Java 基础（11）：枚举类型

一、枚举类型 Enum 的简介

1.什么是枚举类型

枚举类型： 就是由一组具有名的值的有限集合组成新的类型。(即新的类)。
好像还是不懂，别急，咱们先来看一下 为什么要引入枚举类型？
在没有引入枚举类型前，当我们想要维护一组常量集合时，我们是这样做的，看下面的例子：

```
class FavouriteColor_class{
    public static final  int RED   = 1;
    public static final  int BLACK = 3;
    public static final  int GREEN = 2;
    public static final  int BLUE  = 4;
    public static final  int WHITE = 5;
    public static final  int BROWN = 6;
}
```

当我们有枚举类型后，便可以简写成：

```
//枚举类型
public enum FavouriteColor {
    //枚举成员
    RED,GREEN,BLACK,BLUE,WHITE,BROWN
}
```

是不是很简单，很清晰。这样就可以省掉大量重复的代码，使得代码更加易于维护。

现在有点明白枚举类型的定义了吧！在说的再仔细一点，就是**使用关键字 enum 来用一组由常量组成的有限集合来创建一个新的 class 类** 。至于新的 class 类型，请继续往下看。

二、深入分析枚举的特性与实现原理

上面仅仅简单地介绍了枚举类型的最简单的用法，下面我们将逐步深入，掌握枚举类型的复杂的用法，以及其原理。

1. 枚举成员

上面的枚举类 FavouriteColor 里面的成员便都是枚举成员，换句话说，枚举成员就是枚举类中，没有任何类型修饰，只有变量名，也不能赋值的成员。
到这里还是对枚举成员很疑惑,我们先将上面的例子进行反编译一下：

```
public final class FavouriteColor extends Enum {
    public static final FavouriteColor RED;
    public static final FavouriteColor GREEN;
    public static final FavouriteColor BLACK;
    public static final FavouriteColor BLUE;
    public static final FavouriteColor WHITE;
    public static final FavouriteColor BROWN;
}
```

从反编译的结果可以看出，枚举成员都被处理成 public static final 的静态枚举常量。即上面例子的枚举成员都是枚举类 FavouriteColor 的实例。

2. 为枚举类型添加方法、构造器、非枚举的成员

枚举类型在添加方法、构造器、非枚举成员时，与普通类是没有多大的区别，除了以下几个限制：

- 枚举成员必须是最先声明 ,且只能用一行声明(相互间以逗号隔开 ,分号结束声明)。
- 构造器的访问权限只能是 private(可以不写 ,默认强制是 private) ,不能是 public、protected。

```
public enum FavouriteColor {
   //枚举成员
    RED, GREEN(2), BLACK(3), BLUE, WHITE, BROWN;// 必须要有分号
    // 非枚举类型的成员
    private int colorValue;
    public int aa;
    // 静态常量也可以
    public static final int cc = 2;
    //无参构造器
    private FavouriteColor() {
    }
       //有参构造器
    FavouriteColor(int colorValue) {
        this.colorValue = colorValue;
    }
   //方法
```

```
    public void print() {
        System.out.println(cc);
    }
}
```

可以看出，我们其实是可以使用 Eunm 类型做很多事情，虽然，我们一般只使用普通的枚举类型。

仔细看一下所有的枚举成员，我们会发现 GREEN(2), BLACK(3) 这两个枚举成员有点奇怪 !其实也很简单 ,前面说了 ,枚举成员其实就是枚举类型的实例 ,所以 ,GREEN(2), BLACK(3) 就是指明了用带参构造器，并传入参数，即可以理解成 FavouriteColor GREEN = new FavouriteColor(2)。其他几个枚举类型则表示使用无参构造器来创建对象。(事实上，编译器会重新创建每个构造器，为每个构造器多加两个参数)。

3. 包含抽象方法的枚举类型

枚举类型也是允许包含抽象方法的 (除了几个小限制外，枚举类几乎与普通类一样) , 那么包含抽象方法的枚举类型的枚举成员是怎么样的，编译器又是怎么处理的？

我们知道，上面的例子 FavouriteColor 类经过反编译后得到的类是一个继承了 Enum 的 final 类：

```
public final class FavouriteColor extends Enum
```

那么包含抽象方法的枚举类型是不是也是被编译器处理成 final 类，如果是这样，那有怎么被子类继承呢？ 还是处理成 abstract 类呢？

我们看个包含抽象方法的枚举类的例子 , Fruit 类中有三种水果 , 希望能为每种水果输出对应的信息：

```
public enum Frutit {
    APPLE {
        @Override
        public void printFruitInfo() {
           System.out.println("This is apple");
        }
    },BANANA {

        @Override
        public void printFruitInfo() {
            System.out.println("This is apple");
        }
    },WATERMELON {
        @Override
        public void printFruitInfo() {
            System.out.println("This is apple");
```

```
        }
    };

    //抽象方法
    public abstract void printFruitInfo();
    public static void main(String[] arg) {
        Frutit.APPLE.printFruitInfo();
    }
}
```

运行结果：

```
This is apple
```

对于上面的枚举成员的形式也很容易理解，因为枚举成员是一个枚举类型的实例，上面的这种形式就是一种匿名内部类的形式，即每个枚举成员的创建可以理解成：

```
BANANA = new Frutit("BANANA", 1) {//此构造器是编译器生成的，下面会说
        public void printFruitInfo() {//匿名内部类的抽象方法实现。
            System.out.println("This is apple");
        }
    };
```

事实上，编译器确实就是这样处理的，即上面的例子中，创建了三个匿名内部类，同时也会多创建三个 class 文件.
最后，我们反编译一下 fruit 类，看 fruit 类的定义：

```
public abstract class Frutit extends Enum
```

Fruit 类被处理成抽象类，所以可以说，**枚举类型经过编译器的处理，含抽象方法的将被处理成抽象类，否则处理成 final 类。**

4. 枚举类型的父类 – Enum

每一个枚举类型都继承了 Enum，所以是很有必要来了解一下 Enum；

```
public abstract class Enum<E extends Enum<E>>
        implements Comparable<E>, Serializable {
    //枚举成员的名称
    private final String name;
    //枚举成员的顺序，是按照定义的顺序，从 0 开始
    private final int ordinal;
    //构造方法
    protected Enum(String name, int ordinal) {
        this.name = name;
        this.ordinal = ordinal;
```

```
    }
    public final int ordinal() {//返回枚举常量的序数
        return ordinal;
    }
}

public final String name() {//返回此枚举常量的名称，在其枚举声明中对其进行声明。
     return name;
}
public final boolean equals(Object other) {
    return this==other;//比较地址
}
public final int hashCode() {
    return super.hashCode();
}
public final int compareTo(E o) {//返回枚举常量的序数
    //是按照次序 ordinal 来比较的
}
public static <T extends Enum<T>> T valueOf(Class<T> enumType,  String name) { }
public String toString() {
    return name;
}
```

以上都是一些可能会用到的方法，我们从上面可以发现两个有趣的地方：

- **Enum 类实现了 Serializable 接口**，也就是说可以枚举类型可以进行序列化。
- **Enum 的几乎所有方法都是 final 方法**，也就是说，枚举类型只能重写 toString()方法，其他方法不能重写，连 hashcode()、equal()等方法也不行。

5. 真正掌握枚举类型的原理

上面说了这么多，都是片面地、简单地理解了枚举类型，但还没有完全掌握枚举类型的本质，有了上面的基础，我们将如鱼得水。

想要真正理解枚举类型的本质，就得了解编译器是如何处理枚举类型的，也就是老办法 -- 反编译。这次看一个完整的反编译代码，先看一个例子：

```
public enum Fruit {
    APPLE ,BANANA ,WATERMELON ;
    private int value;
    private Fruit() {//默认构造器
       this.value = 0;
```

```
    }
    private Fruit(int value) {//带参数的构造器
        this.value = value;
    }
}
```

反编译的结果：

```
public final class Fruit extends Enum {
   //3 个枚举成员实例
    public static final Fruit APPLE;
    public static final Fruit BANANA;
    public static final Fruit WATERMELON;
    private int value;//普通变量
    private static final Fruit ENUM$VALUES[];//存储枚举常量的枚举数组
    static {//静态域，初始化枚举常量，枚举数组
        APPLE = new Fruit("APPLE", 0);
        BANANA = new Fruit("BANANA", 1);
        WATERMELON = new Fruit("WATERMELON", 2);
        ENUM$VALUES = (new Fruit[]{APPLE, BANANA, WATERMELON});
    }
    private Fruit(String s, int i) {//编译器改造了默认构造器
        super(s, i);
        value = 0;
    }
    private Fruit(String s, int i, int value) {//编译器改造了带参数的构造器
        super(s, i);
        this.value = value;
    }
    public static Fruit[] values() {//编译器添加了静态方法 values()
        Fruit afruit[];
        int i;
        Fruit afruit1[];
        System.arraycopy(afruit = ENUM$VALUES, 0, afruit1 = new Fruit[i =
afruit.length], 0, i);
        return afruit1;
    }
    public static Fruit valueOf(String s) {//编译器添加了静态方法 valueOf()
        return (Fruit) Enum.valueOf(Test_2018_1_16 / Fruit, s);
    }
}
```

从反编译的结果可以看出，编译器为我们创建出来的枚举类做了很多工作：

- **对枚举成员的处理**

编译器对所有的枚举成员处理成 public static final 的枚举常量，并在静态域中进行初始化。

- **构造器**

编译器重新定义了构造器，不仅为每个构造器都增加了两个参数，还添加了父类的构造方法调用。

- **添加了两个类方法**

编译器为枚举类添加了 values() 和 valueOf()。values()方法返回一个枚举类型的数组，可用于遍历枚举类型。valueOf()方法也是新增的，而且是重载了父类的 valueOf()方法。

注意了： 正因为枚举类型的真正构造器是再编译时才生成的，所以我们没法创建枚举类型的实例，以及继承扩展枚举类型（即使是被处理成 abstract 类）。枚举类型的实例只能由编译器来处理创建。

三、枚举类型的使用

1. switch

```
Fruit fruit = Fruit.APPLE;
    switch (fruit) {
    case APPLE:
        System.out.println("APPLE");
        break;
    case BANANA:
        System.out.println("BANANA");
        break;
    case WATERMELON:
        System.out.println("WATERMELON");
        break;
}
```

2. 实现接口

实现接口就不多说了。枚举类型继承了 Enum 类，所以不能再继承其他类，但可以实现接口。

3. 使用接口组织枚举

前面说了，枚举类型是无法被子类继承扩展的，这就造成无法满足以下两种情况的需求：

- 希望扩展原来的枚举类型中的元素；

- 希望使用子类对枚举类型中的元素进行分组；

看一个例子：对食物进行分类，大类是 Food,Food 下面有好几种食物类别，类别上才是具体的食物；

```
public interface Food {
    enum Appetizer implements Food {
        SALAD, SOUP, SPRING_ROLLS
    }
    enum Coffee implements Food {
        BLACK_COFFEE, DECAF_COFFEE, ESPERSSO, TEA;
    }
    enum Dessert implements Food {
        FRUIT, GELATO, TIRAMISU;
    }
}
```

接口 Food 作为一个大类，3 种枚举类型做为接口的子类；Food 管理着这些枚举类型。**对于枚举而言，实现接口是使其子类化的唯一办法**，所以嵌套在 Food 中的每个枚举类都实现了 Food 接口。从而“所有这东西都是某种类型的 Food”。

```
Food food = Food.Coffee.ESPERSSO;//ESPERSSO 不仅是 coffee,也属于大类 Food，达到分类的效果
```

4. 使用枚举来实现单例模式

对于序列化和反序列化，因为每一个枚举类型和枚举变量在 JVM 中都是唯一的，即 Java 在序列化和反序列化枚举时做了特殊的规定，枚举的 writeObject、readObject、readObjectNoData、writeReplace 和 readResolve 等方法是被编译器禁用的，因此，**对于枚举单例，是不存在实现序列化接口后调用 readObject 会破坏单例的问题。**所以，枚举单例是单例模式的最佳实现方式。

```
public enum  EnumSingletonDemo {
    SINGLETON;
    //其他方法、成员等
    public int otherMethod() {
        return 0;
    }
}
```

单例的使用方式：

```
int a = EnumSingletonDemo.SINGLETON.otherMethod();
```

四、EnumSet、EnumMap

此处只是简单地介绍这两个类的使用，并不深入分析其实现原理。

1. EnumSet

EnumSet 是一个抽象类，继承了 AbstractSet 类，其本质上就是一个 Set。**只不过，Enumset 是要与枚举类型一起使用的专用 Set 实现。枚举 set 中所有键都必须来自单个枚举类型，该枚举类型在创建 set 时显式或隐式地指定。**

```
public abstract class EnumSet<E extends Enum<E>> extends AbstractSet<E>
```

尽管 JDK 没有提供 EnumSet 的实现子类，但是 EnumSet 新增的方法都是 static 方法，而且这些方法都是用来创建一个 EnumSet 的对象。因此可以看做是一个对枚举中的元素进行操作的 Set，而且性能也很高。看下面的例子：

```
public static void main(String[] args) {
    //创建对象，并指定 EnumSet 存储的枚举类型
    EnumSet<FavouriteColor> set = EnumSet.allOf(FavouriteColor.class);
    //移除枚举元素
    set.remove(FavouriteColor.BLACK);
    set.remove(FavouriteColor.BLUE);
    for(FavouriteColor color : set) {//遍历 set
        System.out.println(color);
    }
}
```

运行结果：

```
RED
GREEN
WHITE
BROWN
```

EnumSet 不支持同步访问。实现线程安全的方式是：

```
Set<MyEnum> s = Collections.synchronizedSet(EnumSet.noneOf(MyEnum.class));
```

2. EnumMap

EnumMap 是一个类，同样也是**与枚举类型键一起使用的专用 Map 实现。枚举映射中所有键都必须来自单个枚举类型，该枚举类型在创建映射时显式或隐式地指定。枚举映射在内部表示为数组。此表示形式非常紧凑且高效。**

```
public class EnumMap<K extends Enum<K>, V> extends AbstractMap<K, V>
```

简单使用的例子：

```
public static void main(String[] args) {
    EnumMap< FavouriteColor,Integer> map = new EnumMap<>(FavouriteColor.class);
    map.put(FavouriteColor.BLACK,1 );
    map.put(FavouriteColor.BLUE, 2);
    map.put(FavouriteColor.BROWN, 3);
    System.out.println(map.get(FavouriteColor.BLACK));
}
```

同样，防止意外的同步操作：

```
Map<EnumKey, V> m = Collections.synchronizedMap(new EnumMap<EnumKey, V>(...));
```

- 枚举类型继承于 Enum 类，所以只能用实现接口，不能再继承其他类。
- 枚举类型会被编译器处理成抽象类（含抽象方法）或 final 类。
- 枚举成员都是 public static final 的枚举实例常量。枚举成员必须是最先声明,且只能声明一行（逗号隔开，分号结束）。
- 构造方法必须是 private，如果定义了有参的构造器，就要注意枚举成员的声明。没有定义构造方法时，编译器为枚举类自动添加的是一个带两个参数的构造方法,并不是无参构造器。
- 编译器会为枚举类添加 values() 和 valueOf()两个方法。
- 没有抽象方法的枚举类，被编译器处理成 final 类。如果是包含抽象方法的枚举类则被处理成抽象 abstract 类。
- Enum 实现了 Serializable 接口，并且几乎所有方法都是 final 方法。

1) 用于调用存储过程的对象是: C
   A.ResultSet
   B.DriverManager
   C.CallableStatemet
   D.PreparedStatement
2) 描述 forward 和 redirect 的区别
   forward 是服务器请求资源，服务器直接访问目标地址的 URL，目标地址可以接收 request 请求参数，然后把结果发给浏览器，浏览器根本不知道服务器发送的内容是从哪儿来的，所以它的地址栏中还是原来的地址。
   redirect 就是服务端根据逻辑,发送一个状态码,告诉浏览器重新去请求哪个地址，浏览器会重新进行请求，此时不能用 request 传值，浏览器的地址栏会变成新的地址。
3) 局部内部类是否可以访问非 final 变量？
   答案：不能访问局部的，可以访问成员变量（全局的）。

```
class Out{
    private String name = "out.name";
    void print(){
```

```
    final String work = "out.local.work";//若不是final的则不能被Animal 使用.
    int age=10;
    class Animal
    //定义一个局部内部类.只能在print()方法中使用.
    //局部类中不能使用外部的非final的局部变量.全局的可以.
    {
        public void eat(){
            System.out.println(work);//ok
            //age=20;error not final
            System.out.println(name);//ok.
        }
    }
    Animal local = new Animal();
    local.eat();
    }
}
```