Java编程思想》中有这么一句话：“有时恰恰因为它，你才能够‘优雅而干净’地解决问题”——这句话说的是谁呢？就是本篇的主角——枚举（Enum）——大家鼓掌了。

在之前很长时间一段时间里，我都不怎么用枚举，因为总感觉它没什么用处——这其实就是“自我认知”的短见。****当一个人一直蹲在自己的深井里而不敢跳出来的话，那他真的只能看到井口那么大点的天空****。

随着时间的推移，我做的项目越来越多，和枚举见面的机会也越来越多，于是我就渐渐地对它越来越有兴趣，研究得多了，才发现原来枚举如此的优秀。

# 一、枚举的常规用法

一个精简的枚举非常的干净优雅，见下例。

**public** **enum** **Chenmo** {

WANGER, WANGSAN, WANGSI

}

我们为沉默枚举创建了三个值，分别是王二、王三、王四。这段代码实际上调用了3次Enum(String name, int ordinal)（ordinal单词的意思为顺序），也就是：

new Enum<Chenmo>("WANGER", 0);

new Enum<Chenmo>("WANGSAN", 1);

new Enum<Chenmo>("WANGSI", 2);

我们来遍历输出一下枚举：

**for** (Chenmo e : Chenmo.values()) {

System.out.println(e);

}//输出//WANGER//WANGSAN//WANGSI

# 二、作为switch的判断条件

使用枚举作为switch语句判断条件能让我们的代码可读性更强，示例如下。

···  
Chenmo key = Chenmo.WANGER;  
switch (key) {

case WANGSI:

System.**out**.println("今天我送出一个CSDN大鼠标垫");**break**;

case WANGSAN:

System.**out**.println("今天我被坑一个CSDN学院年卡");**break**;**default**:

System.**out**.println("今天我一边高兴，一边失落");**break**;

}  
···

在通过case关键字判断的时候，可以直接使用枚举值，非常简洁。另外，在编译期间限定类型，可以有效的避免越界的情况——字符串常量类型在作为switch判断条件的时候很容易因为误写而发生越界问题。

# 三、枚举实现单例

《Effective Java》一书中对使用枚举实现单例的方式推崇备至：

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。

我觉得“虽然还没有广泛采用”几个字可以去掉了，时至今日，大家应该都知道：使用枚举实现单例是一种非常好的方式。

先来看“双重校验锁”实现的单例：

**public** **class** **SingleTon2** {

// 私有化构造方法

**private** **SingleTon2**() {

}

;

**private** **static** **volatile** SingleTon2 singleTon = **null**;

**public** **static** SingleTon2 **getInstance**() {

// 第一次校验

**if** (singleTon == **null**) {

**synchronized** (SingleTon2.class) {

// 第二次校验

**if** (singleTon == **null**) {

singleTon = **new** SingleTon2();

}

}

}

**return** singleTon;

}

}

再来看枚举实现的单例：

**public** **enum** SingleTon {

INSTANCE;

**public** **void** **method**() {

System.**out**.println("我很快乐！");

}

}

不比不知道，一比吓一跳啊！枚举方式的单例简单到爆——为了不至于看起来太过精简，我还加了一个输出“我很快乐”的方法。

枚举实现的单例可轻松地解决两个问题：

1. 线程安全问题。因为Java虚拟机在加载枚举类的时候，会使用ClassLoader的loadClass方法，这个方法使用了同步代码块来保证线程安全。
2. 避免反序列化破坏单例。因为枚举的反序列化并不通过反射实现。

# 四、枚举可与数据库交互

我们可以配合Mybatis将数据库字段转换为枚举类型。现在假设有一个数据库字段check\_type的类型如下：

`check\_type` **int**(1) DEFAULT NULL COMMENT '检查类型（1：未通过、2：通过）',

它对应的枚举类型为CheckType，代码如下：

**public** **enum** CheckType {

NO\_PASS(0, "未通过"), PASS(1, "通过");

**private** **int** key;

**private** String text;

**private** **CheckType**(**int** key, String text) {

**this**.key = key;

**this**.text = text;

}

**public** **int** **getKey**() {

**return** key;

}

**public** String **getText**() {

**return** text;

}

**private** **static** HashMap<Integer,CheckType> map = **new** HashMap<Integer,CheckType>();

**static** {

**for** (CheckType d : CheckType.values()){

map.put(d.key, d);

}

}

**public** **static** CheckType **parse**(Integer index) {

**if**(map.containsKey(index)){

**return** map.get(index);

}

**return** null;

}

}

CheckType枚举类比我们刚开始见到的那个Chenmo枚举类要复杂一些。

第一，CheckType新添加了构造方法，还有两个字段，key为int型，text为String型。

第二，CheckType中有一个public static CheckType parse(Integer index)方法，可将一个Integer通过key的匹配转化为枚举类型。

那么现在，我们可以在Mybatis的配置文件中使用typeHandler将数据库字段转化为枚举类型。

<resultMap id="CheckLog" type="com.entity.CheckLog">

<id property="id" column="id"/>

<result property="checkType" column="check\_type" typeHandler="com.CheckTypeHandler"></result></resultMap>

其中checkType字段对应的类如下：

**public** **class** **CheckLog** **implements** **Serializable** {

**private** String id;

**private** CheckType checkType;

**public** String **getId**() {

**return** id;

}

**public** **void** **setId**(String id) {

**this**.id = id;

}

**public** CheckType **getCheckType**() {

**return** checkType;

}

**public** **void** **setCheckType**(CheckType checkType) {

**this**.checkType = checkType;

}

}

CheckTypeHandler转换器的类源码如下：

**public** **class** **CheckTypeHandler** **extends** **BaseTypeHandler**<**CheckType**> {

**@Override**

**public** CheckType **getNullableResult**(ResultSet rs, String index) **throws** SQLException {

**return** CheckType.parse(rs.getint(index));

}

**@Override**

**public** CheckType **getNullableResult**(ResultSet rs, **int** index) **throws** SQLException {

**return** CheckType.parse(rs.getint(index));

}

**@Override**

**public** CheckType **getNullableResult**(CallableStatement cs, **int** index) **throws** SQLException {

**return** CheckType.parse(cs.getint(index));

}

**@Override**

**public** **void** **setNonNullParameter**(PreparedStatement ps, **int** index, CheckType val, JdbcType arg3) **throws** SQLException {

ps.setint(index, val.getKey());

}

}

CheckTypeHandler 的核心功能就是调用CheckType枚举类的parse()方法对数据库字段进行转换。

# 五、枚举会比静态常量更消耗内存吗？

说完枚举最常用的4个知识点后，我们来讨论一下“枚举会比静态常量更消耗内存吗？”这个话题——知乎上有人问这样的问题，还有很多人参与回答。

按我的理解，问这个问题的人就好像是在问“0.000,001”比“0.000,000,99”大吗？你说是吗？