JAVA注解与反射

第九艺术家 分类：学习笔记 浏览：140 评论：0 字体 皮肤 左侧展开 最后修改于： 2023/04/17 11:55:32

展开目录

🏆 前言

Java注解与反射

🏆 一、Java注解

📣 1.注解的定义

Java注解又称Java标注，是在 JDK5 时引入的新特性，注解（也被称为元数据）。

Java注解它提供了一种安全的类似注释的机制，用来将任何的信息或元数据（metadata）与程序元素（类、方法、成员变量等）进行关联。

Java注解是附加在代码中的一些元信息，用于一些工具在编译、运行时进行解析和使用，起到说明、配置的功能。

📣 2.注解与注释的区别

注解是对代码的解释和说明，其目的是提高程序代码的可读性。注解是可以被编译器打包进入class文件。

注释只存在于java源代码中，对于编译和运行没有任何作用，也不会被编译到class文件中。注释是给程序员看的，编译器与JVM都不需要知道它。

📣 3.注解的应用

1.生成文档这是最常见的，也是java 最早提供的注解;

2.在编译时进行格式检查，如@Override放在方法前，如果你这个方法并不是覆盖了超类方法，则编译时就能检查出;

3.跟踪代码依赖性，实现替代配置文件功能，比较常见的是spring 2.5 开始的基于注解配置,作用就是减少配置;

4.在反射的 Class, Method, Field 等函数中，有许多于 Annotation 相关的接口,可以在反射中解析并使用 Annotation。

📣 4.注解的分类

📔 标准注解

📝 1.常用的三种：

```java
//@Override    重写的注解
@Override
public String toString() {
    return super.toString();
}

//@Deprecated      不推荐程序员使用，但是可以使用，或者存在更好的方法
@Deprecated
public static void test1() {
    System.out.println("Deprecated");
}

//@SuppressWarnings("关键字")    镇压警告（不推荐使用），将警告消除
@SuppressWarnings("all")
public void test2() {
    List list = new ArrayList();
}

public static void main(String[] args) {
    test1();
}
```

关键字	被抑制的警告类型
all	抑制所有警告
deprecation	抑制使用了过时的类或方法的警告
fallthrough	抑制switch块中没有break语句的警告
finally	抑制finally块中不能正常完成的警告
rawtypes	抑制没有使用泛型的警告
serial	抑制可序列化类没有使用序列化ID的警告
unchecked	抑制未检查操作的警告

2.函数式接口(Functional Interface)就是一个有且仅有一个抽象方法，但是可以有多个非抽象方法的接口。（附加）

元注解

1.Target

枚举值	注解能够被应用的地方
ElementType.ANNOTATION_TYPE	注解类型的声明
ElementType.CONSTRUCTOR	构造方法的声明
ElementType.FIELD	属性的声明
ElementType.LOCAL_VARIABLE	局部变量的声明
ElementType.METHOD	方法的声明
ElementType.PACKAGE	包的声明
ElementType.PARAMETER	方法参数的声明
ElementType.TYPE	类，接口以及枚举的声明

2.Retention

3.Documented

4.Inherited

code:

```java
@MyAnnotation
public class test02 {

    @MyAnnotation
    public void test02() {

    }
}

//自定义一个简单的注解
    //@Target       表示我们的注解可以用在哪些地方

    //@Retention       表示我们的注解在什么地方有效

    //@Documented       表示是否将我们的注解生成在Javadoc中

    //@Inherited       表示子类可以继承父类的注解
@Target(value = {ElementType.METHOD, ElementType.TYPE})
    @Retention(value = RetentionPolicy.RUNTIME)
    @Documented
    @Inherited
@interface MyAnnotation{

}
```

自定义注解

自定义注解的格式

```java
// 元注解
public @interface 注解名称{
    // 属性列表
}
```

示例：

```java
//自定义注解
public class test03 {
    //注解可以显示赋值， 如果没有默认值，我们就必须给注解赋值
    //注解参数的顺序随意
    @MyAnnotation2(age = 18, name = "jacky")
    public void test() {

    }

    @MyAnnotation3("jacky")
    public void test2() {

    }
}

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation2{
    //注解的参数: 参数类型 + 参数名 + ();
    String name() default "";
    int age() default 0;
    int id() default -1;

    String[] schools() default {"peking university"};
}

@Target({ElementType.TYPE, ElementType.METHOD})
@Retention(RetentionPolicy.RUNTIME)
@interface MyAnnotation3{
    //如果注解参数为value赋值时可以直接写值
    String value();
}
```

🏆 二、Java反射

java中的字节码：Java源代码经过虚拟机编译器编译后产生的文件（即扩展为.class的文件），它不面向任何特定的处理器，只面向虚拟机。

📢 1.反射的定义

📔 反射机制

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字节码文件对象。而解剖使用的就是Class类中的方法.所以先要获取到每一个字节码文件对应的Class类型的对象.

📔 什么是反射

反射就是把java类中的各种成分映射成一个个的Java对象

📢 2.Class类

Class 类的实例表示正在运行的 Java 应用程序中的类和接口。也就是jvm中有N多的实例每个类都有该Class对象。（包括基本数据类型）
Class 没有公共构造方法。Class 对象是在加载类时由 Java 虚拟机以及通过调用类加载器中的defineClass 方法自动构造的。也就是这不需要我们自己去处理创建，JVM已经帮我们创建好了

(具体可查看源码，或查看Java api详解)

所有类型的Class对象

```
Class c1 = Object.class;    //类
Class c2 = Comparable.class;    //接口
Class c3 = String.class;    //一维数组
Class c4 = int[][].class;   //二维数组
Class c5 = Override.class;  //注解
Class c6 = ElementType.class;   //枚举
Class c7 = Integer.class;   //基本数据类型
Class c8 = void.class;  //void
Class c9 = Class.class; //Class

//只要元素类型与维度一致，就是同一个Class
int[] a = new int[10];
int[] b = new int[100];
int[][] c = new int[10][10];
System.out.println(a.getClass().hashCode());
System.out.println(b.getClass().hashCode());
System.out.println(c.getClass().hashCode());
```

3.获取反射对象Class五种方式

通过Object类的方法getClass()继承给所有子类

在Object源码中存在getClass方法：

```
public final native Class<?> getClass(); //final修饰无法重写 native原生的
```

示例：

```
Class c1 = new User("",0,0).getClass();
```

任何数据类型都有静态的class属性

示例：

```
Class c2 = User.class;
```

通过Class类的静态方法:forName(String className)

示例：

```
Class c3 = Class.forName("cn.com.reflection.User");
```

基本内置类型的包装类都有一个Type属性

示例：

```
Class c4 = Integer.TYPE;
```

通过子类Class找到父类Class

示例：

```
Class c4 = User.class.getSuperclass();
```

4.类的加载内存分析

加载

加载，是指Java虚拟机查找字节流（查找.class文件），并且根据字节流创建java.lang.Class对象的过程。这个过程，将类的.class文件中的二进制数据读入内存，放在运行时区域的方法区内。然后在堆中创建java.lang.Class对象，用来封装类在方法区的数据结构。

类加载阶段：

（1）Java虚拟机将.class文件读入内存，并为之创建一个Class对象。

（2）任何类被使用时系统都会为其创建一个且仅有一个Class对象。

（3）这个Class对象描述了这个类创建出来的对象的所有信息，比如有哪些构造方法，都有哪些成员方法，都有哪些成员变量等。

链接

链接包括验证、准备以及解析三个阶段。

（1）验证阶段。主要的目的是确保被加载的类（.class文件的字节流）满足Java虚拟机规范，不会造成安全错误。

（2）准备阶段。负责为类的静态成员分配内存，并设置默认初始值。

（3）解析阶段。将类的二进制数据中的符号引用替换为直接引用。

初始化

初始化，则是为标记为常量值的字段赋值的过程。换句话说，只对static修饰的变量或语句块进行初始化。

如果初始化一个类的时候，其父类尚未初始化，则优先初始化其父类。

如果同时包含多个静态变量和静态代码块，则按照自上而下的顺序依次执行。

具体说法如下：

执行类构造器<clinit>()方法的过程。类构造器<clinit>()方法是由编译期自动收集类中所有类变量的赋值动作和静态代码块中的语句合并产生的。(类构造器是构造类信息的，不是构造该类对象的构造器)。

当初始化一个类的时候，如果发现其父类还没有进行初始化，则需要先触发其父类的初始化虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确加锁和同步。

虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确加锁和同步。

```
public class test03 {
public static void main(String[] args) {
    A a = new A();
    System.out.println(a.m);
}
/*
1.加载到内存，会产生一个类对应的Class对象
2.链接，结束后 m = 0;
3.初始化
    <clinit>() {
        System.out.println("A类静态代码块初始化");
        m = 300;
        m = 100;
    }

 m = 100;
 */
}

class A {
    static {
        System.out.println("A类静态代码块初始化");
        m = 300;
    }

    static int m = 100;

    public A() {
        System.out.println("A类无参构造初始化");
    }
}
```

5.分析类初始化

类的主动引用(一定会发生初始化)

- 当虚拟机启动，先初始化main方法所在的类
- new一个类的对象

- 调用类的静态成员(除了**final**常量)和静态方法
- 使用**java,lang.reflect**包的方法对类进行反射调用
- 当初始化一个类，如果其父类未被初始化，则先会初始化其父类

类的被动引用(不会发生类的初始化)

- 当访问一个静态域时，只有真正声明这个域的类才会被初始化。如：当通过子类引用父类的静态变量，不会导致子类的初始化
- 通过数组定义引用，不会出发此类的初始化
- 引用常量不会触发此类的初始化(常量在链接阶段就存入调用类的常量池中)

```
1. public class test04 {
2.
3.     static {
4.         System.out.println("Main类被加载");
5.     }
6.
7.     public static void main(String[] args) throws ClassNotFoundException {
8.         //1.主动引用
9.         Son son = new Son();
10.
11.        //反射也会产生主动引用
12.        Class.forName("cn.com.reflection.Son");
13.
14.        //不会产生类的引用的方法
15.            //子类引用父类的值
16.        System.out.println(Son.n);
17.            //数组只是开辟一个内存空间，也不会初始化
18.        Son[] a = new Son[5];
19.            //引用常量也不会初始化
20.        System.out.println(Son.M);
21.    }
22. }
23.
24. class Father {
25.
26.    static int n = 2;
27.
28.    static {
29.        System.out.println("父类被加载");
30.    }
31. }
32.
33. class Son extends Father {
34.
35.    static {
36.        System.out.println("子类被加载");
37.        m = 300;
38.    }
39.
40.    static int m = 100;
41.    static final int M = 1;
42. }
```

6.类加载器

类加载器作用是用来把类(class)装载进内存的。JVM规范定义了如下类型的类加载器。

系统类加载器

负责java-classpath或-d java.class.path所指的目录下的类与jar包装入工作库，是最常用的加载器。

扩展类加载器

负责jre/lib/ext或-d java.ext.dirs所指的目录下的类与jar包装入工作库

引导类加载器(根加载器)

用c++编写的，是jvm自带的类加载器，负责java平台核心库，用来装载核心类库。该加载器无法直接获取。

各加载器关系

自定义加载器—>System Classloader—>Extension Classloader—>Bootstap Classloader(从左至右对应从底到顶)
自底向上检查类是否已装载
自顶向底尝试加载类

双亲委派机制

自顶向底尝试加载类时，会检查是否存在了该类的包，如果已存在就不会向下加载子类的包。
双亲委派机制保证了安全性。

```java
public class test05 {
    public static void main(String[] args) throws ClassNotFoundException {

        //获取系统类加载器
        ClassLoader systemClassLoader = ClassLoader.getSystemClassLoader();
        System.out.println(systemClassLoader);

        //获取系统类加载器的父类加载器-->扩展类加载器
        ClassLoader parent = systemClassLoader.getParent();
        System.out.println(parent);

        //获取扩展类加载器的父类加载器-->根加载器(c/c++)
        ClassLoader parent1 = parent.getParent();
        System.out.println(parent1);

        //测试当前类是哪个加载器加载的
        ClassLoader classLoader = Class.forName("cn.com.reflection.test05").getClassLoader();
        System.out.println(classLoader);

        //测试JDK内置的类是谁加载的
        ClassLoader classLoader1 = Class.forName("java.lang.Object").getClassLoader();
        System.out.println(classLoader1);

        //如何获取系统类加载器可以加载的路径
        System.out.println(System.getProperty("java.class.path"));

        /*
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\charsets.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\deploy.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\ext\access-bridge-64.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\ext\cldrdata.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\ext\dnsns.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\ext\jaccess.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\ext\jfxrt.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\ext\localedata.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\ext\nashorn.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\ext\sunec.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\ext\sunjce_provider.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\ext\sunmscapi.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\ext\sunpkcs11.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\ext\zipfs.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\javaws.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\jce.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\jfr.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\jfxswt.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\jsse.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\management-agent.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\plugin.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\resources.jar;
        D:\Program Files\Java\jdk1.8.0_241\jre\lib\rt.jar;
        D:\idea_file\AnnotationAndReflection\out\production\AnnotationAndReflection;
        D:\javafile\IDEA\IntelliJ IDEA 2022.2.4\lib\idea_rt.jar
         */
    }
}
```

📣 7.获取类的运行时结构

📒 getXxx()和getDeclaredXxx()

前者获取本类和父类的所有public
后者获取本类全部

📒 getXxx()参数问题

在获取getXxx()时会遇到要求给出参数的class,这是以为多态的原因。比如在获取一个方法时候，不仅仅需要知道方法名字，还需要知道方法参数，才能确定你所需要的确切的方法。

```java
//获取类的信息
public class test06 {
    public static void main(String[] args) throws ClassNotFoundException {
        Class c1 = Class.forName("cn.com.reflection.User");

        //获得类的名字
        System.out.println(c1.getName());  //包名 + 类名
        System.out.println(c1.getSimpleName()); //获得类名

        //获得类的属性
        Field[] fields = c1.getFields(); //获得public属性

        fields = c1.getDeclaredFields(); //获取全部属性

        for(Field f : fields) {
            System.out.println(f);
        }

        //获得类的方法
        Method[] methods = c1.getMethods(); //获得本类和父类的public方法

        methods = c1.getDeclaredMethods();  //获得本类的全部方法

        for(Method m : methods) {
            System.out.println(m);
        }

        //获得类的构造器
        Constructor[] constructors = c1.getConstructors(); //获取public方法

        constructors = c1.getDeclaredConstructors();    //获取全部方法

        for(Constructor c : constructors) {
            System.out.println(c);
        }
    }
}
```

8.动态创建对象执行方法

创建类的对象(两种)

newInstance()

本质上是调用了无参构造器且构造器访问权限满足

构造器创建对象

通过getDeclaredConstructor(参数.class)获取本类指定类型构造器
用构造器.newInstance(“Xxx”)传入构造参数

调用类的方法并获取返回值

通过类得到方法对象，对象中invoke方法 (invoke返回方法返回值)
class —> method —> invoke(对象，方法参数的值且可以为null)

获取类方法的返回值类型

通过类得到方法对象，对象中getReturnType方法
class —> field —> getReturnType()

获取类方法的参数类型

通过类得到方法对象，对象中getParameterTypes方法
class —> field —> getParameterTypes()

调用类的属性

通过类得到方法对象，对象中set方法
class —> field —> set(对象，方法参数的值)

获取调用类的属性

通过类得到方法对象，对象中get方法
class —> field —> get(对象)

setAccessible(boolean x)

启动和禁用访问安全检查的开关
作用：
1.提高反射效率
2.访问私有

```java
//动态的创建对象
public class test07 {
    public static void main(String[] args) throws Exception {
        //获得Class对象
        Class c1 = Class.forName("cn.com.reflection.User");

        //通过newInstance()构造一个对象
        User user = (User)c1.newInstance(); //本质上调用了无参构造器
        System.out.println(user);

        //通过构造器创建对象
        Constructor constructor = c1.getDeclaredConstructor(String.class, int.class, int.class);
        User user1 = (User)constructor.newInstance("jacky", 1, 1);
        System.out.println(user1);

        //通过反射调用普通方法
        User user3 = (User)c1.newInstance();
        //通过反射获取一个方法
        Method setName = c1.getDeclaredMethod("setName", String.class);

        //invoke：激活的意思
        //(对象, "方法的值")
        setName.invoke(user3, "jacky");
        System.out.println(user3.getName());

        //通过反射操作属性
        User user4 = (User)c1.newInstance();
        Field name = c1.getDeclaredField("name");

        //不能直接操作私有属性，需要关闭程序的安全检测，属性或方法的setAccessible(true)
        name.setAccessible(true);
        name.set(user4, "jakcy1");
        System.out.println(user4.getName());
    }
}
```

> 抽象类和接口无法被实例化，但是其中存在静态方法或非抽象方法的时候，无需实例化也可调用静态方法。
> 实际上，任何类的方法都可无需实例化调用。
> isModifiers()判断什么关键词修饰判断是否为抽象或者静态等等。

数组反射

假定一个方法，需要知道对象是否为数组并打印。
下面展示包括多维数组的代码：

```java
public void printdata(Object o) {
    Class c = o.getClass();
    if(c.isArray()) {
        int len = Array.getLength(o);
        for(int i=0; i<len; i++) {
            printdata(Array.get(o, i));
        }
    } else {
        System.out.println(o);
    }
}
```

9.获取泛型信息

Java采用泛型擦除的机制来引入泛型，Java中的泛型仅仅是给编译器Javac使用的，确保数据的安全性和免去强制类型转换的问题，但是，一旦编译完成，所有和泛型有关的类型全部擦除。

```java
public static void main(String[] args) throws NoSuchMethodException {
    Method method = test08.class.getMethod("test01", Map.class, List.class);

    Type[] genericParameterTypes = method.getGenericParameterTypes();

    for(Type genericParameterType : genericParameterTypes) {
        System.out.println("#" + genericParameterType);
        if(genericParameterType instanceof ParameterizedType) {
            Type[] actualTypeArguments = ((ParameterizedType) genericParameterType).getActualTypeArguments();
            for(Type actualTypeArgument : actualTypeArguments) {
                System.out.println(actualTypeArgument);
            }
        }
    }

    Method method2 = test08.class.getMethod("test02", null);

    Type genericReturnType = method2.getGenericReturnType();
    if(genericReturnType instanceof ParameterizedType) {
        Type[] actualTypeArguments = ((ParameterizedType) genericReturnType).getActualTypeArguments();
        for(Type actualTypeArgument : actualTypeArguments) {
            System.out.println(actualTypeArgument);
        }
    }

}

public void test01(Map<String, User> map, List<User> list) {

}

public Map<String, User> test02() {
    return null;
}
`
```

10.获取注解信息

> ORM Object Relationship Map —> 对象关系映射
> 例如
> 数据库中表与代码中对应的类的映射
> 属性与字段对应
> 对象与记录对应

```java
package cn.com.reflection;

import java.io.File;
import java.lang.annotation.*;
import java.lang.reflect.Field;

//练习反射操作注解
public class test09 {

    public static void main(String[] args) throws ClassNotFoundException, NoSuchFieldException {
        Class c1 = Class.forName("cn.com.reflection.test09.Student");

        //通过反射获得注解
        Annotation[] annotations = c1.getAnnotations();
        for (Annotation annotation : annotations) {
            System.out.println(annotation);
        }

        //获得注解value的值
        Table annotation = (Table)c1.getAnnotation(Table.class);
        String value = annotation.value();
        System.out.println(value);

        //获得类指定的注解
        Field name = c1.getDeclaredField("name");
        Field1 annotation1 = name.getAnnotation(Field1.class);
        System.out.println(annotation1.columnName());
        System.out.println(annotation1.type());
        System.out.println(annotation1.length());
    }
}

@Table("db_student")
class Student {
    @Field1(columnName = "db_id", type = "int", length = 10)
    private int id;
    @Field1(columnName = "db_age", type = "int", length = 10)
    private int age;
    @Field1(columnName = "db_name", type = "varchar", length = 3)
    private String name;

    @Override
    public String toString() {
        return "Student{" +
                "id=" + id +
                ", age=" + age +
                ", name='" + name + '\'' +
                '}';
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getName() {
```

```
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@interface Table {
    String value();
}

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@interface Field1{
    String columnName();
    String type();
    int length();
}
```

🏆 总结

Java注解与反射

标签： javase