<https://blog.csdn.net/javazejian/article/details/71333103>

**理解枚举类型**

枚举类型是Java 5中新增特性的一部分，它是一种特殊的数据类型，之所以特殊是因为它既是一种类(class)类型却又比类类型多了些特殊的约束，但是这些约束的存在也造就了枚举类型的简洁性、安全性以及便捷性。下面先来看看什么是枚举？如何定义枚举？

**枚举的定义**

回忆一下下面的程序，这是在没有枚举类型时定义常量常见的方式

/\*\*

\* Created by zejian on 2017/5/7.

\* Blog : http://blog.csdn.net/javazejian [原文地址,请尊重原创]

\* 使用普通方式定义日期常量

\*/

public class DayDemo {

public static final int MONDAY =1;

public static final int TUESDAY=2;

public static final int WEDNESDAY=3;

public static final int THURSDAY=4;

public static final int FRIDAY=5;

public static final int SATURDAY=6;

public static final int SUNDAY=7;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22

上述的常量定义常量的方式称为int枚举模式，这样的定义方式并没有什么错，但它存在许多不足，如在类型安全和使用方便性上并没有多少好处，如果存在定义int值相同的变量，混淆的几率还是很大的，编译器也不会提出任何警告，因此这种方式在枚举出现后并不提倡，现在我们利用枚举类型来重新定义上述的常量，同时也感受一把枚举定义的方式，如下定义周一到周日的常量

//枚举类型，使用关键字enum

enum Day {

MONDAY, TUESDAY, WEDNESDAY,

THURSDAY, FRIDAY, SATURDAY, SUNDAY

}

* 1
* 2
* 3
* 4
* 5

相当简洁，在定义枚举类型时我们使用的关键字是enum，与class关键字类似，只不过前者是定义枚举类型，后者是定义类类型。枚举类型Day中分别定义了从周一到周日的值，这里要注意，值一般是大写的字母，多个值之间以逗号分隔。同时我们应该知道的是枚举类型可以像类(class)类型一样，定义为一个单独的文件，当然也可以定义在其他类内部，更重要的是枚举常量在类型安全性和便捷性都很有保证，如果出现类型问题编译器也会提示我们改进，但务必记住枚举表示的类型其取值是必须有限的，也就是说每个值都是可以枚举出来的，比如上述描述的一周共有七天。那么该如何使用呢？如下：

/\*\*

\* Created by zejian on 2017/5/7.

\* Blog : http://blog.csdn.net/javazejian [原文地址,请尊重原创]

\*/

public class EnumDemo {

public static void main(String[] args){

//直接引用

Day day =Day.MONDAY;

}

}

//定义枚举类型

enum Day {

MONDAY, TUESDAY, WEDNESDAY,

THURSDAY, FRIDAY, SATURDAY, SUNDAY

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17

就像上述代码那样，直接引用枚举的值即可，这便是枚举类型的最简单模型。

**枚举实现原理**

我们大概了解了枚举类型的定义与简单使用后，现在有必要来了解一下枚举类型的基本实现原理。实际上在使用关键字enum创建枚举类型并编译后，编译器会为我们生成一个相关的类，这个类继承了Java API中的java.lang.Enum类，也就是说通过关键字enum创建枚举类型在编译后事实上也是一个类类型而且该类继承自java.lang.Enum类。下面我们编译前面定义的EnumDemo.java并查看生成的class文件来验证这个结论：

//查看目录下的java文件

zejian@zejiandeMBP enumdemo$ ls

EnumDemo.java

//利用javac命令编译EnumDemo.java

zejian@zejiandeMBP enumdemo$ javac EnumDemo.java

//查看生成的class文件，注意有Day.class和EnumDemo.class 两个

zejian@zejiandeMBP enumdemo$ ls

Day.class EnumDemo.class EnumDemo.java

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8

利用javac编译前面定义的EnumDemo.java文件后分别生成了Day.class和EnumDemo.class文件，而Day.class就是枚举类型，这也就验证前面所说的使用关键字enum定义枚举类型并编译后，编译器会自动帮助我们生成一个与枚举相关的类。我们再来看看反编译Day.class文件：

//反编译Day.class

final class Day extends Enum

{

//编译器为我们添加的静态的values()方法

public static Day[] values()

{

return (Day[])$VALUES.clone();

}

//编译器为我们添加的静态的valueOf()方法，注意间接调用了Enum也类的valueOf方法

public static Day valueOf(String s)

{

return (Day)Enum.valueOf(com/zejian/enumdemo/Day, s);

}

//私有构造函数

private Day(String s, int i)

{

super(s, i);

}

//前面定义的7种枚举实例

public static final Day MONDAY;

public static final Day TUESDAY;

public static final Day WEDNESDAY;

public static final Day THURSDAY;

public static final Day FRIDAY;

public static final Day SATURDAY;

public static final Day SUNDAY;

private static final Day $VALUES[];

static

{

//实例化枚举实例

MONDAY = new Day("MONDAY", 0);

TUESDAY = new Day("TUESDAY", 1);

WEDNESDAY = new Day("WEDNESDAY", 2);

THURSDAY = new Day("THURSDAY", 3);

FRIDAY = new Day("FRIDAY", 4);

SATURDAY = new Day("SATURDAY", 5);

SUNDAY = new Day("SUNDAY", 6);

$VALUES = (new Day[] {

MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY

});

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43

从反编译的代码可以看出编译器确实帮助我们生成了一个Day类(注意该类是final类型的，将无法被继承)而且该类继承自java.lang.Enum类，该类是一个抽象类(稍后我们会分析该类中的主要方法)，除此之外，编译器还帮助我们生成了7个Day类型的实例对象分别对应枚举中定义的7个日期，这也充分说明了我们前面使用关键字enum定义的Day类型中的每种日期枚举常量也是实实在在的Day实例对象，只不过代表的内容不一样而已。注意编译器还为我们生成了两个静态方法，分别是values()和 valueOf()，稍后会分析它们的用法，到此我们也就明白了，使用关键字enum定义的枚举类型，在编译期后，也将转换成为一个实实在在的类，而在该类中，会存在每个在枚举类型中定义好变量的对应实例对象，如上述的MONDAY枚举类型对应public static final Day MONDAY;，同时编译器会为该类创建两个方法，分别是values()和valueOf()。ok~，到此相信我们对枚举的实现原理也比较清晰，下面我们深入了解一下java.lang.Enum类以及values()和valueOf()的用途。

**枚举的常见方法**

**Enum抽象类常见方法**

Enum是所有 Java 语言枚举类型的公共基本类（注意Enum是抽象类），以下是它的常见方法：

| **返回类型** | **方法名称** | **方法说明** |
| --- | --- | --- |
| int | compareTo(E o) | 比较此枚举与指定对象的顺序 |
| boolean | equals(Object other) | 当指定对象等于此枚举常量时，返回 true。 |
| Class<?> | getDeclaringClass() | 返回与此枚举常量的枚举类型相对应的 Class 对象 |
| 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 |
| int | ordinal() | 返回枚举常量的序数（它在枚举声明中的位置，其中初始常量序数为零） |
| String | toString() | 返回枚举常量的名称，它包含在声明中 |
| static<T extends Enum<T>> T | static valueOf(Class<T> enumType, String name) | 返回带指定名称的指定枚举类型的枚举常量。 |

这里主要说明一下ordinal()方法，该方法获取的是枚举变量在枚举类中声明的顺序，下标从0开始，如日期中的MONDAY在第一个位置，那么MONDAY的ordinal值就是0，如果MONDAY的声明位置发生变化，那么ordinal方法获取到的值也随之变化，注意在大多数情况下我们都不应该首先使用该方法，毕竟它总是变幻莫测的。compareTo(E o)方法则是比较枚举的大小，注意其内部实现是根据每个枚举的ordinal值大小进行比较的。name()方法与toString()几乎是等同的，都是输出变量的字符串形式。至于valueOf(Class<T> enumType, String name)方法则是根据枚举类的Class对象和枚举名称获取枚举常量，注意该方法是静态的，后面在枚举单例时，我们还会详细分析该方法，下面的代码演示了上述方法：

package com.zejian.enumdemo;

/\*\*

\* Created by zejian on 2017/5/7.

\* Blog : http://blog.csdn.net/javazejian [原文地址,请尊重原创]

\*/

public class EnumDemo {

public static void main(String[] args){

//创建枚举数组

Day[] days=new Day[]{Day.MONDAY, Day.TUESDAY, Day.WEDNESDAY,

Day.THURSDAY, Day.FRIDAY, Day.SATURDAY, Day.SUNDAY};

for (int i = 0; i <days.length ; i++) {

System.out.println("day["+i+"].ordinal():"+days[i].ordinal());

}

System.out.println("-------------------------------------");

//通过compareTo方法比较,实际上其内部是通过ordinal()值比较的

System.out.println("days[0].compareTo(days[1]):"+days[0].compareTo(days[1]));

System.out.println("days[0].compareTo(days[1]):"+days[0].compareTo(days[2]));

//获取该枚举对象的Class对象引用,当然也可以通过getClass方法

Class<?> clazz = days[0].getDeclaringClass();

System.out.println("clazz:"+clazz);

System.out.println("-------------------------------------");

//name()

System.out.println("days[0].name():"+days[0].name());

System.out.println("days[1].name():"+days[1].name());

System.out.println("days[2].name():"+days[2].name());

System.out.println("days[3].name():"+days[3].name());

System.out.println("-------------------------------------");

System.out.println("days[0].toString():"+days[0].toString());

System.out.println("days[1].toString():"+days[1].toString());

System.out.println("days[2].toString():"+days[2].toString());

System.out.println("days[3].toString():"+days[3].toString());

System.out.println("-------------------------------------");

Day d=Enum.valueOf(Day.class,days[0].name());

Day d2=Day.valueOf(Day.class,days[0].name());

System.out.println("d:"+d);

System.out.println("d2:"+d2);

}

/\*\*

执行结果:

day[0].ordinal():0

day[1].ordinal():1

day[2].ordinal():2

day[3].ordinal():3

day[4].ordinal():4

day[5].ordinal():5

day[6].ordinal():6

-------------------------------------

days[0].compareTo(days[1]):-1

days[0].compareTo(days[1]):-2

clazz:class com.zejian.enumdemo.Day

-------------------------------------

days[0].name():MONDAY

days[1].name():TUESDAY

days[2].name():WEDNESDAY

days[3].name():THURSDAY

-------------------------------------

days[0].toString():MONDAY

days[1].toString():TUESDAY

days[2].toString():WEDNESDAY

days[3].toString():THURSDAY

-------------------------------------

d:MONDAY

d2:MONDAY

\*/

}

enum Day {

MONDAY, TUESDAY, WEDNESDAY,

THURSDAY, FRIDAY, SATURDAY, SUNDAY

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83

到此对于抽象类Enum类的基本内容就介绍完了，这里提醒大家一点，Enum类内部会有一个构造函数，该构造函数只能有编译器调用，我们是无法手动操作的，不妨看看Enum类的主要源码：

//实现了Comparable

public abstract class Enum<E extends Enum<E>>

implements Comparable<E>, Serializable {

private final String name; //枚举字符串名称

public final String name() {

return name;

}

private final int ordinal;//枚举顺序值

public final int ordinal() {

return ordinal;

}

//枚举的构造方法，只能由编译器调用

protected Enum(String name, int ordinal) {

this.name = name;

this.ordinal = ordinal;

}

public String toString() {

return name;

}

public final boolean equals(Object other) {

return this==other;

}

//比较的是ordinal值

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;//根据ordinal值比较大小

}

@SuppressWarnings("unchecked")

public final Class<E> getDeclaringClass() {

//获取class对象引用，getClass()是Object的方法

Class<?> clazz = getClass();

//获取父类Class对象引用

Class<?> zuper = clazz.getSuperclass();

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

}

public static <T extends Enum<T>> T valueOf(Class<T> enumType,

String name) {

//enumType.enumConstantDirectory()获取到的是一个map集合，key值就是name值，value则是枚举变量值

//enumConstantDirectory是class对象内部的方法，根据class对象获取一个map集合的值

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);

}

//.....省略其他没用的方法

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65

通过Enum源码，可以知道，Enum实现了Comparable接口，这也是可以使用compareTo比较的原因，当然Enum构造函数也是存在的，该函数只能由编译器调用，毕竟我们只能使用enum关键字定义枚举，其他事情就放心交给编译器吧。

//由编译器调用

protected Enum(String name, int ordinal) {

this.name = name;

this.ordinal = ordinal;

}

* 1
* 2
* 3
* 4
* 5
* 6

**编译器生成的Values方法与ValueOf方法**

values()方法和valueOf(String name)方法是编译器生成的static方法，因此从前面的分析中，在Enum类中并没出现values()方法，但valueOf()方法还是有出现的，只不过编译器生成的valueOf()方法需传递一个name参数，而Enum自带的静态方法valueOf()则需要传递两个方法，从前面反编译后的代码可以看出，编译器生成的valueOf方法最终还是调用了Enum类的valueOf方法，下面通过代码来演示这两个方法的作用：

Day[] days2 = Day.values();

System.out.println("day2:"+Arrays.toString(days2));

Day day = Day.valueOf("MONDAY");

System.out.println("day:"+day);

/\*\*

输出结果:

day2:[MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY]

day:MONDAY

\*/

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

从结果可知道，values()方法的作用就是获取枚举类中的所有变量，并作为数组返回，而valueOf(String name)方法与Enum类中的valueOf方法的作用类似根据名称获取枚举变量，只不过编译器生成的valueOf方法更简洁些只需传递一个参数。这里我们还必须注意到，由于values()方法是由编译器插入到枚举类中的static方法，所以如果我们将枚举实例向上转型为Enum，那么values()方法将无法被调用，因为Enum类中并没有values()方法，valueOf()方法也是同样的道理，注意是一个参数的。

//正常使用

Day[] ds=Day.values();

//向上转型Enum

Enum e = Day.MONDAY;

//无法调用,没有此方法

//e.values();

* 1
* 2
* 3
* 4
* 5
* 6

**枚举与Class对象**

上述我们提到当枚举实例向上转型为Enum类型后，values()方法将会失效，也就无法一次性获取所有枚举实例变量，但是由于Class对象的存在，即使不使用values()方法，还是有可能一次获取到所有枚举实例变量的，在Class对象中存在如下方法：

| **返回类型** | **方法名称** | **方法说明** |
| --- | --- | --- |
| T[] | getEnumConstants() | 返回该枚举类型的所有元素，如果Class对象不是枚举类型，则返回null。 |
| boolean | isEnum() | 当且仅当该类声明为源代码中的枚举时返回 true |

因此通过getEnumConstants()方法，同样可以轻而易举地获取所有枚举实例变量下面通过代码来演示这个功能：

//正常使用

Day[] ds=Day.values();

//向上转型Enum

Enum e = Day.MONDAY;

//无法调用,没有此方法

//e.values();

//获取class对象引用

Class<?> clasz = e.getDeclaringClass();

if(clasz.isEnum()) {

Day[] dsz = (Day[]) clasz.getEnumConstants();

System.out.println("dsz:"+Arrays.toString(dsz));

}

/\*\*

输出结果:

dsz:[MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY]

\*/

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17

正如上述代码所展示，通过Enum的class对象的getEnumConstants方法，我们仍能一次性获取所有的枚举实例常量。

**枚举的进阶用法**

在前面的分析中，我们都是基于简单枚举类型的定义，也就是在定义枚举时只定义了枚举实例类型，并没定义方法或者成员变量，实际上使用关键字enum定义的枚举类，除了不能使用继承(因为编译器会自动为我们继承Enum抽象类而Java只支持单继承，因此枚举类是无法手动实现继承的)，可以把enum类当成常规类，也就是说我们可以向enum类中添加方法和变量，甚至是mian方法，下面就来感受一把。

**向enum类添加方法与自定义构造函数**

重新定义一个日期枚举类，带有desc成员变量描述该日期的对于中文描述，同时定义一个getDesc方法，返回中文描述内容，自定义私有构造函数，在声明枚举实例时传入对应的中文描述，代码如下：

package com.zejian.enumdemo;

/\*\*

\* Created by zejian on 2017/5/8.

\* Blog : http://blog.csdn.net/javazejian [原文地址,请尊重原创]

\*/

public enum Day2 {

MONDAY("星期一"),

TUESDAY("星期二"),

WEDNESDAY("星期三"),

THURSDAY("星期四"),

FRIDAY("星期五"),

SATURDAY("星期六"),

SUNDAY("星期日");//记住要用分号结束

private String desc;//中文描述

/\*\*

\* 私有构造,防止被外部调用

\* @param desc

\*/

private Day2(String desc){

this.desc=desc;

}

/\*\*

\* 定义方法,返回描述,跟常规类的定义没区别

\* @return

\*/

public String getDesc(){

return desc;

}

public static void main(String[] args){

for (Day2 day:Day2.values()) {

System.out.println("name:"+day.name()+

",desc:"+day.getDesc());

}

}

/\*\*

输出结果:

name:MONDAY,desc:星期一

name:TUESDAY,desc:星期二

name:WEDNESDAY,desc:星期三

name:THURSDAY,desc:星期四

name:FRIDAY,desc:星期五

name:SATURDAY,desc:星期六

name:SUNDAY,desc:星期日

\*/

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51

从上述代码可知，在enum类中确实可以像定义常规类一样声明变量或者成员方法。但是我们必须注意到，如果打算在enum类中定义方法，务必在声明完枚举实例后使用分号分开，倘若在枚举实例前定义任何方法，编译器都将会报错，无法编译通过，同时即使自定义了构造函数且enum的定义结束，我们也永远无法手动调用构造函数创建枚举实例，毕竟这事只能由编译器执行。

**关于覆盖enum类方法**

既然enum类跟常规类的定义没什么区别（实际上enum还是有些约束的），那么覆盖父类的方法也不会是什么难说，可惜的是父类Enum中的定义的方法只有toString方法没有使用final修饰，因此只能覆盖toString方法，如下通过覆盖toString省去了getDesc方法：

package com.zejian.enumdemo;

/\*\*

\* Created by zejian on 2017/5/8.

\* Blog : http://blog.csdn.net/javazejian [原文地址,请尊重原创]

\*/

public enum Day2 {

MONDAY("星期一"),

TUESDAY("星期二"),

WEDNESDAY("星期三"),

THURSDAY("星期四"),

FRIDAY("星期五"),

SATURDAY("星期六"),

SUNDAY("星期日");//记住要用分号结束

private String desc;//中文描述

/\*\*

\* 私有构造,防止被外部调用

\* @param desc

\*/

private Day2(String desc){

this.desc=desc;

}

/\*\*

\* 覆盖

\* @return

\*/

@Override

public String toString() {

return desc;

}

public static void main(String[] args){

for (Day2 day:Day2.values()) {

System.out.println("name:"+day.name()+

",desc:"+day.toString());

}

}

/\*\*

输出结果:

name:MONDAY,desc:星期一

name:TUESDAY,desc:星期二

name:WEDNESDAY,desc:星期三

name:THURSDAY,desc:星期四

name:FRIDAY,desc:星期五

name:SATURDAY,desc:星期六

name:SUNDAY,desc:星期日

\*/

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53

**enum类中定义抽象方法**

与常规抽象类一样，enum类允许我们为其定义抽象方法，然后使每个枚举实例都实现该方法，以便产生不同的行为方式，注意abstract关键字对于枚举类来说并不是必须的如下：

package com.zejian.enumdemo;

/\*\*

\* Created by zejian on 2017/5/9.

\* Blog : http://blog.csdn.net/javazejian [原文地址,请尊重原创]

\*/

public enum EnumDemo3 {

FIRST{

@Override

public String getInfo() {

return "FIRST TIME";

}

},

SECOND{

@Override

public String getInfo() {

return "SECOND TIME";

}

}

;

/\*\*

\* 定义抽象方法

\* @return

\*/

public abstract String getInfo();

//测试

public static void main(String[] args){

System.out.println("F:"+EnumDemo3.FIRST.getInfo());

System.out.println("S:"+EnumDemo3.SECOND.getInfo());

/\*\*

输出结果:

F:FIRST TIME

S:SECOND TIME

\*/

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40

通过这种方式就可以轻而易举地定义每个枚举实例的不同行为方式。我们可能注意到，enum类的实例似乎表现出了多态的特性，可惜的是枚举类型的实例终究不能作为类型传递使用，就像下面的使用方式，编译器是不可能答应的：

//无法通过编译,毕竟EnumDemo3.FIRST是个实例对象

public void text(EnumDemo3.FIRST instance){ }

* 1
* 2

在枚举实例常量中定义抽象方法

**enum类与接口**

由于Java单继承的原因，enum类并不能再继承其它类，但并不妨碍它实现接口，因此enum类同样是可以实现多接口的，如下：

package com.zejian.enumdemo;

/\*\*

\* Created by zejian on 2017/5/8.

\* Blog : http://blog.csdn.net/javazejian [原文地址,请尊重原创]

\*/

interface food{

void eat();

}

interface sport{

void run();

}

public enum EnumDemo2 implements food ,sport{

FOOD,

SPORT,

; //分号分隔

@Override

public void eat() {

System.out.println("eat.....");

}

@Override

public void run() {

System.out.println("run.....");

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30

有时候，我们可能需要对一组数据进行分类，比如进行食物菜单分类而且希望这些菜单都属于food类型，appetizer(开胃菜)、mainCourse(主菜)、dessert(点心)、Coffee等，每种分类下有多种具体的菜式或食品，此时可以利用接口来组织，如下(代码引用自Thinking in Java)：

public interface Food {

enum Appetizer implements Food {

SALAD, SOUP, SPRING\_ROLLS;

}

enum MainCourse implements Food {

LASAGNE, BURRITO, PAD\_THAI,

LENTILS, HUMMOUS, VINDALOO;

}

enum Dessert implements Food {

TIRAMISU, GELATO, BLACK\_FOREST\_CAKE,

FRUIT, CREME\_CARAMEL;

}

enum Coffee implements Food {

BLACK\_COFFEE, DECAF\_COFFEE, ESPRESSO,

LATTE, CAPPUCCINO, TEA, HERB\_TEA;

}

}

public class TypeOfFood {

public static void main(String[] args) {

Food food = Appetizer.SALAD;

food = MainCourse.LASAGNE;

food = Dessert.GELATO;

food = Coffee.CAPPUCCINO;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26

通过这种方式可以很方便组织上述的情景，同时确保每种具体类型的食物也属于Food，现在我们利用一个枚举嵌套枚举的方式，把前面定义的菜谱存放到一个Meal菜单中，通过这种方式就可以统一管理菜单的数据了。

public enum Meal{

APPETIZER(Food.Appetizer.class),

MAINCOURSE(Food.MainCourse.class),

DESSERT(Food.Dessert.class),

COFFEE(Food.Coffee.class);

private Food[] values;

private Meal(Class<? extends Food> kind) {

//通过class对象获取枚举实例

values = kind.getEnumConstants();

}

public interface Food {

enum Appetizer implements Food {

SALAD, SOUP, SPRING\_ROLLS;

}

enum MainCourse implements Food {

LASAGNE, BURRITO, PAD\_THAI,

LENTILS, HUMMOUS, VINDALOO;

}

enum Dessert implements Food {

TIRAMISU, GELATO, BLACK\_FOREST\_CAKE,

FRUIT, CREME\_CARAMEL;

}

enum Coffee implements Food {

BLACK\_COFFEE, DECAF\_COFFEE, ESPRESSO,

LATTE, CAPPUCCINO, TEA, HERB\_TEA;

}

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28

**枚举与switch**

关于枚举与switch是个比较简单的话题，使用switch进行条件判断时，条件参数一般只能是整型，字符型。而枚举型确实也被switch所支持，在java 1.7后switch也对字符串进行了支持。这里我们简单看一下switch与枚举类型的使用：

/\*\*

\* Created by zejian on 2017/5/9.

\* Blog : http://blog.csdn.net/javazejian [原文地址,请尊重原创]

\*/

enum Color {GREEN,RED,BLUE}

public class EnumDemo4 {

public static void printName(Color color){

switch (color){

case BLUE: //无需使用Color进行引用

System.out.println("蓝色");

break;

case RED:

System.out.println("红色");

break;

case GREEN:

System.out.println("绿色");

break;

}

}

public static void main(String[] args){

printName(Color.BLUE);

printName(Color.RED);

printName(Color.GREEN);

//蓝色

//红色

//绿色

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34

需要注意的是使用在于switch条件进行结合使用时，无需使用Color引用。

**枚举与单例模式**

单例模式可以说是最常使用的设计模式了，它的作用是确保某个类只有一个实例，自行实例化并向整个系统提供这个实例。在实际应用中，线程池、缓存、日志对象、对话框对象常被设计成单例，总之，选择单例模式就是为了避免不一致状态，下面我们将会简单说明单例模式的几种主要编写方式，从而对比出使用枚举实现单例模式的优点。首先看看饿汉式的单例模式：

/\*\*

\* Created by wuzejian on 2017/5/9.

\* 饿汉式（基于classloder机制避免了多线程的同步问题）

\*/

public class SingletonHungry {

private static SingletonHungry instance = new SingletonHungry();

private SingletonHungry() {

}

public static SingletonHungry getInstance() {

return instance;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

显然这种写法比较简单，但问题是无法做到延迟创建对象，事实上如果该单例类涉及资源较多，创建比较耗时间时，我们更希望它可以尽可能地延迟加载，从而减小初始化的负载，于是便有了如下的懒汉式单例：

/\*\*

\* Created by wuzejian on 2017/5/9..

\* 懒汉式单例模式（适合多线程安全）

\*/

public class SingletonLazy {

private static volatile SingletonLazy instance;

private SingletonLazy() {

}

public static synchronized SingletonLazy getInstance() {

if (instance == null) {

instance = new SingletonLazy();

}

return instance;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18

这种写法能够在多线程中很好的工作避免同步问题，同时也具备lazy loading机制，遗憾的是，由于synchronized的存在，效率很低，在单线程的情景下，完全可以去掉synchronized，为了兼顾效率与性能问题，改进后代码如下：

public class Singleton {

private static volatile Singleton singleton = null;

private Singleton(){}

public static Singleton getSingleton(){

if(singleton == null){

synchronized (Singleton.class){

if(singleton == null){

singleton = new Singleton();

}

}

}

return singleton;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

这种编写方式被称为“双重检查锁”，主要在getSingleton()方法中，进行两次null检查。这样可以极大提升并发度，进而提升性能。毕竟在单例中new的情况非常少，绝大多数都是可以并行的读操作，因此在加锁前多进行一次null检查就可以减少绝大多数的加锁操作，也就提高了执行效率。但是必须注意的是volatile关键字，该关键字有两层语义。第一层语义是可见性，可见性是指在一个线程中对该变量的修改会马上由工作内存（Work Memory）写回主内存（Main Memory），所以其它线程会马上读取到已修改的值，关于工作内存和主内存可简单理解为高速缓存（直接与CPU打交道）和主存（日常所说的内存条），注意工作内存是线程独享的，主存是线程共享的。volatile的第二层语义是禁止指令重排序优化，我们写的代码（特别是多线程代码），由于编译器优化，在实际执行的时候可能与我们编写的顺序不同。编译器只保证程序执行结果与源代码相同，却不保证实际指令的顺序与源代码相同，这在单线程并没什么问题，然而一旦引入多线程环境，这种乱序就可能导致严重问题。volatile关键字就可以从语义上解决这个问题，值得关注的是volatile的禁止指令重排序优化功能在Java 1.5后才得以实现，因此1.5前的版本仍然是不安全的，即使使用了volatile关键字。或许我们可以利用静态内部类来实现更安全的机制，静态内部类单例模式如下：

/\*\*

\* Created by wuzejian on 2017/5/9.

\* 静态内部类

\*/

public class SingletonInner {

private static class Holder {

private static SingletonInner singleton = new SingletonInner();

}

private SingletonInner(){}

public static SingletonInner getSingleton(){

return Holder.singleton;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15

正如上述代码所展示的，我们把Singleton实例放到一个静态内部类中，这样可以避免了静态实例在Singleton类的加载阶段（类加载过程的其中一个阶段的，此时只创建了Class对象，关于Class对象可以看博主另外一篇博文，[深入理解Java类型信息(Class对象)与反射机制](http://blog.csdn.net/javazejian/article/details/70768369)）就创建对象，毕竟静态变量初始化是在SingletonInner类初始化时触发的，并且由于静态内部类只会被加载一次，所以这种写法也是线程安全的。从上述4种单例模式的写法中，似乎也解决了效率与懒加载的问题，但是它们都有两个共同的缺点：

* 序列化可能会破坏单例模式，比较每次反序列化一个序列化的对象实例时都会创建一个新的实例，解决方案如下：
* //测试例子(四种写解决方式雷同)
* public class Singleton implements java.io.Serializable {
* public static Singleton INSTANCE = new Singleton();
* protected Singleton() {
* }
* //反序列时直接返回当前INSTANCE
* private Object readResolve() {
* return INSTANCE;
* }

}

* + 1
  + 2
  + 3
  + 4
  + 5
  + 6
  + 7
  + 8
  + 9
  + 10
  + 11
  + 12
* 使用反射强行调用私有构造器，解决方式可以修改构造器，让它在创建第二个实例的时候抛异常，如下：
* public static Singleton INSTANCE = new Singleton();
* private static volatile boolean flag = true;
* private Singleton(){
* if(flag){
* flag = false;
* }else{
* throw new RuntimeException("The instance already exists ！");
* }

}

* + 1
  + 2
  + 3
  + 4
  + 5
  + 6
  + 7
  + 8
  + 9

如上所述，问题确实也得到了解决，但问题是我们为此付出了不少努力，即添加了不少代码，还应该注意到如果单例类维持了其他对象的状态时还需要使他们成为transient的对象，这种就更复杂了，那有没有更简单更高效的呢？当然是有的，那就是枚举单例了，先来看看如何实现：

/\*\*

\* Created by wuzejian on 2017/5/9.

\* 枚举单利

\*/

public enum SingletonEnum {

INSTANCE;

private String name;

public String getName(){

return name;

}

public void setName(String name){

this.name = name;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

代码相当简洁，我们也可以像常规类一样编写enum类，为其添加变量和方法，访问方式也更简单，使用SingletonEnum.INSTANCE进行访问，这样也就避免调用getInstance方法，更重要的是使用枚举单例的写法，我们完全不用考虑序列化和反射的问题。枚举序列化是由jvm保证的，每一个枚举类型和定义的枚举变量在JVM中都是唯一的，在枚举类型的序列化和反序列化上，Java做了特殊的规定：在序列化时Java仅仅是将枚举对象的name属性输出到结果中，反序列化的时候则是通过java.lang.Enum的valueOf方法来根据名字查找枚举对象。同时，编译器是不允许任何对这种序列化机制的定制的并禁用了writeObject、readObject、readObjectNoData、writeReplace和readResolve等方法，从而保证了枚举实例的唯一性，这里我们不妨再次看看Enum类的valueOf方法：

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);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

实际上通过调用enumType(Class对象的引用)的enumConstantDirectory方法获取到的是一个Map集合，在该集合中存放了以枚举name为key和以枚举实例变量为value的Key&Value数据，因此通过name的值就可以获取到枚举实例，看看enumConstantDirectory方法源码：

Map<String, T> enumConstantDirectory() {

if (enumConstantDirectory == null) {

//getEnumConstantsShared最终通过反射调用枚举类的values方法

T[] universe = getEnumConstantsShared();

if (universe == null)

throw new IllegalArgumentException(

getName() + " is not an enum type");

Map<String, T> m = new HashMap<>(2 \* universe.length);

//map存放了当前enum类的所有枚举实例变量，以name为key值

for (T constant : universe)

m.put(((Enum<?>)constant).name(), constant);

enumConstantDirectory = m;

}

return enumConstantDirectory;

}

private volatile transient Map<String, T> enumConstantDirectory = null;

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

到这里我们也就可以看出枚举序列化确实不会重新创建新实例，jvm保证了每个枚举实例变量的唯一性。再来看看反射到底能不能创建枚举，下面试图通过反射获取构造器并创建枚举

public static void main(String[] args) throws IllegalAccessException, InvocationTargetException, InstantiationException, NoSuchMethodException {

//获取枚举类的构造函数(前面的源码已分析过)

Constructor<SingletonEnum> constructor=SingletonEnum.class.getDeclaredConstructor(String.class,int.class);

constructor.setAccessible(true);

//创建枚举

SingletonEnum singleton=constructor.newInstance("otherInstance",9);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7

执行报错

Exception in thread "main" java.lang.IllegalArgumentException: Cannot reflectively create enum objects

at java.lang.reflect.Constructor.newInstance(Constructor.java:417)

at zejian.SingletonEnum.main(SingletonEnum.java:38)

at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)

at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.lang.reflect.Method.invoke(Method.java:498)

at com.intellij.rt.execution.application.AppMain.main(AppMain.java:144)

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8

显然告诉我们不能使用反射创建枚举类，这是为什么呢？不妨看看newInstance方法源码：

public T newInstance(Object ... initargs)

throws InstantiationException, IllegalAccessException,

IllegalArgumentException, InvocationTargetException

{

if (!override) {

if (!Reflection.quickCheckMemberAccess(clazz, modifiers)) {

Class<?> caller = Reflection.getCallerClass();

checkAccess(caller, clazz, null, modifiers);

}

}

//这里判断Modifier.ENUM是不是枚举修饰符，如果是就抛异常

if ((clazz.getModifiers() & Modifier.ENUM) != 0)

throw new IllegalArgumentException("Cannot reflectively create enum objects");

ConstructorAccessor ca = constructorAccessor; // read volatile

if (ca == null) {

ca = acquireConstructorAccessor();

}

@SuppressWarnings("unchecked")

T inst = (T) ca.newInstance(initargs);

return inst;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21

源码很了然，确实无法使用反射创建枚举实例，也就是说明了创建枚举实例只有编译器能够做到而已。显然枚举单例模式确实是很不错的选择，因此我们推荐使用它。但是这总不是万能的，对于android平台这个可能未必是最好的选择，在android开发中，内存优化是个大块头，而使用枚举时占用的内存常常是静态变量的两倍还多，因此android官方在内存优化方面给出的建议是尽量避免在android中使用enum。但是不管如何，关于单例，我们总是应该记住：线程安全，延迟加载，序列化与反序列化安全，反射安全是很重重要的。

**EnumMap**

**EnumMap基本用法**

先思考这样一个问题，现在我们有一堆size大小相同而颜色不同的数据，需要统计出每种颜色的数量是多少以便将数据录入仓库，定义如下枚举用于表示颜色Color:

enum Color {

GREEN,RED,BLUE,YELLOW

}

* 1
* 2
* 3

我们有如下解决方案，使用Map集合来统计，key值作为颜色名称，value代表衣服数量，如下：

package com.zejian.enumdemo;

import java.util.\*;

/\*\*

\* Created by zejian on 2017/5/10.

\* Blog : http://blog.csdn.net/javazejian [原文地址,请尊重原创]

\*/

public class EnumMapDemo {

public static void main(String[] args){

List<Clothes> list = new ArrayList<>();

list.add(new Clothes("C001",Color.BLUE));

list.add(new Clothes("C002",Color.YELLOW));

list.add(new Clothes("C003",Color.RED));

list.add(new Clothes("C004",Color.GREEN));

list.add(new Clothes("C005",Color.BLUE));

list.add(new Clothes("C006",Color.BLUE));

list.add(new Clothes("C007",Color.RED));

list.add(new Clothes("C008",Color.YELLOW));

list.add(new Clothes("C009",Color.YELLOW));

list.add(new Clothes("C010",Color.GREEN));

//方案1:使用HashMap

Map<String,Integer> map = new HashMap<>();

for (Clothes clothes:list){

String colorName=clothes.getColor().name();

Integer count = map.get(colorName);

if(count!=null){

map.put(colorName,count+1);

}else {

map.put(colorName,1);

}

}

System.out.println(map.toString());

System.out.println("---------------");

//方案2:使用EnumMap

Map<Color,Integer> enumMap=new EnumMap<>(Color.class);

for (Clothes clothes:list){

Color color=clothes.getColor();

Integer count = enumMap.get(color);

if(count!=null){

enumMap.put(color,count+1);

}else {

enumMap.put(color,1);

}

}

System.out.println(enumMap.toString());

}

/\*\*

输出结果:

{RED=2, BLUE=3, YELLOW=3, GREEN=2}

---------------

{GREEN=2, RED=2, BLUE=3, YELLOW=3}

\*/

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60

代码比较简单，我们使用两种解决方案，一种是HashMap，一种EnumMap，虽然都统计出了正确的结果，但是EnumMap作为枚举的专属的集合，我们没有理由再去使用HashMap，毕竟EnumMap要求其Key必须为Enum类型，因而使用Color枚举实例作为key是最恰当不过了，也避免了获取name的步骤，更重要的是EnumMap效率更高，因为其内部是通过数组实现的（稍后分析），注意EnumMap的key值不能为null，虽说是枚举专属集合，但其操作与一般的Map差不多，概括性来说EnumMap是专门为枚举类型量身定做的Map实现，虽然使用其它的Map（如HashMap）也能完成相同的功能，但是使用EnumMap会更加高效，它只能接收同一枚举类型的实例作为键值且不能为null，由于枚举类型实例的数量相对固定并且有限，所以EnumMap使用数组来存放与枚举类型对应的值，毕竟数组是一段连续的内存空间，根据程序局部性原理，效率会相当高。下面我们来进一步了解EnumMap的用法，先看构造函数：

//创建一个具有指定键类型的空枚举映射。

EnumMap(Class<K> keyType)

//创建一个其键类型与指定枚举映射相同的枚举映射，最初包含相同的映射关系（如果有的话）。

EnumMap(EnumMap<K,? extends V> m)

//创建一个枚举映射，从指定映射对其初始化。

EnumMap(Map<K,? extends V> m)

* 1
* 2
* 3
* 4
* 5
* 6

与HashMap不同，它需要传递一个类型信息，即Class对象，通过这个参数EnumMap就可以根据类型信息初始化其内部数据结构，另外两只是初始化时传入一个Map集合，代码演示如下：

//使用第一种构造

Map<Color,Integer> enumMap=new EnumMap<>(Color.class);

//使用第二种构造

Map<Color,Integer> enumMap2=new EnumMap<>(enumMap);

//使用第三种构造

Map<Color,Integer> hashMap = new HashMap<>();

hashMap.put(Color.GREEN, 2);

hashMap.put(Color.BLUE, 3);

Map<Color, Integer> enumMap = new EnumMap<>(hashMap);

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

至于EnumMap的方法，跟普通的map几乎没有区别，注意与HashMap的主要不同在于构造方法需要传递类型参数和EnumMap保证Key顺序与枚举中的顺序一致，但请记住Key不能为null。

**EnumMap实现原理剖析**

EnumMap的源码有700多行，这里我们主要分析其内部存储结构，添加查找的实现，了解这几点，对应EnumMap内部实现原理也就比较清晰了，先看数据结构和构造函数

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

implements java.io.Serializable, Cloneable

{

//Class对象引用

private final Class<K> keyType;

//存储Key值的数组

private transient K[] keyUniverse;

//存储Value值的数组

private transient Object[] vals;

//map的size

private transient int size = 0;

//空map

private static final Enum<?>[] ZERO\_LENGTH\_ENUM\_ARRAY = new Enum<?>[0];

//构造函数

public EnumMap(Class<K> keyType) {

this.keyType = keyType;

keyUniverse = getKeyUniverse(keyType);

vals = new Object[keyUniverse.length];

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26

EnumMap继承了AbstractMap类，因此EnumMap具备一般map的使用方法，keyType表示类型信息，keyUniverse表示键数组，存储的是所有可能的枚举值，vals数组表示键对应的值，size表示键值对个数。在构造函数中通过keyUniverse = getKeyUniverse(keyType);初始化了keyUniverse数组的值，内部存储的是所有可能的枚举值，接着初始化了存在Value值得数组vals，其大小与枚举实例的个数相同，getKeyUniverse方法实现如下

//返回枚举数组

private static <K extends Enum<K>> K[] getKeyUniverse(Class<K> keyType) {

//最终调用到枚举类型的values方法，values方法返回所有可能的枚举值

return SharedSecrets.getJavaLangAccess()

.getEnumConstantsShared(keyType);

}

* 1
* 2
* 3
* 4
* 5
* 6

从方法的返回值来看，返回类型是枚举数组，事实也是如此，最终返回值正是枚举类型的values方法的返回值，前面我们分析过values方法返回所有可能的枚举值，因此keyUniverse数组存储就是枚举类型的所有可能的枚举值。接着看put方法的实现

public V put(K key, V value) {

typeCheck(key);//检测key的类型

//获取存放value值得数组下标

int index = key.ordinal();

//获取旧值

Object oldValue = vals[index];

//设置value值

vals[index] = maskNull(value);

if (oldValue == null)

size++;

return unmaskNull(oldValue);//返回旧值

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12

这里通过typeCheck方法进行了key类型检测，判断是否为枚举类型，如果类型不对，会抛出异常

private void typeCheck(K key) {

Class<?> keyClass = key.getClass();//获取类型信息

if (keyClass != keyType && keyClass.getSuperclass() != keyType)

throw new ClassCastException(keyClass + " != " + keyType);

}

* 1
* 2
* 3
* 4
* 5

接着通过int index = key.ordinal()的方式获取到该枚举实例的顺序值，利用此值作为下标，把值存储在vals数组对应下标的元素中即vals[index]，这也是为什么EnumMap能维持与枚举实例相同存储顺序的原因，我们发现在对vals[]中元素进行赋值和返回旧值时分别调用了maskNull方法和unmaskNull方法

//代表NULL值得空对象实例

private static final Object NULL = new Object() {

public int hashCode() {

return 0;

}

public String toString() {

return "java.util.EnumMap.NULL";

}

};

private Object maskNull(Object value) {

//如果值为空，返回NULL对象，否则返回value

return (value == null ? NULL : value);

}

@SuppressWarnings("unchecked")

private V unmaskNull(Object value) {

//将NULL对象转换为null值

return (V)(value == NULL ? null : value);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21

由此看来EnumMap还是允许存放null值的，但key绝对不能为null，对于null值，EnumMap进行了特殊处理,将其包装为NULL对象，毕竟vals[]存的是Object，maskNull方法和unmaskNull方法正是用于null的包装和解包装的。这就是EnumMap集合的添加过程。下面接着看获取方法

public V get(Object key) {

return (isValidKey(key) ?

unmaskNull(vals[((Enum<?>)key).ordinal()]) : null);

}

//对Key值的有效性和类型信息进行判断

private boolean isValidKey(Object key) {

if (key == null)

return false;

// Cheaper than instanceof Enum followed by getDeclaringClass

Class<?> keyClass = key.getClass();

return keyClass == keyType || keyClass.getSuperclass() == keyType;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

相对应put方法，get方法显示相当简洁，key有效的话，直接通过ordinal方法取索引，然后在值数组vals里通过索引获取值返回。remove方法如下：

public V remove(Object key) {

//判断key值是否有效

if (!isValidKey(key))

return null;

//直接获取索引

int index = ((Enum<?>)key).ordinal();

Object oldValue = vals[index];

//对应下标元素值设置为null

vals[index] = null;

if (oldValue != null)

size--;//减size

return unmaskNull(oldValue);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

非常简单，key值有效，通过key获取下标索引值，把vals[]对应下标值设置为null，size减一。查看是否包含某个值，

判断是否包含某value

public boolean containsValue(Object value) {

value = maskNull(value);

//遍历数组实现

for (Object val : vals)

if (value.equals(val))

return true;

return false;

}

//判断是否包含key

public boolean containsKey(Object key) {

return isValidKey(key) && vals[((Enum<?>)key).ordinal()] != null;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

判断value直接通过遍历数组实现，而判断key就更简单了，判断key是否有效和对应vals[]中是否存在该值。ok~，这就是EnumMap的主要实现原理，即内部有两个数组，长度相同，一个表示所有可能的键(枚举值)，一个表示对应的值，不允许keynull，但允许value为null，键都有一个对应的索引，根据索引直接访问和操作其键数组和值数组，由于操作都是数组，因此效率很高。

**EnumSet**

EnumSet是与枚举类型一起使用的专用 Set 集合，EnumSet 中所有元素都必须是枚举类型。与其他Set接口的实现类HashSet/TreeSet(内部都是用对应的HashMap/TreeMap实现的)不同的是，EnumSet在内部实现是位向量(稍后分析)，它是一种极为高效的位运算操作，由于直接存储和操作都是bit，因此EnumSet空间和时间性能都十分可观，足以媲美传统上基于 int 的“位标志”的运算，重要的是我们可像操作set集合一般来操作位运算，这样使用代码更简单易懂同时又具备类型安全的优势。注意EnumSet不允许使用 null 元素。试图插入 null 元素将抛出 NullPointerException，但试图测试判断是否存在null 元素或移除 null 元素则不会抛出异常，与大多数collection 实现一样，EnumSet不是线程安全的，因此在多线程环境下应该注意数据同步问题，ok~，下面先来简单看看EnumSet的使用方式。

**EnumSet用法**

创建EnumSet并不能使用new关键字，因为它是个抽象类，而应该使用其提供的静态工厂方法，EnumSet的静态工厂方法比较多，如下：

创建一个具有指定元素类型的空EnumSet。

EnumSet<E> noneOf(Class<E> elementType)

//创建一个指定元素类型并包含所有枚举值的EnumSet

<E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType)

// 创建一个包括枚举值中指定范围元素的EnumSet

<E extends Enum<E>> EnumSet<E> range(E from, E to)

// 初始集合包括指定集合的补集

<E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> s)

// 创建一个包括参数中所有元素的EnumSet

<E extends Enum<E>> EnumSet<E> of(E e)

<E extends Enum<E>> EnumSet<E> of(E e1, E e2)

<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3)

<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4)

<E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4, E e5)

<E extends Enum<E>> EnumSet<E> of(E first, E... rest)

//创建一个包含参数容器中的所有元素的EnumSet

<E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> s)

<E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18

代码演示如下：

package zejian;

import java.util.ArrayList;

import java.util.EnumSet;

import java.util.List;

/\*\*

\* Created by wuzejian on 2017/5/12.

\*

\*/

enum Color {

GREEN , RED , BLUE , BLACK , YELLOW

}

public class EnumSetDemo {

public static void main(String[] args){

//空集合

EnumSet<Color> enumSet= EnumSet.noneOf(Color.class);

System.out.println("添加前："+enumSet.toString());

enumSet.add(Color.GREEN);

enumSet.add(Color.RED);

enumSet.add(Color.BLACK);

enumSet.add(Color.BLUE);

enumSet.add(Color.YELLOW);

System.out.println("添加后："+enumSet.toString());

System.out.println("-----------------------------------");

//使用allOf创建包含所有枚举类型的enumSet，其内部根据Class对象初始化了所有枚举实例

EnumSet<Color> enumSet1= EnumSet.allOf(Color.class);

System.out.println("allOf直接填充："+enumSet1.toString());

System.out.println("-----------------------------------");

//初始集合包括枚举值中指定范围的元素

EnumSet<Color> enumSet2= EnumSet.range(Color.BLACK,Color.YELLOW);

System.out.println("指定初始化范围："+enumSet2.toString());

System.out.println("-----------------------------------");

//指定补集，也就是从全部枚举类型中去除参数集合中的元素，如下去掉上述enumSet2的元素

EnumSet<Color> enumSet3= EnumSet.complementOf(enumSet2);

System.out.println("指定补集："+enumSet3.toString());

System.out.println("-----------------------------------");

//初始化时直接指定元素

EnumSet<Color> enumSet4= EnumSet.of(Color.BLACK);

System.out.println("指定Color.BLACK元素："+enumSet4.toString());

EnumSet<Color> enumSet5= EnumSet.of(Color.BLACK,Color.GREEN);

System.out.println("指定Color.BLACK和Color.GREEN元素："+enumSet5.toString());

System.out.println("-----------------------------------");

//复制enumSet5容器的数据作为初始化数据

EnumSet<Color> enumSet6= EnumSet.copyOf(enumSet5);

System.out.println("enumSet6："+enumSet6.toString());

System.out.println("-----------------------------------");

List<Color> list = new ArrayList<Color>();

list.add(Color.BLACK);

list.add(Color.BLACK);//重复元素

list.add(Color.RED);

list.add(Color.BLUE);

System.out.println("list:"+list.toString());

//使用copyOf(Collection<E> c)

EnumSet enumSet7=EnumSet.copyOf(list);

System.out.println("enumSet7:"+enumSet7.toString());

/\*\*

输出结果：

添加前：[]

添加后：[GREEN, RED, BLUE, BLACK, YELLOW]

-----------------------------------

allOf直接填充：[GREEN, RED, BLUE, BLACK, YELLOW]

-----------------------------------

指定初始化范围：[BLACK, YELLOW]

-----------------------------------

指定补集：[GREEN, RED, BLUE]

-----------------------------------

指定Color.BLACK元素：[BLACK]

指定Color.BLACK和Color.GREEN元素：[GREEN, BLACK]

-----------------------------------

enumSet6：[GREEN, BLACK]

-----------------------------------

list:[BLACK, BLACK, RED, BLUE]

enumSet7:[RED, BLUE, BLACK]

\*/

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88
* 89
* 90
* 91
* 92
* 93
* 94
* 95
* 96
* 97

noneOf(Class<E> elementType)静态方法，主要用于创建一个空的EnumSet集合，传递参数elementType代表的是枚举类型的类型信息，即Class对象。EnumSet<E> allOf(Class<E> elementType)静态方法则是创建一个填充了elementType类型所代表的所有枚举实例，奇怪的是EnumSet提供了多个重载形式的of方法，最后一个接受的的是可变参数，其他重载方法则是固定参数个数，EnumSet之所以这样设计是因为可变参数的运行效率低一些，所有在参数数据不多的情况下，强烈***不建议***使用传递参数为可变参数的of方法，即EnumSet<E> of(E first, E... rest)，其他方法就不分析了，看代码演示即可。至于EnumSet的操作方法，则与set集合是一样的，可以看API即可这也不过多说明。什么时候使用EnumSet比较恰当的，事实上当需要进行位域运算，就可以使用EnumSet提到位域，如下：

public class EnumSetDemo {

//定义位域变量

public static final int TYPE\_ONE = 1 << 0 ; //1

public static final int TYPE\_TWO = 1 << 1 ; //2

public static final int TYPE\_THREE = 1 << 2 ; //4

public static final int TYPE\_FOUR = 1 << 3 ; //8

public static void main(String[] args){

//位域运算

int type= TYPE\_ONE | TYPE\_TWO | TYPE\_THREE |TYPE\_FOUR;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11

诸如上述情况，我们都可以将上述的类型定义成枚举然后采用EnumSet来装载，进行各种操作，这样不仅不用手动编写太多冗余代码，而且使用EnumSet集合进行操作也将使代码更加简洁明了。

enum Type{

TYPE\_ONE,TYPE\_TWO,TYPE\_THREE,TYPE\_FOUR

}

public class EnumSetDemo {

public static void main(String[] args){

EnumSet set =EnumSet.of(Type.TYPE\_ONE,Type.TYPE\_FOUR);

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

其实博主认为EnumSet最有价值的是其内部实现原理，采用的是位向量，它体现出来的是一种高效的数据处理方式，这点很值得我们去学习它。

**EnumSet实现原理剖析**

关于EnumSet实现原理可能会有点烧脑，内部执行几乎都是位运算，博主将尽力使用图片来分析，协助大家理解。

**理解位向量**

在分析EnumSet前有必要先了解以下位向量，顾名思义位向量就是用一个bit位(0或1)标记一个元素的状态，用一组bit位表示一个集合的状态，而每个位对应一个元素，每个bit位的状态只可能有两种，即0或1。位向量能表示的元素个数与向量的bit位长度有关，如一个int类型能表示32个元素，而一个long类型则可以表示64个元素，对于EnumSet而言采用的就long类型或者long类型数组。比如现在有一个文件中的数据，该文件存储了N=1000000个无序的整数，需要把这些整数读取到内存并排序再重新写回文件中，该如何解决？最简单的方式是用int类型来存储每个数，并把其存入到数组(int a[m])中，再进行排序，但是这种方式将会导致存储空间异常大，对数据操作起来效率也能成问题，那有没更高效的方式呢？的确是有的，那就是运用位向量，我们知道一个int型的数有4个字节，也就是32位，那么我们可以用N/32个int型数组来表示这N个数：

a[0]表示第1~32个数（0~31）

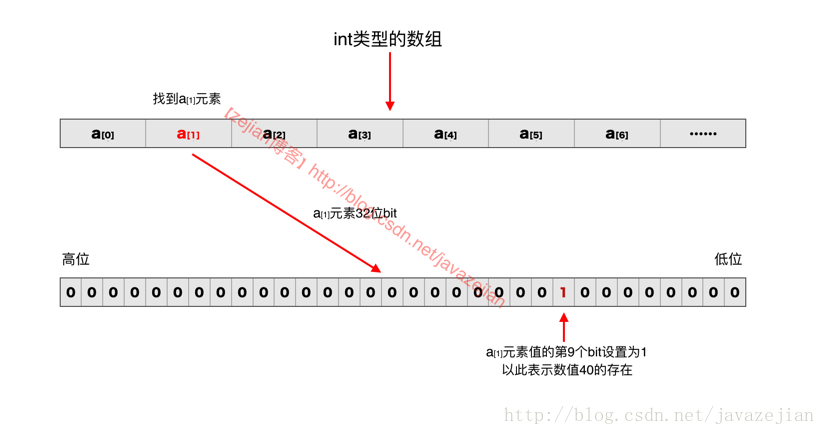
a[1]表示第33~64个数（32~63）

a[2]表示第65~96个数（64~95）

...... 以此类推

* 1
* 2
* 3
* 4

这样，每当输入一个数字m，我们应该先找到该数字在数组的第？个元素，也就是a[?]，然后再确定在这个元素的第几个bit位，找到后设置为1，代表存在该数字。举个例子来说，比如输入40，那么40/32为1余8，则应该将a[1]元素值的第9个bit位置为1(1的二进制左移8位后就是第9个位置)，表示该数字存在，40数字的表示原理图过程如下：



大概明白了位向量表示方式后，上述过程的计算方式，通过以下方式可以计算该数存储在数组的第?个元素和元素中第?个bit位置,为了演示方便，我们这里假设整第?个元素中的?为P，余值设置S

//m 除以 2^n 则商(P)表示为 m >> n

//等同于 m / 2^5 取整数 即：40 / 32 = 1 ，那么P=1就是数组中第2个元素，即a[1]

//位操作过程如下，40的二进制

00000000 00000000 00000000 00101000

//右移5位即 n=5 ， m >> 5 ，即结果转为10进制就是P=1

00000000 00000000 00000000 00000001

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

在这里我们使用的int类型，即32位，所有2^5=32，因此n=5，由此计算出 P的值代表的是数组的第 P 个元素，接着利用下述方式计算出余数（S），以此设置该元素值的第（S+1）个bit位为1

//m 除以2^n 的余数(S)表示为 m & (2^n-1)

//等同于： m % 2^5 取余数 即：40 % 32 = 8

//m=40的二进制

00000000 00000000 00000000 00101000

//2^n-1（31）的二进制

00000000 00000000 00000000 00011111

// m & (2^n-1) 即40与31进行与操作得出余数 即 S=8

00000000 00000000 00000000 00001000

//下面是将a[1]元素值的第(8+1)个bit设置为1，为什么是(8+1)不是8？因为1左移8位就在第9个bit位了，过程如下：

//1的二进制如下：

00000000 00000000 00000000 00000001

//1 << 8 利用余数8对1进行左移动

00000000 00000000 00000001 0000000

//然后再与a[1]执行或操作后就可以将对应的bit位设置为1

//a[P] |= 1 << S 见下述java实现的代码

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22

通过上述二进制位运算过程(关于位运算可以看博主的另一篇博文~[java位运算](http://blog.csdn.net/javazejian/article/details/51181320))就可以计算出整数部分P和余数部分S，并成功设置bit位为1，现在利用java来实现这个运算过程如下：

//定义变量

private int[] a; //数组存储元素的数组

private int BIT\_LENGTH = 32;//默认使用int类型

private int P; //整数部分

private int S; //余数

private int MASK = 0x1F;// 2^5 - 1

private int SHIFT = 5; // 2^n SHIFT=n=5 表示2^5=32 即bit位长度32

* 1
* 2
* 3
* 4
* 5
* 6
* 7

计算代码

/\*\*

\* 置位操作,添加操作

\* @param i

\*/

public void set(int i){

P = i >> SHIFT; //结果等同 P = i / BIT\_LENGTH; 取整数 ①

S = i & MASK; //结果等同 S = i % BIT\_LENGTH; 取余数 ②

a[P] |= 1 << S; //赋值设置该元素bit位为1 ③

//将int型变量j的第k个比特位设置为1， 即j=j|(1<<k),上述3句合并为一句

//a[i >> SHIFT ] |= (1 << (i & MASK)); ④

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12

计算出P和S后，就可以进行赋值了，其中 a[P]代表数组中第P个元素，a[P] |= 1 << S 整句意思是把a[P]元素的第S+1位设置为1，注意从低位到高位设置，即从右到左，①②③合并为④，代码将更佳简洁。当然有添加操作，那么就会有删除操作，删除操作过程与添加类似，只不过删除是把相对应的bit位设置0，代表不存在该数值。

/\*\*

\* 置0操作，相当于清除元素

\* @param i

\*/

public void clear(int i){

P = i >> SHIFT; //计算位于数组中第？个元素 P = i / BIT\_LENGTH;

S = i & MASK; //计算余数 S = i % BIT\_LENGTH;

//把a[P]元素的第S+1个(从低位到高位)bit位设置为0

a[P] &= ~(1 << S);

//更优写法

//将int型变量j的第k个比特位设置为0，即j= j&~(1<<k)

//a[i>>SHIFT] &= ~(1<<(i &MASK));

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

与添加唯一不同的是，计算出余数S，利用1左移S位，再取反(~)操作，最后进行与(&)操作，即将a[P]元素的第S+1个(从低位到高位)bit位设置为0，表示删除该数字，这个计算过程大家可以自行推算一下。这就是位向量表示法的添加和清除方法，然后我们可以利用下述的get方法判断某个bit是否存在某个数字：

/\*\*

\* 读取操作，返回1代表该bit位有值，返回0代表该bit位没值

\* @param i

\* @return

\*/

public int get(int i){

//a[i>>SHIFT] & (1<<(i&MASK));

P = i >> SHIFT;

S = i & MASK;

return Integer.bitCount(a[P] & (1 << S));

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11

其中Integer.bitCount()是返回指定 int 值的二进制补码(计算机数字的二进制表示法都是使用补码表示的)表示形式的 1 位的数量。位向量运算整体代码实现如下：

package com.zejian;

import java.util.ArrayList;

import java.util.List;

import java.util.Random;

/\*\*

\* Created by zejian on 2017/5/13.

\* Blog : http://blog.csdn.net/javazejian [原文地址,请尊重原创]

\* 位向量存储数据

\*/

public class BitVetory {

private int count;

private int[] a; //数组

private int BIT\_LENGTH = 32;//默认使用int类型

private int P; //整数部分

private int S; //余数

private int MASK = 0x1F;// 2^5 - 1

private int SHIFT = 5; // 2^n SHIFT=n=5 表示2^5=32 即bit位长度32

/\*\*

\* 初始化位向量

\* @param count

\*/

public BitVetory(int count) {

this.count = count;

a = new int[(count-1)/BIT\_LENGTH + 1];

init();

}

/\*\*

\* 将数组中元素bit位设置为0

\*/

public void init(){

for (int i = 0; i < count; i++) {

clear(i);

}

}

/\*\*

\* 获取排序后的数组

\* @return

\*/

public List<Integer> getSortedArray(){

List<Integer> sortedArray = new ArrayList<Integer>();

for (int i = 0; i < count; i++) {

if (get(i) == 1) {//判断i是否存在

sortedArray.add(i);

}

}

return sortedArray;

}

/\*\*

\* 置位操作,设置元素

\* @param i

\*/

public void set(int i){

P = i >> SHIFT; //P = i / BIT\_LENGTH; 取整数

S = i & MASK; //S = i % BIT\_LENGTH; 取余数

a[P] |= 1 << S;

//将int型变量j的第k个比特位设置为1， 即j=j|(1<<k),上述3句合并为一句

//a[i >> SHIFT ] |= (1 << (i & MASK));

}

/\*\*

\* 置0操作，相当于清除元素

\* @param i

\*/

public void clear(int i){

P = i >> SHIFT; //计算位于数组中第？个元素 P = i / BIT\_LENGTH;

S = i & MASK; //计算余数 S = i % BIT\_LENGTH;

a[P] &= ~(1 << S);

//更优写法

//将int型变量j的第k个比特位设置为0，即j= j&~(1<<k)

//a[i>>SHIFT] &= ~(1<<(i &MASK));

}

/\*\*

\* 读取操作，返回1代表该bit位有值，返回0代表该bit位没值

\* @param i

\* @return

\*/

public int get(int i){

//a[i>>SHIFT] & (1<<(i&MASK));

P = i >> SHIFT;

S = i & MASK;

return Integer.bitCount(a[P] & (1 << S));

}

//测试

public static void main(String[] args) {

int count = 25;

List<Integer> randoms = getRandomsList(count);

System.out.println("排序前：");

BitVetory bitVetory = new BitVetory(count);

for (Integer e : randoms) {

System.out.print(e+",");

bitVetory.set(e);

}

List<Integer> sortedArray = bitVetory.getSortedArray();

System.out.println();

System.out.println("排序后：");

for (Integer e : sortedArray) {

System.out.print(e+",");

}

/\*\*

输出结果:

排序前：

6,3,20,10,18,15,19,16,13,4,21,22,24,2,14,5,12,7,23,8,1,17,9,11,

排序后：

1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,

\*/

}

private static List<Integer> getRandomsList(int count) {

Random random = new Random();

List<Integer> randomsList = new ArrayList<Integer>();

while(randomsList.size() < (count - 1)){

int element = random.nextInt(count - 1) + 1;//element ∈ [1,count)

if (!randomsList.contains(element)) {

randomsList.add(element);

}

}

return randomsList;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38
* 39
* 40
* 41
* 42
* 43
* 44
* 45
* 46
* 47
* 48
* 49
* 50
* 51
* 52
* 53
* 54
* 55
* 56
* 57
* 58
* 59
* 60
* 61
* 62
* 63
* 64
* 65
* 66
* 67
* 68
* 69
* 70
* 71
* 72
* 73
* 74
* 75
* 76
* 77
* 78
* 79
* 80
* 81
* 82
* 83
* 84
* 85
* 86
* 87
* 88
* 89
* 90
* 91
* 92
* 93
* 94
* 95
* 96
* 97
* 98
* 99
* 100
* 101
* 102
* 103
* 104
* 105
* 106
* 107
* 108
* 109
* 110
* 111
* 112
* 113
* 114
* 115
* 116
* 117
* 118
* 119
* 120
* 121
* 122
* 123
* 124
* 125
* 126
* 127
* 128
* 129
* 130
* 131
* 132
* 133
* 134

**EnumSet原理**

有前面位向量的分析，对于了解EnumSet的实现原理就相对简单些了，EnumSet内部使用的位向量实现的，前面我们说过EnumSet是一个抽象类，事实上它存在两个子类，RegularEnumSet和JumboEnumSet。RegularEnumSet使用一个long类型的变量作为位向量，long类型的位长度是64，因此可以存储64个枚举实例的标志位，一般情况下是够用的了，而JumboEnumSet使用一个long类型的数组，当枚举个数超过64时，就会采用long数组的方式存储。先看看EnumSet内部的数据结构：

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

implements Cloneable, java.io.Serializable

{

//表示枚举类型

final Class<E> elementType;

//存储该类型信息所表示的所有可能的枚举实例

final Enum<?>[] universe;

//..........

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9

EnumSet中有两个变量，一个elementType用于表示枚举的类型信息，universe是数组类型，存储该类型信息所表示的所有可能的枚举实例，EnumSet是抽象类，因此具体的实现是由子类完成的，下面看看noneOf(Class<E> elementType)静态构建方法

public static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType) {

//根据EnumMap中的一样，获取所有可能的枚举实例

Enum<?>[] universe = getUniverse(elementType);

if (universe == null)

throw new ClassCastException(elementType + " not an enum");

if (universe.length <= 64)

//枚举个数小于64，创建RegularEnumSet

return new RegularEnumSet<>(elementType, universe);

else

//否则创建JumboEnumSet

return new JumboEnumSet<>(elementType, universe);

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13

从源码可以看出如果枚举值个数小于等于64，则静态工厂方法中创建的就是RegularEnumSet，否则大于64的话就创建JumboEnumSet。无论是RegularEnumSet还是JumboEnumSet，其构造函数内部都间接调用了EnumSet的构造函数，因此最终的elementType和universe都传递给了父类EnumSet的内部变量。如下：

//RegularEnumSet构造

RegularEnumSet(Class<E>elementType, Enum<?>[] universe) {

super(elementType, universe);

}

//JumboEnumSet构造

JumboEnumSet(Class<E>elementType, Enum<?>[] universe) {

super(elementType, universe);

elements = new long[(universe.length + 63) >>> 6];

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

在RegularEnumSet类和JumboEnumSet类中都存在一个elements变量，用于记录位向量的操作，

//RegularEnumSet

class RegularEnumSet<E extends Enum<E>> extends EnumSet<E> {

private static final long serialVersionUID = 3411599620347842686L;

//通过long类型的elements记录位向量的操作

private long elements = 0L;

//.......

}

//对于JumboEnumSet则是：

class JumboEnumSet<E extends Enum<E>> extends EnumSet<E> {

private static final long serialVersionUID = 334349849919042784L;

//通过long数组类型的elements记录位向量

private long elements[];

//表示集合大小

private int size = 0;

//.............

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18

在RegularEnumSet中elements是一个long类型的变量，共有64个bit位，因此可以记录64个枚举常量，当枚举常量的数量超过64个时，将使用JumboEnumSet，elements在该类中是一个long型的数组，每个数组元素都可以存储64个枚举常量，这个过程其实与前面位向量的分析是同样的道理，只不过前面使用的是32位的int类型，这里使用的是64位的long类型罢了。接着我们看看EnumSet是如何添加数据的，RegularEnumSet中的add实现如下

public boolean add(E e) {

//检测是否为枚举类型

typeCheck(e);

//记录旧elements

long oldElements = elements;

//执行位向量操作，是不是很熟悉？

//数组版：a[i >> SHIFT ] |= (1 << (i & MASK))

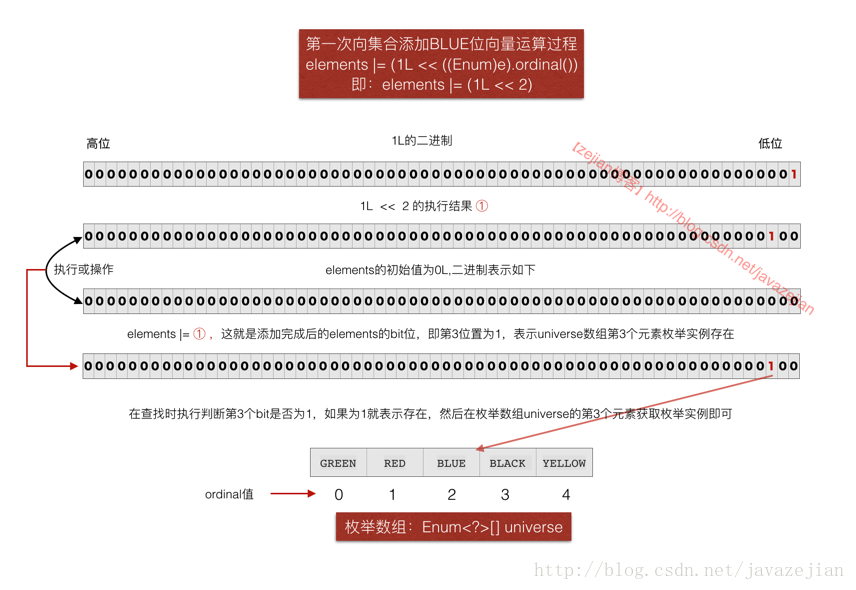
elements |= (1L << ((Enum)e).ordinal());

return elements != oldElements;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

关于elements |= (1L << ((Enum)e).ordinal());这句跟我们前面分析位向量操作是相同的原理，只不过前面分析的是数组类型实现，这里用的long类型单一变量实现，((Enum)e).ordinal()通过该语句获取要添加的枚举实例的序号，然后通过1左移再与 long类型的elements进行或操作，就可以把对应位置上的bit设置为1了，也就代表该枚举实例存在。图示演示过程如下，注意universe数组在EnumSet创建时就初始化并填充了所有可能的枚举实例，而elements值的第n个bit位1时代表枚举存在，而获取的则是从universe数组中的第n个元素值。



这就是枚举实例的添加过程和获取原理。而对于JumboEnumSet的add实现则是如下：

public boolean add(E e) {

typeCheck(e);

//计算ordinal值

int eOrdinal = e.ordinal();

int eWordNum = eOrdinal >>> 6;

long oldElements = elements[eWordNum];

//与前面分析的位向量相同：a[i >> SHIFT ] |= (1 << (i & MASK))

elements[eWordNum] |= (1L << eOrdinal);

boolean result = (elements[eWordNum] != oldElements);

if (result)

size++;

return result;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14

关于JumboEnumSet的add实现与RegularEnumSet区别是一个是long数组类型，一个long变量，运算原理相同，数组的位向量运算与前面分析的是相同的，这里不再分析。接着看看如何删除元素

//RegularEnumSet类实现

public boolean remove(Object e) {

if (e == null)

return false;

Class eClass = e.getClass();

if (eClass != elementType && eClass.getSuperclass() != elementType)

return false;

long oldElements = elements;

//将int型变量j的第k个比特位设置为0，即j= j&~(1<<k)

//数组类型：a[i>>SHIFT] &= ~(1<<(i &MASK));

elements &= ~(1L << ((Enum)e).ordinal());//long遍历类型操作

return elements != oldElements;

}

//JumboEnumSet类的remove实现

public boolean remove(Object e) {

if (e == null)

return false;

Class<?> eClass = e.getClass();

if (eClass != elementType && eClass.getSuperclass() != elementType)

return false;

int eOrdinal = ((Enum<?>)e).ordinal();

int eWordNum = eOrdinal >>> 6;

long oldElements = elements[eWordNum];

//与a[i>>SHIFT] &= ~(1<<(i &MASK));相同

elements[eWordNum] &= ~(1L << eOrdinal);

boolean result = (elements[eWordNum] != oldElements);

if (result)

size--;

return result;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35

删除remove的实现，跟位向量的清空操作是同样的实现原理，如下：   


至于JumboEnumSet的实现原理也是类似的，这里不再重复。下面为了简洁起见，我们以RegularEnumSet类的实现作为源码分析，毕竟JumboEnumSet的内部实现原理可以说跟前面分析过的位向量几乎一样。o~，看看如何判断是否包含某个元素

public boolean contains(Object e) {

if (e == null)

return false;

Class eClass = e.getClass();

if (eClass != elementType && eClass.getSuperclass() != elementType)

return false;

//先左移再按&操作

return (elements & (1L << ((Enum)e).ordinal())) != 0;

}

public boolean containsAll(Collection<?> c) {

if (!(c instanceof RegularEnumSet))

return super.containsAll(c);

RegularEnumSet<?> es = (RegularEnumSet<?>)c;

if (es.elementType != elementType)

return es.isEmpty();

//~elements取反相当于elements补集，再与es.elements进行&操作，如果为0，

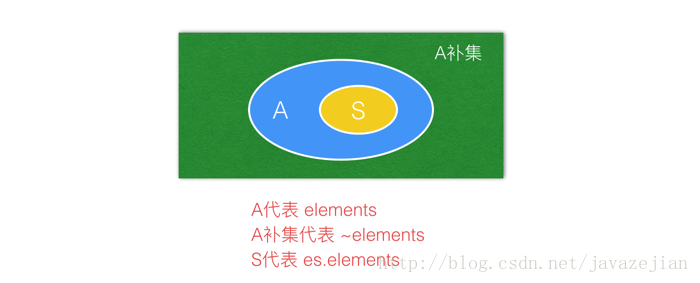
//就说明elements补集与es.elements没有交集，也就是es.elements是elements的子集

return (es.elements & ~elements) == 0;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22

对于contains(Object e) 方法，先左移再按位与操作，不为0，则表示包含该元素，跟位向量的get操作实现原理类似，这个比较简单。对于containsAll(Collection<?> c)则可能比较难懂，这里分析一下，elements变量(long类型)标记EnumSet集合中已存在元素的bit位，如果bit位为1则说明存在枚举实例，为0则不存在，现在执行~elements 操作后 则说明~elements是elements的补集，那么只要传递进来的es.elements与补集~elements 执行&操作为0，那么就可以证明es.elements与补集~elements 没有交集的可能，也就是说es.elements只能是elements的子集，这样也就可以判断出当前EnumSet集合中包含传递进来的集合c了，借着下图协助理解：



图中，elements代表A，es.elements代表S，~elements就是求A的补集，(es.elements & ~elements) == 0就是在验证A’∩B是不是空集，即S是否为A的子集。接着看retainAll方法，求两个集合交集

public boolean retainAll(Collection<?> c) {

if (!(c instanceof RegularEnumSet))

return super.retainAll(c);

RegularEnumSet<?> es = (RegularEnumSet<?>)c;

if (es.elementType != elementType) {

boolean changed = (elements != 0);

elements = 0;

return changed;

}

long oldElements = elements;

//执行与操作，求交集，比较简单

elements &= es.elements;

return elements != oldElements;

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

最后来看看迭代器是如何取值的

public Iterator<E> iterator() {

return new EnumSetIterator<>();

}

private class EnumSetIterator<E extends Enum<E>> implements Iterator<E> {

//记录elements

long unseen;

//记录最后一个返回值

long lastReturned = 0;

EnumSetIterator() {

unseen = elements;

}

public boolean hasNext() {

return unseen != 0;

}

@SuppressWarnings("unchecked")

public E next() {

if (unseen == 0)

throw new NoSuchElementException();

//取值过程，先与本身负执行&操作得出的就是二进制低位开始的第一个1的数值大小

lastReturned = unseen & -unseen;

//取值后减去已取得lastReturned

unseen -= lastReturned;

//返回在指定 long 值的二进制补码表示形式中最低位（最右边）的 1 位之后的零位的数量

return (E) universe[Long.numberOfTrailingZeros(lastReturned)];

}

public void remove() {

if (lastReturned == 0)

throw new IllegalStateException();

elements &= ~lastReturned;

lastReturned = 0;

}

}

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16
* 17
* 18
* 19
* 20
* 21
* 22
* 23
* 24
* 25
* 26
* 27
* 28
* 29
* 30
* 31
* 32
* 33
* 34
* 35
* 36
* 37
* 38

比较晦涩的应该是

//取值过程，先与本身负执行&操作得出的就是二进制低位开始的第一个1的数值大小

lastReturned = unseen & -unseen;

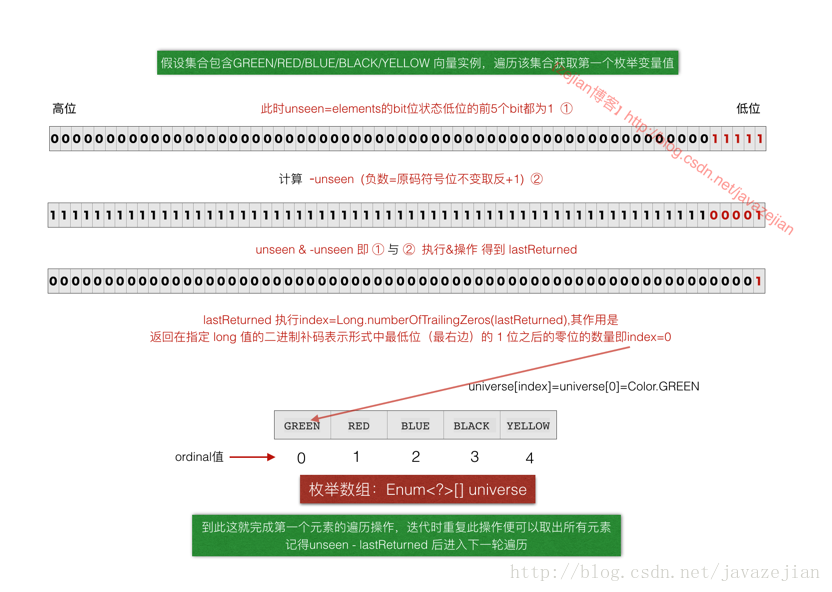
//取值后减去已取得lastReturned

unseen -= lastReturned;

return (E) universe[Long.numberOfTrailingZeros(lastReturned)];

* 1
* 2
* 3
* 4
* 5

我们通过原理图来协助理解，现在假设集合中已保存所有可能的枚举实例变量，我们需要把它们遍历展示出来，下面的第一个枚举元素的获取过程，显然通过unseen & -unseen;操作，我们可以获取到二进制低位开始的第一个1的数值，该计算的结果是要么全部都是0，要么就只有一个1，然后赋值给lastReturned，通过Long.numberOfTrailingZeros(lastReturned)获取到该bit为1在64位的long类型中的位置，即从低位算起的第几个bit，如图，该bit的位置恰好是低位的第1个bit位置，也就指明了universe数组的第一个元素就是要获取的枚举变量。执行unseen -= lastReturned;后继续进行第2个元素的遍历，依次类推遍历出所有值，这就是EnumSet的取值过程，真正存储枚举变量的是universe数组，而通过long类型变量的bit位的0或1表示存储该枚举变量在universe数组的那个位置，这样做的好处是任何操作都是执行long类型变量的bit位操作，这样执行效率将特别高，毕竟是二进制直接执行，只有最终获取值时才会操作到数组universe。



ok~，到这关于EnumSet的实现原理主要部分我们就分析完了，其内部使用位向量，存储结构很简洁，节省空间，大部分操作都是按位运算，直接操作二进制数据，因此效率极高。当然通过前面的分析，我们也掌握位向量的运算原理。好~，关于java枚举，我们暂时聊到这。

参考资料 《Thinking in Java》 And 《Effective Java》

版权声明：本文为博主原创文章，请尊重原创，未经博主允许禁止转载，保留追究权 https://blog.csdn.net/javazejian/article/details/71333103