# 一、JAVA

## 内部类

### 作用

* 如果你想实现一个接口，但是这个接口中的一个方法和你构想的这个类中的一个方法的名称，参数相同。这时候，你可以建一个内部类实现这个接口
* 很好的实现多继承的效果。
* 你甚至连这两个内部类的名字都没有看见！这样，内部类的第一个好处就体现出来了 隐藏你不想让别人知道的操作，也即封装性
* 非静态内部类对象有着指向其外部类对象的引用。即：一个内部类对象可以访问创建它的外部类对象的内容，甚至包括私有变量！这是一个非常有用的特性

### 创建方式

public class Test0530 {

private int valueRateRate = 2;

private class Content implements Contents {

private int i = 11;

public int value() {

// 此处证明作用第四点（非静态内部类对象有着指向其外部类对象的引用）

// return i \* valueRateRate;

// 如果内外两个类变量名相同 还可以这样区分引用外部类的变量

return i \* Test0530.this.valueRateRate;

}

}

protected class GDestination implements Destination {

private String label;

private GDestination(String whereTo) {

label = whereTo;

}

public String readLabel() {

return label;

}

}

public Destination dest(String s) {

return new GDestination(s);

}

public Contents cont() {

return new Content();

}

public static void main(String[] args) {

Test0530 p = new Test0530();

// 创建内部类的第一种方式：证明了作用中的第三点（隐蔽性）

/\*Contents c = p.cont();

Destination d = p.dest("Beijing"); \*/

// 创建内部类的第二种方式：此时创建的话需要实例化的外部类对象，

// 为什呢？这里可以看作用的第四点（非静态内部类对象有着指向其外部类对象的引用）

Test0530.GDestination dest = p.new GDestination("Beijing");

}

}

interface Contents {

int value();

}

interface Destination {

String readLabel();

}

### 静态内部类

**区别就在于静态内部类没有了指向外部的引用**

两者区别：非静态内部类中，都不能有静态数据，静态方法或者又一个静态内部类（内部类的嵌套可以不止一层）。不过静态内部类中却可以拥有这一切。

### 局部内部类

从下面代码中可以看出局部内部类可以写在某个方法里，除了这种还可以在方法中的if语句块中进行编写内部类代码，创建方式多种多样，具体作用具体分析

public Destination dest(String s) {

class GDestination implements Destination {

private String label;

private GDestination(String whereTo) {

label = whereTo;

}

public String readLabel() {

return label;

}

}

return new GDestination(s);

}

### 匿名内部类

当你只需要创建一个类的对象而且用不上它的名字时，可以使用这种方式new interfacename(){......}; 或 new superclassname(){......};，如下具体方式使代码看上去简洁清楚。

public class Goods3 {

public Contents cont() {

return new Contents() {

private int i = 11;

public int value() {

return i;

}

};

}

}

# 二、日志的使用

通过测试以下代码又在MyBlog(ssm)/springboot项目中使用可行

**相关代码：**

// 所在包

import org.apache.log4j.Logger;

// 创建Logger对象

private static Logger log = Logger.getLogger(MainControl.class); // 日志对象

// 操作日志测试

log.info("你好"); // 打印

log.debug("你好"); // 未打印

log.warn("你好"); // 打印

log.error("你好"); // 打印

**Springboot中的依赖：**

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.17</version>

</dependency>

**log4j.properties文件：**

#log4j.rootLogger=ERROR

log4j.rootLogger=ERROR,console,chen

log4j.logger.org.hibernate.cache=error

log4j.appender.console=org.apache.log4j.ConsoleAppender

log4j.appender.console.layout=org.apache.log4j.PatternLayout

log4j.appender.console.layout.ConversionPattern=[%-5p] %d{HH\:mm\:ss\:S} [%C{1}.%M\:%L] %m %n

log4j.logger.freemarker=ERROR

log4j.logger.com.chen=INFO

log4j.logger.net.sf.ehcache=ERROR

log4j.logger.org.apache.shiro=ERROR

log4j.logger.org.springframework.web=ERROR

log4j.logger.net.sf.hibernate=ERROR

log4j.logger.org.hibernate.tool.hbm2ddl=INFO

log4j.logger.org.hibernate.SQL=ERROR

#\u4e3a\u4e86\u663e\u793a\u53c2\u6570TRACE

log4j.logger.org.hibernate.type.descriptor.sql.BasicBinder=ERROR

log4j.logger.org.hibernate.type.descriptor.sql.BasicExtractor=ERROR

#\u67e5\u770b\u67e5\u8be2\u4e2d\u547d\u540d\u53c2\u6570\u7684\u503c

log4j.logger.org.hibernate.engine.QueryParameters=ERROR

log4j.logger.org.hibernate.engine.query.HQLQueryPlan=ERROR

#log4j.appender.stdout.Encoding=GBK

log4j.appender.chen=org.apache.log4j.DailyRollingFileAppender

log4j.appender.chen.File=system/springboottest/chen.log

log4j.appender.chen.immediateFlush=true

log4j.appender.chen.layout=org.apache.log4j.PatternLayout

log4j.appender.chen.layout.ConversionPattern=[%-5p] %d{HH:mm:ss:S} [%C{1}.%M:%L] %m %n

# 三、Mybatis

## 简介

是一款优秀的持久层框架，它支持定制化 SQL、存储过程以及高级映射。

## 核心API

**SqlSessionFactory/SqlSessionFactoryBuilder/SqlSession**

SqlSessionFactory sessionFactory = new SqlSessionFactoryBuilder()

SqlSession session = sessionFactory.openSession() ;

HotelMapper hotelMapper = session.getMapper(HotelMapper.class);

## 映射文件

### #{}、${}的区别

1. **#{}：**

PreparedStatement ps = conn.prepareStatement(sql);

ps.setInt(1,id);

优点：更安全，更迅速，通常也是首选做法。防止sql注入攻击

1. **${}：**

Statement st = conn.createStatement();

ResultSet rs = st.executeQuery(sql);

缺点：以这种方式接受从用户输出的内容并提供给语句中不变的字符串是不安全的，会导致潜在的 SQL 注入攻击，因此要么不允许用户输入这些字段，要么自行转义并检验。

### jdbcType类型归纳

Mybatis中javaType和jdbcType对应关系：

----------------------------------------------------

JDBC Type Java Type

----------------------------------------------------

CHAR String

VARCHAR String

LONGVARCHAR String

NUMERIC java.math.BigDecimal

DECIMAL java.math.BigDecimal

BIT boolean

BOOLEAN boolean

TINYINT byte

SMALLINT short

INTEGER int

BIGINT long

REAL float

FLOAT double

DOUBLE double

BINARY byte[]

VARBINARY byte[]

LONGVARBINARY byte[]

DATE java.sql.Date

TIME java.sql.Time

TIMESTAMP java.sql.Timestamp

CLOB Clob

BLOB Blob

ARRAY Array

DISTINCT mapping of underlying type

STRUCT Struct

REF Ref

DATALINK java.net.URL[color=red][/color]

----------------------------------------------------

例子：

Mybatis中javaType和jdbcType对应和CRUD例子：

<resultMap type="java.util.Map" id="resultjcm">

<result property="FLD\_NUMBER" column="FLD\_NUMBER" javaType="double" jdbcType="NUMERIC"/>

<result property="FLD\_VARCHAR" column="FLD\_VARCHAR" javaType="string" jdbcType="VARCHAR"/>

<result property="FLD\_DATE" column="FLD\_DATE" javaType="java.sql.Date" jdbcType="DATE"/>

<result property="FLD\_INTEGER" column="FLD\_INTEGER" javaType="int" jdbcType="INTEGER"/>

<result property="FLD\_DOUBLE" column="FLD\_DOUBLE" javaType="double" jdbcType="DOUBLE"/>

<result property="FLD\_LONG" column="FLD\_LONG" javaType="long" jdbcType="INTEGER"/>

<result property="FLD\_CHAR" column="FLD\_CHAR" javaType="string" jdbcType="CHAR"/>

<result property="FLD\_BLOB" column="FLD\_BLOB" javaType="[B" jdbcType="BLOB" />

<result property="FLD\_CLOB" column="FLD\_CLOB" javaType="string" jdbcType="CLOB"/>

<result property="FLD\_FLOAT" column="FLD\_FLOAT" javaType="float" jdbcType="FLOAT"/>

<result property="FLD\_TIMESTAMP" column="FLD\_TIMESTAMP" javaType="java.sql.Timestamp" jdbcType="TIMESTAMP"/>

</resultMap>

### resultType 返回类型：

**javaBean ：**

java代码：

Employee getEmpById(Integer id);

配置：

<select id="getEmpById" resultType="employee">

**List<T> ：**

java代码：

List<Employee> getAllEmps();

配置：

<select id="getAllEmps" resultType="employee"> // 返回值类型是集合内存储数据的类型，不是 'list'

**Map<String, Object> ：**

查询结果为一条：

Map中的数据格式为{表字段名， 对应的值}

java代码：

Map<String, Object> getEmpAsMapById(Integer id);

配置：

<select id="getEmpAsMapById" resultType="map">

查询结果为多条：

Map中的数据格式为{表字段名， 对应的值}

java代码：

Map<String, Object> getEmpAsMapById(Integer id);

配置：

<select id="getEmpAsMapById" resultType="map">

**自定义返回一个对象中的那些字段：**

使用心得

再多表查询中可能内心会问，此处配置了关联，对面需要配置关联吗。答案是不需要

java代码：

List<DdChargeWater> gets();

配置：

// resultMap 内容为空 按给定类型DdChargeWater中的属性映射

<resultMap id="userList" type="com.blog.entity.DdChargeWater"> </resultMap>

/\*

在resultMap 中自定义映射字段属性，这多用于多表查询

<resultMap id="userList" type="com.blog.entity.DdChargeWater">

<id column="表的主键字段，或者可以为查询语句中的别名字段" jdbcType="字段类型" property="映射pojo对象的主键属性" />

<association property="pojo的一个对象属性" javaType="pojo关联的pojo对象">

<id column="关联pojo对象对应表的主键字段" jdbcType="字段类型" property="关联pojo对象的主席属性"/>

<result column="任意表的字段" jdbcType="字段类型" property="关联pojo对象的属性"/>

</association>

<!-- 集合中的property须为oftype定义的pojo对象的属性-->

<collection property="pojo的集合属性" ofType="集合中的pojo对象">

<id column="集合中pojo对象对应的表的主键字段" jdbcType="字段类型" property="集合中pojo对象的主键属性" />

<result column="可以为任意表的字段" jdbcType="字段类型" property="集合中的pojo对象的属性" />

</collection>

</resultMap>

\*/

<select id="gets" resultMap="userList" useCache="false">

SELECT \* FROM ${table}

</select>

**以及返回Boolean，Integer 等；**

## 注解的使用

### @Select

### @ResultType

使用该注解来映射查询结果到实体类属性。

@ResultType(Demo.class)

### @Results的总结

MyBatis中使用@Results注解来映射查询结果集到实体类属性。

#### （1）@Results的基本用法。

当数据库字段名与实体类对应的属性名不一致时，可以使用@Results映射来将其对应起来。column为数据库字段名，porperty为实体类属性名，jdbcType为数据库字段数据类型，id为是否为主键。

@Select({"select id, name, class\_id from my\_student"})

@Results({

@Result(column="id", property="id", jdbcType=JdbcType.INTEGER, id=true),

@Result(column="name", property="name", jdbcType=JdbcType.VARCHAR),

@Result(column="class\_id ", property="classId", jdbcType=JdbcType.INTEGER)

})

List<Student> selectAll();

如上所示的数据库字段名class\_id与实体类属性名classId，就通过这种方式建立了映射关系。

#### @ResultMap的用法。

当这段@Results代码需要在多个方法用到时，为了提高代码复用性，我们可以为这个@Results注解设置id，然后使用@ResultMap注解来复用这段代码。

@Select({"select id, name, class\_id from my\_student"})

@Results(id="studentMap", value={

@Result(column="id", property="id", jdbcType=JdbcType.INTEGER, id=true),

@Result(column="name", property="name", jdbcType=JdbcType.VARCHAR),

@Result(column="class\_id ", property="classId", jdbcType=JdbcType.INTEGER)

})

List<Student> selectAll();

@Select({"select id, name, class\_id from my\_student where id = #{id}"})

@ResultMap(value="studentMap")

Student selectById(integer id);

#### @One的用法。

当我们需要通过查询到的一个字段值作为参数，去执行另外一个方法来查询关联的内容，而且两者是一对一关系时，可以使用@One注解来便捷的实现。比如当我们需要查询学生信息以及其所属班级信息时，需要以查询到的class\_id为参数，来执行ClassesMapper中的selectById方法，从而获得学生所属的班级信息。可以使用如下代码。

@Select({"select id, name, class\_id from my\_student"})

@Results(id="studentMap", value={

@Result(column="id", property="id", jdbcType=JdbcType.INTEGER, id=true),

@Result(column="name", property="name", jdbcType=JdbcType.VARCHAR),

@Result(column="class\_id ", property="myClass", javaType=MyClass.class,

one=@One(select="com.my.mybatis.mapper.MyClassMapper.selectById"))

})

List<Student> selectAllAndClassMsg();

#### @Many的用法。

与@One类似，只不过如果使用@One查询到的结果是多行，会抛出TooManyResultException异常，这种时候应该使用的是@Many注解，实现一对多的查询。比如在需要查询学生信息和每次考试的成绩信息时。

@Select({"select id, name, class\_id from my\_student"})

@Results(id="studentMap", value={

@Result(column="id", property="id", jdbcType=JdbcType.INTEGER, id=true),

@Result(column="name", property="name", jdbcType=JdbcType.VARCHAR),

@Result(column="class\_id ", property="classId", jdbcType=JdbcType.INTEGER),

@Result(column="id", property="gradeList", javaType=List.class,

many=@Many(select="com.my.mybatis.mapper.GradeMapper.selectByStudentId"))

})

List<Student> selectAllAndGrade();

# 项目记录

## shopManerge

### 1.1整合Thymeleaf

初步学习springboot

测试th:href 对静态文件的访问，以及链接的请求 （th:href请求指向static和templates

mycss.css

login.html

register.html

LoginController.java

login使用a标签向register页面跳转<a th:href="@{register}">注册</a><br>

login对静态文件的获取mycss.css<link rel="stylesheet" th:href="@{css/mycss.css}" />

### 1.2初步搭建

login.html

LoginController.java

依赖模块Thymeleaf

## Jsptest/springboottest

### 2.1整合jsp

#### 2.1.1依赖配置

<!-- servlet 依赖 -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

</dependency>

<!-- JSTL jsp标准标签库 依赖 -->

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>jstl</artifactId>

</dependency>

<!-- tomcat 依赖 -->

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-tomcat</artifactId>

</dependency>

<!-- jsp引擎 依赖 -->

<dependency>

<groupId>org.apache.tomcat.embed</groupId>

<artifactId>tomcat-embed-jasper</artifactId>

</dependency>

#### 2.1.2application.yml配置文件

在application中指向jsp文件路径 /WEB-INF/jsp/ 和 后缀.jsp

spring:

datasource:

username: chenjiwei

password: cjw168999

url: jdbc:mysql://120.79.149.233:3306/shop

driver-class-name: com.mysql.jdbc.Driver

hikari:

idle-timeout: 60000

maximum-pool-size: 30

minimum-idle: 10

mvc:

view:

prefix: /WEB-INF/jsp/

suffix: .jsp

#### 2.1.3成功访问：

http://localhost:8080/login

### 2.2在项目中使用fastjson替换原来默认的jackson

Controller返回的是一个java对象或者是list集合时自动转换为json数据

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>fastjson</artifactId>

<version>1.2.29</version>

</dependency>

### 2.3日志的使用

具体见一、JAVA 日志的使用

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.17</version>

</dependency>

### 2.4热部署

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-devtools</artifactId>

<!-- optional=true,依赖不会传递 -->

<optional>true</optional>

<scope>true</scope>

</dependency>

<!-- 必须含有build标签的内容，不然热部署失效 -->

<build>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<configuration>

<fork>true</fork>

</configuration>

</plugin>

</plugins>

</build>

### 2.5单元测试

Springboot中的单元测试，其实就是模拟请求，测试请求的数据是否正确

#### 2.5.1依赖配置

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

#### 2.5.2代码：

**LoginControlTest.java**

@RunWith(SpringRunner.class)

@SpringBootTest(classes = App.class)

public class LoginControlTest {

// 注入spring容器

@Autowired

private WebApplicationContext wac;

// 实现了对Http请求的模拟

private MockMvc mvc;

@Before

public void setupMockMvc(){

mvc = MockMvcBuilders.webAppContextSetup(wac).build();

}

@Test

@Transactional

public void getUserInfor(){

// MediaType.APPLICATION\_FORM\_URLENCODED

// Content-Type:"application/x-www-form-urlencoded"

String data="loginName=root";

try {

mvc.perform(MockMvcRequestBuilders.post("/user")

.contentType(MediaType.APPLICATION\_FORM\_URLENCODED)

.accept(MediaType.APPLICATION\_JSON\_UTF8)

.content(data)

).andExpect(MockMvcResultMatchers.status().isOk())

.andDo(MockMvcResultHandlers.print());

} catch (Exception e) {

e.printStackTrace();

}

/\*

// MediaType.APPLICATION\_JSON\_UTF8

// Content-Type:"application/json;charset=UTF-8"

String json="{\"loginName\":\"root\"}";

try {

mvc.perform(MockMvcRequestBuilders.post("/user")

.contentType(MediaType.APPLICATION\_JSON\_UTF8) // Content-Type:"application/json;charset=UTF-8"

.accept(MediaType.APPLICATION\_JSON\_UTF8) // Accept:"application/json;charset=UTF-8"

.content(json)

).andExpect(MockMvcResultMatchers.status().isOk())

.andDo(MockMvcResultHandlers.print());

} catch (Exception e) {

e.printStackTrace();

}\*/

}

}

#### 2.5.3输出

MockHttpServletRequest:

HTTP Method = POST

Request URI = /user

Parameters = {loginName=[root]}

Headers = [Content-Type:"application/x-www-form-urlencoded", Accept:"application/json;charset=UTF-8"]

Body = <no character encoding set>

Session Attrs = {}

Handler:

Type = com.chen.springboottest.control.LoginController

Method = public com.chen.springboottest.entity.User com.chen.springboottest.control.LoginController.user(java.lang.String)

Async:

Async started = false

Async result = null

Resolved Exception:

Type = null

ModelAndView:

View name = null

View = null

Model = null

FlashMap:

Attributes = null

MockHttpServletResponse:

Status = 200

Error message = null

Headers = [Content-Type:"application/json;charset=UTF-8"]

Content type = application/json;charset=UTF-8

Body = {"loginName":"root","password":"123456"}

Forwarded URL = null

Redirected URL = null

Cookies = []