1 枚举类基本特性

方法名称	方法描述	类型
values()	返回枚举实例数组	类方法
valueOf(String)	根据名字返回枚举实例	类方法
ordinal()	返回实例声明次序，从0开始	实例方法
name()	返回实例名字	实例方法

2 反编译枚举类

- 通过 jad 实现反编译
- 没有实现抽象方法的枚举类

```
public enum  Shrubbery implements IEnumtest{
    GROUND("ground",11), CRAWLING("crawling",12), HANGING("hanging",13);
    private String name;
    private Integer age;
    //构造器模式强制是 private，不能是其它
    Shrubbery(String name, Integer age) {
        this.name = name;
        this.age = age;
    }
    public String getName() {
        System.out.println("枚举类的实例方法！");
        return name;
    }
    public static void staticMethod() {
        System.out.println("枚举类中的静态方法！");
    }
    @Override
    public void interfaceMethod() {
        System.out.println("这个是枚举类实现的接口！");
    }
```

```
}
//以下是反编译后的枚举类，反编译后的类继承了 Enum 抽象类！并且是 final 类型，所以不能被继承
public final class Shrubbery extends Enum implements IEnumtest{
    //多了两个静态方法
    public static Shrubbery[] values(){
        return (Shrubbery[])$VALUES.clone();
    }
    //注意这里的 valueOf是 Shrubbery.valueOf(String)，它调用了Enum类的静态方法
Enum.valueOf(Class,String)!
    public static Shrubbery valueOf(String s){
        return (Shrubbery)Enum.valueOf(Shrubbery, s);
    }
    //构造器多了两个参数 s,i：这两个参数用于 ordinal() 和 name() 方法！:可在Enum类中查看其构造器
    private Shrubbery(String s, int i, String s1, Integer integer){
        super(s, i);
        name = s1;
        age = integer;
    }
    //生成了几个静态变量，类装载的时候，这几个变量都没赋实际值
    public static final Shrubbery GROUND;
    public static final Shrubbery CRAWLING;
    public static final Shrubbery HANGING;
    private static final Shrubbery $VALUES[];
    //到第一次初始化（6种方法）的时候才初始化这几个值，即半懒加载（懒加载是第一次调用目标，而这个随便调
用那个所有的都初始化）
    static{
        GROUND = new Shrubbery("GROUND", 0, "ground", Integer.valueOf(11));
        CRAWLING = new Shrubbery("CRAWLING", 1, "crawling", Integer.valueOf(12));
        HANGING = new Shrubbery("HANGING", 2, "hanging", Integer.valueOf(13));
        $VALUES = (new Shrubbery[] {
            GROUND, CRAWLING, HANGING
        });
    }
    public String getName(){
        System.out.println("\u679A\u4E3E\u7C7B\u7684\u5B9E\u4F8B\u65B9\u6CD5\uFF01");
        return name;
    }
    public static void staticMethod(){

 System.out.println("\u679A\u4E3E\u7C7B\u4E2D\u7684\u9759\u6001\u65B9\u6CD5\uFF01");
    }
    public void interfaceMethod(){

 System.out.println("\u8FD9\u4E2A\u662F\u679A\u4E3E\u7C7B\u5B9E\u73B0\u7684\u63A5\u53E3
\uFF01");
    }
    private String name;
    private Integer age;
}
```

- 有抽象方法的枚举类

```
public enum  Shrubbery implements IEnumtest{
```

```
    //有抽象方法，各个枚举实例必须实现该抽象方法
    GROUND("ground",11){
        public void abstractMethod() {
            System.out.println("ground");
        }
    }, CRAWLING("crawling",12){
        public void abstractMethod() {
            System.out.println("crawling");
        }
    }, HANGING("hanging",13){
        public void abstractMethod() {
            System.out.println("hanging");
        }
    };
    private String name;
    private Integer age;
    Shrubbery(String name, Integer age) {
        this.name = name;
        this.age = age;
    }
    public String getName() {
        System.out.println("枚举类的实例方法！");
        return name;
    }
    public static void staticMethod() {
        System.out.println("枚举类中的静态方法！");
    }
    //枚举类可以定义抽象方法！----------------------------------------！
    public abstract void abstractMethod();
    @Override
    public void interfaceMethod() {
        System.out.println("这个是枚举类实现的接口！");
    }
}
//反编译之后，原来的枚举类变成了----- 抽象类！
public abstract class Shrubbery extends Enum implements IEnumtest{
    public static Shrubbery[] values(){
        return (Shrubbery[])$VALUES.clone();
    }
    public static Shrubbery valueOf(String s){
        return (Shrubbery)Enum.valueOf(Shrubbery, s);
    }
    private Shrubbery(String s, int i, String s1, Integer integer){
        super(s, i);
        name = s1;
        age = integer;
    }
    public String getName(){
        System.out.println("\u679A\u4E3E\u7C7B\u7684\u5B9E\u4F8B\u65B9\u6CD5\uFF01");
        return name;
    }
    public static void staticMethod(){
```

```
 System.out.println("\u679A\u4E3E\u7C7B\u4E2D\u7684\u9759\u6001\u65B9\u6CD5\uFF01");
    }
    public void interfaceMethod(){

 System.out.println("\u8FD9\u4E2A\u662F\u679A\u4E3E\u7C7B\u5B9E\u73B0\u7684\u63A5\u53E3
\uFF01");
    }
    public abstract void abstractMethod();
    //同样产生几个静态变量
    public static final Shrubbery GROUND;
    public static final Shrubbery CRAWLING;
    public static final Shrubbery HANGING;
    private String name;
    private Integer age;
    private static final Shrubbery $VALUES[];
    static {
        //所以实现抽象类，必须实现抽象方法
        GROUND = new Shrubbery("GROUND", 0, "ground", Integer.valueOf(11)) {
            public void abstractMethod(){
                System.out.println("ground");
            }
        };
        CRAWLING = new Shrubbery("CRAWLING", 1, "crawling", Integer.valueOf(12)) {
            public void abstractMethod(){
                System.out.println("crawling");
            }
        };
        HANGING = new Shrubbery("HANGING", 2, "hanging", Integer.valueOf(13)) {
            public void abstractMethod(){
                System.out.println("hanging");
            }
        };
        $VALUES = (new Shrubbery[] {
            GROUND, CRAWLING, HANGING
        });
    }
}
//并且生成了3个继承了Shrubbery 抽象类的静态类！！
```

3 反编译枚举类的思考

3.1 枚举类和普通类相同点

- 枚举类和普通类用法基本相同；
- 枚举类也可以定义实例变量、类变量、实例方法、类方法，也可以实现其他接口；
- 和普通实例对象一样，枚举类也可以在实例中覆盖枚举类的实例方法。

3.2 枚举类和普通类不同点

- 枚举类只有私有的构造器，外界甚至不能通过反射创建！
- 枚举类只有固定的实例对象(GROUND，CRAWLING，HANGING)！！而普通类可以实例化任意多的对象；

- 枚举类各个实例，在初始化枚举类的时候统一实例化；
- 关键：枚举类继承了 Enum 抽象类，继承了这个抽象类的所有性质。
- 枚举在同一 JVM 中实现序列化后，仍然是单例的（普通类会破坏单例）。

4 枚举类线程安全说明

- 有上述代码可知，枚举类中的实例都是单例的
- 并且，所有实例都在第一次初始化的时候，统一由构造器初始化（构造器初始化的时候加了锁），所以是线程安全的

5 EnumSet

- EnumSet 是用来代替传统的基于 int 的“标志位”，这种标志位可以用来表示某种“开/关”信息；并且可以方便的表示多种状态的集合，而不用进行位操作来得到多种状态集合。
- EnumSet 中元素的顺序是按照枚举类实例声明的顺序来的，这个特性有时候很有用，比如：洗车枚举类中有很多实例（项目），客户可以随机选择多个项目，但是最终项目的执行还是按照正确流程来做的。

```
public enum Weather {
    刮风，晴天，下雨，下雪，雾霾，闪电，微风
}
public static void main(String[] args) {
    EnumSet<Weather> none = EnumSet.noneOf(Weather.class);
    EnumSet<Weather> all = EnumSet.allOf(Weather.class);
    EnumSet<Weather> mon = EnumSet.of(下雨,刮风);
    EnumSet<Weather> tue = EnumSet.range(雾霾,微风);
    EnumSet<Weather> wed = EnumSet.complementOf(tue);

    System.out.println("none: "+none);//[]
    System.out.println("all: "+all);//[刮风，晴天，下雨，下雪，雾霾，闪电，微风]
    System.out.println("mon: "+mon);//[刮风，下雨]，注意：这里顺序不是使用 EnumSet.of中定义的顺序，而是枚举类实例声明的顺序！
    System.out.println("tue: "+tue);//[雾霾，闪电，微风]
    System.out.println("wed: "+wed);//[刮风，晴天，下雨，下雪]
}
```

6 EnumMap

- EnumMap 和 EnumSet 一样，输出顺序也是实例的声明顺序

```
public class EnumMapTest {
    public static void main(String[] args) {
        EnumMap<Weather,Command> em = new EnumMap(Weather.class);
        em.put(下雪,()->{
            System.out.println("下雪啦！");
        });
        em.put(刮风,()->{
            System.out.println("刮风啦！");
        });
        for (Map.Entry<Weather, Command> entry : em.entrySet()) {
            System.out.print(entry.getKey()+": ");
```

```
                entry.getValue().action();
            }
            em.get(闪电).action();
        }
    }
    interface Command {void action();}
```

参考

反编译工具分析枚举的原理 java枚举类型的实现原理 为什么我墙裂建议大家使用枚举来实现单例 深度分析Java的枚举类型—-枚举的线程安全性及序列化问题 github 源码地址