getEnum(key);
    Assert.assertEquals(targetEnum, anEnum);


    Optional<CoinEnum> enumWithCache = EnumUtils.getEnumWithCache(CoinEnum.class,
CoinEnum::getValue, key);
    Assert.assertTrue(enumWithCache.isPresent());
    Assert.assertEquals(targetEnum, enumWithCache.get());


    Optional<CoinEnum> enumResult = EnumUtils.getEnum(CoinEnum.class,
CoinEnum::getValue, key);
    Assert.assertTrue(enumResult.isPresent());
    Assert.assertEquals(targetEnum, enumResult.get());
}
```

使用上面封装的工具类，不仅能够满足功能要求，还能实现了代码的复用，同时也做到了性能的优化。

通过上面的讲解，希望大家明白“尽信书不如无书”的道理，不要因为看到某个博客、某本书给出一个不错的写法就认为是标准答案，要有自己的思考，要有一定的代码优化意识。

4.1 实现计算

从官方文档中我们可以看到，枚举常量还可以带类方法：

```
enum Operation {
    PLUS {
        double eval(double x, double y) { return x + y; }
    },
    MINUS {
        double eval(double x, double y) { return x - y; }
    },
    TIMES {
        double eval(double x, double y) { return x * y; }
    },
    DIVIDED_BY {
        double eval(double x, double y) { return x / y; }
    };


    abstract double eval(double x, double y);

    public static void main(String args[]) {
        double x = Double.parseDouble(args[0]);
        double y = Double.parseDouble(args[1]);
        for (Operation op : Operation.values())
            System.out.println(x + " " + op + " " + y +
                               " = " + op.eval(x, y));
    }
```

```
}
```

可以在枚举类中定义抽象方法，在枚举常量中实现该方法来提供计算等功能.

JDK 源码中常见的枚举类： `java.util.concurrent.TimeUnit` 类就有类似的用法。

这种策略枚举方式也是替代 if - else if - else 的一种解决方案。

4.2 实现状态机

假设业务开发中需要实现状态流转的功能。

活动有：申报 -> 批准 -> 报名 -> 开始 -> 结束几种状态，依次流转。

我们可以通过下面的代码实现：

```
public enum ActivityStatesEnum {

    DEACLARE(1) {
        @Override
        ActivityStatesEnum nextState() {
            return APPROVE;
        }
    },
    APPROVE(2) {
        @Override
        ActivityStatesEnum nextState() {
            return ENROLL;
        }
    },
    ENROLL(3) {
        @Override
        ActivityStatesEnum nextState() {
            return START;
        }
    },
    START(4) {
        @Override
        ActivityStatesEnum nextState() {
            return END;
        }
    },
    END(5) {
        @Override
        ActivityStatesEnum nextState() {
            return this;
        }
    };

    private int status;

    abstract ActivityStatesEnum nextState();

    ActivityStatesEnum(int status) {
        this.status = status;
```

```
    }

  public ActivityStatesEnum getEnum(int status) {
        for (ActivityStatesEnum statesEnum : ActivityStatesEnum.values()) {
            if (statesEnum.status == status) {
                return statesEnum;
            }
        }
        return null;
    }
}
```

这样做的好处是可以通过 `getEnum` 函数获取枚举，直接通过 `nextState` 来获取下一个状态，更容易封装状态流转的函数，不需要每个状态都通过 `if` 判断再指定下一个状态，也降低出错的概率。

4.3 灵活的特性组合

fastjson 的 `com.alibaba.fastjson.parser.Feature` 类，灵活使用 `java.lang.Enum#ordinal` 和位运算实现了灵活的特性组合。

源码如下：

```
public enum Feature {

    AutoCloseSource,



   Feature(){
        mask = (1 << ordinal());
    }

    public final int mask;

    public final int getMask() {
        return mask;
    }

    public static boolean isEnabled(int features, Feature feature) {
        return (features & feature.mask) != 0;
    }

    public static int config(int features, Feature feature, boolean state) {
        if (state) {
            features |= feature.mask;
        } else {
            features &= ~feature.mask;
        }

        return features;
    }

    public static int of(Feature[] features) {
        if (features == null) {
```

```
            return 0;
        }

        int value = 0;

        for (Feature feature: features) {
            value |= feature.mask;
        }

        return value;
    }
}
```

我们知道 `java.lang.Enum#ordinal` 表示枚举序号。因此可以通过将 1 左移枚举序号个位置，构造各种特性的掩码。

各种特性的掩码可以任意组合，来表示不同的特征组合，也可以根据特性值反向解析出这些特性组合。

本节使用的学习方法有，思考技术的初衷，官方文档，读源码和反汇编。

主要要点如下：

1. 枚举一般表示相同类型的常量。
2. 枚举隐式继承自 `Enum<E>` ，实现了 `Comparable<E>` 和 `Serializable` 接口。
3. `java.util.EnumSet` 和 `java.util.EnumMap` 是两种关于 `Enum` 的数据结构。
4. 枚举类可以使用其 `ordinal` 属性，通过定义抽象函数、实现接口等方式实现高级用法。

更多枚举进阶知识可参考《Effective Java》 第 6 章 枚举和注解。

下一节将讲述 `ArrayList` 类的 `subList` 函数和 `Arrays` 类的 `asList` 函数。

1、通过前几节介绍的 `codota` 来学习两种和 `Enum` 相关的数据结构： `java.util.EnumSet` 和 `java.util.EnumMap` 的用法。

2、请为 `CoinEnum` 枚举类新增一个枚举常量，并将新增的枚举常量通过 Java 序列化到文件中，然后注释掉源码中新增的枚举常量，再反序列化，观察效果。

}