Java 常用武器库之枚举

(https://gitbook.cn/gitchat/author/5b3f67e7b5e02743239c0a55)

文心紫竹 (https://gitboo…

向作者提问 (https://gitbook.cn/m/mazi/author/5b3f67e7b5e02743239c0a55/question)

dubbo最早实战者，现转战springcloud，构建高并发、高性能、高可用系统；带领弟兄进行DDD实战，以解决互联网复杂业务场景；现挑战长链接领域，与socket、线程、锁、并发、分布式为伍

查看本场Chat

(https://gitbook.cn/gitchat/activity/5b670794245b6475efb66084)

1. 枚举基本特征

关键词 enum 可以将一组具名值的有限集合创建成一种新的类型，而这些具名的值可以作为常规程序组件使用。

枚举最常见的用途便是替换常量定义，为其增添类型约束，完成编译时类型验证。

1.1. 枚举定义

枚举的定义与类和常量定义非常类型。使用 enum 关键字替换 class 关键字，然后在 enum 中定义“常量”即可。

例如，需要将用户分为“可用”和“禁用”两种状态，为了达到定义的统一管理，一般会使用常量进行说明，如下：

```
public class UserStatusConstants {
    public static final int ENABLE = 0;
    public static final int DISABLE = 1;
}


public class User1 {
    private String name;
    private int status;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getStatus() {
        return status;
    }

    public void setStatus(int status) {
        this.status = status;
    }
}
```

在 User1 中，描述用户状态的类型为 int，（期望可用值为 0 和 1，但实际可选择值为 int），从使用方角度出发，需要事先知道 UserStatusConstants 中定义的常量才能保障调用的准确性，setStatus 和 getStatus 两个方法绕过了 Java 类型检测。

接下来，我们看一下枚举如何优化这个问题，enum 方案如下：

```
public enum UserStatus {
    ENABLE, DISABLE;
}
```

```
public class User2 {
    private String name;
    private UserStatus status;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public UserStatus getStatus() {
        return status;
    }

    public void setStatus(UserStatus status) {
        this.status = status;
    }
}
```

```
@Test
public void useUserStatus(){
    User2 user = new User2();
    user.setName("test");

    user.setStatus(UserStatus.DISABLE);
    user.setStatus(UserStatus.ENABLE);
}
```

getStatus 和 setStatus 所需类型为 UserStatus，不在是比较宽泛的 int。在使用的时候可以通过 UserStatus.XXX 的方式获取对用的枚举值。

1.2. 枚举的秘密

1.2.1. 枚举的单例性

枚举值具有单例性，及枚举中的每个值都是一个单例对象，可以直接使用 == 进行等值判断。

枚举是定义单例对象最简单的方法。

1.2.2. name 和 ordrial

> 对于简单的枚举，存在两个维度，一个是 name，即为定义的名称；一个是 ordinal，即为定义的顺序。

name 测试如下：

```
@Test
public void nameTest(){
    for (UserStatus userStatus : UserStatus.values()){
        // 枚举的name维度
        String name = userStatus.name();
        System.out.println("UserStatus:" + name);

        // 通过name获取定义的枚举
        UserStatus userStatus1 = UserStatus.valueOf(name);
        System.out.println(userStatus == userStatus1);
    }
}
```

输出结果为：

```
UserStatus:ENABLE
true
UserStatus:DISABLE
true
```

ordrial 测试如下：

```
@Test
public void ordinalTest(){
    for (UserStatus userStatus : UserStatus.values()){
        // 枚举的ordinal维度
        int ordinal = userStatus.ordinal();
        System.out.println("UserStatus:" + ordinal);

        // 通过ordinal获取定义的枚举
        UserStatus userStatus1 = UserStatus.values()[ordinal];
        System.out.println(userStatus == userStatus1);
    }
}
```

1.2.3. 枚举的本质

创建 enum 时，编译器会为你生成一个相关的类，这个类继承自 java.lang.Enum。

先看下 Enum 提供了什么：

```
public abstract class Enum<E extends Enum<E>>
        implements Comparable<E>, Serializable {
    // 枚举的Name维度
    private final String name;
    public final String name() {
        return name;
    }

    // 枚举的ordinal维度
    private final int ordinal;
    public final int ordinal() {
        return ordinal;
    }

    // 枚举构造函数
    protected Enum(String name, int ordinal) {
        this.name = name;
        this.ordinal = ordinal;
    }

    /**
     * 重写toString方法， 返回枚举定义名称
     */
    public String toString() {
        return name;
    }

    // 重写equals，由于枚举对象为单例，所以直接使用==进行比较
    public final boolean equals(Object other) {
        return this==other;
    }

    // 重写hashCode
    public final int hashCode() {
        return super.hashCode();
    }

    /**
     * 枚举为单例对象，不允许clone
     */
    protected final Object clone() throws CloneNotSupportedException {
        throw new CloneNotSupportedException();
    }

    /**
     * 重写compareTo方法，同种类型按照定义顺序进行比较
     */
    public final int compareTo(E o) {
        Enum<?> other = (Enum<?>)o;
        Enum<E> self = this;
        if (self.getClass() != other.getClass() && // optimization
            self.getDeclaringClass() != other.getDeclaringClass())
            throw new ClassCastException();
        return self.ordinal - other.ordinal;
    }

    /**
     * 返回定义枚举的类型
     */
    @SuppressWarnings("unchecked")
    public final Class<E> getDeclaringClass() {
        Class<?> clazz = getClass();
```

```
        Class<?> zuper = clazz.getSuperclass();
        return (zuper == Enum.class) ? (Class<E>)clazz : (Class<E>)zuper;
    }

    /**
     * 静态方法，根据name获取枚举值
     * @since 1.5
     */
    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);
    }

    protected final void finalize() { }

    /**
     * 枚举为单例对象，禁用反序列化
     */
    private void readObject(ObjectInputStream in) throws IOException,
        ClassNotFoundException {
        throw new InvalidObjectException("can't deserialize enum");
    }

    private void readObjectNoData() throws ObjectStreamException {
        throw new InvalidObjectException("can't deserialize enum");
    }
}
```

从 Enum 中我们可以得到：

1. Enum 中对 name 和 ordrial（final）的属性进行定义，并提供构造函数进行初始化。
2. 重写了 equals、hashCode、toString 方法，其中 toString 方法默认返回 name。
3. 实现了 Comparable 接口，重写 compareTo，使用枚举定义顺序进行比较。
4. 实现了 Serializable 接口，并重写禁用了 clone、readObject 等方法，以保障枚举的单例性。
5. 提供 valueOf 方法使用反射机制，通过 name 获取枚举值。

到此已经解释了枚举类的大多数问题。

`UserStatus.values()`， `UserStatus.ENABLE`， `UserStatus.DISABLE`，又是从怎么来的呢？这些是编译器为其添加的。

```
@Test
public void enumTest(){
    System.out.println("Fields");

    for (Field field : UserStatus.class.getDeclaredFields()){
        field.getModifiers();
        StringBuilder fieldBuilder = new StringBuilder();
        fieldBuilder.append(Modifier.toString(field.getModifiers()))
                .append(" ")
                .append(field.getType())
                .append(" ")
                .append(field.getName());

        System.out.println(fieldBuilder.toString());
    }

    System.out.println();
    System.out.println("Methods");
    for (Method method : UserStatus.class.getDeclaredMethods()){
        StringBuilder methodBuilder = new StringBuilder();
        methodBuilder.append(Modifier.toString(method.getModifiers()));
        methodBuilder.append(method.getReturnType())
                .append(" ")
                .append(method.getName())
                .append("(");
        Parameter[] parameters = method.getParameters();
        for (int i=0; i< method.getParameterCount(); i++){
            Parameter parameter = parameters[i];
            methodBuilder.append(parameter.getType())
                    .append(" ")
                    .append(parameter.getName());
            if (i != method.getParameterCount() -1) {
                    methodBuilder.append(",");
            }
        }
        methodBuilder.append(")");
        System.out.println(methodBuilder);
    }
}
```

我们分别对 UserStatus 中的属性和方法进行打印，结果如下：

```
Fields
public static final class com.example.enumdemo.UserStatus ENABLE
public static final class com.example.enumdemo.UserStatus DISABLE
private static final class [Lcom.example.enumdemo.UserStatus; $VALUES

Methods
public staticclass [Lcom.example.enumdemo.UserStatus; values()
public staticclass com.example.enumdemo.UserStatus valueOf(class java.lang.S
tring arg0)
```

由输出，我们可知编译器为我们添加了以下几个特性：

1. 针对每一个定义的枚举值，添加一个同名的 public static final 的属性。
2. 添加一个 `private static final $VALUES` 属性记录枚举中所有的值信息。
3. 添加一个静态的 values 方法，返回枚举中所有的值信息 （$VALUES）。
4. 添加一个静态的 valueOf 方法，用于通过 name 获取枚举值（调用 Enum 中的 valueOf 方法）。

2. 枚举一种特殊的类

虽然编译器为枚举添加了很多功能，但究其本质，枚举终究是一个类。除了不能继承自一个 enum 外，我们基本上可以将 enum 看成一个常规类，因此属性、方法、接口等在枚举中仍旧有效。

2.1. 枚举中的属性和方法

除了编译器为我们添加的方法外，我们也可以在枚举中添加新的属性和方法，甚至可以有 main 方法。

```
public enum  CustomMethodUserStatus {
    ENABLE("可用"),
    DISABLE("禁用");

    private final String descr;

    private CustomMethodUserStatus(String descr) {
        this.descr = descr;
    }

    public String getDescr(){
        return this.descr;
    }

    public static void main(String... args){
        for (CustomMethodUserStatus userStatus : CustomMethodUserStatus.valu
es()){
            System.out.println(userStatus.toString() + ":" + userStatus.getD
escr());
        }
    }
}
```

main 执行输出结果：

```
ENABLE:可用
DISABLE:禁用
```

如果准备添加自定义方法，需要在 enum 实例序列的最后添加一个分号。同时 java 要求必须先定义 enum 实例，如果在定义 enum 实例前定义任何属性和方法，那么在编译过程中会得到相应的错误信息。

enum 中的构造函数和普通类没有太多的区别，但由于只能在 enum 中使用构造函数，其默认为 private，如果尝试升级可见范围，编译器会给出相应错误信息。

2.2. 重写枚举方法

枚举中的方法与普通类中方法并无差别，可以对其进行重写。其中 Enum 类中的 name 和 ordrial 两个方法为 final，无法重写。

```
public enum OverrideMethodUserStatus {
    ENABLE("可用"),
    DISABLE("禁用");

    private final String descr;

    OverrideMethodUserStatus(String descr) {
        this.descr = descr;
    }

    @Override
    public String toString(){
        return this.descr;
    }

    public static void main(String... args){
        for (OverrideMethodUserStatus userStatus : OverrideMethodUserStatus.
values()){
            System.out.println(userStatus.name() + ":" + userStatus.toString
());
        }
    }
}
```

main 输出结果为：

```
ENABLE:可用
DISABLE:禁用
```

重写 toString 方法，返回描述信息。

2.3. 接口实现

由于所有的 enum 都继承自 `java.lang.Enum` 类，而 Java 不支持多继承，所以我们的 enum 不能再继承其他类型，但 enum 可以同时实现一个或多个接口，从而对其进行扩展。

```
public interface CodeBasedEnum {
    int code();
}


public enum  CodeBasedUserStatus implements CodeBasedEnum{
    ENABLE(1), DISABLE(0);

    private final int code;

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

    @Override
    public int code(){
        return this.code;
    }

    public static void main(String... arg){
        for (CodeBasedUserStatus codeBasedEnum : CodeBasedUserStatus.values(
)){
            System.out.println(codeBasedEnum.name() + ":" + codeBasedEnum.co
de());
        }
    }
}
```

main 函数输出结果：

```
ENABLE:1
DISABLE:0
```

3. 枚举集合

针对枚举的特殊性，java 类库对 enum 的集合提供了支持（主要是围绕 ordinal 特性进行优化）。

3.1. EnumSet

Set 是一种集合，只能向其中添加不重复的对象。Java5 中引入了 EnumSet 对象，其内部使用 long 值作为比特向量，以最大化 Set 的性能。

EnumSet 存在两种实现类：

1. RegularEnumSet 针对枚举数量小于等于 64 的 EnumSet 实现，内部使用 long 作为存储。
2. JumboEnumSet 针对枚举数量大于 64 的 EnumSet 实现，内部使用 long 数组进行存储。

```
public enum UserStatus {
    A1, A2 ;
}
public enum MoreUserStatus {
    A1, A2, A3, A4, A5, A6, A7, A8, A9, A10,
    A11, A12, A13, A14, A15, A16, A17, A18, A19, A20,
    A21, A22, A23, A24, A25, A26, A27, A28, A29, A30,
    A31, A32, A33, A34, A35, A36, A37, A38, A39, A40,
    A41, A42, A43, A44, A45, A46, A47, A48, A49, A50,
    A51, A52, A53, A54, A55, A56, A57, A58, A59, A60,
    A61, A62, A63, A64, A65, A66, A67, A68, A69, A70,
    A71, A72, A73, A74, A75, A76, A77, A78, A79, A80,
    A81, A82, A83, A84, A85, A86, A87, A88, A89, A90,
    A91, A92, A93, A94, A95, A96, A97, A98, A99, A100;
}
```

```
@Test
public void typeTest(){
    EnumSet<UserStatus> userStatusesSet = EnumSet.noneOf(UserStatus.class);
    System.out.println("userStatusSet:" + userStatusesSet.getClass());

    EnumSet<MoreUserStatus> moreUserStatusesSet = EnumSet.noneOf(MoreUserSta
tus.class);
    System.out.println("moreUserStatusesSet:" + moreUserStatusesSet.getClass
());
}
```

输出结果为：

```
userStatusSet:class java.util.RegularEnumSet
moreUserStatusesSet:class java.util.JumboEnumSet
```

EnumSet 作为工厂类，提供大量的静态方法，以方便的创建 EnumSet：

1. noneOf 创建 EnumSet，空 Set，没有任何元素
2. allOf 创建 EnumSet，满 Set，所有元素都在其中
3. copyOf 从已有 Set 中创建 EnumSet
4. of 根据提供的枚举值创建 EnumSet
5. range 根据枚举定义的顺序，定义一个区间的 EnumSet

3.2. EnumMap

EnumMap 是一个特殊的 Map，他要求其中的键值必须来着一个 enum。

EnumMap 内部实现：

```
private transient Object[] vals;
public V get(Object key) {
    return (isValidKey(key) ?
            unmaskNull(vals[((Enum<?>)key).ordinal()]) : null);
}

public V put(K key, V value) {
    typeCheck(key);
    int index = key.ordinal();
    Object oldValue = vals[index];
    vals[index] = maskNull(value);
    if (oldValue == null)
        size++;
    return unmaskNull(oldValue);
}
```

由上可见，EnumMap 内部由数组实现（ordrial），以提高 Map 的操作速度。enum 中的每个实例作为键，总是存在，但是如果没有为这个键调用 put 方法来存入相应值的话，其对应的值便是 null。

在使用上，EnumMap 与 key 为枚举的 Map 并无差异。

```
@Test
public void test(){
    Map<UserStatus, String> welcomeMap = new EnumMap<>(UserStatus.class);
    welcomeMap.put(UserStatus.A1, "你好");
    welcomeMap.put(UserStatus.A2, "您好");
}
```

4. 枚举使用案例

枚举作为一种特殊的类，为很多场景提供了更优雅的解决方案。

4.1. Switch

在 Java 1.5 之前，只有一些简单类型（int，short，char，byte）可以用于 switch 的 case 语句，我们习惯采用 ‘ 常量 + case ’ 的方式增加代码的可读性，但是丢失了类型系统的校验。由于枚举的 ordinal 特性的存在，可以将其用于 case 语句。

```
public class FruitConstant {
    public static final int APPLE = 1;
    public static final int BANANA = 2;
    public static final int PEAR = 3;
}
// 没有类型保障
public String nameByConstant(int fruit){
        switch (fruit){
            case FruitConstant.APPLE:
                return "苹果";
            case FruitConstant.BANANA:
                return "香蕉";
            case FruitConstant.PEAR:
                return "梨";
        }
        return "未知";
    }


public enum FruitEnum {
    APPLE,
    BANANA,
    PEAR;
}
// 有类型保障
public String nameByEnum(FruitEnum fruit){
        switch (fruit){
            case APPLE:
                return "苹果";
            case BANANA:
                return "香蕉";
            case PEAR:
                return "梨";
        }
        return "未知";
    }
```

4.2. 单例

Java 中单例的编写主要有饿汉式、懒汉式、静态内部类等几种方式（双重锁判断存在缺陷），
但还有一种简单的方式是基于枚举的单例。

```
public interface Converter<S, T> {
    T convert(S source);
}

// 每一个枚举值都是一个单例对象
public enum Date2StringConverters implements Converter<Date, String>{
    yyyy_MM_dd("yyyy-MM-dd"),
    yyyy_MM_dd_HH_mm_ss("yyyy-MM-dd HH:mm:ss"),
    HH_mm_ss("HH:mm:ss");
    private final String dateFormat;

    Date2StringConverters(String dateFormat) {
        this.dateFormat = dateFormat;
    }

    @Override
    public String convert(Date source) {
        return new SimpleDateFormat(this.dateFormat).format(source);
    }
}

public class ConverterTests {
    private final Converter<Date, String> converter1 = Date2StringConverters
.yyyy_MM_dd;
    private final Converter<Date, String> converter2 = Date2StringConverters
.yyyy_MM_dd_HH_mm_ss;
    private final Converter<Date, String> converter3 = Date2StringConverters
.HH_mm_ss;

    public void formatTest(Date date){
        System.out.println(converter1.convert(date));
        System.out.println(converter2.convert(date));
        System.out.println(converter3.convert(date));

    }
}
```

4.3. 状态机

状态机是解决业务流程中的一种有效手段，而枚举的单例性，为构建状态机提供了便利。

以下是一个订单的状态扭转流程，所涉及的状态包括 Created、Canceled、Confirmed、Overtime、Paied；所涉及的动作包括 cancel、confirm、timeout、pay。

```
graph TB
None{开始}--> |create|Created
Created-->|confirm|Confirmed
Created-->|cancel|Canceld
Confirmed-->|cancel|Canceld
Confirmed-->|timeout|Overtime
Confirmed-->|pay| Paied
```

```
// 状态操作接口，管理所有支持的动作
public interface IOrderState {
    void cancel(OrderStateContext context);

    void confirm(OrderStateContext context);

    void timeout(OrderStateContext context);

    void pay(OrderStateContext context);
}

// 状态机上下文
public interface OrderStateContext {
    void setStats(OrderState state);
}

// 订单实际实现
public class Order{
    private OrderState state;

    private void setStats(OrderState state) {
        this.state = state;
    }

    // 将请求转发给状态机
    public void cancel() {
        this.state.cancel(new StateContext());
    }

    // 将请求转发给状态机
    public void confirm() {
        this.state.confirm(new StateContext());
    }

    // 将请求转发给状态机
    public void timeout() {
        this.state.timeout(new StateContext());
    }

    // 将请求转发给状态机
    public void pay() {
        this.state.pay(new StateContext());
    }

    // 内部类，实现OrderStateContext，回写Order的状态
    class StateContext implements OrderStateContext{

        @Override
        public void setStats(OrderState state) {
            Order.this.setStats(state);
        }
    }
}

// 基于枚举的状态机实现
public enum OrderState implements IOrderState{
    CREATED{
        // 允许进行cancel操作，并把状态设置为CANCELD
        @Override
        public void cancel(OrderStateContext context){
            context.setStats(CANCELD);
        }

        // 允许进行confirm操作，并把状态设置为CONFIRMED
        @Override
        public void confirm(OrderStateContext context) {
            context.setStats(CONFIRMED);
        }

    },
    CONFIRMED{
```

```
            // 允许进行cancel操作，并把状态设置为CANCELD
            @Override
            public void cancel(OrderStateContext context) {
                context.setStats(CANCELD);
            }

            // 允许进行timeout操作，并把状态设置为OVERTIME
            @Override
            public void timeout(OrderStateContext context) {
                context.setStats(OVERTIME);
            }

            // 允许进行pay操作，并把状态设置为PAIED
            @Override
            public void pay(OrderStateContext context) {
                context.setStats(PAIED);
            }

        },
        // 最终状态，不允许任何操作
        CANCELD{

        },

        // 最终状态，不允许任何操作
        OVERTIME{

        },

        // 最终状态，不允许任何操作
        PAIED{

        };

        @Override
        public void cancel(OrderStateContext context) {
            throw new NotSupportedException();
        }

        @Override
        public void confirm(OrderStateContext context) {
            throw new NotSupportedException();
        }

        @Override
        public void timeout(OrderStateContext context) {
            throw new NotSupportedException();
        }

        @Override
        public void pay(OrderStateContext context) {
            throw new NotSupportedException();
        }
    }
```

4.4. 责任链

在责任链模式中，程序可以使用多种方式来处理一个问题，然后把他们链接起来，当一个请求进来后，他会遍历整个链，找到能够处理该请求的处理器并对请求进行处理。

枚举可以实现某个接口，加上其天生的单例特性，可以成为组织责任链处理器的一种方式。

```
// 消息类型
public enum MessageType {
    TEXT, BIN, XML, JSON;
}


// 定义的消息体
@Value
public class Message {
    private final MessageType type;
    private final Object object;

    public Message(MessageType type, Object object) {
        this.type = type;
        this.object = object;
    }
}

// 消息处理器
public interface MessageHandler {
    boolean handle(Message message);
}



// 基于枚举的处理器管理
public enum MessageHandlers implements MessageHandler{
    TEXT_HANDLER(MessageType.TEXT){
        @Override
        boolean doHandle(Message message) {
            System.out.println("text");
            return true;
        }
    },
    BIN_HANDLER(MessageType.BIN){
        @Override
        boolean doHandle(Message message) {
            System.out.println("bin");
            return true;
        }
    },
    XML_HANDLER(MessageType.XML){
        @Override
        boolean doHandle(Message message) {
            System.out.println("xml");
            return true;
        }
    },
    JSON_HANDLER(MessageType.JSON){
        @Override
        boolean doHandle(Message message) {
            System.out.println("json");
            return true;
        }
    };

    // 接受的类型
    private final MessageType acceptType;

    MessageHandlers(MessageType acceptType) {
        this.acceptType = acceptType;
    }

    // 抽象接口
    abstract boolean doHandle(Message message);

    // 如果消息体是接受类型，调用doHandle进行业务处理
    @Override
    public boolean handle(Message message) {
        return message.getType() == this.acceptType && doHandle(message);
    }
}



// 消息处理链
public class MessageHandlerChain {
    public boolean handle(Message message){
        for (MessageHandler handler : MessageHandlers.values()){
            if (handler.handle(message)){
                return true;
            }
        }
        return false;
    }
}
```

4.5. 分发器

分发器根据输入的数据，找到对应的处理器，并将请求转发给处理器进行处理。

由于 EnumMap 其出色的性能，特别适合根据特定类型作为分发策略的场景。

```
// 消息体
@Value
public class Message {
    private final MessageType type;
    private final Object data;

    public Message(MessageType type, Object data) {
        this.type = type;
        this.data = data;
    }
}

// 消息类型
public enum MessageType {
    // 登录
    LOGIN,
    // 进入房间
    ENTER_ROOM,
    // 退出房间
    EXIT_ROOM,
    // 登出
    LOGOUT;
}

// 消息处理器
public interface MessageHandler {
    void handle(Message message);
}


// 基于EnumMap的消息分发器
public class MessageDispatcher {
    private final Map<MessageType, MessageHandler> dispatcherMap =
            new EnumMap<MessageType, MessageHandler>(MessageType.class);

    public MessageDispatcher(){
        dispatcherMap.put(MessageType.LOGIN, message -> System.out.println("
Login"));
        dispatcherMap.put(MessageType.ENTER_ROOM, message -> System.out.prin
tln("Enter Room"));

        dispatcherMap.put(MessageType.EXIT_ROOM, message -> System.out.print
ln("Exit Room"));
        dispatcherMap.put(MessageType.LOGOUT, message -> System.out.println(
"Logout"));
    }

    public void dispatch(Message message){
        MessageHandler handler = this.dispatcherMap.get(message.getType());
        if (handler != null){
            handler.handle(message);
        }
    }
}
```

5. 总结

枚举本身并不复杂，主要理解编译器为我们所做的功能加强。究其本质，枚举只是一个特殊的类型，除了不能继承父类之外，拥有类的一切特性；加之其天生的单例性，可以有效的应用于一些特殊场景。