枚举和注解

30、用 enum 代替 int 常量

int 枚举模式

```
public static final int APPLE_FUJI = 0;
public static final int APPLE_PIPPIN = 1;
public static final int APPLE_GRANNY_SMITH = 2;
public static final int ORANGE_NAVEL = 0;
public static final int ORANGE_TEMPLE = 1;
public static final int ORANGE_BLOD = 2;
```

Java 枚举背后的基本想法非常简单：它们是通过公有的静态 final 域为枚举常量导出实例的类。因为没有可以访问的构造器，枚举类型是真正的 final。

```
public enum PayrollDay {
  MONDAY(PayType.WEEKDAY), TUESDAY(PayType.WEEKDAY), WEDNESDAY(
      PayType.WEEKDAY), THURSDAY(PayType.WEEKDAY), FRIDAY(PayType.WEEKDAY), SATURDAY(
      PayType.WEEKEND), SUNDAY(PayType.WEEKEND);
  private final PayType payType;
  PayrollDay(PayType payType) {
    this.payType = payType;
  }
  double pay(double hoursWorked, double payRate) {
    return payType.pay(hoursWorked, payRate);
  }
  private enum PayType {
    WEEKDAY {
      double overtimePay(double hours, double payRate) {
        return hours <= HOURS_PER_SHIFT ? 0 : (hours - HOURS_PER_SHIFT)
            * payRate / 2;
      }
    },
    WEEKEND {
      double overtimePay(double hours, double payRate) {
        return hours * payRate / 2;
      }
    };
    private static final int HOURS_PER_SHIFT = 8;
    abstract double overtimePay(double hours, double payRate);
    double pay(double hoursWorked, double payRate) {
      double basePay = hoursWorked * payRate;
      return basePay + overtimePay(hoursWorked, payRate);
    }
  }
}
```

总之：与 int 常量相比，枚举的类型优势是不言而喻的。枚举要易读得多，也更加安全，功能更加强大。

31、用实例域代替序数

应该给 enum 添加 int 域，而不是使用 ordinal 方法来导出与枚举关联的序数值。(几乎不应使用 ordinal 方法，除非在编写像 EnumMap 这样的基于枚举的通用数据结构)

```
//错误
public enum Fruit{
    APPLE, PEAR, ORANGE;
    public int numberOfFruit(){
        return ordinal() + 1;
    }
}

//正确
public enum Fruit{
    APPLE(1), PEAR(2), ORANGE(3);
    private final int number;
    Fruit(int num) {number    = num;}
    public int numberOfFruit(){
        return number;
    }
}
```

32、用 EnumSet 代替位域

```
//WRONG（位域）
public class Text{
    private static final int STYLE_BOLD                 = 1 << 0;
    private static final int STYLE_ITALIC               = 1 << 1;
    private static final int STYLE_UNDERLINE        = 1 << 2;
    public void applyStyles(int styles) {...}
}
//use
text.applyStyles(STYLE_BOLD | STYLE_ITALIC);

//RIGHT
public class Text{
    public enum Style{STYLE_BOLD, STYLE_ITALIC, STYLE_UNDERLINE}
    public void applyStyles(Set<Style> styles) {...} //这里不使用 EnumSet<Style>参数是因为考虑到某些客户端可能会传递一些其他的 Set 实现
}
//use
text.applyStyles(EnumSet.of(STYLE_BOLD, STYLE_ITALIC));
```

33、用 EnumMap 代替序数索引

序数索引是指依赖于枚举成员在枚举中的序数来进行数组索引，如：

```
//定义了植物类，其中植物又分为水果，蔬菜，树木三种
public class Plant{
      public enum Type { Fruit, Vegetables, Tree}
      private final String name;
      private final Type type;

      Plant(String name, Type type){
            this.name = name;
            this.type = type;
      }
}

Set<Plant>[] plants = (Set<Plant>[]) new Set[Plant.Type.valuse().lenght];
//根据植物的类型，分别把所有的植物放入三个 set 中
for(int i = 0; i < plant.lenght; i++){
    plant[i] = new HashSet<Plant>();
}

for(Plant p : garden){   //garden 里放了所有的植物
     plant[p.type.ordinal()].add(p)   //反面教材：利用了枚举的序数来得到想要的数组索引，用户在其他地方可以不使用 ordinal 函数，而直接使用 int 值来访问，就可能出错
}
```

应该使用 EnumMap 来实现，EnumMap 内部是采用数组实现的，具有 Map 的丰富功能和类型安全以及数组的效率：

```
Map<Plant.Type, Set<Plant>> plants = new EnumMap<Plant.Type, Set<Plant>>(Plant.Type.class);   //构造函数需要 键 类型的 Class 对象
//根据植物的类型，分别把所有的植物放入三个 set 中
for(Plant.Type type : Plant.Type.valuse()){
    plant.put(type, new HashSet<Plant>);
}

for(Plant p : garden){   //garden 里放了所有的植物
     plant.get(p.type).add(p)   //用户必须使用正确的键值来访问，即 Type 类型
}
```

34、用接口模拟可以伸缩的枚举

由于在 java 中 enum 不是可扩展的，在某些情况下，可能需要对枚举进行扩展，比如操作类型(+-*/等)，就可以考虑：

(1)定义一个接口，比如 public interface Operation{...};

(2)使枚举继承接口：比如 public enum BasicOperation implements Operation{...}

(3)使用时的 API 写成接口(比如，T extends Enum<T> & Operation)，而不是实现（比如 BasicOperation ）

```
private static <T extends Enum<T> & Operation> void function(T t,..); //表示 T 既表示枚举又是 Operation 的子类型
```

(4)当需要扩展 BasicOperation 枚举时，就可以另写一个枚举，且 implements 接口 Operation

35、注解优先于命名模式

优先使用注解来表面针对某些程序元素的特定信息

36、坚持使用 Override 注解

在想要覆盖的方法上使用 Override 注解，编译器就可以帮助发现一些错误。可以不写 Override 的特例：在非抽象类中实现了父类的抽象方法，因为要是没有覆盖，则编译器就会发出错误。

37、用标记接口实现类型

(1)标记接口：没有包含方法声明的接口，只是指明某个类实现了具有某种属性的接口。比如 Serializable 接口

(2)标记注解：如下代码

```
@MyMarker    //这就是标记注解
   public static void myMeth() {
}
```

(2)标记接口与标记注解的最终要的区别在于：标记接口可以在编译时就检查到相应的类型问题，而标记注解则要到运行时

总之，标记接口和标记注解来比较，用标记接口更好。