第14章 注解和反射

#一、注解：

Java 注解（Annotation）又称 Java 标注，是 JDK5.0 引入的一种机制。Java 语言中的类、方法、变量、参数和包等都可以被标注。

#1、Annotation 的定义

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.SOURCE)
public @interface Override {
}

@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface FunctionalInterface {}
```

我们仿照jdk自带注解的方式，自己定义一个注解：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
}
```

结果发现这个注解确实可以使用了，同时我们看到了这几个注解 `@Retention` 和 `@Target` 这两个注解专门给注解加注解，我们称之为元注解。

```
@MyAnnotation
public class MessageWrapper implements Serializable {
```

再来分析，我们不妨看看那几个元注解的源码：

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Retention {
    /**
     * Returns the retention policy.
     * @return the retention policy
     */
    RetentionPolicy value();
}
```

我们发现注解中可以有方法，我们在注解中可以这样定义方法：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface MyAnnotation {
    String name() default "jerry";
    int age();
}
```

我们在使用的时候就可以这样使用了：

```
@MyAnnotation(age = 15)
public class MessageWrapper implements Serializable {
```

关于注解方法，有以下几点注意的：

1、定义的格式是：String name();

2、可以有默认值，也可以没有，如果没有默认值在使用的时候必须填写对应的值。默认值使用default添加。

3、如果想在使用的时候不指定具体的名字，方法名字定义为value() 即可。

```
public @interface MyAnnotation {
    String name() default "jerry";
    int value();
}
```

```
@MyAnnotation(15)
public class MessageWrapper implements Serializable {
```

看到这里，有人可能会问，注解到底有什么用？如果我们没有学习反射，对我们而言，注解确实没什么用，哈哈哈，后边我们结合反射会有例子讲解。

#2、Annotation 组成部分

我们使用javap查看生成的注解类：

```
PS D:\code\test\out\production\test\com\ydlclass\chat> javap -v .\MyAnnotation.class
Classfile /D:/code/test/out/production/test/com/ydlclass/chat/MyAnnotation.class
  Last modified 2021-9-12; size 482 bytes
  MD5 checksum 5b096a2faeef11535277c9cdbe5703d0
  Compiled from "MyAnnotation.java"
  //我们发现字节码中注解其实也是一个接口统一继承自java.lang.annotation.Annotation
public interface com.ydlclass.chat.MyAnnotation extends java.lang.annotation.Annotation
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_INTERFACE, ACC_ABSTRACT, ACC_ANNOTATION
Constant pool:
{
    // 生成了两个抽象方法
  public abstract java.lang.String name();
    descriptor: ()Ljava/lang/String;
    flags: ACC_PUBLIC, ACC_ABSTRACT
    AnnotationDefault:
```

```
        default_value: s#7
    public abstract int value();
      descriptor: ()I
      flags: ACC_PUBLIC, ACC_ABSTRACT
}
SourceFile: "MyAnnotation.java"
RuntimeVisibleAnnotations:
  0: #13(#8=[e#14.#15])
  1: #16(#8=e#17.#18)
PS D:\code\test\out\production\test\com\ydlclass\chat>
```

java Annotation 的组成中，有 3 个非常重要的主干类。它们分别是：

（1）Annotation.java

```
package java.lang.annotation;
public interface Annotation {

    boolean equals(Object obj);

    int hashCode();

    String toString();

    Class<? extends Annotation> annotationType();
}
```

（2）ElementType.java

ElementType 是 Enum 枚举类型，它用来指定 Annotation 的类型。大白话就是，说明了我的注解将来要放在哪里。

```
package java.lang.annotation;

public enum ElementType {
    // 类、接口（包括注释类型）或枚举声明
    TYPE,
    //  字段声明（包括枚举常量
    FIELD,
    //  方法声明
    METHOD,
    //  参数声明
    PARAMETER,
    //  构造方法声明
    CONSTRUCTOR,
    //  局部变量声明
    LOCAL_VARIABLE,
    //   注释类型声明
    ANNOTATION_TYPE,
    //  包声明
    PACKAGE
}
```

（3）RetentionPolicy.java

RetentionPolicy 是 Enum 枚举类型，它用来指定 Annotation 的策略。通俗点说，就是不同 RetentionPolicy 类型的 Annotation 的作用域不同。

1. 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。
2. 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。
3. 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 {
    //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
    SOURCE,
    //编译器将Annotation存储于类对应的.class文件中。但不会加载到JVM中。默认行为
    CLASS,
    // 编译器将Annotation存储于class文件中，并且可由JVM读入，因此运行时我们可以获取。
    RUNTIME
}
```

#3、Java 自带的 Annotation

理解了上面的 3 个类的作用之后，我们接下来可以讲解 Annotation 实现类的语法定义了。

（1）内置的注解

Java 定义了一套注解，共有10 个，6个在 java.lang 中，剩下 4 个在 java.lang.annotation 中。

（1）作用在代码的注解是

- @Override - 检查该方法是否是重写方法。如果发现其父类，或者是引用的接口中并没有该方法时，会报编译错误。
- @Deprecated - 标记过时方法。如果使用该方法，会报编译警告。
- @SuppressWarnings - 指示编译器去忽略注解中声明的警告。
- @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 - Java 8 开始支持，标识一个匿名函数或函数式接口。
- @Repeatable - Java 8 开始支持，标识某注解可以在同一个声明上使用多次。

（2）作用在其他注解的注解(或者说 元注解)是:

- @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。
- @Target - 标记这个注解可以修饰哪些 Java 成员。
- @Inherited - 如果一个类用上了@Inherited修饰的注解，那么其子类也会继承这个注解

（2）常用注解

通过上面的示例，我们能理解：@interface 用来声明 Annotation，@Documented 用来表示该 Annotation 是否会出现在 javadoc 中， @Target 用来指定 Annotation 的类型，@Retention 用来指定 Annotation 的策略。

> @Documented 标记这些注解是否包含在用户文档中。

> @Inherited

@Inherited 的定义如下：加有该注解的注解会被子类继承，注意，仅针对**类，成员属性**、方法并不受此注释的影响。

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Inherited {
}
```

> @Deprecated

@Deprecated 的定义如下：

```
@Documented
@Retention(RetentionPolicy.RUNTIME)
public @interface Deprecated {
}
```

说明：

@Deprecated 所标注内容，不再被建议使用。

```
public class Dog extends Animal{
    @Override
    @Deprecated
    public void eat() { System.out.println("dog is eating!"); }
```

加上这个注解在使用或者重写时会有警告：

> @SuppressWarnings

@SuppressWarnings 的定义如下：

```
@Target({TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
@Retention(RetentionPolicy.SOURCE)
public @interface SuppressWarnings {
    String[] value();
}
```

说明：

SuppressWarnings 的作用是，让编译器对"它所标注的内容"的某些警告保持静默，用于抑制编译器产生警告信息。。例如，"@SuppressWarnings(value={"deprecation", "unchecked"})" 表示对"它所标注的内容"中的 "SuppressWarnings 不再建议使用警告"和"未检查的转换时的警告"保持沉默。

不用记，谁记谁傻X。

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
fallthrough	抑制在switch中缺失breaks的警告
finally	抑制finally模块没有返回的警告
rawtypes	使用generics时忽略没有指定相应的类型
serial	忽略在serializable类中没有声明serialVersionUID变量
unchecked	抑制没有进行类型检查操作的警告
unused	抑制没被使用过的代码的警告

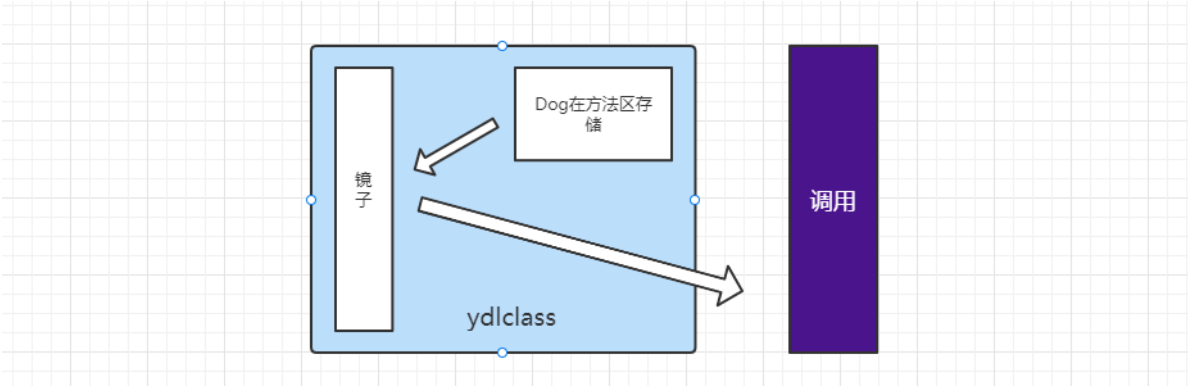
#4、Annotation 的作用

（1）Annotation 具有"让编译器进行编译检查的作用“，这个讲了很多了。

（2）利用反射，和反射配合使用能产生奇妙的化学反应。

#二、反射

我们都知道光是可以反射的，我们无法直接接触方法区中一个类的方法、属性、注解等，那就可以通过一面镜子观察它的全貌，这个镜子就是JDK给我们提供的Class类。

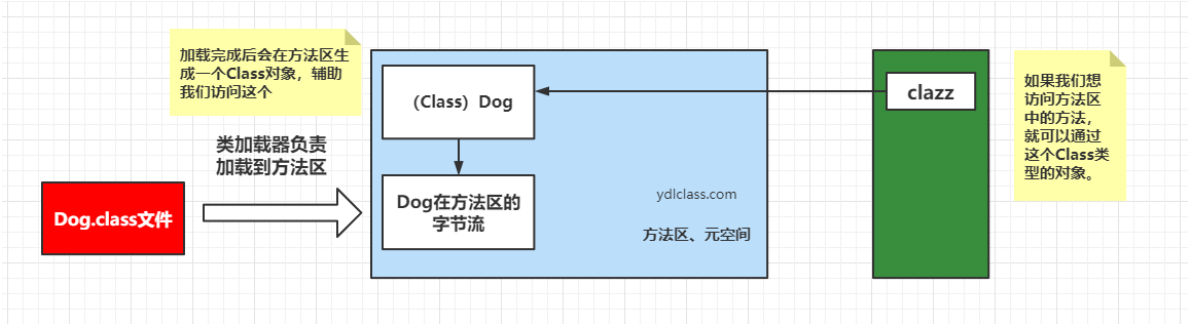


首先我们看一下Class这个类，初步简单的分析一下。我们发现这个类并没有什么成员变量，仅仅存在许多的方法，还有不少是本地方法。通过这些方法的名字我们大致能猜出，这个类能帮我们获取方法、构造器、属性、注解等。

```
public final class Class<T>  {

    // 获得他实现的接口
    public Class<?>[] getInterfaces() {
        ReflectionData<T> rd = reflectionData();
        if (rd == null) {
            // no cloning required
            return getInterfaces0();
        } else {
            Class<?>[] interfaces = rd.interfaces;
            if (interfaces == null) {
                interfaces = getInterfaces0();
                rd.interfaces = interfaces;
            }
            // defensively copy before handing over to user code
            return interfaces.clone();
        }
    }

    private native Class<?>[] getInterfaces0();

    // 获得方法
    @CallerSensitive
    public Method[] getMethods() throws SecurityException {
        checkMemberAccess(Member.PUBLIC, Reflection.getCallerClass(), true);
        return copyMethods(privateGetPublicMethods());
    }

    // 获得他的构造器
    @CallerSensitive
    public Constructor<?>[] getConstructors() throws SecurityException {
        checkMemberAccess(Member.PUBLIC, Reflection.getCallerClass(), true);
        return copyConstructors(privateGetDeclaredConstructors(true));
    }

    // 获得他的属性
    @CallerSensitive
    public Field getField(String name)
        throws NoSuchFieldException, SecurityException {
        checkMemberAccess(Member.PUBLIC, Reflection.getCallerClass(), true);
        Field field = getField0(name);
        if (field == null) {
```

```
            throw new NoSuchFieldException(name);
        }
        return field;
    }

}
```

我们已经学过了类的加载过程，这里我们要介绍的是，每一个类加载完成后会在方法区生成一个Class类型的对象，辅助我们访问这个的方法、构造器、字段等。这个对象是Class的子类，每个类【有且仅有】一个Class类，也叫类对象。



#1、获取类对象的方法

（1）获取方式

```
1、使用类
Class clazz = Dog.class;

2、使用全类名
Class aClass = Class.forName("com.ydl.Dog");

3、使用对象
Dog dog = new Dog();
Class clazz = dog.getClass();
```

（2）对类对象操作

```
//获取类名字
String name = clazz.getName();
//获取类加载器
ClassLoader classLoader = clazz.getClassLoader();
//获取资源
URL resource = clazz.getResource("");
//得到父类
Class superclass = clazz.getSuperclass();
//判断一个类是不是接口，数组等等
boolean array = clazz.isArray();
boolean anInterface = clazz.isInterface();

//重点，使用class对象实例化一个对象
Object instance = clazz.newInstance();
```

#2、对成员变量的操作

在java中万物皆对象成员变量也是对象，他拥有操作一个对象的成员变量的能力。

（1）获取成员变量

getFields只能获取被public修饰的成员变量，当然反射很牛，我们依然可以使用getDeclaredFields方法获取所有的成员变量。

```
//获取字段，只能获取公共的字段（public）
Field name = clazz.getField("type");
Field[] fields = clazz.getFields();
//能获取所有的字段包括private
Field color = clazz.getDeclaredField("color");
Field[] fields = clazz.getDeclaredFields();

System.out.println(color.getType());
```

（2）获取对象的属性

```
Dog dog = new Dog();
dog.setColor("red");
Class clazz = Dog.class;
Field color = clazz.getDeclaredField("color");
System.out.println(color.get(dog));
```

当然你要是明确类型你还能用以下方法：

```
Int i = age.getInt(dog);
xxx.getDouble(dog);
xxx.getFloat(dog);
xxx.getBoolean(dog);
xxx.getChar(dog);
//每一种基本类型都有对应方法
```

（3）设置对象的属性

```
Dog dog = new Dog();
dog.setColor("red");
Class clazz = Dog.class;
Field color = clazz.getDeclaredField("color");
color.set(dog,"blue");
System.out.println(dog.getColor());
```

当然如果你知道对应的类型，我们可以这样：

```
xxx.setBoolean(dog,true);
xxx.getDouble(dog，1.2);
xxx.getFloat(dog,1.2F);
xxx.getChar(dog,'A');
//每一种基本类型包装类都有对应方法
```

```
Field color = dogClass.getDeclaredField("color");
//暴力注入
color.setAccessible(true);
color.set(dog,"red");
```

#3、对方法的操作

（1）获取方法

```
//根据名字和参数类型获取一个方法
Method method = clazz.getMethod("eat",String.class);
Method[] methods = clazz.getMethods();

Method eat = clazz.getDeclaredMethod("eat", String.class);
Method[] declaredMethods = clazz.getDeclaredMethods();
```

（2）对方法的操作

```
Dog dog = new Dog();
dog.setColor("red");
Class clazz = Dog.class;
//获取某个方法，名字，后边是参数类型
Method method = clazz.getMethod("eat",String.class);
//拿到参数的个数
int parameterCount = method.getParameterCount();
//拿到方法的名字
String name = method.getName();
//拿到参数的类型数组
Class<?>[] parameterTypes = method.getParameterTypes();
//拿到返回值类型
Class<?> returnType = method.getReturnType();
//重点。反射调用方法，传一个实例，和参数
method.invoke(dog,"热狗");
```

```
Class dogClass = Class.forName("com.xinzhi.Dog");
Object dog = dogClass.newInstance();

Method eat = dogClass.getMethod("eat");
eat.invoke(dog);

Method eat2 = dogClass.getMethod("eat",String.class);
eat2.invoke(dog,"meat");

Method eat3 = dogClass.getMethod("eat",String.class,int.class);
eat3.invoke(dog,"meat",12);
```

#4、对构造器的操作

（1）获取并构建对象

```
Constructor[] constructors = clazz.getConstructors();
Constructor constructor = clazz.getConstructor();
Constructor[] declaredConstructors = clazz.getDeclaredConstructors();
Constructor declaredConstructor = clazz.getDeclaredConstructor();

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

#5、对注解的操作

（1）从方法、字段、类上获取注解

```
//元注解 要加上runtime
//类上
Annotation annotation = clazz.getAnnotation(Bean.class);
Annotation[] annotations = clazz.getAnnotations();

//字段上
Annotation annotation = field.getAnnotation(Bean.class);
Annotation[] annotations = field.getAnnotations();

//方法上
Annotation annotation = method.getAnnotation(Bean.class);
Annotation[] annotations = method.getAnnotations();
```

#三、写一个小案例

要求：讲src源文件中加了 `@Singleton` 注解的类都在程序启动时以【单例】的形式加载到内存。

提示：

1、获取classpath文件的方法：

```
URL resource = Thread.currentThread().getContextClassLoader().getResource("");
String file = resource.getFile();
```

2、所有的单例放在一个ConcurrentHashMap当中：

```
public class  ApplicationContext {
    private final ConcurrentHashMap<Class<?>,Object> context = new
ConcurrentHashMap<>();

    public void registerSingleton(Class<?> clazz,Object t){
        context.put(clazz, t);
    }

    @SuppressWarnings("unchecked")
    public <T> T getSingleton(Class<T> clazz){
        return  (T)context.get(clazz);
    }
}
```

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Singleton {
}
```

```
public class SingletonHandler {

    public static void handler(List<String> classNames){
        for (String className : classNames) {
            Class<?> clazz = null;
            try {
                clazz = Class.forName(className);
```

```
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            }
            // 获取注解
            Singleton annotation = clazz.getAnnotation(Singleton.class);
            if(annotation != null){
                Object instance = null;
                try {
                    instance = clazz.newInstance();
                } catch (InstantiationException e) {
                    e.printStackTrace();
                } catch (IllegalAccessException e) {
                    e.printStackTrace();
                }
                ApplicationContext.addSingleton(clazz,instance);
            }
        }
    }
}public class ApplicationContext {
    // 维护一个上下文环境
    private final static Map<Class<?>,Object> CONTEXT = new ConcurrentHashMap<>
(8);

    public static void addSingleton(Class<?> clazz,Object entity){
        ApplicationContext.CONTEXT.put(clazz,entity);
    }

    // 把实例对象从容器拿出来
    public static  <T> T getSingleton(Class<T> clazz){
        return (T)ApplicationContext.CONTEXT.get(clazz);
    }
}
```

```
public class FileUtils {

    public static List<String> getAllClassName(File file) {
        // 1、自己定义一个集合
        List<String> classPaths = new ArrayList<>();

        // 2、获取所有的class文件的路径
        findAll(file.getAbsolutePath(),classPaths);
        //
D:\code\javase\out\production\annotationAndReflect\com\ydlclass\Dog.class
        // com.ydlclass.Dog   Class.forName()
        // 遍历绝对路径，变成全限定名
        return classPaths.stream().map(path -> {
            String fileName = file.getAbsolutePath();
            //
D:\code\javase\out\production\annotationAndReflect\com\ydlclass\Dog.class
            // com\ydlclass\Dog.class
            return path.replace(fileName + "\\", "")
                    .replaceAll("\\\\",".")
                    .replace(".class","");
        }).collect(Collectors.toList());
    }


```

```
23
24      private static void findAll(String path,List<String> classPathList) {
25          // 1、尝试列出当前文件夹的文件
26          File file = new File(path);
27          // 2、过滤文件  文件夹和png
28          File[] list = file.listFiles((f, n) -> new File(f, n).isDirectory() ||
    n.contains(".class"));
29
30          if (list == null || list.length == 0) {
31              return;
32          }
33          for (File parent : list) {
34              // 看看是不是一个文件夹，如果是
35              if (parent.isDirectory()) {
36                  // 递归
37                  findAll(parent.getAbsolutePath(),classPathList);
38              } else {
39                  // 如果不是
40                  classPathList.add(parent.getAbsolutePath());
41              }
42          }
43      }
44  }
```

```
1   public class Bootstrap {
2       // 类加载之后就会处理
3       static {
4           // 获取classpath根路径
5           final URL resource =
    Thread.currentThread().getContextClassLoader().getResource("");
6           // 一句话获取权限名称
7           List<String> classNames = FileUtils.getAllClassName(new
    File(resource.getFile()));
8           // 处理对应的全限定名称
9           SingletonHandler.handler(classNames);
10      }
11
12      public static void main(String[] args) {
13          Dog singleton = ApplicationContext.getSingleton(Dog.class);
14          System.out.println(singleton);
15      }
16
17  }
```