# 1. 注解

## 概述

注解与注释，

注解，告诉编译器如何运行程序！

注释，给程序员阅读，对编译、运行没有影响；

注解作用，

1. 告诉编译器如何运行程序；

2. 简化(取代)配置文件 【案例后再看】

常用的注解，

|  |
| --- |
| // 重写父类的方法  @Override  **public** String toString() {  **return** **super**.toString();  }    // 抑制编译器警告  @SuppressWarnings({"unused","unchecked"})  **private** **void** save() {  List list = **null**;  }    // 标记方法以及过时  @Deprecated  **private** **void** ~~save1~~() {  } |

## 自定义注解

注解:@interface

接口:interface

类:class

枚举:enum

通过自定义注解，可以给类、字段、方法上添加描述信息！

### a. 注解基本写法

|  |
| --- |
| /\*\*  \* 自定义注解 (描述一个作者)  \* **@author** zhanghong  \*  \*/  **public** **@interface** Author {  /\*\*  \* 注解属性  \* 1. 修饰为默认或public  \* 2. 不能有主体  \*/  String name();  **int** age();  } |

|  |
| --- |
| 使用 |
| @Author(name = "Jet", age = 30)  **public** **void** save() {  } |

### b.带默认值的注解

|  |
| --- |
| **public** **@interface** Author {  /\*\*  \* 注解属性  \* 1. 修饰为默认或public  \* 2. 不能有主体  \*/  String name();  **int** age() **default** 30; // 带默认值的注解; 使用的时候就可以不写此属性值  } |

### c.默认名称的注解

注解属性名称为value，这就是默认名称

|  |
| --- |
| **public** **@interface** Author {  // 如果注解名称为value,使用时候可以省略名称，直接给值  // (且注解只有一个属性时候才可以省略名称)  String value();  } |
| **使用** |
| **@Author("Jet")** |
| **@Author(value = "Jet")** |

### d.注解属性类型为数组

|  |
| --- |
| **public** **@interface** Author {    String[] value() **default** {"test1","test2"};  } |
| **使用：** |
| @Author（{“”，“”}）  **public** **void** save() {  } |

## 元注解

元注解，表示注解的注解！

作用案例1：指定注解只能使用在类上，或者属性上，或者方法上

作用案例2：用于获取注解信息，即反射上

**指定注解的可用范围：**

@Target({

TYPE, 类

FIELD, 字段

METHOD, 方法

PARAMETER, 参数

CONSTRUCTOR, 构造器

LOCAL\_VARIABLE 局部变量

})

// 元注解 - 2. 指定注解的生命周期

@Retention(RetentionPolicy.**SOURCE**) 注解只在源码级别有效 即编译后就没有注解

@Retention(RetentionPolicy.**CLASS**) 注解在字节码即别有效 **默认值**

@Retention(RetentionPolicy.**RUNTIME**) 注解在运行时期有效

上面的生命周期由小到大

## 注解反射

前提：直接的生命周期必须是RUNTIME

|  |
| --- |
| @Id  @Author(remark = "保存信息！！！", age = 19)  **public** **void** save() **throws** Exception {  // 获取注解信息： name/age/remark      // 1. 先获取代表方法的Method类型;  Class clazz = App.**class**;  Method m = clazz.getMethod("save");    // 2. 再获取方法上的注解  Author author = m.getAnnotation(Author.**class**);  // 获取输出注解信息  System.*out*.println(author.authorName());  System.*out*.println(author.age());  System.*out*.println(author.remark());  } |

# 2. 注解，优化BaseDao的代码

当表名与数据库名称不一致、 字段与属性不一样、主键不叫id， 上面的BaseDao不能用！

这时，可以通过配置文件(XML) 解决！

注解：

简化XML配置， 程序处理非常方便！

(不便于维护： 例如修改字段名，要重新编译！)

XML

便于维护！ 需要些读取代码！

# 3. Log4J日志组件

程序中为什么用日志组件？

简单来说，为了项目后期部署上线后的维护、错误排查！

Log4j, log for java, 开源的日志组件！

使用步骤：

1. 下载组件，引入jar文件;

log4j-1.2.11.jar

2. 配置 : src/log4j.properties

3. 使用

|  |
| --- |
| # 通过根元素指定日志输出的级别、目的地：  # 日志输出优先级： debug < info < warn < error  log4j.rootLogger=info,console,file  ############# 日志输出到控制台 #############  # 日志输出到控制台使用的api类  log4j.appender.console=org.apache.log4j.ConsoleAppender  # 指定日志输出的格式： 灵活的格式  log4j.appender.console.layout=org.apache.log4j.PatternLayout  # 具体格式内容  log4j.appender.console.layout.ConversionPattern=%d %p %c.%M()-%m%n  ############# 日志输出到文件 #############  log4j.appender.file=org.apache.log4j.RollingFileAppender  # 文件参数： 指定日志文件路径  log4j.appender.file.File=../logs/MyLog.log  # 文件参数： 指定日志文件最大大小  log4j.appender.file.MaxFileSize=5kb  # 文件参数： 指定产生日志文件的最大数目  log4j.appender.file.MaxBackupIndex=100  # 日志格式  log4j.appender.file.layout=org.apache.log4j.PatternLayout  log4j.appender.file.layout.ConversionPattern=%d %c.%M()-%m%n |