> 最近重看 Java 枚举，看到这篇觉得还不错的文章，于是简单翻译和完善了一些内容，分享给大家，希望你们也能有所收获。另外，不要忘了文末还有补充哦！
>
> ps: 这里发一篇枚举的文章，也是因为后面要发一篇非常实用的关于 SpringBoot 全局异常处理的比较好的实践，里面就用到了枚举。
>
> 这篇文章由 JavaGuide 翻译，公众号: JavaGuide,原文地址：https://www.baeldung.com/a-guide-to-java-enums 。
>
> 转载请注明上面这段文字。

1.概览

在本文中，我们将看到什么是 Java 枚举，它们解决了哪些问题以及如何在实践中使用 Java 枚举实现一些设计模式。

enum关键字在 java5 中引入，表示一种特殊类型的类，其总是继承java.lang.Enum类，更多内容可以自行查看其官方文档。

枚举在很多时候会和常量拿来对比，可能因为本身我们大量实际使用枚举的地方就是为了替代常量。那么这种方式由什么优势呢?

以这种方式定义的常量使代码更具可读性，允许进行编译时检查，预先记录可接受值的列表，并避免由于传入无效值而引起的意外行为。

下面示例定义一个简单的枚举类型 pizza 订单的状态，共有三种 ORDERED, READY, DELIVERED状态:

```
package shuang.kou.enumdemo.enumtest;

public enum PizzaStatus {
    ORDERED,
    READY,
    DELIVERED;
}
```

简单来说，我们通过上面的代码避免了定义常量，我们将所有和 pizza 订单的状态的常量都统一放到了一个枚举类型里面。

```
System.out.println(PizzaStatus.ORDERED.name());//ORDERED
System.out.println(PizzaStatus.ORDERED);//ORDERED
System.out.println(PizzaStatus.ORDERED.name().getClass());//class java.lang.String
System.out.println(PizzaStatus.ORDERED.getClass());//class
shuang.kou.enumdemo.enumtest.PizzaStatus
```

2.自定义枚举方法

现在我们对枚举是什么以及如何使用它们有了基本的了解，让我们通过在枚举上定义一些额外的API方法，将上一个示例提升到一个新的水平：

```
public class Pizza {
    private PizzaStatus status;
    public enum PizzaStatus {
        ORDERED,
        READY,
        DELIVERED;
    }

    public boolean isDeliverable() {
        return getStatus() == PizzaStatus.READY;
    }

    // Methods that set and get the status variable.
}
```

3.使用 == 比较枚举类型

由于枚举类型确保JVM中仅存在一个常量实例，因此我们可以安全地使用 == 运算符比较两个变量，如上例所示；此外，== 运算符可提供编译时和运行时的安全性。

首先，让我们看一下以下代码段中的运行时安全性，其中 == 运算符用于比较状态，并且如果两个值均为null 都不会引发 NullPointerException。相反，如果使用equals方法，将抛出 NullPointerException：

```
Pizza.PizzaStatus pizza = null;
System.out.println(pizza.equals(Pizza.PizzaStatus.DELIVERED));//空指针异常
System.out.println(pizza == Pizza.PizzaStatus.DELIVERED);//正常运行
```

对于编译时安全性，我们看另一个示例，两个不同枚举类型进行比较：

```
if (Pizza.PizzaStatus.DELIVERED.equals(TestColor.GREEN)); // 编译正常
if (Pizza.PizzaStatus.DELIVERED == TestColor.GREEN);      // 编译失败，类型不匹配
```

4.在 switch 语句中使用枚举类型

```
public int getDeliveryTimeInDays() {
    switch (status) {
        case ORDERED:
            return 5;
        case READY:
            return 2;
        case DELIVERED:
            return 0;
```

```
        }
        return 0;
    }
```

5.枚举类型的属性,方法和构造函数

> 文末有我(JavaGuide)的补充。

你可以通过在枚举类型中定义属性,方法和构造函数让它变得更加强大。

下面，让我们扩展上面的示例，实现从比萨的一个阶段到另一个阶段的过渡，并了解如何摆脱之前使用的if语句和switch语句：

```
public class Pizza {

    private PizzaStatus status;
    public enum PizzaStatus {
        ORDERED (5){
            @Override
            public boolean isOrdered() {
                return true;
            }
        },
        READY (2){
            @Override
            public boolean isReady() {
                return true;
            }
        },
        DELIVERED (0){
            @Override
            public boolean isDelivered() {
                return true;
            }
        };

        private int timeToDelivery;

        public boolean isOrdered() {return false;}

        public boolean isReady() {return false;}

        public boolean isDelivered(){return false;}

        public int getTimeToDelivery() {
            return timeToDelivery;
        }

        PizzaStatus (int timeToDelivery) {
            this.timeToDelivery = timeToDelivery;
        }
```

```
    }

    public boolean isDeliverable() {
        return this.status.isReady();
    }

    public void printTimeToDeliver() {
        System.out.println("Time to delivery is " +
          this.getStatus().getTimeToDelivery());
    }

    // Methods that set and get the status variable.
}
```

下面这段代码展示它是如何 work 的：

```
@Test
public void givenPizaOrder_whenReady_thenDeliverable() {
    Pizza testPz = new Pizza();
    testPz.setStatus(Pizza.PizzaStatus.READY);
    assertTrue(testPz.isDeliverable());
}
```

6.EnumSet and EnumMap

6.1. EnumSet

`EnumSet` 是一种专门为枚举类型所设计的 `Set` 类型。

与`HashSet`相比，由于使用了内部位向量表示，因此它是特定 `Enum` 常量集的非常有效且紧凑的表示形式。

它提供了类型安全的替代方法，以替代传统的基于int的“位标志”，使我们能够编写更易读和易于维护的简洁代码。

`EnumSet` 是抽象类，其有两个实现：`RegularEnumSet` 、`JumboEnumSet`，选择哪一个取决于实例化时枚举中常量的数量。

在很多场景中的枚举常量集合操作（如：取子集、增加、删除、`containsAll`和`removeAll`批操作）使用`EnumSet`非常合适；如果需要迭代所有可能的常量则使用`Enum.values()`。

```
public class Pizza {

    private static EnumSet<PizzaStatus> undeliveredPizzaStatuses =
      EnumSet.of(PizzaStatus.ORDERED, PizzaStatus.READY);

    private PizzaStatus status;

    public enum PizzaStatus {
        ...
```

```
    }

    public boolean isDeliverable() {
        return this.status.isReady();
    }

    public void printTimeToDeliver() {
        System.out.println("Time to delivery is " +
          this.getStatus().getTimeToDelivery() + " days");
    }

    public static List<Pizza> getAllUndeliveredPizzas(List<Pizza> input) {
        return input.stream().filter(
          (s) -> undeliveredPizzaStatuses.contains(s.getStatus()))
            .collect(Collectors.toList());
    }

    public void deliver() {
        if (isDeliverable()) {
            PizzaDeliverySystemConfiguration.getInstance().getDeliveryStrategy()
              .deliver(this);
            this.setStatus(PizzaStatus.DELIVERED);
        }
    }

    // Methods that set and get the status variable.
}
```

下面的测试演示了展示了 `EnumSet` 在某些场景下的强大功能：

```
@Test
public void givenPizaOrders_whenRetrievingUnDeliveredPzs_thenCorrectlyRetrieved()
{
    List<Pizza> pzList = new ArrayList<>();
    Pizza pz1 = new Pizza();
    pz1.setStatus(Pizza.PizzaStatus.DELIVERED);

    Pizza pz2 = new Pizza();
    pz2.setStatus(Pizza.PizzaStatus.ORDERED);

    Pizza pz3 = new Pizza();
    pz3.setStatus(Pizza.PizzaStatus.ORDERED);

    Pizza pz4 = new Pizza();
    pz4.setStatus(Pizza.PizzaStatus.READY);

    pzList.add(pz1);
    pzList.add(pz2);
    pzList.add(pz3);
    pzList.add(pz4);

    List<Pizza> undeliveredPzs = Pizza.getAllUndeliveredPizzas(pzList);
```

```
    assertTrue(undeliveredPzs.size() == 3);
}
```

6.2. EnumMap

EnumMap是一个专门化的映射实现，用于将枚举常量用作键。与对应的 HashMap 相比，它是一个高效紧凑的实现，并且在内部表示为一个数组:

```
EnumMap<Pizza.PizzaStatus, Pizza> map;
```

让我们快速看一个真实的示例，该示例演示如何在实践中使用它：

```
Iterator<Pizza> iterator = pizzaList.iterator();
while (iterator.hasNext()) {
    Pizza pz = iterator.next();
    PizzaStatus status = pz.getStatus();
    if (pzByStatus.containsKey(status)) {
      pzByStatus.get(status).add(pz);
    } else {
      List<Pizza> newPzList = new ArrayList<>();
      newPzList.add(pz);
      pzByStatus.put(status, newPzList);
    }
}
```

下面的测试演示了展示了 EnumMap 在某些场景下的强大功能：

```
@Test
public void givenPizaOrders_whenGroupByStatusCalled_thenCorrectlyGrouped() {
    List<Pizza> pzList = new ArrayList<>();
    Pizza pz1 = new Pizza();
    pz1.setStatus(Pizza.PizzaStatus.DELIVERED);

    Pizza pz2 = new Pizza();
    pz2.setStatus(Pizza.PizzaStatus.ORDERED);

    Pizza pz3 = new Pizza();
    pz3.setStatus(Pizza.PizzaStatus.ORDERED);

    Pizza pz4 = new Pizza();
    pz4.setStatus(Pizza.PizzaStatus.READY);

    pzList.add(pz1);
    pzList.add(pz2);
    pzList.add(pz3);
    pzList.add(pz4);
```

```
    EnumMap<Pizza.PizzaStatus,List<Pizza>> map = Pizza.groupPizzaByStatus(pzList);
    assertTrue(map.get(Pizza.PizzaStatus.DELIVERED).size() == 1);
    assertTrue(map.get(Pizza.PizzaStatus.ORDERED).size() == 2);
    assertTrue(map.get(Pizza.PizzaStatus.READY).size() == 1);
}
```

7. 通过枚举实现一些设计模式

7.1 单例模式

通常，使用类实现 Singleton 模式并非易事，枚举提供了一种实现单例的简便方法。

《Effective Java 》和《Java与模式》都非常推荐这种方式，使用这种方式方式实现枚举可以有什么好处呢?

《Effective Java》

> 这种方法在功能上与公有域方法相近，但是它更加简洁，无偿提供了序列化机制，绝对防止多次实例化，即使是在面对复杂序列化或者反射攻击的时候。虽然这种方法还没有广泛采用，但是单元素的枚举类型已经成为实现 Singleton的最佳方法。 —-《Effective Java 中文版 第二版》

《Java与模式》

> 《Java与模式》中，作者这样写道，使用枚举来实现单实例控制会更加简洁，而且无偿地提供了序列化机制，并由JVM从根本上提供保障，绝对防止多次实例化，是更简洁、高效、安全的实现单例的方式。

下面的代码段显示了如何使用枚举实现单例模式:

```
public enum PizzaDeliverySystemConfiguration {
    INSTANCE;
    PizzaDeliverySystemConfiguration() {
        // Initialization configuration which involves
        // overriding defaults like delivery strategy
    }

    private PizzaDeliveryStrategy deliveryStrategy = PizzaDeliveryStrategy.NORMAL;

    public static PizzaDeliverySystemConfiguration getInstance() {
        return INSTANCE;
    }

    public PizzaDeliveryStrategy getDeliveryStrategy() {
        return deliveryStrategy;
    }
}
```

如何使用呢？请看下面的代码:

```
PizzaDeliveryStrategy deliveryStrategy =
PizzaDeliverySystemConfiguration.getInstance().getDeliveryStrategy();
```

通过 `PizzaDeliverySystemConfiguration.getInstance()` 获取的就是单例的 `PizzaDeliverySystemConfiguration`

7.2 策略模式

通常，策略模式由不同类实现同一个接口来实现的。

这也就意味着添加新策略意味着添加新的实现类。使用枚举，可以轻松完成此任务，添加新的实现意味着只定义具有某个实现的另一个实例。

下面的代码段显示了如何使用枚举实现策略模式：

```
public enum PizzaDeliveryStrategy {
    EXPRESS {
        @Override
        public void deliver(Pizza pz) {
            System.out.println("Pizza will be delivered in express mode");
        }
    },
    NORMAL {
        @Override
        public void deliver(Pizza pz) {
            System.out.println("Pizza will be delivered in normal mode");
        }
    };

    public abstract void deliver(Pizza pz);
}
```

给 `Pizza` 增加下面的方法：

```
public void deliver() {
    if (isDeliverable()) {
        PizzaDeliverySystemConfiguration.getInstance().getDeliveryStrategy()
          .deliver(this);
        this.setStatus(PizzaStatus.DELIVERED);
    }
}
```

如何使用呢？请看下面的代码：

```
@Test
public void givenPizaOrder_whenDelivered_thenPizzaGetsDeliveredAndStatusChanges()
{
    Pizza pz = new Pizza();
    pz.setStatus(Pizza.PizzaStatus.READY);
    pz.deliver();
```

```
    assertTrue(pz.getStatus() == Pizza.PizzaStatus.DELIVERED);
}
```

8. Java 8 与枚举

Pizza 类可以用Java 8重写，您可以看到方法 lambda 和Stream API如何使 `getAllUndeliveredPizzas ()` 和 `groupPizzaByStatus ()` 方法变得如此简洁：

`getAllUndeliveredPizzas ()` ：

```
public static List<Pizza> getAllUndeliveredPizzas(List<Pizza> input) {
    return input.stream().filter(
      (s) -> !deliveredPizzaStatuses.contains(s.getStatus()))
        .collect(Collectors.toList());
}
```

`groupPizzaByStatus ()` ：

```
public static EnumMap<PizzaStatus, List<Pizza>>
  groupPizzaByStatus(List<Pizza> pzList) {
    EnumMap<PizzaStatus, List<Pizza>> map = pzList.stream().collect(
      Collectors.groupingBy(Pizza::getStatus,
      () -> new EnumMap<>(PizzaStatus.class), Collectors.toList()));
    return map;
}
```

9. Enum 类型的 JSON 表现形式

使用Jackson库，可以将枚举类型的JSON表示为POJO。下面的代码段显示了可以用于同一目的的Jackson批注：

```
@JsonFormat(shape = JsonFormat.Shape.OBJECT)
public enum PizzaStatus {
    ORDERED (5){
        @Override
        public boolean isOrdered() {
            return true;
        }
    },
    READY (2){
        @Override
        public boolean isReady() {
            return true;
        }
    },
    DELIVERED (0){
```

```
        @Override
        public boolean isDelivered() {
            return true;
        }
    };

    private int timeToDelivery;

    public boolean isOrdered() {return false;}

    public boolean isReady() {return false;}

    public boolean isDelivered(){return false;}

    @JsonProperty("timeToDelivery")
    public int getTimeToDelivery() {
        return timeToDelivery;
    }

    private PizzaStatus (int timeToDelivery) {
        this.timeToDelivery = timeToDelivery;
    }
}
```

我们可以按如下方式使用 `Pizza` 和 `PizzaStatus`：

```
Pizza pz = new Pizza();
pz.setStatus(Pizza.PizzaStatus.READY);
System.out.println(Pizza.getJsonString(pz));
```

生成 Pizza 状态以以下JSON展示：

```
{
  "status" : {
    "timeToDelivery" : 2,
    "ready" : true,
    "ordered" : false,
    "delivered" : false
  },
  "deliverable" : true
}
```

有关枚举类型的JSON序列化/反序列化（包括自定义）的更多信息，请参阅Jackson-将枚举序列化为JSON对象。

10.总结

本文我们讨论了Java枚举类型，从基础知识到高级应用以及实际应用场景，让我们感受到枚举的强大功能。

11. 补充

我们在上面讲到了，我们可以通过在枚举类型中定义属性,方法和构造函数让它变得更加强大。

下面我通过一个实际的例子展示一下，当我们调用短信验证码的时候可能有几种不同的用途，我们在下面这样定义：

```
public enum PinType {

    REGISTER(100000, "注册使用"),
    FORGET_PASSWORD(100001, "忘记密码使用"),
    UPDATE_PHONE_NUMBER(100002, "更新手机号码使用");

    private final int code;
    private final String message;

    PinType(int code, String message) {
        this.code = code;
        this.message = message;
    }

    public int getCode() {
        return code;
    }

    public String getMessage() {
        return message;
    }

    @Override
    public String toString() {
        return "PinType{" +
                "code=" + code +
                ", message='" + message + '\'' +
                '}';
    }
}
```

实际使用：

```
System.out.println(PinType.FORGET_PASSWORD.getCode());
System.out.println(PinType.FORGET_PASSWORD.getMessage());
System.out.println(PinType.FORGET_PASSWORD.toString());
```

Output:

```
100001
忘记密码使用
PinType{code=100001, message='忘记密码使用'}
```

这样的话，在实际使用起来就会非常灵活方便！