day16 枚举&反射&注解

第一章 枚举

1. 学习目标

- 了解枚举的概念
- 掌握枚举的格式
- 掌握枚举的应用场景
- 掌握枚举的使用

2. 内容讲解

2.1 枚举的概述

枚举是 Java 中一种特殊的类，它可以定义固定数量的枚举实例，例如: 性别、交通信号灯、季节等等

2.2 为什么要使用枚举

假设我们要定义一个人类，人类中包含姓名和性别。通常会将性别定义成字符串类型，效果如下：

```
public class Person {
    private String name;
    private String sex;

    public Person() {
    }

    public Person(String name, String sex) {
        this.name = name;
        this.sex = sex;
    }

    // 省略get/set/toString方法
}
```

```
public class Demo01 {
    public static void main(String[] args) {
        Person p1 = new Person("张三", "男");
        Person p2 = new Person("张三", "abc"); // 因为性别是字符串,所以我们可以传入任意字符串
    }
}
```

不使用枚举存在的问题：可以给性别传入任意的字符串，导致性别是非法的数据，不安全。

2.3 作用

一个方法接收的参数是固定范围之内的时候，那么即可使用枚举类型

2.4 格式

```
enum 枚举名 {
    第一行都是罗列枚举实例,这些枚举实例直接写大写名字即可。
}
```

2.5 入门案例

1. 定义枚举：MALE表示男，FEMALE表示女

```
enum Gender {
    MALE, FEMALE; // 男，女
}
```

2. Perosn中的性别有String类型改为Sex枚举类型

```
public class Person {
    private String name;
    private Gender gender;

    public Person() {
    }

    public Person(String name, Gender gender) {
        this.name = name;
        this.gender = gender;
    }
    // 省略get/set/toString方法
}
```

3. 使用是只能传入枚举中的固定值

```
public class Demo02 {
    public static void main(String[] args) {
        Person p1 = new Person("张三", Gender.MALE);
        Person p2 = new Person("张三", Gender.FEMALE);
        Person p3 = new Person("张三", "abc");
    }
}
```

2.6 枚举中添加成员变量和成员方法

枚举的本质是一个类，所以枚举中还可以有成员变量，成员方法等。

```
public enum Gender {
    MALE("male"), FEMALE("female");
    public String tag;

    Sex(String tag) {
        this.tag = tag;
    }

    public void showTag() {
        System.out.println("它的性别标志是: " + tag);
    }
}
```

```
public class Demo03 {
    public static void main(String[] args) {
        Person p1 = new Person("张三", Sex.BOY);
        Person p2 = new Person("张三", Sex.GIRL);

        Sex.BOY.showTag();
        Sex.GIRL.showTag();
    }
}
```

第二章 反射

1. 学习目标

- 了解类的加载过程
- 理解类初始化过程
- 了解类加载器
- 掌握获取Class对象的四种方式
- 能够运用反射获取类型的详细信息
- 能够运用反射动态创建对象
- 能够运用反射动态获取成员变量并使用
- 能够运用反射动态获取成员方法并使用
- 能够运用反射获取泛型父类的类型参数

2. 内容讲解

2.1 类加载(了解)

类在内存中的生命周期：加载-->使用-->卸载

2.1.1 类的加载过程

当程序主动使用某个类时，如果该类还未被加载到内存中，系统会通过加载、连接、初始化三个步骤来对该类进行初始化，如果没有意外，JVM将会连续完成这三个步骤，所以有时也把这三个步骤统称为类加载。

类的加载又分为三个阶段：

（1）加载：load

就是指将类型的class字节码数据读入内存

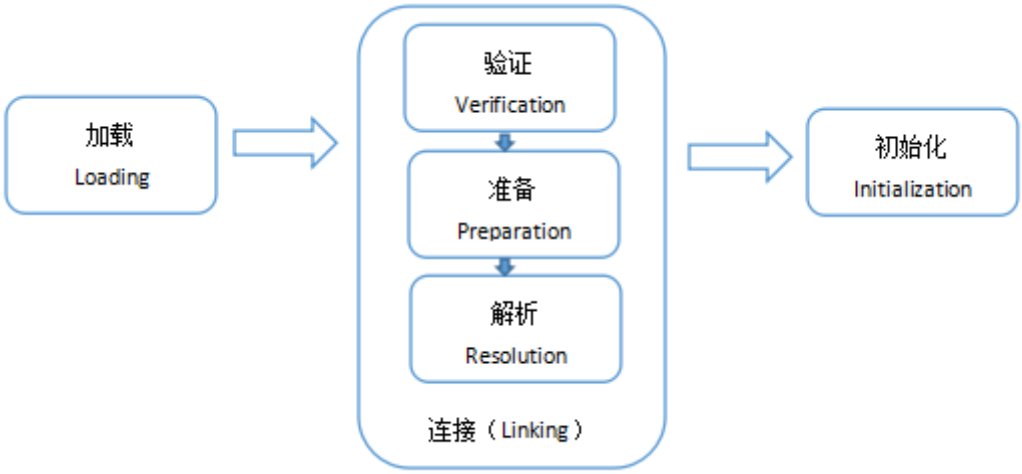
（2）连接：link

①验证：校验合法性等

②准备：准备对应的内存（方法区），创建Class对象，为类变量赋默认值，为静态常量赋初始值。

③解析：把字节码中的符号引用替换为对应的直接地址引用

（3）初始化：initialize（类初始化）即执行类初始化方法，会给类的静态变量赋初始值



2.1.2 类初始化

1、哪些操作会导致类的初始化?

（1）运行主方法所在的类，要先完成类初始化，再执行main方法

（2）第一次使用某个类型就是在new它的对象，此时这个类没有初始化的话，先完成类初始化再做实例初始化

（3）调用某个类的静态成员（类变量和类方法），此时这个类没有初始化的话，先完成类初始化

（4）子类初始化时，发现它的父类还没有初始化的话，那么先初始化父类

（5）通过反射操作某个类时，如果这个类没有初始化，也会导致该类先初始化

```
class Father{
    static{
        System.out.println("main方法所在的类的父类(1)");//初始化子类时，会初始化父类
    }
}

public class TestClinit1 extends Father{
    static{
        System.out.println("main方法所在的类(2)");//主方法所在的类会初始化
    }

    public static void main(String[] args) throws ClassNotFoundException {
        new A();//第一次使用A就是创建它的对象，会初始化A类

        B.test();//直接使用B类的静态成员会初始化B类

        Class clazz = Class.forName("com.atguigu.test02.C");//通过反射操作C类，会初始化C类
    }
```

```
}
class A{
    static{
        System.out.println("A类初始化");
    }
}
class B{
    static{
        System.out.println("B类初始化");
    }
    public static void test(){
        System.out.println("B类的静态方法");
    }
}
class C{
    static{
        System.out.println("C类初始化");
    }
}
```

2、哪些使用类的操作，但是不会导致类的初始化?

（1）使用某个类的静态的常量（static final）

（2）通过子类调用父类的静态变量，静态方法，只会导致父类初始化，不会导致子类初始化，即只有声明静态成员的类才会初始化

（3）用某个类型声明数组并创建数组对象时，不会导致这个类初始化

```
public class TestClinit2 {
    public static void main(String[] args) {
        System.out.println(D.NUM);

        System.out.println(F.num);
        F.test();

        G[] arr = new G[5];
    }
}
class D{
    public static final int NUM = 10;
    static{
        System.out.println("D类的初始化");
    }
}
class E{
    static int num = 10;
    static{
        System.out.println("E父类的初始化");
    }
    public static void test(){
        System.out.println("父类的静态方法");
    }
}
class F extends E{
    static{
```

```
        System.out.println("F子类的初始化");
    }
}

class G{
    static{
        System.out.println("G类的初始化");
    }
}
```

2.1.3 类加载器

很多开发人员都遇到过java.lang.ClassNotFoundException或java.lang.NoClassDefError，想要更好的解决这类问题，或者在一些特殊的应用场景，比如需要支持类的动态加载或需要对编译后的字节码文件进行加密解密操作，那么需要你自定义类加载器，因此了解类加载器及其类加载机制也就成了每一个Java开发人员的必备技能之一。

2.1.3.1 类加载器分为:

（1）引导类加载器（Bootstrap Classloader）又称为根类加载器

```
它负责加载jre/lib中的核心库
它本身不是Java代码实现的，也不是ClassLoader的子类，获取它的对象时往往返回null
```

（2）扩展类加载器（Extension ClassLoader）

```
它负责加载jre/lib/ext扩展库
它是ClassLoader的子类
```

（3）应用程序类加载器（Application Classloader）

```
它负责加载项目的classpath路径下的类

它是ClassLoader的子类
```

（4）自定义类加载器

```
当你的程序需要加载“特定”目录下的类，可以自定义类加载器；
当你的程序的字节码文件需要加密时，那么往往会提供一个自定义类加载器对其进行解码
后面会见到的自定义类加载器：tomcat中
```

2.1.3.2 Java系统类加载器的双亲委托模式

简单描述：

```
下一级的类加载器，如果接到任务时，会先搜索是否加载过，如果没有，会先把任务往上传，如果都没有加载过，一直到根加载器，如果根加载器在它负责的路径下没有找到，会往回传，如果一路回传到最后一级都没有找到，那么会报ClassNotFoundException或NoClassDefError，如果在某一级找到了，就直接返回Class对象。
```

应用程序类加载器 把扩展类加载器视为父加载器，

扩展类加载器 把 引导类加载器视为父加载器。

不是继承关系，是组合的方式实现的。

2.2 java.lang.Class类

Java反射机制是在运行状态中，对于任意一个类，都能够知道这个类的所有属性和方法；对于任意一个对象，都能够调用它的任意一个方法和属性；这种动态获取的信息以及动态调用对象的方法的功能称为Java语言的反射机制。

要想解剖一个类，必须先要获取到该类的Class对象。而剖析一个类或用反射解决具体的问题就是使用相关API（1）java.lang.Class（2）java.lang.reflect.*。所以，Class对象是反射的根源。

2.2.1 哪些类型可以获取Class对象

所有Java类型

用代码示例

```
//（1）基本数据类型和void
例如：int.class
     void.class
//（2）类和接口
例如：String.class
    Comparable.class
//（3）枚举
例如：ElementType.class
//（4）注解
例如：Override.class
//（5）数组
例如：int[].class
```

2.2.2 获取Class对象的四种方式

（1）类型名.class

要求编译期间已知类型

（2）对象.getClass()

获取对象的运行时类型

（3）Class.forName(类型全名称)，通常需要配置文件配置配合使用

可以获取编译期间未知的类型

（4）ClassLoader的类加载器对象.loadClass(类型全名称)

可以用系统类加载对象或自定义加载器对象加载指定路径下的类型

```
//第一种方式获取Class: 类型名.class
//Class clazz = Person.class;

//第二种方式获取Class: 对象.getClass()
//Person person = new Person();
//Class clazz = person.getClass();

//第三种方式获取Class: Class.forName("类的全限定名")
Class clazz = Class.forName("com.atguigu.Person");
```

2.3 反射的概念

反射是一种机制/功能，利用该机制/功能可以在**程序运行**过程中对类进行解剖并操作类中的构造方法，成员方法，成员属性。

2.4 反射的应用场景

各种框架的设计(主要场景)

各大框架的内部实现也大量使用到了反射机制，所以要想学好这些框架，则必须要求了解反射机制

2.5 反射的应用

2.5.1 获取类型的详细信息

可以获取：包、修饰符、类型名、父类（包括泛型父类）、父接口（包括泛型父接口）、成员（属性、构造器、方法）、注解（类上的、方法上的、属性上的）

2.5.1.1 获取包信息

```
Package pkg = clazz.getPackage();
```

2.5.1.2 获取修饰符

```
int mod = clazz.getModifiers();
```

修饰符定义在Modifier类中，该类里面有很多常量值，每一个常量对应一种修饰符

2.5.1.3 获取类名

```
String name = clazz.getName();
```

2.5.1.4 获取父类的字节码对象

```
Class superclass = clazz.getSuperclass();
```

2.5.1.5 获取该类实现的所有接口

```
Class[] interfaces = clazz.getInterfaces();
```

2.5.1.6 获取该类的所有属性

```
Field[] declaredFields = clazz.getDeclaredFields();
```

2.5.1.7 获取该类的所有构造函数

```
Method[] declaredMethods = clazz.getDeclaredMethods();
```

2.5.1.8 获取该类的所有方法

```
Method[] declaredMethods = clazz.getDeclaredMethods();
```

2.5.2 创建任意引用类型的对象(重点)

两种方式：

1、直接通过Class对象来实例化（要求必须有无参构造）

2、通过获取构造器对象来进行实例化

方式一的步骤：

（1）获取该类型的Class对象（2）创建对象

```
@Test
public void test2()throws Exception{
    Class<?> clazz = Class.forName("com.atguigu.test.Student");
    //Caused by: java.lang.NoSuchMethodException: com.atguigu.test.Student.<init>()
    //即说明Student没有无参构造，就没有无参实例初始化方法<init>
    Object stu = clazz.newInstance();
    System.out.println(stu);
}

@Test
public void test1() throws ClassNotFoundException, InstantiationException, IllegalAccessException{
    //使用第一种方式创建Person类的对象
    //强转一定是建立在父子关系的前提下
    /*Person person = (Person) clazz.newInstance();
      System.out.println(person);*/
}
```

方式二的步骤：

（1）获取该类型的Class对象（2）获取构造器对象（3）创建对象

> 如果构造器的权限修饰符修饰的范围不可见，也可以调用setAccessible(true)

示例代码：

```
public static void main(String[] args) throws Exception {
    //获取Person类的Class对象
    Class clazz = Person.class;

    //使用第一种方式创建Person类的对象
```

```
    //强转一定是建立在父子关系的前提下
    /*Person person = (Person) clazz.newInstance();
        System.out.println(person);*/

    //使用第二种方式创建Person类的对象
    //获取无参的构造函数
    //Constructor constructor = clazz.getDeclaredConstructor();

    Constructor constructor =
clazz.getDeclaredConstructor(int.class,String.class, String.class);

    //使用构造函数创建对象
    Person person = (Person) constructor.newInstance(40,"奥巴马","召唤师峡谷");
    System.out.println(person);
}
```

2.3.3 操作任意类型的属性(重点)

(1) 获取该类型的Class对象

```
Class clazz = Class.forName("com.atguigu.bean.User");
```

(2) 获取属性对象

```
Field field = clazz.getDeclaredField("username");
```

(3) 设置属性可访问

```
field.setAccessible(true);
```

(4) 创建实例对象：如果操作的是非静态属性，需要创建实例对象

```
Object obj = clazz.newInstance();
```

(4) 设置属性值

```
field.set(obj,"chai");
```

(5) 获取属性值

```
Object value = field.get(obj);
```

> 如果操作静态变量，那么实例对象可以省略，用null表示，当然一般不会使用反射操作静态变量

示例代码：

```
public static void main(String[] args) throws Exception {
    //1. 获取Person的字节码对象
    Class clazz = Person.class;

    Object obj = clazz.newInstance();
    //2.1 获取Person的所有属性(只能获取自己的，包含公有的和私有的)
```

```
    /*Field[] declaredFields = clazz.getDeclaredFields();
        for (Field declaredField : declaredFields) {
            //获取每个属性的属性名和属性值
            //获取属性名
            String name = declaredField.getName();
            //获取属性的类型
            Class<?> type = declaredField.getType();
            //获取属性的修饰符
            int modifiers = declaredField.getModifiers();

            //暴力反射：通过反射可以访问类的私有成员
            declaredField.setAccessible(true);

            //获取属性的值
            Object value = declaredField.get(obj); //等值于 对象.属性名

            System.out.println(name + "," + value + "," + type + "," +
modifiers);
        }*/

    //2.2 单独获取某一个属性,比如获取name
    Field filed = clazz.getDeclaredField("address");

    //设置其属性值为"北京"
    filed.set(obj,"北京");

    //获取其属性值
    String address = (String) filed.get(obj);
    System.out.println(address);
}
```

2.3.4 调用任意类型的方法

（1）获取该类型的Class对象

```
Class clazz = Class.forName("com.atguigu.service.UserService");
```

（2）获取方法对象

```
Method method = clazz.getDeclaredMethod("login",String.class,String.class);
```

（3）创建实例对象

```
Object obj = clazz.newInstance();
```

（4）调用方法

```
Object result = method.invoke(obj,"chai","123);
```

> 如果方法的权限修饰符修饰的范围不可见，也可以调用setAccessible(true)
>
> 如果方法是静态方法，实例对象也可以省略，用null代替

示例代码：

```java
public static void main(String[] args) throws Exception {
    //使用反射操作类的方法: 获取方法、调用方法
    //1. 获取类的字节码对象
    Class clazz= Person.class;

    Object obj = clazz.newInstance();

    //2. 获取某一个方法,例如: getName()
    //获取无参的getName方法
    Method getNameMethod = clazz.getDeclaredMethod("getName");
    //获取带一个String类型参数的study方法
    Method studyMethod = clazz.getDeclaredMethod("study", String.class,
int.class);

    //调用方法
    String name = (String) getNameMethod.invoke(obj);
    System.out.println("获取到的name:" + name);

    //暴力反射
    studyMethod.setAccessible(true);
    studyMethod.invoke(obj,"Java",180);
}
```

2.3.5 Type接口的介绍(了解)

`java.lang.reflect.Type`接口及其相关接口用于描述java中用到的所有类型，是Java的反射中很重要的组成部分。Type 是 Java 编程语言中所有类型的公共高级接口。它们包括原始类型、参数化类型、数组类型、类型变量和基本类型。

2.3.5.1 使用反射获取Type

有很多场景下我们可以获得Type，比如：

1. 当我们拿到一个Class，用`Class.getGenericInterfaces()`方法得到Type[]，也就是这个类实现接口的Type类型列表。
2. 当我们拿到一个Class，用`Class.getDeclaredFields()`方法得到Field[]，也就是类的属性列表，然后用`Field. getGenericType()`方法得到这个属性的Type类型。
3. 当我们拿到一个Method，用`Method.getGenericParameterTypes()`方法获得Type[]，也就是方法的参数类型列表。
4. 当我们拿到一个Class，用`clazz.getGenericSuperclass()`这样就可以获取父类的泛型实参列表

2.3.5.2 Type的分类

Type接口包含了一个实现类(Class)和四个实现接口(TypeVariable, ParameterizedType, GenericArrayType, WildcardType)，这四个接口都有自己的实现类，但这些实现类开发都不能直接使用，只能用接口。

1. Class: 当需要描述的类型是普通Java类、数组、自定义类、 8种java基本类型 的时候， java会选择Class来作为这个Type的实现类，我们甚至可以直接把这个Type强行转换类型为Class。这些类基本都有一个特点：**基本和泛型无关**，其他4种Type的类型，基本都是泛型的各种形态。
2. ParameterizedType: 当需要描述的类是**泛型类**时，比如List,Map等，不论代码里写没写具体的泛型，java会选择ParameterizedType接口做为Type的实现。ParameterizedType接口有getActualTypeArguments()方法，用于得到泛型的Type类型数组。
3. GenericArrayType: 当需要描述的类型是**泛型类的数组**时，比如比如List[],Map[]，type用GenericArrayType接口作为Type的实现。GenericArrayType接口有

getGenericComponentType()方法，得到数组的组件类型的Type对象。

4. WildcardType: 当需要描述的类型是泛型类，而且泛型类中的泛型被定义为(? extends xxx)或者(? super xxx)这种类型，比如List<? extends TestReflect>，这个类型首先将由ParameterizedType实现，当调用ParameterizedType的getActualTypeArguments()方法后得到的Type就由WildcardType实现。

2.3.6 获取泛型父类信息

示例代码获取泛型父类信息：

```
public class TestGeneric {
    public static void main(String[] args) {
        //需求：在运行时，获取Son类型的泛型父类的泛型实参<String,Integer>

        //（1）还是先获取Class对象
        Class clazz = Son.class;//四种形式任意一种都可以

        //（2）获取泛型父类
        /*
         * getSuperclass()只能得到父类名，无法得到父类的泛型实参列表
         */
        Type type = clazz.getGenericSuperclass();

        // Father<String,Integer>属于ParameterizedType
        ParameterizedType pt = (ParameterizedType) type;

        //（3）获取泛型父类的泛型实参列表
        Type[] typeArray = pt.getActualTypeArguments();
        for (Type type2 : typeArray) {
            System.out.println(type2);
        }
    }
}
//泛型形参：<T,U>
class Father<T,U>{

}
//泛型实参：<String,Integer>
class Son extends Father <String,Integer>{

}
```

2.3.7 动态创建和操作任意类型的数组

在java.lang.reflect包下还提供了一个Array类，Array对象可以代表所有的数组。程序可以通过使用Array类来动态的创建数组，操作数组元素等。

Array类提供了如下几个方法：

public static Object newInstance(Class<?> componentType, int... dimensions)：创建一个具有指定的组件类型和维度的新数组。

public static void setXxx(Object array,int index,xxx value)：将array数组中[index]元素的值修改为value。此处的Xxx对应8种基本数据类型，如果该属性的类型是引用数据类型，则直接使用set(Object array,int index, Object value)方法。

public static xxx getXxx(Object array,int index,xxx value)：将array数组中[index]元素的值返回。此处的Xxx对应8种基本数据类型，如果该属性的类型是引用数据类型，则直接使用get(Object array,int index)方法。

```
public static void main(String[] args) {
    //使用反射操作数组
    //1. 使用反射创建一个String类型的数组，长度是5
    Object array = Array.newInstance(String.class, 5);

    //2. 往数组中存入数据
    for (int i=0;i<5;i++){
        Array.set(array,i,"value"+i);
    }

    //使用Array获取数组中的元素
    for (int i=0;i<5;i++){
        System.out.println(Array.get(array, i));
    }
}
```

第三章 注解

1. 学习目标

- 了解注解的概念
- 了解JDK提供的三种基本注解
- 掌握自定义注解
- 掌握元注解
- 掌握注解解析

2. 内容讲解

2.1 注解概述

2.1.1 什么是注解

注解英文是annotation,是一种代码级别的说明,和类 接口平级关系。相当于一种标记，在程序中加入注解就等于为程序打上某种标记，以后，javac编译器、开发工具和其他程序可以通过反射来了解你的类及各种元素上有无标记，看你的程序有什么标记，就去干相应的事，标记可以加在包、类，属性、方法，方法的参数以及局部变量上定义

2.1.2 注解的作用

执行编译期的检查 例如:@Override

分析代码(主要用途:替代配置文件)； 用在框架里面, 注解开发

2.2 JDK提供的三个基本的注解

1. `@Override`:描述方法的重写.
2. `@SuppressWarnings`:压制警告.
3. `@Deprecated`:标记过时

2.3 自定义注解(重点)

2.3.1 自定义注解语法

语法: `@interface 注解名{}`

示例

```
/**
 * 定义了注解
 *
 */
public @interface Annotation01 {

}
```

2.3.2 注解属性

2.3.2.1 注解属性的作用

注解属性可以让注解具备携带存储数据的功能

2.3.2.2 注解属性的类型

1. 基本类型

2.String

3.枚举类型

4.注解类型

5.Class类型

6.以上类型的一维数组类型

注意:

一旦注解有属性了,使用注解的时候,属性必须有值

- 示例代码

```
/**
 *注解的属性; 格式和接口的方法很类似
 *  1.基本类型
    2.String
    3.枚举类型
    4.注解类型
    5.Class类型
    6.以上类型的一维数组类型

 */
public @interface Annotation02 {
    int a();//基本类型

    String b();//String

    Color c();//枚举类型

```

```
    Annotation01 d();//注解类型

    Class e();//Class类型

    String[] f();//一维数组类型

}
```

2.3.2.3 使用注解时给属性赋值

- 格式

```
@注解名(属性名=值,属性名2=值2)  eg:@MyAnnotation3(i = 0,s="23")
```

2.3.2.4 属性赋值的特殊情况

- **若属性类型的一维数组的时候,当数组的值只有一个的时候可以省略{}**

```
@MyAnnotation4(ss = { "a" })
@MyAnnotation4(ss = "a")
```

- 注解属性可以有默认值

```
属性类型 属性名() default 默认值;
```

- **若属性名为value的时候,且只有这一个属性需要赋值的时候可以省略value【重点】**

2.4 元注解

2.4.1 元注解的作用

元注解是使用在自定义的注解上，为自定义的注解提供支持的

2.4.2 常用的元注解

`@Target`:定义该注解作用在什么上面(位置),默认注解可以在任何位置. 值为: `ElementType` 的枚举值

`METHOD`:方法

`TYPE`:类 接口

`FIELD`:字段

`CONSTRUCTOR`:构造方法声明

`@Retention`:定义该注解保留到那个代码阶段, 值为: `RetentionPolicy` 类型,**默认只在源码阶段保留**

`SOURCE`:只在源码上保留(默认)

`CLASS`:在源码和字节码上保留

`RUNTIME`:在所有的阶段都保留

java (源码阶段) ----编译---> .class(字节码阶段) ----加载内存--> 运行(RUNTIME)

eg:

```java
@Target(value = {ElementType.METHOD,ElementType.TYPE  })
@Retention(value = RetentionPolicy.RUNTIME)
public @interface MyAnnotation03 {
    int a();
    String b();
}
```

2.5 注解解析

java.lang.reflect.AnnotatedElement

- **T getAnnotation(ClassannotationType):得到指定类型的注解引用。没有返回null。**
- **boolean isAnnotationPresent(Class<?extends Annotation> annotationType)**：判断指定的注解有没有。

  Class、Method、Field、Constructor等实现了AnnotatedElement接口.
- Annotation[] getAnnotations()：得到所有的注解，包含从父类继承下来的。
- Annotation[] getDeclaredAnnotations()：得到自己身上的注解。

```java
public @interface Annotation01(){

}

@Annotation01
class Demo01(){

    @Annotation01
    public void fun01(){

    }

    public void fun02(){

    }
}

//1.获得Demo01字节码对象
Class clazz =  Demo01.class;
//2. 获得Demo01上面的注解对象
Annotation01 annotation01 = clazz.getAnnotation(Annotation01.class);
//3.反射获得fun01()方法对象
Method method =  clazz.getMethod("fun01");
//4.判断fun01()方法上面是否有@Annotation01注解
boolean flag = method.isAnnotationPresent(Annotation01.class);
```