

Java 学习笔记

LICHOO

总目录

之一、Java基础

4月18日的五天Java训练营（刘英谦老师主讲，占笔记内容10%）；
Java基础和面向对象（刘苍松老师主讲，占笔记内容40%）；
Java SE核心I和Java SE核心II（范传奇老师主讲，占笔记内容40%）；
其他视频及书籍资料（占笔记内容10%）

目 录

一、 Java技术基础.....	29
1.1 编程语言.....	29
1.2 Java 的特点.....	29
1.3 Java 开发环境.....	29
1.4 Java 开发环境配置.....	30
1.5 Linux 命令与相关知识.....	30
1.6 Eclipse/Myeclipse 程序结构.....	31
二、 Java语言基础.....	32
2.1 基础语言要素.....	32
2.2 八种基本数据类型.....	32
2.3 常量和变量.....	33
2.4 运算符与表达式.....	33
2.5 编程风格.....	34
2.6 流程控制语句.....	35
2.7 数组.....	36
2.8 字符串.....	36
2.9 方法三要素.....	37
2.10 插入排序.....	37
2.11 冒泡排序.....	37
2.12 冒泡排序：轻气泡上浮的方式.....	38
2.13 二分法查找.....	38
2.14 Java 系统 API 方法调用.....	39
2.15 二进制基础.....	39
2.16 Java 基础其他注意事项.....	39
三、 面向对象.....	41
3.1 类.....	41
3.2 对象.....	41
3.3 包.....	41
3.4 方法及其调用.....	42
3.5 引用.....	42
3.6 访问控制（封装）.....	42

3.7 构造器.....	42
3.8 super()、super. 和 this()、this	43
3.9 重载和重写.....	44
3.10 继承.....	45
3.11 static.....	48
3.12 final.....	49
3.13 多态.....	50
3.14 抽象类.....	50
3.15 接口.....	50
3.16 内部类.....	51
3.17 匿名类.....	52
3.18 二维数组和对象数组.....	53
3.19 其他注意事项.....	54
四、 Java SE 核心 I.....	55
4.1 Object 类.....	55
4.2 String 类.....	56
4.3 StringUtils 类.....	58
4.4 StringBuilder 类.....	58
4.5 正则表达式.....	59
4.6 Date 类.....	60
4.7 Calendar 类.....	60
4.8 SimpleDateFormat 类.....	61
4.9 DateFormat 类.....	61
4.10 包装类.....	62
4.11 BigDecimal 类.....	63
4.12 BigInteger 类.....	63
4.13 Collection 集合框架.....	63
4.14 List 集合的实现类 ArrayList 和 LinkedList.....	64
4.15 Iterator 迭代器.....	67
4.16 泛型.....	68
4.17 增强型 for 循环.....	68
4.18 List 高级一数据结构： Queue 队列.....	69
4.19 List 高级一数据结构： Deque 栈.....	69
4.20 Set 集合的实现类 HashSet.....	70
4.21 Map 集合的实现类 HashMap.....	71
4.22 单例模式和模版方法模式.....	73
五、 Java SE 核心 II.....	74
5.1 Java 异常处理机制.....	74
5.2 File 文件类.....	76
5.3 RandomAccessFile 类.....	78
5.4 基本流： FIS 和 FOS.....	80
5.5 缓冲字节高级流： BIS 和 BOS.....	81
5.6 基本数据类型高级流： DIS 和 DOS.....	81
5.7 字符高级流： ISR 和 OSW.....	82

5.8 缓冲字符高级流: BR 和 BW.....	83
5.9 文件字符高级流: FR 和 FW.....	84
5.10 PrintWriter.....	85
5.11 对象序列化.....	85
5.12 Thread 线程类及多线程.....	86
5.13 Socket 网络编程.....	90
5.14 线程池.....	92
5.15 双缓冲队列.....	93

之二、*Oracle*数据库、SQL (薛海璐老师主讲, 占笔记内容 100%);

一、 数据库介绍.....	94
1.1 表是数据库中存储数据的基本单位.....	94
1.2 数据库标准语言.....	94
1.3 数据库 (DB)	95
1.4 数据库种类.....	95
1.5 数据库中如何定义表.....	95
1.6 create database dbname 的含义.....	95
1.7 安装 DBMS	95
1.8 宏观上是数据-->database.....	95
1.9 远程登录: telnet IP 地址.....	95
1.10 TCP/IP 通信协议.....	95
1.11 数据库建连接必须提供以下信息.....	95
1.12 一台机器可跑几个数据库, 主要受内存大小影响.....	96
1.13 源表和结果集.....	96
1.14 几个简单命令.....	96
1.15 tarena 给 jsd1304 授权.....	96
1.16 课程中使用的 5 个表.....	96
二、 select from 语句.....	99
2.1 select 语句功能.....	99
2.2 select 语句基本语法.....	99
2.3 列别名.....	99
2.4 算术表达式.....	99
2.5 空值 null 的处理.....	99
2.6 nvl(p1,p2)函数.....	99
2.7 拼接运算符 	100
2.8 文字字符串.....	100
2.9 消除重复行.....	100
2.10 其他注意事项.....	100
三、 SQL 语句的处理过程.....	101
3.1 SQL 语句处理过程.....	101

3.2 处理一条 select 语句.....	101
四、 where 子句.....	102
4.1 where 子句后面可以跟什么.....	102
4.2 语法和执行顺序.....	102
4.3 字符串是大小写敏感的，在比较时严格区分大小写.....	102
4.4 where 子句后面可以跟多个条件表达式.....	102
4.5 between and 运算符.....	102
4.6 in 运算符（多值运算符）.....	102
4.7 like 运算符.....	103
4.8 is null 运算符.....	103
4.9 比较和逻辑运算符（单值运算符）.....	103
4.10 多值运算符 all、any	103
4.11 运算符的否定形式.....	103
五、 order by 子句.....	104
5.1 语法和执行顺序.....	104
5.2 升降序.....	104
5.3 null 值在排序中显示.....	104
5.4 order by 后面可以跟什么.....	104
5.5 多列排序.....	104
六、 单行函数的使用.....	105
6.1 数值类型.....	105
6.2 日期类型.....	105
6.3 字符类型.....	107
6.4 转换函数.....	108
6.5 其他注意事项.....	108
七、 SQL 语句中的分支.....	110
7.1 分支表达式.....	110
7.2 分支函数.....	110
八、 组函数.....	111
8.1 报表统计常用.....	111
8.2 缺省情况组函数处理什么值.....	111
8.3 当组函数要处理的所有值都为 null 时.....	111
8.4 行级信息和组级信息.....	111
九、 group by 子句.....	112
9.1 语法和执行顺序.....	112
9.2 分组过程.....	112
9.3 常见错误.....	112
9.4 多列分组.....	112
十、 having 子句.....	113
10.1 语法和执行顺序.....	113
10.2 执行过程.....	113
10.3 where 和 having 区别.....	113
十一、 非关联子查询.....	114
11.1 语法.....	114

11.2 子查询的执行过程.....	114
11.3 常见错误.....	114
11.4 子查询与空值.....	114
11.5 多列子查询.....	115
十二、 关联子查询.....	116
12.1 语法.....	116
12.2 执行过程.....	116
12.3 exists.....	116
12.4 exists 执行过程.....	116
12.5 not exists.....	117
12.6 not exists 执行过程.....	117
12.7 in 和 exists 比较.....	117
十三、 多表查询.....	118
13.1 按范式要求设计表结构.....	118
13.2 多表连接的种类.....	118
13.3 交叉连接.....	118
13.4 内连接.....	118
13.5 外连接.....	120
13.6 非等值连接.....	122
13.7 表连接总结.....	122
十四、 集合.....	123
14.1 表连接主要解决的问题.....	123
14.2 集合运算.....	123
14.3 集合运算符.....	123
14.4 子查询、连接、集合总结.....	124
十五、 排名分页问题.....	125
15.1 什么是 rownum.....	125
15.2 where rownum<=5 的执行过程.....	125
15.3 where rownum=5 的执行过程.....	125
十六、 约束 constraint.....	126
16.1 约束的类型.....	126
16.2 primary key: 主键约束.....	126
16.3 not null: 非空约束.....	126
16.4 unique key: 唯一建约束.....	126
16.5 references foreign key: 外键约束.....	127
16.6 check: 检查约束.....	129
十七、 事务.....	130
17.1 transaction.....	130
17.2 定义.....	130
17.3 事务的特性: ACID.....	130
17.4 事务的隔离级别.....	130
17.5 数据库开发的关键挑战.....	130
17.6 锁的概念.....	131
17.7 Oracle 的锁机制.....	131

17.8 事务不提交的后果.....	131
17.9 回滚事务 rollback.....	131
17.10 保留点 savepoint.....	131
十八、 数据库对象：视图 view.....	133
18.1 带子查询的 create table	133
18.2 带子查询的 insert.....	133
18.3 定义缺省值： default.....	133
18.4 视图 view.....	134
18.5 视图的应用场景.....	134
18.6 视图的分类.....	135
18.7 视图的维护.....	135
十九、 数据库对象：索引 index.....	137
19.1 创建 index.....	137
19.2 扫描表的方式.....	137
19.3 索引的结构.....	137
19.4 为什么要使用索引.....	138
19.5 哪些列适合建索引.....	138
19.6 索引的类型.....	138
19.7 哪些写法会导致索引用不了.....	139
二十、 数据库对象：序列号 sequence.....	140
20.1 什么是 sequence.....	140
20.2 创建 sequence.....	140
20.3 缺省是 nocycle (不循环)	140
20.4 缺省 cache 20.....	140
二十一、 其他注意事项.....	142
21.1 删除表，删除列，删除列中的值.....	142
21.2 多对多关系的实现.....	142
21.3 一对多（两张表）	142
21.4 一对一.....	142
21.5 数据库对象.....	142
21.6 缺省（默认）总结：	142

之三、PL/SQL (薛海璐老师主讲, 占笔记内容 100%);

目 录

一、 PL/SQL 简介.....	143
1.1 什么是 PL/SQL.....	143
1.2 PL/SQL 程序结构.....	143
1.3 PL/SQL 运行过程.....	143
1.4 注释.....	144
二、 变量与数据类型.....	145
2.1 数据类型.....	145
2.2 标量类型.....	145
2.3 变量声明.....	145

三、 流程控制语句.....	147
3.1 条件语句.....	147
3.2 循环语句.....	148
四、 PL/SQL 中的 SQL.....	151
4.1 PL/SQL 中的 SQL 分类.....	151
4.2 DML (insert, update, delete) 和 TCL (commit, rollback)	151
4.3 DDL.....	151
五、 PL/SQL 中的 select.....	153
5.1 select 语句的实现.....	153
5.2 record 类型.....	153
5.3 %rowtype.....	154
5.4 record 变量的引用.....	154
5.6 cursor 的分类.....	155
5.7 显式 cursor 的处理.....	155
5.8 显式 cursor 的属性.....	156
5.9 隐式 cursor 的属性.....	156
5.10 cursor 的声明.....	156
5.11 open cursor.....	156
5.12 fetch cursor.....	157
5.13 结果集提取的控制.....	157
5.14 close cursor.....	157
六、 集合.....	161
6.1 什么是 collection.....	161
6.2 什么是关联数组.....	161
6.3 Associative arrays 的定义.....	161
6.4 声明 Associative arrays 类型和变量.....	161
6.5 关联数组的操作.....	162
6.6 Associative arrays 的方法.....	162
6.7 关联数组的遍历.....	163
6.8 Associative arrays 的异常.....	164
6.9 批量绑定.....	164
七、 异常.....	166
7.1 Oracle 错误.....	166
7.2 Oracle 错误处理机制.....	166
7.3 异常的类型.....	166
7.4 PL/SQL 中的异常.....	166
7.5 异常捕获.....	166
7.6 异常的捕获规则.....	167
7.7 Oracle 预定义错误的捕获.....	167
7.8 非 Oracle 预定义异常.....	168
7.9 用户自定义异常.....	168
7.10 异常处理总结.....	169
7.11 sqlcode 和 sqlerrm.....	169
7.12 异常的传播.....	170

八、 子程序.....	171
8.1 子程序.....	171
8.2 有名子程序.....	171
8.3 有名子程序的分类.....	171
8.4 有名子程序的优点.....	171
九、 过程 procedure.....	172
9.1 语法.....	172
9.2 创建存储过程.....	172
9.3 形参和实参.....	172
9.4 形参的种类.....	173
9.5 调用存储过程.....	173
9.6 存储过程中的参数.....	173
9.7 对实际参数的要求.....	174
9.8 形式参数的限制.....	174
9.9 带参数的过程调用.....	174
9.10 使用缺省参数.....	174
9.11 存储过程中的 DDL 语句.....	175
9.12 变量.....	175
9.13 PL/SQL 中的 SQL 分类.....	176
9.14 再一次来看 SQL 语句的处理过程.....	178
9.15 软分析和硬分析.....	178
9.16 对过程 procedure 的基本操作.....	178
9.17 案例.....	178
十、 函数 function.....	180
10.1 语法.....	180
10.2 创建函数.....	180
10.3 调用函数.....	180
10.4 对函数 function 的基本操作.....	181
10.5 过程和函数的比较.....	181
10.6 匿名块中的过程和函数声明.....	181
10.7 案例.....	181
十一、 包 package.....	183
11.1 什么是 package.....	183
11.2 包的组成.....	183
11.3 包的优点.....	183
11.4 package 声明的语法.....	183
11.5 package body 声明的语法.....	184
11.6 编译包和包体.....	184
11.7 案例.....	184
十二、 触发器 trigger.....	186
12.1 面临问题.....	186
12.2 DML 触发器的组成.....	186
12.3 DML 触发器的类型.....	186
12.4 DML 触发器的触发顺序.....	186

12.5 DML 行级触发器.....	186
12.6 :OLD 和:NEW.....	186
12.7 触发器的重新编译.....	188
12.8 触发器的状态.....	188
十三、 其他注意事项.....	189
13.1 PL/SQL 的特点.....	189
13.2 写 PL/SQL 的好处.....	189
13.3 命名建议.....	189
13.4 搞清楚如下内容.....	189
13.5 保证所有对象的状态都是 valid.....	189
13.6 declare 中都可声明什么.....	189
13.7 数据库对象.....	189

之四、*JDBC* (范传奇老师主讲, 占笔记内容 100%);

目 录

一、 JDBC 概述.....	189
1.1 什么是 JDBC.....	189
1.2 什么是驱动.....	190
1.3 SQL lite.....	190
1.4 如何使用 Java 连接某种数据库.....	190
1.5 连接数据库并操作.....	190
1.6 连接数据库时常见的错误.....	190
二、 JDBC 核心 API.....	191
2.1 Connection.....	191
2.2 Statement.....	191
2.3 ResultSet.....	191
2.4 DriverManager.....	191
2.5 UUID.....	192
2.6 案例: 使用 JDBC 连接数据库, 并操作 SQL 语句.....	192
2.7 案例: 通过 JDBC 创建表.....	193
2.8 案例: 使用 JDBC 向表中插入数据.....	193
2.9 遍历 Student_chang 表.....	194
三、 JDBC 核心 API: PreparedStatement.....	195
3.1 Statement 的缺点.....	195
3.2 PreparedStatement 的优点.....	195
3.3 PreparedStatement 的常用方法.....	195
3.4 案例详见第五章 StudentDAO 类.....	196
四、 Connection 封装.....	197
五、 DAO.....	198
5.1 持久类封装.....	198
5.2 DAO 层.....	198
5.3 Properties 类.....	198

5.4 案例：注册系统.....	198
六、 批处理.....	202
6.1 批处理的优点.....	202
6.2 JDBC 批处理 API.....	202
6.3 案例：详见 8.4 案例 step7.....	202
七、 事务处理.....	203
7.1 事务特性 ACID.....	203
7.2 JDBC 中对事务的支持（API）.....	203
八、 DAO 事务封装.....	204
8.1 ThreadLocal 原理.....	204
8.2 原理图.....	204
8.3 ThreadLocal 核心 API.....	204
8.4 案例：登录系统（使用 ThreadLocal 实现连接共享）.....	204
九、 分页查询.....	208
9.1 分页查询的基本原理.....	208
9.2 为何使用分页查询.....	208
9.3 Oracle 分页查询 SQL 语句.....	208
9.4 MySQL 分页查询 SQL 语句.....	208
9.5 “假” 分页.....	208
9.6 案例：分页查询.....	209

之五、XML（范传奇老师主讲，占笔记内容 100%）；

目 录

一、 XML 基本语法.....	211
1.1 XML 介绍.....	211
1.2 XML 元素.....	211
1.3 XML 属性.....	211
1.4 实体引用.....	211
1.5 CDATA 段.....	212
1.6 DTD 声明元素.....	213
1.7 DTD 声明元素：声明空元素.....	213
1.8 DTD 声明元素：含有 PCDATA.....	213
1.9 DTD 声明元素：带有子元素（子元素列表）的元素.....	213
1.10 DTD 声明元素：声明只出现一次的元素.....	214
1.11 DTD 声明元素：声明可多次出现的元素.....	214
1.12 DTD 声明元素：子元素只能是其中之一的情况.....	214
1.13 DTD 声明元素：子元素可以是元素也可以是文本.....	215
1.14 DTD 声明元素：总结.....	215
1.15 DTD 中声明元素的属性.....	215
1.16 属性类型.....	215
1.17 属性值的约束.....	215
1.18 DTD 命名空间介绍.....	216
二、 Schema 简介.....	216

2.1 Schema 的作用.....	216
2.2 Schema 文件的扩展名 xsd.....	216
三、 Java 解析 XML.....	217
3.1 Java 与 XML 共同点.....	217
3.2 Java 解析 XML 有两种方式.....	217
3.3 JDOM/DOM4J.....	217
3.4 DOM 解析.....	217
3.5 SAX 解析.....	217
3.6 案例：使用 DOM4J 包的核心 API 解析 xml 文件.....	217
3.7 案例：使用 DOM4J 包的核心 API 写入 xml 文件.....	219
四、 XPath 语言.....	221
4.1 XPath 基本介绍.....	221
4.2 使用 XPath 的好处.....	221
4.3 XPath 基本语法.....	221
4.4 DOM4J 对 XPath 的支持.....	221
五、 附 db_info.xml 文件.....	222

之六、HTML（王春梅老师主讲，占笔记内容 100%）；

目 录

一、 HTML 概述.....	223
1.1 什么是 HTML.....	223
1.2 Web 浏览器.....	223
二、 HTML 基础语法.....	224
2.1 标记语法.....	224
2.2 封闭类型标记：双标记.....	224
2.3 非封闭类型标记：单标记或者空标记.....	224
2.4 元素和属性.....	224
2.5 注释.....	224
2.6 HTML 文档的标准结构.....	225
2.7 版本信息.....	225
2.8 <head>元素.....	225
2.9 <body> 元素.....	226
2.10 头元素：<title>.....	226
2.11 头元素：<meta>.....	226
2.12 案例：创建一个标准结构的 HTML 文档，并创建头元素。.....	226
三、 文本标记.....	227
3.1 文本标记的作用.....	227
3.2 文本于特数字符.....	227
3.3 标题元素<hn>.....	227
3.4 段落元素<p>.....	227
3.5 换行元素	227
3.6 分区元素和<div>.....	227

3.7 块级元素（block）和行内元素（inline）	227
3.8 案例：使用文本标记为页面添加内容.....	228
四、 图像和连接.....	229
4.1 图像元素.....	229
4.2 链接元素<a>.....	229
4.3 URL.....	229
4.4 锚点.....	229
4.5 案例：使用图像和链接标记.....	230
五、 列表标记.....	231
5.1 列表的作用.....	231
5.2 无序列表.....	231
5.3 有序列表.....	231
5.4 列表的嵌套.....	231
5.5 案例：使用列表标记添加导航目录.....	231
六、 表格.....	232
6.1 表格的作用.....	232
6.2 创建表格.....	232
6.3 表格的常用属性.....	232
6.4 单元格的常用属性.....	232
6.5 表格的标题<caption>.....	232
6.6 行分组（表格特有的）	233
6.7 不规则表格.....	233
6.8 表格的嵌套.....	233
6.9 案例：实现如下图所示表格.....	233
七、 表单.....	234
7.1 表单的作用.....	234
7.2 表单元素<form>.....	234
7.3 <input> 元素.....	234
7.4 文本框与密码框.....	234
7.5 单选框和多选框.....	235
7.6 按钮.....	235
7.7 隐藏域和文件选择框.....	235
7.8 <label> 元素.....	235
7.9 选项框.....	235
7.10 <textarea> 元素.....	236
7.11 表单元素分组（表单元素特有的）	236
7.12 案例：创建复杂表单.....	236
八、 框架.....	237
8.1 浮动框架的作用.....	237
8.2 <iframe> 元素.....	237
九、 其他注意事项.....	237
9.1 Web 应用程序的层次结构.....	238
9.2 界面层中根据类型应用程序的分类.....	238
9.3 单机应用和 Web 应用.....	238

9.4 Web 类型的应用程序.....	238
9.5 本课程分为：	238
9.6 何为 Web 基础.....	238

之七、CSS (王春梅老师主讲, 占笔记内容 100%);

目 录

一、 CSS 概述.....	238
1.1 CSS 的作用.....	238
1.2 什么是 CSS.....	238
1.3 CSS 的基础语法.....	239
二、 如何使用 CSS 样式表.....	240
2.1 内联样式.....	240
2.2 内部样式表.....	240
2.3 外部样式表.....	240
2.4 三种用法的区别.....	240
2.5 CSS 样式表特征和优先级.....	241
三、 CSS 选择器.....	242
3.1 元素选择器.....	242
3.2 类选择器.....	242
3.3 分类选择器.....	242
3.4 元素 id 选择器.....	242
3.5 派生选择器.....	242
3.6 选择器分组.....	243
3.7 伪类选择器.....	243
3.8 选择器优先级.....	243
四、 CSS 单位.....	243
4.1 尺寸.....	243
4.2 颜色.....	243
4.3 尺寸属性.....	243
五、 边框样式.....	244
5.1 简写方式.....	244
5.2 单边定义.....	244
5.3 单边宽度 border-width.....	244
5.4 单边样式 border-style.....	244
5.5 单边颜色 border-color.....	244
5.6 案例.....	244
六、 框模型.....	244
6.1 框模型图.....	244
6.2 width 和 height.....	244
6.3 边框、内外边距对元素尺寸的影响.....	245
6.4 案例.....	245
6.5 内边距.....	245

6.6 外边距.....	245
七、 背景样式.....	246
7.1 背景色.....	246
7.2 背景图像.....	246
7.3 背景平铺.....	246
7.4 背景固定.....	246
7.5 背景定位.....	246
7.6 组合设置.....	246
7.7 案例.....	246
八、 文本/字体样式.....	247
8.1 指定字体.....	247
8.2 字体颜色.....	247
8.3 字体大小.....	247
8.4 字体加粗.....	193
8.5 文本排列.....	248
8.6 行高.....	248
8.7 文字修饰.....	248
8.8 文本缩进.....	248
九、 表格样式.....	232
9.1 垂直对齐.....	250
9.2 边框合并.....	250
9.3 边框边距.....	250
十、 布局.....	251
10.1 浮动定位说明图.....	251
10.2 什么是浮动定位.....	251
10.3 浮动定位移动图.....	251
10.4 float 属性.....	251
10.5 clear 属性.....	252
10.6 display 属性.....	252
十一、 列表样式.....	253
11.1 列表项前的标识符号图像.....	253
11.2 列表项前使用的预设标识符号.....	253
十二、 定位.....	253
12.1 定位概述.....	253
12.2 position (定位) 属性.....	253
12.3 偏移属性.....	253
12.4 堆叠属性.....	254
12.5 相对定位: relative.....	254
12.6 绝对定位: absolute.....	254

之八、JavaScript (王春梅老师主讲, 占笔记内容 100%);

目 录

一、 JavaScript 概述.....	254
1.1 什么是 JavaScript.....	254
1.2 JavaScript 发展史.....	255
1.3 JavaScript 的特点.....	255
1.4 JavaScript 的定义方式.....	255
1.5 JavaScript 的代码错误查看.....	255
1.6 注释.....	256
二、 JavaScript 基础语法.....	257
2.1 编写 JavaScript 代码.....	257
2.2 常量、标识符和关键字.....	257
2.3 变量.....	257
2.4 数据类型.....	257
2.5 string 数据类型.....	257
2.6 number 数据类型.....	257
2.7 boolean 数据类型.....	257
2.8 数据类型的隐式转换.....	258
2.9 数据类型转换函数.....	258
2.10 特殊类型.....	259
2.11 算术运算.....	259
2.12 关系运算.....	259
2.13 逻辑运算.....	259
2.14 条件运算符.....	259
2.15 流程控制语句.....	260
三、 JavaScript 常用内置对象.....	261
3.1 什么是 JavaScript 对象.....	261
3.2 使用对象.....	261
3.3 常用内置对象.....	261
3.4 String 对象.....	261
3.5 String 对象与正则表达式.....	262
3.7 Math 对象.....	264
3.8 Number 对象.....	265
3.9 RegExp 正则表达式对象.....	265
3.10 Date 对象.....	266
3.11 函数与 Function 对象.....	266
3.12 全局函数.....	267
3.13 Arguments 对象.....	268
四、 window 对象.....	269
4.1 DHTML 简介.....	269
4.2 DHTML 对象模型.....	269
4.3 window 对象.....	269
4.4 常用方法：对话框.....	269
4.5 常用方法：窗口的打开和关闭.....	270
4.6 常用方法：周期性定时器.....	270
4.7 常用方法：一次性定时器.....	270

4.8 案例：动态时钟.....	270
五、 Document 对象与 DOM.....	271
5.1 概念.....	271
5.2 根据元素 ID 查找节点.....	271
5.3 根据层次查找节点.....	271
5.4 根据标签名称查找节点.....	272
5.5 读取或者修改节点信息.....	272
5.6 修改节点的样式.....	272
5.7 查找并修改节点.....	272
5.8 三个案例.....	273
5.9 增加新节点.....	275
5.10 删除节点.....	275
5.11 案例：联动菜单.....	275
六、 HTML DOM.....	277
6.1 HTML DOM 概述.....	277
6.2 Select 对象.....	277
6.3 Option 对象.....	277
6.4 案例：联动菜单（HTML DOM 方式）.....	277
6.5 Table 对象.....	278
6.6 TableRow 对象.....	278
6.7 TableCell 对象.....	278
6.8 案例：产品列表.....	278
七、 window 其他子对象（DHTML 模型）.....	280
7.1 screen 对象.....	280
7.2 history 对象.....	280
7.3 location 对象.....	280
7.4 navigator 对象.....	280
7.5 事件.....	280
7.6 event 对象.....	281
八、 面向对象基础.....	283
8.1 属性.....	283
8.2 方法.....	283
8.3 定义对象的三种方式.....	283
8.4 创建通用对象.....	283
8.5 创建对象的模版.....	283
九、 JSON.....	284
9.1 JSON 概述.....	284
9.2 名称可以是属性.....	284
9.3 名称也可以是方法.....	284
9.4 案例：批量提交数据和下拉框版式日历.....	284

之九、Servlet（程祖红老师主讲，占笔记内容 100%）；

目 录

一、 Servlet 概述.....	287
1.1 B/S 架构（了解）.....	287
1.2 什么是 Servlet.....	287
1.3 什么是 Tomcat.....	234
1.4 如何写一个 Servlet（不使用开发工具）.....	289
1.5 使用 MyEclipse 开发 Servlet.....	291
1.6 Servlet 是如何运行的.....	292
1.7 常见错误及解决方式.....	292
1.8 案例：根据请求次数显示结果和显示当前时间.....	293
二、 HTTP 协议.....	294
2.1 什么是 HTTP 协议.....	294
2.2 通讯的过程.....	294
2.3 数据格式.....	294
2.4 TCP/IP Monitor 的使用.....	295
2.5 get 请求与 post 请求.....	295
2.6 如何读取请求参数.....	295
2.7 访问数据库（ MySql ）.....	296
2.8 案例：添加员工（访问 MySql 数据库）.....	296
2.9 异常： IllegalStateException	297
三、 编码问题.....	299
3.1 Java 语言在内存当中默认使用的字符集.....	299
3.2 编码.....	299
3.3 解码.....	299
3.4 Servlet 如何输出中文.....	299
3.6 案例：根据请求正确显示中文.....	299
3.7 将中文数据插入到数据库.....	300
四、 重定向.....	301
4.1 什么是重定向.....	202
4.2 如何重定向.....	301
4.3 注意两个问题.....	301
4.4 两个特点.....	301
4.5 重定向原理图：以 2.8 案例为例.....	301
五、 DAO.....	302
5.1 什么是 DAO.....	302
5.2 如何写一个 DAO.....	302
5.3 工厂类.....	302
六、 工厂设计模式.....	303
6.1 什么是工厂.....	303
6.2 使用工厂设计模式的好处.....	303
6.3 如何使用工厂模式.....	303
6.4 案例：为 2.8 案例添加新功能，并使用 DAO 和工厂模式.....	303
七、 Servlet 容器处理细节.....	307
7.1 Servlet 容器如何处理请求资源路径以及匹配.....	307
7.2 一个 Servlet 如何处理多种请求.....	307

八、 Servlet 的生命周期.....	308
8.1 Servlet 的生命周期的含义.....	308
8.2 Servlet 生命周期的四个阶段.....	308
8.3 实例化.....	308
8.4 初始化.....	308
8.5 就绪.....	308
8.6 销毁.....	309
8.7 Servlet 生命周期图.....	309
8.8 Servlet 生命周期相关的几个接口与类.....	309
九、 JSP (简要介绍, 详细内容见 JSP 笔记)	311
9.1 什么是 JSP.....	311
9.2 为什么要使用 JSP.....	311
9.3 JSP 与 Servlet 的关系.....	311
9.4 如何写一个 JSP 文件.....	311
9.5 JSP 是如何运行的.....	311
9.6 指令.....	312
9.7 案例: 创建 empList.jsp 页面, 将表示逻辑交给 JSP 处理.....	312
十、 请求转发.....	314
10.1 什么是转发.....	314
10.2 如何转发.....	314
10.3 编程需要注意的两个问题.....	314
10.4 转发原理图.....	314
10.5 转发的特点.....	315
10.6 转发和重定向的区别.....	315
10.7 何时用重定向.....	315
10.8 何时用转发.....	315
10.9 案例: 修改 6.4 案例中 step7 中的 ListEmpServlet.java.....	316
十一、 异常的处理.....	318
11.1 用输出流 out.....	318
11.2 用转发的方式.....	318
11.3 让容器处理系统异常.....	318
11.4 案例: 将 10.9 案例中的 step3 中的所有 catch 块修改.....	318
十二、 路径问题.....	319
12.1 什么是相对路径.....	319
12.2 什么是绝对路径.....	319
12.3 如何写绝对路径.....	319
12.4 如何防止硬编码.....	319
12.5 案例: 相对、绝对路径对比.....	319
12.6 四种情况下, 正确的绝对路径写法.....	320
十三、 状态管理.....	321
13.1 什么是状态管理.....	321
13.2 如何进行状态管理.....	321
13.3 cookie.....	321
13.4 如何创建一个 cookie.....	321

13.5 查询 cookie.....	321
13.6 编码问题.....	322
13.7 cookie 的生存时间.....	322
13.8 cookie 的路径问题.....	322
13.9 cookie 的限制.....	323
13.10 案例：写一个 CookieUtil.....	323
13.11 session (会话)	324
13.12 如何创建一个 session 对象.....	324
13.13 HttpSession 接口中提供的常用方法.....	324
13.14 session 的超时.....	325
13.15 用户禁止 cookie 以后，如何继续使用 session.....	325
13.16 url 重写.....	325
13.17 session 的优点.....	325
13.18 session 的缺点.....	326
13.19 案例：session 验证和访问次数.....	326
13.20 案例：验证码.....	327
13.21 案例：购物车.....	329
十四、 过滤器.....	333
14.1 什么是过滤器.....	333
14.2 如何写一个过滤器.....	333
14.3 案例：敏感字过滤器.....	333
14.4 过滤器的优先级.....	334
14.5 初始化参数.....	335
14.6 优点.....	335
十五、 监听器.....	336
15.1 什么是监听器.....	336
15.2 容器会产生两大类事件.....	336
15.3 如何写一个监听器.....	336
15.4 ServletContext (Servlet 上下文)	253
15.5 如何获得 Servlet 上下文.....	336
15.6 Servlet 上下文的作用.....	336
15.7 案例：统计在线人数.....	337
十六、 上传文件.....	339
16.1 如何上传文件.....	339
16.2 案例：上传文件.....	339
十七、 Servlet 线程安全问题.....	341
17.1 为何 Servlet 会有线程安全问题.....	341
17.2 如何处理线程安全问题.....	341
十八、 Servlet 小结.....	342
18.1 Servlet 基础.....	342
18.2 Servlet 核心.....	342
18.3 状态管理.....	342
18.4 数据库访问.....	342
18.5 过滤器和监听器.....	342

18.6 典型案例和扩展.....	342
十九、 其他注意事项.....	343
19.1 连接数据库的工具.....	343
19.2 知名网站.....	343
19.3 C/S 架构: Client/Server.....	343
19.4 B/S 架构: Browser/Server.....	344

之十、 JSP (程祖红老师主讲, 占笔记内容 100%);

目 录

一、 JSP 基础.....	344
1.1 什么是 JSP.....	344
1.2 为什么要使用 JSP.....	344
1.3 JSP 与 Servlet 的关系.....	344
1.4 如何写一个 JSP 文件.....	344
1.5 JSP 是如何运行的.....	345
1.6 隐含对象.....	345
1.7 指令.....	346
1.8 JSP 注释.....	347
1.9 案例: 创建 emplist.jsp 页面, 将表示逻辑交给 JSP 处理.....	347
二、 JSP 标签和 EL 表达式.....	348
2.1 什么是 JSP 标签.....	348
2.2 JSTL 及其使用.....	348
2.3 什么是 EL 表达式.....	348
2.4 EL 表达式的使用.....	348
三、 JSTL 中的几个核心标签.....	351
3.1 if.....	351
3.2 choose.....	351
3.3 forEach.....	351
3.4 url.....	352
3.5 set.....	352
3.6 remove.....	352
3.7 catch.....	352
3.8 import.....	352
3.9 redirect.....	352
3.10 out.....	352
3.11 JSP 标签是如何运行的.....	352
3.12 案例: 将员工列表中的 Java 代码改为 JSP 标签, 并添加分页.....	353
四、 自定义标签.....	354
4.1 如何写一个自定义标签.....	354
4.2 JavaEE5.0 中, 如何使用 EL 表达式和 JSTL.....	354
4.3 案例: 自定义标签.....	354
4.4 案例: 修改之前员工列表中的日期显示.....	355

五、 MVC.....	357
5.1 什么是 MVC.....	357
5.2 使用 MVC 的目的.....	357
5.3 如何使用 MVC 来开发一个 Web 应用程序（JavaEE）	357
5.4 MVC 的优缺点.....	357
5.5 案例：简易贷款（贷款数小于余额数*10）	357
5.6 修改 5.5 案例，使用户无法直接访问 view.jsp 页面.....	359

之十一、Ajax (程祖红老师主讲, 占笔记内容 100%);

目 录

一、 Ajax 概述.....	360
1.1 什么是 Ajax.....	360
1.2 Ajax 对象：如何获得 Ajax 对象.....	360
1.3 Ajax 对象的属性.....	357
1.4 编程步骤.....	361
1.5 编码问题.....	362
1.6 Ajax 的优点.....	363
1.7 缓存问题（IE 浏览器）	363
1.8 案例：简易注册（使用 Ajax 进行相关验证，get 请求）	363
1.9 案例：修改 1.8 案例，使用 post 请求.....	365
1.10 案例：使用 Ajax 实现下拉列表.....	365
二、 JSON.....	367
2.1 什么是 JSON.....	367
2.2 数据交换.....	367
2.3 轻量级.....	367
2.4 JSON 语法（www.json.org）	367
2.5 如何使用 JSON 来编写 Ajax 应用程序.....	368
2.6 案例：股票的实时行情.....	369
2.7 案例：显示热卖的前 3 个商品.....	370
2.8 同步请求.....	371
2.9 案例：修改 1.8 案例 step1 中的 JS 代码（使用同步请求）	371

之十二、jQuery (程祖红老师主讲, 占笔记内容 100%);

目 录

一、 jQuery 基础.....	372
1.1 jQuery 的特点.....	372
1.2 jQuery 编程的步骤.....	373
1.3 jQuery 对象与 DOM 对象如何相互转换.....	373
1.4 如何同时使用 prototype 和 jQuery.....	373
1.5 EL 表达式和 jQuery 函数的区别.....	373

二、 选择器.....	374
2.1 什么是选择器.....	374
2.2 基本选择器.....	374
2.3 层次选择器.....	374
2.4 基本过滤选择器.....	375
2.5 内容过滤选择器.....	302
2.6 可见性过滤选择器.....	376
2.7 属性过滤选择器.....	376
2.8 子元素过滤选择器.....	376
2.9 表单对象属性过滤选择器.....	377
2.10 表单选择器.....	377
三、 DOM 操作.....	378
3.1 查询.....	378
3.2 创建.....	378
3.3 插入节点.....	378
3.4 删除节点.....	378
3.5 如何将 JavaScript 代码与 HTML 分开.....	379
3.6 复制节点.....	379
3.7 属性.....	379
3.8 样式操作.....	380
3.9 遍历节点.....	380
3.10 案例：员工列表（点击某行该行加亮，多选框被选中）.....	381
3.11 案例：员工列表（点击部门隐藏或显示员工）.....	382
四、 事件.....	384
4.1 事件绑定.....	384
4.2 合成事件.....	384
4.3 事件冒泡 可参考 JavaScript 笔记 7.5.....	384
4.4 jQuery 中事件处理.....	385
4.5 动画.....	386
4.6 类数组的操作.....	386
4.7 案例：滚动广告条.....	387
五、 jQuery 对 Ajax 编程的支持.....	389
5.1 load()方法.....	389
5.2 案例：显示机票价格.....	389
5.3 \$.get()方法.....	389
5.4 \$.post()方法.....	390
5.5 案例：修改 Ajax 笔记中 2.6 案例：股票的实时行情.....	390
5.6 \$.ajax()方法.....	390
5.7 案例：搜索栏联想效果（服务器返回 text）.....	390
5.8 案例：下拉列表（服务器返回 xml 文本）.....	392
5.9 案例：表单验证.....	393
5.10 jQuery 的自定义方法.....	396
5.11 \$.param()方法.....	396
5.12 案例：自定义方法和\$.param()方法使用（学了 Struts2 再看）.....	396

之十三、Struts2 (李翊老师主讲, 占笔记内容 100%);

目 录

一、 Struts2 概述.....	397
1.1 为什么要用 Struts.....	397
1.2 什么是 MVC.....	397
1.4 Struts2 发展史.....	398
1.5 衡量一个框架的标准.....	398
1.6 Struts2 使用步骤.....	398
1.7 struts.xml 内容详解.....	400
1.8 Struts2 提供的方便之处.....	401
1.9 案例：简单登录（使用 Strut2）.....	401
1.10 案例：修改 1.6、1.9 案例使用户不能绕过前端控制器.....	402
1.11 NetCTOSS 项目：显示资费列表.....	403
1.12 NetCTOSS 项目：资费列表分页显示.....	407
二、 OGNL 技术.....	410
2.1 什么是 OGNL.....	410
2.2 OGNL 基本语法.....	410
2.3 OGNL 表达式中加 “#” 和不加 “#” 的区别.....	411
2.4 OGNL 体系结构.....	412
2.5 XWord 框架对 OGNL 进行了改造.....	412
2.6 ValueStack 对象结构.....	412
2.7 ValueStack 结构演示.....	413
2.8 Struts2 标签的使用.....	414
2.9 Struts2 对 EL 表达式的支持.....	414
2.10 案例：修改 1.12 案例（使用 Struts2 标签和 OGNL 表达式）.....	415
三、 Action.....	416
3.1 Struts2 的核心组件.....	416
3.2 Struts2 的工作流程.....	416
3.3 在 Action 中访问 Session 和 Application.....	416
3.4 NetCTOSS 项目：用户登录.....	417
3.5 Action 属性注入.....	420
3.6 案例：重构 NetCTOSS 资费列表分页显示（使用属性注入）.....	420
3.7 使用通配符配置 Action.....	491
3.8 案例：通配符配置（资费增、改、查）.....	421
3.9 Struts2 中 Action 的设计经验.....	421
四、 Result.....	422
4.1 Result 注册配置.....	422
4.2 Result 组件利用<result>元素的 type 属性指定 result 类型.....	422
4.3 常见的 Result 组件类型.....	422
4.4 NetCTOSS 项目：资费删除.....	422
4.5 NetCTOSS 项目：基于 StreamResult 生成验证码.....	424

4.6 NetCTOSS 项目：基于 JsonResult 进行验证码检验.....	426
4.7 NetCTOSS 项目：添加资费模块中的验证资费名是否重复.....	428
4.8 自定义一个 Result.....	430
五、 Struts2 标签.....	432
5.1 A 开头的标签.....	432
5.2 B 开头的标签.....	432
5.3 C 开头的标签.....	432
5.4 D 开头的标签.....	432
5.5 E 开头的标签.....	432
5.6 F 开头的标签.....	432
5.7 G 开头的标签.....	433
5.8 H 开头的标签.....	433
5.9 I 开头的标签.....	433
5.10 L 开头的标签.....	433
5.11 M 开头的标签.....	433
5.12 O 开头的标签.....	433
5.13 P 开头的标签.....	433
5.14 R 开头的标签.....	433
5.15 S 开头的标签.....	433
5.16 T 开头的标签.....	434
5.18 所有标签都具备的属性.....	434
5.19 案例：常用标签.....	434
六、 拦截器.....	437
6.1 Struts2 详细流程图.....	437
6.2 拦截器的作用.....	437
6.3 拦截器的常用方法.....	437
6.4 自定义拦截器步骤.....	437
6.5 Struts2 内置的拦截器.....	438
6.6 案例：拦截器入门.....	439
6.7 拦截器栈.....	442
6.8 fileUpload 拦截器原理.....	442
6.9 案例：使用 fileUpload 拦截器实现文件上传.....	442
6.10 NetCTOSS 项目：登录检查拦截器.....	444
七、 Struts2 中如何处理异常.....	446
7.1 异常一般出现在何处.....	446
7.2 如何配置异常.....	446
八、 Struts2 中如何实现国际化.....	448
8.1 i18n.....	448
8.2 如何获得中文的 Unicode 编码.....	448
8.3 浏览器如何决定用哪个资源文件.....	448
8.4 资源文件的命名.....	448
8.5 资源文件的分类.....	448
8.6 实现国际化的步骤.....	448
九、 NetCTOSS 项目.....	450

9.1 DAO 优化、重构、封装!【重要】	450
9.2 资费更新.....	452
9.3 导航条.....	454
十、 项目经验.....	456
10.1 主键用 int 还是 Integer.....	456
10.2 “./” 表示的意思.....	456
10.3 导入静态页面，样式、JS 失效问题.....	456
10.4 <s:hidden>和<s:textarea>标签.....	456
10.5 四种情形下的绝对路径写法.....	456
10.6 URL 和 URI.....	456
10.7 util.Date 和 sql.Date.....	456

之十四、 Hibernate (梁建全老师主讲，占笔记内容 100%);

目 录

一、 Hibernate 的概述.....	457
1.1 Hibernate 框架的作用.....	457
1.2 Hibernate 访问数据库的优点.....	457
1.3 JDBC 访问数据库的缺点.....	457
1.4 Hibernate 的设计思想.....	457
二、 Hibernate 的基本使用.....	458
2.1 Hibernate 的主要结构.....	458
2.2 Hibernate 主要的 API.....	458
2.3 Hibernate 使用步骤.....	458
2.4 HQL 语句 (简要介绍)	463
三、 数据映射类型.....	464
3.1 映射类型的作用.....	464
3.2 type 映射类型的两种写法.....	464
四、 Hibernate 主键生成方式.....	465
4.1 五种生成方式.....	465
五、 Hibernate 基本特性.....	466
5.1 对象持久性.....	466
5.2 处于持久状态的对象具有的特点.....	466
5.3 三种状态下的对象的转换.....	466
5.4 批量操作：注意及时清除缓存.....	466
5.5 案例：三种状态下的对象使用.....	467
5.6 一级缓存机制（默认开启）	467
5.7 一级缓存的好处.....	467
5.8 管理一级缓存的方法.....	468
5.9 延迟加载机制.....	468
5.10 具有延迟加载机制的操作.....	468
5.11 常犯的错误.....	469
5.12 延迟加载的原理.....	469

5.13 Session 的 get 和 load 方法的区别.....	469
5.14 延迟加载的好处.....	469
5.15 案例：测试延迟加载.....	470
5.16 案例：重构 NetCTOSS 资费管理模块.....	470
5.17 Java Web 程序中如何用延迟加载操作（OpenSessionInView）.....	472
六、 关联映射.....	475
6.1 一对多关系 one-to-many.....	475
6.2 多对一关系 many-to-one.....	476
6.3 多对多关联映射 many-to-many.....	476
6.4 关联操作（查询 join fetch/级联 cascade）.....	478
6.5 继承关系映射.....	481
七、 Hibernate 查询方法.....	484
7.1 HQL 查询.....	484
7.2 HQL 和 SQL 的相同点.....	484
7.3 HQL 和 SQL 的不同点.....	484
7.4 HQL 典型案例.....	484
7.5 Criteria 查询.....	487
7.6 Native SQL 原生 SQL 查询.....	488
八、 Hibernate 高级特性.....	489
8.1 二级缓存.....	489
8.2 二级缓存开启方法及测试.....	489
8.3 二级缓存管理方法.....	490
8.4 二级缓存的使用环境.....	490
8.5 查询缓存.....	490
8.6 查询缓存开启方法及测试.....	490
8.7 查询缓存的使用环境.....	491
九、 Hibernate 锁机制.....	492
9.1 悲观锁.....	492
9.2 悲观锁的实现原理.....	492
9.3 悲观锁使用步骤及测试.....	492
9.4 乐观锁.....	493
9.5 乐观锁的实现原理.....	493
9.6 乐观锁使用步骤及测试.....	493
十、 其他注意事项.....	495
10.1 源码服务器管理工具.....	495
10.2 利用 MyEclipse 根据数据表自动生成实体类、 hbm.xml.....	495
10.3 根据实体类和 hbm.xml 生成数据表.....	496
10.4 Hibernate 中分页查询使用 join fetch 的缺点.....	104
10.5 Hibernate 的子查询映射.....	497

之十五、Spring (梁建全老师主讲，占笔记内容 100%);

目 录

一、 Spring 概述.....	498
1.1 Spring 框架的作用.....	498
1.2 Spring 框架的优点.....	498
1.3 Spring 框架的容器.....	498
二、 Spring 容器的基本应用.....	499
2.1 如何将一个 Bean 组件交给 Spring 容器.....	499
2.2 如何获取 Spring 容器对象和 Bean 对象.....	499
2.3 如何控制对象创建的模式.....	499
2.4 Bean 对象创建的时机.....	499
2.5 为 Bean 对象执行初始化和销毁方法.....	499
2.6 案例： Spring 框架的使用以及 2.1 节-2.5 节整合测试.....	500
三、 Spring 框架 IoC 特性.....	502
3.1 IoC 概念.....	502
3.2 DI 概念.....	502
3.3 案例： 测试 IoC (set 注入)	502
3.4 案例： 测试 IoC (构造注入)	503
3.5 案例： 不用 JDBC 访问数据库，而是采用 Hibernate 访问.....	503
四、 Spring 中各种类型的数据注入.....	504
4.1 Bean 对象注入.....	504
4.3 集合的注入.....	504
4.4 案例： 各类数据注入.....	504
五、 AOP 概念.....	507
5.1 什么是 AOP.....	507
5.3 AOP 相关术语.....	507
5.4 案例： AOP 的使用，模拟某些组件需要记录日志的功能.....	508
5.5 通知类型.....	508
5.6 切入点.....	509
5.7 案例： 环绕通知，修改 5.4 案例使之动态显示所执行的操作.....	509
5.8 案例： 利用 AOP 实现异常处理，将异常信息写入文件.....	510
六、 Log4j 日志记录工具.....	511
6.1 Log4j 介绍.....	365
6.2 Log4j 的使用.....	511
6.3 案例： 修改 5.8 案例，使用 Log4j 记录日志.....	511
七、 Spring 注解配置.....	513
7.1 组件扫描功能.....	513
7.2 组件扫描的使用方法.....	513
7.3 注入注解标记使用方法.....	514
7.4 AOP 注解标记使用方法.....	514
八、 Spring 对数据访问技术的支持.....	516
8.1 Spring 提供了统一的异常处理类型.....	516
8.2 Spring 提供了编写 DAO 的支持类.....	516
8.3 Spring 提供了声明式事务管理方法.....	516
8.4 Spring 框架如何使用 JDBC 技术.....	516
8.5 连接池优点.....	519

8.6 Spring 框架如何使用 Hibernate 技术.....	519
8.7 Spring+Hibernate 如何使用 Session、Query 等对象.....	522
8.8 Spring 框架和 Struts2 整合应用.....	522
8.9 案例：采用 SSH 结构重构资费管理模块.....	524
九、 整合开发包 struts-spring-plugin.jar.....	529
9.1 Struts2 创建对象的方式.....	529
9.2 struts-spring-plugin.jar 创建对象的方式.....	529
9.3 struts-spring-plugin.jar 的内部实现.....	529
9.4 原理图 1.....	529
9.5 原理图 2.....	530
9.6 注意事项.....	530
9.7 注入规则.....	530
十、 Spring 的事务管理.....	531
10.1 声明式事务管理（基于配置方式实现事务控制）.....	186
10.2 编程式事务管理（基于 Java 编程实现事务控制），不推荐用！.....	532
10.3 Spring 中常用的事务类型.....	532
十一、 Spring 的 MVC.....	534
11.1 Spring MVC 的体系结构.....	534
11.2 Spring MVC 的工作流程.....	534
11.3 案例：简易登录（基于 XML 配置，不推荐使用）.....	534
11.4 案例：修改 11.3 案例（基于注解配置，推荐使用）.....	536
十二、 其他注意事项.....	537
12.1 Spring 的核心模块.....	538
12.2 表单中 action 属性的相对、绝对路径问题.....	538
12.3 用 SSH 重构 NetCTOSS 项目模块的步骤.....	538

1-JAVA 基础学习笔记

一、Java 技术基础

1.1 编程语言

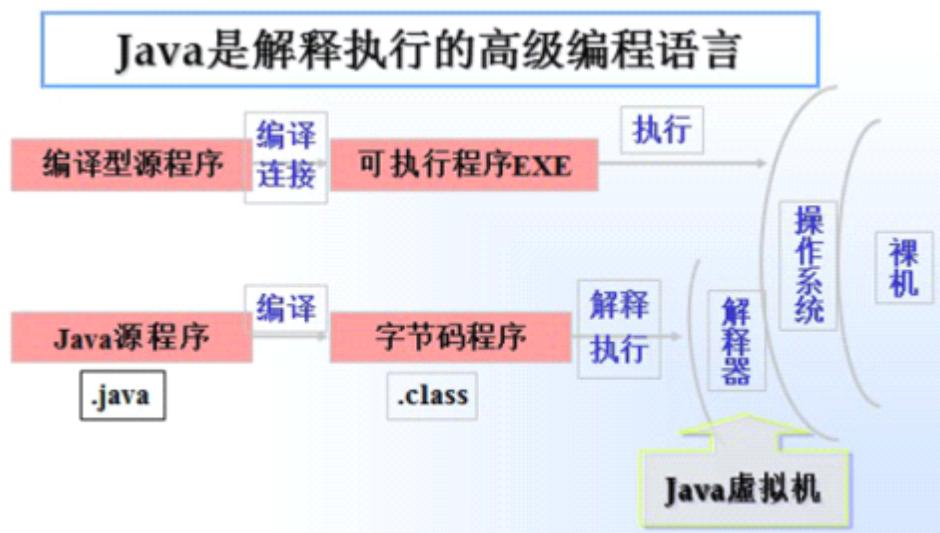
机器语言：0 1 在硬件直接执行

汇编语言：助记符

高级语言： (Java 运行比 C/C++ 慢)

1) 面向过程的高级语言：程序设计的基本单位为函数，如：C/C++语言。

2) 面向对象的高级语言：程序设计的基本单位为类，如：Java、C#。



1.2 Java 的特点

平台无关性、简单性、面向对象、健壮性、多线程、自动内存管理。

平台无关性：指 Java 语言平台无关，而 Java 的虚拟机却不是，需要下载对应平台 JVM 虚拟机的。

自动内存管理：对临时存储的数据自动进行回收，释放内存。如：引用类型的变量没有指向时，被回收；程序执行完后，局部变量被回收。

1.3 Java 开发环境

Java Development Kit——Java 开发工具包，简称 JDK，是由 Sun 公司提供的一个免费的 Java 开发工具，编程人员和最终用户可以利用这个工具来编译、运行 Java 程序。目前版本有 JDK1.0、JDK1.1、JDK1.2、JDK1.3、JDK1.4、JDK1.5 (J2SE5.0)、JDK1.6 (J2SE6.0)、JDK1.7 (J2SE7.0)。

JDK 结构：JDK

```
--开发工具 (Tools) 命令：java、javac、jar、rmic ...
|-- JRE (Java 基本运行环境)
    |-- 系统 API 库，系统类库
        |   系统带来的标准程序库，标准 API
    |-- JVM java 虚拟机
```

1.4 Java 开发环境配置

安装完 JDK 之后，不能立刻使用，需要设置环境变量：

- 1) 设置 PATH: D:\Java\jdk1.6.0\bin (指向 JDK 中 bin 文件夹，有各种编译命令)。
- 2) CLASSPATH: 告诉 Java 程序去哪里查找第三方和自定义类，如果 .class 文件和类源文件在同一文件夹内，则不需要配置 classpath，后续有包，则需要。

A. Windows: 在命令行执行

```
set CLASSPATH=E:\workspace\1304\bin (临时环境配置)
```

```
java day02.Demo1
```

◆ 注意事项：

- ❖ E: \set classpath = c:\ (不加分号就不找当前路径)
= .; c:\ ; d:\ ; (先找 classpath，若无，再找当前路径)
- ❖ C、D 两盘有同名 .class 文件，classpath 设置为 D 盘，而命令行窗口当前盘符为 C 盘，则 JVM 现找 classpath 路径，后找当前路径。

B. Linux: 在控制台执行

- ① 设置 CLASSPATH 环境变量，指向 package 所在的目录，一般是项目文件夹中的 bin 目录。

② 执行 java package.ClassName (包名必须写)。

```
export CLASSPATH=/home/soft01/workspace/1304/bin (临时环境配置)
```

```
java day01.HelloWorld
```

```
java -cp /home/soft01/workspace/1304/bin day01.HelloWorld (二合一)
```

◆ 注意事项：

- ❖ Windows 根目录是反斜线: \
- ❖ Linux 根目录是斜线: /

1.5 Linux 命令与相关知识

- 1) Linux 无盘符，只有一个根目录 (root)
- 2) 终端 == 控制台 == 命令行窗口
- 3) pwd: 打印当前工作目录，显示当前工作目录的位置
- 4) ls: 列表显示目录内容，默认显示当前目录内容
- 5) cd: 改变当前工作目录；cd 后不加参数=返回 home 文件夹；cd ~: 返回 home;
cd /: 切换到根目录；cd .. : 返回上一层目录（相对的）；
- 6) mkdir: 创建文件夹（目录） 注意：目录 == 文件夹
- 7) rm: 删除文件；rm xx xx: 可删多个文件；
rm -rf xx: -为减号，r 表递归，f 表强制
- 8) cat xx: 显示文本文件内容
- 9) 启动 Java 开发工具：cd/opt/eclipse → ./eclipse . 表当前目录下
- 10) 绝对路径：/home （以 / 开始为绝对路径，相对于根目录）
相对路径：home （相对于当前工作目录）
- 11) home (用户主目录，用户的家): /home/username 如: /home/soft01
- 12) 主目录 (home): 有最大访问权限：什么都能干，增删改查、建目录等
其他地方：一般只能查看，不能增删改查、创建目录等

1.6 Eclipse/Myeclipse 程序结构

```
Project 项目文件
|-- src 源文件
|   |-- Package 包
|       |-- .java 源文件
|-- bin
    |-- Package 包
        |-- .class 字节码程序
```

- ◆ 注意事项：
 - ❖ Myeclipse5.5 消耗少， Myeclipse6.5 最稳定

二、Java 语言基础

LICHOO

2.1 基础语言要素

- 1) 标识符: 给类、方法、变量起的名字
 - A. 必须以字母或下划线或 \$ 符号开始, 其余字符可以是字母、数字、\$ 符号和下划线。
 - B. 只能包含两个特殊字符, 即下划线 _ 和美元符号 \$ 。不允许有任何其他特殊字符。
 - C. 标识符不能包含空格。
 - D. 区分大小写。
- 2) 关键字: 只有系统才能用的标识符
 - ◆ 注意事项:
 - ❖ true、false、null 不是关键字! 是字面量。
 - ❖ main 不是关键字! 但是是一个特殊单词, 可以被 JVM 识别, 主函数是固定格式, 作为程序的入口。
- 3) 注释: 单行注释: // 多行注释: /* */ 文档注释: /**.....*/
 - ◆ 注意事项: 开发中类前、属性前、方法前, 必须有文档注视。

2.2 八种基本数据类型

- 1) 四种整数类型(byte、short、int、long):

byte: 8 位, 用于表示最小数据单位, 如文件中数据, -128~127
short: 16 位, 很少用, -32768~32767
int: 32 位、最常用, -2^31-1~2^31 (21 亿)
long: 64 位、次常用

 - ◆ 注意事项:
 - ❖ int i=5; // 5 叫直接量 (或字面量), 即直接写出的常数。
 - ❖ 整数字面量默认都为 int 类型, 所以在定义的 long 型数据后面加 L 或 l。
 - ❖ 小于 32 位数的变量, 都按 int 结果计算。
 - ❖ 强转符比数学运算符优先级高。见常量与变量中的例子。
- 2) 两种浮点数类型(float、double):

float: 32 位, 后缀 F 或 f, 1 位符号位, 8 位指数, 23 位有效尾数。
double: 64 位, 最常用, 后缀 D 或 d, 1 位符号位, 11 位指数, 52 位有效尾数。

 - ◆ 注意事项:
 - ❖ 二进制浮点数: $1010100010 = 101010001.0 * 2 = 10101000.10 * 2^{10}$ (2 次方) = $1010100.010 * 2^{11}$ (3 次方) = $.1010100010 * 2^{10}$ (10 次方)
 - ❖ 尾数: .1010100010 指数: 1010 基数: 2
 - ❖ 浮点数字面量默认都为 double 类型, 所以在定义的 float 型数据后面加 F 或 f; double 类型可不写后缀, 但在小数计算中一定要写 D 或 X.X。
 - ❖ float 的精度没有 long 高, 有效位数 (尾数) 短。
 - ❖ float 的范围大于 long 指数可以很大。
 - ❖ 浮点数是不精确的, 不能对浮点数进行精确比较。
- 3) 一种字符类型(char):

char: 16 位, 是整数类型, 用**单引号**括起来的**1**个字符 (可以是一个中文字符), 使用 Unicode 码代表字符, 0~2^16-1 (65535)。

◆ 注意事项:

- ❖ 不能为 0 个字符。
- ❖ 转义字符: \n 换行 \r 回车 \t Tab 字符 \" 双引号 \\ 表示一个\
- ❖ 两字符 char 中间用 “+” 连接, 内部先把字符转成 int 类型, 再进行加法运算, char 本质就是个数! 二进制的, 显示的时候, 经过“处理”显示为字符。

4) 一种布尔类型(boolean): true 真 和 false 假。

5) 类型转换: char-->

自动转换: byte-->short-->int-->long-->float-->double

强制转换: ①会损失精度, 产生误差, 小数点以后的数字全部舍弃。

②容易超过取值范围。

2.3 常量和变量

变量: 内存中一块存储空间, 可保存当前数据。在程序运行过程中, 其值是可以改变的量。

- 1) 必须声明并且初始化以后使用 (在同一个作用域中不能重复声明变量)!
- 2) 变量必须有明确类型 (Java 是强类型语言)。
- 3) 变量有作用域 (变量在声明的地方开始, 到块 {} 结束)。变量作用域越小越好。
- 4) 局部变量在使用前一定要初始化! 成员变量在对象被创建后有默认值, 可直接用。
- 5) 在方法中定义的局部变量在该方法被加载时创建。

常量: 在程序运行过程中, 其值不可以改变的量。

◆ 注意事项:

- ❖ 字面量、常量和变量的运算机制不同, 字面量、常量由编译器计算, 变量由运算器处理, 目的是为了提高效率。

eg: 小于 32 位数的字面量处理

```
byte b1 = 1; byte b2 = 3;  
//byte b3 = b1+b2;//编译错误, 按照 int 结果, 需要强制转换  
byte b3 = (byte)(b1+b2);  
//byte b3 = (byte)b1+(byte)b2;//编译错误! 两个 byte、short、char 相加还是按 int 算  
System.out.println(b3); //选择结果: A 编译错误 B 运行异常 C 4 D b3  
byte b4 = 1+3;//字面量运算, 编译期间替换为 4, 字面量 4  
//byte b4 = 4; 不超过 byte 就可以赋值
```

- ❖ 不管是常量还是变量, 必须先定义, 才能够使用。即先在内存中开辟存储空间, 才能够往里面放入数据。
- ❖ 不管是常量还是变量, 其存储空间是有数据类型的差别的, 即有些变量的存储空间用于存储整数, 有些变量的存储空间用于存储小数。

2.4 运算符与表达式

1) 数学运算: + - * / % ++ --

◆ 注意事项:

- ❖ + - * / 两端的变量必须是同种类型, 并返回同种类型。
- ❖ % 取余运算, 负数的余数符号与被模数符号相同, -1 % 5 = -1, 1 % -5

= 1; Num % n, n>0, 结果范围[0,n), 是周期函数。

◆ 注意整除问题: 1 / 2 = 0 (整数的除法是整除) 1.0 / 2 = 0.5 1D / 2 = 0.5

◆ 单独的前、后自增或自减是没区别的, 有了赋值语句或返回值, 则值不同!

eg1: 自增自减

```
int a = 1; a = a++; System.out.println("a 的值: "+a);
第1步: 后++, 先确定表达式 a++ 的值 (当前 a 的值) a++ --->1
第2步: ++, 给 a 加 1 a --->2
第3步: 最后赋值运算, 把 a++ 整个表达式的值赋值给 a a --->1
a 被赋值两次, 第1次 a=2, 第2次把 1 赋值给 1
```

eg2: 自增自减

```
x, y, z 分别为 5, 6, 7 计算 z += --y * z++ ; // x = 5, y = 5, z = 42
z = z + --y * z++ → 42 = 7 + 5 * 7 从左到右入栈, 入的是值
```

eg3: 取出数字的每一位

```
d = num%10;//获取 num 的最后一位数 num/=10;//消除 num 的最后一位
```

2) 位运算: & | ~ (取反) ^ (异或) >> << >>>

◆ 注意事项:

◆ 一个数异或同一个数两次, 结果还是那个数。

◆ |: 上下对齐, 有 1 个 1 则为 1; &: 上下对齐, 有 1 个 0 则为 0; (都为二进制)

◆ & 相当于乘法, | 相当于加法; &: 有 0 则为 0, |: 有 1 则为 1, ^: 两数相同为 0, 不同为 1。

3) 关系运算符: > < >= <= == !=

4) 逻辑运算符: && || (短路) ! & |

eg: 短路运算: &&: 前为 false, 则后面不计算; ||: 前为 true, 则后面不计算

```
int x=1,y=1,z=1;
if(x==1 && y==1 || z==1) // || 短路运算后面的不执行了!
System.out.println("x="+x+",y="+y+",z="+z); // 0, 2, 1
```

5) 赋值运算符: = += -= *= /= %=

eg: 正负 1 交替

```
int flag=-1; System.out.println(flag *= -1); .....
```

6) 条件(三目)运算符: 表达式 1 ? 表达式 2 : 表达式 3

◆ 注意事项:

◆ 右结合性: a > b ? a : i > j ? i : j 相当于 a > b ? a : (i > j ? i : j)

◆ 三目运算符中: 第二个表达式和第三个表达式中如果都为基本数据类型, 整个表达式的运算结果由容量高的决定。如: int x = 4; x > 4 ? 99.9 : 9; 99.9 是 double 类型, 而 9 是 int 类型, double 容量高, 所以最后结果为 9.9。

7) 运算符优先级: 括号 > 自增自减 > ~! > 算数运算符 > 位移运算 > 关系运算 > 逻辑运算 > 条件运算 > 赋值运算

2.5 编程风格

MyEclipse/Eclipse 中出现的红色叉叉: 编译错误

编译错误: java 编译器在将 Java 源代码编译为 class 文件的过程出现的错误, 一般是语法使用错误! 当有编译错误时候, 是没有 class 文件产生, 也就不能运行程序。

Java 程序结构：

```

package demo.day01; //必须是小写字母，多个单词用.隔开
import java.util.Scanner;//Java API 一定在当前库中存在

public class HelloWorld { //类名每个单词首字母要大写
    //类体中的成员，要缩进1个Tab 宽
    public static void main(String[] args) {
        //方法中成员也要缩进1个Tab 宽
        //Java的语句以英文分号 ; 为结尾，不是中文的分号！
        //括号要配对使用：先写出成对的括号，然后在中间填代码
        Scanner console = new Scanner(System.in);
        } //方法体的结束也要和方法的声明位置对齐
    } //类体 class body
//括号要配对，声明开始的位置和结束的位置要对齐

```

2.6 流程控制语句

1) 选择控制语句

if 语句： if 、 if-else、 if-else-if： 可以处理一切分支判断。

格式： if(判断){...}、 if(判断){...}else{...}、 if(判断){...}else if(判断){...}

switch 语句： switch(必须整数类型){case 常量 1: ...; case 常量 2: ...;}

◆ 注意事项：

- ❖ int 类型指： byte、 short、 int， 只能写 long 类型， 要写也必须强转成 int 类型； 而 byte、 short 为自动转换成 int。
- ❖ switch-case： 若 case 中无符合的数，并且 default 写在最前（无 break 时）， 则为顺序执行， 有 break 或 } 则退出。
- ❖ switch-case： 若 case 中无符合的数，并且 default 写在最后，则执行 default。
- ❖ switch-case： 若 case 中有符合的数，并且 default 写在最后，并且 default 前面的 case 没有 break 时， default 也会执行。

2) 循环控制语句

①for： 最常用， 用在与次数有关的循环处理， 甚至只用 for 可以解决任何循环问题。

- ◆ 注意事项： for 中定义的用于控制次数的循环变量， 只在 for 中有效， for 结束则循环变量被释放（回收）。

②while： 很常用， 用在循环时候要先检查循环条件再处理循环体， 用在与次数无关的情况。 如果不能明确结束条件的时候， 先使用 while(true)， 在适当条件使用 if 语句加 break 结束循环。

③do-while： 在循环最后判断是否结束的循环。 如： 使用 while(true) 实现循环的时候， 结束条件 break 在 while 循环体的最后， 就可以使用 do-while。 do-while 的结束条件经常是“否定逻辑条件”， 不便于思考业务逻辑， 使用的时候需要注意。 可以利用 while (true) + break 替换。

④循环三要素： A. 循环变量初值 B. 循环条件 C. 循环变量增量（是循环趋于结束的表达式）

⑤for 和 while 循环体中仅一条语句， 也要补全{}， 当有多条语句， 且不写{}时， 它们只执行紧跟着的第一条语句。

⑥循环的替换：

while(布尔表达式){} 等价 for(;布尔表达式;){}

while(true){} 等价 for(;;){}

while(true){} + break 替换 do{}while(布尔表达式);
 for(;;) + break 替换 do{}while(布尔表达式);

3) 跳转控制语句

continue: 退出本次循环, 直接执行下一次循环

break: 退出所有循环

2.7 数组

类型一致的一组数据, 相当于集合概念, 在软件中解决一组, 一堆 XX 数据时候使用数组。

1) 数组变量: 是引用类型变量 (不是基本变量) 引用变量通过数组的内存地址位置引用了一个数组 (数组对象), 即栓到数组对象的绳子。

eg: 数组变量的赋值

```
int[] ary = new int[3];// ary----->{0,0,0}<----ary1
int[] ary1 = ary;// ary 的地址赋值给 ary1, ary 与 ary1 绑定了同一个数组
//ary[1] 与 ary1[1] 是同一个元素, 数组变量不是数组 (数组对象)
```

2) 数组 (数组对象) 有 3 种创建 (初始化) 方式: ①new int[10000] 给元素数量, 适合不知道具体元素, 或元素数量较多时 ②new int[]{3,4,5} 不需要给出数量, 直接初始化具体元素适合知道数组的元素。③ {2,3,4} 静态初始化, 是②简化版, 只能用在声明数组变量的时候直接初始化, 不能用于赋值等情况。

eg: 数组初始化

```
int[] ary1 = new int[]{2,3,4}//创建数组时候直接初始化元素
int[] ary2 = {2,3,4};//数组静态初始化, 只能在声明变量的同时直接赋值
//ary2 = {4,5,6};//编译错误, 不能用于赋值等情况
ary2 = new int[]{4,5,6};
```

3) 数组元素的访问: ①数组长度: 长度使用属性访问, ary.length 获取数组下标。②数组下标: 范围: 0 ~ length-1 就是[0,length), 超范围访问会出现下标越界异常。③使用[index] 访问数组元素: ary[2]。④迭代 (遍历): 就是将数组元素逐一处理一遍的方法。

4) 数组默认初始化值: 根据数组类型的不同, 默认初始化值为: 0 (整数)、0.0 (浮点数)、false (布尔类型)、\u0000 (char 字符类型, 显示无效果, 相当于空格, 编码为 0 的字符, 是控制字符, 强转为 int 时显示 0)、null (string 类型, 什么都没有, 空值的意思)。

5) 数组的复制: 数组变量的赋值, 是并不会复制数组对象, 是两个变量引用了同一个数组对象。数组复制的本质是创建了新数组, 将原数组的内容复制过来。

6) 数组的扩容: 创建新数组, 新数组容量大于原数组, 将原数组内容复制到新数组, 并且丢弃原数组, 简单说: 就是更换更大的数组对象。System.arraycopy() 用于复制数组内容, 简化版的数组复制方法: Arrays.copyOf()方法, 但需 JDK1.5+。

2.8 字符串

字符串(string): 永远用 “” 双引号 (英文状态下), 用字符串连接任何数据 (整数), 都会默认的转化为字符串类型。

字符串与基本数据类型链接的问题: 如果第一个是字符串那么后续就都按字符串处理, 如 System.out.println("(Result)" + 6 + 6); 那么结果就是(Result)66, 如果第一个和第二个…第 n 个都是基本数据, 第 n+1 是字符串类型, 那么前 n 个都按加法计算出结果在与字符串连接, 如下例中的 System.out.println(1+2+"java" + 3+4); 结果为 3java34。

eg: 字符串前后的“+”都是连接符！不是加法运算符！

```
System.out.println("A"+'B');//AB  
System.out.println('A'+'B');//131  
System.out.println(1+2+"java"+3+4);//3java34
```

- ◆ 注意事项：比较字符串是否相等必须使用 equals 方法！不能使用`==`。`"1".equals(cmd)` 比 `cmd.equals("1")` 要好。

2.9 方法三要素

方法：method（函数 function = 功能） $y=f(x)$

- 1) 方法的主要三要素：方法名、参数列表、返回值。
 - 2) 什么是方法：一个算法逻辑功能的封装，是一般完成一个业务功能，如：登录系统，创建联系人，简单说：方法是动作，是动词。
 - 3) 方法名：一般按照方法实现的功能定名，一般使用动词定义，一般使用小写字母开头，第二个单词开始，单词首字母大写。如：`createContact()`。
 - 4) 参数列表：是方法的前提条件，是方法执行依据，是数据。如：
`login(String id, String pwd)`，参数的传递看定义的类型及顺序，不看参数名。
 - 5) 方法返回值：功能执行的结果，方法必须定义返回值，并且方法中必须使用 `return` 语句返回数据；如果无返回值则定义为 `void`，此时 `return` 语句可写可不写；返回结果只能有一个，若返回多个结果，要用数组返回（返回多个值）。
- ◆ 注意事项：递归调用：方法中调用了方法本身，用递归解决问题比较简练，只需考虑一层逻辑即可！但是需要有经验。一定要有结束条件！如：`f(1)=1`；递归层次不能太深。总之：慎用递归！

2.10 插入排序

将数组中每个元素与第一个元素比较，如果这个元素小于第一个元素，则交换这两个元素循环第 1 条规则，找出最小元素，放于第 1 个位置经过 $n-1$ 轮比较完成排序。

```
for(int i = 1; i < arr.length; i++) {  
    int k = arr[i];// 取出待插入元素  
    int j;// 找到插入位置  
    for (j = i - 1; j >= 0 && k < arr[j]; j--) {  
        arr[j + 1] = arr[j];// 移动元素  
    }  
    arr[j + 1] = k;// 插入元素  
    System.out.println(Arrays.toString(arr));  
}
```

2.11 冒泡排序

比较相邻的元素，将小的放到前面。

```
for(int i = 0; i < arr.length - 1; i++) {  
    boolean isSwap = false;  
    for (int j = 0; j < arr.length - i - 1; j++) {  
        if (arr[j] > arr[j + 1]) {  
            int t = arr[j];  
            arr[j] = arr[j + 1];  
            arr[j + 1] = t;  
            isSwap = true;  
        }  
    }  
    if (!isSwap) {  
        break;  
    }  
}
```

```

        arr[j + 1] = t;
        isSwap = true;
    }
}
if (!isSwap){ break; }
System.out.println(Arrays.toString(arr));
}

```

2.12 冒泡排序：轻气泡上浮的方式

冒泡排序法可以使用大气泡沉底的方式，也可以使用轻气泡上浮的方式实现。如下为使用轻气泡上浮的方式实现冒泡排序算法。

```

for (int i = 0; i < arr.length - 1; i++) {
    boolean isSwap = false;
    for (int j = arr.length - 1; j > i; j--) {
        if (arr[j] < arr[j - 1]) {
            int t = arr[j];
            arr[j] = arr[j - 1];
            arr[j - 1] = t;
            sSwap = true;
        }
    }
    if (!isSwap){ break; }
    System.out.println(Arrays.toString(arr));
}

```

2.13 二分法查找

```

intlow = 0;           inthigh = arr.length - 1;           intmid = -1;
while(low <= high) {
    mid = (low + high) / 2;
    if (arr[mid] < value){ low = mid + 1;      }
    else if (arr[mid] > value){ high = mid - 1;     }
    else{ break;}
}
if (low <= high) {   System.out.println("可以找到: index = " + mid + "。");
} else {   System.out.println("无法找到! ");
}

```

二分法思想是取中，比较：

1) 求有序序列 arr 的中间位置 mid。 2) k 为要查找的数字。

若 arr[mid] == k, 查找成功；

若 arr[mid] > k, 在前半段中继续进行二分查找；

若 arr[mid] < k, 则在后半段中继续进行二分查找。

假如有一组数为 3、12、24、36、55、68、75、88 要查给定的值 k=24。可设三个变量 low、mid、high 分别指向数据的上界，中间和下界， $mid = (low+high)/2$ 。

1) 开始令 low=0 (指向 3), high=7 (指向 88), 则 mid=3 (指向 36)。因为 k<mid, 故应在前半段中查找。

2) 令新的 $high=mid-1=2$ (指向 24), 而 $low=0$ (指向 3) 不变, 则新的 $mid=1$ (指向 12). 此时 $k>mid$, 故确定应在后半段中查找。

3) 令新的 $low=mid+1=2$ (指向 24), 而 $high=2$ (指向 24) 不变, 则新的 $mid=2$, 此时 $k=arr[mid]$, 查找成功。

如果要查找的数不是数列中的数, 例如 $k=25$, 当第四次判断时, $k>mid[2]$, 在后边半段查找, 令 $low=mid+1$, 即 $low=3$ (指向 36), $high=2$ (指向 24) 出现 $low>high$ 的情况, 表示查找不到成功。

2.14 Java 系统 API 方法调用

Arrays 类, 是数组的工具类, 包含很多数组有关的工具方法。如:

- 1) `toString()` 连接数组元素为字符串, 方便数组内容输出。
- 2) `equals` 比较两个数组序列是否相等。
- 3) `sort()` 对数组进行排序, 小到大排序。
- 4) `binarySearch(names, "Tom")` 二分查找, 必须在有序序列上使用。

2.15 二进制基础

1) 计算机中一切数据都是 2 进制的! 基本类型, 对象, 音频, 视频。

2) 10 进制是人类习惯, 计算按照人类习惯利用算法输入输出。

"10" -算法转化-> 1010(2) 1010 -算法转化-> "10"

3) 16 进制是 2 进制的简写, 16 进制就是 2 进制!

4) 计算机硬件不支持正负号, 为了解决符号问题, 使用补码算法, 补码规定高位为 1 则为负数, 每位都为 1 则为-1, 如 $1111\ 1111 = -1 = 0xff$

5) 二进制数右移`>>`: 相当于数学 $/ 2$ (基数), 且正数高位补 0, 负数高位补 1; 二进制数左移`<<`: 相当于数学 $* 2$ (基数), 且低位补 0; 二进制数无符号右移`>>>`: 相当于数学 $/ 2$ (基数), 且不论正负, 高位都补 0。

6) 注意掩码运算: 把扩展后前面为 1 的情况除去, 与 `0xff` 做与运算。

eg1: 二进制计算

```
int max = 0xffffffff;      long l = max + max + 2;      System.out.println(l); // 0
```

eg2: 二进制运算 (拼接与拆分)

```
int b1 = 192; int b2 = 168; int b3 = 1; int b4 = 10; int color = 0xD87455;
int ip = (b1<<24) + (b2<<16) + (b3<<8) + b4; // 或者 ip = (b1<<24) | (b2<<16) | (b3<<8) | b4;
int b = color&0xff; // 85   int g = (color >> 8)&0xff; // 116   int r = (color >> 16)&0xff; // 216
```

2.16 Java 基础其他注意事项

- ❖ Java 程序严格区分大小写。
- ❖ 类名, 每个单词首字母必须大写 (公司规范!)。
- ❖ 一个 Java 应用程序, 有且只有一个 `main` 方法, 作为程序的入口点。
- ❖ 每一条 Java 语句必须以分号结束。
- ❖ 类定义关键字 `class` 前面可以有修饰符 (如 `public`), 如果前面的修饰符是 `public`, 该类的类名必须要与这个类所在的源文件名称相同。
- ❖ 注意程序的缩进。
- ❖ `double a[] = new double[2];` //语法可以, 但企业中一定不要这么写, Java 中`[]`建议放前面。
- ❖ **Java 中所有范围参数都是包含 0, 不包含结束**, 如 `int n = random.nextInt(26);` //生

成 0 到 26 范围内的随机数，不包括 26。

- ❖ 任何数据在内存中都是 2 进制的数据，内存中没有 10 进制 16 进制。
- ❖ int n = Integer.parseInt(str); // 将字符串--> int 整数。
- ❖ System.out.println(Long.toBinaryString(maxL)); Long 类型用 Long.XXXX 。
- ❖ 程序：数据+算法 数据即为变量，算法为数据的操作步骤，如：顺序、选择、循环。
- ❖ 字符串按编码大小排序。

三、面向对象

Object: 对象，东西，一切皆对象 == 哪都是东西
面向对象核心：封装、继承、多态。

3.1 类

- 1) 是同类型东西的概念，是对现实生活中事物的描述，映射到 Java 中描述就是 class 定义的类。类是对象的模板、图纸，是对象的数据结构定义。简单说就是“名词”。
- 2) 其实定义类，就是在描述事物，就是在定义属性（变量）和方法（函数）。
- 3) 类中可以声明：属性，方法，构造器；属性（变量）分为：实例变量，局部变量；
实例变量：用于声明对象的结构的，在创建对象时候分配内存，每个对象有一份！实例变量（对象属性）在堆中分配，并作用于整个类中，实例变量有默认值，不初始化也能参与运算；局部变量在栈中分配，作用于方法或语句中，必须初始化，有值才能运算。
- 4) 类与类之间的关系：①关联：一个类作为另一个类的成员变量：需要另一个类来共同完成。class A { public B b } class B {} ②继承：class B extends A {} class A {}
③依赖：个别方法和另一个类相关。class A { public void f(B b) {} //参数里有 B
public B g() {} //返回值里有 B } class B {}

```

package day01;

public class Demo07 {
    public static void main(String[] args) {
        Point p1; //引用“变量”，是对象的名字
        p1 = new Point(); //对象 p1(3,4)
        p1.x = 3; //调用成员运算，可以理解为“的”
        p1.y = 4;
        Point p2 = new Point(); //p2(5,5)
        p2.x = 5;
        p2.y = 5;
        System.out.println(p1.x+"."+p1.y); //3,4
        System.out.println(p2.x+"."+p2.y); //5,5
    }
}
class Point{ //类
    int x;
    int y;
}

```

方法中声明的变量叫局部变量如：p1 p2
类 (.class) 加载到方法区
局部变量p1p2在栈中分配
对象在堆中创建，对象按照属性分配空间
方法区

- 5) **null 与空指针异常：**引用类型变量用于存放对象的地址，可以给引用类型赋值为 null，表示不指向任何对象。当某个引用类型变量为 null 时无法对对象实施访问（因为它没有指向任何对象）。此时，如果通过引用访问成员变量或调用方法，会产生 NullPointerException 空指针异常。

◆ 注意事项：除了 8 中基本类型，其他都是引用类型变量（也叫句柄）。

3.2 对象

是这类事物实实在在存在的个体！利用类做为模板创建的个体实例，本质是数据。

匿名对象：使用方式一：当对对象的方法只调用一次时，可用匿名对象来完成，这样比较简化。如果对一个对象进行多个成员调用，则必须给这个对象起个名字。

使用方式二：可以将匿名对象作为实际参数进行传递。

3.3 包

- 1) 包名必须是小写，多个单词用“.”隔开。在同一个包中，不能有同名的类！

2) 只要在同一个包中，则可直接用 `extends`（类型互知道在哪），若不在同一个包中，则用 `import` 导入。

3.4 方法及其调用

是用于对当前对象数据进行算法计算，实现业务功能。方法是对象的功能，对象的动作，对象的行为。总之是动词！方法名没有规定，建议首单词为小写动词，其他单词首字母大写，必须定义返回值！可有无参数方法。**方法调用只有两种方式：**①对象引用调用②类名调用（即静态类时）。

3.5 引用

是对个体的标识名称。

- 1) 是代词，是对象的引用，就像拴着对象的绳子。
 - 2) 引用本身不是对象！引用指代了对象！
 - 3) 引用的值是对象的地址值（或叫句柄），通过地址值引用了对象。
 - 4) 引用的值不是对象！
- ◆ 注意事项：“.” 叫取成员运算，可以理解为“的”。

3.6 访问控制（封装）

封装：将数据封装到类的内部，将算法封装到方法中。

- 1) 封装原则：将不需要对外提供的内容都隐藏起来，把属性都隐藏，提供公共方法对其访问，通常有两种访问方式：`set` 设置，`get` 获取。
 - 2) 封装结果：存在但是不可见。
 - 3) `public`：任何位置可见，可以修饰：类、成员属性、成员方法、内部类、跨包访问类（需要使用 `import` 语句导入），成员属性 == 成员变量。
 - 4) `protected`：当前包中可见，子类中可见。可以修饰：成员属性、成员方法、内部类（只能在类体中使用，不能修饰类）。
 - 5) 默认的：当前包内部可见，就是没有任何修饰词，可以修饰：类、成员属性、成员方法、内部类，但在实际项目中很少使用。默认类（包内类）的访问范围：当前包内部可见，不能在其他包中访问类，访问受限！`main` 方法若定在默认类中 JVM 将找不到，无法执行，因此必定在 `public` 类中。
 - 6) `private`：仅仅在类内部可见。可以修饰：成员属性、成员方法、内部类（只能在类体中使用，不能修饰类）。私有的方法不能继承，也不能重写。
- ◆ 注意事项：在企业项目中建议：所有类都是公用类。封装的类使用内部类！

3.7 构造器

用于创建对象并初始化对象属性的方法，叫“构造方法”，也叫“构造器”；构造器在类中定义。

- 1) 构造器的名称必须与类名同名，包括大小写。
- 2) 构造器没有返回值，但也不能写 `void`，也不能写 `return`。
- 3) 构造器的参数：一般是初始化对象的前提条件。
- 4) 用 `new` 调用！且对象一建立，构造器就运行且仅运行一次。一般方法可被调用多次。
- 5) 类一定有构造器！这是真的，不需要质疑！
- 6) 如果类没有声明（定义）任何的构造器，Java 编译器会自动插入默认构造器！

7) 默认构造是无参数，方法体是空的构造器，且默认构造器的访问权限随着所属类的访问权限变化而变化。如，若类被 public 修饰，则默认构造器也带 public 修饰符。

8) 默认构造器是看不到的，一旦自己写上构造器则默认构造器就没有了，自己写的叫自定义构造器，即便自己写的是空参数的构造器，也是自定义构造器，而不是默认构造器。

9) 如果类声明了构造器，Java 编译器将不再提供默认构造器。若没手动写出无参构造器，但却调用了无参构造器，将会报错！

eg: 默认构造器

```
public class Demo {    public static void main(String[] args) {  
    Foo foo = new Foo(); //调用了 javac 自动添加的默认构造器!  
    //Koo koo = new Koo(); //编译错误，没有 Koo() 构造器  
    Koo koo = new Koo(8);            }  
    class Foo {} //Foo 有构造器，有无参数的默认构造器!  
    class Koo {    public Koo(int a) { //声明了有参数构造器  
        System.out.println("Call Koo(int)");    }  
    }  
}
```

```
public class Demo02 {  
    public static void main(String[] args) {  
        Point p1=new Point(3,4); //new 运算调用构造器，返回对象  
        p1.up(2); //使用引用调用对象的方法，实现移动的功能  
        System.out.println(p1.y); //2  
        Point p2=new Point(5,5);  
        p2.up(2);  
        System.out.println(p2.y);  
    }  
    class Point{  
        int x; int y;  
        public Point(int x, int y){  
            this.x = x; this.y = y;  
        }  
        public void up(int dy){  
            this.y-=dy;  
        }  
    }  
}
```

Demo02.class
Point.class

Point
x=5
y=3

Point
x=3
y=2

p2
this
dy=2
p1
this
y=4
x=3

实例变量，每个对象实例都有一个！

10) 构造器是可以重载的，重载的目的是为了使用方便，重载规则与方法重载规则相同。

11) 构造器是不能继承的！虽说是叫构造方法，但实际上它不是常说的一般方法。

12) 子类继承父类，那么子类型构造器默认调用父类型的无参数构造器。

13) 子类构造器一定要调用父类构造器，如果父类没有无参数构造器，则必须使用 super(有参数的)，来调用父类有参的构造器。那么，为什么子类一定要访问父类的构造器？因为父类中的数据子类可以直接获取。所以子类对象在建立时，需要先查看父类是如何对这些数据进行初始化的，所以子类在对象初始化时，要先访问一下父类中的构造器。

总之，子类中至少会有一个构造器会访问父类中的构造器，且子类中每一个构造函数内的第一行都有一句隐式 super()。

3.8 super()、super. 和 this()、this.

1) this: 在运行期间，哪个对象在调用 this 所在的方法，this 就代表哪个对象，隐含绑定到当前“这个对象”。

2) super(): 调用父类无参构造器，一定在子类构造器第一行使用！如果没有则是默认存在 super() 的！这是 Java 默认添加的 super()。

3) super. 是访问父类对象，父类对象的引用，与 this. 用法一致

4) `this()`: 调用本类的其他构造器, 按照参数调用构造器, 必须在构造器中使用, 必须在第一行使用, `this()` 与 `super()` 互斥, 不能同时存在

5) `this.` 是访问当前对象, 本类对象的引用, 在能区别实例变量和局部变量时, `this` 可省略, 否则一定不能省!

6) 如果子父类中出现非私有的同名成员变量时, 子类要访问本类中的变量用 `this.` ; 子类要访问父类中的同名变量用 `super.` 。

eg1: 方法参数传递原理 与 `this` 关键字

```
public class Demo11 {
    public static void main(String[] args) {
        Point2 p1 = new Point2();
        p1.x = 10; p1.y=15;
        p1.up(5);
        // up(p1, 5);方法调用时候当前对象引用p1隐含传递给this
        System.out.println(p1.y);
        Point2 p2 = new Point2();
        p2.up(3); //up(p2, 3);
        System.out.println(p2.y); -3
    }
}
class Point2{
    int x; int y;
    public void up/*Point2 this*/ int dy){
        //this 是方法中对当前对象的引用, 本质是方法的隐藏参数
        this.y = this.y - dy;
    }
}
```

eg2: `this.` 和 `this()`

```
Cell c = new Cell(); System.out.println(c.x + "," + c.y);
class Cell { int x; int y;
    public Cell() { this(1,1); //调用本类的其他构造器 }
    public Cell( int x, int y) { this.x = x; this.y = y; }
}
```

eg3: `super()`

```
class Xoo{ public Xoo(int s) { System.out.println("Call Xoo(int)"); }
    //super()用于在子类构造器中调用父类的构造器
class Yoo extends Xoo{
    //public Yoo() {} //编译错误, 子类调用不到父类型无参数构造器
public Yoo(){//super(); //编译错误, 子类调用不到父类型无参数构造器
    super(100); //super(100) 调用了父类 Xoo(int) 构造器 }
}
```

3.9 重载和重写

1) **重写**: 通过类的继承关系, 由于父类中的方法不能满足新的要求, 因此需要在子类中修改从父类中继承的方法叫重写(覆盖)。

①方法名、参数列表、返回值类型与父类的一模一样, 但方法的实现不同。若方法名、参数列表相同, 但返回值类型不同会有变异错误! 若方法名、返回值类型相同, 参数列表不同, 则不叫重写了。

②子类若继承了抽象类或实现了接口, 则必须重写全部的抽象方法。若没有全部实现抽象方法, 则子类仍是一个抽象类!

③子类重写抽象类中的抽象方法或接口的方法时, 访问权限修饰符一定要大于或等于被重写的抽象方法的访问权限修饰符!

④静态方法只能重写静态方法!

2) **重载**: 方法名一样，参数列表不同的方法构成重载的方法（多态的一种形式）。

①调用方法：根据参数列表和方法名调用不同方法。

②与返回值类型无关。

③重载遵循所谓“编译期绑定”，即在编译时根据**参数变量的类型**判断应调用哪个方法。 eg: 重载

```
int[] ary1 = {'A','B','C'};           char[] ary2 = {'A', 'B', 'C'};  
System.out.println(ary1);//println(Object)  
                                //按对象调用，结果为地址值，没有 println(int[])  
System.out.println(ary2);//println(char[]) ABC  
System.out.println('中');//println(char) 中  
System.out.println((int)'中');//println(int) 20013
```

3.10 继承

父子概念的继承：圆继承于图形，圆是子概念（子类型 Sub class）图形是父类型（Super Class 也叫超类），继承在语法方面的好处：子类共享了父类的属性和方法的定义，子类复用了父类的属性和方法，节省了代码。

- 1) 继承是 is a：“是”我中的一种，一种所属关系。
- 2) 子类型对象可以赋值给父类型变量（多态的一种形式），变量是代词，父类型代词可以引用子类型东西。
- 3) 继承只能是单继承，即直接继承，而非间接继承。因为多继承容易带来安全隐患，当多个父类中定义了相同功能，当功能内容不同时，子类无法确定要运行哪一个。
- 4) 父类不能强转成子类，会造形异常！子类向父类转化是隐式的。
- 5) 只有变量的类型定义的属性和方法才能被访问！见下例。
- 6) 重写遵循所谓“运行期绑定”，即在运行的时候根据**引用变量指向的实际对象类型**调用方法。

eg: Shape s, s 只能访问 Shape 上声明的属性和方法

```
Circle c = new Circle(3,4,5);  
Shape s = c;//父类型变量 s 引用了子类型实例  
//s 和 c 引用了同一个对象 new Circle(3,4,5)  
s.up();      System.out.println(c.r);  
System.out.println(c.area());  
//System.out.println(s.area());//编译错误  
//System.out.println(s.r)//在 Shape 上没有定义 r 属性！
```

7) 引用类型变量的类型转换 instanceof

```
public static void main(String[] args) {  
    Circle c = new Circle(3,4,5);      Rect r = new Rect(3,4,5,6);  
    Shape s = c;          Shape s1 = r;  
    //Circle x = s;//编译错误，父类型变量不能赋值给子类型  
    Circle x = (Circle)s;//正常执行  
    //Circle y = (Circle)s1;//运行异常,类型转换异常  
    //instanceof instace: 实例 of:  
    //instanceof 运算 检查变量引用的对象的类型是否兼容  
    //s引用的是圆对象,s instanceof Circle 检查s引用的对象是否是 Circle 类型的实例！
```

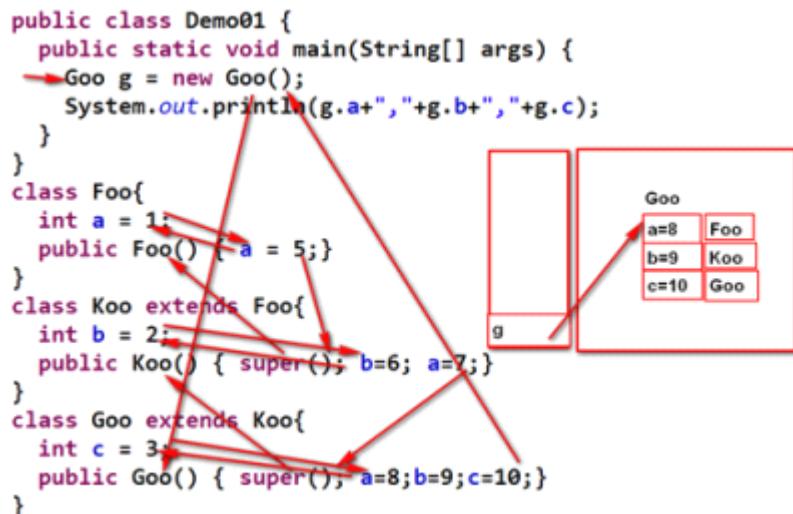
```

        System.out.println(s instanceof Circle);//true
        System.out.println(s1 instanceof Circle);//false
    test(c);      test(r);      }
public static void test(Shape s){//多态的参数
    //if(s instanceof Circle)保护了(Circle)s 不会出现异常
    if(s instanceof Circle){//实现了安全的类型转换
        Circle c = (Circle) s;  System.out.println("这是一个圆, 面积"+c.area());  }
    if(s instanceof Rect){
        Rect r = (Rect) s;  System.out.println("这是一个矩形, 面积"+r.area());  }
}

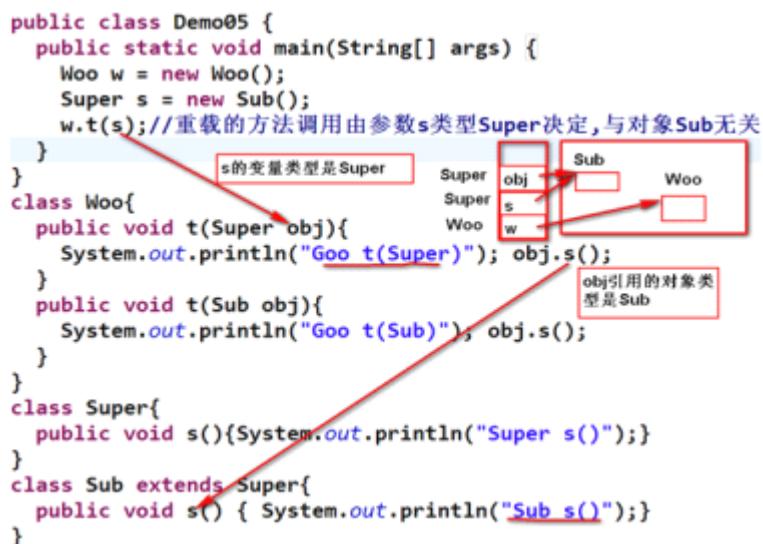
```

8) 继承时候对象的创建过程

①Java首先递归加载所有类搭配方法区。②分配父子类型的内存（实例变量）。③递归调用构造器。



9) 重写方法与重载方法的调用规则



10) 属性绑定到变量的类型，由变量类型决定访问哪个属性；方法动态绑定到对象，由对象的类型决定访问哪个方法。（强转对方法动态绑定到对象无影响，因为强转的是父类的引用，而实例是不变的，只是把实例当作另一个状态去看而已。但是强转对属性动态绑定到变量类型有影响。）其他解释请看多态部分！

```

public class Demo04 {
    public static void main(String[] args) {
        Cheater c = new Cheater();
        Person p = c;
        System.out.println(p.name + "+" + c.name);
        p.whoau(); c.whoau();
    }
}

class Person{
    String name = "灰太狼";
    public void whoau(){
        System.out.println(this.name);
    }
}

class Cheater extends Person{//Cheater:骗子
    String name = "喜羊羊";
    public void whoau() {
        System.out.println(this.name);
    }
}

```

eg1: 方法动态绑定到运行期间对象的方法 实例 1

```

public class Demo02 {
    public static void main(String[] args) {
        Moo moo = new Noo(); //父类型变量引用了子类对象
        moo.test(); //父类型Moo上声明的方法，子类型重写的方法
        //动态绑定到Noo对象，执行Noo对象的方法
    }
}

class Moo{
    public void test(){
        System.out.println("Moo test()");
    }
}

class Noo extends Moo{
    public void test() {
        System.out.println("Noo test()");
    }
}

```

eg2: 方法动态绑定到运行期间对象的方法 实例 2

```

public class Demo03 {
    public static void main(String[] args) {
        Boo b = new Boo();
    }
}

class Aoo{
    int a=1;
    public Aoo() {this.test(); }
    public void test(){
        System.out.println("Aoo "+a);
    }
}

class Boo extends Aoo{
    int b=2;
    public Boo() { super(); } //默认构造器
    public void test() {
        System.out.println("Boo "+a+","+b);
    }
}

```

11) 为何查阅父类功能，创建子类对象使用功能？

Java 中支持多层继承，也就是一个继承体系。想要使用体系，先查阅父类的描述，因为父类中定义的是该体系中共性的功能，通过了共性功能，就可以知道该体系的基本功能，那么这个体系已经可以基本使用了，然而在具体调用时，要创建最（低）子类的对象，原因

如下：①因为父类有可能不能创建对象②创建子类对象，可以使用更多的功能，包括基本的也包括特有的。

12) 属性无继承概念，所以你有你的，我有我的，各自调用各自的，不影响，即使父子类中有同名属性也无影响。

eg: 子父类同名属性无影响

```
class Base { public static final String FOO="foo"; public static void main(String[] args){  
    Base b=new Base(); Sub s=new Sub();  
    Base.FOO;//foo b.Foo; //foo Sub.Foo;//bar s.Foo;//bar  
    ((Base)s).Foo;// foo } }  
class Sub extends Base{ public static final String FOO="bar"; }
```

3.11 static

静态的，只能在类内部使用，可以修饰：属性，方法，内部类。在类加载期间初始化，存在方法区中。

- 1) 静态成员随着类的加载而加载，加载于方法区中，且优先于对象存在。
- 2) 静态修饰的成员：属于类级别的成员，是全体类实例（所有对象）所共享。
- 3) 静态属性：只有一份（而实例变量是每个对象有一份），全体实例共享，类似于全局变量。
- 4) 使用类名访问静态变量，以及类名直接调用方法，不需要创建对象。
- 5) 静态方法只能访问静态成员（静态属性和静态方法），非静态方法既可访问静态，也可访问非静态。
- 6) 静态方法中没有隐含参数 this，因此不能访问当前对象资源。也不能定义 this 和 super 关键字，因为静态优于对象先存在。
- 7) 非静态方法省略的是 this，静态方法省略的是类名（在同一类中），即直接使用属性和方法。
- 8) 静态方法一般用于与当前对象无关工具方法，工厂方法。如：Math.sqrt() Arrays.sort()
- 9) 静态代码块：随着类的加载而执行（用到类的内容才叫加载，只有引用是不加载的），且只执行一次，且优先于主函数，用于给类初始化。
- 10) 代码块（构造代码块）：给所有对象进行统一初始化，且优先于构造器执行；而构造器是给对应的对象进行初始化。

eg: 静态代码块与代码块（构造代码块）

```
class Goo{  
    //代码块（构造代码块），在创建对象时候执行！类似于构造器的作用  
    System.out.println("HI");  
}  
static{//静态代码块，在类的加载期间执行，只执行一次  
    System.out.println("Loading Goo.class");  
}  
public static void main(String[] args) {  
    Point p1 = new Point(3,4); Point p2 = new Point(6,8);  
    //在对象上调用方法，当前对象隐含传递给隐含参数 this  
    System.out.println(p1.distance(p2));//distance(p1,p2)  
    double d = Point.distance(p1, p2); System.out.println(d); //5
```

```

//静态方法调用时候不传递隐含的当前对象参数
}

class Point{      int x; int y;
    public Point(int x, int y) {      this.x = x;      this.y = y;  }
    //静态方法中没有隐含参数 this! 在静态方法中不能访问 this 的属性和方法!
    public static double distance(Point p1, Point p2){
        int a = p1.x - p2.x;  int b = p1.y - p2.y;  return Math.sqrt(a*a + b*b);
    }
    /** 计算当前点 (this) 到另外一个点 (other) 的距离 */
    public double distance(/*Point this*/ Point other){
        int a = this.x - other.x;      int b = this.y - other.y;
        double c = Math.sqrt(a*a + b*b);      return c;
    }
}

```

11) 对象的创建过程及顺序:

Person P = new Person(“chang”, 23); 这句话都做了什么事情?

- ①因为 new 用到了 Person.class, 所以会先找到 Person.class 文件加载到内存中。
- ②执行该类中的 static 代码块 (如果有的话), 给 Person 类.class 类进行初始化。
- ③在堆内存中开辟空间, 分配内存地址, 栈内存中开辟空间也就有了。
- ④在堆内存中建立对象的特有属性, 并进行默认 (隐式) 初始化。
- ⑤对属性进行显式初始化。
- ⑥对对象进行构造代码块初始化。
- ⑦对对象进行对应的构造器初始化。
- ⑧将内存地址赋给栈内存中的 P 变量。

3.12 final

最终的, 可以修饰: 类、方法、变量 (成员变量和局部变量)。

- 1) final 修饰的类: 不能再继承。
- 2) final 修饰的方法: 不能再重写。
- 3) final 的方法和类, 阻止了动态代理模式! 动态代理模式广泛的应用在: Spring Hibernate Struts2
- 4) 企业编程规范: 不允许使用 final 的方法和类!
- 5) final 的变量: final 变量只能初始化一次 (赋值一次, 且方法中不能有给 final 变量赋值的语句! 因为方法可被调用多次!), 不能再修改! 也可在方法的参数列表中添加 final。

eg1: final 的局部变量

```

final int a;      a = 5;//第一次叫初始化! 不是赋值  //a = 8;//编译错误
public static void test(final int a, int b){
    //a++;//编译错误, 不能再修改
    System.out.println(a);
}

```

eg2: final 的数组

```

final String[] ary={"A","B"};      //ary:数组变量, ary[0]数组元素
ary[0] = "Tom"; //数组元素可以修改
//ary=null; //数组变量不能修改

```

eg3: final 的实例变量

```

public static void main(String[] args) {
    Dog d1 = new Dog();      Dog d2 = new Dog();
    //d1.id = 8; //每个实例的 id 不可以再修改
}

```

```

        System.out.println(d1.id+"."+d2.id)+"."+Dog.numOfDogs);      }
class Dog{    final int id;//实例变量，每个对象一份，不能再次修改
              static int numOfDogs=0;//静态，只有一份
              public Dog() {      id = numOfDogs++;      }          }

```

6) static final 共同修饰的叫常量，常量：public static final double PI = 3.14; PI 是直接数的代名词，是名字。字面量(==直接量)：直接写出数值 3.1415926535897 宏观说：字面量和常量都称为常量！

3.13 多态

继承体现了多态：父类型变量可以引用各种各样的子类型实例，也可接收子类对象。

个体的多态：父类型的子类型实例是多种多样的。

行为的多态：父类型定义方法被子类重写为多种多样的，重载也是多态的方法。

1) 千万不能出现将父类对象转成子类类型，会造型异常！

2) 多态前提：必须是类与类之间有关系。要么继承，要么实现。通常还有一个前提：存在覆盖。

3) 多态的好处：多态的出现大大的提高程序的扩展性。

4) 多态的弊端：虽然提高了扩展性，但是只能使用父类的引用访问父类中的成员。

5) 在多态中成员函数的特点：

①在编译时期：参阅引用型变量所属的类中是否有调用的方法。如果有，编译通过，如果没有编译失败。

②在运行时期：参阅对象所属的类中是否有调用的方法。

③简单总结就是：成员方法在多态调用时，编译看左边，运行看右边。

6) 在多态中，成员变量的特点：无论编译和运行，都参考左边(引用型变量所属的类)。

7) 在多态中，静态成员方法和属性的特点：无论编译和运行，都参考做左边。

8) 父类引用指向子类对象，当父类想使用子类中特有属性、方法时，要向下转型。

3.14 抽象类

抽象就是将拥有共同方法和属性的对象提取出来，提取后，重新设计一个更加通用、更加大众化的类，就叫抽象类。

1) abstract 关键字可以修饰类、方法，即抽象类和抽象方法。

2) 抽象类可以有具体的方法，或者全部都是具体方法，但一个类中只要有一个抽象方法，那么这个类就是抽象类，并且必须用 abstract 修饰类。

3) 抽象类可以被继承，则子类必须实现抽象类中的全部抽象方法，否则子类也将是抽象类。抽象类也可主动继承实体类。

4) 抽象类不能实例化，即不能用 new 生成实例。

5) 可以声明一个抽象类型的变量并指向具体子类的对象。

6) 抽象类可以实现接口中的方法。

7) 抽象类中可以不定义抽象方法，这样做仅仅是不让该类建立对象。

3.15 接口

interface 差不多 == abstract class

1) 接口是 like a：“像”我中的一种，是继承体系之外的，用于功能扩展！想扩展就实现，不想就不用实现。

2) 接口中只能声明抽象方法和常量且声明格式都是固定的，只不过可以省略。

eg: 接口中声明常量和抽象方法的格式是固定的

```
interface Runner {  
    /*public abstract final*/int SPEED=100;//声明常量  
    /*public abstract 省略了，写也对*/void run();//声明抽象方法  
}
```

3) 接口中的成员不写修饰符时，默认都是 public。

4) 接口不能有构造器，因为不能实例化何以初始化，接口只能被“实现”。

5) 具体类实现了一个接口，则必须实现全部的抽象方法，若没有全部实现，则该类为抽象类。所以说，接口约定了具体类的方法，约定了类的外部行为。

6) 具体类可以同时实现多个接口，就是多继承现象。

7) 多重继承： class Cat implements Hunter , Runner Cat 即是 Hunter 也是 Runner。

8) 接口用 implements 表实现，实际是继承关系，可有多个接口(实现)，继承用 extends 只能有一个继承关系。

9) 一个类既可以继承的同时，又“实现”接口： class A extends B implements C , D

10) 类与类之间是继承关系，类与接口之间是实现关系，接口与接口之间是继承关系，且只有接口之间可以多继承，即： interface A{}, interface B{}, interface C extends A , B 但接口多继承时要注意，要避免 A、B 接口中方法名相同、参数列表相同，但返回值类型不相同的情况，因为被具体类实现时，不确定调用哪个方法。

11) abstract class 和 interface 有什么区别。

①从语法角度： abstract class 方法中可以有自己的数据成员，也可以有非 abstract 的成员方法，并赋予方法的默认行为，而在 interface 方式中一般不定义成员数据变量，所有的方法都是 abstract，方法不能拥有默认的行为。

②从编程的角度： abstract class 在 java 语言中表示的是一种继承关系，一个类只能使用一次继承关系。而一个类可以实现多个 interface。

③从问题域角度： abstract class 在 Java 语言中体现了一种继承关系，要想使得继承关系合理，父类和派生类之间必须存在"is a"关系，即父类和派生类在概念本质上应该是相同的。对于 interface 来说则不然，并不要求 interface 的实现者和 interface 定义在概念本质上是一致的，仅仅是实现了 interface 定义的契约而已。

3.16 内部类

当描述事物时，事物的内部还有事物，该事物用内部类来描述。因为内部事物在使用外部事物的内容。

在类内部定义的类为成员内部类，在方法里定义的类为局部内部类，被 static 修饰的为静态内部类。一个类中可有多个内部类。

1) 内部类主要用于，封装一个类的声明在类的内部，减少类的暴露。

2) 内部类的实例化：实例化时不需要写出对象，非要写的话为： new 外部类名. 内部类名(); 而不是外部类名. new 内部类名()。

3) 内部类的访问规则：内部类可以直接访问外部类中的成员，包括私有。之所以可以直接访问外部类中的成员，是因为内部类中持有了一个外部类的引用，格式： 外部类名. This 即下面第 4 条。外部类要访问内部类，必须建立内部类对象。

4) 当内部类定义在外部类的成员位置上，而且非私有，则在外部其他类中可以直接建

立内部类对象。格式：外部类名. 内部类名 变量名 = 外部类对象. 内部类对象；

```
Outer.Inner in = new Outer().new Inner();
```

5) 当内部类在成员位置上，就可以被成员修饰符所修饰。比如 `private`：将内部类在外部类中进行封装。

6) 静态内部类：被 `static` 修饰后就具备了静态的特性。当内部类被 `static` 修饰后，只能直接访问外部类中的 `static` 成员，出现了访问局限。

①在外部其他类中，如何直接访问 static 内部类的非静态成员呢？

```
new Outer.Inner().function();
```

②在外部其他类中，如何直接访问 static 内部类的静态成员呢？

```
Outer.Inner.function();
```

◆ 注意事项：

❖ 当内部类中定义了静态成员，该内部类必须是 `static` 的。

❖ 当外部类中的静态方法访问内部类时，内部类也必须是 `static` 的。

7) 内部类想调用外部类的成员，需要使用：外部类名. `this`. 成员，即 `OuterClassName.this` 表示外部类的对象。如果写 `this. 成员==成员`，调用的还是内部类的成员（属性或方法）。

8) Timer 和 TimerTask：继承 TimerTask 重写 `run()` 方法，再用 Timer 类中的 `schedule` 方法定时调用，就能自动启用 `run()`（不像以前似的要用 `. XXX 调用`）。

eg: 内部类

```
class Xoo{      Timer timer = new Timer();  
    public void start(){  
        timer.schedule(new MyTask(), 0, 1000); //0 表示立即开始，无延迟  
        timer.schedule(new StopTask(), 1000*10); //在 10 秒以后执行一次      }  
    class StopTask extends TimerTask{  
        public void run() { timer.cancel(); } //取消 timer 上的任务      }  
    class MyTask extends TimerTask {  
        int i=10;    public void run() { System.out.println(i--); }      }
```

3.17 匿名类

匿名内部类==匿名类

1) 匿名内部类的格式：`new 父类或者接口{}{定义子类的内容}`；如 `new Uoo(){}` 就叫匿名内部类！是继承于 `Uoo` 类的子类或实现 `Uoo` 接口的子类，并且同时创建了子类型实例，其中 {} 是子类的类体，可以写类体中的成员。

2) 定义匿名内部类的前提：内部类必须是继承一个类或者实现接口。

3) 匿名内部类没有类名，其实匿名内部类就是一个匿名子类对象。而且这个对象有点胖。可以理解为带内容的对象。

4) 在匿名内部类中只能访问 `final` 局部变量。

5) 匿名内部类中定义的方法最好不要超过 3 个。

eg1：匿名内部类的创建

```
public static void main(String[] args) {  
    Uoo u = new Uoo(); //创建 Uoo 实例  
    Uoo u1 = new Uoo(){}; //创建匿名内部类实例  
    Uoo u2 = new Uoo(){
```

```

        public void test() { //方法的重写 System.out.println("u2.test()"); }
        u2.test(); //调用在匿名内部类中重写的方法。
        // new Doo(); 编译错误，不能创建接口实例
        Doo doo = new Doo() { //实现接口，创建匿名内部类实例
            public void test() { //实现接口中声明的抽象方法
                System.out.println("实现 test");
            }
        }
        interface Doo { void test(); }
        class Uoo { public void test() {} }
    }

```

eg2: 匿名内部类中只能访问 final 局部变量

```

final Timer timer=new Timer();
timer.schedule(new TimerTask() {
    public void run() { timer.cancel(); //在匿名内部类中只能访问 final 局部变量
    }
}, 1000*10);

```

6) anonymous Inner Class (匿名内部类) 是否可以 extends(继承)其它类？是否可以 implements(实现)interface(接口)？

匿名内部类是可以继承其它类，同样也可以去实现接口的，用法为：

这样的用法在 swing 编程中是经常使用的，就是因为它需要用到注册监听器机制，而该监听类如果只服务于一个组件，那么将该类设置成内部类/匿名类是最方便的。

3.18 二维数组和对象数组

二维数组（假二维数组），Java 中没有真正的二维数组！Java 二维数组是元素为数组的数组。

对象数组：元素是对象（元素是对象的引用）的数组。

```

Point[] ary; // 声明了数组变量 ary
ary = new Point[3]; // 创建了数组对象
// new Point[3]实际情况: { null,null,null }
// 数组元素自动初始化为 null，并不创建元素对象！
System.out.println(ary[1]); // null
ary[0] = new Point(3, 4); ary[1] = new Point(5, 6); ary[2] = new Point(1, 2);
System.out.println(ary[1]); // 默认调用了对象的 toString()
System.out.println(ary[1].toString()); // 结果上面的一样
// toString 是 Object 类定义，子类继承的方法
// 在输出打印对象的时候，会默认调用，重写这个方法可以打印的更好看！
System.out.println(Arrays.toString(ary)); // 输出 3 个对象
System.out.println(ary.toString()); // 地址值
int[] c = {1, 3, 5, 7};
System.out.println(c[1]);
// System.out.println(c[1].toString()); // 错误，不能在 int 类型调用 toString()

```

3.19 其他注意事项

1) Java 文件规则:

一个 Java 源文件中可以有多个类，但只能有一个公有类！其他类只能是默认类（包中类）而且 Java 的文件夹一定与公有类类名一致！如果没有公有类，可以和任何一个文件名一致。

◆ 一般建议：一个文件一个公有类！一般不在一个文件中写多个类

2) JVM 内存结构堆、栈和方法区分别存储的内容：

JVM 会在其内存空间中开辟一个称为“堆”的存储空间，这部分空间用于存储使用 new 关键字创建的对象。

栈用于存放程序运行过程当中所有的局部变量。一个运行的 Java 程序从开始到结束会有多次方法的调用。JVM 会为每一个方法的调用在栈中分配一个对应的空间，这个空间称为该方法的栈帧。一个栈帧对应一个正在调用中的方法，栈帧中存储了该方法的参数、局部变量等数据。当某一个方法调用完成后，其对应的栈帧将被清除。

方法区该空间用于存放类的信息。Java 程序运行时，首先会通过类装载器载入类文件的字节码信息，经过解析后将其装入方法区。类的各种信息都在方法区保存。

四、Java SE 核心 I

4.1 Object 类

在 Java 继承体系中，`java.lang.Object` 类位于顶端（是所有对象的直接或间接父类）。如果一个类没有写 `extends` 关键字声明其父类，则该类默认继承 `java.lang.Object` 类。`Object` 类定义了“对象”的基本行为，被子类默认继承。

1) `toString` 方法：返回一个可以表示该对象属性内容的字符串。

```
MyObject obj=new MyObject(); String info=obj.toString(); System.out.println(info);
```

A. 上例为什么我有 `toString` 方法？

因为所有的类都继承自 `Object`，而 `toString` 方法是 `Object` 定义的，我们直接继承了这个方法。`Object` 的 `toString` 方法帮我们返回一个字符串，这个字符串的格式是固定的：类型@`hashcode`，这个 `hashcode` 是一串数字，在 java 中叫句柄，或叫地址（但不是真实的物理地址，是 java 自己的一套虚拟地址，防止直接操作内存的）。

```
public String toString(){//只能用 public，重写的方法访问权限要大于等于父类中方法的权限  
    return "这个是我们自己定义的 toString 方法的返回值 MyObject!"; }
```

B. 上例为什么要重写 `toString` 方法？

`toString` 定义的原意是返回能够描述当前这个类的实例的一串文字，我们看一串 `hashcode` 没意义，所以几乎是要重写的。

```
public static void main(String[] args){ //System.out.println(toString());//不行！编译错误！  
    Point p=new Point(1,2); System.out.println(p);//输出 p 对象的 toString 方法返回值 }
```

C. 上例为何有编译错误？

不能直接使用 `toString` 方法，因为该方法不是静态的。ava 语法规规定：静态方法中不能直接引用非静态的属性和方法，想引用必需创建对象。非静态方法中可以直接引用静态属性和方法。

2) `equals` 方法：用于对象的“相等”逻辑。

A. 在 `Object` 中的定义：
`public boolean equals(Object obj){ return (this==obj); }`

由此可见，`this==obj` 与直接的`==`（双等于）效果一样，仅仅是根据对象的地址（句柄，那个 `hashcode` 值）来判断对象是否相等。因此想比较对象与给定对象 内容 是否一致，则必须重写 `equals` 方法。

B. “`==`” 与 `equals` 的区别：

用“`==`”比较对象时，描述的是两个对象是否为同一个对象！**根据地址值** 判断。而 `equals` 方法力图去描述两个对象的内容 是否相等，内容相等取决于业务逻辑需要，可以自行定义比较规则。

C. `equals` 方法的重写：如，判断两点是否相等。

```
public boolean equals(Object obj){//注意参数  
    /**若给定的对象 obj 的地址和当前对象地址一致，那么他们是同一个对象，equals  
    方法中有大量的内容比较逻辑时，加上这个判断会节省性能的开销！ */  
    if(this == obj){ return true; }  
    /** equals 比较前要进行安全验证，确保给定的对象不是 null！若 obj 是 null，说明  
    该引用变量没有指向任何对象，那么就不能引用 obj 所指象的对象（因为对象不存在）的属  
    性和方法！若这么做就会引发 NullPointerException，空指针异常！ */  
    if(obj == null){ return false; }
```

```

    /**直接将 Object 转为子类是存在风险的！我们不能保证 Object 和我们要比较的对象是同一个类型的这会引发 ClassCastException！我们称为：类造型异常。*/
    /**重写 equals 时第一件要做的事情就是判断给定的对象是否和当前对象为同一个类型，不是同类型直接返回 false，因为不具备可比性！*/
    if(!(obj instanceof Point)){ return false; }
    Point p=(Point)obj;
    /**不能随便把父类转成子类，因为 Object 是所有类的父类，任何类型都可以传给它所以不能保证 obj 传进来的就是相同类型（点类型）*/
    return this.x==p.x && this.y==p.y;//内容比较逻辑定义
}

```

4.2 String 类

是字符串类型，是引用类型，是“不可变”字符串，无线程安全问题。在 `java.lang.String` 中。

- ◆ 注意事项：`String str="abc";`和`String str=new String("abc");`的区别！

1) `String` 在设计之初，虚拟机就对他做了特殊的优化，将字符串保存在虚拟机内部的字符串常量池中。一旦我们要创建一个字符串，虚拟机先去常量池中检查是否创建过这个字符串，如有则直接引用。`String` 对象因为有了上述的优化，就要保证该对象的内容自创建开始就不能改变！所以对字符串的任何变化都会创建新的对象，而不是影响以前的对象！

2) `String` 的 `equals` 方法：两个字符串进行比较的时候，我们通常使用 `equals` 方法进行比较，字符串重写了 `Object` 的 `equals` 方法，用于比较字符串内容是否一致。虽然 java 虚拟机对字符串进行了优化，但是我们不能保证任何时候 “`==`” 都成立！

3) 编程习惯：当一个字符串变量和一个字面量进行比较的时候，用字面量 `.equals` 方法去和变量进行比较，即：`if("Hello".equals(str))`因为这样不会产生空指针异常。而反过来用，即：`if(str.equals("Hello"))`则我们不能保证变量不是 `null`，若变量是 `null`，我们在调用其 `equals` 方法时会引发空指针异常，导致程序退出。若都为变量则 `if(str!=null&&str.equals(str1))`也可。

4) `String` 另一个特有的 `equals` 方法：`euqalsIgnoreCase`，该方法的作用是忽略大小写比较字符串内容，常用环境：验证码。`if("hello".equalsIgnoreCase(str))`。

5) `String` 的基本方法：

- ①`String toLowerCase()`: 返回字符串的小写形式。如：`str.toLowerCase()`
- ②`String toUpperCase()`: 返回字符串的大写形式。如：`str.toUpperCase()`
- ③`String trim()`: 去掉字符串两边的空白（空格\t\n\r），中间的不去。如：`str.trim()`
- ④`boolean startsWith()`: 判断字符串是否以参数字符串开头。如：`str.startsWith("s")`
- ⑤`boolean endsWith()`: 判断字符串是否以参数字符串结尾。如：`str.endsWith("s")`
- ⑥`int length()`: 返回字符串字符序列的长度。如：`str.length()`

`eg:` 如何让 `HelloWorld` 这个字符串以 `hello` 开头成立

```
if(str.toLowerCase().startsWith("hello"))//有返回值的才能继续 . 先转成小写再判断
```

6) `indexOf` 方法（检索）：位置都是从 0 开始的。

- ①`int indexOf(String str)`: 在给定的字符串中检索 `str`，返回其第一次出现的位置，找不到则返回-1。
- ②`int indexOf(String str,int from)`: 在给定的字符串中从 `from` 位置开始检索 `str`，返回其第一次出现的位置，找不到则返回-1（包含 `from` 位置，`from` 之前的位置不看）。

`eg:` 查找 `Think in Java` 中 `in` 后第一个 `i` 的位置

```
index=str.indexOf("in");           index=str.indexOf("i",index+"in".length());
```

```
//这里对 from 参数加 in 的长度的目的是 从 in 之后的位置开始查找
```

③int lastIndexOf(String str): 在给定的字符串中检索 str, 返回其最后一次出现的位置, 找不到则返回-1 (也可认为从右往左找, 第一次出现的位置)。

④int lastIndexOf(String str,int from): 在给定的字符串中从 from 位置开始检索 str, 返回其最后一次出现的位置, 找不到则返回-1 (包含 from 位置, from 之后的不看)。

7) charAt 方法: char charAt(int index): 返回字符串指定位置 (index) 的字符。

eg: 判断是否是回文: 上海自来水来自海上

```
boolean tf=true;    for(int i=0;i<str2.length()/2;i++){
    //char first=str2.charAt(i);    //char last=str2.charAt(str2.length()-i-1);
    if(str2.charAt(i)!=str2.charAt(str2.length()-i-1)){//优化
        tf = false;      break;//已经不是回文了, 就没有必要再继续检查了    }
}
```

8) substring 方法 (子串): 字符串的截取, 下标从 0 开始的。

①String substring(int start,int end): 返回下标从 start 开始 (包含) 到 end 结束的字符串 (不包含)。

②String substring(int start): 返回下标从 start 开始 (包含) 到结尾的字符串。

9) getBytes 方法 (编码): 将字符串转换为相应的字节。

①byte[] getBytes(): 以当前系统默认的字符串编码集, 返回字符串所对应的二进制序列。如: byte[] array=str.getBytes(); System.out.println(Arrays.toString(array));

②byte[] getBytes(String charsetName): 以指定的字符串编码集, 返回字符串所对应的二进制序列。这个重载方法需要捕获异常, 这里可能引发没有这个编码集的异常, UnsupportedEncodingException, 如: str="常"; byte[] bs=info.getBytes("UTF-8");

◆ 注意事项:

- ❖ Windows 的默认编码集 GBK: 英文用 1 个字节描述, 汉字用 2 个字节描述; ISO-8859-1 欧洲常用编码集: 汉字用 3 个字节描述; GBK 国标; GB2312 国标; UTF-8 编码集是最常用的: 汉字用 3 个字节描述。

- ❖ 编码: 将数据以特定格式转换为字节; 解码: 将字节以特定格式转换为数据。

- ❖ String(byte[] bytes, String charsetName) : 通过使用指定的 charset 解码指定的 byte 数组, 构造一个新的 String。如: String str=new String(bs,"UTF-8");

10) split 方法 (拆分): 字符串的拆分。

String[] split(String regex): 参数 regex 为正则表达式, 以 regex 所表示的字符串为分隔符, 将字符串拆分成字符串数组。其中, regex 所表示的字符串不被保留, 即不会存到字符串数组中, 可理解为被一刀切, 消失!

eg: 对图片名重新定义, 保留图片原来后缀

```
String name="me.jpg";           String[] nameArray=name.split("\\.");
                                //以正则表达式拆分 .有特殊含义, 所以用\\. 转义
System.out.println("数组长度: "+nameArray.length);//如果不用\\.则长度为0
System.out.println(Arrays.toString(nameArray));//任意字符都切一刀, 都被切没了
String newName="123497643."+nameArray[1];
System.out.println("新图片名: "+newName);
```

◆ 注意事项: 分隔符放前、中都没事, 放最后将把无效内容都忽略。

```
String str="123,456,789,456,,";
String[] array=str.split(",");//分隔符放前、中都没事, 放最后将把无效内容都忽略
System.out.println(Arrays.toString(array));//[123, 456, 789, 456]
```

11) replace 方法: 字符串的替换。

String replaceAll(String regex, String replacement): 将字符串中匹配正则表达式 regex 的字符串替换成 replacement。如: String str1=str.replaceAll("[0-9]+", "chang");

12) String.valueOf()方法: 重载的静态方法, 用于返回各类型的字符串形式。

```
String.valueOf(1); //整数, 返回字符串 1 String.valueOf(2.1); //浮点数, 返回字符串 1.2
```

4.3 StringUtils 类

针对字符串操作的工具类, 提供了一系列静态方法, 在 Apache 阿帕奇 Commons-lang 包下中, 需下载。

StringUtils 常用方法:

- 1) String repeat(String str,int repeat): 重复字符串 repeat 次后返回。
- 2) String join(Object[] array,String): 将一个数组中的元素连接成字符串。
- 3) String leftPad(String str,int size,char padChar): 向左边填充指定字符 padChar, 以达到指定长度 size。
- 4) String rightPad(String str,int size,char padChar): 向右边填充指定字符 padChar, 以达到指定长度 size。

4.4 StringBuilder 类

与 String 对象不同, StringBuilder 封装“可变”的字符串, 有线程安全问题。对象创建后, 可通过调用方法改变其封装的字符序列。

StringBuilder 常用方法:

- 1) 追加字符串: StringBuilder append(String str);
- 2) 插入字符串: StringBuilder insert(int index,String str): 插入后, 原内容依次后移
- 3) 删除字符串: StringBuilder delete(int start,int end);
- 4) 替换字符串: StringBuilder replace(int start,int end,String str): 含头不含尾
- 5) 字符串反转: StringBuilder reverse();

eg: 各类操作

```
StringBuilder builder=new StringBuilder();
builder.append("大家好! ") .append("好好学习") .append("天天向上");
                                //返回的还是自己: builder, 所以可以再 .
System.out.println(builder.toString());
builder.insert(4, " ! ");
System.out.println(builder.toString());
builder.replace(5,9,"Good Good Study!"); System.out.println(builder.toString());
builder.delete(9, builder.length()); System.out.println(builder.toString());
```

◆ 注意事项:

- ❖ 该类用于对某个字符串频繁的编辑操作, 使用 StringBuilder 可以在大规模修改字符串时, 不开辟新的字符串对象, 从而节约内存资源, 所以, 对有着大量操作字符串的逻辑中, 不应使用 String 而应该使用 StringBuilder。
- ❖ append 是有返回值的, 返回类型是 StringBuilder, 而返回的 StringBuilder 其实就是自己 (this), append 方法的最后一句是 return this;
- ❖ StringBuilder 与 StringBuffer 区别: 效果是一样的。

StringBuilder 是线程不安全的, 效率高, 需 JDK1.5+。

StringBuffer 是线程安全的, 效率低, “可变” 字符串。

在多线程操作的情况下应使用 StringBuffer, 因为 StringBuffer 是线程安全

的，他难免要顾及安全问题，而进行必要的安全验证操作。所以效率上要比 StringBuilder 低，根据实际情况选择。

4.5 正则表达式

实际开发中，经常需要对字符串数据进行一些复杂的匹配、查找、替换等操作，通过正则表达式，可以方便的实现字符串的复杂操作。

正则表达式是一串特定字符，组成一个“规则字符串”，这个“规则字符串”是描述文本规则的工具，正则表达式就是记录文本规则的代码。

[]	表示一个字符
[abc]	表示 a、b、c 中任意一个字符
[^abc]	除了 a、b、c 的任意一个字符
[a-z]	表示 a 到 z 中的任意一个字符
[a-zA-Z0-9_]	表示 a 到 z、A 到 Z、0 到 9 以及下滑线中的任意一个字符
[a-zA-Z0-9_][^bc]	表示 a 到 z 中除了 b、c 之外的任意一个字符，&& 表示“与”的关系
.	表示任意一个字符
\d	任意一个数字字符，相当于 [0-9]
\D	任意一个非数字字符，相当于 [^0-9]
\s	空白字符，相当于 [\t\n\f\r\x0B]
\S	非空白字符，相当于 [^s]
\w	任意一个单词字符，相当于 [a-zA-Z0-9_]
\W	任意一个非单词字符，相当于 [^w]
^	表示字符串必须以其后面约束的内容开始
\$	表示字符串必须以其前面约束的内容结尾
?	表示前面的内容出现 0 到 1 次
*	表示前面的内容出现 0 到多次
+	表示前面的内容出现 1 到多次
{n}	表示前面的字符重复 n 次
{n,}	表示前面的字符至少重复 n 次
{n,m}	表示前面的字符至少重复 n 次，并且小于 m 次 X>=n && X<m

◆ 注意事项：

- ❖ 邮箱格式的正则表达式 @无特殊含义，可直接写，也可[@]
- ❖ 使用 Java 字符串去描述正则表达式的时候，会出现一个冲突，即如何正确描述正则表达式的“.”。

起因：在正则表达式中我们想描述一个“.”，但“.”在正则表达式中有特殊含义，他代表任意字符，所以我们在正则表达式中想描述“.”的原义就要写成“\.”但是我们用 java 字符串去描述正则表达式的时候，因为“.”在 java 字符串中没有特殊意义，所以 java 认为我们书写 String s="\. "是有语法错误的，因为“.”不需要转义，这就产生了冲突。

处理：我们实际的目的很简单，就是要让 java 的字符串描述“\.”又因为在 java 中“\”是有特殊含义的，代表转义字符我们只需要将“\”转义为单纯的斜杠，即可描述“\.”了所以我们用 java 描述“\.”的正确写法是 String s="\\. ";

- ❖ 若正则表达式不书写^或\$，正则表达式代表匹配部分内容，都加上则表示权匹配

eg: 测试邮箱正则表达式：Pattern 的作用是描述正则表达式的格式支持，使用静态方法

compile 注册正则表达式，生成实例。

```
String regStr="^@[a-zA-Z0-9_]+@[a-zA-Z0-9]+\(\.\com|\.\cn|\.\net)+$";
Pattern pattern=Pattern.compile(regStr);//注册正则表达式
String mailStr="chang_2013@chang.com.cn";
//匹配字符串，返回描述匹配结果的 Matcher 实例
Matcher matcher=pattern.matcher(mailStr);
//通过调用 Matcher 的 find 方法得知是否匹配成功
if(matcher.find()){ System.out.println("邮箱匹配成功！"); }
else{ System.out.println("邮箱格式错误！"); }
```

4.6 Date 类

java.util.Date 类用于封装日期及时间信息，一般仅用它显示某个日期，不对它作任何操作处理，作处理用 Calendar 类，计算方便。

```
Date date=new Date();//创建一个 Date 实例，默认的构造方法创建的日期代表当前系统时间
System.out.println(date);//只要输出的不是类名@hashcode 值，就说明它重写过 toString()
long time=date.getTime();//查看 date 内部的毫秒值
date.setTime(time+1000*60*60*24);//设置毫秒数让一个时间 Date 表示一天后的当前时间
int year=date.getYear();//画横线的方法不建议再使用：1、有千年虫问题。2、不方便计算
```

4.7 Calendar 类

java.util.Calendar 类用于封装日历信息，其主要作用在于其方法可以对时间分量进行运算。

1) 通过 Calendar 的静态方法获取一个实例该方法会根据当前系统所在地区来自行决定时区，帮我们创建 Calendar 实例，这里要注意，实际上根据不同的地区，Calendar 有若干个子类实现。而 Calendar 本身是抽象类，不能被实例化！我们不需要关心创建的具体实例为哪个子类，我们只需要根据 Calendar 规定的方法来使用就可以了。

2) 日历类所解决的根本问题是简化日期的计算，要想表示某个日期还应该使用 Date 类描述。Calendar 是可以将其描述的时间转化为 Date 的，我们只需要调用其 getTime()方法就可以获取描述的日期的 Date 对象了。

3) 通过日历类计算时间：为日历类设置时间，日历类设置时间使用通用方法 set。

set(int field,int value), field 为时间分量，Calendar 提供了相应的常量值，value 为对应的值。

4) 只有月份从 0 开始：0 为 1 月，以此类推，11 为 12 月，其他时间是正常的从 1 开始。也可以使用 Calendar 的常量 calendar.NOVEMBER……等。

5) Calendar.DAY_OF_MONTH 月里边的天---号；

Calendar.DAY_OF_WEEK 星期里的天---星期几

Calendar.DAY_OF_YEAR 年里的天

```
Calendar calendar=Calendar.getInstance();//构造出来表示当前时间的日历类
Date now=calendar.getTime();//获取日历所描述的日期
calendar.set(Calendar.YEAR, 2012);//设置日历表示 2012 年
calendar.set(Calendar.DAY_OF_MONTH,15);//设置日历表示 15 号
calendar.add(Calendar.DAY_OF_YEAR, 22);//想得到 22 天以后是哪天
calendar.add(Calendar.DAY_OF_YEAR, -5);//5 天以前是哪天
calendar.add(Calendar.MONTH, 1);得到 1 个月后是哪天
System.out.println(calendar.getTime());
```

6) 获取当前日历表示的日期中的某个时间单位可以使用 get 方法。

```

int year=calendar.get(Calendar.YEAR);
int month=calendar.get(Calendar.MONTH);
int day=calendar.get(Calendar.DAY_OF_MONTH);
System.out.println(year+"年"+(month+1)+"月"+day+"日");//month 要处理

```

LICHOO

4.8 SimpleDateFormat 类

java.text.SimpleDateFormat 类，日期转换类，该类的作用是可以很方便的在字符串和日期类之间相互转换。

1) 这里我们在字符串与日期类间相互转换是需要一些约束的，"2012-02-02"这个字符串如何转换为 Date 对象？Date 对象又如何转为字符串？

parse 方法用于按照特定格式将表示时间的字符串转化成 Date 对象。

format 方法用于将日期数据（对象）按照指定格式转为字符串。

2) 常用格式字符串

字符	含义	示例
y	年	yyyy 年-2013 年; yy 年-13 年
M	月	MM 月-01 月; M 月-1 月
d	日	dd 日-06 日; d 日-6 日-
E	星期	E-星期日 (Sun)
a	AM 或 PM	a-下午 (PM)
H	24 小时制	a h 时-小午 12 时
h	12 小时制	HH: mm: ss-12: 46: 33
m	分钟	hh(a): mm: ss-12 (下午): 47: 48
s	秒	

eg: 字符串转成 Date 对象

```

//创建一个 SimpleDateFormat 并且告知它要读取的字符串格式
SimpleDateFormat sdf=new SimpleDateFormat("yyyy-MM-dd");
String dateFormat="2013-05-14";//创建一个日期格式字符串
//将一个字符串转换为相应的 Date 对象
Date date=sdf.parse(dateFormat);//要先捕获异常
System.out.println(date);//输出这个 Date 对象

```

eg: Date 对象转成字符串

```

SimpleDateFormat sdf=new SimpleDateFormat("yyyy-MM-dd");
Date now=new Date();      String nowStr=sdf.format(now);//把日期对象传进去

```

3) 在日期格式中 - 和 空格 无特殊意义，无特殊含义的都将原样输出。

eg: 将时间转为特定格式

```

//将当前系统时间转换为 2012/05/14 17:05:22 的效果
SimpleDateFormat format1=new SimpleDateFormat("yyyy/MM/dd hh:mm:ss");
nowStr=format1.format(now);      System.out.println(nowStr);

```

4.9 DateFormat 类

java.text.DateFormat 类（抽象类）是 SimpleDateFormat 类的父类，用的少，没 SimpleDateFormat 灵活。

创建用于将 Date 对象转换为日期格式的字符串的 DateFormat，创建 DateFormat 对象的实例，使用静态方法 getDateInstance(style,aLocale)，style 为输出日期格式的样式：DateFormat

有对应的常量；`aLocale` 为输出的地区信息，影响字符串的语言和表现形式。

```
Date now=new Date();    DateFormat format=DateFormat.getDateInstance(  
                           DateFormat.MEDIUM, Locale.CHINA);
```

4.10 包装类

Java 语言的 8 种基本类型分别对应了 8 种“包装类”。每一种包装类都封装了一个对应的基本类型成员变量，同时还提供了针对该数据类型的实用方法。

- 1) 包装类的目的：用于将基本类型数据当作引用类型看待。
- 2) 包装类的名字：除了 `Integer(int)`, `Character(char)` 外，其余包装类名字都是基本类型名首字母大写。
- 3) 拆、装箱：`Integer i=new Integer(1);` 创建一个以对象形式存在的整数 1，这种从基本类型转为引用类型的过程称之为“装箱”，反之叫“拆箱”。
- 4) 装箱：方式一：`Double d=new Double(2.2);`//装箱
 方式二：`Double d=Double.valueOf(2.2);`//基本类型都有 `valueOf` 方法
- 5) 拆箱：`double num=d.doubleValue();`//拆箱
- 6) 包装类使用前提：JDK1.5+

```
public static void say(Object obj){    System.out.println(obj);    }  
int a=1;//基本类型，不是 Object 子类！  
say(a);//在 java 1.4 版本的时候，这里还是语法错误的！因为 int 是基本类型，不是 Object 对象，要自己写 8 种基本类型对应的方法
```

- 7) 包装类的使用：实例化一个对象，该对象代表整数 1；`Integer` 的作用是让基本类型 `int` 作为一个引用类型去看待。这样就可以参与到面向对象的编程方式了。由此我们可以将一个 `int` 当作一个 `Object` 去看待了，也成为了 `Object` 的子类。

```
Integer i=new Integer(a);//装箱，或者写 Integer i=new Integer(1);  
Integer ii=Integer.valueOf(a);//装箱另一种方式  
int num=i.intValue();//拆箱    say(i);//Integer 是 Object 的子类，可以调用！
```

- 8) JDK1.5 包装类自动拆装箱（原理）：在编译源程序的时候，编译器会预处理，将未作拆箱和装箱工作的语句自动拆箱和装箱。可通过反编译器发现。

```
say(Integer.valueOf(a));自动装箱    num=i;//引用类型变量怎么能复制给基本类型呢?  
//num=i.intValue();自动拆箱
```

- 9) 包装类的一些常用功能：将字符串转换为其类型，方法是：`parseXXX`，`XXX` 代表其类型。这里要特别注意！一定要保证待转换的字符串描述的确实是或者兼容要转换的数据类型！否则会抛出异常！

```
String numStr="123";    System.out.println(numStr+1);//1231  
int num=Integer.parseInt(numStr);    System.out.println(num+1)//124  
long longNum=Long.parseLong(numStr);  System.out.println(longNum);//123  
double doubleNum=Double.parseDouble(numStr);  System.out.println(doubleNum);//123.0
```

- 10) `Integer` 提供了几个有趣的方法：将一个整数转换为 16 进制的形式，并以字符串返回；将一个整数转换为 2 进制的形式，并以字符串返回。

```
String bStr=Integer.toBinaryString(num);    String hStr=Integer.toHexString(num);
```

- 11) 所有包装类都有几个共同的常：获取最大、最小值。

```
int max=Integer.MAX_VALUE;//int 最大值    int min=Integer.MIN_VALUE;//int 最小值  
System.out.println(Integer.toBinaryString(max)); System.out.println(Integer.toBinaryString(min));
```

4.11 BigDecimal 类

表示精度更高的浮点型，在 `java.math.BigDecimal` 包下，该类可以进行更高精度的浮点运算。需要注意的是，`BigDecimal` 可以描述比 `Double` 还要高的精度，所以在转换为基本类型时，可能会丢失精度！

1) `BigDecimal` 的使用：创建一个 `BigDecimal` 实例，可以使用构造方法 `BigDecimal(String numberFormatString)` 用字符串描述一个浮点数作为参数传入。

```
BigDecimal num1=new BigDecimal("3.0");
BigDecimal num2=new BigDecimal("2.9"); //运算结果依然为 BigDecimal 表示的结果
BigDecimal result=num1.subtract(num2); //num1-num2
System.out.println(result);
float f=result.floatValue(); //将输出结果转换为基本类型 float
int i=result.intValue(); //将输出结果转换为基本类型 int
```

2) `BigDecimal` 可以作加 `add`、减 `subtract`、乘 `multiply`、除 `divide` 等运算：这里需要注意除法，由于除法存在结果为无限不循环小数，所以对于除法而言，我们要制定取舍模式，否则会一直计算下去，直到报错（内存溢出）。

```
result=num1.divide(num2,8,BigDecimal.ROUND_HALF_UP);
//小数保留 8 位，舍去方式为四舍五入
```

4.12 BigInteger 类

使用描述更长位数的整数“字符串”，来表示、保存更长位数的整数，在 `java.math.BigInteger` 包下。

1) `BigInteger` 的使用：创建 `BigInteger`

```
BigInteger num=new BigInteger("1"); //不可以，没有这样的构造器
//这种方式我们可以将一个整数的基本类型转换为 BigInteger 的实例
num=BigInteger.valueOf(1);
```

2) 理论上：`BigInteger` 存放的整数位数只受内存容量影响。

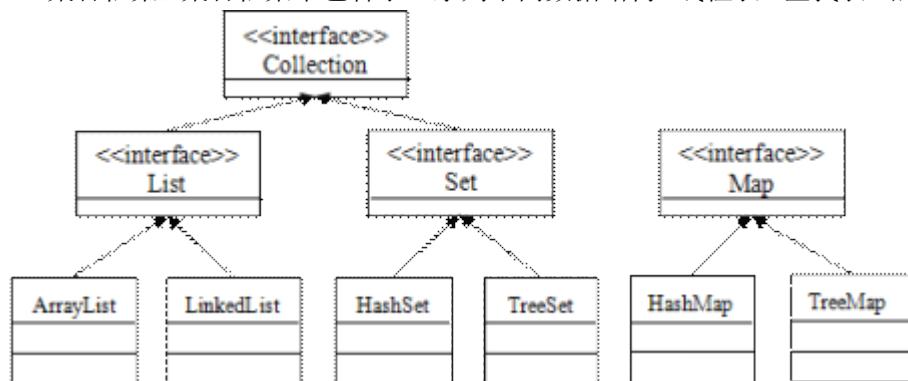
3) `BigInteger` 同样支持加 `add`、减 `subtract`、乘 `multiply`、除 `divide` 等运算。

eg: 1-200 的阶乘

```
for(int i=1;i<=200;i++){
    num=num.multiply(BigInteger.valueOf(i));
}
System.out.println("结果"+num.toString().length()+"位");
System.out.println(num);
```

4.13 Collection 集合框架

在实际开发中，需要将使用的对象存储于特定数据结构的容器中。而 JDK 提供了这样的容器——集合框架，集合框架中包含了一系列不同数据结构（线性表、查找表）的实现类。



1) `Collection` 常用方法：

- ①int size(): 返回包含对象个数。
- ②boolean isEmpty(): 返回是否为空。
- ③boolean contains(Object o): 判断是否包含指定对象。
- ④void clear(): 清空集合。
- ⑤boolean add(E e): 向集合中添加对象。
- ⑥boolean remove(Object o): 从集合中删除对象。
- ⑦boolean addAll(Collection<? extends E> c): 另一个集合中的所有元素添加到集合
- ⑧boolean removeAll(Collection<?> c): 删除集合中与另外一个集合中相同的原素
- ⑨Iterator<E> iterator(): 返回该集合的对应的迭代器

2) Collection 和 Collections 的区别

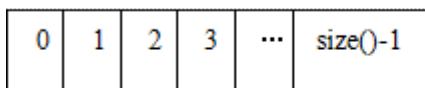
Collection 是 java.util 下的接口，它是各种集合的父接口，继承于它的接口主要有 Set 和 List；Collections 是个 java.util 下的类，是针对集合的帮助类，提供一系列静态方法实现对各种集合的搜索、排序、线程安全化等操作。

4.14 List 集合的实现类 ArrayList 和 LinkedList

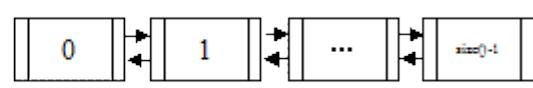
List 接口是 Collection 的子接口，用于定义线性表数据结构，元素可重复、有序的；可以将 List 理解为存放对象的数组，只不过其元素个数可以动态的增加或减少。

1) List 接口的两个常见的实现类：ArrayList 和 LinkedList，分别用动态数组和链表的方式实现了 List 接口。List、ArrayList 和 LinkedList 均处于 java.util 包下。

2) 可以认为 ArrayList 和 LinkedList 的方法在逻辑上完全一样，只是在性能上有一定的差别，ArrayList 更适合于随机访问，而 LinkedList 更适合于插入和删除，在性能要求不是特别苛刻的情形下可以忽略这个差别。



ArrayList



LinkedList

3) 使用 List 我们不需要在创建的时候考虑容量集合的容量是根据其所保存的元素决定的换句话说，集合的容量是可以自动扩充的。

4) List 的实现类会重写 toString 方法，依次调用所包含对象的 toString 方法，返回集合中所包含对象的字符串表现。

5) 常用方法：

①add(Object obj): 向想集合末尾追加一个新元素，从该方法的参数定义不难看出，集合可以存放任意类型的元素，但在实际编程中我们发现，几乎不会向集合中存放一种以上的不同类型的元素。

②size()方法：返回当前集合中存放对象的数量。

③clear()方法：用于清空集合。

④isEmpty()方法：用于返回集合是否为空。

```

List list=new ArrayList(); list.add("One"); list.add("Two"); list.add("Three");
//list.add(1); //不建议这样的操作！尽量不在同一个集合中存放不用类型元素
System.out.println("集合中元素的数量: "+list.size());
System.out.println(list); //System.out.println(list.toString());
//ArrayList 重写了 toString()方法返回的字符串是每个元素的 toString()返回值的序
列
list.clear(); //清空 System.out.println("清空后元素的数量: "+list.size());
System.out.println("集合是否为空？ : "+list.isEmpty());

```

⑤contains(Object obj)方法：检查给定对象是否被包含在集合中，检查规则是将 obj

对象与集合中每个元素进行 equals 比较，若比对了所有元素均没有 equals 为 true 的则返回 false。

- ◆ 注意事项：根据情况重写 equals：若比较是否是同一个对象，则不需要重写，直接用 contains 里的 equals 比较即可。若重写 equals 为内容是否相同，则按内容比较，不管是否同一个对象。是否重写元素的 equals 方法对集合的操作结果有很大的效果不同！

⑥boolean remove(Object obj)方法：删除一个元素，不重写 equals，不会有元素被删除（因为比较的是对象的地址，都不相同），重写 equals 为按内容比较，则删除第一个匹配的就退出，其他即使内容相同也不会被删除。

```
List list=new ArrayList();  
//多态的写法 //ArrayList arrayList=new ArrayList();  
list.add(new Point(1,2));  
//Point point =new Point(1,2);    list.add(point);等量代换  
list.add(new Point(3,4));      list.add(new Point(5,6));  
System.out.println("集合中元素的数量: "+list.size());    System.out.println(list);  
Point p=new Point(1,2);  
//创建一个 Point 对象  
System.out.println("p 在集合中存在么? "+list.contains(p));  
//不重写为 false 重写为 true  
System.out.println("删前元素: "+list.size());  
list.remove(p);  
//将 p 对象删除，不重写 equals，不会有元素被删除  
System.out.println("删后元素: "+list.size());    System.out.println(list);
```

⑦E remove(int index)方法：移除此列表中指定位置上的元素。向左移动所有后续元素（将其索引减 1）。因此在做删除操作时集合的大小为动态变化的，为了防止漏删，必须从后往前删！

```
ArrayList list=new ArrayList();  
list.add("java");    list.add("aaa");  
list.add("java");    list.add("java");    list.add("bbb");  
//相邻的元素删不掉!  
for(int i=0;i<list.size();i++){  if( "java".equals(list.get(i))) list.remove(i);      }  
//可以删干净!  
for(int i=list.size()-1;i>=0;i-){  if("java".equals(list.get(i))){ list.remove(i);      }}
```

⑧addAll(Collection c)方法：允许将 c 对应的集合中所有元素存入该集合，即并集。注意，这里的参数为 Collection，所以换句话说，任何集合类型都可以将其元素存入其他集合中！

⑨removeAll(Collection c)方法：删除与另一个集合中相同的元素。它的“相同”逻辑通过 equals 方法来判断。

⑩retainAll(Collection c)方法：保留与另一个集合中相同的元素，即交集。它的“相同”逻辑通过 equals 方法来判断。

```
list1.addAll(list2);  
//并集  
list1.removeAll(list3);  
//从 list1 中删除 list3 中相同>equals 为 true 的元素  
list1.retainAll(list2);  
//保留 list1 中删除 list2 中相同>equals 为 true 的元素
```

⑪Object get(int index)方法：根据元素下标获取对应位置的元素并返回，这里元素的下标和数组相似。

⑫Object set(int index, Object newElement)方法：将 index 位置的元素修改为 newElement 修改后会将被修改的元素返回。因此，可实现将 List 中第 i 个和第 j 个元素交换的功能：list.set(i, list.set(j, list.get(i)));

⑬add(int index, Object newElement)方法：使用 add 的重载方法，我们可以向 index 指定位置插入 newElement，原位置的元素自动向后移动，即所谓的“插队”。

⑭Object remove(int index)方法：将集合中下标为 index 的元素删除，并将被删除的元素返回（不根据 equals，根据下标删除元素）。

```
List list=new ArrayList(); list.add("One"); list.add("Two"); list.add("Three");
//因为 get 方法是以 Object 类型返回的元素，所以需要造型，默认泛型 Object
String element=(String)list.get(2);//获取第三个元素 System.out.println(element);
for(int i=0;i<list.size();i++){//遍历集合 System.out.println(list.get(i)); }
Object old=list.set(2, "三");
System.out.println("被替换的元素: "+old); System.out.println(list);
list.add(2, "二");//在 Two 与 “三” 之间插入一个“二”
System.out.println(list);
Object obj=list.remove(1); System.out.println("被删除的元素: "+obj);
```

⑮indexOf(Object obj)方法：用于在集合中检索对象，返回对象第一次出现的下标。

⑯lastIndexOf(Object obj)方法：用于在集合中检索对象，返回对象最后一次出现的下标。

⑰Object[] toArray()方法：该方法继承自 Collection 的方法，该方法会将集合以对象数组的形式返回。

⑱toArray()的重载方法，T[] toArray(T[] a): 可以很方便的让我们转换出实际的数组类型，如下例，参数 new Point[0] 的作用是作为返回值数组的类型，所以参数传入的数组不需要有任何长度，因为用不到，就没有必要浪费空间。

```
Object[] array=list.toArray();//将集合以对象数组的形式返回
for(int i=0;i<array.length;i++){ Point p=(Point)array[i]; System.out.println(p.getX()); }
Point[] array1=(Point[])list.toArray(new Point[0]);//toArray()的重载方法
for(int i=0;i<array1.length;i++){ Point p=array1[i];//不需要每次都强转了
System.out.println(p.getX()); }
```

⑲List<E> subList(int fromIndex, int toIndex)方法：获取子集合，但在获取子集后，若对子集合的元素进行修改，则会影响原来的集合。

```
List<Integer> list=new ArrayList<Integer>();
for(int i=0;i<10;i++){ list.add(i); }
List<Integer> subList=list.subList(3, 8);//[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] 取子集(3-7)
System.out.println(subList);//[3, 4, 5, 6, 7]
//我们在获取子集后，若对自己元素进行修改，会影响原来的集合
for(int i=0;i<subList.size();i++){
    int element=subList.get(i);
    element*=10; subList.set(i, element);
    //subList.set(i, subList.get(i)*10);同上三步
}
System.out.println(list);//原集合内容也被修改了
```

⑳Comparable 接口：针对对象数组或者集合中的元素进行排序时，首选需要确定对象元素的“比较”逻辑（即哪个大哪个小）。Comparable 接口用于表示对象间的大小关系，我们需要实现 Comparable 接口，并重写 compareTo() 方法定义比较规则。

```
public int compareTo(ComparablePoint o){ int r=x*x+y*y;//自身点到原点的距离
int other=o.x*x+o.y*y;//参数点到原点的距离
//返回结果大于 0，自身比参数大；小于 0，自身比参数小；等于 0，自身和参数相等；
//equals 返回 true 的时候，compareTo 的返回值应该为 0
return r-other; }
```

㉑Collections.sort()方法：需要集合中的对象实现 Comparable 接口，从而可以调用

其 compareTo 方法判断对象的大小，否则 sort 将无法判断。该方法会依次调用集合中每个元素的 compareTo 方法，并进行自然排序（从小到大）。

```
List<ComparablePoint> list=new ArrayList<ComparablePoint>();  
list.add(new ComparablePoint(1,5));    list.add(new ComparablePoint(3,4));  
list.add(new ComparablePoint(2,2));  System.out.println(list);//输出顺序与存方时一致  
Collections.sort(list);      System.out.println(list);
```

②Comparator 接口：比较器。一旦 Java 类实现了 Comparable，其比较逻辑就已经确定了，如果希望在排序中的操作按照“临时指定规则”，即自定义比较规则。可以采用 Comparator 接口回调方式。

Comparator 比较器创建步骤：A. 定义一个类并实现 Comparator 接口。B. 实现接口中的抽象方法 compare(E o1,E o2)。C. 实例化这个比较器 D. 调用 Collections 的重载方法：sort(Collection c,Comparator comparator)进行排序。通常使用匿名类方式创建一个实例来定义比较器。

```
Comparator<ComparablePoint> c=new Comparator<ComparablePoint>(){  
    public int compare(ComparablePoint o1,ComparablePoint o2){  
        return o1.getX()-o2.getX(); //两个点的 X 值大的大    }  
    Collections.sort(list, c);  System.out.println(list);
```

4.15 Iterator 迭代器

所有 Collection 的实现类都实现了 iterator 方法，该方法返回一个 Iterator 接口类型的对象，用于实现对集合元素迭代的便利。在 java.util 包下。

1) Iterator 定义有三个方法：

- ①boolean hasNext()方法：判断指针后面是否有元素。
- ②E next()方法：指针后移，并返回当前元素。E 代表泛型，默认为 Object 类型。
- ③void remove()方法：在原集合中删除刚刚返回的元素。

2) 对于 List 集合而言，可以通过基于下标的 get 方法进行遍历；而 iterator 方法是针对 Collection 接口设计的，所以，所有实现了 Collection 接口的类，都可以使用 Iterator 实现迭代遍历。

3) 迭代器的使用方式：先问后拿。问：boolean hasNext() 该方法询问迭代器当前集合是否还有元素；拿：E next() 该方法会获取当前元素。迭代器的迭代方法是 while 循环量身定制的。

```
List list=new ArrayList();  list.add("One");      list.add("#");  
Iterator it=list.iterator();  
while(it.hasNext()){//集合中是否还有下一个元素  
    Object element=it.next();//有就将其取出  
    System.out.println(element);    }
```

4) 迭代器中的删除问题：在迭代器迭代的过程中，我们不能通过“集合”的增删等操作，来改变该集合的元素数量！否则会引发迭代异常！若想删除迭代出来的元素，只能通过 Iterator。迭代器在使用自己的 remove() 方法时，可以将刚刚获取的元素从集合中删除，但是不能重复调用两次！即在不迭代的情况下，不能在一个位置删两次。

```
while(it.hasNext()){//集合中是否还有下一个元素  
String element=(String)it.next();//有就将其取出，next 返回值为 E(泛型)默认为 Object  
所以需要强转  
if("#".equals(element)){ //list.remove(element);不可以！
```

```
it.remove(); //删除当前位置元素 } }
```

4.16 泛型

- 1) 泛型是 JDK1.5 引入的新特性，泛型的本质是参数化类型。在类、接口、方法的定义过程中，所操作的数据类型为传入的指定参数类型。所有的集合类型都带有泛型参数，这样在创建集合时可以指定放入集合中的对象类型。同时，编译器会以此类型进行检查。
- 2) ArrayList 支持泛型，泛型尖括号里的符号可随便些，但通常大写 E。
- 3) 迭代器也支持泛型，但是迭代器使用的泛型应该和它所迭代的集合的泛型类型一致！
- 4) 泛型只支持引用类型，不支持基本类型，但可以使用对应的包装类
- 5) 如果泛型不指定类型的话，默认为 Object 类型。

```
ArrayList<Point> list=new ArrayList<Point>();  
list.add(new Point(1,2)); list.add(new Point(3,4));  
//list.add("哈哈");//定义泛型后，只运行 Point 类型，否则造型异常  
for(int i=0;i<list.size();i++){ Point p/*(Point)也不需要强转造型了*/list.get(i);  
System.out.println(p.getX()); }  
Iterator<Point> it=list.iterator(); while(it.hasNext()){ Point p=it.next();  
//也不需要强转了 System.out.println(p); }
```

6) 自定义泛型

```
Point p=new Point(1,2);//只能保存整数  
//把 Point 类的 int 都改成泛型 E，或者也可设置多个泛型 Point<E,Z>  
Point<Double> p1=new Point<Double>(1.0,2.3);//设置一个泛型  
Point<Double,Long> p2=new Point<Double,Long>(2.3,3L);//设置多个泛型
```

4.17 增强型 for 循环

JDK 在 1.5 版本推出了增强型 for 循环，可以用于数组和集合的遍历。

- ◆ 注意事项：集合中要有值，否则直接退出（不执行循环）。

- 1) 老循环：自己维护循环次数，循环体自行维护获取元素的方法。

```
int[] array=new int[]{1,2,3,4,5,6,7};  
for(int i=0;i<array.length;i++)//维护循环次数  
int element=array[i];//获取数组元素 System.out.print(element); }
```

- 2) 新循环：自动维护循环次数（由遍历的数组或集合的长度决定），自动获取每次迭代的元素。

```
int[] array=new int[]{1,2,3,4,5,6,7};  
for(int element:array){ System.out.print(element); }
```

- 3) 新循环执行流程：遍历数组 array 中的每个元素，将元素一次赋值给 element 后进入循环体，直到所有元素均被迭代完毕后退出循环。

- ◆ 注意事项：使用新循环，element 的类型应与循环迭代的数组或集合中的元素类型一致！至少要是兼容类型！

- 4) 新循环的内部实现是使用迭代器完成的 Iterator。

- 5) 使用新循环遍历集合：集合若使用新循环，应该为其定义泛型，否则我们只能使用 Object 作为被接收元素时的类型。通常情况下，集合都要加泛型，要明确集合中的类型，集合默认是 Object。

```
ArrayList<String> list=new ArrayList<String>();  
list.add("张三"); list.add("李四"); list.add("王五");
```

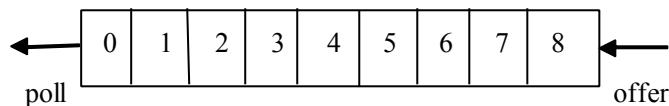
```
for(String str:list){//默认是 Object 自己加泛型 String    System.out.println(str);}
```

4.18 List 高级—数据结构：Queue 队列

队列（Queue）是常用的数据结构，可以将队列看成特殊的线性表，队列限制了对线性表的访问方式：只能从线性表的一端添加（offer）元素，从另一端取出（poll）元素。Queue 接口：在包 java.util.Queue。

1) 队列遵循先进先出原则：FIFO（First Input First Output）队列不支持插队，插队是不道德的。

2) JDK 中提供了 Queue 接口，同时使得 LinkedList 实现了该接口（选择 LinkedList 实现 Queue 的原因在于 Queue 经常要进行插入和删除的操作，而 LinkedList 在这方面效率较高）。



3) 常用方法：

①boolean offer(E e): 将一个对象添加至队尾，如果添加成功则返回 true。

②poll(): 从队列中取出元素，取得的是最早的 offer 元素，从队列中取出元素后，该元素会从队列中删除。若方法返回 null 说明 队列中没有元素了。

③peek(): 获取队首的元素（不删除该元素!）

eg: 队列相关操作

```
Queue<String> queue=new LinkedList<String>();
queue.offer("A");    queue.offer("B");    queue.offer("C");
System.out.println(queue);//[A, B, C]
System.out.println("队首: "+queue.peek());//获取队首元素,但不令其出队
String element=null; while((element=queue.poll())!=null){
    System.out.println(element); }
```

4.19 List 高级—数据结构：Deque 栈

栈（Deque）是常用的数据结构，是 Queue 队列的子接口，因此 LinkedList 也实现了 Deque 接口。栈将双端队列限制为只能从一端入队和出队，对栈而言即是入栈和出栈。子弹夹就是一种栈结构。在包 java.util.Deque 下。

1) 栈遵循先进后出的原则：FILO(First Input Last Output)。

2) 常用方法：

①push:压入，向栈中存入数据。

②pop:弹出，从栈中取出数据。

③peek: 获取栈顶位置的元素，但不取出。

◆ 注意事项：我们在使用 pop 获取栈顶元素之前，应现使用 peek 方法获取该元素，确定该元素不为 null 的情况下才应该将该元素从栈中弹出”，否则若栈中没有元素后，我们调用 pop 会抛出异常 “NoSuchElementException”。

eg: 栈相关操作

```
Deque<Character> deque=new LinkedList<Character>();
for(int i=0;i<5;i++){    deque.push((char)('A'+i));        }
System.out.println(deque);
//注意使用 peek 判断栈顶是否有元素
```

```
while(deque.peek()!=null){      System.out.print(deque.pop()+" ");      }
```

4.20 Set 集合的实现类 HashSet

Set 是无序，用于存储不重复的对象集合。在 Set 集合中存储的对象中，不存在两个对象 equals 比较为 true 的情况。

1) HashSet 和 TreeSet 是 Set 集合的两个常见的实现类，分别用 hash 表和排序二叉树的方式实现了 Set 集合。HashSet 是使用散列算法实现 Set 的。

2) Set 集合没有 get(int index)方法，我们不能像使用 List 那样，根据下标获取元素。想获取元素需要使用 Iterator。

3) 向集合添加元素也使用 add 方法，但是 add 方法不是向集合末尾追加元素，因为无序。

4) 宏观上讲：元素的顺序和存放顺序是不同的，但是在内容不变的前提下，存放顺序是相同的，但在我门使用的时候，要当作是无序的使用。

```
Set<String> set=new HashSet<String>();//多态  
//也可 HashSet<String> set=new HashSet<String>();  
set.add("One");    set.add("Two");    set.add("Three");          Iterator<String>  
it=set.iterator();  
while(it.hasNext()){    String element=it.next();    System.out.print(element+" ");    }  
for(String element:set){    System.out.print(element+" ");    }//新循环遍历 Set 集合
```

5) hashCode 对 HashSet 的影响：若我们不重写 hashCode，那么使用的就是 Object 提供的，而该方法是返回地址（句柄）！换句话说，就是不同的对象， hashCode 不同。

6) 对于重写了 equals 方法的对象，强烈要求重写继承自 Object 类的 hashCode 方法的，因为重写 hashCode 方法与否会对集合操作有影响！

7) 重写 hashCode 方法需要注意两点：

①与 equals 方法的一致性，即 equals 比较返回为 true 的对象其 hashCode 方法返回值应该相同。

②hashCode 返回的数值应该符合 hash 算法要求，如果有很多对象的 hashCode 方法返回值都相同，则会大大降低 hash 表的效率。一般情况下，可以使用 IDE（如 Eclipse）提供的工具自动生成 hashCode 方法。

8) boolean contains(Object o)方法：查看对象是否在 set 中被包含。下例虽然有新创建的对象，但是通过散列算法找到了位置后，和里面存放的元素进行 equals 比较为 true，所以依然认为是被包含的（重写 equals 了时）。

```
Set<Point> set=new HashSet<Point>();    set.add(new Point(1,2));  
set.add(new Point(3,4));    System.out.println(set.contains(new Point(1,2)));
```

9)HashCode 方法和 equals 方法都重写时对 HashSet 的影响：将两个对象同时放入 HashSet 集合，发现存在，不再放入（不重复集）。当我们重写了 Point 的 equals 方法和 hashCode 方法后，我们发现虽然 p1 和 p2 是两个对象，但是当我们把它们同时放入集合时， p2 对象并没有被添加进集合。因为 p1 在放入后， p2 放入时根据 p2 的 hashCode 计算的位置相同，且 p2 与该位置的 p1 的 equals 比较为 true， HashSet 认为该对象已经存在，所以拒绝将 p2 存入集合。

```
Set<Point> set=new HashSet<Point>();  
Point p1=new Point(1,2);    Point p2=new Point(1,2);  
System.out.println("两者是否同一对象: "+(p1==p2));  
System.out.println("两者内容是否一样: "+p1.equals(p2));
```

```

System.out.println("两者 hashCode 是否一样: "+ (p1.hashCode()==p2.hashCode()));
set.add(p1); set.add(p2); System.out.println("hashset 集合的元素数"+set.size());
for(Point p:set){ System.out.println(p); }

```

10) 不重写 hashCode 方法, 但是重写了 equals 方法对 HashSet 的影响: 两个对象都可以放入 HashStet 集合中, 因为两个对象具有不同的 hashCode 值, 那么当他们在放入集合时, 通过 hashCode 值进行的散列算法结果就不同。那么他们会被放入集合的不同位置, 位置不相同, HashSet 则认为它们不同, 所以他们可以全部被放入集合。

11) 重写了 hashCode 方法, 但是不重写 equals 方法对 HashSet 的影响: 在 hashCode 相同的情况下, 在存放元素时, 他们会在相同的位置, HashSet 会在相同位置上将后放入的对象与该位置其他对象一次进行 equals 比较, 若不相同, 则将其存入在同一个位置存入若干元素, 这些元素会被放入一个链表中。由此可以看出, 我们应该尽量使得多种类的不同对象的 hashCode 值不同, 这样才可以提高 HashSet 在检索元素时的效率, 否则可能检索效率还不如 List。

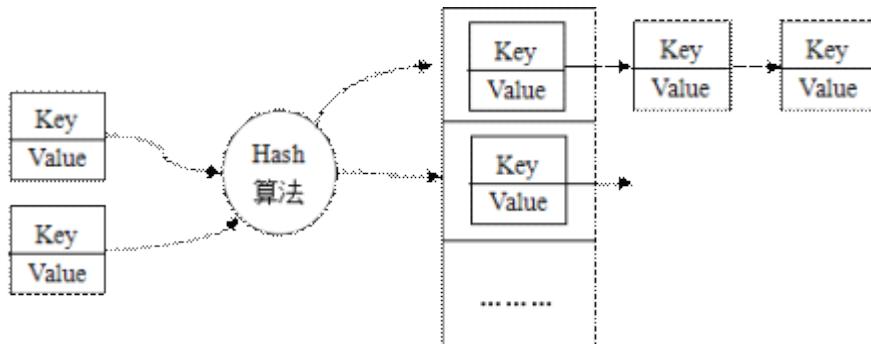
12) 结论: 不同对象存放时, 不会保存 hashCode 相同并且 equals 相同的对象, 缺一不可。否则 HashSet 不认为他们是重复对象。

4.21 Map 集合的实现类 HashMap

Map 接口定义的集合又称为查找表, 用于存储所谓 “Key-Value” 键值对。Key 可以看成是 Value 的索引。而往往 Key 是 Value 的一部分内容。

- 1) Key 不可以重复, 但所保存的 Value 可以重复。
- 2) 根据内部结构的不同, Map 接口有多种实现类, 其中常用的有内部为 hash 表实现的 HashMap 和内部为排序二叉树实现的 TreeMap。同样这样的数据结构在存放数据时, 也不建议存放两种以上的数据类型, 所以, 通常我们在使用 Map 时也要使用泛型约束存储内容的类型。
- 3) 创建 Map 时使用泛型, 这里要约束两个类型, 一个是 key 的类型, 一个是 value 的类型。

4) 基本原理图:



5) HashMap 集合中常用的方法:

- ① `V put(K Key,V value)`: 将元素以 Key-Value 的形式放入 map。若重复保存相同的 key 时, 实际的操作是替换 Key 所对应的 value 值。
- ② `V get(Object key)`: 返回 key 所对应的 value 值。如果不存在则返回 null。
- ③ `boolean containsKey(Object Key)`: 判断集合中是否包含指定的 Key。
- ④ `boolean containsValue(Object value)`: 判断集合中是否包含指定的 Value。

6) 若给定的 key 在 map 中不存在则返回 null, 所以, 原则上在从 map 中获取元素时要先判断是否有该元素, 之后再使用, 避免空指针异常的出现。Map 在获取元素时非常有针

对性，集合想获取元素需要遍历集合内容，而 Map 不需要，你只要给他特定的 key 就可以获取该元素。

```
Map<String,Point> map=new HashMap<String,Point>();
map.put("1,2", new Point(1,2)); map.put("3,4", new Point(3,4));
Point p=map.get("1,2"); System.out.println("x="+p.getX()+"y="+p.getY());
map.put("1,2", new Point(5,6));//会替换之前的
p=map.get("1,2"); System.out.println("x="+p.getX()+"y="+p.getY());
p=map.get("haha");System.out.println("x="+p.getX()+"y="+p.getY());//会报空指异常
```

eg: 统计每个数字出现的次数。步骤: ①将字符串 str 根据 “,” 拆分。②创建 map。③循环拆分后的字符串数组。④将每一个数字作为 key 在 map 中检查是否包含。⑤包含则对 value 值累加 1。⑥不包含则使用该数字作为 key, value 为 1 存入 map。

```
String str="123,456,789,456,789,225,698,759,456";
String[] array=str.split(",");
Map<String,Integer> map=new HashMap<String,Integer>();
for(String number:array){ if(map.containsKey(number)){
    int sum=map.get(number);//将原来统计的数字取出 sum++; //对统计数字加 1
    map.put(number, sum); //放回 map.put(number, map.get(number)+1);等同上三
部
} else{ map.put(number, 1); //第一次出现 value 为 1 } }
System.out.println(map);//HashMap 也重写了 toString()
```

7) 计算机中有这么一句话: 越灵活的程序性能越差, 顾及的多了。

8) 遍历 HashMap 方式一: 获取所有的 key 并根据 key 获取 value 从而达到遍历的效果 (即迭代 Key)。keySet()方法: 是 HashMap 获取所有 key 的方法, 该方法可以获取保存在 map 下所有的 key 并以 Set 集合的形式返回。

```
Map<String,Point> map=new HashMap<String,Point>();
map.put("1,2", new Point(1,2)); map.put("2,3", new Point(2,3));
map.put("3,4", new Point(3,4)); map.put("4,5", new Point(4,5));
/** 因为 key 在 HashMap 的泛型中规定了类型为 String, 所以返回的 Set 中的元素
也是 String, 为了更好的使用, 我们在定义 Set 类型变量时也应该加上泛型 */
Set<String> keyset=map.keySet();
for(String key:keyset){ Point p=map.get(key); //根据 key 获取 value
    System.out.println(key+":"+p.getX()+" "+p.getY()); }
for(Iterator<String> it=keyset.iterator(); it.hasNext(); ){//普通 for 循环
    String key=it.next(); Point p=map.get(key);
    System.out.println(key+":"+p.getX()+" "+p.getY()); }
```

9) LinkedHashMap: 用法和 HashMap 相同, 内部维护着一个链表, 可以使其存放元素时的顺序与迭代时一致。

10) Entry 类, 遍历 HashMap 方式二: 以“键值对”的形式迭代。Map 支持另一个方法 entrySet(): 该方法返回一个 Set 集合, 里面的元素是 map 中的每一组键值对, Map 以 Entry 类的实例来描述每一个键值对。其有两个方法: getKey()获取 key 值; getValue()获取 value 值。Entry 也需要泛型的约束, 其约束的泛型应该和 Map 相同! Entry 所在位置: java.util.Map.Entry。

```
Map<String,Point> map=new LinkedHashMap<String,Point>();
map.put("1,2", new Point(1,2)); map.put("2,3", new Point(2,3));
```

```

map.put("3,4", new Point(3,4));      map.put("4,5", new Point(4,5)); // 泛型套泛型
Set<Entry<String,Point>> entrySet=map.entrySet(); // Set 的泛型不会变，就是 Entry
for(Entry<String,Point> entry:entrySet){
    String key=entry.getKey(); // 获取 key  Point p=entry.getValue(); // 获取 value
    System.out.println(key+","+p.getX()+","+p.getY());
}

```

11) List、Map、Set 三个接口存储元素时各有什么特点：

①List：是有序的 Collection，使用此接口能够精确的控制每个元素插入的位置。用户能够使用索引（元素在 List 中的位置，类似于数组下标）来访问 List 中的元素，这类似于 Java 的数组。

②Set：是一种不包含重复的元素的 Collection，即任意的两个元素 e1 和 e2 都有 e1.equals(e2)=false，Set 最多有一个 null 元素。

③Map：请注意，Map 没有继承 Collection 接口，Map 提供 key 到 value 的映射。

4.22 单例模式和模版方法模式

设计模式是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。

1) 使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。简单的说：设计模式是经典问题的模式化解决方法。

2) 经典设计模式分为三种类型，共 23 类。

创建模型式：单例模式、工厂模式等

结构型模式：装饰模式、代理模式等

行为型模式：模版方法模式、迭代器模式等

3) 单例设计模式：意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。

适用性：当类只能有一个实例而且客户可以从一个众所周知的访问点访问它。任何情况下，该类只能创建一个实例！

4) 单例设计模式创建步骤：①定义一个私有的静态的当前类型的属性。②私有化构造方法。③定义一个静态的可以获取当前类实例的方法。这个方法中我们可以判断是否创建过实例，创建过就直接返回，从而达到单例的效果。

```

private static DemoSingleton obj;
// 或 private static DemoSingleton obj=new DemoSingleton();
private DemoSingleton() { }
public static DemoSingleton getInstance(){
    if(obj==null){          obj= new DemoSingleton();      }
    return obj;             }

```

5) 模版方法模式：意图：定义一个操作中的算法过程的框架，而将一些步骤延迟到子类中实现。类似于定义接口或抽象类，子类去实现抽象方法。

五、Java SE 核心 II

LICHOO

5.1 Java 异常处理机制

异常结构中的父类 Throwable 类，其子类 Exceptionlei 类和 Error 类。我们在程序中可以捕获的是 Exception 的子类异常。

Error 系统级别的错误：Java 运行时环境出现的错误，我们不可控。

Exception 是程序级别的错误：我们可控。

1) 异常处理语句：try-catch，如果 try 块捕获到异常，则到 catch 块中处理，否则跳过忽略 catch 块（开发中，一定有解决的办法才写，无法解决就向上抛 throws）。

```
try{//关键字，只能有一个 try 语句  
    可能发生异常的代码片段  
}catch(Exception e){//列举代码中可能出现的异常类型，可有多个 catch 语句  
    当出现了列举的异常类型后，在这里处理，并有针对性的处理  
}
```

2) 良好的编程习惯，在异常捕获机制的最后书写 catch(Exception e)（父类，顶极异常）捕获未知的错误（或不需要针对处理的错误）。

3) catch 的捕获是由上至下的，所以不要把父类异常写在子类异常的上面，否则子类异常永远没有机会处理！在 catch 块中可以使用方法获取异常信息：

- ①getMessage()方法：用来得到有关异常事件的信息。
- ②printStackTrace()方法：用来跟踪异常事件发生时执行堆栈的内容。

4) throw 关键字：用于主动抛出一个异常

当我们的方法出现错误时（不一定是真实异常），这个错误我们不应该去解决，而是通知调用方法去解决时，会将这个错误告知外界，而告知外界的方式就是 throw 异常（抛出异常）catch 语句中也可抛出异常。虽然不解决，但要捕获，然后抛出去。

使用环境：

我们常在方法中主动抛出异常，但不是什么情况下我们都应该抛出异常。原则上，自身决定不了的应该抛出。那么方法中什么时候该自己处理异常什么时候抛出？

方法通常有参数，调用者在调用我们的方法帮助解决问题时，通常会传入参数，若我们方法的逻辑是因为参数的错误而引发的异常，应该抛出，若是我们自身的原因应该自己处理。

```
public static void main(String[] args) {  
    try{/**通常我们调用方法时需要传入参数的话，那么这些方法，JVM 都不会自动  
    处理异常，而是将错误抛给我们解决*/  
        String result=getGirlFirend("女神");    System.out.println("追到女神了么？  
"+result);  
    }catch(Exception e){  
        System.out.println("没追到");//我们应该在这里捕获异常并处理。  
    }  
}  
  
public static String getGirlFirend(String name){  
    try{ if("春哥".equals(name)){    return "行";  
    }else if("曾哥".equals(name)){    return "行";  
    }else if("我女朋友".equals(name)){    return "不行";  
    }  
}
```

```

    }else{/**当出现了错误(不一定是真实异常)可以主动向外界抛出一个异常!*/
        throw new RuntimeException("人家不干！");
    }
}catch(NullPointerException e){
    throw e;//出了错不解决，抛给调用者解决
}
}

```

5) throws 关键字：不希望直接在某个方法中处理异常，而是希望调用者统一处理该异常。声明方法的时候，我们可以同时声明可能抛出的异常种类，通知调用者强制捕获。就是所谓的“丑话说前面”。原则上 throws 声明的异常，一定要在该方法中抛出。否则没有意义。相反的，若方法中我们主动通过 throw 抛出一个异常，应该在 throws 中声明该种类异常，通知外界捕获。

◆ 注意事项：

- ❖ 注意 throw 和 throws 关键字的区别：抛出异常和声明抛出异常。
- ❖ 不能在 main 方法上 throws，因为调用者 JVM 直接关闭程序。

```

public static void main(String[] args) {
    try{ Date today=stringToDate("2013-05-20"); } catch (ParseException e){
        //catch 中必须含有有效的捕获 stringToDate 方法 throws 的异常
        // 输出这次错误的栈信息可以直观的查看方法调用过程和出错的根源
        e.printStackTrace(); }
}

```

eg：将一个字符串转换为一个 Date 对象，抛出的异常是字符格式错误
java.text.ParseException

```

public static Date stringToDate(String str) throws ParseException{
    SimpleDateFormat format=new SimpleDateFormat("yyyy-MM-DD");
    Date date=format.parse(str); return date;
}

```

6) 捕获异常两种方式：上例 SimpleDateFormat 的 parse 方法在声明的时候就是用了 throws，强制我们调用 parse 方法时必须捕获 ParseException，我们的做法有两种：一是添加 try-catch 捕获该异常，二是在我们的方法中声明出也追加这种异常的抛出（继续往外抛）。

7) java 中抛出异常过程：java 虚拟机在运行程序时，一但在某行代码运行时出现了错误，JVM 会创建这个错误的实例，并抛出。这时 JVM 会检查出错代码所在的方法是否有 try 捕获，若有，则检查 catch 块是否有可以处理该异常的能力（看能否把异常实例作为参数传进去，看有没有匹配的异常类型）。若没有，则将该异常抛给该方法的调用者（向上抛）。以此类推，直到抛至 main 方法外仍没有解决（即抛给了 JVM 处理）。那么 JVM 会终止该程序。

8) java 中的异常 Exception 分为：

①非检测异常（RuntimeException 子类）：编译时不检查异常。若方法中抛出该类异常或其子类，那么声明方法时可以不在 throws 中列举该类抛出的异常。常见的运行时异常有： NullPointerException、IllegalArgumentException、

ClassCastException、NumberFormatException、
ArrayIndexOutOfBoundsException、ArithmaticException

②可检测异常（非 RuntimeException 子类）：编译时检查，除了运行时异常之外的异常，都是可检查异常，则必须在声明方法时用 throws 声明出可能抛出的异常种类！

9) finally 块：finally 块定义在 catch 块的最后（所有 catch 最后），且只能出现一次（0 - 1 次），无论程序是否出错都会执行的块！无条件执行！通常在 finally 语句中进行资源

的消除工作，如关闭打开的文件，删除临时文件等。

```

public static void main(String[] args) {
    System.out.println( test(null)+"."+test("0")+"."+test("") );
    /**输出结果？1,0,2 ? 4,4,4 为正确结果 */
}

public static int test(String str){
    try{      return str.charAt(0)-'0';
    }catch(NullPointerException e){      return 1;
    }catch(RuntimeException e){      return 2;
    }catch(Exception e){      return 3;
    }finally{//无条件执行      return 4;      }
}

```

10) 重写方法时的异常处理

如果使用继承时，在父类别的某个地方上宣告了 throws 某些异常，而在子类别中重新定义该方法时，可以：①不处理异常（重新定义时不设定 throws）。②可仅 throws 父类别中被重新定义的方法上的某些异常（抛出一个或几个）。③可 throws 被重新定义的方法上的异常之子类别（抛出异常的子类）。

但不可以：①throws 出额外的异常。②throws 被重新定义的方法上的异常之父类别（抛出了异常的父类）。

5.2 File 文件类

java 使用 File 类（java.io.File）表示操作系统上文件系统中的文件或目录。换句话说，我们可以使用 File 操作硬盘上的文件或目录进行创建或删除。

File 可以描述文件或目录的名字，大小等信息，但不能对文件的内容操作！File 类的构造器都是有参的。

1) 关于路径的描述：不同的文件系统差异较大，Linux 和 Windows 就不同！最好使用相对路径，不要用绝对路径。

2) “.” 代表的路径：当前目录（项目所处的目录），在 eclipse_workspace/project_name 下，File.separator：常量，目录分隔符，推荐使用！根据系统自动识别用哪种分割符，windows 中为/，Linux 中为\。

3) 创建该对象并不意味着硬盘上对应路径上就有该文件了，只是在内存中创建了该对象去代表路径指定的文件。当然这个路径对应的文件可能根本不存在！

```

File file=new File("./"+File.separator+"data.dat");// 效果为./data.dat
//File file=new File("e:/XX/XXX.txt");不建议使用

```

4) createNewFile()中有 throws 声明，要求强制捕获异常！

5) 新建文件或目录：

①boolean mkdir(): 只能在已有的目录基础上创建目录。

②boolean mkdirs(): 会创建所有必要的父目录（不存在的自动创建）并创建该目录。

③boolean createNewFile(): 创建一个空的新文件。

6) 创建目录中文件的两种方式：

①直接指定 data.dat 需要创建的位置，并调用 createNewFile()，前提是目录都要存在！

②先创建一个 File 实例指定 data.dat 即将存放的目录，若该目录不存在，则创建所有不存在的目录，再创建一个 File 实例，代表 data.dat 文件，创建是基于上一个代表目录的 File 实例的。使用 File(File dir, String fileName) 构造方法创建 File 实例，然后再调用

createNewFile(): 在 dir 所代表的目录中表示 fileName 指定的文件

```
File dir=new File(".");
if(!dir.exists()){ dir.mkdirs();//不存在则创建所有必须的父目录和当亲目录 }
File file=new File(dir,"data.dat");
if(!file.exists()){file.createNewFile();System.out.println("文件创建完毕！");}
}
```

7) 查看文件或目录属性常用方法

- ①long length(): 返回文件的长度。
- ②long lastModified(): 返回文件最后一次被修改的时间。
- ③String getName(): 返回文件或目录名。 ⑧String getPath(): 返回路径字符串。
- ④boolean exists(): 是否存在。 ⑨boolean isFile(): 是否是标准文件。
- ⑤boolean isDirectory(): 是否是目录。 ⑩boolean canRead(): 是否可以读取。
- ⑥boolean canWrite(): 是否可以写入、修改。
- ⑦File[] listFiles(): 获取当亲目录的子项（文件或目录）

eg1: File 类相关操作

```
File dir=new File(".");
if(dir.exists()&&dir.isDirectory()){ //是否为一个目录
    File[] files=dir.listFiles(); //获取当前目录的子项（文件或目录）
    for(File file:files){ //循环子项
        if(file.isFile()){ //若这个子项是一个文件
            System.out.println("文件: "+file.getName());
        } else{ //有路径显示，输出 File 的 toString()
            System.out.println("目录: "+file.getName()); } } }
```

eg2: 递归遍历出所有子项

```
File dir=new File(".");
File[] files=dir.listFiles();
if(files!=null&&files.length>0){ //判断子项数组有项
    for(File file:files){ //遍历该目录下的所有子项
        if(file.isDirectory()){ //若子项是目录
            listDirectory(file); //不到万不得已，不要使用递归，非常消耗资源
        } else{ System.out.println("文件: "+file); //有路径显示，输出 File 的 toString()
            //file.getName()无路径显示，只获取文件名 } } }
```

8) 删除一个文件: boolean delete(): ①直接写文件名作为路径和"./data.dat"代表相同文件，也可直接写目录名，但要注意第 2 条。②删除目录时：要确保该目录下没有任何子项后才可以将该目录删除，否则删除失败！

```
File dir=new File(".");
File[] files=dir.listFiles();
if(files!=null&&files.length>0){ for(File file:files){ if(file.isDirectory()){
    deleteDirectory(file); //递归删除子目录下的所有子项 } else{
    if(!file.delete()){ throw new IOException("无法删除文件: "+file); }
    System.out.println("文件: "+file+"已被删除!"); } } }
```

9) FileFilter: 文件过滤器。FileFilter 是一个接口，不可实例化，可以规定过滤条件，在获取某个目录时可以通过给定的删选条件来获取满足要求的子项。accept()方法是用来定义过滤条件的参数 pathname 是将被过滤的目录中的每个子项一次传入进行匹配，若我们认为该子项满足条件则返回 true。如下重写 accept 方法。

```
FileFilter filter=new FileFilter(){
    public boolean accept(File pathname){
        return pathname.getName().endsWith(".java"); //保留文件名以.java 结尾的
        //return pathname.length()>1700;按大小过滤 } };
```

```

File dir=new File(".");
File[] sub=dir.listFiles(filter);
for(File file:sub){
    System.out.println(file);
}

```

10) 回调模式: 我们定义一段逻辑, 在调用其他方法时, 将该逻辑通过参数传入。这个方法在执行过程中会调用我们传入的逻辑来达成目的。这种现象就是回调模式。最常见的应用环境: 按钮监听器, 过滤器的应用。

5.3 RandomAccessFile 类

可以方便的读写文件内容, 但只能一个字节一个字节 (byte) 的读写 8 位。

- 1) 计算机的硬盘在保存数据时都是 byte by byte 的, 字节挨着字节。
- 2) RandomAccessFile 打开文件模式: rw: 打开文件后可进行读写操作; r: 打开文件后只读。
- 3) RandomAccessFile 是基于指针进行读写操作的, 指针在哪里就从哪里读写。
 - ①void seek(long pos)方法: 从文件开头到设置位置的指针偏移量, 在该位置发生下一次读写操作。
 - ②getFilePointer()方法: 获取指针当前位置, 而 seek(0)则将指针移动到文件开始的位置。
 - ③int skipBytes(int n)方法: 尝试跳过输入的 n 个字节。
- 4) RandomAccessFile 类的构造器都是有参的。
 - ①RandomAccessFile 构造方法 1:

```
RandomAccessFile raf=new RandomAccessFile(file,"rw");
```

 - ②RandomAccessFile 构造方法 2:

```
RandomAccessFile raf=new RandomAccessFile("data.dat","rw");
```

直接根据文件路径指定, 前提是确保其存在!
- 5) 读写操作完了, 不再写了就关闭: close();
- 6) 读写操作:

```

File file=new File("data.dat");//创建一个 File 对象用于描述该文件
if(!file.exists()){//不存在则创建该文件
    file.createNewFile();//创建该文件, 应捕获异常, 仅为演示所以抛给 main 了 }
RandomAccessFile raf=new RandomAccessFile(file,"rw");//创建 RandomAccessFile,
并将 File 传入, RandomAccessFile 对 File 表示的文件进行读写操作。
/**1 位 16 进制代表 4 位 2 进制; 2 位 16 进制代表一个字节 8 位 2 进制;
 * 4 字节代表 32 位 2 进制; write(int) 写一个字节, 且是从低 8 位写*/
int i=0x7fffffff;//写 int 值最高的 8 位 raf.write(i>>>24);//00 00 00 7f
raf.write(i>>>16);//00 00 7f ff      raf.write(i>>>8);// 00 7f ff ff
raf.write(i);//    7f ff ff ff
byte[] data=new byte[]{0,1,2,3,4,5,6,7,8,9};//定义一个 10 字节的数组并全部写入文件
raf.write(data);//写到这里, 当前文件应该有 14 个字节了
/**写字节数组的重载方法: write(byte[] data,int offset,int length), 从 data 数组的 offset
位置开始写, 连续写 length 个字节到文件中 */
raf.write(data, 2, 5);// {2, 3, 4, 5, 6}
System.out.println("当前指针的位置: "+raf.getFilePointer());
raf.seek(0);//将指针移动到文件开始的位置
int num=0;//准备读取的 int 值

```

```

int b=raf.read();//读取第一个字节 7f 也从低 8 位开始
num=num | (b<<24);//01111111 00000000 00000000 00000000
b=raf.read();//读取第二个字节 ff
num=num| (b<<16);//01111111 11111111 00000000 00000000
b=raf.read();//读取第三个字节 ff
num=num| (b<<8);//01111111 11111111 11111111 00000000
b=raf.read();//读取第四个字节 ff
num=num| b;//01111111 11111111 11111111 11111111
System.out.println("int 最大值: "+num);      raf.close();//写完了不再写了就关了

```

LICHOO

7) 常用方法:

- ①write(int data): 写入第一个字节, 且是从低 8 位写。
- ②write(byte[] data): 将一组字节写入。
- ③write(byte[] data,int offset,int length): 从 data 数组的 offset 位置开始写, 连续写 length 个字节到文件中。
- ④writeInt(int): 一次写 4 个字节, 写 int 值。
- ⑤writeLong(long): 一次写 8 个字节, 写 long 值。
- ⑥writeUTF(String): 以 UTF-8 编码将字符串连续写入文件。
write.....
- ①int read(): 读一个字节, 若已经读取到文件末尾, 则返回-1。
- ②int read(byte[] buf): 尝试读取 buf.length 个字节。并将读取的字节存入 buf 数组。返回值为实际读取的字节数。
- ③int readInt(): 连续读取 4 字节, 返回该 int 值
- ④long readLong(): 连续读取 8 字节, 返回该 long 值
- ⑤String readUTF(): 以 UTF-8 编码将字符串连续读出文件, 返回该字符串值
read.....

```

byte[] buf=new byte[1024];//1k 容量    int sum=raf.read(buf);//尝试读取 1k 的数据
System.out.println("总共读取了: "+sum+"个字节");
System.out.println(Arrays.toString(buf));    raf.close();//写完了不再写了就关了

```

8) 复制操作: 读取一个文件, 将这个文件中的每一个字节写到另一个文件中就完成了复制功能。

```

try { File srcFile=new File("chang.txt");
        RandomAccessFile src=new RandomAccessFile(srcFile,"r");//创建一个用于读取文件的 RandomAccessFile 用于读取被拷贝的文件
        File desFile=new File("chang_copy.txt");   desFile.createNewFile();//创建复制文件
        RandomAccessFile des=new RandomAccessFile(desFile,"rw");//创建一个用于写入文件的 RandomAccessFile 用于写入拷贝的文件
        //使用字节数组作为缓冲, 批量读写进行复制操作比一个字节一个字节读写效率高得多!
        byte[] buff=new byte[1024*100];//100k 创建一个字节数组, 读取被拷贝文件的所有字节并写道拷贝文件中
        int sum=0;//每次读取的字节数
        while((sum=src.read(buff))>0){   des.write(buff,0,sum);//注意! 读到多少写多少! }
        src.close();      des.close();  System.out.println("复制完毕! ");
} catch (FileNotFoundException e) {   e.printStackTrace();
}

```

```

} catch (IOException e) {    e.printStackTrace();    }
//int data=0;//用于保存每一个读取的字节
//读取一个字节，只要不是-1（文件末尾），就进行复制工作
//while((data=src.read())!=-1){      des.write(data);//将读取的字符写入    }

```

9) 基本类型序列化：将基本类型数据转换为字节数组的过程。`writeInt(111)`: 将 int 值 111 转换为字节并写入磁盘；持久化：将数据写入磁盘的过程。

5.4 基本流：FIS 和 FOS

Java I/O 输入/输出

流：根据方向分为：输入流和输出流。方向的定了是基于我们的程序的。流向我们程序的流叫做：输入流；从程序向外流的叫做：输出流

我们可以把流想象为管道，管道里流动的水，而 java 中的流，流动的是字节。

1) 输入流是用于获取（读取）数据的，输出流是用于向外输出（写出）数据的。

`InputStream`: 该接口定义了输入流的特征

`OutputStream`: 该接口定义了输出流的特征

2) 流根据源头分为：

基本流（节点流）：从特定的地方读写的流类，如磁盘或一块内存区域。即有来源。

处理流（高级流、过滤流）：没有数据来源，不能独立存在，它的存在是用于处理基本流的。是使用一个已经存在的输入流或输出流连接创建的。

3) 流根据处理的数据单位不同划分为：

字节流：以一个“字节”为单位，以 Stream 结尾

字符流：以一个“字符”为单位，以 Reader/Writer 结尾

4) `close()`方法：流用完一定要关闭！流关闭后，不能再通过其读、写数据

5) 用于读写文件的字节流 FIS/FOS（基本流）

①`FileInputStream`: 文件字节输入流。 ②`FileOutputStream`: 文件字节输出流。

6) `FileInputStream`常用构造方法：

① `FileInputStream(File file)`: 通过打开一个到实际文件的连接来创建一个 `FileInputStream`，该文件通过文件系统中的 File 对象 file 指定。即向 file 文件中写入数据。

② `FileInputStream(String filePath)`: 通过打开一个到实际文件的连接来创建一个 `FileInputStream`，该文件通过文件系统中的文件路径名指定。也可直接写当前项目下文件名。

常用方法：

①`int read(int d)`: 读取 int 值的低 8 位。

②`int read(byte[] b)`: 将 b 数组中所有字节读出，返回读取的字节个数。

③`int read(byte[] b,int offset,int length)`: 将 b 数组中 offset 位置开始读出 length 个字节。

④`available()`方法：返回当前字节输入流 可读取的总字节数。

7) `FileOutputStream` 常用构造方法：

① `FileOutputStream(File File)`: 创建一个向指定 File 对象表示的文件中写入数据的文件输出流。会重写以前的内容，向 file 文件中写入数据时，若该文件不存在，则会自动创建该文件。

② `FileOubputStream(File file,boolean append)`: append 为 true 则对当前文件末尾进行写操作（追加，但不重写以前的）。

③ `FileOubputStream(String filePath)`: 创建一个向具有指定名称的文件中写入数据的文件输出流。前提路径存在，写当前目录下的文件名或者全路径。

④FileOubputStream(String filePath,boolean append): append 为 true 则对当前文件末尾进行写操作（追加，但不重写以前的）。

常用方法：

①void write(int d): 写入 int 值的低 8 位。

②void write(byte[] d): 将 d 数组中所有字节写入。

③void write(byte[] d,int offset,int length): 将 d 数组中 offset 位置开始写入 length 个字节。

5.5 缓冲字节高级流：BIS 和 BOS

对传入的流进行处理加工，可以嵌套使用。

1) BufferedInputStream: 缓冲字节输入流

A. 构造方法：BufferedInputStream(InputStream in)

BufferedInputStream(InputStream in, int size)

B. 常用方法：

①int read(): 从输入流中读取一个字节。

②int read(byte[] b,int offset,int length): 从此字节输入流中给定偏移量 offset 处开始将各字节读取到指定的 byte 数组中。

2) BufferedOutputStream: 缓冲字节输出流

A. 构造方法：BufferedOutputStream(OutputStream out)

BufferedOutputStream(OutputStream out, int size)

B. 常用方法：

①void write(int d): 将指定的字节写入此缓冲的输出流。

②void write(byte[] d,int offset,int length): 将指定 byte 数组中从偏移量 offset 开始的 length 个字节写入此缓冲的输出流。

③void flush(): 将缓冲区中的数据一次性写出，“清空”缓冲区。

C. 内部维护着一个缓冲区，每次都尽可能的读取更多的字节放入到缓冲区，再将缓冲区中的内容部分或全部返回给用户，因此可以提高读写效率。

3) 辨别高级流的简单方法：看构造方法，若构造方法要求传入另一个流，那么这个流就是高级流。所以高级流是没有空参数的构造器的，都需要传入一个流。

4) 有缓冲效果的流，一般为写入操作的流，在数据都写完后一定要 flush, flush 的作用是将缓冲区中未写出的数据一次性写出：bos.flush(); 即不论缓存区有多少数据，先写过去，缓冲区再下班～确保所有字符都写出

5) 使用 JDK 的话，通常情况下，我们只需要关闭最外层的流。第三方流可能需要一层一层关。

5.6 基本数据类型高级流：DIS 和 DOS

是对“流”功能的扩展，简化了对基本类型数据的读写操作。

1) DataInputStream(InputStream in): 可以直接读取基本数据类型的流

常用方法：

①int readInt(): 连续读取 4 个字节（一个 int 值），返回该 int 值

②double readDouble(): 连续读取 8 个字节（一个 double 值），返回 double 值

③String readUTF(): 连续读取字符串

.....

2) DataOutputStream(OutputStream out): 可以直接写基本数据类型的流

常用方法：

- ①void writeInt(int i): 连续写入 4 个字节（一个 int 值）
- ②void writeLong(long l): 连续写入 8 个字节（一个 long 值）
- ③void writeUTF(String s): 连续写入字符串
- ④void flush(): 将缓冲区中的数据一次性写出，“清空”缓冲区。
-

5.7 字符高级流：ISR 和 OSW

以“单个”“字符”为单位读写数据，一次处理一个字符(unicode)。

字符流底层还是基于字节形式读写的。

在字符输入输出流阶段，进行编码修改与设置。

所有字符流都是高级流。

1) OutputStreamWriter: 字符输出流。

A. 常用构造方法：

OutputStreamWriter(OutputStream out): 创建一个字符集的输出流。

OutputStreamWriter(OutputStream out, String charsetName): 创建一个使用指定字符集的输出流。

B. 常用方法：

①void write(int c): 写入单个字符。

②void write(char c[], int off, int len): 写入从字符数组 off 开头到 len 长度的部分

③void write(String str, int off, int len): 写入从字符串 off 开头到 len 长度的部分。

④void flush(): 将缓冲区中的数据一次性写出，“清空”缓冲区。

⑤void close(): 关闭流。

eg: 向文件中写入字符：①创建文件输出流（字节流）。②创建字符输出流（高级流），处理文件输出流，目的是我们可以以字节为单位写数据。③写入字符。④写完后关闭流。

```
OutputStreamWriter writer=null;//不写 try-catch 外的话 finally 找不到流，就无法关闭
try{ FileOutputStream fos=new FileOutputStream("writer.txt");
    // writer=new OutputStreamWriter(fos); //默认构造方法使用系统默认的编码
集
    writer=new OutputStreamWriter(fos,"UTF-8");//最好指定字符集输出
    writer.write("你好！ ");
    writer.flush(); //将缓冲区数据一次性写出
} catch(IOException e){ throw e; }
finally{ if(writer!=null){ writer.close(); } }
```

2) InputStreamReader: 字符输入流。

A. 常用构造方法：

InputStreamReader(InputStream in): 创建一个字符集的输入流。

InputStreamReader(InputStream in, String charsetName): 创建一个使用指定字符集的输入流。

B. 常用方法：

①int read(): 读取单个字符。

②int read(char cbuf[], int offset, int length): 读入字符数组中从 offset 开始的 length 长度的字符。

③void close(): 关闭流。

eg: 读取文件中的字符

```
InputStreamReader reader=null;
try{//创建用于读取文件的字节出入流
    FileInputStream fis=new FileInputStream("writer.txt");
    //创建用于以字符为单位读取数据的高级流
    reader=new InputStreamReader(fis,"UTF-8");    int c=-1;//读取数据
    while((c=reader.read())!=-1){//InputStreamReader 只能一个字符一个字符的读
        System.out.println((char)c);
    }
} catch(IOException e){    throw e;    }
finally{ if(reader!=null){    reader.close();    }}
```

5.8 缓冲字符高级流：BR 和 BW

可以以“行”为单位读写“字符”，高级流。

在字符输入输出流修改编码。

1) BufferedWriter: 缓冲字符输出流，以行为单位写字符

A. 常用构造方法：

BufferedWriter(Writer out): 创建一个使用默认大小的缓冲字符输出流。

BufferedWriter(Writer out,int size): 创建一个使用给定大小的缓冲字符输出流。

B. 常用方法：

①void write(int c): 写入单个字符。

②void write(char[] c,int off,int len): 写入字符数组从 off 开始的 len 长度的字符。

③void write(String s,int off,int len): 写入字符串中从 off 开始的 len 长度的字符。

④void newLine(): 写入一个行分隔符。

⑤flush(): 将缓冲区中的数据一次性写出，“清空”缓冲区。

⑥close(): 关闭流。

◆ 注意事项：BufferedWriter 的构造方法中不支持给定一个字节输出流，只能给定一个字符输出流 Writer 的子类，Writer 是字符输出流的父类。

```
//创建用于写文件的输出流
FileOutputStream fos=new FileOutputStream("buffered.txt");
//创建一个字符输出流，在字符输入输出流修改编码
OutputStreamWriter osw=new OutputStreamWriter(fos,"UTF-8");
BufferedWriter writer=new BufferedWriter(osw);
writer.write("你好啊!! ");
writer.newLine();//输出一个换行
writer.write("我是第二行!! ");
writer.newLine();//输出一个换行
writer.write("我是第三行!! ");
writer.close();//输出流关闭后，不能再通过其写数据
```

2) BufferedReader: 缓冲字符输入流，以行为单位读字符

A. 常用构造方法：

BufferedReader(Reader in): 创建一个使用默认大小的缓冲字符输入流。

BufferedReader(Reader in,int size): 创建一个使用指定大小的缓冲字符输入流。

B. 常用方法：

①int read(): 读取单个字符。如果已到达流末尾，则返回-1。

②int read(char cbuf[], int off, int len): 从字符数组中读取从 off 开始的 len 长度

的字符。返回读取的字符数，如果已到达流末尾，则返回-1。

③String readLine(): 读取一个文本行。通过下列字符之一即可认为某行已终止：换行 ('\n')、回车 ('\r') 或回车后直接跟着换行。如果已到达流末尾，则返回 null。EOF: end of file 文件末尾。

④void close(): 关闭流。

eg: 读取指定文件中的数据，并显示在控制台

```
FileInputStream fis=new FileInputStream(  
    "src"+File.separator+"day08"+File.separator+"DemoBufferedReader.java");  
InputStreamReader isr=new InputStreamReader(fis);  
BufferedReader reader=new BufferedReader(isr);      String str=null;  
if((str=reader.readLine())!=null){//readLine()读取一行字符并以字符串形式返回  
    System.out.println(str);        }           reader.close();
```

eg: 读取控制台输入的每以行信息，直到在控制台输入 exit 退出程序

```
//1 将键盘的字节输入流转换为字符输入流  
InputStreamReader isr=new InputStreamReader(System.in);  
//2 将字符输入流转换为缓冲字符输入流，按行读取信息  
BufferedReader reader=new BufferedReader(isr);  
// 循环获取用户输入的信息并输出到控制台  
String info=null;  while(true){  info=reader.readLine();  
if("exit".equals(info.trim())){ break;          }  
    System.out.println(info);//输出到控制台          }           reader.close();
```

5.9 文件字符高级流：FR 和 FW

用于读写“文本文件”的“字符”输入流和输出流。

1) FileWriter 写入：继承 OutputStreamWriter

A. 常用构造方法

FileWriter(File file)、FileWriter(File file, boolean append)
FileWriter(String filePath)、FileWriter(String fileName, boolean append)
意思和 FileOutputStream 的四个同类型参数的构造方法一致。

◆ 注意事项：FileWriter 的效果等同于：FileOutputStream + OutputStreamWriter。

B. 常用方法：

- ①void write(int c): 写入单个字符。
- ②void write(char c[], int off, int len): 写入字符数组从 off 到 len 长度的部分
- ③void write(String str, int off, int len): 写入字符串从 off 到 len 长度的部分。
- ④void flush(): 将缓冲区中的数据一次性写出，“清空”缓冲区。
- ⑤void close(): 关闭流。

```
FileWriter writer=new FileWriter("filewriter.txt");  
//File file=new File("filewriter.txt");  
//FileWriter writer=new FileWriter(file);  
writer.write("hello!FileWriter!");           writer.close();
```

2) FileReader 读取：继承 InputStreamReader

A. “只能”以“字符”为单位读取文件，所以效率低

B. 常用构造方法

FileReader(File file)、FileReader(String filePath)

意思和 FileInputStream 的两个同类型参数的构造方法一致。

C. 常用方法:

①int read(): 读取单个字符。

②int read(char cbuf[], int offset, int length): 读入字符数组中从 offset 开始的 length 长度的字符。

③void close(): 关闭流。

```
FileReader reader=new FileReader("filewriter.txt");
//int c=-1;//只能以字符为单位读取文件
//while((c=reader.read())!=-1){    System.out.println((char)c);      }
//将文件字符输入流转换为缓冲字符输入流便可以行为单位读取
BufferedReader br=new BufferedReader(reader);
String info=null;  while((info=br.readLine())!=null){  System.out.println(info);  }
br.close();
```

5.10 PrintWriter

另一种缓冲“字符”输出流，以“行”为单位，常用它作输出，BufferedWriter 用的少。

1) Servlet: 运行在服务器端的小程序，给客户端发送相应使用的输出流就是 PrintWriter。

2) 写方法: println(String data): 带换行符输出一个字符串，不用手动换行了。

println.....

3) 构造方式:

PrintWriter(File file): 以行为单位向文件写数据

PrintWriter(OutputStream out): 以行为单位向字节输出流写数据

PrintWriter(Writer writer): 以行为单位向字符输出流写数据

PrintWriter(String fileName): 以行为单位向指定路径的文件写数据

```
PrintWriter writer=new PrintWriter("printwriter.txt"); //向文件写入一个字符串
writer.println("你好！ PrintWriter");//自动加换行符
```

/*我们要在确定做写操作的时候调用 flush()方法，否则数据可能还在输出流的缓冲区中，没有作真实的写操作！*/

```
writer.flush();    writer.close();
```

eg: 将输出流写入文件

```
System.out.println("你好！！");          PrintStream out=System.out;
PrintStream fileOut=new PrintStream( new FileOutputStream("SystemOut.txt") );
System.setOut(fileOut);//将我们给定的输出流赋值到 System.out 上
System.out.println("你好！我是输出到控制台的!");    System.setOut(out);
System.out.println("我是输出到控制台的!");           fileOut.close();
```

5.11 对象序列化

将一个对象转换为字节形式的过程就是对象序列化。序列化还有个名称为串行化，序列化后的对象再被反序列化后得到的对象，与之前的对象不再是同一个对象。

1) 对象序列化必须实现 Serializable 接口，但该接口无任何抽象方法，不需要重写方法，只为了标注该类可序列化。

2) 且同时建议最好添加版本号（编号随便写）: serialVersionUID。版本号，用于匹配当前类与其被反序列化的对象是否处于同样的特征（属性列表一致等）。反序列化时，ObjectInputStream 会根据被反序列化对象的版本与当前版本进行匹配，来决定是否反序列

化。不加版本号可以，但是可能存在反序列化失败的风险。

- 3) JDK 提供的大多数 java bean 都实现了该接口
- 4) transient 关键字：序列化时忽略被它修饰的属性。
- 5) 对象的序列化使用的类： ObjectOutputStream
 writeObject(Object obj): ①将给定对象序列化。②然后写出。
- 6) 对象的反序列化使用的类： ObjectInputStream
 Object readObject(): 将读取的字节序列还原为对象
- 7) 对于 HTTP 协议：通信一次后，必须断开连接，想再次通信要再次连接。
- 8) 想要实现断点续传，我们必须告诉服务器我们当前读取文件的开始位置。相当于我们本地调用的 seek(), 因为我们不可能直接调用服务器的对象的方法，所以我们只能通过某种方式告诉服务器我们要干什么。让它自行调用自己流对象的 seek() 到我们想读取的位置。
bytes=0- 的意思是告诉服务器从第一个字节开始读，即 seek(0) 从头到尾； bytes=128- 的意思是告诉服务器从第 128 个字节开始读，即 seek(128)。String prop="bytes="+info newPos()+"-";

eg: 序列化和反序列化

```

try{ DownloadInfo info=new
DownloadInfo("http://www.baidu.com/download/xxx.zip"
            , "xxx.zip" );
    info.setPos(12587);   info.setFileSize(5566987);
    File file=new File("obj.tmp");//将对象序列化以后写到文件中
    FileOutputStream fos=new FileOutputStream(file);
    //通过 oos 可以将对象序列化后写入 obj.tmp 文件中
    ObjectOutputStream oos=new ObjectOutputStream(fos);
    oos.writeObject(info);//将 info 序列化后写出      oos.close();
    ///反序列化操作
    FileInputStream fis=new FileInputStream(file);
    ObjectInputStream ois=new ObjectInputStream(fis);
    DownloadInfo obj=(DownloadInfo)ois.readObject();//反序列化
    System.out.println(obj.getUrl());      System.out.println(obj.getFileName());
    System.out.println(obj.getFileSize());  System.out.println(obj newPos());
    System.out.println(info==obj);         ois.close();
} catch(Exception e){  e.printStackTrace();  System.out.println("非常 sorry! ");}

```

5.12 Thread 线程类及多线程

进程：一个操作系统中可以同时运行多个任务（程序），每个运行的任务（程序）被称为一个进程。即系统级别上的多线程（多个任务）。

线程：一个程序同时可能运行多个任务（顺序执行流），那么每个任务（顺序执行流）就叫做一个线程。即在进程内部。

并发：线程是并发运行的。操作系统将时间化分为若干个片段（时间片），尽可能的均匀分配给每一个任务，被分配时间片后，任务就有机会被 CPU 所执行。微观上看，每个任务都是走走停停的。但随着 CPU 高效的运行，宏观上看所有任务都在运行。这种都运行的现象称之为并发，但不是绝对意义上的“同时发生”。

1) Thread 类的实例代表一个并发任务。任何线程对象都是 Thread 类的（子类）实例。Thread 类是线程的模版，它封装了复杂的线程开启等操作，封装了操作系统的差异性。因此并发的任务逻辑实现只要重写 Thread 的 run 方法即可。

2) 线程调度：线程调度机制会将所有并发任务做统一的调度工作，划分时间片（可以被cpu执行的时间）给每一个任务，时间片尽可能的均匀，但做不到绝对均匀。同样，被分配时间片后，该任务被cpu执行，但调度的过程中不能保证所有任务都是平均的获取时间片的次数。只能做到尽可能平均。这两个都是程序不可控的。

3) 线程的启动和停止：void start(): 想并发操作不要直接调用run方法！而是调用线程的start()方法启动线程！void stop(): 不要使用stop()方法来停止线程的运行，这是不安全的操作，想让线程停止，应该通过run方法的执行完毕来进行自然的结束。

4) **线程的创建方式一：**1: 继承自 Thread。2: 重写 run 方法：run方法中应该定义我们需要并发执行的任务逻辑代码。

5) **线程的创建方式二：**将线程与执行的逻辑分离开，即实现 Runnable 接口。因为有了这样的设计，才有了线程池。关注点在于要执行的逻辑。

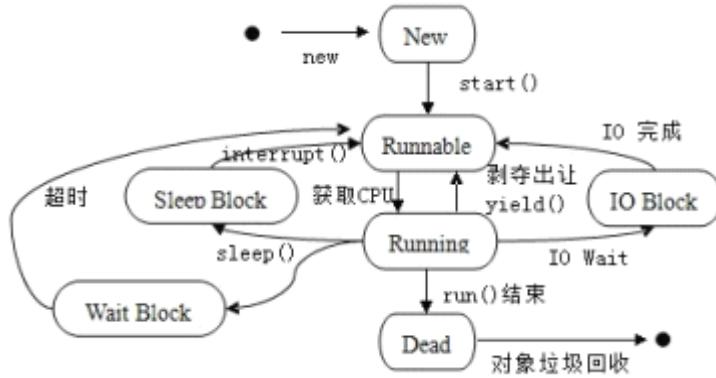
6) Runnable 接口：用于定义线程要执行的任务逻辑。我们定一个类实现 Runnable 接口，这时我们必须重写 run 方法，在其中定义我们要执行的逻辑。之后将 Runnable 交给线程去执行。从而实现了线程与其执行的任务分离开。将任务分别交给不同的线程并发处理，可以使用线程的重载构造方法：Thread(Runnable runnable)。解耦：线程与线程体解耦，即打断依赖关系。Spring 的 ioc 就是干这个的。

```
/**创建两个需要并发的任务，MyFirstRunnable 和 MySecRunnable 都继承了 Runnable  
接口并重写了 run()方法 */  
Runnable r1=new MyFirstRunnable();      Runnable r2=new MySecRunnable();  
Thread t1=new Thread(r1);      Thread t2=new Thread(r2);  
t1.start();      t2.start();
```

7) **线程的创建方式三：**使用匿名内部类方式创建线程

```
new Thread(){public void run(){...}}.start();    /** * 匿名类实现继承 Thread 形式*/  
Thread t1=new Thread(){  
    public void run(){  
        for(int i=0;i<1000;i++){  
            System.out.println(i);  
        }  
    }  
};  
new Thread(new Runnable(){public void run(){...}}).start();  
/**匿名类实现 Runnable 接口的形式 */  
Thread t2=new Thread(new Runnable(){  
    public void run(){  
        for(int i=0;i<1000;i++){  
            System.out.println("你好"+i+"次");  
        }  
    }  
});
```

8) 线程生命周期：



9) 线程睡眠阻塞：使当前线程放弃 cpu 时间，进入阻塞状态。在阻塞状态的线程不会分配时间片。直到该线程结束阻塞状态回到 Runnable 状态，方可再次获得时间片来让 cpu 运行（进入 Running 状态）。

①static void sleep(times)方法：让当前线程主动进入 Block 阻塞状态，并在 time 毫秒后回到 Runnable 状态。

◆ 注意事项：使用 Thread.sleep()方法阻塞线程时，强制让我们必须捕获“中断异常”。引发情况：当前线程处于 Sleep 阻塞期间，被另一个线程中断阻塞状态时，当前线程会抛出该异常。

```

int i=0;      while(true){  System.out.println(i+"秒");  i++;
try {    Thread.sleep(1000);
} catch (InterruptedException e) {    e.printStackTrace();  }
}

```

10) void interrupt()方法：打断/唤醒线程。一个线程可以提前唤醒另外一个 sleep Block 的线程。

◆ 注意事项：方法中定义的类叫局部内部类：局部内部类中，若想引用当前方法的其他局部变量，那么该变量必须是 final 的。

```

final Thread lin=new Thread(){
    public void run(){      System.out.println("林：睡觉了……");
    try {  Thread.sleep(1000000);  } catch (InterruptedException e) {
        System.out.println("林：干嘛呢！干嘛呢！干嘛呢！");
        System.out.println("林：都破了相了！ ");
    }  }
    lin.start();//启动第一个线程
}

Thread huang=new Thread(){
    public void run(){  System.out.println("80一锤子，您说砸哪儿？ ");
    for(int i=0;i<5;i++){  System.out.println("80！");
    try {  Thread.sleep(1000);  } catch (InterruptedException e) {
        e.printStackTrace();
    }
    System.out.println("咣当！");
    System.out.println("黄：搞定！");
    lin.interrupt();//中断第一个线程的阻塞状态
    }
    huang.start();//启动第二个线程
}

```

11) 线程的其他方法：

①static void yield(): 当前线程让出处理器（离开 Running 状态）即放弃当前时间片，主动进入 Runnable 状态等待。

②final void setPriority(int): 设置线程优先级；优先级越高的线程，理论上获取 cpu

的次数就越多。但理想与现实是有差距的……设置线程优先级一定要在线程启动前设置！

③final void join(): 等待该线程终止。

```
Thread t1=new Thread(){
    public void run(){ for(int i=0;i<100;i++){ System.out.println("我是谁啊？");
        Thread.yield(); } }
};

Thread t2=new Thread(){
    public void run(){ for(int i=0;i<100;i++){ System.out.println("我是修水管的");
        Thread.yield(); } }
};

Thread t3=new Thread(){
    public void run(){ for(int i=0;i<100;i++){ System.out.println("我是打酱油的");
        Thread.yield(); } }
};

t1.setPriority(Thread.MAX_PRIORITY); t2.setPriority(Thread.MIN_PRIORITY);
t1.start(); t2.start(); t3.start();
```

12) 线程并发安全问题: synchronized 关键字, 线程安全锁、同步监视器。

多线程在访问同一个数据时(写操作), 可能会引发不安全操作。

①哪个线程报错不捕获, 则线程死, 不影响主程序。

②同步: 同一时刻只能有一个执行, A 和 B 配合工作, 步调一致的处理(B 得到 A 的执行结果才能继续)。如一群人上公交车。

异步: 同一时刻能有多个执行, 并发, 各自干各自的。如一群人上卡车。

③synchronized 可以修饰方法也可以单独作为语句块存在(同步块)。作用是限制多线程并发时同时访问该作用域。

④synchronized 修饰方法后, 会为方法上锁。方法就不是异步的了, 而是同步的。锁的是当前对象。

⑤synchronized 同步块: 分析出只有一段代码需要上锁, 则使用。效率比直接修饰方法要高。

⑥线程安全的效率低, 如 Vector、Hashtable。线程不安全的效率高, 如 ArrayList、HashMap

```
synchronized void getMoney(int money){ if(count==0){
    throw new RuntimeException("余额为 0"); }
    Thread.yield(); count-=money; }

void getMoney(int money){
    synchronized(this){//synchronized(Object){需要同步的代码片段}
        if(count==0){ throw new RuntimeException("余额为 0"); }
        Thread.yield(); count-=money; }
```

13) Daemon 后台线程也称为守护线程: 当当前进程中“所有”“前台”线程死亡后, 后台线程将被强制死亡(非自然死亡), 无论是否还在运行。

①守护线程, 必须在启动线程前调用。

②main 方法也是靠线程运行的, 且是一个前台线程。

③正在运行的线程都是守护线程时, JVM 退出。

14) wait/notify 方法

这两个方法不是在线程 Thread 中定义的方法, 这两个方法定义在 Object 中。两个

方法的作用是用于协调线程工作的。

①等待机制与锁机制密切关联：wait/notify 方法必须与 synchronized 同时使用，谁调用 wait 或 notify 方法，就锁谁！

②wait()方法：当条件不满足时，则等待。当条件满足时，等待该条件的线程将被唤醒。如：浏览器显示一个图片，displayThread 要想显示图片，则必须等待下载线程 downloadThread 将该图片下载完毕。如果图片没有下载完成，则 displayThread 可以暂停。当 downloadThread 下载完成后，再通知 displayThread 可以显示了，此时 displayThread 继续执行。

③notify()方法：随机通知、唤醒一个在当前对象身上等待的线程。

④notifyAll 方法：通知、唤醒所有在当前对象身上等待的线程。

5.13 Socket 网络编程

Socket 套接字。在 java.net.Socket 包下。

1) 网络通信模型：C/S：client/server，客户端/服务器端；B/S：browser/server，浏览器端/服务器端；C/S 结构的优点：应用的针对性强，画面绚丽，应用功能复杂。缺点：不易维护。B/S 结构的优点：易于维护。缺点：效果差，交互性不强。

2) Socket：封装着本地的地址，服务端口等信息。ServerSocket：服务端的套接字。

服务器：使用 ServerSocket 监听指定的端口，端口可以随意指定（由于 1024 以下的端口通常属于保留端口，在一些操作系统中不可以随意使用，所以建议使用大于 1024 的端口），等待客户连接请求，客户连接后，会话产生；在完成会话后，关闭连接。

客户端：使用 Socket 对网络上某一个服务器的某一个端口发出连接请求，一旦连接成功，打开会话；会话完成后，关闭 Socket。客户端不需要指定打开的端口，通常临时的、动态的分配一个 1024 以上的端口。

3) 永远都是 Socket 去主动连接 ServerSocket。一个 ServerSocket 可以接收若干个 Socket 的连接。网络通信的前提：一定要捕获异常。

4) Socket 连接基于 TCP/IP 协议，是一种长连接（长时间连着）。

5) 读取服务器信息会阻塞，写操作不会。

6) 建立连接并向服务器发送信息步骤：①通过服务器的地址及端口与服务器连接，而创建 Socket 时需要以上两个数据。②连接成功后可以通过 Socket 获取输入流和输出流，使用输入流接收服务端发送过来的信息。③关闭连接。

7) 连接服务器：一旦 Socket 被实例化，那么它就开始通过给定的地址和端口号去尝试与服务器进行连接（自动的）。这里的地址"localhost"是服务器的地址，8088 端口是服务器对外的端口。我们自身的端口是系统分配的，我们无需知道。

8) 和服务器通信（读写数据）：使用 Socket 中的 getInputStream() 获取输入流，使用 getOutputStream() 获取输出流。

9) ServerSocket 构造方法要求我们传入打开的端口号，ServerSocket 对象在创建的时候就向操作系统申请打开这个端口。

10) 通过调用 ServerSocket 的 accept 方法，使服务器端开始等待接收客户端的连接。该方法是一个阻塞方法，监听指定的端口是否有客户端连接。直到有客户端与其连接并接收客户端套接字，否则该方法不会结束。

eg1.1：客户端 ClientDemo 类

```
private Socket socket;
```

```

public void send(){
    try{ System.out.println("开始连接服务器");      socket=new Socket("localhost",8088);
    InputStream in=socket.getInputStream();//获取输入流
    OutputStream out=socket.getOutputStream();//获取输出流
    /**将输出流变成处理字符的缓冲字符输出流*/
    PrintWriter writer=new PrintWriter(out);          writer.println("你好！ 服务器！ ");
    /**注意，写到输出流的缓冲区里了，并没有真的发给服务器。想真的发送就要作
    真实的写操作，清空缓冲区*/
    writer.flush();
    /**将输入流转换为缓冲字符输入流*/
    BufferedReader reader=new BufferedReader(new InputStreamReader(in));
    /**读取服务器发送过来的信息*/
    String info=reader.readLine(); //读取服务器信息会阻塞   System.out.println(info);
    writer.println("再见！ 服务器！ ");    writer.flush();
    info=reader.readLine();      System.out.println(info);
    }catch(Exception e){   e.printStackTrace();   }
}
public static void main(String[] args){
    ClientDemo demo=new ClientDemo();      demo.send(); //连接服务器并通信
}

```

eg1.2：服务器端 ServerDemo 类（不使用线程）

```

private ServerSocket socket=null;      private int port=8088;
/**构建 ServerDemo 对象时就打开服务端口*/
public ServerDemo(){
    try{ socket=new ServerSocket(port); }catch(Exception e){ e.printStackTrace(); }
    /**开始服务，等待收受客户端的请求并与之通信*/
public void start(){
    try{ System.out.println("等待客户端连接……");    Socket s=socket.accept();
        //获取与客户端通信的输入输出流
        InputStream in=s.getInputStream();    OutputStream out=s.getOutputStream();
        //包装为缓冲字符流
        PrintWriter writer=new PrintWriter(out);
        BufferedReader reader=new BufferedReader(new InputStreamReader(in));
        //先听客户端发送的信息
        String info=reader.readLine(); //这里同样会阻塞   System.out.println(info);
        //发送信息给客户端
        writer.println("你好！ 客户端");    writer.flush();
        info=reader.readLine();      System.out.println(info);
        writer.println("再见！ 客户端");    writer.flush();
        socket.close(); //关闭与客户端的连接
    }catch(Exception e){   e.printStackTrace();   }
}
public static void main(String[] args){
    System.out.println("服务器启动中……");
    ServerDemo demo=new ServerDemo();      demo.start();
}

```

eg2：服务器端 ServerDemo 类（使用线程），start()方法的修改以及 Handler 类

```

public void start(){
    try{while(true){ System.out.println("等待客户端连接……"); Socket s=socket.accept();

```

```

    /**
     * 当一个客户端连接了，就启动一个线程去接待它 */
    Thread clientThread=new Thread(new Handler(s)); clientThread.start();
} catch(Exception e){ e.printStackTrace(); }
/** 定义线程体，该线程的作用是与连接到服务器端的客户端进行交互操作 */
class Handler implements Runnable{
    private Socket socket;//当前线程要进行通信的客户端 Socket
    public Handler(Socket socket){//通过构造方法将客户端的 Socket 传入
        this.socket=socket;
    }
    public void run(){
        try{ //获取与客户端通信的输入输出流
            InputStream           in=socket.getInputStream();OutputStream
out=socket.getOutputStream();
            PrintWriter writer=new PrintWriter(out);//包装为缓冲字符流
            BufferedReader reader=new BufferedReader(new InputStreamReader(in));
            String info=reader.readLine();//先听客户端发送的信息，这里同样会阻塞
            System.out.println(info);
            //发送信息给客户端
            writer.println("你好！客户端"); writer.flush();
            info=reader.readLine(); System.out.println(info);
            writer.println("再见！客户端"); writer.flush();
            socket.close();//关闭与客户端的连接
        } catch(Exception e){ e.printStackTrace(); }
    }
    public static void main(String[] args){
        System.out.println("服务器启动中……");
        ServerDemo demo=new ServerDemo(); demo.start();
    }
}

```

5.14 线程池

线程若想启动需要调用 `start()` 方法。这个方法要做很多操作。要和操作系统打交道。注册线程等工作，等待线程调度。`ExecutorService` 提供了管理终止线程池的方法。

1) 线程池的概念：首先创建一些线程，它们的集合称为线程池，当服务器接收到一个客户请求后，就从线程池中取出一个空闲的线程为之服务，服务完后不关闭该线程，而是将该线程还回到线程池中。在线程池的编程模式下，任务是提交给整个线程池，而不是直接交给某个线程，线程池在拿到任务后，它就在内部找有无空闲的线程，再把任务交给内部某个空闲的线程，一个线程同时只能执行一个任务，但可以同时向一个线程池提交多个任务。

2) 线程池的创建都是工厂方法。我们不要直接去 `new` 线程池，因为线程池的创建还要作很多的准备工作。

3) 常见构造方法：

①`Executors.newCachedThreadPool()`:可根据任务需要动态创建线程，来执行任务。若线程池中有空闲的线程将重用该线程来执行任务。没有空闲的则创建新线程来完成任务。理论上池子里可以放 int 最大值个线程。缓存线程生命周期 1 分钟，得不到任务直解 kill

②`Executors.newFixedThreadPool(int threads)`:创建固定大小的线程池。池中的线程数是固定的。若所有线程处于饱和状态，新任务将排队等待。

③`Executors.newScheduledThreadPool()`:创建具有延迟效果的线程池。可将带运行的任务延迟指定时长后再运行。

④`Executors.newSingleThreadExecutor()`:创建单线程的线程池。池中仅有一个线程。

所有未运行的任务排队等待。

5.15 双缓冲队列

BlockingQueue: 解决了读写数据阻塞问题，但是同时写或读还是同步的。

1) 双缓冲队列加快了读写数据操作，双缓冲对列可以规定队列存储元素的大小，一旦队列中的元素达到最大值，待插入的元素将等。等待时间是给定的，当给定时间到了元素还没有机会被放入队列那么会抛出超时异常。

2) **LinkedBlockingQueue** 是一个可以不指定队列大小的双缓冲队列。若指定大小，当达到峰值后，待入队的将等待。理论上最大值为 int 最大值。

eg1.1: log 服务器写日志文件，客户端 ClientDemo 类，try 语句块中修改如下

```
try{ System.out.println("开始连接服务器");
    socket=new Socket("localhost",8088);
    OutputStream out=socket.getOutputStream();
    PrintWriter writer=new PrintWriter(out);
    while(true){
        writer.println("你好！服务器！");
        writer.flush();
        Thread.sleep(500); } }
```

eg1.2: log 服务器写日志文件，服务器端 ServerDemo 类，增加线程池和双缓冲队列两个属性，删掉与原客户端的输出流

```
private ExecutorService threadPool;//线程池
private BlockingQueue<String> msgQueue; //双缓冲队列
public ServerDemo(){
    try{ socket=new ServerSocket(port);
        //创建 50 个线程的固定大小的线程池
        threadPool=Executors.newFixedThreadPool(50);
        msgQueue=new LinkedBlockingQueue<String>(10000);
        /**创建定时器，周期性的将队列中的数据写入文件*/
        Timer timer=new Timer();
        timer.schedule(new TimerTask(){
            public void run(){
                try{ //创建用于向文件写信息的输出流
                    PrintWriter writer=new PrintWriter(new FileWriter("log.txt",true));
                    //从队列中获取所有元素，作写出操作
                    String msg=null;
                    for(int i=0;i<msgQueue.size();i++){
                        /**参数 0: 时间量 TimeUnit.MILLISECONDS: 时间单位*/
                        msg=msgQueue.poll(0,TimeUnit.MILLISECONDS);
                        if(msg==null){ break; }
                        writer.println(msg);//通过输出流写出数据
                    }
                    writer.close();
                } catch(Exception e){ e.printStackTrace(); }
            }
        });
    } }
```

```

        }, 0,500);
    }catch(Exception e){    e.printStackTrace();    }
public void start(){
    try{ while(true){  System.out.println("等待客户端连接……");
        Socket s=socket.accept();
        /**将线程体（并发的任务）交给线程池，线程池会自动将该任务分配给一个空闲线程
去执行。 */
        threadPool.execute(new Handler(s));
        System.out.println("一个客户端连接了， 分配线程");
    }catch(Exception e){    e.printStackTrace();    }
    /**定义线程体，该线程的作用是与连接到服务器端的客户端进行交互操作*/
    class Handler implements Runnable{
        private Socket socket;//当前线程要进行通信的客户端 Socket
        public Handler(Socket socket){//通过构造方法将客户端的 Socket 传入
            this.socket=socket;
        }
        public void run(){
            try{ //获取与客户端通信的输入输出流
                InputStream in=socket.getInputStream();
                //包装为缓冲字符流
                BufferedReader reader=new BufferedReader(new InputStreamReader(in));
                String info=null;
                while(true){//循环读取客户端发送过来的信息
                    info=reader.readLine();
                    if(info!=null){//插入对列成功返回 true， 失败返回 false
                        //该方法会阻塞线程，若中断会报错！
                        boolean b=msgQueue.offer(info, 5, TimeUnit.SECONDS);
                    }
                }catch(Exception e){    e.printStackTrace();    }
            }
        }
    }
}

```

2-Oracle 数据库、SQL 学习笔记

六、数据库介绍

1.1 表是数据库中存储数据的基本单位

1.2 数据库标准语言

结构化查询语言 SQL: Structured Query Language

1) 数据定义语言 DDL: Data Definition Language

create table 列表结构、alter table 修改列、drop table 删除列

2) 数据操作语言 DML: Data Manipulation Language

insert 增加一行, 某些列插入值、update 修改一行, 这一行的某些列、delete 删除一行, 跟列无关

3) 事务控制语言 TCL: Transaction Control Language

commit 确认, 提交(入库)、rollback 取消, 回滚, 撤销

4) 数据查询语言 DQL: Data Query Language

select 语句

5) 数据控制语言 DCL: Data Control Language

系统为多用户系统因此有隐私权限问题: grant 授权、revoke 回收权限

1.3 数据库 (DB)

DATABASE 关系数据库使用关系或二维表存储信息。

关系型数据库管理系统 (EDBMS): Relationship Database Management System 是一套软件, 用于在数据库中存储数据、维护数据、查询数据等。

1.4 数据库种类

Oracle 10g (Oracle)、DB2 (IBM)、SQL SERVER (MS)

1.5 数据库中如何定义表

先画列即表头 (列名, 数据类型及长度, 约束); 数据类型有字符、数值 number、日期 date。

1.6 create database dbname 的含义

创建数据库即创建可用空间, 创建出一堆数据文件 data file

1.7 安装 DBMS

职位: DBA 数据库管理员 (DataBase Administrator)

1.8 宏观上是数据-->database

开发流程: create table DML TCL -> DQL select

1.9 远程登录: telnet IP 地址

sql developer 在 linux 系统-->连接-->database 在 solaris 系统

1.10 TCP/IP 通信协议

两台机器上的两个应用程序要通信, 必须依赖网络, 依赖 TCP/IP 通信协议。

IP: IP 协议包中提供要连接机器的 IP 地址, 用于标识机器。

TCP: TCP 协议包中提供与机器上的哪个具体应用程序通信, 通过端口号实现, oracle 数据库服务缺省端口为 1521, 用于标识 Oracle 此数据库应用。

1.11 数据库建连接必须提供以下信息

ip 地址 (确认机器)、port 号 (确认进程 (程序) 确认 Oracle)

SID: 一个端口可以为多个 oracle 数据库提供监听, 因此还需要提供具体的数据库名。(确认数据库里的哪个数据库)

username、password: 要想访问数据库, 必须是该数据库上一个有效的用户。(确认身份)

1.12 一台机器可跑几个数据库, 主要受内存大小影响

1.13 源表和结果集

源表: 被查询的表 结果集: select 语句的查询结果

1.14 几个简单命令

show user: 查看当前用户 desc 表名: 查看表结构

drop table 表名 purge;删除表, Oracle 中删除表不是真正的删除, 而是占空间的移动到别的地方, 因为了不占空间, 真正的删除需要用 purge。

delete from 表名: 删除表中所有值; 若加上 where 列名=value 则删除某列中的值

1.15 tarena 给 jsd1304 授权

```
connect tarena/tarena
grant select on account to jsd1304;           grant select on service to jsd1304;
grant select on cost to jsd1304;
jsd1304 select tarena 的表
connect jsd1304/jsd1304
create synonym 创建同义词          create synonym account for tarena.account;
create synonym service for tarena.service;  create synonym cost for tarena.cost;
```

1.16 课程中使用的 5 个表

表 - 1 资费信息表 (在SQL课上使用单位费用 : 元/小时, 即不加灰部分)

ID 资 费 编 号	NAME 资费 名称	BASE_ COST 月固定 费用	BASE_ DURA TION 月包在 线时长	UNIT_ COST 单位 费用	S T A T U S 状 态	DESCR 资费 说明	CREA TIME 创建 时间	STAR TIME 启用 时间
1	5.9 元套餐	5.90	20	0.0111 0.40		5.9 元 20 小时/月,超出部分 0.0111 分/秒 (0.4 元/时)		
2	6.9 元套餐	6.90	40	0.0083 0.30		6.9 元 40 小时/月,超出部分 0.0083 分/秒 (0.3 元/时)		
3	8.5 元套餐	8.50	100	0.0056 0.20		8.5 元 100 小时/月,超出部分 0.0056 分/秒 (0.2 元/时)		
4	10.5 元套餐	10.50	200	0.0028 0.10		10.5 元 200 小时/月,超出部 分 0.0028 分/秒 (0.1 元/时)		
5	计时收费			0.0139 0.50		0.0139 分/秒 (0.5 元/时), 不使用不收费		
6	包月	20.00				每月 20 元,不限制使用时间		

表 - 2 帐务信息表

ID 帐务 帐号 编码	REC OM MEN DER_ ID 推荐 人帐 务帐 号 ID	LOGIN - NAME 登录 Net CTOSS 系统的 名称	LOGIN _PASS WD 登录 Net CTOSS 的口令	ST A T U S	CREATE_ DATE 帐务帐号创 建日期	REAL_ NAME 客户 姓名	IDCARD_NO 身份证号	TELEP HONE 电话
1005		taiji001	256528	1	15-MAR-08	zhangsanfeng	41038119430225 6528	136693 51234
1010		xl18z6 0	190613	1	10-JAN-09	guojing g	33068219690319 0613	133389 24567
1011	1010	dgbf70	270429	1	01-MAR-09	huangrong	33090219710827 0429	136378 11357
1015	1005	mjjzh6 4	041115	1	12-MAR-10	zhang wiji	61012119890604 1115	135729 52468
1018	1011	jmdxj0 0	010322	1	01-JAN-11	guofur on g	35058120020101 0322	186178 32562
1019	1011	ljxj90	310346	1	01-FEB-12	luwush uang	32021119930731 0346	131864 54984
1020		lohxhd2 0	012115	1	20-FEB-12	weixia obao	32102220001001 2115	139534 10078

表 - 3 业务信息表

ID 业务 帐号 编号	ACCOU NT_ID 帐务帐号 编号	UNIX_HOST UNIX 服务器 IP 地址	OS_ USERNAME UNIX 服务器 OS 帐号	LOGIN_ PASSWD 登录 UNIX 服务器口令	STAT US 状态	CREATE_ DATE 创建日期	COST_ ID 资费 编码
2001	1010	192.168.0.26	guojing	guo1234	0	10-MAR-09	1
2002	1011	192.168.0.26	huangr	huang234	0	01-MAR-09	1
2003	1011	192.168.0.20	huangr	huang234	0	01-MAR-09	3
2004	1011	192.168.0.23	huangr	huang234	0	01-MAR-09	6
2005	1019	192.168.0.26	luwsh	luwu2345	0	10-FEB-12	4
2006	1019	192.168.0.20	luwsh	luwu2345	0	10-FEB-12	5
2007	1020	192.168.0.20	weixb	wei12345	0	10-FEB-12	6
2008	1010	192.168.0.20	guojing	guo09876	0	11-FEB-12	6

表 - 4 UNIX服务器信息表

ID unix 服务器的 ip 地址	NAME 主机名	LOCATION 主机所在位置
192.168.0.26	sunv210	beijing
192.168.0.20	sun-server	beijing
192.168.0.23	sun280	beijing
192.168.0.200	Ultra10	beijing

表 - 5 年龄分段信息表

ID 年龄分段编号	NAME 年龄分段的名称	LOWAGE 年龄段下限	HIAGE 年龄段上限
0	少年逆反期	11	14
1	少年成长期	15	17
2	青年青春期	18	28
3	青年成熟期	29	40
4	中年壮实期	41	48
5	中年稳健期	49	55
6	中年调整期	56	65
7	老年初老期	66	72

七、select from 语句

LICHOO

2.1 select 语句功能

- 1) 投影操作：结果集是源表中的部分“列”
- 2) 选择操作：结果集是源表中的部分“行”
- 3) 选择操作+投影操作：结果集是源表中的部分“行”部分“列”
- 4) 连接操作 join：多表查询，结果集来自多张表，把多张的记录按一定条件组合起来

2.2 select 语句基本语法

- 1) select colname(列名) from tablename (表名)
- 2) select 中指定多个列名，则用“逗号”分隔：select colname1,colname2 from tablename
- 3) * 号表示所有列：select * from tablename
- 4) select 语句：可有多个子句
- 5) select 子句：投影操作（列名）、列表达式、函数、from 子句等

2.3 列别名

- 1) 给列起一个别名，能够改变一个列、表达式的标识。
- 2) 不写的话默认都是转成大写。 3) 适合计算字段。
- 4) 在原名和别名之间可以使用 as 关键字。
- 5) 别名中包含空格、特数字符或希望大小写敏感的，用“”双引号将其括起来。

2.4 算术表达式

在 number 类型上使用算术表达式（加减乘除）。

eg: 一个月使用了 250 小时,每种资费标准下应缴纳的费用(首次实现)

```
select base_cost + (250 - base_duration)*unit_cost fee from cost;
```

2.5 空值 null 的处理

未知的，没写数

- 1) 空值不等于 0
- 2) 空值不等于空格
- 3) 在算术表达式中包含空值导致结果为空
- 4) 在算术表达式中包含空值需要用空值转换函数 nvl 处理

2.6 nvl(p1,p2)函数

空值转换函数

- 1) 两个参数类型要一致！
- 2) 参数的数据类型可以是数值 number、字符 character、日期 date
- 3) 但 null 转成字符串，null 也要用 to_char()转化。
- 4) 实现过程：

```
if p1 is null then return p2  
else      return p1  
end if
```

- 5) 实现空值转换：null->非 null 值 0

eg: 一个月使用了 250 小时,每种资费标准下应缴纳的费用(再次实现)

```
select nvl(base_cost,0) + (250 - nvl(base_duration,0)) * nvl(unit_cost,0) fee from cost;
```

2.7 拼接运算符 ||

表达字符（串）的拼接，可以将某几列或某列与字符串拼接在一起。

```
select colname1||colname2 from tablename
```

2.8 文字字符串

select 语句后面可以包含的文字值：字符、表达式、数字。

1) 字符常量（或字符串）必须用 ‘’ 单引号括起来，作为“定界符”使用。

2) 表达单引号本身，需要两个单引号 '()' 1, 4 定界 2, 3 表单引号。

3) 对于文字值每行输出一次。

eg: 显示客户姓名的身份证号是.....

```
select real_name || ''''s IDCARD NO is ' || idcard_no || '.' client from account;
```

4) 函数转换大小写，尽量在进入数据时操作。

2.9 消除重复行

distinct 去重复行（对整条记录返回的结果去重，不是对后面的某个列去重），若后面有多列，则所有列联合起来唯一，即每列的值都可以重复，但组合不能重复。

eg1: 哪些 unix 服务器提供远程登录业务

```
select distinct unix_host from service;
```

eg2: 每一台 unix 服务器在哪些天开通了远程登录业务

```
select distinct unix_host,create_date from service;
```

2.10 其他注意事项

1) 调常量时用单行单列的 dual 表，系统提供的表。

2) invalid identifier 无效标识名，列名不。

3) table or view does not exist 表名不对。

八、SQL 语句的处理过程

LICHOO

3.1 SQL 语句处理过程

用户进程 sqlplus → 建立连接 → 服务进程 Server process oracleSID
↑ -- 创建会话 -- Oracle server

3.2 处理一条 select 语句

1) 分析语句:

① 搜索是否有相同语句

② 用 hash value 计算 select 语句是否长得一样：大小写，关键字，空格要都一样，不一样则为两条语句，则服务进程会重新分析。若为统一语句，则直接从内存拿执行计划，计算结果

③ 检查语法、表名、权限 ④ 在分析过程中给对象加锁

⑤ 生成执行计划

2) 绑定变量：给变量赋值

3) 执行语句：

4) 获取数据：将数据返回给用会进程

九、where 子句

LICHOO

用 where 子句对表里的记录进行过滤，where 子句跟在 from 子句后面。

4.1 where 子句后面可以跟什么

跟条件表达式：列名、常量、比较运算符（单、多值运算符）、文字值；不能跟组函数！不能跟列别名！

- ◆ 注意事项：对列不经过运算的条件表达式效率会更高，建议在写 where 子句时尽量不要对列进行运算。

eg: 一年的固定费用为 70.8 元，计算年包在线时长

```
select base_duration*12 ann_duration from cost where base_cost*12=70.8; 没下面效率高  
select base_duration*12 ann_duration from cost where base_cost=70.8/12;
```

4.2 语法和执行顺序

语法顺序：select from where

执行顺序：from where select

4.3 字符串是大小写敏感的，在比较时严格区分大小写

- 1) upper(): 函数将字符串转换成大写。
- 2) lower(): 函数将字符串转换成小写。
- 3) initcap(): 函数将字符串转换成首字符大写（是将列中的值大小写转换然后去和等号后的字符串比，而不是把转字符串转换去和列比）。

eg: 哪些 unix 服务器上开通了 os 帐号 huangr

```
select unix_host,os_username from service  
where os_username = 'huangr';(有结果)  
where lower(os_username)='HUANGR';(无结果)  
where lower(os_username)='huangr';(有结果)  
where upper(os_username)='HUANGR';(有结果)
```

4.4 where 子句后面可以跟多个条件表达式

条件表达式之间用 and、or 连接，也可用 () 改变顺序。

4.5 between and 运算符

表示一个范围，是闭区间，含义为大于等于并且小于等于。

eg: 哪些资费的月固定费用在 5 元到 10 元之间

```
select base_duration,base_cost,unit_cost from cost  
where base_cost >= 5 and base_cost <= 10;      where base_cost between 5 and 10;
```

4.6 in 运算符（多值运算符）

表示一个集合，是离散值，含义为等于其中任意一个值，等价于 any。

eg: 哪些资费的月固定费用是 5.9 元,8.5 元,10.5 元

```
select base_duration,base_cost,unit_cost from cost  
where base_cost = 5.9 or base_cost = 8.5 or base_cost = 10.5;  
where base_cost in(5.9,8.5,10.5);          where base_cost =any(5.9,8.5,10.5);
```

4.7 like 运算符

在字符串比较中，可用 like 和通配符进行模糊查找。

1) 通配符：%表示 0 或多个字符；_表示任意“一个”字符（要占位的）。

◆ 注意事项：若要查找%和_本身，则需要 escape 进行转移。

eg: 哪些 unix 服务器上的 os 帐号名是以 h 开头的

```
select os_username from service where os_username like 'h%'
```

eg: 哪些 unix 服务器上的 os 帐号名是以 h_开头的

```
select os_username from service where os_username like 'h\_%' escape '\';
```

4.8 is null 运算符

测试 null 值需要用 is null。

1) null 不能用等于号“=”和不等于号“<>”跟任何值比较，包括它自身。所以不能用“=”和“<>”来测试是否有空值。

2) 即：null=null 是不成立的；null 不等于 null 也不成立；null 和任何值比较都不成立。

eg: 列出月固定费用是 5.9 元，8.5 元，10.5 元或者没有月固定费。

```
select base_duration,base_cost,unit_cost from cost
```

where base_cost in (5.9,8.5,10.5,null);(错误)

where base_cost in (5.9,8.5,10.5) or base_cost is null;(正确)

4.9 比较和逻辑运算符（单值运算符）

1) 比较运算符：= > >= < <=

2) SQL 比较运算符：between and、in、like、is null

3) 逻辑运算符：and、or、not

4.10 多值运算符 all、any

1) >all：大于所有的，等价于 >(select max()…).

2) >any：大于任意的，等价于 >(select min()…).

4.11 运算符的否定形式

1) 比较运算符：<> != ~=

2) SQL 比较运算符：not between and not in not like is not null

◆ 注意事项：

❖ in 相当于=or =or =or 等价于 any

❖ not in 等价于 <>and <>and <>and 等价于 <>all

❖ not between and 小于下界 or 大于上界

❖ 集合中有 null，对 in 无影响；但对 not in 有影响，有一个就没有返回值！

eg: 哪些资费信息的月固定费用不是 5.9 元,8.5 元,10.5 元

```
select base_duration,base_cost,unit_cost from cost
```

```
where nvl(base_cost,0)<> 5.9 and nvl(base_cost,0)<> 8.5 and nvl(base_cost,0)<> 10.5;
```

```
where nvl(base_cost,0) not in (5.9,8.5,10.5);
```

十、order by 子句

select 语句输出的结果按记录在表中的存储顺序显示，order by 子句能够改变记录的输出顺序。

order by 子句对查询出来的结果集进行排序，即对 select 子句的计算结果排序。

5.1 语法和执行顺序

语法顺序：select from where order by

执行顺序：from where select order by

5.2 升降序

ASC—升序，可以省略，默认值

order by nvl(base_cost,0);

DESC—降序

order by unix_host,create_date desc;

◆ 注意事项：order by 是 select 语句中最后一个子句

5.3 null 值在排序中显示

1) 被排序的列如果包含 null 值，用 ASC 方式 null 值的在最后；

2) 用 DESC 方式 null 在最前面；

5.4 order by 后面可以跟什么

可以跟列名、列别名、列位置（数字）、表达式、函数。

order by 1: 表示列位置为 1 的列

select 1 from: 表示常量 1

eg: 按年固定费用从大到小的顺序显示资费信息

方式一： select id,base_cost*12 ann_cost,base_duration ann_duration from cost
order by base_cost desc;

方式二： select id,base_cost ann_cost,base_duration ann_duration from cost
order by base_cost*12 desc; 排序的效果和上面是一样的，但前一个效率高。

5.5 多列排序

order by 子句后面可以跟多列，而 order by 后面的列可以不出现在 select 后面。结果集先按第一列升序排列，若列值一样，再按第二列降序排列。

eg: 按 unix 服务器 ip 地址升序，开通时间降序显示业务帐号信息

select id,unix_host,os_username,create_date from service
order by unix_host,create_date desc;

十一、单行函数的使用

SQL 函数的两种类型：单行函数、多行函数（组函数）。

单行函数：数值类型、日期类型、字符类型、转换函数。处理一列数据，返回一个结果。

6.1 数值类型

1) 定义：create table tablename

```
( c1  number,c2  number(6),c3  number(4,3),c4  number(3,-3),c5  number(2,4) );
```

2) 数值类型说明

①number：不写数值，表可写 38 位数

②number(6): 6 位整数	999999.1	999999
-------------------	----------	--------

③number(4,3): 数字 4 位，小数点占 3 位，四舍五入	1.234567	1.235
------------------------------------	----------	-------

④number(3,-3): 小数点前三位不写数，四舍五入，然后有效位 3 位	1234
---	------

1000

⑤number(2,4): 小数点后 4 位，有效位 2 位	0.00991	0.0099
--------------------------------	---------	--------

3) 数值函数：参数类型为 number

①round(): 四舍五入函数，“缺省转成数字”；也可对日期

②trunc(): 截取函数（不管多大值直接舍去）；也可对日期

eg: round 和 trunc

round(45.923, 2): 45.92	round(45.923, 0): 46	round(45.923, -1): 50
-------------------------	----------------------	-----------------------

trunc(45.923, 2): 45.92	trunc(45.923): 45	trunc(45.923, -1): 40
-------------------------	-------------------	-----------------------

6.2 日期类型

1) Oracle 用 7 个字节来存储日期和时间：世纪、年、月、日、时、分、秒。Date 不存在定宽度，就是 7 个字节。

2) 缺省（默认）日期格式为 DD-MON-RR，格式敏感。

3) sysdate 是一个系统函数，返回当前系统时间和日期。

4) 改变 session（会话）中的日期格式：session 和 connection 是同时建立的，两者是对同一件事情的不同层次的描述。connection 是物理上的客户机同服务器端的通信链路；session 是逻辑上的用户同服务器的通信交互，SQL 语句的运行环境。

eg: 显示的日期包含世纪、年、月、日、时、分、秒

alter session set nls_date_format = 'yyyy mm dd hh24:mi:ss';
--

5) 日期格式

yyyy	用数字表达的四位年（2013 年）
mm	用数字表达的两位月（01 月）
dd	用数字表达的两月日（01 日）
hh24	用数字表达的 24 进制的小时（20 点）
h12	用数字表达的 12 进制的小时（8 点）
mi	用数字表达的分钟（30 分）
ss	用数字表达的小时（30 秒）
D	用数字表达的一周内的第几天（周日：1）
day	用全拼表达的星期几（sunday）
month	用全拼表达的月（march）

mon

用简拼表达的月 (mar)

eg: 案例

```
select to_char(sysdate,'DDD') from dual;年中的第几天  
select to_char(sysdate,'DD') from dual;月中的第几天  
select to_char(sysdate,'D') from dual;星期中的第几天
```

6) 在数据库中如何处理日期类型

```
create table test(c1 date);  
insert into test values ('01-JAN-08');  
insert into test values ('2008-08-08'); (报错)  
insert into test values (to_date('2008-08-08','yyyy-mm-dd'));  
select c1 from test;  
select to_char(c1,'yyyy-mm-dd') from test;
```

在 create table 中定义日期类型 date 时一定不能指定宽度。日期在数据库中用固定的 7 个字节存储，表示世纪、年、月、日、时、分、秒。缺省的日期格式为'DD-MON-RR', '01-JAN-08' 符合缺省日期格式可以插入表中，因为系统会自动调用 to_date 函数将它转成日期。

'2008-08-08'插入时报错，原因是不符合缺省格式，需要手工使用函数 to_date 对字符串的格式进行说明，如'2008-08-08'的格式说明串为'yyyy-mm-dd'。select 时日期按缺省日期格式显示，若用指定日期格式，需要使用 to_char 函数。

7) 日期与字符串相互转换：

to_date(char,date)函数：将字符串转换成一个日期值。对应 java 中 parse。

to_char(date,char)函数：第一个参数为要处理的日期，第二个参数为格式；可获取一个日期的任意一部分信息；对应 java 中 format。

eg: 创建一张表，包含 date 类型的列，插入 2008 年 8 月 8 日 8 点 8 分 8 秒并显示。

```
insert into test values (to_date('2008-08-08 08:08:08','yyyy-mm-dd hh24:mi:ss'));  
select to_char(c1,'yyyy-mm-dd hh24:mi:ss') from test;
```

◆ 注意事项：

- ❖ 格式必须用单引号括起来，并且大小写敏感。
- ❖ 必须是有效的日期格式。
- ❖ fm 能去掉前导 0 和两端的空格。
- ❖ 对日期去重复问题，to_char 获取当天日期即可，时分秒忽略，加上 distinct 即可做到

eg: 案例

```
where to_char(create_date,'mm')='03';
```

若等式右边写成 '3'，'03' = '3' 不成立，需要在 'mm' 前增加 'fm'。

```
where to_char(create_date,'fm'mm')='3'
```

```
where to_number(to_char(create_date,'mm'))= 3;
```

若等式右边写成 3，'03' = 3 成立，'03' 是字符类型，3 是数字类型，等式两边相等，说明系统做了隐式数据转换，缺省做法将字符转化为 number。

8) 日期函数：参数类型为 date

①add_months(): 一个日期加、减一个月。

②months_between(): 两个日期之间相差多少个月。

③last_day(): 同一个月的最后一天 ④next_day(): 根据参数，出现下一个的日期。

eg1: 昨天，今天，明天

```
alter session set nls_date_format = 'yyyy mm dd hh24:mi:ss';
```

```
select sysdate-1,sysdate,sysdate+1 from dual;
```

eg2: 十分钟之后

```
alter session set nls_date_format = 'yyyy mm dd hh24:mi:ss';
```

```
select sysdate,sysdate + 1/144 from dual;
```

eg3: 每台 unix 服务器上的 os 帐号开通了多长时间 (以天为单位)

```
select unix_host,os_username,create_date,round(sysdate - create_date) days from service;
```

eg4: 上个月的今天, 今天, 下个月的今天

```
alter session set nls_date_format = 'yyyy mm dd hh24:mi:ss';
```

```
select add_months(sysdate,-1),sysdate,add_months(sysdate,1) from dual;
```

eg5: 当前月的最后一日

```
alter session set nls_date_format = 'yyyy mm dd hh24:mi:ss';
```

```
select last_day(sysdate) from dual;
```

eg6: 用户注册多长时间了

```
select trunc(months_between(sysdate,create_date)) from service;
```

6.3 字符类型

1) 定义: create table tablename

```
(c1 char(10), c2 varchar2(10));
```

2) char 和 varchar2 区别:

①varchar2 必须定义长度, 按字符串的实际长度存, 最大长度 4000 字节, 更省空间。

②char 可以不定义长度, 默认为 1, 按定义长度存, 最大长度 2000 字节, 操作更快。

③列的取值是定长, 定义成 char 类型。

④列的取值长度不固定, 定义成 varchar2。

◆ 注意事项:

❖ 在字符串比较中, varchar2 按实际字符串比, 对空格是敏感的, 对大小些敏感。

❖ char 会将短字符串补齐后, 再与字符串比, 对空格不敏感。

❖ varchar 类型是 ANSI 定义的, varchar2 类型是 Oracle 定义的, 目前是等价的。但如果 ANSI 对 varchar 类型定义有变化, 则 Oracle varchar2 类型不变。

eg: 案例

varchar2(10):	'abc'='abc'	yes	;	'abc'='abc'	no
char(10):	'abc	'=abc'	yes	'abc	'='abc'

3) 字符函数: 参数类型为字符

①upper(): 函数将字符串转换成大写。②lower(): 函数将字符串转换成小写。

③initcap(): 函数将字符串转换成首字符大写 (是将列中的值大小写转换然后去和等号后的字符串比, 而不是把转字符串转换去和列比)。

④length(): 字符串的长度。⑤rpad()、lpad(): 将字符补成同样长度, l 和 r 表左右。

⑥rtrim()、ltrim(): 去除字符串, l 和 r 表左右, 与 fm 相同效果。

⑦concat(): 拼接函数与 “||” 相似。⑧substr(): 求子串函数。

eg: 相关操作

```

select rpad('FEBRARY',9,'*') from dual;
where to_char(create_date,'fmMONTH')='MARCH';
where rtrim(to_char(create_date,'MONTH'))='MARCH';
select concat('ab ','c')from dual;      ↓ 从左往右      ↓ 从右往左
select os_username,substr(os_username,1,2),substr(os_username,-2,2)from service;

```

6.4 转换函数

1) `to_number()`函数: 将字符(串)转换成 `number` 数值类型, 这也是系统的缺省做法, 即 `to_number('03')=3`。

- ◆ 注意事项: 若 `to_number` 函数处理的字符串为'ab', 则系统报错, 若转换后的值是十进制的, 则要求字符串必须是数字字符。

```
select to_number('ab') from dual; (报错 invalid number)
```

2) `to_char(date,char)`函数: 第一个参数为要处理的日期, 第二个参数为格式; 可获取一个日期的任意一部分信息; 对应 java 中 `format`。

3) 函数格式说明:

9	代表数位
0	定义宽度大于实际宽度时, 0 会被强制显示在前面, 以补齐位数
\$	美元符号
L	本地货币符号
.	小数点
,	每千位显示一个逗号

- ◆ 注意事项: 如果显示位数不足(定义宽度小于实际宽度), 用#代替。

eg1: 相关操作

```

select to_char(base_cost,'L99.99') from cost;
select to_char(base_cost,'L00.00') from cost;
select to_char(base_cost,'$00.00') from cost;

```

eg2: 显示月固定费用, 单位费用, 单位费用为 null, 显示 no unit cost

```
select base_cost,nvl(to_char(unit_cost),'no unit cost') unit_cost
```

3) `to_date(char,date)`函数: 将字符串转换成一个日期值。对应 java 中 `parse`。

4) `number`、字符、`date`间的转化

①`to_char()`: `number->字符` `date->字符`

②`to_number()`: `字符->number`

③`to_date()`: `字符->date`

5) 显式隐式转换

①隐式数据类型转换, 系统调用转换函数

```
where create_date like '%3%';隐式
```

②显式数据类型转换, 用户调用转换函数

```
where to_char(create_date,'mm')='03';显式
```

6.5 其他注意事项

1) `insert into 表名 values(1, 2, 3, 4, null)`有多列时, 插入值必须都写, 没值的也要写 `null`

2) `insert into 表名(C5) values(1234)` 表名最多 30 个字符且不能有特殊字符

3) `alter session set nls_language='AMERICAN',28-MAY-13`

alter session set nls_language='SIMPLIFIED CHINESE';28-5 月 -13

- 4) alter session set nls_territory = 'AMERICA';
alter session set nls_territory = 'CHINA';

LICHOOL

十二、SQL 语句中的分支

LICHOO

7.1 分支表达式

1) case when (then), 用于解决不同记录需要不同处理方式的问题。when 后面跟条件表达式, 当所有 when 条件都不满足时, 若有 else, 表达式的返回结果为其后的值, 否则返回 null 值。

2) 寻找 when 的优先级: 从上到下再多的 when, 也只有一个出口, 即其中有一个满足了表达式 expr 就马上退出 case。

3) else expr 和 return expr 的数据类型必须相同。

eg: 当月包在线时长为 20 小时, 单位费用涨 5 分, 为 40 小时涨 3 分, 其他不变 (用 CASE WHEN 实现)

```
select base_duration,unit_cost,case when base_duration=20 then unit_cost+0.05  
                                     when base_duration=40 then unit_cost+0.03  
                                     else unit_cost  
                                     end new_nuit_cost  
from cost;
```

7.2 分支函数

decode, 是简版的 case when。

1) decode(value,if1,then1,if2,then2,……,else)标识如果 value 等于 if1 时, 返回 then1。如果不等于任何一个 if 值, 则返回 else。

eg: 当月包在线时长为 20 小时, 单位费用涨 5 分, 为 40 小时涨 3 分, 其他不变 (用 decode 实现)

```
select base_duration,unit_cost,decode(base_duration,20,unit_cost+0.05,  
                                      40,unit_cost+0.03,  
                                      unit_cost)n_base_cost  
from cost;
```

十三、组函数

操作在一组行（记录）上，每组返回一个结果。

8.1 报表统计常用

- 1) avg(distinct|all|n): 平均值，参数类型只能为 number。
- 2) sum(distinct|all|n): 求和，参数类型只能为 number。
- 3) count(distinct|all|expr|*): 计数，参数类型为 number、字符、date。
- 4) max(distinct|all|expr): 最大值，参数类型为 number、字符、date。
- 5) min(distinct|all|expr): 最小值，参数类型为 number、字符、date。

◆ 注意事项：

- ❖ distinct 去重复时，会保留一个。

```
select count(distinct base_duration)from cost;//4, distinct 保留一个空，但 count 统计时不  
算
```

- ❖ count (*) 不管 null，统计“记录”数。
- ❖ count (列名) 返回的是列中非 null 值的数量。

8.2 缺省情况组函数处理什么值

所有的非空值。

8.3 当组函数要处理的所有值都为 null 时

count 函数返回 0，其他函数返回 null。

8.4 行级信息和组级信息

返回的结果集包含多条记录，是行级信息；返回的结果集包含一条记录，是统计汇总信息，是组级别的信息；两者不能同时显示出来！

处理方式：将行级信息变成组标识或进行组函数处理。

eg1：单位费用的总和、平均值、最大值、最小值个数

```
select sum(unit_cost) sum1,avg(unit_cost) avg1,max(unit_cost) max1,  
min(unit_cost) min1,count(unit_cost) cnt from cost;
```

eg2：若 null 值参与运算，必须将 null 值转换成非 null 值

```
select avg(nvl(unit_cost,0)),sum(unit_cost)/count(*) from cost;
```

eg3：若 unit_cost 列中参与运算的数据都为 null，avg(unit_cost) 的函数值为 null，count(unit_cost) 的函数值为 0。

```
select avg(unit_cost),count(unit_cost) from cost where unit_cost is null;
```

eg4：每台 unix 服务器上开通的 os 帐号数即开户数？

```
select unix_host,count(os_username) from service group by unix_host;
```

eg5：tarena26 (192.168.0.26) 上开通的 os 帐号数即开户数？

```
select max(unix_host),count(os_username) cnt from service where unix_host = '192.168.0.26';  
用 min(unix_host) 也可
```

十四、group by 子句

LICHOOL

将表中的记录进行分组

9.1 语法和执行顺序

语法顺序：select from where group by order by

执行顺序：from where group by select order by

9.2 分组过程

根据 group by 子句指定的表达式，将要处理的数据分成若干组（若有 where 子句即为通过条件过滤后的数据）。每组有唯一的组标识，组内有若干条记录，根据 select 后面的组函数对每组的记录进行计算，每组对应一个返回值。

9.3 常见错误

若没有 group by 子句，select 后面有一个是组函数，则其他都必须是组函数（记录（行）信息和组信息不能放一起，要么都是组函数，要么都是单行函数）。

若有 group by 子句，select 后面跟 group by 后面跟的表达式以及组函数，其他会报错。

9.4 多列分组

包含多列用“,” 分开，分组的个数多了，每组的记录少了。

eg: 根据 unix 服务器 ip 地址、开通时间统计开通的 os 帐号数即开户数

```
select unix_host,to_char(create_date,'yyyymmdd') create_date,count(os_username) cnt  
from service  
group by unix_host,to_char(create_date,'yyyymmdd');
```

十五、**having** 子句

LICHOO

对分组过滤。

10.1 语法和执行顺序

语法顺序: select from where group by having order by

执行顺序: from where group by having select order by

10.2 执行过程

行被分组，将 having 子句的条件应用在每个分组上，只有符合 having 条件的组被保留，再应用 select 后面的组函数对每组的数据进行处理。

10.3 where 和 having 区别

1) where: 过滤的是行（记录），后面可跟任意列名，单行函数，不能跟组函数（无法对应到具体记录），先执行，不允许用列别名。

2) having: 过滤的是分组（组标识、每组数据的聚合结果），后面只能包含 group by 后面的表达式和组函数（能表达组信息的），后执行，不允许用列别名。

eg1: 哪些 unix 服务器开通的 os 帐号数即开户数多于 2 个

```
select unix_host,count(os_username) cnt from service  
group by unix_host  
having count(os_username)> 2;
```

eg2: 哪些 unix 服务器在哪几天的开户数多于 1 个

```
select unix_host,to_char(create_date,'yyyymmdd') create_date,count(os_username) cnt  
from service  
group by unix_host,to_char(create_date,'yyyymmdd')  
having count(os_username)> 1;
```

十六、非关联子查询

LICHOO

子查询就是在一条 SQL (DDL、DML、TCL、DQL、DCL) 语句中嵌入 select 语句。

11.1 语法

```
select colname, … from tablename where expr operator(select colname2 from subtablename);
```

11.2 子查询的执行过程

先执行子查询，子查询的返回结果作为主查询的条件，再执行主查询。子查询只执行一遍。若子查询的返回结果为多个值，Oracle 会自动去掉重复值后，再将结果返回给主查询。
注意事项：不需要 distinct，会自动去重的。

eg1：哪些 os 帐号的开通时间是最早的

```
select unix_host,os_username,create_date from service  
where create_date = (select min(create_date) from service);
```

eg2：哪些 os 帐号的开通时间比 unix 服务器 192.168.0.26 上的 huangr 晚

```
select unix_host,create_date,os_username from service  
where create_date > (select create_date from service  
                  where os_username = 'huangr' and unix_host = '192.168.0.26');
```

eg3：哪些 os 帐号的开通时间比 huangr 晚？（多台 unix 服务器上都有名为 huangr 的 os 帐号）

```
select unix_host,create_date,os_username from service  
where create_date > all (select create_date from service  
                          where os_username = 'huangr');大于所有的  
where create_date > (select max(create_date) from service  
                          where os_username = 'huangr');大于最大的  
where create_date > any (select create_date from service  
                          where os_username = 'huangr');大于任意一个  
where create_date > (select min(create_date) from service  
                          where os_username = 'huangr');大于最小的
```

11.3 常见错误

单行子查询返回多条记录！此时要注意运算符的选择：

- 1) 若子查询的返回结果仅为一个值，可用单值运算符，如“=”号。
- 2) 若子查询的返回结果可能为多个值，必须用多值运算符，如 in 等。

eg：哪些客户是推荐人

```
select real_name from account  
where id in (select recommender_id from account);
```

11.4 子查询与空值

若子查询的返回结果中包含空值 null，并且运算为 not in，那么整个查询不会返回任何行。not in 等价于 <> all，任何值跟 null 比（包括 null 本身），结果都不为 true。

eg：哪些客户不是推荐人

```
select real_name from account  
where id not in (select recommender_id from account where recommender_id is not null);
```

11.5 多列子查询

where 子句后面可以跟多列条件表达式。

eg1: 哪些 os 帐号的开通时间是所在 unix 服务器上最早的? (每台 unix 服务器上最早开通的 os 帐号)

```
select unix_host,os_username,create_date from service  
where (unix_host,create_date) in (select unix_host,min(create_date) from service  
group by unix_host);
```

eg2: 哪些 os 帐号的开通时间比所在 unix 服务器上最早开通时间晚九天

```
select unix_host,os_username,create_date from service  
where (unix_host,to_char(create_date,'yyyymmdd')) in  
(select unix_host,to_char(min(create_date) + 9,'yyyymmdd') from  
service  
group by unix_host);
```

十七、关联子查询

LICHOO

关联子查询采用的是循环（loop）的方式。

12.1 语法

```
select column1, … from table1 o where column1 operator  
      (select column1, column2 from table2 i where i.expr1=o.expr2);
```

12.2 执行过程

- 1) 外部查询得到一条记录（查询先从 outer 表中读取数据），并将其传入到内部的表查询。
- 2) 内部查询基于传入的值执行。
- 3) 内部查询从其结果中把值传回到外部查询，外部查询使用这些值来完成处理，若符合条件，outer 表中得到的那条记录就放入结果集中，否则放弃。
- 4) 重复执行 1—3，直到把 outer 表中的所有记录判断一遍。子查询执行 n 遍。

eg:哪些 os 帐号的开通天数比同一台 unix 服务器上的平均开通天数长。

```
select unix_host, os_username, create_date, round(sysdate - create_date) open_age  
from service o  
where round(sysdate - create_date) > (select avg(round(sysdate - create_date)) from service i  
                                         where o.unix_host = i.unix_host);
```

12.3 exists

exists 采用的是循环（loop）的方式，判断 outer 表中是否存在在 inner 表中找到的一条匹配的记录。

12.4 exists 执行过程

- 1) 外部查询得到一条记录（查询先从 outer 表中读取数据），并将其传入到内部的表进行查询。
- 2) 对 inner 表中的的记录依次扫描，若根据条件，存在一条记录与 outer 表中的记录匹配，则立即停止扫描，返回 true，将 outer 表中的记录放入结果集中；若扫描了全部记录，没有任何一条记录符合匹配条件，则返回 false，outer 表中的该记录被过滤掉，不能出现在结果集中。
- 3) 重复执行 1—2，直到把 outer 表中的所有记录判断一遍。

eg1：哪些客户是推荐人

```
select real_name from account o where exists (select 1 from account i  
                                              where o.id = i.recommender_id);  
//1 可随便写，不关心结果什么样，只关心是否有满足的条件返回
```

eg2：哪些客户申请了远程登录业务

非关联子查询：

```
select real_name from account where id in (select account_id from service);
```

关联子查询：

```
select real_name from account o where exists (select 1 from service i  
                                              where o.id = i.account_id);
```

12.5 not exists

采用的是循环（loop）的方式，判断 outer 表中是否存在记录（它能在 inner 表中找到匹配的记录）。

12.6 not exists 执行过程

1) 外部查询得到一条记录（查询先从 outer 表中读取数据），并将其传入到内部的表进行查询。

2) 对 inner 表中的记录依次扫描，若根据条件，存在一条记录与 outer 表中的记录匹配，则立即停止扫描，返回 false，将 outer 表中的记录过滤掉，不能出现在结果集中；若扫描了全部记录，没有任何一条记录符合匹配条件，则返回 true，outer 表中的该记录放入结果集中。

3) 重复执行 1—2，直到把 outer 表中的所有记录判断一遍。

eg1：哪些客户不是推荐人

```
select real_name from account o where not exists (select 1 from account i  
where o.id = i.recommender_id);
```

eg2：哪些客户没有申请远程登录业务

非关联子查询：

```
select real_name from account where id not in (select account_id from service);
```

关联子查询：

```
select real_name from account o where not exists (select 1 from service i  
where o.id = i.account_id);
```

12.7 in 和 exists 比较

1) exists 是用循环（loop）的方式，有 outer 表的记录数决定循环次数，对于 exists 影响最大，所以，外表的记录数要少。

2) in 先执行子查询，子查询的结果去重之后，再执行主查询，所以，子查询的返回结果越少，越适合用该方式。

十八、多表查询

LICHOO

结果集中的记录保存在多张表中。

13.1 按范式要求设计表结构

第二范式：每个非主属性必须完全依赖于主属性（主键 pk 列）（避免多对多合表造成数据冗余）。

第三范式：每个非主属性不能依赖于另一个非主属性（避免一对多合表造成数据冗余，不一致）。

13.2 多表连接的种类

交叉连接（cross join）、内连接（inner join）、外连接（outer join）。

13.3 交叉连接

数学中的组合问题。

1) 假设 table1 表中有 m 条记录, table2 表中有 n 条记录, 交叉连接产生的结果集为 $m \times n$. 该结果产生的结果集为笛卡尔积。

2) 语法:

```
select tablename1.colname1,tablename2.colname2 from tablename1 cross join tablename2;
```

eg: 案例

```
select a.real_name,a.id,s.account_id,s.unix_host,s.os_username  
from account a cross join service s;
```

13.4 内连接

核心解决匹配问题，建议用 on and and 多条件组合，不用 where。

1) 语法:

```
select tablename1.colname1,tablename2.colname2 from tablename1 join tablename2  
on tablename1.colname1=tablename2.colname2 and 其他条件;
```

2) 如果有多个条件表达式，on 关键字后面跟一个，其余用 and 条件连接。

eg: 客户 huangrong 在哪些 unix 服务器上申请了远程登录业务

```
select a.real_name,s.unix_host,s.os_username,s.create_date  
from account a join service s on a.id = s.account_id and a.real_name = 'huangrong';
```

3) 内连接原理一：

t1 和 t2 表作内连接，连接条件为 on t1.c1=t2.c2，假设 t1 表作驱动表，t2 表作匹配表，记录过程如下：

①从 t1 表中读取一条记 r1，若它的列 c1 值为 1

②根据该值到 t2 表中查找匹配的记录，即需要遍历 t2 表，从 t2 表中的第一条记录开始，若查找的记录的 c2 列的值为 1，我们就说这两条记录能够匹配上，那么 t1 的 r1 和 t2 中刚刚匹配的该条记录组合起来，作为结果集里的一条记录，否则检测 t2 表中的下一条记录。

③按照步骤 2 依次将 t2 表中所有的记录检测一遍，只要匹配就放入结果集中。

④从 t1 表中读取第二条记录，依次重复步骤 2 和 3，产生最终的结果集。

eg: 列出申请了远程登录业务的客户姓名以及在 unix 服务器上的开通信息

```
select a.real_name,s.unix_host,s.os_username,s.create_date
```

```
from account a join service s on a.id = s.account_id;
```

4) 内连接原理二:

t1 和 t2 表作内连接, 连接条件为 on t1.c1=t2.c2, 假设 t1 表作驱动表, t2 表作匹配表, 记录的匹配有如下三种情况:

- ①t1 表中的某条记录在 t2 表中找不到任何一条匹配的记录, 那么 t1 表中的该条记录不会出现在结果集中。
- ②t1 表中的某条记录在 t2 表中只有一条匹配的记录, 那么 t1 表中的该记录和 t2 表中匹配的记录组合成新的记录出现在结果集中。
- ③t1 表中的某条记录在 t2 表中有多条匹配的记录, 那么 t1 表中的该记录会和 t2 表中每一条匹配的记录组合成新的记录出现在结果集中。

◆ 注意事项: 内连接的核心为: 任何一张表里的记录一定要在另一张表中找到匹配的结果, 否则不能出现在结果集中。

5) 内连接原理三:

t1 和 t2 表作内连接, 连接条件为 on t1.c1=t2.c2, 以下两种方式都可以得到相同的结果集:

- ①一种 t1 作驱动表, t2 作匹配表
- ②一种 t2 作驱动表, t1 作匹配表
- ③无论那种方式, 最终得到的结果集都一样, 所不同的是效率。

6) 内连接的结果集结构:

t1.c1 t1.c2 t1.c3 t2.c1 t2.c2 t2.c3

7) 内连接的语句执行顺序:

先根据 on 和 and 条件对要连接的表进行过滤, 将过滤后的结果集进行内连接操作 (join on), 再根据 select 语句的定义生成最终的结果集。

注意事项: 内连接中使用 on 和 where 都可以。

8) from 后面可跟子查询

eg1: 列出客户姓名以及开通的远程登录业务的数量

方式一: 先连接再统计

```
select a.id,max(a.real_name),count(a.id)
from account a join service s on a.id = s.account_id
group by a.id;
```

方式二: 先统计再连接, 效率更高。

```
select a.real_name,count(a.id)
from account a join (select account_id count(id) cnt from service group by account_id) c
on a.id = c.account_id
```

eg2: 列出客户姓名以及他的推荐人 (考查了内连接、空值转换、decode)

```
select a1.real_name recommended,decode(a2.id,a1.id,'No Recommender',a2.real_name)
recommender from account a1 join account a2
on nvl(a1.recommender_id,a1.id) = a2.id;
```

9) 自连接:

①同一张表的行 (记录) 之间的匹配关系可以用同一张表的列之间的条件表达式描述。

②通过给表起别名, 将同一张表的列之间的关系转换成不同表的列之间的条件表达式。

eg: 哪些客户是推荐人

```
select distinct a2.id,a2.real_name from account a1 join account a2 on a1.recommender_id = a2.id;
```

10) 其他案例

eg1: 显示客户姓名, 开通的远程登录业务的数量。(结果集中只包含开通了远程登录业务的客户)

```
select t1.real_name,t2.cnt from account t1 join  
(select account_id,count(*) cnt from service group by account_id) t2  
on t2.account_id=t1.id;
```

◆ 注意事项: count(*)已经到了不得不起别名的地步, 组函数不可作与单行函数在一起显示的。

eg2: 显示客户姓名, 开通的远程登录业务的数量。(结果集中只包含开通了远程登录业务的客户)

```
select min(a.real_name),count(s.account_id)  
from account a join service s on a.id=s.account_id group by a.id;
```

◆ 注意事项:

- ❖ eg1 比 eg2 的效率高! 两个表出统计结果时, 一个表就能出来结果的就先单表统计再连接。否则, 就先连接再统计!!!!
- ❖ in (非关联子查询) exists (关联子查询) join (表查询) 都是在解决匹配问题。
- ❖ 匹配是记录和记录的匹配, 是逻辑上的匹配, 不一定非要是物理上独立个体的匹配, 也可在一个表的记录间相互匹配。

13.5 外连接

作用: ①把匹配和不匹配的都找出来。②只找不匹配的, 匹配的交给内连接作。

1) 语法: left right full 定驱动表的

from t1 left (outer) join t2 on t1.c1=t2.c2 outer 可省 左表为驱动表

from t1 right (outer) join t2 on t1.c1=t2.c2 右表为驱动表

from t1 full (outer) join t2 on t1.c1=t2.c2 左右表都为驱动表

2) 外连接原理一:

t1 和 t2 表作外连接, 连接条件为 from t1 left outer join t2 on t1.c1=t2.c2, t1 表必须作驱动表, t2 表作匹配表, 记录的匹配过程如下:

①从 t1 表中读取一条记 r1, 若它的列 c1 值为 1

②根据该值到 t2 表中查找匹配的记录, 即需要遍历 t2 表, 从 t2 表中的第一条记录开始, 若查找的记录的 c2 列的值为 1, 我们就说这两条记录能够匹配上, 那么 t1 的 r1 和 t2 中刚刚匹配的该条记录组合起来, 作为结果集里的一条记录, 否则检测 t2 表中的下一条记录。

③按照步骤 2 依次将 t2 表中所有的记录检测一遍, 只要匹配就放入结果集中。若扫描完后, t1 的 r1 记录在 t2 表中找不到任何匹配的记录, t2 表中模拟一条 null 记录与 t1 表中的 r1 组合起来, 放入结果集中。

④从 t1 表中读取第二条记录, 依次重复步骤 2 和 3, 产生最终的结果集。

3) 外连接原理二:

t1 和 t2 表作外连接, 连接条件为 from t1 left outer join t2 on t1.c1=t2.c2, t1 表必须作驱动表, t2 表作匹配表:

①外连接的结果集=内连接结果集+t1 表中匹配不上的记录和一条 null 记录 (按 t2

表的结构) 组成的记录的组合。

②外连接的核心可以将 t1 中匹配不上的记录 (按 on 条件在 t2 中找不到对应的匹配记录) 也显示出来, 而不像内连接直接过滤掉, 即 t1 中的记录一个都不少的出现在结果集中。

③外连接结果集的记录数不一定是驱动表的记录数 (结果集记录数 \geq 驱动表记录数)。

4) 外连接原理三:

t1 和 t2 表作外连接, 连接条件为 from t1 right outer join t2 on t1.c1=t2.c2, t2 表必须作驱动表, t1 表作匹配表:

①外连接的结果集=内连接结果集+t2 表中匹配不上的记录和一条 null 记录 (按 t1 表的结构) 组成的记录的组合。

②外连接的核心可以将 t2 中匹配不上的记录 (按 on 条件在 t1 中找不到对应的匹配记录) 也显示出来, 而不像内连接直接过滤掉, 即 t2 中的记录一个都不少的出现在结果集中。

③外连接结果集的记录数不一定是驱动表的记录数 (结果集记录数 \geq 驱动表记录数)。

5) 外连接原理四:

t1 和 t2 表作外连接, 连接条件为 from t1 full outer join t2 on t1.c1=t2.c2, t1 表必须作驱动表, t2 表作匹配表:

①外连接的结果集=内连接结果集+t1 表中匹配不上的记录和一条 null 记录 (按 t2 表的结构) 组成的记录+t2 表中匹配不上的记录和一条 null 记录 (按 t1 表的结构) 组成的记录的组合。

②外连接结果集的记录数不一定是 t1 表和 t2 表的记录数之和。

eg1: 列出客户姓名以及他的推荐人

```
select a1.real_name customer,nvl(a2.real_name,'No Recommender') recommender  
from account a1 left join account a2 on a1.recommender_id = a2.id;
```

eg2: 列出客户姓名以及所开通的远程登录业务的信息 (没有申请远程登录业务的客户也要出现在结果集中)

```
select a.id,a.real_name,s.unix_host,s.os_username from account a left join service s  
on a.id =  
s.account_id;
```

eg3: 哪些客户不是推荐人

```
select a1.real_name recommender  
from account a1 left join account a2 on a1.id = a2.recommender_id  
where a2.id is null;
```

6) 外连接语句的执行顺序

若 on 子句后面有 and 条件, 则现对匹配表进行过滤, 然后再进行外连接 (join on), 再对外连接的结果集用 where 子句进行过滤, 最后用 select 语句生成最终的结果集。on 和 where 后面都可以跟多个条件表达式, 表达式之间用 and 连接

eg: 哪些 UNIX 服务器上没有 os 帐号 weixb

```
select h.id,h.name,h.location  
from host h left join service s on h.id=s.unix_host and s.os_username='weixb'  
where s.id is null;
```

①先过滤 service 表, 用 s.os_username='weixb'

②过滤后的结果集作匹配表，host 表作驱动表，进行外连接，用 where 对外连接的结果集进行过滤，产生最终结果。

◆ 注意事项：

- ❖ 驱动表和匹配表的关系，也就是指驱动表中的记录和匹配表中的记录的关系，通过 on 联系；要想统计出正确的数量 count，必须统计匹配表的“非空列”！
- ❖ 对内连接 and、where 用谁都行，但外连接则有严格的使用位置。
- ❖ 过滤驱动表一定用 where 子句。

13.6 非等值连接

不同表没有共同属性的列，但两张表的列可以写成一个 SQL 条件表达式。

eg1：显示客户的年龄段

```
select t1.real_name,round((sysdate-t1.birthdate)/365) age,t2.name  
from account t1 join age_segment t2 on round((sysdate-t1.birthdate)/365)  
                                between t2.lowage and t2.hiage;
```

eg2：显示客户 huangrong 的年龄段

```
select t1.real_name,round((sysdate-t1.birthdate)/365) age,t2.name  
from account t1 join age_segment t2 on round((sysdate-t1.birthdate)/365)  
                                between t2.lowage and t2.hiage  
                                and real_name='huangrong';
```

eg3：显示青年年龄段中的客户数

```
select t1.real_name,round((sysdate-t1.birthdate)/365) age,t2.name  
from account t1 join age_segment t2 on round((sysdate-t1.birthdate)/365)  
                                between t2.lowage and t2.hiage  
                                and t2.name like'青年%';
```

eg4：显示各个年龄段的客户数（没有客户的年龄段的客户数为 0）

```
select max(t2.name),count(t1.id)  
from account t1 right join age_segment t2 on round((sysdate-t1.birthdate)/365)  
                                between t2.lowage and t2.hiage  
group by t2.id;搞清楚为何用 t1.id 统计（思考连接过程）;  
若没有客户的年龄段不用出现在结果集中则采用内连接。
```

13.7 表连接总结

1) 内连接，解决匹配问题

①等值连接：on 子句后有等值条件。

②非等值连接：不同表没有共同属性的列，但两张表的列可以写成一个 SQL 条件表达式。

③自连接：同一张表，通过起别名，表达列之间的关系。

2) 外连接，解决不匹配问题和表中所有记录出现在结果集

①等值连接：on 子句后有等值条件。

②非等值连接：不同表没有共同属性的列，但两张表的列可以写成一个 SQL 条件表达式。

③自连接：同一张表，通过起别名，表达列之间的关系。

3) 交叉连接，笛卡尔积

十九、集合

14.1 表连接主要解决的问题

- 1) 两张表记录之间的匹配问题。
- 2) 两张表记录之间的不匹配问题。
- 3) 匹配问题+不匹配问题。

14.2 集合运算

- 1) 若将两张表看成集合，匹配问题就是集合运算中的交集。
- 2) 若将两张表看成集合，不匹配问题就是集合运算中的差。
- 3) 匹配问题+不匹配问题就是集合运算中的并集。

14.3 集合运算符

- 1) **union**: 结果集为两个查询结果的并集，是去掉重复值的，最后有自动升序。
 - 2) **union all**: 结果集为两个查询结果的并集，是包含重复值的，输出效果为记录升序。
 - 3) **tersect**: 结果集为两个查询结果的交集，不包含重复值。
 - 4) **minus**: 结果集为属于第一个查询的结果集，但不属于第二个查询的结果集，即从第一个查询的结果集中减去他们的交集，不包含重复值； $A-B=C$, A 为被减数，B 为减数，C 为差；从 A 中减去和 B 中相同的部分。
- ◆ 注意事项：集合运算要求两个 select 语句是同构的，即列的个数和数据类型必须一致。

eg1: 当月包在线时长为 20 小时，单位费用涨 5 分，为 40 小时涨 3 分，其他不变（用 union all 实现）

```
select base_duration,unit_cost+0.05 from cost where base_duration=20
union all
select base_duration,unit_cost+0.03 from cost where base_duration=40
union all
select base_duration,unit_cost from cost where base_duration not in(20,40)
or base_duration is null;效率低，换成 case when 较好
```

eg2: 列出客户姓名以及他的推荐人

```
select t2.real_name,t1.real_name from account t1 join account t2 or
t1.id=t2.recommender_id
union all
select real_name,'No recommender' from account where recommender_id is null;
```

eg3: sun280 和 sun-server 上的远程登录业务使用了哪些相同的资费标准

```
方式一: select name from cost where id in(
    select cost_id
    from host h join service s on h.id=s.unix_host and h.name='sun280'
    intersect
    select cost_id
    from host h join service s on h.id=s.unix_host and
h.name='sun-server');
```

```
方式二: select name from cost where id in(
    select cost_id
    from service s where s.unix_host in (
        select id from host where name='sun280')
    intersect
    select cost_id
    from host h join service s on h.id=s.unix_host and
h.name='sun-server');
```

eg4: 哪台 UNIX 服务器上没有开通远程登录业务

```
select id from host minus select unix_host from service;
```

14.4 子查询、连接、集合总结

- 1) 匹配问题: in、exists、inner join、intersect
- 2) 不匹配问题: not in、not exists、(outer join+where 匹配表非空列 is null)、minus
- 3) 匹配+不匹配问题: outer join、union、union all

二十、排名分页问题

LICHOO

15.1 什么是 rownum

rownum 是一个伪列，对查询返回的行编号即行号，由 1 开始依次递增。

◆ 注意事项：关键点：Oracle 的 rownum 数值是在获取每行之后才赋予的！

15.2 where rownum<=5 的执行过程

- 1) Oracle 获取第一个符合条件的 1 行，将它叫做第一行。
- 2) 有 5 行了吗？如果没有，Oracle 就再返回行，因为它要满足行号小于等于 5 的条件。如果到了 5 行，那么 Oracle 就不再返回行。
- 3) Oracle 获取下一行，并递增行号（从 2 到 3 再到 4 再到 5…）。
- 4) 返回到第 2 步。

15.3 where rownum=5 的执行过程

- 1) 由于 Oracle 没有获取到第一个符合条件的 1 行，即第一行。
- 2) 所以 Oracle 无法获取下一行，即无法从编号为 1 的第一行开始递增行号（从 2 到 3 再到 4 再到 5…）。
- 3) 最终结果为空。

eg1：找出帐务信息表的前三条记录

```
select rownum,id,real_name,create_date from account where rownum <= 3;
```

eg2：找出帐务信息表的第四条到第六条记录？

```
select rn,real_name from (select rownum rn,real_name from account where rownum<=6)  
where rn>=4;
```

◆ 注意事项：此时 rownum 必须有别名，否则结果将按照子查询后的表进行伪列查找，结果就为空了。

eg3：最晚开通系统的前三个客户？

```
select rownum,real_name,create_date from (select real_name,create_date from account  
order by create_date desc)  
where rownum<=3;
```

◆ 注意事项：要先排序后过滤，注意 where、order by 的执行顺序。

eg4：最晚开通系统的第四到第六名客户？

```
select rn,real_name,create_date from (select rownum rn,real_name,create_date  
from (select real_name,create_date from account  
order by create_date desc)  
where rownum<=6)  
where rn>=4;
```

二十一、约束 constraint

LICHOOL

面临的问题：某列必须有值而且唯一；某列的取值受到另一列的限制。

数据库提供的解决方法：限制无效的数据进入到表中；数据库层面的“安检”。

16.1 约束的类型

primary key、not null、unique key、references foreign key、check

16.2 primary key：主键约束

一张表只能有一个主键约束，其他约束没有这个限制！

1) 创建主键约束

①列级约束：

```
create table test(
    c1 number(2) constraint test_c1_pk primary key,
    c2 number);
```

②表级约束：

```
create table test(
    c1 number(2),
    c2 number,
    constraint test_c1_pk primary key(c1));
```

2) 主键约束解决的问题是：不允许表中有 null 记录，不允许表中有重复的记录

3) 增加删除操作

①增加一列：alter table test add(同建表时的列定义方法，表级约束语法)；

②删除一列：alter table test drop(列名)；

③删除约束：alter table test drop constraint 约束名；

④删除主键约束：alter table test drop primary key；

⑤增加主键约束：alter table test add constraint 约束名 primary key(列名)；

◆ 注意事项：

❖ 约束一定要有名字，自己不写，系统也会加上“SYS_C+一堆数字”。

❖ 约束名是在同一用户下的，因此不能有相同的名称，建议“表名_列名_约束类型”。

16.3 not null：非空约束

不允许将该列的值置为空，只有列级约束形式。

1) 创建非空约束

```
create table test(
    c1 number constraint test_c1_pk primary key,
    c2 number not null);
```

2) 修改非空约束：

alter table test modify(c1 null);not null->null

alter table test modify(c1 default 1 not null);null->not null 默认、缺省值为 1

16.4 unique key：唯一键约束

一张表可有多个唯一键约束！

1) 创建唯一键约束

①列级约束:

```
create table test(
    c1 number constraint test_c1_pk primary key,
    c2 number constraint test_c2_uk unique,
    c3 number constraint test_c3_uk unique);注意, unique 后没有 key 单词
```

②表级约束:

```
create table test(
    c1 number constraint test_c1_pk primary key,
    c2 number ,
    c3 number ,
    constraint test_c3_uk unique(c2,c3));
```

◆ 注意事项:

- ❖ 唯一性约束和主键约束的共同点是都能保证列值的唯一性,都可以被外键列引用。
- ❖ 联合唯一键约束与在两列上分别定义唯一键约束不一样。

2) 唯一键的列值不允许重复。

3) 唯一键上的任意一列的取值允许为空, 并且可以是多个 null 值。

4) primary key = unique key + not null, 表示主键列要求非空, 而唯一键列允许为 null, 并且可以是多个 null 值。

eg: 用 DDL 语句创建一张表, 有三列, 每列的值都要求唯一且非空。

```
create table test( c1 number constraint test_c1_pk primary key,
                    c2 number not null constraint test_c2_uk unique,
                    c3 number not null constraint test_c3_uk unique);
```

16.5 references foreign key: 外键约束

1) 创建外键约束

①列级约束:

```
create table child(
    c1 number(2) constraint child_c1_pk primary key,
    c2 number(3) constraint child_c2_fk references parent(c1));注意 references 有 “s”
```

②表级约束 1:

```
create table child(
    c1 number(2) constraint child_c1_pk primary key,
    c2 number(3) constraint child_c2_fk foreign key(c2) references parent(c1));
表级定义外键约束时用 foreign key
```

③表级约束 2:

```
create table child1(
    c1 number(2) constraint child1_c1_pk primary key,
    c2 number(3) constraint child1_c2_fk references parent(c1) on delete cascade);
```

④表级约束 3:

```
create table child2(
    c1 number(2) constraint child1_c1_pk primary key,
    c2 number(3) constraint child1_c2_fk references parent(c1) on delete set null);
```

2) 外键约束的作用：外键约束是用来解决一对多关系的。

3) 外键约束下父子表的联系：

①建立外键约束：首先先创建父表，并且引用的列必须唯一键或主键，至于被引用的列本身是否为 null，外键约束并没有要求。然后子表再去创建外键约束。

②插入外键约束列中的值：首先先向父表中被引用列插入值，然后再向子表中外键约束列插入值。

③删除外键约束列中的值：首先先删除子表中外键约束列的值，然后再删除父表中被引用列的值。

④删除表：首先先删除子表，然后再删除父表

◆ 注意事项：

- ❖ 通过外键 FK 可以与同一张表的主键 PK 或唯一键 UK 建立引用关系，也可以与不同表的主键 PK 或唯一键 UK 建立引用关系。
- ❖ 外键的取值必须匹配父表中已有的值或空值。

4) 删除外键约束：alter table child drop constraint child_c2_fk;

5) cascade constraints：级联约束

通过级联约束删除表：drop table parent cascade constraints purge; 通过级联约束删表不用管父子表的约束，直接删表。否则要先删子表，再删父表。

◆ 注意事项：

- ❖ 此语句 constraints 有“s”!
- ❖ 等价于执行两个动作：alter table child (子表) drop constraint child_c2_fk (子表中的约束)； drop table parent (父表) purge;

6) on delete cascade：级联删除

即在删除父表记录的时候，系统会将子表的记录先删除，再删除父表的记录。

7) on delete set null：将子表中的记录某列置空，再删除父表中的记录。

```
insert into parent values (1);
insert into child2 values (1,1);
delete from parent where c1 = 1;
```

相当于以下两条语句：

```
update child2 set c2 = null where c2 = 1;
delete from parent where c1 = 1;
```

8) 外键约束关键字：

①foreign key：用表级约束定义外键时使用该关键字。

②references：表示引用父表中的某列。

③on delete cascade：级联删除，删除父表的记录前，先删除子表里的相关记录。

④on delete set null：删除父表的记录前，先将子表中的外键列的相关值置空。

9) update 语句：更新表中已经存在的记录，即修改记录的某列的值。

```
update tabname set colname=value[,colname=value] [where condition]
```

delete 语句：删除已经存在的记录。

```
delete [from] tabname [where condition];
```

◆ 注意事项：

- ❖ integrity constraint：看到这个就是违反了外键约束。
- ❖ 被引用的记录应先放入表中。

16.6 check：检查约束

1) 创建检查约束

①列级约束：

```
create table test(
    c1 number(3) constraint test_c1_pk primary key,
    c2 number(3) constraint test_c2_ck check ( c2 > 100 ));
```

②表级约束：

```
create table test(
    c1 number(3) constraint test_c1_pk primary key,
    c2 number(2),
    c3 number(2),constraint test_c2_ck check (( c2 + c3 )> 100 ));
```

2) 定义条件表达式，每个列值必须满足该条件。

3) 以下表达式不允许

①伪列： curval、 nextval、 level、 rownum

②函数： sysdate、 uid、 user、 userenv

③引用其他记录的其他值

4) 在 create table 时定义约束，表已经存在，用 alter table 追加约束。

二十二、事务

LICHOO

17.1 transaction

交易，事务；（一笔）交易，（一项）事务。

client 端建立连接

```
connect jsd1304/jsd1304(sqlplus/sql developer/jdbc)
```

DDL 操作：

```
create table (column 列 datatype 数据类型 constraint 约束);
```

alter table 对列、约束等的操作

```
drop table cascade constraint purge;
```

DML 操作：insert、update、delete

TCL 操作：commit、rollback，数据才真正入库

table:account

```
column id,balance 余额    row: A  B
```

```
update account set balance = balance-1000 where id='A';
```

```
update account set balance = balance+1000 where id='B';
```

交易：包含 2 个 update，一笔交易应看作一个原子操作：要做一起作，要么都不作

帐平，不是单条 DML，而是一组 DML+committ、rollback (transaction)

17.2 定义

事务是由一组 DML 语句和 commit/rollback(TCL)组成，是改变数据库数据的最小逻辑单元。

- 1) 如果是 commit，表示数据入库；如果是 rollback，表示取消所有的 DML 操作。
- 2) 事务的结束：committ/rollback，DDL 语句自动提交，DML 一定要有显式 commit。
用程序传数据入库也要传一个事务！
3) 事务的开始：上一个事务的结束就是下一个事务的开始。
◆ 注意事项：我们传送的是一个事务，而不是简单的 DML 语句了！数据库中也都是一个个事务。切记！！

17.3 事务的特性：ACID

- 1) 原子性 (atomicity)：一个事务或者完全发生，或者完全不发生。由 DML 和 TCL 共同完成的!!
- 2) 一致性 (consistency)：事务把数据库从一个一致状态转变到另一个状态。
- 3) 隔离性 (isolation)：在事务提交之前，其他事务觉察不到事务的影响。
- 4) 持久性 (durability)：一旦事务提交，它是永久的。

17.4 事务的隔离级别

数据库应用程序中最常用的隔离级别。

Read committed：一个事务只可以读取在事务开始之前提交的数据和本事务正在修改的数据。

17.5 数据库开发的关键挑战

在开发多用户、数据库驱动的应用程序中，关键性的挑战之一是要使并行的访问量达到

最大化，同时还要保证每一个用户（会话）可以以一致的方式读取并修改数据。

- 1) 锁 (lock) 机制：用来管理对一个共享资源的并行访问
- 2) 多版本一致读：
 - ① 非阻塞查询：写不阻塞读，读不阻塞写
 - ② 一致读查询：在某一时刻查询产生一致结果

17.6 锁的概念

- 1) 排他锁 (X 锁)：如果一个对象上加了 X 锁，在这个锁被采用后，直到 commit 或 rollback 释放它之前，该对象上不能施加任何其他类型的锁。
- 2) 共享锁 (S 锁)：如果一个对象被加上了 S 锁，该对象上可以加其他类型的 S 锁，但是，在该锁释放之前，该对象不能被加任何其他类型的 X 锁。

17.7 Oracle 的锁机制

为确保并发用户能正确使用与管理共享资源，如表中的记录，Oracle 引进锁机制。

- 1) 对表的数据进行写操作时，系统会自动加两类，共 3 种锁。
 - ① DML 锁：用于保护数据的完整性，会加以下两种锁（对 DML 操作而言的，用排队机制 wait）。
 - TX 锁：即事务锁（行级锁），类型为 X 锁；操作哪行记录就加排他锁
 - TM 锁：即意向锁（表级锁），属于一种 S 锁；操作哪个表就加共享锁
 - ◆ 注意事项：
 - ❖ 不提交，锁不会被释放！就会把别人堵塞住！
 - ❖ 不提交，锁不会被释放！别人也看不到你对数据的操作。
 - ❖ select 语句是读操作，没有上锁。
- ② DDL 锁：用于保护数据库对象的结构（例如表、索引的结构定义）（正在写操作时，不可修改表的结果，否则直接报错 error）。

X 类型的 DDL 锁：这些锁定防止其他的会话，自己获得 DDL 锁定或 TM(DML) 锁定。这意味着可以在 DDL 其间查询一个表，但不可以以任何方式进行修改。

 - ◆ 注意事项：出现 wait 和 no wait（即 DDL 报错）都是因为不能再加 X 锁导致的。

17.8 事务不提交的后果

- 1) 其他事务看不见它的操作结果。
- 2) 表和行上的锁不释放，会阻塞其他事务的操作。
- 3) 它所操作的数据可以恢复到之前的状态。
- 4) 占用的回滚端资源不释放，rollback segment/undo segment 会滚段（公共空间）。

17.9 回滚事务 rollback

- 1) 数据的改变就像从未发生过一样
- 2) 插入的数据没有了，更新前和删除前的数据都恢复出来。
- 3) 锁被释放。

17.10 保留点 savepoint

用 savepoint 在当前事务里创建一个保留点，用 rollback to savepoint 命令将事务回滚到标记点。

```
SQL>insert.....;      SQL>update.....;      SQL>savepoint update_done;
      savepoint created.
```

SQL>insert.....;

SQL>rollback to update _done;

rollback complete.

LICHCO

二十三、数据库对象：视图 view

LICHOO

18.1 带子查询的 create table

```
create table tablename[column(column…)] as subquery;
```

- 1) 根据子查询语句创建表并插入数据（根据已有的表创建新表）。
- 2) 表结构由子查询的 select 语句决定，create table 指定的列的数量要跟 select 语句指定的列的数量一致。
- 3) create table 定义列只能定义列名、缺省值、完整性约束，不能定义数据类型。
- 4) 约束不能被复制过来，但非空约束不需要定义可以直接复制过来。

eg1:20 机器上的业务信息

```
create table service_20 as select * from service where unix_host='192.168.0.20';
```

eg2: 创建一张表 account_90, 表结构与 account 一致，没有数据

```
create table account_90 as select * from account  
where 1 = 2 ;
```

- ◆ 注意事项：where 1=2;是通用的，所以记录不符合条件；若子查询的返回记录数为 0，新建的表就只有结构。1 = 2 是永假式，任何表都不会返回记录。where 1=1;所有记录符合条件

18.2 带子查询的 insert

```
insert into new_tab(colname1,colname2,…colnamen)  
select colname1,colname2,…colnamen  
from old_tab  
where condition;
```

- 1) 根据子查询语句向表中插入数据。
 - 2) insert 指定的列的数量要跟 select 语句指定的列的数量一致。
 - 3) 一次可以插入多条记录，不能用 values 子句。
- ◆ 注意事项：若插入多个列，则需要把所有的非空列都选出，否则报错，因为非空约束直接复制过来了。

eg: account_90 表中包含所有的 90 后客户

```
insert into account_90_chang  
select * from account  
where to_char(birthdate,'yyyy') between 1990 and 1999;
```

18.3 定义缺省值： default

…colname date default sysdate,…

- 1) 缺省值的数据类型必须匹配列的数据类型。
 - 2) 有效的缺省值为文字值，表达式、sql 函数： sysdate、 user 等。
 - 3) 无效的缺省值为另一个列的列名或伪列。
 - 4) default 可以用于 insert 语句、 update 语句。
- ◆ 注意事项： insert 语句、 update 语句都可写非关联自查询。

eg1: 案例 1

```
drop table test purge;  
create table test( c1 number default 1,c2 number);
```

```
insert into test (c2) values (2);      insert into test values (default,3);
```

eg2: 案例 2

```
insert into test values (4,4);
```

```
update test set c1 = default where c1=4;把 c1 列等于 4 的都换成默认值
```

18.4 视图 view

- 1) 视图在数据库中不存储数据值，即不占空间。
- 2) 只在系统表中存储对视图的定义。
- 3) 视图实际就是一条 select 语句。
- 4) 类似 windows 中的快捷方式。
◆ 注意事项：对象类型都按大写存储的，所以查找也写大写。

eg1: 创建 view

```
create or replace view test_v1 as select * from test where c1=2
```

eg2: 查找 view test_v1 是如何定义的，找对应的 select 语句

```
select view_name,text from user_views
```

```
where view_name = 'TEST_V1';
```

eg3: 查找 view test_v1 当前的状态，若为 invalid 则源表出问题了

```
select object_name,object_type,status from user_objects
```

```
where object_name = 'TEST_V1';
```

eg4: 当视图无效时

```
alter view test_v1 compile;
```

//当视图无效时，尝试先编译视图，若不能编译，则源表不存在，要创建表。

```
create table test(c1 number,c2 number); //此时系统不会自动编译。
```

```
select * from test_v1; //当从视图中查询时，系统会做 alter view test_v1 compile 即编译操作，此时视图就有效了。
```

18.5 视图的应用场景

- 1) 简化操作，屏蔽了复杂的 SQL 语句，直接对视图操作。
- 2) 控制权限，只允许查询一张表中的部分数据。解决办法：对其创建视图，授予用户读视图的权限，而非读表的权限。

eg: 允许授权 account 表部分数据给 jsd1304

```
create or replace view account_1304
```

```
as
```

```
select * from account
```

```
where 条件表达式
```

```
grant select on account_1304 to jsd1304;
```

- 3) 通过视图将多张表 union all 成一张逻辑表，作为单独一个数据库对象，实现表的超集。

eg: 数据库中有分区表

table heap table 堆表，我们当前用的这些无序的一般表

table partition table 分区表

```
create table haidian
```

```
create table haidian1
```

```
create table xicheng
```

```

create table changping      .....
create or replace view beijing as select * from haidian
union all
select * from haidian1
union all
select * from xicheng
union all
select * from changping

```

eg: 每个客户选择了哪些资费标准

方式一:

```

create or replace view cost_account_service
as
select a.real_name,s.unix_host,c.name
from account a join service s on a.id=s.account_id
join cost c on s.cost_id=c.id;

```

方式二:

```

create or replace view cost_account_service
as
select a.real_name,s.unix_host,c.name
from account a left join service s on a.id=s.account_id
left join cost c on s.cost_id=c.id;

```

方式三:

```

create or replace view cost_account_service
as
select a.real_name,t.unix_host,t.name
from account a left join (select s.account_id,s.unix_host,c.name
                           from service s join cost c
                           on s.cost_id = c.id) t
on a.id = t.account_id;

```

18.6 视图的分类

- 1) 简单视图: 基于单张表并且不包含函数或表达式的视图, 在该视图上可以执行 DML 语句 (即可执行增、删、改操作)。
- 2) 复杂视图: 包含函数、表达式或者分组数据的视图, 在该视图上执行 DML 语句时必须要符合特定条件。
在定义复杂视图时必须为函数或表达式定义别名。
- 3) 连接视图: 基于多个表建立的视图, 一般来说不会在该视图上执行 insert、update、delete 操作 (即不可进行 DML 操作)。

18.7 视图的维护

- 1) 视图中的 with check option 约束

```

create or replace view test_ck as select * from test
where c1=1 with check option;

```

通过“视图”插入数据时, 必须符合条件才能插入 (避免不符合逻辑的问题: 能插

入其他值，但通过视图查询时看不到其他值的情况，即能操作却不能看)。

2) 视图中的 with read only 约束

```
create or replace view test_ck as select * from test  
where c1=1 with read only;
```

只读视图，只能看不能操作，报错提示这些是虚拟列。

3) 视图的 DDL 语句

```
create or replace view view_name  
alter view  
drop view
```

二十四、数据库对象：索引 index

LICHOO

19.1 创建 index

```
create index index_name on table_name(columnname);
```

eg: 创建 service 表中的 account_id 索引

```
create index service_account_id_idx on service(account_id);
```

19.2 扫描表的方式

1) 全表扫描 FTS(Full Table Scan)

高水位线 HWM(High Water Mark): 曾经插入数据的最远块 (数据存储的最小单元数据块)。数据块: 口口口回回 | 口口, 1、2 被删掉后, 高水位线不动, 所以叫曾经……

delete from tablename 中 delete 也同理, 清除数据但不释放空间, 高水位线不动, count(*) 花很长时间, 但结果为 0; 同时数据也要写入 rollback 回滚段, 因此时间又长了, 但好处是不提交数据是可恢复的; 若等了半天, 出现回滚段空间不足, 则又会把数据返回表中 --! 所以, delete 不适合删除大表的所有数据!

truncate table tablename; DDL 操作, 空间释放, 高水位线前移, 时间短, 不写回滚段, 数据是不可恢复的 (删除表中的所有行, 但表结构及其列、约束、索引等保持不变)。

eg: 将扫描高水位线以下的所有数据块

```
select real_name from account where id=1010;不建立索引, 则会扫描全表
```

2) 通过 rowid (伪列) 来扫描数据

rowid: 标识一条记录的物理位置 (唯一标识), rowid 的数据类型就是 rowid, 会隐式把字符转成 rowid 类型。

rowid 包含如下信息:

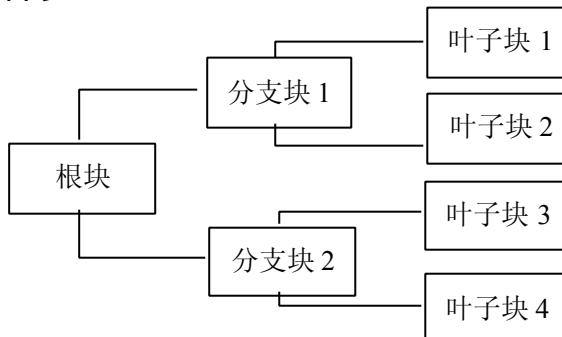
①该记录属于哪张表的 (哪个数据库对象): data_object_id

②该记录在数据文件的第几个数据块里: block_id

③该记录在数据块里是第几条记录: row_id

而索引则会记录: key, rowid 键值对, 即为 index entry 索引项。

19.3 索引的结构



B*tree 索引由根块(root block)、分支块(branch block)、叶子块(leaf block)组成。

1) 根块下面是分支块, 用于导航结构, 包含了索引列范围和另一非根块(可以是分支块或叶子块)的地址。

2) 最底层为叶子块, 包含索引项(index entry), 索引由 key 值(被索引列的值)和该列所在行的 rowid 组成。

3) 叶子块实际上是双向链表的表。一旦找到叶子块的“开始”点(一旦找到第一个值),

对值进行顺序扫描（索引范围扫描）是很容易的。不必再做结构导航，只要通过叶子块转发就行。最主要的就是索引对数据进行了排序。

19.4 为什么要使用索引

- 1) Oracle server 通过 rowid 快速定位要找的行。
- 2) 通过 rowid 定位数据能有效降低读取数据块（data block）的数量。
- 3) 索引的使用和维护是自动的，一般情况下不需要用户干预。

19.5 哪些列适合建索引

- 1) 经常出现在 where 子句的列。
- 2) 经常用于表连接的列。
 - ◆ 注意事项：A、B 两表连接，B 表有引用 A 表的外键，则从外键约束角度看 A 表为父表，B 表为子表；一般父表数据量小，子表数据量大，所以从表连接角度看 A 表应作为驱动表，B 表应作匹配表，把匹配表作索引，则匹配时不再进行全表扫描，效率将提高。
- 3) 该列是高基数数据列（高基数数据列是指有很多不同的值）。
- 4) 该列包含许多 null 值。
 - ◆ 注意事项：where is null 一定是全表扫描，因为索引不记录 null 值。
- 5) 表很大，查询的结果集小。
- 6) 主键（PK）列、唯一键（UK）列。
- 7) 外键（FK）列。
- 8) 经常需要排序（order by）和分组（group by）的列。
 - ◆ 注意事项：
 - ❖ 索引不是万能的，结果集和源表数据差不多时，使用索引就不好了，因为系统还要去读索引。
 - ❖ 但没有索引是万万不能的。

19.6 索引的类型

- 1) 唯一性索引（unique）：等价于唯一性约束，唯一性约束用唯一性索引实现的。
- 2) 非唯一性索引：用于提高查询效率

eg: 创建唯一性索引

```
create unique index test_c1_uniidx on test(c1);
insert into test values (1);
insert into test values (1);
ERROR at line 1:
ORA-00001: unique constraint (JSD1302.TEST_C1_UNIIDX) violated
注意: 唯一性约束的名字是唯一性索引的名字
结论: 唯一性约束是通过唯一性索引实现的, 二者是等价的。
```

- 3) 单列索引：索引建在一列上
- 4) 联合索引：索引建在多列上

eg: 创建联合索引

```
create unique index srt_cour_pkid on stu_cour(sid,cid);联合主键索引: 则
index entry 存储的为 1011, 10, rowid
where c1=2 and c2=1;联合列索引: 则 index entry 存储的为 2, 1, rowid
```

19.7 哪些写法会导致索引用不了

- 1) 函数导致索引用不了: where upper(colname)='carmen'
- 2) 表达式导致索引用不了: where colname*12=12000
- 3) 部分隐式数据类型导致索引用不了: where colname=2(c1 为 varchar2 类型)
 - ◆ 注意事项: 经过计算的结果, 在索引里是找不到的。只能建函数或表达式的索引。
- 4) like 和 substr

where colname like 'ca%';只要通配符不在前面, 就可以用索引
where substr(colname,1,2)='ca';不可用索引

- 5) 查询所有的 null 值: where colname is null
- 6) 否定形式: not in、<>
 - ◆ 注意事项: 把 not in 转成 not exist 可以用索引。

eg: 创建函数索引

```
create index test_c2_funidx on test(round(c2));
```

二十五、数据库对象：序列号 sequence

LICHOO

面临问题：主键约束和唯一键约束要求列中每个值都必须是唯一的
程序员怎样获得唯一值：

使用 Oracle 提供的数据库对象 sequence：序列号

程序员自己写代码实现

20.1 什么是 sequence

Oracle 提供的数据库对象。

- 1) 为了解决主键值和唯一键值的唯一性（即解决如果保证插入的数据是唯一的）。
- 2) 按照预定义的模式自动生成整数的一种机制，保证数字的自动增长。

20.2 创建 sequence

```
create sequence seq_name
[increment by 1 | integer]
[start with integer]
[maxvalue integer | nomaxvalue]
[minvalue integer | nomaxvalue]
[cycle | nocycle]      (有缺省值)
[cache 20 (缺省值) | integer | no cache]
```

◆ 注意事项：integer 一定为数值类型

eg1：创建序列号，创建 sequence 最简单的方式：create sequence s1

```
create sequence s1
start with 1
increment by 1
maxvalue 5
```

◆ 注意事项：表、视图、索引、序列号都不能重名！

eg2：若找到重名的对象

```
select object_type from user_object where object_name='s1';找出重名的s1是那个对象中的
drop sequence s1;发现是 sequence 中的 s1，则删除 sequence s1
```

20.3 缺省是 nocycle (不循环)

eg1：不循环情况

```
create sequence s1 start with 1 maxvalue 5;
select s1.nextval from dual;
```

select 语句连续执行 6 遍，最后一次：报错

```
ORA-08004:sequence S110.NEXTVAL exceeds MAXVALUE and cannot be instantiated
```

eg2：循环情况

```
create sequence s2 start with 1 maxvalue 5 cycle cache 4
```

select 语句连续执行 6 遍，最后一次：第 6 次重新从 1 开始

20.4 缺省 cache 20

当没写 cycle 时，即为缺省值 nocycle 时，由于为 cache 20，所以一次性取出 20 个数，存入到内存中，以后的用户直接在内存中取数，若内存中没有，则再次读取序列号。

◆ 注意事项：

- ❖ 当 cycle (循环) 时, cache 的值要比最大值小才能成功建立!
- ❖ 当 nocycle (不循环) 时, 大于最大值会报错。

eg1: 获取当前序列号的值

```
select s1.curval from dual;
```

eg2: 从 user_sequences 获取所有序列号信息

```
select sequence_name,cache_size,last_number from user_sequences;
```

二十六、其他注意事项

LICHOO

21.1 删除表，删除列，删除列中的值

- 1) drop table 表名 purge;删除表
- 2) alter table test drop(列名);删除一列
- 3) delete from 表名: 删除表中所有值 (列保留); 若加上 where 列名=value 则删除某列中的值
- 4) alter table test add(同建表时的列定义方法);增加一列
- 5) truncate table tabname;删除表中的所有行, 但表结构及其列、约束、索引等保持不变

21.2 多对多关系的实现

都需要增加一个中间表 (三张表)

stu	course	stu-cour
sid	cid	sid fk-stu(sid)
name	name	cid fk-course(cid) pk(sid,cid)

21.3 一对多 (两张表)

p 表, c 表, 对 pid 设置(fk pk)

21.4 一对一

husband wife

一张表: h info w info

两张表: husband id pk fk->w(id) wife id pk

另一种案例: man id pk fk wid->woman(id) uk woman id pk

21.5 数据库对象

table、view、index、sequence

12.6 缺省 (默认) 总结:

- 1) oracle 数据库服务缺省端口为 1521。
- 2) 日期类型缺省格式为 DD-MON-RR。
- 3) round(): 四舍五入函数, 缺省转成数字; 也可对日期。
- 4) 数据类型转换: 缺省将字符转化为 number (即会调用 to_number()函数, 要求字符串必须是数字字符)。
- 5) 组函数缺省处理所有的非空值。
- 6) 定义缺省值: c1 default 1
- 7) sequence 缺省 nocycle 和 cache 20
- 8) 不写列别名的话, 默认都是转成大写。
- 9) ASC—升序, 可以省略, 缺省。
- 10) char 可以不定义长度, 默认为 1, 按定义长度存, 最大长度 2000 字节, 操作更快。

3 PL/SQL 学习笔记

LICHOO

二十七、PL/SQL 简介

1.1 什么是 PL/SQL

PL/SQL (Procedural Language/SQL) 是 Oracle 在标准 SQL 的基础上增加了过程化处理，把 DML 和 select 语句组织在 PL/SQL 代码的过程性单元中，通过逻辑判断、循环等操作，实现复杂的功能或者计算的程序语言。

扩展：变量和类型、控制结构、过程与函数。

- ◆ 注意事项：
 - ❖ java 中是写方法，把复杂的业务逻辑写入方法中，再调用方法。
 - ❖ PL/SQL 是写过程、函数，把复杂的业务逻辑写入过程、函数中，再调用它们。

1.2 PL/SQL 程序结构

PL/SQL 块：包含三部分，声明部分：declare；执行部分：begin；异常处理：exception
eg1：语法

```
declare
    v_AccountID number(5):=1001;
    v_RealName  varchar(20);
begin
    select real_name info v_RealName from account where id=v_AccountID;
exception
    when on_data_found then
        insert info Fee_Log(desrc) values('Account 1001 dece not exit!')
        commin;
end;-- (不需要理解代码的具体含义!)
```

- ◆ 注意事项：最简写的方式为 begin end;

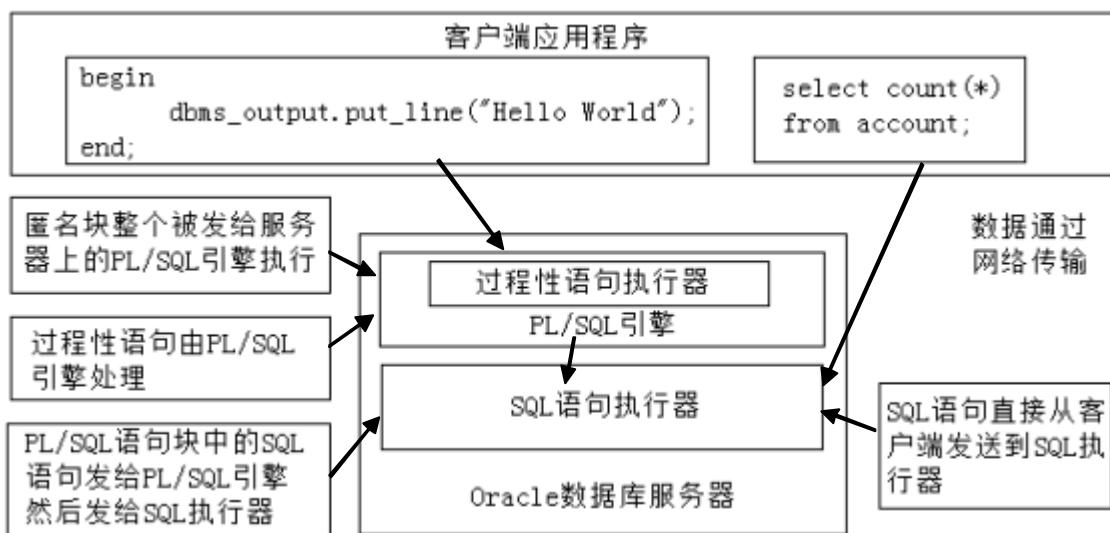
eg2：打印 Hello World

```
begin
    dbms_output.put_line('Hello World');
end;
```

- ◆ 注意事项：
 - ❖ dbms_output 是系统提供的包 package，包含多个过程、函数。其中的过程 put_line 实现的是输出功能，只有一个参数，只能为“字符类型”（日期和数值也可，系统自动转换！布尔类型不行！但 java 中可以），用于接收需输出的字符串。在 sql 工作表子窗口中可以调用存储过程。
 - ❖ 想要在屏幕上输出需要写：set serveroutput on（在 begin 上面写）。
 - ❖ 如何调用过程：begin 包名.过程名();所有过程都是没有返回值的，即 java 中的 void。

1.3 PL/SQL 运行过程

如下图所示：



1.4 注释

增加可阅读性，使程序更容易理解。编译时将被忽略。

- 1) 单行注释：由两个连字符 “--” 开始，到行尾都是注释。
- 2) 多行注释：由 “/*” 开头，由 “*/” 结尾。

二十八、变量与数据类型

LICHOO

2.1 数据类型

标量类型：数字型、字符型、日期型、布尔型；可以直接用的。

复合类型：record、associative array、nested table、varray；需要自己去定义的。

2.2 标量类型

1) 数字类型：①number ②number 的子类型 dec(38)、float(38)、real(18)…

③binary_integer（只能在 PL/SQL 中用），按 10 进制赋值，但存的时候会换成 2 进制存。优势是计算快。

2) 字符类型：①varchar2、varchar（长度：1~32767）②string（只能在 PL/SQL 中用，长度：1~32767）③char（长度：1~32767）④long

3) 日期类型：date

4) 布尔类型：boolean

①用于存储逻辑值 true, false, null（java 中只有 true, false）。

②不能向数据库中插入 boolean 类型的数据。

③不能将列值保存到 boolean 变量中。

④只能对 boolean 变量执行逻辑操作。

2.3 变量声明

1) 语法：var_name type [constant][not null][:=value];

◆ 注意事项：PL/SQL 规定没有初始化的变量为 null。

2) 直接定义类型

```
declare
    v_n1 number :=1;--赋初值要有“冒号”，PL/SQL 中赋值和等号是区分开的
    v_c1 varchar2(10);
    v_d1 date :=sysdate;--sysdate 是函数，有返回值
begin
```

◆ 注意事项：

❖ PL/SQL 中没返回值的叫过程；有返回值的叫函数，且必须有指向（要么打印出，要么赋值给变量）。所以，过程和函数的调用是不一样的。

❖ java 中的方法一种没有返回值，一种是有返回值；有返回值的如果没返回，则会丢弃。但 PL/SQL 不行！

3) %type 方式：变量具有与数据库的表中的某列或其他变量相同的类型。

eg1：三种声明方式

```
declare v_RealName varchar2(20);
declare v_RealName account.real_name%type;
declare v_TempVar number(7,3) not null :=12.3;
    v_AnotherVar v_TempVar%type :=12.3;
```

eg2：定义变量，打印系统时间，并输出它们的值

```
set serveroutput on
declare
    v_d1 date :=sysdate;
    v_c1 varchar2(20);
begin
    v_c1 :=to_char(v_d1,'yyyy mm dd hh24:mi:ss');
    dbms_output.put_line('Current date is ' || v_c1);
    --put_line 只有一个参数，所以必须拼接！
end;
```

4) 复合类型

record 类型、集合类型。详细内容见后面的第五、六章！

二十九、流程控制语句

LICHOO

条件语句：if、case； 循环语句：loop、while、for

3.1 条件语句

1) if 语句

方式一：

```
if then
    statement;
end if; //最简写的方式
```

方式二：

```
if then
    statement;
else
    statement1;
end if;
```

方式三：

```
if boolean_expr then
    statement;
elsif boolean_expr then
    statement1;
else
    statement2;
end if;
```

◆ 注意事项：else if 在 PL/SQL 中的写法是 elsif！

eg：输出布尔类型的变量的值

```
declare
    v_b1 boolean :=false;
begin
    if v_b1 then
        dbms_output.put_line('true');
    elsif v_b1 is null then
        dbms_output.put_line('null');
    else
        dbms_output.put_line('false');
    end if;
end; --注意：put_line 中的参数只能为字符类型，但日期和数值会隐式转换，布尔类型不行！
```

2) case 语句

```
case when then
      when then
else
end;
```

3.2 循环语句

1) loop 循环（无条件进入）

语法:

```
loop
    statement1;
    statement2;
    exit when<condition>
end loop;
```

◆ 注意事项:

- ❖ exit when<condition>子句是必须的，否则循环将无法停止。
- ❖ when 后面直接写条件，没有尖括号或圆括号。

eg1: 循环输出数字 1~10

```
declare v_index binary_integer :=1;
begin
    loop
        dbms_output.put_line(v_index);
        v_index := v_index+1;
        exit when v_index>10;
    end loop;
end;
```

eg2: 循环插入 10 条记录（静态）

```
declare v_index binary_integer :=1;
begin
    loop
        insert into test_chang values(v_index);
        v_index := v_index+1;
        exit when v_index>10;
    end loop;
end;
```

eg3: 循环输出语句 insert into test_chang values('i'), i 为字符。

```
declare v_index binary_integer :=1;
begin
    loop
        dbms_output.put_line('insert into test_chang values'||v_index||');
        v_index := v_index+1;
        exit when v_index>10;
    end loop;
end;
```

2) while 循环

语法:

```
while<boolean expr>loop
    statement1;    statement2;
end loop;
```

①循环语句执行的顺序是先判断<boolean expr>的真假，如果为 true 则循环执行，否则退出循环。

②在 while 循环语句中仍然可以使用 exit 或 exit when 子句。

◆ 注意事项：while 后面直接写条件，没有尖括号或圆括号。

eg1: 循环输出数字 1~10

```
declare v_index binary_integer :=1;
begin
    while v_index <=10 loop
        dbms_output.put_line(v_index);
        v_index := v_index+1;
    end loop;
end;
```

eg2: 循环插入 10 条记录（动态）

```
declare v_index binary_integer :=1;
begin
    while v_index <=10 loop
        execute immediate 'insert into test_chang values'||v_index||';
        v_index := v_index+1;
    end loop;
end;
```

3) for 循环

语法：

```
for 循环计数器 in 下限 .. 上限 loop
    statement1;
    statement2;
end loop;
```

①循环计数器是一个变量，这个变量不需要声明（和 java 不一样）。它的作用域仅是在循环中。

②每循环一次，循环变量自动加 1；使用关键字 reverse，循环变量自动减 1。

③可以使用 exit 或者 exit when 子句退出循环。

◆ 注意事项：

❖ 跟在 in、in reverse 后面的数字必须是从小到大的顺序，但不一定是整数，可以是能够转换成整数的变量或表达式。

❖ 不能用 null 作上限或下限下标！

eg1: 循环输出数字 1~10

```
begin
    for i in 1..10 loop
        dbms_output.put_line(i);
    end loop;
end;
```

eg2: 使用 in reverse 循环输出数字 1~10

```
begin
    for i in reverse 1..10 loop
        dbms_output.put_line(i);
```

end loop;

end;

eg3: 循环插入 10 条记录 (静态)

begin

for i in 1..10 loop

insert into test_chang values(i);

end loop;

end;

LICHO0

三十、PL/SQL 中的 SQL

LICHOO

4.1 PL/SQL 中的 SQL 分类

1) 静态 SQL

在 PL/SQL 块中使用的 SQL 语句在编译时是明确的，执行的是确定对象，即 SQL 语句是在 PL/SQL 编译阶段编译的。效率高。能用静态就不用动态！

2) 动态 SQL

在 PL/SQL 块编译时 SQL 语句是不确定的，如根据用户输入的参数的不同而执行不同的操作。编译程序对动态语句部分不进行处理，只是在程序运行时动态的创建语句、对语句进行语法分析并执行该语句。效率低。

4.2 DML (insert, update, delete) 和 TCL (commit, rollback)

1) 它们可以直接在 PL/SQL 中使用标准的 SQL 语句。

2) 语法：

```
begin
    insert into host(id) values('10.0.0.11');
    commit;
end;
```

4.3 DDL

1) 不能原封不动的像 DML 和 TCL 那样直接写。

2) 所有的 DDL 语句要在 PL/SQL 中操作，必须用如下的语法形式（转成字符串），即本地动态 SQL 是使用 execute immediate 语句来实现的。

eg1：可执行

```
begin
    execute immediate 'create table test_chang(c1 number)';
end;--本地动态 SQL 执行 DDL 语句
```

eg2：不可执行

```
begin
    execute immediate 'create table test_chang(c1 number)';
    insert into test values(1);
    commit;
end;--报错！是编译错误，说 test_chang 不存在！说明表还没有创建，只是在编译期编译。
```

eg3：可执行

```
begin
    execute immediate 'create table test_chang(c1 number)';
    execute immediate 'insert into test_chang values(1)';
    commit;
end;--编译，执行都通过！使用动态 SQL
```

- ◆ 注意事项：在一段 PL/SQL 程序中，先 create 再 insert，必须用 execute immediate（静态 sql 和动态 sql）

eg4: 用循环向 test_chang 表中插入 10 条记录。(从 1 到 10)

```
begin
    execute immediate 'create table test_chang(c1 number)';
    for i in 1..10 loop
        execute immediate 'insert into test_chang values'||i';
    end loop;
    commit;
end;
```

三十一、PL/SQL 中的 select

LICHOO

5.1 select 语句的实现

根据 select 语句返回的记录数，将 select 语句的实现分为两类：

- 1) 当且仅当只返回“一条”记录：用 select...into...语句实现（将结果放入到变量中）。
- 2) 返回“0条或多条”记录：用 cursor 实现。

◆ 注意事项：select into 确认 select 只返回一条记录，才不报错，否则加异常处理（没有返回记录或返回记录太多）。

eg: 提供客户 ID，打印客户姓名和年龄

```
declare
    v_realname account.real_name%type;
    v_age number(3);
begin
    select real_name,round((sysdate-birthdate)/365)//减出的是天数
        into v_realname,v_age
    from account
    where id=1005;--换成 where l=2 报错：异常，无数据；换成 where l=1 报错：返回
记录太多
    dbms_output.put_line('姓名: '||v_realname||' 年龄: '||v_age);
end;
```

3) select...into 后的查询结果集种类

①若查询结果是单行单列，into 子句后用标量类型（标准类型），与 select 子句后的目标数据类型一致即可。

②若查询结果是单行多列，into 子句后的变量个数、顺序、每个变量的数据类型，应该与 select 子句后的目标数据相匹配；也可以用记录类型的变量。

5.2 record 类型

记录类型，处理单行多列数据。

eg1: 语法

```
declare
    type t_cost_rec id record(base_cost cost.base_cost%type,--t_cost_rec 是记录类型
                                base_duration cost.base_duration%type,--base_cost 是成员
                                unix_cost cost.unit_cost%type);
    v_cost t_cost_rec;--v_cost 是记录类型的变量
    v_cost_1 t_cost_rec;
```

eg2: 打印每个客户的名字，年龄，身份证

```
declare
    type t_account_rec is record(real_name account.real_name%type,
                                    age number(3), idcard_no char(18));
    v_account t_account_rec;
begin
    select real_name,round((sysdate-birthdate)/365),idcard_no into v_account
    from account
```

```

where id=1005;
dbms_output.put_line('姓名: "'||v_account.real_name);
dbms_output.put_line('年龄: "'||v_account.age);
dbms_output.put_line('身份证: "'||v_account.idcard_no);
end;--t_account_rec 是记录类型, real_name 是成员, v_account 是记录类型的变量。

```

5.3 %rowtype

用表结构或视图结构定义变量

当使用%rowtype 定义记录变量时, record 成员的名称和类型与表或视图的列名称和类型完全相同。

eg: 语法

```
declare v_cost cost%rowtype;
```

5.4 record 变量的引用

1) 记录类型变量的属性引用方法是“.”引用, 即变量名.属性名。

eg1: 语法

```

declare
    v_cost.base_cost :=5.9;
    v_cost.base_duration :=20;
    v_cost.unit_cost :=0.4;
    v_cost_1 :=v_cost;
    select base_cost,base_duration,unit_cost into v_cost_1 from cost where id =2;

```

eg2: 语法

```

declare
    v_cost cost%rowtype;
begin
    select * into v_cost from cost where id=1;
    dbms_output.put_line(v_cost.name);
    dbms_output.put_line(v_cost.base_duration);
    dbms_output.put_line(v_cost.descr);
end;

```

◆ 注意事项: 打印记录类型, 一定是打印它的成员!

2) 若用 PL/SQL 程序对表进行 DML 操作, 则该表必须事先存在, 用 SQL 建表。

eg: 语法

```
create table cost_t1 as select base_cost,base_duration,unix_host from cost where id=3;
```

3) 在 insert 语句和 update 语句中可以使用记录类型变量。

eg: 语法

```

begin
    insert into cost_t1 values v_cost;
    update cost_t1 set row = v_cost_1;
    commit;
end;

```

4) 案例

eg: 提供客户 ID, 打印客户名称和身份证号, 若该客户不存在, 打印客户不存在

```
declare
    type t_account_rec is record(real_name account.real_name%type,idcard_no
char(18));
    v_account t_account_rec;
    v_cnt number;
begin
    select count(*) into v_cnt from account where id =1005;
    if v_cnt=1 then
        select real_name,idcard_no into v_account from account where id=1005;
        dbms_output.put_line('姓名: '|v_account.real_name);
        dbms_output.put_line('身份证: '|v_account.idcard_no);
    else
        dbms_output.put_line('用户不存在');
    end if;
end;
```

5.5 cursor 的概念

- 1) 根据 select 语句返回的记录数, 若返回记录数是 0 条或多条用 cursor 实现。
- 2) Oracle 使用专有 SQL 工作区 (private SQL workarea) 来执行 SQL 语句, 存储处理信息。这个工作区称为 “cursor”。
- 3) Oracle 所执行的每一个 SQL 语句都有唯一的 cursor 与之相对应。
- 4) 程序员用 PL/SQL 的 cursor 定义所需执行的 select 语句。

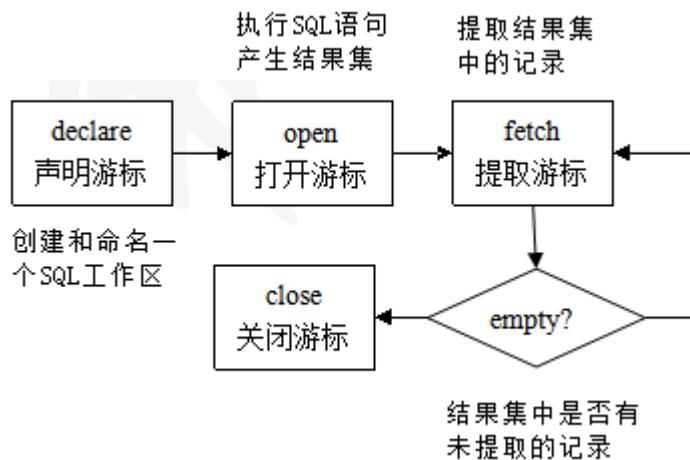
5.6 cursor 的分类

隐式: select...into 语句、DML 语句

显式: 返回多条记录的 select 语句用显式 cursor 实现。

5.7 显式 cursor 的处理

如下图所示:



5.8 显式 cursor 的属性

目的：获取有关显式 cursor 的状态信息。

属性	类型	描述
cursor_name%isopen	布尔	如果 cursor 是 open 的，其值为 true
cursor_name%notfound	布尔	如果前一个 fetch 语句没有返回一行记录，其值为 true
cursor_name%found	布尔	如果前一个 fetch 语句返回记录，其值为 true
cursor_name%rowcount	数值	到目前为止，cursor 已提取的总行数

- ◆ 注意事项：
 - ❖ 布尔值（true、false、null），当不 fetch 时，open 后直接%found，则返回 null。
 - ❖ %notfound 和%found 缺省为 null，由 fetch 去改变。
 - ❖ sqlplus 命令：exec 过程名，即可直接调用过程。

5.9 隐式 cursor 的属性

目的：获取有关隐式 cursor 的状态信息。

属性	类型	描述
sql%isopen	布尔	DML 执行中为 true，结束后为 false
sql%notfound	布尔	与 SQL%found 属性返回值相反
sql%found	布尔	值为 true 表示 DML 操作成功
sql%rowcount	数值	表示 DML 语句成功执行的数据行数

- ◆ 注意事项：sql 是指最近的 sql 语句。

eg: 相关操作

```
begin
    insert into test_chang values(100);
    dbms_output.put_line(sql%rowcount);
    update test_chang set c1=10 where c1=100;
    dbms_output.put_line(sql%rowcount);
end;
```

5.10 cursor 的声明

- 1) 在游标声明中使用标准的 select 语句。
- 2) 如果需要按指定的次序处理行，可在查询中使用 order by 子句。
- 3) 可以在查询中引用变量，但必须在 cursor 语句之前声明这些变量。

eg: 语法

```
declare
    cursor c_service_id(p_cost_id    number)is    select    id    from    service    where
cost_id=p_cost_id;
begin
```

5.11 open cursor

- 1) 通过 open cursor 来执行 select 语句并标识结果集。
- 2) select 语句如果没有返回记录，不会出现异常。
- 3) 语法：open c_service_id(5);

5.12 fetch cursor

- 1) 检索当前行，把值赋给变量。
- 2) 变量可以是 record 类型或简单变量。
- 3) 如果是简单变量：①包含相同数量的变量。②把每一个变量和相应的列进行位置匹配。
- 4) 通过循环检测 cursor 中是否包含数据行。
- 5) 语法：

```
fetch cursor_name into [var1,var2,...|record_name];
fetch c_service_id into v_service_id
```

- ◆ 注意事项：
 - ❖ fetch into 变量：变量一定是记录类型，与 select 后的列的个数、类型相同。
 - ❖ cursor 不会因为 select 子句的结果集为 0 或多值报错。

5.13 结果集提取的控制

- 1) 使用循环处理显式 cursor 结果集中的多行数据。
- 2) 每次 fetch 一行，反复进行。
- 3) 使用%notfound 属性检测一次不成功的提取操作。
- 4) 使用显式 cursor 的属性检测每一次提取是否成功，避免出现无限循环。

5.14 close cursor

- 1) 处理完结果集中的数据后，应该关闭 cursor。
- 2) 如果需要，可以再次打开该 cursor。
- 3) cursor 一旦关闭，所有和该 cursor 相关的资源都会被释放，不可再从关闭的 cursor 中提取数据。
- 4) 任何对关闭的 cursor 的操作都会引发 invalid_cursor 错误。
- 5) 每个 session 能打开的 cursor 数量由 open_cursor 参数决定。
- 6) 语法：close c_service_id;

eg1：打印每个客户的名字和身份证号码（使用 loop 循环处理 cursor）。

```
declare
  cursor c_account is select real_name from account where 1=1;
  v_realname varchar2(20);
begin
  open c_account;--open 则结果集已经在内存中了
  loop
    fetch c_account into v_realname;
    exit when c_account%notfound;
    dbms_output.put_line(v_realname);--若输出在判断前，则最后的记录多打印一次
  end loop;
  close c_account;
end;
```

eg2：打印每个客户的名字和身份证号码，若查询没有记录返回，打印客户不存在（使

用 loop 循环处理 cursor)。 where 1=1 输出所有用户名， where 1=2 输出 no account

```
declare
    cursor c_account is select real_name from account where 1=1;
    v_realname varchar2(20);
begin
    open c_account;
    fetch c_account into v_realname;
    if c_account%notfound then
        dbms_output.put_line('no account');
    else
        loop
            dbms_output.put_line(v_realname);
            fetch c_account into v_realname;
            exit when c_account%notfound;
        end loop;
    end if;
    close c_account;
end;
```

eg3：打印每个客户的名字和身份证号码，若查询没有记录返回，打印客户不存在（使用 loop 循环处理 cursor)。 v_realname 也可用 rowtype 定义为 cursor 类型，但要注意变量的引用。

```
declare
    cursor c_account is select real_name from account where 1=1;
    v_realname c_account%rowtype;
begin
    open c_account;
    fetch c_account into v_realname.real_name;
    if c_account%notfound then
        dbms_output.put_line('no account');
    else
        loop
            dbms_output.put_line(v_realname.real_name);
            fetch c_account into v_realname.real_name;
            exit when c_account%notfound;
        end loop;
    end if;
    close c_account;
end;
```

eg4：打印每个客户的名字和身份证号码，若查询没有记录返回，打印客户不存在（使用 while 循环处理 cursor)。

```
declare
    cursor c_account is select real_name,idcard_no from account where 1=1;
    v_account c_account%rowtype;
begin
```

```

open c_account;
fetch c_account into v_account;
if c_account%notfound then
    dbms_output.put_line('no account');
else
    while c_account%found loop
        dbms_output.put_line('姓名 : '||v_account.real_name||'身份证 : '
'||v_account.idcard_no);
        fetch c_account into v_account;
    end loop;
end if;
close c_account;
end;

```

- ◆ 注意事项：用 while 循环和 loop 循环时，注意 fetch 和 put_line() 的顺序；否则 while 会少打记录，loop 会多打记录。

eg5：打印每个客户的名字和身份证号码（使用 for 循环处理 cursor）

方式一：for 的最简单写法

```

begin
    for i in(select real_name,idcard_no from account)
    loop
        dbms_output.put_line('姓名: '||i.real_name||'身份证: '||i.idcard_no);
        --dbms_output.put_line(c_account%rowcount);无法打印，因为拿不到 cursor 的名
字
    end loop;
end;

```

- ◆ 注意事项：for 循环把 cursor 的操作集成了（open、fetch、close），自动打开关闭，自动把值 fetch 给 i，所以 i 的类型是记录类型。

方式二：定义一个 cursor

```

declare
    cursor c_account is select real_name,idcard_no from account;
begin
    for i in c_account
    loop
        dbms_output.put_line('姓名: '||i.real_name||'身份证: '||i.idcard_no);
        dbms_output.put_line(c_account%rowcount);
    end loop;
end;

```

eg6：打印每个客户的名字和身份证号码，若查询没有记录返回，打印客户不存在（使用 for 循环处理 cursor）

方式一：

```

declare
    cursor c_account is select real_name,idcard_no from account where 1=1;
    v_account c_account%rowtype;
begin

```

```
open c_account;
fetch c_account into v_account;
if c_account%notfound then
    dbms_output.put_line('no account');
else
    close c_account;
    for i in c_account
    loop
        dbms_output.put_line('姓名: "'||i.real_name||' 身份证: "'||i.idcard_no);
    end loop;
end if;
close c_account;
end;
```

方式二:

```
declare
    cursor c_account is select real_name,idcard_no from account where 1=1;
    v_cnt binary_integer := 0;
begin
    for i in c_account loop
        dbms_output.out_line(rpad(i.real_name,12,'')||i.idcard_no);
        v_cnt := 1;
    end loop;
    if v_cnt = 0 then
        dbms_output.put_line('account not exists!');
    end if;
end;
```

三十二、集合

LICHOO

6.1 什么是 collection

collection 是按某种顺序排列的一组元素，所有的元素有相同的数据类型，每个元素有唯一一个下标标识其在这一组元素中的位置。

分类：

- 1) Associative arrays (关联数组) 又称 index-by table，使用键值访问。
- 2) Nested table 嵌套表 (本课程不讲解)。
- 3) Varray 数据，变长数组，定义时需要指定数组大小 (本课程不讲解)。

6.2 什么是关联数组

- 1) 关联数组就是键值对的集合，其中键是唯一的，用于确定数组中对应的值。
- 2) 键可以是整数或字符串。第一次使用键来指派一个对应的值就是添加元素，而后续这样的操作就是更新元素。
- 3) 关联数组能帮我们存放任意大小的数据集合，快速查找数组中的元素。它像一个简单的 SQL 表，可以按主键来检索数据。

6.3 Associative arrays 的定义

- 1) 同种类型的一维、无边界的稀疏集合，只能用于 PL/SQL。
 - 2) type_name 是关联数组的类型名，element_type 是关联数组中存储的元素类型。
 - 3) index by 后面的数据类型是下标的数据类型。
- ◆ 注意事项：
- ❖ 关联数组和下标不要求连续，不要求从 0、从 1 开始。
 - ❖ 集合对下标是自动排序的，按元素排序要自己写算法。
 - ❖ 两个下标一样，则是覆盖操作！

eg1：语法

```
type type_name is table of element_type [not null]
index by [binary_integer|pls_integer|varchar2(size_limit)];
```

eg2：集合下标自动排序

```
dbms_output.put_line('index:'||v_account.first);--输出排序后的第一个下标
dbms_output.put_line('index:'||v_account.last); --输出排序后的最后一个下标
```

6.4 声明 Associative arrays 类型和变量

eg：声明类型

```
type associative_array_type is table of number index by binary_integer;
```

eg：声明变量

```
V1 associative_array_type;
V2 v1%type;
```

◆ 注意事项：

```
v_account account%rowtype;
type is table of v_account index by binary_integer;
-报错 v_account 是变量，不是数据类型，改为 account%rowtype 才正确
```

6.5 关联数组的操作

eg: 语法

```

declare
    type t_indtab is table of number
        index by binary_integer;
    v_indtab t_indtab;
begin
    v_indtab(1) :=1;
    v_indtab(5) :=5;
    v_indtab(6) :=6;
    v_indtab(10) :=10;
    dbms_output.put_line(v_indtab(10));
    dbms_output.put_line(v_indtab(7));
end;

```

6.6 Associative arrays 的方法

1) exists 方法:

- ①exists(1), 判断第 1 个元素是否存在。
- ②在使用 count 方法前, 建议先用 exists 来判断一下, 这样可以避免抛出异常。

eg: 语法

```

if courses.exists(i) then
    courses(i) := new_course;
end if;

```

2) count 方法:

- ①返回联合数组的元素个数, 不包括被删除的元素。
- ②对于空的联合数组, 返回值也是 0;

eg: 语法

```
if projects.count = 25 then .....
```

3) first 和 last: 返回最小和最大下标号, 如果 collection 为空, 则返回 null。

4) prior(n)和 next(n): 返回第 n 个元素的前一个和后一个, 如果不存在, 则返回 null。

5) trim(n): 从最后一个元素删除 n 个元素。

6) delete 方法:

- ①delete: 删除所有元素
- ②delete(n): 删除第 n 个元素
- ③delete(m,n): 从第 m 个元素删除到第 n 个元素

eg: 相关操作

```

declare
    type t_account_idxtab is table of varchar2(20)--元素的数据类型
        index by binary_integer;--下标的数据类型
    cursor c_account is select id,real_name from account;--用 id 作下标
    v_account t_account_idxtab;
begin
    for i in c_account loop

```

```

    v_account(i.id) := i.real_name;
    dbms_output.put_line('index:'||i.id);
    dbms_output.put_line('element:'||v_account(i.id));
end loop;
    dbms_output.put_line('element:'||v_account(1005));
    dbms_output.put_line('index:'||v_account.first);
    dbms_output.put_line('index:'||v_account.last);
end;

```

6.7 关联数组的遍历

eg1: 语法

```

declare .....
begin .....
    for i in v_indtab.first .. v_indtab.last loop
        dbms_output.put_line(v_indtab(i));
    end loop;
end;

```

eg2: 关联数组的遍历 (用 while 循环)

```

declare
    type t_account_idxtab is table of varchar2(20)--元素的数据类型
        index by binary_integer;--下标的数据类型
    cursor c_account is select id,real_name from account;--用 id 作下标
    v_account t_account_idxtab;
    v_index binary_integer;

begin
    for i in c_account loop
        v_account(i.id) := i.real_name;
    end loop;
    v_index := v_account.first;
    while v_index <= v_account.last loop
        dbms_output.put_line(v_account(v_index));
        v_index := v_account.next(v_index);--找 1005 的下一个下标
    end loop;
end;

```

eg3: 关联数组的遍历 (用 for 循环)

```

declare
    type t_account_idxtab is table of varchar2(20)--元素的数据类型
        index by binary_integer;--下标的数据类型
    cursor c_account is select id,real_name from account;--用 id 作下标
    v_account t_account_idxtab;

begin
    for i in c_account loop
        v_account(i.id) := i.real_name;
    end loop;

```

```

for i in v_account.first .. v_account.last loop
    if v_account.exists(i) then
        dbms_output.put_line(v_account(i));
    end if;
end loop;
end;

```

- ◆ 注意事项：
- ❖ 对于不连续的下标，用 for 循环会空转，效率低。
 - ❖ 下标是连续的用 for 循环好。
 - ❖ 下标不连续的用 while 循环好（匹配集合中的 next()方法）。

6.8 Associative arrays 的异常

现象：

```

declare
    type numlist is table of number index by binary_integer;
    nums numlist;--自动初始化为 null
begin
    nums(null) := 3;--触发 value_error, 没下标, 下标不能为空
    if nums(1)=1 then --触发 no_data_found, 下标有, 但没元素

```

6.9 批量绑定

- 1) 通过 bulk collect 减少 loop 处理的开销。
- 2) 采用 bulk collect 可以将查询结果一次性的加载到 collections 中。
- 3) 可以在 select into、fetch into、returning into 语句中使用 bulk collect。
 - ◆ 注意事项：选择操作的几种方式：select、select into、cursor fetch into、select bulk collect into、fetch bulk collect into、fetch bulk collect into limit 10
- 4) 注意在使用 bulk collect 时，所有的 into 变量都必须是 collections。
 - ◆ 注意事项：
 - ❖ 当返回多条记录时可用 bulk collect 和 cursor，但 cursor 是一条一条处理，bulk collect 是批量处理。
 - ❖ 集成度越高，可控的内容越少，如下标不可控了。
 - ❖ 使用 bulk collect 时，下标自动生成从 1 开始。
 - ❖ 使用问题：如果内存空间不够，则将无法执行下去，所以要限制 limit 一次取出的数量。

eg1: fetch bulk collect into

```

declare
    cursor is v_account 集合变量
begin
    open c_account;
    fetch bulk collect into v_account limit 3;
    close c_account;

```

eg2: 打印每个客户的名字、年龄、以及累计年龄

方式一：

```
declare
    type t_account_rec is record (real_name varchar2(20),age binary_integer);
    type t_account_idxtab is table of t_account_rec index by binary_integer;
    v_account t_account_idxtab;
    v_sum binary_integer :=0;
begin
    select real_name,round((sysdate-birthdate)/365) bulk collect into v_account from account where 1=1;
    dbms_output.put_line('RealName'||      ''||'Age'||      ''||'SumAge');
    if v_account.count<>0 then
        for i in v_account.first .. v_account.last loop
            v_sum :=v_sum + v_account(i).age;
            dbms_output.put_line(rpad(v_account(i).real_name,12,'')||
'|rpad(v_account(i).age,4,'')||' ''||v_sum);
        end loop;
    else
        dbms_output.put_line('no account');
    end if;
end;
```

方式二：

```
declare
    cursor c_account is select real_name,round((sysdate-birthdate)/365) age--想引用必须起别名
                           from account;
    type t_account_idxtab is table of c_account%rowtype index by binary_integer;
    v_account t_account_idxtab;
    v_culage binary_integer :=0;
begin
    select real_name,round((sysdate-birthdate)/365)age bulk collect into v_account
                           from account
                           where 1=1;--用 bulk collect 也可处理多条记录，下标自动生成从 1 开始
    if v_account.first is not null then
        for i in v_account.first .. v_account.last loop
            v_culage :=v_culage + v_account(i).age;
            dbms_output.put_line(rpad(v_account(i).real_name,12,'')||
'|rpad(v_account(i).age,4,'')||' ''||v_culage);
        end loop;
    else
        dbms_output.put_line('no account');
    end if;
end;
```

三十三、异常

LICHOO

7.1 Oracle 错误

- 1) PL/SQL 错误：编译时错误、运行时错误。
- 2) 运行时的错误：Oracle 错误 (ORA-XXXXX)、PL/SQL 运行错误、用户定义的错误。

7.2 Oracle 错误处理机制

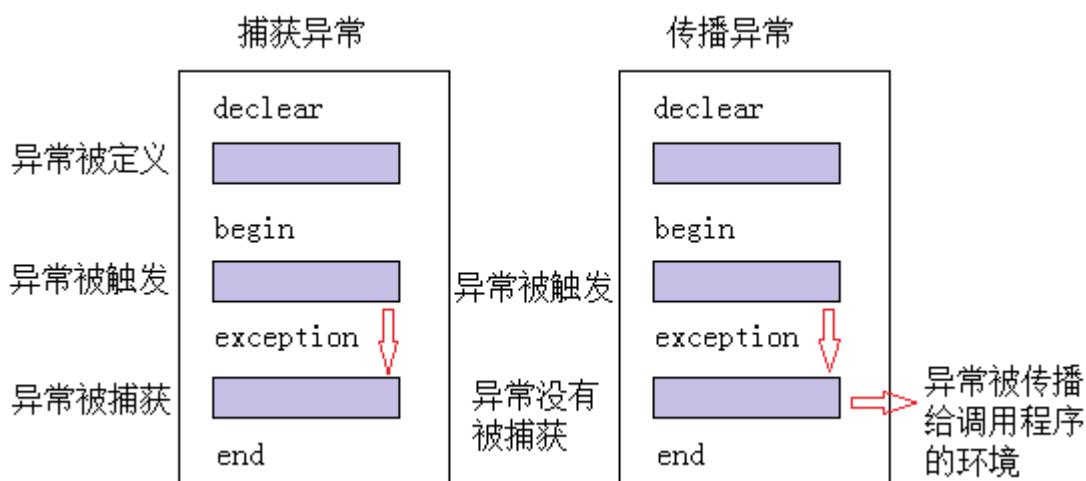
- 1) 在程序运行期间的错误对应一个异常 (exception)。
- 2) 一个错误对应一个异常，当错误产生时抛出相应的异常，并被异常处理器捕获，程序控制权传递给异常处理器，由异常处理器来处理运行时错误。

7.3 异常的类型

- 1) 隐式触发：Oracle 预定义异常、非 Oracle 预定义异常。
- 2) 显式触发：用户自定义异常。

7.4 PL/SQL 中的异常

如下图所示：



7.5 异常捕获

- 1) 异常的捕获通过异常处理器实现。它是程序的一个独立部分，把错误与程序的其他部分分离开来，使程序逻辑更加易于理解。
- 2) 当异常产生时，控制权立即转移到异常处理器。一旦执行控制权被转移到异常处理器，就无法再回到本语句块的可执行部分。如没有异常部分，则被传播到外层语句块（类似于 Java 中的异常一直向上抛）。
- 3) 异常处理器语法：

```
exception
    when exception_name1 then
        sequence_of_statements1;
    when exception_name2 then
        sequence_of_statements2;
    [when others then]
```

```
sequence_of_statements3;]  
end;
```

LICHOO

7.6 异常的捕获规则

- 1) exception 关键字，标识异常处理的开始区域。
- 2) 一个异常处理器可以捕获多个异常，只需要在 when 子句中用 or 连接即可。
- 3) 允许有多个异常处理器。
- 4) 一个异常只能被一个异常处理器捕获，并进行处理。
- 5) others 异常处理器总是作为异常处理部分的最后一个异常处理器，负责处理那些没有被其他异常处理器捕获的异常。

7.7 Oracle 预定义错误的捕获

- 1) no_date_found(ORA-1403): 如没有找到数据。
- 2) too_many_rows(ORA-1422): 如返回的行数超出请求行数。
- 3) invalid_cursor(ORA-1001): 如 cursor 没有 open 就使用。
- 4) zero_divide(ORA-1476): 如 0 做了除数。
- 5) dup_val_on_index(ORA-0001): 如多个重复索引（违反了唯一键约束）。
- 6) value_error(ORA-6502): 如集合（数组）中下标为 null（或者说元素为空）。
- 7) 不需要定义，不需要触发，只要捕获到即可。
- 8) Oracle 预定义错误的捕获

eg1: 语法

```
begin  
    select ..... commit;  
exception  
    when no_date_found then  
        statement1;  
    when too_many_rows then  
        statement2;  
    when others then  
        statement3;  
end;
```

eg2: 相关操作

```
declare  
    v_realname varchar2(20);  
begin  
    insert into test_chang values(1);  
    insert into test_chang values(1);  
    commit;--保证没有把锁带进异常  
    select real_name into v_realname from account where l=1;  
    dbms_output.put_line('a');--异常时，此句不执行  
exception--执行一个异常后  
    when no_data_found then--若不接异常，则报错 ORA—1403: no date found  
        dbms_output.put_line('no data found');  
    when too_many_rows then--若不接异常，则报错 ORA—1422: exact fetch returns
```

LICHOO

```

more than requested number of rows
    dbms_output.put_line('too many rows');
when dup_val_on_index then
    rollback;--要做都做，要不做都不做，回滚保证没有锁
end;

```

7.8 非 Oracle 预定义异常

将一个经过命名的异常和一个 Oracle 错误相关联。

- 1) 在语句块的声明部分声明一个异常名称: e_integrity exception;
- 2) 通过 pragma exception_init 将异常与一个 Oracle 错误号相关联:

```
    pragma exception_init(e_integrity,-2291)
```

- 3) 在异常处理部分捕获并处理异常: when e_integrity then

eg: 非预定义异常

```

declare
    e_noparent exception;
    pragma exception_init(e_noparent,-2291);
    e_childest exception;
    pragma exception_init(e_childest,-2292);
begin
    delete from parent_chang where c1=1;
    insert into child_chang values(1,2);
exception
    when e_noparent then
        dbms_output.put_line('no parent');
    when e_childest then
        dbms_output.put_line('e_childest');
end;

```

7.9 用户自定义异常

- 1) 用户自定义异常必须在声明部分进行声明。
- 2) 当异常发生时, 系统不能自动触发, 需要用户使用 raise 语句(类似 java 中的 throw)。
- 3) 在异常处理部分捕获并处理异常。

eg1: 自定义大于 100 为异常

```

declare
    v_n1 number :=101;
    e_more100 exception;
begin
    if v_n1 > 100 then
        raise e_more100;
    end if;
exception
    when e_more100 then
        dbms_output.put_line('>100');
end;

```

eg2: others 接受所有未捕获的异常, sqlcode、sqlerrm

```
declare
    v_n1 number :=101;
    e_more100 exception;
begin
    insert into test values('abc')--异常
    if v_n1 > 100 then
        raise e_more100;
    end if;
exception
    when e_more100 then
        dbms_output.put_line('>100');
    when others then
        dbms_output.put_line('a'||sqlcode)--获取错误的编号
        dbms_output.put_line('a'||sqlerrm)--获取错误的信息
end;
```

7.10 异常处理总结

- 1) 在声明部分为错误定义异常时，包括非 Oracle 预定义异常和用户定义异常
 - ①e_exception exception;
 - ②pragma exception_init(e_exception,-####);
- 2) 在执行过程中当错误产生时抛出与错误对应的异常：raise excepname。
- 3) 在异常处理部分通过异常处理器捕获异常，并进行异常处理。

7.11 sqlcode 和 sqlerrm

通过以下两个函数获取错误相关信息。

- 1) **sqlcode:** 返回当前错误代码
 - ①如果是用户定义错误返回值为 1；
 - ②如果是 ORA-1403: no data found 错误，返回值为 100
 - ③其他 Oracle 内部错误返回相应的错误号。

eg: 语法

```
exception--执行一个异常后
    when no_data_found then--若不接异常，则报错 ORA—1403: no date found
        dbms_output.put_line('no data found');
        dbms_output.put_line('a'||sqlcode)--100
    when too_many_rows then--若不接异常，则报错 ORA—1422: exact fetch returns more
than requested number of rows
        dbms_output.put_line('too many rows');
        dbms_output.put_line('a'||sqlcode)--1422
end;
```

- 2) **sqlerrm:** 返回当前错误的消息文本

- ①如果是 Oracle 内部错误，返回系统内部的错误描述
- ②如果是用户定义的错误，则返回信息文本为“user-defined exception”。

eg: 语法

```
declare
    v_n1 number :=101;
    e_more100 exception;
begin
    insert into test values('abc')--异常
    if v_n1 > 100 then
        raise e_more100;
    end if;
exception
    when e_more100 then
        dbms_output.put_line('>100');
    when others then
        dbms_output.put_line('a'||sqlcode)--a-1722
        dbms_output.put_line('a'||sqlerrm)--aORA-01722: 无效数字
end;
```

- ◆ 注意事项: `insert into log_tab values(sqlcode,sqlerrm,sysdate);`sqlerrm 不能直接插入表中, 要赋给一个变量! `v_errm :=sqlerrm;`
`insert into log_tab values(sqlcode,v_errm,sysdate);`--才能执行

7.12 异常的传播

1) 可执行部分异常的传播:

如果当前语句块有该异常的处理器, 则执行之, 并且成功完成该语句块。然后, 控制权传递到外层语句块。

如果当前语句块没有该异常的处理器, 则通过在外层语句块中产生该异常来传播该异常。然后, 对外层语句块执行步骤 1。如果没有外层语句块, 则该异常将传播到调用环境。

2) 声明部分异常的传播:

声明部分的异常立刻传播到外层语句块, 即使当前语句块有异常处理器。

3) 异常处理部分的异常的传播:

异常处理器中产生的异常, 可以有 `raise` 语句显式产生, 也可以通过运行时错误而隐含产生。异常立即被传播到外层语句块。

三十四、子程序

LICHOO

8.1 子程序

- 1) 匿名子程序，即匿名块 declare begin exception end;
 - ①匿名块不存在数据库中
 - ②每次使用时都会进行编译
 - ③不能在其他块中相互调用
- 2) 有名子程序

8.2 有名子程序

- 1) 命名的 PL/SQL 块，编译并存储在数据库中，可以在任何需要的地方调用。
- 2) 子程序的组成部分：子程序头、声明部分、可执行部分、异常处理部分（可选）。
 - ◆ 注意事项：有名子程序编译和运行是分开的。

8.3 有名子程序的分类

procedure: 过程, put_line()	function: 函数
package: 包, dbms_output	trigger: 触发器

8.4 有名子程序的优点

- 1) 模块化：将程序分解为逻辑模块。
- 2) 可重用性：可以被任意数目的程序调用。
- 3) 可维护性：简化维护操作。
- 4) 安全性：通过设置权限，是数据更安全。

三十五、过程 procedure

LICHOOL

9.1 语法

```
create [or replace] procedure proc_name [(arg_name[ {in|out|in out} ]) type,⋯ ]
{ is | as }
    <local variable declaration>--存储过程要用到的局部变量
begin
    <executable statements>
exception
    <exception handlers>
end;
```

9.2 创建存储过程

就是编译存储过程的代码，目的是为了调用它。

eg：语法

```
create or replace procedure p_accountno_chang--只要是 create 出来的都是数据库对象，类型是 procedure
is
    v_cnt number;--存储过程要用到的局部变量
begin
    select count(id) into v_cnt from account;
    dbms_output.put_line(v_cnt);
end;
```

◆ 注意事项：

- ❖ select status from user_objects where object_name='P_ACCOUNTNO_CHANG';--查看状态，名字必须大写，存的时候是按大写存的
- ❖ select type from user_source where name='P_ACCOUNTNO_CHANG';--查看是哪种有名子程序
- ❖ --create or replace view(procedure)编译不成功，但数据库对象已创建，只不过状态是 invalid

9.3 形参和实参

1) 在创建过程语句中的参数为形参

```
create or replace procedure procchang(p_n1 number)--p_n1 是形参
is
    v_n1 number;--v_n1 是过程中的局部变量
begin
    v_n1:=p_n1;
end;
```

2) 在调用过程时括号内的参数为实参

```
declare
    v_n1 number :=1;
```

```
begin
    procchang(v_n1);--v_n1 是实参
    procchang(2);--2 是实参
end;
```

◆ 注意事项：

- ❖ 形参相当于实参的“占位符”。
- ❖ 形参的名字和列名相同，系统则默认按列名。

9.4 形参的种类

- 1) in：参数的缺省模式！在调用过程的时候，实际参数的值被传递给该过程；在过程内部，形参是只可读的。
- 2) out：在调用过程时，任何的实参将被忽略；在过程内部，形参是只可写的。
- 3) in out：是 in 与 out 的组合。在调用过程的时候，实参的值可以被传递给该过程；在过程内部，形参可以被读出也可以被写入；过程结束时，控制会返回给控制环境，而形式参数的内容将赋给调用时的实际参数。

◆ 注意事项：

- ❖ in 相当于 java 中八大基本类型+string 的类似效果。
- ❖ in out 相当于 java 中的引用类型的类似效果。
- ❖ out 无对应。

9.5 调用存储过程

1) 用匿名块调用

```
begin
    procedure_name();
end;
```

2) 在 SQL 动作表中直接调用

```
exec procedure_name();
```

9.6 存储过程中的参数

eg1：语法

```
create or replace procedure procchang(
    p_c1 in varchar2,--只有此时 varchar 才不用写长度，长度由传入的实参决定
    p_c2 out varchar2,p_c3 in out varchar2)
is
    v_c1 varchar2(10);--存储过程要用到的局部变量
begin
    v_c1 :=p_c1;
    --p_c1 :=p_c1||'d';--报错，不能作赋值目标。in 在调用过程中，形参只可“读取”实参的值，即不能被写（不能在赋值语句左边）。
    p_c2 :=p_c2||'d';--out 在调用过程中，任何实参都被忽略，形参只可“写入”实参中（在过程内部可读，但无值，即 p_c2 :=p_c2||'d';不报错）。
    p_c3 :=p_c3||'d';--只有变量才能作赋值目标
end;
```

eg2: 用匿名块调用过程

```

declare
    v_c2 varchar2(20) := 'abc';--对于传出参数没必要定义初值，即也可 v_c2 varchar2(20);
    v_c3 varchar2(20) := 'abc';
begin
    procchang('abc',v_c2,v_c3);--过程的调用可直接单独写出，但函数不行，不要有变量
接收返回值
    dbms_output.put_line(v_c2);--存储过程依靠 out 返回， d
    dbms_output.put_line(v_c3);--abcd
end;

```

9.7 对实际参数的要求

- 1) 模式为 in 的形参对应的实际参数可以是“常量或变量”。
- 2) 模式为 in out 或 out 的形参对应的实际参数必须是“变量”，用于存储返回的值，所以不能是常量或表达式。

9.8 形式参数的限制

- 1) 在调用过程当中，实际参数在将值传递给过程时，也传递了对变量的限制。
- 2) 形式参数不能声明长度，但可使用%type 来进行限制。

9.9 带参数的过程调用

- 1) 位置表示法：调用时添入所有参数，实参与形参按顺序一一对应。
- 2) 名字表示法：调用时给出形参名字，并给出实参：

```

procname(12,p_outparm=> v_var1,p_inout=> 10);
        前为形参，后为实参；理解为形参对应的实参

```

- 3) 两种方法可以混用：混用时，第一个参数必须通过位置来指定，名字表示对于参数很多时，可提高程序的可读性。

9.10 使用缺省参数

- 1) 形参可以指明缺省值：parm_name [mode] type {:= | default} init_value
- 2) 位置标示法时，所有的缺省值都放在最后面。
- 3) 使用名字标示法则无所谓。
- 4) 声明时，如果有缺省值，尽量将缺省值放在参数表的末尾。

eg: 使用缺省参数 default

```

create or replace procedure pro_chang1(p_c1 number,p_c2 number,p_c3 number,
p_c4 number default 1,p_c5 number default 1)
is
begin

```

```

    dbms_output.put_line(p_c1);      dbms_output.put_line(p_c2);
    dbms_output.put_line(p_c3);      dbms_output.put_line(p_c4);
    dbms_output.put_line(p_c5);
end;

```

exec pro_chang1(1,2,3);--1,2,3,1,1 把缺省值放最后

```
exec pro_chang1(1,2,3,4,5);--1,2,3,4,5
```

9.11 存储过程中的 DDL 语句

eg: 语法

```
create or replace procedure proc1  
is  
begin  
    execute immediate 'create table test(c1 number)' ;  
end;  
exec proc1
```

ORA-01031:权限不足

◆ 注意事项: 为何会出现创建 table 权限不足问题? 明明创建 table 已经好几天了? !

问题分析:

1) DBA 创建用户:

```
create user jsd1304;--创建的用户初始什么都做不了, 只能慢慢授权各种权限
```

```
create role connect--角色 connect, 是缺省创建的
```

```
create role resource--另一种角色 resource
```

2) 把角色授权给用户:

```
grant connect,resource to jsd1304;--授权两种角色给用户 1304
```

```
user role:enable、disable--用户角色分为: 可用和不可用
```

3) 举例说明:

```
create role role1--创建角色 role1
```

```
grant create table to role1;--把创建 table 的权限授予角色 role1
```

```
grant create index to role1;--把创建 index 的权限授予角色 role1
```

```
grant create view to role1;--把创建 view 的权限授予角色 role1
```

```
grant role1 to jsd1304;--把角色 role1 所拥有的权限授予用户 jsd1304
```

4) 用户能创建 table 的原因: 用户被授予了一个角色, 而角色是可用的

```
connect role enable priviledge 可用
```

5) 而在处理有名块时, 把用户的所有角色 disable, priviledge 不可用了, 所以会出现权限不足问题

6) 解决方法: grant create table to jsd1304;--直接授予的权限不会被 disable

7) 因此, 可总结如下:

①在过程中执行 DDL 操作, 所需的权限必须通过直接授予的方式, 不能通过角色授予。

②调用过程时, 所有角色都是 disable 的 (系统做的), 即角色中包含的所有权限都不能生效。

③调用其他用户的过程, 必须由过程的属主授予执行权限: grant execute on procname to username;

④权限-->角色-->用户<--权限

9.12 变量

1) 局部变量: 在匿名块或存储过程中定义的变量为局部变量, 即作用域在整个匿名块或存储过程中。程序运行结束, 该变量也就不存在了。

2) 绑定变量 (宿主变量): 在 PL/SQL 的 SQL 中可以直接使用绑定变量: bv_name, 即

也可不需要定义。标志是以“冒号”开头。

eg1：绑定变量的定义

--variable i number--绑定变量的定义，也可不声明；且不能定义宽度；且最后一定要有赋值

```
begin
    for i in 1..10 loop
        execute immediate 'insert into test_chang values (:i)' using i;
        --系统使用 using 自动给绑定变量赋值，用绑定变量是占位置的，若不声明则系统认为存在！
    end loop;
    dbms_output.put_line(:i);
    commit;
end;
```

eg2：绑定变量的赋值

```
variable i number
begin
    i :=1;
end;
print i;--1,print 是环境里的命令，不是 PL/SQL 命令
exec :i :=10000--绑定变量的赋值
print i;
```

3) 各类变量总结

①variable i: 绑定变量，不退出环境，变量始终存在，跨 session 的，生命周期最长。

②declare .. begin 之间和 is .. begin 之间定义的变量：叫局部变量，匿名块退出，变量不存在。

③for i: i 变量不需要声明，for 循环结束，变量不存在。

④调用一个函数、过程时，需要提供实参，绑定变量和局部变量可作实参。作用域匿名块里。

⑤定义一个函数、过程时，用形参，形参作用域在函数、过程里。

⑥在包 package 声明中的变量 v1，在包中的过程、函数里都能直接调用，不用写包名；如果出了包：匿名块、过程、函数、其他包里可以用包名调，pkg.v1。也称为 session 里的全局变量。

9.13 PL/SQL 中的 SQL 分类

1) 静态 SQL: 在 PL/SQL 块中使用的 SQL 语句在编译时是明确的，执行的是确定对象，即 SQL 语句是在 PL/SQL 编译阶段编译的。

◆ 注意事项：

- ❖ SQL 书写是有规范的，目的是减少硬分析（硬分析对 cpu、内存的耗费代价最高）。
- ❖ SQL 中使用绑定变量（非常重要）！目的是减少硬分析。
- ❖ SQL 出现在各种开发工具中，两种形式：静态 SQL，动态 SQL。比如：jdbc 只能提供动态 SQL 方式，PL/SQL 可以使用静态方式（运行时 sql 已经编译过了，oracle 对其做额外的优化，如 cursor cache，减少软分析的次

- 数)
- ❖ 一个 session 中可以同时执行 sql 的数量是有限的（每打开一个 cursor 就会占内存，内存限制）。
 - ❖ 一个 session 可以打开的 cursor 是有限的（参数限制 open_cursor）。

2) 动态 SQL: 在 PL/SQL 块编译时 SQL 语句是不确定的，如根据用户输入的参数的不同而执行不同的操作。编译程序对动态语句部分不进处理，只是在程序运行时动态地创建语句、对语句进行语法分析并执行该语句。

- ◆ 注意事项：在编译期 execute immediate 相当于 SQL 语句，后面的就是单纯的字符串，不看内容。只有在执行时才看、报错。

3) PL/SQL 中的静态 SQL:

①Oracle 在解析 SQL 时会吧 PL/SQL 中定义的变量转化为绑定变量，insert into test_chang values(i);减少了硬分析次数。

②server process 将执行完的 SQL cache 起来，不关闭！当再次执行 SQL 时，不需要软分析。

③静态高效的原因是系统自动使用了绑定变量，同时动态 SQL 也可用绑定变量方式。

④静态 SQL 的 1 次软分析是由于 cursor 用完没有被关闭，而是下次继续使用。

⑤过程中的参数会自动转化为绑定变量。

eg1: 静态 SQL

```
create or replace procedure proc1
is
begin
    for i in 1..1000 loop
        insert into test_chang values(i);
    end loop;
    commit;
end;--编译存储过程时编译 SQL 语句
begin      proc1      end; --1 次硬分析，1 次软分析，1000 次执行，静态 SQL 的 1 次软分析是由于 cursor 用完没有被关闭，而是下次继续使用
```

eg2: 动态 SQL (未使用绑定变量)

```
create or replace procedure proc1
is
begin
    for i in 1..1000 loop
        execute immediate 'insert into test_chang values('||i|| ')';
    end loop;
    commit;
end;--编译存储过程时不编译 SQL 语句
begin      proc1      end; --1000 次硬分析，1000 次软分析，1000 次执行
```

eg3: 动态 SQL (使用绑定变量)

```
create or replace procedure proc1
is
begin
    for i in 1..1000 loop
```

```

    execute immediate  'insert into test_chang values(:i) using j;
end loop;      commit;
end;--编译存储过程时不编译 SQL 语句
begin      proc1      end; --1 次硬分析, 1000 次软分析, 1000 次执行

```

LICHOO

9.14 再一次来看 SQL 语句的处理过程

- 1) 语法检查 (syntax check): 检查此 SQL 的拼写是否符合语法。
- 2) 语义检查 (semantic check): 诸如检查 SQL 语句中的访问对象是否存在及该用户是否具备相应的权限。
- 3) 对 SQL 语句进行解析 (parse): 利用内部算法对 SQL 进行解析, 生成解析树 (parse tree) 及执行计划 (execute plan)。
- 4) 执行 SQL, 返回结果 (execute and return)。

9.15 软分析和硬分析

Oracle 利用内部的 hash 算法来取得该 SQL 的 hash 值, 然后在 library cache 里查找是否存在该 hash 值。

- 1) 假设存在, 则将此 SQL 与 cache 中的进行比较; 假设“相同”, 就将利用已有的解析树与执行计划, 而省略了优化器的相关工作。这也就是软解释的过程。
 - 2) 如果上面两个假设中有任意一个不成立, 那么优化器都将进行创建解析树、生成执行计划的动作。这个过程就叫做硬解析。
- ◆ 注意事项: hash 算法对大小写是敏感的

9.16 对过程 procedure 的基本操作

- 1) 创建并编译过程: create or replace procedure
- 2) 编译过程: alter procedure procname compile;
- 3) 调用过程: 用匿名子程序调用, 直接写过程名; 用有名子程序调用, 直接写过程名。
- 4) 删除过程: drop procedure;

9.17 案例

eg1: 提供客户 ID, 返回客户名称、年龄, 若该客户不存在则返回 no account

```

create or replace procedure pro_chang(p_id number,name out varchar2,age out number)
is
begin
select real_name,round((sysdate-birthdate)/365)into name,age from account
where id=p_id;
exception
when no_data_found then
name:='no account';
age:=0;
end;

declare
v_realname varchar2(20);
v_age number(3);

```

```
begin  
    pro_chang(105,v_realname,v_age);  
    dbms_output.put_line(v_realname||'||v_age);  
end;
```

eg2: 使用绑定变量

```
variable b_realname varchar2;  
variable b_age number;  
begin  
    pro_chang(1005,:b_realname,:b_age);  
end;  
  
print b_realname;  
print b_age;
```

◆ 注意事项: 也可使用 exec pro_chang(1005,:b_realname,:b_age);直接调用

三十六、函数 function

LICHOO

10.1 语法

```
create [or replace] function func_name[(arg_name [{in|out|in out}]) type,⋯⋯]
return type
{is|as}
    <local variable declaration>--函数中要用到的局部变量
begin
    <executable statements>
    return value;
exception
    <exception handlers>
end;
```

10.2 创建函数

```
create or replace function fun1(p_in number,p_out out number) return number
is
begin
    p_out :=2;
    return p_in;
end;
```

10.3 调用函数

方式一：

```
declare
    v_out number(3);
begin
    dbms_output.put_line(fun1(10,v_out)||''||v_out);
end;
```

方式二：

```
declare
    v_out number(3);
    v_fun1 number;--java 中函数可以单独使用，但 PL/SQL 中不可以，有传出要定义变量接收。
begin
    v_fun1 :=fun1(10,v_out);--函数不能单独放，函数的调用方式为表达式右边
    dbms_output.put_line(v_fun1||''||v_out);
end;
```

方式三：

```
variable b_out number
exec dbms_output.put_line(fun1(10,b_out));
print b_out
```

10.4 对函数 function 的基本操作

- 1) 创建并编译函数: create or replace function
- 2) 编译函数: alter function funcname compile;
- 3) 调用函数: 用匿名子程序调用, 允许写值的位置就可以调用函数; 用有名子程序调用, 允许写值的位置就可以调用函数; 在 DML 语句和 select 语句中调用。
- 4) 删除函数: drop function;

10.5 过程和函数的比较

过程	函数
作为 PL/SQL 语句执行	作为表达式的一部分调用
在过程头中不包含 return	必须在函数头中包含 return 子句
不返回任何值	必须返回值
可以包含 return 语句, 但不能有返回值, 表示过程终止运行	必须包含至少一条 return 语句, 多条时, 只有一条被执行

◆ 注意事项: 过程的那一套东西也都适合函数。

10.6 匿名块中的过程和函数声明

```
declare
    v_n1 number :=1;
    function fun1(p_in number) return number
    is
    begin
        return p_in;
    end;
    procedure proc1
    is
    begin
        dbms_output.put_line(fun1(v_n1));
    end;
begin
    proc1;
end;
```

◆ 注意事项: 过程和函数只能在本匿名块中调用

10.7 案例

eg1: 创建一个函数并调用

```
create or replace function fun1(p_c1 varchar2,p_c2 out varchar2)
return varchar2
is
begin
    p_c2 :=p_c1||'d';
    return p_c1;
end;
```

```
declare
  v_c2 varchar2(20);
begin
  dbms_output.put_line(fun1('abc',v_c2));--abc
  dbms_output.put_line(v_c2);--abcd
end;
```

eg2: 提供客户 ID, 返回客户名称、年龄, 若该客户不存在则 null

```
create or replace function getaccount(p_id number,p_age out number) return varchar2
is
  type t_account_rec is record(real_name account.real_name%type,age number(3));
  v_account t_account_rec;
begin
  select real_name,round((sysdate-birthdate)/365) into v_account
  from account
  where id = p_id;
  p_age := v_account.age ;
  return v_account.real_name;
exception
  when no_data_found then
    v_account.real_name := 'null';
    return v_account.real_name;
end;

declare
  v_age number(3);
begin
  dbms_output.put_line(getaccount(10111,v_age)|| v_age);
end;
```

LICHOOL

三十七、包 package

LICHOO

11.1 什么是 package

1) package (包) 是一个可以将相关对象存储在一起的 PL/SQL 结构。package 包含了两个分离的组成部分：specification (package 的声明，即包声明) 和 body (声明中的程序实现，即包体)。每个部分都单独被存储在数据字典中。包声明是一个操作接口，对应用来说是可见的。

2) 包体是黑盒，对应用来说隐藏了实现细节。

11.2 包的组成

将相关的若干程序单元组织到一块，用一个包来标识这个集合，包中可以包含以下的程序单元：

程序单元	描述
过程 (procedure)	带有参数的程序
函数 (function)	带有参数的程序，该程序有返回值
变量 (variable)	用于存储变化值的存储单元
游标 (cursor)	定义一条 SQL 语句
类型 (type)	定义复合类型 (record、collection)
常量 (constant)	定义常量
异常 (exception)	标识异常

11.3 包的优点

- 1) 方便对存储过程和函数的组织：
①将相关的过程和函数组织在一起。②在一个用户环境中解决命名的冲突问题。
- 2) 方便对存储过程和函数的管理：
①在不改变包的声明定义是可以改变包体的实现的。②限制过程、函数的依赖性。
③在包体未实现时，其他程序中可以调用包中的对象，对自己的程序进行编译，可以并行地对程序开发。
- 3) 方便对存储过程和函数的安全性管理：
①整个包的访问权限只需一次性授权。②区分公用过程和私有过程。
- 4) 改善性能：
①在包被首次调用时作为一个整体全部调入内存。②减少多次调用时磁盘 I/O 次数。

11.4 package 声明的语法

```
create or replace package pkg_name
{is|as}
    公共变量 (variable) 的定义 | 公共类型 (type) 的定义 |
    公共异常 (Exception) 的定义 | 公共游标 (cursor) 的定义 |
    函数说明 | 过程说明
end;
```

11.5 package body 声明的语法

```
create or replace package body pkg_name
{is|as}
    函数实现;--调用一次执行一次
    过程实现;--调用一次执行一次
begin
    --初始化代码;--首次调用包中的任意对象则执行一次（只执行1次）
end;
```

11.6 编译包和包体

```
alter package pkg1 compile;
alter package pkg1 compile body;
```

11.7 案例

eg1:创建一个包并调用

```
create or replace package pkg_chang--创建 package
is
    type t_rec is record(m1 number,m2 varchar2(10));
    v_rec t_rec;
    procedure proc1;
    function fun1(p_in number)return number;
end;
--创建 package body
create or replace package body pkg_chang
is
    procedure proc1--存储在数据库中的 package 包里
    is
        begin
            dbms_output.put_line(v_rec.m1);
        end;
    function fun1(p_in number)return number
    is
        begin
            return p_in;
        end;
    --begin
    --v_rec.m1 :=100;
    end;
--调用 package
begin
    pkg_chang.v_rec.m1 :=pkg_chang.fun1(10);
    pkg_chang.proc1;
end;
```

eg2：创建一个包并调用

```
--创建 package
create or replace package pkg_chang1
is
    type t_rec is record(realname varchar2(30),age number);
    v_account t_rec;
    procedure p_account;
    function f_account(p_id number)return t_rec;
end;

--创建 package body
create or replace package body pkg_chang1
is
    procedure p_account
    is
    begin
        dbms_output.put_line('姓名：'||v_account.realname||' 年龄：'||v_account.age);
    end;

    function f_account(p_id number)return t_rec
    is
        --v_account t_rec;
    begin
        select real_name,round((sysdate-birthdate)/365) into v_account from account
        where id=p_id;
        return v_account;
    end;
end;

--调用 package
begin
    pkg_chang1.v_account :=pkg_chang1.f_account(1005);
    pkg_chang1.p_account;
end;
```

◆ 注意事项：

- ❖ to_char、nvl 等常用函数都在系统提供的 standard package 包中，只不过可以省略包名，其他包都要写包名；包内部可不写包名，出了包要写。
- ❖ 包里的过程和函数也可以重载。
- ❖ 包声明变了，包体一定要变。
- ❖ 可以有包声明，而没有包体，但不能只有包体而没有包声明。
- ❖ 删包，则包体也被删除了。
- ❖ 若 package 没有更新，只更新了 package body，调用包的程序代码不需要修改，也不需要重新编译。

三十八、触发器 trigger

LICHOO

12.1 面临问题

- 1) 在作 DML 操作时，不需要提供主键值，系统自动生成。
- 2) 如何实现级联更新。

12.2 DML 触发器的组成

组成部分	描述	可能值
触发时间	触发事件的时间次序	before、after
触发事件	DML 语句是触发事件	insert、update、delete
触发器类型	触发器体被执行的次数	statement（语句级）、row（行级）
触发器体	该触发器将要执行的动作	完整的 PL/SQL 块

12.3 DML 触发器的类型

- 1) 语句级触发器。
- 2) 行级触发器。
- 3) 行级触发器与语句级触发器的区别：触发的次数不同，如果 DML 语句只影响一行，则语句级与行级触发器效果一样；如果影响多行，则行级触发器触发的次数比语句级触发器触发的次数多。

12.4 DML 触发器的触发顺序

根据触发的时间、类型不同，可以组合为四种 DML 触发器。

触发时间	级别	描述
before	statement	在 SQL 语句执行之前执行一次
before	row	SQL 语句影响的每条记录被 update、delete 或 insert 之前执行一次
after	row	SQL 语句影响的每条记录被 update、delete 或 insert 之后执行一次
after	statement	在 SQL 语句执行之后执行一次

- ◆ 注意事项：如在 update 表时分为 5 个执行区间：做 update 语句之前；修改每一条符合条件的记录之前；修改记录；修改每一条符合条件的记录之后；做完 update 语句之后；

12.5 DML 行级触发器

for each row 子句创建一个行级触发器，使其在受到触发事件影响的每一行上都被触发。

eg: 语法

```
create [or replace] trigger trig_name {before|after} insert [or update...]
on tab_name
for each row [when restricting_condition]
PL/SQL block;
```

12.6 :OLD 和:NEW

在行级触发器中，在列名前加上：old 标识符表示该列变化前的值，加上：new 标识符表示变化后的值（绑定变量）。

触发事件	: old.列名	: new.列名
insert	所有的字段是 null	insert 语句中要插入的值
update	在 update 之前该列的原始值	update 语句中要更新的新值
delete	在 delete 行之前的该列的原始值	所有字段都是 null

eg1：自动产生主键值

--创建表

```
create table test_chang(c1 number primary key,c2 number);
```

--产生序列号

```
create sequence s1_chang;
```

--创建触发器，以序列号作为 c1 列的值

```
create or replace trigger gen_pk_chang
```

```
before insert on test_chang
```

```
for each row
```

```
begin
```

```
select s1_chang.nextval into :new.c1
```

```
from dual;
```

```
end;
```

--只插入 c2 列的值，发现不报错了。原因：系统自动将序列号插入 c1 列

```
insert into test_chang(c2) values(1);
```

--把 c1 列的值也插入，发现数字 10 被序列号替换了，before insert for each row 覆盖了 :new 的值

```
insert into test_chang values(10,5);
```

若不想被替换，则匿名块修改为：

```
if :new.c1 is null then
```

```
    select s1_chang.nextval into :new.c1
```

```
    from dual;
```

```
end if;
```

◆ 注意事项：: new 和 : old 是记录类型。

- ❖ : new 和 : old 只能用于行级触发器，不能用于语句级触发器。

- ❖ : new tabname%rowtype，绑定变量 new 和定义触发器的 table 的行记录结构相同。

- ❖ : old tabname%rowtype，绑定变量 old 和定义触发器的 table 的行记录结构相同。

- ❖ after 行级触发器不可以修改 : new 的值，但可以查看 : new 的值。

- ❖ insert into test_chang(c2) values(1); :new.c1=null :new.c2=1 --若没有触发器，则报错。原因：将 null 插入了主键列。

eg2：实现级联更新

--创建父子表

```
create table parent_chang(c1 number primary key,c2 number);
```

```
create table child_chang(c1 number primary key,c2 constraint child_c2_fk references parent_chang(c1));
```

--向父子表中插入数据

```
insert into parent_chang values(11,1);
```

```
insert into parent_chang values(12,2);
```

```
insert into child_chang values(21,11);
insert into child_chang values(22,12);
```

--若无触发器，更新 c1 列的值则报错。

```
update parent_chang set c1=1 where c1=11;
```

--若定义触发器

```
create or replace trigger cascade_update_chang
```

```
after update on parent_chang
```

```
for each row
```

```
begin
```

```
    update child_chang set c2=:new.c1
```

```
    where c2=:old.c1;
```

```
end;
```

--再次执行更新语句，则不报错，两表数据同时被更新

```
update parent_chang set c1=1 where c1=11;
```

◆ 注意事项：

- ❖ 在触发器里，缺省状态下，不能写提交和回滚的！
- ❖ 另一状态：Oracle 自治事务，可写。
- ❖ 约束检查发生在触发器完成之后。

12.7 触发器的重新编译

1) 如果触发器内调用其他函数或过程，当他们被删除或修改后，触发器的状态被表示为无效。当 DML 语句激活一个无效触发器时，Oracle 将重新编译触发器代码，如果编译时发现错误，这将导致 DML 语句执行失败。

2) 调用 alter trigger 语句重新编译已创建的触发器：

```
alter trigger [schema.] trig_name compile;
```

◆ 注意事项：查看触发器列表

```
desc user_triggers;      select * from user_triggers;
```

12.8 触发器的状态

1) 有效状态 (enable)：当触发事件发生时，处于有效状态的 trigger 将被触发。

2) 无效状态 (disable)：当触发事件发生时，处于无效状态的 trigger 将不会被触发。

3) trigger 的两种状态可以相互转换，格式为：

```
alter trigger trig_name [disable|enable];
```

4) alter trigger 语句一次只能改变一个触发器的状态，而 alter table 语句则一次能改变与指定表相关的所有触发器的使用状态。

```
alter table [schema.] tab_name {enable|disable} all triggers;
```

◆ 注意事项：drop table 后：

- ❖ 表中的 all constraints drop、表中的 all index drop、表中的 all trigger drop
- ❖ 无效的有：view、synonym、procedure、function、package

三十九、其他注意事项

LICHOO

13.1 PL/SQL 的特点

- 1) 结构化模块编程。
- 2) 良好的可移植性。
- 3) 良好的可维护性。
- 4) 提升系统性能。
- 5) 不便于向异构数据库移植应用程序。

13.2 写 PL/SQL 的好处

client 端可以不用写 sql，人员要求低，安全，权限控制。

相对于 C/S 结构，代码存在数据库中，维护的成本低（JDBC 的 java 程序相当于 C/S 结构）。

B/S 结构：sql 代码在 as 上（application server）。

能写静态 sql，可以预先编译的，效率高。

JDBC 只能写动态 sql，程序在运行时，sql 语句才编译。

13.3 命名建议

b_绑定变量 v_普通变量 p_形参

13.4 搞清楚如下内容

- 1) 过程参数 in、out、in out。
- 2) 过程，角色，权限的关系。
- 3) 静态 sql、动态 sql、绑定变量。

13.5 保证所有对象的状态都是 valid

```
alter procedure procname compile;
alter package pkg1 compile;
alter package pkg1 compile body;
```

13.6 declare 中都可声明什么

变量、类型、cursor、exception、过程、函数

13.7 数据库对象

table、view、index、sequence、synonym、procedure、function、package、package body

4 JDBC 学习笔记

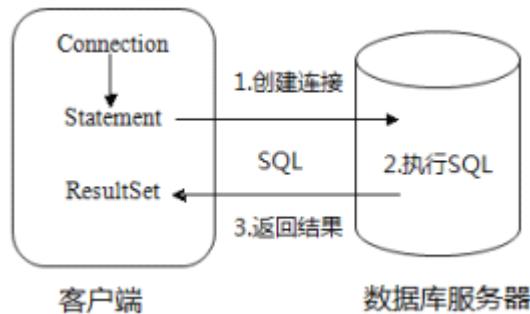
四十、JDBC 概述

1.1 什么是 JDBC

- 1) Java 的设计者希望使用相同的方式访问不同的数据库。
- 2) JDBC 是 Java 用于统一连接数据库并操作数据库的一组通用接口定义（即通过一系列接口定义了访问数据库的通用 API）。
- 3) JDBC 是连接数据库的规范，不同的数据库厂商若想让 Java 语言可以对其操作，就

需要实现一组类，这组类需要实现 Java 提供的这组用于连接数据库的接口，并实现其中定义的相关方法。那么不同的数据库厂商根据各自数据库的特点，去提供对 JDBC 的实现（实现类包），那么这组类就是该数据库的驱动包了。

4) 原理图：



1.2 什么是驱动

简单的说就是让软件知道如何去操作硬件。

1.3 SQL lite

是轻量级的数据库，常用于嵌入式。

1.4 如何使用 Java 连接某种数据库

需要两个部分：1) 使用 JDBC 连接数据库（导入某数据库的.jar 包）。
2) 提供对该数据库的驱动包（使用静态方法 Class.forName 注册驱动）。

1.5 连接数据库并操作

- 1) 打开与数据库的连接（使用 DriverManager.getConnection 获取连接）。
- 2) 执行 SQL 语句（使用 Statement 或者 PreparedStatement）。
- 3) 得到结果。

1.6 连接数据库时常见的错误

- 1) 报错 ClassNotFoundException 则有两种情况：
 - ①驱动包没导入。
 - ②Class.forName()中的字符串拼写有误。
 - 2) 报错 port number，应注意：
 - ①连接数据库时输入数据库路径时没有添加端口号。
 - ②Oracle 数据库的完整写法应为：jdbc:oracle:thin:@IP 地址:端口号:数据库名
- ◆ 注意事项：Oracle 数据库默认端口号 1521。MySQL 数据库默认端口号为 3306。

四十一、 JDBC 核心 API

2.1 Connection

接口，需导入 `java.sql.Connection` 包，与特定数据库进行连接（会话）。

2.2 Statement

接口，需导入 `java.sql.Statement` 包，用于执行静态 SQL 语句并返回它所生成结果的对象。

- 1) `ResultSet executeQuery(String sql) throws SQLException` 方法：执行给定的 SQL 语句（通常为静态 SQL SELECT 语句），该语句返回单个 `ResultSet` 对象。
- 2) `boolean execute(String sql) throws SQLException` 方法：执行给定的 SQL 语句，该语句可能返回多个结果。如果第一个结果为 `ResultSet` 对象，则返回 `true`；如果其为更新计数或者不存在任何结果，则返回 `false`。详细介绍请看 2.6 案例注释。
- 3) `int executeUpdate(String sql) throws SQLException` 方法：执行给定 SQL 语句，该语句可能为 INSERT、UPDATE、DELETE（DML 语句），或者不返回任何内容的 DDL 语句。
返回值：①对于数据操作语句（DML 语句），返回行计数。②对于 DDL 语句，返回 0。
- 4) `boolean execute(String sql)` 方法：返回结果为 `true`、`false`，常用与执行表级操作的 SQL 语句，如建表、删表等，创建表若失败实际上是会直接抛出异常的。`false`：为建表成功的标志。
- 5) `execute()` 方法：原则上可以执行任意 SQL 语句。返回 `true`：若执行结果为一个结果集（`ResultSet`）。返回 `false`：为其他信息（如影响表数据总条数等）。所以我们通常不会使用 `execute` 去执行查询语句。
- 6) `int executeUpdate(String sql) throws SQLException` 方法：返回值 `int`，返回值为当前执行的 SQL 语句影响了数据库数据的总条数；该方法常用与执行 `insert`、`update`、`delete` 语句。
- 7) 在底层一定会用到网络 `Socket` 和流，但我们不用关心使用字符还是字节接收，都由 `Statement` 做了。

2.3 ResultSet

接口，表示数据库结果集的数据表（很像一个集合），通常通过执行查询数据库的语句生成。

- 1) `ResultSet` 特点：按行遍历，按字段取值。
- 2) 它的 `next()` 方法包含了是否有下一条记录的 `hasNext()` 方法。
- 3) 按字段取值时，`getString(int)` 方法中的 `int`，代表结果集的第几列，
◆ 注意事项：这里的 `int` 从 1 开始，和 Java 对索引的习惯不同。

2.4 DriverManager

它是管理一组 JDBC 驱动程序的类。

- 1) `Connection getConnection(String url, String user, String password)` 方法：静态方法，建立与给定数据库 URL 的连接（`DriverManager` 试图从已注册的 JDBC 驱动程序集中选择一个适当的驱动程序）。

- 2) `DriverManager` 如何知道某种数据库已注册的？

例如：`oracle.jdbc.driver.OracleDriver` 类在 `Class.forName()` 的时候被载入 JVM；而 `OracleDriver` 是 JDBC 中 `Driver` 的子类，它被要求在静态初始化的时候要将自身驱动的信

息通过 DriverManager 的静态方法注册进去，这样 DriverManager 就知道应该如何通过 OracleDriver 去连接该数据库了。所以之后就可以通过 DriverManager 的另一个静态方法： getConnection() 来根据之前注册的驱动信息获取连接了：

```
Connection conn=DriverManager.getConnection("", "", "");
```

2.5 UUID

UUID 为通用唯一标识码 (Universally Unique Identifier) 对于大数据量的表来说，UUID 是存放 ID 最好的方式。

1) Java 提供的支持 (用法详见 2.8 案例)

UUID 类： UUID.randomUUID().toString(): 获得一个 36 位不重复的字符串。

2) Oracle 提供的支持 (用法详见 8.4 案例 step7)

函数 sys_guid(): 获取一个 32 位不重复的字符串。

2.6 案例：使用 JDBC 连接数据库，并操作 SQL 语句

```
/** 连接数据库一定要捕获异常的 */
Connection conn=null;//定义在 try 外面是用于在 finally 块中关闭它, 同时局部变量在使用前,一定要初始化!!
try{ /** 与数据库进行连接分为两步：1) 注册驱动：不同的数据库实现不尽相同，所以要使用不同数据库厂商提供的驱动包。连接不同数据库，传入的字符串不尽相同，但是目的相同，都是注册驱动。而对于驱动包路径，名字是固定的，基本上不会变的！2) 根据数据库的位置（路径）以及用户名和密码进行连接 */
    Class.forName("oracle.jdbc.driver.OracleDriver");
    /** 路径：不同数据库连接的路径写法不尽相同，Oracle 的写法：
    jdbc:oracle:thin:@HOST:DB_NAME
        其中 HOST 包含两部分：IP 地址和端口号；本机则使用 localhost 或 127.0.0.1 */
    conn=DriverManager.getConnection("jdbc:oracle:thin:@192.168.0.20:1521:tarena",
                                    "jsd1304","jsd1304");
    /** 使用 SQL 语句来操作数据库，若想执行 SQL 语句，我们需要使用一个专门处理 SQL 语句的类，这个类叫做 Statement */
    Statement state=conn.createStatement();
    /** user_tables 是 Oracle 用于存储当前用户创建的所有表的信息，其中一个字段叫做 table_name 用户保存的表名 */
    String sql="SELECT table_name FROM user_tables";
    /** 通过 Statement 执行查询语句，当查询完毕后，数据库会将查询结果返回，Statement 会将查询结果存储到 ResultSet 中 */
    ResultSet rs=state.executeQuery(sql);
    while(rs.next()){ //按行遍历，包含了是否有下一条记录的方法 hasnext()
        /** 按字段取值；整数参数：结果集的第几列。注意：这里从 1 开始，和 Java 对索引的习惯不同 */
        String tableName=rs.getString(1);
        System.out.println(tableName);
    }
    /** 底层一定会用到网络 socket 和流，但我们不用关心使用字符还是字节接收，都由 Statement 做了 */
}
```

```

        rs.close();    state.close();
    }catch(Exception e){    e.printStackTrace();
    }finally{ if(conn!=null){    try {    conn.close();    } catch (SQLException e) {
                e.printStackTrace();    }    }    }

```

◆ 注意事项：养成良好的编码习惯：所有 SQL 关键字用纯大写，其他内容用纯小写。

2.7 案例：通过 JDBC 创建表

```

Connection conn=null;
try{ //1 注册驱动
    Class.forName("oracle.jdbc.driver.OracleDriver");
//2 打开连接，支持 import java.sql.*，但全导入较耗费性能
conn=DriverManager.getConnection(
    "jdbc:oracle:thin:@192.168.0.20:1521:tarena","jsd1304","jsd1304");
//3 创建用于执行 SQL 语句的 Statement
Statement state=conn.createStatement();
//创建建表语句
String sql="CREATE TABLE Student_chang(
    id varchar2(36) PRIMARY KEY," + "name varchar2(30)," +
    "age number(2," + "sex varchar2(2)" + ")";
//execute()方法详见 2.2 节
if(!state.execute(sql)){  System.out.println("创建表成功！ ");
} else{  System.out.println("创建失败！ ");  }
state.close();
}catch (Exception e){  e.printStackTrace();
}finally{ if(conn!=null){    try {    conn.close();    } catch (SQLException e) {
                e.printStackTrace();    }    }    }

```

2.8 案例：使用 JDBC 向表中插入数据

```

Connection conn=null;
try{ Class.forName("oracle.jdbc.driver.OracleDriver");
conn=DriverManager.getConnection(
    "jdbc:oracle:thin:@192.168.0.20:1521:tarena","jsd1304","jsd1304");
Statement state=conn.createStatement();
//UUID 详见 2.5
String uuid=UUID.randomUUID().toString();  System.out.println(uuid);
String sql="INSERT INTO Student_chang VALUES("+uuid +"','Chang',22,'1')";
//或 String sql="INSERT INTO Student_chang VALUES(sys_guid(),'chang',23,'1')";
//判断 insert 语句是否成功，看返回值是否大于 0， executeUpdate 方法详见 2.2
if(state.executeUpdate(sql)>0){  System.out.println("插入数据成功");  }
state.close();
}catch(Exception e){  e.printStackTrace();
}finally{ if(conn!=null){    try {    conn.close();    } catch (SQLException e) {
                e.printStackTrace();    }    }    }

```

2.9 遍历 Student_chang 表

```
Connection conn=null;
try{ Class.forName("oracle.jdbc.driver.OracleDriver");
    conn=DriverManager.getConnection(
        "jdbc:oracle:thin:@192.168.0.20:1521:tarena","jsd1304","jsd1304");
    Statement state=conn.createStatement();
    String sql="SELECT * FROM Student_chang";
    ResultSet rs=state.executeQuery(sql);
    while(rs.next()) { String id=rs.getString(1);
        String name=rs.getString("name");//不知第几列也可写列名
        int age=rs.getInt("age");
        String sex=rs.getString(4).equals("1")?"男":"女";
        System.out.println(id+","+name+","+age+","+sex);
    }
    rs.close(); state.close();
} catch(Exception e){ e.printStackTrace();
} finally{ if(conn!=null){ try { conn.close(); } catch (SQLException e) {
        e.printStackTrace(); } } }
```

四十二、JDBC 核心 API：PreparedStatement

3.1 Statement 的缺点

- 1) 用 Statement 操作时代码的可读性和可维护性差，编写 SQL 语句复杂。
- 2) Statement 操作 SQL 语句，每执行一次都要对传入的语句编译一次，效率比较差。
- 3) 不安全可能出现 SQL 注入攻击，详见 9.6 案例 step3。
- 4) 扩展：XSS 攻击、html 代码注入攻击、struts2 OGNL 存在可以远程执行底层操作系统命令的漏洞。

3.2 PreparedStatement 的优点

- 1) PreparedStatement 实例包含已编译的 SQL 语句。包含于 PreparedStatement 对象中的 SQL 语句可具有一个或多个 IN 参数。IN 参数的值在 SQL 语句创建时未被指定。该语句为每个 IN 参数保留一个问号（“？”）作为占位符，不考虑类型。每个问号的值必须在该语句执行之前，通过适当的 setString、setInt、setDouble……等方法来提供。
 - 2) 由于 PreparedStatement 对象已预编译过，所以其执行速度要快于 Statement 对象。因此，多次执行的 SQL 语句经常创建为 PreparedStatement 对象，以提高效率。
 - 3) PreparedStatement 继承于 Statement，其中三种方法：execute、executeQuery、executeUpdate 都已被更改为不再需要参数了。因为我们在获取 PreparedStatement 时已经将 SQL 语句传入了。所以执行就可以，不需要再传入 SQL。
 - 4) PreparedStatement 可以进行批量处理。
 - 5) 可以防止 SQL 注入攻击。
- ◆ 注意事项：
- ❖ 使用预编译语句，你传入的任何内容就不会和原来的语句发生任何匹配的关系，只要全使用预编译语句，你就不用对传入的数据作任何的过滤。
 - ❖ 对一个表只作一个操作用 PreparedStatement，效率高、方便
 - ❖ 对表进行 2 种及以上的操作用 Statement。

3.3 PreparedStatement 的常用方法

- 1) boolean execute(): 可以是任何种类的 SQL 语句。一些预处理过的语句返回多个结果，execute 方法处理这些复杂的语句，executeQuery 和 executeUpdate 处理形式更简单的语句。如果第一个结果是 ResultSet 对象，则返回 true；如果第一个结果是更新计数或者没有结果，则返回 false。
- 2) ResultSet executeQuery(): 在此 PreparedStatement 对象中执行 SQL 查询，并返回该查询生成的 ResultSet 对象。
- 3) int executeUpdate(): 在此 PreparedStatement 对象中执行 SQL 语句，该语句必须是一个 DML 语句，比如 INSERT、UPDATE 或 DELETE 语句；或者是无返回内容的语句，比如 DDL 语句。
- 4) void addBatch(): 将一组参数添加到此 PreparedStatement 对象的批处理命令中。其他批处理方法，见第六章。
- 5) void setObject(int parameterIndex, Object x): 第一个参数用于设置“？”中的值，且从 1 开始！该方法可将任意给定的参数类型转换为所对应的 SQL 类型。类似 setInt、setString 等。

3.4 案例详见第五章 StudentDAO 类

LICHOOL

四十三、Connection 封装

可建立一个 DBUtils 类，用于对数据库连接的封装。

```
private static String driver;    private static String url;
private static String user;      private static String password;
static { //此处详细说明见 5.4 案例 step2
    try { Properties props = new Properties(); //Properties 类详见 5.3
        props.load(DBUtils.class.getClassLoader().getResourceAsStream(
            "db.properties"));
        driver = props.getProperty("driver");
        url = props.getProperty("url");
        user = props.getProperty("user");
        password = props.getProperty("password");
        Class.forName(driver); //注意这里没有引号！
    } catch (IOException e) {    e.printStackTrace();
    } catch (ClassNotFoundException e) {    e.printStackTrace();    }
}
/**封装连接操作，供其他类继承，再次使用数据库连接的时候，可以直接调用
openConnection 获取数据连接，其中 url、user、pwd 通过上面的.properties 文件加载 */
protected static Connection getConnection() throws SQLException{
    return DriverManager.getConnection(url, user, pwd);
}
/** 封装关闭操作，供其他类继承，关闭连接时调用该方法 */
protected static void closeConnection(Connection conn){
    if(conn!=null){
        try{ conn.close(); } catch (SQLException e) {    }
    } //catch 块内容可不写，即所谓的“安静”的关闭连接
}
```

四十四、DAO

LICHOO

5.1 持久类封装

对象关系映射（ORM）使用描述对象和数据库之间映射的元数据，将 Java 程序中的对象自动持久化到关系数据库中。1) 表和类对应。2) 表中的字段和类的属性对应。3) 记录和对象对应。

5.2 DAO 层

1) DAO：数据连接对象（DataAccessObjects）

2) 作用：将数据库中的数据转化为 Java 的对象并返回（即读数据），将 Java 的对象转化为数据库中表的一条数据（即写数据）。

3) Java 对象在这里就是所谓的实体 entity，DAO 要达到的目的：对数据库数据的操作面向对象化。

4) 实体：用 Java 中的对象去描述数据库中的某表中的某一条记录。

比如：Student 表有字段 id、name、age、sex，则对应的 Java 类中有 Student 类，属性有 id、name、age、sex

5) 实体类：用于对应数据库中的表。通常实体类的名字和数据库中表的名字一致。

◆ 注意事项：实体类代表表，属性代表字段，对象代表一条数据。

5.3 Properties 类

用于读取 “.properties” 文本文件的类，导入 java.util.Properties 包。

1) “.properties” 文件是一个纯文本文件，里面定义的内容格式有要求，必须是 key=value 的形式，并且以行为单位。一行只记录一条数据！

2) Properties 类可以方便的读取 properties 文件，并将内容以类似 HashMap 的形式进行读取。

3) db.properties 文件里的内容如下：

```
jdbc.driver=oracle.jdbc.driver.OracleDriver  
jdbc.url=jdbc:oracle:thin:@192.168.0.20:1521:tarena  
jdbc.user=jsd1304  
jdbc.pwd=jsd1304
```

◆ 注意事项：读取的都是字符串！不用写双引号，无空格！

4) getProperty(String key) 方法：该方法可以从 properties 文件中获取数据，如：jdbc.driver=oracle.jdbc.driver.OracleDriver。获取方式是将 jdbc.driver 以 key 作为参数调用方法。返回的就是等号右面的值 oracle.jdbc.driver.OracleDriver 了。

5.4 案例：注册系统

step1：Student 实体类

```
public class Student implements Serializable {  
    private static final long serialVersionUID=1L;  
    private String id;    private String name;    private int age;    private String sex;  
    .....各自的 get/set 方法  
}
```

◆ 注意事项：该类描述数据库中 Student 表，其每一个实例都可以代表 Student 表的一行数据。通常情况下实体都是可以序列化的！

step2: BaseDAO 父类 (基础类, 提供所有 DAO 都需要具备的特性)

```
private static Properties properties=new Properties();
private static String driver="";://"oracle.jdbc.driver.OracleDriver";
private static String url="";://"jdbc:oracle:thin:@192.168.0.20:1521:tarena";
private static String user="";://"jsd1304";
private static String pwd="";://"jsd1304";
/** 在静态初始化中注册驱动, 驱动不需要重复注册, 所以静态初始化最合适注册驱动
*/
static{    try {/** 加载配置文件, 读取配置信息 db.properties 中的内容见 5.3 */
    properties.load(BaseDAO.class.getClassLoader()
        .getResourceAsStream("day01pm/dao/db.properties"));
    System.out.println(properties.getProperty("jdbc.driver")); //可输出检查一下
    /** 获取值 */
    driver=properties.getProperty("jdbc.driver");
    url=properties.getProperty("jdbc.url");
    user=properties.getProperty("jdbc.user");
    pwd=properties.getProperty("jdbc.pwd");
    //加载驱动, 并注册到 DriverManager, 详见 2.4
    Class.forName(driver); //反射机制
} catch (Exception e) { e.printStackTrace();
    throw new RuntimeException(e); //若注册失败, 我们要通知调用者 }
}
/** 获取数据库连接对象 Connection 让外界去捕获异常, 连接不上数据库该怎么办?
*/
protected static Connection getConnection() throws SQLException{
    return DriverManager.getConnection(url,user,pwd);
}
/** 将给定的数据库连接关闭 */
protected static void closeConnection(Connection conn){
    if(conn!=null){ try { conn.close(); } catch (SQLException e) { //catch 块可不写内容
        e.printStackTrace(); } }
}
}
```

step3: StudentDAO 类 (用于操作数据库 Student 表) 并继承 BaseDAO

```
public Student findStudentByName(String name){
Connection conn=null;
try{//    Class.forName("oracle.jdbc.driver.OracleDriver");
//    conn=DriverManager.getConnection(
//
"jdbc:oracle:thin:@192.168.0.20:1521:tarena","jsd1304","jsd1304");
//通过父类 BaseDAO 的 getConnection()方法获取数据库连接
conn=getConnection();
//    Statement state=conn.createStatement(); //使用 Statement
//    String sql="SELECT * FROM student_chang WHERE name='"+name+"'";
String sql="SELECT * FROM student_chang WHERE name=?";
PreparedStatement state=conn.prepareStatement(sql); //使用 PreparedStatement
state.setString(1, name); //给第一个问号赋值
}
}
```

```

    /**
     * 根据用户名查询该用户信息，并将这条数据转化为一个 Student 对象并返回 */
    // ResultSet rs=state.executeQuery(sql); //使用 Statement 时
    ResultSet rs=state.executeQuery(); //使用 PreparedStatement 时，不需要参数了
    if(rs.next()){
        Student student=new Student();
        student.setId(rs.getString("id"));
        student.setName(rs.getString("name"));
        student.setAge(rs.getInt("age"));
        student.setSex(rs.getString("sex"));
        return student;
    }
    }catch(Exception e){
        e.printStackTrace();
    }finally{
        // if(conn!=null){
        try {
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        // 调用父类的关闭连接方法
        closeConnection(conn);
    }
    return null;
}

/**
 * 持久化 Student 对象，即将 Student 对象的数据保存到数据库中 */
public boolean saveStudent(Student student){
    Connection conn=null;
    try{ //Class.forName("oracle.jdbc.driver.OracleDriver");
        // conn=DriverManager.getConnection(
        //     "jdbc:oracle:thin:@192.168.0.20:1521:tarena","jsd1304","jsd1304");
        // 通过父类 BaseDAO 的 getConnection() 方法获取数据库连接
        conn=getConnection();
        // Statement state=conn.createStatement();
        // String sql="INSERT INTO student_chang VALUES(" +
        //             "sys_guid()," +
        //             ""+student.getName()+"+"+
        //             student.getAge()+"+"+
        //             ""+student.getSex()+"+"+
        //             ")";
        // 预编译的 SQL，将 SQL 中变化的内容用“？”代替，然后通过 PreparedStatement 执行该 SQL 时用给定的参数代替？，来达到插入不同数据的目的，预编译 SQL 更像是一个格式或者模版 */
        String sql="INSERT INTO student VALUES(sys_guid(),?, ?, ?)"; //问号上不考虑类型，就是占位
        PreparedStatement state=conn.prepareStatement(sql);
        state.setString(1, student.getName()); //将第一个问号替换为学生的姓名
        state.setInt(2, student.getAge());
        state.setString(3, student.getSex());
        if(state.executeUpdate(>0)){
            return true;
        } //使用 PreparedStatement
        // if(state.executeUpdate(sql)>0){
        //     return true;
        // } //使用 Statement
    }catch(Exception e){
        e.printStackTrace();
    }finally{
        // if(conn!=null){
        try {
            conn.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        // 调用父类的关闭连接方法
    }
}

```

```
    closeConnection(conn);          }
    return false;                  }
```

step4: StudentService 类 (业务逻辑类)

```
private StudentDAO studentDAO=new StudentDAO();
public void reg(String name,int age,String sex){
    /** 必要的验证 */
    if(name==null||"".equals(name)){ System.out.println("名字不能为空! ");
    }else if(age<1||age>99){ System.out.println("年龄只能在 1—99 之间! ");
    }else if !"1".equals(sex)&&"0".equals(sex)){System.out.println("性别只能为 0 或 1!");
    }else if(studentDAO.findStudentByName(name)!=null){
        System.out.println("该用户已存在! ");
    }else{//1 将用户输入的信息转化为一个 Student 对象
        Student student=new Student(); student.setName(name);
        student.setAge(age); student.setSex(sex);
        //2 将该对象交给 DAO 进行持久化 3 根据保存结果通知用户
        if(studentDAO.saveStudent(student)){      System.out.println("注册成功! ");
        }else{      System.out.println("注册失败! ");      }
    }
    public void findStudentByName(String name){
        //对用户输入的信息进行必要的判断,null 和空字符串
        if(name!=null && !"".equals(name)){
            //向 DAO 获取学生信息
            Student student=studentDAO.findStudentByName(name);
            if(student!=null){
                System.out.println("学生: "+student.getName()+"年龄: "+student.getAge());
            }else{      System.out.println("查无此人! ");      }
        }
    }
}
```

step5: 测试类

```
String studentName="chang";
StudentService service=new StudentService();
service.findStudentByName(studentName);
String studentName1="changyb";
int age=16;
String sex="1";
StudentService service1=new StudentService();
service1.reg(studentName1, age, sex);
```

四十五、批处理

LICHOO

6.1 批处理的优点

一个批处理是被发送到数据库以作为单个单元执行的一组更新语句。这降低了应用程序和数据库之间的网络调用。相比单个 SQL 语句的处理，处理一个批处理中的多个 SQL 语句是一种更为有效的方式。

6.2 JDBC 批处理 API

- 1) `addBatch(String sql)`: Statement 类的方法，“多次调用”该方法可以将给定的“多条”SQL 语句添加到 Statement 对象的命令列表中缓存（每调用一次缓存一条 SQL）。
- 2) `addBatch()`: PreparedStatement 类的方法，“多次调用”该方法可以将“多条”预编译的 SQL 语句添加到 PreparedStatement 对象的命令列表中缓存（每调用一次缓存一条 SQL）。
- 3) `executeBatch()`: 把 Statement 对象或 PreparedStatement 对象命令列表中缓存的所有 SQL 语句一次性提交给数据库进行处理。这样可以有效的减少网络通信带来的性能消耗。
- 4) `clearBatch()`: 清空当前 SQL 命令列表中缓存的所有 SQL 语句。

6.3 案例：详见 8.4 案例 step7

四十六、事务处理

LICHOO

7.1 事务特性 ACID

- 1) 原子性 (atomicity): 事务必须是原子工作单元; 对其数据修改, 要么全都执行, 要么全都不执行。
- 2) 一致性 (consistency): 事务在完成时, 必须使所有的数据都保持一致状态。
- 3) 隔离性 (isolation): 由并发事务所作的修改必须与任何其他并发事务所作的修改隔离。
- 4) 持久性 (durability): 事务完成之后, 它对于系统的影响是永久性的。

7.2 JDBC 中对事务的支持 (API)

- 1) JDBC 默认是每执行一条 SQL 语句都会提交事务。这对于批量处理数据来说性能开效果大。而且不满足事务本该管理的原则。
- 2) `conn.setAutoCommit(false);`//不自动提交
- 3) `conn.commit();`//提交事务
- 4) `conn.rollback();`//回滚事务

四十七、DAO 事务封装

LICHOO

8.1 ThreadLocal 原理

1) Java 提供了一个类 ThreadLocal: 用于实现线程内的数据共享, 即对于相同的程序代码, 多个模块(方法)在同一个线程中运行时要共享一份数据, 而在另外线程中运行时又共享另外一份数据。

2) 数据共享是根据线程区分的。不同线程间不共享数据。

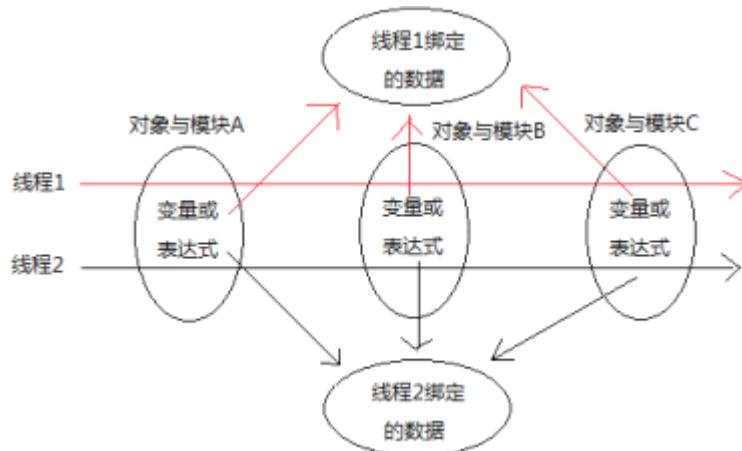
3) 原理分析:

ThreadLocal 内部很简单: 就是维护者一个 HashMap, 其中 key 存放的是每一个线程, value 存放这个线程在不同模块间要共享的数据。每一个 ThreadLocal 实例, 只能保存一个线程的一个共享数据, 不同线程看到的应该是同一个 ThreadLocal 实例, 但是获取的是不同的数据, 因为线程间不共享。每个线程在不同模块中共享数据。比如: BaseDAO 中, getConnection()、commi()、rollback()、closeConnection() 这些方法, 都共享同一个 Connection 实例。

```
HashMap hashMap = new HashMap();
void set(Object obj){    hashMap.put(Thread.currentThread(),obj);    }
Object get(){        return hashMap.get(Thread.currentThread());    }
```

4) 多线程共享数据: 可用单例模式, 但要加锁。

8.2 原理图



8.3 ThreadLocal 核心 API

1) public T get(): 用于获取线程共享的数据。

2) public void set(T value): 用于保存线程共享的数据。

3) public void remove(): 移除线程中保存的数据。

◆ 注意事项: 使用 remove() 则 key 也没了, 使用 set(null) 则 key 还保留。

8.4 案例: 登录系统 (使用 ThreadLocal 实现连接共享)

step1: 添加实体类 UserInfo, 并设置相应的 get/set 方法

```
private static final long serialVersionUID=1L;
private String id;    private String name;    private String password;
private int age;     private String sex;      private String email;
```

step2: 修改 5.2 案例中的 BaseDAO 中的 getConnection()方法

```
protected static Connection getConnection() throws SQLException{  
    /** 当一个线程调用该方法要获取连接时，我们先检查之前这个线程是否已经获取过一个连接了，若有就不再创建了 */  
    Connection conn=localConn.get();//get 用于获取线程共享的数据  
    if(conn==null){//若是空的，说明这个线程第一次获取连接  
        //创建和数据库的连接  
        conn=DriverManager.getConnection(url,user, pwd);  
        //将创建出来的连接放入线程共享中，set 方法用于保存线程共享的数据  
        localConn.set(conn);  
    }  
    return conn;  
}
```

step3: 修改 5.2 案例中的 BaseDAO 中的 closeConnection()方法

```
protected static void closeConnection(){  
    try{ Connection conn=localConn.get();  
        if(conn!=null){ conn.close();//关闭连接  
            //连接关闭后，这个连接就没有存在的意义了，应该从线程共享中将其删除！  
            localConn.remove();//key 也没了，或者 localConn.set(null);key 还保留  
        }  
    }catch(Exception e){ e.printStackTrace(); }  
}
```

step4: 添加 begin 方法: 开始事务

```
protected static void begin(){  
    try{ Connection conn=localConn.get();  
        if(conn!=null){ conn.setAutoCommit(false);//取消事务的自动提交 }  
    }catch(Exception e){ e.printStackTrace(); }  
}
```

step5: 添加 commit 方法: 提交数据到数据库。

```
protected static void commit(){  
    /** 先去线程共享中看看当前调用该方法的线程是否有共享过 Connection，若有，就提交事务。不需要 UserInfoDAO 再告诉我提交哪个 conn，即不用再传参数 */  
    Connection conn=localConn.get();  
    if(conn!=null){ try { conn.commit(); } catch (SQLException e) {  
        e.printStackTrace(); } }  
}
```

step6: 添加 rollback 方法，回滚事务。

```
protected static void rollback(){  
    /** 不需要 UserInfoDAO 再告诉我回滚哪个 conn，即不用再传参数 */  
    Connection conn=localConn.get();  
    if(conn!=null){ try { conn.rollback(); } catch (SQLException e) {  
        e.printStackTrace(); } }  
}
```

step7: 添加 UserInfoDAO 类，并继承 BaseDAO

```
private static final String INSERT="INSERT INTO userinfo_chang(" +  
    "id,name,password,age,sex,email) VALUES(sys_guid(),?, ?, ?, ?, ?);  
/** 保存给定的所有用户信息 */  
public boolean save(List<UserInfo> userInfos){  
    Connection conn=null;
```

```

try{ conn=getConnection();      conn.setAutoCommit(false);//禁止自动提交
    PreparedStatement state=conn.prepareStatement(INSERT);
    long start=System.currentTimeMillis();
    //    for(int i=0;i<50;i++){//可快速插入 50 条记录
    //        state.setString(1,"test"+ i);          state.setString(2,"12345"+ i);
    //        state.setInt(3,22);                  state.setString(4,"1");
    //        state.setString(5,"test"+ i +"@mail.com");
    //        //state.executeUpdate();//每执行一次都提交一次事务，涉及到硬件的写操作
    //        state.addBatch();//而 Statement 需要把 sql 语句写入参数列表
    //    }
    for(UserInfo userinfo:userInfos){//读取集合中的记录，并存入数据库，集合不能为空，为空则不执行了。
        state.setString(1,userinfo.getName());
        state.setString(2,userinfo.getPassword());
        state.setInt(3,userinfo.getAge()); state.setString(4,userinfo.getSex());
        state.setString(5,userinfo.getEmail());
        //state.executeUpdate();//每执行一次都提交一次事务，涉及到硬件的写操作
        state.addBatch();//而 Statement 需要把 sql 语句写入参数列表
    }
    state.executeBatch();//批处理执行之前缓存的所有 SQL
    //conn.commit();
    commit();    long end=System.currentTimeMillis();
    System.out.println("耗时：" +(end-start)+"毫秒");
    return true;//返回 true，告知调用者保存成功
} catch(Exception e){
    e.printStackTrace();
    //    if(conn!=null){//若执行保存过程中出错，要回滚事务
    //        //conn.rollback();
    //    }
    rollback();
} finally{ //closeConnection(conn);
    closeConnection();
    return false;
}
/** 创建表 */
public void createTable(){
    Connection conn=null;
    try{ conn=getConnection();
        Statement state=conn.createStatement();
        String sql="CREATE TABLE userinfo_chang(" +
                    "id VARCHAR2(36) PRIMARY KEY,name VARCHAR2(30)," +
                    "password VARCHAR2(50),age NUMBER(2)," +
                    "sex VARCHAR2(2),email VARCHAR2(50))";
        if(!state.execute(sql)){ System.out.println("创建完毕"); }
    }

```

```
    } catch(Exception e){    e.printStackTrace();
    } finally{ if(conn!=null){      closeConnection(conn);      }
}
```

LJC00

四十八、分页查询

LICHOO

9.1 分页查询的基本原理

```
Connection conn = getConnection();
PreparedStatement state= conn.prepareStatement(sql);
int start = rowsPerPage * (page -1)+1;//每页的开始
int end = start + rowsPerPage;//每页的结束
state.setInt(1,end); state.setInt(2,start);
ResultSet rs =state.executeQuery();
List<Service> list = new ArrayList<Service>();
while(rs.next()) { list.add(toService(rs)); } //把每条数据转成对象后添加
```

9.2 为何使用分页查询

1) 每次只向数据库要求一页的数据量，频繁访问数据库，内存压力小，适合大数据量。
2) 数据库中的表可能会存储若干数据，若我们一次性将所有数据获取显然是不理智的。这可能产生很多坏处，比如占用内存过大，而对于用户而言，数据过于多也不利于查看等。为此，我们可以将数据分批次的检索出来。既可以节省资源，也利于用户查看。不同数据库对分页支持的 SQL 语句不尽相同。所以，使用不同的数据库，我们要适应当前数据库对分页语句的定义。当然，hibernate 屏蔽了数据库分页的差异。可以让我们很方便的使用统一方式进行分页查询。

9.3 Oracle 分页查询 SQL 语句

1) Oracle 中的分页使用了一个字段 rownum，使用该字段对查询的行数进行限制，从而达到获取某一区间的数据。实现分页查询。

2) rownum 它是 oracle 系统顺序分配的查询返回的行的编号，查询到第一条数据 rownum 返回第一行分配的行号 1，查询到第二条数据 rownum 返回第二行分配的行号 2，依此类推。也就是说行号是查询到数据后才产生的。

3) 起始、结束位置计算：

每页的起始位置 start=每页显示的记录数 rows × (当前要请求的页数 page-1) +1

每页的结束位置 end=每页的起始位置 start + 每页显示的记录数 rows (或 end=page * rows 含头不含尾 java 的一种习惯，不是必须的算法)。比如：

```
select
id,account_id,host,user_name,login_password,status,create_date,pause_date,close_date,cost_id
from (select id,account_id,host,user_name,login_password,status,create_date,pause_date,
close_date,cost_id,rownum r from service where rownum <?) where r >=?
```

◆ 注意事项：子查询不能写 *，写了不会有 rownum，记得给 rownum 起列别名！

9.4 MySQL 分页查询 SQL 语句

```
select * from table limit start,pageSize
```

◆ 注意事项：start 从 0 开始，pageSize 为每页的条数。

9.5 “假” 分页

1) 一次性把数据全部取出来放在缓存中，根据用户要看的页数（page）和每页记录数

(pageSize), 计算把哪些数据输出显示。

- 2) 只访问数据库一次, 第一次取数比较慢, 以后每页都从缓存中取, 比较快。
- 3) 比较适合小数据量, 如果数据量大, 对内存压力比较大。
- 4) 一次性将数据库数据读入结果集, 每次查看指定的页时, 要求结果集的指针能够跳到指定的行, 即指针能够跳到整个结果集的任一位置。

9.6 案例：分页查询

step1：在 8.4 案例中 step7 的 UserInfoDAO 类添加分页查询方法和根据用户名、密码获取用户信息方法

```
/** 分页查询用户信息。page: 第几页。rows: 每页显示的条数 */
public List<UserInfo> findPaging(int page,int rows){
    try{/** 根据页数和每页显示的条数, 计算起始行号和结束行号 */
        int start=(page-1)*rows+1;
        int end=page*rows;//含头不含尾 java 的一种习惯, 不是必须的算法
        /** 执行分页查询的 Oracle 的 SQL 语句, 这里我们要进行两次查询, 第一次先将后区间定位, 第二次再将前区间定位。从而获取区间中的数据。 */
        //子查询不能写 * , 写了不会有 rownum, 注意 r 后有一个空格
        String sql="SELECT * FROM " +
                   "(SELECT id,name,password,age,sex,email,rownum r " +
                   "FROM userinfo_chang WHERE rownum <?) WHERE r >=? ";
        //获取预编译 Statement
        //Connection conn=getConnection();
        //PreparedStatement state=conn.prepareStatement(sql);
        //现在不需要关 conn 了, 由 BaseDAO 去关 (统一线程数据共享)
        PreparedStatement state=getConnection().prepareStatement(sql);//上面二合一
        state.setInt(1, end);//设置后区间 (第一个问号)
        state.setInt(2, start);//设置前区间 (第二个问号)
        ResultSet rs=state.executeQuery();//获取结果集
        ArrayList<UserInfo> userinfos=new ArrayList<UserInfo>();
        /** 这里我们可以在循环外面创建一个引用变量, 从而节省循环带来的不必要的内存开销。但是绝对不能在循环外面创建一个对象, 在循环里重复设置内容, 否则看到的效果就是虽然查询出来了若干数据, 但是集合中保存的却是最后一条数据, 而且保存了若干次而已 (所有引用指向最后一个对象, 所以内容相同且为最后一条数据的)。 */
        UserInfo userinfo=null;
        while(rs.next()){//引用变量可同一个, 但对象不能同一个 (即定义在循环外)
            userinfo=new UserInfo(); userinfo.setId(rs.getString("id"));
            userinfo.setName(rs.getString("name"));
            userinfo.setPassword(rs.getString("password"));
            userinfo.setAge(rs.getInt("age")); userinfo.setSex(rs.getString("sex"));
            userinfo.setEmail(rs.getString("email")); userinfos.add(userinfo);
        }
        rs.close(); state.close(); return userinfos;
    }catch(Exception e){ e.printStackTrace(); throw new RuntimeException(e);
    }finally{ closeConnection(); }
    //return null;
}
```

```

//当 catch 块写了 throw new RuntimeException(e); 就不用写 return 语句了，因为执行不到
/** 根据用户名、密码获取用户信息 */
public UserInfo findUserInfoByNameAndPwd(String name, String password) {
    try {
        String sql = "SELECT * FROM userinfo_chang "
                    + "WHERE name=" + name + " AND password=" + password + " ";
        Statement state = getConnection().createStatement();
        ResultSet rs = state.executeQuery(sql);
        UserInfo userinfo = null;
        if (rs.next()) {
            userinfo = new UserInfo();
            userinfo.setId(rs.getString("id"));
            userinfo.setName(rs.getString("name"));
            userinfo.setPassword(rs.getString("password"));
            userinfo.setAge(rs.getInt("age"));
            userinfo.setSex(rs.getString("sex"));
            userinfo.setEmail(rs.getString("email"));
        }
        rs.close();
        state.close();
        return userinfo;
    } catch (Exception e) {
        e.printStackTrace();
        throw new RuntimeException(e);
    } finally {
        closeConnection();
    }
}

```

step2: main 方法测试

```

public static void main(String[] args) {
    UserInfoDAO dao = new UserInfoDAO();
    // dao.createTable(); //建表
    // dao.save(new ArrayList()); //快速插入数据
    List<UserInfo> list = dao.findPaging(2, 10);
    for (UserInfo user : list) {
        System.out.println(user.getName());
    }
}

```

step3: main 方法添加如下语句，测试 SQL 注入

```

/** SQL 注入攻击：对于 SQL 语句 "SELECT * FROM userinfo WHERE name='chang' AND password='123456'" 
把密码修改为当前形式：AND password='1234' OR '1'='1' 即拼写部分为：
1234' OR '1'='1' 呢？

（第 2 个 1 最后无单引号，拼的字符串里有，第一个密码同理）则发现也可登录成功！
*/
// UserInfo userinfo = new UserInfo(); //添加用户
// userinfo.setName("chang");
// userinfo.setPassword("123456");
// List<UserInfo> list = new ArrayList<UserInfo>();
// list.add(userinfo);
// dao.save(list);
UserInfo user = dao.findUserInfoByNameAndPwd("chang", "123456");
// dao.findUserInfoByNameAndPwd("chang", "1234' OR '1'='1"); //SQL 注入攻击，则发现也可登录成功！
if (user != null) {
    System.out.println("欢迎你：" + user.getName());
} else {
    System.out.println("登录失败！");
}

```

5 XML 学习笔记

LICHO0

四十九、基本语法

1.1 XML 介绍

- 1) XML 是可扩展标记语言 (EXtensible Markup Language)。
- 2) XML 是独立于软件和硬件的信息传输工具。
- 3) XML 是以文本的形式存在于一个文本文件中的，一般该文件的后缀名就是 “.xml”，例如：user.xml。
- 4) XML 的设计宗旨是传输信息 (尤其是结构比较复杂的数据)，而不是显示数据。
- 5) XML 可以描绘树状结构的数据。因为这个特点，除了传输数据外，更多时候我们使用 XML 作为配置文件。
- 6) XML 是一种标记语言，很类似 HTML。
- 7) XML 标签没有预先定义，需要自行定义标签。
- 8) XML 被设计为具有自我描述性。
- 9) XML 是 W3C 推荐的标准 (W3C, 万维网联盟, World Wide Web Consortium, 这个建立于 1994 年的组织，其宗旨是通过促进通用协议的发展并确保其通用型，以激发 web 世界的全部潜能)。
- 10) XML 注释：<!--注释内容-->

1.2 XML 元素

- 1) XML 文档包含 XML 元素。
- 2) XML 元素指的是从开始标签 (包含) 到结束标签 (包含) 的部分。
- 3) 元素可包含其他元素 (标签嵌套使用)、文本或者两者的混合物。
- 4) 元素也可以拥有属性。

例如：XML 文件内容都是标签，<tag></tag>标签是成对出现的。

1.3 XML 属性

1) XML 元素可以在开始标签中包含属性(即属性是在前标签中定义的)，属性(Attribute)通常不是用于保存数据的，而是用于设定、描述标签的一些特征，是提供关于元素的额外(附加)信息的。属性通常提供不属于数据组成部分的信息，但是对需要处理这个元素的应用程序来说却很重要。

◆ 注意事项：

- ❖ 属性必须是属性名=属性值的形式。
- ❖ 元素可以包含元素，但是不能交叉使用！嵌套关系必须完整。

2) XML 属性的属性值必须使用引号，单引号 ‘’ 或双引号 “” 都可以！如果属性值本身包含双引号，那么有必要使用单引号包围它，或者可以使用实体引用。

例如：oracle user="chang" 的连接" id="sss">

3) 在标签名的后面可以定义若干个属性，每个属性间应该以空格隔开。

1.4 实体引用

为了解决属性值中使用 XML 中的特殊字符，我们可以使用类似的转义字符去描述。

字符	被替换为	转移字符（实体引用）
<		<
>		>
&		&
'		'
"		"

◆ 注意事项：

- ❖ 文本中也不能有特数字符。
- ❖ 实际上，在属性值中&和<是确认不合法的特殊字符（但>是合法的），必须要转义。其余的可以不用，但是更好的习惯是遇到这种在 XML 中有特殊含义的字符时都使用转义去表达。

1.5 CDATA 段

在某些情况下，我们在 xml 中要使用大量 XML 敏感的字符，而我们又不希望逐一的对其进行转移。这时候使用 CDATA 段是最理想的。

- 1) 语法格式：<! [CDATA [忽略检查的文本]] >
- 2) 在 CDATA 中将文本的内容写入，那么这段文本内容会被忽略检查，无论里面是否包含 XML 敏感内容，全部被当作普通的文本去看待。例如：

```
<content>
  <![CDATA[
    <script language="javascript">
      function sayhello(){    alert("hello!");    }
    </script>
  ]]>
</content>
```

- 3) 行业内交换数据时要求 xml 文件格式相同，所以需要大家遵守规范的 xml 文件格式，比如两份 xml 文件要有相同的元素嵌套关系、相同的属性定义、相同的元素顺序、元素出现相同的次数等。

如下为两份相同数据，但是结构不同的 xml 文件，无法交换数据

```

1 A学校的xml文件中
2 <计算机书籍>
3   <书名 isbn="1234">XML的前世今生</书名>
4   <价格>50</价格>
5   <简介>一本介绍XML的书</简介>
6   <作者>李毅</作者>
7 </计算机书籍>
8
9 B学校的xml文件中
10 <Computer_book>
11   <isbn>1234</isbn>
12   <bookname author="李毅">XML的前世今生</bookname>
13   <price>50</price>
14   <brief>一本介绍XML的书</brief>
15 </Computer_book>

```

1.6 DTD 声明元素

1) 在一个 DTD 中，元素通过元素声明来进行声明（用于声明和约束元素）。

- ◆ 注意事项：XML 只允许有一个根标记（根节点）。

2) DTD 声明元素语法：

```
<!ELEMENT 元素名 (元素内容) >
```

```
<!ELEMENT 元素名 元素类别>
```

- ◆ 注意事项：若使用 DTD 声明了元素（也可说使用了 DTD 验证），那么在 XML 中只能使用被声明过的元素，包括元素下的子元素。不能使用不在 DTD 中声明的元素了。

1.7 DTD 声明元素：声明空元素

即声明一个空标签，标签中不含有任何内容。

例如：`<!ELEMENT page EMPTY>`

对应：`<page/>`或`<page></page>`

- ◆ 注意事项：简写：`<page/>`通常 XML 中若没有后标记可以这样作，HTML 中被大量使用。前提是标签中没内容！

1.8 DTD 声明元素：含有 PCDATA

定义元素中的内容为文本内容（写什么都可以）。

1) 语法：`<!ELEMENT 元素名 (#PCDATA)>`

例如：`<!ELEMENT page (#PCDATA)>`

对应：`<page>chang</page>`或`<page></page>`

- ◆ 注意事项：PCDATA 是会被解析器解析的文本，这些文本将被解析器检查实体以及标记。

2) 这时候 `page` 标签中只能出现文本内容，但要注意，该文本内容是需要检查的，就是说不能出现 XML 敏感字符。

`<page>7<0</page>`这样不行！

`<page><![CDATA[7<0]]></page>`这样可以

1.9 DTD 声明元素：带有子元素（子元素列表）的元素

即约束标签中出现标签，或者说约束元素中有子元素。

1) 语法：`<!ELEMENT 元素名 (子元素 1, 子元素 2, …)>`

2) 在声明当前元素所包含的子元素时，我们要在下面声明出这些子元素！

例如：`<!ELEMENT jdbc (oracle,mysql)>`

```
<!ELEMENT oracle (#PCDATA)>
```

```
<!ELEMENT mysql (#PCDATA)>
```

上面的声明表示有一个叫 `jdbc` 的标签，它其中只能包含两个标签，分别是 `oracle` 和 `mysql`，而这两个标签也声明了，它们的内容可以是任意文本。

对应：`<jdbc>`

```
<oracle>chang</oracle>
```

```
<mysql>booooo</mysql>
```

```
</jdbc>
```

- ◆ 注意事项：`jdbc` 标签中 `oracle` 和 `mysql` 的出现顺序必须与 `jdbc` 声明的元素的顺序

一致！

LICHOOL

错误一：

```
<jdbc>
    <mysql>boooo</mysql>这样就不行！
    <oracle>chang</oracle>
</jdbc>
```

错误二：

```
<jdbc>
    <oracle>lalala</oracle>
    <mysql>oooooo</mysql>
    <mysql>oooooo</mysql>这样就不行！只能出现1次！
</jdbc>
```

1.10 DTD 声明元素：声明只出现一次的元素

1) 语法: <!ELEMENT 元素名 (子元素名称)>

例如: <!ELEMENT page (prev)>

对应: <page>

```
<prev>3</prev>
```

```
</page>
```

2) 上例声明了 prev 子元素必须出现一次，并且必须只在“page”元素中出现一次。

1.11 DTD 声明元素：声明可多次出现的元素

对于在子元素中出现的次数，DTD 也有规范，不支持具体数，但支持量词。

1) 子元素可以出现的量词：

?	0-1
*	0-多次
+	1-多次

例如: <!ELEMENT jdbc (oracle?,mysql+)>

◆ 注意事项：

- ❖ 使用了以上的量词的话，子元素出现顺序就不是必须一致了。
- ❖ 没用量词，则子元素必须出现且 1 次，且出现顺序必须与声明的元素的顺序一致！

1.12 DTD 声明元素：子元素只能是其中之一的情况

例如: <!ELEMENT sex (man|woman)>

```
<!ELEMENT man (#PCDATA)>
```

```
<!ELEMENT woman (#PCDATA)>
```

对应: <sex>

```
<woman>abc</woman><!-- 标签二选一 -->
```

```
</sex>
```

◆ 注意事项：与就是直接用“，”逗号，如<!ELEMENT sex (man,woman)>

1.13 DTD 声明元素：子元素可以是元素也可以是文本

定义标签中的子元素可以是元素也可以是文本。

例如：<!ELEMENT sex ANY>

1.14 DTD 声明元素：总结

类型名称	类型说明
CDATA	值为字符数据
(en1 en2 ...)	值为枚举列表中的一个值
ID	值为唯一的 ID
IDREF	值为另外一个元素的 ID
IDREFS	值为其他元素 ID 的列表
NMTOKEN	值为合法的 XML 名称
NMTOKENS	值为合法的 XML 名称列表
ENTITY	值是一个实体
ENTITIES	值是一个实体列表
NOTATION	此值是符号的名称
xml	值是一个预定义的 XML 值

1.15 DTD 中声明元素的属性

1) 在 DTD 中，属性通过 ATTLIST 来声明属性。

2) 声明属性的语法：<!ATTLIST 元素名称 属性名 属性类型 默认值>

例如：<!ATTLIST oracle user CDATA "user">

<oracle></oracle><!-- 默认值起作用 -->

<oracle user="user"></oracle><!-- 默认值不起作用 -->

◆ 注意事项：仅仅是定义元素中的内容为文本内容用#PCDATA，其他地方都用 CDATA

例如：<!ELEMENT package (#PCDATA)>

<!ELEMENT package color CDATA "fffffff">

1.16 属性类型

1) CDATA：文本内容。

2) ID：属性的值在 xml 中是唯一的（该属性值不可重复）。

3) 枚举 (en1 | en2…): 属性的值只能是列举的其中之一。

1.17 属性值的约束

1) #REQUIRED：当前这个属性必须在标签里出现（即要写出来），则此时默认值无效了。

例如：<!ATTLIST oracle user CDATA "user" #REQUIRED>

2) #FIXED：属性里的值为固定值（值要给出），或者值是其列举的其中之一。

例如：<!ATTLIST oracle user CDATA "user" #FIXED ("user"|"admin")>

3) 默认值"xxx"：若在默认值位置直接给定值，那么就是默认值。

例如: <!ATTLIST oracle id CDATA "sss">

1.18 DTD 命名空间介绍

1) 命名空间 (NameSpace), XML 文件允许自定义标记, 所以可能出现来自不同源 DTD 或 Schema 文件的同名标记, 为了区分这些标记, 就需要使用命名空间。

2) 命名空间的目的是有效的区分来自不同 DTD 的相同标记, 例如下例 xml 文件中使用了命名空间区分开“表格”和“桌子”。

例如: <html:table>
 <line><column>这是一个表格</column></line>
 </html:table>
 <product:table>
 <type>coffee table</type>
 <meterial>wood</meterial>
 </product:table>

五十、Schema 简介

2.1 Schema 的作用

因为 DTD 无法解决命名冲突问题, 所以出现了 Schema, 它是 DTD 的替代者。DTD 和 Schema 的功能都是用于描述 XML 结构的。

Schema: W3C 提出的一套用于约束 XML 元素的标准, 支持命名空间, 和 DTD 的作用一致。DTD 因为定义语法相对困难, 且不是标准的 XML 形式去描述定义的。而 Schema 本身就是 xml (所以也被称作是自描述的语言), 去约束另一个 xml 元素的内容相对 DTD 更易维护。

2.2 Schema 文件的扩展名 xsd

XML Schema Definition (简称 XSD, 遵循 W3C 标准)。

五十一、Java 解析 XML

LICHOO

3.1 Java 与 XML 共同点

有很多共同点，比如跨平台、与厂商无关，目前为止 Java 对 XML 的解析比其他语言更完善（Java 是支持 XML 最好的语言）。

3.2 Java 解析 XML 有两种方式

- 1) DOM：文本对象模型（Document Object Model）
- 2) SAX：基于 xml 的简单 API（Simple API For XML）
- 3) 嵌入式设备中常用 SAX 进行解析。例如 android 中就是使用 SAX 作为解析 xml 文件的工具的。Android 中还有一种叫做 Pull 解析。

3.3 JDOM/DOM4J

目前常用的 2 种解析 XML 文件的 API。

3.4 DOM 解析

解析 XML 是以树状结构进行解析的。DOM 在解析 XML 的时候会将整个 xml 内容解析出来，以 Element（元素）描绘每个节点和嵌套关系，并载入内存。

- 1) 关键字：树（Document）
- 2) 优点：把 xml 文件在内存中构造树型结构，可以遍历和修改节点，因为它知道节点的所有关系。
- 3) 缺点：因为解析时就将整个 xml 文件全部载入到内存，所以解析过长，内存开销大。

3.5 SAX 解析

解析 XML 是把 xml 文件作为输入流，触发标记开始，内容开始，标记结束等动作。

- 1) 关键字：流（Stream）
- 2) 优点：解析可以立即开始，速度快，没有内存压力。
- 3) 缺点：不能对节点做修改。

3.6 案例：使用 DOM4J 包的核心 API 解析 xml 文件

我们使用 DOM 工具，来自 DOM4J，非常流行的用于解析 xml 的 DOM 工具，还有一种常用的叫做 JDOM，解析步骤：

- 1) 创建用于解析 Xml 文件的读取器 SAXReader；
- 2) 使用 SAXReader 读取指定的输入流来解析 xml 文件；
- 3) 第 2 步的方法会返回一个 Document 对象，描述整个文档，通过该文档对象获取根标签（标记）Root；
- 4) 根据树的组成形式，逐一解析。

step1：创建 DBinfo 类，其中属性有

private String url;//连接的 url	private String driver;//连接的驱动
private String username;//数据库用户名	private String password;//数据库密码
private String dbName;//数据库连接名	private String attUser;//数据库标签的属性 user
.....各自对应的 get/set 方法	

step2：创建 XMLUtils 工具类用于解析 xml 文件（xml 文件内容见第五章）属性如下：

```
/** 定义常量，用于描述当前解析的 XML 中出现的标签名 */
private static final String ELEMENT_JDBC="jdbc";
private static final String ELEMENT_ORACLE="oracle";
private static final String ELEMENT_MYSQL="mysql";
private static final String ELEMENT_URL="url";
private static final String ELEMENT_DRIVER="driver";
private static final String ELEMENT_USERNAME="username";
private static final String ELEMENT_PASSWORD="password";
private static final String ATTRIBUTE_USER="user";
```

step3：在 XMLUtils 工具类中，添加 xmlToDBInfo(InputStream input)方法

```
/** 方法 xmlToDBInfo： 解析 xml 文件，将配置的所有数据库连接返回 */
```

```
public static List<DBInfo> xmlToDBInfo(InputStream input){
```

```
    /** 参数 InputStream input 说明：因为我们要使用 DOM 去解析 XML 文件，那么
```

一定会通过输入流去获取 Xml 文件中的数据，无论这个文件来自网络还是本地文件，我们在程序中都是通过输入流的形式读取的。 */

```
//1 使用 dom4j 需要导包： dom4j-1.6.1.jar
```

```
SAXReader reader=new SAXReader();
```

```
/** 解析 xml 前调用该方法！该方法默认值为 false，设置为 true 的作用是检查 xml 的合法性，验证 DTD。 */
```

```
reader.setValidation(false); //此处建议先 false，否则 DTD 稍有错误就将报错！
```

```
//2 导包： org.dom4j.Document
```

```
Document document;
```

```
try{    document =reader.read(input); //读取并解析文件
```

```
}catch(Exception e){    e.printStackTrace();
```

```
    throw new RuntimeException("数据读取错误！ ",e);        }
```

```
//3 导包： org.dom4j.Element
```

```
/** root.getName()用于获取当前标签的名字，这里应该是 jdbc */
```

```
Element root=document.getRootElement();
```

```
if(!"jdbc".equals(root.getName())){
```

```
    throw new RuntimeException("数据格式错误,根应该是:"+ELEMENT_JDBC);}
```

```
//4 逐一解析
```

```
/** root.elements();该方法获取 root 元素下的所有子元素并以集合的形式返回。
```

```
* root.element(String name);获取指定名字的子元素
```

```
* root.elements(String name);获取指定名字的所有子元素 */
```

```
List<Element> childList=root.elements();
```

```
/** 将 oracle 元素和 mysql 元素保存的内容存放到相应的 DBInfo 对象中，再将这些对象存入一个集合并返回，最终完成解析工作。 */
```

```
List<DBInfo> infos=new ArrayList<DBInfo>();
```

```
for(Element child:childList){
```

```
    //将 oracle 或 mysql 标签转化为 DBInfo 对象并存入集合 infos
```

```
    DBInfo info=toDBInfo(child);    infos.add(info);        }
```

```
return infos;    }
```

step4: 将标签内容转化为 DBInfo 对象，添加 toDBInfo(Element element)方法

```

private static DBInfo toDBInfo(Element element){
    DBInfo info=new DBInfo();
    //本
    Element urlElement=element.element(ELEMENT_URL);//现拿到标签，再拿标签内容
    String url=urlElement.getText();
    String url=element.elementText(ELEMENT_URL); //同上面两步，直接获取子元素文
    String driver=element.elementText(ELEMENT_DRIVER);
    String username=element.elementText(ELEMENT_USERNAME);
    String password=element.elementText(ELEMENT_PASSWORD);
    String attUser=element.attributeValue(ATTRIBUTE_USER); //获取标签的属性值
    String dbName=element.getName(); //获取当前标签的名字
    info.setAttUser(attUser);      info.setDbName(dbName);
    info.setDriver(driver);        info.setPassword(password);
    info.setUrl(url);            info.setUsername(username);
    return info;
}

```

step5: 测试

```

FileInputStream fis=new FileInputStream("src"+File.separator+
                                         "day03"+File.separator+"part2"+File.separator+"db_info.xml");
List<DBInfo> infos=XMLUtils.xmlToDBInfo(fis);
for(DBInfo info:infos){
    System.out.println("db_name: "+info.getDbName());
    System.out.println("url: "+info.getUrl());
    System.out.println("driver: "+info.getDriver());
    System.out.println("username: "+info.getUsername());
    System.out.println("attUser: "+info.getAttUser());
    System.out.println("");
}

```

3.7 案例：使用 DOM4J 包的核心 API 写入 xml 文件

将数据写入 xml 文件中的步骤：

- 1) 创建一个文档对象 Document。
- 2) 向文档对象中添加根标记。
- 3) 向根标记中按照需求添加子标记，最终构成 xml 的树状结构。
- 4) 将根标记写出（等同于将整个树写出）这样整个结构就转化为 xml 形式写出了。

step1: 在 XMLUtils 工具类中，添加 writeDBInfosToXml()方法

```

/** 将指定的一组数据以 XML 的格式通过输出流写出 */
public static void writeDBInfosToXml(
    OutputStream out,List<DBInfo> infos) throws Exception{
    //1 创建文档对象
    Document document=DocumentHelper.createDocument();
    //2 添加根标记
    /** Document 的 addElement(String name)方法会向文档对象添加名为参数的根标

```

记，该方法有返回值，返回值为添加的根标记对象 Element */

```

Element root=document.createElement(ELEMENT_JDBC);
//3 添加子标记
/** 向根标记中添加子标记，添加多少个子标记是由给定的数据决定的 */
for(DBInfo info:infos){
    Element element=toDBInfoElement(info);    root.add(element);    }
//4 将根标记写出
XMLWriter writer=new XMLWriter(out);
writer.write(root);//将根标记写出，就相当于将整个树写出了，以 XML 的格式写出
的。
writer.flush();//清空缓存
writer.close();    System.out.println("写出完毕！");
}

```

step2：在 XMLUtils 工具类中，添加 toDBInfoElement(DBInfo info)方法

```

/** 将 DBinfo 转化为元素 Element */
public static Element toDBInfoElement(DBInfo info){
    //1 创建一个空的 Element 对象，该对象仅代表 oracle 或 mysql 标签，没有子元素
    //和属性
    Element element=DocumentHelper.createElement(info.getDbName());
    //2 添加子元素，如 url、driver 等标签
    Element url=DocumentHelper.createElement(ELEMENT_URL);
    element.add(url);
    Element url=element.addElement(ELEMENT_URL);//等同于上面两步
    url.setText(info.getUrl());
    Element driver=element.addElement(ELEMENT_DRIVER);
    driver.setText(info.getDriver());
    element.addElement(ELEMENT_DRIVER).setText(info.getDriver());//又等同于上两
    //步
    element.addElement(ELEMENT_USERNAME).setText(info.getUsername());
    element.addElement(ELEMENT_PASSWORD).setText(info.getPassword());
    //3 为当前标记加属性，如 user="user"
    element.addAttribute(ATTRIBUTE_USER, info.getAttUser());
    return element;
}

```

step3：测试

```

FileInputStream fis=new FileInputStream("src"+File.separator+
        "day03"+File.separator+"part2"+File.separator+"db_info.xml");
List<DBInfo> infos=XMLUtils.xmlToDBInfo(fis);
File file=new File("out.xml");
if(!file.exists()){    file.createNewFile();    }
FileOutputStream fos=new FileOutputStream(file);
XMLUtils.writeDBInfosToXml(fos, infos);

```

◆ 注意事项：XML 常被用于在不同系统间传递数据使用。还有一种比较常见的是 json。前两个以字符串传递。还有一个比较常见的是 google protocol buffer 以字节传递。

五十二、XPath 语言

XPath 是一种用于在 XML 中查找信息语言。XPath 可用来在 XML 文档中对元素和属性进行遍历。

4.1 XPath 基本介绍

- 1) XPath 使用路径表达式在 XML 文档中进行导航。
- 2) XPath 包含一个标准函数库。
- 3) XPath 是 XSLT 中的主要元素。
- 4) XPath 是一个 W3C 标准。

4.2 使用 XPath 的好处

当我们单纯使用 DOM 定位节点时，大部分时间需要一层一层的处理，如果有了 XPath，我们定位节点将变得很轻松，它可以根据路径、属性，甚至是条件进行节点的检索。

4.3 XPath 基本语法

XPath 的常见规则：

- 1) 从根节点开始查找，如：/bookshelf/book，即从根标记下查找所有的 book 标记。
- 2) 查找某个节点，不管其位置，如：//book，即忽略位置，查找所有 book 标记。
- 3) @选择属性，如：//@ID，即获取指定名称的属性，获取属性名为 ID 的属性，同样获取所有的 ID 属性。
- 4) [index]根据索引选择节点，如：/bookshelf/book[1]，即获取根标记下第一个 book 标记。
◆ 注意事项：索引从 1 开始！
- 5) 根据属性选取节点，如：/bookshelf/book[@ID='1234']，即获取根标记下所有属性 ID 为 1234 的 book 标记。
- 6) 根据子元素的值获取节点，如：/bookshelf/book@student='金三顺'即获取根标记下 book 标记的子标记 student 的值为“金三顺”的 book 标记。

4.4 DOM4J 对 XPath 的支持

```
SAXReader saxReader = new SAXReader();
Document document = saxReader.read(new File(filename));
List list=document.selectNodes("/books/book/@show");
Iterator iter=list.iterator();
```

五十三、附 db_info.xml 文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdbc [
    <!ELEMENT jdbc (oracle*,mysql*)>
    <!ELEMENT oracle ((url|uri),driver,username,password)>
    <!ELEMENT mysql (url,driver,username,password)>
    <!ELEMENT uri (#PCDATA)>
    <!ELEMENT url (#PCDATA)>
    <!ELEMENT driver (#PCDATA)>
    <!ELEMENT username (#PCDATA)>
    <!ELEMENT password (#PCDATA)>
    <!ATTLIST oracle user CDATA "user" #FIXED ("user"|"admin")>
    <!ATTLIST oracle id CDATA "xxx">
]>
<jdbc>
    <oracle user="user" id="sss">
        <url>jdbc:oracle:thin:@192.168.0.20:1521:tarena</url>
        <driver>oracle.jdbc.driver.OracleDriver</driver>
        <username>jsd1304</username>
        <password>jsd1304</password>
    </oracle>

    <oracle user="admin">
        <url>jdbc:oracle:thin:@192.168.0.26:1521:tarena</url>
        <driver>oracle.jdbc.driver.OracleDriver</driver>
        <username>jsd1304</username>
        <password>jsd1304</password>
    </oracle>

    <mysql>
        <url>jdbc:mysql://192.168.0.20:3306/tarena</url>
        <driver>com.mysql.jdbc.Driver</driver>
        <username>jsd1304</username>
        <password>jsd1304</password>
    </mysql>
</jdbc>
```

6 HTML 学习笔记

LICHOOL

五十四、HTML 概述

1.1 什么是 HTML

- 1) HTML (HyperText Markup Language) 是一种超文本标记语言，是一种纯文本型的语言，是用来设计网页的标记语言。
- 2) 用该语言编写的文件，以.html 或者.htm 为后缀。
- 3) 由浏览器解释运行。
- 4) HTML 是一个扩展性很强的语言，可以嵌套用脚本语言编写的程序段，如：VBScript、JavaScript。嵌入 JavaScript 代码可以实现动态效果，同时也可以使用 CSS 定义样式。

1.2 Web 浏览器

- 1) 主要功能：
 - ①代理访问者提交请求。
 - ②作为 HTML 解释器和内嵌脚本程序执行器。
 - ③用图形化的方式显示 HTML 文档。
- 2) 主要 Web 浏览器产品
IE、Firefox、Chrome、Opera、Safari

五十五、HTML 基础语法

2.1 标记语法

- 1) HTML 用于描述功能的符号称为“标记”，比如<p>、<h1>等。
- 2) 标记在使用时必须使用尖括号括起来，有些标记还必须成对出现。

2.2 封闭类型标记：双标记

- 1) 语法：
 <标记>内容</标记>
 <标记 属性 1="值" 属性 2="值">内容</标记>
 - 2) 属性的声明必须位于开始标记里。
 - 3) 一个标记的属性可能不止一个，多个属性之间用空格隔开。
- ◆ 注意事项：
- ❖ 封闭类型的标记必须成对出现。
 - ❖ 如果一个应该封闭的标记没有被封闭，则会产生意想不到的错误。

2.3 非封闭类型标记：单标记或者空标记

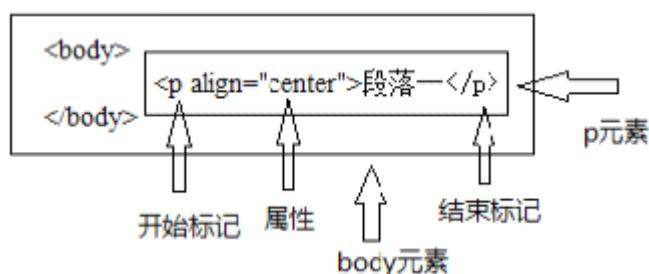
- 1) 语法：
 <标记>或者<标记 />
- 2) 不需要结束标记，不能包含内容，可以设置属性。

例如： hello word
hello word
hello word
hello word

- ◆ 注意事项：
- ❖
为当前标准，
为早期版本。
 - ❖ 对于单标记，建议写法
，而不是
。

2.4 元素和属性

- 1) 元素：每一对尖括号包围的部分，如<body></body>包围的部分就叫做 body 元素。
- 2) 属性：用来修饰元素，每个属性都有值，属性放在开始标记中。



2.5 注释

为代码添加适当的注释是一种良好的编码习惯。

- 1) 注释只在编辑文档情况下可见，在浏览器展示页面时并不会显示。

- 2) 添加注释的语法：

<!--注释的文本内容-->

- ◆ 注意事项：
 - ❖ “<!--” 和 “-->” 之间的任何内容都不会显示在浏览器中。
 - ❖ 注释不可以嵌套在其他注释中。

2.6 HTML 文档的标准结构

- 1) 结构:

```
版本信息
<html><!--HTML 页面-->
  <head></head><!--文件头-->
  <body></body><!--文件主体部分-->
</html>
```

- 2) 例如:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head></head>
  <body></body>
</html>
```

2.7 版本信息

1) 在文档的起始用 DOCTYPE 声明指定的版本和风格，让浏览器清楚文档的版本、类型和风格。版本信息分为三种：严格型、传统型（过渡型）、框架型。

- 2) Strict DTD:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

- 3) Transitional DTD: (常用)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
```

- 4) Frameset DTD: (不常用)

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Frameset//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-frameset.dtd">
```

- ◆ 注意事项: 传统型不需要命名空间，严格型需要命名空间。

2.8 <head> 元素

- 1) <head> 元素用于为页面定义全局信息
 - ①所有其他头元素的容器。 ②紧跟在起始标签<html>之后。
- 2) 定义整个文档相关的信息，常包含如下子元素:
 - ①<title>: 标题。
 - ②<meta>: 元数据元素，定义页面的编码格式或者刷新频率等。
 - ③<script>: JavaScript 脚本（或引入 Ajax、jQuery 脚本等）。
 - ④<style>: 定义内部样式表。
 - ⑤<link>: 为当前页面引入其他资源（如外部样式表）。

2.9 <body> 元素

文档的主体，包含所有要显示的内容。

2.10 头元素：<title>

标题元素<title></title>用于为文档定义标题。

- 1) 标题元素的内容出现在浏览器顶部。
- 2) 没有属性。
- 3) 必须出现在<head>元素中。
- 4) 一个文档只能有一个标题元素。

例如：<head>

```
<title>第一个网页</title>
</head>
```

2.11 头元素：<meta>

元数据元素<meta />用于定义网页的基本信息。

- 1) 为空标记。
- 2) 常用属性有：content、http-equiv

例如：<head>

```
<title>第一个网页</title>
<meta http-equiv="refresh" content="3" /><!--3 秒刷新一次-->
<!--文档内容为：文本格式的 html，字符集采用 utf-8-->
<meta http-equiv="content-type" content="text/html;charset=utf-8" />
</head>
```

2.12 案例：创建一个标准结构的 HTML 文档，并创建头元素。

```
<!--版本信息-->
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<!--html 元素，表示整个文档-->
<html>
    <!--头元素：描述整个文档的相关信息-->
    <head>
        <title>第一个网页</title>
        <meta http-equiv="refresh" content="3" /><!--3 秒刷新-->
        <meta http-equiv="content-type" content="text/html;charset=utf-8" />
    </head>
    <!--文档主体：显示-->
    <body>
    </body>
</html>
```

- ◆ 注意事项：若不想出现乱码，则存储时的物理编码和查看时的编码需要一致；若出现乱码，则一看保存时的编码，二指定查看编码。

五十六、文本标记

LICHOO

3.1 文本标记的作用

- 1) 文本是网页上的基本成分。
- 2) 直接书写的文本会用浏览器默认的样式显示。
- 3) 包含在标记中的文本则会被显示为标记所拥有的样式：特殊字符、注释、标题元素、段落元素、换行元素、分区元素。

3.2 文本于特数字符

- 1) 空格折叠：多个空格或制表符压缩成单个空格，即只显示一个空格。
- 2) 特殊字符（如空格），可以用转义字符，也称为字符实体。

例如：The <p> element. ©2013 by chang.

对应：The <p> element. ©2013 by chang.

3.3 标题元素<hn>

- 1) 标题元素让文字以醒目的方式显示，往往用于文章的标题。
- 2) 基本语法：`<h#>……</h#>`, #：可以为 1、2、3、4、5、6
从

到，即标题 1 到标题 6
- ◆ 注意事项：`<h1>最大，<h6>最小。`

3.4 段落元素<p>

- 1) <p>元素提供了结构化文本的一种方式。
- 2) <p>元素中的文本会用单独的段落显示：
 - ①与前后的文本都换行分开（即 p 中的内容会独占一行）。
 - ②添加一段额外的垂直空白距离，作为段落间距（与
相比，间距较大）。

常用属性：align（可用的值有：left、right、center）

例如：<p>The first paragraph.</p>

<p align="right">The first paragraph.</p>

3.5 换行元素

使用
元素在任何地方创建手动换行，该元素为空标记，语法为：
。相当于回车，间距较小。

3.6 分区元素和<div>

- 1) 分区元素用于为元素分组，常用于页面布局。即对某些元素进行一些统一的设置。
- 2) 文本：不会影响布局，常用于同一行中部分元素。
- 3) <div>文本</div>：独占一行，常用于多行的情况下。

3.7 块级元素（block）和行内元素（inline）

- 1) 块级元素：默认情况下，块级元素会独占一行，即前后都会自动换行，比如：<div>、<p>、<hn>、
- 2) 行内元素：可以和其他元素位于同一行，即不会换行，比如：、、<a>

3.8 案例：使用文本标记为页面添加内容

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
  <head>
    <title>一个 HTML 文档</title>
    <meta http-equiv="content-type" content="text/html; charset=utf-8" />
  </head>
  <body>
    <h1 align="center">Java&nbsp;语言基础&nbsp;<span style="color:red;">
      &lt;Day03&gt;</span></h1>
    <h2>1&nbsp;个人所得税计算器</h2>
    <h3>1.1&nbsp;问题</h3>
    <p>计算个人所的税</p>
    <h2>1.2&nbsp;方案</h2>
    <p>使用 if 语句来完成该程序</p>
    <h2>1.3&nbsp;实现</h2>
    <p>使用记事本，代码如下：</p>
    <p>public class IncomeTax<br />
    {<br /><br /><br />}<br />
    </p>
  </body>
</html>
```

五十七、图像和连接

LICHOOL

4.1 图像元素

- 1) 使用元素将图像添加到页面，该元素为空标记，语法为。
- 2) 必须属性：src
- 3) 常用属性：width、height
- 4) 语法：``

例如：``

◆ 注意事项：

```
<!--本地的绝对路径，放在 Web 里是不行的！-->  
<!--相对路径，当前项目下-->  
<!--绝对路径，全路径-->
```

不建议宽高都设置，因为不知原图的比例，都设置会变形，只设置一个，系统会自动按比例缩放。

4.2 链接元素

- 1) 使用元素创建一个超级链接：点击、去往其他资源（常见的去页面）。
- 2) 语法：`被单击的内容、文本或图片`
 - ① href 属性：链接 URL
 - ② target 属性：目标，可取的值有：`_self`: 默认值，替换当前页
`_blank`: 打开新的空白页，显示页面

4.3 URL

- 1) URL (Uniform Resource Locator): 统一资源定位器，用来标识网络中的任何资源。
如：文本、图片、音视频文件、段落或其他超文本。
- 2) 完整 URL 的组成：协议、机名、路径名、文件名。
- 3) URL 中的路径名表示方法有：相对路径和绝对路径。

4.4 锚点

- 1) 锚点是文档中某行的一个记号，用于链接到文档中的某一行。即实现当前页面的不同位置之间的跳转。
- 2) 如何使用锚点：
第一步：使用 a 在目标位置定义一个锚点，``
第二步：使用连接 a 链接到锚点（在锚点名前加上#），href 指向 link1，``
- ◆ 注意事项：
 - ❖ #代表后面的不是页面，而是一个锚点。
 - ❖ 页面不同位置之间的跳转要在有滚动条的情况下才有效！
- 3) 直接回到页面的顶端
 - ①早期版本里，要先定义锚点，再定义链接。
 - ②因为非常常用，所以现在简化了，直接写个#，不用先去定义锚点了：
`test`

4.5 案例：使用图像和链接标记

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
<title>一个 HTML 文档</title>
<meta http-equiv="content-type" content="text/html; charset=utf-8" />
</head>
<body>
<h1 align="center">Java&nbsp;语言基础&nbsp;<span style="color:red;">
&lt;Day03&gt;</span></h1>
<h2>1&nbsp;个人所得税计算器</h2>
<h3>1.1&nbsp;问题</h3>
<p>计算个人所的税</p>
<div align="center">
<a href="http://tts6.tarena.com.cn" target="_blank" >

</a>
<p>图-1</p>
</div>
<h2>1.2&nbsp;方案</h2>
<p>使用 if 语句来完成该程序</p>
<div align="center">
<a href="http://tts6.tarena.com.cn" target="_blank" >

</a>
<p>图-2</p>
</div>
<h2>1.3&nbsp;实现</h2>
<p>使用记事本，代码如下：</p>
<p>public class IncomeTax<br />
{<br /><br /><br />}<br />
</p>
<p align="center"><a href="#">To Top</a></p>
</body>
</html>
```

五十八、列表标记

5.1 列表的作用

- 1) 列表是指将具有相似特征或者具有先后顺序的几行文字进行对其排列。
- 2) 所有的列表都由列表类型和列表项组成：
 - ①列表类型有：有序列表``（ordered list）和无序列表``（unordered list）。
 - ②列表项有：``（list item），用于指示具体的列表内容。
- ◆ 注意事项：定义一个列表必须使用``或``。列表的每个内容，使用一个``。

5.2 无序列表``

- 1) ``元素表示无序列表，用于列出页面上没有特定次序的一些项目。
- 2) ``元素中只能包含具体的列表项元素``，列表中包含的每一项都必须包含在起始标记``和结束标记``之间。

例如：` onetwo `

5.3 有序列表``

- 1) ``元素表示有序列表，用于列出页面上有特定次序的一些项目。
- 2) ``元素中只能包含具体的列表项元素``，列表中包含的每一项都必须包含在起始标记``和结束标记``之间。

例如：` onetwo `

5.4 列表的嵌套

- 1) 将列表元素嵌套使用，可以创建多层列表，即可以将整个列表放在某个`li`里。
- 2) 常用于创建文档大纲、导航菜单等。

例如：``

```

<li>Web 基础知识
    <ul><li>Web 的工作原理</li></ul>
</li>
<li>HTML 快速入门
    <ul><li>基础语法</li></ul>
</li>
</ul>
```

5.5 案例：使用列表标记添加导航目录

```

<ol>
    <li>个人所得税计算器
        <ul>
            <li><a href="#link1">问题</a></li>
            <li><a href="#link2">方案</a></li>
        </ul>
    </li>
</ol>
```

五十九、表格

LICHOO

6.1 表格的作用

- 1) 表格通常用来组织结构化的信息。
- 2) 表格是一些被称作单元格的矩形框按照从左到右，从上到下的顺序排列在一起形成的。
- 3) 表格的数据保存在单元格里。
- 4) 显示网格数据，常用于页面的布局。

6.2 创建表格

- 1) 定义表格：使用成对的<table></table>标记。
- 2) 创建表行：使用成对的<tr></tr>标记。
- 3) 创建单元格：使用成对的<td></td>标记（table definition）。

```
例如: <table border="1">
    <tr> <td>第 1 行, 第 1 列</td><td>第 1 行, 第 2 列</td></tr>
    <tr> <td>第 2 行, 第 1 列</td><td>第 2 行, 第 2 列</td></tr>
</table>
```

6.3 表格的常用属性

- 1) border 属性：表格外边框宽度，会为每个单元格应用边框！
 - 2) width/height 属性：表格的宽和高。
 - 3) align 属性：水平对齐方式，可选的值有：left、right、center
 - 4) cellpadding 属性：单元格内容和单元格边框之间的距离。
 - 5) cellspacing 属性：单元格之间的距离。
- ◆ 注意事项：
- ❖ 表格的高宽默认自适应（按内容长度自适应）。
 - ❖ 对表格的宽设定值，则每列的宽按单元格内容的长度的比例分配，如第一列 5 个字符，第二列 4 个字符，则分配比例为 5: 4。
 - ❖ 对列设置宽，会影响整列。对列设置高，会影响整行。

6.4 单元格的常用属性

- 1) width/height 属性：单元格的宽和高。
- 2) align 属性：水平对齐方式，可选的值有：left、right、center
- 3) valign 属性：垂直对齐方式，可选的值有：top、middle、bottom

6.5 表格的标题<caption>

- 1) 使用<caption>元素为表格定义标题，默认情况下，标题将在表格上方居中显示。
- 2) <caption>标签只能位于表格<table>里，且只能对每个表格定义一个标题，且作为第一个子元素存在！

```
例如: <table border="1">
    <caption>我的表格</caption>
    <tr> <td>第 1 行, 第 1 列</td><td>第 1 行, 第 2 列</td></tr>
</table>
```

6.6 行分组（表格特有的）

将多个行作为一组，进行统一设置的时候，使用专门的对表格实现分组的元素。

- 1) 表格可以划分为 3 个部分：表头、表主体和表尾。
 - 2) 表头行分组：`<thead></thead>`只能出现一次
 - 3) 表主体行分组：`<tbody></tbody>`可出现多次
 - 4) 表尾行分组：`<tfoot></tfoot>`只能出现一次
- ◆ 注意事项：行分组元素只能出现在 `table` 里，只能包含 `tr` 元素。

6.7 不规则表格

对`<td>`元素而言，有 `colspan`、`rowspan` 属性设置单元格跨列或者跨行。

- 1) 跨列：`colspan`

水平方向（横着）延伸单元格，值为正整数，代表此单元格水平延伸的单元格数。

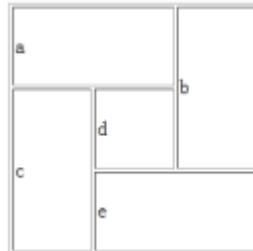
- 2) 跨行：`rowspan`

垂直方向（竖着）延伸单元格，值为正整数，代表此单元格垂直延伸的单元格数。

6.8 表格的嵌套

- 1) 在单元格中放置另外一个表在表单元格中，即`<td>`元素中再包含`<table>`元素。
- 2) 使用嵌套的表格以设计复杂表格或者复杂布局。

6.9 案例：实现如下图所示表格



```
<table border="1" width="300" height="300">
  <tr>
    <td colspan="2">a</td>
    <td rowspan="2">b</td>
  </tr>
  <tr>
    <td rowspan="2">c</td>
    <td>d</td>
  </tr>
  <tr>
    <td colspan="2">e</td>
  </tr>
</table>
```

六十、表单

LICHOO

7.1 表单的作用

- 1) 动态交互：通过查看、填写（页面上录入数据）并提交表单信息到服务器端。
- 2) 表单有两个基本部分：
 - ①实现数据交互的可见的界面元素，比如文本框或按钮。
 - ②提交后的表单处理。
- 3) 界面元素：
 - ①使用<form>元素创建表单。
 - ②在<form>元素中添加其他表单可以包含的控件元素。

7.2 表单元素<form>

- 1) 定义表单：使用成对的<form></form>标记。
- 2) 承载其他交互的元素，以表单为单位提交数据。
- 3) 主要属性：
 - ①action 属性：定义表单被提交时发生动作，通常包含服务方脚本的 URL（比如 JSP、PHP）。
 - ②method 属性：指出表单数据提交的方式，取值为 get 或者 post。
 - ③enctype 属性：表单数据进行编码的方式。

例如：<form action="login.jsp" method="post">
 地址信息：包含文本框、密码框等录入数据的元素
 包含一个提交按钮<!--此处的提交按钮只对当前 form 里的数据进行提交-->
 </form>
 <form action="">发票信息：</form>

7.3 <input>元素

- 1) <input>元素用于收集用户信息。
- 2) 该元素是一个空标记，语法为：<input />
- 3) 主要属性：
 - ①type 属性：依靠 type 属性的取值决定元素的类型。
 - ②value 属性：取决于元素的类型，用作初始值（文本框/密码框）、用于提交的值（单/多选框）、按钮上的文本显示（提交/重置/普通按钮）。
 - ③name 属性：单选或者多选框的分组。同一组 name 应相同。

7.4 文本框与密码框

- 1) 文本框<input type="text" />
- 2) 密码框<input type="password" />
- 3) 主要属性：
 - ①value 属性：用作初始值。
 - ②maxlength 属性：限制输入的字符数。
 - ③readonly 属性：设置文本控件只读，老版本可写 readonly=""，当前写成 readonly="readonly"。

7.5 单选框和多选框

- 1) 单选框: <input type="radio" />
- 2) 多选框: <input type="checkbox" />
- 3) 主要属性:
 - ①value 属性: 当提交 form 时, 如果选中了此单选框, 那么 value 的值就被发送到服务器。
 - ②name 属性: 单选或者多选框的分组。同一组 name 应相同。
 - ③checked 属性: 设置初始状态是否为选中, 老版本可写 checked="" , 当前写成 checked="checked"。

7.6 按钮

- 1) 提交按钮: <input type="submit" />, 传送表单数据给服务器端或其他程序处理, 并且页面刷新。
- 2) 重置按钮: <input type="reset" />, 清空表单的内容并把所有表单控件恢复到默认值状态。
- 3) 普通按钮: <input type="button" />, 用于执行客户端脚本, 为其设置 onclick 事件, 才会有功能。
- 4) 主要属性: value 属性: 按钮上的文本显示。

7.7 隐藏域和文件选择框

- 1) 隐藏域: <input type="hidden" />, 不会显示, 常用于在页面上隐藏那些不希望被用户看到的数据。
- 2) 文件选择框: <input type="file" />, 选择要上传的文件。

7.8 <label>元素

- 1) 语法: <label>文本</label>
- 2) 主要属性: for 属性: 表示与该元素相关联的控件的 ID 值。
- 3) 作用: 使用 for 属性关联其他元素, 使单击文本时, 就像单击关联的控件 (元素) 一样

例如: <input type="radio" name="sex" value="0" id="s1"/><label for="s1">女士</label>
 <input type="radio" name="sex" value="1" id="s2" /><label for="s2">男士</label>

7.9 选项框

- 1) 两种: 下拉选项框和滚动列表
- 2) <select>: 创建选项框
 - ①name 属性: 选项框命名。
 - ②size 属性: 大于 1, 则为滚动列表, 即由 size 属性区分下拉和滚动列表。
 - ③multiple: 设置多选。
- 3) <option>: 选项
 - ①value 属性: 选项用于提交的值。
 - ②selected 属性: 预选中, 老版本可写 selected="" , 当前写成 selected="selected"。

7.10 <textarea>元素

- 1) 多行文本输入框: <textarea>文本</textarea>
- 2) 主要属性:
 - ①cols 属性: 指定文本区域的列数。
 - ②rows 属性: 指定文本区域的行数。
 - ③readonly 属性: 设置只读。

7.11 表单元素分组（表单元素特有的）

- 1) <fieldset>元素: 为表单控件分组。
- 2) <legend>元素: 为分组指定一个标题

例如: <fieldset><!--表单元素特有分组-->

```
<legend>角色</legend><!--表单分组标题名-->
<input type="checkbox" name="role" value="0" id="r1" />
<label for="r1">超级管理员</label><br /><!--label 关联元素-->
<input type="checkbox" name="role" value="1" id="r2" />
<label for="r2">帐单管理员</label>
</fieldset>
```

◆ 注意事项: 所有标签都可以加 ID 属性, 为了作唯一区分。

7.12 案例: 创建复杂表单

```
<h2>增加管理员</h2>
<form> <table> <tr> <td align="right">姓名: </td>
    <td><input type="text" value="marry" /></td>
    <td>10 个字符以内</td> </tr>
    <tr> <td align="right">密码: </td>
        <td><input type="password" /></td>
        <td>10 个字符以内</td> </tr>
    <tr> <td align="right">性别: </td><!--label 关联元素-->
        <td><input type="radio" name="sex" value="0" id="s1" /><label for="s1"> 女士 </label>
        <input type="radio" name="sex" value="1" id="s2" /><label for="s2"> 男士 </label>
    </td><td></td> </tr>
    <tr> <td align="right">角色: </td>
        <td><fieldset><!--表单元素特有分组-->
            <legend>角色</legend><!--表单分组标题名-->
            <input type="checkbox" name="role" value="0" id="r1" />
            <label for="r1">超级管理员</label><br /><!--label 关联元素-->
            <input type="checkbox" name="role" value="1" id="r2" />
            <label for="r2">帐单管理员</label>
        </fieldset></td>
        <td>至少选择一个角色</td> </tr>
        <tr> <td align="right">上传头像: </td><td><input type="file" /></td><td></td></tr>
```

```
<tr><td align="right">状态: </td>
    <td><select><!-- 下拉菜单-->
        <option value="1">启用</option>
        <option value="2">停用</option>
        <option value="3">删除</option></select> </td><td></td> </tr>
<tr><td align="right">自我描述: </td>
    <td><textarea></textarea><!--多行文本框--></td><td></td> </tr>
<tr> <td></td><td align="center">
    <input type="submit" value="保存" />&nbsp;<input type="reset" value="重置" />
    <input type="hidden" value="123" /><!--隐藏域-->
    <input type="button" value="按钮 1" /></td>
</tr>
</table> </form>
```

六十一、框架

8.1 浮动框架的作用

浮动框架有时称为内联框架，用于在当前文档上引入并显示其他页面（嵌入），常用于类似于广告页面的嵌入。

8.2 <iframe>元素

- 1) 语法: <iframe src="URL"></iframe>
- 2) 主要属性:
 - ①src 属性: 指定在<iframe>中显示的文档的 URL。
 - ②width/height 属性: 指定框架的宽度和高度。

六十二、其他注意事项

9.1 Web 应用程序的层次结构

DB 数据层-->数据访问层（统称业务层）-->IU 层（用户界面）

9.2 界面层中根据类型应用程序的分类

DOS（控制台）、桌面应用程序（独立安装、独立运行）、Web 类型

9.3 单机应用和 Web 应用

单机应用：安装后，有更新则需要用户去升级。

Web 应用：通过浏览器，输入一个 url 地址，用户不需要升级，升级是在服务器端进行的。

9.4 Web 类型的应用程序

程序位于服务器，用户只需要通过浏览器去访问和交互；

又分两类：网站、程序（OA、CRM 等）。

9.5 本课程分为：

客户端：页面的创建，如搭建（HTML）、美化（CSS）、动态的显示效果（JavaScript）

服务器端：页面和数据库的交互，如 JSP 和 Servlet

◆ 注意事项：Ajax&Jquery 也可实现动态显示效果；PHP/ASP.NET 也可实现页面和数据库的交互。

9.6 何为 Web 基础

HTML+CSS+JavaServlet，是 JSP、PHP、ASP.NET 公共的部分。

7 CSS 学习笔记

六十三、CSS 概述

1.1 CSS 的作用

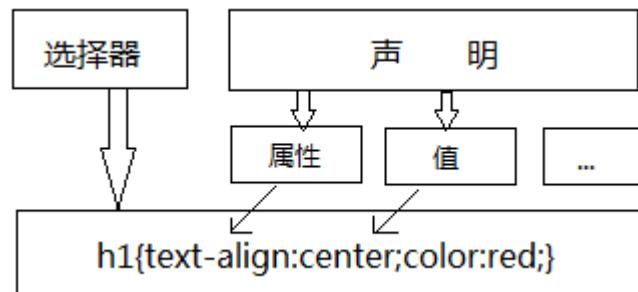
早期，依靠 HTML 元素的属性设置样式，比如：border/align；而每个元素的属性不尽相同，所以样式设置比较混乱；因此，W3C 推出了一套标准：使用某种样式声明后，所有的元素通用，即 CSS 产生，它是对页面的样式进行统一的定义（声明）的。

1.2 什么是 CSS

- 1) CSS (Cascading Style Sheets)：层叠样式表，又叫级联样式表，简称样式表。
- 2) 用于 HTML 文档中元素的样式定义。
- 3) 实现了将内容与表现分离。
- 4) 提高了代码的可重用性和可维护性。

1.3 CSS 的基础语法

- 1) 样式表由多个样式规则组成，每个样式规则有两个部分：选择器和声明。
- 2) 选择器：决定哪些元素使用这些规则。
- 3) 声明：由一个或者多个属性/值对组成，用于设置元素的外观表现。



六十四、如何使用 CSS 样式表

LICHOO

一共有三种使用方式：内联样式、内部样式表、外部样式表。

2.1 内联样式

样式定义在单个的 HTML 元素中。

- 1) 样式定义在 HTML 元素的标准属性 style 里。
- 2) 不需要定义选择器，也不需要大括号。
- 3) 只需要将分号隔开的一个或者多个属性/值对，作为元素的 style 属性的值。

例如：`<h1 style="background-color:silver;color:blue">文本</h1>`

2.2 内部样式表

样式定义在 HTML 页的头元素中。

- 1) 样式表规则位于文档头元素的`<style>`元素内。
- 2) 在文档的`<head>`元素内添加`<style>`元素，在`<style>`元素中添加样式规则。

例如：`<head><style type="text/css">
 h1 { color:green; }
</style></head>
<body><h1>文本</h1></body>`

- ◆ 注意事项：`<style>`元素中的属性 `type="text/css"` 可省略。且`<style>`元素里的注释要用 CSS 规定的注释/*注释*/，而不是 HTML 里的注释`<!--注释-->`。

2.3 外部样式表

将样式定义在一个外部的 CSS 文件中（.css 文件），由 HTML 页面引用样式表文件。

- 1) 首先需要创建一个单独的样式表文件，用来保存样式规则。
 - ①一个纯文本文件。②该文件中只能包含 CSS 样式规则。③文件后缀为.css。
- 2) 然后在需要使用该样式表文件的页面上，使用`<link>`元素链接需要的外部样式表文件。

例如：`myStyle.css` 文件：

```
h1 { color:green; }  
p { background-color:silver;color:blue; }  
html 文件里的<head>元素：  
<head>  
    <link rel="stylesheet" type="text/css" href="myStyle.css" />  
</head>
```

- ◆ 注意事项：`rel`: 代表做什么用；`href`: 代表引入的文件在哪；`type`: 代表引入的文件是什么类型的；`text/css`: 代表纯文本类型的 CSS 代码。

2.4 三种用法的区别

- 1) 内联样式：将样式定义在元素的 style 属性里；但没有重用性。
- 2) 内部样式表：将样式定义在`<head>`元素里的`<style>`里；但仅限于当前文档范围重用。
- 3) 外部样式表：将样式定义在单独的.css 文件里，有页面引入它；但可维护性和可重用性高，同时实现了数据（内容）和表现的分离。

2.5 CSS 样式表特征和优先级

- 1) 继承性：大多数 CSS 的样式规则可以被继承（子元素继承父元素的样式）。
- 2) 层叠性：可以定义多个样式表；不冲突时，多个样式表中的样式可层叠为一个，即不冲突时采用并集方式。
 - 3) 优先级：即冲突时采用优先级
内联 > 内部 或者 外部。
内部和外部：优先级相同的情况下，采取就近原则，以最后一次定义的为优先。
所以，当修改时，不想去找，就在 CSS 中最后的位置重新写一遍新的样式。这也是 CSS 文件越来越大的原因。
 - ◆ 注意事项：还应注意浏览器的缺省设置。
- 4) 所以级联（层叠）样式表 CSS 的特点是：继承+并集+优先级。

六十五、CSS 选择器

LICHOOL

3.1 元素选择器

HTML 文档的元素名称就是元素选择器。

- 1) 语法, 例如: `html{color:black;}`、`h1{color:blue;}`、`p{color:silver;}`
- 2) 缺点: 不同的元素样式相同, 即不能跨元素。所以做不到同一类元素下的细分。

3.2 类选择器

自定义的某种选择器。

- 1) 语法: `.className{样式声明}`

例如: `.myClass{ background-color:pink;font-size:35pt; }`
`<h2 class="myClass">h2 中的文本</h2>`
`<p class="myClass">p 中的文本</p>`

- ◆ 注意事项:
 - ❖ html 文件中, 所有元素都有一个 `class` 属性, 如: `<p class="name"></p>`
 - ❖ 类选择器还一种用法: `<div id="d1" class="s1 s2">hello</div>`, 样式 `s1` 和样式 `s2` 对 `div` 共同起作用。

3.3 分类选择器

将类选择器和元素选择器结合起来使用, 以实现同一类元素下不同样式的细分控制。如 `<input>` 元素, 又有按钮又有文本框的, 采用分类选择器。

- 1) 语法: 元素选择器.`.className{样式声明}`

例如: `p.myClass{ color:red;font-size:20pt }`
`<h2 class="myClass">h2 中的文本</h2>`
`<p class="myClass">p1 中的文本</p>`
`<p>p2 中的文本</p>`

3.4 元素 id 选择器

以某个元素 `id` 的值作为选择器。比较特殊的、页面整体结构的划分一般使用 `id` 选择器。

- 1) 语法: 定义 `id` 选择器时, 选择器前面需要有一个 “#” 号, 选择器本身则为文档中某个元素的 `id` 属性的值。

例如: `#header{ color:red;background:yellow; }`
`<h1 id="header">This is Title</h1>`

- ◆ 注意事项:
 - ❖ html 文件中, 所有元素都有一个 `id` 属性。且某个 `id` 选择器仅使用一次。

3.5 派生选择器

依靠元素的层次关系来定义。某一包含元素下的一些相同子元素使用派生选择器。

- 1) 语法: 通过依据元素在其位置的上下文关系来定义样式, 选择器一端包括两个或多个用空格分隔的选择器。

例如: `h1 span{color:red;}`
`<h1>This is aimportantheading</h1>`

3.6 选择器分组

对某些选择器定义一些统一的设置（相同的部分）。

1) 语法：选择器声明为以逗号隔开的元素列表。

例如：h1,p,div{ border:1px solid black;}

3.7 伪类选择器

伪类用于向某些选择器添加特殊的效果。

1) 语法：使用冒号“：“作为结合符，结合符左边是其他选择器，右边是伪类。

2) 常用伪类：有些元素有不同的状态，典型的是[元素](#)。

①:link: 未访问过的链接 ②:active: 激活

③:visited: 访问过的链接 ④:hover: 悬停，鼠标移入，所有元素都能用

⑤:focus: 获得焦点

例如：a:link{ color:black;font-size:15pt; }

a:visited{ color:pink;font-size:15pt; }

a:hover{ font-size:20pt; }

a:active{ color:red; }

3.8 选择器优先级

1) 基本规则：

内联样式 > id 选择器 > 类选择器 > 元素选择器

2) 优先级从低到高排序：

div < .class < div.class < #id < div#id < #id.class < div#id.class

例如：<div id="id" class="class" style="color:black;"></div>

六十六、CSS 单位

4.1 尺寸

1) %: 百分比

2) in: 英寸

3) cm: 厘米

4) mm: 毫米

5) pt: 磅（1pt 等于 1/72 英寸，绝对）

6) px: 像素（计算机屏幕上的一个点，相对）

7) em: 1em 等于当前字体尺寸，2em 等于当前字体尺寸的两倍

4.2 颜色

1) rgb(x,x,x): RGB 值，如 rgb(255,0,0)

2) rgb(x%,x%,x%): RGB 百分比值，如 rgb(100%,0%,0%)

3) #rrggbb: 十六进制数，如#ff0000

4) #rgb: 简写的十六进制数，如#f00（两两相同可简写）

4.3 尺寸属性

1) width 和 height: 宽度和高度。

2) overflow: 当内容溢出元素框时如何处理，可取的值有：visible/hidden/scroll（总是显示滚动条）/auto（溢出时才会显示滚动条）

六十七、边框样式

5.1 简写方式

```
border:width style color;
```

5.2 单边定义

```
border-left/right/top/bottom:width style color;
```

5.3 单边宽度 border-width

```
border-left/right/top/bottom-width;
```

5.4 单边样式 border-style

```
border-left/right/top/bottom-style;
```

5.5 单边颜色 border-color

```
border-left/right/top/bottom-color;
```

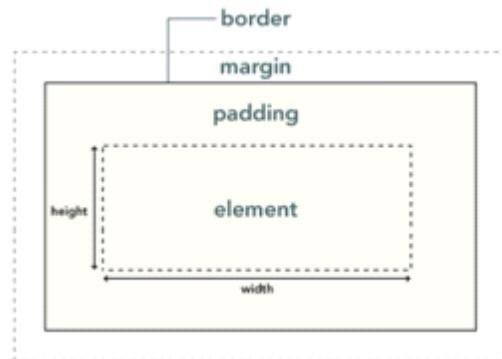
5.6 案例

```
input.btn {height:30px; width:80px; color:black; background:#f0f0f0;  
border-left:2px solid #fff; border-top:2px solid #fff;  
border-right:2px solid #6a6a6a; border-bottom:2px solid #6a6a6a;  
background:#c0c0c0; }
```

六十八、框模型

框模型（box model）定义了元素边框、内容、其他元素之间的内边距和外边距的问题。

6.1 框模型图



6.2 width 和 height

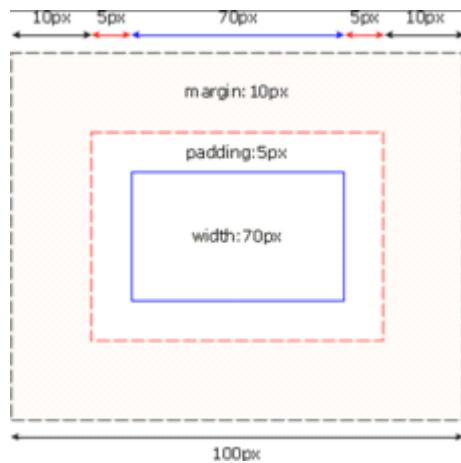
指定内容域的宽度和高度。

6.3 边框、内外边距对元素尺寸的影响

增加内边距、边框和外边距不会影响内容域的尺寸，但是会增加元素框的总尺寸。

6.4 案例

```
#box { width:70px; margin:10px; padding:5px }
```



◆ 注意事项：注意元素所占面积的变化。

6.5 内边距

和内容元素之间的距离。

- 1) 语法: padding:30px;
- 2) 单边设置: padding-left/top/right/bottom:30px;
- 3) 简写方式: ①padding:10px 20px; 上下 左右
 ②padding:10px 20px 30px 40px; 顺时针, 上右下左

6.6 外边距

与下一个元素之间的空间量。

- 1) 语法: margin:30px;
- 2) 单边设置: margin-left/top/right/bottom:30px;
- 3) 简写方式: ①margin:10px 20px; 上下 左右
 ②margin:10px 20px 30px 40px; 顺时针, 上右下左
- 4) 特殊写法: margin:0px auto; 对 margin 而言, margin 可取值为 auto, 使某个框居中显示; 屏幕宽度减去元素宽度, 除以 2, 左右平分

六十九、背景样式

LICHOOL

7.1 背景色

background-color:颜色;

7.2 背景图像

background-image:url(a.jpg);/*注意， url 里没有双引号*/

7.3 背景平铺

background-repeat:repeat (默认的) /no-repeat/repeat-x/repeat-y;

7.4 背景固定

background-attachment:scroll/fixed; /*scroll:背景和内容一起滚动;fixed:背景固定，类似于水印的效果，走到哪跟到哪*/

7.5 背景定位

background-position:left top; /*可写 px、%、center 单词*/

7.6 组合设置

background:background-color || background-image || background-repeat
|| background-attachment || background-position;

7.7 案例

background:#9dcdfc url(a.jpg) no-repeat fixed left top;

- ◆ 注意事项：
 - ❖ 如使用组合属性定义其单个参数，则其他参数的默认值将无条件覆盖各自对应的单个属性设置。比如设置 background:white 等价于设置
`background:white none repeat scroll 0% 0%`
如果在此之前设置 background-image 属性，则其设置将被 background-image 的默认值 none 覆盖。
 - ❖ `` 错误，img 没有占一定的面积、空间，所以不能设置背景。

七十、文本/字体样式

LICHOOL

8.1 指定字体

1) 语法:

font-family:name;
font-family:ncursive/fantasy/monospace/serif/sans-serif;

2) 取值说明:

name: 字体名称。按优先顺序排列。以逗号隔开。如果字体名称包含空格，则应使用引号括起第二种声明方式使用所列出的字体序列名称。如果使用 fantasy 序列，将提供默认字体序列。

8.2 字体颜色

1) 语法: color:color;

◆ 注意事项: 用颜色名称指定 color 不被一些浏览器接受。但使用 RGB 值指定颜色能被所有浏览器识别并正确显示。

8.3 字体大小

1) 语 法 :
font-size:xx-small/x-small/small/medium/large/x-large/xx-large/larger/smaller/length;
2) 取值说明:
①xx-small: 绝对字体尺寸。根据对象字体进行调整。最小
②x-small: 绝对字体尺寸。根据对象字体进行调整。较小
③small: 绝对字体尺寸。根据对象字体进行调整。小
④medium: 默认值。绝对字体尺寸。根据对象字体进行调整。正常
⑤large: 绝对字体尺寸。根据对象字体进行调整。大
⑥x-large: 绝对字体尺寸。根据对象字体进行调整。较大
⑦xx-large: 绝对字体尺寸。根据对象字体进行调整。最大
⑧larger: 相对字体尺寸。相对于父对象中字体尺寸进行相对增大。使用成比例的 em 单位计算
⑨smaller: 相对字体尺寸。相对于父对象中字体尺寸进行相对减小。使用成比例的 em 单位计算
⑩length: 百分数/由浮点数字和单位标识符组成的长度值, 不可为负值。其百分比取值是基于父对象中字体的尺寸。

◆ 注意事项: 字体一般为 10-12 磅。

8.4 字体加粗

1) 语法: font-weight:normal:bold/bolder/lighter/100/200/300/400/500/600/700/800/900;

2) 取值说明:

①normal: 默认值, 正常的字体, 相当于 400 。声明此值将取消之前任何设置
②bold : 粗体。相当于 700 。也相当于 b 对象的作用
③bolder: 比 normal 粗
④lighter: 比 normal 细
⑤100: 字体至少像 200 那样细

- ⑥200: 字体至少像 100 那样粗, 像 300 那样细
- ⑦300: 字体至少像 200 那样粗, 像 400 那样细
- ⑧400: 相当于 normal
- ⑨500: 字体至少像 400 那样粗, 像 600 那样细
- ⑩600: 字体至少像 500 那样粗, 像 700 那样细
- ⑪700: 相当于 bold
- ⑫800: 字体至少像 700 那样粗, 像 900 那样细
- ⑬900: 字体至少像 800 那样粗

8.5 文本排列

- 1) 语法: `text-align:left/right/center/justify;`
- 2) 取值说明:
 - ①left: 默认值。左对齐 ②right: 右对齐
 - ③center: 居中对齐 ④justify: 两端对齐

◆ 注意事项:

 - ❖ 此属性作用于所有块级元素, 在一个 div 对象里的所有块级元素会继承此属性值。
 - ❖ 只对块级元素生效, 对行内元素无效!

8.6 行高

常用于控制文本的垂直布局。

- 1) 语法: `line-height:normal/length;`
- 2) 取值说明:
 - ①normal: 默认值。默认行高
 - ②length: 百分比数字/由浮点数字和单位标识符组成的长度值, 允许为负值。其百分比取值是基于字体的高度尺寸。

◆ 注意事项: 设置为和包含元素一样高, 即可实现垂直居中效果。

8.7 文字修饰

- 1) 语法: `text-decoration:none || underline || blink || overline || line-through;`
- 2) 取值说明:
 - ①none: 默认值。无装饰 ②blink: 闪烁
 - ③underline: 下划线 ④line-through: 贯穿线
 - ⑤overline: 上划线

◆ 注意事项:

 - ❖ 假如 none 值在属性声明的最后, 所有的先前的其他取值都会被清除。
例如, 声明 `text-decoration: underline overline blink none`
等于声明 `text-decoration: none`。
 - ❖ 指定块对象的此属性将影响其所有内联子对象。

8.8 文本缩进

首行缩进, 常用于段落。

- 1) 语法: `text-indent:length;`
- 2) 取值说明: length: 百分比数字/由浮点数字和单位标识符组成的长度值, 允许为负

值， 默认值为 0 。

LICHO0

七十一、表格样式

是表格特有的属性。

9.1 垂直对齐

vertical-align:top/middle (默认) /bottom; 单元格中垂直方向上的对齐

9.2 边框合并

border-collapse:separate (默认) /collapse; 对 table 设置此属性，对 td 设置无效

9.3 边框间距

border-spacing:value;

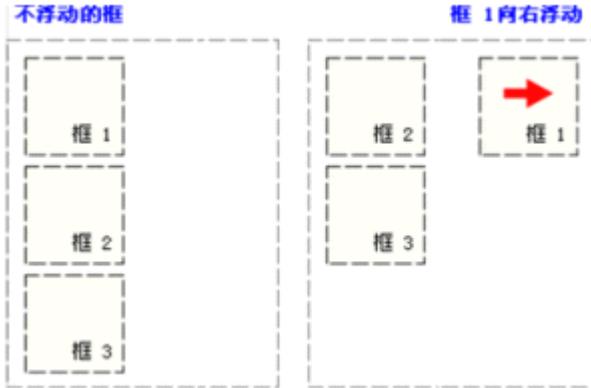
◆ 注意事项：

- ❖ 对 table 设置边框，仅仅是显示表格的外边框（不是单元格的）。
- ❖ margin:0px auto; /*块级元素可用 margin 居中*/
- ❖ text-align:right; 对行内不管用

七十二、布局

LICHOO

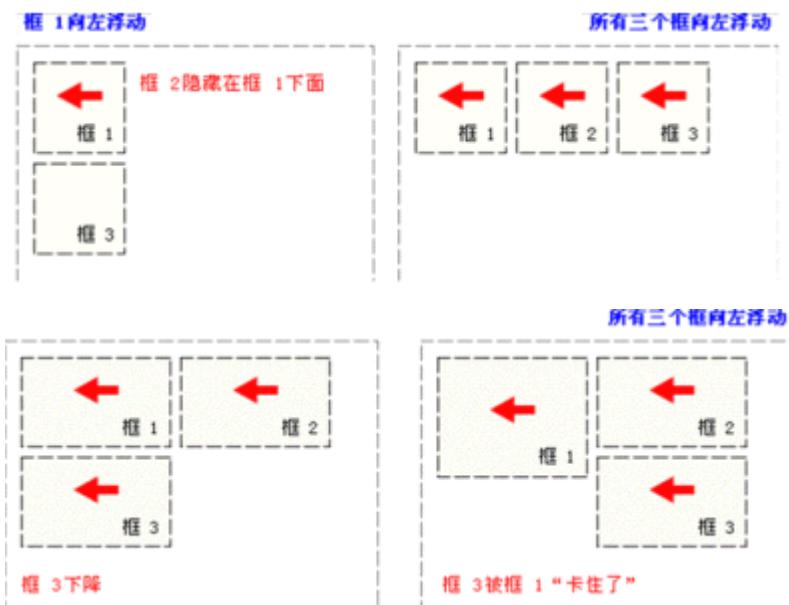
10.1 浮动定位说明图



10.2 什么是浮动定位

- 1) 将元素排除在普通流之外。
 - 2) 将浮动元素放置在包含框的左边或者右边。
 - 3) 浮动元素依旧位于包含框之内！
 - 4) 浮动的框可以向左或向右移动，直到它的外边缘碰到包含框或另一个浮动框为止！
- ◆ 注意事项：流模式：从上到下，从左到右

10.3 浮动定位移动图



10.4 float 属性

- 1) 语法：设置浮动，float:none/left/right;
- 2) 取值说明：
 - ①none：默认值。对象不飘浮
 - ②left：文本流向对象的右边
 - ③right：文本流向对象的左边

- ◆ 注意事项：
 - ❖ 多元素浮动，建议高度都一样。
 - ❖ 多元素飘在同一行，要确认总长度不超过父元素的长度。
 - ❖ 某个元素浮动，可能会影响后续的元素，后续的元素自动向上补齐！

10.5 clear 属性

clear 属性用于清除某测浮动元素所带来的影响。

1) 语法： clear:none/left/right/both;

2) 取值说明：

- | | |
|-------------------------|-------------------|
| ①none: 默认值。允许两边都可以有浮动对象 | ②left: 不允许左边有浮动对象 |
| ③right: 不允许右边有浮动对象 | ④both: 不允许有浮动对象 |

例如： <div style="float:left; border:1px solid black;">div text</div>

<p>p text</p> ---> <p style="clear:both;">p text</p>

10.6 display 属性

设置元素的显示方式。

1) HTML 元素分为行内元素和块级元素

①行内元素：位于一行中，高和宽自适应。②块级元素：独占一行，自定义高和宽。
所以有些时候，希望改变元素的显示方式。

例如： a 的大小自适应，它是行内元素 inline。

div 就可以用 width/height 属性，它是块级元素 block。

2) 语法： display:inline/block/none;

3) 取值说明：

①block: 将元素转为块级元素。②inline: 将元素转为行级元素。③none: 不显示。

◆ 注意事项：

- ❖ 常会结合 JS 实现一些动态的效果。
- ❖ li 属于块级元素。

七十三、列表样式

特定于列表元素。

11.1 列表项前的标识符号图像

- 1) 语法: `list-style-image:none/url(url);`
- 2) 取值说明:
 - ①`none`: 默认值, 不指定图像。
 - ②`url(url)`: 使用绝对或相对 url 地址指定图像。
 - ◆ 注意事项: 若此属性值为 `none` 或指定 url 地址的图片不能被显示时, `list-style-type` 属性将发生作用。

11.2 列表项前使用的预设标识符号

- 1) 语法: `list-style-type:disc/circle/square/decimal/none;`
- 2) 取值说明:

① <code>disc</code> : 默认值, 实心圆	② <code>circle</code> : 空心圆	③ <code>square</code> : 实心方块
④ <code>decimal</code> : 阿拉伯数字	⑤ <code>none</code> : 不使用项目符号	

七十四、定位

12.1 定位概述

- 1) 普通定位:
页面中的块级元素框从上到下一个接一个地排列, 每一个块级元素都会出现在一个新行中, 内联元素将在一行中从左到右排列水平布置。
- 2) 浮动定位: 见第十章。
- 3) 相对/绝对定位。

12.2 position (定位) 属性

更改定位模式为相对定位、绝对定位、固定定位

- 1) 语法: `position:static/absolute/fixed/relative;`
- 2) 取值说明:
 - ①`static`: 默认值。无特殊定位, 元素遵循 HTML 定位规则 (即默认的流布局模式)
 - ②`absolute`: 将元素从文档流中拖出, 使用 `left`、`right`、`top`、`bottom` 等属性相对于最近的有 `position` 属性的祖先元素, 如果没有, 那么它的位置相对最初的包含块, 比如按 `<body>` 进行绝对定位。而其层叠通过 `z-index` 属性定义
 - ③`relative`: 元素不可层叠, 但将依据 `left`、`right`、`top`、`bottom` 等属性在正常文档流中偏移位置
 - ④`fixed`: 元素定位遵从绝对定位, 但是要遵守一些规范。低版本的 IE 中, 这个属性无效

12.3 偏移属性

实现元素框位置的便宜。

- 1) 语法: `top/bottom/right/left:auto/length;`

2) 取值说明:

①auto: 默认值, 无特殊定位, 根据 HTML 定位规则在文档流中分配。

②length: 由浮点数字和单位标识符组成的长度值/百分数。必须定义 position 属性值为 absolute 或者 relative 此取值方可生效

12.4 堆叠属性

1) 语法: z-index:auto/number;

2) 取值说明:

①auto: 默认值, 为 0, 遵从其父元素的定位。

②number: 无单位的整数值, 可为负数。

◆ 注意事项:

- ❖ 较大 number 值的元素会覆盖在较小 number 值的元素之上。如两个绝对定位元素的此属性具有同样的 number 值, 那么将依据它们在 HTML 文档中声明的顺序层叠。
- ❖ 此属性仅仅作用于 position 属性值为 relative 或 absolute 的元素。
- ❖ 默认布局使用堆叠无效。

12.5 相对定位: relative

1) 元素仍保持其未定位前的形状。

2) 原本所占的空间仍保留。

3) 元素框会相对它原来的位置偏移某个距离。

4) 在相对定位元素之后的文本或元素占有他们自己的空间而不会覆盖被定位元素的自然空间。

5) 相对定位会保持元素在正常的 HTML 流中, 但是它的位置可以根据它的前一个元素进行偏移。

6) 在相对定位元素在可视区域之外, 滚动条不会出现。

12.6 绝对定位: absolute

1) 绝对定位会将元素拖离出正常的 HTML 流, 而不考虑它周围内容的布局。

2) 要激活元素的绝对定位, 必须指定 left、right、top、bottom 属性中的至少一个。

3) 绝对定位元素之后的文本或元素在被定位元素被拖离正常 HTML 流之前会占有它的自然空间。

4) 绝对定位元素在可视区域之外会导致滚动条出现。

8 JavaScript 学习笔记

七十五、JavaScript 概述

1.1 什么是 JavaScript

1) JavaScript 是一种网页编程技术, 用来向 HTML 页面添加动态交互效果。

2) JavaScript 是一种基于对象和事件驱动的解释性脚本语言，具有与 Java 和 C 语言类似的语法。

3) JavaScript 可直接嵌入 HTML 页面。由浏览器解释执行代码，不进行预编译。

1.2 JavaScript 发展史

1) JavaScript 的正式名称是“ECMAScript”，此标准由 ECMA 组织发展和维护。

2) ECMA-262 是正式的 JavaScript (Netscape) 和 JScript (Microsoft)。

3) 网景公司在 Netscape2.0 首先推出了 JavaScript。微软公司从 IE3.0 开始提供对客户端 JavaScript 的支持，并另取名为 JScript。

◆ 注意事项：与 Java 没任何关系。

1.3 JavaScript 的特点

1) 可以使用任何文本编辑工具编写，只需要浏览器就可以执行程序。

2) 解释执行：事先不解释，逐行执行。

3) 基于对象：内置大量现成对象。

4) 适宜：客户端数据计算、客户端表单合法性验证、浏览器事件的触发、网页特殊显示效果制作。

1.4 JavaScript 的定义方式

1) 方式一：直接定义在事件中

例如：`<input type="button" value="第一个按钮" onclick="alert('hello world');"/>`

2) 方式二：在页面上`<head></head>`标签中嵌入`<script></script>`标签，标签中放置 JavaScript 代码

例如：`<head>`

`<script language="javascript" type="text/javascript">`

`/*不指定语言，直接写代码，那么浏览器用哪种语言规则解释？依靠浏览器的默认设置*/`

`//封装方法`

`function firstMethod()//必须用关键字 function，不需要返回值`

`{ alert("he\ll\lo ja\nva");//不封装的话，一加载就运行，按从上往下解释执行`

`alert("\u4e00");//一，中文字符编码的开始`

`alert("\u9fa5");//中文字符编码的结束 } </script> </head>`

◆ 注意事项：

❖ alter 修改，alert 警告。

❖ onclick="alert('hello world');"字符串用单引号或双引号隔开，此处用单引号，因为会和前面的双引号成对。

3) 方式三：将代码写在单独的.js 文件中，在 html 页面的`<head>`里使用`<script>`引入

例如：`<head> <script language="javascript" src="myScript.js"></script> </head>`

1.5 JavaScript 的代码错误查看

1) 解释性代码，若代码错误，则页面无任何效果。

2) IE 浏览器：使用开发工具。

3) Firefox 浏览器：使用错误控制台查看。

1.6 注释

单行注释用 “//”， 多行注释用： /* */

LICHOOL

七十六、JavaScript 基础语法

LICHOO

2.1 编写 JavaScript 代码

- 1) 由 Unicode 字符集编写。
- 2) 语句：表达式、关键字、运算符组成；大小写敏感；使用分号结束。

2.2 常量、标识符和关键字

- 1) 常量：直接在程序中出现的数据值（字面量），用完就丢了。如 `alert("hello")`；
- 2) 标识符：由不以数字开头的字母、数字、下划线、\$组成。常用于表示函数、变量等的名称。不能和保留关键字重复，如 `break`、`if` 等。
- 3) 关键字：只有系统才能用的标识符。

2.3 变量

- 1) 变量声明：使用关键字 `var` 声明变量，如 `var x,y;`
 - ◆ 注意事项：以 `var` 关键字声明，声明的时候不确定类型，变量的类型以赋值为准。
例如：`var x,y,z; x=10; y="mary"; z=true;`
- 2) 变量初始化：使用等号赋值
 - ◆ 注意事项：没有初始化的变量则自动取值为 `undefined`，如：`var count=0;`
 - 3) 变量命名同标识符的规则，大小写敏感。

2.4 数据类型

- 1) 基本类型：`number` 数字、`string` 字符串、`boolean` 布尔
- 2) 特殊类型：`null` 空、`undefined` 未定义
- 3) 复杂类型：`array` 数组、`object` 对象

2.5 string 数据类型

- 1) 表示文本：由 Unicode 字符、数字、标点符号组成的序列。
- 2) 首尾由单引号或双引号括起来。
- 3) 特殊字符需要转义，用转义符\，如：`\n`, `\\"`, `\\'`, `\“`

例如：`var a = "欢迎来到\"JavaScript 世界\"";`

- ◆ 注意事项：可用在正则表达式，只允许录入汉字[^\u4e00-\u9fa5\$]，每个汉字都有转义符。

2.6 number 数据类型

- 1) 不区分整型数值和浮点型数值：所有数字都采用 64 位浮点格式存储，类似于 `double` 格式。
- 2) 整数：16 进制数前面加上 `0x`, 8 进制前面加 `0`。
- 3) 浮点数：使用小数点记录数据，如 `3.4`, `5.6`; 使用指数记录数据，如 `4.3e23=4.3X10^23`

2.7 boolean 数据类型

- 1) 仅有两个值：`true` 和 `false`；实际运算中 `true=1`, `false=0`
- 2) 多用于结构控制语句。

2.8 数据类型的隐式转换

- 1) JavaScript 属于松散类型的程序语言
 - ①变量在声明时不需要指定数据类型。
 - ②变量由赋值操作确定数据类型。
- 2) 不同类型数据在计算过程中会自动进行转换
 - ①数字+字符串：数字转换为字符串
 - ②数字+布尔值：true 转换为 1, false 转换为 0
 - ③字符串+布尔值：布尔值转换为字符串 true 或 false
 - ④布尔值+布尔值：布尔值转换为数值 1 或 0

```
例如: var s="a";      var n=1;      var b1=true;      var b2=false;
      alert(s + n);//a1    alert(s + b1);//atrue   alert(n + b1);//2    alert(b1 + b2);//1
```

2.9 数据类型转换函数

- 1) 转换方式:
 - ①隐式转换：直接转，默认的规则
 - ②显式转换：利用转换的方法
 - ◆ 注意事项：不建议用隐式转换。
- 2) 显式转换
 - ①toString: 转成字符串，所有数据类型均可转换为 string 类型。
 - ②parseInt: 强制转换成整数，如果不能转换，则返回 NaN。

例如：parseInt("6.12")=6 (无四舍五入)
 - ③parseFloat: 强制转换成浮点数，如果不能转换，则返回 NaN。

例如：parseFloat("6.12")=6.12
 - ④typeof: 查询数值当前类型，返回 string/number/boolean/object。

例如：typeof("test"+3)="string"
- 3) NaN: not a number，非常特殊，它不是数字，所以任何数跟它都不相等，甚至 NaN 本身也不等于 NaN
- 4) isNaN(str): is not a number，判断文本是否为数值，false 为数值，true 为非数值
- 5) 案例

eg1: 转换函数：得到录入数值的整数部分

```
<input type="text" id="txtData" />
<input type="button" value="得到录入数据的整数部分" onclick="getInt();"/>
function getInt() {
  var str = document.getElementById("txtData").value;
  if (isNaN(str)) alert("请录入数值");
  else {
    var data = parseInt(str);
    alert("整数部分为：" + data);
  }
}
```

eg2: 转换函数：计算录入数值的平方

```
<input type="text" id="txtData" />
<input type="button" value="计算平方" onclick="getSquare();"/>
function getSquare() {
  var str = document.getElementById("txtData").value;
  if (isNaN(str)) alert("请录入数值");
  else {
    var data = parseFloat(str);
    var result = data * data; alert(result);
  }
}
```

2.10 特殊类型

1) null: null 在程序中代表“无值”或者“无对象”。可以通过给一个变量赋值 null 来清除变量的内容。

2) undefined: 声明了变量但从未赋值或这对象属性不存在。

2.11 算术运算

1) 加 (+) 减 (-) 乘 (*) 除 (/) 余数 (%)

① “-”: 可以表示减号，也可以表示负号。

② “+”: 可以表示加法，也可以用于字符串的连接。

2) 递增 (++) 递减 (--)

i++等价于 i=i+1, i--等价于 i=i-1

2.12 关系运算

1) 用于判断数据之间的大小关系: “>”、“<”、“>=”、“<=”、“==”、“!=”

◆ 注意事项: “==”比较的是值(内容)。

2) 关系表达式的值为 boolean 类型 (“true”或“false”)

3) 严格相等: “==”类型、数值都相同。

4) 非严格相等: !=

例如: var a = "10"; var b = 10; if(a == b) alert("equal"); if(a === b) alert("same");

2.13 逻辑运算

1) 逻辑非 (!) 逻辑与 (&&) 逻辑或 (||)

2) 逻辑运算的操作数均为 boolean 表达式

b1	b2	b1&&b2	b1 b2	!b1
false	false	false	false	true
false	true	false	true	
true	false	false	true	false
true	true	true	true	

2.14 条件运算符

1) 条件运算符又称“三目”或“三元”运算符。

2) 语法: boolean 表达式?表达式 1:表达式 2

①先计算 boolean 表达式的值, 如果为 true, 则整个表达式的值为表达式 1 的值。

②如果为 false, 则整个表达式的值为表达式 2 的值。

eg: 猜数字

```
<input type="text" id="txtData" />
<input type="text" onblur="guessNumber(this.value);"/>!--this 代表当前对象-->
function guessNumber(str) { //内置结果    var result = 10;
    if (isNaN(str))    alert("请录入数值");
    else {    var data = parseFloat(str);
        var info = data > result ? "大了" : "小了";    alert(info);    }    }
```

2.15 流程控制语句

程序默认情况下顺序执行，改变或者控制执行顺序。

1) if 语句: ①if(表达式){语句块 1}else{语句块 2}

②if(表达式 1){语句块 1}else if(表达式 2){语句块 2}else{语句块 3}

2) switch-case 语句:

```
switch(表达式){  
    case 值 1:语句 1;break;  
    case 值 2:语句 2;break;  
    default:语句 3;  
}
```

3) for 语句:

```
for(初始化;条件;增量){  
    语句;  
}
```

◆ 注意事项：初始化中的局部变量用 var 声明。

4) while 语句

```
while(条件){  
    语句 1;  
}
```

◆ 注意事项：使用 break 或者 continue 中止循环

eg: 阶乘计算

```
<input type="button" value="求 10 的阶乘" onclick="getFac();"/>  
function getFac() {    var result = 1;  
    for (var i = 1; i <= 10; i++) {        result *= i;    }  
    alert("10 的阶乘为: " + result);}
```

七十七、JavaScript 常用内置对象

3.1 什么是 JavaScript 对象

- 1) JavaScript 是一种基于对象的语言，对象是 JavaScript 中最重要的元素。
- 2) JavaScript 包含多种对象：内置对象、自定义对象、浏览器对象、HTML DOM 对象、ActiveX 对象

3.2 使用对象

- 1) 对象由属性和方法封装而成。
- 2) 属性的引用：使用点“.”运算符、通过下标的方式引用。
- 3) 对象的方法的引用：ObjectName.methods()。

3.3 常用内置对象

- 1) 简单数据对象：String、Number、Boolean
- 2) 组合对象：Array、Date、Math
- 3) 复杂对象：Function、RegExp

3.4 String 对象

- 1) 创建字符串对象

方式一：var str1 = "hello world";
方式二：var str2 = new String("hello world");

- 2) String 对象的属性：length，例如：alert(str1.length);
- 3) String 对象的方法：

①大小写转换：toLowerCase(): 转为小写；toUpperCase(): 转为大写。

```
例如：var str1="AbcdEfgh";
      var str2=str1.toLowerCase();alert(str2);//结果为 abcdefgh
      var str3=str1.toUpperCase();alert(str3);//结果为 ABCDEFGH
```

②获取指定字符：charAt(index): 返回指定位置 index 的字符；
charCodeAt(index): 返回指定位置 index 的字符的 Unicode 编码

◆ 注意事项：下标 index 从 0 开始。

```
例如：var str1="JavaScript 网页教程";
      var str2=str1.charAt(12);alert(str2);//结果为 教
      var str3=str1.charCodeAt(12);alert(str3);//结果为 25945
```

③查询指定字符串：

indexOf(findstr,index): 从前往后，从位置 index 开始查找指定的字符串 findstr，并返回出现的首字符的位置。

lastIndexOf(findstr): 从后往前，查找指定的字符串 findstr，并返回出现的首字符的位置。

◆ 注意事项：index 可省略，代表从 0 开始找。如果没有找到则返回-1。

```
例如：var str1="JavaScript 网页教程";
      var str2=str1.indexOf("a");alert(str2);//结果为 1
      var str3=str1.lastIndexOf("a");alert(str3);//结果为 3
```

④获取子字符串：substring(start,end): 从 start 开始，到 end 结束，不包含 end。

例如： var str1="abcdefghijklm";
var str2=str1.substring(2,4);alert(str2);//结果为 cd

⑤替换子字符串： replace(oldstr,newstr): 返回替换后的字符串。

例如： var str1="abcde";
var str2=str1.replace("cd","aaa");alert(str2);//结果为 abaaee

⑥拆分子字符串： split(bystr): 用 bystr 分割字符串，并返回分割后的字符串数组。

例如： var str1="一,二,三,四,五,六,日";
var strArray=str1.split(",");alert(strArray[1]);//结果为 二

3.5 String 对象与正则表达式

String 对象有几个方法可以结合正则表达式使用。

1) 方法：

- ①replace(regexp,"replacestr"): 返回替换后的结果。
- ②match(regexp): 返回匹配字符串的数组。
- ③search(regexp): 得到匹配字符串的“首”字符位置的索引。

2) JavaScript 中使用正则表达式： 使用两个斜杠， / 表达式 / 匹配模式

①正则表达式回顾： \d 或者 [a-z]{3,5}。就是纯文本的表达式，用来表示某种匹配模式。在不同的语言环境下，提供了不同的类，执行或者使用正则表达式，实现文本的各种处理。

②匹配模式： g: global， 全局匹配； m: multilin， 多行匹配； i: 忽略大小写匹配。

例如： var str1="abc123def";
var str2=str1.replace(/\d/g,"*");alert(str2);//abc***def， 如果不用全局匹配
则只替换一个数字

var array=str1.match(/\d/g);//1,2,3
var index=str1.search(/\d/);alert(index);//3

3) 案例

eg1：查找并替换文本框中录入的子字符串 js 为 *

```
<input type="text" id="txtString" />  
<input type="button" value="过滤特殊字符 (js)" onclick="searchStringAndReplace();"/>  
function searchStringAndReplace() { var str = document.getElementById("txtString").value;  
var count = 0; var index = str.indexOf("js", 0);  
while (index > -1) { str = str.replace("js", "*"); index = str.indexOf("js", index + 1); }  
document.getElementById("txtString").value = str; }
```

eg2：字符查询与过滤（使用正则表达式）

```
<input type="text" id="txtString" /><br />  
<input type="button" value="查找字符并过滤" onclick="stringByRegex();"/>  
function stringByRegex() { var str = document.getElementById("txtString").value;  
var result = str.match(/js/gi);  
document.getElementById("txtString").value = str.replace(/js/gi, "*");  
alert("共替换了" + result.length + "处。"); }
```

3.6 Array 对象

一列有多个数据。JavaScript 中只有数组没有集合。

1) 创建方式

方式一： var arr = ["mary",10,true];//用中括号！不是大括号；常用；声明的同时并赋值。

方式二： var arr = new Array("mary",10,true);//声明的同时并赋值。

方式三： var arr = new Array();或 var arr = new Array(7); //可有长度也可没有长度。即便写了长度，也能把第 8 个数据存入！先声明后赋值。

```
例如：arr[0] = "mary"; arr[1] = 10; arr[2] = true; alert(arr[3]);//undefined
```

2) 数组的属性： length

3) 创建二维数组：通过指定数组中的元素为数组的方式可以创建二维甚至多维数组。

```
例如：var week=new Array(7); for(var i=0;i<=6;i++){ week[i]=new Array(2); }  
week[0][0]="星期日";week[0][1]="Sunday"; .....  
week[6][0]="星期六";week[6][1]="Saturday";
```

4) 方法：数组转换为字符串

①join(bystr): 以 bystr 作为连接数组中元素的分隔字符，返回拼接后的数组。

②toString(): 输出数组的内容（以逗号隔开）。

```
例如：var arr1=[1,2,3,4]; alert(arr1.toString());//1,2,3,4 alert(arr1.join("-"));//1-2-3-4
```

5) 方法：连接数组， concat(value,...): value 作为数组元素连接到数组的末尾（数组连接），返回连接后的数组。

```
例如：var a=[1,2,3]; var b=a.concat(4,5); alert(a.toString());//1,2,3  
alert(b.toString());//1,2,3,4,5
```

◆ 注意事项：concat 方法并不改变原来数组的内容！

6) 方法：获取子数组， slice(start,end): 截取从 start 开始到 end 结束（不包含 end）的数组元素。end 省略代表从 start 开始到数组结尾。

```
例如：var arr1=['a','b','c','d','e','f','g','h'];  
var arr2=arr1.slice(2,4);alert(arr2.toString());//c,d  
var arr3=arr1.slice(4);alert(arr3.toString());//e,f,g,h
```

7) 方法：数组反转， reverse(): 改变原数组元素的顺序。

```
例如：var arr1=[32,12,111,444]; alert(arr1.reverse());//444,111,12,32
```

8) 方法：数组排序

①sort(): 数组排序，默认按照字符串的编码顺序进行排序。

②sort(sortfunc): sortfunc 用来确定元素顺序的函数名程。

```
例如：var arr1=[32,12,111,444]; arr1.sort();alert(arr1.toString());//111,12,32,444  
function sortFunc(a,b){ return a-b; }  
arr1.sort(sortFunc);alert(arr1.toString());//12,32,111,444
```

9) 案例

eg1：数组倒转与排序

```
<input type="text" id="txtNumbers" value="12,4,3,123,51" />  
<input type="button" value="数组倒转" onclick="operateArray(1);"/>  
<input type="button" value="数组排序（文本）" onclick="operateArray(2);"/>  
<input type="button" value="数组排序（数值）" onclick="operateArray(3);"/>  
function operateArray(t) {
```

```

//拆分为数组
var array = document.getElementById("txtNumbers").value.split(",");
switch (t) {
    case 1: array.reverse(); break;
    case 2: array.sort(); break;
    case 3: array.sort(sortFunc); break;
}
alert(array.toString());
function sortFunc(a, b) { return a - b; }

```

eg2：统计文本框中录入的各个字符的各数（使用二维数组）

```

<h2>统计文本框中录入的各字符的个数</h2>
<input type="text" onblur="countString(this.value);"/>
function countString(str) {
    var result = new Array();
    for (var i = 0; i < str.length; i++) { //直接循环 str
        var curChar = str.charAt(i); //得到当前字符
        var isHas = false; //声明一个变量，标识 char 在结果中是否出现过
        for (var j = 0; j < result.length; j++) { //循环判断当前字符是否在 result 中出现过
            //如果出现过，则设置标识为 true，并增加数量，最后跳出循环
            if (curChar == result[j][0]) {
                isHas = true; result[j][1]++; break; } }
        if (!isHas) //循环 result 完毕，没有出现过，则加入 result
            result.push(new Array(curChar, 1));
    }
    alert(result.toString());
}

```

eg3：彩票双色球生成器

```

<h2>彩票双色球生成器</h2>
<input type="button" value="机选" onclick="doubleBall();"/>
function randomNumber(min, max) { //随机数，包含下限，不包含上限
    var n = Math.floor(Math.random() * (max - min)) + min; return n;
}
function doubleBall() { //彩票双色球
    var result = new Array(); //声明一个数组，用于存放结果
    var i = 0; while (i < 6) { //先产生 6 个红球
        var curCode = randomNumber(1, 34); //先生成一个 1 到 33 之间的号码
        var isHas = false; //判断该号码是否出现过
        for (var j = 0; j < result.length; j++) {
            if (result[j] == curCode) { isHas = true; break; } }
        if (!isHas) //没有出现过，则加入且计数器加 1
            result.push(curCode); i++; }
    result.sort(sortFunc); //产生完 6 个红球后，先排序，再产生一个蓝球
    var info = result.toString(); var blueBall = randomNumber(1, 17);
    alert(info + " | " + blueBall); //返回结果
}
function sortFunc(a, b) { return a - b; } //排序用的方法

```

3.7 Math 对象

用于执行相关的数学计算。

1) 没有构造函数 Math()。

2) 不需要创建，直接把 Math 作为对象使用就可以调用其所有属性和方法。如：不需要创建 var data=Math.PI; 直接使用 Math.PI; 像 Java 中的静态类一样。

3) 常用属性：都是数学常数，如：Math.E（自然数）、Math.PI（圆周率）、Math.LN2（ln2）、Math.LN10（ln10）等

4) 常用方法：

①三角函数：Math.sin(x)、Math.cos(x)、Math.tan(x)等

②反三角函数：Math.asin(x)、Math.acos(x)等

③计算函数：Math.sqrt(x)、Math.log(x)、Math.exp(x)等

④数值比较函数：Math.abs(x)、Math.max(x,y,...)、Math.random()、Math.round(x)等

◆ 注意事项：

❖ Math.random(): 是一个 ≥ 0 且 <1 的正小数

❖ Math.floor(x): 地板，小于等于 x，并且与它最接近的整数。注意：将负数舍入为更小的负数，而不是向 0 进行舍入。

❖ Math.ceil(x): 天花板，返回大于等于 x，并且与它最接近的整数。注意：不会将负数舍入为更小的负数，而是向 0 舍入。

5) 案例

eg：随机得到 3-9 之间的一个整数（包含 3，不包含 9）

分析：① * (9-3) 则得到 0-6 之间的小数。② +3 则得到 3-9 之间的小数。

```
function getRandom(){  var min = 3;  var max = 9;  var seed = Math.random();  var n = Math.floor( seed * (max-min) + min);  alert(n); }
```

3.8 Number 对象

Number 对象是原始数值的包装对象。

1) 创建 Number 对象

方式一：var myNum=new Number(value);

方式二：var myNum=Number(value);

2) 常用方法

①toString：数值转换为字符串。

②toFixed(n)：数值转换为字符串，并保留小数点后 n 位数，多了就截断，四舍五入。少了就用 0 补足，常用作数值的格式化。比如：

```
var data=23.56789; alert(data.toFixed(2));//23.57 data=23.5; alert(data.toFixed(2));//23.50
```

3.9 RegExp 正则表达式对象

1) 创建正则表达式对象

方式一：var reg1=/^d{3,6}\$/;

方式二：var reg2=new RegExp("^\d{3,6}\$");

2) JavaScript 中，正则表达式的应用分为两类：

一类：和 String 对象的三个方法结合使用，实现对 string 的操作，如：replace/match/search

二类：调用正则表达式对象的常用 test 方法测试，RegExpObject.test(string)：如果字符串 string 中含有与 RegExpObject 匹配的文本，则返回 true，否则返回 false

3) 案例

eg: 输入验证

用户名 (3 到 6 位字母数字的组合):

```
<input type="text" onblur="validateUserName(this.value);"/><br/>
```

电话号码 (6 位数字):

```
<input type="text" onblur="validateUserPhone(this.value);"/><br/><!--this 代表当前对象-->
```

```
--> function validateUserName(name) { //验证用户名
```

```
    var reg = /^[a-zA-Z0-9]{3,6}$/; //reg 是一个对象
```

```
    var isRight = reg.test(name);      if (!isRight){ alert("录入错误!"); } }
```

```
function validateUserPhone(phone) { //验证电话号码
```

```
    var reg = /\d{6}/; var isRight = reg.test(phone);
```

```
    if (!isRight){ alert("录入错误!"); } }
```

3.10 Date 对象

Data 对象用于处理日期和时间。

1) 创建 Data 对象

方式一: var now = new Date(); //当前日期和时间

方式二: var now = new Date("2013/01/01 12:12:12");

2) 常用方法

①读取日期的相关信息: getDay()、getDate()、getMonth()、getFullYear()...

②修改日期: setDate()、setDate()、setMonth()、setHours()

③转换为字符串: 得到某种格式的字符串显式, 常用于页面显式, 如: toString(),
toDateString()、toLocaleTimeString()

3) 案例

eg: 计算查询时段

```
<input type="radio" onclick="getDateRange(3); name="date" />三天内
```

```
<input type="radio" onclick="getDateRange(7); name="date" />一周内
```

```
function getDateRange(days) { //时间的查询
```

```
    var end = new Date(); //得到当前日期
```

```
    var endString = end.toLocaleDateString();
```

```
    end.setDate(end.getDate() - days + 1); //修改日期
```

```
    var s = "开始日期为: " + end.toLocaleDateString() + "\n"; //显示结果
```

```
    s += "结束日期为: " + endString; alert(s); }
```

3.11 函数与 Function 对象

1) 函数的概述: 函数 (方法) 是一组可以随时随地运行的语句。Function 对象可以表示开发者定义的任何函数。函数实际上是功能完整的对象。

2) 函数的定义: function 函数名(参数){ 函数体; return; }

◆ 注意事项:

❖ 由关键字 function 定义。

❖ 函数名的定义规则与标识符一致, 大小写敏感。

❖ 可以使用变量、常量或表达式作为函数调用的参数。

❖ 返回值必须使用 return。

3) 函数的调用: 函数可以通过其名字加上括号中的参数进行调用。如果有多个参数,

则参数之间用逗号隔开。如果函数有返回值，则可以声明变量等于函数的结果即可。比如：

```
function sum(n1,n2){    return n1+n1;    }    var result=sum(1,1);    alert(result);//2
```

4) JavaScript 中，如何创建一个方法

方式一：使用 `function` 关键字明文的声明一个方法（最常用，用于功能相关的方法）

```
例如：function AA(a,b){    alert(a+b);    }
```

方式二：使用 `Function` 对象创建函数，语法：

```
var functionName=new Function(arg1,...argN,functionBody);
```

最后一个参数是方法体，前面的其它参数是方法的参数。

例如：`<input type="button" value="使用 Function 对象创建函数" onclick="testFunction();"/>`

```
function testFunction(){    var f = new Function("a","b","alert(a+b);");    f(10,20);    }
```

◆ 注意事项：

- ❖ 需求：有些方法不需要显式的存在，只是为其他方法里所使用（类似于 Java 中的内部类）。
- ❖ 缺陷：方法体不能复杂。

方式三：创建匿名函数，语法：

```
var func=function(arg1,...argN){    func_body;    return value;    }
```

```
例如：var f = function(a,b){    alert(a+b);    }
```

5) 案例

eg：数组按数值排序（使用 `Function` 对象和匿名函数）

```
<input type="button" value="排序" onclick="sortArray();"/>
function sortArray() {//使用 Function 对象
    var array = [34, 2, 14, 56, 43];
    array.sort(new Function("a", "b", "return a-b;"));
    alert(array.toString());
}

function sortArray2{//使用匿名函数
    var array=[12,3,45,9];
    var f = function(a,b){    return a-b;    };
    array.sort(f);
    alert(array.toString());
}
```

3.12 全局函数

全局函数可用于所有内建的 JavaScript 对象。常用的全局函数有：

1) decodeURI/encodeURI

①`encodeURI(str)`：把字符串作为 URI 进行编码（大写 i）。

②`decodeURI(str)`：对 `encodeURI()` 函数编码过的 URI 进行解码。

例如：`function test(){ var s="http://sss.jsp?name=张三&code=李四"; var r1=encodeURI(s); var r2=decodeURI(r1); alert(r1); alert(r2); }`

2) parseInt/parseFloat

①`parseInt(str)`：强制转换成整数，如果不能转换，则返回 `Nan`。

例如：`parseInt("6.12")=6`（无四舍五入）

②`parseFloat(str)`：牵制转换成浮点数，如果不能转换，则返回 `Nan`。

例如：`parseFloat("6.12")=6.12`

3) isNaN(str)：is not a number，判断文本是否为数值，`false` 为数值，`true` 为非数值。

4) eval(str)：用于计算某个字符串，以得到结果；或者用于执行其中的 JavaScript 代码。

◆ 注意事项：`eval(str)` 只接受原始字符串作为参数，如果参数中没有合法的表达式

式和语句，则抛出异常

例如： var str="2+3"; alert(str);//2+3 alert(eval(str));//5

5) 案例

eg: 简单计算器

```
<input type="button" value="1" onclick="calculate(this.value);"/>
<input type="button" value="2" onclick="calculate(this.value);"/>
<input type="button" value="3" onclick="calculate(this.value);"/>
<input type="button" value="4" onclick="calculate(this.value);"/>
<input type="button" value="+" onclick="calculate(this.value);"/>
<input type="button" value="-" onclick="calculate(this.value);"/>
<input type="button" value="/" onclick="calculate(this.value);"/><br />
<input type="text" id="txtNumber" />

function calculate(s){
    if( s == "="){//如果单击的是=则计算，否则拼接
        var s1 = document.getElementById("txtNumber").value;
        var r = eval(s1);
        document.getElementById("txtNumber").value = r;
    }else{    document.getElementById("txtNumber").value +=s;    }
}
```

3.13 Arguments 对象

1) arguments 是一种特殊对象，在函数代码中，代表当前方法被传入的所有的参数，形成一个数组。

2) 在函数代码中，可以使用 arguments 访问所有参数。

①arguments.length: 函数的参数个数； ②arguments.[i]: 第 i 个参数

3) JavaScript 中，没有传统意义上的重载（方法名相同，但是参数不同），即：如果方法名相同，则以最后一次定义的为准。如果想在 JavaScript 中实现类似于重载的效果，就需要使用 arguments 对象。

4) 案例

eg: 模拟一个方法的重载

```
<input type="button" value="计算平方" onclick="myMethod(12);"/>
<input type="button" value="计算加法" onclick="myMethod(12,45);"/>
//Javascript 代码中，相同名称的方法如果重复定义，将会用新定义的方法覆盖已有的同名方法，因此，只能创建一个名为 myMethod 的方法，且不需要为该方法定制参数。
function myMethod() {
    if (arguments.length == 1) {//计算平方
        var n = parseInt(arguments[0]);//避免隐式转换，主动显式转换
        alert(n + " 的平方为：" + n * n);
    } else if (arguments.length == 2) {//计算和
        var n = parseInt(arguments[0]);
        var m = parseInt(arguments[1]);
        var result = n + m;
        alert(n + " 与 " + m + " 的和为：" + result);
    }
}
```

七十八、window 对象

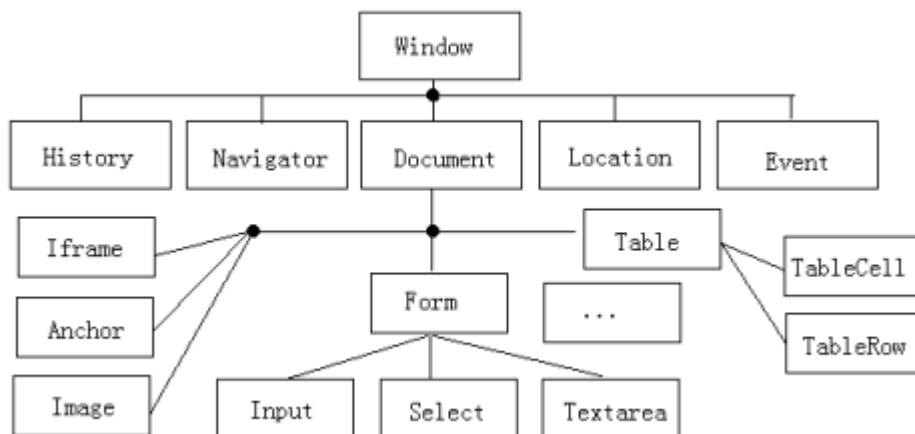
LICHOO

4.1 DHTML 简介

- 1) 操作 HTML 以创造各种动态视觉效果，是一种浏览器端的动态网页技术。
- 2) DHTML 的功能：
 - ① 动态改变页面元素。
 - ② 事件响应机制制作动态折叠的树形结构和菜单。
 - ③ 与用户进行交互等。
- 3) DHTML 对象模型包括浏览器对象模型和 DOM 对象模型。

4.2 DHTML 对象模型

将整个窗口均实现了对象化，结构如下：



4.3 window 对象

window 对象表示浏览器中打开的窗口。也是父对象。

- 1) 常用属性
 - ① name: 窗口名称。
 - ② opener: 打开当前窗口的 window 对象（引用）。
 - ③ status: 窗口底部状态栏信息。
- 2) 常用子对象
 - ① document: 代表给定浏览器窗口中的 HTML 文档。
 - ② history: 包含了用户浏览过的 URL 信息。
 - ③ location: 包含关于当前 URL 的信息。
 - ④ navigator: 包含 Web 浏览器的信息。
 - ⑤ screen: 包含关于客户屏幕和渲染能力的信息。
 - ⑥ event: 代表事件状态。

4.4 常用方法：对话框

- 1) alert(str): 提示对话框，显示 str 字符串的内容。

例如：window.alert("aa"); //window. 可省

- 2) confirm(str): 确认对话框，像是 str 字符串的内容，按“确定”返回 true，其他操作返回 false。

3) prompt(str,value): 输入对话框，采用文本框输入信息，str为提示信息，value为初始值，按“确定”返回输入值，其他操作返回 undefined，value 可省。

```
例如: function testConfirm(){  
    var r = window.confirm("Are you really...");  
    alert(r); window.prompt("请输入 ID: ");//因为不能控制它，所以很少用 }
```

4.5 常用方法：窗口的打开和关闭

- 1) window.open(url): 重复打开。
- 2) window.open(url,name): 采取命名方式，避免重复打开。
- 3) window.open(url,name,config): config 设置新窗口外观如高和宽。
- 4) window.close(): 关闭窗口。

```
例如: function testNewWindow(){  
    var config="toolbar=no,location=no,width=300,height=200";  
    window.open("http://www.baidu.com","a",config); }
```

4.6 常用方法：周期性定时器

- 1) setInterval(exp,time): 周期性触发代码 exp，返回已经启动的定时器。exp: 执行语句，time: 时间周期，单位为毫秒。
- 2) clearInterval(tObj): 停止启动的定时器。tObj: 启动的定时器对象。

4.7 常用方法：一次性定时器

- 1) setTimeout(exp,time): 一次性触发代码 exp，返回已经启动的定时器。exp: 执行语句，time: 时间周期，单位为毫秒。
- 2) clearTimeout(tObj): 停止启动的定时器。tObj: 启动的定时器对象。

4.8 案例：动态时钟

```
<input type="button" value="显式时间" onclick="showTime();"/>  
<input type="button" value="启动时钟" onclick="startClock();"/>  
<input type="button" value="停止时钟" onclick="stopClock();"/>  
<input type="button" value="5s 后弹出一个 Hello" onclick="wait();"/>  
  
function showTime() {//显式时间  
    var t = new Date();//将当前时间显式在一个文本框中  
    document.getElementById("txtTime").value = t.toLocaleTimeString(); }  
  
var timer;//设置全局变量  
function startClock() {//启动时钟  
    timer = window.setInterval(showTime, 1000); //有返回值的。 }  
  
function stopClock() {//停止时钟  
    window.clearInterval(timer); }  
  
var timer1;//设置全局变量  
function wait() {//5 秒后弹出一个 Hello  
    timer1 = window.setTimeout("alert('Hello')", 5000); //可以执行合法的文本表达式 }
```

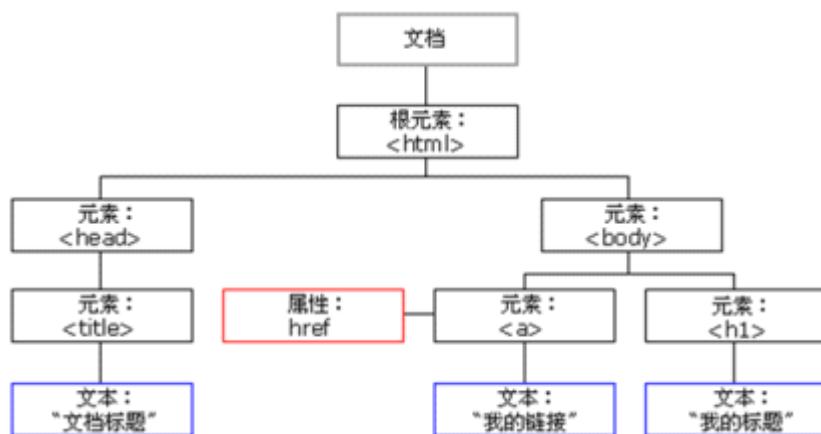
- ◆ 注意事项：showTime 没有括号，代表每隔一定时间让浏览器去找 showTime 对象并启动一次；showTime()则为立即启动方法。

七十九、Document 对象与 DOM

LICHOO

5.1 概念

- 1) Document 对象: 每个载入浏览器的 HTML 文档都会成为 Document 对象。
 - ①通过使用 Document 对象, 可以从脚本中对 HTML 页面中的所有元素进行访问(操作文档中的任何内容, 都过 Document)。
 - ②Document 对象是 window 对象的一部分, 可通过 window.document 方式对其进行访问。
- 2) DOM: Document Object Model, 文档对象模型, HTML 文档中的所有节点组成了一个文档树 (或节点树)。
 - ①树起始于文档节点, Document 对象是一颗文档树的根。
 - ②HTML 文档中的每个元素、属性、文本等都代表这树中的一个节点。



5.2 根据元素 ID 查找节点

- 1) 方法: document.getElementById(idValue)
 - 2) 忽略文档的结构, 通过指定的 ID 来返回元素节点。
 - 3) 可以查找整个 HTML 文档中的任何 HTML 元素。
- ◆ 注意事项: 如果 ID 值错误, 则返回 null。

5.3 根据层次查找节点

- 1) 遵循文档的层次结构, 查找单个节点: parentNode、firstChild、lastChild
- 2) 遵循文档的层次结构, 查找多个节点:

childNodes: 以 s 结尾的都是数组, 则有 length 属性。

- 3) 案例

eg: 根据层次查找节点

```
<select id="sell">
    <option>a</option>
    <option>b</option>
    <option>c</option>
</select><!-- 注意按此格式 -->
var sObj=document.getElementById("sell"); alert(sObj.childNodes.length);//7? 为何是 7? !
var s1=sObj.firstChild;      alert(s1.innerHTML);//undefined? 这又为何为未定义?
```

解释：childNodes 会找到 sell 下的所有子节点，除了真正的子节点外，还有空格这种特殊节点，所以长度为 7。只有不空格，都写一行结果才为 3，如：

```
<select id="sell"><option>a</option><option>b</option><option>c</option></select>
```

同理，第一个子节点为空格，firstChild 的内容当然也就是 undefined 了。

5.4 根据标签名称查找节点

1) getElementsByTagName("namestr")：在某个节点的所有后代里查找某种类型的元素（根据指定的标签名），并返回所有的元素（返回一个节点列表）。

①忽略文档的结构，查找整个 HTML 文档中的所有元素。

②如果标签名错误，则返回长度为 0 的节点列表。

◆ 注意事项：单词 Elements 结尾有 s，所以为数组。

2) length 属性：返回的是一个节点列表，是个数组，因此可用 length 属性获取元素个数。使用[index]可定位具体的元素。例如：

```
<body>      <span>文本</span>      <h1>一周畅销<span>榜</span></h1>      </body>
var spans=document.getElementsByTagName("span");
alert(spans.length);//2  alert(spans[1].innerHTML);//榜
```

5.5 读取或者修改节点信息

1) 规则一：将 HTML 标签对象化，操作前先明确被对象化的标签都有什么属性。

如：aObj 代表一个[元素](#)，有 aObj.href 属性，但没有 aObj.value 和 aObj.src

2) 规则二：innerHTML：设置或获取位于对象起始和结束标签内的内容。

如：aObj.innerHTML 可修改[元素](#)标签中的“元素”两字

◆ 注意事项：单标签无法用 innerHTML 修改内容。

3) 规则三：nodeName：得到某个元素节点和属性节点（即可得到标签或属性名称）及其类型位置。xxx.nodeName：当未知节点类型时，使用该属性获得节点的名称，全大写方式。

如：if(xxx.nodeName == "IMG") xxx.src = "http://...";

4) 节点属性：getAttribute()方法：根据属性名称获取属性的值。

将 HTML 标签、属性、CSS 样式都对象化。

5.6 修改节点的样式

1) style 属性：语法：node.style.color; node.style.fontSize

◆ 注意事项：CSS 样式中属性名内有“-”的，这里省略，并把后面单词首字母大写！

2) className 属性：语法：var Obj=document.getElementById("p1");

如：Obj.className="样式类名称";//可用于设置不同的样式。

5.7 查找并修改节点

1) 使用 getElementById()方法找到元素节点。

2) 修改元素内容：innerHTML

3) 修改样式：style 属性或 className 属性

4) 修改属性：html 属性

5.8 三个案例

eg1：表单验证

```

<form>
    Name: <input type="text" id="txtName" onblur="validName();"/>
          <span id="nameInfo">3-5 个小写字母</span><br />
    Age: <input type="text" id="txtAge" onblur="validAge();"/>
          <span id="ageInfo">2 位数字</span><br />
    <input type="submit" value="保存" onclick="return validDatas();"/>
</form>

function validName(){//验证用户名
    var name = document.getElementById("txtName").value;//得到录入的用户名
    var r = /^[a-z]{3,5}$/;//验证
    if(r.test(name))//根据验证结果显示不同的样式
        document.getElementById("nameInfo").className = "success";
    else
        document.getElementById("nameInfo").className = "fail";
    return r.test(name);//添加返回
}

function validAge(){//验证年龄
    var age = document.getElementById("txtAge").value;//得到录入的年龄
    var r = /^[0-9]{2}$/;//验证
    if(r.test(age))//根据验证结果显示不同的样式
        document.getElementById("ageInfo").className = "success";
    else
        document.getElementById("ageInfo").className = "fail";
    return r.test(age);//添加返回
}

function validDatas(){//验证所有数据，验证 name 和 age，return true 和 false
    var r1 = validName();  var r2 = validAge();  return r1&&r2
}

```

- ◆ 注意事项： onclick="return validDatas(); 返回值为 true 则提交表单，否则取消提交，即为 return false 时取消某事件。

eg2：读取或修改节点信息

```

<input type="button" value="读取或修改节点信息" onclick="testDocument();"/>
<h2 id="h2">h2 文本</h2><p id="p1">段落文本</p>
h2.style1{ border-top:1px solid red; border-right:2px solid blue; }
function testDocument(){
    var imgObj = document.getElementById("img1");
    imgObj.src = "ok.png";//修改图片
    //修改段落：字体颜色，背景色，字体大小，段落文本
    var pObj = document.getElementById("p1");
    pObj.style.color = "red";//注： pObj.style = "color:red"; 不够对象化
    pObj.style.backgroundColor = "silver";
    //注： pObj.style.background-color 不认识减号，去掉减号，下个单词首字母大即可。
    pObj.style.fontSize = "25";//单位由浏览器的默认值决定
    pObj.innerHTML = "new text";
    //修改某元素的样式：样式复杂时用
}

```

```

document.getElementById("h2").className = "style1";
//复杂样式先定义一个样式类 h2.style1{} (定义在内部样式或外部样式都可以), 再用
className 操作, 不能用 class, 被系统用了。正常情况是标签中直接写 class="style1", 但为
了实现动态效果而使用 className 进行操作。
}

```

eg3: 购物车

```

<form> <table border="1" id="shoppingCart">
    <tr> <td>价格</td><td>数量</td><td>小计</td> </tr>
    <tr> <td>10.00</td>
        <td><input type="button" value="-" onclick="sub(this);"/>
            <input type="text" value="1"/>
            <input type="button" value "+" onclick="add(this);"/></td>
        <td>10.00</td> </tr>
    <tr> <td>20.00</td>
        <td><input type="button" value="-" onclick="sub(this);"/>
            <input type="text" value="1"/>
            <input type="button" value "+" onclick="add(this);"/></td>
        <td>20.00</td> </tr> </table>
    总计: <span id="spanTotal">00.00</span> </form>
<!--总计不能写到表格中, 与前几行格式不一致, 取得的数为 null 或未定义, 无法计算-->
function add(btnObj){//增加购物的数量
    var tdObj = btnObj.parentNode;//得到当前按钮所在的 td
    for(var i=0;i<tdObj.childNodes.length;i++){//得到 td 的所有子元素, 找到那个文本框
        var childNode = tdObj.childNodes[i];
        if(childNode.nodeName == "INPUT" && childNode.type == "text"){
            var oldData = parseInt(childNode.value);//得到旧数据, 操作, 并显示
            oldData++; childNode.value = oldData; }
    }
    calTotal();
}
function sub(btnObj){//减少购物的数量, 代码与增加一样, 仅仅将 oldDate--即可!
}
function calTotal(){//计算小计和总计
    var tableObj = document.getElementById("shoppingCart");//得到表格的所有行
    var rows = tableObj.getElementsByTagName("tr");
    var total = 0; for(var i=1;i<rows.length;i++){//循环从第二行开始
        var curRow = rows[i];
        //得到第一格中的价格
        var price = curRow.getElementsByTagName("td")[0].innerHTML;
        //得到第二格中的第二个 input 的 value
        var q = curRow.getElementsByTagName("td")[1].getElementsByTagName("input")[1].value;
        var sum = parseFloat(price) * parseFloat(q);//算完, 写入第三格
        curRow.getElementsByTagName("td")[2].innerHTML =
sum.toFixed(2);
        total += sum;//总计
    }
    document.getElementById("spanTotal").innerHTML = total.toFixed(2);//显示
}

```

5.9 增加新节点

1) 步骤:

第一步：创建一个新元素，`document.createElement("elementName")`;比如：元素名可为 `a/input(option)` 等；返回新创建的节点。

第二步：为新元素设置相关数据，比如：

```
var newNode=document.createElement("input");
```

```
newNode.type="text";  newNode.value="mary";    newNode.style.color="red";
```

第三步：把新节点加入树上（新元素加入文档），作为树上某个节点的子节点存在。

```
xxx.appendChild(newNode);//追加
```

```
xxx.insertBefore(newNode,oldNode);//新节点放在旧节点之前
```

2) 案例

eg: 添加新节点

```
<form id="form1">
<input type="button" value="新节点" onclick="addNewNode();"/>      </form>
function addNewNode(){
    var formObj = document.getElementById("form1");
    var obj = document.createElement("input");//为 form 添加一个文本框
    obj.value = "mary";           formObj.appendChild(obj);
    var aObj = document.createElement("a");//form 最后添加一个超级连接
    aObj.href = "http://tts6.tarena.com.cn";    aObj.innerHTML = "click me";
    formObj.appendChild(aObj);
    var btnObj = document.createElement("input");//在原有按钮的前面加入一个按钮
    btnObj.type = "button";    btnObj.value = "new button";
    btnObj.onclick = function() {    alert("hello");    };//function 方法
    formObj.insertBefore(btnObj,formObj.firstChild); //由于第一个子节点为空白，所以此处 oldNode 可用 firstChild 去定位（放在空白前）。
}
```

5.10 删 除 节 点

1) 语 法: `parentNode.removeChild(childNode);`

◆ 注意事项：一定是从父节点删除子节点，不能直接删除子节点。

例如：<input id="btn1" type="button" value="删除节点" onclick="deleteNode();"/>

```
<a id="a1" href="#">link1</a>
function deleteNode(){
    var delNode=document.getElementById("a1");
    delNode.parentNode.removeChild(delNode);
}
```

5.11 案例：联动菜单

```
<select id="sel1" onchange="showCities();">
<option>请选择</option>
<option>北京</option>
<option>河北</option>
</select>
```

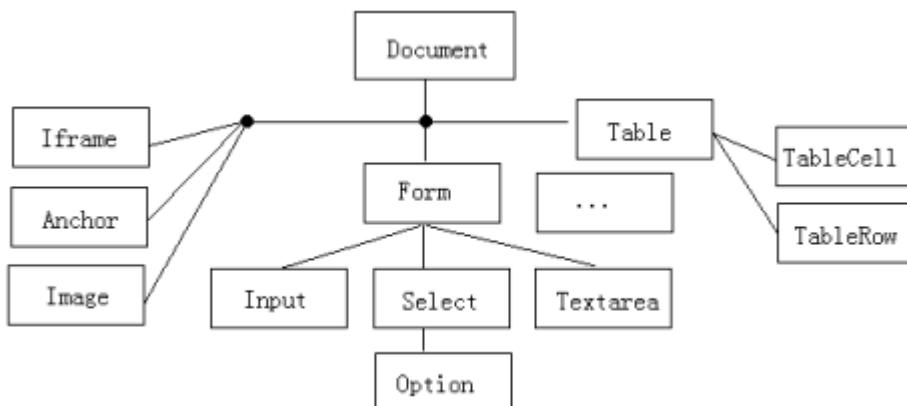
```
<select id="sel2">
    <option>请选择</option>
</select>
//以下为 JS 代码： 声明一个数组类型的全局变量用于存储所有的城市数据
var array = new Array();
array[0] = ["请选择"];
array[1] = ["海淀", "朝阳", "西城", "东城"];
array[2] = ["石家庄", "邢台", "保定"];
function showCities() {//根据省显示城市
    //得到第一个选择框的选中的选项的索引
    var i = document.getElementById("sel1").selectedIndex;
    //先删除选择框中原有的元素
    var sellObj = document.getElementById("sel2");
    // for(var j=0;j<sellObj.childNodes.length;j++){
    //     sellObj.removeChild(sellObj.childNodes[j]);
    // }注意事项：这样操作是删不干净的！
    while(sellObj.childNodes.length>0){
        sellObj.removeChild(sellObj.firstChild);
    }
    var cities = array[i];//根据索引找到城市数据
    //循环城市数据， 创建 option 元素， 文本写入
    for(var index=0;index<cities.length;index++){
        var newNode = document.createElement("option");
        newNode.innerHTML = cities[index];
        document.getElementById("sel2").appendChild(newNode);
    }
}
```

八十、HTML DOM

LICHOO

6.1 HTML DOM 概述

- 1) HTML DOM 定义了用于 HTML 的一系列标准的对象，以及访问和处理 HTML 文档的标准方法（对 DOM 操作进行了封装，实现代码的简化）。
- 2) HTML 标签对象化：将网页中的每个元素都看作一个对象。
- 3) 常用 HTML DOM 对象



- 4) 如何解决浏览器兼容性问题：首先代码要符合 W3C 标准，然后了解各浏览器特殊的地方。
- 5) 多种对象中，重要了解 Select 对象和 Table 对象。

6.2 Select 对象

- 1) Select 对象代表 HTML 表单中的一个下拉列表，<select>标签即表示一个 Select 对象。
- 2) 常用属性：options（选项数组）、selectedIndex（索引从 0 开始）、size
- 3) 常用方法：add(option)、remove(index)
- 4) 事件：onchange

例如：<select onchange="alert(this.selectedIndex);">
<option value="1">a</option> <option value="2">b</option> </select>

6.3 Option 对象

- 1) Option 对象代表 HTML 表单中下拉列表中的一个选项，<option>标签表示一个 Option 对象
- 2) 创建对象：var obj=new Option(text,value);
- 3) 常用属性：index、text、value、selected

例如：(结合上例修改 onchange="selFunc();" id="s1")
function selFunc(){
 var selObj=document.getElementById("s1");
 var value=selObj.options[selObj.selectedIndex].value;
 alert(value); var option=new Option("c","3"); selObj.add(option); }

6.4 案例：联动菜单（HTML DOM 方式）

修改 5.11 案例中的 Javascript：

```

function showCities(){
    //得到第一个选择框选中的选项的索引
    var i = document.getElementById("sel1").selectedIndex;
    //删除原有的选项
    var sellObj = document.getElementById("sel2");
    sellObj.options.length = 0;
    var cities = array[i];//根据索引找到程序数据
    //循环城市数据，创建 option 元素，文本写入
    for(var index=0;index<cities.length;index++){
        var newNode = new Option(cities[index]);
        sellObj.add(newNode);
    }
}

```

6.5 Table 对象

- 1) Table 对象代表一个 HTML 表格，<table>标签表示一个 Table 对象。
- 2) 常用属性：rows（返回所有行数组）、cells（返回所有单元格数组）
- 3) 常用方法：①table.insertRow(index): 返回 TableRow 对象（插入新行）。
②table.deleteRow(index): 删除 TableRow 对象（删除行）。

6.6 TableRow 对象

- 1) TableRow 对象代表一个 HTML 表格行，<tr>标签表示一个 TableRow 对象
- 2) 常用属性：cells、innerHTML、rowIndex
- 3) 常用方法：①row.insertCell(index): 返回 TableCell 对象（插入单元格）。
②row.deleteCell(index): 删除 TableCell 对象（删除单元格）。

6.7 TableCell 对象

- 1) TableCell 对象代表一个 HTML 表格单元格，<td>标签表示一个 TableCell 对象。
- 2) 常用属性：cellIndex、innerHTML、colSpan、rowSpan

6.8 案例：产品列表

```

名称: <input type="text" id="txtID"/><br />
价格: <input type="text" id="txtName"/><br />
<input type="button" value="增加" onclick="modiTable();"/>
<table id="t1" border="1">
    <tr>      <td>产品 ID</td><td>产品名称</td><td></td>      </tr>      </table>
function modiTable() {//增加行
    var table = document.getElementById("t1");//得到表格对象
    var row = table.insertRow(table.rows.length);//创建新行
    var cell1 = row.insertCell(0);//为行创建产品名称单元格
    cell1.innerHTML = document.getElementById("txtID").value;
    var cell2 = row.insertCell(1);//为行创建价格单元格
    cell2.innerHTML = document.getElementById("txtName").value;
    var buttonCell = row.insertCell(2);//为行创建操作按钮的单元格
    var button = document.createElement("input");
    button.type = "button";      button.value = "删除";
}

```

```
button.onclick = function () {    delFunc(this);    };
buttonCell.appendChild(button);
function delFunc(btnObj) {//删除按钮的单击事件
    var isDel = confirm("真的要删除吗？");
    if (!isDel)      return;
    //找到当前行的 ID
    var rowObj = btnObj.parentNode.parentNode;
    var id = rowObj.getElementsByTagName("td")[0].innerHTML;
    //循环行，根据 id 定位需要删除的行，并删除
    var table = document.getElementById("t1");
    for (var i = 1; i < table.rows.length; i++) {
        if (table.rows[i].cells[0].innerHTML == id) {
            table.deleteRow(i);      break;          }
    }
    alert("删除 ID 为 " + id + " 的数据。");//提示
}
```

八十一、window 其他子对象 (DHTML 模型) LICHOO

7.1 screen 对象

1) 包含有关客户端显示屏幕的信息（封装了屏幕相关的信息，供读取）。

2) 常用属性: width/height、availWidth/availHeight

例如: var n = screen.width//ok! screen.width = 1000;//error! 只能读不能写

7.2 history 对象

1) 包含用户在浏览器窗口中访问过的 URL（封装了历史访问记录）。

2) length 属性: 浏览器历史列表中的 URL 数量。

3) 方法: ①history.back(): 单击后退按钮。

②history.forward(): 单击向前按钮。

③history.go(n): 单击 n 次后退按钮。

7.3 location 对象

1) 包含有关当前 URL 的信息（地址栏），常用于获取和改变当前浏览的网址。

2) href 属性: 当前窗口正在浏览的网页地址。

3) 方法: ①location.href="url": 在当前页面打开，保留历史访问记录。

②location.replace("url"): 在当前页面打开 url, 不保留历史访问记录。

③location.reload(): 重新载入当前网址，等同于按刷新。

◆ 注意事项:

❖ location.href="url"也等同于 location["href"]，但很少这么用。

❖ location="url"好像也可以。

7.4 navigator 对象

包含浏览器软件的相关信息，常用于获取客户端和操作系统信息。

eg: 遍历 navigator 对象的所有属性

```
function testNavigator(){ var s="";
for(var p in navigator){ //p 代表对象中的每一个属性
    s += p +": "+navigator[p]+"\n" } alert(s); }
```

7.5 事件

1) 事件: 指 DHTML 对象在状态改变、鼠标操作或键盘操作时触发的动作。

2) 事件的类别:

①鼠标事件: onclick/ondblclick/onmouseover/onmouseout

②键盘事件: onkeydown/onkeyup

③状态改变事件: onblur/onfocus/onchange.onload(body 里)/onsubmit(form 里)

3) 如何定义事件:

①<标签里 onXXX="代码">: 静态, 写在 html 代码中

②obj.onclick = function() {}: 动态, 在 JavaScript 代码中定义

4) 事件可以被取消: onXXX = "return false;"

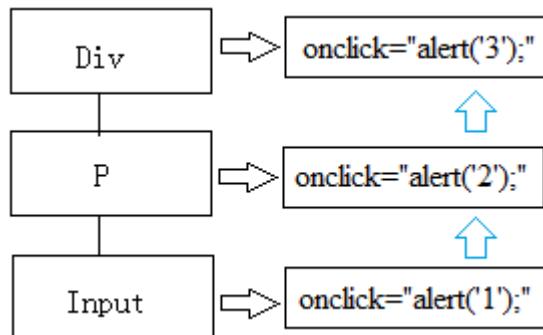
例如: ID: <input id="txtID" type="text" />
<input type="submit" value="Delete" onclick="return deleteFunc();"/>

```

<span id="info"></span>
function deleteFunc(){ var data=document.getElementById("txtID").value;
    var spanObj=document.getElementById("info");
    if(isNaN(data)){ info.innerHTML="请录入数值";
        return false;
    }
    else return confirm("真的要删除吗? ");
}

```

5) 事件的处理机制（冒泡机制）



事件冒泡机制：当处于 DHTML 对象模型底部的对象事件发生时，会依次激活上面对象定义的同类事件处理（即当有层次关系，且定义了相同事件时，会发生）。

例如：现象：单击按钮，则会弹出“1”、“2”、“3”。

单击段落，则会弹出“2”、“3”

单击 div，则会弹出“3”

```

<div style="width:100px;height:100px;border:1px solid red;" onclick="alert('3');">
    <p onclick="alert('2');">p text
        <input type="button" value="1" onclick="alert('1');" />
    </p>
</div>

```

注意事项：使用 event 对象可以禁止事件冒泡，也可详见 jQuery 笔记。

7.6 event 对象

- 1) 任何事件触发后将会产生一个 event 对象。
- 2) event 对象记录事件发生时的鼠标位置、键盘按键状态和触发对象等信息（获得 event 对象后，可以获得相关信息）。
- 3) 常用属性：clientX/clientY/cancelBubble=true（取消冒泡）……
- 4) 使用 event 对象
 - ①如何获得 event 对象
 - A. IE/Chrome 等浏览器：直接使用 event 关键字。
 - B. Firefox 浏览器：在事件定义中，使用 event 关键字将事件对象作为参数传入方法。
 - ◆ 注意事项：在 Firefox 里直接在 JavaScript 中使用 event，则不行！不认识！即只能在 html 页面获得 event 对象！其他浏览器既可以在 JavaScript 中获得，也可以在 html 页面中获得。
 - ②如何获得事件源
 - A. IE/Chrome 等浏览器：event.srcElement
 - B. Firefox 浏览器：event.target
 - ◆ 注意事项：两种获得事件源对象的方式最新的 Chrome 浏览器都支持。

```

例如: <div onclick="func();>div text</div>
//IE/Chrome 等浏览器
function func(){
    alert(event.clientX+":"+event.clientY);
    alert(event.srcElement.nodeName);//DIV (大写的)
}

//Firefox 浏览器
<div onclick="func(event);>div text</div>
<!--注意把 event 传入方法中! 这样也可以在 JavaScript 中写复杂代码了! -->
function func(e){
    alert(e.clientX+":"+e.clientY);
    alert(e.target.nodeName);//DIV (大写的)
}

```

5) 考虑各浏览器的兼容性

```

<div onclick="func(event);>div text</div>
<!--注意把 event 传入方法中! 这样也可以在 JavaScript 中写复杂代码了! -->
//如下操作可兼容各浏览器
function func(e){
    alert(e.clientX+":"+e.clientY);
    var obj=e.srcElement || e.target;
    alert(obj.nodeName);//DIV (大写的)
}

```

6) 案例

eg: 简单计算器 (简化版)

```

<div style="border:1px solid red;" onclick="cal(event);>
    <input type="button" value="1" />           <input type="button" value="2" />
    <input type="button" value="3" />           <input type="button" value="4" />
    <input type="button" value="+" />           <input type="button" value="-" />
    <input type="button" value="/" />           <input type="text" id="txtNumber" />
</div>

function cal(e){
    var obj = e.target || e.srcElement;//获得被单击的对象
    //判断只有单击的是 button
    if(obj.nodeName == "INPUT" && obj.type == "button"){
        if(obj.value == "="){//按钮进行 eval; 其他按钮做拼接操作
            var s = document.getElementById("txtNumber").value;
            var data = eval(s);
            document.getElementById("txtNumber").value = data;
        }else{
            document.getElementById("txtNumber").value +=obj.value;
        }
    }
}

```

八十二、面向对象基础

LICHOO

对象是一种特殊的数据类型，由属性和方法封装而成。

8.1 属性

属性指的是与对象有关的值：对象名.属性名

8.2 方法

方法指的是对象可以执行的行为或可以完成的功能：对象名.方法名()

8.3 定义对象的三种方式

1) 创建对象的实例。2) 创建对象的模版。3) 使用 JSON (相当于 Java 中的 Map)。

8.4 创建通用对象

使用 Object 创建对象以实现简单的封装，但不方便重用！

```
例如: <input type="button" value="使用 Object 创建对象" onclick="testObject();"/>
function TestObject() { //创建对象，封装数据和行为
    var s=new Object();           s.name="mary";      s.age=18;
    s.sing=function(){alert("hello");}; //匿名方法，赋进去一个 function，所以 sing 是个方法
    alert(s.name);   alert(s.age);   s.sing(); }
```

◆ 注意事项：. 后写啥属性名都行，因为 JavaScript 是松散类型语言。

8.5 创建对象的模版

可重用的封装；定义构造函数，以创建自定义的对象。

语法：function ObjName(参数 1, 参数 2,...){}

```
例如: <input type="button" value="自定义对象" onclick="testOwnObject();"/>
function Student(n1,a1){//定义一个对象的模版： Student
    this.name=n1;//用了 this 关键字，就认为 Student 是一个类，而不是方法
    this.age=a1;  this.introduceSelf=function(){
        alert("i am "+this.name+", i am "+this.age+" year
old");  }
}
function testOwnObject(){//测试自定义的 Student 对象
    var p1=new Student("mary",18);          var p2=new Student("join",20);
    alert(p1.name);  alert(p2.age);  p1.introduceSelf();  p2.introduceSelf(); }
```

八十三、JSON

LICHOO

9.1 JSON 概述

1) 数据的传递

①数据在 JavaScript 范围里传递，使用 Object 创建对象或者创建对象的模版，两种都可用。

②数据传递到服务器端，采用一种通用的格式，Xml 或者 JSON。

2) JSON (JavaScript Object Notation) 是一种轻量级的数据交换格式。

3) 使用名/值对的方式定义。

4) 名称需要使用“”引起来。

5) 多对定义之间使用“，”隔开。

例如：var obj={ "firstName":"chang", "lastName":"yanbo" };

6) 相当于 Java 中的 Map<String,obj>，最简单的创建对象的方式：var obj={}，相当于 Java 中 new HashMap();键值对中值为字符串才写引号（单或双），值为数值，则直接写，

例如：var obj={ name:"chang", age:23 }

◆ 注意事项：

- ❖ 为了简化，键值对中，键的引号可以省略，但只有在无歧义的时候才能省。当键里有“.”时，如：user.name，则必须加引号。
- ❖ 例如：var obj={user.name:"chang", age:23}，获取 user.name 所对应的值"chang"时，将会有歧义，如：var name=obj[user.name]。user.name 应该是一个整体，一个字符串。而上述写法的意思就是 user 对象中的 name 属性了，所以在定义对象属性和获取属性值 user.name 时必须都加引号。
- ❖ 即 var obj={"user.name":"chang", age:23}; var name=obj["user.name"];

7) 相关操作：

①var name=obj["name"]; //相当于 obj.name，取键所对应的值

②obj["score"]=99; //放入一个新的键值对

9.2 名称可以是属性

字符串类型的属性值，需要使用“”引起来。

9.3 名称也可以是方法

为其赋值 function 对象

例如：function testJSON(){

```
var obj={ "name":"changyanbo", "age":"23",
  "introduce":function() { //一般不需要定义方法，因为主要是为了数据交换的
    var info = "i am " + this.name + ", i am " + this.age + " years old.";
    alert(info);
  }
}
alert(obj.name); alert(obj.age); obj.introduce(); //测试对象
}
```

9.4 案例：批量提交数据和下拉框版式日历

eg1：批量提交数据

```
<form> <h2>批量提交数据</h2>
  <input type="button" value="增加产品" onclick="addNewRow();"/><br />
```

```

<table id="table1">
    <tr> <td>产品 ID</td><td>产品名称</td><td>产品价格</td> </tr>
</table><br />
        <input type="button" value="保存" onclick="saveData();"/> </form>
function addNewRow() {//为表格添加行
    var table = document.getElementById("table1");//得到表格对象
    var row = table.insertRow(table.rows.length);//创建新行
    var idCell = row.insertCell(0);//为行创建 id 单元格
    var input1 = document.createElement("input");
    idCell.appendChild(input1);
    var nameCell = row.insertCell(1);//为行创建 name 单元格
    var input2 = document.createElement("input");
    nameCell.appendChild(input2);
    var priceCell = row.insertCell(2);//为行创建 price 单元格
    var input3 = document.createElement("input");
    priceCell.appendChild(input3);
}
function saveData() {//保存数据
    var datas = new Array();//定义数组，用于存储数据
    var table = document.getElementById("table1");//循环表格，并收集数据
    for (var i = 1; i < table.rows.length; i++) {//获得页面的数据
        var id = table.rows[i].cells[0].firstChild.value;
        var name = table.rows[i].cells[1].firstChild.value;
        var price = table.rows[i].cells[2].firstChild.value;
        var o = new Data(id, name, price);//定义对象以封装数据
        datas[i - 1] = o;
    }
    for (var i = 0; i < datas.length; i++) {//提交数据（测试数据）
        datas[i].show();
    }
}
function Data(id, n, p) {//自定义对象
    this.id = id;//属性      this.name = n;      this.price = p;
    this.show = function () {//方法
        alert("id:" + this.id + "\nname:" + this.name + "\nprice:" + this.price);
    };
}

```

eg2：下拉框版式日历

```

<body onload="initialCalendar();">
    <form><h2>下拉框版日历</h2>
        <select id="selYear" onchange="dateChanged();"></select>年
        <select id="selMonth" onchange="dateChanged();"></select>月
        <select id="selDate"></select>日
    </form> </body>
function initialCalendar() {//初始化日历的相关择框
    var startYear = 2005; //定义开始年份
    var index = 0;//声明 option 对象的索引
    //初始化年份选择框
    var yearObj = document.getElementById("selYear");
    var endYear = (new Date()).getFullYear();
    for (var i = startYear; i <= endYear; i++) {

```

```
var optionObj = new Option(i,i);
yearObj.options[index] = optionObj;           index++;      }

//初始化月份选择框
var monthObj = document.getElementById("selMonth");
index = 0;
for (var i = 1; i <= 12; i++) {
    var optionObj = new Option(i,i);
    monthObj.options[index] = optionObj;           index++;      }

//初始化日期选择框
var dateObj = document.getElementById("selDate");
index = 0;
for (var i = 1; i <= 31; i++) {
    var optionObj = new Option(i,i);
    dateObj.options[index] = optionObj;           index++;      }

function dateChanged() { //年份和月份选择改变后，修改日期下拉框
    //得到所选择的年份
    var yearObj = document.getElementById("selYear");
    var year = yearObj.options[yearObj.selectedIndex].value;
    //得到所选择的月份
    var monthObj = document.getElementById("selMonth");
    var month = monthObj.options[monthObj.selectedIndex].value;
    //得到当月最大天数
    var days = getDays(year, month);
    //删除原有天数
    var dateObj = document.getElementById("selDate");
    dateObj.options.length = 0;
    //重新添加天
    var index = 0;
    for (var i = 1; i <= days; i++) {
        var optionObj = new Option(i, i);
        dateObj.options[index] = optionObj;           index++;      }

    function getDays(year, month) { //获得某年某月的总天数
        if (month == 2) {
            var isLeap = (year % 4 == 0 && year % 100 != 0) || year % 400 == 0;
            return isLeap ? 29 : 28;
        }
        else if (month == 4 || month == 6 || month == 9 || month == 11)      return 30;
        else      return 31;      }
    }
}
```

9 Servlet 学习笔记

LICHOO

八十四、Servlet 概述

1.1 B/S 架构（了解）

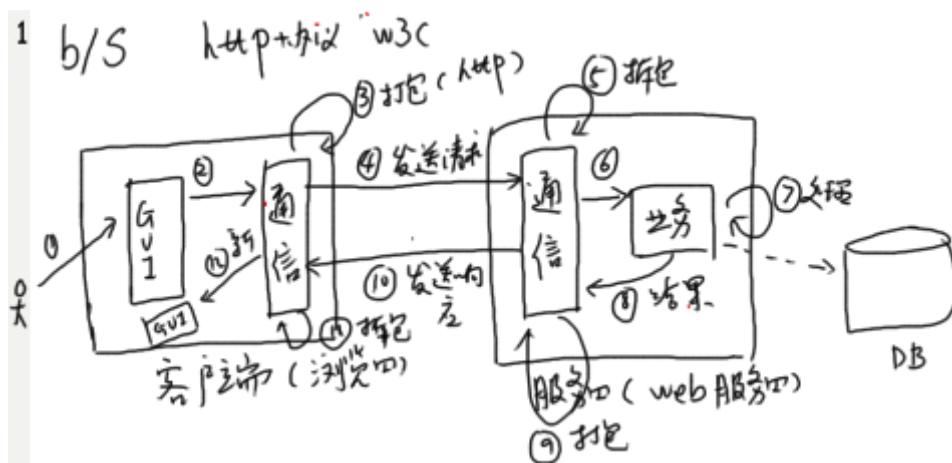
1) 什么是 B/S 架构

客户端使用浏览器，服务端使用 web 浏览器，客户端跟服务器之间使用 HTTP 协议进行通讯。

2) 优点

① 客户端不需要单独安装（因为使用浏览器）；C/S 架构比较麻烦的是需要单独安装每个客户端，并且一旦客户端版本发生改变，就需要再次安装。

② 开发相对简单；C/S 架构需要我们分别在，客户端和服务器端编写相应的通信处理模块和自定义协议，而 B/S 架构使用标准的 HTTP 协议（即不再需要自定义协议），而且浏览器与 Web 服务器已经包含了相应的通信模块了。



1.2 什么是 Servlet

Sun 公司制订的一种用来扩展 Web 服务器功能的组件规范。

1) 扩展 web 服务器功能

早期（2000 年左右）的 Web 服务器：apache 的 web server、微软的 iis。只能够处理静态资源（即需要事先将 html 文件写好），不能够处理动态资源的请求（即需要依据请求参数然后进行计算，生成相应的页面）。

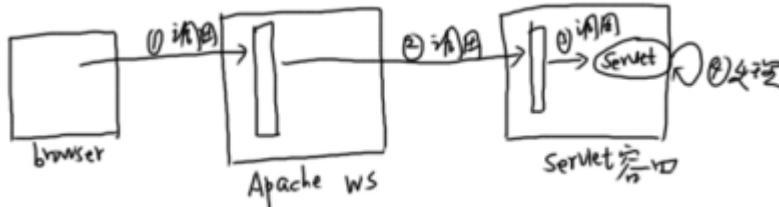
为了让这些 web 服务器能够处理动态资源的请求，需要扩展他们的功能。

早期使用的是 CGI 技术（Common Gateway Interface 通用网关接口），可以使用很多语言编写，如 perl, C/C++ 等来开发 CGI 程序。但是 CGI 程序有几个问题，比如开发比较复杂（因为需要程序员自己去分析请求参数）、性能不佳（因为当 Web 服务器收到请求之后，会启动一个 CGI 进程来处理请求）、CGI 程序依赖平台（可移植性不好）。

现在，可以使用 Servlet 来扩展。

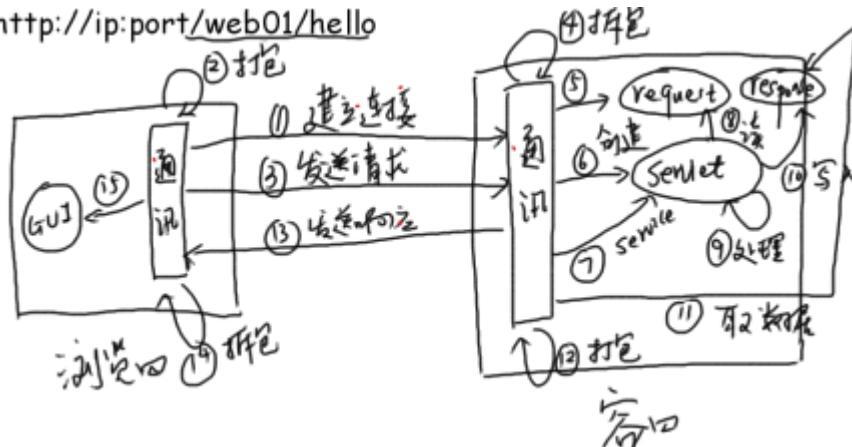
当浏览器将请求发送给 Web 服务器（比如：apcahe 的 web server），Web 服务器会向 Servlet 容器发送请求，Servlet 容器负责解析请求数据包。当然，也包括网络通讯相关的一些处理，然后将解析之后的数据交给 Servlet 来处理（Servlet 只需要关注具体的业务处理）。

不用关心网络通讯相关的问题)。



LICHOO

- ◆ 注意事项：可以不使用服务器，而直接向 Servlet 容器发送请求，因为 Servlet 容器里面也有个通信模块，所以也可直接把 Servlet 容器当作简单的 Web 服务器来使用。



2) 组件规范

① 组件：是符合一定规范，并且实现部分功能的可以单独部署的软件模块。组件必须要部署到容器里面才能运行。

② 容器：也是符合一定规范，并且提供组件的运行环境的程序。

- ◆ 注意事项：单个的组件、单个的容器都是没意义的，都不能单独运行，需要放在一起才能运行。

1.3 什么是 Tomcat

Tomcat 本身是一个 Servlet 容器，即可以提供 Servlet 运行环境的一个程序，但是 Tomcat 还提供了 Web 服务器所具有的所有功能，所以我们也称 Tomcat 是一个 Web 服务器。Tomcat 的默认端口是 8080。

1) 安装 Tomcat (www.apache.org 下载)

① Linux 系统下安装和配置的步骤

step1：解压到/home/soft01 下

step2：配置环境变量

```
cd /home/soft01  
vi .bash_profile  
JAVA_HOME:jdk 的主目录  
CATALINA_HOME:tomcat 的主目录  
PATH:CATALINA_HOME/bin
```

step3：启动 Tomcat

```
cd /home/soft01/tomcat 主目录/bin  
sh startup.sh 或者 sh catalina.sh run
```

接下来，可以打开浏览器，输入 <http://localhost:8080>，验证是否配置成功。

step4：关闭 Tomcat

```
cd /home/soft01/tomcat 主目录/bin
```

```
sh shutdown.sh
```

②Windows 系统下安装和配置的步骤

step1：解压到某个盘下

step2：配置环境变量

JAVA_HOME (JDK 的主目录) 必须配置

CATALINA_HOME (Tomcat 的主目录) 可以不配置

PATH (Tomcat 的 bin 目录) 可以不配置

“我的电脑”右键“属性”，打开“系统属性”的“高级”选项卡，找到“环境变量。

新建“用户变量”。

新建 JAVA_HOME。

◆ 注意事项：新建系统变量或新建用户变量均可，建议新建用户变量。

新建 CATALINA_HOME (可以不配置)。

新建 PATH (可以不配置)。

◆ 注意事项：

❖ 如果环境变量中已经有 PATH，修改该 PATH 即可，使用“；”

分号作分隔，如下所示：

```
PATH C:\Program Files\Java\jdk1.6.0_06\bin ;
```

```
D:\apache-tomcat-5.5.23\bin
```

❖ Windows 操作系统下以“；”分号为分隔符；linux 系统下以“：“冒号为分隔符。

step3：启动 Tomcat

进入启动 Tomcat 的 bin 目录下，双击“startup.bat”。

接下来，可以打开浏览器，输入 <http://localhost:8080>，验证是否配置成功。

step4：关闭 Tomcat

进入启动 Tomcat 的 bin 目录下，双击“shutdown.bat”。

2) Tomcat 目录结构简介

①bin 目录：存放启动和关闭服务器的一些脚本（命令）。

②common 目录：共享(部署在该服务器上的所有程序都可以使用)的一些 jar 包。

③conf 目录：存放服务器的一些配置文件。

④webapps 目录：部署目录。

⑤work 目录：服务器运行时，生成的一些临时文件。

1.4 如何写一个 Servlet (不使用开发工具)

step1：先写一个 Java 类，实现 Servlet 接口或者继承 HttpServlet 抽象类。

```
public class HelloWorldServlet extends HttpServlet {  
    //Tomcat 会自动调用 service，自己不用再去写方法调用了，所以必须把名字写对！  
    public void service(HttpServletRequest request, HttpServletResponse response)  
        throws ServletException, IOException {  
        //异常必须写两个，不能写成 Exception，也不能再多个异常  
        //设置一个消息头 content-type，告诉浏览器返回的数据类型是一个 html 文档，以及编码格式。此外，还可以告诉服务器，在使用 out 输出时，使用指定的编码格式进行编码  
        response.setContentType("text/html;charset=utf-8");  
    }  
}
```

```

    //通过响应回对象，获得一个输出流
    PrintWriter out=response.getWriter();
    //调用流的方法进行输出，其实质是将处理结果写到了 response 对象上
    out.println("<span style='color:red;font-size:30px;'>Hello World</span>");
    /** out.close()不调用也可以，因为 Servlet 方法执行完毕，容器会自动调用 out.close 方法
 */
    out.close(); } }

```

step2：执行编译操作：javac -d . HelloWorldServlet.java

但是会报错，找不到某些类！其中 -d . 代表把编译后的文件放在当前文件夹下。

把 Tomcat 安装目录中 common 中 lib 中的 servlet-api.jar 和刚编写类放一起（是编译时需要的 jar）。

再次编译：javac -cp servlet-api.jar -d . HelloWorldServlet.java，其中 -cp servlet-api.jar 表示告诉 Java 编译器去哪里找需要的 class 文件（到 servlet-api.jar 的 jar 包中找）。

step3：打包，即创建一个具有如下结构的文件夹：

```

appname (文件夹名起应用名)
|---WEB-INF (必须大写)
|   |---classes (文件夹，放.class 文件)
|   |---lib (文件夹，放.jar 文件，可没有)
|   |---web.xml (部署描述文件 url-pattern)

```

将编译好的.class 文件放入 classes 文件夹中（如有包，则一起放入）。

web.xml 文件内容：

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" <!--根元素-->
    xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
        <servlet>
            <servlet-name>helloWorld</servlet-name>
            <!--类名要完整（包名.类名都要有）-->
            <servlet-class>Servletday01.HelloWorldServlet</servlet-class>
        </servlet>
        <servlet-mapping>
            <servlet-name>helloWorld</servlet-name>
            <url-pattern>/hello</url-pattern><!-- 注意斜线开头 -->
        </servlet-mapping>
    </web-app>

```

step4：部署，将 step3 创建的文件夹拷贝到 Servlet 容器特定的文件夹下面（拷贝到 Tomcat 的 webapps 目录下）。

◆ 注意事项：也可以将 step3 创建的文件夹使用 jar 命令进行压缩，生成.war 为后缀的文件，然后拷贝。

step5：启动 Servlet 容器，访问 Servlet。

访问格式：http://ip:port/appname/url-pattern

比如在浏览器地址栏输入：http://localhost:8080/web01/hello

1.5 使用 MyEclipse 开发 Servlet

step1：配置 MyEclipse，使得 MyEclipse 能够管理 Tomcat。

1) 点击工具栏上的“Run/Stop/Restart MyEclipse Servers”图标旁边的下拉箭头，选择“Configure Server”。

2) 在弹出的对话框“Preferences”中展开“MyEclipse”--“Servers”--“Tomcat”--“Tomcat5.X”

◆ 注意事项：选择你目前电脑上 Tomcat 的版本，此处以 Tomcat5 为例。

3) 将 Tomcat server 选项置为“Enable”（默认为“Disable”）。

4) 点击“Tomcat home directory”之后的“Browse”按钮，选择 Tomcat 主目录，确定，然后“Tomcat base directory”和“Tomcat temp directory”自动生成，点击“OK”。

◆ 注意事项：两项可改可不改的：

❖ Tomcat 下的 JDK--“Tomcat JDK name”是自己已安装的 JDK（Tomcat 也是 Java 写的也得依赖 JDK）。

❖ 建议 Tomcat 下的 Launch--“Tomcat launch mode”设置为 Run model，默认为 Debug mode 而该模式在有些时候会显示不正常。

5) 回到工具栏上的“Run/Stop/Restart MyEclipse Servers”图标旁边的下拉箭头，选择 Tomcat 5.x，点击“Start”。

6) 当在控制台显示“Server startup in XXX ms”，则 Tomcat 启动成功。

◆ 注意事项：如果出现“Address already in use: JVM_Bind”异常，则说明已经启动了一个 Tomcat。解决办法：运行 shutdown 命令，关闭之前开启的 Tomcat。

step2：建立一个 Web Project（Web 工程），填写“Project name”，JDK 最好选 5.0，其他选项默认，点击“Finish”。

step3：编写 Java 类和 web.xml 文件。

step4：部署项目到 Tomcat 服务器。

1) 点击工具栏“Deploy MyEclipse J2EE Project to Server”按钮。

2) 弹出对话框“Project Deployments”，点击“Add”按钮。

3) 弹出“New Deployment”对话框，选择“Tomcat 5.x”，点击“Finish”，最后点击“OK”。

◆ 注意事项：在对话框“Project Deployments”对话框有 4 个按钮，常用的为

① “Add”按钮：在 Tomcat 服务器上增加新应用。

② “Remove”按钮：删除 Tomcat 服务器上的新应用。

③ “Redeploy”按钮：重新部署该应用，一般每次修改后都需要重新部署一下。

step5：访问 Tomcat 服务器上的 Servlet 实例。

访问格式：<http://ip:port/appname/url-pattern>

比如在浏览器地址栏输入：<http://localhost:8080/web01/hello>

◆ 注意事项：

❖ 在 IDE 工具（集成开发环境）中启动 Tomcat 部署项目后，不需要重新启动服务器，系统会自动部署。

❖ IDE 工具简化了 Servlet 的开发步骤：

第 1 步 写一个 java 类 手动

第 2 步 编译 自动

第 3 步 打包 自动

第 4 步 部署 不用手动拷贝，点一下

第 5 步 启动服务器，访问 servlet 手动

默认情况下，应用名和工程名相同。

- ❖ 工程--属性--MyEclipse--Web--Web Context 可修改部署后的应用名。
- ❖ 新建 Web 工程时也可以手动在 Context root RUL 中更改应用名。

LICHOO

1.6 Servlet 是如何运行的

比如，在浏览器地址栏输入：http://ip:port/web01/hello（web01/hello 为请求资源路径）。

step1：浏览器依据 ip 和 port 建立与 Servlet 容器之间的连接。

step2：浏览器将请求数据打包（即按照 http 协议的要求，将相关数据封装成一个数据包，一般称之为请求数据包）并发送给 Servlet 容器。

step3：Servlet 容器解析请求数据包，并将解析之后得到的数据放到已创建的 request 对象上，同时，容器也已经创建好了一个 response 对象。

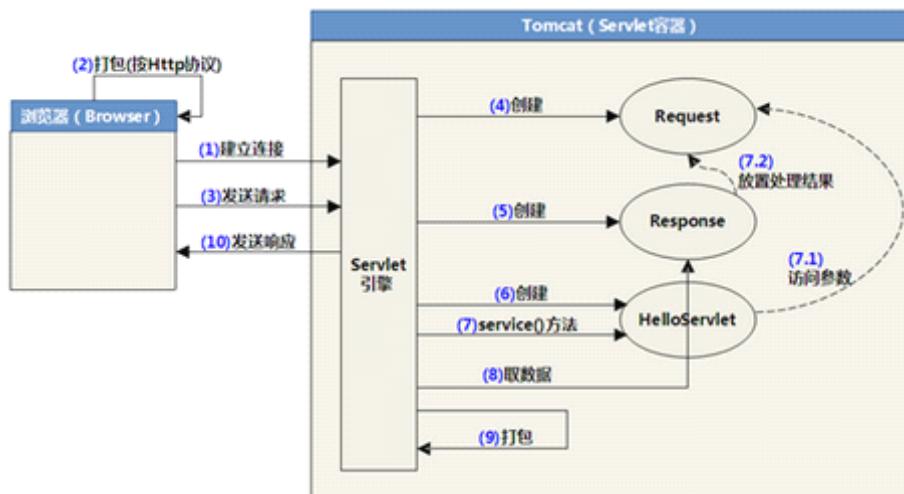
step4：Servlet 容器依据请求资源路径（即/web01/hello）找到 Servlet 的配置，然后创建 Servlet 对象（根据 xml 文件里配置的类而创建）。

step5：Servlet 容器接下来调用 Servlet 对象的 service 方法，并且会将事先创建好的 request 对象和 response 对象作为 service 方法的参数传递给 Servlet 对象。

step6：Servlet 可以通过 request 对象获得请求参数，进行相应的处理，然后将处理结果写到 response 对象上。

step7：Servlet 容器读取 response 对象上的数据，然后将处理结果打包（响应数据包）并发送给浏览器。

step8：浏览器解析响应数据包，将返回的数据展现给用户。



1.7 常见错误及解决方式

- 1) 404: 是一个状态码（是一个三位数字，由服务器发送给浏览器，告诉浏览器是否正确处理了请求），404 的意思是说：服务器依据请求资源路径，找不到对应的资源。
解决：
 - ①依据 http://ip:port/appname/url-pattern 检查你的请求地址是否正确。
 - ②仔细检查 web.xml，特别要注意 servlet-name 是否一致。
- 2) 500: 服务器处理出错，一般是因为程序运行出错。
解决：
 - ①检查程序的代码，比如：是否继承。
 - ②检查 web.xml，类名要填写正确。
- 3) 405: 服务器找不到对应的 service 方法。
解决：检查 service 方法的签名（方法名、参数类型、返回类型、异常类型）。

1.8 案例：根据请求次数显示结果和显示当前时间

eg1：在地址栏输入 `http://localhost:8080/web01/hello?qty=5` 后，获得 5 个 Hello World

◆ 注意事项：qty：请求参数名字。5：请求次数为 5。

```
public void service(HttpServletRequest request,
                     HttpServletResponse response) throws ServletException, IOException {
    //读请求参数值，返回值是 String
    String qtyStr=request.getParameter("qty"); int qty=Integer.parseInt(qtyStr);//转成整数
    String rs=""; for(int i=0;i<qty;i++){//处理请求，生成处理结果
        rs+="<div style='font-size:30px'>Hello World</div>";
    }
    //输出处理结果，设置一个消息头 content-type，告诉浏览器返回的数据类型是一个
    //html 文档
    response.setContentType("text/html");
    PrintWriter out=response.getWriter();//通过响应对象，获得一个输出流
    //调用流的方法进行输出，其实质是将处理结果写到了 response 对象上
    out.println(rs); out.close();
}
```

eg2：在网页上显示当前时间

step1：Java 类中写

```
public void service (HttpServletRequest request,
                     HttpServletResponse response) throws ServletException, IOException {
    response.setContentType("text/html");//告诉浏览器返回的数据类型是一个 html 文档

    PrintWriter out=response.getWriter();//通过响应对象，获得一个输出流
    Date date=new Date(); SimpleDateFormat sdf=new
    SimpleDateFormat("yyyy-MM-dd");
    String now=sdf.format(date); out.println(now); //输出到页面 out.close();
}
```

step2：web.xml 中写：

```
<servlet>
    <servlet-name>Date</servlet-name>
    <servlet-class>Servletday01.DateDemo</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Date</servlet-name>
    <url-pattern>/date</url-pattern>
</servlet-mapping>
```

八十五、HTTP 协议

LICHOO

2.1 什么是 HTTP 协议

HTTP (Hypertext transport protocol) 是超文本传输协议。是一种应用层协议，由 W3C 制定，它定义了浏览器（或者其他客户端）与 Web 服务器之间通讯的过程及数据格式。

2.2 通讯的过程

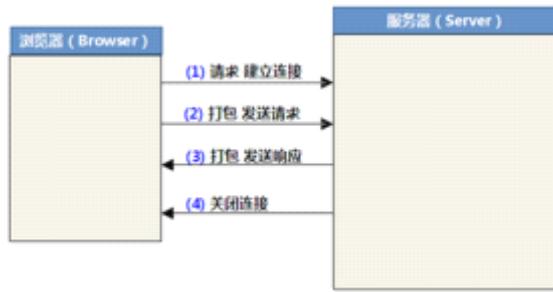
step1：浏览器建立与 Web 服务器之间的连接（Socket）。

step2：浏览器要将请求数据打包（请求数据包），然后发送给 Web 服务器。

step3：Web 服务器将处理结果打包（响应数据包），然后发送给浏览器。

step4：Web 服务器关闭连接。

◆ 注意事项：特点：一次请求，一次连接。优点：Web 服务器可以利用有限的连接个数为尽可能多的客户服务（效率高）。如果浏览器要再发请求，就必须重新建立一个新的连接。



2.3 数据格式

1) 请求数据包

①请求行：请求方式+请求资源路径+协议描述。

②若干消息头：消息头是一些键值对，一般由 W3C 定义，有特定的含义。浏览器和服务器之间，可以通过发送消息头来传递一些特定的信息，比如：浏览器可以通过 user-agent 消息头来通知服务器浏览器的类型和版本。

③实体内容：只有当请求方式为 post 时，浏览器才会将请求参数添加到实体内容里面，如果请求方式为 get，浏览器会将请求参数添加到请求资源路径的后面。

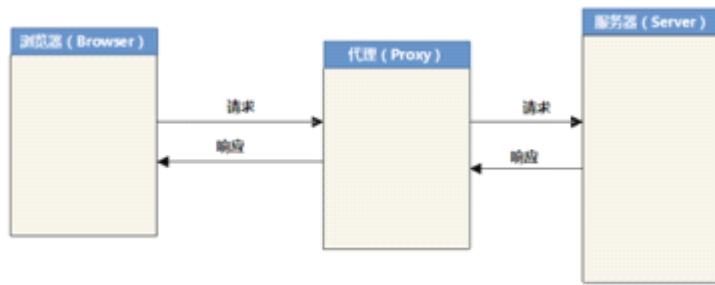
2) 响应数据包

①状态行：协议描述+状态码+状态描述。

②若干消息头：服务器也可以发送一些消息头给浏览器，比如 content-type，告诉浏览器服务器返回的数据类型和编码格式（字符集，比如：UTF-8、ISO-8859-1）。

③实体内容：程序处理之后，返回的结果。

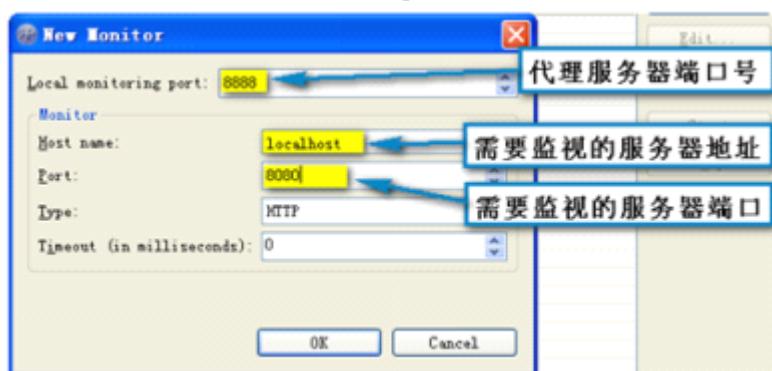
3) 截获数据包，使用 MyEclipse 中的 TCP/IP Monitor，TCP/IP Monitor 相当于一个代理服务器，它的原理图如下：



LICHOO

2.4 TCP/IP Monitor 的使用

- 1) Window--Show View--Other--MyEclipse Common--TCP/IP Monitor
- 2) 在 TCP/IP Monitor 的空白处右键--Properties--弹出对话框点 Add, 增加新的监视器:



- 3) 点击 Start, 启动代理服务器--OK。
- 4) 测试 1.8 案例 eg1, 地址栏输入: <http://localhost:8888/web01/hello?qty=5>, 注意使用代理口号! 执行完毕, 即可查看 TCP/IP Monitor 出现的内容。

2.5 get 请求与 post 请求

- 1) 哪些情况下, 浏览器会使用 get 方式发请求:
 - ①直接在浏览器地址输入某个地址。
 - ②点击链接地址。
 - ③表单默认的提交方法: <form method="get(默认)/post">。
- 2) 哪些情况下, 浏览器会用 post 方法发请求:
 - ①设置表单的 method 属性值为 “post”。
- 3) get 请求的特点:
 - ①get 请求会将请求参数添加到请求资源路径的后面, 因为请求行存放的数据大小有限 (也就是地址栏的最长字节数), 所以 get 请求只能提交少量的数据。
 - ②get 请求会将请求参数显示在浏览器地址栏, 不安全 (比如, 路由器会记录整个地址)。
- 4) post 请求的特点:
 - ①post 请求会将请求参数添加到实体内容里面, 所以, 可以提交大量的数据。
 - ②post 请求不会将请求参数显示在浏览器地址栏, 相对安全一些。但是, post 请求并不会对请求参数进行加密处理。用 HTTPS 协议进行加密处理。
 - ◆ 注意事项: 服务器不关心是用浏览器还是 Java 程序发送的请求, 只要符合协议格式, 都会处理。

2.6 如何读取请求参数

- 1) 方法一: String request.getParameter(String paraName);

- ①如果 paraName (即参数名称) 与实际的参数名称不一致, 会获得 null (不报错)。
 ②在使用表单提交数据时, 如果用户没有填写任何的值, 会获得空字符串 ""。
- 2) 方法二: String[] request.getParameterValues(String paraName);
 ①当有多个参数且名称相同时, 使用该方法。比如: ?city=bj&city=cs&city=wh
 ◆ 注意事项: getParameterValues 方法也可用于只有一个参数的情况。

2.7 访问数据库 (MySql)

- 1) 使用 MySql 数据库
- ①登录 MySql: mysql -uroot;//登录 mysql, 使用 root 用户权限
 - ②查看当前所有的数据库: show databases;
 - ③创建一个新的数据库: create database db_chang default character set utf8;//创建 db_chang 数据库, 默认是用 utf8 编码集 (不能写减号 -)
 - ④使用某个数据库: use db_chang;
 - ⑤查看当前数据库有哪些表: show tables; drop table tablename;//删表
 - ⑥建表:
- ```
create table chang_emp(id int primary key auto_increment,
 name varchar(50), salary double, age int);
insert into chang_emp(name,salary,age) values("tom",10000,23);
```
- ◆ 注意事项: auto\_increment: 自增长列, 即插入记录时, 数据库会自动为该列赋一个唯一的值 (相当于 Oracle 中的序列 sequence)。

- 2) 使用 JDBC 访问数据库
- step1: 将 JDBC 驱动放到 WEB-INF\lib 下  
 step2: 编写 JDBC 代码, 需要注意异常的处理!

## 2.8 案例: 添加员工 (访问 MySql 数据库)

step1: addEmpMySql.html 页面内容:

```
<form action="addempmysql" method="post">
<fieldset><legend>添加员工</legend>
姓名: <input name="name"/>

薪水: <input name="salary"/>

年龄: <input name="age"/>

<input type="submit" value="确认"/>
</fieldset></form>
```

step2: ListEmpServlet.java 中 service 方法内容:

```
Connection conn=null;
try { Class.forName("com.mysql.jdbc.Driver");
conn=DriverManager.getConnection("jdbc:mysql://localhost:3306/db_chang","root","");
Statement state=conn.createStatement();
ResultSet rs=state.executeQuery("select * from chang_emp order by id");
response.setContentType("text/html;charset=utf-8");
PrintWriter out=response.getWriter();
out.println("<table border='1' width='60%' cellpadding='0' cellspacing='0'>");
out.println("<tr><td>id</td><td>姓名</td><td>薪水</td><td>年龄</td></tr>");
while(rs.next()) { long id=rs.getLong("id"); String name=rs.getString("name");
double salary=rs.getDouble("salary"); int age=rs.getInt("age");
out.println("<tr><td>" + id + "</td><td>" + name + "</td><td>" + salary + "</td> +
<td>" + age + "</td></tr>"); }
}
```

```

 out.println("</table>");
 out.println("再次添加新员工"); out.close();
 }catch(Exception e) { e.printStackTrace();
 }finally{ if(conn!=null){ try { conn.close();
 }catch (SQLException e) { e.printStackTrace();} } }

```

step3: AddEmpMySql.java 中 service 方法内容:

```

request.setCharacterEncoding("utf-8"); String nameStr=request.getParameter("name");
String salaryStr=request.getParameter("salary"); String ageStr=request.getParameter("age");
PrintWriter out=response.getWriter();
Connection conn=null;//将员工信息插入到数据库
try { Class.forName("com.mysql.jdbc.Driver");
 conn=DriverManager.getConnection("jdbc:mysql://localhost:3306/db_chang","root","");
/** 此处不妥～若插入中文数据，会出现乱码！具体措施详见第三章！！ */
PreparedStatement prep=conn.prepareStatement(
 "insert into chang_emp(name,salary,age) values(?, ?, ?)");
prep.setString(1, nameStr); prep.setDouble(2, Double.parseDouble(salaryStr));
prep.setInt(3, Integer.parseInt(ageStr)); prep.executeUpdate();
/** response.setContentType("text/html;charset=utf-8"); out.println("添加成功");
 out.println("再次添加");//不使用重定向 */
response.sendRedirect("list");//使用重定向，详见第四章 out.close();
} catch (Exception e) {
/** 异常发生之后，首先记录日志，一般会将日志记录到文件里面，可以用一些日志工具，比如 log4j 来记录 */
 e.printStackTrace();
/** 接下来，看异常能否恢复，如果能，则编写处理代码。如果不能，则提示用户稍后重试。一般来说，系统异常是不能恢复的，所谓系统异常：指的是不是因为程序的原因产生的异常，比如：数据库服务停止、连接数据库的网络中断等 */
 response.setContentType("text/html;charset=utf-8");
 out.println("系统繁忙，稍后重试");
 out.println("重试");
} finally{ //与 step2 相同，此处略 }

```

step4: web.xml 内容（为了省纸，采取如下格式）

```

<servlet><servlet-name>AddEmpMySql</servlet-name>
 <servlet-class>Servletday02.AddEmpMySql</servlet-class></servlet>
<servlet><servlet-name>ListEmpServlet</servlet-name>
 <servlet-class>Servletday02.ListEmpServlet</servlet-class></servlet>
<servlet-mapping> <servlet-name>AddEmpMySql</servlet-name>
 <url-pattern>/addempmysql</url-pattern></servlet-mapping>
<servlet-mapping> <servlet-name>ListEmpServlet</servlet-name>
 <url-pattern>/list</url-pattern></servlet-mapping>

```

step5: 在浏览器地址栏输入 <http://localhost:8080/> 应用名/list 则访问出员工列表页面。

## 2.9 异常：IllegalStateException

出现如下错误可以不用理会，是因为 Tomcat 热部署造成的，重新手动部署一下即可：

信息: Illegal access: this web application instance has been stopped at  
java.lang.IllegalStateException  
at org.apache.catalina.loader.WebappClassLoader.loadCl...  
at org.apache.catalina.loader.WebappClassLoader.loadCl...  
at java.lang.ClassLoader.loadClassInternal(ClassLoader.j...

## 八十六、编码问题

LICHOO

### 3.1 Java 语言在内存当中默认使用的字符集

默认会用“Unicode”编码格式（字符集）来保存字符。

### 3.2 编码

把 Unicode 这种编码格式对应的字节数组，转换成某种本地编码格式（比如 GBK）对应的字节数组，从而保存。Unicode->GBK。

### 3.3 解码

把某种本地编码格式的字节数组转换成 Unicode 这种编码格式对应的字节数组。GBK-->Unicode。

- ◆ 注意事项：服务器默认使用 ISO-8859-1 编码格式，使用 1 个字节保存，无法存中文！因此中文会出现乱码。

### 3.4 Servlet 如何输出中文

需要调用：response.setContentType("text/html;charset=utf-8");其中 charset=utf-8 表示：

- 1) 用来指定编码格式，只要支持中文即可，比如也可设置为 charset=gbk。
- 2) 作用两个：
  - ①生成一个 content-type 消息头，告诉浏览器返回的数据类型和编码格式。
  - ②服务器在输出时，会使用指定的编码格式进行编码。

### 3.5 如果表单有中文参数值，也需要注意编码问题

因为，当表单提交的时候，浏览器会对表单中的数据进行编码（会使用打开表单时的编码格式进行编码），而服务器默认情况下，会使用 ISO-8859-1 去解码，所以，会产生乱码问题。

- 1) 解决方式一：

step1：先保证表单所在的页面按照指定的编码格式打开。即：

<meta http-equiv="content-type" content="text/html;charset=utf-8" />已是一种规范（模拟 content-type 消息头，告诉浏览器正在解析的数据类型和编码格式）。

step2：调用 request.setCharacterEncoding("utf-8");意思是告诉服务器，使用指定的编码格式进行解码。

- ◆ 注意事项：该方法只能用于"post"请求！注意代码放置顺序，在 request.getParameter()方法前。

- 2) 解决方式二：

step1：同方式一的第一步。

step2：使用 new String(str.getBytes("iso-8859-1"),"utf-8");

比如：String name=request.getParameter("uname");

name = new String(name.getBytes("iso-8859-1"),"utf-8");

### 3.6 案例：根据请求正确显示中文

step1：表单页面

```
<form method="post" action="hello" >
```

```
 请输入数量： <input name="qty" /> 打招呼的人： <input name="uname" />
```

```
<input type="submit" value="确定" /></form>
```

step2: HelloWorldServlet 类中 service 方法写

```
//request.setCharacterEncoding("utf-8");//方式一
String qtyStr=request.getParameter("qty");//读请求参数值
int qty=Integer.parseInt(qtyStr);//转成整数
String uname=request.getParameter("uname");
uname = new String(uname.getBytes("iso-8859-1"),"utf-8");//方式二
String rs="";//处理请求，生成处理结果
for(int i=0;i<qty;i++){ rs+="/** 输出处理结果，设置一个消息头 content-type，告诉浏览器返回的数据类型是一个
html 文档，以及编码格式。此外，还可以告诉服务器，在使用 out 输出时，使用指定的编码
格式进行编码 */
response.setContentType("text/html;charset=utf-8");
PrintWriter out=response.getWriter();//通过响应对象，获得一个输出流
out.println(rs);//调用流的方法进行输出，其实质是将处理结果写到了 response 对象
上
out.close();
```

### 3.7 将中文数据插入到数据库

程序从内存 Unicode 编码格式--->数据库中的某种本地格式。

step1: 要确保数据库支持中文，即正确设置数据库的字符集。

```
例如：建 MySql 的数据库时设置编码：create database db_chang default character set utf8;
```

step2: JDBC 驱动必须能够正确地进行编码和解码。有些 MySql 的驱动不能够正确进行编码和解码（默认使用 ISO-8859-1 进行编码和解码），可以在 JDBC 的 URL 后添加：

“useUnicode=true&characterEncoding=utf8”。

```
例如：conn=DriverManager.getConnection("jdbc:mysql://localhost:3306/db_chang?" +
"useUnicode=true&characterEncoding=utf8","root","");

```

◆ 注意事项：

- ❖ MySql 中 utf 不能写 “-”，记得数据库名后有个 “\_”。
- ❖ MySql 客户端也是一个程序，也会有编码问题，若你的程序中数据显示正常，而 MySql 客户端中查询的数据为乱码，则更改下编码，使用：set names gbk; 命令。

# 八十七、重定向

LICHOO

## 4.1 什么是重定向

服务器发送一个 302 状态码及一个 Location 消息头（值是一个地址，称为重定向地址），通知浏览器立即向重定向地址发请求。

## 4.2 如何重定向

使用 response.sendRedirect(String url);

## 4.3 注意两个问题

1) 重定向之前，不要调用 out.close();会报错！

```
out.println("添加成功");//能看到这个输出！(如果不写这个输出，则响应为空白页)
out.close();
response.sendRedirect("list");//看不到重定向结果！
```

2) 重定向之前，服务器会先清空 response 对象上缓存的数据。Servlet 只允许同时发送一个响应。

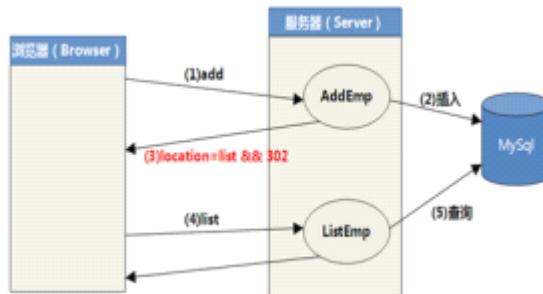
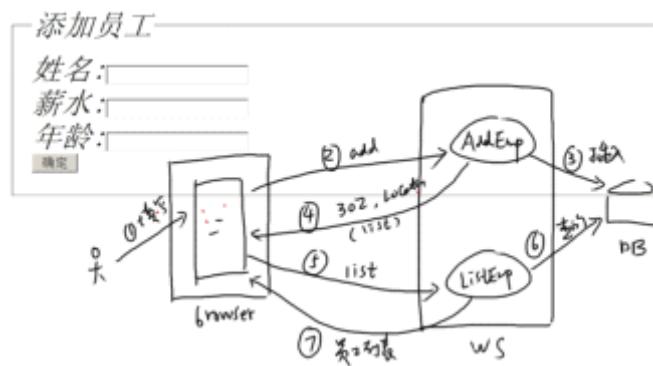
```
out.println("添加成功");//看不到这个输出！缓存数据被清空（响应一）
response.sendRedirect("list");//能看到重定向结果！（响应二）
out.close();
```

## 4.4 两个特点

1) 重定向的地址是任意的（前提要存在否则报 404）。

2) 重定向之后，浏览器地址栏的地址会变成重定向地址。

## 4.5 重定向原理图：以 2.8 案例为例

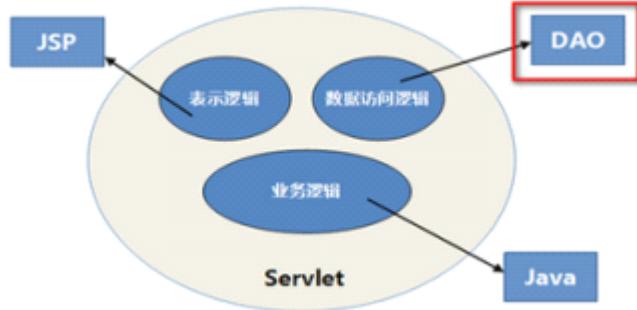


# 八十八、DAO

LICHOO

## 5.1 什么是 DAO

Data Access Object，即封装了数据访问逻辑的一个模块。



## 5.2 如何写一个 DAO

### step1：实体类

为了方便对数据库中的记录进行操作而封装的一个普通的 Java 类。比如，有一个表 chang\_emp，可以根据表中的字段设计一个 Employee 类，属性名与表中的字段名相同，且有 get/set 方法。

### step2：声明 DAO 接口

声明一系列方法，这些方法不应该涉及任何具体的数据库访问技术。比如，EmployeeDAO 接口：

```
public ResultSet findAll(); //错误的！因为 ResultSet 限制了只能是用 JDBC 来实现
public List<Employee> findAll() throws SQLException; //错误的！因为限制了只能是用 JDBC 来实现，SQLException 是 JDBC 的错误
public List<Employee> findAll(); //正确的！
public List<Employee> findAll() throws Exception; //正确的！
```

### step3：DAO 实现

使用具体的技术来实现 DAO 接口。比如，EmployeeDAOJdbcImpl 类

### step4：DAO 工厂

Connection conn=DriverManager.getConnection... //DriverManager 就是个工厂类，调用者通过 DriverManager 工厂类获得一个符合 Connection 接口要求的对象 conn

## 5.3 工厂类

封装了对象的创建细节，为调用者提供符合要求的对象。详细介绍见第六章。

# 八十九、工厂设计模式

LICHOO

## 6.1 什么是工厂

工厂是一个设计模式，所谓设计模式，是为解决一类相同或者相似的问题提出的一套解决方案，并且会对这个解决方案命名，比如工厂就是一个常用的设计模式。

## 6.2 使用工厂设计模式的好处

1) 工厂为调用者提供符合接口要求的对象(对这对象一般称为产品)，这样做的好处是，调用者不用了解对象的创建细节，这样，当对象创建过程发生改变，不会影响到调用者。

2) 工厂也可以从配置文件来获得接口与其对应的实现类的配置信息。这样，当实现类发生改变时，也不用再去修改工厂类的源代码了。

## 6.3 如何使用工厂模式

现在做这样一个假设，假如我们的底层数据访问不再使用 JDBC，而采用 Hibernate(框架技术，后续会学习，此处做了解)。

那么我们需要为 EmployeeDAO 增加一个实现 EmployeeDAOHibernateImpl.java

并且修改之前的代码，DelEmpServlet、AddEmpServlet、ListEmpServlet、LoadEmpServlet、ModifyEmpServlet 都要做相应修改(new 的地方都要换成 EmployeeDAOHibernateImpl 对象)。

## 6.4 案例：为 2.8 案例添加新功能，并使用 DAO 和工厂模式

step1：创建 DBUtil.java 类，专门用于连接和关闭数据库：

```
public static Connection getConnection() throws Exception{
 Connection conn=null;
 try { Class.forName("com.mysql.jdbc.Driver");
 conn=DriverManager.getConnection("jdbc:mysql://localhost:3306/db_chang?" +
 "useUnicode=true&characterEncoding=utf8","root","");
 } catch (Exception e) { e.printStackTrace(); throw e; }
 return conn;
}
public static void close(Connection conn){
 if(conn!=null){ try { conn.close(); } catch (SQLException e) {
 e.printStackTrace(); } }
}
```

step2：创建 Employee.java 实体类：有 id、name、salary、age 四个属性，以及 get/set 方法。

step3：创建 DAO 接口 EmployeeDAO

```
public List<Employee> findAll() throws Exception;
public void add(Employee e) throws Exception;
public void delete(long index) throws Exception;
public void modify(Employee e) throws Exception;
public Employee findById(long id) throws Exception;
```

step4：创建 DAO 接口的实现类 EmployeeDAOJdbcImpl：

```
public List<Employee> findAll() throws Exception{//查找所有员工
 List<Employee> employees=new ArrayList<Employee>();
 Connection conn=DBUtil.getConnection();
 Statement state=conn.createStatement();
```

```

 ResultSet rs=state.executeQuery("select * from chang_emp order by id");
 while(rs.next()) { int id=rs.getInt("id"); String name=rs.getString("name");
 double salary=rs.getDouble("salary"); int age=rs.getInt("age");
 Employee e=new Employee(); e.setId(id); e.setName(name);
 e.setSalary(salary); e.setAge(age); employees.add(e); }
 DBUtil.close(conn); return employees; }

public void add(Employee e) throws Exception{//添加员工
 Connection conn=DBUtil.getConnection();
 PreparedStatement prep=conn.prepareStatement(
 "insert into chang_emp(name,salary,age) value(?, ?, ?)");
 prep.setString(1, e.getName()); prep.setDouble(2, e.getSalary());
 prep.setInt(3, e.getAge()); prep.executeUpdate(); DBUtil.close(conn); }

public void delete(long index) throws Exception{//删除员工
 Connection conn=DBUtil.getConnection();
 PreparedStatement prep=conn.prepareStatement("delete from chang_emp where id=?");
 prep.setLong(1, index); prep.executeUpdate(); DBUtil.close(conn); }

public void modify(Employee e) throws Exception{//修改员工
 Connection conn=DBUtil.getConnection();
 PreparedStatement prep=conn.prepareStatement(
 "update chang_emp set name=?,salary=?,age=? where id=?");
 prep.setString(1, e.getName()); prep.setDouble(2, e.getSalary());
 prep.setInt(3, e.getAge()); prep.setLong(4, e.getId()); prep.executeUpdate();
 DBUtil.close(conn); }

public Employee findById(long id) throws Exception {//根据 id 查找员工
 Connection conn=DBUtil.getConnection();
 PreparedStatement prep=conn.prepareStatement("select * from chang_emp where
id=?");
 prep.setLong(1, id); ResultSet rs=prep.executeQuery(); Employee e=null;
 if(rs.next()) { e=new Employee(); e.setId(id); e.setName(rs.getString("name"));
 e.setSalary(rs.getDouble("salary")); e.setAge(rs.getInt("age")); }
 DBUtil.close(conn); return e; }
}

```

step5：创建 Factory.java 工厂类：

```

public static Object getInstance(String type){ Object obj=null;
if("EmployeeDAO".equals(type)){
 obj=new EmployeeDAOJdbcImpl();
 /** obj=new EmployeeDAOHibernateImpl(); 当不再使用 JDBC 连接数据库时，如
HibernateI 连接时，就需要更改每个 new EmployeeDAOJdbcImpl(); 不方便维护。使用工厂模
式就可以统一更改了 */
 return obj; }
}

```

step6：修改 AddEmpMySql.java 中的 try 块， finally 块也不需要了（DAO 中调用了 DBUtil 的关闭）

```

try { EmployeeDAO dao=(EmployeeDAO) Factory.getInstance("EmployeeDAO");
 Employee e=new Employee(); e.setName(name); e.setSalary(salary);
 e.setAge(age); dao.add(e); response.sendRedirect("list"); }
}

```

step7：修改 ListEmpServlet.java 中的 try 块，finally 块也不需要了（DAO 中调用了 DBUtil 的关闭）

```
try { EmployeeDAO dao=(EmployeeDAO) Factory.getInstance("EmployeeDAO");
 List<Employee> Employees=new ArrayList(); Employees=dao.findAll();
 out.println("<style>tr:hover {color:red;background:silver;}</style>" +
 "<table border='1' width='60%' cellpadding='1' cellspacing='1'>");
 out.println("<tr><td>id</td><td>姓名</td><td>薪水</td><td>年龄</td>" +
 "<td>操作</td></tr>");
 for(int i=0;i<Employees.size();i++){ Employee e=Employees.get(i);
 out.println("<tr><td>" + e.getId() + "</td><td>" + e.getName() + "</td>" +
 "<td>" + e.getSalary() + "</td><td>" + e.getAge() + "</td>" +
 "<td>删除&nbsp" +
 "修改</td></tr>"); }
 out.println("</table>"); out.println("再次添加新员工"); }
```

step8：增加 DelEmpServlet.java 类用于删除员工，其中 service 方法内容为：

```
response.setCharacterEncoding("utf-8"); PrintWriter out=response.getWriter();
long id=Long.parseLong(request.getParameter("id"));
try{ EmployeeDAO dao=(EmployeeDAO) Factory.getInstance("EmployeeDAO");
 dao.delete(id); response.sendRedirect("list");
} catch(Exception e){ e.printStackTrace(); out.println("删除失败！请稍后再试！"); }
```

step9：增加 LoadEmpServlet.java 类用于读取员工，其中 service 方法内容为：

```
response.setCharacterEncoding("utf-8"); PrintWriter out=response.getWriter();
long id = Long.parseLong(request.getParameter("id"));
try{ EmployeeDAO dao=(EmployeeDAO) Factory.getInstance("EmployeeDAO");
 Employee e=dao.findById(id); response.setContentType("text/html;charset=utf-8");
 out.println("<form action='modify?id=" + id + "' method='post'>");
 out.println("id: " + id + "
");
 out.println("姓名: <input name='name' value='" + e.getName() + "' />
");
 out.println("薪水: <input name='salary' value='" + e.getSalary() + "' />
");
 out.println("年龄: <input name='age' value='" + e.getAge() + "' />
");
 //或 out.println("<input type='hidden' name='id' value='" + id + "' />"); out.println("<input type='submit' value='确定' />"); out.println("</form>"); }
catch(Exception e){ e.printStackTrace(); out.println("读取失败！稍后再试！"); }
```

step10：增加 ModifyEmpServlet.java 类用于修改员工，其中 service 方法内容为：

```
response.setCharacterEncoding("utf-8"); request.setCharacterEncoding("utf-8");
PrintWriter out = response.getWriter(); long id =
Long.parseLong(request.getParameter("id"));
String name=request.getParameter("name");
double salary=Double.parseDouble(request.getParameter("salary"));
int age=Integer.parseInt(request.getParameter("age"));
try { EmployeeDAO dao=(EmployeeDAO) Factory.getInstance("EmployeeDAO");
 Employee e=new Employee(); e.setId(id); e.setName(name); e.setSalary(salary);
 e.setAge(age); dao.modify(e); response.sendRedirect("list"); }
```

```
 } catch (Exception e) { e.printStackTrace(); out.println("修改失败！稍后重试"); }
```

step11: 在浏览器地址栏输入 <http://localhost:8080/> 应用名/list 则访问出员工列表页面。

## 九十、Servlet 容器处理细节

LICHOO

### 7.1 Servlet 容器如何处理请求资源路径以及匹配

比如：我们在浏览器地址栏输入：`http://ip:port/appname/abc.html` 浏览器会将`"/appname/abc.html"`作为请求资源路径发送给 Servlet 容器。

step1：Servlet 容器会先假设访问的是一个 Servlet，会依据应用名（`appname`）找到应用所在的文件夹，然后找到 `web.xml` 文件。

step2：匹配`<url-pattern>`

- 1) 精确匹配（完全匹配）：“/”、大小写、名字完全一样。
- 2) 通配符匹配：使用“\*”来匹配任意长度的字符串，比如：`<url-pattern>/*</url-pattern>`
- 3) 后缀匹配：使用“\*.”开头，后接任意的字符串，比如：`<url-pattern>*.do</url-pattern>`

◆ 注意事项：`*.do` 表示匹配所有以`.do` 结尾的请求，注意不写“/”。

比如地址栏输入：`/del.do` 正确      `/aaaaaaaa/bbbbbbb.do` 正确

step3：如果都不匹配，则容器认为访问的是一个静态资源文件（比如 `html` 文件），然后容器会查找该文件，如果找到则返回，否则会返回 404。

### 7.2 一个 Servlet 如何处理多种请求

step1：使用后缀匹配模式，比如`<url-pattern>*.do</url-pattern>`

step2：分析请求资源路径，然后依据分析的结果分别进行不同的处理，需使用 `String request.getRequestURI();`

◆ 注意事项：是 URI 不是 URL。

# 九十一、Servlet 的生命周期

LICHOO

## 8.1 Servlet 的生命周期的含义

Servlet 容器如何去创建 Servlet 对象，如何给 Servlet 对象分配资源，如何调用 Servlet 对象的方法来处理请求，以及如何去销毁 Servlet 对象的整个过程。

## 8.2 Servlet 生命周期的四个阶段

实例化、初始化、就绪、销毁。

## 8.3 实例化

- 1) 指的是容器调用 Servlet 的构造器，创建 Servlet 对象。
- 2) 何时实例化？

情况 1：容器收到请求之后才创建 Servlet 对象。在默认情况下，容器只会为 Servlet 创建唯一的一个实例（多线程，有安全问题。每次请求创建一个线程，由线程去调用方法）。

情况 2：容器事先（容器启动时）将某些 Servlet（需要配置 load-on-startup 参数）对象创建好。load-on-startup 参数值必须是 $\geq 0$  的整数，越小，优先级越高（即先被实例化）。参数加在 web.xml 配置文件里的某个<servlet>标签里，如<load-on-startup>1</load-on-startup>

◆ 注意事项：注意标签的先后顺序，是有要求的，要符合 .xsd 文档要求。

```
<servlet><servlet-name></servlet-name><servlet-class></servlet-class>
 <load-on-startup>1</load-on-startup></servlet>
```

## 8.4 初始化

- 1) 指的是容器在创建好 Servlet 对象之后，会立即调用 Servlet 对象的 init 方法。
- 2) init 方法：
  - ① init 方法只会执行一次。  
② GenericServlet 已经实现了 init 方法，该方法会将容器创建好的 ServletConfig 对象作为参数传递给 init 方法。  
③ ServletConfig 对象提供了一个 getInitParameter 方法来访问 Servlet 的初始化参数。  
step1：在 web.xml 文件里，使用<init-param>来配置初始化参数，加在某个<servlet>标签里。

```
<servlet><init-param>
 <param-name>name</param-name>
 <param-value>常</param-value>
</init-param></servlet>
```

step2：使用 String getInitParameter(String paraName) 获取初始化参数。

④ 如果 GenericServlet 的 init 方法提供的初始化操作不满足需要，可以覆盖（override）init()方法，覆盖的是不带参数的，对有参数的无影响。有参数的 init 方法执行时，同时无参的也被执行（有参的 init 方法里调用了无参的 init 方法），叫钩子方法。

## 8.5 就绪

- 1) 就绪指的是 Servlet 对象可以接受调用了，容器收到请求之后，会调用 Servlet 对象的 service 方法来处理，且可以被执行多次。
- 2) HttpServlet 已经实现了 service 方法，该方法会依据请求类型（get/post）分别调用 doGet 或 doPost 方法。所以我们在写一个 Servlet 时，有两种选择：

选择一：覆盖 HttpServlet 的 doGet, doPost 方法

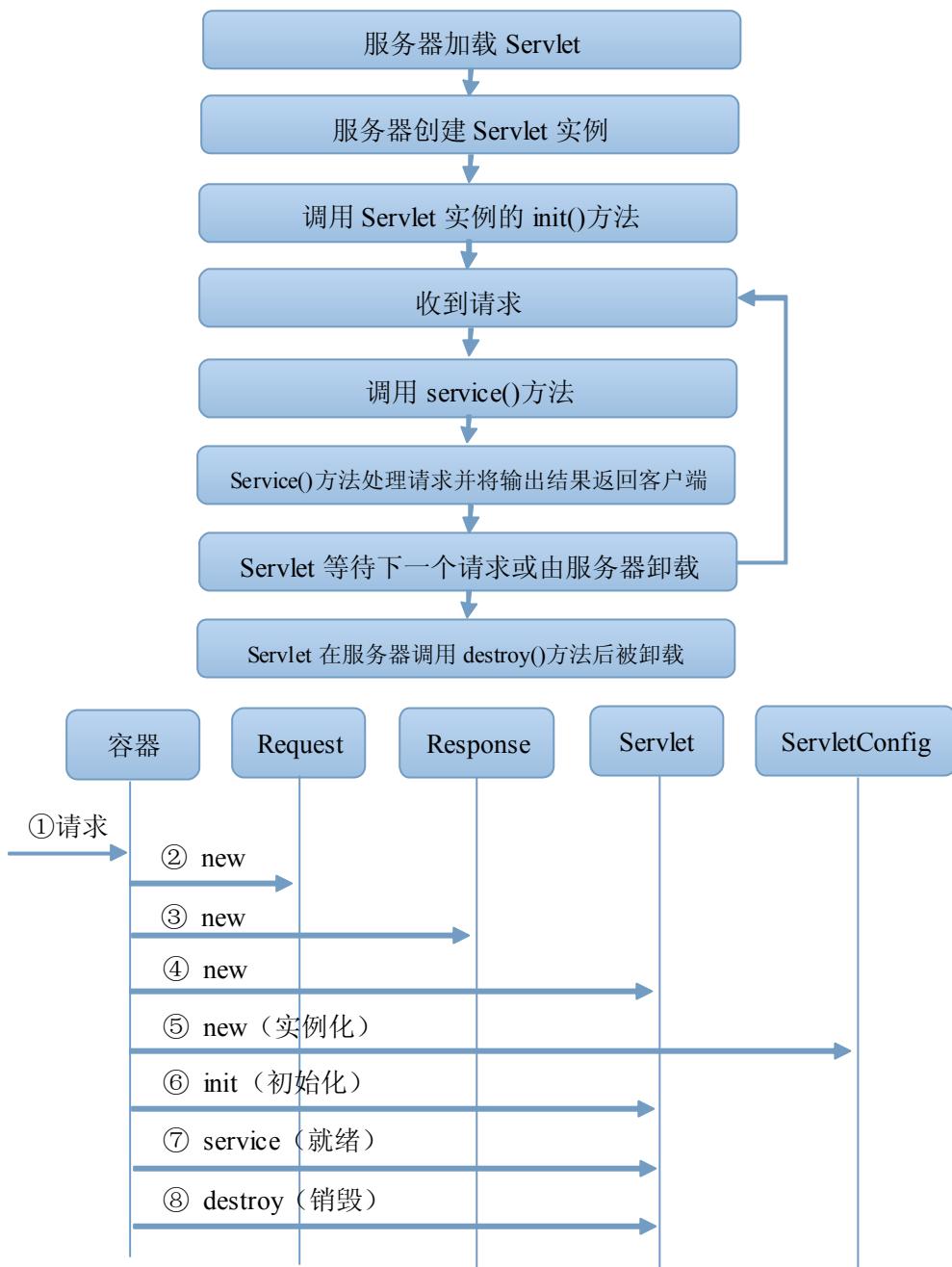
选择二：覆盖 HttpServlet 的 service 方法

LICHOO

## 8.6 销毁

销毁指的是 Servlet 容器在销毁 Servlet 对象之前，会调用 destroy 方法，且只会执行一次。

## 8.7 Servlet 生命周期图



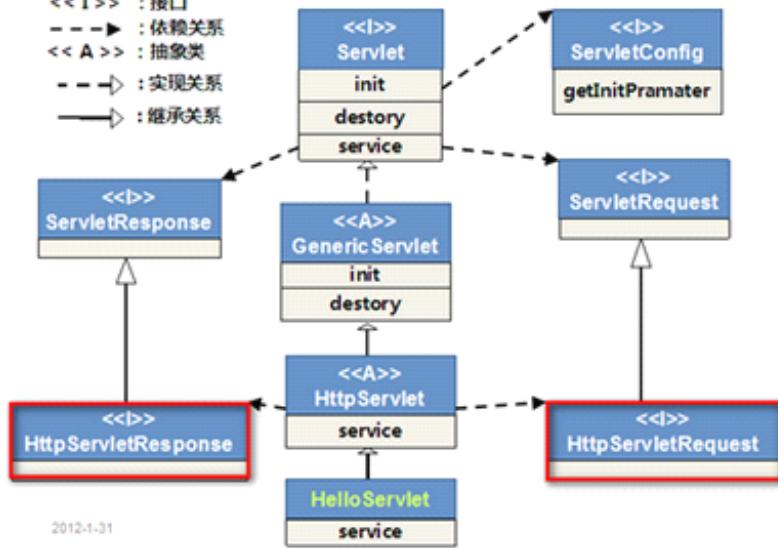
## 8.8 Servlet 生命周期相关的几个接口与类

- 1) Servlet 接口：
  - ① init(ServletConfig config)//有参的 init 方法
  - ② service(ServletRequest req, ServletResponse res)    ③ destroy()

- 2) GenericServlet 抽象类：实现了 Servlet 接口中的 init, destroy 方法
- 3) HttpServlet 抽象类：继承了 GenericServlet 抽象类，实现了 service 方法
- 4) ServletConfig 接口：String getInitParameter(String paraName)
- 5) ServletRequest 接口是 HttpServletRequest 的父接口
- 6) ServletResponse 接口是 HttpServletResponse 的父接口

说明：

<< I >> : 接口  
--> : 依赖关系  
<< A >> : 抽象类  
- -> : 实现关系  
—> : 继承关系



2012-1-31

## 九十二、JSP（简要介绍，详细内容见 JSP 笔记）

### 9.1 什么是 JSP

JSP (Java Server Page) 是 Java 服务器端动态页面技术。是 sun 公司制订的一种服务器端的动态页面生成技术规范。

### 9.2 为什么要使用 JSP

因为直接使用 Servlet，虽然也可以生成动态页面。但是，编写繁琐（需要使用 `out.println` 来输出），并且维护困难（如果页面发生了改变，需要修改 Java 代码），所以 sun 指定了 JSP 规范。

### 9.3 JSP 与 Servlet 的关系

JSP 其实是一个以.jsp 为后缀的文件，容器会自动将.jsp 文件转换成一个.java 文件（其实就是一个 Servlet），然后调用该 Servlet。所以，从本质上讲，JSP 其实就是一个 Servlet。

### 9.4 如何写一个 JSP 文件

step1：创建一个以“.jsp”为后缀的文件

step2：在该文件里面，可以添加如下的内容

- 1) HTML (CSS、JS): 直接写即可
- 2) Java 代码: 形式一: Java 代码片段: `<% Java 代码 %>`  
形式二: JSP 表达式: `<%= Java 表达式 %>`  
形式三: JSP 声明: `<%! %>`
- 3) 指令

### 9.5 JSP 是如何运行的

step1：容器依据.jsp 文件生成.java 文件（也就是先转换成一个 Servlet）

- 1) HTML (CSS、JS) 放到 service 方法里，使用 `out.write` 输出
- 2) `<% %>` 也放到 service 方法里，照搬，不改动。
- 3) `<%= %>` 也会放到 service 方法里，使用 `out.print` 输出。
- 4) `<%! %>` 给 Servlet 添加新的属性或者新的方法。

这样就把一个 JSP 变成了一个 Servlet 容器。

- ◆ 注意事项：`out.writer` 方法只能输出简单的字符串，对象是没法输出的。优点是把 null 自动转换成空字符串输出。如：`<% out.println(new Date()); %>` 不能用 `writer`

step2：容器接下来就会调用 Servlet 来处理请求了（会将之前生成的.java 文件进行编译，然后实例化、初始化、调用相应的方法处理请求）

step3：隐含对象

- 1) 所谓隐含对象（共 9 个，详细见 JSP 总结），指的是在.jsp 文件里面直接可以使用的对象，比如 `out`、`request`、`response`、`session`、`application` (`ServletContext` 上下文) 等。
- 2) 之所以能直接使用这些对象，是因为容器会自动添加创建这些对象的代码 (JSP 仅仅是个草稿，最终会变为一个 Servlet)。

## 9.6 指令

### 1) 指令是什么

通知容器，在将.jsp 文件转换成.java 文件时，作一些额外的处理，比如导包。

### 2) 指令的语法

```
<%@指令名称 属性名=属性值 %>
```

### 3) page 指令（更多属性，见 JSP 笔记）

①import 属性：导包

例如：`<%@page import="java.util.*"%><!-- 注意：没有分号！-->`

```
<%@page import="java.util.* ,java.text.*"%>
```

`<!-- 多个包以逗号隔开！都在一个双引号里-->`

eg：页面显示系统时间

```
current time:<% out.println(new Date()); %>

```

```
current time:<%=new Date()%>
```

◆ 注意事项：两个是等价，new Date()会被放入 out.print();里

②contentType 属性：设置 response.setContentType 的内容

例如：`<%@page import="java.util.*" contentType="text/html;charset=utf-8" %>`

`<!-- 属性之间用空格隔开 -->`

③pageEncoding 属性：告诉容器.jsp 的文件的编码格式，这样，容器杂在获取.jsp 文件的内容（即解码）时，不会出现乱码。最好加入，有些容器默认以 ISO 8859-1 编码。

例如：`<%@page contentType="text/html;charset=utf-8" pageEncoding="utf-8" %>`

◆ 注意事项：页面最好添加：`<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">`，会影响到 CSS 样式，甚至 JS。如 IE 有个混杂模式，即自动降级，为了兼容老的页面。

### 4) include 指令

对于页面的公共部分，我们可以使用相同的.jsp 文件，并使用 include 指令导入，如此实现代码的优化。

告诉容器，在将.jsp 文件转换成.java 文件时，在指令所在的位置插入相应的文件的“内容”。插入的页面并未运行，而是机械的将内容插入。

比如：`<%@include file="head.jsp" %>`

## 9.7 案例：创建 emplist.jsp 页面，将表示逻辑交给 JSP 处理

```
<%@page contentType="text/html;charset=utf-8" pageEncoding="utf-8"
 import="java.util.* ,dao.* ,util.* ,entity.* ,dao.impl.*" %>
<head>
 <style>
 .row1 {background-color:silver;}
 .row2 {background-color:yellow;}
 </style>
</head>
<body style="font-size:20px">
 <table cellpadding="0" cellspacing="0" border="1" width="60%">
 <tr><td>ID</td><td>姓名</td><td>薪水</td><td>年龄</td></tr>
 <% //使用 DAO 来访问数据库 %>
 </table>
</body>
</html>
```

```
EmployeeDAO dao=(EmployeeDAO)Factory.getInstance("EmployeeDAO");
List<Employee> employees=dao.findAll();
for(int i=0;i<employees.size();i++){
 Employee e=employees.get(i);
 %>
 <tr class="row<%=i%2+1 %>">
 <td><%=e.getId()%></td>
 <td><%=e.getName()%></td>
 <td><%=e.getSalary()%></td>
 <td><%=e.getAge()%></td>
 </tr>
 <%
 }
 %>
</table>
</body>
```

## 九十三、请求转发

LICHOO

### 10.1 什么是转发

一个 Web 组件（Servlet/JSP）将未完成的处理通过容器转交给另外一个 Web 组件继续完成。常见的情况是：一个 Servlet 将数据处理完毕之后，转交给一个 JSP 去展现。

### 10.2 如何转发

step1: 绑定数据到 request: request 里有个 HashMap。

```
request.setAttribute(String name, Object obj); //name: 绑定名。obj: 绑定值。
```

另一个方法获取绑定值：

```
Object request.getAttribute(String name); //如果绑定对象的值不存在，会返回 null
```

step2: 获得一个转发器: url: 要转发给哪一个 Web 组件

```
RequestDispatcher rd = request.getRequestDispatcher(String url);
```

step3: 转发

```
rd.forward(request, response); //JSP 和 Servlet 会共享相同的请求和响应对象
```

step4: 在转发的目的地，可以使用 request.getAttribute 方法获得绑定的数据，然后进行处理。

### 10.3 编程需要注意的两个问题

1) 转发之前，先清空 response 对象中缓存的数据。

```
out.println("hello"); //看不到结果
```

```
rd.forward(request, response); //能看到结果
```

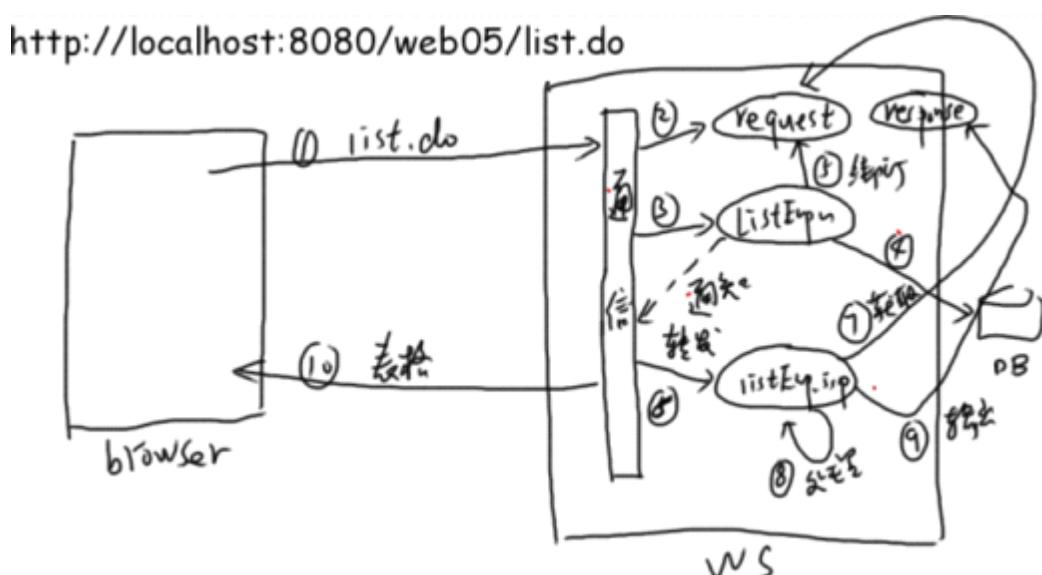
2) 转发之前，不能调用 out.close() 或着 out.flush 只能同时处理一个响应

```
out.println("hello"); //看到结果
```

```
out.close();
```

```
rd.forward(request, response); //看不到结果
```

### 10.4 转发原理图

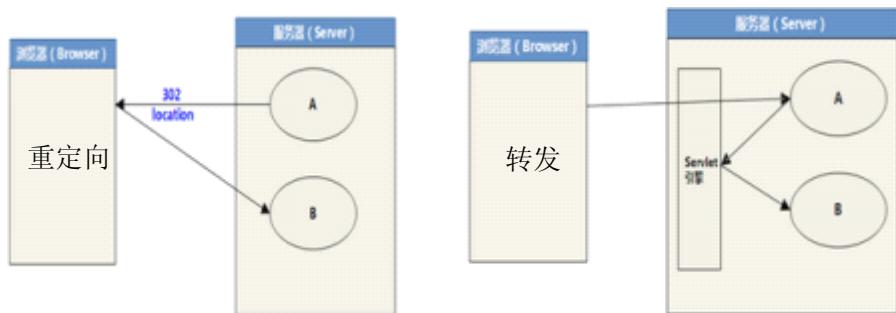


## 10.5 转发的特点

- 1) 转发的目的地只能是同一个应用内部的某个组件的地址。
- 2) 转发之后，浏览器地址栏的地址不变。
- 3) 转发所涉及的各个 Web 组件可以共享同一个 request 对象和 response 对象

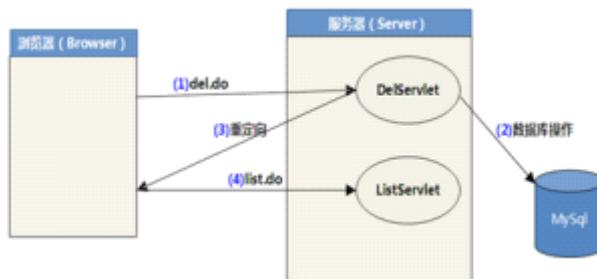
## 10.6 转发和重定向的区别

- 1) 转发的目的地只能是同一个应用内部某个组件的地址，而重定向的目的地是任意的。
- 2) 转发之后，浏览器地址栏的地址不变，而重定向会变。
- 3) 转发所涉及的各个 Web 组件可以共享 request 对象，而重定向不可以。
- 4) 转发是一件事情未做完，而重定向是一件事情已经做完。



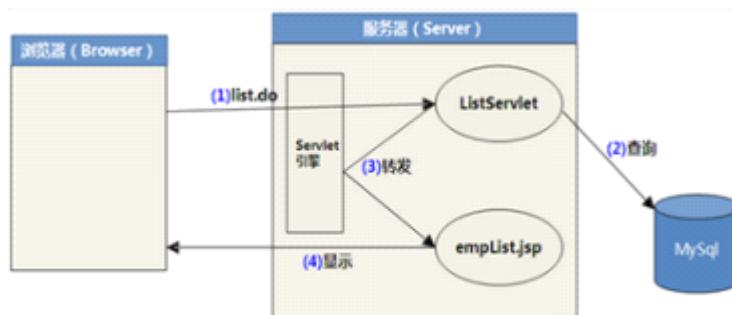
## 10.7 何时用重定向

- 1) 比如用户作删除操作时，删除操作已做完，重定向访问 list.do



## 10.8 何时用转发

- 1) 用户调用 list.do
- 2) 有 ListServlet 到数据库查询数据
- 3) ListServlet 将查询到的结果通过 Servlet 引擎(通信模块)转发给负责显示的 empList.jsp
- 4) empList.jsp 将数据通过友好的界面显示给用户，比如用户作删除操作时，删除操作已做完，重定向访问 list.do



## 10.9 案例：修改 6.4 案例中 step7 中的 ListEmpServlet.java

step1：删除 try 块里面的输出流 out.println();即将表示逻辑删除。使用转发，将表示逻辑交给 emplist.jsp 展示

```
try{ //使用工厂来访问数据库
 EmployeeDAO dao=(EmployeeDAO) Factory.getInstance("EmployeeDAO");
 List<Employee> employees=dao.findAll();
 request.setAttribute("employees", employees); //转发 step1:绑定数据
 RequestDispatcher rd=request.getRequestDispatcher("emplist.jsp"); //step2:获得转发器
 rd.forward(request, response); //step3:转发
}
```

step2：将 9.7 案例中的 emplist.jsp 修该为接收绑定的数据，注意导包

```
<tr><td>ID</td><td>姓名</td><td>薪水</td><td>年龄</td></tr>
<% List<Employee> employees=
 (List<Employee>)request.getAttribute("employees"); //接收绑定的数据
for(int i=0;i<employees.size();i++){
 Employee e=employees.get(i);
%>
<tr class="row<%=i%2+1 %>">
 <td><%=e.getId()%></td>
 <td><%=e.getName()%></td>
 <td><%=e.getSalary()%></td>
 <td><%=e.getAge()%></td>
 <td><a href="del.do?id=<%=e.getId()%>" onclick="return confirm('确定删除 <%=e.getName()%> 吗？')">删除员工

 <a href="load.do?id=<%=e.getId()%>">修改员工
 </td></tr>
<%
}
%>
```

step3：同理，将其他 Servlet 中有表示逻辑的都使用转发，并创建相应的.jsp 页面，无表示逻辑的使用重定向。将几种 Servlet 合并到一个 Servlet 中，命名为 ActionServlet.java，其中的 service 方法内容为：

```
request.setCharacterEncoding("utf-8");
response.setContentType("text/html;charset=utf-8");
PrintWriter out=response.getWriter();
String uri=request.getRequestURI();
String action=uri.substring(uri.lastIndexOf("/"),uri.lastIndexOf(".")); //截取从 "/" 开始到"."
结束，不包括" ." 的字符串
if(action.equals("/list")){//使用工厂来访问数据库
 try { EmployeeDAO dao=(EmployeeDAO) Factory.getInstance("EmployeeDAO");
 List<Employee> employees=dao.findAll();
 request.setAttribute("employees", employees); //转发 step1:绑定数据
 }
}
```

```

RequestDispatcher rd=request.getRequestDispatcher("emplist.jsp");//step2:获得转发器
 rd.forward(request, response);//step3:转发
 }catch(Exception e) { e.printStackTrace();out.println("系统繁忙！稍后再试"); }
}else if(action.equals("/add")){
 String name=request.getParameter("name");
 Double salary=Double.parseDouble(request.getParameter("salary"));
 int age=Integer.parseInt(request.getParameter("age"));
 try {//将员工信息插入到数据库
 EmployeeDAO dao=(EmployeeDAO) Factory.getInstance("EmployeeDAO");
 Employee e=new Employee(); e.setName(name); e.setSalary(salary);
 e.setAge(age); dao.add(e); response.sendRedirect("list.do");
 } catch (Exception e) { e.printStackTrace(); out.println("添加失败！稍后再试"); }
}else if(action.equals("/del")){
 long id=Long.parseLong(request.getParameter("id"));
 try{ EmployeeDAO dao=(EmployeeDAO) Factory.getInstance("EmployeeDAO");
 dao.delete(id); response.sendRedirect("list.do");
 }catch(Exception e){ e.printStackTrace(); out.println("删除失败！稍后再试"); }
}else if(action.equals("/load")){
 long id = Long.parseLong(request.getParameter("id"));
 try{ EmployeeDAO dao=(EmployeeDAO) Factory.getInstance("EmployeeDAO");
 Employee e = dao.findById(id); request.setAttribute("e_modify", e);//转发
 request.getRequestDispatcher("modifyEmp.jsp").forward(request,response);
 }catch(Exception e){ e.printStackTrace(); out.println("读取失败！稍后再试"); }
}else if(action.equals("/modify")){
 long id = Long.parseLong(request.getParameter("id"));
 String name=request.getParameter("name");
 double salary=Double.parseDouble(request.getParameter("salary"));
 int age=Integer.parseInt(request.getParameter("age"));
 try{ EmployeeDAO dao=(EmployeeDAO) Factory.getInstance("EmployeeDAO");
 Employee e=new Employee(); e.setId(id); e.setName(name);
 e.setSalary(salary); e.setAge(age); dao.modify(e);
 response.sendRedirect("list.do");
 }catch(Exception e){ e.printStackTrace(); out.println("修改失败！稍后再试！"); }
}
}

```

step4: web.xml 配置如下

```

<servlet>
 <servlet-name>ActionServlet</servlet-name>
 <servlet-class>web.ActionServlet</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>ActionServlet</servlet-name>
 <url-pattern>*.do</url-pattern>
</servlet-mapping>

```

## 九十四、异常的处理

LICHOO

### 11.1 用输出流 out

```
catch(Exception e) { e.printStackTrace(); out.println("系统繁忙，稍后再试！"); }
```

### 11.2 用转发的方式

不用输出流 out 了。

step1：Servlet 中 catch 块

```
catch(Exception e) {
 request.setAttribute("syserror", "系统繁忙，稍后重试！");
 request.getRequestDispatcher("error.jsp").forward(request, response);
}
```

step2：error.jsp 页面

```
<body style="font-size:30px;font-style:italic;">
<% String msg = (String)request.getAttribute("syserror"); %>
<%=msg%>
</body>
```

### 11.3 让容器处理系统异常

step1：将异常抛给容器

不用输出流 out 了。

```
catch(Exception e) {
 e.printStackTrace();//写 throw e;报错，子类异常个数不能大于父类
 throw new ServletException(e);//这里将异常转化
}
```

step2：编写一个错误处理页面，比如 system\_error.jsp

```
<body style="font-size:30px;font-style:italic;color:red;">系统繁忙，稍后重试。</body>
```

step3：配置错误处理页面（让容器知道，当捕获到了相应的异常后，应该调用哪一个页面）

```
<!-- 配置错误处理页面 -->
<error-page>
 <exception-type>javax.servlet.ServletException</exception-type>
 <location>/system_error.jsp</location>
</error-page>
```

### 11.4 案例：将 10.9 案例中的 step3 中的所有 catch 块修改

修改为将异常抛给容器，具体步骤见 11.3 节。

## 九十五、路径问题

即链接地址、表单提交地址、重定向、转发这四种情况下，如何写正确的路径（地址）。

```

<form action="add.do">
response.sendRedirect("list.do");
request.getRequestDispatcher("emplist.jsp");
```

### 12.1 什么是相对路径

不以“/”开头的路径。

- ◆ 注意事项：
  - ❖ <a href="../a2.jsp">点击访问 a2.jsp</a><!-- .. / 的用法为：向上跳一级 -->
  - ❖ 相对路径较易出错，在实际开发中建议使用绝对路径。

### 12.2 什么是绝对路径

以“/”开头的路径。

### 12.3 如何写绝对路径

链接地址、表单提交地址、重定向的绝对路径，应该从“应用名”开始写，而转发应该从“应用名之后”开始写。可看案例 12.5。

### 12.4 如何防止硬编码

所谓硬编码就是固定、写死在某个文件中，如 JSP 页面中。如果 Web 应用的应用名被更改，则应用会因为路径问题而无法正常运行。

解决此问题需要使用 String request.getContextPath() 方法来即时获取应用名。

### 12.5 案例：相对、绝对路径对比

应用名 web05\_path 中的 WebRoot 文件夹下有 a2.jsp 和 app01 文件夹，而 app01 文件夹中有 a1.jsp。

a1.jsp 页面：

```
<body style="font-size:30px;color:red;"><!-- a1.jsp -->
 a1 的内容……

 使用相对路径
访问 a2.jsp

 使用绝对路径：硬编码
访问 a2.jsp

 使用绝对路径：方法

 <a href="<% =request.getContextPath() %>/a2.jsp">访问 a2.jsp </body>
```

a2.jsp 页面

```
<body style="font-size:30px;color:red;"><!-- a2.jsp -->
 a2 的内容……

 使用相对路径
访问 a1.jsp

 使用 绝 对 路 径： 硬 编 码
 访 问
 a1.jsp

 使用绝对路径：方法

 <a href="<% =request.getContextPath() %>/app01/a1.jsp">访问 a1.jsp </body>
```

## 12.6 四种情况下，正确的绝对路径写法

1) 链接地址：从“应用名”开始写

```
<a href="<% =request.getContextPath() %>/app01/a1.jsp">点击，访问 a1.jsp
```

2) 表单提交地址：从“应用名”开始写

```
<form method="post" action="<% =request.getContextPath() %>/add">……</form>
```

3) 重定向而：从“应用名”开始写

```
response.sendRedirect(request.getContextPath() + "/app01/a1.jsp");
```

4) 转发：从“应用名之后”开始写

```
request.getRequestDispatcher("app01/a1.jsp").forward(request, response);
```

# 九十六、状态管理

LICHOO

## 13.1 什么是状态管理

将浏览器与 Web 服务器之间多次交互当作一个整体来看待，并且将多次交互所涉及的数据保存下来。

## 13.2 如何进行状态管理

第一大类：客户端状态管理技术，即将状态（也就是多次交互所涉及的数据）保存在客户端（浏览器）。

第二大类：服务器端状态管理技术，即将状态保存在 Web 服务器端。

## 13.3 cookie

什么是 cookie：①是一种客户端的状态管理技术。②当浏览器访问服务器的时候，服务器可以将少量的数据以 set-cookie 消息头的方式发送给浏览器，浏览器会将这些数据保存下来。当浏览器再次访问服务器时，会将之前保存的这些数据以 cookie 消息头的方式发送给服务器。

◆ 注意事项：有几个 cookie，就有几个 set-cookie 消息头。

## 13.4 如何创建一个 cookie

step1：创建 Cookie

```
Cookie c=new Cookie(String name,String value); //name: cookie 的名称。value: cookie 的值
```

step2：添加 Cookie： response.addCookie(c); //调用响应对象的 addCookie 方法

例如：AddCookieServlet 类中的 service 方法中写

```
response.setContentType("text/html;charset=utf-8");
PrintWriter out=response.getWriter();
Cookie c1=new Cookie("username","tom"); response.addCookie(c1);
Cookie c2=new Cookie("city","beijing");
response.addCookie(c2); //消息头中将会有两个 set-cookie
out.println("<h1>添加 Cookie 成功</h1>"); out.close();
```

## 13.5 查询 cookie

1) 方法：Cookie[] request.getCookies(); //如果没有任何的 cookie，则返回 null

String cookie.getName(); //返回 cookie 的名称

String cookie.getValue(); //返回 cookie 的值

2) 案例：FindCookieServlet 类中的 service 方法中写

```
response.setContentType("text/html;charset=utf-8");
PrintWriter out = response.getWriter(); Cookie[] cookies=request.getCookies();
if(cookies!=null){ for(int i=0;i<cookies.length;i++){
 Cookie c=cookies[i]; String name=c.getName();
 String value=c.getValue();
 out.println("name: "+name+", value: "+value+"
"); }
} else{ out.println("没有 cookie"); }
```

## 13.6 编码问题

cookie 的值或者名称只允许合法的 ASCII 码字符串，如果是中文，需要将中文转换成 ASCII 码字符串。

- 需要用到的两个方法

```
String URLEncoder.encode(String str, String code);
String URLDecoder.decode(String str, String code);
```

- 例如：上例代码修改如下

```
Cookie c1=new Cookie("username",URLEncoder.encode("常","utf-8"));
out.println("name: "+name+", value: "+URLDecoder.decode(value,"utf-8")+"
");
```

## 13.7 cookie 的生存时间

默认情况下，浏览器会将 cookie 保存在内存里，只要浏览器不关闭，cookie 会一直存在。

- 启动 FireFox 后，操作系统会为浏览器开辟一块内存空间

- ①浏览器（FireFox）访问服务器。
- ②服务器生成消息头 setCookie 发回给浏览器。
- ③浏览器将服务器生成的消息头保存到内存区域。
- ④当浏览器关闭，系统回收为浏览器分配的内存，cookie 也相应消失。

- 设置生存时间：使用 cookie.setMaxAge(int seconds)方法。

- ◆ 注意事项：①单位是秒。②当 seconds>0 时，浏览器会将 cookie 保存在硬盘上，当 cookie 保存的时间超过了 seconds，则 cookie 会被浏览器删除。③当 seconds<0 时，缺省值（浏览器会将 cookie 保存在内存里）。④当 seconds=0 时，删除 cookie，所以 Cookie 中无删除方法，设置为 0 即可。而修改，则为覆盖，用相同的 name。

- 案例

eg1：删除名称为 userId 的 cookie

```
Cookie c=new Cookie("userId","");
c.setMaxAge(0);
response.addCookie(c);
```

eg2：读取名为 name，值为 zs 的 cookie，若不存在则创建。service 方法如下

```
response.setContentType("text/html;charset=utf-8");
PrintWriter out=response.getWriter();
Cookie[] cookies=request.getCookies();
if(cookies!=null){
 boolean flag=false;
 for(int i=0;i<cookies.length;i++){
 Cookie c=cookies[i];
 if(c.getName().equals("name")){
 out.println(c.getValue());
 flag=true;
 }
 if(!flag){
 Cookie c=new Cookie("name","zs");
 response.addCookie(c);
 }
 }
}
```

## 13.8 cookie 的路径问题

1) 路径问题指的是：浏览器在向服务器上的某个地址发请求时，会比较请求地址与 cookie 的路径是否匹配，只有匹配的 cookie 才会发送。

2) cookie 有一个默认的路径，值等于创建该 cookie 的组件的路径，比如：/web06\_Cookie/

app01/addCookie.jsp 创建了一个 cookie，则该 cookie 默认的路径为：/web06\_Cookie/app01。

3) 匹配规则：只有当访问的地址是 cookie 的路径或者其子路径时，浏览器才会将这个 cookie 进行发送。

4) 案例：若 cookie 的默认路径是 “/web\_Cookie/app01”，则访问。

step1: addCookie.jsp 页面，且放在 web06\_Cookie/app01 目录中，并执行

```
<% Cookie c=new Cookie("userid","abc"); response.addCookie(c); %>
add cookie success.
```

step2: findCookie1.jsp 页面（2、3 相同），1 放在 web06\_Cookie 目录中，2 放在 web06\_Cookie/app01 目录中，3 放在 web06\_Cookie/app01/sub 目录中

```
<% Cookie[] cookies=request.getCookies();
if(cookies!=null){ for(int i=0;i<cookies.length;i++){
 Cookie c=cookies[i];
 out.println(c.getName()+" "+c.getValue()+"
"); } } %>
```

step3: 分别执行各个 jsp

```
web06_Cookie/findCookie1.jsp //是不能访问，不是 no cookie 提示(页面内容为空白。
忽略 JSESSIONID)
web06_Cookie/app01/findCookie2.jsp //可以访问
web06_Cookie/app01/sub/findCookie3.jsp //可以访问
```

5) 修改 cookie 的默认路径

可以调用 cookie.setPath(String path)方法，一般我们会设置 cookie.setPath("/appname")，这样，应用内部的某个组件所添加的 cookie 可以被该应用内部的其他组件访问到。

例如：上例修改如下，则三种地址都可访问。

```
<% Cookie c=new Cookie("userid","abc"); c.setPath("/web06_Cookie");
response.addCookie(c); %>
```

## 13.9 cookie 的限制

- 1) cookie 可以被用户禁止。
- 2) 因为 cookie 保存在浏览器端，所以 cookie 不安全。对于一些敏感的数据，需要加密处理。
- 3) cookie 只能够保存少量的数据（大约是 4K 左右）。
- 4) cookie 的个数也有限制（浏览器只能保存大约 300 个左右的 cookie，另外对于某个服务器也有 cookie 个数的限制，大约是 20 个）。
- 5) cookie 只能够保存字符串，对象、集合、数组都不能保存。

## 13.10 案例：写一个 CookieUtil

step1: 创建 CookieUtil 类

```
//appname 是应用名，在使用该工具类时，将应用名改为实际部署时的应用名
private static String appname="/web06_Cookie";
//添加一个 cookie 需要考虑编码问题
public static void addCookie(String name,String value
,HttpServletResponse response,int age) throws UnsupportedEncodingException{
Cookie c=new Cookie(name,URLEncoder.encode(value,"utf-8"));
c.setMaxAge(age); c.setPath(appname); response.addCookie(c); }
```

```

//依据 cookie 名称返回 cookie 的值，找不到则返回 null
public static String findCookie(String name,HttpServletRequest request){
 String value=null; Cookie[] cooikes=request.getCookies();
 if(cooikes!=null){ for(int i=0;i<cooikes.length;i++){
 Cookie c=cooikes[i];
 if(c.getName().equals(name)){ value=c.getValue(); } }
 }
 return value;
}
//删除一个 cookie
public static void deleteCookie(String name,HttpServletResponse response){
 Cookie c=new Cookie(name,"");
 c.setMaxAge(0); response.addCookie(c);
}

```

step2：测试，创建 FindCookieServlet 类，其中的 service 方法为：

```

response.setContentType("text/html"); PrintWriter out = response.getWriter();
String value=CookieUtil.findCookie("name", request);
if(value==null){ CookieUtil.addCookie("name", "zs", response, 3600);
} else{ out.println(value); } out.close();

```

## 13.11 session（会话）

什么是 session：①是服务器端的状态管理技术。②当浏览器访问服务器时，服务器会创建一个 session 对象（该对象有一个唯一的 id 号，称之为 sessionId）。接下来，服务器在默认情况下，会使用 set-cookie 消息头将这个 sessionId 发送浏览器，浏览器会将这个 sessionId 保存下来（内存中，因为指定生存时间）。当浏览器再次访问服务器时，会将 sessionId 使用 cookie 消息头发送给服务器，服务器依据这个 sessionId 就可以找到之前创建的 session 对象。③用户与服务器之间的多次交互叫一次会话。

## 13.12 如何创建一个 session 对象

1) 方式一： HttpSession s=request.getSession(boolean flag); // HttpSession 是一个接口，返回一个符合要求的对象（工厂）。

当 flag=true 时：

服务器会先检查请求当中是否有 sessionId，如果没有则创建一个 session 对象。

如果有 sessionId，则服务器会依据 sessionId 查找对应的 session 对象，如果找到了，则返回 session 对象。根据 sessionId 找不到，则服务器会创建一个新的 session 对象。

当 flag=false 时：

服务器会先检查请求当中是否有 sessionId，如果没有则返回 null。

如果有 sessionId，则服务器会依据 sessionId 查找对应的 session 对象，如果找到了，则返回 session 对象。根据 sessionId 找不到，则服务器返回 null。

2) 方式二： HttpSession s=request.getSession();

等价于 HttpSession s=request.getSession(true);

## 13.13 HttpSession 接口中提供的常用方法

- 1) String getId(): 获得 sessionId
- 2) setAttribute(String name, Object obj): 绑定一个对象到 session 对象上
- 3) getAttribute(String name): 获得绑定对象，如果不存在则返回 null
- 4) removeAttribute(String name): 解除绑定（request 也有）

## 13.14 session 的超时

1) 服务器会将空闲时间过长的 session 对象删除。大部分服务器默认的 session 超时限制是 30 分钟。

2) 修改超时限制：可以修改这个默认的超时限制。

例如：可以修改 Tomcat 的 web.xml（影响 Tomcat 服务器下的所有应用），修改之后，需要重新启动 Tomcat 服务器。也可以修改某个应用的 web.xml。

```
<session-config>
 <session-timeout>30</session-timeout>
</session-config>
```

也可以通过编程的方式来修改超时的限制：setMaxInactiveInterval(int seconds);

3) 立即删除 session： invalidate(); // 实际是清空 session 对象的内容，然后可以给其他人继续使用该 session 对象

## 13.15 用户禁止 cookie 以后，如何继续使用 session

1) 如果用户禁止 cookie，服务器仍然会将 sessionId 以 cookie 的方式发送给浏览器，但是，浏览器不再保存这个 cookie（即 sessionId）。

2) 如果想要继续使用 session，需要采取其他方式来实现 sessionId 的跟踪，如可以使用 url 重写来实现 sessionId 的跟踪。

## 13.16 url 重写

1) 什么是 url 重写

浏览器在访问服务器上的某个地址时，不能够直接写这个组件（Servlet/JSP）的地址，而应该使用服务器生成的包含有 sessionId 的地址。比如以 13.19 案例中访问次数为例：

当一个新页面有个链接时 <a href="SomeServlet">someServlet</a> 是错误的！点刷新是不会计数的，因为 cookie 被禁止了。

应该是 <a href="<% = response.encodeURL("SomeServlet") %>">someServlet</a> 此时去点击刷新 count 会自动加 1 的。

◆ 注意事项：当用户禁止 cookie 时，encodeURL 方法会在“SomeServlet”后面添加 sessionId（不禁止不会有）。例如：

`http://localhost:8080/web08/SomeServlet;jsessionid=CD20CBCCC38B4AA3137084D9966D2874`

2) 如何进行 url 重写

① `response.encodeURL(String url);` // encodeURL 方法用在链接地址、表单提交地址。

例如： <form action="<% = response.encodeURL("SomeServlet") %>">

◆ 注意事项：还有一个 encodeUrl() 是个老方法，内部实现和 encodeURL 一模一样！

② `response.encodeRedirectURL(String url);` // 用于重定向

例如： `response.sendRedirect(response.encodeRedirectURL("list.do"));`

③ 转发不用考虑！是服务器内部，不是浏览器访问服务器。

## 13.17 session 的优点

1) session 比较安全（相对于 cookie）因为将所有状态都写在服务器端。

2) session 能够保存的数据类型更加丰富（cookie 只能保存字符串）。

3) session 能够保存的数据大小更大（cookie 只能保存大约 4K 左右的数据），理论上

session 是没有限制的。

4) session 没有被禁止的问题，也就不需要 url 重写（cookie 可以被用户禁止）。

### 13.18 session 的缺点

1) session 会将数据放在服务器端，所以，对服务器的资源的占用比较大，而 cookie 会将数据保存在浏览器端，对服务器资源的占用没有。

可以将 session 存到数据库或文件中（钝化），采取最近最少使用原则。当再次使用 session 时，从数据库或文件中激活。

2) session 默认情况下，会将 sessionId 以 cookie 的方式发送给浏览器，浏览器会将 session 保存到内存中，如果浏览器关闭，浏览器发请求时就没有 sessionId，服务器端的 session 对象就找不到了。

### 13.19 案例： session 验证和访问次数

1) session 验证： session 验证经常用于保护一些需要登录之后才能访问的资源。比如：只有登录成功以后，才能访问 main.jsp

step1：登录成功以后，绑定一些数据到 session 对象上。

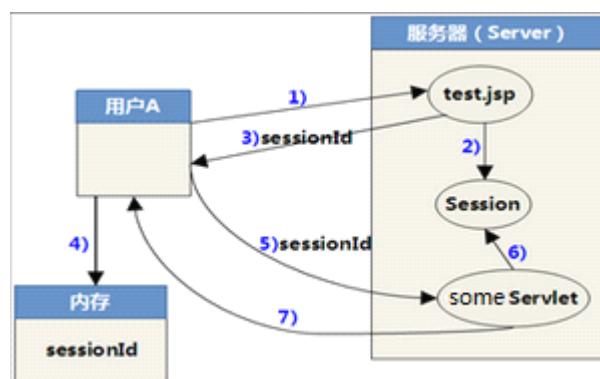
例如： session.setAttribute("user", user);

step2：对于需要保护的资源，添加 session 验证的代码。例如：为 main.jsp 设置 session 验证

```
<% Object obj=session.getAttribute("user");
 if(obj==null){
 //没有登录成功，或者因为 session 超时，服务器已经删除了之前的 session 对象
 response.sendRedirect("login.jsp");
 return;
 }
%>
```

◆ 注意事项：若不写 return，则该页面中 body 标签中的内容，虽然看不到，但还是会执行！

2) 访问次数



step1： SomeServlet 中 service 方法

```
response.setContentType("text/html;charset=utf-8"); PrintWriter out
response.getWriter();
HttpSession session=request.getSession(); //获得一个 session 对象
//设置超时限制，注意：若不到 10 秒刷新则又重新开始计时
//session.setMaxInactiveInterval(10);
```

```

String sessionId=session.getId();//获得 sessionId
System.out.println("sessionId: "+sessionId);
Integer count=(Integer)session.getAttribute("count");//绑定必须是个对象，所以用包装类
if(count==null){ count=1;//是第一次访问，则设置 count=1
}else{ count++; //不是第一次访问，则原有值基础上加 1 }
session.setAttribute("count", count);
out.println("<h1>你是第: "+count+" 次访问</h1>");
//session.invalidate(); //立即删除 session 对象
out.close();

```

step2: web.xml 配置

```

<servlet>
 <servlet-name>SomeServlet</servlet-name>
 <servlet-class>web.SomeServlet</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>SomeServlet</servlet-name>
 <url-pattern>/SomeServlet</url-pattern>
</servlet-mapping>

```

## 13.20 案例：验证码

step1: CheckcodeServlet 中 service 方法：生成验证码图片

```

/** step1: 绘图 */
//1 创建一个内存映像对象（类似一个画布）
BufferedImage image=
 new BufferedImage(WIDTH,HEIGHT,BufferedImage.TYPE_INT_RGB);
Graphics g=image.getGraphics(); //2 获得画布
Random r=new Random(); //3 给笔上色
g.setColor(new Color(r.nextInt(255),r.nextInt(255),r.nextInt(255)));
g.fillRect(0,0,WIDTH,HEIGHT); //4 给画布设置一个背景颜色
String number=getNumber(5); //5 绘图
//或 String number=r.nextInt(99999)+""; //作用是把整数转成字符串
//6 绑定 number 到 session 对象上
HttpSession session=request.getSession(); session.setAttribute("number", number);
g.setColor(new Color(0,0,0));
//7 字体类型、风格、高度
g.setFont(new Font(null,Font.BOLD|Font.ITALIC,18)); //null 表示缺省字体
g.drawString(number,10,20); //第一个字符的左下角的点离左的距离和字高度
//8 加一些干扰线
for(int i=0;i<5;i++){ g.drawLine(r.nextInt(80),r.nextInt(30), r.nextInt(80),r.nextInt(30)); }
/** step2: 将图片压缩，然后输出到客户端（浏览器） */
/** 设置 content-type 消息头，告诉浏览器返回的数据是一张 jpeg 格式的图片 */
response.setContentType("image/jpeg");
OutputStream os=response.getOutputStream(); //获得字节输出流（要输出的是图片）
/** 将原始图片 image 按照指定的压缩格式 jpeg 进行压缩，然后，将压缩之后生成的数

```

据通过 os 输出到 response 对象上，服务器会从 response 对象上取出数据，打包发送给浏览器 \*/

```
javax.imageio.ImageIO.write(image, "jpeg", os);
```

step2: CheckcodeServlet 中 getNumber 方法

```
private String getNumber(int size) {
 String str="ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789";
 String rs=""; Random r=new Random();
 for(int i=0;i<size;i++){ rs +=str.charAt(r.nextInt(str.length())); }
 return rs;
}
```

step3: ActionServlet 中 service 方法：对登录的判断

```
request.setCharacterEncoding("utf-8");
response.setContentType("text/html;charset=utf-8"); PrintWriter out=response.getWriter();
String uri=request.getRequestURI();
String action=uri.substring(uri.lastIndexOf("/"),uri.lastIndexOf("."));
if(action.equals("/login")){//先看验证码是否正确，比较两个验证码
String number1=request.getParameter("number");//number1：表单提交时用户填写的验证码
HttpSession session=request.getSession();//number2：获得绑定的 number
String number2=(String)session.getAttribute("number");
if(!number1.equalsIgnoreCase(number2)){//如果验证码错误，则提示用户验证码错误
 request.setAttribute("number_error","验证码错误");
 request.getRequestDispatcher("login.jsp").forward(request, response);
 return;//没有必要往下执行了
}
String username=request.getParameter("username");String
pwd=request.getParameter("pwd");
UserDAO dao=new UserDAO();
try { User user=dao.findByUsername(username);
 //登录成功，跳转到一个只有登录成功后才能访问的页面
 if(user!=null && user.getPwd().equalsIgnoreCase(pwd)){
 //添加 session 验证的代码：先绑定一些数据到 session 对象上，用于 session 验证
 session.setAttribute("user", user); response.sendRedirect("main.jsp");
 }else{ request.setAttribute("login_error", "用户名或密码错误！");
 request.getRequestDispatcher("login.jsp").forward(request, response); }
} catch (Exception e) { e.printStackTrace(); throw new ServletException(e); }
```

step4: web.xml 配置

```
<servlet>
 <servlet-name>CheckcodeServlet</servlet-name>
 <servlet-class>web.CheckcodeServlet</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>CheckcodeServlet</servlet-name>
 <url-pattern>/checkcode</url-pattern>
</servlet-mapping>
```

step5: login.jsp 页面

```

<tr><td valign="middle" align="right">验证码: </td>
 <td valign="middle" align="left">
 <input type="text" name="number" />
 <% String msg2=(String)request.getAttribute("number_error"); %>
 <%=(msg2==null?" ":msg2)%></td></tr>
<tr><td></td><td valign="middle" align="left">

 <a href="javascript:;" onclick="document.getElementById('img1').src=
 'checkcode?' + Math.random();">看不清, 换一个
</td></tr>

```

- ◆ 注意事项: Math.random();获得随机数,(new Date()).getTime();获得毫秒数,加在问号“?”后面都可以起到欺骗服务器作用,以达到刷新验证码的作用!

## 13.21 案例: 购物车

step1: 建表

```

create table chang_product(
 id bigint primary key auto_increment, model varchar(20), pic varchar(50),
 prodDesc text, price double);
insert into chang_product(model,pic,prodDesc,price) values('x200','x200.jpg','还算便宜',2000);
insert into chang_product(model,pic,prodDesc,price) values('x500','x500.jpg','性价比最好',4000);
insert into chang_product(model,pic,prodDesc,price) values('x600','x600.jpg','性能不错',6000);

```

step2: Product 实体类

```

private int id; private String model; private String pic; private String
prodDesc;
private double price;//相应的 get/set 方法

```

step3: DBUtil 工具类

```

public static Connection getConnection() throws Exception{
 Connection conn=null;
 try { Class.forName("com.mysql.jdbc.Driver");
 conn=DriverManager.getConnection("jdbc:mysql://localhost:3306/db_chang?" +
 "useUnicode=true&characterEncoding=utf8","root","");
 } catch (Exception e) { e.printStackTrace(); throw e; }
 return conn;
}
public static void close(Connection conn){
 if(conn!=null){ try { conn.close(); } catch (SQLException e) {
 e.printStackTrace(); } }
}

```

step4: ProductDAO 类

```

public List<Product> findAll() throws Exception{
 List<Product> products=new ArrayList<Product>(); Connection conn=null;
 try { conn=DBUtil.getConnection(); Statement stat=conn.createStatement();
 ResultSet rs=stat.executeQuery("select * from chang_product");
 while(rs.next()){ Product p=new Product(); p.setId(rs.getInt("id"));
 p.setModel(rs.getString("model")); p.setPic(rs.getString("pic"));
 }
 }
}

```

```

 p.setProdDesc(rs.getString("prodDesc")); p.setPrice(rs.getDouble("price"));
 products.add(p); }
 } catch (Exception e) { e.printStackTrace();
 } finally{ DBUtil.close(conn); } return products;
}

public Product findById(int id) { Product p=null; Connection conn=null;
try { conn=DBUtil.getConnection(); Statement stat=conn.createStatement();
ResultSet rs=stat.executeQuery("select * from chang_product where id="+id);
while(rs.next()){ p=new Product(); p.setId(rs.getInt("id"));
p.setModel(rs.getString("model")); p.setPic(rs.getString("pic"));
p.setProdDesc(rs.getString("prodDesc")); p.setPrice(rs.getDouble("price")); }
} catch (Exception e) { e.printStackTrace();
} finally{ DBUtil.close(conn); } return p;
}

```

step5：新建 CartItem.java 商品条目类，作用是为了方便实现 Cart 类的操作而设计的。

```
private Product p; private int qty;//get/set 方法
```

step6：新建 Cart.java 类，原因：用户发送请求（buy）购买，服务器创建 Session 对象，不需要再直接访问 Session 的方法，而是 Session 绑定一个 Cart 对象，然后用户购买商品时，只调用 Cart 的 add 方法进行操作，而不再直接操作 Session，删除（修改）时也一样。如此，我们以后只要操作 Cart 类的方法就 OK 了，不再直接操作 Session。这是一个经典的设计。

```

//items: 用来保存用户所购买的所有商品
private List<CartItem> items=new ArrayList<CartItem>();

/** 添加商品：需要对 items 进行遍历，如果已经购买过该商品，则不再添加并且返回 false，否则，将商品添加到 items 然后返回 true */
public boolean add(CartItem item) {
 for(int i=0;i<items.size();i++){
 CartItem curr=items.get(i);
 if(curr.getP().getId()==item.getP().getId()){//已经购买过
 return false;
 }
 items.add(item);//没有买过
 return true;
 }
}

/** 计价 */
public double cost(){
 double total=0;
 for(int i=0;i<items.size();i++){
 CartItem curr=items.get(i);
 total +=curr.getQty()*curr.getP().getPrice();
 }
 return total;
}

/** 返回用户所购买的所有商品 */
public List<CartItem> list() { return items; }

/** 清空购物车 */
public void clear() { if(items.size()>0){ items.clear(); } }

/** 删除购物车中某项 */
public void delete(int id){
 for(int i=0;i<items.size();i++){
 CartItem curr=items.get(i);
 if(curr.getP().getId()==id){ items.remove(curr); }
 }
}

/** 修改商品数量 */
public void modify(int id,int qty){
 for(int i=0;i<items.size();i++){
 CartItem curr=items.get(i);
 if(curr.getP().getId()==id){ curr.setQty(qty); }
 }
}

```

step7: ActionServlet 类中的 service 方法

```
String uri = request.getRequestURI();
String action = uri.substring(uri.lastIndexOf("/"),uri.lastIndexOf("."));
if(action.equals("/list")){
 ProductDAO dao = new ProductDAO();
 try { List<Product> products = dao.findAll();
 request.setAttribute("products", products);
 request.getRequestDispatcher("productList.jsp").forward(request, response);
 } catch (Exception e) { e.printStackTrace(); throw new ServletException(e); }
} else if(action.equals("/buy")){
 HttpSession session=request.getSession();
 Cart cart=(Cart)session.getAttribute("cart");
 if(cart==null){ cart=new Cart(); session.setAttribute("cart", cart); }//第一次购买
 int id=Integer.parseInt(request.getParameter("id"));
 ProductDAO dao=new ProductDAO();
 try { Product p=dao.findById(id);//将 Product 封装成一个 CartItem 对象
 CartItem item=new CartItem(); item.setP(p); item.setQty(1);
 boolean flag=cart.add(item);
 if(flag){//如果返回值为 true, 则提示用户购买成功
 request.setAttribute("buy_msg_"+id, "购买成功");
 } else{ request.setAttribute("buy_msg_"+id, "已经购买过"); }
 request.getRequestDispatcher("list.do").forward(request, response);
 } catch (Exception e) { e.printStackTrace(); throw new ServletException(e); }
} else if(action.equals("/delete")){
 HttpSession session=request.getSession(); Cart cart=(Cart)session.getAttribute("cart");
 cart.delete(id); response.sendRedirect("cart.jsp");
} else if(action.equals("/clear")){
 HttpSession session=request.getSession();
 Cart cart=(Cart)session.getAttribute("cart"); cart.clear();
 response.sendRedirect("cart.jsp");
} else if(action.equals("/modify")){
 int id=Integer.parseInt(request.getParameter("id"));
 int qty=Integer.parseInt(request.getParameter("qty"));
 HttpSession session=request.getSession(); Cart cart=(Cart)session.getAttribute("cart");
 cart.modify(id,qty); response.sendRedirect("cart.jsp");
}
```

step8: productList.jsp 页面动态部分

```
<% List<Product> products = (List<Product>)request.getAttribute("products");
for(int i=0;i<products.size();i++){ Product p = products.get(i); %>
<tr> <td class="altbg2"> <%=p.getModel()%></td>
<td class="altbg2"></td>
<td class="altbg2"><%=p.getProdDesc()%></td>
<td class="altbg2">¥<%=p.getPrice()%></td>
<td class="altbg2">
<% String msg=(String)request.getAttribute("buy_msg_"+p.getId()); %>
<a href="buy.do?id=<%=p.getId()%>">购买
<%=(msg==null?"":msg)%></td></tr>
<% } %>
```

```

<center>
 <input class="button" type="button" value="查看购物车" name="settingsubmit"
 onclick="location = 'cart.jsp';">
</center>

```

step9: cart.jsp 页面动态部分

```

<% //从 session 当中，取出 cart 对象
 Cart cart=(Cart)session.getAttribute("cart");
 if(cart!=null&&cart.list().size()>0){ List<CartItem> items=cart.list();
 for(int i=0;i<items.size();i++){
 CartItem item=items.get(i);
 }
 %>
<tr> <td class="altbg2"><%=item.getP().getModel()%></td>
 <td class="altbg2"><%=item.getP().getPrice()%></td>
 <td class="altbg2"><%=item.getQty()%></td>
 <td class="altbg2"><input type="text" size="3" value="" id="num_<%=item.getP().getId()%>" /></td><!-- 动态生成 id -->
 <td class="altbg2">
 <a href="javascript:;" onclick="location='modify.do? id=<%=item.getP().getId()%>&qty='+document.getElementById('num_<%=item.getP().getId()%>').value;">
 更改数量</td><!-- 注意！ location 中的地址不能用回车换行！ -->
 <td class="altbg2"><a href="delete.do?id=<%=item.getP().getId()%>">删除</td>
 </td>
</tr>
<% } %>
<tr><td class="altbg1" colspan="6">总价格：￥<%=cart.cost()%></td></tr>
<%
 }else{
%>
<tr><td class="altbg2" colspan="6">还没有选购商品</td></tr>
<%
 }
%>
<center>
 <input class="button" type="button" value="返回商品列表" name="settingsubmit"
 onclick="location = 'list.do';">
 <input class="button" type="button" value="清空购物车" name="settingsubmit"
 onclick="location = 'clear.do';">
</center>

```

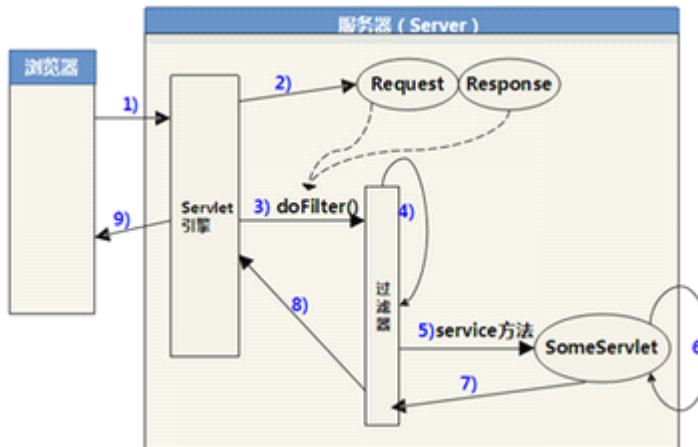
step10: 测试，输入 <http://localhost:8080/shoppingcart/list.do>

# 九十七、过滤器

LICHOO

## 14.1 什么是过滤器

Servlet 规范当中定义的一种特殊的组件，可以拦截 Servlet 容器的调用过程并进行相应的处理。某个过滤器只有一个实例，即单例模式。



## 14.2 如何写一个过滤器

step1：写一个 Java 类，实现 Filter 接口

step2：在 doFilter 方法里面，编写处理逻辑

step3：配置过滤器

◆ 注意事项：Filter 的 xml 配置在 Servlet 的 xml 配置前！

## 14.3 案例：敏感字过滤器

step1：创建 comment.jsp 页面

```
<form action="comment" method="post">
 请输入你的评论: <input name="content" /><input type="submit" value="发表"/>
</form>
```

step2：创建 CommentFilter1 过滤器

```
public class CommentFilter1 implements Filter{//实现 Filter 接口
 private FilterConfig config;
 public CommentFilter1(){ System.out.println("Filter1's constructor..."); }
 /** 容器在销毁过滤器之前，会调用 destroy 方法。该方法只会执行一次。 */
 public void destroy(){ System.out.println("Filter1's destroy ..."); }
 /** 容器会调用 doFilter 方法来处理请求，类似于容器调用 Servlet 的 service 方法一样。
 * 如果调用了 FilterChain(过滤器链)的 doFilter 方法，容器会继续调用后续的过滤器或者
 * Servlet，如果没有调用 FilterChain 的 doFilter 方法，则容器不再向后调用。 */
 public void doFilter(ServletRequest arg0,ServletResponse arg1,
 FilterChain arg2) throws IOException, ServletException {
 System.out.println("Filter1's doFilter begin...");
 HttpServletRequest request =
 (HttpServletRequest)arg0;//强转为子接口，是为了调用更多方法
 HttpServletResponse response =(HttpServletResponse)arg1;
```

```

request.setCharacterEncoding("utf-8");
String content = request.getParameter("content");
response.setContentType("text/html;charset=utf-8");
PrintWriter out = response.getWriter();
//通过 FilterConfig 对象读取初始化参数
String illegalStr = config.getInitParameter("illegalStr");//初始化参数详见 14.5
if(content.indexOf(illegalStr) != -1){//有敏感字，提示用户评论当中有敏感字
 out.println("<h1>评论当中有敏感字</h1>");
} else{ arg2.doFilter(arg0, arg1); } //无敏感字，继续向后调用
System.out.println("Filter1's doFilter end.");
/** 容器在启动时，会创建过滤器实例，接下来，会立即调用 init 方法完成初始化操作。
该方法只会执行一次。容器会事先创建好一个符合 FilterConfig 接口要求的对象。可以通过
FilterConfig.getInitParameter 方法来访问过滤器的初始化参数。 */
public void init(FilterConfig arg0) throws ServletException {
 System.out.println("Filter1's init...");
/** 因为 init 方法只执行一次，方法执行完之后，arg0 变量就会被销毁，所以需要将 arg0
的值赋给 config 属性，这样，就可以在 doFilter 方法里面访问到 FilterConfig 对象了。 */
 config = arg0;
}
}

```

step3：配置 web.xml 文件

```

<filter>
 <filter-name>filter1</filter-name>
 <filter-class>web.CommentFilter1</filter-class>
 <init-param><!-- 过滤器的初始化参数 详见 14.5-->
 <param-name>illegalStr</param-name>
 <param-value>cat</param-value>
 </init-param>
</filter>
<filter-mapping>
 <filter-name>filter1</filter-name>
 <url-pattern>/comment</url-pattern>
</filter-mapping>
<servlet>
 <servlet-name>CommentServlet</servlet-name>
 <servlet-class>web.CommentServlet</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>CommentServlet</servlet-name>
 <url-pattern>/comment</url-pattern>
</servlet-mapping>

```

## 14.4 过滤器的优先级

如果有多个过滤器都满足过滤的条件，则容器会依据<filter-mapping>的先后顺序来调用过滤器。比如：

step1：14.3 案例中再添加一个 CommentFilter2 过滤器，用于判断长度，部分代码修改如下

```
int length =Integer.parseInt(config.getInitParameter("length")); //初始化参数详见 14.5
if(content.length() > length){
 out.println("<h1>评论长度不能超过"+length+"个字符</h1>");
} else{ arg2.doFilter(arg0, arg1); } //长度符合要求，继续向后调用
```

step2: web.xml 配置

```
<filter>
 <filter-name>filter2</filter-name>
 <filter-class>web.CommentFilter2</filter-class>
 <init-param><!-- 初始化参数详见 14.5 -->
 <param-name>length</param-name>
 <param-value>20</param-value>
 </init-param>
</filter>
<filter-mapping>
 <filter-name>filter2</filter-name>
 <url-pattern>/comment</url-pattern>
</filter-mapping>
```

## 14.5 初始化参数

step1：使用<init-param>配置初始化参数

step2：调用 FilterConfig.getInitParameter(String parameter);返回一个字符串

## 14.6 优点

- 1) 可以实现代码的“可插拔性”（增加或减少某个模块，不会影响程序的正常运行）。
- 2) 可以将多个组件相同的处理逻辑集中写在过滤器里面，方便代码的维护。如：四个 Web 组件都需要 session 验证，把验证写在一个过滤器即可。配置中写“\*”，拦截所有请求。

# 九十八、监听器

LICHOO

## 15.1 什么是监听器

Servlet 规范当中定义的一种特殊的组件，用来监听容器产生的事件并进行处理。

## 15.2 容器会产生两大类事件

1) 生命周期相关的事件：容器在创建或者销毁 Request 对象、Session 对象、ServletContext 对象（Servlet 上下文）时产生的事件。

2) 绑定相关的事件：容器调用了 Request 对象、Session 对象、ServletContext 的 setAttribute、removeAttribute 时产生的事件。

## 15.3 如何写一个监听器

step1：写一个 Java 类，实现监听器接口（依据监听的事件类型来选择对应的接口，8 种，如继承 HttpSessionListener）

step2：在监听器接口定义的方法里面，编写处理逻辑

step3：配置监听器

## 15.4 ServletContext（Servlet 上下文）

容器在启动的时候，会为每一个应用创建唯一的一个符合 ServletContext 接口要求的对象（Servlet 上下文），该对象一直存在，只有非容器关闭时对象被销毁。

## 15.5 如何获得 Servlet 上下文

1) GenericServlet.getServletContext(); //通过 GenericServlet 抽象类获得上下文（其实也是调用了 ServletConfig 的 getServletContext 方法）

2) HttpSession.getServletContext(); //通过 Session 获得上下文

3) ServletConfig.getServletContext(); //通过 ServletConfig 接口获得上下文

4) FilterConfig.getServletContext(); //通过过滤器获得上下文

## 15.6 Servlet 上下文的作用

1) 绑定数据：setAttribute、removeAttribute、getAttribute

Request、Session、ServletContext 都提供了绑定数据的相关的三个方法，如果都满足使用的条件，应该优先使用生命周期短的（Request<Session<ServletContext）。

① Request 对象上绑定的数据只有同一个请求所涉及的各个 Web 组件可以共享，例如：一个 Servlet 将数据绑定到 Request，然后转发到一个 jsp。请求先交给过滤器来处理，然后调用 Servlet。

② Session 对象上绑定的数据是用一个会话所涉及的各个 Web 组件可以共享。

③ ServletContext 对象绑定的数据是公开的，谁都可以访问，而且随时可访问。

2) 访问全局的初始化参数

即使用 <context-param> 配置的初始化参数，调用 String getInitParameter(String paramName) 方法，可以被同一个应用中的所有的 Servlet、Filter 共享。例如：

```
<context-param><!-- 全局初始化参数，放在 servlet 前面 -->
 <param-name>company</param-name>
 <param-value>chang 全局</param-value>
```

```

</context-param>
<servlet><!-- 局部初始化参数，放在某个 servlet 里面 -->
 <servlet-name>A</servlet-name>
 <servlet-class>web.A</servlet-class>
 <init-param>
 <param-name>company</param-name>
 <param-value>chang 局部</param-value>
 </init-param>
</servlet>

```

3) 依据逻辑路径 (path) 获得实际部署时的物理路径

```
String getRealPath(String path); //详情见 16.2 案例 step3
```

## 15.7 案例：统计在线人数

step1: index.jsp 页面

```

在线人数: <%=application.getAttribute("count") %>

退出系统

```

step2: 创建 CountListener 监听类，并实现 HttpSessionListener 接口

```

public class CountListener implements HttpSessionListener {
 private int count=0;//计数器
 /** session 对象创建之后，容器会产生 HttpSessionEvent 事件，然后调用 sessionCreated 方法 */
 public void sessionCreated(HttpSessionEvent arg0) {
 System.out.println("sessionCreated...");
 //通过事件对象（HttpSessionEvent）找到 session
 HttpSession session=arg0.getSession();
 ServletContext sctx=session.getServletContext();
 //将人数绑定到 servletContext，这样可以随时访问
 count++; sctx.setAttribute("count", count);
 }
 /** 容器在销毁 session 对象时候，会调用 sessionDestroyed 方法 */
 public void sessionDestroyed(HttpSessionEvent arg0) {
 System.out.println("sessionDestroyed...");
 HttpSession session=arg0.getSession();
 ServletContext sctx=session.getServletContext();
 count--; sctx.setAttribute("count", count);
 }
}

```

step3: logout.jsp 页面

```
<%session.invalidate();%>
```

step4: web.xml 配置

```

<listener>
 <listener-class>web.CountListener</listener-class><!-- 直接写包名.类名即可 -->
</listener>
<welcome-file-list>
 <welcome-file>index.jsp</welcome-file>
</welcome-file-list>

```

◆ 注意事项：

- ❖ 因为多个火狐浏览器在一台计算机共享一块内存，所以不论启动多少个窗口都显示是一个用户。同时使用火狐浏览器和其它浏览器打开，因为使用的不是同一块内存，所以用户数会增加。
- ❖ web.xml 配置文件中配置信息的顺序如下：

```
<!-- 全局配置参数 -->
<!-- 过滤器 -->
<!-- 监听器 -->
<!-- servlet -->
<!-- 错误配置页面 -->
<!-- 首页 http://ip:port/appname 访问时写到应用名就能访问了-->
```

# 九十九、上传文件

LICHOO

## 16.1 如何上传文件

step1：给表单设置 `enctype="multipart/form-data"` (http 协议的要求)，并且，表单只能使用 `post` 方式来提交。

step2：在服务器端，不能使用 `request.getParameter` 方法来获得参数值。此时，需要调用 `request.getInputStream` 获得一个 `InputStream` (字节流)，然后分析这个流来获得数据。一般，使用一些工具来分析这个流（比如：apache 提供的 `commons-fileupload-1.2.1.jar`）

## 16.2 案例：上传文件

step1：将 `commons-fileupload-1.2.1.jar` 和 `commons-io-1.4.jar` 包导入

step2：创建 `form.jsp` 页面

```
<form action="fileupload" method="post" enctype="multipart/form-data">
 <fieldset><legend>上传文件</legend>
 用户名：<input name="username" />
照片：<input type="file" name="file1"/>
 <input type="submit" value="确定" /></fieldset>
</form>
```

step3：`FileUploadServlet` 类中 `service` 方法

```
/** step1 创建一个 DiskFileItemFactory 工厂类对象，该对象为解析器提供了解析时的缺省配置 */
DiskFileItemFactory dfif = new DiskFileItemFactory();
ServletFileUpload sfu = new ServletFileUpload(dfif);/** step2 创建解析器 */
/** step3 使用解析器来解析，解析器会调用 request.getInputStream 获得一个流，然后分析这个流，并且将分析的结果封装到 FileItem 对象里面。一个 FileItem 对象封装了一个表单域中的所有数据。 */
try { List<FileItem> items = sfu.parseRequest(request);
 //只需要遍历 items 集合就可以访问表单中的每一个表单域的数据。
 for(int i=0;i<items.size();i++){
 FileItem curr = items.get(i);
 if(curr.isFormField()){//如果是普通表单域
 String fieldname = curr.getFieldName();
 System.out.println("fieldname:" + fieldname);
 String username = curr.getString();
 System.out.println("username:" + username);
 }else{//如果是上传文件域
 ServletContext sctx = getServletContext();
 //依据逻辑路径获得实际部署时的物理路径
 String path = sctx.getRealPath("upload");
 System.out.println("path:" + path);
 String filename = curr.getName();//获得文件名
 File file = new File(path + "/" + filename);
 curr.write(file);//注意，windows 用"\\"，Linux 用“//” }
 }
 } catch (Exception e) { e.printStackTrace(); }
}
```

step4: web.xml 配置

```
<servlet>
 <servlet-name>FileUploadServlet</servlet-name>
 <servlet-class>web.FileUploadServlet</servlet-class>
</servlet>
<servlet-mapping>
 <servlet-name>FileUploadServlet</servlet-name>
 <url-pattern>/fileupload</url-pattern>
</servlet-mapping>
```

# 一百、Servlet 线程安全问题

LICHOO

可使用 jmeter 压力测试工具。

## 17.1 为何 Servlet 会有线程安全问题

当容器收到一个请求之后，会启动一个线程来处理该请求，如果有多个请求到达容器，并且这多个请求要访问的是同一个 Servlet，则会发生多个线程调用同一个 Servlet 的情况，此时，就需要考虑线程安全问题了，比如，这多个线程都要修改 Servlet 的属性值。

## 17.2 如何处理线程安全问题

- 1) 加锁：使用 synchronized 关键字对方法或者代码块加锁。加锁会影响一些性能
- 2) 让一个 Servlet 实现 SingleThreadMode 接口：容器会为这样的 Servlet（实现 SingleThreadMode 接口的）创建多个实例（一个线程一个实例）。因为有可能会产生过多的 Servlet 实例，所以，在比较大型的应用当中，尽量少用。

# 一百〇一、Servlet 小结

LICHOO

## 18.1 Servlet 基础

- 1) 什么是 Servlet
- 2) 如何写一个 Servlet
- 3) Servlet 容器是什么
- 4) Servlet 依不依赖某个特定的 Servlet 容器? (不依赖)
- 5) HTTP 协议
- 6) get 请求和 post 请求
- 7) 表单的处理: ①如何获得表单中的参数值 ②中文参数值如何获得
- 8) Servlet 如何输出中文

## 18.2 Servlet 核心

- 1) 重定向和转发
- 2) 如何获得请求参数值
- 3) 容器如何处理请求资源路径
- 4) Servlet 的生命周期
- 5) 线程安全问题
- 6) Servlet 上下文

## 18.3 状态管理

- 1) 什么是状态管理
- 2) Cookie:
  - ①Cookie 是什么 ②如何创建一个 Cookie ③Cookie 的编码问题
  - ④Cookie 的生成时间 ⑤Cookie 的路径问题 ⑥Cookie 的限制
- 3) Session:
  - ①Session 是什么 ②如何获得一个 Session ③Session 的常用方法
  - ④Session 的超时 ⑤删除 Session
  - ⑥如果用户禁止 Cookie, 如何继续使用 Session
  - ⑦Session 的优缺点 (和 Cookie 比较)

## 18.4 数据库访问

- 1) 如何使用 JDBC 访问数据库
- 2) DAO

## 18.5 过滤器和监听器

- 1) 过滤器:
  - ①什么是过滤器 ②如何写一个过滤器 ③过滤器的优先级
  - ④初始化参数 ⑤优点
- 2) 监听器:
  - ①什么是监听器 ②如何写一个监听器

## 18.6 典型案例和扩展

- 1) 员工管理
- 2) Session 验证
- 3) 验证码
- 4) 购物车
- 5) 上传文件

## 一百〇二、其他注意事项

LICHOO

### 19.1 连接数据库的工具

- 1) toad: 3 星
- 2) sql-front: 2 星
- 3) eclipse 自带的

### 19.2 知名网站

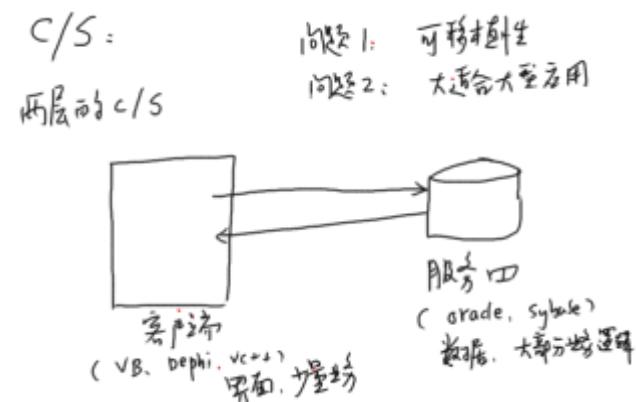
www.iteye.com

### 19.3 C/S 架构: Client/Server

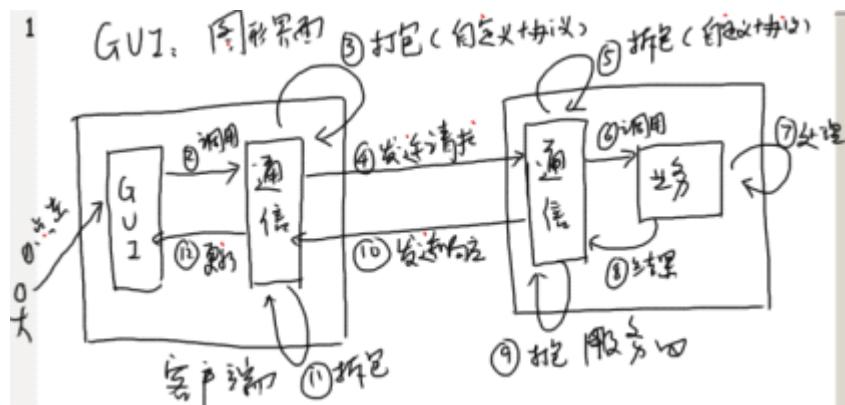
- 1) 两层的 C/S 架构:

优点: 发开效率高。

缺点: ①对数据库而言可移植性差。②不适合大型应用 (服务器提供的连接个数是有限的)。



- 2) 三层的 C/S 架构:



3) C/S 架构一定有自定义协议。

4) TCP/IP 协议有两个问题: ①拆包问题。②粘包问题。

5) 三层的 C/S 架构执行过程:

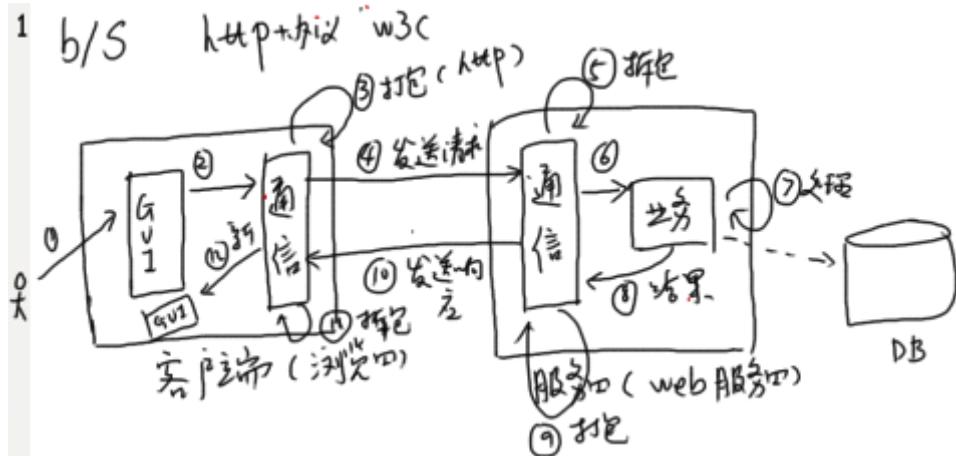
用户点击 GUI (图形界面), GUI 调用通信, 通信打包 (自定义协议), 发送请求给服务器通信, 通信拆包 (自定义协议), 调用业务, 业务处理, 结果发给通信, 通信打包, 发送响应给客户端通信, 通信拆包, 然后更新 GUI。

6) C/S 架构客户端需要单独安装。

## 19.4 B/S 架构: Browser/Server

和 C/S 三层架构相同，只是客户端不用写，服务器不用写，通信也不用写，浏览器里都有，采用 http 协议，是 w3c 标准，不需要自定义协议了。

只需要写界面（负责输入和输出）和业务逻辑（用 Java 写 Servlet、Jsp）。



用户点击 GUI（图形界面），GUI 调用通信，通信打包（http 协议），发送请求给 web 服务器通信，通信拆包（http 协议），调用业务，业务处理，结果发给通信，通信打包，发送响应给客户端通信，通信拆包，然后重新生成一个 GUI。

# 10 JSP 学习笔记

## 一百〇三、JSP 基础

### 1.1 什么是 JSP

JSP (Java Server Page) 是 Java 服务器端动态页面技术。是 sun 公司制订的一种服务器端的动态页面生成技术规范。

### 1.2 为什么要使用 JSP

因为直接使用 Servlet，虽然也可以生成动态页面。但是，编写繁琐（需要使用 `out.println` 来输出），并且维护困难（如果页面发生了改变，需要修改 Java 代码），所以 sun 指定了 JSP 规范。

### 1.3 JSP 与 Servlet 的关系

JSP 其实是一个以.jsp 为后缀的文件，容器会自动将.jsp 文件转换成一个.java 文件（其实就是一个 Servlet），然后调用该 Servlet。所以，从本质上讲，JSP 其实就是一个 Servlet。

### 1.4 如何写一个 JSP 文件

step1：创建一个以“.jsp”为后缀的文件。

step2：在该文件里面，可以添加如下的内容：

1) HTML (CSS、JS): 直接写即可

2) Java 代码:

形式一：Java 代码片段：`<% Java 代码 %>`

形式二：JSP 表达式：`<%= Java 表达式 %>`

形式三：JSP 声明：`<%! %>`

### 3) 指令

## 1.5 JSP 是如何运行的

step1：容器依据.jsp 文件生成.java 文件（也就是先转换成一个 Servlet）：

- 1) HTML (CSS、JS) 放到 service 方法里，使用 `out.write` 输出
- 2) `<% %>` 也放到 service 方法里，照搬，不改动。
- 3) `<%= %>` 也会放到 service 方法里，使用 `out.print` 输出。
- 4) `<%! %>` 给 Servlet 添加新的属性或者新的方法（转成.java 文件后，声明内的部分添加在 service 方法之外）。

这样就把一个 JSP 变成了一个 Servlet 容器。

◆ 注意事项：`out.writer` 方法只能输出简单的字符串，对象是没法输出的。优点是把 null 自动转换成空字符串输出。如：`<% out.println(new Date()); %>` 不能用 `writer`

step2：容器接下来就会调用 Servlet 来处理请求了（会将之前生成的.java 文件进行编译、然后实例化、初始化、调用相应的方法处理请求）

step3：隐含对象

1) 所谓隐含对象（共 9 个），指的是在.jsp 文件里面直接可以使用的对象，比如 `out`、`request`、`response`、`session`、`application`（`ServletContext` 上下文）、`exception`、`pageContext`、`config`、`page`。

2) 之所以能直接使用这些对象，是因为容器会自动添加创建这些对象的代码。（JSP 仅仅是个草稿，最终会变为一个 Servlet。）

## 1.6 隐含对象

1) exception 对象：当 jsp 页面运行时发生了异常，容器会将异常信息封装到该对象，可以使用该对象提供的方法来输出异常的信息。另外，必须在页面设置 `isErrorPage="true"` 指令才能使用该隐含对象。

2) pageContext 对象：容器会为每一个 JSP 实例（JSP 所对应的那个 Servlet 对象）创建唯一的一个符合 `pageContext` 接口的对象，称之为 `page` 上下文。该对象会一直存在，除非 JSP 实例被销毁。它作用：①绑定数据（绑定的数据只有对应的 JSP 实例才可以访问）：`setAttribute`、`removeAttribute`、`getAttribute`。②找到其他八个隐含对象（详情看 JSP 标签）。

3) config 对象：即 `ServletConfig`，可以使用该对象获得初始化参数。

例如：step1：a5.jsp 页面

```
company: <%=config.getInitParameter("company") %>
```

step2：web.xml 配置

```
<servlet> <servlet-name>a5</servlet-name>
 <jsp-file>/a5.jsp</jsp-file>
 <init-param><!-- jsp 初始化参数 -->
 <param-name>company</param-name>
 <param-value>chang</param-value>
 </init-param> </servlet>
<servlet-mapping> <servlet-name>a5</servlet-name>
```

```
</servlet-mapping>
step3: 测试输入/a5.html
```

4) page 对象: JSP 实例本身 (一般不用)。

## 1.7 指令

1) 指令是什么

通知容器, 在将.jsp 文件转换成.java 文件时, 作一些额外的处理, 比如导包。

2) 指令的语法

```
<%@指令名称 属性名=属性值 %>
```

3) page 指令

①import 属性: 导包。

例如: <%@page import="java.util.\*"%><!-- 注意: 没有分号! -->

```
<%@page import="java.util.*;java.text.*"%>
```

<!-- 多个包以逗号隔开! 都在一个双引号里-->

eg: 页面显示系统时间

```
current time:<% out.println(new Date()); %>

```

```
current time:<%=new Date()%>
```

◆ 注意事项: 两个是等价, new Date()会被放入 out.print(); 里

②contentType 属性: 设置 response.setContentType 的内容。

例如: <%@page import="java.util.\*" contentType="text/html;charset=utf-8" %>

<!-- 属性之间用空格隔开 -->

③pageEncoding 属性: 告诉容器.jsp 的文件的编码格式, 这样, 容器在获取 jsp 文件的内容 (即解码) 时, 不会出现乱码。最好加入, 有些容器默认以 ISO-8859-1 编码。

例如: <%@page contentType="text/html;charset=utf-8" pageEncoding="utf-8" %>

④session 属性: true/false, 缺省值 true, 如果值为 false, 则容器不会添加获得 session 的语句。

例如: <%@page session="false" %>

sessionID: <%=session.getId()%><!-- 报错: session cannot be resolved -->

⑤isELIgnored 属性: true/false, 缺省值 true, 如果值为 false, 则告诉容器不要忽略 el 表达式。J2EE5.0 需要使用 isELIgnored="false", 否则 EL 表达式无效。

⑥isErrorPage 属性: true/false, 缺省值 false, 如果值为 true, 表示这是一个错误处理页面 (即专门用来处理其他 JSP 产生的异常, 只有值为 true 时, 才能使用 exception 隐含对象去获取错误信息)。

例如: <%=exception.getMessage()%>

⑦errorPage 属性: 设置一个错误处理页面。

例如: step1: a3.jsp 页面, 测试输入 a3.jsp?num=100a

```
<%@page isErrorPage="a4.jsp" %>
```

```
<% String num=request.getParameter("num");
```

```
int sum=Integer.parseInt(num)+100; out.println(sum); %>
```

step2: a4.jsp 页面

```
<%@page isErrorPage="true" pageEncoding="UTF-8"
```

```
contentType="text/html;charset=utf-8"%>
```

<%=exception.getMessage()%><!-- 获取到错误信息: For input string: "100a" -->

->

- ◆ 注意事项：页面最好添加：`<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">`，否则会影响到 CSS 样式，甚至 JS。如 IE 有个混杂模式，即自动降级，为了兼容老的页面。

#### 4) include 指令

file 属性：①对于页面的公共部分，我们可以使用相同的.jsp 文件，并使用 include 指令导入，如此实现代码的优化。②告诉容器，在将.jsp 文件转换成.java 文件时，在指令所在的位置插入相应的文件的“内容”（由 file 属性来指定）。插入的页面并未运行，而是机械的将内容插入。

例如：`<%@include file="head.jsp" %>`

#### 5) taglib 指令

导入 JSP 标签`<%@taglib uri="命名空间" prefix="前缀" %>`

- ①uri：在 standard.jar/META-INF/c.tld 中查找，详情看 2.2
- ②prefix：前缀，用于代表命名空间。

例如：`<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>`

`<c:if test="\${1>0}">今天运气太好了</c:if>`

## 1.8 JSP 注释

1) `<!-- 注释内容 -->`：允许注释的内容是 Java 代码，如果是 Java 代码，会被容器执行，但是执行的结果会被浏览器忽略（不会显示出来）

2) `<%-- 注释内容 --%>`：注释的内容不能是 Java 代码，如果是 Java 代码，会被容器忽略。

## 1.9 案例：创建 emplist.jsp 页面，将表示逻辑交给 JSP 处理

详见 Servlet 笔记第 9 章。

## 一百〇四、JSP 标签和 EL 表达式

### 2.1 什么是 JSP 标签

sun 公司制定的一种技术规范，利用 JSP 标签（类似于 HTML 标签）来代替 JSP 中的 Java 代码。这样作的好处是，JSP 页面会更加简洁，并且更好维护（比如，将这样的页面交给美工，可以很方便地去修改）。

### 2.2 JSTL 及其使用

1) 什么是 JSTL: Java Standard Taglib (Java 标准标签库, apache 开发了一套标签，捐献给了 sun, sun 将其命名为 JSTL)。

2) 如何使用:

step1: 将 JSTL 标签对应的 jar (标签类) 文件拷贝到 WEB-INF\lib 下, standard.jar、jstl.jar

step2: 使用 taglib 指令引入 JSP 标签

### 2.3 什么是 EL 表达式

是一套简单的计算规则，用于给 JSP 标签的属性赋值，也可以直接输出。

注意事项: 新建工程, J2EE 选 1.4 可直接使用 EL 表达式, JavaEE5.0 需要使用 isELIgnored 属性，否则无法使用!! 详见 1.7。

### 2.4 EL 表达式的使用

1) 访问 bean 的属性 (就是普通的 Java 类，有属性和 get/set 方法)

第一种方式: 例如: \${user.name}，容器会依次从 4 个隐含对象中 pageContext、request、session、application 中查找 (getAttribute) 绑定名为"user"的对象。接下来，会调用该对象的"getName"方法 (自动将 n 变大写然后加 get)，最后输出执行结果。

优点: ①会自动将 null 转换成""输出。

②如果绑定名称对应的值不存在，会不报 null 指针异常，会输出""。

◆ 注意事项: 依次是指先从 pageContext 中查找，如果找不到，再查找 request，如果找到了，则不再向下查找。如果要**指定查找范围**，可以使用 pageScope、requestScope、sessionScope、applicationScope 来指定查找的范围。

eg: jsp 页面测试

```
<% User user=new User(); user.setName("chang"); user.setAge(22);
request.setAttribute("user",user); User user2=new User(); user2.setName("bo");
user2.setAge(22); session.setAttribute("user",user2); %>
name: <% User user=(User)request.getAttribute("user"); out.println(user.getName());
%>

<!-- 用 Java 代码输出 --><!-- chang -->
name: ${user.name}
<!-- 用 EL 表达式输出 --><!-- chang -->
name: ${sessionScope.user.name} <!-- 指定查找范围 --><!-- bo -->
```

第二种方式: 比如 \${user["name"]}，与第一种方式是等价的！容器会依次从 4 个隐含对象中 pageContext、request、session、application 中查找 (getAttribute) 绑定名为"user"的对象。接下来，会调用该对象的"getName"方法 (自动将 n 变大写然后加 get)，最后输出执行结果。

优点: ①中括号[]里面可以出现变量。②中括号[]里面可以出现下标从 0 开始的数

组。

◆ 注意事项：

- ❖ 中括号[]里的字符串用双引号、单引号都可以！
- ❖ EL 表达式中没引号的为变量，有引号的为字符串。

eg: jsp 页面测试

step1: User 类

```
private String name; private int age; private String[] interest;
private IdCard card; get/set 方法
```

step2: IdCard 类

```
private String cardNo; get/set 方法
```

step3: jsp 页面

```
<% User user=new User(); user.setName("chang"); user.setAge(22);
 user.setInterest(new String[]{"台球","乒乓球"});
 user.setCard(new IdCard("1008611")); request.setAttribute("user",user);
%>
name: ${user["name"]}
<!-- 基本类型，与${user.name}等价 -->
<% request.setAttribute("propname","age"); %>
name: ${user[propname]}
<!-- 变量 -->
interest: ${user.interest}
<!-- 数组地址 -->
interest: ${user.interest[1]}
<!-- 数组某个元素的值 -->
IdCard: ${user.card.cardNo }<!-- 引用类型 -->
```

2) 进行一些简单的计算，计算的结果可以用来给 JSP 标签的属性赋值，也可以直接输出。

①算术运算：“+”、“-”、“\*”、“/”、“%”

◆ 注意事项：“+”号操作不能连接字符串。如 "abc"+"bcd" 会报错！

"100"+ "200"=\${"100"+ "200"} 可以！

②关系运算：“>”、“>=”、“<”、“<=”、“!=”、“==”

◆ 注意事项：“eq”也可判断是否相等

eg: 相关测试

```
1>0?${1>0 }
<!-- true -->
<% request.setAttribute("str1","abc"); request.setAttribute("str2","bcd"); %>
${str1==str2}
<!-- false --> ${str1=="abc"}
<!-- true -->
eq: ${str1 eq "abc" }
<!-- true --> ${sessionScope.str1=="abc"}
<!-- false -->
```

③逻辑运算：“&&”、“||”、“!”，与 Java 中的一样

④empty 运算：判断是否是一个空字符串，或者是一个空的集合，如果是，返回 true。

以下四种情况都是 true: A.空字符串。B.空集合。C.null。

D.根据绑定名找不到值。

eg: 相关测试

```
<% request.setAttribute("str3","");
%>
空字符串: ${empty str3 }
<!-- true -->
<% List<String> list1=new ArrayList<String>(); //list1.add("abc");//false
 request.setAttribute("list1",list1); %>
空集合: ${empty list1 }
<!-- true -->
<% request.setAttribute("obj",null); %>
```

null: \${empty obj }<br /><!-- true -->  
找不到值: \${empty abc }<br /><!-- true -->

LICHOO

3) 获取请求参数值

- ①\${param.username} 等价于 request.getParameter("username");
  - ②\${paramValues.city} 等价于 request.getParameterValues("city");
- eg: 相关测试 a6.jsp 页面

```
username: ${param.username }<!-- 测试输入: a6.jsp?usernme=1 -->

interest: ${paramValues.interest[0] }
<!-- 测试输入: a6.jsp?username=6&interest=cooking&interest=fishing -->
```

# 一百〇五、JSTL 中的几个核心标签

LICHOO

## 3.1 if

1) 语法: <c:if test="" var="" scope=""></c:if>

当 test 属性值为 true, 执行标签体的内容, test 属性可以使用 EL 表达式。

2) var 属性: 用来指定绑定名称。

3) scope 属性: 指定绑定范围, 可以是 page (pageContext)、request、session、application

◆ 注意事项: 可以在 if 标签里写 Java 代码。

## 3.2 choose

1) 语法: <c:choose><!-- 用于分支, 当满足某个条件, 执行某一个分支 -->

<c:when test=""><!-- 分支, 可多次出现 -->

</c:when>

...

<c:otherwise><!-- 当其他分支都不满足条件, 则执行该标签的内容 -->

</c:otherwise>

</c:choose>

◆ 注意事项:

❖ when 和 otherwise 必须要放到 choose 标签里面才能使用。

❖ when 可以出现 1 次或者多次, otherwise 可以出现 0 次或者 1 次, 表例外。

eg: 相关测试

```
<% Person p=new Person(); p.setName("常"); p.setGender("x");
 request.setAttribute("p",p); %>
性别: <c:choose>
 <c:when test="${p.gender=='m'}">男</c:when>
 <c:when test="${p.gender=='f'}">女</c:when>
 <c:otherwise>保密</c:otherwise>
</c:choose>
```

## 3.3 forEach

1) 语法:

用法一 (遍历集合): <c:forEach var="" items="" carStatus=""></c:forEach>

①items 属性: 用来指定要遍历的集合, 可以使用 EL 表达式。

②var 属性: 指定绑定名, 绑定范围是 pageContext, 绑定值是从集合中取出的某个元素。

③carStatus 属性: 指定绑定名, 绑定范围是 pageContext, 绑定值是一个由容器创建的一个对象, 该对象封装了当前迭代的状态。比如, 该对象提供了 getIndex、getCount 方法, 其中, getIndex 会返回当前迭代的元素的下标 (从 0 开始), getCount 会返回当前迭代的次数 (从 1 开始)。

用法二 (指定位置迭代): <c:forEach var="" begin="" end=""></c:forEach>

①begin: 如果指定了 items, 那么迭代就从 items[begin]开始进行迭代; 如果没有指定 items, 那么就从 begin 开始迭代。它的类型为整数。

②end: 如果指定了 items, 那么就在 items[end]结束迭代; 如果没有指定 items, 那

么就在 end 结束迭代。它的类型也为整数。

- ◆ 注意事项：forEach 还一个属性为 step=""：迭代的步长。

## 3.4 url

- 1) 语法：`<c:url value="">`
  - ①当用户禁止 cookie 以后，会自动在地址后面添加 sessionId。
  - ②当使用绝对路径时，会自动在地址前添加应用名。
- 2) value 属性：指定地址，在表单提交、链接当中，可以使用该标签。

## 3.5 set

- 1) 语法：`<c:set var="" scope="" value="">`，绑定一个对象到指定的范围。
- 2) value 属性：绑定值。

## 3.6 remove

- 1) 语法：`<c:remove var="" scope="">`，解除绑定。

## 3.7 catch

1) 语法：`<c:catch var="">`，处理异常，会将异常信息封装成一个对象，绑定到 pageContext 对象上。

## 3.8 import

- 1) 语法：`<c:import url="">`
- 2) url 属性：指定一个 jsp 文件的地址，jsp 会在运行时调用这个 jsp。

## 3.9 redirect

- 1) 语法：`<c:redirect url="">`，重定向到另外一个地址。
- 2) url 属性：指定重定向的地址。

## 3.10 out

- 1) 语法：`<c:out value="" default="" escapeXml="">`，用于输出 el 表达式的值。
- 2) value 属性：指定输出的值。
- 3) default 属性：指定缺省值。
- 4) escapeXml 属性：设置成 true，会将 value 中的特殊字符替换成相应的实体。缺省值就是 true。

## 3.11 JSP 标签是如何运行的

容器依据命名空间找到标签的描述文件 (.tld 文件)，接下来，依据标签的名称找到标签类的类名，然后将该标签实例化，最后，调用标签实例的相应的方法。

- ◆ 注意事项：容器会从 WEB-INF 下查找，如果找不到，还会查找 WEB-INF\lib 下的 jar 文件。

### 3.12 案例：将员工列表中的 Java 代码改为 JSP 标签，并添加分页

step1：删除该页上所有 Java 代码，并修改如下

```
<c:forEach var="e" items="${employees}" varStatus="s">
 <tr class="row${s.index%2+1}">

<td>${e.id}</td><td>${e.name}</td><td>${e.salary}</td><td>${e.age}</td>
 <td>删除员工

 修改员工</td></tr>
 </c:forEach>
```

step2：JSP 页面添加分页

```
<h2> <c:choose>
 <c:when test="${page > 1}">
 上一页
 </c:when>
 <c:otherwise>上一页</c:otherwise>
</c:choose>
第${page}页
<c:choose>
 <c:when test="${page < totalPages}">
 下一页
 </c:when>
 <c:otherwise>下一页</c:otherwise>
</c:choose>
总共${totalPages}页 </h2>
```

step3：修改 ActionServlet 中 service 方法中的 if 语句

```
if(action.equals("/list")){
 String page=request.getParameter("page");//读页数 page
 if(page==null){ page="1"; }/刚打开页面时，应该显示第 1 页
 try {//使用工厂来访问数据库
 EmployeeDAO dao=(EmployeeDAO) Factory.getInstance("EmployeeDAO");
 //List<Employee> employees=dao.findAll();
 List<Employee> employees=dao.findByPage(Integer.parseInt(page), 5);
 int totalPages = dao.totalPages(5);
 request.setAttribute("page", Integer.parseInt(page));
 request.setAttribute("totalPages", totalPages);
 request.setAttribute("employees", employees);//绑定数据
 RequestDispatcher rd=request.getRequestDispatcher("emplist.jsp");//转发器
 rd.forward(request, response);/转发
 }catch(Exception e){}
```

# 一百〇六、自定义标签

LICHOO

## 4.1 如何写一个自定义标签

step1：写一个 Java 类（标签类），且必须继承 SimpleTagSupport 类。

step2：在 doTag 方法里面（覆盖 doTag 方法），编写相应的处理逻辑。标签有哪些属性，则标签类也有哪些属性，并且类型要匹配。此外，这些属性必须提供相应的 set 方法

step3：在.tld 文件当中，描述该标签。.tld 文件可以放在 WEB-INF 下（或它的子文件夹下），也可以放到 META-INF 下，可以参考 c.tld 文件来写。

### ◆ 注意事项：

- ❖ 简单标签技术是新技术，新的规范。
- ❖ <body-content></body-content>的作用是告诉容器标签是否有标签体（即开始、结束标签之间的内容），如果有标签体可以有三个值：
  - 1) empty：没有标签体。
  - 2) scriptless：可以有标签体，但是标签体里面不能够出现 Java 代码（三种形式都不行！）。
  - 3) JSP：有标签体，并且标签体的内容可以是 Java 代码。只有复杂标签技术支持 JSP。简单标签技术只支持 empty 和 scriptless。

## 4.2 JavaEE5.0 中，如何使用 EL 表达式和 JSTL

Tomcat5.5	对应 Servlet2.4 规范	对应 J2EE1.4
Tomcat6.0	对应 Servlet2.5 规范	对应 JavaEE5.0
Tomcat7.0	对应 Servlet3.0 规范	对应 JavaEE6.0

在 JavaEE5.0 以上版本中，已经将 JSTL 标签库对应的 jar 文件包含进来了，不用再去将那两个 jar 文件拷贝到 WEB-INF\lib 下了。

## 4.3 案例：自定义标签

step1：JSP 页面

```
<%@taglib uri="http://www.chang.com.cn/mytag" prefix="c1" %>
<c1:hello msg="hello world" qty="\${1+9}" />
```

step2：编写 HelloTag，继承 SimpleTagSupport

```
public class HelloTag extends SimpleTagSupport{
 private String msg; private int qty;
 public HelloTag(){ System.out.println("HelloTag's constructor..."); }
 public void setMsg(String msg){
 System.out.println("setMsg..." +msg); this.msg = msg; }
 public void setQty(int qty){
 System.out.println("setQty..." +qty); this.qty = qty; }
 public void doTag() throws JspException, IOException {
 System.out.println("HelloTag's doTag...");
 //通过 SimpleTagSupport 类提供的 getJspContext 方法获得 pageContext 对象
 PageContext ctx=(PageContext)getJspContext();
```

LICHOO

```
//pageContext 提供了获得其他所有隐含对象的方法
JspWriter out=ctx.getOut();
for(int i=0;i<qty;i++){ out.println(msg+"
"); } }
```

step3：在 mytag.tld 文件当中，描述该标签（注意该文件的放置位置）

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
 http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd" version="2.0">
 <tlib-version>1.1</tlib-version>
 <short-name>c1</short-name>
 <uri>http://www.chang.com.cn/mytag</uri>
 <tag>
 <name>hello</name>
 <tag-class>tag>HelloTag</tag-class>
 <body-content>empty</body-content>
 <attribute>
 <name>msg</name>
 <required>true</required><!-- 属性值是否是必须赋值的 -->
 <rtpvalue>false</rtpvalue><!-- 是否可以动态赋值（如 EL 表达式）-->
 </attribute>
 <attribute>
 <name>qty</name>
 <required>true</required><!-- 属性值是否是必须赋值的 -->
 <rtpvalue>true</rtpvalue><!-- 是否可以动态赋值（如 EL 表达式）-->
 </attribute>
 </tag>
</taglib>
```

## 4.4 案例：修改之前员工列表中的日期显示

step1：JSP 页面

```
<%@taglib uri="http://www.chang.com.cn/mytag" prefix="c1" %>
<c1:date pattern="yyyy-MM-dd" />
```

step2：编写 DateTag，继承 SimpleTagSupport

```
public class DateTag extends SimpleTagSupport{
 private String pattern;
 public void setPattern(String pattern) { this.pattern = pattern; }
 public void doTag() throws JspException, IOException {
 PageContext ctx=(PageContext) getJspContext();
 JspWriter out=ctx.getOut();
 Date date=new Date();
 SimpleDateFormat sdf=new SimpleDateFormat(pattern);
 out.println(sdf.format(date)); }}
```

step3：在 mytag.tld 文件当中，描述该标签（注意该文件的放置位置）

```
<tag>
 <name>date</name>
 <tag-class>tag.DateTag</tag-class>
 <body-content>empty</body-content>
 <attribute>
 <name>pattern</name>
 <required>true</required><!-- 属性值是否是必须赋值的 -->
 <rteprvalue>false</rteprvalue><!-- 是否可以动态赋值（如 EL 表达式）-->
 </attribute>
</tag>
```

# 一百〇七、MVC

LICHOO

## 5.1 什么是 MVC

Model View Controller，是一种软件架构的思想，将一个软件的模块划分成三种不同类型的模块，分别是模型（封装业务逻辑）、视图（实现表示逻辑）、控制器（协调模型和视图，即模型返回的结果要先交给控制器，由控制器来选择合适的视图来展示处理结果；另外，视图发送请求给控制器，由控制器来选择对应的模型来处理。）

## 5.2 使用 MVC 的目的

使用 MVC 思想来设计软件，最根本的目的是为了实现模型的复用：即模型只返回处理结果，并不关心这些结果如何展现，展现由不同的视图来处理；另外可以使用不同的视图来调用同一个模型。

## 5.3 如何使用 MVC 来开发一个 Web 应用程序（JavaEE）

使用 Java 类来实现模型（将业务逻辑写在 Java 类里面，写完之后可以立即测试），使用 Servlet 或者 Filter 来实现控制器，使用 JSP 来实现视图。

- ◆ 注意事项：
  - ❖ 一般模型产生应用异常（即不是系统的原因产生的异常，而是用户使用不当造成的，需要提示用户采取正确的操作）时，抛自定义异常给控制器（自定义异常可自己写个最简单的类，但是，当有许多应用异常时可采用异常编号方式），控制器再返回结果给视图。
  - ❖ 对于客户端，WEB-INF 下的文件都是受保护的，不能直接访问！只有服务器之间的组件可以访问。某个结果页面不想被用户访问，可以设置主页，然后使用 ActionServlet 采取转发方式。

## 5.4 MVC 的优缺点

- 1) 优点：①可以实现模型的复用。②模型或者视图发生改变，不会相互影响。  
③方便测试（比如，将业务逻辑写在 Java 类里面，可以直接测试，如果写在 Servlet 里面，必须要先部署才能测试）。
- 2) 缺点：使用 MVC 后，①会增加设计的难度。②代码量也会增加。③相应地也会增加软件的开发成本。

## 5.5 案例：简易贷款（贷款数小于余额数\*10）

step1: 实体类 Account

```
private int id; private String accountNo; private double balance;get/set 方法
```

step2: AccountDAO 类

```
public Account findByAccountNo(String accountNo){
 Account account=null; Connection conn=null;
 try { conn=DBUtil.getConnection();//借用之前案例中的 DBUtil 类
 PreparedStatement prep=conn.prepareStatement(
 "select * from chang_account where accountNo=?");
 prep.setString(1,accountNo); ResultSet rs=prep.executeQuery();
 while(rs.next()){ account = new Account();
```

```

 account.setAccountNo(accountNo);
 account.setBalance(rs.getDouble("balance"));
 account.setId(rs.getInt("id"));
 }
} catch (Exception e) {
 e.printStackTrace();
} finally {
 DBUtil.close(conn);
}
return account;
}

```

step3: AccountService 业务逻辑类

```

public String apply(String accountNo,double amount) throws Exception{
 String number = "";
 //step1: 检查帐号是否存在, 如果不存在, 要提示用户帐号不存在, 否则进行下一步
 AccountDAO dao = new AccountDAO();
 Account a = dao.findByAccountNo(accountNo);
 if(a == null){//帐号不存在: 抛出一个自定义异常
 throw new AccountNotExistException();
 }
 //throw new ApplicationException("10001000");//当应用异常过多时, 可采异常编号方式
 //step2: 检查余额是否充足, 如果余额不足, 要提示用户, 否则, 进行下一步
 if(a.getBalance() * 10 < amount){//余额不足:
 throw new AccountLimitException();
 }
 //step3: 生成一个序列号, 并且保存该序列号到数据库
 Random r = new Random(); number = r.nextInt(1000000) + "";
 //保存该序列号到数据库, 此处略...
 return number;
}

```

step4: apply.jsp 页面

```

${apply_error}
<form action="apply.do" method="post">
 <fieldset><legend>申请贷款</legend>
 帐号:<input name="accountNo"/>

 金额:<input name="amount"/>

 <input type="submit" value="提交"/></fieldset></form>

```

step5: ActionServlet 中 service 方法

```

String uri = request.getRequestURI();
String action = uri.substring(uri.lastIndexOf("/"), uri.lastIndexOf("."));
// step1: 分析请求资源路径, 依据请求调用相应的模型来处理。
if (action.equals("/apply")) {
 String accountNo = request.getParameter("accountNo");
 double amount = Double.parseDouble(request.getParameter("amount"));
 AccountService service = new AccountService();
 try { String number = service.apply(accountNo, amount);
 //step2: 依据模型返回的结果来选择合适的视图
 request.setAttribute("number", "贷款申请成功,请记住序列号:" + number);
 request.getRequestDispatcher("view.jsp").forward(request, response);
 } catch (Exception e) {
 e.printStackTrace();
 }
}

```

```
if (e instanceof AccountLimitException) { //应用异常
 request.setAttribute("apply_error", "余额不足");
 request.getRequestDispatcher("apply.jsp").forward(request, response);
} else if (e instanceof AccountNotExistException) {
 request.setAttribute("apply_error", "帐号不存在");
 request.getRequestDispatcher("apply.jsp").forward(request, response);
} else { throw new ServletException(e); //系统异常 } }
```

step6: 自定义异常

```
public class AccountLimitException extends Exception {}
public class AccountNotExistException extends Exception {}
```

step7: view.jsp 页面

```
 ${number}
```

## 5.6 修改 5.5 案例，使用户无法直接访问 view.jsp 页面

若直接访问则页面为空，因为没有转发。

step1: 将 apply.jsp、view.jsp 页面放到 WEB-INF 文件夹中，可建一个文件夹名为 jsp。相关介绍看 5.3 注意事项。

step2: 写一个主页 index.jsp，主页放在 WebRoot 文件夹下

```
申请贷款
```

step3: web.xml 配置主页

```
<welcome-file-list>
 <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

step4: 修改 ActionServlet 中 service 方法

1) 添加 if 判断语句

```
if(action.equals("/to_apply")) { //绝对路径
 request.getRequestDispatcher("/WEB-INF/jsp/apply.jsp").forward(request, response);
}
```

2) try 块中转发修改

```
request.getRequestDispatcher("/WEB-INF/jsp/view.jsp").forward(request, response);
```

3) catch 块中转发修改

```
if (e instanceof AccountLimitException) {
 request.setAttribute("apply_error", "余额不足");

 request.getRequestDispatcher("/WEB-INF/jsp/apply.jsp").forward(request, response);
} else if (e instanceof AccountNotExistException) {
 request.setAttribute("apply_error", "帐号不存在");

 request.getRequestDispatcher("/WEB-INF/jsp/apply.jsp").forward(request, response);
```

# 11 Ajax 学习笔记

LICHOO

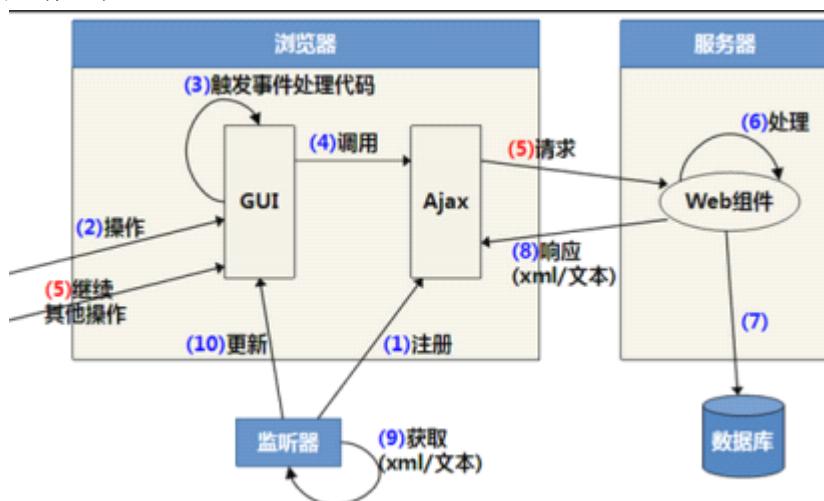
## 一百〇八、Ajax 概述

### 1.1 什么是 Ajax

Asynchronous Javascript And Xml (异步的 JavaScript 和 Xml)。是一种用来改善用户体验的技术，其实质是利用浏览器内置的一个特殊对象（XMLHttpRequest，一般称之为 Ajax 对象）异步地（Ajax 对象在向服务器发送请求时，浏览器并不会销毁当前页面，用户仍然可以对当前页面作其他的操作）向服务器发送请求，服务器送回部分数据（不是一个完整的新的页面，而是文本或者 Xml 文档），在浏览器端，可以利用这些数据部分更新当前页面。整个过程，页面无刷新，不打断用户的操作。

之前，都是先销毁原来的页面，然后发送请求，等待服务器发送响应，再生成新页面。

Ajax 的工作流程：



### 1.2 Ajax 对象：如何获得 Ajax 对象

由于 XMLHttpRequest (Ajax 对象) 没有标准化，所以要**区分浏览器**。

```
function getXhr(){//注意：后面的案例都将用到此函数
 var xhr=null;
 if(window.XMLHttpRequest){
 xhr=new XMLHttpRequest();//非 IE 浏览器
 }else{
 xhr=new ActiveXObject('Microsoft.XMLHttp');//IE 浏览器
 }
 return xhr;
}
```

◆ 注意事项：后面的案例**也会用到**以下函数

```
function $(id){//依据 id 返回 dom 节点
 return document.getElementById(id);
}
function $F(id){//返回 id 对应的值
}
```

```
 return $(id).value;
}
```

LICHOON

## 1.3 Ajax 对象的属性

1) `onreadystatechange`: 绑定一个事件处理函数（监听器），该函数用来处理 `readystatechange` 事件。Ajax 对象的 `readyState` 属性发生改变，比如从 0 到 1，则会产生 `onreadystatechange` 事件。

- 2) `responseText`: 获得服务器返回的文本数据。
- 3) `responseXML`: 获得服务器返回的 Xml 文档。
- 4) `status`: 获得状态码。
- 5) `readyState`: 返回 Ajax 对象与服务器通讯的状态，返回值是一个 `number` 类型的值。

一共有 5 个值，分别是：

- ①0: (未初始化) 对象已建立，但是尚未初始化（尚未调用 `open` 方法）。
- ②1: (初始化) 对象已建立，尚未调用 `send` 方法。
- ③2: (发送数据) `send` 方法已调用。
- ④3: (数据传送中) 已接收部分数据。
- ⑤4: (响应结束) Ajax 对象已经获得了服务器返回的所有的数据。

## 1.4 编程步骤

1) 发送 get 请求：

step1: 获得 Ajax 对象，比如：`var xhr=getXhr();`//调用之前定义的函数

step2: 使用 Ajax 对象发送 get 请求

    ①调用 `xhr.open('get','check_username.do?username=chang&age=23,true);` 方法：  
建立与服务器之间的连接，三个参数依次为：请求方式、请求资源路径、请求是同步还是异步。

`true`: 表示异步请求（Ajax 对象发送请求时，用户可以对当前页面作其他的操作，不会销毁页面）。

`false`: 表示同步请求（Ajax 对象发送请求时，浏览器会锁定当前页面，用户只能等待，不会销毁页面）。

    ②`xhr.onreadystatechange=func1();`: 绑定一个事件处理函数（监听器）

    ③`xhr.send(null);`: 发送请求参数，因为参数已经写在了请求资源路径中，所以这里为 `null`。

step3: 编写服务器端的处理程序，跟以前相比，有一点点改变，就是一般不需要返回一个完整的页面，只需要返回部分的数据。

step4: 编写事件处理函数

```
function f1(){
 if(xhr.readyState==4){
 var txt=xhr.responseText;
 dom 操作.....
 }
}
```

2) 发送 post 请求：

step1: 获得 Ajax 对象，比如：`var xhr=getXhr();`//调用之前定义的函数

step2: 使用 Ajax 对象发送 post 请求

①xhr('post','check\_username.do',true);: 建立连接  
②xhr.setRequestHeader('content-type','application/x-www-form-urlencoded');: 发送一个 content-type 消息头  
③xhr.onreadystatechange=func1();: 绑定一个事件处理函数（监听器）  
④xhr.send('username=chang');: 发送请求参数

◆ 注意事项：

- ❖ 与 get 请求不同，请求参数要放到 send 方法里面。
- ❖ 因为按照 HTTP 协议的要求，发送 post 请求时，应该发送一个 content-type 消息头，而 Ajax 对象在默认情况下，不会发送这个消息头，所以，需要调用 setRequestHeader 方法来添加。

step3：编写服务器端的处理程序，跟以前相比，有一点点改变，就是一般不需要返回一个完整的页面，只需要返回部分的数据。

step4：编写事件处理函数

## 1.5 编码问题

### 1) 发 get 请求

#### ①乱码产生的原因：

IE 浏览器内置的 Ajax 对象会使用“GBK”或“GB2312”对中文参数进行编码，而其他浏览器（Chrom、Firefox）内置的 Ajax 对象会使用“utf-8”对中文参数进行编码。服务器端，默认会使用“ISO-8859-1”去解码。因为编码与解码所使用的字符集（编码格式）不一致，所以，会出现乱码问题。

#### ②解决：

step1：设置服务器使用指定的字符集去解码。比如，可以修改 Tomcat 的 server.xml 配置（conf 文件夹中），添加 URIEncoding="utf-8"（告诉服务器，对于所有的 get 请求，默认使用“utf-8”去解码），修改之后重新启动服务器。

```
<Connector port="8080" maxHttpHeaderSize="8192"
 maxThreads="150" minSpareThreads="25" maxSpareThreads="75"
 enableLookups="false" redirectPort="8443" acceptCount="100"
 connectionTimeout="20000" disableUploadTimeout="true" URIEncoding="utf-8" />
```

step2：使用 encodeURI() 函数（JS 中内置函数）对请求地址进行编码。encodeURI() 函数会使用“utf-8”进行编码

```
xhr.open('get','check_username.do?username='+$F('username'),true);
var uir='check_username.do?username='+$F('username');
xhr.open('get',encodeURI(uir),true);
```

### 2) 发 post 请求

#### ①乱码问题产生的原因：

所有浏览器（一般指三大浏览器：Chrom、Firefox、IE）内置的 Ajax 对象都会使用“utf-8”对中文参数进行编码，而服务器默认情况下，会使用“ISO-8859-1”去解码。

◆ 注意事项：Firefox 特殊，本应是乱码，但能正常显示。通过截取消息头发现 Firefox 会在消息头中自动添加“charset=utf-8”。

#### ②解决：

服务器端添加：request.setCharacterEncoding("utf-8");

## 1.6 Ajax 的优点

- 1) 页面无刷新，不打断用户的操作。
- 2) 按需要获取数据，客户端（浏览器）与服务器端之间的数据传输量大大减少。
- 3) 是一种标准化的技术，不需要下载任何插件。

## 1.7 缓存问题（IE 浏览器）

- 1) 当发送 get 请求时

IE 浏览器（其他浏览器没这个问题）内置的 Ajax 对象会检查请求地址是否访问过，如果访问过，则不再向服务器发送请求。

- 2) 解决方式

方式一：在请求地址后面添加一个随机数，用于欺骗 IE，例如：

```
xhr.open('get','getNumber.do?' + Math.random(),true);
```

方式二：使用 post 方式发请求。

- 3) 案例：在 IE 浏览器中测试缓存问题

step1: getNumber.jsp 页面

```
点这儿，在链接底下显示一个随机数
```

```
<div id="d1"></div>
```

step2: JS 代码 getNumber 函数

```
function getNumber(){
 var xhr=getXhr();
 xhr.open('get','getNumber.do?' + Math.random(),true);//若没随机数，则点链接不会变化
 xhr.onreadystatechange=function(){
 if(xhr.readyState==4){
 var txt=xhr.responseText;
 $('#d1').innerHTML=txt;
 }
 xhr.send(null);
 }
}
```

step3: Servlet 中 service 方法中的 if 判断

```
if(action.equals("/getNumber")){
 Random r=new Random();
 int number=r.nextInt(10000);
 System.out.println(number);
 out.println(number);
}
```

## 1.8 案例：简易注册（使用 Ajax 进行相关验证，get 请求）

step1: 编写 myScript.js 并放在 js 文件夹中

此处省略三个函数，详看 1.2

```
function check_username(){//检查用户名，get 请求
 var xhr=getXhr();//step1 获得 Ajax 对象
 //step2 使用 Ajax 对象发送请求
 xhr.open('get','check_username.do?username='+$F('username'),true);
 //绑定一个事件处理函数，里面的代码在状态改变时执行，且状态为 4 时执行 if 语句
}
```

```

xhr.onreadystatechange=function(){
 if(xhr.readyState==4){
 if(xhr.status==200){//服务器返回了正确的结果
 //只有 readyState 为 4 时， Ajax 对象才获得服务器返回的所有数据

 var txt=xhr.responseText;
 $('#username_msg').innerHTML=txt;//利用服务器返回的数据更新页
面

 }else{//服务器运行出错
 $('#username_msg').innerHTML='验证出错'; } }
 };
 $('#username_msg').innerHTML='正在验证...';//模拟用户量较大的情况，显示正在验
证

 xhr.send(null); }

function check_number(){//检查验证码
 var xhr=getXhr();
 xhr.open('get','check_number.do?number='+$F('number'),true);
 xhr.onreadystatechange=function(){
 if(xhr.readyState==4){
 var txt=xhr.responseText;
 $('#number_msg').innerHTML=txt; }
 xhr.send(null); }
 }
}

```

step2：编写 regist.jsp 页面， get 请求

```

<form action="regist.do" method="get">

 <fieldset>
 <legend>注册</legend>
 用户名: <input type="text" name="username" id="username"
 onblur="check_username();"/>

 真实姓名: <input type="text" name="name" />

 验证码: <input type="text" name="number" id="number"
 onblur="check_number();"/>

 </>

 看不清换一个

 <input type="submit" value="提交" />
 </fieldset>
</form>

```

- ◆ 注意事项： href="javascript:;" 相当于 href="#"，写成 href="javascript:;" 一般要和 onclick 事件一起使用，表示 a 元素不再指向一个地址，而是点击后触发一个事件。

step3：ActionServlet 中 service 方法

```
response.setContentType("text/html;charset=utf-8");
```

```

PrintWriter out = response.getWriter();
String uri = request.getRequestURI();
String action = uri.substring(uri.lastIndexOf("/"), uri.lastIndexOf("."));
if(action.equals("/check_username")) {
 String username=request.getParameter("username");
 // try {//模拟耗时操作
 // Thread.sleep(6000); } catch (InterruptedException e) { e.printStackTrace(); }
 if(1==1){//模拟一个系统异常
 throw new ServletException("some error");
 }
 if(username.equals("常")){
 out.println("已被占用"); } else{ out.println("可以使用"); }
} else if(action.equals("/check_number")){
 String number1=request.getParameter("number");
 HttpSession session=request.getSession();
 String number2=(String)session.getAttribute("number");
 if(number1.equalsIgnoreCase(number2)){ out.println("验证码正确");
 } else{ out.println("验证码错误"); }
} else if(action.equals("/regist")){
 //加上验证代码，比如检查用户名是否正确，验证码是否正确，此处略
 System.out.println("注册成功");
}

```

step4: CheckcodeServlet 借用之前 Servlet 笔记中 13.20 案例，随机生成验证码

## 1.9 案例：修改 1.8 案例，使用 post 请求

step1：修改 regist.jsp 页面请求方式为 post

step2：添加 JS 验证代码 check\_username\_post

```

function check_username_post(){
 var xhr=getXhr();
 xhr.open('post','check_username.do',true);
 //添加一个消息头
 xhr.setRequestHeader('content-type','application/x-www-form-urlencoded');
 xhr.onreadystatechange=function(){
 if(xhr.readyState==4){
 var txt=xhr.responseText;
 $('#username_msg').innerHTML=txt; }
 }
 xhr.send('username='+$F('username'));
}

```

## 1.10 案例：使用 Ajax 实现下拉列表

step1：ActionServlet 中 service 方法中的 if 判断

```

if(action.equals("/select")){
 String name = request.getParameter("name");
 if(name.equals("qq")){ out.println("性价比高");
 } else if(name.equals("bmw")){ out.println("驾驶性能出众");
 } else{ out.println("好车,也贵"); }
}

```

step2：select.jsp 页面

```
<select id="s1" onchange="change(this.value);>
 <option value="qq">QQ</option><option value="bmw">宝马</option>
 <option value="ff">法拉利</option></select>
<div id="d1"></div>
```

### step3: JavaScript 代码

```
function change(value){
 var xhr=getXhr(); xhr.open('get','select.do?name='+value,true);
 xhr.onreadystatechange=function(){
 if(xhr.readyState==4){ var txt=xhr.responseText; $('#d1').innerHTML=txt; }
 };
 xhr.send(null);
}
```

# 一百〇九、JSON

LICHOO

## 2.1 什么是 JSON

JavaScript Object Notation，是一种轻量级的数据交换技术规范（因为借鉴了 JavaScript 对象创建的一种语法结构，故命名为 JSON，详情见 JavaScript 第 8 章）。

## 2.2 数据交换

将数据转换成一种中间的，与平台无关的数据格式（比如 Xml 或者 JSON 字符串）发送给另外一方来处理。

## 2.3 轻量级

JSON 相对于 Xml，所需的数据大小要小的多，并且解析的速度更快。因此 Xml 现在用的少了（Ajax 中的 x 即指用 Xml 交换数据）。

## 2.4 JSON 语法（[www.json.org](http://www.json.org)）

### 1) 如何表示一个对象

- { 属性名: 属性值, 属性名: 属性值…… }
- ◆ 注意事项：
  - ❖ 属性名要使用引号括起来。
  - ❖ 属性值如果是字符串，要使用引号括起来。
  - ❖ 属性值可以是 string、number、boolean、null、object。

例如：function f1(){//表示一个对象

```
 var obj={'name':'chang','age':22}; alert(obj.name);
 }

function f2(){//表示一个对象
 var obj={'name':'bo','address':{ 'city':'beijing', 'room':'1101' }};
 alert(obj.address.room);
}
```

### ◆ 注意事项：JavaScript 中创建对象的三种方式（也可看 JavaScript 笔记中的第 8 章）：

方式一：使用 Json 语法来创建

```
var p = {'name':'zs','age':22};
```

方式二：使用 Object 来创建

```
var obj = new Object(); obj.name = 'zs'; obj.age = 22;
```

方式三：利用 JavaScript 函数来创建（对象模版）

```
function Person(name,age){
 this.name = name; this.age = age; }
var person1 = new Person("zs",22)
```

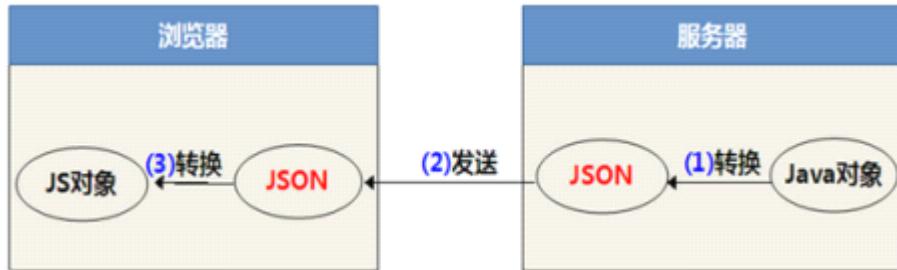
### 2) 如何表示一个对象组成的数组，[ {},{},{} ,…… ]

例如：function f3(){//表示一个对象组成的数组

```
 var arr=[{'name':'chang','age':22},{'name':'bo','age':23}];
 alert(arr[1].name);
}
```

## 2.5 如何使用 JSON 来编写 Ajax 应用程序

LICHOO



step1：Java 对象转换成 JSON 字符串（就是普通字符串用了 JSON 语法而已的字符串）

一般使用 JSON 官方提供的 API (json-lib) 来实现转换 (6 个包：1 主 5 副)。也可用谷歌提供的 API。

例如：股票实体类 Stock，有 name、code、price 三个属性和相应的 get/set 方法

情形一：Java 对象转换成一个 JSON 字符串，使用 JSONObject.fromObject()

```
/** 效果: {"name": "chang", "code": "10086", "price": 12.8} */
public static void test1() {
 Stock s=new Stock(); s.setName("chang"); s.setCode("10086"); s.setPrice(12.8);
 JSONObject obj=JSONObject.fromObject(s); //得到一个 json 对象
 String jsonStr = obj.toString(); System.out.println(jsonStr);
}
```

情形二：Java 对象组成的集合转换成一个 JSON 字符串，使用 JSONArray.fromObject()

```
/** 效果: [{"name": "chang", "code": "10086", "price": 12.8},
{"name": "chang", "code": "100861", "price": 12.81}, {"name": "chang", "code": "1008611", "price": 12.82}]
*/
public static void test2() {
 List<Stock> stocks=new ArrayList<Stock>();
 Random r=new Random(); DecimalFormat df=new DecimalFormat("#.##");
 for(int i=0;i<3;i++){ Stock s=new Stock(); s.setName("chang"+r.nextInt(10));
 s.setCode("60001"+r.nextInt(10)); double
 price=r.nextInt(100)+r.nextDouble();
 s.setPrice(Double.parseDouble(df.format(price))); stocks.add(s);
 }
 JSONArray array=JSONArray.fromObject(stocks);
 String jsonStr=array.toString(); System.out.println(jsonStr);
}
```

情形三：Java 对象组成的数组转换成一个 JSON 字符串，使用 JSONArray.fromObject()

```
public static void test3() {
 Stock[] stocks = new Stock[3]; Random r=new Random();
 DecimalFormat df=new DecimalFormat("#.##");
 for(int i=0;i<3;i++){ Stock s=new Stock(); s.setName("chang"+r.nextInt(10));
 s.setCode("60001"+r.nextInt(10)); double
 price=r.nextInt(100)+r.nextDouble();
 s.setPrice(Double.parseDouble(df.format(price))); stocks[i]=s;
 }
 JSONArray array=JSONArray.fromObject(stocks);
 String jsonStr=array.toString(); System.out.println(jsonStr);
}
```

step2：JSON 字符串转换成 JavaScript 对象

可以使用一些工具提供的方法，比如 prototype 提供了 evalJSON() 函数，prototype 是

一个 js 文件，里面提供了很多常用的函数，比如：

- 1) \$(id): document.getElementById(id);
- 2) \$F(id): \$(id.value);
- 3) \$(id1,id2,id3...): 分别依据 id1,id2...查找对应的节点，然后放到一个数据里面返回。
- 4) strip(): 除掉字符串两端的空格。trim()函数也有，但有的浏览器不支持。
- 5) evalJSON(): 将 JSON 字符串转换成对应的 JavaScript 对象或者 JavaScript 对象组成的数组。

例如：//将一个 JSON 字符串转成一个 JavaScript 对象

```
function f4(){ var str="{'name':'chang','age':24}"; //alert(typeof str);
 //使用 prototype 框架提供的 evalJSON 函数，将字符串转成一个 JavaScript 对象
 var obj=str.evalJSON(); //别忘记引入.js 文件
 alert(obj.name);
}
//将 JSON 字符串转换成一个 JavaScript 对象组成的数组
function f5(){ var str="[{ 'name':'chang','age':22},{'name':'bo','age':23}]";
 var arr=str.evalJSON();
 alert(arr[1].name);
}
```

## 2.6 案例：股票的实时行情

step1：股票实体类 Stock，有 name、code、price 三个属性和相应的 get/set 方法

step2：ActionServlet 中 service 方法中的 if 判断

```
if(action.equals("/quo")){
/** 模拟生成八只股票的价格信息，然后将这些信息转换成 JSON 字符串，并发送到客户端 */
 List<Stock> stocks=new ArrayList<Stock>();
 Random r=new Random(); DecimalFormat df=new
DecimalFormat("#.##");
 for(int i=0;i<8;i++){ Stock s=new Stock();
s.setName("chang"+r.nextInt(30));
s.setCode("6000"+r.nextInt(30)); double
price=r.nextInt(100)+r.nextDouble();
s.setPrice(Double.parseDouble(df.format(price))); stocks.add(s); }
 JSONArray array=JSONArray.fromObject(stocks); //将 Java 数组转成 JSON 字符串
 String jsonStr=array.toString(); System.out.println(jsonStr);
 out.println(jsonStr); //将 JSON 字符串发送到客户端
 }
```

step3：stock.jsp 页面表格（表格样式忽略）

```
<body onload="f1();">
<div id="d1"><div id="d2">股票实时行情</div>
<div id="d3"><table cellpadding="0" cellspacing="0" width="100%">
 <thead><tr><td>股 票 名 称 </td><td>股 票 代 码 </td><td>股 票 价 格
</td></tr></thead>
 <tbody id="tb1"></tbody></table></div>
</div></body>
```

step4: JavaScript 代码

```
function f1(){ setInterval(quoto,3000); }
function quoto(){ var xhr = getxhr(); xhr.open('get','quoto.do',true);
 xhr.onreadystatechange = function(){
 if(xhr.readyState == 4){ var txt = xhr.responseText;
 var arr = txt.evalJSON(); //将 json 字符串转换成 javascript 对象组成的数组
 var html = ""; //将数组中的数据取出来，添加到 tbody
 for(i=0;i<arr.length;i++){ html += '<tr><td>' + arr[i].name + '</td><td>' +
 arr[i].code + '</td><td>' + arr[i].price + '</td></tr>';
 }
 $('#tb1').innerHTML = html;
 }
 xhr.send(null);
 }
}
```

- ◆ 注意事项：innerHTML 属性对于 IE 浏览器只能对 td 赋值！对表格 table 里的其他节点，如 tr、tbody、thead、caption 都不能赋值，只能用它去读（兼容性问题）。

## 2.7 案例：显示热卖的前 3 个商品

step1: 建表，实体类 Sale 有属性 name 和 qty，以及相应 get/set 方法。DBUtil 类此处省略

step2: SaleDAO

```
public List<Sale> limit(int top) throws Exception{
 List<Sale> list=new ArrayList<Sale>(); Connection conn=DBUtil.getConnection();
 PreparedStatement prep=conn.prepareStatement(
 "select * from chang_sale order by qty desc limit ?");
 prep.setInt(1,top); ResultSet rs=prep.executeQuery(); Sale sale=null;
 while(rs.next()){ sale=new Sale(); sale.setName(rs.getString("name"));
 sale.setQty(rs.getInt("qty")); list.add(sale);
 }
 DBUtil.close(conn); return list;
}
```

step3: ActionServlet 中 service 方法中的 if 判断

```
if(action.equals("/limit")){
 int top=Integer.parseInt(request.getParameter("qty"));
 SaleDAO dao=new SaleDAO(); List<Sale> sales=new ArrayList<Sale>();
 try { sales=dao.limit(top); JSONArray array=JSONArray.fromObject(sales);
 String jsonStr=array.toString(); out.println(jsonStr);
 } catch (Exception e) { e.printStackTrace(); }
}
```

step4: JavaScript 代码 (jsp 页面表格与 2.6 案例 step3 类似，此处忽略)

```
function f1(){ setInterval(quoto,3000); }
function quoto(){ var xhr = getxhr(); var top=5;
 xhr.open('get','limit.do?qty='+top,true);
 xhr.onreadystatechange = function(){
 if(xhr.readyState == 4){ var txt = xhr.responseText;
 var arr = txt.evalJSON(); var html = "";
 for(i=0;i<arr.length;i++){ html += '<tr><td>' + arr[i].name +
 '</td><td>' + arr[i].qty + '</td></tr>';
 }
 $('#tb1').innerHTML = html;
 }
 xhr.send(null);
 }
}
```

## 2.8 同步请求

### 1) 什么是同步请求

Ajax 对象在向服务器发送请求时，浏览器会锁定当前页面，用户不能够对当前页面做任何的操作。

### 2) 如何发送同步请求

`open(请求方式, 请求地址, false)`

3) 优先使用异步，因为同步会影响性能，当服务器端处理比较慢的时候，浏览器会锁定当前页面（“假死”）。

4) 只有当客户端需要等待服务器的响应之后，才能继续向下执行时，应该使用同步。

◆ 注意事项：Firefox 的某些版本（低版本，如 3、4）对于同步的支持比较特殊：

不能使用 `xhr.onreadystatechange` 来绑定一个事件处理函数，而应该在 `send` 方法执行之后，才调用 `xhr.responseText` 方法来获得服务器返回的数据。例如：

```
function getTyoe() {//返回浏览器类型
 if(navigator.userAgent.indexOf('Firefox')!=-1){ return 'firefox';
 } else{ return 'other'; }//其他浏览器
}
function quofo(){
 if(getType() != 'firefox'){
 xhr.onreadystatechange=function(){
 if(xhr.readyState == 4){ var txt = xhr.responseText; }; }
 xhr.send('username=' + $F('username'));
 if(getType() == 'firefox'){ var txt = xhr.responseText; }
 }
}
```

## 2.9 案例：修改 1.8 案例 step1 中的 JS 代码（使用同步请求）

step1：form 表单增加属性 `onsubmit="return beforesubmit();"`

step2：ActionServlet 中 service 方法中的 if 判断

```
if(action.equals("/check_username")){
 String username=request.getParameter("username");
 if(username.equals("chang")){
 out.print("error");//不能用 println 服务器会把换行也返回，则永远不会匹配成功
 } else{ out.print("ok"); }
} else if(action.equals("/check_number")){
 String number1=request.getParameter("number");
 HttpSession session=request.getSession();
 String number2=(String)session.getAttribute("number");
 if(number1.equalsIgnoreCase(number2)){ out.print("ok");
 } else{ out.print("error"); }
}
```

step3：增加 JS 代码

```
function check_username() {//step1：检查用户名是否为空
 $('#username_msg').innerHTML="";//先清空之前的提示信息
 if($F('username').strip().length==0){
 $('#username_msg').innerHTML='用户名不能为空';
 return false;
 }
 var flag=false;//step2：检查用户名是否被占用
}
```

```

var xhr=getXhr(); xhr.open('post','check_username.do',false);
xhr.setRequestHeader('content-type','application/x-www-form-urlencoded');
xhr.onreadystatechange=function(){
 if(xhr.readyState==4){ var txt=xhr.responseText;
 if(txt=='ok'){//注意服务器端 println 问题
 $('username_msg').innerHTML='用户名可以使用'; flag=true;
 }else{ $('username_msg').innerHTML='用户名被占用'; flag=false; }
 }
 xhr.send('username='+$F('username'));
 //如果是同步请求，浏览器不会执行以下的代码，而是等待服务器响应回来，在此期间，浏览器会锁定当前页面
 return flag;
}
function check_number(){//检查验证码
 $('number_msg').innerHTML="";
 if($F('number').strip().length==0){
 $('number_msg').innerHTML='验证码不能为空'; return false; }
 var flag=false; var xhr=getXhr();
 xhr.open('get','check_number.do?number='+$F('number'),false);
 xhr.onreadystatechange=function(){
 if(xhr.readyState==4){
 var txt=xhr.responseText;
 if(txt=='ok'){//注意服务器端 println 问题
 $('number_msg').innerHTML='验证码正确'; flag=true;
 }else{ $('number_msg').innerHTML='验证码错误'; flag=false; }
 }
 xhr.send(null); return flag;
 }
}
function beforesubmit(){//提交之前先检查
 var flag=check_username() && check_name() && check_number();
 return flag;
}

```

# 12 jQuery 学习笔记

## 一百一十、jQuery 基础

### 1.1 jQuery 的特点

- 1) jQuery 是一种框架，对于浏览器的兼容问题，95%不用再去考虑了。
- 2) jQuery 利用选择器（借鉴了 CSS 选择器的语法）查找要操作的节点（DOM 对象），然后将这些节点封装成一个 jQuery 对象（封装的目的有两个：①是为了兼容不同的浏览器，②也为了简化代码）。通过调用 jQuery 对象的方法或者属性来实现对底层的 DOM 对象的操作。
- 3) jQuery 特点简单概括就是：选择器 + 调方法。

## 1.2 jQuery 编程的步骤

step1：引入 jQuery 框架 ([www.jquery.org](http://www.jquery.org) 下载)， min 为去掉所有格式的压缩版

```
<script language="javascript" src="js/jquery-1.4.1.min.js"></script>
```

step2：使用选择器查找要操作的节点（该节点会被封装成一个 jQuery 对象，并返回）

```
var $obj=$('#d1');//ID 选择器，查找的节点 ID 为 d1
```

step3：调用 jQuery 对象的方法或者属性

```
$obj.css('font-size','60px');//调用 jQuery 的 css()方法
```

- ◆ 注意事项：jQuery 是一个大的匿名函数，且内部有很多函数（类似 Java 中的内部类），它的大部分函数返回对象都是 jQuery 对象（它自己），所以可以继续“.”，例如：

```
function f1(){
 var $obj=$('#d1');//为了强调返回的是 jQuery 对象，命名习惯用$开头来声明变量
 $obj.css('font-size','60px').css('font-style','italic');
}
```

## 1.3 jQuery 对象与 DOM 对象如何相互转换

1) dom 对象如何转化为 jQuery 对象

使用函数：\$(dom 对象)即可，例如：

```
function f2(){
 var obj=document.getElementById('d1');
 var $obj=$(obj);//将 dom 节点封装成 jQuery 对象
 $obj.html('hello java');
}
```

2) jQuery 对象如何转化为 dom 对象

方式一：\$obj.get(0); 方式二：\$obj.get()[0];

```
function f3(){
 var $obj=$('#d1'); //方式一： var obj=$obj.get(0);
 var obj=$obj.get()[0];//方式二 obj.innerHTML='hello perl';
}
```

## 1.4 如何同时使用 prototype 和 jQuery

step1：先导入 prototype.js，再导入 jQuery.js

step2：将 jQuery 的\$函数换一个名字：var \$a=jQuery.noConflict();//注意大小写

- ◆ 注意事项：函数名就是一个变量，指向函数对象，例如：

```
<script language="javascript" src="js/prototype-1.6.0.3.js"></script>
<script language="javascript" src="js/jquery-1.4.3.js"></script>
function f1(){//无效 var obj=$('#d1'); }
//无效是因为 jQuery 是后引入的，所以 prototype 被 jQuery 替换
function f1(){//为了避免冲突，可以将 jQuery 的$函数换一个名字$a
 var $a=jQuery.noConflict();
 obj.innerHTML='hello prototype'; $a('#d1').html('hello jQuery');
}
```

## 1.5 EL 表达式和 jQuery 函数的区别

1) \${}：EL 表达式，在服务器端运行（JSTL 标签库也在服务器端运行，EL 和 JSTL 标签库本质是 Java 代码）。

2) \$(): jQuery 函数，在浏览器中运行（JavaScript 也在浏览器中运行）。

# 一百一十一、选择器

LICHOO

## 2.1 什么是选择器

jQuery 模仿 CSS 选择器的语法提供了一种用来方便查找要操作的节点的语法规则。

## 2.2 基本选择器

- 1) #id: ID 选择器, 如: \$('#d1').css('color','red');//编号 1 变
- 2) .class: 类选择器, 如: \$('.s1').css('font-size','60px');//编号 2 和 3 变
- 3) element: 元素选择器, 如: \$('div').css('font-size','60px');//编号 1 和 2 变
- 4) selector1,selector2...selectorn: 选择器合并, 如: \$('#d1,p').css('font-size','60px');//编号 1 和 3 变
- 5) \*: 所有选择器, 如: \$('\*').css('font-size','60px');
- 6) 案例:

```
<div id="d1">hello jquery</div><!-- 1 -->
<div class="s1">hello java</div><!-- 2 -->
<p class="s1">hello perl</p><!-- 3 -->
<input type="button" value="基本选择器的使用" onclick="f1();"/>
```

- ◆ 注意事项: 当 jQuery 选择器查找到了多个 DOM 节点, 则仍然是封装成“一个”jQuery 对象, 在调用 jQuery 对象的属性或者方法时, 默认情况下, 会作用于底层所有的 DOM 节点之上。如: \$('.s1').css('font-size','60px'); 则编号 2 和 3 都变。

## 2.3 层次选择器

- 1) select1 select2: 所有后代 (要符合 select2 的要求)。

例如: \$('#d1 div').css('font-size','60px');//d2d3d4d5 变

- 2) select1>select2: 只考虑子节点 (要符合 select2 的要求), 孙子不管~

例如: \$('#d1>div').css('font-size','60px');//d2d3d5 变

- 3) select1+select2: 下一个兄弟 (要符合 select2 的要求), 儿子不管~

例如: \$('#d3+div').css('font-size','60px');//d5 变, d2 不管

- 4) select1~select2: 下面所有的兄弟 (要符合 select2 的要求), 上面的兄弟不管~兄弟中的儿子也不管~

例如: \$('#d2~div').css('background-color','yellow');//d3d5 变

- 5) 案例:

```
<div id="d1">
 <div id="d2">hello 1</div>
 <div id="d3" style="width:200px;height:200px;background-color:red;">
 <div id="d4" style="width:150px;height:150px;background-color:silver;">hello
2</div>
 </div>
 <div id="d5">hello 3</div>
</div>
<input type="button" value="层次选择器的使用" onclick="f1();"/>
```

## 2.4 基本过滤选择器

- 1) :first: 第一行。 2) :last: 最后一行。 3) :not(selector): 把满足要求的选择器排除在外。
- 4) :even: 偶数行, 下标从 0 开始。 5) :odd: 奇数行, 下标从 0 开始。
- 6) :eq(index): 等于下标的元素, 下标从 0 开始。
- 7) :gt(index): 大于下标的元素, 下标从 0 开始。
- 8) :lt(index): 小于下标的元素, 下标从 0 开始。
- ◆ 注意事项: 过滤器前是没有空格的。是 xx:first 而不是 xx :first。
- 9) 案例: step1: 页面表格

```
<table id="t1" border="1" width="60%" cellpadding="0" cellspacing="0">
<thead><tr><td>姓名</td><td>年龄</td></tr></thead>
<tbody><tr><td>岳飞</td><td>33</td></tr>
<tr id="tr2"><td>赵构</td><td>32</td></tr>
<tr><td>韩世忠</td><td>31</td></tr>
<tr><td>梁红玉</td><td>22</td></tr></tbody></table>
<input type="button" value="点这儿" onclick="f1();"/>
```

step2: jQuery 代码

```
<script language="javascript" src="../js/jquery-1.4.3.js"></script>
<!-- 相对路径, ../表示向上跳一级 -->
<script language="javascript" type="text/javascript">
function f1(){
 $('#t1 tr:first').css('background-color','#cccccc');
 $('#t1 tr:last').css('background-color','#cccccc');
 $('tbody tr:even').css('background-color','#fff8dc');
 $('tbody tr:odd').css('background-color','yellow'); }

function f2(){ $('tbody tr:even').css('font-style','italic').css('font-size','50px'); }
function f3(){ $('tbody tr:odd').css('font-size','50px'); }
function f4(){ $('tbody tr:even td:even').css('background-color','red'); }
 //过滤器前（“:”前）是没有空格的
}
</script></pre>

```

## 2.5 内容过滤选择器

- 1) :contains(text): 匹配包含给定文本的元素。
- 2) :empty: 匹配所有不包含子元素或者文本的空元素。
- 3) :has(selector): 匹配含有选择器所匹配的元素的元素。
- 4) :parent: 匹配含有子元素或者文本的元素 (与 empty 正好相反)。
- 5) 案例: step1: 页面

```
<div>吃饭了吗? </div><div></div>
<div><p>一会要下课了</p></div><input type="button" value="点这儿" onclick="f1();"/>
```

step2: jQuery 代码

```
<script language="javascript" src="../js/jquery-1.4.3.js"></script>
<script language="javascript" type="text/javascript">
function f1(){ $('div:contains(吃饭)').css('font-size','50px'); }
```

```

function f2(){//当参数比较多，采用对象的方式传递
 $('div:empty').css({'width':'400px','height':'80px','border':'2px solid red'});
}
function f3(){ $('div:has(p)').css('font-size','80px');}
function f4(){ $('div:parent').css('border','2px solid blue');}
}
</script>

```

## 2.6 可见性过滤选择器

1) :hidden 匹配所有不可见元素，或者 type 为 hidden 的元素。

2) :visible 匹配所有的可见元素。

3) 案例：step1：页面

```

<div>hell jjQuery</div><div style="display:none;">hello java</div>
<input type="button" value="点这儿" onclick="f1();" />

```

step2：jQuery 代码

```

<script language="javascript" src="../js/jquery-1.4.3.js"></script>
<script language="javascript" type="text/javascript">
 function f1(){ $('div:hidden').css('display','block');//或$('div:hidden').show('slow');
 $('div:hidden').show(500); //毫秒
 }
 function f2(){ $('div:visible').hide(800);
 }
</script>

```

## 2.7 属性过滤选择器

1) [attribute]: 有某个属性的元素。

例如： \$('div[id]').css('font-size','60px');//div 中有 id 属性的元素，编号 1 变

2) [attribute=value]: 某个属性的值与指定的值相同的元素。

例如： \$('div[id=d1]').css('font-size','60px');//编号 1 变

3) [attribute!=value]: 某个属性的值与指定的值不相同的元素。

例如： \$('div[id!=d1]').css('font-size','60px');//编号 2 变

4) 案例：

```

<div id="d1">hell jjQuery</div><!-- 1 --> <div>hello java</div><!-- 2 -->
<input type="button" value="点这儿" onclick="f1();" />

```

## 2.8 子元素过滤选择器

1) :nth-child(index/even/odd): 对符合条件的每个节点的子节点作相同操作。

例如： \$('ul li:nth-child(2)').css('font-size','60px');//item2 和 item22 一起变

2) 案例：

```

item1item2item3
item11item22item33
<input type="button" value="点这儿" onclick="f1();" />

```

◆ 注意事项：

❖ 子元素过滤器中 index 从 1 开始。

❖ 基本过滤器中 eq 的 index 从 0 开始，如上例中： \$('ul li:eq(1)').css('font-size', '60px'); 则为 item2 变化。

## 2.9 表单对象属性过滤选择器

1) :enabled: 没有被禁用。

例如: `$('#form1 input:enabled').attr('disabled',true);`//设置属性，可见元素为不可见

2) :disabled: 被禁用。

例如: `$('#form1 input:disabled').css('border','1px dotted red');`//设置不可见元素的样式

`$('#form1 input:disabled').attr('disabled',false);`//设置属性，不可见元素为可见

3) :checked: 单选框、多选框中被选中的选项。

例如: `alert($('#form2 input:checked').val());`//把值输出，类似于 value 属性

4) :selected: 下拉列表中被选中的选项。

例如: `alert($('#form3 option:selected').val());`

5) 案例:

```
<form id="form1">username:<input name="username" />

 name:<input name="name" disabled="disabled"/>
</form>
<form id="form2">
 爱好: 做饭<input type="checkbox" name="interest" value="cooking"/>
 钓鱼<input type="checkbox" name="interest" value="fishing" checked="checked"/>
 足球<input type="checkbox" name="interest" value="football"/></form>
<form id="form3">
 <option value="bj">北京</option>
 <option value="cs" selected="selected">长沙</option>
 <option value="wh">武汉</option></form>
 <input type="button" value="点这儿" onclick="f1();"/>
```

## 2.10 表单选择器

- |                      |                   |                    |
|----------------------|-------------------|--------------------|
| 1) :input: input 元素。 | 2) :text: 文本框。    | 3) :password: 密码框。 |
| 4) :radio: 单选。       | 5) :checkbox: 多选。 | 6) :submit: 提交按钮。  |
| 7) :image: 图片。       | 8) :reset: 重置按钮。  | 9) :button: 普通按钮。  |
| 10) :file: 文件。       | 11) :hidden: 隐藏域。 |                    |

# 一百一十二、DOM 操作

LICHOO

## 3.1 查询

利用选择器找到要操作的节点之后，获得节点的值、属性值、文本以及 html 内容。

1) `html()`: html 内容，如：`alert($('#d1').html())`，相当于 `innerHTML` 属性，下例中也会把 `span` 输出（输出标记中的所有内容），即`<span>hello jQuery</span>`。

2) `text()`: 文本，如：`alert($('#d1').text())`，相当于 `innerText` 属性，由于有兼容性问题，所以没讲。只输出文本内容 `hello jQuery`（标记中的文本内容）。

3) `val()`: 节点的值，如：`alert($('#username').val())`，结果为文本框中输入的值。

4) `attr()`: 属性值，如：`alert($('#d1').attr('id'))`，结果为 `d1`。

◆ 注意事项：此外，这几个方法也可以用来修改节点的内容、值、文本内容、属性值。如：

```
$('#d1').html('hello java'); $('#username').val('chang');
$('#d1').attr('style','color:red;font-size:50px;');
```

5) 案例：

```
<div id="d1">hello jQuery</div>
username: <input name="username" id="username"/>

<input type="button" value="点这儿" onclick="f1();"/>
```

## 3.2 创建

`$(html);`//直接写 html 语句即可，如：`var $obj=$('<div>常</div>');`

## 3.3 插入节点

1) `append()`: 向每个匹配的元素内部最后追加内容（添加的元素作为最后一个孩子）

```
例如： var $obj=$('<div>抗金英雄</div>'); $('body').append($obj);
```

2) `prepend()`: 向每个匹配的元素内部最前添加内容（添加的元素作为第一个孩子）

```
例如： var $obj=$('<div>抗金英雄</div>'); $('body').prepend($obj);
```

3) `after()`: 向每个匹配的元素之后插入内容（在该元素之后添加兄弟节点）

```
例如： $('ul').after('<p>hello</p>');
```

4) `before()`: 向每个匹配的元素之前插入内容（在该元素之前添加兄弟节点）

```
例如： $('ul').before('<p>hello</p>');
```

◆ 注意事项：都可以简化为：`($('body').append/prepend/after/before('<div>抗金英雄</div>');`

5) 案例：

```
岳飞是谁？
item1item2item3
```

## 3.4 删除节点

1) `remove()`: 删除节点，如：`$('#ul li: eq(1)').remove();`

2) `remove(selector)`: 删除满足 selector 的节点，如：`($('#ul li').remove('#l2'));`

3) `empty()`: 清空节点，相当于 `innerHTML=""`，如：`($('#ul li: eq(1)').empty());`

4) 案例：

```
item1<li id="l2">item2item3
<input type="button" value="点这儿" onclick="f1();"/>
```

## 3.5 如何将 JavaScript 代码与 HTML 分开

### 1) 为何要分开?

为了使行为和数据分开。

### 2) 问题分析

```
<script language="javascript">
 var obj=document.getElementById("d1");
 obj.onclick=function(){ this.innerHTML='hello java'; }
</script>
```

此处的 `var obj=document.getElementById("d1");` 没有写在一个函数里（即直接在 `script` 标签中写的），再由于浏览器是逐行解析，那么 `obj` 将是 `null`，因为 DOM 树还没有生成。

### 3) 当引入自己写的 JS 文件时，JS 代码写

```
window.onload=function(){//这里的 JS 代码会在页面加载完成之后执行。
```

```
 var obj=document.getElementById("d1");
 obj.onclick=function(){ this.innerHTML='hello java'; };
```

### 4) 当引入 jQuery 框架时，JS 代码：

① 使用如下主结构

```
$(function(){ //这的 JS 代码会在页面加载完成之后执行。});
```

② 在主结构中添加点击事件使用如下结构（即不在 html 页面添加 `onclick`）

```
$(function(){//主结构，当页面加载完毕后会执行这里的代码
 $('#d1').click(function(){//现在是个 jQuery 对象，不是 dom 对象，
 //所以不能用$('#d1').onclick，要用 jQuery 的函数
 //this 代表绑定了该事件的 dom 对象，this.innerHTML='hello java dom';//方式一
 $(this).html('hello java jquery');//方式二
 });
});
```

③ 上述做法的另一个好处是：可以不用修改代码，直接把代码放入.js 文件中

## 3.6 复制节点

1) `clone()`: 复制节点（不复制行为）。

2) `clone(true)`: 使复制的节点也具有行为（将事件处理代码一块复制）。

3) 案例：step1：页面

```
item1item2item3
<input type="button" value="点这儿" id="b1"/>
```

step2：jQuery 代码

```
<script language="javascript" type="text/javascript">
 $(function(){ $('#ul li:eq(2)').click(function(){//给 item2 动态绑定一个行为
 $(this).css('font-size', '60px'); });
 $('#b1').click(function(){//给 button 动态绑定一个行为
 var $obj=$('#ul li:eq(2)').clone(true);//连同行为一起复制
 $('#ul').append($obj); //追加为 ul 元素的最后一个节点 });
 });
</script>
```

## 3.7 属性

1) `attr('attrName')`: 读取属性。

2) `attr('attrName', 'value')`: 设置一个属性。

3) attr({"attrName1":"value1","attrName2":"value2"}): 设置多个属性。

例如: \$("img").attr({ src: "test.jpg", alt: "Test Image" });

◆ 注意事项:

- ❖ 此处属性名可不用引号（单引或双引），但属性值必须用引号！
- ❖ 不要把样式当属性了。

4) removeAttr('attrName'): 删除属性。

## 3.8 样式操作

1) attr('class',"或者 attr('style',''): 读取和设置。

例如: 读取样式: alert(\$("#d1").attr('class'));

设置样式: \$('#d1').attr('class','s1'); 或 \$('#d1').attr('style','color:red;font-style:italic;');

2) addClass(): 追加。

例如: \$('#d1').addClass('s1 s2');//追加 s1 和 s2 两种样式

3) removeClass(): 移除。

例如: \$('#d1').removeClass('s1');//移除样式 s1

4) removeClass('s1 s2 ...sn'): 移除多个样式

例如: \$('#d1').removeClass('s1 s2');//移除样式 s1 和 s2

5) removeClass(): 删除所有样式。

6) toggleClass(): 样式来回切换，有该样式就删除，没有就添加。

例如: \$('#d1').toggleClass('s3');//样式 s3 一会有一会没（来回切换）

7) hasClass(): 是否有某个样式。

例如: alert(\$('#d1').hasClass('s3'));//返回值 true 或 false

8) css(): 只能读取 style 样式里某个属性的值。

◆ 注意事项: 无法读取某个样式类

例如: <div id="d1" style="font-size:60px;" class="s3">hello jQuery</div>, 则

alert(\$("#d1").css('font-size'));只能读出 60px, 若写 alert(\$("#d1").css('s3'));则内容为空，读不出来。

9) css(""): 设置一个 CSS 样式。

例如: \$('#d1').css('border','1px solid red');

10) css({":":}): 设置多个样式。

例如: \$('#d1').css({'border':'1px solid red','font-size':'50px'});

## 3.9 遍历节点

1) children(): 只考虑子元素（孩子），不考虑其它后代元素（孙子）

2) children(selector): 只考虑子元素（孩子），不考虑其它后代元素（孙子），然后还要满足 selector 的要求（再次过滤）

3) next(): 下一个兄弟

4) next(selector): 下一个兄弟，然后还要满足 selector 的要求（再次过滤）

5) prev(): 上一个兄弟

6) prev(selector): 上一个兄弟，然后还要满足 selector 的要求（再次过滤）

7) siblings(): 兄弟们（上下都算）

8) siblings(selector): 兄弟们（上下都算），然后还要满足 selector 的要求（再次过滤）

9) find(selector): 从某一种节点开始查找所有符合 selector 要求的后代

10) parent(): 父节点

### 11) 案例: step1: 页面

```
<div id="d1">
 <div id="d2">hello 1</div><div id="d3">hello 2</div>
 <div id="d4">hello 3</div><p>hello 4</p></div>
<input type="button" value="点这儿" id="b1"/>
```

### step2: jQuery 代码

```
<script language="javascript" type="text/javascript">
$(function() { $('#b1').click(function() {
 var $obj=$('#d1').children('div');//obj 包含了 4 个节点
 //length 属性: 获得 jQuery 对象包含的 dom 对象的个数
 //alert($obj.length);//4
 //$('#d4').next().css('font-size','40px');//p 变
 //$('#d3').siblings('div').css('font-size','40px');
 //d2 和 d4 变, 若无过滤器, 则 d2、d4、p 都变
 $('#d1').find('p').css('font-size','40px');//p 变
 alert($('#d2').parent().attr('id'));//d1 });
});
```

变

## 3.10 案例: 员工列表 (点击某行该行加亮, 多选框被选中)

### step1: 页面表格

员工信息				
	姓名	薪水	年龄	
	张三	4000	33	
	李四	5000	32	
	王五	3500	31	
	马六	5500	22	

### step2: 样式

```
<style><!-- 注意样式顺序 -->
.row1{background-color:#ff8dc;}
.row2{background-color:#fff0f5;}
.selected{background-color:#ffff00;} </style></pre>
```

### step3: jQuery 代码

```
<script language="javascript" type="text/javascript">
$(function() { $('#tbody tr:even').addClass('row1');
```

```

 $('tbody tr:odd').addClass('row2');
 $('tbody :checkbox:checked').parent().parent().addClass('selected');
 $('tbody tr').click(function(){
 if($(this).hasClass('selected')){
 $(this).removeClass('selected');
 $(this).find(':checkbox').attr('checked',false);
 }else{
 $(this).addClass('selected');
 $(this).find(':checkbox').attr('checked',true);
 }
 });

```

### 3.11 案例：员工列表（点击部门隐藏或显示员工）

step1：页面表格

```

<table width="50%" border="1" cellpadding="0" cellspacing="0">
 <caption style="font-weight:800;">员工信息</caption>
 <thead><tr><th>姓名</th><th>薪水</th><th>年龄</th></tr></thead>
 <tbody>
 <tr class="p" id="p1"><td colspan="3">部门一</td></tr>
 <tr class="c_p1"><td>张三</td><td>20000</td><td>23</td></tr>
 <tr class="c_p1"><td>李四</td><td>22000</td><td>22</td></tr>
 <tr class="c_p1"><td>王五</td><td>14000</td><td>26</td></tr>
 <tr class="c_p1"><td>马六</td><td>15000</td><td>21</td></tr>
 <tr class="p" id="p2"><td colspan="3">部门二</td></tr>
 <tr class="c_p2"><td>张三</td><td>20000</td><td>23</td></tr>
 <tr class="c_p2"><td>李四</td><td>22000</td><td>22</td></tr>
 <tr class="c_p2"><td>王五</td><td>14000</td><td>26</td></tr>
 <tr class="c_p2"><td>马六</td><td>15000</td><td>21</td></tr>
 <tr class="p" id="p3"><td colspan="3">部门三</td></tr>
 <tr class="c_p3"><td>张三</td><td>20000</td><td>23</td></tr>
 <tr class="c_p3"><td>李四</td><td>22000</td><td>22</td></tr>
 <tr class="c_p3"><td>王五</td><td>14000</td><td>26</td></tr>
 <tr class="c_p3"><td>马六</td><td>15000</td><td>21</td></tr>
 </tbody>
</table>

```

step2：样式

```

<style>
 thead{background-color:#cccccc;}
 .p{ background-color:#FFE7BA;}
 .selected{background-color:#FFFF00;} </style>

```

step3：jQuery 代码

```

<script language="javascript" type="text/javascript">
$(function(){
 $('.p').toggle(function(){
 $(this).addClass('selected'); $(this).siblings('.c_').hide(500);
 },function(){
 $(this).removeClass('selected');
 $(this).siblings('.c_').show(500); });
});
```

</script>

LICHCO

## 一百一十三、事件

## 4.1 事件绑定

1) 方式一: bind( type,function(){} )

例如: `$('#d1').bind('click',function() { //正式写法, 常用于解决浏览器兼容性  
    $(this).html('hello java');                   });`

2) 方式二（绑定的简写形式）: click( function(){} );

例如： `$('#d1').click(function(){//简写形式  
$(this).html('hello java')})`

## 4.2 合成事件

1) hover(enter,leave): 模拟光标悬停事件。

例如：\$(function(){ \$('.s1').hover(function(){ \$(this).addClass('s2');//光标进入 },function(){ \$(this).removeClass('s2');//光标离开 });});

2) 不使用合成事件的方式。

```
例如: $('.s1').mouseenter(function(){//鼠标移入
 $(this).addClass('s2');//绑定了 mouseenter 事件的 div });
$('.s1').mouseleave(function(){//鼠标移出
 $(this).removeClass('s2');});
```

3) 案例: 上面两点的样式与页面

```
<style><!-- 样式 -->
 .s1 { width:100px; height:100px; background-color:silver; }
 .s2 { background-color:yellow; } </style>
<div class="s1"></div><!-- 页面 -->
```

- ◆ 注意事项：细节！为何不用 ID 属性，即<div id="s1"></div>，样式选择器改名为#s1{...}。当执行函数时，相当于<div id="s1" class="s2"></div>，由于 ID 样式选择器比 class 样式选择器优先级高，所以不会有移入时背景变色效果！

4) `toggle(function1(){}),function2(){},...)`: 模拟光标连续单击事件。

4.3 事件冒泡 可参考 JavaScript 笔记 7.5

### 1) 什么是事件冒泡

子节点产生的事件，会依次向上抛出给相应的父节点。

## 2) 如何取消事件冒泡

使用 event 对象，`e.cancelBubble=true`:

### 3) 事件对象有何作用

①取消冒泡: e.cancelBubble=true;

```

 例如: <div id="d1" onclick="f2(event);">
 Cilck Me</div>
 function f1(e){ alert('点击了连接'); e.cancelBubble=true;//取消冒泡 }
 function f2(e){ alert('点击了 div'); }

```

②获得光标点击的坐标: 两个属性: e.clientX e.clientY

```

 例如: function f2(e){ alert(e.clientX+","+e.clientY); }

```

③找到事件源 (详见 JavaScript 笔记 7.6): 使用 event 对象: e.target || e.srcElement; 要区分浏览器。

◆ 注意事项: 两种获得事件源对象的方式最新的 Chrome 浏览器都支持。

## 4.4 jQuery 中事件处理

1) 获得事件对象

```

click(function(e){ //e: 对底层的事件对象做了一个封装 });

```

2) 事件对象的属性

- ①event.type: 事件类型
- ②event.target: 返回事件源 (是 DOM 对象)
- ③event.pageX/pageY: 返回点击的坐标

例如: \$(function() {

```

 $('a').click(function(e){ //对两个对象都绑定了 click
 //依据事件对象获得事件源, e 是 jQuery 对象 (封装了底层的事件对象)
 var obj=e.target;//target 属性返回的是一个 dom 对象
 alert(obj.innerHTML);
 alert(e.pageX+","+e.pageY);//通过事件对象获得光标点击的 X、Y 坐标
 alert(e.type);
 });
 Cilck 1Cilck 2

```

3) 停止冒泡: event.stopPropagation()

例如: \$(function() { //e 是 Query 封装后的对象, 不能再用底层的 cancelBubble 属性

```

 $('a').click(function(e){
 alert('点击的是连接');
 e.stopPropagation(); //停止冒泡
 });
 $('div').click(function(e){
 alert('点击的是 div');
 });
 <div id="d1">Cilck Me</div>

```

4) 停止默认行为: event.preventDefault()

◆ 注意事项: 原来的写法为<a href="del.do" onclick="return false"></a>

如: \$(function() {

```

 $('a').click(function(e){
 var flag=confirm('确定删除吗? ');
 if(!flag){ //阻止浏览器的默认行为, 即不再向连接地址发请求
 e.preventDefault();
 }
 });
 删除

```

5) 模拟操作: trigger('click')

```

$(function(){

```

```

 $('#b1').click(function(){
 //模拟用户点击了 username 文本框, 即让 username 文本框产生焦点获得事件
 $('#username').trigger('focus');
 //$('#username').focus(); //简写形式, 模拟获得焦点
 //$('#username').click(); //模拟点击
 });

```

## 4.5 动画

### 1) show()/hide()

①作用：通过同时改变元素的“宽度”和“高度”来实现显示或隐藏。

②用法：show(速度,[回调函数]); (hide 同理)

A.回调函数：整个动画执行完毕之后，会执行该函数。

B.速度：'slow', 'fast', 'normal' 或者使用毫秒数。

### 2) slideUp()/slideDown()

①作用：通过同时改变元素的“高度”来实现显示或隐藏。

②用法：同上。

### 3) 案例：将 4.2 案例 4) 中显示机票价格修改

```
$(function() { $('#a').toggle(function() { $('#d1').slideDown(800); //滑下
}, function() { $('#d1').slideUp(800); //收起 })); })
```

### 4) fadeIn()/fadeOut()

①作用：通过改变元素的不透明度来实现显示或隐藏。0 透明 0.5 半透明 1 不透明

②用法：同上。

### 5) 案例：将 4.2 案例 4) 中显示机票价格修改

```
$(function() { $('#b1').toggle(function() { $('#d1').fadeIn('slow'); //淡入
}, function() { $('#d1').fadeOut('slow'); //淡出 })); })
```

### 6) 自定义动画 animate(params,speed,[callback])

①params：是一个 JavaScript 对象，描述动画执行结束之后元素的样式，比如：

{'height':'200px'}

②speed：速度，只能用毫秒数。 ③callback：回调函数。

### 7) 案例：step1：页面

```
<div id="d1"></div>
```

step2：样式

```
#d1 {width:100px; height:100px; background-color:#fff8cd; border:1px solid red;
position:absolute; /*必须定义 position 属性，否则动不起来*/ }
```

step3：jQuery 代码

```
$(function() { $('#d1').click(function() {
//$(this).animate({ 'left':'500px','top':'300px'},5000); //走最短距离，往右下走
$(this).animate({ 'left':'500px'},5000); //动画队列先执行完第一个，再执行第二个
$(this).animate({ 'top':'300px'},3000).fadeOut('slow'); })); })
```

## 4.6 类数组的操作

jQuery 对象里面可能包含多个 DOM 对象，所谓类数组，指的就是这些 DOM 对象。

1) each(function(i))：循环遍历每一个元素，this 代表被遍历的 DOM 对象，\$(this) 代表被迭代的 jQuery 对象，i 代表正在被遍历的那个对象的下标。下标从 0 开始。

例如：\$('#b1').click(function() { var \$obj=\$('#ul li');

\$obj.each(function(i){ //i：表示正在被遍历的那个节点的下标，下标从 0 开始

if(i==0){ //this：表示正在被遍历的那个 DOM 对象

\$(this).css('font-size','50px');

}else if(i==1){ \$(this).css('font-style','italic');

```
 } else{ $(this).css('color','red'); });});
```

2) eq(index): 返回 index 位置处的 jQuery 对象。

例如: \$('#b1').click(function(){ var \$obj=\$('#ul li'); //eq: 取下标对应处的 DOM 对象, 然后将这个 DOM 对象封装成一个 jQuery 对象 var \$o=\$obj.eq(1); \$o.css('font-size','50px');//item2 变 });};

3) index(obj): 返回下标, 其中 obj 可以是 DOM 对象或者 jQuery 对象。

例如: \$('#b1').click(function(){ var \$obj=\$('#ul li'); var \$o=\$obj.eq(1);//item2 var index=\$obj.index(\$o);//index: 返回节点在类数组中的下标 alert(index);//1 });};

4) length 属性: 获得 jQuery 对象包含的 DOM 对象的个数。

例如: \$('#b1').click(function(){ var \$obj=\$('#ul li'); alert(\$obj.length);//3 });};

5) get(): 返回 DOM 对象组成的数组。

例如: \$('#b1').click(function(){ var \$obj=\$('#ul li'); var arr=\$obj.get(); arr[2].innerHTML='hello');//item3 变 });};

6) get(index): 返回第 index 个 DOM 对象。

例如: \$('#b1').click(function(){ var \$obj=\$('#ul li'); var obj=\$obj.get(1);//get(index): 取 index 对应处的 DOM 对象 obj.innerHTML='hello jQuery',//item2 变 });};

7) 案例: 页面

```
item1item2item3
<input type="button" value="点这儿" id="b1"/>
```

## 4.7 案例：滚动广告条

step1: 页面无序列表

```
<div id="d1">
 <ul id="adv">

 <ul id="num">12345</div>
```

step2: 样式

```
*{ margin:0px; padding:0px; }
#d1{ border: 1px solid #aaaaaa; margin-left:200px; margin-top:40px; width:548px;
 height:177px; overflow:hidden; position:relative; }

#adv,#num{ position:absolute; }

ul li{ list-style:none; display:inline; }

ul img{ width:548px; height:177px; display:block; }

#num{ right:5px; bottom:5px; }

#num li{ float: left; color: #FF7300; text-align: center; line-height: 16px;
 width: 16px; height: 16px; font-family: Arial; font-size: 12px; cursor: pointer;

 overflow: hidden; margin: 3px 1px; border: 1px solid #FF7300; }

.on{ line-height: 21px; width: 21px; height: 21px; font-size: 16px; margin: 0 1px; }
```

```
border: 0; background-color:red; font-weight: bold; }
```

step3: jQuery 代码

```
$(function(){
 $('#num li').mouseenter(function(){
 var index = $('#num li').index(this);//需要知道光标指向的是哪一个 li
 showImage(index);//依据 index(下标), 滚动图片
 }).eq(0).mouseenter();
 var i = 0; var taskId;//光标进入, 停止滚动图片, 光标离开, 开始滚动图片
 $('#d1').hover(function(){ clearInterval(taskId);//光标进入, 清除定时任务
 },function(){ taskId = setInterval(function(){//光标离开, 开始定时任务
 showImage(i); i++;
 if(i == 5){ i = 0; } },2000);
 }).mouseleave();
 //滚动图片函数, stop(true): 在执行当前动画之前, 先清空之前累积的动画
 function showImage(index){
 $('#adv').stop(true).animate({ 'top':-index * 177},1000);
 $('#num li').eq(index).addClass('on').siblings().removeClass('on');//加亮光标指定的 li }
})
```

# 一百一十四、jQuery 对 Ajax 编程的支持

LICHOO

## 5.1 load()方法

1) 作用：将服务器返回的数据直接插入到符合要求的节点之上，相当于：  
obj.innerHTML=服务器返回的数据。

2) 用法：\$obj.load(url,[data]);

①url：请求地址，服务器上的某个组件的地址。

②data：请求参数，有两种形式：

A. 请求字符串形式："name=chang&age=23"

B. 对象形式：{'name':'chang','age':23}

◆ 注意事项：load 方法：当没有请求参数时，会使用 get 方式向服务器发请求，如果有请求参数，会使用 post 方式向服务器发请求。也不支持异步！

## 5.2 案例：显示机票价格

step1：ActionServlet 中 service 方法中 if 判断

```
if(action.equals("/priceInfo")){
 String airline=request.getParameter("airline");
 if(airline.equals("CA1000")){//简单操作，不读取数据库了
 out.println("头等舱：¥ 2400
经济舱：¥ 1200");
 }else{
 out.println("头等舱：¥ 2200
经济舱：¥ 1000");
 }
}
```

step2：页面表格

航班号	机型	起飞时间	到达时间
&ampnbsp	经济舱价格	&ampnbsp	&ampnbsp
CA1000	<a href="#">波音 777</a>	8:00am	
10:00am	<a href="javascript:;">显示所有票价</a>	<div>&amp;ampnbsp</div> <dt>¥ 1200</dt>	
<input type="button" value="订票" />			
MU1949	<a href="#">空客 230</a>	18:00am	
20:00am	<a href="javascript:;">显示所有票价</a>	<div>&amp;ampnbsp</div> <dt>¥ 800</dt>	
<input type="button" value="订票" />			

step3：jQuery 代码

```
$(function(){
 $('a.s1').toggle(function(){
 var airline=$(this).parent().siblings().eq(0).text();
 $(this).next().load('priceInfo.do','airline='+airline);
 //或用对象形式$(this).next().load('priceInfo.do',{'airline':airline});
 $(this).html('显示经济舱价格');
 },function(){
 $(this).next().empty();
 $(this).html('显示所有票价');
 });
});
```

## 5.3 \$.get()方法

1) 作用：使用 get 方式向服务器发请求。

2) 用法：\$.get(url,[data],[callback],[type]);

- ①url: 请求地址。 ②data: 请求参数, 有两种形式: 同上。
- ③callback: 是一个回调函数, 格式: function(data,statusText), 其中, data 是服务器返回的数据, statusText 是服务器处理的状态。
- ④type: 服务器返回的数据的类型, 有五种:
  - A.html: 返回的是一个 html 文档。
  - B.text: 返回的是纯文本。
  - C.json: 返回的是 json 字符串。
  - D.xml: 返回的是一个 Xml 文档。
  - E.script: 返回的是一个 javascript 脚本。

## 5.4 \$.post() 方法

- 1) 作用: 使用 post 方式向服务器发请求。 2) 用法: 与\$.get()方法相同

## 5.5 案例: 修改 Ajax 笔记中 2.6 案例: 股票的实时行情

step1: 导入 jQuery 框架, 删除 step4 中 Ajax (JS) 代码

step2: 写 jQuery 代码, 如下:

```

$(function(){
 setInterval(quoto,3000);
});

function quoto(){
 $.get('quoto.do',function(data){ //注意 IE 浏览器的缓存问题
 //$.get()/$.post()/$.ajax()会自动将服务器返回的 JSON 字符串转换成 JavaScript 对象
 $('#tb1').empty(); //先清空之前的内容
 for(i=0;i<data.length;i++){
 $('#tb1').append('<tr><td>' + data[i].name
 +'</td><td>' + data[i].code + '</td><td>' + data[i].price + '</td></tr>');
 }
 },'json'); //ActionServlet 中发送的是 JSON 字符串, 所以此处写 json, 记得导 6 个包,
 //详见 Ajax 笔记 2.5
}

```

## 5.6 \$.ajax() 方法

- 1) 用法: \$.ajax( { ... } ), 参数顺序无所谓。
- 2) 参数如下:
  - ①url(string): 请求地址。 ②type(string): GET/POST。
  - ③data(object/string): 请求参数。 ④dataType(string): 预期服务器返回的数据类型。
  - ⑤ success(function) : 请求成功后调用的回调函数, 有两个参数 function(data,textStatus)
    - , 其中 data 是服务器返回的数据, textStatus 描述状态的字符串。
    - ⑥error(function): 请求失败时调用的函数, 有三个参数 function(xhr,textStatus,errorThrown):
      - A.xhr: 底层的 XMLHttpRequest 对象。
      - B.textStatus: 错误的描述。
      - C.errorThrown: 一般为 null。
    - ⑦async: true (缺省, 异步) /false (同步)。

## 5.7 案例: 搜索栏联想效果 (服务器返回 text)

step1: ActionServlet 中 service 方法中 if 判断

```

if(action.equals("/find")){
 //if(l==1){ throw new ServletException("some error"); } //模拟错误
 String key=request.getParameter("key");
 if(key.equals("小")){
 out.println("小学生作文,小学生,小米,小米 2,小学生守则");
 }
}

```

```

} else if(key.equals("小学")){
 out.println("小学生作文,小学生");
} else if(key.equals("岳")){
 out.println("岳飞,岳不群");
}

```

step2: 样式

```

<style> table { margin-left:400px; margin-top:100px; font-size:20px; }
 .selected{background-color:#fff8cd; } </style>

```

step3: 页面

```

<table cellpadding="0" cellspacing="0">
 <tr><td><input name="key" id="key"/></td>
 <td><input type="button" value="搜索"/></td></tr>
 <tr><td colspan="2"><div id="tips"/></div></td></tr></table>

```

step4: jQuery 代码：火狐某些低版本切换中文输入法时 keyup 事件失效，是浏览器的 BUG

```

<script language="javascript" type="text/javascript">
$(function(){
 $('#key').keyup(function(){
 $.ajax({
 'url':'find.do', 'type':'post', 'data':'key='+$('#key').val(), 'dataType':'text',
 'success':function(data){//data 表示服务器返回的数据，如：岳飞,岳不群
 var arr=data.split(',');//分解服务器返回的数据
 $('#tips').empty();
 for(i=0;i<arr.length;i++){ $('#tips').append(
 "<div class='item'>"+arr[i]+"</div>"); }
 //当光标经过提示项时加亮
 $('.item').mouseenter(function(){
 $(this).addClass('selected').siblings().removeClass('selected'); });
 //当光标点击某个选项时，将该选项的值复制到 key
 $('.item').click(function(){
 $('#key').val($(this).text()); $('#tips').empty(); });
 //success 回调函数的括号
 });//这的 } 是$.ajax 中的括号
 });//这的 } 是keyup 中的括号
 });//这的 } 是$函数中的括号
</script>

```

step5: 为了解决 keyup 失效问题，使用 input 或 propertychange 事件都可，但 IE 浏览器只认识 propertychange 事件，其他浏览器只认识 input 事件，所以再次为了解决兼容性问题，修改如下：

```

<script language="javascript" type="text/javascript">
$(function(){
 var event_type='input';//要判断浏览器类型
 if(navigator.userAgent.indexOf('MSIE')!= -1){//IE 浏览器
 event_type='propertychange';
 }
 $('#key').bind(event_type,fn);
});
function fn(){
 $.ajax({
 'url':'find.do', 'type':'post', 'data':'key='+$('#key').val(), 'dataType':'text',

```

```

'success':function(data){//data 表示服务器返回的数据, 如: 岳飞,岳不群
 var arr=data.split(',');//分解服务器返回的数据
 $('#tips').empty();
 for(i=0,i<arr.length;i++){
 $('#tips').append(
 "<div class='item' style='border:1px solid silver'>" + arr[i] + "</div>");
 }
 //当光标经过提示项时加亮
 $('.item').mouseenter(function(){
 $(this).addClass('selected').siblings().removeClass('selected');
 });
 //当光标点击某个选项时, 将该选项的值复制到 key
 $('.item').click(function(){
 $('#key').val($(this).text());
 $('#tips').empty();
 });
},
'error':function(){
 alert('出错');
}
});//这的 } 是$.ajax 中的括号
}

```

## 5.8 案例：下拉列表（服务器返回 xml 文本）

step1：页面

```

<select style="width:120px;" id="s1">
 <option value="bmw520">宝马 520</option><option value="qq">QQ</option>
 <option value="maiten">迈腾</option>

```

step2：样式

```

<style>
 #d1{ width:300px; height:80px; background-color:#fff8dc; }
</style>

```

step3：jQuery 代码

```

<script type="text/javascript">
$(function(){
 $('#s1').change(function(){
 $('#d1').remove();
 $.ajax({ 'url':'carinfo.do', 'type':'post', 'data':{'name':$('#s1').val()},
 'dataType':'xml',
 'success':function(data,statusText){
 //data 是一个 dom 节点, 指向 xml 文档对应的那棵 dom 树。
 var $obj = $(data);
 $('#d1').after("<div id='d1'></div>");
 $('#d1').html('报价:' + $obj.find('price').text()
 + '
 描述:' + $obj.find('desc').text());
 },
 'error':function(xhr,e1,e2){
 //xhr 表示 XMLHttpRequest 对象, e1,e2 表示具体的错误信息
 alert('系统出错');
 }
 });
 });
});

```

```

});//这的 } 是$.ajax 中的括号
});//这的 } 是 change 中的括号
});//这的 } 是$函数中的括号
</script>

```

step4: ActionServlet 中 service 方法中 if 判断

```

if(path.equals("/carinfo")){
 System.out.println("carInfo..");
 //生成一份 xml 文档，返回给客户端
 response.setContentType("text/xml;charset=utf-8");
 PrintWriter out = response.getWriter();
 StringBuffer sb = new StringBuffer();
 sb.append("<msg>");
 String name = request.getParameter("name");
 if(name.equals("bmw520")){
 sb.append("<price>50</price>");
 sb.append("<desc>还不错</desc>");
 }else if(name.equals("qq")){
 sb.append("<price>5</price>");
 sb.append("<desc>非常不错</desc>");
 }else{
 sb.append("<price>18</price>");
 sb.append("<desc>没开过不知道</desc>"); }
 sb.append("</msg>"); out.println(sb.toString()); out.close(); }

```

step5: 访问 <http://localhost:8080/应用名/carinfo.do?name=bmw520> , 则会返回一颗 Xml 树  
访问 <http://localhost:8080/应用名/carinfo.jsp> 正常显示。

## 5.9 案例：表单验证

1) 用户输入时，动态提示信息

step1: 样式

```

<style> #d1{ width:500px; height:250px; border:1px solid red;
background-color:#fff8dc; margin:20 auto; }</style>

```

step2: 页面

```

<div id="d1">
 <form class="cmxform" id="signupForm">
 <p><label for="username">用 户 名 </label><input id="username" name="username"
/></p>
 <p><label for="password">密 码 </label><input id="password" name="password"
type="password"
/></p>
 <p><label for="confirm_password">确认密 码 </label>
 <input id="confirm_password" name="confirm_password" type="password"
/></p>
 <p><label for="email">邮 箱 </label><input id="email" name="email" /></p>
 <p><label for="agree">是否同意 </label><input type="checkbox" id="agree"
name="agree"
/></p>
 <p><input class="submit" type="submit" value="Submit" /></p>
 </form>
</div>

```

step3: jQuery 代码

```

<script type="text/javascript" src="../js/jquery-1.4.3.js"></script>
<script type="text/javascript" src="../js/jquery.validate.js"></script><!-- 导包 -->
<script type="text/javascript">
$(function(){
 $("#signupForm").validate(
 { rules:
 { username:
 { required: true, minlength: 2 },
 password:
 { required: true, minlength: 5 },
 confirm_password:
 { required: true, minlength: 5, equalTo: "#password" },
 email:
 { required: true, email: true }
 },
 }
);
});
</script>

```

2) 使用自定义错误提示信息, 如动态提示信息为中文

step1: 修改 jQuery 代码

```

<script type="text/javascript">
$(function() {
 $("#signupForm").validate(
 { rules:
 { username:
 { required: true, minlength: 2 },
 password:
 { required: true, minlength: 5 },
 confirm_password:
 { required: true, minlength: 5, equalTo: "#password" },
 email:
 { required: true, email: true }
 },
 messages:
 { username:
 { required: "请输入用户名", minlength: "用户名至少2个字符" },
 password:
 { required: "请输入密码", minlength: "密码不能少于5个字符" },
 confirm_password:
 { required: "请输入确认密码", minlength: "确认密码不能少于5个字符",equalTo: "密码输入不一致" },
 email: "请输入有效邮箱"
 }
 }
);
});
</script>

```

```

 }
);
}
);
</script>

```

3) 给验证信息添加图片，如点击提交，提示信息包括相应提示图片

step1：添加样式

```

span { font-weight: bold; padding-right: 1em; vertical-align: top; }
span.error { background:url("../image/unchecked.gif") no-repeat 0px 0px;
 padding-left: 16px; }
span.success { background:url("../image/checked.gif") no-repeat 0px 0px;
 padding-left: 16px; }

```

step2：修改 jQuery 代码

```

<script type="text/javascript">
$(function(){
 $("#signupForm").validate(
 { rules:
 { username:
 { required: true, minlength: 2 },
 password:
 { required: true, minlength: 5 },
 confirm_password:
 { required: true, minlength: 5, equalTo: "#password" },
 email:
 { required: true, email: true }
 },
 messages:
 { username:
 { required: "请输入用户名", minlength: "用户名至少 2 个字符"
 },
 password:
 { required: "请输入密码", minlength: "密码不能少于 5 个字符"
 },
 confirm_password:
 { required: "请输入确认密码", minlength: "确认密码不能少于 5
 个字符",equalTo: "密码输入不一致"
 },
 email: "请输入有效邮箱"
 },
 errorElement: "span", //用来创建错误提示信息标签
 success: function(label) {//验证成功后执行的回调函数
 //label 指向上面那个错误提示信息标签
 //清空错误提示信息
 label.text(" ").addClass("success");//加上自定义的 success 类
 }
 }
);
});

```

LICHOOL

```

 }
 });
});
</script>

```

LICHOO

## 5.10 jQuery 的自定义方法

1) 语法: `$.fn.funName=function(){...}`

2) 案例:

```

$.fn.red=function(){
 this.css("backgroundColor","red");
}
$("#msg_error").red(); //调用自定义的方法

```

## 5.11 \$.param()方法

1) 语法:

```

var obj={表单的名字:"表单的值"};//组成一个对象
$.param(obj);

```

## 5.12 案例: 自定义方法和\$.param()方法使用(学了 Struts2 再看)

step1: 创建自定义方法

```

$.fn.remote=function(url,errorMsg,errorCtn){
 var b=false; var value=this.val(); var name=this.attr("name");
 var obj={};//做了一个对象
 obj[name]=value;//添加一个键值对
 var params=$.param(obj);//考虑提交是中文的情况，先编码再传
 $.ajax({ url:url,
 date:params,
 type:"post",
 dataType:"json",
 async:false,
 success:function(data){//data 为接收到的 boolean 值变量
 if(data){ $(errorCtn).text(""); b=true;//验证成功
 } else{ $(errorCtn).text(errorMsg); //验证失败
 }
 }
);
 return b;
}

```

step2: JSP 页面调用自定义函数

```

<script type="text/javascript">
$(function(){
 $('#save').click(function(){
 var b=validateCostForm();
 if(b){ $('#costForm').submit(); }
 });
})
function validateCostForm(){
 var b1=false;
 b1=$('#costName').remote("validateName.action","资源名称已被占用",
 $('#msg_costName'));
}

```

```
 return b1 ;
 </script>
```

step3：创建验证用的 Action，并返回一个 boolean 值变量

step4：在 struts-cost.xml 配置文件中，配置 ValueStack 栈顶返回的 boolean 值变量名字

◆ 注意事项：此处，需要学习 Struts2 后，并结合 NetCTOSS 项目，代码略……

# 13 Struts 学习笔记

## 一百一十五、Struts2 概述

### 1.1 为什么要用 Struts

1) JSP 用 HTML 与 Java 代码混用的方式开发，把表现与业务逻辑代码混合在一起给前期开发与后期维护带来巨大的复杂度。

2) 解决办法：把业务逻辑代码从表现层中清晰的分离出来。

3) 2000 年，Craig McClanahan 采用了 MVC 的设计模式开发 Struts 主流的开发技术，大多数公司在使用。

### 1.2 什么是 MVC

#### 1) M-Model 模型

模型（Model）的职责是负责业务逻辑。包含两部分：业务数据和业务处理逻辑。比如实体类、DAO、Service 都属于模型层。

#### 2) V-View 视图

视图（View）的职责是显示界面和用户交互（收集用户信息）。属于视图的类是不包含业务逻辑和控制逻辑的 JSP。

#### 3) C-Controller 控制器

控制器是模型层 M 和视图层 V 之间的桥梁，用于控制流程。比如我们之前项目中写的 ActionServlet。

### 1.3 JSP Model 1 和 JSP Model 2

#### 1) JSP Model 1

最早的 JSP/Servlet 模式被我们称之为 JSP Model 1 (JSP 设计模式 1)，在其中有模型层 M，但是视图层 V 的 JSP 中包含了业务逻辑或控制逻辑，JSP 和 Servlet 紧密耦合。即：JSP (数据的展现和业务流程的控制) + JavaBean (对数据的访问和运算，Model)。

#### 2) JSP Model 2

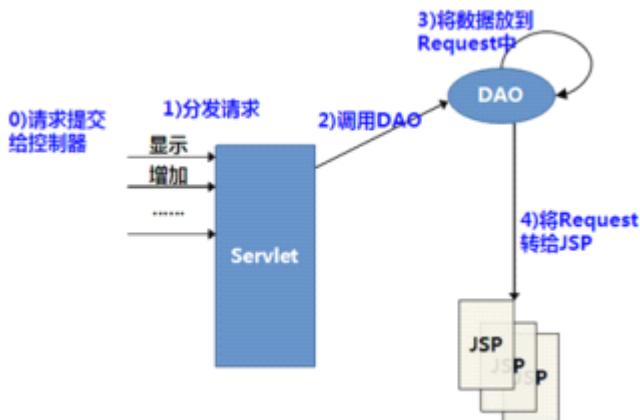
JSP Model 2 (JSP 设计模式 2) 和 JSP Model1 不同之处再与将 JSP 中的业务逻辑和控制逻辑全部剔除，并全部放入控制层 C 中，JSP 仅仅具有显示页面和用户交互的功能。

编程理念：高内聚低耦合。即：

M: Model，用 JavaBean 来做 (对数据的访问和运算)

V: View，用 JSP 来作 (数据的展现)

C: Controller，用 Servlet 来作 (业务流程的控制)



## 1.4 Struts2 发展史

- 1) 最早出现的 Struts1 是一个非常著名的框架，它实现了 MVC 模式。Struts1 简单小巧，其中最成熟的版本是 Struts1.2。
- 2) 之后出现了 WebWork 框架，其实现技术比 Struts1 先进，但影响力不如 Struts1。
- 3) 在框架技术不断发展的过程中，有人在 WebWork 的核心 XWork 的基础上包装了 Struts1（算是两种框架的整合），由此，结合了 Struts1 的影响力和 WebWork 的先进技术，Struts2 诞生了。
- 4) Struts2 不是 Struts1 的升级，它更像是 WebWork 的升级版本。

## 1.5 衡量一个框架的标准

- 1) 健壮性：Struts2.0 不健壮（带 0 的就是实验品），Struts2.1.8 是健壮的（2.1.6 也不健壮，该版本 BUG 较多），3 颗星。
- 2) 易用性：易用性越好，原理则越复杂，4 颗星。
- 3) 扩展性：Struts2，5 颗星。
- 4) 侵入性：即对写代码的要求（如：必须继承某个类才能怎么怎么样……），越低越好。Struts2 低但也有侵入性，4 颗星。

## 1.6 Struts2 使用步骤

step1：创建一个 JavaWeb Project，命名为 struts01（Struts2 只能用在 Web 项目里）

step2：拷贝 Struts2 的核心 Jar 包到 WEB-INF/lib/下，基本功能核心 jar 包 5 个：

- 1) xwork-core-2.1.6.jar：Struts2 核心包，是 WebWork 内核。
- 2) struts-core-2.1.8.jar：Struts2 核心包，是 Struts 框架的“外衣”。
- 3) ognl-2.7.3.jar：用来支持 OGNL 表达式的，类似于 EL 表达式，功能比 EL 表达式强大的多。
- 4) freemarker-2.3.15.jar：freemarker 是比 JSP 更简单好用，功能更加强大的表现层技术，用来替代 JSP 的。在 Struts2 中提倡使用 freemarker 模板，但实际项目中使用 JSP 也很多。
- 5) commons-fileupload.jar：用于实现文件上传功能的 jar 包。

step3：在 web.xml 中配置 Struts2 的前端控制器，Struts2 是用 Filter（过滤器）实现的前端控制器，它只负责根据请求调用不同的 Action

◆ 注意事项：原来的项目是用 Servlet 的方式作为控制器。

web.xml 内容：

```
<?xml version="1.0" encoding="UTF-8"?>
```

LICHOO

```

<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
 http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">
 <!-- 上面的内容是规定必须写的，复制粘贴即可 -->
 <filter><!-- 前端控制器 -->
 <filter-name>Struts2</filter-name>
 <filter-class>
 org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter
 </filter-class>
 </filter>
 <filter-mapping>
 <filter-name>Struts2</filter-name>
 <url-pattern>/*</url-pattern> <!-- /*表示所有的请求都要经过该过滤器 -->
 </filter-mapping>
</web-app>

```

step4: 在 WebRoot 下新建 jsp 文件夹，并在其中创建 nameform.jsp 和 welcome.jsp

1) nameform.jsp

```

<form action="welcome.action" method="post">
 <input name="name" type="text" /><input value="提交" type="submit" />
 ${error}
</form>

```

2) Welcome.jsp

```
<h1>Welcome, ${name}</h1>
```

step5: 在 com.tarena.action 包下创建 WelcomeAction 类

```

private String name; private String error; 各自的 get/set 方法
public String execute() { System.out.println("WelcomeAction.execute()...");
 if(name==null||"".equals(name)){ error="用户名不能为空"; return "fail"; }
 System.out.println("name: " + name); //用于测试
 if ("test".equalsIgnoreCase(name)) { error="不能使用名字 text 登录";
 return "fail";
 }else{ return "success"; }
}

```

step6: 写 Struts2 所需要的控制器配置文件 struts.xml, 非常重要! 该文件写请求与各个 Action (Java 类) 的对应关系, 前端控制器会根据配置信息调用对应的 Action (Java 类) 去处理请求

- ◆ 注意事项:
  - ❖ 在编写时 struts.xml 放在 src 目录中, 而部署后该文件位于 WEB-INF/classes/ 下!
  - ❖ 同样的, .properties 文件也放 src 下, 部署后也在 WEB-INF/classes/ 下!
  - ❖ 程序一启动, struts.xml 就会被加载到内存, 并不是现读的。

struts.xml 内容如下:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC "-//Apache Software Foundation//DTD Struts Configuration
2.1//EN" "http://struts.apache.org/dtds/struts-2.1.dtd">

```

```

<!-- 以上内容是规定必须写的，复制粘贴即可 -->
<struts><!-- struts 的根标签，也是规定，各个标签详解见 1.7 节 -->
 <package name="helloworld" namespace="/day01" extends="struts-default" >
 <action name="welcome" class="com.tarena.action.WelcomeAction">
 <result name="success">/jsp/welcome.jsp</result>
 <result name="fail">/jsp/nameform.jsp</result>
 </action>
 </package>
</struts>

```

- ◆ 注意事项：xml 文件内容严格按照.dtd 格式要求，如：<?xml...?>不在第一行可能会报错，标签中随意换行也会引起 404！

step7：部署，地址栏输入：<http://localhost:8080/struts01/day01/welcome.action> 进行测试，发现一进入页面，则错误提示信息已经显示！为了解决该问题，需要再定义一个<action>用于读取页面，而不是发请求后立即执行 WelcomeAction 类进行处理，处理应当在点击提交后才执行。

step8：在 struts.xml 中添加新的<action>仅仅用于读取页面

```

<action name="welcomeform"><!--不指明 class，Struts2 会自动创建一个类，详见 1.7 节
-->
 <result name="success">/jsp/nameform.jsp</result>
</action>

```

step9：重新部署，地址栏输入：<http://localhost:8080/struts01/day01/welcomeform.action> 进行测试，发现问题已经解决，可正常执行

## 1.7 struts.xml 内容详解

1) <package>：作用是为<action>分组，<struts>标签下可有多个<package>，而<package>标签有如下属性：

- ①name="helloworld"：唯一的标识，表示包名为 helloworld。
  - ◆ 注意事项：在相同包里的不能重复，不同包的可以重复！
- ②namespace="/day01"：用来设置该包中的 action 的地址的命名空间。
  - ◆ 注意事项：
    - ❖ 命名空间中的 “/” 要写！
    - ❖ 也可以这么写：namespace="/"，表示根命名空间，此时 Action 的访问路径为：<http://localhost:8080/appName/actionName.action>  
即：<http://localhost:8080/struts01/welcome.action>
    - ❖ 一般写法为：namespace="/day01"，此时 Action 的访问路径为：<http://localhost:8080/appName/namespace/actionName.action>  
即：<http://localhost:8080/struts01/day01/welcome.action>

③extends="struts-default"：继承的包名，一般继承 Struts2 默认提供的 struts-default 包，该包中定义了很多 Struts2 应用必须的组件（比如：拦截器）；该 package 声明的位置在 struts2-core-2.1.8.jar/struts-default.xml 文件中。

- ◆ 注意事项：
  - ❖ appName 是部署时项目的名字！
  - ❖ 包只能继承包。不能说包继承某个类！

2) <action>：作用是指明每个请求对应的 Action 类之间的对应关系（一个<action>对应

一个请求 ), <package>下可有多个<action>。而<action>标签有如下属性:

①name="welcome": 表示请求的名字为 welcome (即 welcome.action)

②class="com.tarena.action.WelcomeAction": 格式为“包名.类名”。指定了对应的 Action 类 (Java 类) 来处理请求。class 属性可以不写, 不写则 Struts2 会默认认为该<action>添加一个 class (自动创建一个类, 该类中的 execute()方法只有一个 return "success";语句), 作用是转发到对应的<result name="success">中指定的页面。

③method="xxx": 用于指定在对应的 Action 类中要执行的方法名, 该方法的返回值必须是 String 类型 (这是规定!!)

例如: public String xxx(){...return string;}//返回的 string 是<result>标签中某个 name 属性的值 (结果的名字)!

若没有 method 属性或 method="" , 则默认调用方法名为 execute 的方法, 即:  
public String execute(){...return string}。

3) <result>: 作用是指明执行相应的 Action 类之后, 显示哪种结果页。而<result>标签有如下属性:

①name="success": 是该 result 的名称, Action 返回哪一个 result 的 name 值, 意味着要转发到哪一个 result 所对应的 JSP 地址, <result>不写 name 属性, 则默认为 success。

②type"": 默认 dispatcher (转发), 还可写 json、stream、……等 10 种

## 1.8 Struts2 提供的方便之处

1) 数据的自动封装 (输入属性)

根据页面组件的 name 属性, 自动封装到 Action 中对应的 name 属性中, 即自动调用某属性的 set 方法。

例如: 在 JSP 页面<input name="name" type="text" />, 而在 Action 中则会自动给属性 private String name 赋值。

◆ 注意事项: 必须有 name 属性对应的 get 和 set 方法。

2) 数据的自动传递 (输出属性)

Action 中的属性在 JSP 页面可以直接用 EL 表达式拿到, 即自动调用某属性的 get 方法。

例如: Action 中的属性 private String name; 在 JSP 页面上可以直接 \${name} 得到对应的值。

## 1.9 案例: 简单登录 (使用 Struts2)

step1: 新建 Web 工程 struts01, 导入 Struts2 的五个基本核心包, 并在 web.xml 中配置 Struts2 的前端控制器, web.xml 代码, 见 1.6 中 step3

step2: 在 WebRoot 下创建 jsp 文件夹, 并放入 loginform.jsp 和 welcome.jsp

1) loginform.jsp

```
<form action="/struts01/day01/login.action" method="post"><!-- 使用的是绝对路径-->
 <table><tr><td>用户名: </td><td><input name="name" type="text" /></td></tr>
 <tr><td>密 码 : </td><td><input name="pwd" type="password" /></td></tr>
 </table>
 <input value="提交" type="submit" />
 ${errorMsg}<!-- 接收错误信息 -->
</form>
```

◆ 注意事项:

- ❖ 应用名前的“/”要写。
- ❖ <form>标签中的 action 属性只写 action="login.action" 也可！如果写成 action="day01/login.action"，则会出现叠加问题！可参考 Spring 笔记 12.2 节。

## 2) welcome.jsp

```
<h1>Welcome, ${name}</h1>
```

step3：在包 com.tarena.action 下新建 LoginAction 类

```
private String name; private String pwd; private String errorMsg;
.....各自的 get/set 方法
public String execute(){
 if("chang".equals(name)&&"123".equals(pwd)){//模拟登录成功
 return "success";// 与 struts.xml 某一个<result>对应，用来生产视图
 }else{ errorMsg="用户名或密码错误"; return "fail"; }
}
```

step4：在 struts.xml 中配置<action>

```
<struts> <!-- 包只能继承包 -->
 <package name="one" namespace="/day01" extends="struts-default">
 <!--请求地址： http://localhost:8080/appName/namespace/actionName.action -->
 <action name="login" class="com.tarena.action.LoginAction"><!--包名.类名-->
 <result name="success">/jsp/welcome.jsp</result>
 <result name="fail">/jsp/loginform.jsp</result>
 </action>
 </package>
</struts>
```

step5：部署，地址栏输入：http://localhost:8080/struts01/day01/login.action 进行测试，发现一进入页面，则错误提示信息已经显示！为了解决该问题，需要再定义一个<action>用于读取页面，而不是发请求后立即执行 LoginAction 类进行处理，处理应当在点击登录后才执行。

step6：在 struts.xml 中添加新的<action>仅仅用于读取页面

```
<action name="loginform" class="com.tarena.action.LoginFormAction">
 <result name="success">/jsp/loginform.jsp</result>
</action>
```

◆ 注意事项：当前配置指明了 class，则需要自己再写一个 LoginFormAction 类

step7：在包 com.tarena.action 下新建 LoginFormAction 类

```
public class LoginFormAction { public String execute(){ return "success"; } }
```

step8：重新部署，地址栏输入：http://localhost:8080/struts01/day01/loginform.action 进行测试，发现问题已经解决，可正常执行

step9：优化：step6 也可这么写，此时也不用写 step8 中的 LoginFormAction 类了

```
<action name="loginform" ><!--不指明 class，则 Struts2 会自动创建一个类，详见 1.7 节-->
 <result name="success">/jsp/loginform.jsp</result>
</action><!--result 中的 name 属性也可以不写-->
```

## 1.10 案例：修改 1.6、1.9 案例使用户不能绕过前端控制器

1) 问题描述：1.6 和 1.9 案例都存在一个问题，用户可以绕过前端控制器，直接访问 JSP 页面，如地址栏直接输入：http://localhost:8080/struts01/jsp/welcome.jsp，可显示页面，只是获取的名字为空。

## 2) 如何解决:

step1: 将 jsp 文件夹放入 WEB-INF 目录下。因为 WEB-INF 目录中的内容是受保护的，不能直接访问！但能转发过来。

step2: 修改 1.9 案例 struts.xml 中 result 的转发路径，1.6 也同理

```
<action name="loginform"><!-- result 的 type 属性不写，默认为转发 -->
 <result name="success">/WEB-INF/jsp/loginform.jsp</result>
</action>
<action name="login" class="com.tarena.action.LoginAction">
 <result name="success">/WEB-INF/jsp/welcome.jsp</result><!-- 修改路径 -->
 <result name="fail">/WEB-INF/jsp/loginform.jsp</result><!-- 修改路径 -->
</action>
```

step3: 测试

http://localhost:8080/struts01/WEB-INF/jsp/welcome.jsp 不能直接访问

http://localhost:8080/struts01/WEB-INF/jsp/loginform.jsp 也不能直接访问

http://localhost:8080/struts01/day01/loginform.action 只能根据请求，访问登录页面

## 1.11 NetCTOSS 项目：显示资费列表

step1: 新建 Web 工程: NetCTOSS，导入需要的包: Struts2 核心包、Oracle 数据库驱动包、JSTL 包（若为 J2EE5.0 则不需要导这个包了，若为 J2EE1.4 则需要导入！）

step2: 创建 COST\_CHANG 表，并插入数据，别忘记 commit

```
create table cost_chang(
 id number(4) constraint cost_chang_id_pk primary key,
 name varchar2(50) not null,
 base_duration number(11),
 base_cost number(7,2),
 unit_cost number(7,4),
 status char(1) constraint cost_chang_status_ck check(status in (0,1)),
 descr varchar2(100),
 creatime date default sysdate,
 starttime date,
 cost_type char(1));
insert into cost_chang values (1,'5.9 元套餐',20,5.9,0.4,0,'5.9 元 20 小时/月,超出部分 0.4 元
时',default,null,'2');
insert into cost_chang values (2,'6.9 元套餐',40,6.9,0.3,0,'6.9 元 40 小时/月,超出部分 0.3 元
时',default,null,'2');
insert into cost_chang values (3,'8.5 元套餐',100,8.5,0.2,0,'8.5 元 100 小时/月,超出部分 0.2
元/时',default,null,'2');
insert into cost_chang values (4,'10.5 元套餐',200,10.5,0.1,0,'10.5 元 200 小时/月,超出部分
0.1 元/时',default,null,'2');
insert into cost_chang values (5,'计时收费',null,null,0.5,0,'0.5 元 / 时 , 不使用不收费
',default,null,'3');
insert into cost_chang values (6,'包月 ',null,20,null,0,'每月 20 元 , 不限制使用时间
',default,null,'1');
```

- ◆ 注意事项：在 Oracle 数据库中，语句大小写都可，但最终在数据库中显示的是大写形式。

step3：新建 db.properties 到 src 目录下，将 username 和 password 修改为自己的数据库的用户名和密码

```
user=system
password=chang
url=jdbc:oracle:thin:@localhost:1521:dbchang
driver=oracle.jdbc.driver.OracleDriver
```

step4：在 com.tarena.netctoss.util 包中创建 DBUtils 工具类，用于打开连接接和关闭连接

```
public class DBUtils {
 private static String driver; private static String url; private static String user;
 private static String password;
 static {
 try { Properties props = new Properties();
 props.load(DBUtils.class.getClassLoader().getResourceAsStream("db.properties"));
 driver = props.getProperty("driver"); url = props.getProperty("url");
 user = props.getProperty("user"); password = props.getProperty("password");
 Class.forName(driver);
 } catch (Exception e) { throw new RuntimeException("数据库驱动加载错误",e);}
 }
 public static Connection openConnection() throws SQLException {
 Connection con = DriverManager.getConnection(url, user, password);
 return con;
 }
 public static void closeConnection(Connection con) {
 if (con != null) { try { con.close(); } catch (SQLException e) { } }
 }
 public static void main(String[] args) throws Exception {
 Connection con = openConnection();//简单测试下
 System.out.println(con);
 }
}
```

step5：在 com.tarena.netctoss.entity 包中创建实体类 Cost

```
private Integer id; //资费 ID private String name; //资费名称 NAME
private Integer baseDuration; //包在线时长 BASE_DURATION
private Float baseCost; //月固定费 BASE_COST
private Float unitCost; //单位费用 UNIT_COST
private String status; //0: 开通; 1: 暂停; STATUS
private String descr; //资费信息说明 DESCRIPTOR
private Date startTime; //启用日期 STARTTIME
private Date creaTime;//创建时间 CREATETIME
private String costType;//资费类型 COST_TYPE
.....各自的 get/set 方法.....
```

step6：在 com.tarena.netctoss.dao 包中新建 DAOException 类，并继承 Exception

```
public class DAOException extends Exception {
 public DAOException(String message, Throwable cause) {
 super(message, cause);
 }
}
```

step7: 在 com.tarena.netctoss.dao 包中新建 CostDAO 接口

```
public interface CostDAO { public List<Cost> findAll() throws DAOException; }
```

step8: 在 com.tarena.netctoss.dao.impl 包中新建 CostDAOImpl 类, 并实现 CostDAO

```
public class CostDAOImpl implements CostDAO {
 private static String findAll = "select ID, NAME, BASE_DURATION, BASE_COST,
 UNIT_COST, CREATIME, STARTIME, STATUS, DESCR, COST_TYPE from
 COST_CHANG";
 public List<Cost> findAll() throws DAOException {
 Connection con = null;
 try { con = DBUtils.openConnection();
 PreparedStatement stmt = con.prepareStatement(findAll);
 ResultSet rs = stmt.executeQuery();
 List<Cost> costList = new ArrayList<Cost>();
 while (rs.next()) { costList.add(toCost(rs)); }
 return costList;
 } catch (SQLException e) { e.printStackTrace();
 throw new DAOException("访问异常", e); // 抛自己定义的 DAOException
 } finally { DBUtils.closeConnection(con); }
 }
 /** 将结果集中的每一条数据转换成每一个 Cost 对象 */
 private Cost toCost(ResultSet rs) throws SQLException {
 Cost cost = new Cost(); cost.setId(rs.getInt("ID"));
 cost.setName(rs.getString("NAME"));
 cost.setBaseDuration(rs.getInt("BASE_DURATION"));
 cost.setBaseCost(rs.getFloat("BASE_COST"));
 cost.setUnitCost(rs.getFloat("UNIT_COST"));
 cost.setStartTime(rs.getDate("STARTIME"));
 cost.setStatus(rs.getString("STATUS"));
 cost.setCreaTime(rs.getDate("CREATIME"));
 cost.setDescr(rs.getString("DESCR"));
 cost.setCostType(rs.getString("COST_TYPE")); return cost; } } }
```

step9: 在 com.tarena.netctoss.dao 包中新建 DAOFactory 工厂类

```
private static CostDAO costDAO = new CostDAOImpl();
public static CostDAO getCostDAO() { return costDAO; }
```

step10: 在 com.tarena.netctoss.test 包中新建 TestCostDAO 类, 用于测试 DAO。用 JUnit 或 main 方法测试, 此处先用 main 方法测

```
public static void main(String[] args) throws Exception {
 CostDAO costDAO = DAOFactory.getCostDAO();
 List<Cost> costList = costDAO.findAll();
 for (Cost cost : costList) { System.out.println(cost.getId() + "," + cost.getName()); } }
```

step11: 在 com.tarena.netctoss.action.cost 包中新建 CostListAction, 用于接收请求并处理

```
private List<Cost> costList; // 输出属性
private CostDAO costDAO = DAOFactory.getCostDAO(); // DAO 属性不用设置 get/set 方法
.....costList 属性的 get/set 方法.....
public String execute() throws Exception {
```

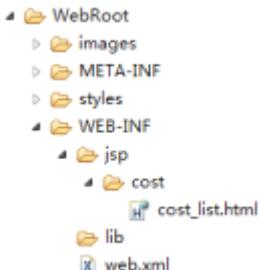
```
costList = costDAO.findAll(); return "success";
```

```
}
```

step12: 在 web.xml 中配置前端控制器

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
 http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"><!--以上都为固定格式-->
<filter><!-- 前端控制器 -->
 <filter-name>Struts2</filter-name>
 <filter-class>
 org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter
 </filter-class>
</filter>
<filter-mapping><!-- 前端控制器映射 -->
 <filter-name>Struts2</filter-name>
 <url-pattern>/*</url-pattern>
</filter-mapping>
</web-app>
```

step13: 拷贝 NetCTOSS 静态页面、图片、样式到工程中，结构如下：



step14: 将静态页面 “.html” 后缀改为动态页面 “.jsp” 后缀

- 1) 首先在页面顶部添加页面指令：<%@page pageEncoding="utf-8"%>
- 2) 然后保存页面，最后再去将后缀改为 “.jsp”，如： cost\_list.jsp

step15: 在 cost\_list.jsp 页面中保留标题行，使用 JSTL 标签和 EL 表达式修改数据行

```
<c:forEach items="#{costList}" var="cost"><!--使用 JSTL 中 forEach 标签循环显示-->
 <tr><td>${cost.id}</td><!--使用 EL--> <td>${cost.name}</td>
 <td>${cost.baseDuration }小时</td> <td>${cost.baseCost }元</td>
 <td>${cost.unitCost }元/小时</td> <td>${cost.creaTime }</td>
 <td>${cost.startTime }</td>
 <td> <c:if test="${cost.status == 0}">开通</c:if>
 <c:if test="${cost.status == 1}">暂停</c:if>
 <c:if test="${cost.status == 2}">删除</c:if></td>
 <td><c:if test="${cost.status == 0}">
 <input type="button" value="暂停" class="btn_pause" /></c:if>
 <c:if test="${cost.status == 1}">
 <input type="button" value="启用" class="btn_start" /></c:if>
 <input type="button" value="修改" class="btn_modify" />
 <input type="button" value="删除" class="btn_delete" /></td>
```

```
</tr></c:forEach>
```

- ◆ 注意事项：
  - ❖ 当学习了 Struts2 的标签和 OGNL 表达式后，上面的 JSTL 标签和 EL 表达式可都被替换掉。
  - ❖ JSP 页面 记 得 引 入 : <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

step16: 【项目经验】在 src 目录下分别创建 struts.xml 和 struts-cost.xml 配置文件

1) struts.xml, 相当于主配置，分别引入其他配置文件。为何要用这种形式？因为当配置文件太大时，采用该方式便于管理

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
 "-//Apache Software Foundation//DTD Struts Configuration 2.1//EN"
 "http://struts.apache.org/dtds/struts-2.1.dtd"><!--以上为固定格式-->
<struts>
 <include file="struts-cost.xml" /><!-- 当配置文件太大时，采用该方式 -->
</struts>
```

2) struts-cost.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
 "-//Apache Software Foundation//DTD Struts Configuration 2.1//EN"
 "http://struts.apache.org/dtds/struts-2.1.dtd">
<struts>
 <package name="cost" namespace="/cost" extends="struts-default">
 <action name="list" class="com.tarena.netctoss.action.cost.ListCostAction">
 <result name="success"/>/WEB-INF/jsp/cost/cost_list.jsp</result>
 </action>
 </package>
</struts>
```

step17: 部署，测试，地址栏输入：http://localhost:8080/NetCTOSS/cost/list.action，效果如下：

资费ID	资费名称	基本时长	基本费用	单位费用	创建时间	开通时间	状态	操作
1	5.9元套餐	20小时	5.9元	0.4元/小时	2013-08-29	2013-08-30	开通	 暂停  修改 
2	6.9元套餐	40小时	6.9元	0.3元/小时	2013-08-29	2013-08-30	开通	 暂停  修改 
3	8.5元套餐	100小时	8.5元	0.2元/小时	2013-08-29	2013-08-30	开通	 暂停  修改 
4	10.5元套餐	200小时	10.5元	0.1元/小时	2013-08-29		暂停	 启用  修改 
5	计时收费	0小时	0.0元	0.5元/小时	2013-08-29		暂停	 启用  修改 
6	包月	0小时	20.0元	0.0元/小时	2013-08-29		暂停	 启用  修改 

- ◆ 注意事项：为了看效果，此处在数据库中修改了几个资费状态值和开通时间。

## 1.12 NetCTOSS 项目：资费列表分页显示

step1: 此例是在 1.11 案例基础之上做修改。也可以复制 1.11 案例的工程再修改，如果采取复制，则步骤如下：

1) 点击 NetCTOSS 工程，右键点击“Copy”，然后右键点击“Paste”，起个工程名，如：NetCTOSS2

2) 但是这样复制的工程，在 MyEclipse 中工程名虽然为 NetCTOSS2，但是当要部署到

Tomcat 下时，发现无法部署，因为已经有同名的应用被部署了！所以，此时需要修改 工程名 为 NetCTOSS2 的工程，在部署时的应用名！

3) 修改部署时的应用名：右键工程-->选择“Properties”-->选择“MyEclipse”-->选择“Web”-->在 Web Context-root 文本框中，为当前工程部署时的应用名，将它修改为“/NetCTOSS2”，点击“OK”即可，记得要写“/”。此时，工程名和应用名一致。

◆ 注意事项：要注意区分工程名和应用名的问题！

step2：在 CostDAO 接口中添加方法定义

```
public List<Cost> findAll(int page,int rowsPerPage) throws DAOException;//分页查询
public int getTotalPages(int rowsPerPage) throws DAOException;//获得总页数
```

step3：在 CostDAOImpl 中实现方法

```
private static String findAllByPage="select ID, NAME, BASE_DURATION, BASE_COST,
 UNIT_COST, CREATIME, STARTIME, STATUS, DESCR,COST_TYPE from (select
 ID, NAME, BASE_DURATION, BASE_COST, UNIT_COST, CREATIME, STARTIME,
 STATUS,DESCR,COST_TYPE, rownum n from COST_CHANG where rownum<?) where
n>=?";

public List<Cost> findAll(int page, int rowsPerPage) throws DAOException { //分页查询
 Connection con = null;
 try { con = DBUtils.openConnection();
 PreparedStatement stmt = con.prepareStatement(findAllByPage);
 int start=(page-1)*rowsPerPage +1; int end=start+rowsPerPage;
 stmt.setInt(1,end); stmt.setInt(2,start); ResultSet rs = stmt.executeQuery();
 List<Cost> costList = new ArrayList<Cost>();
 while (rs.next()) { costList.add(toCost(rs)); } return costList;
 } catch (SQLException e) {
 e.printStackTrace(); throw new DAOException("访问异常", e);
 } finally { DBUtils.closeConnection(con); } }

public int getTotalPages(int rowsPerPage) throws DAOException {
 Connection con = null;
 try { con = DBUtils.openConnection();
 PreparedStatement stmt = con.prepareStatement(
 "select count(*) from COST_CHANG");
 ResultSet rs = stmt.executeQuery();
 List<Cost> costList = new ArrayList<Cost>();
 rs.next(); int totalRows=rs.getInt(1);
 if(totalRows%rowsPerPage==0){ return totalRows/rowsPerPage;
 }else { return (totalRows/rowsPerPage)+1; }
 } catch (SQLException e) {
 e.printStackTrace(); throw new DAOException("访问异常", e);
 } finally { DBUtils.closeConnection(con); } }
```

step4：修改 CostListAction

```
private int page=1;//添加当前页属性， input、 output 双重属性
private int totalPages;//添加总页数属性， output
.....各自 get/set 方法.....
public String execute() throws Exception { costList = costDAO.findAll(page,2);//每页 2 条
```

```

totalPages=costDAO.getTotalPages(); //每页 2 条数据, 计算出的总页数
return "success";
}

```

step5：修改 cost\_list.jsp 中的分页条

```

<div id="pages">
 首页
 <c:choose><c:when test="${page > 1}">
 上一页
 </c:when>
 <c:otherwise>上一页</c:otherwise>
 </c:choose>
 <c:forEach begin="1" end="${totalPages}" var="p">!--forEach 用法见 JSP 笔记
3.3-->
 <c:choose><c:when test="${page == p}">
 ${p}
 </c:when>
 <c:otherwise>${p}
 </c:otherwise>
 </c:choose>
 </c:forEach>
 <c:choose><c:when test="${page < totalPages}">
 下一页
 </c:when>
 <c:otherwise>下一页</c:otherwise>
 </c:choose>
 末页
</div>

```

- ◆ 注意事项：
  - ❖ JSP 页 面 记 得 引 入 : <%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
  - ❖ 如果切换上下页，出现错位的情况，在 JSP 页面按下 Ctrl+Shift+F，格式化一下，然后保存即可。

step6：部署，测试，地址栏输入：http://localhost:8080/NetCTOSS/cost/list.action，效果如下：

资源ID	资源名称	基本时长	基本费用	单位费用	创建时间	开通时间	状态	
3	8.5元套餐	100小时	8.5元	0.2元/小时	2013-08-29	2013-08-30	开通	<input type="radio"/> 暂停 <input checked="" type="checkbox"/> 修改 <input type="checkbox"/> 删除
4	10.5元套餐	200小时	10.5元	0.1元/小时	2013-08-29		暂停	<input type="radio"/> 启用 <input checked="" type="checkbox"/> 修改 <input type="checkbox"/> 删除

首页 上一页 1 2 3 下一页 末页

# 一百一十六、OGNL 技术

LICHOO

## 2.1 什么是 OGNL

OGNL 是 Object-Graph Navigation Language（对象图形导航语言）的缩写，它是一种功能强大的表达式语言。

OGNL 可以让我们用非常简单的表达式访问对象层，它用一个独立的 lib 形式出现（封装于 ognl.jar 中），方便我们使用或构建自己的框架。

## 2.2 OGNL 基本语法

OGNL 引擎访问对象的格式: Ognl.getValue("OGNL 表达式",root 对象); //root 对象是 Ognl 要操作的对象。

1) 访问基本类型属性: (见案例 1)

```
System.out.println(Ognl.getValue("id", foo));//100
System.out.println(Ognl.getValue("name", foo));//Java
```

2) 访问数组属性:

```
System.out.println(Ognl.getValue("arry[0]", foo));//one
```

3) 访问集合 List 属性:

```
System.out.println(Ognl.getValue("list[1]", foo));//B
```

4) 访问集合 Map 属性:

```
System.out.println(Ognl.getValue("map.one", foo));//Java
或 System.out.println(Ognl.getValue("map['two']", foo));//JavaJava
```

5) 基本运算:

```
System.out.println(Ognl.getValue("id+100", foo));//200
System.out.println(Ognl.getValue("\\"What is "+name", foo));//What is Java
System.out.println(Ognl.getValue("\\"name: "+ name + " id: "+id",foo));
System.out.println(Ognl.getValue("id > 150", foo));//false //name: Java
id:100
```

6) 调用方法:

```
System.out.println(Ognl.getValue("name.toUpperCase()", foo));//JAVA
System.out.println(Ognl.getValue("list.size()", foo));//3
```

◆ 注意事项: 方法的参数也可以使用属性

```
System.out.println(Ognl.getValue("map['three'].lastIndexOf(name)", foo));//8
```

7) 调用静态方法: 调用静态方法的格式: @类名@方法名

```
System.out.println(Ognl.getValue("@java.util.Arrays@toString(arry)",foo));//[one,two,three]
```

8) 创建的 List 对象:

```
Object obj = Ognl.getValue("{1,2,3,4,5}", null); //这种方法更方便地临时创建一个 List 对象
System.out.println(obj.getClass().getName());//java.util.ArrayList
System.out.println(obj);//[1,2,3,4,5]
```

9) 创建的 Map 对象:

```
obj = Ognl.getValue("#{1:'java',2:'javajava',3:'javajavajava'}", null); //注意: “#”号不能丢
System.out.println(obj.getClass().getName());//java.util.LinkedHashMap
```

```
System.out.println(obj); // [1=java, 2=javajava, 3=javajavajava]
```

◆ 注意事项：OGNL 中只能创建 List 和 Map 对象。

10) 案例 1：step1：新建工程，导入 Struts2 核心包，配置前端控制器

step2：创建 Foo 类，属性如下

```
private Integer id; private String name; private String[] arry;
private List<String> list; private Map<String, String> map; ...各自的 get/set 方法
```

step3：创建 TestOgnl 类，在 main 方法中先进行赋值，然后执行各类型测试

```
Foo foo = new Foo(); foo.setId(100); foo.setName("Java");
foo.setArry(new String[] { "one", "two", "three" });
foo.setList(Arrays.asList("A", "B", "C")); // 返回一个受指定数组支持的固定大小的集合。

HashMap<String, String> map = new HashMap<String, String>();
map.put("one", "Java"); map.put("two", "JavaJava");
map.put("three", "JavaJavaJava"); foo.setMap(map);
```

11) 访问对象中的属性：（见案例 2）

```
System.out.println(Ognl.getValue("name", emp1)); // chang1
System.out.println(Ognl.getValue("dept.name", emp2)); // R&D
System.out.println(Ognl.getValue("emps[2].salary", dept)); // 3000.0
System.out.println(Ognl.getValue("emps[0].salary + 1000", dept)); // 2000.0
System.out.println(Ognl.getValue("emps[1].salary > 10000", dept)); // false
System.out.println(Ognl.getValue("emps[2].name.toUpperCase()", dept)); // CHANG3
```

12) 案例 2：step1：创建 Emp 员工类

```
private String name; private double salary; private Dept dept; // 又是一个对象类型
public Emp(String name, double salary) { this.name=name; this.salary=salary; } // 构造器
.....各自的 get/set 方法
```

step2：创建 Dept 部门类

```
private String name; private List<Emp> emps=new ArrayList<Emp>();
public Dept(String name){ this.name=name; } // 构造器
.....各自的 get/set 方法
public void addEmp(Emp emp){ emps.add(emp); // 将员工加入集合
emp.setDept(this); // 将当前 Dept 对象的名称给 emp 对象的 dept 属性赋值 }
```

step3：在 main 方法中先赋值，然后执行访问对象属性的测试

```
Dept dept=new Dept("R&D"); Emp emp1=new Emp("chang1", 1000);
Emp emp2=new Emp("chang2", 2000); Emp emp3=new Emp("chang3", 3000);
dept.addEmp(emp1); dept.addEmp(emp2); dept.addEmp(emp3);
```

## 2.3 OGNL 表达式中加“#”和不加“#”的区别

step1：在 2.2 节的工程中，新建 Bar 类，添加 name 属性，并设置 get/set 方法

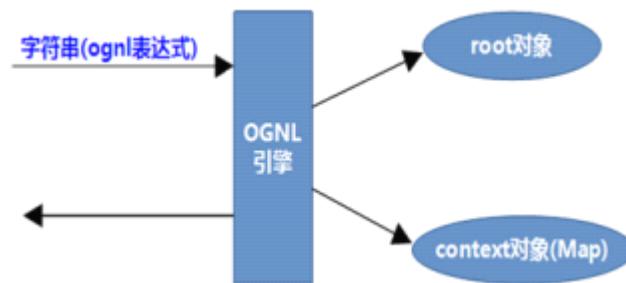
step2：创建 TestOgnl2 类，在 main 方法中进行测试

```
// 自定义 context 对象（见 2.4-2.5 节 OGNL 体系结构），如果不写系统会自动加 -
```

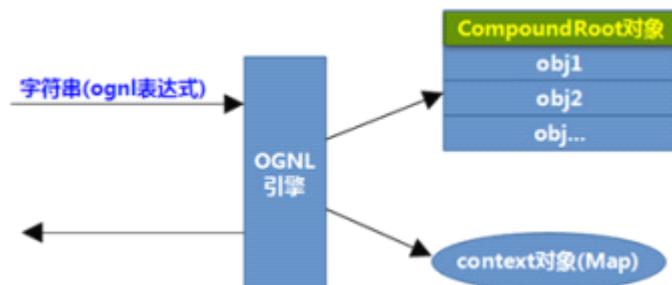
个

```
Map<String, Object> ctx = new HashMap<String, Object>(); ctx.put("num", 10);
Bar root = new Bar(); //创建 root 对象 root.setName("bar");
//不加"#", 表示从业务对象 root 中取数据
System.out.println(Ognl.getValue("name", ctx, root)); //bar
//加"#", 表示从公共对象 context 中取数据
System.out.println(Ognl.getValue("#num", ctx, root)); //10
```

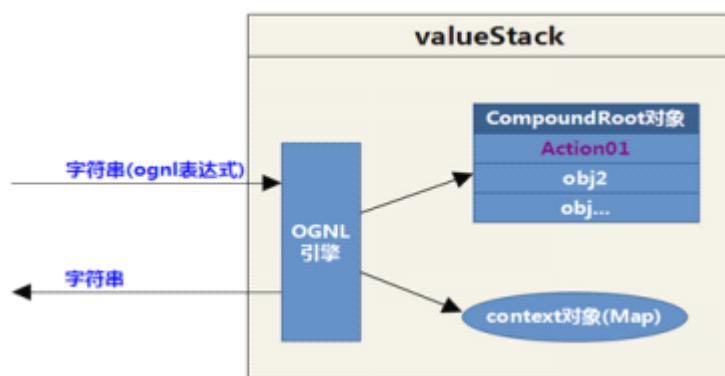
## 2.4 OGNL 体系结构

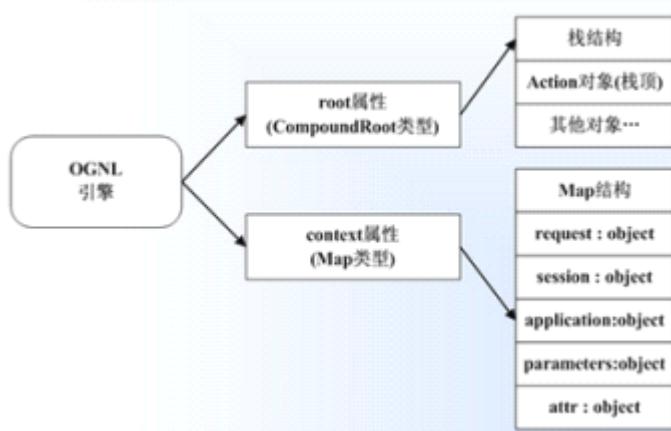


## 2.5 XWord 框架对 OGNL 进行了改造



## 2.6 ValueStack 对象结构





## 2.7 ValueStack 结构演示

step1：新建工程 struts02，导入 Struts2 核心包，配置前端控制器

step2：新建 DebugAction 类

```

private String name; private int id; private String[] arry;
public String execute() { name = "java"; id = 100;
 arry = new String[] {"struts","hibernate","spring" }; return "success";
 各自 get/set 方法

```

step3：配置 struts.xml

```

<package name="valuestack" namespace="/day02" extends="struts-default" >
 <action name="debug" class="action.DebugAction">
 <result name="success"/>/WEB-INF/jsp/debug.jsp</result>
 </action>
</package>

```

step4：在 WEB-INF/jsp 下新建 debug.jsp

```

<%@page pageEncoding="utf-8"%>
<%@page import="java.util.*" %>
<% Enumeration en = request.getAttributeNames(); //获取 request 对象中所有属性
while(en.hasMoreElements()){//循环打印出所有属性
 out.println(en.nextElement() + "
"); }
Object obj = request.getAttribute("struts.valueStack");//打印出 struts.valueStack
out.println("<hr/>struts.valueStack 对象:
" + obj); %>

```

step5：部署，地址栏输入：<http://localhost:8080/struts02/day02/debug.action>，测试结果如下：

```

javax.servlet.forward.request_uri
javax.servlet.forward.context_path
javax.servlet.forward.servlet_path
struts.request_uri
struts.view_uri
_cleanup_recursion_counter
struts.actionMapping
struts.valueStack

```

---

```

struts.valueStack 对象：
com.opensymphony.xwork2.ognl.OgnlValueStack@162e703

```

step6：Struts2 提供了专门的标记库，用于查看 ValueStack 中的内容，一般用于调试程序，在 debug.jsp 尾部添加标签

```
<s:debug />
```

step7：地址栏输入：<http://localhost:8080/struts02/day02/debug.action>，点击[debug]，显示如下：

The screenshot shows the 'Struts ValueStack Debug' interface. At the top, there is a button labeled '[Debug]'. Below it, the title 'Struts ValueStack Debug' is displayed. The interface is divided into two main sections: 'Value Stack Contents' and 'Stack Context'.

**Value Stack Contents:** This section displays the contents of the Value Stack. It has a header row with columns 'Object' and 'Property Name' and 'Property Value'. A yellow box highlights the 'Object' column. Below this, there are two rows of data:

Object	Property Name	Property Value
com.tarena.outman.DebugAction	id	10 案顶
com.opensymphony.xwork2.DefaultTextProvider	name	java
	texts	null

A yellow box highlights the 'Property Name' column for the first row. The 'Property Value' column for the first row is also highlighted.

**Stack Context:** This section displays context items available using the #key notation. It has a header row with 'Key' and 'Context 对象'. A yellow box highlights the 'Key' column. Below this, there is a single row of data:

Key	Context 对象
com.opensymphony.xwork2.dispatcher.HttpServletRequest	org.apache.struts.

A yellow box highlights the 'Key' column for the first item.

## 2.8 Struts2 标签的使用

Struts2 的很多标记就是通过访问 ValueStack 获得数据的。

使用前要在页面中要引入：`<%@taglib uri="/struts-tags" prefix="s"%>`，prefix：表示前缀

1) 通过 OGNL 从 ValueStack 取数据，并且显示。

```
<s:property value="ognl 表达式"/>
```

2) 省略 value，取出 ValueStack 的栈顶（默认从栈顶取值）。

```
<s:property />
```

3) 通过 OGNL 从 ValueStack 取出集合，依次将集合中的对象置于栈顶，在循环中，ValueStack 栈顶即为要显示的数据。

```
<s:iterator value="ognl 表达式指定的集合名">
 <s:property value="name" /><!--将每个对象中 name 属性的值输出-->
</s:iterator>
```

将 OGNL 指定的集合属性循环迭代，在迭代中，会将当前元素压入 root 栈顶位置，因此使用“属性”格式的 OGNL，此时定位的是元素属性，而不是 Action 属性。Iterator 循环结束后，Action 对象又恢复成栈顶。与 JSTL 标签中的 forEach 类型，也有相同的属性。其中 var、status 属性指定的变量存储在 context 区域，需要使用“#变量”访问。

4) if-else 判断标签

```
<s:if test=“ognl 判断表达式”>满足条件时</s:if> <s:else>不满足条件时</s:else>
```

5) 可在 value 属性中做字符串拼接

```
<s:property value="name + 'java'" /><!--接 2.7 节，结果为：java java-->
```

6) 可调用方法

```
<s:property value="arry[1].toUpperCase()" /><!--接 2.7 节，结果为：HIBERNATE-->
```

7) 其他详细用法，可参考 2.2 节或第五章，只不过我们在 Struts2 标签中直接写 OGNL 表达式（字符串）即可！

## 2.9 Struts2 对 EL 表达式的支持

EL 表达式默认是访问 page、request、session、application 范围的数据。

在 Struts2 中也可以使用 EL 表达式访问 Action 对象的属性。访问顺序是 page、request、root 栈（action）、session、application。当 EL 从 request 中获取不到信息后，会去 ValueStack 对象的 root 栈获取，原因是：Struts2 重写了 HttpServletRequest 的方法。

## 2.10 案例：修改 1.12 案例（使用 Struts2 标签和 OGNL 表达式）

```
<div id="pages"> 首页
 <s:if test="page > 1">上一页</s:if>
 <s:else>上一页</s:else>
 <s:iterator begin="1" end="totalPages" var="p">
 <s:if test="top==page"><s:property /></s:if>
 <s:else><s:property /></s:else>
 <!-- Struts2 标签写在属性里也行，但不符合习惯，一般写 EL 表达式-->
 </s:iterator>
 <s:if test="page < totalPages">下一页</s:if>
 <s:else>下一页</s:else>
 末页
</div>
```

# 一百一十七、Action

LICHOO

## 3.1 Struts2 的核心组件

- 1) FC: 前端控制器
- 2) VS: ValueStack
- 3) Action: 动作
- 4) Result: 结果
- 5) Interceptor: 拦截器
- 6) Tags: 标记库

## 3.2 Struts2 的工作流程

- 1) 所有请求提交给 FC。
- 2) 根据配置信息确定要调用的 Action。
- 3) 创建一个 ValueStack 对象（每个请求都有一个独立的 VS 对象）。
- 4) 创建 Action 对象，把 Action 对象放到 VS 的栈顶，将 VS 对象存入到 request 中，存储的 key 为 “struts.valueStack”。
- 5) 控制器调用 Action 对象接收请求参数，并在方法中根据输入属性算输出属性。
- 6) 在调用 Action 之前或之后调用一系列 Interceptor。
- 7) 根据 Action 返回的字符串确定 Result (10 种类型)。
- 8) 调用 Result 对象，将 VS 中的数据按照特定的格式输出。
- 9) 很多情况下，Result 将转发到 JSP，JSP 页面用 Tags 取出数据并显示。
- 10) 请求处理完后，将 ValueStack 对象和 Action 对象销毁。

## 3.3 在 Action 中访问 Session 和 Application

- 1) 利用 ActionContext，返回 Map 类型

```
ActionContext context=ActionContext.getContext();
Map<String, Object> session=context.getSession();
session.put("username", "chang");
Map<String, Object> request=(Map)context.get("request");
Map<String, Object> app=context.getApplication();
```

- ◆ 注意事项：不推荐使用 ActionContext 访问 Session 的方式，因为这种方式的“侵入性”较强。ActionContext 是 Struts2 的 API，如果使用其他框架代替目前的 Struts2 框架，而我们实际项目中的 Action 的数量非常庞大，每个类都修改的话，会非常麻烦。推荐使用实现 SessionAware 接口的方式。

- 2) ServletActionContext 返回的是 Servlet 使用类型

```
HttpServletRequest httpreq=ServletActionContext.getRequest();
HttpSession httpsession=httpreq.getSession();
ServletContext httpapp=ServletActionContext.getServletContext();
```

- 3) 在 Action 中实现 Aware 接口，由框架底层将对象注入给 Action 变量

```
RequestAware: Map 类型的 request
SessionAware: Map 类型的 session
ApplicationAware: Map 类型的 application
ServletRequestAware: http 类型的 request
ServletResponseAware: http 类型的 response
ServletContextAware: http 类型的 application
```

### 3.4 NetCTOSS 项目：用户登录

在原项目的基础上，添加登录模块。

step1：创建 ADMIN\_CHANG 表，并插入数据，别忘记 commit

```
create table admin_chang(
 id number(11) constraint admin_info_id_pk_chang primary key,
 admin_code varchar2(30) unique not null,
 password varchar2(30) not null,
 name varchar2(30) not null,
 telephone varchar2(30),
 email varchar2(50),
 enrolldate date not null
);
insert into admin_chang values(1001,'admin','123','管理员','13688997766','admin@sin.com'
,to_date('2013-08-05 15:30:30', 'yyyy-mm-dd hh24:mi:ss'));
insert into admin_chang values(1002,'user','123','用户','13688997766','shiyi@sin.com'
,to_date('2013-08-22 08: 30: 23','yyyy-mm-dd hh24:mi:ss'));
```

step2：在 com.tarena.netctoss.entity 包下，新建 Admin 实体类

```
private Integer id; // ID private String adminCode; // ADMIN_CODE
private String password; // PASSWORD private String name; // NAME
private String telephone; // TELEPHONE private String email; // EMAIL
private Date enrollDate; // ENROLLDATE各自的 get/set 方法
```

step3：在 com.tarena.netctoss.dao 包下，新建 AdminDAO 接口

```
public interface AdminDAO {
 public Admin findByCodeAndPwd(String code, String pwd) throws DAOException; }
```

step4：在 com.tarena.netctoss.dao.impl 包下，新建 AdminDAOImpl 类，并实现 AdminDAO

```
public class AdminDAOImpl implements AdminDAO {
 private static final String findByCodeAndPwd="select ID,ADMIN_CODE,
PASSWORD, NAME, TELEPHONE, EMAIL, ENROLLDATE from ADMIN_CHANG where
ADMIN_CODE=? and PASSWORD=?";
 public Admin findByCodeAndPwd(String code, String pwd) throws DAOException {
 Connection con = null; Admin admin=null;
 try { con = DBUtils.openConnection();
 PreparedStatement stmt = con.prepareStatement(findByCodeAndPwd);
 stmt.setString(1, code); stmt.setString(2, pwd);
 ResultSet rs = stmt.executeQuery();
 if(rs.next()){ admin = toAdmin(rs); }
 return admin;
 } catch (Exception e) { e.printStackTrace();
 throw new DAOException("访问异常", e); //抛自己定义的 DAOException
 } finally{ DBUtils.closeConnection(con); }
 }
 /** 将结果集中的每一条数据转换成每一个 Admin 对象 */
 private Admin toAdmin(ResultSet rs) throws SQLException{
```

```

 Admin admin = new Admin(); admin.setId(rs.getInt("ID"));
 admin.setAdminCode(rs.getString("ADMIN_CODE"));
 admin.setPassword(rs.getString("PASSWORD"));
 admin.setName(rs.getString("NAME"));
 admin.setTelephone(rs.getString("TELEPHONE"));
 admin.setEmail(rs.getString("EMAIL"));
 admin.setEnrollDate(rs.getDate("ENROLLDATE")); return admin; } }

```

step5：在 DAOFactory 工厂类中添加 Admin 实例

```

private static AdminDAO adminDAO = new AdminDAOImpl();
public static AdminDAO getAdminDAO() { return adminDAO; }

```

step6：在 com.tarena.netctoss.test 包中新建 TestAdminDAO 类，用于测试 DAO。此处用 JUnit

```

@Test
public void testFindAllByPage() throws Exception {
 AdminDAO adminDAO=DAOFactory.getAdminDAO();
 Admin admin=adminDAO.findByName("admin", "123");
 System.out.println(admin.getId()); System.out.println(admin.getName());
}

```

step7：在 com.tarena.netctoss.action 包中新建 LoginAction，用于接收客户端的请求并处理

```

private Admin admin; private String errorMsg="用户名或密码错误";
private AdminDAO adminDAO = DAOFactory.getAdminDAO();//DAO 不用设置 get/set
.....各自的 get/set 方法.....
public String execute() throws Exception {
 admin = adminDAO.findByName("admin", "123");
 if (admin != null) { session.put(ADMIN_KEY, admin);
 return "success"; // 使用 Session，需要实现 SessionAware 接口
 } else { return "fail"; }
}

```

step8：在 com.tarena.netctoss.action 包中新建 BaseAction，BaseAction 实现 SessionAware，然后 step7 中的 LoginAction 继承 BaseAction

### 1) BaseAction

```

public class BaseAction implements SessionAware {
 protected Map<String, Object> session;
 public void setSession(Map<String, Object> session) { this.session = session; }
}

```

#### ◆ 注意事项：

- ❖ 该类实现了 SessionAware 接口，表明 Struts2 在启动 BaseAction 时，首先，创建出该 Action 的对象，放入“栈顶”然后，会调用 BaseAction 的 setSession 方法，把 session 传入给 BaseAction 对象（注意：如果是普通的 Action，Struts2 在启动时仅创建出该 Action 的对象，然后放入“栈顶”）由此，我们定义了属性 session，以便在之后其它的方法中使用。为了让子类也能使用，所以访问控制符为 protected。
- ❖ 按照这样的机制，我们可以让所有的 Action（如 LoginAction）继承实现了 SessionAware 接口的 BaseAction，当需要更换 Struts2 框架为其他框架时，只需要修改 BaseAction 即可（另外的 框架只要提供一个类似 SessionAware 接口的接口，由 BaseAction 继承）

### 2) LoginAction

```

public class LoginAction extends BaseAction { ... }

```

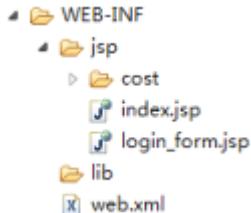
step9: 为了方便统一管理 KEY，所以在 com.tarena.netctoss 包下新建 Constants 接口，用于定义常量 KEY

```
public interface Constants {
 public static final String ADMIN_KEY = "com.tarena.netctoss.admin.key";
}
```

step10: 所以 BaseAction 要再实现 Constants

```
public class BaseAction implements SessionAware, Constants { ... }
```

step11: 拷贝登录页面、主页的静态页面片工程中，并修改后缀为 “.jsp”，结构如下：



step12: 修改 login\_form.jsp，添加<form>标签、帐号和密码添加 name 属性，用于提交给服务器、页面引入 Struts2 标签：<%@taglib uri="/struts-tags" prefix="s"%>

```
<form id="loginForm" action="login.action" method="post">
 <table><input name="admin.adminCode" type="text" class="width150" />
 <input name="admin.password" type="password" class="width150" />
 <td class="login_button" colspan="2"><!--loginForm 为 form 的 id 值-->

 </td><!--登录按钮图片-->
 <td><s:property value="errorMsg"/></td>
 </form>
```

- ◆ 注意事项：
  - ❖ 在<a>标签中， href="javascript:;"一般和 onclick 一起使用，主要用于触发点击事件，单独用相当于空连接： href="#"。
  - ❖ 由于放置位置的原因， login\_form.jsp 中的样式、图片链接，都要加“..”，如： href="../styles/global.css"

step13: 在 src 目录下新建 struts-login.xml

```
<package name="login" namespace="/login" extends="struts-default">
 <action name="loginform"><!--仅用于读取页面-->
 <result name="success">/WEB-INF/jsp/login_form.jsp</result>
 </action>
 <action name="login" class="com.tarena.netctoss.action.LoginAction">
 <result name="success">/WEB-INF/jsp/index.jsp</result>
 <result name="fail">/WEB-INF/jsp/login_form.jsp</result>
 </action>
</package>
```

step14: 在 struts.xml 中引入

```
<include file="struts-login.xml" />
```

- ◆ 注意事项：在 xml 文件中，有时多个空行都会报错！

step15: 部署，测试，地址栏输入：http://localhost:8080/NetCTOSS/login/loginform.action，输入正确则进入主页面，输入错误才出现提示信息：



### 3.5 Action 属性注入

在<action>配置中，为 Action 组件的属性指定一个初始值，该值在创建 Action 对象时注入，可以将一些变化的参数值，利用该方法指定。例如：pageSize、password 密码、dir 存储路径等。例如：

```
public class ParamAction{ private String param1; private int param2; ... 各自
get/set}
<action name="param" class="com.tarena.ParamAction">
 <param name="param1">ABC</param>
 <param name="param2">10</param>
</action>
```

### 3.6 案例：重构 NetCTOSS 资费列表分页显示（使用属性注入）

一般情况下，为该 Action 定义一些属性，有些是用于输入的（比如表单提交用户名、密码），有些是用于输出的（比如员工列表、当前页数）此外，还有一些属性是 Action 自用的参数，比如 1.12 节分页案例中，每页显示多少行的行数，我们是写死在程序里的。

例如 1.12 案例 step4：

```
costList = costDAO.findAll(page,2); //每页 2 条
totalPages=costDAO.getTotalPages(2); //每页 2 条数据，计算出的总页数
```

但是，我们可以把它声明为一个属性，方便更改。pageSize 就是参数，而参数注入就是说，像 pageSize 这样的参数，我们可以写在配置文件中。

step1：修改 1.12 案例 step4 中的 ListCostAction

```
private int pageSize =2; //添加 pageSize 属性，以及它的 get/set 方法
costList = costDAO.findAll(page,pageSize); //修改方法中的每页条数
totalPages=costDAO.getTotalPages(pageSize);
```

step2：修改 struts-cost.xml

```
<action name="list" class="com.tarena.netctoss.action.cost.ListCostAction">
 <param name="pageSize">3</param>!--会把原来定义的每页 2 条修改为 3 条
-->
 <result name="success">/WEB-INF/jsp/cost/cost_list.jsp</result>
</action>
```

### 3.7 使用通配符配置 Action

1) \*\_\*：代表名称中有“\_”下划线的所有 Action。

2) 例如：

```
<action name="*_*_*" class="com.tarena.{1}Action" method="{2}">
```

UCHOO

```
<result name="success">/WEB-INF/jsp/{3}.jsp</result>
```

```
</action>
```

- ①{1}Action：表示 name 属性中第一个“\*”星号匹配的字符串。
- ②method="{2}"：表示方法名为 name 属性中第二个“\*”星号匹配的字符串。
- ③{3}.jsp：表示显示的页面为 name 属性中第三个“\*”星号匹配的字符串。

### 3.8 案例：通配符配置（资费增、改、查）

step1：新建项目 struts03，导入 Struts2 核心包，配置前端控制器

step2：在 WebRoot 目录下新建 cost.jsp

```
资费查看
资费更新
资费添加
```

step3：在 action 包中新建 CostAction，并定义三个方法

```
public String add(){ System.out.println("资费添加操作"); return "success"; }
public String list(){ System.out.println("资费查看操作"); return "success"; }
public String update(){ System.out.println("资费更新操作"); return "success"; }
```

step4：新建 struts.xml

```
<package name="cost" extends="struts-default">
 <action name="*_*" class="action.{1}Action" method="{2}">
 <result name="success">cost.jsp</result>
 </action>
</package>
```

step5：部署，测试，地址栏输入：<http://localhost:8080/struts03/cost.jsp>，效果如下：

[资费查看](#) [资费更新](#) [资费添加](#)

step6：点击三个连接，则分别执行了各自的方法：

|  
| 资费查看操作  
| 资费更新操作  
| 资费添加操作  
|

### 3.9 Struts2 中 Action 的设计经验

- 1) 配置文件的拆分。
- 2) 控制好 Action 类的粒度（不能太多或就 1 个）。
- 3) Action 和 Servlet API 的耦合度要低，如若非要用底层 API，实现 XXXAware 接口，可以封装拦截器，如：sessionAware。
- 4) 适当用通配符。
- 5) 根据输入算输出。

# 一百一十八、Result

Result 组件是一个类，职责是生成视图。该类必须实现 Result 接口，并且将约定的 execute 方法实现，在该方法中编写了生成响应输出的逻辑代码。

视图可以是多种多样的（比如 JSP、JSON、报表、freeMarker 等），这些视图都可以由 Result 负责。

## 4.1 Result 注册配置

```
<package>
 <result-types>
 <result-type name="类型名" class="实现类"
 //...其它类型的 result
 </result-type>
 </package>
```

## 4.2 Result 组件利用<result>元素的 type 属性指定 result 类型

```
<action>
 <result type="result 类型">
 //...result 参数指定
 </result>
</action>
```

## 4.3 常见的 Result 组件类型

- 1) dispatcher (默认): 以请求转发方式调用一个 JSP，生成响应视图。
- 2) redirect: 以重定向方式调用一个 JSP，生成响应视图。
- 3) redirectAction: 以重定向方式调用一个 Action。  
参数: ①actionName: 要定向到的 Action 的 name 值。  
②namespace: 要定向到的 Action 所在包的命名空间的名字。
- 4) chain: 以请求转发方式调用一个 Action。
- 5) stream: 以字节流方式响应，将 Action 中指定的一个 InputStream 类型的属性以字节流方式输出。  
参数: ①inputName: OGNL 表达式，表示要输出的一个输入流对象（要输出数据的来源）。  
②contentType: 用于设置响应中 contentType。
- 6) json: 以 json 字符串响应，将 Action 中指定的属性，拼成一个 json 字符串输出。  
参数: ①root: OGNL 表达式，要做成 JSON 字符串的对象

## 4.4 NetCTOSS 项目：资费删除

step1: 在 CostDAO 接口中添加删除

```
public void delete(int id) throws DAOException;
```

step2: 在 CostDAOImpl 类中实现该方法

```
private static String delete="delete from COST_CHANG where ID=?"; //SQL 删除语句
public void delete(int id) throws DAOException {
 Connection con = null;
 try { con = DBUtils.openConnection();
```

```

 PreparedStatement stmt = con.prepareStatement(delete);
 stmt.setInt(1, id); stmt.executeUpdate();
 } catch (Exception e) { e.printStackTrace();
 throw new DAOException("访问异常", e);
 } finally { DBUtils.closeConnection(con); }
}

```

step3：在 com.tarena.netctoss.action.cost 包下，新建 DeleteCostAction

```

private int id; private CostDAO costDAO=DAOFactory.getCostDAO();//不需要 get/set
.....id 属性的 get/set 方法
public String execute() throws DAOException{ costDAO.delete(id); return "success"; }

```

step4：在 struts-cost.xml 中添加删除配置

```

<action name="delete" class="com.tarena.netctoss.action.cost.DeleteCostAction">
 <result name="success" type="redirectAction">list</result>
</action>

```

- ◆ 注意事项：上面代码中的 type="redirectAction" 表示重定向到一个 action，重定向到一个 jsp 可以使用 type="redirect"。转发到一个 action 使用 type="chain"，转发到一个 jsp 使用 type="dispatcher"。如果 result 没有定义 type 属性，那么默认值就是 "dispatcher"。

step5：为 cost\_list.jsp 中的删除按钮添加 onclick 事件

```

<input type="button" value="删除" onclick="location.href='delete.action?id=${cost.id}'"../>

```

- ◆ 注意事项：由于此时还是用的 JSTL 中的 forEach 标签，所以必须要有 var 属性绑定变量，而之前 var 绑定的是 cost，即 var="cost"，所以 EL 表达式中要写成 \${cost.id}。如果用的是 Struts2 标签进行循环，则 EL 表达式写 \${id} 即可。

step6：将 cost\_list.jsp 中的循环改为 Struts2 标签和 OGNL 表达式：（记得导入 Struts2 标签）

```

<s:iterator value="costList">
 <tr><td><s:property value="id" /></td>
 <td><s:property value="name" /></td>
 <td><s:property value="baseDuration" />小时</td>
 <td><s:property value="baseCost" />元</td>
 <td><s:property value="unitCost" />元/小时</td>
 <td><s:date name="creaTime" format="yyyy-MM-dd hh:mm:ss"/></td>
 <td><s:property value="startTime" /></td>
 <td><s:property value="statusOptions[status]" /></td>
 <!--<s:if test="status==1">暂停</s:if><s:else>开通</s:else>
 这些状态也是数据，写这不好，所以采取定义到 Action 或 DAO 中-->
 <td><input type="button" value="启用" class="btn_start" />
 <input type="button" value="修改" class="btn_modify" />
 <input type="button" value="删除" class="btn_delete"
 onclick="location='delete.action?id=${id}';" /></td>
 </tr>
</s:iterator>

```

step7：将状态数据定义到 DAO 中，所以在 CostDAO 接口中添加定义

```
public Map<String, String> getCostStatusOptions();
```

step8：在 CostDAOImpl 中实现该 Map 方法

```

public Map<String, String> getCostStatusOptions() {
 Map<String, String> statusOptions;
 statusOptions=new LinkedHashMap<String, String>(); statusOptions.put("0","开通");

 statusOptions.put("1","暂停"); statusOptions.put("2","删除");
 return statusOptions;
}

```

step9: 在 ListCostAction 中添加 statusOptions 属性，并计算

```

private Map<String, String> statusOptions; get/set 方法
public String execute() throws Exception {
 statusOptions=costDAO.getCostStatusOptions();
}

```

step10: 部署，测试，地址栏输入: <http://localhost:8080/NetCTOSS/cost/list.action>，点击删除，执行成功并重定向到 list.action

## 4.5 NetCTOSS 项目：基于 StreamResult 生成验证码

struts-default.xml 中：

```
<result-type name="stream" class="org.apache.struts2.dispatcher.StreamResult"/>
```

step1: 在 com.tarena.netctoss.util 包下，新建 VerifyCodeUtils 类，用于生成验证码

```

private String code; private byte[] codeArr;
private static char[] seq = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q',
'R',
'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' };
public String getCode() { return code; }
public byte[] getCodeArr() { return codeArr; }
public void generate(int width, int height, int num) throws Exception {
 Random r = new Random();
 // 图片的内存映像
 BufferedImage image = new BufferedImage(width, height,
 BufferedImage.TYPE_INT_RGB);
 // 获得画笔对象
 Graphics g = image.getGraphics();
 g.setColor(new Color(r.nextInt(255), r.nextInt(255), r.nextInt(255)));
 g.fillRect(0, 0, width, height); g.setColor(new Color(0, 0, 0));
 // 用于存储随即生成的验证码
 StringBuffer number = new StringBuffer();
 // 绘制验证码
 for (int i = 0; i < num; i++) {
 g.setColor(new Color(r.nextInt(255), r.nextInt(255), r.nextInt(255)));
 int h = (int) ((height * 60 / 100) * r.nextDouble() + (height * 30 / 100));
 g.setFont(new Font(null, Font.BOLD | Font.ITALIC, h));
 String ch = String.valueOf(seq[r.nextInt(seq.length)]);
 number.append(ch); g.drawString(ch, i * width / num * 90 / 100, h); }
 // 绘制干扰线
 for (int i = 0; i <= 12; i++) {
 g.setColor(new Color(r.nextInt(255), r.nextInt(255), r.nextInt(255)));

```

```

 g.drawLine(r.nextInt(width), r.nextInt(height), r.nextInt(width), r.nextInt(height)); }
code = number.toString();
// 字节数组输出流，向字节数组中输出信息
ByteArrayOutputStream baos = new ByteArrayOutputStream();
JPEGImageEncoder encoder = JPEGCodec.createJPEGEncoder(baos); //压缩成 JPEG
// 图片的二进制信息输出到了内存中
encoder.encode(image); codeArr = baos.toByteArray(); }
}

```

step2：在 com.tarena.netctoss.action 包下，创建 VerifyCodeAction 类，由于要使用 session，所以继承 BaseAction

```

public class VerifyCodeAction extends BaseAction {
 private InputStream codeInputStream; // StreamResult 只能输出 InputStream 类型属性值
 private String code; // 输入用户提交的验证码
 private boolean ok = false; // 输出验证的结果
 /**
 * 注意：boolean 类型的变量自动生成 get/set 方法后，get 方法名叫 isXXX()，而不是 getXXX()！如此例中的 ok，则是 isOk()，不是 getOk()。如果是 Boolean 类型（字母 b 大写），则自动生成的 get 方法为 getXXX() */
 public String code() throws Exception { // 生成验证码
 VerifyCodeUtils verifyCodeUtils = new VerifyCodeUtils();
 verifyCodeUtils.generate(80, 30, 4);
 String code = verifyCodeUtils.getCode();
 byte[] codeArr = verifyCodeUtils.getCodeArr();
 session.put(VISITCODE_KEY, code); System.out.println(code);
 codeInputStream = new ByteArrayInputStream(codeArr);
 return "success";
 }
}

```

step3：在 Constants 接口中，添加 KEY 常量

```
public static final String VISITCODE_KEY = "com.tarena.netctoss.verifycode.key";
```

step4：在 struts-login.xml 中添加获取验证码的配置

```

<action name="code" class="com.tarena.netctoss.action.VerifyCodeAction" method="code">
 <result name="success" type="stream"><!-- StreamResult 负责输出二进制信息 -->
 <!-- 给 StreamResult 对象的 inputStream 属性赋值。inputName 是一个 OGNL 表达式，该表达式可以从 VS 中获取一个 InputStream 对象。而 StreamResult 就是从这个 inputStream 对象中读取出要输出的二进制信息。codeInputStream 与 VerifyCodeAction 中的属性相对应 -->
 <param name="inputName">codeInputStream</param>
 <param name="contentType">image/jpg</param> <!-- 输出类型为图片 -->
 </result>
 <result name="fail">/WEB-INF/jsp/login_form.jsp</result>
</action>

```

step5：修改 login\_form.jsp 中的验证码

```

<tr><td class="login_info">验证码： </td>
 <input id="verifycode" name="code" type="text" class="width70" /></td>
<td><!-- 放在 img 标签中，火狐浏览器就知道这里是一个图片，而不是下载 -->
 </td><!--这么写的效果为：点击图片更换验证码-->
<td></td></tr>

- ◆ 注意事项： onclick="this.src='code.action?#'+(new Date()).getTime();"， 中间的“#”号要加，否则点击图片更换验证码时，会报异常： java.lang.NumberFormatException。
- step6：部署，测试，地址栏输入：<http://localhost:8080/NetCTOSS/login/loginform.action>，验证码可正常显示，点击图片也会更换



4.6 NetCTOSS 项目：基于 JsonResult 进行验证码检验

Struts2 的 struts-default.xml 中没有提供 json 类型的 Result，所以要扩展 Result。

step1：导入 struts2-json-plugin-2.1.8.jar …… 等一系列包到工程中

step2：该 jar 包中有个 struts-plugin.xml，里面添加了 json 的配置，并且继承了 struts-default.xml，所以以后的配置文件只要继承 json-default 包，也就有了 struts-default 中的配置

```
<package name="json-default" extends="struts-default">
    <result-types>
        <result-type name="json" class="org.apache.struts2.json.JSONResult"/>
    </result-types>
    <interceptors>
        <interceptor name="json" class="org.apache.struts2.json.JSONInterceptor"/>
    </interceptors>
</package>
```

step3：修改 struts-login.xml，使包继承 json-default

```
<package name="login" namespace="/login" extends="json-default">
```

step4：在 com.tarena.netctoss.action 包下的 VerifyCodeAction 中，添加验证方法

```
public String verify() //进行验证
    String c=(String)session.get(VERIFYCODE_KEY);
    if(c!=null){    ok=c.equalsIgnoreCase(code.trim());    }    return "success";
```

step5：在 struts-login.xml 中配置验证

```
<action name="verifycode" class="com.tarena.netctoss.action.VerifyCodeAction"
    method="verify"><!--获取、验证都定义在一个 Action 中，但调用不同的方法-->
    <!-- JsonResult 默认将会把 VS 的栈顶（当前的 Action）做成 JSON 字符串返回 -->
    <result name="success" type="json">
        <!-- JsonResult 对象的 root 属性是一个 OGNL 表达式，通过该表达式可以从 VS
            中获取一个对象， JsonResult 将把这个对象做成 JSON 字符串并返回 -->
        <param name="root">ok</param><!--配置参数后只把该对象中的 ok 属性输出
    -->
    </result>
</action>
```

- ◆ 注意事项： json 的三种使用方式：
 - ❖ 将一个属性以 json 格式返回

```
<result type="json">
```

```
<param name="root">属性名</param>
```

```
</result>
```

- ❖ 将所有属性以 json 格式返回

```
<result type="json"></result>
```

- ❖ 将一部分属性以 json 格式返回

```
<result type="json">
```

```
<param name="includeProperties">属性 1,属性 2,...</param>
```

```
</result>
```

step6：部署，测试，地址栏输入：<http://localhost:8080/NetCTOSS/login/code.action>，先产生一个验证码，然后地址栏再输入：<http://localhost:8080/NetCTOSS/login/verifycode.action?code=1234>，1234 是随便写的，此时页面返回 false。



step7：在 WebRoot 目录下新建 js 文件夹，将 jquery-1.4.1.min.js（jquery 框架）放入。同时新建 validation.js 文件，在其中添加自定义验证方法，可参考 jQuery 笔记 5.12 案例

```
$.fn.required=function(errorMsg,errorCtn){//必须填写
    var value=this.val();
    if(value!=null&&value.length>0){      $(errorCtn).text("");      return true;
    }else{      $(errorCtn).text(errorMsg);      return false;  }
}
$.fn.remote=function(url,errorMsg,errorCtn){//与服务器间的数据进行验证
    var b=false;  var value=this.val();  var name=this.attr("name");
    $.ajax( {  url:url+"?"+name+"="+value,  dataType:"json",  async:false,
        success:function(data){
            if(data){  $(errorCtn).text("");  b=true;
            }else{  $(errorCtn).text(errorMsg);  }
        }
    );
    return b;
}
```

step8：在 login_form.jsp 中引入这两个 js 文件，并编写验证代码

```
<script language="javascript" src=".//js/jquery-1.4.1.min.js"></script>
<script language="javascript" src=".//js/validation.js"></script>
<script language="javascript" type="text/javascript">
    function validateForm(){//普通方法
        var b1=$("#admincode").required("帐号必须填写","#msg_admincode");
        /* 这么写也行，传进去的是字符串 */
        var b2=$("#password").required("密码必须填写",$("#msg_password"));
        /* 这种方式，传进去的是对象，虽然 validation.js 中的方法也写了$,
但这样写也可以，因为设计者无法确定传进去的是什么，所以加个$，更灵活 */
        var b3=$("#verifycode").remote("verifycode.action","验证码错误~",
                                         $("#msg_verifycode"));
        return b1&&b2&&b3;
    }
    $(function(){//一加载页面就执行的方法
        $("#submit").click(function(){  var b = validateForm();
    
```

```

        if(b) {      $("#loginForm").submit();    }           });
    
```

step9：在 login_form.jsp 中添加 id 属性，修改登录按钮使之采用脚本验证后再提交

```

<form id="loginForm" action="login.action" method="post">
    <input id="admincode" name="admin.adminCode" .....
        <span id="msg_admincode" class="required"></span>
    <input id="password" name="admin.password" .....
        <span id="msg_password" class="required"></span>
    <input id="verifycode" name="code" type="text" .....
        <span id="msg_verifycode" class="required"> .....
    <a id="submit" href="javascript:;" ></a> .....
</form>

```

step10：部署，测试，地址栏输入：<http://localhost:8080/NetCTOSS/login/loginform.action>，点击登录按钮，出现如下提示，只有都填写，且正确才可进入主页



step11：进入主页后发现地址栏还是：<http://localhost:8080/NetCTOSS/login/login.action>，且点 F5 刷新，会弹出“确认重新提交表单”的框，这不符合实际业务需求，因此要修改 struts-login.xml 中的配置

step12：修改 3.4 案例 step13 中的 struts-login.xml，使用 redirectAction 类型的 Result

```

<action name="login" class="com.tarena.netctoss.action.LoginAction">
    <result name="success" type="redirectAction">
        <param name="namespace"/>/main</param>
        <param name="actionName">main</param>
    </result>
    <result name="fail">/WEB-INF/jsp/login_form.jsp</result>
</action>

```

step13：添加 struts-main.xml

```

<package name="main" namespace="/main" extends="struts-default">
    <action name="main">
        <result name="success">/WEB-INF/jsp/index.jsp</result>
    </action>
</package>

```

step14：在 struts.xml 中引入 struts-main.xml

```

<include file="struts-main.xml" />

```

step15：部署，测试，当正确填写用户名、密码和验证码后，进入主页。此时，地址栏已经变为 <http://localhost:8080/NetCTOSS/main/main.action>，并且按 F5 刷新，不会出现弹框。

4.7 NetCTOSS 项目：添加资费模块中的验证资费名是否重复

实现方式也是利用 json。

step1：在 CostDAO 接口中定义 findByName

```
public Cost findByName(String name) throws DAOException;
```

step2: 在 CostDAOImpl 中实现该方法

```
private static String findByName= "select ID, NAME, BASE_DURATION, BASE_COST,  
UNIT_COST, CREATIME, STARTIME, STATUS, DESCR,COST_TYPE from COST_CHANG  
where NAME=?";  
  
public Cost findByName(String name) throws DAOException {  
    Connection con = null;  
    try { con = DBUtils.openConnection();  
        PreparedStatement stmt = con.prepareStatement(findByName);  
        stmt.setString(1, name);      ResultSet rs = stmt.executeQuery();  
        Cost cost=null; if (rs.next()) { cost=toCost(rs); } return cost;  
    } catch (Exception e) { e.printStackTrace();  
        throw new DAOException("访问异常", e);  
    } finally{ DBUtils.closeConnection(con); }
```

step3: 测试 findByName, 在 TestCostDAO 中添加方法

```
@Test  
public void testFindByName() throws Exception {  
    CostDAO costDAO=DAOFactory.getCostDAO();  
    Cost cost=costDAO.findByName("包月");  
    System.out.println(cost.getId()+" "+cost.getName()); }
```

step4: 在 com.tarena.netctoss.action.cost 包中, 新建 ValidateCostNameAction

```
//private String name;//不用这个了, 否则后面的 JS 报错!  
private Cost cost=new Cost(); private boolean ok=false; .....各自 get/set 方法  
private CostDAO costDAO=DAOFactory.getCostDAO();  
public String execute() throws DAOException {  
    System.out.println(cost.getName());  
    cost1=costDAO.findByName(cost.getName());  
    if(cost1==null){ ok=true; } return "success"; }
```

- ◆ 注意事项: 报错的原因是因为页面中, name 属性的值为 “cost.name”, 如果 action 中的属性用 name 接收, 则无法接收。报错信息如下:

```
Error setting expression 'cost.name' with value '[Ljava.lang.String;@54996c'  
ognl.OgnlException: target is null for setProperty(null, "name" ....
```

step5: 在 struts-cost.xml 中添加配置, 并让 cost 包继承 json-default

```
<action name="validateName"  
       class="com.tarena.netctoss.action.cost.ValidateCostNameAction">  
    <result name="success" type="json">  
        <param name="root">ok</param>  
    </result>  
</action>
```

step6: 部署, 测试, 地址栏输入: <http://localhost:8080/NetCTOSS/cost/validateName.action?cost.name=包月>, 页面显示 true, 且控制台输出 “????”, 这是由于中文乱码产生的问题! 如果有个资费名称不包含中文的, 则页面显示 false。对于中文乱码问题, 后面的 JS 脚本会解决

step7: 将静态页面资费添加拷贝到 cost 文件夹中, 改为 cost_add.jsp

```

<script language="javascript" src="../js/jquery-1.4.1.min.js"></script>
<script language="javascript" src="../js/validation.js"></script>
<script language="javascript" type="text/javascript">
    function validateCostForm(){
        b=$('#costName').remote("validateName.action","资费名称已被占用",
                               $('#msg_costName'));
    }
</script>
<input id="costName" name="cost.name" onblur="validateCostForm();" type="text" .....
<div id="msg_costName" class="validate_msg_short"> .....

```

step8：修改 validation.js 中的 remote 函数，用于解决中文乱码问题，可与 4.6 案例 step7 比较

```

$.fn.remote=function(url,errorCtn){ //与服务器间的数据进行验证
    var b=false;      var value=this.val();      var name=this.attr("name");
    var params =name+"="+value;
    $.ajax( { url:url,   data:params,   dataType:"json",   type:"post",   async:false,
              success:function(data){
                  if(data){ $(errorCtn).text("");   b=true;
                  } else{   $(errorCtn).text(errorMsg);   }
              });
    return b;
}

```

- ◆ 注意事项：用 post 方式不用考虑中文乱码，因为 \$.ajax() 函数默认使用 utf-8，而且 Struts2 框架也默认使用 utf-8。如果 url:url+"?" + name +"=" + value，则还是乱码。

step9：在 struts-cost.xml 中添加配置

```

<action name="addform">
    <result name="success">/WEB-INF/jsp/cost/cost_add.jsp</result>
</action>

```

step10：部署，测试，地址栏输入：<http://localhost:8080/NetCTOSS/cost/addform.action>，输入“包月”，则显示资费名已被占用

4.8 自定义一个 Result

此例模拟 json Result。

step1：新建工程 struts_result，导入一系列包，并在 result 包中，新建 SomeResult

```

public class SomeResult implements Result {
    private String root;      .....get/set 方法
    public void execute(ActionInvocation arg0) throws Exception {
        ValueStack vs=arg0.getStack(); //获得 VS      Object obj;
        if(root==null){   obj=vs.findValue("top"); //从栈顶获得
        } else{   obj=vs.findValue(root);   }
        JSONObject json=JSONObject.fromObject(obj); //json 的原理
        String jsonstr=json.toString();
        HttpServletResponse response = ServletActionContext.getResponse();
        PrintWriter out=response.getWriter();      out.println(jsonstr);   }
}

```

step2：在 entity 包中，新建 Emp

```

private String name;      private double salary;      .....get/set 方法

```

```
public Emp(String name, double salary) { this.name = name; this.salary = salary; }
```

step3: 在 action 包中，新建 TestAction

```
private Emp emp; private String name; .....get/set 方法  
public String execute() throws Exception{  
    emp=new Emp("chang",10000); name="java"; return "success"; }
```

step4: 新建 struts.xml，进行配置

```
<package name="test" namespace="/" extends="struts-default">  
    <result-types><!--声明自定义的 result-->  
        <result-type name="some" class="result.SomeResult"></result-type>  
    </result-types>  
    <action name="test" class="action.TestAction">  
        <result name="success" type="some"><!-- 模拟 json -->  
            <param name="root">emp</param>  
            <!-- 去掉上面的参数，会返回 json 字符串：  
                {"name":"java","emp":{"name":"chang","salary":10000}}  
                加上只返回 emp 对象的字符串：  
                {"name":"chang","salary":10000} -->  
        </result>  
    </action>  
</package>
```

step5: 部署，测试，地址栏输入：http://localhost:8080/struts_result/test.action，显示结果为：

{ "name": "chang", "salary": 10000 }，如果把配置中的参数去掉，则显示为：

{ "name": "java", "emp": { "name": "chang", "salary": 10000 } }

一百一十九、Struts2 标签

LICHOO

5.1 A 开头的标签

```
<s:a href=""></s:a>: 超链接, 类似于 html 里的<a></a>
<s:action name=""></s:action>: 执行一个 view 里面的一个 action
<s:actionerror/>: 如果 action 的 errors 有值那么显示出来
<s:actionmessage/>: 如果 action 的 message 有值那么显示出来
<s:append></s:append>: 添加一个值到 list, 类似于 list.add();
<s:autocompleter></s:autocompleter>: 自动完成<s:combobox>标签的内容, 这个是 ajax
```

5.2 B 开头的标签

```
<s:bean name=""></s:bean>: 类似于 struts1.x 中的, JavaBean 的值
```

5.3 C 开头的标签

```
<s:checkbox></s:checkbox>: 复选框
<s:checkboxlist list=""></s:checkboxlist>: 多选框
<s:combobox list=""></s:combobox>: 下拉框
<s:component></s:component>: 图像符号
```

5.4 D 开头的标签

```
<s:date/>: 获取日期格式
<s:datetimepicker></s:datetimepicker>: 日期输入框
<s:debug></s:debug>: 显示错误信息
<s:div></s:div>: 表示一个块, 类似于 html 的<div></div>
<s:doubleselect list="" doubleName="" doubleList=""></s:doubleselect>: 双下拉框
```

5.5 E 开头的标签

```
<s:if test="判断条件">满足条件时执行此处</s:if>
<s:elseif test="判断条件">满足条件时执行此处</s:elseif>
<s:else></s:else>: 这 3 个标签可一起使用, 表示条件判断
```

5.6 F 开头的标签

```
<s:fielderror></s:fielderror>: 显示文件错误信息
<s:file></s:file>: 文件上传
<s:form action="" theme="" method="" namespace=""></s:form>: 类似于 html 中的<form>
```

◆ 注意事项:

- ❖ <s:form>标签中主题 theme 默认为 xhtml (以表格布局, 可查看源代码, 自动添加了 table 表格), 如果不喜欢该主题, 就设置 theme="simple", 此时就可以使用自定义的样式了。
- ❖ Struts2 的<s:form>标签可以自动将 Action 的属性填写到页面 form 表单中, 但密码除外!

5.7 G 开头的标签

<s:generator separator="" val=""></s:generator>： 和<s:iterator>标签一起使用

5.8 H 开头的标签

<s:head>： 在<head></head>里使用， 表示头文件结束

<s:hidden></s:hidden>： 隐藏值

5.9 I 开头的标签

<s:i18n name=""></s:i18n>： 加载资源包到值堆栈

<s:include value=""></s:include>： 包含一个输出， servlet 或 jsp 页面

<s:inputtransferselect list=""></s:inputtransferselect>： 获取 form 的一个输入

<s:iterator></s:iterator>： 用于遍历集合

5.10 L 开头的标签

<s:label></s:label>： 只读的标签

5.11 M 开头的标签

<s:merge></s:merge>： 合并遍历集合出来的值

5.12 O 开头的标签

<s:optgroup></s:optgroup>： 获取标签组

<s:optiontransferselect doubleList="" list="" doubleName=""></s:optiontransferselect>： 左右选择框

5.13 P 开头的标签

<s:param></s:param>： 为其他标签提供参数

<s:password size="" maxlen="" readonly=""></s:password>： 密码输入框

<s:property/>： 得到 VS 栈顶的元素

<s:push value=""></s:push>： value 的值 push 到栈中， 从而使 property 标签能够获取 value 的属性

◆ 注意事项： size 属性规定输入字段的宽度（即是文本框只显示 size 个字符大小的宽度）， 由于 size 属性是一个可视化的设计属性， 所以我们可以使用 CSS 中的 width 来代替它。

5.14 R 开头的标签

<s:radio list="OGNL 需要迭代的集合" listValue="用作于每一个选项的提示"

listKey="用作于每一个要提交的值"></s:radio>： 单选按钮

<s:reset></s:reset>： 重置按钮

5.15 S 开头的标签

<s:select list="OGNL 需要迭代的集合" listValue="用作于每一个选项的提示"

listKey="用作于每一个要提交的值"></s:select>： 单选框

<s:set name=""></s:set>： 赋予变量一个特定范围内的值

<s:sort comparator=""></s:sort>: 通过属性给 list 分类
<s:submit></s:submit>: 提交按钮
<s:subset></s:subset>: 为遍历集合输出子集

LICHOO

5.16 T 开头的标签

<s:tabbedPanel id=""></s:tabbedPanel>: 表格框
<s:table></s:table>: 表格
<s:text name=""></s:text>: I18n 文本信息
<s:textarea rows="" cols=""></s:textarea>: 文本域输入框
<s:textfield size="" maxlength="" readonly=""></s:textfield>: 文本输入框
<s:token></s:token>: 拦截器
<s:tree></s:tree>: 树
<s:treenode label=""></s:treenode>: 树的结构

5.17 U 开头的标签

<s:updownselect list=""></s:updownselect>: 多选择框
<s:url></s:url>: 创建 url

- ◆ 注意事项：
 - ❖ 以上标记，有的可以用单标记，有的也可用双标记。
 - ❖ 以上标记，有的属性并未写全！在页面中的标签内，使用“Alt + /”可显示该标签所有的属性。

5.18 所有标签都具备的属性

1) label: input 类型的按钮，不能用 label 设置按钮上的文本，只能用 value。在主题 theme 为 simple 下，label、tooltip 不起作用。

- | | | |
|--|----------------------------|------------|
| 2) labelposition | 3) required | 4) tooltip |
| 5) tooltipIconPath | 6) cssClass: html 中的 class | |
| 7) cssStyle: html 中的 style | 8) value: 用于获取值 | |
| 9) name: 在 Struts2 标签中，用于提交值或获取值；在普通 HTML 标签中用于提交值 | | |

- ◆ 注意事项：
 - ❖ Struts2 中的标签 name 属性有双重作用，获取值，提交值，前提是获取和提交值的名字一致，如果不一致则需要分别写 value 属性和 name 属性。
 - ❖ HTML 标签中如若有回显效果，也要提交值，则必须写 value 和 name 属性。

5.19 案例：常用标签

step1：新建工程 struts04，导入 Struts2 核心包，配置前端控制器

step2：在 com.tarena.action 包下，新建 BaseAction，并继承 RequestAware

```
public class BaseAction implements RequestAware{
    protected Map<String, Object> request;
    public void setRequest(Map<String, Object> request) {
        this.request = request;
    }
}
```

step3：在 com.tarena.entity 包下，新建 Favor 实体类

```
private int id;      private String name;      .....各自 get/set 方法
public Favor(int id, String name) {      this.id = id;      this.name = name; }
```

step4: 在 com.tarena.dao 包下，新建 FavorDAO 类，用于模拟访问数据库

```
public List<Favor> findAll(){  
    List<Favor> list = new ArrayList<Favor>();  
    list.add(new Favor(1,"睡觉"));    list.add(new Favor(2,"上网"));  
    list.add(new Favor(3,"吃喝"));    list.add(new Favor(4,"编程"));    return list;    }
```

step5: 在 com.tarena.action 包下，新建 FormAction

```
private String name;//output      private String password;      private int age;  
private String sex;              private String desc;  
private boolean marry;//单个 checkbox 选中  
private int[] favorsKey;//多个 checkbox 选中  
private int favor;// select 选中  
public String execute() {//模拟从数据取出原有数据  
    name = "常";      password = "123";      age = 24;      sex = "M";  
    desc = "我是一个好人";      marry = true;  
    FavorDAO favorDao = new FavorDAO();//获取数据  
    List<Favor> list = favorDao.findAll();      request.put("favors", list);//放入 request 中  
    favorsKey = new int[] { 1, 4 };//设置哪些 checkbox 选中  
    favor = 4;//设置 select 选中      return "success";    }
```

step6: 在 WEB-INF 中创建 jsp 文件夹，并将 form.jsp 放入

```
<s:form action="form" theme="simple">  
    姓名: <s:textfield name="name" maxlength="8" size="30"></s:textfield><br/>  
    密码: <s:password name="password"></s:password><br/>  
    年龄: <s:textfield name="age"></s:textfield><br/>  
    性别: <s:radio list="#{'M':'男','F':'女'}" name="sex"></s:radio><br/>  
    婚姻状况: <s:checkbox name="marry"></s:checkbox><br/>  
    兴趣爱好: <s:checkboxlist list="#request.favors" listKey="id"  
        listValue="name" name="favorsKey"></s:checkboxlist><br/>  
    select: <s:select list="#request.favors" listKey="id" listValue="name" name="favor">  
        </s:select><br/>  
    简介: <s:textarea name="desc" cols="10" rows="3"></s:textarea><br/>  
</s:form>
```

step7: 创建 struts.xml 文件，并进行配置

```
<struts>  
    <package name="uitag" extends="struts-default">  
        <action name="form" class="com.tarena.action.FormAction">  
            <result>/WEB-INF/jsp/form.jsp</result>  
        </action>  
    </package>  
</struts>
```

step8: 部署，测试，地址栏输入：http://localhost:8080/struts04/form.action，其中 password 标签没有回显效果：

LICHOOL

姓名:

密码:

年龄:

性别: 男 女

婚姻状况:

兴趣爱好: 睡觉 上网 吃喝 编程

select:

我是第一个
简介:

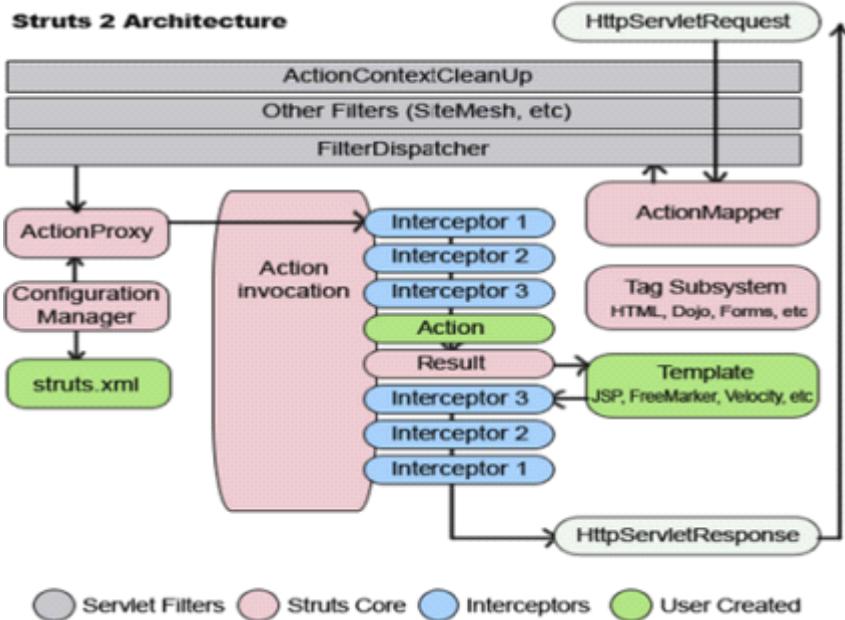
step9: 不用 Struts2 标签, 则以前的方式为:

```
<%@taglib uri="/struts-tags" prefix="s"%>
<form><!-- 以前的方式 -->
    Name: <input type="text" name="" value="${emp.name}" /><br />
    Salary: <input type="text" name="" value="${emp.salary}" /><br />
    Sex: <s:if test="emp.sex.equals('m')">
        <input type="radio" name="sex" value="m" checked="checked">Male
        <input type="radio" name="sex" value="f">Female
    </s:if>
    <s:else>
        <input type="radio" name="sex" value="m">Male
        <input type="radio" name="sex" value="f" checked="checked">Female
    </s:else>
</form>
```

一百二十、拦截器

LICHOO

6.1 Struts2 详细流程图



6.2 拦截器的作用

拦截器适合封装一些共通处理，便于重复利用。例如：请求参数给 Action 属性，日志的记录，权限检查，事务处理等。拦截器是通过配置方式调用，因为使用方法比较灵活，便于维护和扩展。

6.3 拦截器的常用方法

- 1) 如果不调用下面的两种方法，那么就不会调用 Action，也不会调用后面的 Interceptor，拦截器中的 return 的 String 觉定了最后的 Result（极端情况下）。
- 2) arg0.invoke(); 调用 Action，也包括 Result，拦截器中的 return 的内容无效。
- 3) arg0.invokeActionOnly(); 调用 Action，不包括后面的 Interceptor 和 Result，拦截器中的 return 的 String 决定了最后的 Result。
- 4) 在拦截器中，如何调用底层 API：
 - ① 可调用 ActionInvocation 对象的 getStack 方法获取 VS。
 - ② 可调用 ServletActionContext 的静态方法获取 Servlet API，如 getResponse 获取 response。

6.4 自定义拦截器步骤

step1：编写拦截器组件，组件类实现 **Interceptor** 接口，实现约定的 **interceptor** 方法。在该方法中添加自定义的共通处理

```
public class DemoInterceptor implements Interceptor{  
    public String interceptor(ActionInvocation ai){  
        //拦截器前部分处理  
        ai.invoke(); //执行 action 和 result，或 ai.invokeActionOnly()  
    }  
}
```

```
//拦截器后续处理
```

```
}
```

step2：将拦截器注册给 Struts2 框架，在 struts.xml 中注册（添加该拦截器的声明）

```
<package>
    <interceptors>
        <interceptor name="名称" class="实现类" />
        //...其他 interceptor
    </interceptors>
</package>
```

step3：使用拦截器组件（添加该拦截器的引用）

1) 方式一：定义默认拦截器

```
<default-interceptor-ref name="拦截器名" />
```

◆ 注意事项：默认(不写)情况下，一个 Action 在执行时，会默认调用 defaultStack 拦截器栈。

2) 方式二：显式指定调用哪个拦截器

```
<action>
    <interceptor-ref name="拦截器名"/>
    //...可以写多个
</action>
```

◆ 注意事项：当指定 Action 调用的拦截器后，默认的 defaultStack 将不再执行！

6.5 Struts2 内置的拦截器

在 struts-default.xml 中可查看。

```
<interceptors>
    <interceptor name="cookie" class="org.apache.struts2.interceptor.CookieInterceptor"/>
    <interceptor name="i18n" class="com.opensymphony.xwork2.interceptor.I18nInterceptor"/>
    <interceptor name="token" class="org.apache.struts2.interceptor.TokenInterceptor"/>
    <interceptor name="store" class="org.apache.struts2.interceptor.MessageStoreInterceptor" />
    <interceptor name="checkbox" class="org.apache.struts2.interceptor.CheckboxInterceptor" />
    ...
    ....
    ....
<interceptor-stack name="defaultStack">
    <interceptor-ref name="exception"/>
    <interceptor-ref name="alias"/>
    <interceptor-ref name="servletConfig"/>
    <interceptor-ref name="chain"/>
    ...
    ....
</interceptor-stack>
    ....
    ....
</interceptors>
    ....
    ....
<default-interceptor-ref name="defaultStack"/>
```

6.6 案例：拦截器入门

step1：新建工程 struts05，导入 Struts2 核心包，配置前端控制器

step2：在 com.tarena.intercept 包下，新建 SomeInterceptor 类，并实现 Interceptor

```
public class SomeInterceptor implements Interceptor {
    private String name; // 设置拦截器参数 .....get/set 方法
    public void destroy() { System.out.println("SomeInterceptor.destroy()..."); }
    public void init() { System.out.println("SomeInterceptor.init()..."); }
    public String intercept(ActionInvocation arg0) throws Exception {
        /** 自定义拦截器，一般会用到底层的 request、response、session、servletContext，但是在 Action 里不允许出现底层的 API，就是写纯业务代码！ */
        /** ActionContext ctx=arg0.getInvocationContext();
            ValueStack stack=ctx.getValueStack(); // 拿到 VS
            // stack.setValue("ognl", obj); // 放
            // Object obj=stack.findValue("ognl"); // 取
            stack.setValue("name", "chang");
            HttpServletRequest request = ServletActionContext.getRequest(); // 获得 request 对象
            System.out.println(request);
            HttpServletResponse response = ServletActionContext.getResponse(); // 获得 response
            System.out.println(response);
            ServletContext application = ServletActionContext.getServletContext();
            System.out.println(application);
            */
            System.out.println("SomeInterceptor:" + name);
            // ActionInvocation 封装了 Action 和 Result 的整个调用流程
            System.out.println("SomeInterceptor.intercept()...before"); // 在调用 Action 之前的操作
            arg0.invoke(); // String result=arg0.invokeActionOnly(); System.out.println(result);
            System.out.println("SomeInterceptor.intercept()...after"); // 调用 Action 之后的操作
            // return "success"; // 若不调用 invoke 和 invokeActionOnly 方法，则当前是极端情况。
            // 则跟 Action 没关系了，由拦截器决定返回哪个页面
            return "error"; // 当调用 invoke() 方式时，return 什么都行
    }
}
```

◆ 注意事项：

- ❖ invoke(): 调用 Action，同时已经决定了 Result，所以 Result 不能再换了，由 Action 决定 Result。同时继续调用后面的拦截器。
- ❖ invokeActionOnly(): 调用 Action，但是还没有决定 Result，则 Result 由拦截器说了算，同时后面的拦截器也不调用了，直接掉 Action。

step3：在 com.tarena.action 包下，新建 SomeAction 类

```
private String name; .....get/set 方法
public String execute() {
    System.out.println("Action:" + name);
    System.out.println("SomeAction.execute()..."); return "success";
}
```

step4：新建 struts.xml 文件，将拦截器注册给 Struts2 框架，指定拦截器作用的 Action

```
<package name="day05" namespace="/" extends="struts-default">
<interceptors><!-- 声明拦截器，注意和 action 的先后顺序 -->
<!-- 单个拦截器 -->
<interceptor name="some" class="com.tarena.intercept.SomeInterceptor"/>
```

```

</interceptors>
<action name="some" class="com.tarena.action.SomeAction">
    <interceptor-ref name="basicStack"/><!--引用拦截器，必须写，见注意事项-->
    <!--<interceptor-ref name="some"/> 引用拦截器，不加参数这么写 -->
    <!--前面的拦截器若调用 invokeActionOnly 则后面的拦截器都失效，见
step10-->
    <interceptor-ref name="some"><!-- 引用拦截器，加参数这么写 -->
        <param name="name">ABC</param>
    </interceptor-ref><!-- 自己定义的拦截器放里面，贴身（离 Result 近）的 -->
    <result name="success">/WEB-INF/jsp/some.jsp</result>
    <result name="error">/WEB-INF/jsp/error.jsp</result>
</action>
</package>

```

- ◆ 注意事项：
 - ❖ 上面代码中，如果不写拦截器，则默认是 defaultStack 拦截器栈，但自己写上拦截器后，系统不会再添加默认拦截器！所以自己写项目时，最少也要写 basicStack，也可写 defaultStack。一般情况下，自己定义的拦截器放里面，贴身（离 Result 近）的。
 - ❖ 自己写拦截器的目的是扩展拦截器，增加 Struts2 的功能，是通用性的，不要把业务代码也写里面，所以程序员自己写拦截器的情况很少！

step5：在 WEB-INF/jsp 下，新建 some.jsp

```
<h1>Some jsp</h1>
```

step6：部署，测试，地址栏输入：<http://localhost:8080/struts05/some.action?name=chang>，页面显示“Some jsp”，而控制台输出如下：

```

SomeInterceptor:ABC
SomeInterceptor.intercept()...before
Action:chang
SomeAction.execute()...
SomeInterceptor.intercept()...after

```

step7：在 com.tarena.intercept 包下，再创建一个拦截器 OtherInterceptor

```

public class OtherInterceptor implements Interceptor{
    public void destroy() { System.out.println("OtherInterceptor.destroy()..."); }
    public void init() { System.out.println("OtherInterceptor.init()..."); }
    public String intercept(ActionInvocation arg0) throws Exception {
        System.out.println("OtherInterceptor.intercept()...before");
        arg0.invoke();
        System.out.println("OtherInterceptor.intercept()...after");
        return "success";// 因为采取 invoke()，所以 return null 也可以
    }
}

```

step8：在 struts.xml 中添加该拦截器的声明和引用

```

<interceptors>
    <interceptor name="some" class="com.tarena.intercept.SomeInterceptor"/>
    <interceptor name="other" class="com.tarena.intercept.OtherInterceptor"/>
</interceptors>
<action name="some" class="com.tarena.action.SomeAction">

```

```

<interceptor-ref name="basicStack"/>
<!-- 前面的拦截器若调用 invokeActionOnly，则后面的拦截器都失效了，见 step10
-->
<interceptor-ref name="some">
    <param name="name">ABC</param>
</interceptor-ref>
<interceptor-ref name="other"/><!-- other 离 Result 最近，可调用 invokeActionOnly，对前面的拦截器没有影响 -->
<result name="success">/WEB-INF/jsp/some.jsp</result>
<result name="error">/WEB-INF/jsp/error.jsp</result>
</action>

```

step9：部署，测试，地址栏输入：<http://localhost:8080/struts05/some.action?name=chang>，页面显示“Some jsp”，而控制台输出如下：

```

SomeInterceptor:ABC
SomeInterceptor.intercept()...before
OtherInterceptor.intercept()...before
Action:chang
SomeAction.execute()...
OtherInterceptor.intercept()...after
SomeInterceptor.intercept()...after

```

step10：测试“前面的拦截器若调用 invokeActionOnly，则后面的拦截器都失效了”这句话。把 SomeInterceptor 拦截器中的 arg0.invoke();改为 arg0.invokeActionOnly();结果由拦截器决定

step11：由于 SomeInterceptor 拦截器的返回结果为：return "error";所以在 WEB-INF/jsp 下，新建 error.jsp

```
<h1>Error jsp</h1>
```

step12：部署，测试，地址栏输入：<http://localhost:8080/struts05/some.action?name=chang>，页面显示“Error jsp”，而控制台输出如下：即 OtherInterceptor 拦截器没被调用、失效

```

SomeInterceptor:ABC
SomeInterceptor.intercept()...before
Action:chang
SomeAction.execute()...
SomeInterceptor.intercept()...after

```

step13：把 SomeInterceptor 改回 arg0.invoke();，把 OtherInterceptor 改成 arg0.invokeActionOnly();看结果有何变化

step14：部署，测试，地址栏输入：<http://localhost:8080/struts05/some.action?name=chang>，页面显示“Some jsp”，而控制台输出如下：即修改 OtherInterceptor 拦截器为 invokeActionOnly(),则对前面的拦截器没有影响

```

SomeInterceptor:ABC
SomeInterceptor.intercept()...before
OtherInterceptor.intercept()...before
Action:chang
SomeAction.execute()...
OtherInterceptor.intercept()...after

```

6.7 拦截器栈

拦截器栈相当于一组拦截器的组合。在引用时，就把它当成一个拦截器。

例如，6.5 案例中的 struts.xml 也可这么写：

```
<package name="day05" namespace="/" extends="struts-default">
    <interceptors><!-- 注意和 action 的先后顺序，假设存在拦截器 ThreeInterceptor -->
        <interceptor name="some" class="com.tarena.intercept.SomeInterceptor"/>
        <interceptor name="other" class="com.tarena.intercept.OtherInterceptor"/>
        <interceptor name="three" class="com.tarena.intercept.ThreeInterceptor"/>
        <interceptor-stack name="someother">
            <interceptor-ref name="some" />
            <interceptor-ref name="other" />
        </interceptor-stack>
        <interceptor-stack name="all">
            <interceptor-ref name="someother" />
            <interceptor-ref name="three" />
        </interceptor-stack> -->
    </interceptors>
    <action name="some" class="com.tarena.action.SomeAction">
        <interceptor-ref name="basicStack"/>
        <interceptor-ref name="all"/>
        <result name="success">/WEB-INF/jsp/some.jsp</result>
    ....

```

6.8 fileUpload 拦截器原理

该拦截器首先会调用 commons-file-upload.jar 组件，将客户端上传的文件保存到服务器临时目录下，之后将临时目录下的文件对象给 Action 属性赋值。当 Action 和 Result 调用完毕之后，清除临时目录下的文件。因此在 Action 业务方法中，需要做文件复制，将临时文件转移到目标目录中。

6.9 案例：使用 fileUpload 拦截器实现文件上传

step1：在原有 5 个核心包的基础上添加 commons-io-1.3.2.jar 到 struts05 工程的 lib 中

step2：在 WEB-INF/jsp 下，新建 upload.jsp

```
<h1>文件上传</h1>
<form action="upload.action" method="post" enctype="multipart/form-data">
    <input type="file" name="some">
    <input type="submit" value="提交" />
</form>
```

◆ 注意事项：method 必须为 post! enctype 必须为 multipart/form-data!

step3：在 com.tarena.action 包下，新建 BaseAction，并实现 ServletContextAware，用于获取 application 对象，在后面会使用

```
public class BaseAction implements ServletContextAware {
    protected ServletContext application;
```

```

    public void setServletContext(ServletContext context) { this.application = context; }
    public String toRealPath(String path) { // 该方法功能：得到绝对路径
        return application.getRealPath(path);
    }
}

```

step4：根据 fileUpload 的实现原理，在 com.tarena.util 包下，新建 FileUtils 类，用于复制上传的文件

```

public static boolean copy(File src, File dest) {
    BufferedInputStream bis = null;    BufferedOutputStream bos = null;
    try { bis = new BufferedInputStream(new FileInputStream(src));
        bos = new BufferedOutputStream(new FileOutputStream(dest));
        byte[] bts = new byte[1024];
        int sum = -1;
        while ((sum = bis.read(bts)) != -1) {    bos.write(bts, 0, sum);    }
        return true;
    } catch (Exception e) {    e.printStackTrace();    return false;
    } finally { if (bis != null) { try { bis.close(); } catch (IOException e) {
            e.printStackTrace();        }    }
        if (bos != null) { try { bos.close(); } catch (IOException e) {
            e.printStackTrace();        }    }    }
}

```

step5：在 com.tarena.action 包下，新建 UploadAction，并继承 BaseAction

```

public class UploadAction extends BaseAction{
    private File some;//临时文件对象，属性名和表单提交写的名要相同！
    private String someFileName; //原文件名， XXXFileName 是固定写法，自动获得上传时的文件名
    private String someContentType; //原文件类型
    private String filePath;//存放文件的路径
    .....各自 get/set 方法
    public String execute()//测试文件大小超出范围，这的输出要注释掉，否则空指针
        System.out.println(some);           System.out.println(some.length());
        System.out.println(someFileName);   System.out.println(someContentType);
        if(some == null){      return "error";      }
        String fileName = "file_" + System.currentTimeMillis()
            + someFileName.substring(someFileName.lastIndexOf("."));
        System.out.println("fileName:" + fileName);//随机生成上传文件名
        filePath = "upload/" + fileName;//上传文件的相对路径，页面的链接使用
        //上传文件的绝对路径，写文件时使用， toRealPath()方法定义于 BaseAction 中
        String realFilePath = toRealPath(filePath);
        System.out.println("realFilePath ." + realFilePath);
        FileUtils.copy(some, new File(realFilePath));//从缓存中读取图片文件
        //IOUtils.copy(bis, bos);//commons.io.IOUtils 提供的复制
        return "success";
    }
}

```

step6：在 WEB-INF/jsp 中新建 ok.jsp 和 error.jsp

1) ok.jsp

```

<h1>文件上传成功</h1>


```

2) error.jsp

```
<h1>文件上传失败，文件太大了。请<a href="fileform.action">重试</a></h1>
```

step7：在 struts.xml 中添加配置

```
<action name="fileform">
    <result>/WEB-INF/jsp/upload.jsp</result>
</action>
<action name="upload" class="com.tarena.action.UploadAction">
    <interceptor-ref name="fileUpload"><!--Struts2 带的拦截器，注意和 basicStack 的顺序-->
        <param name="maximumSize">30000</param><!--单位字节-->
    </interceptor-ref>
    <interceptor-ref name="basicStack" /><!-- 它不包含 fileUpload 拦截器 -->
        <result name="success">/WEB-INF/jsp/ok.jsp</result>
        <result name="error">/WEB-INF/jsp/error.jsp</result>
    </interceptor-ref>
</action>
```

step8：部署，测试，地址栏输入：<http://localhost:8080/struts05/fileform.action>，上传图片大小小于 30KB 的，则显示如下：

文件上传成功



step9：如果上传图片太大，则显示如下（记得把 UploadAction 中的输出语句注释掉，否则报空指针异常）：

文件上传失败，文件太大了。请[重试](#)

6.10 NetCTOSS 项目：登录检查拦截器

step1：在 com.tarena.netctoss.interceptor 包下，新建 SessionValidateInterceptor 拦截器类，该拦截器的作用是：从 session 中取 KEY 为 “com.tarena.netctoss.admin.key” 的 value，如果取到的 value 是 null，说明该用户没有登录，返回到登录页面。

```
public class SessionValidateInterceptor implements Interceptor {
    private String key;      private String errorResult;//设置这两个参数，是为了通用性
    .....各自 get/set 方法
    public void destroy() { }
    public void init() { }
    public String intercept(ActionInvocation arg0) throws Exception {
        if(key==null){//拦截器有通信性，当用户没有配置 key 时，应该是拦截器失效，而不是空指针等，灾难性错误（key 的值会在 xml 文件中配置）。
            arg0.invoke();      return null;      }
        HttpSession session = ServletActionContext.getRequest().getSession();
        Object value=session.getAttribute(key);
        if(value == null){      return errorResult;//errorResult 的值也会在 xml 文件中配置
        }else{      arg0.invoke();      return null;      }
    }
}
```

step2：在 struts-main.xml 中添加拦截器配置，加一个新的<package>

```
<package name="netctoss-default" abstract="true" extends="json-default"><!--继承 json 包-->
```

```

-->
<!-- abstract="true" 表此包可以被继承 -->
<interceptors>
    <interceptor name="sessionValidate"
        class="com.tarena.netctoss.interceptor.SessionValidateInterceptor">
    </interceptor>
    <interceptor-stack name="netctossStack">
        <interceptor-ref name="defaultStack" />
        <interceptor-ref name="session Validate">
            <param name="key">com.tarena.netctoss.admin.key</param>
            <param name="errorResult">loginForm</param><!-- 下面有定义-->
        </interceptor-ref>
    </interceptor-stack><!-- 注意：个别多余的空行也会报错！ -->
</interceptors>
<default-interceptor-ref name="netctossStack" /><!--所有 Action 默认用该拦截器栈-->
<global-results><!-- 包里所有的 Action 都能用的 Result -->
    <result name="loginForm" type="redirectAction"><!-- 这里用重定向好些 -->
        <param name="namespace"/>/login</param><!-- 跨包了，所以这么写 -->
        <param name="actionName">loginform</param>
    </result>
</global-results><!-- 注意和 default-interceptor-ref 的顺序 -->
</package>

```

step3：修改 struts-cost.xml，使该包继承 netctoss-default 包

```
<package name="cost" namespace="/cost" extends="netctoss-default">
```

step4：修改 struts-main.xml，使 main 包继承 netctoss-default 包

```
<package name="main" namespace="/main" extends="netctoss-default">
```

step5：部署，测试，地址栏输入：<http://localhost:8080/NetCTOSS/cost/list.action>，则会跳转到登录页面；地址栏输入：<http://localhost:8080/NetCTOSS/main/main.action>，也会跳转到登录页面；只有登录后才可正常访问

一百二十一、Struts2 中如何处理异常

LICHOO

7.1 异常一般出现在何处

MVC 模式，一般 M 出现异常！

M 要做状态恢复、连接关闭、资源关闭、数据库回滚，然后抛出异常。

7.2 如何配置异常

根据抛出不同的异常，转到不同的异常页面。

step1：新建工程 struts06，导入 Struts2 核心包，配置前端控制器

step2：在 exception 包下，新建 DAOException，并继承 Exception

```
public class DAOException extends Exception { //无内容的 }
```

step3：在 dao 包下，新建 FooDAO，用于模拟 DAO 层

```
public class FooDAO {  
    public void foo() throws DAOException{ throw new DAOException(); }}
```

step4：在 action 包下，新建 TestAction，用于模拟控制层

```
public class TestAction {  
    public String execute() throws Exception{ System.out.println("TestAction.execute()...");  
        FooDAO fooDAO=new FooDAO(); //模拟自定义异常  
        //FooDAO fooDAO=null; //模拟运行时异常  
        fooDAO.foo(); return "success"; }}
```

step5：在 WEB-INF/jsp 下新建 dao_error.jsp 和 rt_error.jsp

1) dao_error.jsp

```
<h1 style="color:red;">DAO 异常</h1>
```

2) rt_error.jsp

```
<h1 style="color:red;">运行时异常</h1>
```

step6：新建 struts.xml 文件，进行异常的配置

```
<package name="test" namespace="/" extends="struts-default">  
    <action name="test" class="action.TestAction"><!--注意元素间的顺序，自定义异常-->  
        <exception-mapping result="daoError" exception="exception.DAOException" />  
        <exception-mapping result="rtError" exception="java.lang.RuntimeException" />  
        <result name="daoError">/WEB-INF/jsp/dao_error.jsp</result>  
        <result name="rtError">/WEB-INF/jsp/rt_error.jsp</result>  
    </action>  
</package>
```

step7：部署，测试，地址栏输入：<http://localhost:8080/struts06/test.action>，若模拟 DAO 异常，则页面显示 DAO 异常；若模拟运行时异常，则页面显示运行时异常

step8：也可定义全局异常，注意：在<action>前定义

```
<global-results><result name="rtError">/WEB-INF/jsp/rt_error.jsp</result></global-results>  
<global-exception-mappings>  
    <exception-mapping result="rtError" exception="java.lang.RuntimeException" />  
</global-exception-mappings>
```

- ◆ 注意事项：若全局和局部同名，听局部的。

LICHOOL

一百二十二、Struts2 中如何实现国际化

8.1 i18n

把页面中的文字全部写在属性文件“.properties”中（也可叫资源文件）。由于该文件不能写中文，所以对于中文，要写对应的 Unicode 编码 \uxxxx\uxxxx\uxxxx。

8.2 如何获得中文的 Unicode 编码

step1：Windows 系统：从开始菜单，进入命令控制行，或搜索栏直接输入 cmd

step2：输入 jdk 命令：源文件 目标文件

```
native2ascii 1.properties 2.properties
```

8.3 浏览器如何决定用哪个资源文件

资源文件的命名有规范！命名正确则自动切换，由浏览器发送的请求来确定是哪个国家哪个语言，放 Action 包中。

zh：中文的国际编码	CN：中国的国际编码
en：英文的国际编码	US：美国的国际编码

8.4 资源文件的命名

```
例如：TestAction_zh_CN.properties 和 TestAction_en_US.properties
```

但是这么写的话，只能是 TextAction 用，若 Action 有共同的部分，则把前面的 TestAction 替换为 package，例如：package_zh_CN.properties，此时该属性文件被所在包中的所有 Action 都可用。

8.5 资源文件的分类

- 1) 类级：TestAction_zh_CN.properties。
- 2) 包级：package_zh_CN.properties，类级和包级资源文件放 Action 包中。
- 3) 全局：放 src 下，message_zh_CN.properties，前面的名字（即 message）可随便起，但必须在 struts.xml 中配置：

```
<constant name="struts.custom.i18n.resources" value="message"></constant>
```

◆ 注意事项：

- ❖ 全局的配置，在<package>标签外配置。
- ❖ 资源信息先去类级里找，然后包级，最后全局。
- ❖ 若局部和全局的命名相同，则听局部的。

8.6 实现国际化的步骤

step1：新建工程 struts07，导入 Struts2 核心包，配置前端控制器，准备好资源文件

- 1) TestAction_zh_CN.properties，放 Action 中

```
jsp.welcome=\u6B22\u8FCE\u4F60
```

- 2) TestAction_en_US.properties，放 Action 中

```
jsp.welcome=Welcome!  
jsp.name=chang
```

- 3) package_en_US.properties，放 Action 中

```
jsp.name=hello
```

4) message_en_US.properties, 放 src 目录下

```
jsp.logo=Struts2
```

step2: 在 action 包中, 新建 TestAction, 且必须继承 ActionSupport 类

```
public class TestAction extends ActionSupport {  
    private String name;      .....get/set 方法  
    public String execute() throws Exception {  
        System.out.println("TestAction.execute(..)...");  
        name = getText("jsp.name");      return "success";    }  
}
```

- ◆ 注意事项: Action 中的错误提示信息, 也可以放入资源文件中, 用 key 表示。由于继承了 ActionSupport, 所以有了 getText("键的名字");方法, 可以把这个返回值赋给 Action 中的错误提示信息属性。

step3: 在 WEB-INF/jsp 文件夹中, 新建 main.jsp, 并在页面中使用标签: <s:text name="文件中共同的地方, 键" />

```
<%@taglib uri="/struts-tags" prefix="s"%>  
<s:text name="jsp.welcome"/>  
<s:text name="jsp.name"/>  
<s:text name="jsp.logo"/>  
<s:property value="name"/>
```

step4: 在 struts.xml 中配置全局资源文件

```
<struts>  
    <constant name="struts.custom.i18n.resources" value="message" />  
    <package name="test" namespace="/" extends="struts-default">  
        <action name="i18n" class="action.TestAction">  
            <result name="success">/WEB-INF/jsp/main.jsp</result>  
        </action>  
    </package>  
</struts>
```

step5: 部署, 测试, 地址栏输入: http://localhost:8080/struts07/i18n.action, 由于当前为中文操作系统, 所以显示结果为: 欢迎你 jsp.name jsp.logo jsp.name

step6: 如何模拟英文操作系统? 可以采取修改浏览器首选语言的方式, 修改后重新发请求, 则显示结果为: Welcome! chang Struts2 chang



- ◆ 注意事项: 服务器安装的操作系统是哪个国家的语言, 则是默认的语言资源文件。如: 服务器上安的是中文的系统, 则中文资源文件是默认的, 那么第三方其他国家的用户看得则是中文。

一百二十三、NetCTOSS 项目

LICHOO

9.1 DAO 优化、重构、封装！【重要】

step1：在 com.tarena.netctoss.entity 包中，新建 Entity 抽象类，且内容为空，目的是为了封装后，返回的对象是父类型

```
public abstract class Entity { //内容为空 }
```

step2：让每个实体类继承 Entity 类

```
public class Cost extends Entity{ ... }
```

```
public class Admin extends Entity{ ... }
```

step3：在 com.tarena.netctoss.dao.impl 包中，新建 BaseDAO 抽象类

```
public abstract class BaseDAO { //该封装可以和之前写的方法进行比较~  
/** 去除 List 的泛型，加 final 修饰，为的是防止不小心重写，其中的一个参数为对象数组，对象数组可以放任何类型的参数。该方法的封装只可进行查询操作 */  
protected final List query(String sql, Object[] params) throws DAOException {  
    Connection conn=null;  
    try { conn = DBUtils.openConnection(); //根据指定的 SQL 进行查询  
        PreparedStatement stmt=conn.prepareStatement(sql);  
        if(params!=null){  
            for(int i=0;i<params.length;i++){  
                stmt.setObject(i+1, params[i]); //第一个问号?是从 1 开始的  
                /** 如果需要任意参数类型转换，使用 setObject 方法，该方法将  
                    给定的参数转换为相应的 SQL 类型。 */  
            }  
        }  
        ResultSet rs=stmt.executeQuery(); //执行查询操作  
        List entityList=new ArrayList(); //返回 entityList 集合  
        while(rs.next()){//根据子类分别重写的 toEntity 方法，进行转换  
            entityList.add(toEntity(rs));  
        }  
        return entityList;  
    } catch (SQLException e) {  
        e.printStackTrace();  
        throw new DAOException("访问异常", e);  
    } finally{  
        DBUtils.closeConnection(conn);  
    }  
    /** 定义抽象方法 toEntity()，由子类去实现：将结果集中的每一条数据转换成对应的每一个实体对象 */  
    public abstract Entity toEntity(ResultSet rs) throws SQLException;  
    /** 加 final 修饰，为的是防止不小心重写。该方法的封装可进行增、删、改操作 */  
    protected final int update(String sql, Object[] params) throws DAOException {  
        Connection conn=null;  
        try { conn = DBUtils.openConnection(); //根据指定的 SQL 进行查询  
            PreparedStatement stmt=conn.prepareStatement(sql);  
            if(params!=null){  
                for(int i=0;i<params.length;i++){  
                    stmt.setObject(i+1, params[i]); //第一个问号?是从 1 开始的
```

```

        }
    }

    return stmt.executeUpdate(); //执行更新操作，会返回影响的记录数
} catch (SQLException e) {
    e.printStackTrace();
    throw new DAOException("访问异常", e);
} finally {
    DBUtils.closeConnection(conn);
}
}
}

```

step4：让每个实现类都继承 BaseDAO

```

public class CostDAOImpl extends BaseDAO implements CostDAO { ... }

public class AdminDAOImpl extends BaseDAO implements AdminDAO { ... }

```

step5：重构 CostDAOImpl 中的方法，并实现 toEntity()方法

```

public List<Cost> findAll() throws DAOException {//重构后
    return query(findAll, null);
}

public List<Cost> findAll(int page, int rowsPerPage) throws DAOException {//重构后
    int start=(page-1)*rowsPerPage+1;           int end=start+rowsPerPage;
    return query(findAllByPage,new Object[]{end,start});
}

public int getTotalPages(int rowsPerPage) throws DAOException {//方法不变 ... }

public void delete(int id) throws DAOException {//重构后
    update(delete, new Object[]{id});
}

public Cost findByName(String name) throws DAOException {//重构后
    List<Cost> costList=query(findByName, new Object[]{name});
    if(costList!=null&&costList.size()>0){      return costList.get(0);
    } else {   return null;   }
}

.....以后其他的方法能用当前的封装就用，不能用则单独写.....
/** 在 Cost 实体类中实现 toEntity 方法，之前的 toCost(ResultSet rs)方法删除掉 */
public Entity toEntity(ResultSet rs) throws SQLException {
    Cost cost = new Cost();      cost.setId(rs.getInt("ID"));
    cost.setName(rs.getString("NAME"));
    cost.setBaseDuration(rs.getInt("BASE_DURATION"));
    cost.setBaseCost(rs.getFloat("BASE_COST"));
    cost.setUnitCost(rs.getFloat("UNIT_COST"));
    cost.setStartTime(rs.getDate("STARTIME"));
    cost.setStatus(rs.getString("STATUS"));
    cost.setCreateTime(rs.getDate("CREATIME"));
    cost.setDescr(rs.getString("DESCR"));
    cost.setCostType(rs.getString("COST_TYPE"));
    return cost;//此处 Cost 类要继承 Entity 类，才能返回 Cost 类型
}

```

step6：重构 AdminDAOImpl 中的方法，并实现 toEntity()方法

```

public Admin findByCodeAndPwd(String code, String pwd) throws DAOException {//重构
后
    List<Admin> adminList = query(findByCodeAndPwd, new Object[]{code,pwd});
    if (adminList != null && adminList.size() > 0) {      return adminList.get(0);
    } else {   return null;   }
}

.....以后其他的方法能用当前的封装就用，不能用则单独写.....
/** 在 Admin 实体类中实现 toEntity 方法，之前的 toAdmin(ResultSet rs)方法删除掉 */

```

```

public Entity toEntity(ResultSet rs) throws SQLException {
    Admin admin = new Admin();           admin.setId(rs.getInt("ID"));
    admin.setAdminCode(rs.getString("ADMIN_CODE"));
    admin.setPassword(rs.getString("PASSWORD"));
    admin.setName(rs.getString("NAME"));
    admin.setTelephone(rs.getString("TELEPHONE"));
    admin.setEnrollDate(rs.getDate("ENROLLDATE"));
    admin.setEmail(rs.getString("EMAIL"));
    return admin;//此处 Admin 类要继承 Entity 类，才能返回 Admin 类型
}

```

step7：部署，测试，一切正常

9.2 资费更新

step1：在 DAO 优化、重构、封装的基础上进行

step2：在 CostDAO 接口中添加方法定义

```

public Cost findById(Integer id) throws DAOException;
public void update(Cost cost) throws DAOException;

```

step3：在 CostDAOImpl 中实现方法

```

private static String findById= "select ID, NAME, BASE_DURATION, BASE_COST,
UNIT_COST, CREATIME, STARTIME, STATUS, DESC,COST_TYPE from COST_CHANG
where ID=?";
private static String update="update COST_CHANG set NAME=?,BASE_DURATION=?,
BASE_COST=?,UNIT_COST=?,DESCR=?,COST_TYPE=? where ID=?";
public Cost findById(Integer id) throws DAOException {
    List<Cost> costList=query(findById, new Object[]{id});
    if(costList!=null&&costList.size()>0){      return costList.get(0);
    } else {      return null;    }
}
public void update(Cost cost) throws DAOException {//实现接口的方法
    Object[] params={cost.getName(),cost.getBaseDuration(),cost.getBaseCost(),
        cost.getUnitCost(),cost.getDescr(),cost.getCostType(),cost.getId()};
    update(update,params);//封装类中的方法
}

```

step4：在 BaseAction 中，实现 RequestAware 接口，并添加 set 方法

```

public class BaseAction implements SessionAware,Constants,RequestAware {
    protected Map<String, Object> session;
    protected Map<String, Object> request;
    public void setSession(Map<String, Object> session) { this.session = session; }
    public void setRequest(Map<String, Object> request) { this.request = request; }
    public String execute() throws Exception { return "success"; }
}

```

step5：在 com.tarena.netctoss.action.cost 包中，新建 UpdateCostAction 和 UpdateCostFormAction

1) UpdateCostAction，并继承 BaseAction

```

private Cost cost; private int page;//页面的 form 把 page 传入 .....各自 get/set 方法
private CostDAO costDAO=DAOFactory.getCostDAO();
public String execute()throws DAOException{
    try{      costDAO.update(cost);      request.put("ok",0);
    }catch (Exception e) {      e.printStackTrace();      request.put("ok",1);
    }
}

```

```
        return "success";  
    }  
}
```

2) UpdateCostFormAction

```
private Cost cost;      private int id;      private int page;//用于保存请求时的页数  
private CostDAO costDAO=DAOFactory.getCostDAO();//不需要 get/set 方法  
.....各自 get/set 方法  
public String execute() throws DAOException{  
    cost=costDAO.findById(id);      return "success";  
}
```

- ◆ 注意事项：上面两个 Action 可以合成一个，使用不同的方法名区分，然后在配置文件指定各自的方法名即可。

step6：在 struts-cost.xml 中添加配置

```
<action name="updateForm"  
       class="com.tarena.netctoss.action.cost.UpdateCostFormAction">  
    <result name="success">/WEB-INF/jsp/cost/cost_modi.jsp</result>  
</action>  
<action name="update" class="com.tarena.netctoss.action.cost.UpdateCostAction">  
    <result name="success">/WEB-INF/jsp/cost/cost_modi.jsp</result>  
</action>
```

step7：修改 cost_list.jsp，增加修改连接

```
<input type="button" value="修改" class="btn_modify"  
       onclick="location='updateForm.action?id=${id}&page=${page}'"/>
```

step8：复制资费静态页面到工程中，并修改为 cost_modi.jsp，记得导入各种资源

```
<%@taglib uri="/struts-tags" prefix="s"%>  
<script language="javascript" src="../js/jquery-1.4.1.min.js"></script>  
<script language="javascript" type="text/javascript">  
    $(function(){ $('#save').click(function(){ $('#costForm').submit(); }); })  
    //body 定义 onload 事件，页面刷新时执行，根据 request 中 ok 的值显示不同的提示。  
    function showMsg(){  
        feeTypeChange(${cost.costType});//读取页面判断资费类型从而显示不同效果  
        var flag="${ok}";//若 ok 值没有，则 var flag="";如果不写引号，则 var flag= ;,  
        有些浏览器不认识  
        var msg_div=document.getElementById("save_result_info")  
        if(flag=="0"){ msg_div.innerHTML="修改成功！"; showResult();}  
        else if(flag=="1"){//直接写 else 则初始状态为空则也显示失败！  
            msg_div.innerHTML="修改失败！"; showResult(); } }  
        //注意：把原来的自费类型切换函数改一改：.className="width148"，.readOnly 为  
        大写的字母 “o”，否则无效  
    </script>  
    <body onload="showMsg()"><!--页面一刷新则调用--><!--请求中加 page 是为了返回用  
-->  
    <form id="costForm" action="update.action?page=${page}" method="post" .....  
        <input name="cost.id" type="text" class="readonly" readonly  
              value=<s:property value='cost.id' /> .....  
        <input name="cost.name" id="costName" type="text"  
              value=<s:property value='cost.name' /> .....
```

```

<s:radio list='{"1":"包月","2":"套餐","3":"计时"}' name="cost.costType"
          onclick="feeTypeChange(this.value);"/>
.....
<input name="cost.baseDuration" type="text"
       value=">" /> .....
<input name="cost.baseCost" type="text"
       value=">" /> .....
<input name="cost.unitCost" type="text"
       value=">" /> .....
<s:textarea name="cost.descr" cssClass="width300 height70"/>
<input type="button" value="取消" class="btn_save"
       onclick="location.href='list.action?page=${page}'"; /> .....

```

- ◆ 注意事项：使用 HTML 中的<textarea>标签，会在描述信息的前、后，莫名其妙的多出空格。

step9：部署，测试，一切正常

9.3 导航条

step1：为了方便使用，把页面中共同的部分：导航条提取出来。这样处理后，我们只要修改一次连接地址，则所有页面都可生效

step2：在 WEB-INF/jsp 文件夹中，新建 head.jsp，该 jsp 页面只有如下内容，没有其他 HTML 标签

```

<%@page pageEncoding="utf-8"%>
<script language="javascript" type="text/javascript">
$(function() { <% String uri=request.getRequestURI();
System.out.println("请求的 uri 为: "+uri); %>
var uri="<%request.getRequestURI() %>";//获得 uri
$('#menu a').each(function(){//相当于 foreach 得到 a 链接的整个元素，去挨个比较
if(uri.indexOf($(this).attr("id"))>0){//找到后，更改样式
    $(this).removeClass($(this).attr("id")+"off");
    $(this).addClass($(this).attr("id")+"on");
} else if(uri.indexOf("cost_")>0){//样式中没有 cost_ 所以单独处理。或改页面名字
    $("#cost_").removeClass("fee_off");
    $("#cost_").addClass("fee_on");
} else if(uri.indexOf("index")>0){//主页也特殊，要单独处理。或者改主页的名字
    $("#index").removeClass("index_off");
    $("#index").addClass("index_on");
}
});
</script>
<ul id="menu">
<li><a id="index" href="/NetCTOSS/main/main.action" class="index_off"></a></li>
<li><a id="role_" href="" class="role_off"></a></li>
<li><a id="admin_" href="" class="admin_off"></a></li>
<li><a id="cost_" href="/NetCTOSS/cost/list.action" class="fee_off"></a></li>

```

```
<li><a id="account_" href="" class="account_off"></a></li>
<li><a id="service_" href="" class="service_off"></a></li>
<li><a id="bill_" href="" class="bill_off"></a></li>
<li><a id="report_" href="" class="report_off"></a></li>
<li><a id="information_" href="" class="information_off"></a></li>
<li><a id="password_" href="" class="password_off"></a></li>
</ul>
```

step3：将所有页面中...部分删除，<div id="navi"></div>要保留

step4：在<div id="navi"></div>中添加 Struts2 的 include 标签

```
<div id="navi"><s:include value="../head.jsp" /></div>
```

step5：主页 index.jsp 特殊一些

```
<div id="index_navi"><s:include value="../jsp/head.jsp" /></div>
```

- ◆ 注意事项：今后所有页面都引入该 head.jsp，而连接只要修改一次即可所有页面生效。

step6：所有页面添加 jQuery 框架

```
<script language="javascript" src="../js/jquery-1.4.1.min.js"></script>
```

- ◆ 注意事项：所有页面记得导入 Struts2 标签：`<%@taglib uri="/struts-tags" prefix="s"%>`

step7：部署，测试，连接正常！样式修改也正常！

一百二十四、项目经验

LICHOO

10.1 主键用 int 还是 Integer

主键用 int、Integer 都可以，但在 Hibernate 框架中，习惯用 Integer。而非主键最好用封装类 Integer，这样可以允许有空值，否则至少有个 0。

10.2 “..” 表示的意思

表示向上跳一级目录，看的是浏览器地址！如：localhost:8080/应用名/命名空间/xx.action，用“..”则跳到应用名，而“命名空间/xx.action”整体为一级，共同决定响应后显示的页面。

10.3 导入静态页面，样式、JS 失效问题

项目结构和发布后的结构不一样的，发布后结构为 webapps/应用名/然后是 WebRoot 中那一堆东西，即 WEB-INF 文件夹啊、images 文件夹啊、js 文件夹啊等。

所以，样式和 js 的导入路径为：

```
<link type="text/css" rel="stylesheet" media="all" href="../styles/global_color.css" />
<script language="javascript" src="../js/jquery-1.4.1.min.js"></script>
```

即从当前请求地址跳到应用名，然后找到 js 文件夹，然后找到 jquery。

10.4 <s:hidden> 和 <s:textarea> 标签

<s:hidden value=""> 中 value 属性不是 OGNL 表达式，使用 value="%{ognl}" 可使里面的字符串强制转成 ognl 表达式。

<s:textarea value=""> 中的 value 属性也不是 OGNL 表达式，也使用 value="%{ognl}" 强转。

10.5 四种情形下的绝对路径写法

绝对路径：前面一定有个 “/”，且从应用名开始写。以下为 4 种情形的绝对路径写法：

1) 链接：/应用名/... 2) 提交：/应用名/.. 3) 重定向：/应用名/..

4) 转发：/应用名后开始写

◆ 注意事项：例如在 cost_list.jsp 页面，添加 add 页面的连接时，可以只写 add.action（相对路径），因为它们是同包的，或者写绝对路径，但不能写成 cost/add.action，会出现叠加问题，可参考 Spring 笔记 12.2 节。

10.6 URL 和 URI

http://ip:port/NetCTOSS/.....

```
|-->    uri    -->|
|-->      url      -->|
```

10.7 util.Date 和 sql.Date

当要把日期插入数据库时，要注意程序导入的是 util.Date 还是 sql.Date。

如：new Date(System.currentTimeMillis());

它们之间的转换：

```
Date date=new Date();//util.Date
java.sql.Date date1=new java.sql.Date(date.getTime());
```

14 Hibernate 系列笔记

一百二十五、Hibernate 的概述

1.1 Hibernate 框架的作用

Hibernate 框架是一个数据访问框架（也叫持久层框架，可将实体对象变成持久对象，详见第 5 章）。通过 Hibernate 框架可以对数据库进行增删改查操作，为业务层构建一个持久层。可以使用它替代以前的 JDBC 访问数据。

1.2 Hibernate 访问数据库的优点

- 1) 简单，可以简化数据库操作代码。
- 2) Hibernate 可以自动生成 SQL，可以将 ResultSet 中的记录和实体类自动的映射（转化）。
- 3) Hibernate 不和数据库关联，是一种通用的数据库框架（支持 30 多种数据库），可以方便数据库移植。任何数据库都可以执行它的 API。因为 Hibernate 的 API 中是不涉及 SQL 语句的，它会根据 Hibernate 的配置文件，自动生成相应数据库的 SQL 语句。

1.3 JDBC 访问数据库的缺点

- 1) 需要编写大量的复杂的 SQL 语句、表字段多时 SQL 也繁琐、设置各个问号值。
- 2) 需要编写实体对象和记录之间的代码，较为繁琐。
- 3) 数据库移植时需要修改大量的 SQL 语句。

1.4 Hibernate 的设计思想

Hibernate 是基于 ORM（Object Relation Mapping）思想设计的，称为对象关系映射。负责 Java 对象和数据库表数据之间的映射。

Hibernate 是一款主流的 ORM 工具，还有其他很多 ORM 工具，如：MyBatis（以前叫 iBatis）、JPA。Hibernate 功能比 MyBatis 强大些，属于全自动类型，MyBatis 属于半自动。但全自动会有些不可控因素，因此有些公司会用 MyBatis。

ORM 工具在完成 Java 对象和数据库之间的映射后：

- 1) 在查询时，直接利用工具取出“对象”（不论是查询一条记录还是多条记录，取出的都是一个个对象，我们不用再去转化实体了）。
- 2) 在增删改操作时，直接利用工具将“对象”更新到数据库表中（我们不用再去把对象转成数据了）。
- 3) 中间的 SQL+JDBC 细节，都被封装在了工具底层，不需要程序员参与。
 - ◆ 注意事项：
 - ❖ Java 程序想访问数据库，只能通过 JDBC 的方式，而 Hibernate 框架也就是基于 ORM 思想对 JDBC 的封装。
 - ❖ Hibernate 是以“对象”为单位进行数据库的操作。

一百二十六、Hibernate 的基本使用

LICHOO

2.1 Hibernate 的主要结构

- 1) hibernate.cfg.xml (仅 1 个): Hibernate 的主配置文件, 主要定义数据连接参数和框架设置参数。
 - ◆ 注意事项: 就是个 xml 文件, 只是名字比较奇葩!
- 2) Entity 实体类 (n 个, 一个表一个): 主要用于封装数据库数据。
- 3) hbm.xml 映射文件 (n 个): 主要描述实体类和数据表之间的映射信息。描述表与类, 字段与属性的对应关系。
 - ◆ 注意事项: hbm.xml 是个后缀, 如: 命名可写 Cost.hbm.xml。

2.2 Hibernate 主要的 API

- 1) Configuration: 用于加载 hibernate.cfg.xml 配置信息。用于创建 SessionFactory。
 - 2) SessionFactory: 存储了 hbm.xml 中描述的信息, 内置了一些预编译的 SQL, 可以创建 Session 对象。
 - 3) Session: 负责对数据表执行增删改查操作。表示 Java 程序与数据库的一次连接会话, 是对以前的 Connection 对象的封装。和 JSP 中的 session 不是一回事, 就是名字一样而已。
 - 4) Query: 负责对数据表执行特殊查询操作。
 - 5) Transaction: 负责 Hibernate 操作的事务管理。默认情况下 Hibernate 事务关闭了自动提交功能, 需要显式的追加事务管理 (如调用 Transaction 对象中的 commit(); 提交事务)!
- ◆ 注意事项:
- ❖ 这些 API 都是在 Hibernate 包下的, 导包别导错!
 - ❖ 第一次访问数据库比较慢, 比较耗资源, 因为加载的信息多。

2.3 Hibernate 使用步骤

step1: 建立数据库表。

step2: 建立 Java 工程 (Web 工程也可), 引入 Hibernate 开发包和数据库驱动包。必须引入的包: hibernate3.jar、cglib.jar、dom4j.jar、commons-collections.jar、commons-logging.jar……等

step3: 添加 hibernate.cfg.xml 配置文件, 文件内容如下:

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="dialect"><!-- 指定方言, 决定 Hibernate 生成哪种 SQL -->
            org.hibernate.dialect.OracleDialect<!-- 不知道数据库版本就写 OracleDialect -->
        </property><!-- 可在 hibernate3.jar 中 org.hibernate.dialect 包下查看名字 -->
        <property name="connection.url">
            jdbc:oracle:thin:@localhost:1521:dbchang
        </property>
        <property name="connection.username">system</property>
```

```

<property name="connection.password">chang</property>
<property name="connection.driver_class">
    oracle.jdbc.driver.OracleDriver
</property>
<!-- 框架参数，将 hibernate 底层执行的 SQL 语句从控制台显示 -->
<property name="show_sql">true</property>
<!-- 格式化显示的 SQL -->
<property name="format_sql">true</property>
<!-- 指定映射描述文件 -->
<mapping resource="org/tarena/entity/Cost.hbm.xml" />
</session-factory>
</hibernate-configuration>

```

属性名	含义
dialect	指定数据库的方言。虽然各种数据库都符合 SQL 标准，但不同的数据库所实现的 SQL 语句略有不同，此所谓数据库的方言。应根据所采用数据库的不同设置不同的方言类。例如： org.hibernate.dialect.Oracle9Dialect(针对 Oracle) org.hibernate.dialect.MySQLDialect(针对 MySQL) org.hibernate.dialect.SQLServerDialect (针对 SQLServer)
connection.driver_class	指定数据库的 JDBC 驱动类
connection.url	指定 JDBC 连接数据库的 URL
connection.username	指定连接数据库的用户名
connection.password	指定连接数据库的密码
Show_sql	可选参数，如果设置为 "true"，则 hibernate 将以日志的形式输出所执行的 SQL 语句，用于跟踪和调试基于 hibernate 的应用，其默认值为 "false"
Format_sql	可选参数，如果设置为 "true"，则 hibernate 输出 SQL 语句时整理格式，按照合理分行和缩进的形式显示 SQL 语句，如果为 false，则不整理格式，其默认值为 "false"

- ◆ 注意事项：应该放在源文件的 src 目录下，默认为 hibernate.cfg.xml。文件内容是 Hibernate 工作时必须用到的基础信息。

step4：编写 Entity 实体类（也叫 POJO 类），例如：资费实体类 Cost

private Integer id; //资费 ID	private String feeName; //资费名称
private Integer baseDuration; //基本时长	private Float baseCost; //基本定费
private Float unitCost; //单位费用	private String status; //0: 开通; 1: 暂停;
private String descr; //资费信息说明	private Date createTime; //创建日期
private Date startTime; //启用日期	private String costType; //资费类型
.....getter/setter 方法	

- ◆ 注意事项：POJO 类表示普通类（Plain Ordinary Old Object），没有格式的类，只有属性和对应的 getter/setter 方法，而没有任何业务逻辑方法的类。这种类最多再加

入 equals()、 hashCode()、 toString() 等重写父类 Object 的方法。不承担任何实现业务逻辑的责任。

step5：编写 hbm.xml 映射（文件）描述信息：映射文件用于指明 POJO 类和表之间的映射关系（xx 属性对应 xx 字段），一个类对应一个映射文件。例如：Cost.hbm.xml 内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<!-- 定义 COST_CHANG 表和 Cost 类型之间的映射信息 -->
<hibernate-mapping><!-- <hibernate-mapping package="包名写这也行" -->
    <!-- name: 包名.类名，指定是哪个类； table: 数据库中哪个表； catalog: 对 Oracle 而言为哪个数据库，对 MySQL 而言为某个用户（MySQL 是在用户下建表， Oracle 是在库中建表）,
不写也行（若用工具则会自动生成）。例如， select * from cost_chang 则会在 hibernate.cfg 配置文件中定义的库（或用户）下去找表。若写了则为 select * from system.cost_chang
-->
    <class name="org.tarena.entity.Cost" table="COST_CHANG" catalog="system">
        <!-- <id></id> 表明此为主键列，且必须写否则 xml 报错，主键映射 -->
        <id name="id" type="java.lang.Integer">
            <column name="ID" /><!-- 或双标签<column name="ID"></column> -->
            <!-- 指定主键值生成方式，采用序列方式生成主键，仅对添加操作有效-->
            <generator class="sequence">
                <param name="sequence">COST_SEQ_CHANG</param> <!-- 指定序列名-->
            </generator>
        </id>
        <property name="name" type="java.lang.String"><!-- 以下为非主键映射 -->
            <column name="NAME" /><!-- 可有 length、 not-null 属性，如： length="20" -->
        </property>
        <property name="baseDuration" type="java.lang.Integer"><!-- 映射顺序没关系 -->
            <column name="BASE_DURATION" />
        </property>
        <property name="baseCost" type="java.lang.Float"><!-- 类型要和实体定义的相同
-->
            <column name="BASE_COST" />
        </property>
        <property name="startTime" type="java.sql.Date"><!-- 列名写错则报错读不到实体-->
            <column name="STARTTIME" /><!-- junit 测试右键点 Copy Trace 查看错误列-->
        </property>
        <!-- 也可写成<property name="" type="" column="" /></property>，主键列同理！-->
        ..... 其他省略 .....
    </class>
</hibernate-mapping>
```

◆ 注意事项：

- ❖ 映射文件默认与 POJO 类放在一起；命名规则为：类名.hbm.xml。
- ❖ hbm.xml 中已写出的属性与字段的映射要一一对应，若表中没有某个字段，却

写了映射关系，则报错：找不到实体类。

step6：利用 Hibernate API 实现 DAO

1) 新建 HibernateUtil 类，用于封装创建 Session 的方法。如下：每个用户会对应一个 Session，但是 SessionFactory 是共享的。

```
public class HibernateUtil {
    private static SessionFactory sf;
    static{//不用每次都加载配置信息，所以放 static 块中，否则每次都加载会耗费资源
        Configuration conf=new Configuration();//加载主配置 hibernate.cfg.xml
        conf.configure("/hibernate.cfg.xml");
        sf=conf.buildSessionFactory();//获取 SessionFactory
    }
    public static Session getSession(){//获取 Session
        Session session =sf.openSession();      return session;
    }
}
```

2) 新建 CostDAO 接口

```
public Cost findById(int id);      public void save(Cost cost);
public void delete(int id);       public void update(Cost cost);
public List<Cost> findAll();
```

3) 新建 CostDAOImpl 类，用于实现 CostDAO 接口

```
public class CostDAOImpl implements CostDAO {
    private Session session;
    public CostDAOImpl () {//不想老写获得 session 的方法，就写在构造器中
        session=HibernateUtil.getSession();
    }
    /** get 方法执行查询，按主键当条件查询，如何判断是主键，是根据写的描述文件来定。
     * get 方法就是 findById，就是按主键去查，需指定：操作哪个类和 id（主键）条件值即可，其他条件查询做不了 */
    public Cost findById(int id) {
        //Session session=HibernateUtil.getSession();
        Cost cost=(Cost)session.get(Cost.class,id); session.close(); return cost;
    }
    /** save 方法执行增加操作，注意 1：获取事务并开启，增删改要注意，查询可以不管事务，因为没对数据库进行修改；注意 2：主键值根据 hbm.xml 中的<generator>定义生成，执行后，会先获取序列值，再去做 insert 操作。
     * 即先： select COST_SEQ_CHANG.nextval from dual; 然后： insert into ..... */
    public void save(Cost cost) {
        //Session session=HibernateUtil.getSession();
        Transaction tx=session.beginTransaction(); //打开事务      session.save(cost);
        tx.commit(); //提交事务      session.close(); //释放
    }
    /** delete 方法执行删除操作，由于 Hibernate 以“对象”为单位进行数据库操作，所以这里要传进去一个对象，虽然是个对象，但还是按主键做条件删除，只要把主键值设置上就行，其他非主键值不用管。也可先通过 id 查再删 */
    public void delete(int id) {
        //Session session=HibernateUtil.getSession();
        Transaction tx=session.beginTransaction();      Cost cost=new Cost();
        cost.setId(id); session.delete(cost); tx.commit(); session.close();
    }
    /** update 方法执行修改操作， */
    public void update(Cost cost) {
```

```

    //Session session=HibernateUtil.getSession();
    Transaction tx=session.beginTransaction();
    session.update(cost);//将 cost 对象更新到数据库      tx.commit();
    session.close();                                }

/** 特殊查询， SQL 语句： String sql="select * from COST_CHANG";
HQL 语句： String hql="from Cost"; （Hibernate Query Language）是面向对象的查询语句。
from 后写映射的类名，它是 Hibernate 中特有的查询语句，根据映射的类去查询。 */
public List<Cost> findAll() {
    //Session session=HibernateUtil.getSession();
    String hql="from Cost";//HQL 语句
    Query query=session.createQuery(hql);
    List<Cost> list=query.list();//执行查询，返回 List 集合
    session.close();      return list;          }  }

```

4) 新建 TestCostDAO 类，使用 junit 测试

```

@Test
public void testFindById(){//当 get 方法没有记录时，返回 null
    CostDAO costDao = new CostDAOImpl();    Cost cost = costDao.findById(1);
    System.out.println(cost.getName());System.out.println(cost.getBaseDuration());
    System.out.println(cost.getBaseCost());System.out.println(cost.getUnitCost());
    System.out.println(cost.getDescr());          }

@Test
public void testSave(){//id 主键列由 Hibernate 管理，这里不用设置
    Cost cost=new Cost();  cost.setName("2013 计时");
    cost.setUnitCost(0.8f);  cost.setDescr("2013-08-09 计时， 0.8 元/小时。");
    cost.setStatus("0");      cost.setCreateTime(new Date(System.currentTimeMillis()));
    CostDAO costDao = new CostDAOImpl();      costDao.save(cost);          }

@Test
public void testUpdate(){//开通某个资费，把状态由 0 变为 1
    CostDAO costDAO=new CostDAOImpl();
    /** 注意事项： 更新部分字段，不能和实现类中的删除那样，做一个对象出来！否则没设置的字段将被改为空！ 即不能： Cost cost=new Cost();  cost.setId(90);
    cost.setStatus("1");  cost.setStartTime(new Date(System.currentTimeMillis()));  */
    Cost cost=costDAO.findById(90);//只能先通过 id 找到带有所有值的对象
    cost.setStatus("1");//然后再对部分字段进行更新，才能避免把其他字段更新为空
    cost.setStartTime(new Date(System.currentTimeMillis()));
    costDAO.update(cost);          }

@Test
public void testDelete(){
    CostDAO costDAO=new CostDAOImpl();    costDAO.delete(90);          }

@Test
public void testfindAll(){
    CostDAO costDAO=new CostDAOImpl();  List<Cost> list=costDAO.findAll();
}

```

```
for(Cost c:list){    System.out.println(c.getName());    }
```

2.4 HQL 语句（简要介绍）

简要介绍见 2.3 节中 step6 中的 3) 特殊查询（本页最上）。详细介绍见第七章。

一百二十七、数据映射类型

hbm.xml 在描述字段和属性映射时，采用 type 属性来指定映射类型。

3.1 映射类型的作用

主要负责实现属性和字段值之间的相互转化。

3.2 type 映射类型的两种写法

1) 指定 Java 类型，例如：java.lang.String、java.lang.Integer …，不能写成 String …

2) 指定 Hibernate 类型，例如：

①整数：byte、short、integer、long；

②浮点数：float、double； ③字符串：string；

④日期和时间：date（只处理年月日），time（只处理时分秒），timestamp（处理年月日时分秒）；

⑤布尔值：true/false<-yes_no->char(1)（数据库存 Y/N，显示时自动转为 true/false）、true/false<-true_false->char(1)（数据库存 T/F，显示时自动转为 true/false）、true/false<-boolean->bit（数据库存 1/0，显示时自动转为 true/false）；

⑥其他：blob（以字节为单位存储大数据）、clob（以字符为单位存储大数据）、big_decimal、big_integer；

◆ 注意事项：Hibernate 类型都是小写！建议使用 Hibernate 类型。

3) 所以 2.3 节中 step5 的 Cost.hbm.xml 内的 type 也可这样写：

```
<property name="name" type="string"><column name="NAME" /></property>
<property name="baseCost" type="float"><column name="BASE_COST" /></property>
<property name="startTime" type="date"><column name="STARTTIME" /></property>
```

◆ 注意事项：

❖ java.util.Date 有年月日时分秒毫秒，但如果用 date 映射，则只把年月日存进数据库；java.sql.Date 只有年月日。java.sql.Timestamp 有年月日时分秒毫秒。

❖ 若在页面显示按特定格式显示则用 Struts2 标签：

<s:date name="属性名" format="yyyy-MM-dd HH:mm:ss"/>

Hibernate 映射类型	对应的 Java 数据类型	对应的标准 SQL 字段类型
integer	int 或 java.lang.Integer	INTEGER
long	long 或 java.lang.Long	BIGINT
short	short 或 java.lang.Short	SMALLINT
float	float 或 java.lang.Float	FLOAT
double	double 或 java.lang.Double	DOUBLE
big_decimal	java.math.BigDecimal	NUMERIC
character	java.lang.String	CHAR(1)
string	java.lang.String	VARCHAR
byte	byte 或 java.lang.Byte	TINYINT
boolean	boolean 或 java.lang.Boolean	BIT
yes_no	boolean 或 java.lang.Boolean	CHAR(1) ('Y' or 'N')
true_false	boolean 或 java.lang.Boolean	CHAR(1) ('T' or 'F')
date	java.util.Date 或 java.sql.Date	DATE

一百二十八、Hibernate 主键生成方式

LICHOO

Hibernate 负责管理主键值。它提供了多种主键生成方式。

4.1 五种生成方式

1) sequence: 可以按指定序列生成主键值。只适用于 Oracle 数据库。不担心并发量！

例如: <generator class="sequence">

```
<param name="sequence">序列名字</param></generator>
```

◆ 注意事项: 创建序列时如果不指定参数, 默认从 1 开始, 步进是 1。

2) identity: 按数据库自动增长机制生成主键值。一般适用于 MySql、SQLServer 数据库。

例如: <generator class="identity"></generator>

3) native: Hibernate 会根据方言类型不同, 选择不同的主键生成方式。如果是 OracleDialect 则会选择 sequence, 如果是 MySQLDialect 则会选择 identity。

例如: <generator class="native"></generator>

◆ 注意事项: 如果是 MySql 数据库, <param name="sequence">序列名字</param>是不起作用的, 但也不会出错; 如果是 Oracle 数据库, <param name="sequence">序列名字</param>就会起作用, 所以一般我们会加上这句话, 这样通用性更强。

4) assigned: Hibernate 会放弃主键生成, 采用此方法, 需要在程序中指定主键值。

例如: <generator class="assigned"></generator>

5) increment: Hibernate 先执行 select max(id)...语句获取当前主键的最大值, 执行加 1 操作, 然后再调用 insert 语句插入。Oracle 和 MySQL 都可用。但不适合并发量很大的情况!

例如: <generator class="increment"></generator>

6) uuid/hilo: uuid: 按 UUID 算法生成一个主键值(字符串类型); hilo: 按高低位算法生成一个主键值(数值类型)。

例如: <generator class="hilo"></generator>

◆ 注意事项:

- ❖ 主键一般都是自动生成的。我们一般不使用业务数据作为主键, 因为业务逻辑的改变有可能会改变主键值。
- ❖ 主键生成方式是枚举类型, 只能从一个有限的范围内选择, 不能自定义。其中, sequence 是使用序列生成主键(Oracle 数据库经常使用)。

一百二十九、Hibernate 基本特性

LICHOO

5.1 对象持久性

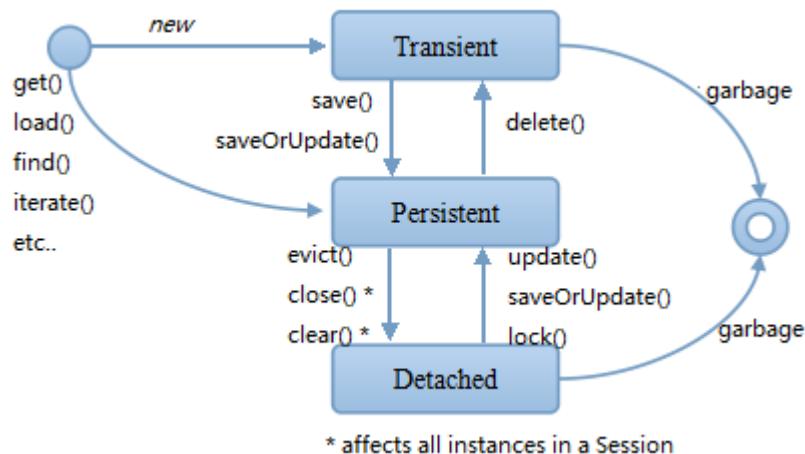
在 Hibernate 使用过程中，实体对象可以具有以下三种状态：

- 1) 临时状态：采用 new 关键字创建的对象，该对象未与 Session 发生关联（未调用 Session 的 API）。也叫临时对象。临时状态的对象会被 Java 的垃圾回收机制回收。
- 2) 持久状态：实体对象与 Session 发生关联（调用了 Session 的 get、load、save、update 等 API）。也叫持久对象。
- 3) 游离状态：原来是持久状态，后来脱离了 Session 的管理。如：Session 被关闭，对象将从持久状态变为游离状态，同时垃圾回收机制可以回收掉，不再占用缓存空间了。

5.2 处于持久状态的对象具有的特点

- 1) 对象生命周期持久，垃圾回收机制不能回收。
 - 2) 对象的数据可以与数据库同步（即对象中的数据发生改变，则数据库中的数据自动同步）。由 Session 对象负责管理和同步。
 - 3) 对象在 Session 的一级缓存中存放（或者说在 Session 缓存中的对象都是持久对象）。
- ◆ 注意事项：Session.close(); 有两个作用：①关闭连接、释放资源②使对象变为游离状态，当对象的引用不存在时，对象才被回收。没被回收时，对象中的数据还在！

5.3 三种状态下的对象的转换



5.4 批量操作：注意及时清除缓存

```
Transaction tx = session.beginTransaction();
for(int i=0;i<100000;i++){
    Foo foo = new Foo();
    session.save(foo); //设置 foo 属性
    if(i%50==0){ //够 50 个对象，与数据库同步下，并清除缓存
        session.flush(); //同步
        session.clear(); //清除缓存
    }
}
tx.commit();
```

5.5 案例：三种状态下的对象使用

当持久对象数据改变后，调用 session.flush()方法，会与数据库同步更新。

commit()方法，内部也调用了 flush()方法，因此使用 commit()方法可以省略 flush()方法的调用。

```

@Test
public void test1() {//Foo 实体有 id、name、salary、hireDate、marry 等属性
    Session session=HibernateUtil.getSession(); Transaction tx=session.beginTransaction();
    Foo foo=(Foo)session.get(Foo.class, 1); //foo 具有持久性
    foo.setName("chang");      foo.setSalary(6000);
    /** 提交事务，若后面不写 flush，只写提交 commit，则也能执行更新操作。因为 commit 在内部会先调用 flush，再提交事务，所以此时 flush 可不写 */
    tx.commit();
    /** 触发同步动作，同步和提交是两回事。数据有变化才同步（更新），没变化不会更新 */
    session.flush();           session.close(); //关闭 session 释放资源
}

@Test
public void test2(){
    Foo foo=new Foo();      foo.setName("tigger");      foo.setSalary(8000);
    foo.setMarry(true);     foo.setHireDate(new Date(System.currentTimeMillis()));
    Session session=HibernateUtil.getSession(); Transaction tx=session.beginTransaction();
    session.save(foo); //以上都是临时状态，此时由临时状态转为持久状态
    foo.setSalary(10000); //修改 foo 持久对象的数据，也是更新操作，可不写 update 方法
    tx.commit(); //同步、提交           session.close();
}

@Test
public void test3(){
    Session session=HibernateUtil.getSession(); Transaction tx=session.beginTransaction();
    Foo foo=(Foo)session.get(Foo.class, 2); //foo 具有持久性
    session.clear(); //下面的则不会与数据库同步了
    foo.setName("常");      foo.setSalary(8800);
    tx.commit(); //session.flush() //数据有变化才同步 update，没变化不会 update
    session.close();
}

```

5.6 一级缓存机制（默认开启）

一级缓存被称为 Session 级别的缓存：每个 Session 都有自己独立的缓存区，该缓存区随着 Session 创建而开辟（由 SessionFactory 创建），随着 Session.close()而释放。

该缓存区可以存储当前 Session 关联的持久对象。只有在缓存区中，Session 才管该对象。

5.7 一级缓存的好处

Hibernate 在查询时，先去缓存当中查找，如果缓存中没有，才去数据库查询。如果利用 Session 对同一个对象查询多次，第一次去数据库查，后续的会从缓存查询，从而减少了与数据库的交互次数。

5.8 管理一级缓存的方法

- 1) session.evict()方法：将对象清除。
- 2) session.clear()方法：清除所有对象。
- 3) session.close()方法：清除所有对象，并关闭与数据库的连接。
 - ◆ 注意事项：不同的 session 的缓存区不能交叉访问。
- 4) 案例：测试一级缓存

```

@Test
public void test1(){
    Session session=HibernateUtil.getSession();
    Foo foo1=(Foo)session.get(Foo.class, 1); //第一次查询
    System.out.println(foo1.getName()); //能出现 SQL 查询语句
    Foo foo2=(Foo)session.get(Foo.class, 1); //后续查询
    System.out.println(foo2.getSalary()); //不能出现 SQL 查询语句
    session.close();
}

@Test
public void test2(){
    Session session=HibernateUtil.getSession();
    Foo foo1=(Foo)session.get(Foo.class, 1); //第一次查询
    System.out.println(foo1.getName()); //能出现 SQL 查询语句
    session.evict(foo1); //或 session.clear(foo1);
    Foo foo2=(Foo)session.get(Foo.class, 1); //后续查询
    System.out.println(foo2.getSalary()); //能出现 SQL 查询语句
    session.close();
}

@Test
public void test3(){
    Session session=HibernateUtil.getSession();
    Foo foo1=(Foo)session.get(Foo.class, 1); //不同的对象，所以查询两次
    System.out.println(foo1.getName());
    Foo foo2=(Foo)session.get(Foo.class, 2); System.out.println(foo2.getName());
}

@Test
public void test4(){
    Session session=HibernateUtil.getSession();
    Foo foo1=(Foo)session.get(Foo.class, 1); System.out.println(foo1.getName());
    session.close();
    session=HibernateUtil.getSession(); //不同的 session 的缓存区不能交叉访问
    Foo foo2=(Foo)session.get(Foo.class, 1); //又一次查询
    System.out.println(foo2.getName());
}

```

5.9 延迟加载机制

Hibernate 在使用时，有些 API 操作是具有延迟加载机制的。

延迟加载机制的特点：当通过 Hibernate 的 API 获取一个对象结果后，该对象并没有数据库数据。当通过对对象的 getter 方法获取属性值时，才去数据库查询加载。

5.10 具有延迟加载机制的操作

- 1) session.load();//延迟加载查询，session.get()是立即加载查询
- 2) query.iterator();//查询
- 3) 获取关联对象的属性信息

- ◆ 注意事项：这些方法返回的对象，只有 id 属性（主键）有值，其他属性数据在使用的时候（调用 getXXX() 方法时，除主键外，调主键 get 方法不会发 SQL）才去获取。

5.11 常犯的错误

1) 报错：LazyInitializationException: could not initialize proxy - no Session，原因：代码中使用了延迟加载操作，但是 session 在加载数据前关闭了。只要看到这个类名：LazyInitializationException 就都是 session 过早关闭，后面的描述可能不同。

2) 报错：NonUniqueObjectException: a different object with the same identifier value was already associated with the session，原因：有两个不同对象，但是主键，即 id 却相同。例如：

```
Session session=HibernateUtil.getSession(); Transaction tx=session.beginTransaction();
Account account1=(Account)session.get(Account.class, 1010); //将 account1 放入缓存
System.out.println(account1.getRealName()); System.out.println(account1.getIdcardNo());
Account account2=new Account(); account2.setId(1010);
//update 操作会将 account2 放入缓存，此时会出现异常，缓存中已经存在一个 1010 对象，不允许再放入 id 为 1010 的对象
session.update(account2); tx.commit(); session.close();
```

5.12 延迟加载的原理

在使用延迟加载操作后，Hibernate 返回的对象是 Hibernate 利用 CGLIB 技术 (cglib.jar) 新生成的一个类型（动态的在内存中生成）。在新类型中，将属性的 getter 方法重写。新生成的类是原实体类的子类（继承关系）。

```
例如：public class Foo$$EnhancerByCGLIB$$87e5f322 extends Foo{
    public String getName(){
        //判断是否已加载过数据，如果加载过，返回 name 值
        //没有加载过，则发送 SQL 语句查询加载数据，然后返回 name 值
    }
}
```

- ◆ 注意事项：一般情形： *.java --> *.class --> 载入类加载器 --> 执行
延迟加载：javassist.jar/cglib.jar（生成新类型） --> 类加载器 --> 执行

5.13 Session 的 get 和 load 方法的区别

- 1) 相同点：两者都是按“主键”做条件查询。
- 2) 不同点：
 - ①get 是立刻加载；load 是延迟加载。
 - ②get 返回的对象类型是实体类型；load 返回的是动态生成的一个代理类（动态代理技术），该代理类是实体类的子类。
 - ③get 未查到数据返回 null；load 未查到数据抛出 ObjectNotFoundException 异常。
- ◆ 注意事项：若实体类用了 final 修饰，则破坏了延迟加载机制，那么 load 效果与 get 就完全相同了。

5.14 延迟加载的好处

- 1) 提高了内存的使用效率。
- 2) 可以使数据访问降低并发量。

5.15 案例：测试延迟加载

```

@Test
public void test10{      Session session=HibernateUtil.getSession();
    //load 是延迟加载, foo 没有数据
    Foo foo=(Foo)session.load(Foo.class, 1); //此时还没去数据库查询
    //session.close(); //放这里报错, session 关的过早 could not initialize proxy - no Session
    System.out.println(foo.getName()); //第一次调用属性的 getter 方法时触发查询
    session.close(); //放这里不报错, 对象没被回收
    System.out.println(foo.getSalary());
}

@Test
public void test20{      Session session=HibernateUtil.getSession();
    Foo foo=(Foo)session.load(Foo.class, 1); //此时还没去数据库查询
    //类 org.tarena.entity.Foo$$EnhancerByCGLIB$$87e5f322, 由 cglib.jar 生产
    System.out.println(foo.getClass().getName());           session.close();
}

```

5.16 案例：重构 NetCTOSS 资费管理模块

step1：引入 Hibernate 开发框架（jar 包和主配置文件）

step2：采用 Hibernate 操作 COST_CHANG 表

1) 添加实体类

```

private Integer id; //资费 ID      private String name; //资费名称 NAME
private Integer baseDuration; //包在线时长 BASE_DURATION
private Float baseCost; //月固定费 BASE_COST
private Float unitCost; //单位费用 UNIT_COST
private String status; //0: 开通 1: 暂停; STATUS
private String descr; //资费信息说明 DESCRIPTOR
private Date startTime; //启用日期 STARTTIME
private Date creaTime; //创建时间 CREATETIME

```

2) 追加 Cost.hbm.xml

```

<hibernate-mapping><!-- <hibernate-mapping package="包名写这也行" -->
<class name="com.tarena.netctoss.entity.Cost" table="COST_CHANG" catalog="system">
    <id name="id" type="java.lang.Integer">
        <column name="ID" />
        <generator class="sequence">
            <param name="sequence">COST_SEQ_CHANG</param>
        </generator>
    </id>
    <property name="name" type="java.lang.String">
        <column name="NAME" />
    </property>
    <property name="baseDuration" type="java.lang.Integer">
        <column name="BASE_DURATION" />
    </property>
    <property name="baseCost" type="java.lang.Float"><!-- 类型要和实体定义的相同

```

```

-->
    <column name="BASE_COST" />
</property>
.....其他略.....
</class>
</hibernate-mapping>

```

LICHOO

◆ 注意事项：

- ❖ 实体类和 hbm.xml 必须保持一致！列名写错则会报：不能读取实体类。
- ❖ junit 测试右键点 Copy Trace 查看错误列。

step3：借用 2.3 节中 step6 的 HibernateUtil 类

step4：按 CostDAO 接口重构一个 DAO 实现组件：HibernateCostDAOImpl

```

public void delete(int id) throws DAOException {
    Session session=HibernateUtil.getSession();
    Transaction tx=session.beginTransaction();
    Cost cost=new Cost();      cost.setId(id);      session.delete(cost);
    tx.commit();      session.close();
}

public List<Cost> findAll() throws DAOException {
    Session session=HibernateUtil.getSession();
    Transaction tx=session.beginTransaction();
    String hql="from Cost";      Query query=session.createQuery(hql);
    List list=query.list();      tx.commit();      session.close();      return list;
}

public List<Cost> findAll(int page, int rowsPerPage) throws DAOException {
    Session session=HibernateUtil.getSession(); //分页查询
    Transaction tx=session.beginTransaction();      String hql="from Cost";
    Query query=session.createQuery(hql);
    int start=(page-1)*rowsPerPage; //设置分页查询参数
    query.setFirstResult(start); //设置抓取记录的起点，从 0 开始（第一条“记录”）
    query.setMaxResults(rowsPerPage); //设置抓取多少条记录
    List list=query.list(); //按分页参数查询
    tx.commit();      session.close();      return list;
}

public Cost findById(Integer id) throws DAOException {
    Session session=HibernateUtil.getSession();
    Transaction tx=session.beginTransaction();
    Cost cost=(Cost)session.load(Cost.class,id);      tx.commit();
    String name=cost.getName();      session.close();      return cost;
}

public Cost findByName(String name) throws DAOException {
    //select * from COST_CHANG where NAME=?
    String hql="from Cost where name=?";
    Session session=HibernateUtil.getSession();
    Transaction tx=session.beginTransaction();
    Query query=session.createQuery(hql);
    query.setString(0, name); //注意 Hibernate 赋值从 0 开始，即第一个问号
    Cost cost=(Cost)query.uniqueResult(); //适用于只有一行查询结果返回
}

```

```

    //如果返回记录为多条，则会报错，多条用 query.list();
    tx.commit(); session.close(); return cost;
}

public int getTotalPages(int rowsPerPage) throws DAOException {
    //select count(*) from COST_CHANG
    String hql="select count(*) from Cost";//类名
    Session session=HibernateUtil.getSession();
    Transaction tx=session.beginTransaction();
    Query query=session.createQuery(hql); Object obj=query.uniqueResult();
    int totalRows=Integer.parseInt(obj.toString());
    tx.commit(); session.close();
    if(totalRows%rowsPerPage==0){ return totalRows/rowsPerPage;
    }else { return (totalRows/rowsPerPage)+1; }
}

public void save(Cost cost) throws DAOException {
    Session session=HibernateUtil.getSession();
    Transaction tx=session.beginTransaction();//下面的设置属性建议写到 Action 中
    cost.setStatus("1"); cost.setCreateTime(new Date(System.currentTimeMillis()));
    session.save(cost); tx.commit(); session.close();
}

public void update(Cost cost) throws DAOException {
    Session session=HibernateUtil.getSession();
    Transaction tx=session.beginTransaction();//下面设置属性建议写到 Action 中，不
    写 DAO 中，因为 update 有通用性可封装，到时无法确定 setXX 方法
    Cost cost1=(Cost)session.get(Cost.class, cost.getId());
    cost1.setName(cost.getName()); cost1.setBaseDuration(cost.getBaseDuration());
    cost1.setUnitCost(cost.getUnitCost()); cost1.setDescr(cost.getDescr());
    session.update(cost1); tx.commit(); session.close();
}

```

step5：修改 DAOFactory

```
private static CostDAO costDAO = new HibernateCostDAOImpl();
```

5.17 Java Web 程序中如何用延迟加载操作

(OpenSessionInView)

1) Java Web 程序工作流程：

*.action-->Action-->DAO（延迟 API）-->JSP（利用标签或 EL 获取数据，会触发延迟加载数据）-->生成响应 HTML 页面给浏览器。

2) 基于上述原因，在 DAO 中不能关闭 Session，需要将 Session 关闭放到 JSP 解析之后（把 Session 的关闭延迟到 View 组件运行完之后），这种模式被称为 OpenSessionInView。

3) OpenSessionInView 和 ThreadLocal：

使用 OpenSessionInView 必须满足 Session 的线程单例，一个线程分配一个 Session，在该线程的方法中可以获得该 Session，具体使用 ThreadLocal（一个线程为 key 的 Map）。

Hibernate 支持的 Session 线程单例，配置文件中：

```
<property name="current_session_context_class">thread</property>
```

然后调用：sessionFactory.getCurrentSession();//自动实现线程单例

4) OpenSessionInView 模式也可以采用以下技术实现：

①利用 Struts2 的拦截器（将关闭 session 的操作写在拦截器中）。

step1：基于 ThreadLocal 技术改造 2.3 中 step6 的 HibernateUtil 类

```
public class HibernateUtil { private static SessionFactory sf;
    private static ThreadLocal<Session> sessionLocal = new ThreadLocal<Session>();
    static{//不用每次都加载配置信息，所以放 static 块中，否则耗费资源
        Configuration conf=new Configuration();
        conf.configure("/hibernate.cfg.xml");//加载主配置 hibernate.cfg.xml
        sf=conf.buildSessionFactory();//获取 SessionFactory
    }
    /** 同一个线程，只创建一个 session，创建出来后利用 ThreadLocal 将 session 与当前线程绑定 */
    public static Session getSession(){ Session session=sessionLocal.get();
        if(session==null){//当前线程第一次调用，创建一个
            session =sf.openSession();
            sessionLocal.set(session);//将 session 存取 ThreadLocal
        }
        return session;//如果能取到 session，说明当前线程已经创建过 session
    }
    /** 把关闭 session 也封装一下 */
    public static void closeSession(){ Session session=sessionLocal.get();
        sessionLocal.set(null); if(session.isOpen()){
            session.close();//关闭 session 和释放 ThreadLocal 空间
        }
    }
    /** 简单测试一下 */
    public static void main(String[] args){
        Session session1 = HibernateUtil.getSession();
        Session session2 = HibernateUtil.getSession();
        System.out.println(session1==session2);//true
    }
}
```

step2：创建拦截器

```
public class OpenSessionInViewInterceptor extends AbstractInterceptor {//继承抽象类
    @Override
    public String intercept(ActionInvocation arg0) throws Exception {
        Session session=HibernateUtil.getSession();
        //开启事务，或 Transaction tx=session.getTransaction(); tx.begin();
        Transaction tx=session.beginTransaction();//等于以上两步
        System.out.println("开启事务");
        try{ arg0.invoke();//执行 action、result --> jsp
            if(!tx.wasCommitted()){//当两个 action 连续调用时，避免重复提交
                tx.commit();//提交事务 System.out.println("提交事务");
            }
            return null;
        }catch (Exception e){ tx.rollback();//回滚事务 System.out.println("回滚事务");
            e.printStackTrace(); throw e;//受 AbstractInterceptor 类影响，必须抛异常
        }
        finally{ HibernateUtil.closeSession(); //关闭 session
            System.out.println("关闭事务");
        }
    }
}
```

常

- ◆ 注意事项：重复提交的情况 redirectAction： add.action-->拦截器开启事务-->AddCostAction-->拦截器开启事务-->ListCostAction-->cost_list.jsp-->拦截器提交事务-->拦截器提交事务

step3：在 NetCTOSS 项目中的 struts-cost.xml 中配置拦截器，添加内容如下：

```
<interceptors>
    <interceptor name="opensessioninview"
        class="com.tarena.netctoss.interceptor.OpenSessionInViewInterceptor">
    </interceptor>
    <interceptor-stack name="opensessionStack">
        <interceptor-ref name="opensessioninview"></interceptor-ref>
        <interceptor-ref name="defaultStack"></interceptor-ref>
    </interceptor-stack><!--默认的拦截器不能丢，否则表单接收等不正常了-->
</interceptors>
<!-- 定义全局拦截器引用，若其他配置文件也有全局拦截器，则覆盖原来的，找最近的
    (即在当前 struts-cost.xml 中有效) 类似于 Java 中的局部变量 --&gt;
    &lt;default-interceptor-ref name="opensessionStack"&gt;&lt;/default-interceptor-ref&gt;</pre>
```

step4：将 5.16 案例中 step4 中的 HibernateCostDAOImpl 类里的所有方法中的开启事务、提交事务、关闭 Session 全部删除。

②利用 Filter 过滤器。

```
public void doFilter(request,response,chain){
    //前期处理逻辑
    chain.doFilter(request,response);//调用后续 action,result 组件
    //后期处理逻辑。关闭 session
}
```

③利用 Spring 的 AOP 机制。

一百三十、关联映射

关联映射主要是在对象之间建立关系。开发者可以通过关系进行信息查询、添加、删除和更新操作。

如果不使用 Hibernate 关联关系映射，我们也可以取到用户对应的服务。

```
Account account = (Account)session.get(Account.class, 1); //取到用户信息  
String hql = "from Service s where s.accountId=1";  
Query query = session.createQuery(hql); //取到用户对应的服务  
List<Item> list = query.list();
```

而 Hibernate 提供的关联映射，更方便一些。

6.1 一对多关系 one-to-many

step1：新建项目，导入 Hibernate 开发包，借用 NetCTOSS 项目中的实体：Account 和 Service，配置 hibernate.cfg.xml 和两个实体的 hbm.xml 映射文件。

step2：一个 Account 帐号对应多个 Service 服务，所以为一对多关系。因此为 One 方 Account 实体类添加 Set 集合属性，以及对应的 get/set 方法。

```
//追加属性，用于存储相关联的 Service 信息  
private Set<Service> services = new HashSet<Service>();
```

step3：在 One 方 Account.hbm.xml 映射文件中，加入 Set 节点的映射

简单说明：

```
<set name="属性名">  
    <!-- 关联条件，column 写外键字段，会默认的与 ACCOUNT 表的主键相关联 -->  
    <key column="指定关联条件的外键字段"></key>  
    <!-- 指定采用一对多关系，class 指定关联的类型 -->  
    <one-to-many class="要关联的另一方 (N 方)" />  
</set>
```

具体实现：

```
<!-- 描述 services 属性，采用一对多关系加载 service 记录 -->  
<!-- 是 list 集合用<list name=""></list> set 集合用<set name=""></set> -->  
<set name="services">  
    <key column="ACCOUNT_ID"></key> <!-- ACCOUNT_ID 是 Service 表中字段 -->  
    <one-to-many class="org.tarena.entity.Service"/>  
</set>
```

step4：借用 5.17 节 4) 中的 step1 中的 HibernateUtil 类

step5：新建 TestOneToMany.java 类用于测试

```
@Test  
public void test1(){  
    Session session = HibernateUtil.getSession();  
    Account account = (Account) session.load(Account.class, 1011); //第一次发送 SQL 查询  
    System.out.println(account.getRealName()); System.out.println(account.getIdcardNo());  
    /** 显示与当前帐号相关的 Service 业务帐号，以前的方式需要写 hql:  
    String hql = "from Service where ACCOUNT_ID=1011"; 用了关联映射则不用写 hql 了  
    */  
    Set<Service> services = account.getServices(); //延迟加载，第二次发送 SQL 查询  
    for(Service s:services){
```

```
System.out.println(s.getId()+" "+s.getOsUsername()+" "+s.getUnixHost()); }  
session.close(); }
```

6.2 多对一关系 many-to-one

step1: 新建项目, 导入 Hibernate 开发包, 借用 NetCTOSS 项目中的实体: Account 和 Service, 配置 hibernate.cfg.xml 和两个实体的 hbm.xml 映射文件。

step2: 可多个 Service 服务对应一个 Account 帐号, 所以为多对一关系。因此为 N 方 Service 实体类添加 Account 属性, 以及对应的 get/set 方法。

```
//追加属性, 用于存储关联的 Account 信息  
private Account account;//已经包含 accountId 了, 原来的 accountId 属性删! 否则报错
```

- ◆ 注意事项: Service 实体原来的 accountId 属性删, 相应的 get/set 方法也删, Service 的映射文件对应的描述也删! 否则报错: org.hibernate.MappingException: Repeated column in mapping for entity: org.tarena.entity.Service column: ACCOUNT_ID

step3: 在 N 方 Service.hbm.xml 映射文件中描述 account 属性

简单说明:

```
<many-to-one name="属性名" class="要关联的另一方类型 Account"  
column="关联条件的外键字段"/> <!-- 指明外键字段, 不写主键 -->
```

- ◆ 注意事项: 此时没有<set name="属性名"></set>标签。

具体实现:

```
<!-- 描述 account, 采用多对一关系加载 -->  
<many-to-one name="account" class="org.tarena.entity.Account"  
column="ACCOUNT_ID"/> <!-- 指明外键字段, 不写主键 -->
```

step4: 借用 5.17 节 4) 中的 step1 中的 HibernateUtil 类

step5: 新建 TestManyToOne.java 类用于测试

```
@Test  
public void test1(){ Session session=HibernateUtil.getSession();  
Service service=(Service)session.load(Service.class, 2002);  
//System.out.println(service.getId());//结果为 2002, 但没去查数据库! 主键传啥显示啥  
System.out.println(service.getOsUsername());//第一次 SQL 查询  
System.out.println(service.getUnixHost());  
//查看账务账号的真实名字、身份证、Account_ID  
System.out.println(service.getAccount().getId());//第二次 SQL 查询  
System.out.println(service.getAccount().getRealName());  
System.out.println(service.getAccount().getIdcardNo()); session.close(); }
```

6.3 多对多关联映射 many-to-many

数据库设计是采用 3 张数据表, 有一张是关系表。例如:

ADMIN_INFO-->ADMIN_ROLE<--ROLE

中间的关系表不用映射, 只映射两端的表。但如何对关系表进行操作呢?

答: Hibernate 会自动的通过已经映射的表, 进行关系表操作。

- ◆ 注意事项:
 - ◆ 多对多关系是默认的级联操作, 可加 cascade 属性, 但是加了之后相当于进入

一个循环，若删除 A 数据，则所有表中有 A 的数据则都被删除！因此一般都是采用一对多或多对一，从而破坏这个循环。详情可看 6.4 节。

- ❖ 多对多关系一定有第三张表，间接实现多对多关系，两表之间不可能有多对多关系！

案例 1：step1：在 Admin 实体类中添加一个 Set 集合属性

```
//追加属性，用于存储相关联的 Role 信息  
private Set<Role> roles=new HashSet<Role>();
```

step2：在 Admin.hbm.xml 中定义属性的映射描述

简单说明：

```
<set name="关联属性名" table="中间的关系表">  
    <key column="关系表中与当前一方关联的字段"></key>  
    <many-to-many class="关联的另一方类型"  
        column="关系表中与另一方关联的字段" />  
</set>
```

具体实现：

```
<!-- 描述 roles 属性，采用多对多加载 Role 对象的数据 -->  
<set name="roles" table="ADMIN_ROLE_CHANG">  
    <key column="ADMIN_ID"></key>  
    <many-to-many class="org.tarena.entity.Role" column="ROLE_ID" />  
</set>
```

step3：新建 TestManyToMany 类，用于测试

```
@Test  
public void testFind(){//测试查询  
    Session session=HibernateUtil.getSession();  
    Admin admin=(Admin)session.get(Admin.class, 1001);  
    System.out.println(admin.getName());      System.out.println(admin.getEmail());  
    Set<Role> roles=admin.getRoles();//具有的角色  
    for(Role role:roles){  System.out.println(role.getId()+" "+role.getName()); } }
```

```
@Test  
public void testAdd(){//测试添加管理员，指定角色  
    Session session=HibernateUtil.getSession(); Transaction tx=session.beginTransaction();  
    Admin admin=new Admin();  admin.setName("常 1");  admin.setCode("chang1");  
    admin.setPassword("123123");  
    admin.setEnrollDate(new Date(System.currentTimeMillis()));  
    Role role1=(Role)session.load(Role.class, 20);//追加角色，角色 ID 为 20  
    Role role2=(Role)session.load(Role.class, 24);//追加角色，角色 ID 为 24  
    admin.getRoles().add(role1);      admin.getRoles().add(role2);//把角色加入集合中  
    session.save(admin);          tx.commit();          session.close(); }
```

```
@Test  
public void testDelete(){    Session session=HibernateUtil.getSession();  
    Transaction tx=session.beginTransaction();  
    Admin admin=(Admin)session.get(Admin.class, 23);//查找 ID 为 23 的管理员  
    Role role1=(Role)session.get(Role.class,2);//查找 ID 为 2 的角色  
    admin.getRoles().remove(role1);//取消该管理员 ID 为 2 的角色
```

```
//增加角色 admin.getRoles().add(role1); //和上个方法相同  
session.update(admin); //更新 admin tx.commit(); session.close();
```

案例 2: step1: 在 Role 实体类中添加一个 Set 集合属性

```
//追加属性，用于存储 admin 信息  
private Set<Admin> admins=new HashSet<Admin>();
```

step2: 在 Role.hbm.xml 中定义属性的映射描述

```
<!-- 采用多对多加载 admin 信息 -->  
<set name="admins" table="ADMIN_ROLE_CHANG">  
    <key column="ROLE_ID"></key><!-- 与 admin.hbm.xml 是相反的 -->  
    <many-to-many column="ADMIN_ID" class="org.tarena.entity.Admin">  
    </many-to-many>  
</set>
```

step3: 在 TestManyToMany 类中添加方法，用于测试

```
@Test  
public void testFindRole(){//根据 role 查询相关 admin  
    Session session=HibernateUtil.getSession(); Role role=(Role)session.load(Role.class, 1);  
    System.out.println(role.getName()); //查找 ID 为 1 的角色，然后显示名称  
    for(Admin admin:role.getAdmins()){ //显示哪些 admin 具有此角色  
        System.out.println(admin.getId()+" "+admin.getCode()+" "+admin.getName());  
    } session.close();}
```

6.4 关联操作（查询 join fetch/级联 cascade）

1) **查询操作:** 建立关联映射后，默认情况下在调用关联属性的 getter 方法时，会再发送一个 SQL 加载关系表数据。如果需要将关联数据与主对象一起加载（两个 SQL 查询合成一个 SQL 查询），可以采用下面的方法：

① 在 hbm.xml 关联属性映射描述中（下例是在 Account.hbm.xml 中），使用 lazy="false" fetch="join"（不推荐使用！影响面太广）

```
<!-- 关联属性的 lazy 属性默认为 true，主属性默认为 false，主属性也可加 lazy 属性-->  
<!-- <set name="services" lazy="false" fetch="join">, lazy="false"关闭了延迟操作，与主对象一起实例化。fetch 默认 select: 单独发送一个 SQL 查询。join: 表连接用一个 SQL 查询。-->
```

不推荐用因为影响的映射范围太广，推荐使用 HQL -->

```
<set name="services" lazy="false" fetch="join">  
    <key column="ACCOUNT_ID"></key><!-- ACCOUNT_ID 是 Service 表中字段 -->  
    <one-to-many class="org.tarena.entity.Service"/>  
</set>
```

② 编写 HQL，采用 join fetch 关联属性方式实现。（推荐使用！）

```
@Test  
/** 我们期望：当执行(Account)session.get(Account.class, 1001);语句，取出 Account 后，在属性 services 已经填充了所有的服务项 service */  
public void test1(){ Session session=HibernateUtil.getSession();  
    Account account=(Account)session.get(Account.class, 1011);  
    session.close(); //当 hbm.xml 中关联属性设置了 lazy="false"，在这里关闭能正常执行
```

```

Set<Service> services=account.getServices();//若 lazy="true"默认值, 则报错
for(Service s:services){
    System.out.println(s.getId()+" "+s.getOsUsername()+" "+s.getUnixHost()); }
System.out.println(account.getRealName()); }

@Test /** TestOneToMany 中添加方法 */
public void test2(){
    Session session=HibernateUtil.getSession();
    //String hql="from Account where id=?"; //与 TestOneToMany 中 test1 效果一样
    //在下面的 hql 中把需要关联的属性写出来, 效果与在 hbm.xml 加 fetch 的效果一样
    String hql="from Account a join fetch a.services where a.id=?"; //只执行一次 SQL 查询
    Query query=session.createQuery(hql);
    query.setInteger(0, 1011);          Account account=(Account)query.uniqueResult();
    System.out.println(account.getRealName()); System.out.println(account.getIdcardNo());
    Set<Service> services=account.getServices();
    for(Service s:services){
        System.out.println(s.getId()+" "+s.getOsUsername()+" "+s.getUnixHost()); }
    session.close(); }

@Test /** TestManyToOne 中添加方法 */
public void test20(){
    Session session=HibernateUtil.getSession();
    String hql="from Service s join fetch s.account where s.id=?"; //只执行一次 SQL 查询
    Query query=session.createQuery(hql);
    query.setInteger(0, 2002); //Hibernate 设置问号从 0 开始
    Service service=(Service)query.uniqueResult();
    System.out.println(service.getOsUsername());
    System.out.println(service.getUnixHost());
    //查看账务账号的真实名字、身份证、Account_ID
    System.out.println(service.getAccount().getId());
    System.out.println(service.getAccount().getRealName());
    System.out.println(service.getAccount().getIdcardNo()); session.close(); }

```

2) 级联操作

当对主对象增删改时, 可以对关系属性中的数据也相应的执行增删改。

例如: session.delete(account); 当 account 中有 services 且 services 有值, 则删除 account 时, 相关联的 services 也被删除。

级联操作执行过程: 首先级联操作要开启, 然后判断级联属性是否有数据, 若无数据则没有级联操作, 若有数据则看是否有更改, 有更改才有级联操作。

级联操作默认是关闭的, 如果需要使用, 可以在关联属性映射部分添加 cascade 属性, 属性值有: ①none: 默认值, 不支持级联。②delete: 级联删除。③save-update: 级联添加和更新④All: 级联添加、删除、更新等。案例: 级联增加:

step1: 修改 Account.hbm.xml:

```

<set name="services" cascade="all">
    <key column="ACCOUNT_ID"></key>
    <one-to-many class="org.tarena.entity.Service"/>
</set>

```

step2: 新建 TestCascade 类, 用于测试级联操作

```
@Test
```

```

public void testAdd(){//采用级联方式添加一个 Account 和两个 Service
    Account account=new Account();//一个 Account, 简单操作: 只把非空列设置上
    account.setLoginName("chang");      account.setLoginPasswd("123");
    account.setRealName("常");           account.setIdcardNo("1234567890");
    account.setTelephone("123456789");
    Service service1=new Service();//两个 Service, 简单操作: 只把非空列设置上
    service1.setAccount(account);//因 accountId 被删了, 所以这里用 account, 它里面有
id
    service1.setOsUsername("chang1");    service1.setLoginPassword("111");
    service1.setUnixHost("192.168.0.20"); service1.setCostId(1);
    Service service2=new Service();//同理
    service2.setAccount(account);       service2.setOsUsername("chang2");
    service2.setLoginPassword("222");     service2.setUnixHost("192.168.0.23");
    service2.setCostId(2);
    /** 将 service1 和 service2 添加到 account.services 集合中, 否则不会级联添加这两个
service, 同时 Hibernate 会检测, 新增数据若是 services 中的原有的, 则不往数据库添加 */
    account.getServices().add(service1);   account.getServices().add(service2);
    Session session=HibernateUtil.getSession(); Transaction tx=session.beginTransaction();
    //hbm.xml 添加 cascade="all", 则会对 account.service 中的数据执行级联添加
    session.save(account);      tx.commit();      session.close();    }
}

```

3) inverse 属性作用

默认情况下, 采用了级联操作, Hibernate 在执行 insert、update、deleted 基本操作后, 还要执行 update 关系字段的操作(即关系维护工作, 上例中维护的为 ACCOUNT_CHANG 表中的 ID 和 SERVICE_CHANG 表中的 ACCOUNT_ID)。

默认是关联对象双方都要负责关系维护。所以在上例中, 控制台在最后会有两个 update 语句, 因为当前添加了一个 Account 和两个 Service, 所以 One 方要维护两个 Service, 即两个 update 语句。如果数据量很大, 则要维护 N 个 Service, 则有 N 个 update 语句, 此时就会影响性能。

为了使 update 语句不出现, 可以在 Account.hbm.xml 中的级联属性加 inverse="true" 属性, 即当前一方放弃关系维护, 将这项工作交给对方负责。

例如: <set name="services" inverse="true" cascade="all">...</set>

◆ 注意事项: 遇到一对多、多对一关系映射时, 把 inverse="true" 属性加到 one-to-many 一方 (One 方放弃, Many 方维护)。能起到一定的优化作用。

4) 级联删除

在 TestCascade 类中添加方法:

```

@Test
public void testDelete(){
    /** Account account=new Account(); account.setId(id); 级联删除, 不要采用
new Account()方法, 因为 new 出来的 account, 它的 services 是空的, 那么将是单表操
作, 将报错: 违反完整性约束 */
    Session session=HibernateUtil.getSession(); Transaction tx=session.beginTransaction();
    Account account=(Account)session.load(Account.class, 500);//应该先查找
    session.delete(account);//再删除      tx.commit();      session.close();    }
}

```

Hibernate 级联删除的缺点：delete 是按 id（主键）一条一条删的，不是按关系字段删的，当数据量小时可用 Hibernate 的级联删除，简单方便些。

但是，但当数据量大时，Hibernate 的级联删除效率低，则不建议使用 Hibernate 的级联删除，建议采用 HQL 语句的方式，例如：

```
delete from Account where id=?      delete from Service where account.id=?
```

◆ 注意事项：

- ❖ 级联删除，不写 inverse="true"，且数据库中 SERVICE_CHANG 表中的 ACCOUNT_ID 为 NOT NULL 约束，那么程序最后会执行 update，会设置 ACCOUNT_ID=null，那么将与数据库冲突！报错！所以，应当加上 inverse="true"。

6.5 继承关系映射

可以将数据表映射成具有继承关系的实体类。Hibernate 提供了 3 方式的继承映射。

- 1) 将一个表映射成父类和子类：采用<subclass>描述子类。
- 2) 将子类表映射成父类和子类（无父类表）：采用<union-subclass>描述。
- 3) 将父类表和子类表映射成父类和子类：采用<joined-subclass>描述子类继承关系映射。

```
<joined-subclass name="子类类型" table="子类表" extends="父类类型">  
    <key column="子类表与父类表关联的字段"></key>  
    //子类中其他属性的映射 property 元素  
</joined-subclass>
```

- 4) 案例：将父类表和子类表映射成父类和子类

step1：PRODUCT 表为父表，CAR 表和 BOOK 表都为子表。建表语句如下：

```
CREATE TABLE PRODUCT(  
    ID      NUMBER(5) CONSTRAINT PRODUCT_ID_PK PRIMARY KEY,  
    NAME    VARCHAR2(20),  
    PRICE   NUMBER(15,2),  
    PRODUCT_PIC  VARCHAR2(100)      );  
CREATE SEQUENCE product_seq;  
CREATE TABLE BOOK(  
    ID      NUMBER(5) CONSTRAINT BOOK_ID_PK PRIMARY KEY,  
    AUTHOR    VARCHAR2(20),  
    PUBLISHING  VARCHAR2(50),  
    WORD_NUMBER VARCHAR2(20),  
    TOTAL_PAGE VARCHAR2(20)      );  
CREATE TABLE CAR(  
    ID      NUMBER(5) CONSTRAINT CAR_ID_PK PRIMARY KEY,  
    BRAND   VARCHAR2(20),  
    TYPE    VARCHAR2(1),  
    COLOR   VARCHAR2(50),  
    DISPLACEMENT VARCHAR2(20)      );
```

- ◆ 注意事项：父表即把子表中相同的字段提取出来，作为父表。如：书和汽车都有 ID、名字、价格、产品图片。

step2：创建实体类 Product、Book 和 Car，其中 Book 和 Car 类需要继承 Product。

```
例如：public class Book extends Product{ ... }、public class Car extends Product{ ... }
```

step3：添加映射文件：Product.hbm.xml、Book.hbm.xml、Car.hbm.xml

1) Product.hbm.xml

```
<hibernate-mapping>
    <class name="org.tarena.entity.Product" table="PRODUCT">
        <id name="id" type="integer" column="ID">
            <generator class="sequence"><!-- 指定序列 -->
                <param name="sequence">PRODUCT_SEQ</param>
            </generator>
        </id>
        <property name="name" type="string" column="NAME"></property>
        .....其他部分略
    </class>
</hibernate-mapping>
```

2) Book.hbm.xml

```
<hibernate-mapping><!-- 描述了 Book 与 Product 的描述信息 -->
    <!-- name: 指明当前子类。table: 哪个表。extends: 继承哪个父类。 -->
    <joined-subclass name="org.tarena.entity.Book" table="BOOK"
        extends="org.tarena.entity.Product">
        <key column="ID"></key><!-- BOOK 表中哪个字段与 PRODUCT 表关联 -->
        <!-- 这里不自动增长, Hibernate 会自动的把 Product 中的主键值拿过来-->
        <property name="author" type="string" column="AUTHOR"></property>
        .....其他部分略
    </joined-subclass>
</hibernate-mapping>
```

3) Car.hbm.xml

```
<hibernate-mapping><!-- 描述了 Book 与 Product 的描述信息 -->
    <joined-subclass name="org.tarena.entity.Car" table="CAR"
        extends="org.tarena.entity.Product">
        <key column="ID"></key><!-- CAR 表中哪个字段与 PRODUCT 表关联 -->
        <!-- 这里不自动增长, Hibernate 会自动的把 Product 中的主键值拿过来-->
        <property name="brand" type="string" column="BRAND"></property>
        .....其他部分略
    </joined-subclass>
</hibernate-mapping>
```

step4：创建 TestExtends 类，用于测试继承关系映射

```
@Test
public void testAddBook() {      Session session=HibernateUtil.getSession();
    Transaction tx=session.beginTransaction();      Book book=new Book();
    book.setName("常的书");//设置 product 属性      book.setPrice(50);
    book.setProductPic("a.jsp");
    book.setAuthor("常");//设置 book 属性      book.setPublishing("BO 出版社");
    book.setTotalPage("20");                      book.setWordNumber("10000");
    session.save(book);//执行保存      tx.commit();      session.close();  }
}

@Test
public void testFindBook() {      Session session=HibernateUtil.getSession();
    Book book=(Book)session.load(Book.class, 1);  System.out.println(book.getName());
    System.out.println(book.getPrice());           System.out.println(book.getAuthor());
    System.out.println(book.getPublishing());       session.close();  }
}

@Test
public void testDeleteBook(){      Session session=HibernateUtil.getSession();}
```

```
Transaction tx=session.beginTransaction();
Book book=(Book)session.get(Book.class, 3);           session.delete(book);
tx.commit();      session.close();                      }

@Test
public void testAddCar(){   Session session=HibernateUtil.getSession();
    Transaction tx=session.beginTransaction();
    Car car=new Car();//product 信息      car.setName("Q7");      car.setPrice(800000);
    car.setProductPic("b.jsp");
    car.setBrand("奥迪");//car 信息       car.setType("J");//J 轿车 K 卡车
    car.setColor("黑色");                  car.setDisplacement("3.0");//排量
    session.save(car);//执行保存        tx.commit();      session.close();    }

@Test
public void testFindAllBook(){      Session session=HibernateUtil.getSession();
    String hql="from Book";//from car 为所有汽车, from product 为所有商品
    Query query=session.createQuery(hql);      List<Book> books=query.list();
    for(Book book:books){ System.out.println(book.getId()+" "+book.getName()); } }
```

一百三十一、Hibernate 查询方法

7.1 HQL 查询

Hibernate Query Language 简称 HQL。

HQL 语句是面向对象的一种查询语言。HQL 针对 Hibernate 映射之后的实体类型和属性进行查询（若出现表名和字段名则为错误的！）。

7.2 HQL 和 SQL 的相同点

- 1) 都支持 select、from、where、group by、order by、having 子句……。
- 2) 都支持+、-、*、/、>、<、>=、<=、<>等运算符和表达式。
- 3) 都支持 in、not in、between...and、is null、is not null、like、or 等过滤条件。
- 4) 都支持分组统计函数 count、max、min、avg、sum。

7.3 HQL 和 SQL 的不同点

- 1) HQL 区分大小写（除了关键字外）。
 - 2) HQL 使用类名和属性名，不能使用表名和字段名。
 - 3) HQL 不能使用 select * 写法。
 - 4) HQL 不能使用 join...on 中的 on 子句。
 - 5) HQL 不能使用数据库端的函数。
- ◆ 注意事项：HQL 中 select count(*) 可以使用。

7.4 HQL 典型案例

- 1) 案例 1：一个主键的情况

step1：借用之前的 Account 实体、Account.hbm.xml、hibernate.cfg.xml

step2：新建 TestHQL 类，用于测试 HQL。查询操作可不写事务控制语句。

```

@Test //SQL: select * from ACCOUNT_CHANG
public void test1() {//查询所有账务账号信息           String hql="from Account";
    Session session=HibernateUtil.getSession();      Query
query=session.createQuery(hql);      List<Account> list=query.list();//如果查询出多条结果
    for(Account a:list){
        System.out.println(a.getId()+" "+a.getRealName()+" "+a.getIdcardNo()); }
    session.close();                                }

@Test //SQL: select * from ACCOUNT_CHANG where REAL_NAME like ?
public void test2() {//按真名模糊查询
    //多个条件也可继续加 and XX=? or XX=?
    String hql="from Account where realName like ?"; //方式一
    //String hql="from Account where realName like \n"; //方式二
    Session session=HibernateUtil.getSession();      Query
query=session.createQuery(hql);
    query.setString(0, "zhang%");//方式一：设置查询参数，从 0 开始表示第一个？
    //query.setString("n", "zhang%");//方式二：给\n赋值，就不用数问号是第几个了
    //zhang_ 表示以 zhang 开头的两个字名字
    List<Account> list=query.list();//如果查询出多条结果
}

```

```

for(Account a:list){
    System.out.println(a.getId()+" "+a.getRealName()+" "+a.getIdcardNo()); }
    session.close(); }

@Test //SQL: select ID,REAL_NAME,IDCARD_NO from ACCOUNT_CHANG
public void test3() {//查询部分字段方式一
    String hql="select id,realName,idcardNo from Account";
    Session session=HibernateUtil.getSession(); Query
query=session.createQuery(hql);
/** 部分查询，默认采用 Object[] 封装一行记录的字段值，数组的大小、顺序和写的 HQL
属性的个数、顺序一致！注意事项：只要是单独列出的属性，则返回的就是 Object 数组！ */
*/
    List<Object[]> list=query.list();
    for(Object[] objs:list){//若写成原实体 Account 则报错类型转换异常
        System.out.println(objs[0]+" "+objs[1]+" "+objs[2]); }
    session.close(); }

@Test //SQL: select ID,REAL_NAME,IDCARD_NO from ACCOUNT_CHANG
public void test4() {//查询部分字段方式二
    String hql="select new Account(id,realName,idcardNo) from Account";
    /** 为了支持它，需要在 Account 实体中加构造方法，无参的也加上（否则影响其他地方的使用）。或者写个新的实体类也可以。 */
    Session session=HibernateUtil.getSession(); Query
query=session.createQuery(hql);
//部分查询，采用指定的 Account 的构造方法封装一行记录的字段值
List<Account> list=query.list();
for(Account a:list){//注意：显示其他属性将会是初始值
    System.out.println(a.getId()+" "+a.getRealName()+" "+a.getIdcardNo()); }
    session.close(); }
}

```

step3: Account 实体中加构造方法和无参构造方法

```

public Account(){}
public Account(Integer id,String realName,String idcardNo){
    this.id=id; this.realName=realName; this.idcardNo=idcardNo; }
}

```

step4: 在 Account.hbm.xml 中添加 HQL 语句，step5 中会用到

```

<class name="org.tarena.entity.Account" table="ACCOUNT_CHANG" ></class>
<!-- 和 class 是平级的！一个 query 标签写一个 HQL 语句 -->
<query name="findAll"><!-- 起个名字 -->
    <![CDATA[from Account]]><!-- 怕特殊符号影响语句，放 CDATA 段中作为纯文本
-->
</query>

```

step5: 其他测试

```

@Test
public void test5() {//将 HQL 定义到 hbm.xml 中
    Session session=HibernateUtil.getSession();
    //session.getNamedQuery() 方法，会去 hbm.xml 中找 HQL 语句
    Query query=session.getNamedQuery("findAll");
}

```

```

List<Account> list=query.list(); //如果查询出多条结果
for(Account a:list){
    System.out.println(a.getId()+" "+a.getRealName()+" "+a.getIdcardNo()); }
session.close();
}

@Test
public void test60{//分页查询
String hql="from Account"; Session session=HibernateUtil.getSession();
Query query=session.createQuery(hql);
//设置分页参数，就会按分页形式查询
query.setFirstResult(0);
/** 设置抓取记录的起点（每页的第一条记录），0 代表第一条“记录”，不是页数 */
query.setMaxResults(5); //设置最大抓取数量
List<Account> list=query.list(); //如果查询出多条结果
for(Account a:list){
    System.out.println(a.getId()+" "+a.getRealName()+" "+a.getIdcardNo()); }
session.close();
}

@Test
/** SQL: select s.ID,s.OS_USERNAME,s.UNIX_HOST,a.REAL_NAME,a.IDCARD_NO
 from SERVICE_CHANG s join ACCOUNT_CHANG a on(a.ID=s.ACCOUNT_ID) */
public void test70{//对象关联查询
String hql=""; //只要是单独列出的属性则返回的就是 object 数组！
//hql += "select s.id,s.osUsername,s.unixHost,a.realName,a.idcardNo "; //方式一 *
//hql += "from Service s join s.account a "; //不能写 on 子句，多对一
//hql += "from Account a join a.services s "; //或反过来，一对多
/** 方式二：一定要有“属性”才能“.”点出来，若属性是集合、数组，则它们里面
没有其他属性，不能“.”出来 */
hql += "select s.id,s.osUsername,s.unixHost, s.account.realName,s.account.idcardNo ";
hql += "from Service s";
//因为不能写 on 子句，再加上 Service 中有 account 属性，所以关联属性写 s.account
Session session=HibernateUtil.getSession(); Query
query=session.createQuery(hql);
List<Object[]> list=query.list();
for(Object[] objs:list){ System.out.println(objs[0]+" "+objs[1]+" "+objs[2]+" "
+objs[3]+" "+objs[4]); }
session.close();
}

```

2) 案例 2：联合主键的情况

step1：创建 PERSON 表

```

CREATE TABLE PERSON(
    FIRST_NAME VARCHAR2(20), LAST_NAME VARCHAR2(20), AGE NUMBER );
ALTER TABLE PERSON ADD CONSTRAINT PERSON_KEY
    PRIMARY
KEY(FIRST_NAME,LAST_NAME);

```

step2：1) 创建 Person 实体， Hibernate 要求联合主键要再封装

```

    //private String firstName;  private String lastName;
    private PersonKey id;//主属性
    private Integer age;      .....get/set 方法

```

2) 创建 PersonKey 实体, 用作 Person 实体的联合主键

```

/** 必须实现 Serializable 否则 load、get 方法不能调用了, 因为它们的第二个参数是
Serializable 类型 */
public class PersonKey implements Serializable {
    private String firstName;  private String lastName;      .....get/set 方法
}

```

step3: 添加 Person.hbm.xml 映射文件

```

<class name="org.tarena.entity.Person" table="PERSON">
    <!-- 联合主键 -->
    <composite-id name="id" class="org.tarena.entity.PersonKey">
        <!-- 主键自动生成这里就不适合了, 要通过程序操作 -->
        <key-property name="firstName" type="string" column="FIRST_NAME">
        </key-property>
        <key-property name="lastName" type="string" column="LAST_NAME">
        </key-property>
    </composite-id>
    <property name="age" type="integer" column="AGE"></property>
</class>

```

7.5 Criteria 查询

1) 基本使用方式:

```
Criteria c=session.createCriteria(实体类.class);      List list=c.list();
```

◆ 注意事项: 分组、太复杂的语句用不了。

2) 案例: 创建 TestCriteria 类, 用于测试 Criteria 查询 (借助以前的实体和映射)

```

@Test
public void test10{//没有任何子句      Session session=HibernateUtil.getSession();
    Criteria c=session.createCriteria(Account.class);      List<Account> list=c.list();
    for(Account a:list){
        System.out.println(a.getId()+" "+a.getRealName()+" "+a.getIdcardNo()); }
    session.close(); }
}

@Test
public void test20{//模糊查询      Session session=HibernateUtil.getSession();
    Criteria c=session.createCriteria(Account.class);
    //Criteria c1=session.createCriteria(Service.class);//可以有第二个表
    //追加条件, 都被封装在了 Restrictions 中
    c.add(Restrictions.like("realName", "zhang%"));      List<Account> list=c.list();
    for(Account a:list){
        System.out.println(a.getId()+" "+a.getRealName()+" "+a.getIdcardNo()); }
    session.close(); }

@Test
public void test30{//追加两个条件
    Session session=HibernateUtil.getSession();
}

```

```

Criteria c=session.createCriteria(Account.class);
//c.add(Restrictions.like("realName", "zhang%"));
c.add//当条件多时，就比较麻烦了
    Restrictions.and(  Restrictions.like("realName", "zhang%"),
                      Restrictions.eq("idcardNo", "410381194302256528") );
c.addOrder(Order.desc("id"));//追加排序      List<Account> list=c.list();
for(Account a:list){
    System.out.println(a.getId()+" "+a.getRealName()+" "+a.getIdcardNo()); }
session.close(); }
```

7.6 Native SQL 原生 SQL 查询

1) 基本使用:

```

SQLQuery query=session.createSQLQuery(sql);
List<Object[]> list=query.list();//默认封装成 Object 数组
```

◆ 注意事项：一些特殊的函数 Hibernate 无法执行需使用原生 SQL，极其复杂的操作也可用原生 SQL。

2) 案例：创建 TestSQLQuery 类，用于测试原生 SQL 查询（同样借助以前的实体和映射）

```

@Test
public void test1()//没有用到映射文件
{
    String sql="select * from ACCOUNT_CHANG";
    Session session=HibernateUtil.getSession();
    SQLQuery query=session.createSQLQuery(sql);
    query.setFirstResult(0);//分页查询      query.setMaxResults(5);
    List<Object[]> list=query.list();
    for(Object[] objs:list){  System.out.println(objs[0]+" "+objs[1]+" "+objs[2]); }
    session.close(); }
```



```

@Test
public void test2()//改变 test1 中的封装类型
{
    String sql="select * from ACCOUNT_CHANG";
    Session session=HibernateUtil.getSession();
    SQLQuery query=session.createSQLQuery(sql);
    /** 改变封装类型，利用该类型映射（映射描述将起作用），封装一条记录（全字段的，部分字段不行） */
    query.addEntity(Account.class);      List<Account> list=query.list();
    for(Account a:list){  System.out.println(a.getId()+" "+a.getRealName()); }
    session.close(); }
```

一百三十二、Hibernate 高级特性

LICHOO

8.1 二级缓存

二级缓存是 SessionFactory 级别的。由 SessionFactory 对象负责管理。通过 Factory 创建的 Session 都可以访问二级缓存。

二级缓存默认是关闭的。如果二级缓存打开，则先去一级缓存找对象，找不到则去二级缓存，还找不到则访问数据库。

◆ 注意事项：

- ❖ 一级缓存和二级缓存都是存“单个对象”的！不能存集合、数组！
- ❖ Hibernate 本身没有提供组件，需要第三方提供的组件。有很多第三方组件，这里用 ehcache-1.2.3.jar。

8.2 二级缓存开启方法及测试

step1：引入 ehcache-1.2.3.jar，在 src 下添加 ehcache.xml 配置文件

ehcache.xml 配置文件说明：

```
<diskStore path="java.io.tmpdir"/><!-- 磁盘存储路径 -->
<defaultCache           <!--默认参数设置-->
    maxElementsInMemory="2000" <!--存储的最大对象个数-->
    eternal="false"          <!--缓存对象的有效期，true 为永久存在-->
    timeToIdleSeconds="20"   <!--空闲时间：某个对象空闲超过 20 秒则清出二级缓存-->
    timeToLiveSeconds="120"  <!--某个对象生存了 120 秒，则自动清出二级缓存-->
    overflowToDisk="true"    <!--当存储个数超出设置的最大值时，把超出对象存到磁盘-->
/>
```

step2：在 hibernate.cfg.xml 中添加开启配置参数，指定缓存类型

```
<!-- 开启二级缓存 -->
<property name="hibernate.cache.use_second_level_cache">true</property>
<!-- 指定二级缓存组件类型，类似于 JDBC 中不同的数据库驱动类型 -->
<property name="hibernate.cache.provider_class">
    net.sf.ehcache.hibernate.EhCacheProvider
</property><!--在 ehcache-1.2.3.jar 中 net.sf.ehcache.hibernate 下找-->
```

step3：在需要缓存的实体类 Account.hbm.xml 中，添加

```
<cache usage="read-only" region="采用 ehcache.xml 中哪组参数缓存"/>
<class name="org.tarena.entity.Account" table="ACCOUNT_CHANG">
    <!-- read-only：只读。read-write：读写。region：指明用哪组参数缓存 -->
    <cache usage="read-only" region="sampleCache1"/>
    <id name="id" type="integer" column="ID"></id>
    .....其他部分略
```

step4：创建 TestSecondCache 类，用于测试二级缓存

```
@Test
public void test10{//查询一次，一级缓存的作用
    Session session=HibernateUtil.getSession();}
```

```

Account account=(Account)session.load(Account.class, 1010);
System.out.println(account.getRealName()); System.out.println(account.getIdcardNo());
Account account1=(Account)session.get(Account.class, 1010);
System.out.println(account1.getEmail()); session.close(); }

@Test
public void test2()//若没有开启二级缓存，则查询两次，因为是不同的 Session 对象
    test1(); System.out.println("-----"); test1(); //开启二级缓存后，只查询一次 }

```

8.3 二级缓存管理方法

- 1) SessionFactory.evict(class); //清除某一类型的对象
- 2) SessionFactory.evict(class,id); //清除某一个对象
- 3) SessionFactory.evict(list); //清除指定的集合

8.4 二级缓存的使用环境

- 1) 共享数据，数据量不大。
- 2) 该数据更新频率低（一更新就涉及到同步，就要有性能开销了）。

8.5 查询缓存

一级和二级缓存只能缓存单个对象。对于其他类型的数据，例如：一组字段或一组对象集合、数组都不能缓存。这种特殊结果，可以采用查询缓存存储。查询缓存默认是关闭的！

8.6 查询缓存开启方法及测试

step1：开启查询对象的二级缓存（因为也是跨 Session 的）。

step2：在 hibernate.cfg.xml 中设置开启查询缓存参数。

```

<!-- 开启查询缓存 -->
<property name="hibernate.cache.use_query_cache">true</property>

```

step3：创建 TestQueryCache 类，用于测试查询缓存，并在执行 query.list() 方法前，设置 query.setCacheable(true);

```

@Test
public void test1()//当传入的参数相同时，只执行一次查询。不同时执行两次查询
    show("zhang%"); System.out.println("-----"); show("zhang%");//一次查询 }
private void show(String name){
    String hql="from Account where realName like ?";
    /** 看 SQL 不看 HQL，SQL 相同则从查询缓存中取，不一样则访问数据库查询 */
    Session session=HibernateUtil.getSession(); Query
query=session.createQuery(hql);
    query.setString(0, name);
    /** 采用查询缓存机制进行查询（在开启二级缓存的基础上）！去缓存查找，执行过此次 SQL，将结果集返回。未执行过，去数据库查询，并将 SQL 和结果存入缓存 */
    query.setCacheable(true); List<Account> list=query.list();//执行查询
    for(Account a:list){ System.out.println(a.getId()+" "+a.getRealName()); }
    session.close(); }

```

8.7 查询缓存的使用环境

- 1) 共享数据的结果集，结果集数据量不应太大（几十到几百条即可）。
- 2) 查询的结果集数据变化非常小（最好不要变化，几天变一次还可接受，先清原来的缓存再重新载入）。

一百三十三、Hibernate 锁机制

LICHOO

Hibernate 提供了乐观锁和悲观锁机制，主要用于解决事务并发问题。

9.1 悲观锁

Hibernate 认为任何操作都可能发生并发，因此在第一个线程查询数据时，就把该条记录锁住。此时其他线程对该记录不能做任何操作（即增删改查四种操作都不能）。必须等当前线程事务结束才可以进行操作。

9.2 悲观锁的实现原理

Hibernate 悲观锁机制实际上是采用数据库的锁机制实现。

数据库中 SQL 语句最后加 for update 则把记录直接锁死，其他用户增删改查都不行，只能等待：select * from TRAIN where id=1 for update;

Hibernate 中 load 重载方法：session.load(Train.class,1,LockMode.UPDATE);只能等待当前用户提交或回滚，若等待超时则报异常！

悲观锁的缺点：处理效率很低。

9.3 悲观锁使用步骤及测试

step1：创建表 TRAIN（火车）

```
CREATE TABLE TRAIN(
    ID NUMBER PRIMARY KEY,    T_START VARCHAR2(20),
    T_END VARCHAR2(20),        T_TICKET NUMBER
);
INSERT INTO TRAIN VALUES(1,'beijing','shanghai',100);
```

step2：创建 Train 实体类

```
private int id;    private String start;    private String end;    private int ticket; ....get/set
```

step3：添加 Train.hbm.xml 映射文件

```
<class name="org.tarena.entity.Train" table="TRAIN" >
    <id name="id" type="integer" column="ID"></id>
    <property name="start" type="string" column="T_START"></property>
    <property name="end" type="string" column="T_END"></property>
    <property name="ticket" type="integer" column="T_TICKET"></property>
</class>
```

step4：创建 ThreadClient 类，用于模拟一个用户操作

```
public class ThreadClient extends Thread {//继承 Thread
    public void run()//模拟购票操作
        Session session=HibernateUtil.getSession();
        Transaction tx=session.beginTransaction();
        //Train train=(Train)session.load(Train.class, 1);//不加锁时，查询出火车信息
        Train train=(Train)session.load(Train.class,1,LockMode.UPGRADE);//设置悲观锁
        if(train.getTicket()>=1){//判断剩余票数>=购买票数
            try {//满足购票条件，进行购票操作
                Thread.sleep(2000);//模拟用户操作
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
}
```

```

        int ticket=train.getTicket()-1;//将票数更新
        train.setTicket(ticket);      System.out.println("购票成功! ");
    }else {  System.out.println("票数不足, 购买失败! ");
    tx.commit();//持久对象, 提交就自动同步, 相当于执行了 update
    session.close();
}
}

```

step5：创建 TestTrain 类，用于模拟并发操作

```

public static void main(String[] args) {
    ThreadClient c1=new ThreadClient();      c1.start();
    ThreadClient c2=new ThreadClient();      c2.start();
    /* 若 ThreadClient 类为无锁查询, 则显示 2 个购买成功, 但数据库却减少 1 张票。若
    ThreadClient 类加了悲观锁, 则显示 2 个购买成功, 数据库减少 2 张票。 */
}

```

9.4 乐观锁

认为发生并发几率非常小。相同的记录不同的用户都可以查询访问, 当多个人都要修改该记录时, 只有第一个提交的用户成功, 其他的会抛出异常, 提示失败!

9.5 乐观锁的实现原理

乐观锁机制是借助于一个“版本”字段实现, 当第一个更新用户提交成功后, Hibernate 会自动将该“版本”字段值+1, 当其他用户提交, 如果版本字段小于数据库中的值, 则会抛出异常, 提示失败。如果不使用框架技术, 那么我们需要手工做对比, 使用 Hibernate 框架后, Hibernate 可以帮助我们做 version 对比的操作。

9.6 乐观锁使用步骤及测试

step1：向 TRAIN 表中添加一个 T_VERSION 版本字段

```
ALTER TABLE TRAIN ADD T_VERSION NUMBER;
```

step2：在 Train 实体类中添加 version 属性, 以及对应的 get/set 方法

```
private int version;//版本属性
```

step3：在 Train.hbm.xml 中定义版本字段映射

简单说明：

```

<class name="org.tarena.entity.Train" table="TRAIN" optimistic-lock="version">
    <!-- 采用“版本”机制, 不写也可, 默认是 version -->
    <id name="" type="" column=""></id>
    <version name="属性名" type="类型" column="字段名"></version>

```

具体实现：

```

<class name="org.tarena.entity.Train" table="TRAIN" optimistic-lock="version">
    <id name="id" type="integer" column="ID"></id>
    <!-- 定义乐观锁的版本字段, 注意顺序 -->
    <version name="version" type="integer" column="T_VERSION"></version>
    <property name="start" type="string" column="T_START"></property>

```

step4：将 9.3 节 step4 中的 ThreadClient 类, 改为不加锁的 load 方法

```
Train train=(Train)session.load(Train.class, 1);
```

step5：再次执行 9.3 节 step5 中的 TestTrain 类

执行结果：先提交的事务执行成功，后提交的事务执行失败，报错信息为：

StaleObjectStateException: Row was updated or deleted by another transaction

一百三十四、其他注意事项

LICHOO

10.1 源码服务器管理工具

今后工作中会使用到的源码服务器（或叫版本服务器）管理工具：

- 1) CVS 常用 乐观锁机制
- 2) SVN 常用 乐观锁机制
- 3) VSS 用的少 悲观锁机制

10.2 利用 MyEclipse 根据数据表自动生成实体类、hbm.xml

step1：进入 DB Browser，建立一个与数据库的连接



step2：新建工程，右键工程-->MyEclipse-->Add Hibernate Capabilities

step3：弹出添加 Hibernate 开发环境界面：

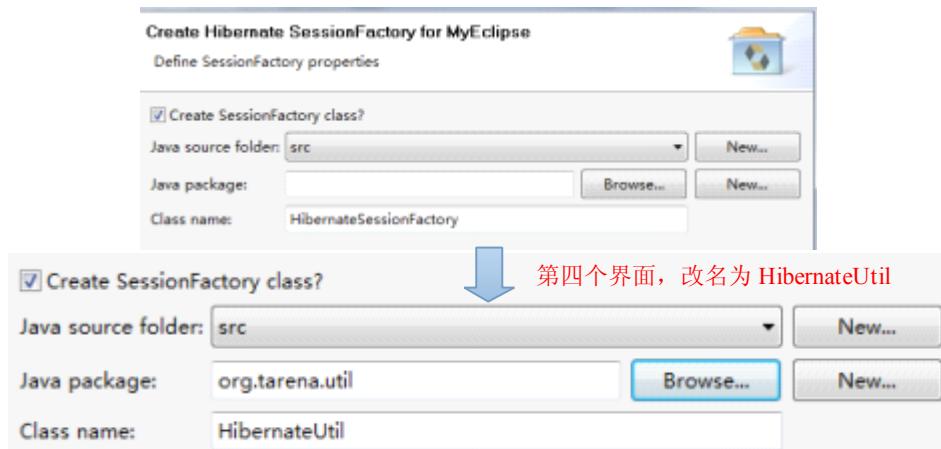
第一个界面选择 Hibernate 版本，然后点 Next；

- ◆ 注意事项：第一个界面中 Select the libraries to add to the buildpath 是选择要导入的 Hibernate 的 Jar 包，有两种选择：MyEclipse 自带的“MyEclipse Libraries”，还有自己导入的“User Libraries”，如果已经手动将所有 Jar 包拷贝到了 lib 下，不用再导入了。那么我们都不选，去掉“MyEclipse Libraries”前面的 checkbox 的对勾。

第二个界面添加 hibernate.cfg.xml 主配置文件（可默认），然后点 Next；

第三个界面在 DB Driver 下拉菜单中选择第一步建立的连接，然后点 Next；

第四个界面创建 HibernateSessionFactory 类，其实就是我们写的 HibernateUtil，然后点击 Finish。



- ◆ 注意事项：主配置文件中：

- ❖ 对于 MySql 连接，连接字符串中如若有&符号，需要写成&
- ❖ <property name="myeclipse.connection.profile">

oracle.jdbc.driver.OracleDriver</property>为连接名，可以删掉。

step4：在工程中，新建一个 package，用于存放实体和 hbm.xml

step5：进入 DB Browser 选中需要生成的数据表，右键选择 Hibernate Reverse Engineering...

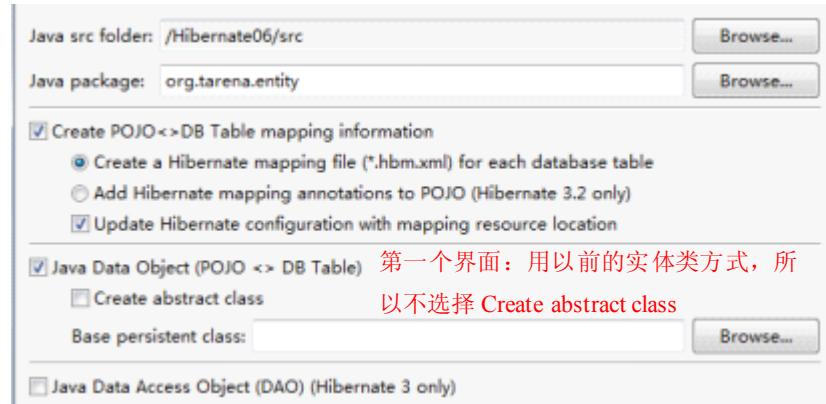
第一个界面选择生成 hbm.xml 和 POJO（实体类），及其存放路径，然后点 Next；

第二个界面选择 Type Mapping 为 Hibernat types（其他设置可默认），然后点 Next；

第三个界面选择数据表，可以设定映射的 POJO 类名（包名.类名）和主键生成方式，选择字段可以设置属性名和映射类型（要改，否则不符合 Java 命名规范），最后点击 Finish。

- ◆ 注意事项：第一个界面有个 Create abstract class，创建抽象类。意思为：

```
public class Foo extends AbstractFoo implements java.io.Serializable {
    //里面为构造方法，没有属性和 getter/setter 方法
}
public abstract class AbstractFoo implements java.io.Serializable{
    //里面为属性和 getter/setter 方法
}
```



Generate artifacts for the following tables:

- T_FOO
 - T_ID
 - T_NAME
 - T_SALARY
 - T_HIRE_DATE
 - T_MARRY
 - T_LAST_LOGIN_TIME

Table details

Name: T_FOO
Catalog:
Schema: SYSTEM
Class name: org.tarena.entity.Foo
Id Generator: sequence

Generate artifacts for the following tables:

- T_FOO
 - T_ID
 - T_NAME
 - T_SALARY
 - T_HIRE_DATE
 - T_MARRY
 - T_LAST_LOGIN_TIME

Column details

Name: T_HIRE_DATE
 Exclude column from reverse engineering
JDBC type:
Property name: hireDate
Hibernate type: date

step6：完成后，实体类、hbm.xml 和 hibernate.cfg.xml 都已生成、修改好了。

- ◆ 注意事项：

- ❖ Hibernate 生成的映射文件会有一些问题，最好再修改一下。
- ❖ 可以按“Ctrl”键选择多个表，之后步骤相同，可以同时创建多个表的 POJO 类和映射文件。

10.3 根据实体类和 hbm.xml 生成数据表

利用 Hibernate 框架自带的 hbm2ddl 工具，根据实体类和 hbm.xml 生成数据表。

step1：在 hibernate.cfg.xml 中添加开启工具的设置

```
<!-- 根据实体类和 hbm.xml 生成数据表 -->
<property name="hbm2ddl.auto">update</property>
```

step2：在执行添加、查询等操作时，会自动创建数据表，然后再执行操作。

10.4 Hibernate 中分页查询使用 join fetch 的缺点

使用 Hibernate 分页查询时，若语句中使用了 join fetch 语句，则由“假分页”机制实现。

实际采用可滚动 ResultSet 游标实现的数据抓取。最后的显示结果一样，但是 SQL 语句不同。

假分页 SQL 语句：select * from bill，返回大量数据，然后利用游标 result.absolute(begin); 即跳着查询，而 result.next(); 为下一条记录。

真分页 SQL 语句： MySql: select * from bill limit ?,?

Oracle: select ... (select ... rownum<?) where rownum>?

那么假分页的结果就是，一次性把数据全部取出来放在缓存中，比较适合小数据量，如果数据量大，对内存压力比较大。

所以，建议 join fetch 语句不要和分页查询一起用！

“假分页”详情可看 JDBC 笔记 9.5 节。

10.5 Hibernate 的子查询映射

1) 在映射文件中使用如下格式：

```
<property name="属性名" type="映射类型" formula="(子查询语句)"></property>
```

在对当前对象查询时，Hibernate 会将子查询执行结果给指定的属性赋值。

2) formula 使用注意事项：

①必须将子查询语句用括号括起来。

②子查询语句必须是 SQL 语句。

③子查询中使用的表必须用别名(在查询条件中没有别名的字段都默认是当前映射实体类的字段)。

④当使用了子查询映射后，如 HQL 语句：from Host，默认查询所有字段和子查询结果。如果只需要部分字段值，不需要子查询结果，可以采用 select 单独指定需要的字段属性。

3) 案例：对于 NetCTOSS 项目中的每台服务器资费使用量的查询

step1：在实体类 Host 中添加三个属性，并设置 set/get 方法

```
private int id;      private String name;      private String location;  
private Integer c1;//追加属性，用于存储包月使用数量  
private Integer c2;//追加属性，用于存储套餐使用数量  
private Integer c3;//追加属性，用于存储计时使用数量
```

step2：在映射文件 Host.hbm.xml 中添加这三个属性的描述（采用子查询形式）

```
<!-- 采用子查询映射将包月使用量结果给 c1，缺点：只要写 from HOST_CHANG 就会  
嵌入子查询，默认查询所有字段和子查询结果，若只需要部分字段值，不要子查询结果，可  
以采用：select 部分字段名 from HOST_CHANG -->
```

```
<property name="c1" type="integer" formula="(select count(*) from SERVICE_CHANG  
s,COST_CHANG c where s.COST_ID=c.ID and s.UNIX_HOST=ID and c.COST_TYPE='1')">  
</property><!-- s.UNIX_HOST 不能写死了，把 ID 赋给它，ID 为 HOST 中的字段 -->  
<!-- 采用子查询映射将套餐使用量结果给 c2 -->  
<property name="c2" type="integer" formula="(select count(*) from SERVICE_CHANG  
s,COST_CHANG c where s.COST_ID=c.ID and s.UNIX_HOST=ID and c.COST_TYPE='2')">  
</property>  
<!-- 采用子查询映射将计时使用量结果给 c3 -->  
<property name="c3" type="integer" formula="(select count(*) from SERVICE_CHANG  
s,COST_CHANG c where s.COST_ID=c.ID and s.UNIX_HOST=ID and c.COST_TYPE='3')">  
</property>
```

15 Spring 学习笔记

LICHO0

一百三十五、Spring 概述

我们学习 Spring 框架的最终目的是用它整合 Struts2、Hibernate 框架（SSH）。

1.1 Spring 框架的作用

Spring 框架主要负责技术整合（可以整合很多技术），该框架提供 IoC 和 AOP 机制，基于这些特性整合，可以降低系统组件之间的耦合度，便于系统组件的维护、扩展和替换。

1.2 Spring 框架的优点

其实与 Spring 框架的作用相同：

在 SSH 中，主要是利用 Spring 容器管理我们程序中的 Action、DAO 等组件，通过容器的 IoC 机制，可以降低 Action、DAO 之间的耦合度（关联度），利用 AOP 进行事务管理等共通部分的处理。

在 SSH 中，Struts2 主要是利用它的控制器，而不是标签、表达式；Hibernate 主要利用它的数据库访问；Spring 主要是利用它的整合。

1.3 Spring 框架的容器

Spring 框架的核心是提供了一个容器（是我们抽象出来的，代指后面的类型）。该容器类型是 BeanFactory 或 ApplicationContext（建议用这个类型，它是 BeanFactory 的子类，功能更多）。

该容器具有以下功能：

- 1) 容器可以创建和销毁组件对象，等价于原来“工厂”类的作用。
- 2) 容器可以采用不同的模式创建对象，如单例模式创建对象。
- 3) 容器具有 IoC 机制实现。
- 4) 容器具有 AOP 机制实现。

一百三十六、Spring 容器的基本应用

LICHOO

2.1 如何将一个 Bean 组件交给 Spring 容器

- 1) Bean 组件其实就是一个普通的 Java 类！
- 2) 方法：在 applicationContext.xml 中添加以下定义，见 2.6 案例中 step4。

```
<bean id="标识符" class="Bean 组件类型"></bean>
```

2.2 如何获取 Spring 容器对象和 Bean 对象

- 1) 实例化容器：

```
ApplicationContext ac=new ClassPathXmlApplicationContext("/applicationContext.xml");  
//FileSystemXmlApplicationContext(""); //去指定的磁盘目录找，上面的为去 Class 路径找
```

- 2) 利用 getBean("标识符")方法获取容器中的 Bean 对象。见 2.6 案例中 step5。

2.3 如何控制对象创建的模式

Spring 支持 singleton（单例）和 prototype（原型，非单例）两种模式。

默认是 singleton 模式，可以通过<bean>的 scope 属性修改为 prototype 模式。以后在 Web 程序中，通过扩展可以使用 request、session 等值。见 2.6 案例中 step4、step7。

例如：<bean id="标识符" scope="prototype" class="Bean 组件类型"></bean>

- ◆ 注意事项：对于 NetCTOSS 项目，一个请求创建一个 Action，所以用 Spring 时必须指明 prototype，否则默认使用 singleton 会出问题。而 DAO 则可用 singleton 模式。

2.4 Bean 对象创建的时机

- 1) singleton 模式的 Bean 组件是在容器实例化时创建。
- 2) prototype 模式是在调用 getBean()方法时创建。
- 3) singleton 模式可以使用<bean>元素的 lazy-init="true"属性将对象的创建时机推迟到调用 getBean()方法。也可以在<beans>（根元素）中使用 default-lazy-init="false"推迟所有单例 Bean 组件的创建时机。见 2.6 案例中 step3、step4。

例如：<bean id="标识符" scope="singleton" lazy-init="true" class="Bean 组件类型"></bean>
<beans default-lazy-init="false"></beans>

2.5 为 Bean 对象执行初始化和销毁方法

- 1) 初始化：
 - ① 可以利用<bean>元素的 init-method="方法名"属性指定初始化方法。
② 指定的初始化方法是在构造方法调用后自动执行。若非单例模式，则每创建一个对象，则执行一次初始化方法（单例、非单例模式都可）。见 2.6 案例中 step3、step4。
 - ◆ 注意事项：
 - ❖ 初始化的三种方式：写构造方法中；或写{}中（代码块）；Spring 框架中<bean>元素写 init-method="方法名"属性。
 - ❖ 初始化不能用 static {}，它是类加载调用，比创建对象要早。
- 2) 销毁：
 - ① 可以利用<bean>元素的 destroy-method="方法名"属性执行销毁方法。
② 指定的销毁方法是在容器关闭时触发，而且只适用于 singleton 模式的组件（只能为单例模式）。见 2.6 案例中 step3、step4、step6。

2.6 案例：Spring 框架的使用以及 2.1 节-2.5 节整合测试

step1: 导入 Spring 开发包: spring.jar、commons-logging.jar 和配置文件: applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:jee="http://www.springframework.org/schema/jee"
       xsi:schemaLocation="http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
                           http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-2.5.xsd
                           http://www.springframework.org/schema/jee
                           http://www.springframework.org/schema/jee/spring-jee-2.5.xsd">
</beans>
```

step2: 在 org.tarena.dao 包下, 创建接口 CostDAO, 添加两个方法

```
public void delete();      public void save();
```

step3: 在 org.tarena.dao 包下, 创建 JdbcCostDAO 类, 并实现 CostDAO 接口

```
public JdbcCostDAO(){    System.out.println("创建 CostDAO 对象");    }
public void myinit(){    System.out.println("初始化 CostDAO 对象");    }
public void mydestroy(){    System.out.println("销毁 CostDAO 对象");    }
public void delete() {    System.out.println("利用 JDBC 技术实现删除资费记录");    }
public void save() {    System.out.println("利用 JDBC 技术实现保存资费记录");    }
```

step4: 在 applicationContext.xml 配置文件中, 将 Bean 组件 (Java 类) 交给 Spring 容器

```
<bean id="jdbcCostDAO" scope="singleton" lazy-init="true" init-method="myinit"
      destroy-method="mydestroy" class="org.tarena.dao.JdbcCostDAO">
</bean>
```

step5: 在 org.tarena.test 包下, 创建 TestApplicationContext 类, 获取 Spring 容器对象, 并测试

```
@Test
public void test10{    String conf="/applicationContext.xml";
                      ApplicationContext ac=new ClassPathXmlApplicationContext(conf);//实例化容器
                      CostDAO costDAO=(CostDAO)ac.getBean("jdbcCostDAO");//获取 Bean 对象
                      costDAO.save();    costDAO.delete();    }
```

step6: 在 TestApplicationContext 类中添加方法, 测试销毁对象

```
@Test
/**关闭容器才会触发销毁, 但关闭容器方法封装在 AbstractApplicationContext 类中 */
```

```
LICHOO  
public void test20{      String conf="/applicationContext.xml";  
    ApplicationContext ac=new ClassPathXmlApplicationContext(conf);  
    AbstractApplicationContext ac=new ClassPathXmlApplicationContext(conf);  
    CostDAO costDAO=(CostDAO)ac.getBean("jdbcCostDAO");      ac.close();    }
```

step7: 在 TestApplicationContext 类中添加方法，测试单例

```
@Test  
public void test3(){      String conf="/applicationContext.xml";  
    ApplicationContext ac= new ClassPathXmlApplicationContext(conf);  
    CostDAO costDAO1=(CostDAO)ac.getBean("jdbcCostDAO");  
    CostDAO costDAO2=(CostDAO)ac.getBean("jdbcCostDAO");  
    System.out.println(costDAO1==costDAO2);//true, 所以 Spring 默认为单例模式 }
```

一百三十七、Spring 框架 IoC 特性

LICHOO

3.1 IoC 概念

- 1) Inverse of Controller 被称为控制反转或反向控制，其实真正体现的是“控制转移”。
- 2) 所谓的控制指的是负责对象关系的指定、对象创建、初始化和销毁等逻辑。
- 3) IoC 指的是将控制逻辑交给第三方框架或容器负责（即把 Action 中的控制逻辑提出来，交给第三方负责），当两个组件关系发生改变时，只需要修改框架或容器的配置即可。
- 4) IoC 主要解决的是两个组件对象调用问题，可以以低耦合方式建立使用关系。

3.2 DI 概念

- 1) Dependency Injection 依赖注入。
- 2) Spring 框架采用 DI 技术实现了 IoC 控制思想。
- 3) Spring 提供了两种形式的注入方法：

①setter 方式注入（常用）：依靠 set 方法，将组件对象传入（可注入多个对象）。

A.首先添加属性变量和 set 方法。

B.在该组件的<bean>定义中采用下面的描述方式：

```
<property name="属性名" ref="要注入的 Bean 对象的 id 值"></property>
```

◆ 注意事项：例如 CostAction 中有 costDAO 属性，而它的标准 set 方法名为 setCostDAO，那么配置文件中的 name 就应该写 costDAO（去掉 set，首字母小写）。如果 set 方法名为 setCost，那么 name 就应该写 cost（去掉 set 首字母小写）！确切的说，name 不是看定义的属性名，而是 set 方法名。

②构造方式注入（用的少）：依靠构造方法，将组件对象传入。

A.在需要注入的组件中，添加带参数的构造方法。

B.在该组件的<bean>定义中，使用下面格式描述：

```
<constructor-arg index="参数索引" ref="要注入的 Bean 对象的 id 值"></constructor-arg>
```

3.3 案例：测试 IoC（set 注入）

step1：接 2.6 案例，在 org.tarena.action 包下，创建 CostAction 类，调用 save 方法

```
private CostDAO costDAO;//利用 Spring 的 IOC 机制使用 CostDAO 组件对象，set 注入
public void setCostDAO(CostDAO costDAO) {    this.costDAO = costDAO;    }
public String execute(){    System.out.println("处理资费添加操作");
    costDAO.save();//调用 CostDAO 中的 save 方法    return "success";    }
```

step2：在 applicationContext.xml 配置文件中，将 CostAction 组件交给 Spring 容器

```
<bean id="costAction" scope="prototype" class="org.tarena.action.CostAction">
    <!-- 利用 setCostDAO 方法接收 jdbcCostDAO 对象 -->
    <property name="costDAO" ref="jdbcCostDAO"></property>
    <!-- name：与 CostAction 中对应的 set 方法匹配的名。ref：指明哪个对象 -->
</bean><!--此处用到了 2.6 案例中 step3 描述的组件 JdbcCostDAO-->
```

step3：在 org.tarena.test 包下，创建 TestIoc 类，用于测试 IOC 机制

```
@Test //测试 set 注入
public void test10{    String conf="/applicationContext.xml";
    ApplicationContext ac=new ClassPathXmlApplicationContext(conf);}
```

```
    CostAction action=(CostAction)ac.getBean("costAction");//获得 CostAction 类的对象  
    action.execute();  
}
```

step4：测试结果为：

创建 CostDAO 对象	初始化 CostDAO 对象	处理资费添加操作
利用 JDBC 技术实现保存资费记录。		

3.4 案例：测试 IoC（构造注入）

step1：接 3.3 案例，在 org.tarena.action 包下，创建 DeleteAction 类，调用 delete 方法

```
private CostDAO costDAO;  
public DeleteAction(CostDAO costDAO){ this.costDAO=costDAO; }//构造注入  
public String execute() { System.out.println("处理资费删除操作");  
    costDAO.delete();//调用 CostDAO 中的 delete 方法 return "success"; }
```

step2：在 applicationContext.xml 配置文件中，将 DeleteAction 组件交给 Spring 容器

```
<bean id="deleteAction" scope="prototype" class="org.tarena.action.DeleteAction">  
    <!-- 索引 0：给构造方法中第一个参数注入一个 jdbcCostDAO 对象。  
        若多个参数则重复追加 constructor-arg 元素即可 -->  
    <constructor-arg index="0" ref="jdbcCostDAO"></constructor-arg><!-- 构造注入 -->  
    </bean>
```

step3：在 TestIoc 类中添加方法，测试构造注入

```
@Test //测试构造注入  
public void test20{ String conf="/applicationContext.xml";  
    ApplicationContext ac=new ClassPathXmlApplicationContext(conf);  
    DeleteAction action=(DeleteAction)ac.getBean("deleteAction");  
    action.execute(); }
```

step4：测试结果为：

创建 CostDAO 对象	初始化 CostDAO 对象	处理资费删除操作
利用 JDBC 技术实现删除资费记录。		

3.5 案例：不用 JDBC 访问数据库，而是采用 Hibernate 访问

接 3.3 案例，如果不使用 JDBC 访问数据库，而是采用 Hibernate 访问，则替换组件过程为：

step1：创建 HibernateCostDAO 类，并实现 CostDAO 接口

```
public void delete() { System.out.println("利用 Hibernate 技术实现删除资费记录"); }  
public void save() { System.out.println("利用 Hibernate 技术实现保存资费记录"); }
```

step2：在 applicationContext.xml 配置文件中，将 HibernateCostDAO 组件交给 Spring 容器

```
<bean id="hibernateCostDAO" class="org.tarena.dao.HibernateCostDAO"></bean>
```

step3：修改 3.3 案例中 step2 中 CostAction 组件的描述

```
<bean id="costAction" scope="prototype" class="org.tarena.action.CostAction">  
    <!-- 修改 ref 属性的指引 -->  
    <property name="costDAO" ref="hibernateCostDAO"></property>  
    <!-- name：与 CostAction 中添加的属性名相同。ref：指明哪个对象 -->  
    </bean>
```

step4：再次执行 3.3 案例 step3 中 test1 方法，测试结果为：

处理资费添加操作	利用 Hibernate 技术实现保存资费记录
----------	-------------------------

一百三十八、Spring 中各种类型的数据注入

Spring 可以为对象注入以下类型的数据。

4.1 Bean 对象注入

```
<property name="属性名" ref="要注入的 Bean 对象的 id 值"></property>
```

4.2 基本数据的注入

1) 字符串、数字

```
<property name="属性名" value="要注入的值"></property>
```

4.3 集合的注入

1) List、Set

```
<property name="集合属性名">
    <list><!-- 普通值用<value>标签, 对象用<bean>标签 -->
        <value>集合中的值 1</value><value>集合中的值 2</value> .....
    </list>
</property>
```

2) Map

```
<property name="集合属性名">
    <map><!-- 普通值用<value>标签, 对象用<bean>标签 -->
        <entry key="键 1" value="值 1"></entry>
        <entry key="键 2" value="值 2"></entry> .....
    </map>
</property>
```

3) Properties

```
<property name="集合属性名">
    <props>
        <prop key="键 1">值 1</prop>
        <prop key="键 2">值 2</prop> .....
    </props>
</property>
```

4) 特殊用法: set 方法接收字符串, 内部进行处理 (如分割), 再存入集合属性

```
<property name="集合属性名" value="字符串"></property>
```

4.4 案例：各类数据注入

step1: 对象注入参考 3.3、3.4 案例

step2: 创建 MessageBean 类, 并定义不同类型的数据以及对应的 set 方法

```
private String username;//用户名      private String fileDir;//上传路径
private List<String> hbms;           private Set<String> cities;
private Map<Integer, String> books;   private Properties props;
private Set<String> types;//允许上传类型
/** 注意 set 方法的名字! 不是看属性名, 而是看 set 方法名去掉 set, 首字母大写的名 */
```

```

public void setName(String username) {    this.username = username; } //手动更改过名字
public void setDir(String fileDir) {    this.fileDir = fileDir; } //手动更改过名字
.....其他属性名字没改，其他属性代码略
public void setTypes(String str) { //特殊用法：注入一个字符串，分析之后给 set 集合赋值
    String[] arr = str.split(",");    types = new HashSet<String>();
    for (String s : arr) {    types.add(s);    }
}
public void show() {
    System.out.println("用户名：" + username);    System.out.println("上传路径：" + fileDir);
    System.out.println("--hbms 文件如下--");
    for (String s : hbms) {    System.out.println(s);    }
    System.out.println("--city 城市如下--");
    for (String c : cities) {    System.out.println(c);    }
    System.out.println("--book 图书信息--");
    Set<Entry<Integer, String>> ens = books.entrySet();
    for (Entry en : ens) {    System.out.println(en.getKey() + " " + en.getValue());    }
    System.out.println("--props 参数如下--");
    Set keys = props.keySet(); //另一种方式遍历
    for (Object key : keys) {    System.out.println(key + " +
        props.getProperty(key.toString()));    }
    System.out.println("--允许上传类型如下--"); //特殊用法
    for (String type : types) {    System.out.println(type);    }
}

```

step3: applicationContext.xml 配置文件中

```

<!-- 各种数据类型的注入 -->
<bean id="messageBean" class="org.tarena.dao.MessageBean">
    <!-- 注意名字 name 指的是 set 方法名去掉 set, 首字母大写的名, 不看属性名！ -->
    <property name="name" value="root"></property> <!-- 手动更改过 set 方法名 -->
    <property name="dir" value="D:\\images"></property> <!-- 手动更改过 set 方法名 -->
    <property name="hbms">
        <list><value>/org/tarena/entity/Cost.hbm.xml</value>
            <value>/org/tarena/entity/Admin.hbm.xml</value>
        </list></property>
    <property name="cities">
        <set><value>北京</value><value>上海</value></set></property>
    <property name="books">
        <map><entry key="1" value="Java 语言基础"></entry>
            <entry key="2" value="Java Web 入门"></entry>
        </map></property>
    <property name="props">
        <props><prop key="hibernate.show_sql">true</prop>
            <prop
key="hibernate.dialect_sql">org.hibernate.dialect.OracleDialect</prop>
        </props>
    </property>
    <!-- 特殊用法， set 方法传入字符串，内部进行处理，再存入集合 -->
    <property name="types" value="jpg,gif,jpeg"></property>

```

```
</bean>
```

step4： 创建 TestInjection 类用于测试各类数据的注入

```
@Test  
public void test1(){  
    String conf="/applicationContext.xml";  
    ApplicationContext ac=new ClassPathXmlApplicationContext(conf);  
    MessageBean bean=(MessageBean)ac.getBean("messageBean");  
    bean.show();}
```

一百三十九、AOP 概念

LICHOO

5.1 什么是 AOP

Aspect Oriented Programming，被称为面向方面编程。对单个对象（一对一）的解耦用 IOC，而当有个共通组件，它对应多个其他组件（一对多），则解耦用 AOP。如，拦截器。这也是为何在程序中大量的用 IoC，而 AOP 却用的很少，因为程序中不可能有很多的共通部分。

5.2 AOP 和 OOP 的区别

OOP 是面向对象编程，AOP 是以 OOP 为基础的。

OOP 主要关注的是对象，如何抽象和封装对象。

AOP 主要关注的是方面，方面组件可以以低耦合的方式切入到（作用到）其他某一批目标对象方法中（类似于 Struts2 中的拦截器）。

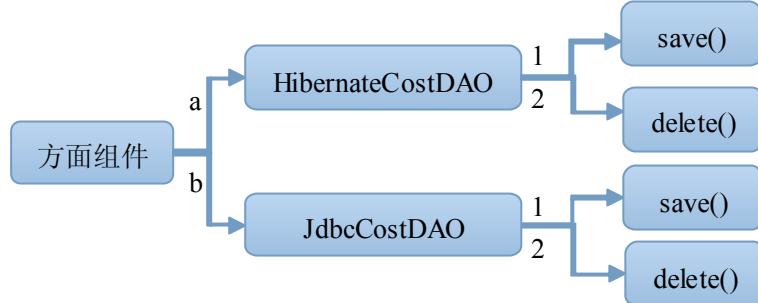
AOP 主要解决共通处理和目标组件之间解耦。

5.3 AOP 相关术语

1) 方面 (Aspect): 指的是封装了共通处理的功能组件。该组件可以作用到某一批目标组件的方法上。

2) 连接点 (JoinPoint): 指的是方面组件和具体的哪一个目标组件的方法有关系。

3) 切入点 (Pointcut): 用于指定目标组件的表达式。指的是方面组件和哪一批目标组件方法有关系。多个连接点组成的集合就是切入点。如：a、b 为切入点，1、2 为连接点。



4) 通知 (Advice): 用于指定方面组件和目标组件方法之间的作用时机。例如：先执行方面组件再执行目标方法；或先执行目标方法再执行方面组件。

5) 目标 (Target): 利用切入点指定的组件和方法。

6) 动态代理 (AutoProxy): Spring 同样采用了动态代理技术实现了 AOP 机制。当使用 AOP 之后，从容器 getBean() 获取的目标组件，返回的是一个动态生成的代理类。然后通过代理类执行业务方法，代理类负责调用方面组件功能和原目标组件功能。

Spring 提供了下面两种动态代理技术实现：

1) 采用 CGLIB 技术实现（目标组件没有接口采用此方法）

例如：public class 代理类 extends 原目标类型 {}
CostAction action=new 代理类(); //代理类中有原来类的方法

2) 采用 JDK Proxy API 实现（目标组件有接口采用此方法，即实现了某个接口）

例如：Public class 代理类 implements 原目标接口 {}
CostDAO costDAO=new 代理类(); //代理类去实现了原目标接口，所以没

有原来类的方法

5.4 案例：AOP 的使用，模拟某些组件需要记录日志的功能

接3.3、3.4案例，想让所有的操作进行日志记录，那么按以前的方式就需要给所有 Action 或 DAO 中添加记录日志的代码，如果 Action 或 DAO 很多，那么不便于维护。而使用 AOP 机制，则可以很方便的实现上述功能：

step1：导入 AOP 需要的包：aopalliance.jar、aspectjrt.jar、aspectjweaver.jar、cglib-nodep-2.1_3.jar
step2：在 org.tarena.aop 包下新建 LoggerBean 类，并添加 logger 方法用于模拟记录日志功能

```
public void logger(){    System.out.println("记录了用户的操作日志"); }
```

step3：在 applicationContext.xml 配置文件中，添加 AOP 机制

```
<bean id="loggerBean" class="org.tarena.aop.LoggerBean"></bean>
<aop:config>
    <!--定义切入点，指定目标组件和方法。id：可任意起个名字。expression：指定哪些组件是目标，并作用在这些目标的方法上。下面表示所有 Action 中的所有方法为切入点-->
    <aop:pointcut id="actionPointcut" expression="within(org.tarena.action.*)" />
    <!--定义方面，将 loggerBean 对象指定为方面组件，loggerBean 从普通 Bean 组件升级为了方面组件-->
    <aop:aspect id="loggerAspect" ref="loggerBean">
        <!-- aop:before 在操作前执行 aop:after 操作后执行 -->
        <!-- 定义通知，aop:before：指定先执行方面组件的 logger 方法，再执行切入点指定的目标方法。aop:after：与 aop:before 相反 -->
        <aop:before pointcut-ref="actionPointcut" method="logger"/>
    </aop:aspect>
</aop:config>
```

step4：执行 3.3 案例 step3，则发现添加操作已有了记录日志功能

创建 CostDAO 对象 初始化 CostDAO 对象 记录了用户的操作日志

处理资费添加操作 利用 JDBC 技术实现保存资费记录

step5：执行 3.4 案例 step3，则发现删除操作已有了记录日志功能，记得加无参构造方法！

记录了用户的操作日志 处理资费删除操作

利用 Hibernate 技术实现删除资费记录

◆ 注意事项：DeleteAction 用的是构造注入，所以此处要把无参构造器再加上！因为 AOP 底层调用了 DeleteAction 的无参构造方法。不加则报错：Superclass has no null constructors but no arguments were given

5.5 通知类型

通知决定方面组件和目标组件作用的关系。主要有以下几种类型通知：

- 1) 前置通知：方面组件在目标方法之前执行。
- 2) 后置通知：方面组件在目标方法之后执行，目标方法没有抛出异常才执行方面组件。
- 3) 最终通知：方面组件在目标方法之后执行，目标方法有没有异常都会执行方面组件。
- 4) 异常通知：方面组件在目标方法抛出异常后才执行。
- 5) 环绕通知：方面组件在目标方法之前和之后执行。

```
try{ //前置通知执行时机<aop:before>
```

```
    //执行目标方法
```

```

    //后置通知执行时机<aop:after-returning>
    }catch(Exception e){//异常通知执行时机<aop:after-throwing>
        }finally{ //最终通知执行时机<aop:after>
        }//环绕通知等价于前置+后置<aop:around>

```

5.6 切入点

切入点用于指定目标组件和方法，Spring 提供了多种表达式写法：

1) 方法限定表达式：指定哪些方法启用方面组件。

①形式：execution(修饰符? 返回类型 方法名(参数列表) throws 异常?)

②示例：

```

execution(public * *(..)), 匹配容器中, 所有修饰符是 public (不写则是无要求的), 返回类型、方法名都没要求, 参数列表也不要求的方法。
execution(* set*(..)), 匹配容器中, 方法以 set 开头的所有方法。
execution(* org.tarena.CostDAO.*(..)), 匹配 CostDAO 类中的所有方法。
execution(* org.tarena.dao.*.*(..)), 匹配 dao 包下所有类所有方法。
execution(* org.tarena.dao..*.*(..)), 匹配 dao 包及其子包中所有类所有方法。

```

2) 类型限定表达式：指定哪些类型的组件的所有方法启用方面组件（默认就是所有方法都启用，且知道类型，不到方法）。

①形式：within(类型) ②示例：

```

within(com.xyz.service.*), 匹配 service 包下的所有类所有方法
within(com.xyz.service..*), 匹配 service 包及其子包中的所有类所有方法
within(org.tarena.dao.CostDAO), 匹配 CostDAO 所有方法

```

◆ 注意事项：within(com.xyz.service..*.*), 为错误的，就到方法名！

3) Bean 名称限定：按<bean>元素的 id 值进行匹配。

①形式：Bean(id 值) ②示例：

```

bean(costDAO), 匹配 id=costDAO 的 bean 对象。
bean(*DAO), 匹配所有 id 值以 DAO 结尾的 bean 对象。

```

4) args 参数限定表达式：按方法参数类型限定匹配。

①形式：args(类型) ②示例：

```

args(java.io.Serializable), 匹配方法只有一个参数, 并且类型符合 Serializable 的方法, public void f1(String s)、public void f2(int i) 都能匹配。

```

◆ 注意事项：上述表达式可以使用&&、|| 运算符连接使用。

5.7 案例：环绕通知，修改 5.4 案例使之动态显示所执行的操作

step1：新建 opt.properties 文件，自定义格式：包名.类名.方法名=操作名。在高版本 MyEclipse 中，切换到 Properties 界面，点击 Add 直接输入键和值，则中文会自动转为 ASCII 码。低版本的则需要使用 JDK 自带的转换工具：native2ascii.exe

```

#第一个为资费添加, 第二个为资费删除
org.tarena.action.CostAction.execute=\u8D44\u8D39\u6DFB\u52A0
org.tarena.action.DeleteAction=\u8D44\u8D39\u5220\u9664

```

step2：新建 PropertiesUtil 工具类，用于解析 properties 文件

```

private static Properties props = new Properties();
static {
    try {
        props.load(PropertiesUtil.class.getClassLoader()
            .getResourceAsStream("opt.properties"));
    }
}

```

```

        }catch(Exception ex){  ex.printStackTrace();    }
    }

    public static String getValue(String key){
        String value =  props.getProperty(key);
        if(value == null){  return "";    }else{    return value;    }
    }
}

```

step3：使用环绕通知，将 5.4 案例 step3 中的<aop:before />标签换为<aop:around />

```
<aop:around pointcut-ref="actionPointcut" method="logger"/>
```

step4：修改 5.4 案例 step2 中的 LoggerBean 类

```

public Object logger(ProceedingJoinPoint pjp) throws Throwable{//采用环绕通知，加参数
    //前置逻辑
    String className=pjp.getTarget().getClass().getName();//获取要执行的目标组件类名
    String methodName=pjp.getSignature().getName();//获取要执行的方法名
    //根据类名和方法名，给用户提示具体操作信息
    String key=className+"."+methodName;      System.out.println(key);
    //解析 opt.properties，根据 key 获取 value
    String value=PropertiesUtil.getValue(key);
    //XXX 用户名可以通过 ActionContext.getSession 获取
    System.out.println("XXX 执行了"+value+"操作！操作时间：" +
        new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(
            new Date(System.currentTimeMillis())));
    Object obj=pjp.proceed();//执行目标方法
    //后置逻辑
    return obj;
}

```

step5：分别执行 3.3 案例 step3 和 3.4 案例 step3，执行结果动态显示所执行的操作及时间

```
XXX 执行了资费添加操作！操作时间：2013-08-19 20:14:47
```

```
XXX 执行了资费删除操作！操作时间：2013-08-19 20:15:45
```

5.8 案例：利用 AOP 实现异常处理，将异常信息写入文件

1) 分析：方面：将异常写入文件。切入点：作用到所有 Action 业务方法上
within(org.tarena.action..*)。通知：异常通知<aop:after-throwing>。

2) 实现：step1：在 org.tarena.aop 包中创建 ExceptionBean 类

```

public class ExceptionBean {//模拟，将异常信息写入文件
    public void exec(Exception ex){//ex 代表目标方法抛出的异常
        System.out.println("将异常记录文件"+ex); //记录异常信息
    }
}

```

step2：在 applicationContext.xml 配置文件中进行配置

```

<bean id="exceptionBean" class="org.tarena.aop.ExceptionBean"></bean>
<aop:pointcut id="actionPointcut" expression="within(org.tarena.action..*)" />
<!-- 定义方面组件，将 exceptionBean 指定为方面 -->
<aop:aspect id="exceptionAspect" ref="exceptionBean">
    <!-- throwing：和自定的方法中的参数名相同。一定要把异常抛出来才行！
        try-catch 了则不行！ -->
    <aop:after-throwing pointcut-ref="actionPointcut" method="exec" throwing="ex"/>
</aop:aspect>

```

step3：在 DeleteAction 的 execute 方法中添加异常

```
String str=null;      str.length();
```

step4：执行 3.3 案例 step3 则添加操作执行正常；执行 3.4 案例 step3 则删除操作报空指针异常！显示结果：将异常记录文件 java.lang.NullPointerException

一百四十、Log4j 日志记录工具

6.1 Log4j 介绍

Log4j 主要用于日志信息的输出。可以将信息分级别（错误、严重、警告、调式信息）按不同方式（控制台、文件、数据库）和格式输出。

Log4j 主要有以下 3 部分组件构成：

- 1) 日志器 (Logger): 负责消息输出，提供了各种不同级别的输出方法。
- 2) 输出器 (Appender): 负责控制消息输出的方式，例如输出到控制台、文件输出等。
- 3) 布局器 (格式器，Layout): 负责控制消息的输出格式。

6.2 Log4j 的使用

step1：引入 log4j.jar

step2：在 src 下添加 log4j.properties（定义了消息输出级别、采用哪种输出器、采用哪种布局器）

```
#level: 大小写都可, myconsole 是自己随便起的 appender 名字, 可以写多个 appender
log4j.rootLogger=debug,myconsole,myfile
#appender: 可在 org.apache.log4j 中找自带的类
log4j.appender.myconsole=org.apache.log4j.ConsoleAppender
log4j.appender myfile=org.apache.log4j.FileAppender
#log4j.appender myfile.File=D:\\error.txt
log4j.appender myfile.File=D:\\error.html
#layout: 可在 org.apache.log4j 中找自带的类
log4j.appender.myconsole.layout=org.apache.log4j.SimpleLayout
log4j.appender myfile.layout=org.apache.log4j.HTMLLayout
```

◆ 注意事项：级别从小到大为：debug、info、warn、error、fatal

step3：创建 TestLog4j 类，测试利用日志器不同的方法输出消息。

```
public class TestLog4j {
    public static Logger logger=Logger.getLogger(TestLog4j.class);
    public static void main(String[] args) {
        //能显示就显示，不显示也不会影响主程序后面的运行，仅是个辅助工具
        logger.debug("调试信息");      logger.info("普通信息");
        logger.warn("警告信息");       logger.error("错误信息");
        logger.fatal("致命信息");      }
    }
```

◆ 注意事项：

- ❖ 导包为 org.apache.log4j.Logger。
- ❖ 若在 log4j.properties 中指定的级别为 debug，则五种信息都会显示；若指定的级别为 error，则只显示 error 和 fatal 信息。

6.3 案例：修改 5.8 案例，使用 Log4j 记录日志

step1：继续使用 6.2 节 step1 和 step2

step2：修改 5.8 案例 step1

```

public class ExceptionBean {//将异常信息写入文件
    Logger logger=Logger.getLogger(Exception.class);
    public void exec(Exception ex){//ex 代表目标方法抛出的异常
        logger.error("====异常信息====");//记录异常信息
        logger.error("异常类型"+ex);
        StackTraceElement[] els=ex.getStackTrace();
        for(StackTraceElement el:els){ logger.error(el); }
    }
}

```

step3：执行 3.4 案例 step3 则删除操作报空指针异常（前提：已进行了 5.8 案例 step3 操作）！
由于 log4j.properties 配置了两种输出方式，所以两种方式都有效。

控制台的显示结果：

```

XXX 执行了资费删除操作！操作时间：2013-08-20 12:47:54
ERROR - =====异常信息=====
ERROR - 异常类型 java.lang.NullPointerException
..... .... ...

```

HTML 显示结果：

Log session start time Tue Aug 20 12:47:53 CST 2013

Time	Thread	Level	Category	Message
0	main	ERROR	java.lang.Exception	=====异常信息=====
2	main	ERROR	java.lang.Exception	异常类型java.lang.NullPointerException

一百四十一、Spring 注解配置

注解技术从 JDK5.0 推出，之后很多框架开始提供注解配置形式。Spring 框架从 2.5 版本开始支持注解配置。注解配置的优点：简单、快捷。

7.1 组件扫描功能

Spring 可以按指定的包路径扫描内部的组件，当发现组件类定义前有一下的注解标记，会将该组件纳入 Spring 容器中。

- 1) @Component (其他组件)
 - 2) @Controller (Action 组件，负责调 Service)
 - 3) @Service (Service 组件，负责调 DAO，处理一些额外逻辑)
 - 4) @Repository (DAO 组件，负责访问数据库)
- ◆ 注意事项：
- ❖ 括号中的为推荐用法，上述 4 个注解任意用也可以，但不符合规范。
 - ❖ 注解只能用在类定义前、方法定义前、成员变量定义前！

7.2 组件扫描的使用方法

step1：在 applicationContext.xml 配置文件中开启组件扫描配置

```
<!-- 开启组件扫描，base-package 指定扫描包路径。使用前提：要有 xmlns:context 命名空间的引入。base-package="org.tarena" 这么写，则 dao 和 action 都能被扫描-->
<context:component-scan base-package="org.tarena" />
```

step2：在要扫描的组件的类定义前使用上述注解标记即可。例如在 JdbcCostDAO 类前使用

```
@Repository
public class JdbcCostDAO implements CostDAO {
    public JdbcCostDAO(){    System.out.println("创建 CostDAO 对象");    }
    @PostConstruct//等价于设置了 init-method="方法名"属性
    public void myinit(){    System.out.println("初始化 CostDAO 对象");    }
    @PreDestroy//等价于设置了 destroy-method="方法名"属性
    public void mydestroy(){    System.out.println("销毁 CostDAO 对象");    }
    .....
}
```

@Repository 等价于原来配置文件中的：

```
<bean id="jdbcCostDAO" class="org.tarena.dao.JdbcCostDAO"></bean>
```

加上@Scope("prototype")等价于原配置文件中的：

```
<bean id="jdbcCostDAO" scope="prototype" class="org.tarena.dao.JdbcCostDAO"></bean>
```

- ◆ 注意事项：
- ❖ 上述标记将组件扫描到容器后，id 属性默认是类名首字母小写。如果需要自定义 id 值，可以使用@Repository("自定义 id 值")，其他注解也同理。
 - ❖ 类的命名和变量的命名要规范！首字母大写，第二个字母要小写！否则在使用框架时会有冲突或无法识别，如类名为 JDBCOSTDAO 时无法识别它的 id 值：jDBCCostDAO，此时以它的类名作为 id 值却可识别：JDBCCostDAO。
 - ❖ 默认采用 singleton 模式创建 Bean 对象，如果需要改变，可以使用 @Scope("prototype") 定义。
 - ❖ lazy-init="true" 属性只能在<beans>根元素定义了，没有对应的注解。

step3：创建 TestAnnotation 类，用于测试注解

```

@Test //组件扫描
public void test10{      String conf="/applicationContext.xml";
    ApplicationContext ac=new ClassPathXmlApplicationContext(conf);
    //获取扫描到容器的 Bean 对象
    CostDAO costDAO=(CostDAO)ac.getBean("jdbcCostDao");
    costDAO.delete(); //组件扫描， 默认的 id 为类名首字母小写！且默认单例模式 }

```

7.3 注入注解标记使用方法

如果容器中两个符合要求可被注入同一个组件的 Bean 对象，可以采用下面注解标记：

1) @Resource， 默认按**类型匹配**注入（JDK 自带的）。若有多个符合要求的类型，则报错：匹配不唯一，那么就需要采取按名称注入的方式，它的使用格式为：

```
@Resource(name="需要注入的 Bean 对象 id 值")。
```

2) @Autowired， 默认按**类型匹配**注入（Spring 提供的）。若有多个符合要求的类型，则采取按名称注入的方式，它的使用格式为：

```
@Autowired
@Qualifier("需要注入的 Bean 对象 id 值")
```

◆ 注意事项：注入标记在成员变量定义前，但@Resource 也可以在 set 方法前使用！

3) 案例：id 为 hibernateCostDao 的 Bean 对象和 id 为 costDao 的 Bean 对象，都符合 CostDAO 接口，在 CostAction 组件中注入，那么此时将会报错：匹配不唯一。解决如下：

step1：修改 CostAction，添加注入标记

```

@Controller("costAction")
@Scope("prototype")
public class CostAction {
    //@Resource//将 costDao 注入，按类型匹配注入，JDK 自带的
    //@Autowired//将 costDao 注入，按类型匹配注入，Spring 提供的
    //@Resource(name="hibernateCostDao")//当有多个符合要求的类型，则按名称注入
    @Autowired
    @Qualifier("hibernateCostDao")//当有多个符合要求的类型，则按名称注入
    private CostDAO costDAO;
    public void setCostDAO(CostDAO costDAO) { this.costDAO = costDAO; }
    .....
}

```

step2：在 TestAnnotation 类，添加方法测试注入标记

```

@Test //注入标记测试
public void test20{      String conf="/applicationContext.xml";
    ApplicationContext ac=new ClassPathXmlApplicationContext(conf);
    CostAction costAction=(CostAction)ac.getBean("costAction");
    costAction.execute();
}

```

step3：可正常执行，如果没写注入标记则报错：NullPointerException

7.4 AOP 注解标记使用方法

step1：在 applicationContext.xml 配置文件中开启 AOP 注解

```
<aop:aspectj-autoproxy /><!--之前的配置可都删除，只留根元素-->
```

step2：在方面组件中，使用下面注解标记：

1) 首先使用@Component 将组件扫描到 Spring 容器。

- 2) 然后使用@Aspect 将组件定义为方面组件。
- 3) 之后定义一个空方法（方法名随便起）在方法前使用@Pointcut 定义切入点表达式。
- 4) 最后在方面组件的处理方法前使用 @Around、@Before、@AfterReturning、
@AfterThrowing、@After

例如：修改 5.7 案例 step4 中的 LoggerBean 类

```
@Component//将组件扫描到 Spring 容器
@Aspect//将该组件定义为方面组件
public class LoggerBean {
    //定义切入点
    @Pointcut("within(org.tarena.action..*)")
    public void mypoint(){}//主要目的是使用@Pointcut标记,id则为它的方法名 mypoint
    //采用环绕通知
    @Around("mypoint()")//方法即为下面的方法名
    public Object logger(ProceedingJoinPoint pjp) throws Throwable{
        //.....方法体内容没变.....}
    }}
```

- ◆ 注意事项：@Pointcut 注解在 JDK1.7 中不能识别，只能把切入点表达式写在通知中：

@Around("within(org.tarena.action..*)")。而此用法 JDK1.6 也支持。

step3：再次执行 3.3 案例 step3，则也可正常执行

step4：把 6.3 案例 step2 中的 ExceptionBean 修改为使用 AOP 注解

```
@Component//将组件扫描到 Spring 容器
@Aspect//将该组件定义为方面组件
public class ExceptionBean {
    Logger logger=Logger.getLogger(Exception.class);
    @Pointcut("within(org.tarena.action..*)")
    public void mypoint(){}
    @AfterThrowing(pointcut="mypoint()",throwing="ex")//方法名即为下面方法的名字
    public void exec(Exception ex){ //.....方法体内容没变.....}
}
```

step5：执行 6.3 案例 step3，则正常执行，控制台显示空指针异常，异常信息也被写入 HTML 文件。

一百四十二、Spring 对数据访问技术的支持

8.1 Spring 提供了统一的异常处理类型

- 1) SQLException 是 JDBC 抛的异常。
- 2) org.hibernate.XXXException 是 Hibernate 抛出的异常。
- 3) Spring 提供的异常根类: DataAccessException, 但是很多异常都被 try-catch 了, 所以出错后看不到提示, 因此要用 Log4j。

8.2 Spring 提供了编写 DAO 的支持类

- 1) DaoSupport 类: JdbcDaoSupport、HibernateDaoSupport, 自己写的 DAO 按使用的访问技术, 有选择的继承它们 (类似于以前写的 BaseDAO 类)。
- 2) Template 类: JdbcTemplate、HibernateTemplate, 封装了通用操作, 如: 增删改查。特殊操作, 如: 分页查询, 则仍需要使用 Hibernate 原来的方式, 详见 8.6 节。
- 3) 继承 DaoSupport 类后, 就可通过 getJdbcTemplate()、getHibernateTemplate()方法获得对应的 Template 类对象, 即可进行通用操作:

①update(): 实现增删改查。	②query(): 实现查询多行记录。
③queryForObject(): 实现查询单行记录。	④queryForInt(): 实现查询单个 int 值。
- 4) 将一条记录转换成一个实体对象, 需要实现 Spring 提供的 RowMapper 接口 (将实体与记录间的转换写在它的实现类中), 因为 Spring 提供的 Template 对象中的查询方法 query() 有 RowMapper 类型的参数。

8.3 Spring 提供了声明式事务管理方法

基于 AOP 配置实现, 不需要写 Java 代码, 加注解标签即可。详情见第十章。

8.4 Spring 框架如何使用 JDBC 技术

以前的 DBUtil 则不需要写了, 交给 Spring 配置。此例包含注解配置和 xml 配置。

step1: 新建一个工程, 引入 Spring 开发包 (IoC 和 AOP 开发包) 和配置文件

spring.jar/commons-logging.jar/aopalliance.jar/aspectjrt.jar/aspectjweaver.jar/cglib-nodep-2.1_3.jar

step2: 引入 JDBC 技术相关的开发包 (驱动包)

step3: 根据要操作的表, 编写对应的实体类。此处用 Hibernate 笔记 5.16 案例中 Cost 实体

step4: 编写 DAO 接口 CostDAO 和实现类 JdbcCostDAO (实现类继承 JdbcDaoSupport, 并使用其提供的 getJdbcTemplate 对象实现增删改查操作)

1) CostDAO 接口

```
public Cost findById(int id);    public void save(Cost cost);    public void delete(int id);
public void update(Cost cost);   public int count();           public List<Cost> findAll();
```

2) JdbcCostDAO 实现类

```
public class JdbcCostDAO extends JdbcDaoSupport implements CostDAO {
    public void delete(int id) {      String sql="delete from COST_CHANG where ID=?";
        Object[] params={id};  getJdbcTemplate().update(sql,params);      }
    public List<Cost> findAll() {     String sql="select * from COST_CHANG";
    /** 将一条记录转换成一个 Cost 对象, 需要实现 Spring 提供的 RowMapper 接口 */
    }
```

```

        RowMapper mapper=new CostRowMapper();
        List<Cost> list=getJdbcTemplate().query(sql,mapper);      return list;      }
    public int count(){      String sql="select count(*) from COST_CHANG";
        int size=getJdbcTemplate().queryForInt(sql);                  return size;      }
    public Cost findById(int id) {      String sql="select * from COST_CHANG where ID=?";
        Object[] params={id};      RowMapper mapper=new CostRowMapper();
        Cost cost=(Cost)getJdbcTemplate().queryForObject(sql, params,mapper);
        return cost;      }
    public void save(Cost cost) {
        String sql="insert into COST_CHANG(ID,NAME,BASE_DURATION," +
            "BASE_COST,UNIX_COST,STATUS,DESCR," +
            "STARTTIME) values(COST_SEQ_CHANG.nextval,?,?,?,?,?,?)";
        Object[] params={ cost.getName(),                           cost.getBaseDuration()
            cost.getBaseCost(),
            cost.getUnitCost(),  cost.getStatus(),  cost.getDescr(),
            cost.getStartTime() };
        getJdbcTemplate().update(sql,params);
    }
    public void update(Cost cost) {
        String sql="update COST_CHANG set NAME=?,BASE_DURATION=?,"
            + "BASE_COST=?,UNIX_COST=?,STATUS=?,DESCR=?," +
            "STARTTIME=? where ID=?";
        Object[] params={ cost.getName(),                           cost.getBaseDuration()
            cost.getBaseCost(),
            cost.getUnitCost(),  cost.getStatus(),  cost.getDescr(),  cost.getStartTime(),
            cost.getId() };
        getJdbcTemplate().update(sql,params);
    }
}

```

3) 在 org.tarena.entity 包中创建 CostRowMapper 类

```

public class CostRowMapper implements RowMapper{//实现 RowMapper 接口
    /** rs: 结果集。index: 当前记录的索引。Object: 返回实体对象。 */
    public Object mapRow(ResultSet rs, int index) throws SQLException {
        Cost cost = new Cost();      cost.setId(rs.getInt("ID"));
        cost.setName(rs.getString("NAME"));
        cost.setBaseDuration(rs.getInt("BASE_DURATION"));
        cost.setBaseCost(rs.getFloat("BASE_COST"));
        cost.setUnitCost(rs.getFloat("UNIT_COST"));
        cost.setStartTime(rs.getDate("STARTTIME"));
        cost.setStatus(rs.getString("STATUS"));      return cost;
    }
}

```

step5：将 DAO 组件交给 Spring 容器，在 applicationContext.xml 中进行相关配置

1) 定义 DAO 组件的<bean>元素

```

<bean id="costDao" class="org.tarena.dao.impl.JdbcCostDAO">
    <!-- 此处需要注入连接资源给 DaoSupport，用于实例化 Template 对象，否则没有
    与数据库的连接！ dataSource: 代表连接池对象。此处实际是给 JdbcDaoSupport 注入连接，
    用于实例化 JdbcTemplate 对象。 -->
    <property name="dataSource" ref="MyDataSource"></property>

```

```
</bean>
```

2) 需要 DAO 的 Bean 注入一个 dataSource 对象。dataSource 对象采用一个连接池构建（此处使用 dbcp 连接池），先引入 dbcp 连接池开发包（ commons-pool.jar、commons-dbcp-1.2.2.jar、commons-collections-3.1.jar），再定义 dataSource 对象的<bean>元素。

```
<!-- 定义连接池 Bean 对象 -->
<bean id="MyDataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
    <!-- 注入数据库连接参数 -->
    <property name="url" value="jdbc:oracle:thin:@localhost:1521:dbchang"></property>
    <property
      name="driverClassName"
      value="oracle.jdbc.driver.OracleDriver"></property>
    <property name="username" value="system"></property>
    <property name="password" value="chang"></property>
    <property name="maxActive" value="20"></property><!-- 设置连接最大数 -->
    <!-- 连接池实例化时初始创建的连接数 -->
    <property name="initialSize" value="2"></property>
</bean>
```

step6：创建 TestJdbcCostDAO 类，用于测试 xml 配置，可正常执行

```
@Test
public void testFindAll() {//测试基于 xml 配置方法
    String conf="/applicationContext.xml";
    ApplicationContext ac = new ClassPathXmlApplicationContext(conf);
    CostDAO costDAO=(CostDAO)ac.getBean("costDao");
    List<Cost> list=costDAO.findAll();    System.out.println("总数: "+costDAO.count());
    for(Cost c:list){    System.out.println(c.getId()+" "+c.getName());    }
    System.out.println("findById: "+costDAO.findById(1).getName());    }
```

step7：使用注解配置，保留 step5 中的连接资源 MyDataSource，而 Bean 组件 costDao 删除，并添加开启组件扫描的配置

```
<context:component-scan base-package="org.tarena" /><!-- 开启组件扫描 -->
```

step8：在 JdbcCostDAO 中添加注解

```
@Repository //id 值默认为类名首字母小写
@Scope("prototype") //设置非单例模式
public class JdbcCostDAO extends JdbcDaoSupport implements CostDAO {
    @Resource //将容器中的 myDataSource 按类型匹配注入
    /** 虽然继承了 JdbcDaoSupport，但我们无法在别人的代码中添加注解，所以我们添加一个 set 方法，注意名字别用 setDataSource! 因为 JdbcDaoSupport 有同名的 set 方法，且是 final 修饰的，所以需要稍微改变一下。 */
    public void setMyDataSource(DataSource ds){
        super.setDataSource(ds); //将注入的 dataSource 给 DaoSupport 注入
    }
    .....
}
```

step9：在 TestJdbcCostDAO 类添加方法，用于测试注解配置，可正常执行

```
@Test
public void testFindAllByAnnotation() {//测试基于注解配置方法
    String conf="/applicationContext.xml";
```

```

ApplicationContext ac = new ClassPathXmlApplicationContext(conf);
CostDAO costDAO=(CostDAO)ac.getBean("jdbcCostDAO");
List<Cost> list=costDAO.findAll();    System.out.println("总数: "+costDAO.count());
for(Cost c:list){      System.out.println(c.getId()+" "+c.getName());      }
System.out.println("findById: "+costDAO.findById(1).getName());      }

```

- ◆ 注意事项：此时由于一开始定义的 id 为 costDao 的 Bean 已被删除，所以 TestJdbcCostDAO 类中的 testfindAll 方法已无法执行。在测试注解配置时，可以新建 applicationContext-annotation.xml 文件，把注解开启、连接资源写这里，之后在测试方法中，指定它的名字即可！这样两个方法都可执行。

step10：优化，当大量 Bean 组件需要采取注解配置时，每个 Bean 都要写 set 方法注入 dataSource 连接资源，所以比较麻烦，可采取继承方式：

- 1) 新建 JdbcBaseDAO 类，并继承 JdbcDaoSupport，把 set 方法写入

```

public class JdbcBaseDAO extends JdbcDaoSupport {
    @Resource //将容器中的 myDataSource 按类型匹配注入
    public void setMyDataSource(DataSource ds){//注意名字，详见 step8
        super.setDataSource(ds);      }      }

```

- 2) 删除 step8 中 JdbcCostDAO 的 set 方法及其前面的注解，并继承 JdbcBaseDAO

```
public class JdbcCostDAO extends JdbcBaseDAO implements CostDAO { ..... }
```

8.5 连接池优点

- 1) 增强数据访问的稳定性。
- 2) 连接池可以将连接数控制在安全的范围内。
- 3) 连接池中的连接对象始终与数据库保持联通状态，它的 close 方法被重写，不是真正的关闭，而是把连接又放回池中，避免了重复的新建连接和释放连接过程。

8.6 Spring 框架如何使用 Hibernate 技术

Hibernate 的主配置也不需要了，也交给 Spring 配置。此例包含注解配置和 xml 配置。

step1：新建一个工程，引入 Spring 开发包（IoC 和 AOP 开发包）和配置文件

```
spring.jar/commons-logging.jar/aopalliance.jar/aspectjrt.jar/aspectjweaver.jar/cglib-nodep-2.1_3.jar
```

step2：引入 Hibernate 相关的开发包（Hibernate 开发包+驱动）

step3：编写实体类和 hbm.xml 映射描述文件，此处用 Hibernate 笔记 5.16 案例中 Cost 实体及其 Cost.hbm.xml 映射文件

step4：编写 DAO 接口和 HibernateCostDAO 实现类（实现类继承 HibernateDaoSupport，并使用其提供的 HibernateTemplate 对象实现增删改查操作）

- 1) CostDAO 接口，与 8.4 节 step4 中 1) 基本相同，但额外添加一个方法

```
public List<Cost> findPage(int page,int pageSize);//分页查询
```

- 2) HibernateCostDAO 实现类

```

public class HibernateCostDAO extends HibernateDaoSupport implements CostDAO {
    public int count() {      String hql="select count(*) from Cost";
        List list=getHibernateTemplate().find(hql);
        int size=Integer.parseInt(list.get(0).toString());
        /** 最好先变成 string 再转成 int。不要强转，Spring 的不同版本返回值可能是 long、Integer
        类型 */      return size;      }

```

```

public void delete(int id) {      Cost cost=findById(id);
    getHibernateTemplate().delete(cost); } LICHOO
public List<Cost> findAll() {      String hql="from Cost";
    List<Cost> list=getHibernateTemplate().find(hql);      return list; }
public Cost findById(int id) {
    Cost cost=(Cost)getHibernateTemplate().get(Cost.class, id);  return cost; }
public void save(Cost cost) {      getHibernateTemplate().save(cost); }
public void update(Cost cost) {      getHibernateTemplate().update(cost); }
public List<Cost> findPage(final int page,final int pageSize) { //分页查询，参数为 final
    List<Cost> list=(List<Cost>)getHibernateTemplate().execute(
        new HibernateCallback(){//回调函数及下面的 Session 如何使用详见 8.7 节
            public Object doInHibernate(Session session)//记得改参数名
                throws HibernateException, SQLException {
                String hql="from Cost";
                Query query=session.createQuery(hql);//使用 session 对象
                int begin=(page-1)*pageSize;
                query.setFirstResult(begin);
                query.setMaxResults(pageSize);
                return query.list();
            }
        }
    );
    return list; } } 
```

step5：将 DAO 组件交给 Spring 容器管理，在 applicationContext.xml 中进行相关配置

```

<bean id="hibernateCostDao" scope="prototype"
      class="org.tarena.dao.impl.HibernateCostDAO">
    <property name="sessionFactory" ref="MySessionFactory"></property>
</bean><!-- name: 代表 Hibernate 的连接资源，该资源要么是 hibernateTemplate 类型，  

要么是 sessionFactory 类型（按 Alt+/就会显示），我们用 sessionFactory 类型。ref: 名字任意起，是我们配置的 sessionFactory 连接资源，有了该资源，getHibernateTemplate()方法才能执行 -->
```

step6：在 applicationContext.xml 中配置 sessionFactory 资源（相当于 Hibernate 的主配置）

```

<bean id="MySessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <!-- LocalSessionFactoryBean 是 Hibernate 中 SessionFactory 的子类，我们不用写 -->
    <!-- 注入数据库连接信息，此处要再配置一下 dataSource 数据库连接资源 -->
    <property name="dataSource" ref="MyDataSource"></property><!--ref: 名字任意起  

-->
    <!-- 注入 Hibernate 配置参数 -->
    <property name="hibernateProperties">
        <props><!-- 放 Spring 中属性要加个前缀 hibernate -->
            <prop key="hibernate.dialect">org.hibernate.dialect.OracleDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.format_sql">true</prop>
    
```

```

        </props>
    </property>
    <!-- 注入映射描述 -->
    <property name="mappingResources">
        <list><value>org/tarena/entity/Cost.hbm.xml</value></list>
    </property>
</bean>

```

step7：在 applicationContext.xml 中配置 dataSource 数据库连接资源，与 8.4 案例 step5 中 2) 相同，dataSource 对象采用一个连接池构建（此处使用 dbcp 连接池），先引入 dbcp 连接池开发包（commons-pool.jar、commons-dbcp-1.2.2.jar、commons-collections-3.1.jar）

```

<!-- 定义连接池 Bean 对象 -->
<bean id="MyDataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
    <!-- 注入数据库连接参数 -->
    <property name="url" value="jdbc:oracle:thin:@localhost:1521:dbchang"></property>
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"></property>
    <property name="username" value="system"></property>
    <property name="password" value="chang"></property>
    <property name="maxActive" value="20"></property><!-- 设置连接最大数 -->
    <!-- 连接池实例化时初始创建的连接数 -->
    <property name="initialSize" value="2"></property>
</bean>

```

◆ 注意事项：此案例的 step5-7 的注入关系为：dataSource 对象（dbcp 连接池构建的）注入给 sessionFactory 对象，sessionFactory 对象注入给 DAO（Bean）对象。

step8：创建 TestHibernateCostDAO 类，用于测试 xml 配置，可正常执行

```

@Test //测试基于 xml 配置方法
public void testFindAll() {  String conf="/applicationContext.xml";
    ApplicationContext ac = new ClassPathXmlApplicationContext(conf);
    CostDAO costDAO=(CostDAO)ac.getBean("hibernateCostDao");
    List<Cost> list=costDAO.findAll();  System.out.println("总数: "+costDAO.count());
    for(Cost c:list){  System.out.println(c.getId()+" "+c.getName());  }
    System.out.println("findById: "+costDAO.findById(1).getName());  }

```

step9：使用注解配置，新建 applicationContext-annotation.xml 配置文件（此处使用的为 8.4 案例 step9 中注意事项说的方式），在其中添加 step6 中的 sessionFactory 资源，添加 step7 中的 dataSource 数据库连接资源（不要 step5 中的 Bean 组件 hibernateCostDao），并添加开启组件扫描的配置

```
<context:component-scan base-package="org.tarena" /><!-- 开启组件扫描 -->
```

step10：在 HibernateCostDAO 中添加注解

```

@Repository("hibernateCostDao")
@Scope("prototype")
public class HibernateCostDAO extends HibernateDaoSupport implements CostDAO {
    @Resource //setSessionFactory 名字用不了是 final 的，同理说明见 8.4 案例 step8
    public void setMySessionFactory(SessionFactory sf){

```

```
super.setSessionFactory(sf); //将注入的 sessionFactory 给 HibernateDaoSupport 传  
入  
}  
.....
```

step11: 在 TestHibernateCostDAO 类添加方法, 用于测试注解配置, 可正常执行

```
@Test //测试基于注解配置方法  
public void testFindAllByAnnotation() { String conf="/applicationContext-annotation.xml";  
    ApplicationContext ac = new ClassPathXmlApplicationContext(conf);  
    CostDAO costDAO=(CostDAO)ac.getBean("hibernateCostDao");  
    List<Cost> list=costDAO.findAll(); System.out.println("总数: "+costDAO.count());  
    for(Cost c:list){ System.out.println(c.getId()+" "+c.getName()); }  
    System.out.println("findById: "+costDAO.findById(1).getName()); }
```

step12: 在 TestHibernateCostDAO 类添加方法, 用于测试分页查询

```
@Test //分页查询  
public void testFindByPage() { String conf="/applicationContext-annotation.xml";  
    ApplicationContext ac = new ClassPathXmlApplicationContext(conf);  
    CostDAO costDAO=(CostDAO)ac.getBean("hibernateCostDao");  
    List<Cost> list=costDAO.findPage(2, 4);System.out.println(" 总 数 :  
"+costDAO.count());  
    for(Cost c:list){ System.out.println(c.getId()+" "+c.getName()); } }
```

8.7 Spring+Hibernate 如何使用 Session、Query 等对象

1) 方式一: 利用 HibernateDaoSupport 提供的 getSession() 方法:

没用到延迟加载的 API, 那么用这个方式简单些。但是有延迟加载的 API, 则会出现问题: session 关闭早了, 页面可能获取不到数据; 不关闭吧, 一但出了方法体则关不上了! 而多次访问数据库后, 就发现没结果, 因为连接数用完了。

```
Session session=getSession();  
.....利用 session 进行一些操作.....  
session.close(); //注意, 一定要释放!
```

◆ 注意事项: getHibernateTemplate 中的 API 都有释放操作, 所以自己不用再写。

2) 方式二: 利用 HibernateTemplate.execute()方法, 以回调函数方式使用。这种方式不用担心方式一出现的问题, session 的关闭由 HibernateTemplate 统一管理。

```
getHibernateTemplate().execute(  
    new HibernateCallback(){//实现该接口的 doInHibernate 方法  
        public Object doInHibernate(Session session)  
            throws HibernateException, SQLException {  
                //回调函数中使用 session, 可见 8.6 案例 step4  
            }  
    }  
)
```

8.8 Spring 框架和 Struts2 整合应用

step1: 新建一个工程, 引入 Spring 开发包 (IoC 和 AOP 开发包)、配置文件和 Struts2 的五个基本核心包

spring.jar/commons-logging.jar/aopalliance.jar/aspectjrt.jar/aspectjweaver.jar/cglib-nodep-2.1_3.jar/xwork-core.jar/struts-core.jar/ognl.jar/freemarker.jar/commons-fileupload.jar

step2: 在 org.tarena.action 包中新建 HelloAction

```
private String name;//output  
public String getName() { return name; }  
public void setName(String name) { this.name = name; }  
public String execute(){ name="Chang"; return "success"; }
```

step3: 新建 applicationContext.xml 文件并配置, 将 Action 或 DAO 等组件交给 Spring 容器

```
<bean id="helloAction" scope="prototype" class="org.tarena.action.HelloAction"></bean>
```

step4: 引入 Struts2 和 Spring 整合的开发包: struts-spring-plugin.jar。该开发包的作用为: 当 Struts2 请求过来时, Action 对象将交给整合包去 Spring 容器获取, 即 Struts2 不再产生 Action 对象, 详解介绍可看第九章。

step5: 新建 struts.xml 文件, 并配置<action>, 将 class 属性与 Spring 容器中<bean>元素的 id 属性保持一致。(整合包利用 class 值当作 id 标识去 Spring 容器获取 Bean 对象)

```
<struts>  
  <package name="spring05" extends="struts-default">  
    <!--关键: 利用 struts-spring-plugin.jar 去 Spring 容器寻找 Bean 对象  
        利用 class 属性当作 id 值去 Spring 容器获取, 详细介绍看第九章 -->  
    <!-- 原来为 class="org.tarena.action.HelloAction", 现在改为 id 值: helloAction  
-->  
    <action name="hello" class="helloAction"><result>/hello.jsp</result></action>  
  </package>  
</struts>
```

step6: 在 WebRoot 目录中, 新建 hello.jsp 和 index.jsp

1) hello.jsp

```
<body style="font-size:30px;font-style:italic;">${name }你好! </body>
```

2) index.jsp

```
<a href="hello.action">Hello 示例</a>
```

step7: 在 web.xml 中添加前端控制器

```
<filter>  
  <filter-name>StrutsFilter</filter-name>  
  <filter-class>  
    org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter  
  </filter-class>  
</filter><!-- 前端控制器 -->  
<filter-mapping>  
  <filter-name>StrutsFilter</filter-name>  
  <url-pattern>/*</url-pattern>  
</filter-mapping>
```

step8: 部署, 启动服务器, 准备测试。但是发现报错:

You might need to add the following to web.xml:

```
<listener>
```

```
<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
```

```
</listener>
```

step9：在 web.xml 中添加 ContextLoaderListener 组件配置（可以在启动服务器时，实例化 Spring 容器），但它的默认查找路径为：/WEB-INF/applicationContext.xml，所以要指定路径（src 中），否则还是会报错：java.io.FileNotFoundException

```
<!-- 指定 Spring 配置文件位置和名称 -->
<context-param><!-- 下面的名字是 ContextLoaderListener 里约定好的，必须这么写 -->
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value><!--classpath 代表
src-->
</context-param>
<!-- 在服务器启动时，实例化 Spring 容器，在中间过程无法插入实例化代码，因为都是自动的，所以没地方写！因此要配置 listener，启动服务器则创建 Spring 容器。 -->
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

◆ 注意事项：在前端控制器前添加！！否则无法生效。

step10：在浏览器中输入 http://localhost:8080/Spring05_Struts2t/index.jsp 进行测试

8.9 案例：采用 SSH 结构重构资费管理模块

修改 Hibernate 笔记 5.16 案例中的资费管理模块（原功能采用 Struts2+Hibernate 结构）。

采用 SSH 结构需要追加以下步骤：

step1：引入 Spring 开发包（IoC 和 AOP）和 applicationContext.xml 配置文件

```
spring.jar/commons-logging.jar/aopalliance.jar/aspectjrt.jar/aspectjweaver.jar/cglib-nodep-2.1_3.j
ar
```

=====重构 Spring+Hibernate=====

step2：编写 DAO 组件，采用 Spring+Hibernate 方式实现，如：新建 SpringHibernateCostDAOImpl

```
public class SpringHibernateCostDAOImpl extends HibernateDaoSupport implements
CostDAO { public void delete(int id) throws DAOException { Cost cost=findById(id);
getHibernateTemplate().delete(cost); }
public List<Cost> findAll() throws DAOException { String hql="from Cost";
List<Cost> list=getHibernateTemplate().find(hql); return list; }
public List<Cost> findAll(final int page,final int rowsPerPage) throws DAOException {
List list=(List)getHibernateTemplate().execute(
new HibernateCallback(){ @Override
public Object doInHibernate(Session session)
throws HibernateException, SQLException {
String hql="from Cost";
Query query=session.createQuery(hql);
int start=(page-1)*rowsPerPage;//设置分页查询参数
query.setFirstResult(start);//设置抓取记录的起点，从 0 开始（第一条记
录）
query.setMaxResults(rowsPerPage);//设置抓取多少条记录
}
}
}); }
```

```

        return query.list();//按分页参数查询      }      }
    );
    return list;
}

public Cost findById(Integer id) throws DAOException {
    Cost cost=(Cost)getHibernateTemplate().load(Cost.class, id);
    return cost;//有延迟问题！要配置 web.xml，详见 step10
}

public Cost findByName(String name) throws DAOException {
    String hql = "from Cost where name=?";          Object[] params = {name};
    List<Cost> list = getHibernateTemplate().find(hql,params);
    if(!list.isEmpty()){      return list.get(0);
    } else{      return null;      }
}

public int getTotalPages(int rowsPerPage) throws DAOException {
    String hql="select count(*) from Cost";//类名
    List list=getHibernateTemplate().find(hql);
    int totalRows=Integer.parseInt(list.get(0).toString());
    if(totalRows%rowsPerPage==0){      return totalRows/rowsPerPage;
    } else {      return (totalRows/rowsPerPage)+1;      }
}

public void save(Cost cost) throws DAOException {
    cost.setStatus("1");
    cost.setCreateTime(new Date(System.currentTimeMillis()));
    getHibernateTemplate().save(cost);
}

public void update(Cost cost) throws DAOException {
    getHibernateTemplate().update(cost);
}
}

```

step3：在 Spring 容器的配置文件中定义配置

1) 将 DAO 扫描到 Spring 容器

```
<context:component-scan base-package="com.tarena.netctoss" /><!--开启组件扫描-->
```

2) 配置 sessionFactory

<!-- 注意：这里的 id 值不能自定义了！必须写 sessionFactory，因为配置了
OpenSessionInViewFilter 后（session 关闭延迟到 JSP 解析后，详见 step10），默认找的就是这个名字 -->

```

<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <!-- LocalSessionFactoryBean 是 Hibernate 中 SessionFactory 的子类，我们不用写-->
    <!-- 注入数据库连接信息 -->
    <property name="dataSource" ref="MyDataSource"></property>
    <!-- 注入 Hibernate 配置参数 -->
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect"><!-- 放 Spring 中属性加个前缀 hibernate -->
                org.hibernate.dialect.OracleDialect
            </prop>
            <prop key="hibernate.show_sql">true</prop>
            <prop key="hibernate.format_sql">true</prop>
        </props>
    </property>

```

```

<!-- 注入映射描述 -->
<property name="mappingResources">
    <list>
        <value>com/tarena/netctoss/entity/Cost.hbm.xml</value>
        <value>com/tarena/netctoss/entity/Admin.hbm.xml</value>
    </list>
</property>
</bean>

```

3) 引入 dbcp 开发包 (连接池), 定义 DataSource

```

<!-- 定义连接池 Bean 对象 -->
<bean id="MyDataSource" class="org.apache.commons.dbcp.BasicDataSource"
      destroy-method="close">
    <!-- 注入数据库连接参数 -->
    <property name="url" value="jdbc:oracle:thin:@localhost:1521:dbchang"></property>
    <property name="driverClassName"
              value="oracle.jdbc.driver.OracleDriver"></property>
    <property name="username" value="system"></property>
    <property name="password" value="chang"></property>
    <!-- 设置连接最大数 -->
    <property name="maxActive" value="20"></property>
    <!-- 设置连接池实例化时初始创建的连接数 -->
    <property name="initialSize" value="1"></property>
</bean>

```

4) 将 sessionFactory 给 DAO 注入

```

@Repository      @Scope("prototype")//排版需要, 这里写一行了
public class SpringHibernateCostDAOImpl extends HibernateDaoSupport implements
CostDAO {
    @Resource
    public void setMySessionFactory(SessionFactory sf){//注意名字问题
        super.setSessionFactory(sf);
    }
    .....
}

```

step4: 测试 DAO 组件, 在 TestCostDAO 中添加方法, 可正常执行

```

@Test //Spring+Hibernate
public void testFindAllByPageBySpring() throws Exception{
    String conf="applicationContext.xml";
    ApplicationContext ac = new ClassPathXmlApplicationContext(conf);
    CostDAO costDao = (CostDAO)ac.getBean("springHibernateCostDAOImpl");
    List<Cost> list = costDao.findAll(1,5);
    for(Cost c : list){ System.out.println(c.getId()+" "+c.getName()); }
}

```

=====重构 Spring+Struts2=====

step5: (*) 将资费模块中所有 Action 定义到 Spring 容器 (step3 已开启组件扫描), 如果使用了 DAO, 采用注入方式使用 (此处是新建了 ListCostActionSpring 类, 与原来的 ListCostAction 相同)

```

@Controller      @Scope("prototype")//排版需要, 这里写一行了

```

```
public class ListCostActionSpring {    @Resource  
    private CostDAO costDAO;    ...    ...}
```

step6: 引入 struts-spring-plugin.jar 插件包

step7: (*) 修改 Action 的 struts 配置, 将 class 属性修改为容器中 Action 组件的 id 值

```
<action name="list" class="listCostActionSpring">  
    <result name="success">/WEB-INF/jsp/cost/cost_list.jsp</result>  
</action><!-- class 可写包名.类名, 即和以前的方式相同, 详情见 9.3 节-->
```

step8: 在 web.xml 中添加 ContextLoaderListener 配置, 用于实例化 Spring 容器, 详细说明见 8.8 案例 step9

```
<!-- 指定 Spring 配置文件位置和名称 -->  
<context-param><!--默认在 WEB-INF 下查找 applicationContext.xml 的, 不指定则报错-->  
    <param-name>contextConfigLocation</param-name><!--名字必须这么写 -->  
    <param-value>classpath:applicationContext.xml</param-value><!--classpath 表示  
src-->  
</context-param>  
<!-- 在服务器启动时, 实例化 Spring 容器对象 -->  
<listener>  
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>  
</listener>
```

step9: 测试 SSH, 发现修改资费时, 无法回显数据! 这是因为 Hibernate 中的 load 方法为延迟加载, 而 session 被早关闭, 所以 JSP 页面获取不到数据。为了支持 Hibernate 延迟加载的使用, 在 web.xml 中可以配置 Spring 提供的 OpenSessionInViewFilter。将 Session 关闭动作推迟到 JSP 解析之后。

step10: 在 web.xml 中添加 OpenSessionInViewFilter, 利用它控制 Session 关闭

```
<!-- 追加 OpenSessionInViewFilter, 不需要自己写了, Spring 提供了! 作用: 将 Template  
中的 session 关闭动作推迟到 jsp 解析之后 -->  
<filter><!--注意: 配置的 Filter 顺序, 要在 Struts 控制器 Filter 之前配置才能生效! -->  
    <filter-name>openSessionInViewFilter</filter-name>  
    <filter-class>  
        org.springframework.orm.hibernate3.support.OpenSessionInViewFilter  
    </filter-class>  
    </filter>  
    <filter-mapping>  
        <filter-name>openSessionInViewFilter</filter-name>  
        <url-pattern>/*</url-pattern>  
    </filter-mapping>
```

step11: 测试 SSH, 发现增、删、改会报错, 这是由于配置 OpenSessionInViewInterceptor 后产生的 (查询正常), 详细说明见下面的注意事项

```
org.springframework.dao.InvalidDataAccessApiUsageException:  
    Write operations are not allowed in read-only mode (FlushMode.NEVER/MANUAL):  
    Turn your Session into FlushMode.COMMIT/AUTO or remove 'readOnly' marker from  
    transaction definition.
```

step12: 在 applicationContext.xml 配置文件中添加 AOP 事务控制, 使用注解的方式, 代码及

详情见 10.3 节 2)

step13：分别给 Action 添加事务注解（可类定义前，也可 execute 或自定义方法前）

1) AddCostAction、DeleteCostAction、UpdateCostAction 中添加，事务类型详见 10.3 节

```
@Transactional(propagation=Propagation.REQUIRED)
```

2) DetailCostAction、ListCostActionSpring、ValidateCostNameAction 中添加

```
@Transactional(readOnly=true,propagation=Propagation.REQUIRED)
```

◆ 注意事项：

- ❖ 注意配置的 Filter 顺序！该 Filter 需要在 Struts 控制器 Filter 之前配置才能生效！！
- ❖ 当配置 OpenSessionInViewFilter 后，Spring 容器中 SessionFactory 组件的 id 值必须为 sessionFactory，不能再随便起了。
- ❖ 当配置 OpenSessionInViewFilter 后，会默认将 session 操作置成 readOnly 状态，此时需要添加 AOP 事务才能执行增、删、改，否则报错（查询没问题）。
- ❖ 由于 Hibernate 笔记 5.16 案例中用到了 5.17 节 4) step2 的自定义拦截器 OpenSessionInViewInterceptor，并按 step3 修改了 struts-cost.xml，此时会有冲突：资费模块的权限验证会失效（无权限的也能进入）。当项目为 SSH 结构时，用 Spring 提供的 OpenSessionInViewFilter 就行了，之前定义的 OpenSessionInViewInterceptor 不用了！把 struts-cost.xml 中自定义的拦截器的声明、引用、默认全局都删掉！包 cost 继承 netctoss-default，权限验证即可生效。其他配置也都继承 netctoss-default（之前为了方便测试所以都继承了 json-default）。

一百四十三、整合开发包 struts-spring-plugin.jar

9.1 Struts2 创建对象的方式

Struts2-->ObjectFactory-->StrutsObjectFactory

9.2 struts-spring-plugin.jar 创建对象的方式

在它的包中有个 StrutsSpringObjectFactory，而它的配置文件 struts-plugin.xml，则把 Struts2 中的 ObjectFactory 指定成了 StrutsSpringObjectFactory，这样一来，当 Struts2 请求过来时，Action 对象将交给整合包去 Spring 容器获取，即 Struts2 不再产生 Action 对象，可在该包的 struts-plugin.xml 中查看。

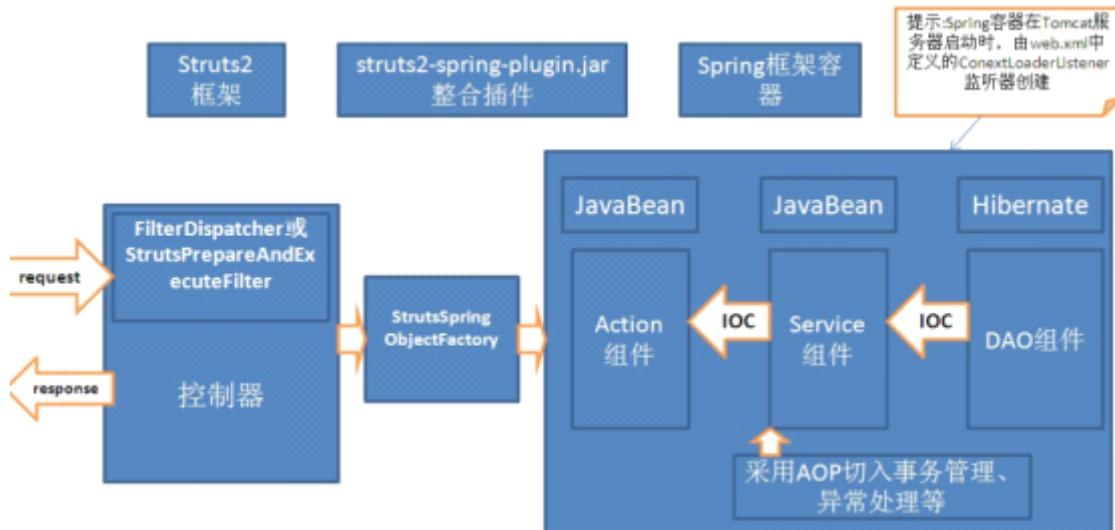
```
<bean type="com.opensymphony.xwork2.ObjectFactory" name="spring"  
      class="org.apache.struts2.spring.StrutsSpringObjectFactory" />  
<constant name="struts.objectFactory" value="spring" />
```

9.3 struts-spring-plugin.jar 的内部实现

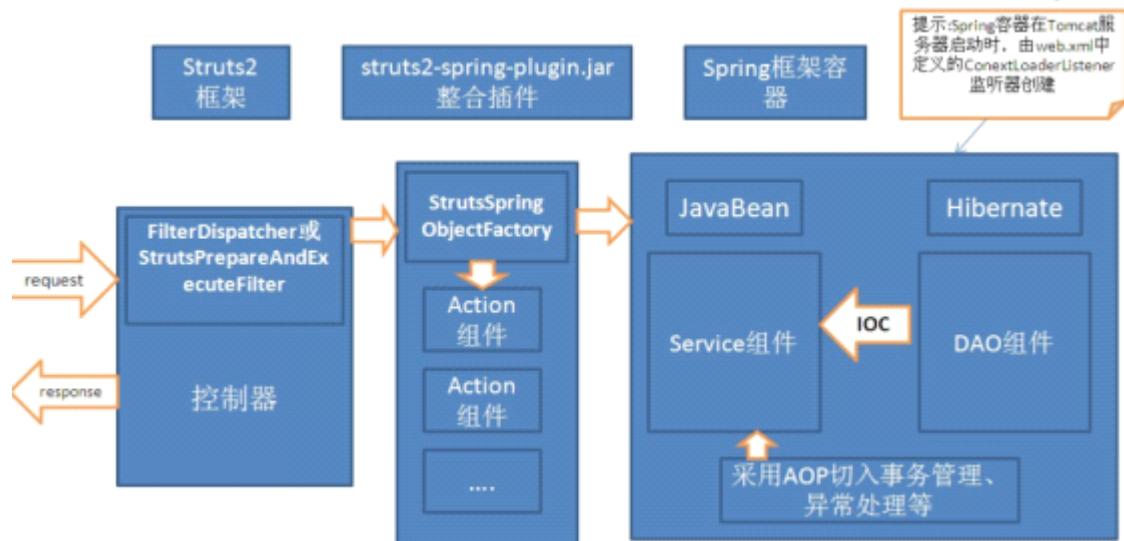
通过它的内部代码，可以发现该整合开发包获取 Bean 对象时，有两种方式：

```
try{//第一种 见原理图 1  
    Object action = ac.getBean(class 属性); //id 值  
}catch(){//第二种 见原理图 2  
    Class class = Class.forName(class 属性); //step1: 包名.类名, 利用反射机制生成 Action  
    Object action = class.newInstance();  
    //step2: 然后自动注入, 注入规则见 9.7 节!  
}
```

9.4 原理图 1



9.5 原理图 2



9.6 注意事项

第一种方式：创建 Action 由 Spring 容器负责。

第二种方式：创建 Action 由插件负责，与第一种相比 Action 脱离了 Spring 容器，所以不能用 AOP 机制了！也不能用事务管理了！所以，为了使 Action 只负责调用而不涉及业务逻辑，开发中一般会有个 Service 组件，把业务逻辑、AOP 机制、事务管理都给 Service 组件。

9.7 注入规则

9.3 节中 catch 块的注入规则：

- 1) 组件中，不加@Resource（默认）是按名称匹配，即属性名和 id 值一致才可以。
- 2) 组件中，如果添加了@Resource，则按类型匹配。

一百四十四、Spring 的事务管理

LICHOO

Spring 提供以下两种方式管理事务。

10.1 声明式事务管理（基于配置方式实现事务控制）

1) 以 8.9 案例为例，在 applicationContext.xml 配置文件中使用 xml 方式配置事务：

```
<!--事务管理配置-->
<!--定义事务管理 Bean (用于管理事务)，不用我们写了，直接用 Spring 提供的类-->
<bean id="txManager"
      class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <!-- 注入 session 资源 -->
    <property name="sessionFactory" ref="sessionFactory"></property>
</bean>
<!-- 定义方面和通知，直接使用 Spring 提供的命名空间： xmlns:tx=http://.../schema/tx -->
<tx:advice id="txAdvice" transaction-manager="txManager">
    <!--可以指定目标对象中不同方法采用不同的事务管理机制。注意：name 要根据目标对象中的方法名写。read-only="true"：只读事务，只能查询。propagation="REQUIRED"为默认值其他可取的值见 10.3 节 -->
    <tx:attributes><!--注释中的是给 DAO 添加事务，但不好！没注释的是给 Action 加-->
        <!--<tx:method name="save" propagation="REQUIRED"/>
        <tx:method name="update" propagation="REQUIRED"/>
        <tx:method name="delete" propagation="REQUIRED"/>
        <tx:method name="find*" read-only="true" propagation="REQUIRED"/>
        <tx:method name="get*" read-only="true" propagation="REQUIRED"/>
        <tx:method name="*" propagation="REQUIRED"/>  -->
    <!-- action 中 execute 方法都采取 REQUIRED，而 REQUIRED 把默认的只读操作模式改了，此时可增、删、改、查。由于 NetCTOSS 项目没加 Service 组件，所以把事务加到 Action 上，否则就是给 Service 加的。而不加在 DAO 中是因为有些操作可能要执行多个 DAO 共同完成一项功能，如果加在 DAO 中，那么里面的一个方法就是一个事务（会提交），那么当该功能中途出现问题，则之前的操作将无法回滚了！ -->
        <tx:method name="execute" propagation="REQUIRED"/>
        <!--其他没有考虑到的按默认的事务管理机制值-->
        <tx:method name="*" propagation="REQUIRED"/>
    </tx:attributes>
</tx:advice>
<!--定义切入点，AOP 切入--><!--proxy-target-class="true"：表示强制采用 CGLIB 方式生成代理类，若不写则容器会根据是否有接口，而去选择用哪种技术产生代理类，但这样就会有问题！如：Action 继承 BaseAction，而 BaseAction 实现某个接口，那么容器选择的代理类技术而产生的代理 Action 是没有原来 Action 中的方法的！若是作用在 DAO 上，则可不写 -->
<aop:config proxy-target-class="true"><!-- 类型匹配 -->
    <aop:pointcut id="actionPointcut" expression="within(com.tarena.netctoss.action..*)"/>
    <!-- 将切入点和通知结合 -->
```

```
<aop:advisor advice-ref="txAdvice" pointcut-ref="actionPointcut"/>
</aop:config>
```

2) 以 8.9 案例为例, 在 applicationContext.xml 配置文件中使用注解方式配置:

step1: 定义 HibernateTransactionManager (事务管理) Bean 组件

```
<bean id="txManager"
      class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <!-- 注入 session 资源 -->
    <property name="sessionFactory" ref="sessionFactory"></property>
</bean>
```

step2: 开启事务的注解配置

```
<!-- 开启事务注解配置 -->           <!-- 指明让 txManager 来管理事务 -->
<tx:annotation-driven proxy-target-class="true" transaction-manager="txManager"/>
```

step3: 然后在业务组件的类定义前或方法中使用@Transactional 注解即可, 例如:

①AddCostAction, 在类定义前使用 (对类中除了 get/set 方法外的所有方法, 都使用相同的事务管理)

```
@Transactional(propagation=Propagation.REQUIRED)
public class AddCostAction extends BaseAction{ ..... }
```

②DeleteCostAction, 在方法前使用

```
@Transactional(propagation=Propagation.REQUIRED)
public String execute() throws DAOException{ ..... }
```

③UpdateCostAction, 在类定义前使用

```
@Transactional(propagation=Propagation.REQUIRED)
public class UpdateCostAction extends BaseAction { ..... }
```

④ListCostAction, 在方法前使用

```
@Transactional(readOnly=true,propagation=Propagation.REQUIRED)
public String execute() throws Exception { ..... }
```

◆ 注意事项: 如果将 Action 当作目标, 需要在<tx:annotation-driven>添加 proxy-target-class="true"属性, 表示不管有没有接口, 都采用 CGLIB 方式生成代理类。

10.2 编程式事务管理(基于 Java 编程实现事务控制), 不推荐用!

主要是利用 transactionTemplate 的 execute()方法, 以回调方式将多个操作封装在一个事务中。

10.3 Spring 中常用的事务类型

- 1) REQUIRED: 支持当前事务, 如果当前没有事务, 就新建一个事务。这是最常见的选择, 也是默认值。
- 2) SUPPORTS: 支持当前事务, 如果当前没有事务, 就以非事务方式执行。
- 3) MANDATORY: 支持当前事务, 如果当前没有事务, 就抛出异常。
- 4) REQUIRES_NEW: 新建事务, 如果当前存在事务, 把当前事务挂起。
- 5) NOT_SUPPORTED: 以非事务方式执行操作, 如果当前存在事务, 就把当前事务挂起。
- 6) NEVER: 以非事务方式执行, 如果当前存在事务, 则抛出异常。
- 7) NESTED: 如果当前存在事务, 则在嵌套事务内执行。如果当前没有事务, 则进行

与 REQUIRED 类似操作。拥有多个可以回滚的保存点，内部回滚不会对外部事务产生影响。只对 DataSourceTransactionManager 有效。

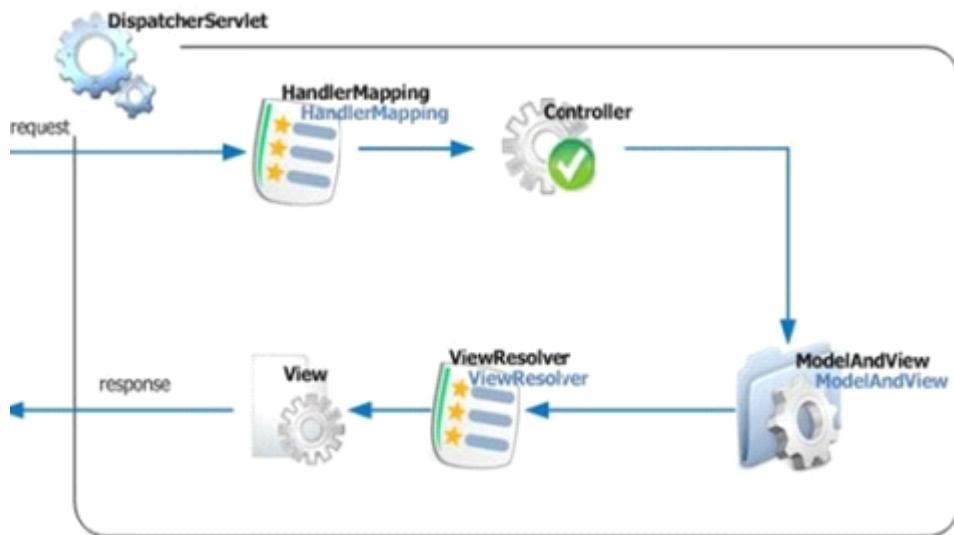
LICHOU

一百四十五、Spring 的 MVC

LICHOO

11.1 Spring MVC 的体系结构

- 1) 控制器 (两种): ①DispatcherServlet (等价于 Struts2 中的 Filter)
②Controller (等价于 Struts2 中的 Action)
- 2) 映射处理器: HandlerMapping (完成请求和 Controller 之间的调用, 等价于 Struts2 中的 ActionMapping)
- 3) 模型视图组件: ModelAndView (封装了模型数据和视图标识)
- 4) 视图解析器: ViewResolver (等价于 Struts2 中的 Result)
- 5) 视图组件: 主要用 JSP



11.2 Spring MVC 的工作流程

- 1) 客户端发送请求, 请求到达 DispatcherServlet 主控制器。
- 2) DispatcherServlet 控制器调用 HandlerMapping 处理。
- 3) HandlerMapping 负责维护请求和 Controller 组件对应关系。HandlerMapping 根据请求调用对应的 Controller 组件处理。
- 4) 执行 Controller 组件的业务处理, 需要访问数据库, 可以调用 DAO 等组件。
- 5) Controller 业务方法处理完毕后, 会返回一个 ModelAndView 对象。该组件封装了模型数据和视图标识。
- 6) Servlet 主控制器调用 ViewResolver 组件, 根据 ModelAndView 信息处理。定位视图资源, 生成视图响应信息。
- 7) 控制器将响应信息给用户输出。

11.3 案例: 简易登录 (基于 XML 配置, 不推荐使用)

由于此案例没有用到 JDBC、Hibernate 等访问数据库的技术, 所以 AOP 包可以不用导入!

- step1: 导入 spring-webmvc.jar 包
- step2: 导入 Spring 的 IoC 开发包 (spring.jar、commons-logging.jar)
- step3: 配置 web.xml

```
<servlet>
```

```

<servlet-name>springmvc</servlet-name>
<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
<init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>classpath:applicationContext.xml</param-value>
</init-param><!-- 让容器从指定的 src 目录下查找 applicationContext.xml 文件-->
</servlet>
<servlet-mapping>
    <servlet-name>springmvc</servlet-name>
    <url-pattern>*.do</url-pattern><!-- 不能写/*了，影响太广，自定义一种请求形式 -->
</servlet-mapping>

```

step4：在/WEB-INF/jsp 中新建 login.jsp 和 ok.jsp

1) login.jsp

```

<h1>Spring MVC 登录</h1><font size="5" color="red">${error}</font>
<form action="login.do" method="post">
    用户名: <input type="text" name="username" /><br/>
    密码: <input type="password" name="password" /><br/>
    <input type="submit" value="登录"/></form>

```

2) ok.jsp

```
<h1>欢迎${user}，登录成功！</h1>
```

step5：在 WebRoot 下新建 index.jsp

```
<a href="toLogin.do">spring mvc(xml)</a>
```

step6：在 org.tarena.controller 包中新建 ToLoginController 类，并实现 Controller 接口

```

public class ToLoginController implements Controller {//必须实现 Controller，并重写方法
    @Override //记得改改参数名字
    public ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws Exception {//默认执行的业务方法
        ModelAndView mv=new ModelAndView("login");//调用 login.jsp，指定视图名称
        return mv; }
    }
}

```

step7：在 org.tarena.controller 包中新建 LoginController 类，并实现 Controller 接口

```

public class LoginController implements Controller {//必须实现 Controller，并重写方法
    @Override //记得改改参数名字
    public ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws Exception {//默认执行的业务方法
        String name=request.getParameter("username");
        String pwd=request.getParameter("password");
        Map<String, Object> map=new HashMap<String, Object>();
        if("chang".equals(name) && "123".equals(pwd)){//简单模拟，不访问数据库了
            map.put("user", name); return new ModelAndView("ok",map); }
        map.put("error", "用户名或密码错误");
        return new ModelAndView("login",map);//指定视图名称 }
    }
}

```

step8：新建 applicationContext.xml 文件，并进行配置

```

<!-- 定义 handlermapping，即定义请求和 Controller 的映射信息 -->
<bean id="handlerMapping"

```

```

        class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
<property name="mappings">
    <props>
        <prop key="toLogin.do">toLoginController</prop>
        <prop key="login.do">loginController</prop>
    </props>
</property>
</bean>
<!-- 定义视图解析器，负责根据 ModelAndView 信息调用 View 组件 -->
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/"></property><!-- 声明前缀 -->
    <!-- 因返回的 ModelAndView 对象仅有个名字，所以要定义前后缀 -->
    <property name="suffix" value=".jsp"></property><!-- 声明后缀 -->
</bean>
<!-- 定义 Controller -->
<bean id="toLoginController" class="org.tarena.controller.ToLoginController"></bean>
<bean id="loginController" class="org.tarena.controller.LoginController"></bean>

```

step9：部署，测试

11.4 案例：修改 11.3 案例（基于注解配置，推荐使用）

由于此案例没有用到 JDBC、Hibernate 等访问数据库的技术，所以 AOP 包可以不用导入！

- step1：导入 spring-webmvc.jar 包
- step2：导入 Spring 的 IoC 开发包（spring.jar、commons-logging.jar）
- step3：配置 web.xml，与 9.3 案例 step3 相同
- step4：在/WEB-INF/jsp 中新建 login.jsp 和 ok.jsp

1) login.jsp

```

<h1>Spring MVC 登录</h1><font size="5" color="red">${error}</font>
<!-- 模拟请求有多级，即 user/login.do，action 写 user/login.do 为相对路径，出现叠加问题；写绝对路径需要加“/”，同时也要写应用名。详细说明见 10.2 节-->
<form action="/Spring06_MVC2/user/login.do" method="post">
    用户名：<input type="text" name="username" /><br/>
    密码：<input type="password" name="password" /><br/>
    <input type="submit" value="登录"/></form>

```

2) ok.jsp，与 9.3 案例 step4 中 2) 相同

step5：在 WebRoot 下新建 index.jsp

```
<a href="toLogin.do">spring mvc(annotation)</a>
```

step6：新建 applicationContext.xml 文件，并进行配置

```

<!-- 开启组件扫描 -->
<context:component-scan base-package="org.tarena" />
<!-- 定义映射处理器，采用注解 AnnotationMethodHandlerAdapter 指定映射 -->
<bean id="annotationMapping"
      class="org.springframework.web.servlet.mvc.annotation.AnnotationMethodHandlerAdapter">

```

```

    </bean>
    <!-- 定义视图解析器，负责根据 ModelAndView 信息调用 View 组件（JSP）-->
    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"></property><!-- 声明一个前缀 -->
        <property name="suffix" value=".jsp"></property><!-- 声明一个后缀 -->
    </bean>

```

step7：在 org.tarena.controller 包中新建 ToLoginController 类，并使用注解

```

@Controller //将组件扫描到 Spring 容器
@Scope("prototype")
public class ToLoginController {//不用再实现接口了，就是写个普通的类
    @RequestMapping(value="/toLogin.do",method=RequestMethod.GET)
    public String execute()//写个最简单的方法,返回类型也可写 ModelAndView
        return "login";//返回视图名称
}

```

step8：在 org.tarena.controller 包中新建 LoginController 类，并使用注解

```

@Controller //将组件扫描到 Spring 容器
@Scope("prototype")
@RequestMapping("/user/*")//当请求有多级，即有共同前缀，可写类定义前
public class LoginController {//不用去实现 Controller 接口了，就是写个普通的类
    //@RequestMapping(value="/login.do",method=RequestMethod.POST)//没有共同前缀
    @RequestMapping(value="login.do",method=RequestMethod.POST)//有共同前缀
    public String execute(User user,Model model){//Model 是可以传到下一个页面的
        Map<String, Object> map=new HashMap<String, Object>();
        if("chang".equals(user.getUsername()) && "123".equals(user.getPassword())){
            map.put("user", user.getUsername());
            model.addAttribute("user", user.getUsername()); return "ok";
        }
        model.addAttribute("error", "用户名或密码错误");
        return "login";
    }
}

```

◆ 注意事项：

- ❖ @RequestMapping 注解：value 属性指定请求； method 属性指定请求提交方式。
- ❖ Controller 中业务方法可以定义成以下格式：
 - 1) public String f1(){}
 - 2) public ModelAndView f1(){}
 - 3) public String f1(HttpServletRequest request){} //需要 request 就加上，不需要可不写
 - 4) public String f1(HttpServletRequest request, HttpServletResponse response){}
 - 5) public String f1(User user){} //自定义实体类，属性与表单中的提交名一致
 - 6) public String f1(Model model){} //org.springframework.ui.Model 中的
 - 7) public String f1(User user, Model model){}

step9：step8 中 execute 方法用到了实体 User，所以在 org.tarena.entity 包下创建 User 实体

```

private String username; private String password;//和表单提交名一样 .....get/set 方法

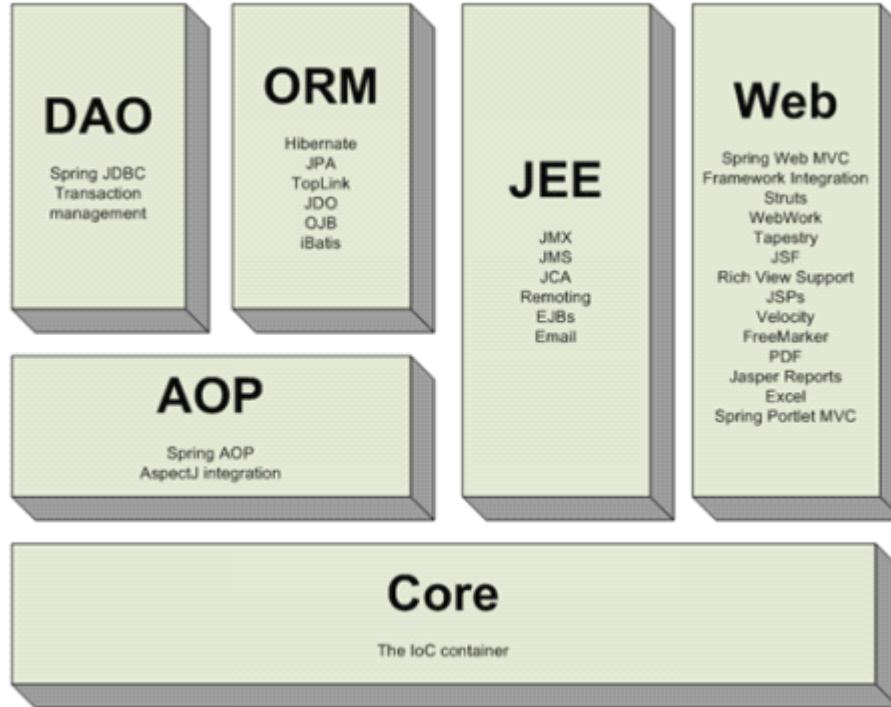
```

step10：部署，测试

一百四十六、其他注意事项

LICHOO

12.1 Spring 的核心模块



12.2 表单中 action 属性的相对、绝对路径问题

例如 9.4 案例 step4 中的 login.jsp，当请求有多级时（项目中不需要创建什么 user 目录，仅是通过请求实现的分类而已）：

<form>标签中的 action 属性，如果写“user/login.do”，即相对路径，会出现叠加问题：

```
<form action="user/login.do" method="post">
```

因为它相对的是当前请求地址，比如当前请求页面地址为：

```
http://localhost:8080/Spring06_MVC2/user/login.do
```

那么，第一次向 user/login.do 发请求没问题，但第二次发请求则出现叠加问题！（点两次登录就能出现该问题）

```
http://localhost:8080/Spring06_MVC2/user/user/login.do
```

即，第二次请求把后面的 login.do 又替换为了 user/login.do，而之前的 user 仍在，则出现叠加问题。

解决方式：写绝对路径，但要加“/”，同时也要写应用名，即：

```
<form action="/Spring06_MVC2/user/login.do" method="post">
```

12.3 用 SSH 重构 NetCTOSS 项目模块的步骤

- 1) 了解原功能的处理流程，例如资费模块：/cost/add.action-->AddCostAction.execute-->CostDAO.save-->list.action
- 2) 重构 CostDAO（Spring+Hibernate）
 - ①追加 Spring 开发包和配置文件。
 - ②追加 Cost 类（已存在）和 Cost.hbm.xml。

- ③在 Spring 的 sessionFactory 中加载 hbm.xml。
 - ④基于 HibernateDaoSupport 和 HibernateTemplate 编写 CostDAO 实现组件。
- 3) CostDAO 的 Spring 配置
- ①将 CostDAO 扫描到 Spring 容器。
 - ②将容器中 dataSource 注入到 sessionFactory, 然后 sessionFactory 注入到 DAO 组件。
 - ③测试 DAO。
- 4) 修改 AddCostAction
- ①将 Action 扫描到 Spring 容器。
 - ②采用注入方式使用 DAO 组件对象。
 - ③在业务方法上定义@Transactional 事务注解。
- 5) 修改 AddCostAction 的 struts 配置
- ①将 class 属性改成与扫描到 Spring 容器后的 Action 组件 id 值。
- 6) 检查共通的操作是否完成
- ①是否引入 struts-spring-plugin.jar。
 - ②是否在 web.xml 中添加 ContextLoaderListener。
 - ③是否开启了组件扫描配置。
 - ④是否开启了注解事务配置。
 - ⑤是否配置 dataSource、sessionFactory。