**Java**语言高级特性**——**注解与反射

Java知识是作为Android开发的语言基础，虽然现在已经推出了kotlin，但是基于以下原因我们还是需要好好牢牢掌 握java：

1. SDK还是java，kotlin也需要编译成为java运行；
2. Java语言应用不仅仅在Android，就是在后台开发中也是一个最流行的语言；
3. 大公司面试都要求我们有扎实的Java语言基础。

所以，请大家不要轻视提高自己Java基础的机会，请大家认真学习，做好笔记，争取取得更大的进步。

# 注解

Java 注解（Annotation）又称 Java 标注，是 JDK5.0 引入的一种注释机制。 注解是元数据的一种形式，提供有关于程序但不属于程序本身的数据。注解对它们注解的代码的操作没有直接影响。

## 注解声明

#### 声明一个注解类型

Java中所有的注解，默认实现

Annotation

接口：

package java.lang.annotation; public interface Annotation {

boolean equals(Object obj); int hashCode();

String toString();

Class<? extends Annotation> annotationType();

}

与声明一个"**Class**"不同的是，注解的声明使用 关键字。一个注解的声明如下：

@interface

public @interface Lance{

}

#### 元注解

在定义注解时，注解类也能够使用其他的注解声明。对注解类型进行注解的注解类，我们称之为 meta- annotation（元注解）。一般的，我们在定义自定义注解时，需要指定的元注解有两个 ：

另外还有**@Documented** 与 **@Inherited** 元注解，前者用于被javadoc工具提取成文档，后者表示允许子类继承父类中定义的注解。

##### @Target

注解标记另一个注解，以限制可以应用注解的 Java 元素类型。目标注解指定以下元素类型之一作为其值：

可以应用于注解类型。可以应用于构造函数。

ElementType.CONSTRUCTOR

ElementType.ANNOTATION\_TYPE

ElementType.FIELD 可以应用于字段或属性。

可以应用于局部变量。ElementType.METHOD 可以应用于方法级注解。 ElementType.PACKAGE 可以应用于包声明。 ElementType.PARAMETER 可以应用于方法的参数。 ElementType.TYPE 可以应用于类的任何元素。

ElementType.LOCAL\_VARIABLE

##### @Retention

注解指定标记注解的存储方式：

- 标记的注解仅保留在源级别中，并被编译器忽略。

RetentionPolicy.SOURCE

RetentionPolicy.CLASS - 标记的注解在编译时由编译器保留，但 Java 虚拟机( JVM)会忽略。

- 标记的注解由 JVM 保留，因此运行时环境可以使用它。

RetentionPolicy.RUNTIME

@Retention 三个值中 SOURCE < CLASS < RUNTIME，即CLASS包含了SOURCE，RUNTIME包含SOURCE、CLASS。下文会介绍他们不同的应用场景。

下面来看例子：

//@Target(ElementType.TYPE) 只能在类上标记该注解 @Target({ElementType.TYPE,ElementType.FIELD}) // 允许在类与类属性上标记该注解 @Retention(RetentionPolicy.SOURCE) //注解保留在源码中

public @interface Lance {

}

#### 注解类型元素

在上文元注解中，允许在使用注解时传递参数。我们也能让自定义注解的主体包含 *annotation type element (*注解类型元素*)* 声明，它们看起来很像方法，可以定义可选的默认值。

@Target({ElementType.TYPE,ElementType.FIELD}) @Retention(RetentionPolicy.SOURCE)

public @interface Lance {

String value(); //无默认值

int age() default 1; //有默认值

}

注意：在使用注解时，如果定义的注解中的类型元素无默认值，则必须进行传值。

SOURCE

@ Lance ("帅") //如果只存在value元素需要传值的情况，则可以省略:元素名= @Andy(value="帅",age = 2)

int i;

## 注解应用场景

按照**@Retention** 元注解定义的注解存储方式，注解可以被在三种场景下使用：

### SOURCE

，作用于源码级别的注解，可提供给IDE语法检查、APT等场景使用。

RetentionPolicy.SOURCE

在类中使用 级别的注解，其编译之后的class中会被丢弃。



**IDE**语法检查

在Android开发中， 与

support-annotations

androidx.annotation)

下：

中均有提供

注解，此注解的定义如

@IntDef

@Retention(SOURCE) //源码级别注解 @Target({ANNOTATION\_TYPE})

public @interface IntDef { int[] value() default {};

boolean flag() default false;

boolean open() default false;

}

Java中Enum(枚举)的实质是特殊单例的静态成员变量，在运行期所有枚举类作为单例，全部加载到内存中。 比常量多5到10倍的内存占用。

此注解的意义在于能够取代枚举，实现如方法入参限制。

如：我们定义方法为:

test

，此方法接收参数

需要在：**Lance**、**Alvin**中选择一个。如果使用枚举能够实现

public enum Teacher{ LANCE,ALVIN

}

public void test(Teacher teacher) {

}

而现在为了进行内存优化，我们现在不再使用枚举，则方法定义为：

public static final int LANCE = 1; public static final int ALVIN = 2;

public void test(int teacher) {

}

然而此时，调用 方法由于采用基本数据类型int，将无法进行类型限定。此时使用@IntDef增加自定义注解：

test

public static final int LANCE = 1; public static final int ALVIN = 2;

@IntDef(value = {MAN, WOMEN}) //限定为LANCE，ALVIN @Target(ElementType.PARAMETER) //作用于参数的注解 @Retention(RetentionPolicy.SOURCE) //源码级别注解 public @interface Teacher {

}

public void test(@Teacher int teacher) {

}

此时，我们再去调用错)。

test

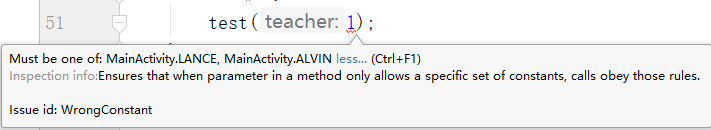
方法，如果传递的参数不是 或者

则会显示 **Inspection** 警告(编译不会报

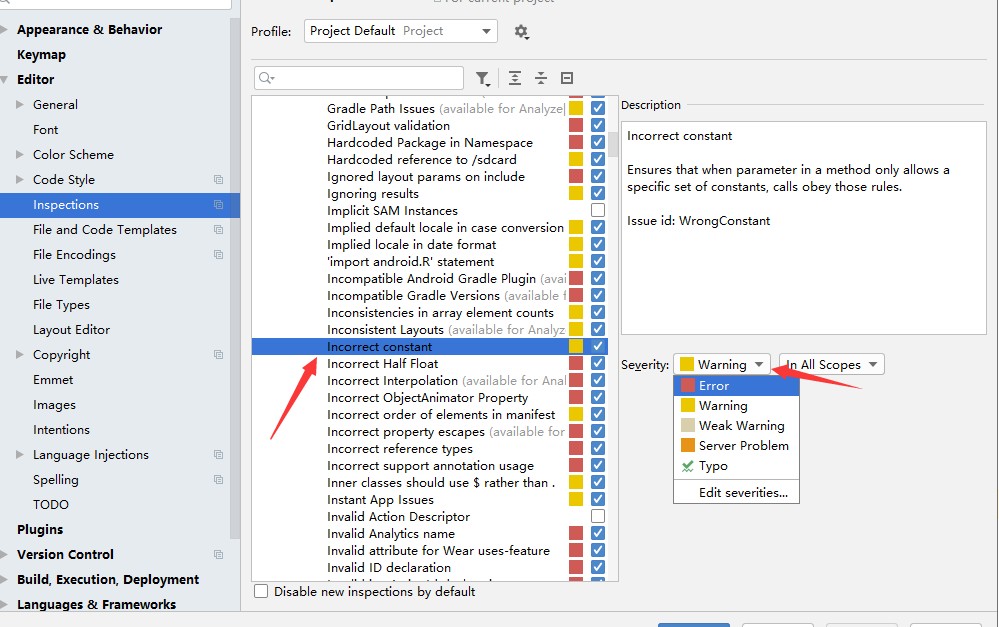
teacher

LANCE

ALVIN



可以修改此类语法检查级别：



以上注解均为 级别，本身IDEA/AS 就是由Java开发的，工具实现了对Java语法的检查，借助注解能对被注

SOURCE

解的特定语法进行额外检查。

**APT**注解处理器

APT全称为："Anotation Processor Tools"，意为注解处理器。顾名思义，其用于处理注解。编写好的Java源文件，需要经过 javac 的编译，翻译为虚拟机能够加载解析的字节码Class文件。注解处理器是 javac 自带的一个工具，用来在编译时期扫描处理注解信息。你可以为某些注解注册自己的注解处理器。 注册的注解处理器由

javac

调起，并将注解信息传递给注解处理器进行处理。

注解处理器是对注解应用最为广泛的场景。在Glide、EventBus3、Butterknifer、Tinker、ARouter等等常用

SOURCE

框架中都有注解处理器的身影。但是你可能会发现，这些框架中对注解的定义并不是 级别，更多的

CLASS

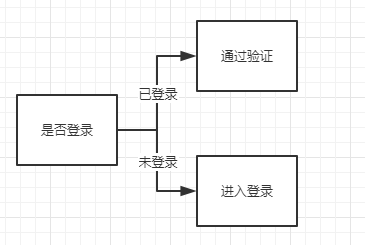
是 级别，别忘了：**CLASS**包含了**SOURCE**，**RUNTIME**包含**SOURCE**、**CLASS**。

关于注解处理器的实现，在后续课程中会有相当多的介绍。此处先不进行详细介绍。

### CLASS

定义为 CLASS 的注解，会保留在class文件中，但是会被虚拟机忽略(即无法在运行期反射获取注解)。此时完全符合此种注解的应用场景为字节码操作。如：AspectJ、热修复Roubust中应用此场景。

所谓字节码操作即为，直接修改字节码Class文件以达到修改代码执行逻辑的目的。在程序中有多处需要进行是否 登录的判断。



如果我们使用普通的编程方式，需要在代码中进行 的判断，也许存在十个判断点，则需要在每个判断点加

if-else

需要登录

无需登录

入此项判断。此时，我们可以借助AOP(面向切面)编程思想，将程序中所有功能点划分为： 与两种类型，即两个切面。对于切面的区分即可采用注解。

//Java源码 @Target(ElementType.METHOD) @Retention(RetentionPolicy.CLASS) public @interface Login {

}

@Login

public void jumpA(){

startActivity(new Intent(this,AActivity.class));

}

public void jumpB(){

startActivity(new Intent(this,BActivity.class));

}

在上诉代码中， jumpA 方法需要具备登录身份。而 注解的定义被设置为 CLASS 。因此我们能够在该类所编

Login

译的字节码中获得到方法注解 Login 。在操作字节码时，就能够根据方法是否具备该注解来修改class中该方法的

if-else

内容加入 的代码段：

//Class字节码

@Login

public void jumpA() { if (this.isLogin) {

this.startActivity(new Intent(this, LoginActivity.class));

} else {

this.startActivity(new Intent(this, AActivity.class));

}

}

public void jumpB() {

startActivity(new Intent(this,BActivity.class));

}

注解能够设置类型元素(参数)，结合参数能实现更为丰富的场景，如：运行期权限判定等。

### RUNTIME

注解保留至运行期，意味着我们能够在运行期间结合反射技术获取注解中的所有信息。

# 反射

一般情况下，我们使用某个类时必定知道它是什么类，是用来做什么的，并且能够获得此类的引用。于是我们直接 对这个类进行实例化，之后使用这个类对象进行操作。

反射则是一开始并不知道我要初始化的类对象是什么，自然也无法使用 new 关键字来创建对象了。这时候，我们使用 JDK 提供的反射 API 进行反射调用。反射就是在运行状态中**,**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方法和属性**;**并且能改变它的属性。是Java被视为动态语言的关键。

Java反射机制主要提供了以下功能：

在运行时构造任意一个类的对象

在运行时获取或者修改任意一个类所具有的成员变量和方法在运行时调用任意一个对象的方法（属性）

**Class**

反射始于Class，**Class**是一个类，封装了当前对象所对应的类的信息。一个类中有属性，方法，构造器等，比如说 有一个Person类，一个Order类，一个Book类，这些都是不同的类，现在需要一个类，用来描述类，这就是Class，它应该有类名，属性，方法，构造器等。Class是用来描述类的类。

Class类是一个对象照镜子的结果，对象可以看到自己有哪些属性，方法，构造器，实现了哪些接口等等。对于每 个类而言，JRE 都为其保留一个不变的 Class 类型的对象。一个 Class 对象包含了特定某个类的有关信息。 对象只能由系统建立对象，一个类（而不是一个对象）在 JVM 中只会有一个Class实例。

获得 **Class** 对象

获取Class对象的三种方式

|  |  |  |
| --- | --- | --- |
| 1. 通过类名获取 | 类名.class |  |
| 2. 通过对象获取 | 对象名.getClass() |  |
| 3. 通过全类名获取 | Class.forName(全类名) | classLoader.loadClass(全类名) |

使用 Class 类的 静态方法

forName

public static Class<?> forName(String className)

直接获取某一个对象的 class

Class<?> klass = int.class; Class<?> classInt = Integer.TYPE;

调用某个对象的 方法

getClass()

StringBuilder str = new StringBuilder("123"); Class<?> klass = str.getClass();

## 判断是否为某个类的实例

一般地，我们用 关键字来判断是否为某个类的实例。同时我们也可以借助反射中 Class 对象的

instanceof

isInstance()

方法来判断是否为某个类的实例，它是一个 native 方法：

public native boolean isInstance(Object obj);

判断是否为某个类的类型

public boolean isAssignableFrom(Class<?> cls)

## 创建实例

通过反射来生成对象主要有两种方式。

使用Class对象的newInstance()方法来创建Class对象对应类的实例。

Class<?> c = String.class; Object str = c.newInstance();

先通过Class对象获取指定的Constructor对象，再调用Constructor对象的newInstance()方法来创建实例。这 种方法可以用指定的构造器构造类的实例。

//获取String所对应的Class对象 Class<?> c = String.class;

//获取String类带一个String参数的构造器

Constructor constructor = c.getConstructor(String.class);

//根据构造器创建实例

Object obj = constructor.newInstance("23333"); System.out.println(obj);

## 获取构造器信息

得到构造器的方法

Constructor getConstructor(Class[] params) -- 获得使用特殊的参数类型的public构造函数(包括父类）

Constructor[] getConstructors() -- 获得类的所有公共构造函数

Constructor getDeclaredConstructor(Class[] params) -- 获得使用特定参数类型的构造函数(包括私有) Constructor[] getDeclaredConstructors() -- 获得类的所有构造函数(与接入级别无关)

获取类构造器的用法与上述获取方法的用法类似。主要是通过Class类的getConstructor方法得到Constructor类的 一个实例，而Constructor类有一个newInstance方法可以创建一个对象实例:

public T newInstance(Object ... initargs)

## 获取类的成员变量（字段）信息

获得字段信息的方法

Field getField(String name) -- 获得命名的公共字段

Field[] getFields() -- 获得类的所有公共字段

Field getDeclaredField(String name) -- 获得类声明的命名的字段

Field[] getDeclaredFields() -- 获得类声明的所有字段

## 调用方法

获得方法信息的方法

Method getMethod(String name, Class[] params) -- 使用特定的参数类型，获得命名的公共方法

Method[] getMethods() -- 获得类的所有公共方法

Method getDeclaredMethod(String name, Class[] params) -- 使用特写的参数类型，获得类声明的命名的方法

Method[] getDeclaredMethods() -- 获得类声明的所有方法

当我们从类中获取了一个方法后，我们就可以用

invoke()

方法来调用这个方法。

方法的原型为:

public Object invoke(Object obj, Object... args)

## 利用反射创建数组

invoke

数组在Java里是比较特殊的一种类型，它可以赋值给一个Object Reference 其中的Array类为

java.lang.reﬂect.Array类。我们通过Array.newInstance()创建数组对象，它的原型是:

public static Object newInstance(Class<?> componentType, int length);

## 反射获取泛型真实类型

当我们对一个泛型类进行反射时，需要的到泛型中的真实数据类型，来完成如json反序列化的操作。此时需要通

过 体系来完成。

Type

Type

TypeVariable

接口包含了一个实现类(Class)和四个实现接口，他们分别是：

泛型类型变量。可以泛型上下限等信息；

ParameterizedType

具体的泛型类型，可以获得元数据中泛型签名类型(泛型真实类型)

GenericArrayType

当需要描述的类型是泛型类的数组时，比如List[],Map[]，此接口会作为Type的实现。

WildcardType

通配符泛型，获得上下限信息；

### TypeVariable

public class TestType <K extends Comparable & Serializable, V> {

K key; V value;

public static void main(String[] args) throws Exception {

// 获取字段的类型

Field fk = TestType.class.getDeclaredField("key"); Field fv = TestType.class.getDeclaredField("value");

TypeVariable keyType = (TypeVariable)fk.getGenericType(); TypeVariable valueType = (TypeVariable)fv.getGenericType();

// getName 方法

System.out.println(keyType.getName());

System.out.println(valueType.getName());

// getGenericDeclaration 方法

System.out.println(keyType.getGenericDeclaration());

// K

// V

// class com.test.TestType

System.out.println(valueType.getGenericDeclaration()); // class com.test.TestType

// getBounds 方法

System.out.println("K 的上界:");

for (Type type : keyType.getBounds()) { System.out.println(type);

}

System.out.println("V 的上界:");

for (Type type : valueType.getBounds()) { System.out.println(type);

}

// 有两个

// interface java.lang.Comparable

// interface java.io.Serializable

// 没明确声明上界的, 默认上界是 Object

// class java.lang.Object

}

}

**ParameterizedType**

public class TestType { Map<String, String> map;

public static void main(String[] args) throws Exception { Field f = TestType.class.getDeclaredField("map");

System.out.println(f.getGenericType()); // java.util.Map<java.lang.String, java.lang.String>

ParameterizedType pType = (ParameterizedType) f.getGenericType(); System.out.println(pType.getRawType()); // interface java.util.Map for (Type type : pType.getActualTypeArguments()) {

System.out.println(type); // 打印两遍: class java.lang.String

}

}

}

**GenericArrayType**

public class TestType<T> { List<String>[] lists;

public static void main(String[] args) throws Exception { Field f = TestType.class.getDeclaredField("lists");

GenericArrayType genericType = (GenericArrayType) f.getGenericType(); System.out.println(genericType.getGenericComponentType());

}

}

**WildcardType**

public class TestType {

private List<? extends Number> a; // 上限

private List<? super String> b; //下限

public static void main(String[] args) throws Exception { Field fieldA = TestType.class.getDeclaredField("a"); Field fieldB = TestType.class.getDeclaredField("b");

// 先拿到范型类型

ParameterizedType pTypeA = (ParameterizedType) fieldA.getGenericType(); ParameterizedType pTypeB = (ParameterizedType) fieldB.getGenericType();

// 再从范型里拿到通配符类型

WildcardType wTypeA = (WildcardType) pTypeA.getActualTypeArguments()[0]; WildcardType wTypeB = (WildcardType) pTypeB.getActualTypeArguments()[0];

// 方法测试

System.out.println(wTypeA.getUpperBounds()[0]); // class java.lang.Number System.out.println(wTypeB.getLowerBounds()[0]); // class java.lang.String

// 看看通配符类型到底是什么, 打印结果为: ? extends java.lang.Number

System.out.println(wTypeA);

}

}

**Gson**反序列化

static class Response<T> {

T data; int code;

String message;

@Override

public String toString() { return "Response{" +

"data=" + data + ", code=" + code +

", message='" + message + '\'' + '}';

}

public Response(T data, int code, String message) {

this.data = data; this.code = code; this.message = message;

}

}

static class Data { String result;

public Data(String result) { this.result = result;

}

@Override

public String toString() { return "Data{" +

"result=" + result + '}';

}

}

public static void main(String[] args) {

Response<Data> dataResponse = new Response(new Data("数据"), 1, "成功");

Gson gson = new Gson();

String json = gson.toJson(dataResponse); System.out.println(json);

//为什么TypeToken要定义为抽象类？

Response<Data> resp = gson.fromJson(json, new TypeToken<Response<Data>>() {

}.getType()); System.out.println(resp.data.result);

}

在进行GSON反序列化时，存在泛型时，可以借助要被定义为抽象类呢？

TypeToken

TypeToken

获取Type以完成泛型的反序列化。但是为什么

因为只有定义为抽象类或者接口，这样在使用时，需要创建对应的实现类，此时确定泛型类型，编译才能够将泛型

signature信息记录到Class元数据中。