枚举 & 注解 & 反射

1、枚举

1.1、简介

JDK1.5引入了新的类型--枚举。
在JDK1.5 之前，我们定义常量都是： public static fianl.... 。很难管理。
枚举，可以把相关的常量分组到一个枚举类型里，而且枚举提供了比常量更多的方法。

用于定义有限数量的一组同类常量，例如：

    错误级别：
        低、中、高、急
    一年的四季：
        春、夏、秋、冬
    商品的类型：
        美妆、手机、电脑、男装、女装...

在枚举类型中定义的常量是该枚举类型的实例。

1.2、定义格式

```
权限修饰符 enum 枚举名称  {
    实例1,实例2，实例3，实例4;
}
```

```
public enum Level {
  LOW(30), MEDIUM(15), HIGH(7), URGENT(1);

  private int levelValue;

  private Level(int levelValue) {
    this.levelValue = levelValue;
  }

  public int getLevelValue() {
    return levelValue;
  }
}
```

1.3、枚举类的主要方法

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。

1.4、实现接口的枚举类

所有的枚举都继承自java.lang.Enum类。由于Java 不支持多继承，所以枚举对象不能再继承其他类。

每个枚举对象，都可以实现自己的抽象方法

```
public interface LShow{
    void show();
}
public enum Level  implements LShow{
  LOW(30){
    @Override
    public void show(){
        //...
    }
  }, MEDIUM(15){
    @Override
    public void show(){
        //...
    }
  },HIGH(7){
    @Override
    public void show(){
        //...
    }
  },URGENT(1){
    @Override
    public void show(){
        //...
    }
  };

  private int levelValue;
```

```
  private Level(int levelValue) {
    this.levelValue = levelValue;
  }

  public int getLevelValue() {
    return levelValue;
  }
}
```

1.5、注意事项

- 一旦定义了枚举，最好不要妄图修改里面的值，除非修改是必要的。
- 枚举类默认继承的是java.lang.Enum类而不是Object类
- 枚举类不能有子类，因为其枚举类默认被final修饰
- 只能有private构造方法
- switch中使用枚举时，直接使用常量名，不用携带类名
- 不能定义name属性，因为自带name属性
- 不要为枚举类中的属性提供set方法，不符合枚举最初设计初衷。

2、注解

2.1、简介

Java 注解（Annotation）又称 Java 标注，是 JDK5.0 引入的一种注释机制。

Java 语言中的类、方法、变量、参数和包等都可以被标注。和注释不同，Java 标注可以通过反射获取标注内容。在编译器生成类文件时，标注可以被嵌入到字节码中。Java 虚拟机可以保留标注内容，在运行时可以获取到标注内容 。 当然它也支持自定义 Java 标注。

- 主要用于：
  - 编译格式检查
  - 反射中解析
  - 生成帮助文档
  - 跟踪代码依赖
  - 等

2.2、学习的重点

理解 Annotation 的关键，是理解 Annotation 的语法和用法.

学习步骤：

```
1.  概念
```

2. 怎么使用内置注解
3. 怎么自定义注解
4. 反射中怎么获取注解内容

2.3、内置注解

- @Override ： 重写 *
  - 定义在java.lang.Override
- @Deprecated：废弃 *

    - 定义在java.lang.Deprecated
- @SafeVarargs
    - Java 7 开始支持，忽略任何使用参数为泛型变量的方法或构造函数调用产生的警告。
- @FunctionalInterface： 函数式接口 *
    - Java 8 开始支持，标识一个匿名函数或函数式接口。
- @Repeatable：标识某注解可以在同一个声明上使用多次
    - Java 8 开始支持，标识某注解可以在同一个声明上使用多次。
- SuppressWarnings：抑制编译时的警告信息。 *
    - 定义在java.lang.SuppressWarnings
    - 三种使用方式

```
1. @SuppressWarnings("unchecked")  [^ 抑制单类型的警告]
2. @SuppressWarnings("unchecked","rawtypes")  [^ 抑制多类型的警告]
3. @SuppressWarnings("all")  [^ 抑制所有类型的警告]
```

    - 参数列表：

关键字	用途
all	抑制所有警告
boxing	抑制装箱、拆箱操作时候的警告
cast	抑制映射相关的警告
dep-ann	抑制启用注释的警告
deprecation	抑制过期方法警告
fallthrough	抑制确在switch中缺失breaks的警告
finally	抑制finally模块没有返回的警告
hiding	抑制相对于隐藏变量的局部变量的警告
incomplete-switch	忽略没有完整的switch语句
nls	忽略非nls格式的字符
null	忽略对null的操作
rawtypes	使用generics时忽略没有指定相应的类型
restriction	抑制禁止使用劝阻或禁止引用的警告
serial	忽略在serializable类中没有声明serialVersionUID变量
static-access	抑制不正确的静态访问方式警告
synthetic-access	抑制子类没有按最优方法访问内部类的警告
unchecked	抑制没有进行类型检查操作的警告
unqualified-field-access	抑制没有权限访问的域的警告
unused	抑制没被使用过的代码的警告

2.4、元注解

2.4.1、简介

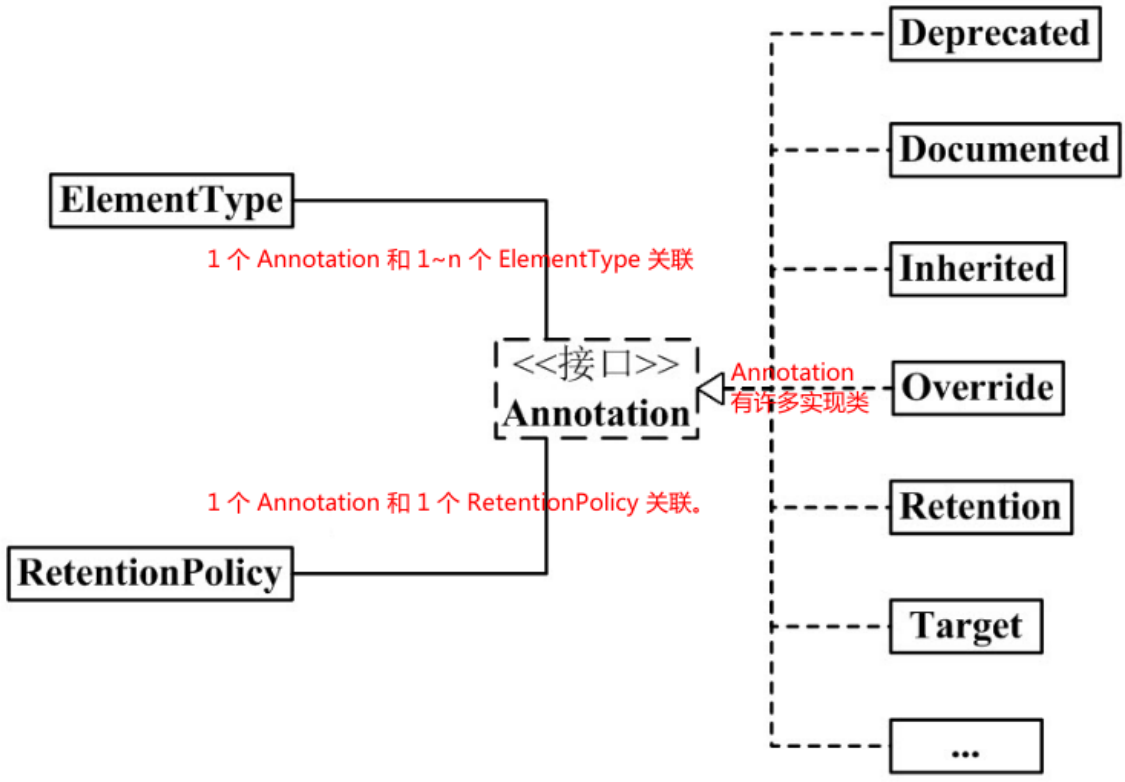
作用在其他注解的注解

2.4.2、有哪些？

- @Retention - 标识这个注解怎么保存，是只在代码中，还是编入class文件中，或者是在运行时可以通过反射访问。
- @Documented - 标记这些注解是否包含在用户文档中 javadoc。
- @Target - 标记这个注解应该是哪种 Java 成员。
- @Inherited - 标记这个注解是自动继承的
    - ```
      1.  子类会继承父类使用的注解中被@Inherited修饰的注解
      2.  接口继承关系中，子接口不会继承父接口中的任何注解，不管父接口中使用的注解有没有被@Inherited修饰
      3.  类实现接口时不会继承任何接口中定义的注解
      ```

2.5、自定义注解

2.5.1、注解架构



01) Annotation与RetentionPolicy 与ElementType 。

每 1 个 Annotation 对象，都会有唯一的 RetentionPolicy 属性；至于 ElementType 属性，则有 1~n 个。

(02) ElementType(注解的用途类型)
      ```

"每 1 个 Annotation" 都与 "1 ~ n 个 ElementType" 关联。当 Annotation 与某个 ElementType 关联时，就意味着：Annotation有了某种用途。例如，若一个 Annotation 对象是 METHOD 类型，则该 Annotation 只能用来修饰方法。

```
package java.lang.annotation;

public enum ElementType {
    TYPE,               /* 类、接口（包括注释类型）或枚举声明  */

    FIELD,              /* 字段声明（包括枚举常量）  */

    METHOD,             /* 方法声明  */

    PARAMETER,          /* 参数声明  */

    CONSTRUCTOR,        /* 构造方法声明  */

    LOCAL_VARIABLE,     /* 局部变量声明  */

    ANNOTATION_TYPE,    /* 注释类型声明  */

    PACKAGE             /* 包声明  */
}
```

(03) RetentionPolicy（注解作用域策略）。

"每 1 个 Annotation" 都与 "1 个 RetentionPolicy" 关联。

- a) 若 Annotation 的类型为 SOURCE，则意味着：Annotation 仅存在于编译器处理期间，编译器处理完之后，该 Annotation 就没用了。 例如，" @Override" 标志就是一个 Annotation。当它修饰一个方法的时候，就意味着该方法覆盖父类的方法；并且在编译期间会进行语法检查！编译器处理完后，"@Override" 就没有任何作用了。
- b) 若 Annotation 的类型为 CLASS，则意味着：编译器将 Annotation 存储于类对应的 .class 文件中，它是 Annotation 的默认行为。
- c) 若 Annotation 的类型为 RUNTIME，则意味着：编译器将 Annotation 存储于 class 文件中，并且可由JVM读入。

```
package java.lang.annotation;
public enum RetentionPolicy {
    SOURCE,             /* Annotation信息仅存在于编译器处理期间，编译器处理完之后就没有该Annotation信息了  */

    CLASS,              /* 编译器将Annotation存储于类对应的.class文件中。默认行为  */

    RUNTIME             /* 编译器将Annotation存储于class文件中，并且可由JVM读入 */
}
```

2.5.2、定义格式

- @interface 自定义注解名{}

2.5.3、注意事项

1. 定义的注解，自动继承了java.lang,annotation.Annotation接口
2. 注解中的每一个方法，实际是声明的注解配置参数

- 方法的名称就是 配置参数的名称
- 方法的返回值类型，就是配置参数的类型。只能是：基本类型/Class/String/enum

3. 可以通过default来声明参数的默认值
4. 如果只有一个参数成员，一般参数名为value
5. 注解元素必须要有值，我们定义注解元素时，经常使用空字符串、0作为默认值。

2.5.4、案例

```
@Documented
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation1 {
    参数类型 参数名() default 默认值;
}
```

上面的作用是定义一个 Annotation，我们可以在代码中通过 "@MyAnnotation1" 来使用它。@Documented, @Target, @Retention, @interface 都是来修饰 MyAnnotation1 的。含义：

(01) @interface

使用 @interface 定义注解时，意味着它实现了 java.lang.annotation.Annotation 接口，即该注解就是一个Annotation。

定义 Annotation 时，@interface 是必须的。

注意：它和我们通常的 implemented 实现接口的方法不同。Annotation 接口的实现细节都由编译器完成。通过 @interface 定义注解后，该注解不能继承其他的注解或接口。

(02) @Documented

类和方法的 Annotation 在缺省情况下是不出现在 javadoc 中的。如果使用 @Documented 修饰该 Annotation，则表示它可以出现在 javadoc 中。

定义 Annotation 时，@Documented 可有可无；若没有定义，则 Annotation 不会出现在 javadoc 中。

(03) @Target(ElementType.TYPE)

前面我们说过，ElementType 是 Annotation 的类型属性。而 @Target 的作用，就是来指定 Annotation 的类型属性。

@Target(ElementType.TYPE) 的意思就是指定该 Annotation 的类型是 ElementType.TYPE。这就意味着，MyAnnotation1 是来修饰"类、接口（包括注释类型）或枚举声明"的注解。

定义 Annotation 时，@Target 可有可无。若有 @Target，则该 Annotation 只能用于它所指定的地方；若没有 @Target，则该 Annotation 可以用于任何地方。

(04) @Retention(RetentionPolicy.RUNTIME)

前面我们说过，RetentionPolicy 是 Annotation 的策略属性，而 @Retention 的作用，就是指定 Annotation 的策略属性。

@Retention(RetentionPolicy.RUNTIME) 的意思就是指定该 Annotation 的策略是 RetentionPolicy.RUNTIME。这就意味着，编译器会将该 Annotation 信息保留在 .class 文件中，并且能被虚拟机读取。

定义 Annotation 时，@Retention 可有可无。若没有 @Retention，则默认是 RetentionPolicy.CLASS。

3、反射

3.1、概述

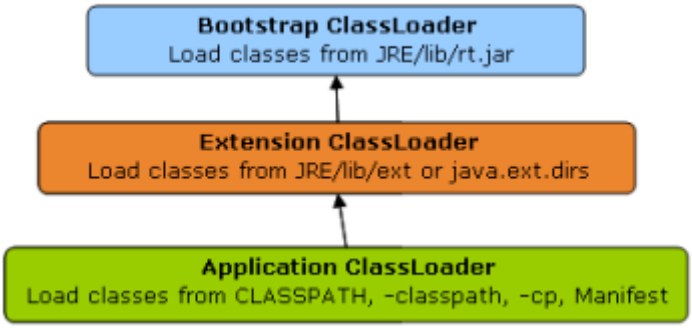
JAVA反射机制是在运行状态中，获取任意一个类的结构 ， 创建对象 ， 得到方法，执行方法 ， 属性 !；这种在运行状态动态获取信息以及动态调用对象方法的功能被称为java语言的反射机制。

3.2、类加载器

Java类加载器（Java Classloader）是Java运行时环境（Java Runtime Environment）的一部分，负责动态加载Java类到Java虚拟机的内存空间中。

java默认有三种类加载器，BootstrapClassLoader、ExtensionClassLoader、App ClassLoader。

BootstrapClassLoader（引导启动类加载器）：
　　嵌在JVM内核中的加载器，该加载器是用C++语言写的，主要负载加载JAVA_HOME/lib下的类库，引导启动类加载器无法被应用程序直接使用。
ExtensionClassLoader（扩展类加载器）：
　　ExtensionClassLoader是用JAVA编写，且它的父类加载器是Bootstrap。
　　是由sun.misc.Launcher$ExtClassLoader实现的，主要加载JAVA_HOME/lib/ext目录中的类库。
　　它的父加载器是BootstrapClassLoader
App ClassLoader（应用类加载器）：
　　App ClassLoader是应用程序类加载器，负责加载应用程序classpath目录下的所有jar和class文件。它的父加载器为Ext ClassLoader



类通常是按需加载，即第一次使用该类时才加载。由于有了类加载器，Java运行时系统不需要知道文件与文件系统。学习类加载器时，掌握Java的委派概念很重要。

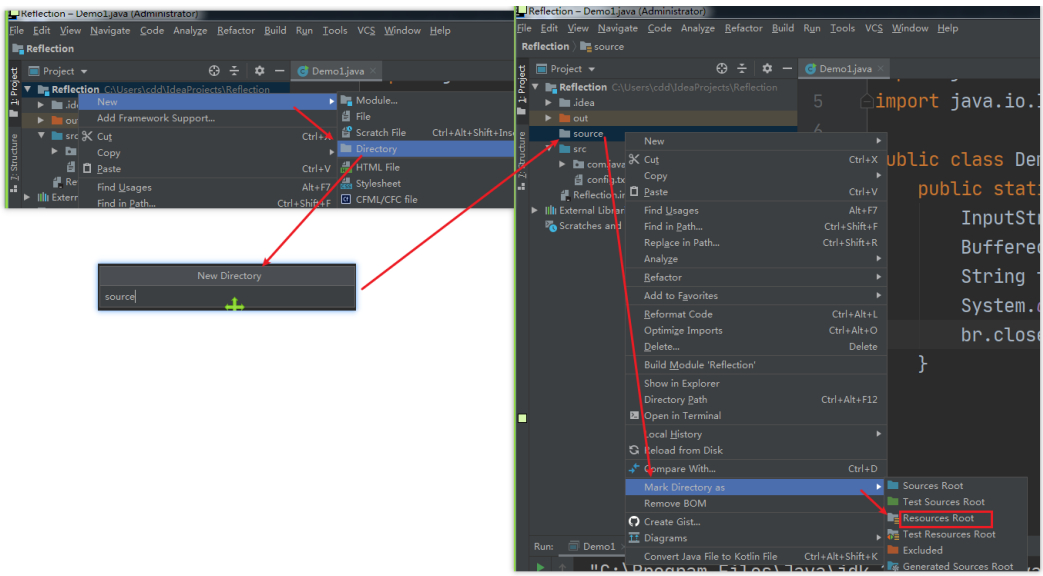
双亲委派模型：如果一个类加载器收到了一个类加载请求，它不会自己去尝试加载这个类，而是把这个请求转交给父类加载器去完成。每一个层次的类加载器都是如此。因此所有的类加载请求都应该传递到最顶层的启动类加载器中，只有到父类加载器反馈自己无法完成这个加载请求（在它的搜索范围没有找到这个类）时，子类加载器才会尝试自己去加载。委派的好处就是避免有些类被重复加载。

- 源码

```
protected synchronized Class<?> loadClass(String paramString, boolean paramBoolean)
    throws ClassNotFoundException
  {
    //检查是否被加载过
    Class localClass = findLoadedClass(paramString);
    //如果没有加载，则调用父类加载器
    if (localClass == null) {
      try {
        //父类加载器不为空
        if (this.parent != null)
          localClass = this.parent.loadClass(paramString, false);
        else {
          //父类加载器为空，则使用启动类加载器
          localClass = findBootstrapClass0(paramString);
        }
      }
      catch (ClassNotFoundException localClassNotFoundException)
      {
        //如果父类加载失败，则使用自己的findClass方法进行加载
        localClass = findClass(paramString);
      }
    }
    if (paramBoolean) {
      resolveClass(localClass);
    }
    return localClass;
  }
```

3.2.1、加载配置文件

- 给项目添加resource root目录
  - 如图



- 通过类加载器加载资源文件
  - 默认加载的是src路径下的文件，但是当项目存在resource root目录时，就变为了加载resource root下的文件了。

3.3、所有类型的Class对象

要想了解一个类,必须先要获取到该类的字节码文件对象.
在Java中，每一个字节码文件，被夹在到内存后，都存在一个对应的Class类型的对象

3.4、得到Class的几种方式

1. 如果在编写代码时，指导类的名称，且类已经存在，可以通过
   包名.类名.class 得到一个类的 类对象

2. 如果拥有类的对象，可以通过
   Class 对象.getClass() 得到一个类的 类对象

3. 如果在编写代码时，知道类的名称 ，可以通过
   Class.forName(包名+类名): 得到一个类的 类对象

上述的三种方式，在调用时，如果类在内存中不存在，则会加载到内存 ！ 如果类已经在内存中存在，不会重复加载，而是重复利用 ！

(一个class文件 在内存中不会存在两个类对象)

特殊的类对象

    基本数据类型的类对象:
        基本数据类型.clss
        包装类.type
    基本数据类型包装类对象:
        包装类.class

3.5、获取Constructor

3.5.1、通过class对象 获取一个类的构造方法

1. 通过指定的参数类型，获取指定的单个构造方法

    getConstructor(参数类型的class对象数组)

    例如:

        构造方法如下: Person(String name,int age)

        得到这个构造方法的代码如下:

        Constructor c = p.getClass().getConstructor(String.class,int.class);

2. 获取构造方法数组

    getConstructors();

3. 获取所有权限的单个构造方法
    getDeclaredConstructor(参数类型的class对象数组)

4. 获取所有权限的构造方法数组
    getDeclaredConstructors();

3.5.2、Constructor 创建对象

常用方法:

    newInstance(Object... para)
        调用这个构造方法， 把对应的对象创建出来
        参数: 是一个Object类型可变参数， 传递的参数顺序 必须匹配构造方法中形式参数列表的顺序!

   setAccessible(boolean flag)
        如果flag为true 则表示忽略访问权限检查 !(可以访问任何权限的方法)

3.6、获取Method

3.6.1、通过class对象 获取一个类的方法

```
1.  getMethod(String methodName , class.. clss)
    根据参数列表的类型和方法名，得到一个方法(public修饰的)

2.  getMethods();
    得到一个类的所有方法 (public修饰的)

3.  getDeclaredMethod(String methodName , class.. clss)
    根据参数列表的类型和方法名，得到一个方法(除继承以外所有的:包含私有，共有，保护，默认)

4.  getDeclaredMethods();
    得到一个类的所有方法 (除继承以外所有的:包含私有，共有，保护，默认)
```

3.6.2、Method 执行方法

```
invoke(Object o,Object... para) :

    调用方法 ，
    参数1. 要调用方法的对象
    参数2. 要传递的参数列表

getName()
    获取方法的方法名称


setAccessible(boolean flag)

        如果flag为true  则表示忽略访问权限检查 !(可以访问任何权限的方法)
```

3.7、获取Field

3.7.1、通过class对象 获取一个类的属性

```
1.  getDeclaredField(String filedName)
        根据属性的名称，获取一个属性对象 (所有属性)

2.  getDeclaredFields()
        获取所有属性

3.  getField(String filedName)
        根据属性的名称，获取一个属性对象 (public属性)

4.  getFields()
        获取所有属性 (public)
```

3.7.2、Field 属性的对象类型

```
常用方法:

    1.  get(Object o );
        参数: 要获取属性的对象
        获取指定对象的此属性值

    2.  set(Object o , Object value);
```

```
        参数1. 要设置属性值的 对象
        参数2. 要设置的值
        设置指定对象的属性的值

    3.  getName()
        获取属性的名称

    4.  setAccessible(boolean flag)

        如果flag为true  则表示忽略访问权限检查 !(可以访问任何权限的属性)
```

3.8、获取注解信息

3.8.1、获取类/属性/方法的全部注解对象

```
Annotation[] annotations01 = Class/Field/Method.getAnnotations();
for (Annotation annotation : annotations01) {
    System.out.println(annotation);
}
```

3.8.2、根据类型获取类/属性/方法的注解对象

```
注解类型 对象名 = (注解类型) c.getAnnotation(注解类型.class);
```

4、内省

4.1、简介

```
基于反射 , java所提供的一套应用到JavaBean的API

    一个定义在包中的类 ,
            拥有无参构造器
            所有属性私有,
            所有属性提供get/set方法
            实现了序列化接口

    这种类, 我们称其为 bean类 .

Java提供了一套java.beans包的api , 对于反射的操作, 进行了封装 !
```

4.2、Introspector

```
获取Bean类信息

方法:
    BeanInfo getBeanInfo(Class cls)

    通过传入的类信息, 得到这个Bean类的封装对象 .
```

4.3、BeanInfo

常用的方法：

MethodDescriptor[] getPropertyDescriptors():

获取bean类的 get/set方法 数组

4.4、MethodDescriptor

常用方法：

1. Method getReadMethod();

   获取一个get方法

2. Method getWriteMethod();
   获取一个set方法

有可能返回null 注意 ,加判断 ！