基础增强：

# 1. 课程回顾

XML技术

Servlet/Jsp

数据库

Jdbc技术

JavaWeb增强

过滤器、监听器、文件上传与下载、国际化

学过的组件：

Dom4J/Xpath、DBUtils、C3p0、BeanUtils、log4J…

Struts / Spring / Hibernate

目标：

1. 泛型

2. 注解

3. log4J 日志 (项目中)

4. 枚举

# 2. 泛型

掌握的知识：

**基本用法、泛型擦除、泛型类/泛型方法/泛型接口**、泛型关键字、反射泛型(案例)！

## a. 概述

泛型是JDK1.5以后才有的， 可以在编译时期进行类型检查，且可以避免频繁类型转化！

|  |
| --- |
| // 运行时期异常  @Test  **public** **void** testGeneric() **throws** Exception {  // 集合的声明  List list = **new** ArrayList();  list.add("China");  list.add(1);    // 集合的使用  String str = (String) list.get(1);    }    // 使用泛型  @Test  **public** **void** testGeneric2() **throws** Exception {  // 声明泛型集合的时候指定元素的类型  List<String> list = **new** ArrayList<String>();  list.add("China");  // list.add(1);// 编译时期报错    String str = list.get(1);  } |

**泛型擦除，**

**泛型只在编译时期有效，编译后的字节码文件中不存在有泛型信息！**

|  |
| --- |
| /\*  \* 泛型擦除实例    public void save(List<Person> p){  }  public void save(List<Dept> d){ // 报错： 与上面方法编译后一样  }  \*/ |

**泛型写法:**

|  |
| --- |
| // 泛型写法  @Test  **public** **void** testGeneric3() **throws** Exception {  // 声明泛型集合，集合两端类型必须一致  List<Object> list = **new** ArrayList<Object>();  List<String> list1 = **new** ArrayList<String>();  List list2 = **new** ArrayList<String>();  List<Integer> list3 = **new** ArrayList();    // 错误  //List<Object> list4 = new ArrayList<String>();  // 错误： 泛型类型必须是引用类型,不能为基本类型  List<**int**> list5 = **new** ArrayList<**int**>();  } |

## b. 泛型方法/泛型类/泛型接口

作用：

设计公用的类、方法，对公用的业务实现进行抽取！

使程序更灵活！

**1. 泛型方法:**

|  |
| --- |
| **public** **class** GenericDemo {  // 定义泛型方法  **public** <K,T> T save(T t,K k) {  **return** **null**;  }    // 测试方法  @Test  **public** **void** testMethod() **throws** Exception {  // 使用泛型方法: 在使用泛型方法的时候，确定泛型类型  save(1.0f, 1);  }  } |

**2. 泛型类：**

|  |
| --- |
| **public** **class** GenericDemo<T> {  // 定义泛型方法  **public** <K> T save(T t,K k) {  **return** **null**;  }    **public** **void** update(T t) {  }    // 测试方法  @Test  **public** **void** testMethod() **throws** Exception {    // 泛型类： 在创建爱泛型类对象的时候，确定类型  GenericDemo<String> demo = **new** GenericDemo<String>();  demo.save("test", 1);  }  } |

**3. 泛型接口：**

|  |
| --- |
| /\*\*  \* 泛型接口  \* **@author** Jie.Yuan  \*  \* **@param** <T>  \*/  **public** **interface** IBaseDao<T> {  **void** save(T t );  **void** update(T t );  } |
| 泛型接口类型确定： 实现泛型接口的类也是抽象，那么类型在具体的实现中确定或创建泛型类的时候确定 |
| **public** **class** BaseDao<T> **implements** IBaseDao<T> { |
| 泛型接口类型确定： 在业务实现类中直接确定接口的类型 |
| **public** **class** PersonDao **implements** IBaseDao<Person>{ |
|  |

## c. 泛型关键字

泛型中：

? 指定只是接收值

extends 元素的类型必须继承自指定的类

super 元素的类型必须是指定的类的父类

**关键字 ： ?**

|  |
| --- |
| /\*\*  \* 泛型, 涉及到一些关键字  \*  \* Ctrl + shift + R 查看当前项目中类  \* Ctrl + shift + T 查看源码jar包中的类  \* **@author** Jie.Yuan  \*  \*/  **public** **class** App\_extends\_super {    //只带泛型特征的方法  **public** **void** save(List<?> list) {  // 只能获取、迭代list; 不能编辑list  }  @Test  **public** **void** testGeneric() **throws** Exception {    // ? 可以接收任何泛型集合， 但是不能编辑集合值； 所以一般在方法参数中用  List<?> list = **new** ArrayList<String>();  //list.add("");// 报错  }  } |

**关键字 ： extends 【上限】**

|  |
| --- |
| **public** **class** App\_extends\_super {      /\*\*  \* list集合只能处理 Double/Float/Integer等类型  \* 限定元素范围：元素的类型要继承自Number类 (上限)  \* **@param** list  \*/  **public** **void** save(List<? **extends** Number> list) {  }  @Test  **public** **void** testGeneric() **throws** Exception {  List<Double> list\_1 = **new** ArrayList<Double>();  List<Float> list\_2 = **new** ArrayList<Float>();  List<Integer> list\_3 = **new** ArrayList<Integer>();    List<String> list\_4 = **new** ArrayList<String>();    // 调用  save(list\_1);  save(list\_2);  save(list\_3);  //save(list\_4);  }  } |

**关键字 ： super 【下限】**

|  |
| --- |
| /\*\*  \* 泛型, 涉及到一些关键字  \*  \* Ctrl + shift + R 查看当前项目中类  \* Ctrl + shift + T 查看源码jar包中的类  \* **@author** Jie.Yuan  \*  \*/  **public** **class** App\_super {      /\*\*  \* super限定元素范围：必须是String父类 【下限】  \* **@param** list  \*/  **public** **void** save(List<? **super** String> list) {  }  @Test  **public** **void** testGeneric() **throws** Exception {  // 调用上面方法，必须传入String的父类  List<Object> list1 = **new** ArrayList<Object>();  List<String> list2 = **new** ArrayList<String>();    List<Integer> list3 = **new** ArrayList<Integer>();  //save(list3);  }  } |

## d. 泛型的反射

案例，设置通用方法，会用到反射泛型！

步骤：

1. 案例分析 / 实现

2. 涉及知识点(jdk api)

3. 优化 / 反射泛型

反射泛型涉及API：

Student 类型的表示

Id name

[ParameterizedType](mk:@MSITStore:E:\API\JDK_API_1_6_zh_CN.CHM::/java/lang/reflect/ParameterizedType.html) 参数化类型的表示

**ArrayList<String>();**

Type 接口，任何类型默认的接口！

包括： 引用类型、原始类型、**参数化类型**

**List<String> list = new ArrayList<String>();**

**泛型集合： list**

**集合元素定义：new ArrayList<String>(); 中的String**

**参数化类型：** [ParameterizedType](mk:@MSITStore:E:\API\JDK_API_1_6_zh_CN.CHM::/java/lang/reflect/ParameterizedType.html)

**即：“ArrayList<String> ” 为参数化类型**

**反射泛型案例**

|  |
| --- |
| **public** **class** AdminDao **extends** BaseDao<Admin> {}  **public** **class** AccountDao **extends** BaseDao<Account> {} |
| /\*\*  \* 所有dao的公用的方法，都在这里实现  \* **@author** Jie.Yuan  \*  \*/  **public** **class** BaseDao<T>{    // 保存当前运行类的参数化类型中的实际的类型  **private** Class clazz;  // 表名  **private** String tableName;        // 构造函数： 1. 获取当前运行类的参数化类型； 2. 获取参数化类型中实际类型的定义(class)  **public** BaseDao(){  // this 表示当前运行类 (AccountDao/AdminDao)  // this.getClass() 当前运行类的字节码(AccountDao.class/AdminDao.class)  // this.getClass().getGenericSuperclass(); 当前运行类的父类，即为BaseDao<Account>  // 其实就是“参数化类型”， ParameterizedType  Type type = **this**.getClass().getGenericSuperclass();  // 强制转换为“参数化类型” 【BaseDao<Account>】  ParameterizedType pt = (ParameterizedType) type;  // 获取参数化类型中，实际类型的定义 【new Type[]{Account.class}】  Type types[] = pt.getActualTypeArguments();  // 获取数据的第一个元素：Accout.class  clazz = (Class) types[0];  // 表名 (与类名一样，只要获取类名就可以)  tableName = clazz.getSimpleName();  }    /\*\*  \* 主键查询  \* **@param** id 主键值  \* **@return** 返回封装后的对象  \*/  **public** T findById(**int** id){  /\*  \* 1. 知道封装的对象的类型  \* 2. 表名【表名与对象名称一样， 且主键都为id】  \*  \* 即，  \* ---》得到当前运行类继承的父类 BaseDao<Account>  \* ----》 得到Account.class  \*/    String sql = "select \* from " + tableName + " where id=? ";  **try** {  **return** JdbcUtils.*getQuerrRunner*().query(sql, **new** BeanHandler<T>(clazz), id);  } **catch** (SQLException e) {  **throw** **new** RuntimeException(e);  }  }      /\*\*  \* 查询全部  \* **@return**  \*/  **public** List<T> getAll(){  String sql = "select \* from " + tableName ;  **try** {  **return** JdbcUtils.*getQuerrRunner*().query(sql, **new** BeanListHandler<T>(clazz));  } **catch** (SQLException e) {  **throw** **new** RuntimeException(e);  }  }  } |

# 4. 反射复习

反射，可以在运行时期动态创建对象；获取对象的属性、方法；

|  |
| --- |
| **public** **class** Admin {  // Field  **private** **int** id = 1000;  **private** String name = "匿名";    // Constructor  **public** Admin(){  System.*out*.println("Admin.Admin()");  }  **public** Admin(String name){  System.*out*.println("Admin.Admin()" + name);  }    // Method  **public** **int** getId() {  **return** id;  }  **public** **void** setId(**int** id) {  **this**.id = id;  }  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }    } |
| // 反射技术  **public** **class** App {  // 1. 创建对象  @Test  **public** **void** testInfo() **throws** Exception {  // 类全名  String className = "cn.itcast.c\_reflect.Admin";  // 得到类字节码  Class<?> clazz = Class.*forName*(className);    // 创建对象1: 默认构造函数简写  //Admin admin = (Admin) clazz.newInstance();    // 创建对象2： 通过带参数构造器创建对象  Constructor<?> constructor = clazz.getDeclaredConstructor(String.**class**);  Admin admin = (Admin) constructor.newInstance("Jack");    }  @Test  //2. 获取属性名称、值  **public** **void** testField() **throws** Exception {    // 类全名  String className = "cn.itcast.c\_reflect.Admin";  // 得到类字节码  Class<?> clazz = Class.*forName*(className);  // 对象  Admin admin = (Admin) clazz.newInstance();    // 获取所有的属性名称  Field[] fs = clazz.getDeclaredFields();  // 遍历：输出每一个属性名称、值  **for** (Field f : fs) {  // 设置强制访问  f.setAccessible(**true**);  // 名称  String name = f.getName();  // 值  Object value = f.get(admin);    System.*out*.println(name + value);  }  }    @Test  //3. 反射获取方法  **public** **void** testMethod() **throws** Exception {    // 类全名  String className = "cn.itcast.c\_reflect.Admin";  // 得到类字节码  Class<?> clazz = Class.*forName*(className);  // 对象  Admin admin = (Admin) clazz.newInstance();    // 获取方法对象 public int getId() {  Method m = clazz.getDeclaredMethod("getId");  // 调用方法  Object r\_value = m.invoke(admin);    System.*out*.println(r\_value);  }    } |

# 3. 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

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** Jie.Yuan  \*  \*/  **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();  // 带默认值的注解; 使用的时候就可以不写此属性值  **int** age1() **default** 30;  } |

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

### b.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")** |

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！

**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. 指定注解的声明周期

@Retention(RetentionPolicy.**SOURCE**) 注解只在源码级别有效

@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

## 注解反射

|  |
| --- |
| @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\_2.**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());  } |

# 4. 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！

这是，

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！

# 5. 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 |