# 反射

JAVA反射机制是在**运行状态**中，对于任意一个实体类，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；这种动态获取信息以及动态调用对象方法的功能称为java语言的反射机制。

框架：半成品项目。我们可以再框架的基础上进行软件开发，简化编码。

反射：框架的基础，也是框架的灵魂。将类的各个组成部分封装成其他的对象。

反射的好处：可以在程序运行过程中，操作这些对象。可以提高程序扩展性和复用性，可以解耦。

## Class

**[Class](https://developer.android.google.cn/reference/java/lang/Class" \t "_blank)代表类的实体**，在运行的Java应用程序中表示类和接口。在这个类中提供了很多有用的方法，这里对他们简单的分类介绍。

* **获得类相关的方法**

| **方法** | **用途** |
| --- | --- |
| asSubclass(Class<U> clazz) | 把传递的类的对象转换成代表其子类的对象 |
| getClassLoader() | 获得类的加载器 |
| getClasses() | 返回一个数组，数组中包含该类中所有公共类和接口类的对象 |
| getDeclaredClasses() | 返回一个数组，数组中包含该类中所有类和接口类的对象 |
| forName(String className) | 根据类名返回类的对象 |
| getName() | 获得类的完整路径名字 |
| newInstance() | 创建类的实例 |
| getPackage() | 获得类的包 |
| getSimpleName() | 获得类的名字 |
| getSuperclass() | 获得当前类继承的父类的名字 |
| getInterfaces() | 获得当前类实现的类或是接口 |
| .class | 获取当前对象的类 |

* **获得类中字段相关的方法**

| **方法** | **用途** |
| --- | --- |
| getField(String name) | 获得某个公有的字段对象 |
| getFields() | 获得所有公有的字段对象 |
| getDeclaredField(String name) | 获得某个字段对象 |
| getDeclaredFields() | 获得所有字段对象 |

* **获得类中注解相关的方法**

| **方法** | **用途** |
| --- | --- |
| getAnnotation(Class<A> annotationClass) | 返回该类中与参数类型匹配的公有注解对象 |

* **获得类中构造器相关的方法**

| **方法** | **用途** |
| --- | --- |
| getConstructor(Class...<?> parameterTypes) | 获得该类中与参数类型匹配的公有构造方法 |
| getConstructors() | 获得该类的所有公有构造方法 |
| getDeclaredConstructor(Class...<?> parameterTypes) | 获得该类中与参数类型匹配的构造方法 |
| getDeclaredConstructors() | 获得该类所有构造方法 |

* **获得类中方法相关的方法（交给大家练习）**

| **方法** | **用途** |
| --- | --- |
| getMethod(String name, Class...<?> parameterTypes) | 获得该类某个公有的方法 |
| getMethods() | 获得该类所有公有的方法 |
| getDeclaredMethod(String name, Class...<?> parameterTypes) | 获得该类某个方法 |
| getDeclaredMethods() | 获得该类所有方法 |

* **类中其他重要的方法**

| **方法** | **用途** |
| --- | --- |
| isAnnotation() | 如果是注解类型则返回true |
| isAnnotationPresent(Class<? extends Annotation> annotationClass) | 如果是指定类型注解类型则返回true |
| isArray() | 如果是一个数组类则返回true |
| isEnum() | 如果是枚举类则返回true |
| isInstance(Object obj) | 如果obj是该类的实例则返回true |
| isInterface() | 如果是接口类则返回true |

## Field

[Field](https://developer.android.google.cn/reference/java/lang/reflect/Field" \t "_blank)代表类的成员变量。**成员变量（字段）和成员属性是两个概念。比如：当一个User类中有一个name变量，那么这个时候我们就说它有name这个字段。但是如果没有getName和setName这两个方法，那么这个类就没有name属性。反之，如果这个类拥有getAge和setAge这两个方法，不管有没有age字段，我们都认为它有age这个属性**

| **方法** | **用途** |
| --- | --- |
| get(Object obj) | 获得obj中对应的属性值 |
| set(Object obj, Object value) | 设置obj中对应属性值 |
| SetAccessible | 暴力反射，忽略访问权限修饰符 |

## Method

[Method](https://developer.android.google.cn/reference/java/lang/reflect/Method" \t "https://www.jianshu.com/p/_blank)代表类的方法（不包括构造方法）。

| **方法** | **用途** |
| --- | --- |
| invoke(Object obj, Object... args) | 传递object对象及参数调用该对象对应的方法 |
| getName | 获取方法名 |
| SetAccessible | 暴力反射，忽略访问权限修饰符 |

Invoke方法的用处：SpringAOP在切面方法执行的前后进行某些操作，就是使用的invoke方法。

## Constructor(用的比较少)

[Constructor](https://developer.android.google.cn/reference/java/lang/reflect/Constructor" \t "https://www.jianshu.com/p/_blank)代表类的构造方法。

| **方法** | **用途** |
| --- | --- |
| newInstance(Object... initargs) | 根据传递的参数创建类的对象 |

Constructor类在实际开发中使用极少，几乎不会使用Constructor。因为：Constructor违背了Java的一些思想，比如：私有构造不让用户去new对象；单例模式保证全局只有一个该类的实例。而Constructor则可以破坏这个规则

案例：写一个小工具，可以帮我们创建任意类的对象，并且执行其中任意方法

案例：通过反射，验证运行中泛型会被擦除

# 注解

概念：注解就是说明程序的一个标识，给计算机看的。

注释：用文字描述程序，给程序员看的。

定义：也叫作元数据，是一种代码级别的说明。它是JDK1.5引入的一个特性，是一种特殊的接口。它可以声明在类、字段、方法、变量、参数、包等前面，作为一个描述去使用。

作用分类：

编写文档：通过代码中标识的注解生成文档（Swagger）

代码分析：通过代码里的注解对代码进行分析（逻辑判断）

编译检查：通过代码里对应的注解让编译器实现基本的编译检查（Override，Deprecated，FunctionalInterface）

JDK中预定义的一些注解

Override：检测该注解标识的方法是否继承自父类

Deprecated：标识方法、类、字段等已经过时，后续的版本可能会将其移除

SuppressWarnings：压制警告

## 自定义注解

### 格式

元注解

Public @interface 注解名称 {

属性列表

}

本质：注解本质上是一个接口，该接口事实上默认继承自Annotation接口

### 属性

事实上是接口中的抽象方法

1. 如果定义了属性，在使用属性的时候需要给属性赋值。
2. 如果只有一个属性需要赋值，并且这个属性名称是value，则可以省略value
3. 数组赋值时，需要使用{}包起来。如果数组中只有一个元素，则大括号可以省略

属性中的返回值类型有下列取值：

\* 基本数据类型

\* String

\* 枚举

\* 注解

\* 以上类型的数组

### 元注解

用于描述注解的注解

@Target：描述该注解作用范围

ElementType取值：

Type：作用于类

METHOD：作用于方法

FIELD：作用于字段

@Retention：描述注解被保留的阶段

RetentionPolicy.RUNTIME：当前描述的注解，会保留到class字节码文件中，并被jvm读取到  
@Documented：描述注解是否被抽取到api文档中

@Inherited：描述注解是否可以被继承

## 案例

编写一个缓存注解，该注解用于缓存指定方法的返回值。

|  |
| --- |
| public final class CacheUtils {   */\*\*  \* HashMap是线程不安全的，这里应该用ConcurrentHashMap  \*/* private static Map<String, Object> *cacheMap* = new ConcurrentHashMap<>();   private CacheUtils() {  }   */\*\*  \* 执行当前指定对象的指定方法  \*  \** ***@param*** *obj  \** ***@param*** *methodName  \** ***@param*** *params  \** ***@return*** *\*/* public static Object invokeMethod(Object obj, String methodName, Object... params) {  Class<?> objClass = obj.getClass();  Object result = null;   try {  Method method;  if (params.length > 0) {  Class<?>[] classArr = new Class[params.length];  Object[] valueArr = new Object[params.length];  for (int i = 0; i < params.length; i++) {  classArr[i] = params[i].getClass();  valueArr[i] = params[i];  }  method = objClass.getDeclaredMethod(methodName, classArr);   // 获取缓存注解  Cache cacheAnnotation = method.getAnnotation(Cache.class);  // 先判断注解是否为空  if (cacheAnnotation != null) {  // 方法有参数，以第一个参数为小key。  Object paramsKey = params[0];  // 获取大key  String key = cacheAnnotation.key();  // 拼接key  String cacheKey = key + "." + paramsKey;  // 获取缓存  Object cacheValue = *cacheMap*.get(cacheKey);  // 判断缓存是否存在，如果存在，直接返回缓存的值  if (cacheValue != null) {  return cacheValue;  }  }  method.setAccessible(true);  result = method.invoke(obj, valueArr);  // 方法执行完了，将数据放入到缓存中  if(cacheAnnotation!=null) {  // 获取key  // 方法有参数，以第一个参数为小key。  Object paramsKey = params[0];  // 获取大key  String key = cacheAnnotation.key();  // 拼接key  String cacheKey = key + "." + paramsKey;  *cacheMap*.put(cacheKey, result);  }  } else {  method = objClass.getDeclaredMethod(methodName);  // 获取缓存注解  Cache cacheAnnotation = method.getAnnotation(Cache.class);  // 先判断注解是否为空  if (cacheAnnotation != null) {  // 获取大key  String key = cacheAnnotation.key();  // 获取缓存  Object cacheValue = *cacheMap*.get(key);  // 判断缓存是否存在，如果存在，直接返回缓存的值  if (cacheValue != null) {  return cacheValue;  }  }  method.setAccessible(true);  result = method.invoke(obj);  // 方法执行完了，将数据放入到缓存中  if(cacheAnnotation!=null) {  String key = cacheAnnotation.key();  *cacheMap*.put(key, result);  }  }  } catch (Exception e) {  e.printStackTrace();  }  return result;  }  } |