

简谈Java Enum

 **泊浮目** 发布于 2017-02-15

问题

我们偶尔能在项目中看到如下风格的代码：

```
public class ResponseCode {
    public static final int SUCCESS = 0;
    public static final int FAILURE = 10000;
    public static final int ILLEGAL_ARGUMENT = 10001;
    //.....
}
```

这样的代码一般被叫做int枚举模式。其包含着大量缺点：

- int枚举是编译时常量，被编译到客户端中，如果枚举常量关联的int发生变化，客户端必须重新编译，如果没有重新编译，程序仍可以运行，但行为就不确定了。
- 将int枚举常量翻译成可打印的字符串很麻烦。

在本例中，我要向客户端返回一个错误码和错误信息，错误码可以通过**ResponseCode.FAILURE**获得，但是错误信息恐怕只能以**"FAILURE"**这样的hard code返回了。

- 遍历一个组中所有的int枚举常量，获得int枚举组的大小，没有可靠的方法。

在这种**int枚举模式**的基础上，还有一种变体，我们称之为**String枚举模式**。虽然它提供了可打印的字符串，但这种方式存在性能问题，因为依赖于字符串的比较操作。

还有一种代码类似：

```
public interface CascadeConstant {
    String DELETION_CHECK_CODE = "deletion.check";
    String DELETION_DELETE_CODE = "deletion.delete";
    String DELETION_FORCE_DELETE_CODE = "deletion.forceDelete";
    String DELETION_CLEANUP_CODE = "deletion.cleanup";

    List<String> DELETION_CODES = Arrays.asList(DELETION_CHECK_CODE, DELETION_DELETE_CODE,
DELETION_FORCE_DELETE_CODE);
}
```

这样的代码我们一般称之为常量接口(constant interface)——这种接口不包含任何方法，它只包含静态的final域，每个域都导出一个常量。

类实现接口时，接口就充当可以引用这个类的实例类型。因此，**类实现了接口，就表明客户端对这个类的实例可以实施某些动作**。为了任何其他目的而定义的接口是不恰当的。

常量接口是对接口的一种不良使用。类在内部使用某些常量，纯粹是实现细节，实现常量接口，会导致把这样的实现细节泄露到该类的导出API中，因为接口中所有的域都是及方法public的。类实现常量接口，这对于这个类的用户来讲并没有实际的价值。实际上，这样做返回会让他们感到更糊涂，这还代表了一种承诺：如果在将来的发行版本中，这个类被修改了，它不再需要使用这些常量了，依然必须实现这个接口，以确保二进制兼容性。如果非final类实现了常量接口，它的所有子类的命名空间都受到了污染。[Java](#)平台类库中存在几个常量接口，如java.io.ObjectStreamConstants，这些接口都是反面典型，不值得效仿。

那既然不适合存在全部都是导出常量的常量接口，那么如果需要导出常量，它们应该放在哪里呢？**如果这些常量与某些现有的类或者接口紧密相关，就应该把这些常量添加到这个类或者接口中，注意，这里说添加到接口中并不是指的常量接口**。在Java平台类库中所有的数值包装类都导出MIN_VALUE和MAX_VALUE常量。如果这些常量最好被看作是枚举类型成员，那就应该用枚举类型来导出。否则，应该使用**不可实例化**的工具类来导出这些常量。

实践

我们先看改良版的ResponseCode：

```
public enum ResponseCode {
    SUCCESS(0),
    ERROR(10000),
    ILLEGAL_ARGUMENT(10001);

    private final int code;

    ResponseCode(int code) {
        this.code = code;
    }

    public int getCode() {
        return code;
    }

    public static ResponseCode getEnum(int value) {
        for (ResponseCode responseCode : ResponseCode.values()) {
            if (responseCode.getCode()==value) {
                return responseCode;
            }
        }
        return null;
    }
}
```

这样就克服了我们之前提到的缺点：**类型确定**。
在int枚举模式中或者String枚举模式中，我们会写出这样的方法签名：

```
//int 枚举模式
SendResponse(String description,int value)
//String枚举模式
//SendResponse(String description,int value)
```

这样的代码其实并不可靠，我们可以考虑：

```
SendResponse(String description,int code)
```

那么在调用的时候实质上是：

```
SendResponse(ResponseCode.ERROR.toString(),ResponseCode.ERROR.getCode())
```

为了更加方便，我们可以在此之上简单的封装一层：

```
AutoBuildAndSendResponse(ResponseCode responseCode){
    SendResponse(responseCode.toString(),responseCode.getCode());
}
```

这就很美滋滋。

在这个情况下，我们甚至还可以考虑在SendResponse方法中再加一个名为errorDetail的参数。利用方法重载，使Reponse的返回信息更为灵活。

不仅如此，Java的枚举类是很强大的。其本质是int值，并且背后基本原理也非常简单：它们就是通过共有的静态final域为每个枚举常量导出实例的类。因为没有可以访问的构造器，枚举类是真正的final。因为客户端既不能创建枚举类型的实例，也不能对它进行扩展，因此很可能没有实例，而只有声明过的枚举常量。换句话说，枚举类型是实例受控的。**而且，Java的枚举类还可以添加任意的方法和域，并实现任意接口，而且也提供了所有的Object方法的高级实现。**

关于StringValue的较佳实践

```
public enum SourceDiskType {  
  
    SYSTEM("system"),  
    DATA("data"),;  
  
    private String stringValue;  
  
    SourceDiskType(String stringValue) {  
        setStringValue(stringValue);  
    }  
  
    public String getStringValue() {  
        return stringValue;  
    }  
  
    public void setStringValue(String stringValue) {  
        this.stringValue = stringValue;  
    }  
  
    public static SourceDiskType getEnum(String stringValue) {  
        if (null == stringValue) {  
            return null;  
        }  
  
        for (SourceDiskType sourceDiskType : SourceDiskType.values()) {  
            if (sourceDiskType.getStringValue().equals(stringValue)) {
```

这是阿里云早期版本SDK中的一段代码。

java

阅读 1.3k • 更新于 2020-09-13

👍 赞 2

🔖 收藏 4

🔗 分享

本作品系原创，采用《署名-非商业性使用-禁止演绎 4.0 国际》许可协议



泊浮说

作者是个热爱分享交流的人，所以有了这个专栏。你的点赞是我最大的更新动力。

关注专栏



泊浮目

ZStack Committer，患有严重的代码洁癖症，对Zookeeper、Kafka等开源软件源码饶有兴趣。现主要负责沃趣科技分布式异构数据库流转平台——QMatrix的开发，并致力于构建可靠、可维护、可扩展的软件系统。

4.8k 声望 244 粉丝

关注作者

0 条评论

得票数 最新



撰写评论 ...

提交评论