

JavaWeb

整个 javaweb 阶段技术介绍

刘银朋-Teacher

客户端(用户使用的)-浏览器、app、微信

服务器-web 核心

数据库-保存、获取数据的

客户端->发请求->服务端->操作数据->数据库->服务端(得到结果)->客户端(返回响应)

编写网页：

- html
- css
- JavaScript
- **jQuery**
- Bootstrap 框架
- **Ajax(重点)**

数据库：

- MySql
- Oracle
- Redis 缓存数据库

服务端：

1、和客户端交互的代码(重点)

- **Servlet**
 - **response**
 - 会话技术 **cookie/session**
 - **jsp/el/jstl**
 - 过滤器
-

2、操作数据库的代码

- JDBC
- **连接池技术**
- **JDBCTemplate**

工具：

- Tomcat 软件
- Linux 操作系统
- maven...

1-MySQL 单表上

1-1_数据库概述

1-1-1_什么是数据库

数据库：存储数据的仓库。是以结构的方式保存数据，可以快速的对保存的数据进行查询、修改、删除的操作。数据库的本质仍然是文件系统。在实际的 web 应用时，需要通过程序代码来对数据库里的数据进行增、删、改、查的操作。

1-1-2_有哪些常见的数据库

- Oracle: Oracle 公司收费的大型数据库软件。通常用于大型系统
- SQLServer: Microsoft 公司的收费的大型数据库软件。通常是.net 的系统使用 SQLServer。 c#
- DB2: IBM 公司的大型的收费数据库软件。银行、金融项目使用 DB2 比较多
- Sybase: 目前已经逐步退出历史舞台，但是提供了一个强大的数据库建模工具 PowerDesigner
- MySql: Oracle 公司的小型收费/免费的数据库软件，通常小型项目会使用 MySql

1-1-3 MySql 安装与卸载

- 安装
- 卸载

1-1-4 MySql 服务管理以及登录

通过 Windows 的服务管理来操作 MySql 服务

windows+R-->输入: services.msc-->找到 MySQL 服务

右键-->启动: 开启 MySql 服务，开启之后就可以登录 MySql，访问操作 MySql 了

右键-->停止: 关闭 MySql 服务

在（管理员运行） cmd 里使用命令来启动/关闭 MySql 服务

开启: **net start mysql**

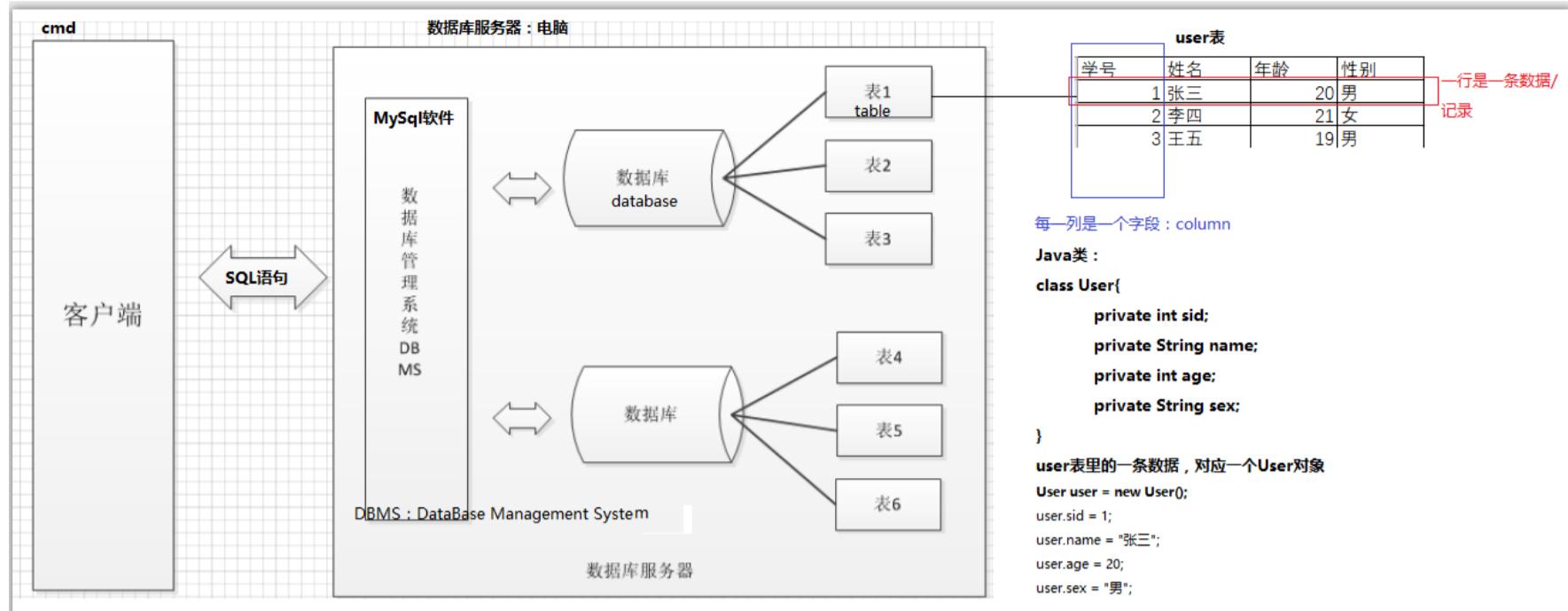
关闭: **net stop mysql**

在 cmd 窗口里输入命令进行登录:

mysql -u 用户名 -p 密码 [-h MySql 的 ip 地址 -P MySql 的端口]

退出: **exit/quit**

1-1-5 MySql 的结构



1-2_SQL 语言

1-2-1_什么是 SQL 语言

SQL: structure query language, 结构化查询语言。是数据库提供的，用来操作数据库的语法。

标准 SQL 和 SQL 方言之分：

标准 SQL: 标签的 SQL 规范

SQL 方言: 不同数据库有一些个性化的内容，操作方式也有一些区别

1-2-2_按照 SQL 的功能划分的类(面试题)

- DDL: Data Definition Language, 数据定义语言，主要是用来定义 database、table、column。常见的: create、alter、drop 等等
- DML: Data Manipulation Language, 数据操作语言，主要是用来插入、修改、删除数据/记录。常见的: insert、update、delete
- DQL: Data Query Language, 数据查询语言，主要是用来查询数据的。常见: select
- DCL: Data Controller Language, 数据库控制语言，主要是用来管理数据库用户、权限等等，通常是 DBA 来使用的 (DataBase Administrator 数据库管理员)。常见的: grant, revoke

1-3_DDL-操作 database

1-3-1_创建 database

语法: **create database** 数据库名;

create database user;

1-3-2_查看 database

列出所有数据库：

语法: **show databases;**

查看某个数据库的详细创建信息

语法: **show create database 数据库名;**

1-3-3_切换 database

语法: **use 数据库名;**

查看在哪个库里: **select database();**

1-3-4_删除 database

语法: **drop database 数据库名**

drop database user;

1-4_DDL-操作 table

1-4-1_MySQL 的数据类型

- **int:** 整型。对应 Java 的 int
- **double(m,d):** 双精度小数, m 表示数字的总位数,d 表示小数位。**double(5,2)**最大范围: 999.99。对应 Java 的 double
- **char(n):** 固定长度字符串, 最大可以保存 n 位。**char(5)**类型, 保存” abc”, 存储的是” abc ”。对应 Java 的 String
- **varchar(n):** 可变长度字符串, 最大可以保存 n 位。**varchar(5)**类型, 保存 “abc”, 存储的就是” abc ”。对应 Java 的 String
- **date:** 日期类型。对应 Java 的 java.sql.Date
- **datetime:** 日期时间类型 **yyyy-MM-dd HH:mm:ss**。可以保存 1000~9999 年的范围, 对应的 Java 的 java.sql.Timestamp, java.sql.Date。
- **timestamp:** 时间戳类型 **yyyy-MM-dd HH:mm:ss**。可以保存 1970~2038 年的范围。对应的 Java 类型是 java.sql.Timestamp

1-4-2_创建 table

语法:

create table 表名称(

```
字段名 字段类型 [约束],  
.....  
字段名 字段类型 [约束]  
);
```

1-4-3_查看 table

- 查看所有表: **show tables;**
- 查看表结构: **desc 表名称;**
- 查看表创建信息: **show create table 表名称;**

1-4-4_修改 table

- 重命名表: **rename table 表名称 to 新名称;**
- 添加字段: **alter table 表名称 add 字段名 字段类型 [约束];**
- 修改字段名称: **alter table 表名称 change 字段名 新字段名 字段类型 [约束];**
`alter table users change address addr varchar(50);`
- 修改字段类型: **alter table 表名称 modify 字段名 字段类型 [约束];**
`alter table users modify addr varchar(100);`
- 删除字段: **alter table 表名称 drop 字段名;**
`alter table users drop addr;`

1-4-5_删除 table

```
语法: drop table 表名称;
```

1-5_DML-操作数据

1-5-1_插入数据

```
语法: insert into 表名称 (字段 1,字段 2,...) values (值 1,值 2,...);
```

注意:

前边有几个字段, 后边就要有几个值

插入的值必须要符合字段类型的精度要求。比如有字段 `varchar(5)`, 插入值: 黑马程序员 41 期
值可以字符串类型, 插入任意数据

```
语法: insert into 表名称 values (值 1,值 2,...);
```

是给表里所有字段插入数据。插入数据时需要注意：值的顺序必须和字段的顺序一样

1-5-2_修改数据

语法：**update 表名称 set 字段 1=值 1, 字段 2=值 2,...** [where 条件];

update 表名称 set 字段 1=值 1, 字段 2=值 2,... where id = 1;

1-5-3_删除数据

语法：**delete from 表名称 [where 条件];**数据库底层会一条一条删，效率低，但是可以加 where 条件

语法：**truncate 表名称;**数据库会把整个表数据一次性清除掉，效率高，但是不能加 where 条件

1-6_DQL-查询数据

1-6-1_基本查询

- 查询全部：**select * from 表名称;**
- 查询指定列：**select 字段 1, 字段 2, ... from 表名称;**
- 查询并运算：**select 字段 1 + 值, 字段 2 - 值, 字段 3 * 值, ... from 表名称;**
- **查询并空值处理：****select ifnull(字段, 为空的值) + 值, 字段 2, 字段 3, ... from 表名称;**
- 查询并起别名：**select ifnull(字段, 为空的值) + 值 as 别名 1, 字段 2 别名 2 from 表名称;**
- **查询并去重：****select distinct 字段 1, 字段 2, ... from 表名称;**

说明：**distinct** 后边所有字段的值都一样，才是重复值，需要去掉的

1-6-2_条件查询

语法：基本查询 **where** 条件

条件：

比较：**>, <, >=, <=, =, <>**

范围：**字段名 between ... and ...** 包含头和尾

集合：**字段名 in (值 1, 值 2, ...)**

模糊查询：**字段名 like ‘石%’** : %表示任意个任意字符；_表示一个任意字符

多条件连接

and : 多条件同时生效

or : 多条件中任意一个条件符合即可

not(条件): 排除掉符合这个条件的

1-7_插入数据时，乱码问题

解决方案一：先执行命令 **set names gbk;** 然后再操作就可以正常插入中文数据了

只能临时解决，针对当前这一次连接有效。重新登录之后就又恢复到原本的乱码状态了

永久解决方案：

找到 MySql 安装目录里的 my.ini

搜索[mysql]下边的 default-character-set=utf8 改成 gbk

保存文件

重新启动 MySql 的服务

2-Mysql 单表下

2-1_单表

2-1-1_DQL 查询

1、排序查询(重点)

查询出来数据之后，按照我们指定的顺序显示

语法： **order by** 排序字段 1 排序规则 1, 排序字段 2 排序规则 2,...

排序字段：想按照哪个字段排序

排序规则：

升序： **ASC**, 默认排序规则

降序： **DESC**

示例 1：查询所有的员工信息，按照年龄从小到大排序

select * from employee order by age asc

示例 2：查询所有员工信息，按照年龄从小到大排序；如果年龄一样，按照工资从高到低排序

select * from employee order by age asc, salary desc

2、聚合函数

针对某一列的所有数据进行统计，叫聚合统计。 MySql 提供了一些聚合函数来实现聚合统计，

使用方法：**select 聚合函数 from 表名称 [where 条件]**

常用的有：

统计个数：**count(*)**

统计 employee 表里的总数：**select count(*) from employee**

统计 employee 里男员工的数量：**select count(*) from employee where gender = '男'**

求和统计：**sum(字段)**

统计给所有员工发的工资总和: `select sum(salary) from employee`

求平均值: `avg(字段)`

统计所有员工的平均工资: `select avg(salary) from employee`

求最大值: `max(字段)`

查询所有员工的最高工资: `select max(salary) from employee`

求最小值: `min(字段)`

查询所有员工的最低工资: `select min(salary) from employee`

注意: 聚合函数会忽略 `null` 值

3、分组查询

查询结果并进行分组统计。

语法: `select 分组字段, 聚合函数 1, 聚合函数 2, ... from 表名称 [where 条件] group by 分组字段`

查询并统计所有员工信息: 按部门统计

`select dept,count(*), max(salary), min(salary), sum(salary), avg(salary) from employee group by dept`

查询并统计所有男性员工信息: 按照部门统计

`select dept,count(*), max(salary), min(salary), sum(salary), avg(salary) from employee where gender = '男' group by dept`

语法: `select 分组字段, 聚合函数 1, 聚合函数 2, ... from 表名称 group by 分组字段 having count(*) > 5`

having: 是对分组后的结果进行过滤的, 而不是原始表数据过滤

查询并统计所有员工信息: 按部门统计, 只要部门人数大于 5 的数据

`select dept,count(*), max(salary), min(salary), sum(salary), avg(salary) from employee group by dept having count(*) > 5`

注意:

1. 在分组查询里 `select` 可以查询的字段: 只有分组字段, 以及聚合函数

2. `where` 条件只能对原始表进行过滤; 而 `having` 是对分组后的数据进行过滤

过滤的对象不同: `where` 过滤原始表数据; `having` 过滤分组后的数据

过滤的时机不同: `where` 先执行; 分组后才会有 `having` 过滤执行

过滤的条件不同: `where` 对表字段的过滤; `having` 对分组后结果列进行过滤

4、分页查询(重点)

分页: 主要是指在程序开发过程中, 需要在页面上显示数据。如果数据过多, 就需要分页显示。

比如: 每页显示 10 条, 第 1 页时就要显示第 1 个 10 条数据; 第 2 页就要显示第 2 个 10 条数据;

语法: `limit index, size`

`index:` 表示从哪个索引开始查询数据

size: 表示要查询几条数据

示例：每页 10 条，显示第 1 个 10 条： limit 0, 10

```
select * from employee limit 0, 10
```

注意：分页必须要写在 SQL 的最后

5、综合查询的语法

```
select *|字段 1… from 表名称 where 条件 group by 分组字段 having 分组过滤 order by 排序字段 排序规则 limit index, size
```

2-1-2_数据库的约束(理解)

1、什么是约束

约束：对数据库里的数据进行限制，必须要符合约束的要求，才可以正常的插入、修改、删除数据

2、主键约束

主键

主键：在创建表的时候，要求每张表都要有且只有一个主键字段；是数据的唯一性标识

主键约束：设置为主键的字段，要求值必须是非空唯一的

语法： primary key

创建表时设置主键：

```
create table student
```

主键自增

主键自增策略：如果主键是数字类型的话，可以设置主键为自增。我们在插入数据时，就不需要指定主键值，数据库会自动生成不重复的主键值。

语法： **primary key auto_increment**

主键自增示例：

```
CREATE TABLE student(
```

```
    sid INT PRIMARY KEY AUTO_INCREMENT,  
    sname VARCHAR(20) NOT NULL,  
    semail VARCHAR(50) UNIQUE,  
    sage INT DEFAULT 80  
);
```

#插入数据： 主键自增

```
INSERT INTO student (sid, sname, semail, sage) VALUES (NULL, '马超', 'machao@163.com', 20);
```

```
INSERT INTO student (sname, semail, sage) VALUES ('白江', 'baijiang@163.com', 20);
```

3、唯一性约束

唯一性约束：添加了唯一性约束的字段，要求值必须是唯一不重复的
语法：unique

4、非空约束

非空约束：添加了非空约束的字段，要求值不能为 null
语法：not null

5、默认值约束

默认值约束：添加了默认值约束的字段，如果没有设置值，就会使用默认值
语法：default 默认值

6、约束的练习

```
CREATE TABLE student(
    sid INT PRIMARY KEY,
    sname VARCHAR(20) NOT NULL,
    semail VARCHAR(50) UNIQUE,
    sage INT DEFAULT 80
);

#插入数据：符合所有约束的要求，可以插入成功
INSERT INTO student (sid, sname, semail, sage) VALUES (1, '马超', 'machao@163.com', 20);
#插入数据：违反主键要求 唯一非空
INSERT INTO student (sid, sname, semail, sage) VALUES (NULL, '白江', 'baijiang@163.com', 20);
INSERT INTO student (sid, sname, semail, sage) VALUES (1, '白江', 'baijiang@163.com', 20);
#插入数据：违反唯一性约束
INSERT INTO student (sid, sname, semail, sage) VALUES (2, '白江', 'machao@163.com', 20);
#插入数据：违反非空约束
INSERT INTO student (sid, sname, semail, sage) VALUES (2, NULL, 'baijiang@163.com', 20);
#插入数据：违反默认值约束。设置了默认值的字段不插入值，会使用默认值
INSERT INTO student (sid, sname, semail, sage) VALUES (2, '马超', 'baijiang@163.com', NULL);
INSERT INTO student (sid, sname, semail) VALUES (3, '张锟', 'zhangkun@163.com');
```

2-1-3 数据的备份与恢复

1、使用 SQLyog 备份与恢复

2、使用 cmd 备份与恢复

```
备份: mysqldump -u 用户名 -p 密码 -h ip 地址 -P 端口 要备份的库名称>E:\backup.sql
```

```
恢复: mysql -u 用户名 -p 密码 -h ip 地址 -P 端口 恢复到哪个库名称<E:\backup.sql
```

注意: 执行恢复命令时, 必须要保证所恢复的库已经创建好了

2-2 多表

2-2-1 多表场景

有商品表如下:

商品编号 pid	商品名称 pname	商品价格 price	库存数量 pcount	所属分类 category
1	小米 8	2599	100	手机数码
2	华为 P20	3000	100	手机数码
3	格力空调	2699	20	大型家电
4	小米电视	4999	20	大型家电

存在问题:

商品信息和分类信息耦合到一起了。

假如要增加一个分类, 但是还没有加商品---添加异常;

假如要删除一个分类手机数码, 需要所相关的所有商品也删除掉---删除异常

假如要把大型家电修改成家用电器, 需要修改库里 n 多条数据---修改异常

解决方案: 拆成两张表--商品表和分类表

拆成商品表和分类表如下:

商品编号 pid	商品名称 pname	商品价格 price	库存数量 pcount	所属分类编号 cid (外键)
1	小米 8	2599	100	1
2	华为 P20	3000	100	1
3	格力空调	2699	20	2
4	小米电视	4999	20	2

分类编号 cid (主键)	分类名称 cname
---------------	------------

1	手机数码
2	大型家电

解决的问题:

要增加一个分类，直接在分类表里添加即可

要删除一个分类，直接在分类表里删除数据即可

要修改一个分类，直接修改分类表的数据：大型家电 --> 家用电器

存在的问题:

按照正常的逻辑要求，商品表的 **cid** 值必须是从分类表的 **cid** 取值。但是，如果有以下操作，会影响数据的完整性：

“手机数码”分类在商品表有数据，但是我可以在分类表直接把“手机数码”删除掉。商品表的数据就会找不到分类，数据不完整了，脏数据

要新增商品“登山鞋”，所属分类 3，但是 3 在分类表里没有数据。那么添加的就是不完整的数据，脏数据

问题的根源:

两张表之间缺乏数据的完整性约束，需要约束：商品表里的 **cid**，值必须是来自于分类表的 **cid**。

如果有这个约束存在，数据库帮我们自动检查数据是否符合这个约束，就能够避免脏数据。

如果数据库已经加了约束:

要新增商品“登山鞋”，分类是 3。数据库就会帮我们去分类表里检查，3 对应的分类是否存在；不存在的话不允许插入

要删除分类“手机数码”。假如数据库帮我们去商品表检查是否有对应的数据存在；如果有数据存在，不允许删除

就能够避免脏数据，多表的数据能保证完整性了。

2-2-2_外键与外键约束

1、主表和从表

从表的数据，来自于主表。就可以说从表依赖于主表。

比如：商品表里的 **cid** 数据，来自于分类表的主键 **cid**。商品表是分类表的从表

2、外键和外键约束

外键：从表里某个字段，取值来自于主表的主键。那么这个字段就是外键

比如：从表 **product** 里的 **cid**，来自于主表 **category** 的主键 **cid**，那么：**product** 表的 **cid** 就是外键

外键约束：设置为外键的字段，要求值必须来自于主表的主键的数据

作用：保证多表数据的一致性和完整性，避免出现脏数据

语法：

在创建表的时候增加外键：

```
#先创建主表: category
```

```

CREATE TABLE category (
    cid INT PRIMARY KEY AUTO_INCREMENT,
    cname VARCHAR(20) NOT NULL UNIQUE
);

#再创建从表: product
CREATE TABLE product(
    pid INT PRIMARY KEY AUTO_INCREMENT,
    pname VARCHAR(50),
    price DOUBLE(9,2),
    pcount INT,
    cid INT,
    #增加外键设置
    CONSTRAINT pro_cat_fk1 FOREIGN KEY(cid) REFERENCES category(cid)
);
#准备数据
INSERT INTO category(cname) VALUES ('手机数码');
INSERT INTO category(cname) VALUES ('大型家电');

INSERT INTO product (pname, price, pcount, cid) VALUES ('小米 8', 2599, 100, 1);
INSERT INTO product (pname, price, pcount, cid) VALUES ('华为 P20', 2599, 100, 1);
INSERT INTO product (pname, price, pcount, cid) VALUES ('格力空调', 2599, 100, 2);
INSERT INTO product (pname, price, pcount, cid) VALUES ('小米电视', 2599, 100, 2);
#外键约束的验证
#验证外键的作用：保证数据的一致性和完整性
#删除主表数据，在从表 里有对应的数据存在，是否能删除？ cid 为 1 的分类，在 product 表里有数据
DELETE FROM category WHERE cid = 1;

#修改主表数据，把 cid 为 1 的值改成 cid 为 5
UPDATE category SET cid = 5 WHERE cid = 2;

#增加从表数据，但是外键值在主表中不存在，是否能增加？ 增加一个商品，分类是 3
INSERT INTO product (pname, price, pcount, cid) VALUES ('登山鞋', 599, 100, 3);

```

3、外键的级联操作

因为有外键约束的存在，导致：删除主表数据可能失败、修改主表数据可能失败。

所以需要给外键约束增加级联的操作：删除主表数据时，一并把相关的从表数据删除掉；修改主表数据时，一并把相关的从表 数据修改掉。

语法：constraint 约束名称 foreign key(外键字段) references 主表(主键) on delete cascade on update cascade

级联操作的验证

#先创建主表：category

```
CREATE TABLE category (
    cid INT PRIMARY KEY AUTO_INCREMENT,
    cname VARCHAR(20) NOT NULL UNIQUE
);
```

#再创建从表：product

```
CREATE TABLE product(
    pid INT PRIMARY KEY AUTO_INCREMENT,
    pname VARCHAR(50),
    price DOUBLE(9,2),
    pcount INT,
    cid INT,
    #增加外键设置
```

```
CONSTRAINT pro_cat_fk1 FOREIGN KEY(cid) REFERENCES category(cid) ON DELETE CASCADE ON UPDATE CASCADE
```

```
);
```

#准备数据

```
INSERT INTO category(cname) VALUES ('手机数码');
```

```
INSERT INTO category(cname) VALUES ('大型家电');
```

```
INSERT INTO product (pname, price, pcount, cid) VALUES ('小米 8', 2599, 100, 1);
```

```
INSERT INTO product (pname, price, pcount, cid) VALUES ('华为 P20', 2599, 100, 1);
```

```
INSERT INTO product (pname, price, pcount, cid) VALUES ('格力空调', 2599, 100, 2);
```

```
INSERT INTO product (pname, price, pcount, cid) VALUES ('小米电视', 2599, 100, 2);
```

#删除主表中 cid 为 1 的数据，会把 product 表中分类为 1 的数据一并删除掉

```
DELETE FROM category WHERE cid = 1;
```

#修改主表数据，把 cid 为 2 的值改成 cid 为 5，会把 product 表中原本分类是 2 的值一并修改成 5

```
UPDATE category SET cid = 5 WHERE cid = 2;
```

2-2-3_多表与多表关系

1、多表之间的关系

一对一是：

一对多：分类表和商品表、用户表和订单表

多对多：学生和学科、订单表和商品表

2、一对一(了解)

一对一关系的表不常见，因为通常可以合并成一张表。但是有些特殊的场景可能需要拆表：

出现业务考虑，要拆表：用户帐户信息表，可以拆成用户表和帐户表，提供不同的业务处理的功能

出现效率考虑，要拆表：表有 100 个字段，常用的只有 20 个字段。可以拆成两张表，常用的数据一张表，不常用的数据另外一张表。

3、一对多

比如：分类表和商品表、用户表和订单表

建表原则：在多的一方（从表一方）增加外键，指向一的一方（主表）的主键

例如，创建用户表和订单表

#一对多建表。一的一方是用户表（主表），多的一方是订单表（从表）

#先创建主表

```
CREATE TABLE `user` (
    uid INT PRIMARY KEY AUTO_INCREMENT,
    uname VARCHAR(20)
);
```

#再创建从表

```
CREATE TABLE orders (
    oid INT PRIMARY KEY AUTO_INCREMENT,
    ordertime DATETIME,
    total DOUBLE(9,2),
    uid INT,
    #一对多建表时，需要在从表增加外键约束，指向主表的主键
    CONSTRAINT orders_user_fk1 FOREIGN KEY(uid) REFERENCES `user`(uid)
);
```

4、多对多

比如：学生和学科、订单和商品

建表原则：建立一张中间关系表，来维护多对多的关系。

建表示例：订单表和商品表

#多对多建表：商品和订单表

#先创建订单表

```
CREATE TABLE orders (
```

```
    oid INT PRIMARY KEY AUTO_INCREMENT,
```

```
    ordertime DATETIME,
```

```
    total DOUBLE(9,2)
```

```
);
```

#再创建商品表

```
CREATE TABLE product(
```

```
    pid INT PRIMARY KEY AUTO_INCREMENT,
```

```
    pname VARCHAR(50),
```

```
    price DOUBLE(9,2),
```

```
    pcount INT
```

```
);
```

#最后创建中间关系表

```
CREATE TABLE product_order(
```

```
    poid INT PRIMARY KEY AUTO_INCREMENT,
```

```
    oid INT,
```

```
    pid INT,
```

```
    CONSTRAINT po_fk1 FOREIGN KEY(oid) REFERENCES orders(oid),
```

```
    CONSTRAINT po_fk2 FOREIGN KEY(pid) REFERENCES product(pid)
```

```
);
```

2-2-4 权限经典五张表

权限经典五张表

用户表

张三
李四
王五
赵六
小七
老八

用户-角色关系表
id 用户id 角色id

角色表

研发
项目经理
CEO

角色-资源关系表
id 角色id 资源id

资源表

查看个人工资
请假
审批假条
查看项目合同信息

3-MySQL 多表、事务和 DCL

3-1_复习

3-1-1_单表部分

1、DQL-排序查询

```
order by 排序字段 排序规则, 排序字段 排序规则  
排序规则: ASC 升序; DESC 降序
```

2、DQL-聚合统计

```
count(*) 统计个数  
sum(字段) 求和统计  
avg(字段) 求平均值  
max(字段) 求最大值  
min(字段) 求最小值  
注意: 聚合函数会忽略 null 值
```

3、DQL-分组查询

```
group by 分组字段 having 分组后过滤条件
```

4、DQL-分页查询

```
limit index, size  
index: 从哪个索引开始查询  
size: 查询几条数据  
注意: 分页的 limit 必须要写在 SQL 的最后
```

5、约束

```
主键约束: primary key  
要求主键字段唯一非空。  
如果主键是数字类型, 可以设置成为自增: primary key auto_increment  
唯一性约束: unique  
要求字段不能重复  
非空约束: not null  
要求字段不能为空  
默认值约束: default 默认值
```

如果字段没有设置值，就取默认值
外键约束

3-1-2_多表部分

1、外键和外键约束

语法: `constraint 约束名称 foreign key (外键字段) references 主表(主键字段)`
作用: 保证多表之间数据的一致性和完整性，避免出现脏数据

2、多表和多表关系

一对一

学生和身份证。通常一对一的表可以合并成一张表。

一对多

比如: 用户和订单， 教室和学生

建表原则: 在从表上 (多的一方) 建立一个外键，指向主表 (一的一方) 的主键字段

多对多

比如: 订单和商品、学生和课程

建表原则: 需要建立一个中间关系表，来维护多对多的关系

3-1-3_数据库范式 (了解)

1、什么是数据库的范式(NF)

范式 NF: Normal Form 在构建数据库的时候，如果要构造一个比较科学的规范的数据库，所需要遵循的规则和规范。
如果在创建数据库的时候，范式和业务需求有冲突，或者和性能有冲突: 业务需求 > 性能 > 范式

2、有哪些数据库的范式

有六种数据库范式: 第一范式(1NF)、第二范式(2NF)、第三范式(3NF)、巴斯-科德范式(BCNF)、第四范式(4NF)、第五范式(5NF， 完美范式)。

如果在创建数据库的时候，要遵循的最基本的要求，就是 1NF。在 1NF 基础上满足更多要求，就是 2NF。以此类推。

但是在实际创建数据库的时候，只要到 3NF 即可

3、三大范式

1NF

是最基本的要求: 要求表里所有字段是不可分割的。

2NF

在 1NF 基础上，要求所有的列都要完全依赖于主属性。

主要指：主属性是由多字段组合成的情况

3NF

在 2NF 基础上，要求所有的列都要直接依赖于主属性

依赖传递：A->B->C 就存在依赖传递关系 A->C

3-2_多表查询（重点）

3-2-1_什么是多表查询

在建表的时候，存在拆分表的情况。把数据拆分到多张表里保存，数据之间是有关系的。

拆分成多表之后，要查询数据，就需要从这多张表中查询有关联的数据了。

多表查询的关键，在于：把多表关联起来，关联的时候一定要有关联条件。

笛卡尔积：多表关联的时候，没有写关联条件，就得到了无意义的多表数据的排列组合。

在多表查询中一定要避免笛卡尔积的出现：一定要有关联条件

3-2-2_内连接查询

从多表中查询一定有关联的数据

隐式内连接查询：

语法：select * from 表 1, 表 2 where 关联条件

显式内连接查询

语法：select * from 表 1 inner join 表 2 on 关联条件

3-2-3_外连接查询

查询一张表所有数据，以及另外一张表有关联的数据。左外连接和右外连接本质完全相同，仅仅是方向相反

左外连接：查询左表所有数据，以及右表相关联的数据

语法：select * from 左表 left [outer] join 右表 on 关联条件

右外连接：查询右表所有数据，以及左表相关联的数据

语法：select * from 左表 right [outer] join 右表 on 关联条件

3-2-4_子查询

子查询：两层查询嵌套，没有固定语法，只是一种查询技巧

单行单列子查询-----子查询的结果是一个值

#查询 oid 为 1 的用户信息

#第一步：查询 oid 为 1 的 uid 的值

SELECT uid FROM orders WHERE oid = 1;

#第二步：查询 user 表里 uid 为上一步查询结果值的用户信息

```
SELECT * FROM USER WHERE uid = 1;  
#合成一条 SQL:  
SELECT * FROM USER WHERE uid = (SELECT uid FROM orders WHERE oid = 1);  
多行单列子查询----子查询结果是一个集合  
#查询 2018 年下过订单的用户信息  
#第一步：查询 2018 年下过的订单信息  
SELECT uid FROM orders WHERE ordertime BETWEEN '2018-01-01' AND '2018-12-31'  
#第二步：查询 user 表里 uid 是刚刚查询的结果的用户信息  
SELECT * FROM USER WHERE uid IN(1, 2)  
#合成一条 SQL  
SELECT * FROM USER WHERE uid IN(SELECT uid FROM orders WHERE ordertime BETWEEN  
'2018-01-01' AND '2018-12-31')  
多行多列子查询----子查询结果是一张虚拟表。拿虚拟表和其它表关联查询  
#查询 2018 年下过订单的用户信息，以及订单信息  
SELECT * FROM USER u, orders o WHERE u.uid = o.uid AND o.ordertime BETWEEN '2018-01-01'  
AND '2018-12-31';  
  
#第一步：查询 2018 年下过的所有订单信息  
SELECT * FROM orders WHERE ordertime BETWEEN '2018-01-01' AND '2018-12-31'  
#第二步：把 2018 年的订单信息（虚拟表） 和 用户表进行关联查询  
SELECT * FROM USER u, (SELECT * FROM orders WHERE ordertime BETWEEN '2018-01-01'  
AND '2018-12-31') t WHERE u.uid = t.uid
```

3-3 MySql 的事务（理解）

3-3-1 什么是事务

事务：是数据库的概念，逻辑上的一个操作需要由多个步骤共同完成。组成一个事务的多个操作单元，要么全部成功，要么全部失败。

事务的作用：保证组成事务的多个 SQL 操作，要么全部成功，要么全部失败

使用事务的场景：多条数据变更 SQL，要求一起成功的时候，才需要使用到事务

事务的经典场景：**银行转账业务**，需要保证两步要么全部成功，要么全部失败。如果成功一半，要能够撤销已经执行的 SQL。

事务的使用步骤：

开启事务：

第一步：转账人扣钱操作，执行一条 SQL ----数据变更不会立即保存数据库，而是被 MySql 暂时保存起来

第二步：收款人加钱操作，执行一条 SQL ----数据变更不会立即保存数据库，而是被 MySql 暂时保存起来

关闭事务：

提交事务：开启事务之后执行的所有 SQL 立即全部生效，真正的保存到数据库里去

回滚事务：开启事务之后执行的所有 SQL 全部撤消，所有的数据变更不会生效，回到事务开启之前的状态

3-3-2 MySql 的事务管理

1、默认管理方式

MySql 默认是自动提交的。有一个变量”**autocommit**”值默认是 1，是开启状态的。

如果把这个开关关闭掉，再执行多条 SQL 的数据变更就不会立即生效了，而是被暂时保存起来了。

查看自动提交的状态：**select @@autocommit;** ----1：开启状态；0：关闭状态

关闭自动提交：**set autocommit = 0;**

默认管理方式的步骤：

执行 SQL1 -----数据变更不会立即生效

执行 SQL2 -----数据变更不会立即生效

提交事务 **commit**/回滚事务 **rollback** -----数据立即生效/全部回滚撤消

注意：关闭自动提交，仅仅对当前本次连接有效。

手动管理方式

在默认自动提交开启的状态下，可以通过 SQL 命令，单独手动开启一个事务。然后执行多条 SQL 语句，之后再关闭事务。

开启事务：**start transaction**

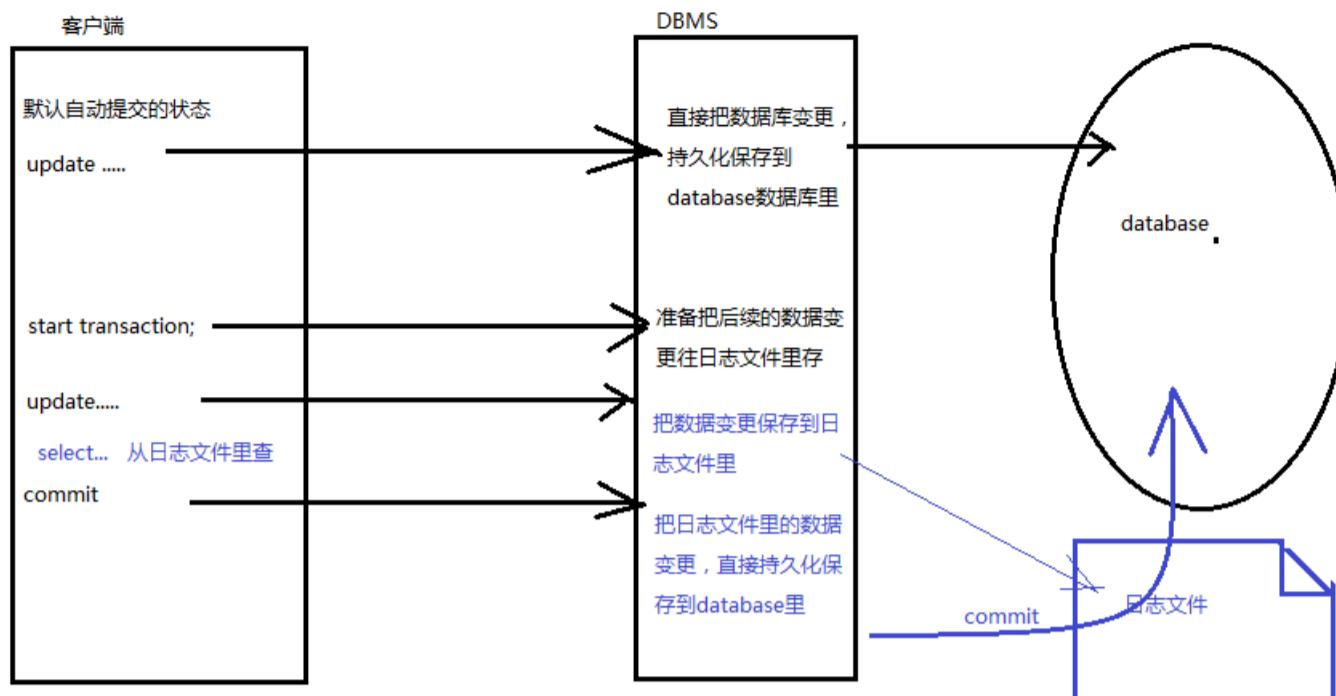
执行 n 条 SQL-----数据变更不会立即生效，而是被暂时保存起来了

关闭事务：

提交事务：**commit**; -----刚刚执行的所有 SQL 全部同时生效

回滚事务：**rollback**; -----刚刚执行的所有 SQL 全部撤消变更，回到事务开启之前的状态

3-3-3_事务的原理



3-3-4_事务的回滚点

默认状态下，回滚直接到事务开启之前的状态

设置回滚点，可以回滚到回滚点的位置，而不是事务开启之前的状态

```
开启事务: start transaction
执行 n 条 SQL
设置一个回滚点: savepoint 回滚点名称
执行 n 条 SQL
回滚:
    可以回滚到事务开启之前: rollback
    也可双回滚到回滚点的位置: rollback to 回滚点名称
```

3-3-5_事务的特性 ACID

1、事务的 ACID 四个特性 (面试题)

A: Atomicity, 原子性。 表示事务里的操作是不可分割的，要么全部成功，要么全部失败。不存在成功一半的情况。

C: Consistency, 一致性。 表示事务执行的前后，数据应该是完整的一致的。

I: Isolation, 隔离性。 表示多个事务同时并发执行时，事务之间应该是相互独立、互不干扰的。

D: Durability, 持久性。 表示事务执行之后，数据会被持久化保存到数据库里。保存之后不管出现什么问题，数据都在数据库里保存着。

3-3-6_事务并发的问题

在不考虑隔离性，或者隔离性不够高的时候，事务并发出现的问题。

脏读：一个事务，读取到了另外一个事务未提交的数据。是最严重的事务并发问题，必须要避免的问题

不可重复读：在一个事务里，多次读取的数据不一致。主要是受到其它事务的 update 操作影响

虚读/幻读：在一个事务里，多次读取的数据不一致。主要是受到其它事务的 insert、delete 影响

3-3-7 事务的隔离级别（了解）

事务并发有问题的原因是：隔离级别不够高。隔离级别有四种：

read uncommitted: 读未提交。是最低的隔离级别。存在脏读、不可重复读、虚读/幻读

read committed: 读已提交。解决了：脏读 存在：不可重复读、虚读/幻读

repeatable read: 重复读。解决：脏读、不可重复读 存在：虚读/幻读

serializable: 串行化。解决了所有并发问题

安全性： serializable > repeatable read > read committed > read uncommitted

性能： serializable < repeatable read < read committed < read uncommitted

隔离级别的操作：

查看隔离级别：**select @@tx_isolation**

设置隔离级别：**set session transaction isolation level 隔离级别**

级别	名字	隔离级别	脏读	不可重复读	幻读	数据库默认隔离级别
1	读未提交	read uncommitted	是	是	是	
2	读已提交	read committed	否	是	是	Oracle和SQL Server
3	可重复读	repeatable read	否	否	是	MySQL
4	串行化	serializable	否	否	否	

3-3-8 不同隔离级别的演示

假定：A 客户端主要演示用，B 客户端作为干扰事务

1、read uncommitted

设置 A 的隔离级别为 read uncommitted

A 和 B 同时开启事务

A 查询一下数据---主要是看原始数据是什么样的

B 执行 update 操作，但是不提交事务

A 再查询数据----如果查到的数据变化了，说明 A 读取到了 B 未提交的数据，存在：脏读

2、read committed

设置 A 的隔离级别为 read committed

A 和 B 同时开启事务

A 查询一下数据---主要是看原始数据是什么样的

B 执行 update 操作，但是不提交事务

A 再查询数据----如果查到的数据不变，说明解决了脏读问题

B 提交事务

A 再查询数据----看数据是否有变化，说明在 A 一个事务里多次查询数据不一致，存在：不可重复读问题

3、repeatable read

设置 A 的隔离级别为 repeatable read

A 和 B 同时开启事务

A 查询一下数据---主要是看原始数据是什么样的

B 执行 update 操作，但是不提交事务

A 再查询数据----如果查到的数据不变，说明解决了脏读问题

B 提交事务

A 再查询数据----看数据是否有变化，如果数据没有变化，说明解决了：不可重复读问题

4、Serializable

设置 A 的隔离级别为 serializable

A 和 B 开启事务

A 执行一个查询

B 执行一个 update 操作----处于等待状态。

A 事务结束，B 会立即执行

A 事务一直不结题，B 等待超时

3-4_DCL 语句

3-4-1_什么是 DCL

DCL: DataBase Control Language, 数据库控制语言，主要是 DBA 用来管理数据库的用户、权限等等。

需要在登录管理员帐号才能操作

3-4-2_管理用户

1、创建用户

语法: create user ‘用户名’ @’主机名’ identified by ‘密码’

用户名: 创建的帐号名称

主机名: 创建的帐号，可以在哪台电脑上登录这个数据库。localhost: 表示帐号只能在 MySql 本机登录；%: 表示可以在任意电脑上登录 MySql

密码: 创建的帐号的密码

示例:

创建用户名:liuyp，可以在任意电脑上登录，密码是:liuyp

create user ‘liuyp’@’%’ identified by ‘liuyp’;

2、修改用户密码

语法: **set password for ‘用户名’ @’主机名’ = password(‘新密码’);**

示例: SET PASSWORD FOR 'liuyp'@'%' = PASSWORD('12345');

3、删除用户

语法: **drop user ‘用户名’ @’主机名’**

示例: DROP USER 'liuyp'@'%';

3-4-3_管理权限

1、增加授权

语法: **grant 权限 1, 权限 2,... on 数据库名.表名 to ‘用户名’ @’主机名’**

权限: **select, update, delete, alter, drop** 等等。all 表示所有权限

数据库名: 这些权限可以操作哪个数据库

表名: 这些权限可以操作哪张表

‘用户名’ @’主机名’: 把权限授给哪个用户

示例: 给 liuyp 用户增加授权: 可以任意操作 heima41 的所有表

2、查看授权

语法: **show grants for ‘用户名’ @’主机名’**

示例: SHOW GRANTS FOR 'liuyp'@'%';

3、取消授权

语法: **revoke 权限 1, 权限 2,... on 数据库名.表名 from ‘用户名’ @’主机名’**

示例: 取消 liuyp 用户的 update 权限

4-JDBC

4-1_上节课回顾

4-1-1_多表查询

1、内连接查询

查询多表中必然有关联的数据。

隐式内连接

select * from 表 1, 表 2 where 关联条件 [and 过滤条件]

显式内连接

```
select * from 表 1 inner join 表 2 on 关联条件 [where 过滤条件]
```

2、外连接查询

查询一张表的全部数据，以及另一张表有关联的数据。如果另外一张表没有关联的数据，是 null

左外连接

```
select * from 左表 left join 右表 on 关联条件 where 过滤条件
```

右外连接

```
select * from 左表 right join 右表 on 关联条件 where 过滤条件
```

3、子查询

- 单行单列子查询---结果是一个值
- 多行单列子查询---结果是一个集合
- 多行多列子查询---结果是一张虚拟表，拿虚拟表和其它表进行关联查询

4-1-2 MySql 的事务管理

1、默认事务管理

关闭自动提交: set autocommit = 0;

执行 n 条 SQL 语句----不会立即生效

关闭事务:

提交事务: commit ----所有 SQL 同时生效

回滚事务: rollback ----所有 SQL 全部撤消

2、手动事务管理

手动开启事务: start transaction;

执行 n 条 SQL 语句----不会立即生效

关闭事务:

提交事务: commit ----所有 SQL 同时生效

回滚事务: rollback ----所有 SQL 全部撤消

3、事务的特性 ACID

A: 原子性，表示事务是不可分割的，不存在成功一半的情况

C: 一致性，表示事务执行前后数据是一致的和完整的

I: 隔离性，表示事务并发执行时，应该是相互独立，互不干扰的

D: 持久性，表示事务执行完成，数据被持久化保存到数据库里

4-2 JDBC 概述

4-2-1 什么是 JDBC

驱动：让程序代码来操作某些硬件设备的组件。数据库里的驱动：让程序代码来操作数据库的组件。

JDBC：Java DataBase Connectivity，Java 数据库连接。是 Sun 公司提供，操作数据库的一组 Java API，实现 Java 程序对不同数据库的统一操作。

数据库的驱动：各数据库厂商，根据 JDBC 规范提供的操作数据库的实现类。这些代码通常是以 jar 包的形式提供的。

4-2-2 数据库的驱动包

要使用 JDBC 操作数据库，必须有对应数据的驱动包才可以。

MySql 的驱动包：mysql-connector-java-5.1.37-bin.jar

4-2-3 JDBC 里相关的类和方法

1、JDBC 里的接口和类所在的包：

java.sql.*： JDBC 操作数据库的基础包

javax.sql.*： JDBC 操作数据库的扩展包

软件包	描述
java.sql	提供使用 Java 编程语言访问和处理存储在数据源（通常是关系数据库）中的数据的 API。
javax.sql	通过 Java 编程语言为服务器端数据源访问和处理提供 API。

2、常用的 JDBC 接口和类

- java.sql.DriverManager： JDBC 提供的工具类，主要是用来注册驱动，以及获取数据库连接
- java.sql.Connection： JDBC 的接口，规定的数据库连接对象应该实现的方法。
- java.sql.Statement： JDBC 的接口，是执行 SQL 的平台对象，是用来执行 SQL 语句的，规定了执行 SQL 的方法
- java.sql.ResultSet： JDBC 的结果，是 SQL 中 DQL 执行的结果集对象，规定了循环遍历查询结果集的方法

4-2-4 JDBC 的快速入门

首先：导入数据库驱动包

在 modules 创建一个文件夹： lib

把 jar 包拷贝粘贴到 lib 文件夹

在 jar 包上右键-->Add as Library—[设置在为在当前 Module 里使用，起一名称]，确定

使用代码操作数据库 CURD

注册驱动

获取连接：获取 **Connection** 接口的实现类对象

创建 **SQL** 执行平台：获取 **Statement** 接口的实现类对象

执行 **SQL** 语句：如果执行了 **DQL**，就会得到一个 **ResultSet** 对象

处理结果：处理 **ResultSet** 对象

释放资源：把 **ResultSet**、**Statement**、**Connection** 对象关闭掉

4-3 API 详解

4-3-1 注册驱动

1、什么是驱动

让 Java 程序操作数据库的类。

JDBC 规范规定了，所有的数据库驱动类，都必须要实现 **java.sql.Driver** 接口，实际注册驱动要注册这个接口的实现类。

MySql 的驱动类：**com.mysql.jdbc.Driver**

2、怎样注册驱动类

API:

`DriverManager.registerDriver(Driver driver)`

注册方式一(不推荐):

`DriverManager.registerDriver(new com.mysql.jdbc.Driver());`

原因 1：硬编码。代码里写死了注册的是哪个数据库的驱动类。如果更改了数据，就必须更改程序源码。

原因 2：注册了 2 次

注册方式二：

`Class.forName("com.mysql.jdbc.Driver");`

好处 1：注册了 1 次

好处 2：非硬编码。驱动类的名称是一个字符串，可以配置到配置文件中，使用程序加载配置文件，得到要注册的驱动类

4-3-2 获取连接

1、什么是连接对象

要操作数据库，就必须要先连接上数据库才可以。JDBC 规范规定了一个数据库连接的接口：**java.sql.Connection**。

获取连接实际上获取的是这个接口的实现类对象

2、怎样获取连接

API

`java.sql.DriverManager` 中有一个方法:

`static Connection getConnection(String url, String user, String password)`。

参数:

`url`: 数据库连接地址。JDBC 规定了 `url` 的写法: 协议:子协议:database 地址

协议: 固定值 `jdbc`

子协议: 通常是要连接数据库类型。连接 MySql, 这个值是: `mysql` `oracle`

`database` 的地址: 格式是由数据库厂商决定的, 不同数据库的写法不同。

MySql 的写法: `//ip:port/database` 名称。 `//localhost:3306/heima41`

如果连接是本机默认端口的 MySql, 可以简写成: `///database` 名称

示例: `jdbc:mysql:///heima41`

`user`: 数据库的用户名

`password`: 数据库的密码

返回值:

数据库连接对象。注册了哪个数据库的驱动, 获取的就是哪个数据库的连接对象。

4-3-3_创建 SQL 执行平台

1、什么是 SQL 执行平台

要执行 SQL 语句, 就必须要在这个执行 SQL 的平台对象里进行操作。这个对象就是 JDBC 规定的: `java.sql.Statement`。

要获取的执行平台对象, 实际上获取的就是这个接口的实现类对象

2、怎样获取执行平台对象

API:

`java.sql.Connection` 对象里有一个方法, 是用来获取 SQL 执行平台的

`Statement createStatement() throws SQLException`

Creates a Statement object for sending SQL statements to the database.

```
Statement statement = connection.createStatement();
```

4-3-4_执行 SQL 语句

通过 `Statement` 对象来执行 SQL 语句。

1、执行 DQL 的方法: **select**

```
ResultSet executeQuery(String sql) throws SQLException
```

2、执行 DML 的方法: **insert、update、delete**

```
int executeUpdate(String sql) throws SQLException
```

返回:

int: DML 语句影响的行数

3、执行 DDL 的方法: **create、alter、drop 等等**

```
boolean execute(String sql) throws SQLException
```

execute()方法不仅可以执行 DDL、还可以执行 DQL、DML、DCL 语句

返回:

boolean: 如果执行的 select, 结果有 ResultSet 对象, 就是 true; 如果执行的不是 select, 返回 false

4-3-5_处理结果 (ResultSet)

1、ResultSet 对象

是 select 语句执行查询的结果集对象，里边封装了查询的所有结果数据。相当于是一张二维表。如果想要获取 ResultSet 里的数据，就需要对这个对象进行循环遍历：得到其中每一行的每一个字段的数据。

2、ResultSet 对象的 API

向下移动一行: **next()**

返回: boolean

true: 下一行存在，移动成功

false: 下一行不存在，移动失败

向上移动一行: **previous()**

返回: boolean

true: 上一行存在，移动成功

false: 上一行不存在，移动失败

根据列序号获取字段值: **getXXX(int 列序号)**

XXX: 表示这一列的数据类型。如果是 varchar, 要写成 String; 如果是 int, 要写 int

根据列名称获取字段值: **getXXX(String 列名称)**

XXX: 表示这一列的数据类型。如果是 varchar, 要写成 String; 如果是 int, 要写 int

4-3-6_释放资源

```
resultSet.close();
statement.close();
connection.close();
```

注意：

前边用到了几个 JDBC 的对象，就要关闭掉几个对象。Connection、Statement、ResultSet 对象。
关闭的顺序，和开启的顺序相反。

4-4_封装 JdbcUtils 工具类

4-5_JDBC 的事务管理

4-5-1_JDBC 事务管理相关的 API

JDBC 的事务管理，是基于 Connection 对象的。所有事务操作的方法，都是 Connection 对象提供的方法。

开启事务：setAutoCommit(boolean autoCommit)

提交事务：commit()

回滚事务：rollback()

4-5-2_使用事务管理的步骤

- 注册驱动获取连接
- 开启事务
- 创建 SQL 执行平台
- 执行 SQL 语句
- 处理执行结果
- 关闭事务：提交/回滚
- 释放资源

4-5-3_JDBC 事务管理示例

5-连接池 DataSource

5-1_上节课回顾

5-1-1_JDBC 的步骤

导入 jar 包：数据库驱动包

代码操作的步骤：

```
//注册驱动
```

```
Class.forName("com.mysql.jdbc.Driver");
//获取连接
Connection conn = DriverManager.getConnection("jdbc:mysql://heima41","root","root");
//创建 SQL 执行平台
Statement statement = conn.createStatement();
//执行 SQL 语句: 执行 DQL, executeQuery; 执行 DML: int count = statement.executeUpdate()
ResultSet resultSet = statement.executeQuery("select * from user");
//处理结果
while(resultSet.next()){
    String uname = resultSet.getString("uname");
    System.out.println(uname);
}
//释放资源
resultSet.close();
statement.close();
conn.close();
```

5-1-2 JDBC 的 API

1、注册驱动

方式一： DriverManager.registerDriver(new com.mysql.jdbc.Driver());

缺点 1：硬编码。代码里写死了注册的是 MySQL 的驱动。如果更改数据库的话，就必须要更改源码

缺点 2：注册了 2 次

方式二： Class.forName(“com.mysql.jdbc.Driver”);

2、获取连接

DriverManager.getConnection(url, username, password=);

参数：

url 格式： jdbc:子协议:database 的地址

子协议：使用的数据库类型。MySQL 的值：mysql

database 的地址：由数据库厂商决定的。MySQL 的写法： //ip:port/database\

如果连接的是本机的默认端口 MySQL，可以简写成： //database

3、处理结果集 ResultSet

ResultSet：类似于一张二维表，是 select 语句查询的结果集对象。

到下一行： next()

返回值： boolean。表示是否移动成功。

到上一行: `previous()`

返回值: `boolean`。表示是否移动成功。

获取指定列的数据:

`getXXX(列序号): XXX 表示数据类型。列序号是从 1 开始`

`getXXX(列名称): XXX 表示数据类型。`

5-2_JDBC 的预编译对象

5-2-1_SQL 注入漏洞 (SQLInject)

什么是 SQL 注入漏洞: 通过在页面上构造一些特殊的数据, 提交到 Java 程序。Java 程序接收到之后使用 `Statement` 去数据库里执行 SQL 语句。但是执行的 SQL 是根据页面传递的数据拼接而成的, 如果这些数据是恶意构造的内容, 可能会导致 SQL 的结构会发生变化, 从而绕过登录验证的目的。

问题出现的原因:

通过一些特殊的参数, 拼接 SQL 时, 改变了 SQL 的结构。

原本条件: 用户名正确 并且 密码正确

SQL 注入之后: (用户名正确 并且 密码正确/错误) 或者 始终成立的恒等式

解决 SQL 注入漏洞的方案: 不再使用 `Statement` 执行 SQL 语句, 而是使用预编译对象: `PreparedStatement`

5-2-2_预编译对象 PreparedStatement

1、什么是 PreparedStatement

`public interface PreparedStatement extends Statement`

是 `Statement` 的一个子接口

2、怎样获取 PreparedStatement 对象

数据库连接对象 `Connection` 有一个方法:

`PreparedStatement prepareStatement(String sql) throws SQLException`

参数: `sql`

`sql` 的写法和普通的 SQL 不一样。需要使用占位符? 来代替参数值。

原本: `select * from user where uname = '用户名参数值' and password = '密码的参数值';`

预编译对象要求的: `select * from user where uname = ? and password = ?;`

方法会把使用占位符的 SQL, 预先编译。编译之后 SQL 的结构就固定下来了

3、给预编译的 SQL 设置参数值

`setXXX(?的序号, ?的值)`

`XXX: 要设置的参数的类型。`

参数:

?的序号: 给第几个占位符?设置参数值。从 1 开始

?的值: 设置的参数值

4、执行 SQL 语句

执行 DQL: `executeQuery()` 方法无参

返回: `ResultSet` 对象

执行 DML: `executeUpdate()` 方法无参

返回: `int` 表示 SQL 语句影响的行数

执行 DDL 或者 DCL: `execute()` 方法无参

返回: `boolean` 表示是否执行了 `select` 语句

5-2-3_预编译对象应用示例

```
//创建 SQL 执行平台: 预编译对象: 登录的 SQL 语句 select * from user where uname = 'lisi' and password = 'lisi'  
String sql = "select * from user where uname = ? and password = ?";  
PreparedStatement preparedStatement = connection.prepareStatement(sql);  
preparedStatement.setString(1, "lisi");  
preparedStatement.setString(2, "" or '1'='1");  
//执行 SQL 语句  
ResultSet resultSet = preparedStatement.executeQuery();
```

5-2-4_预编译对象的好处

解决了 SQL 注入漏洞

预编译对象 SQL 执行效率比 Statement 对象高

5-3_连接池 DataSource

5-3-1_连接池概述

1、什么是连接池

连接池: 里边保存维护了一堆连接(Connection 对象)的容器。之后如果要操作数据库, 需要使用数据库连接的话, 不需要创建连接对象, 而是可以从连接池里获取一个使用(租用), 使用完成之后, 再把这个连接归还到连接池里, 而不是关闭掉。

2、连接池的作用

解决了操作数据库的效率问题

可以循环使用 Connection

避免的数据库服务器内存溢出

3、JDBC 提供的连接池规范接口

JDBC 规范提供的连接池接口：`javax.sql.DataSource`。要求所有组织提供连接池工具的时候，都必须要实现这个接口。因为接口里提供了操作连接的统一的方法：`getConnection()`；

5-3-2_包装类-方法功能增强的方式

1、增加的方式有三种：

创建子类，继承父类，重写父类的方法。 在使用的时候必须要使用子类对象，才可增强。

`Connection conn =`

如果有子类继承 `Connection` 对象的话，原本对象里的数据没有办法处理

`Connection` 是一个接口。

包装类的方式，增强某个方法的功能----可以使用现有的知识来实现

代理类的方式，增强某个方法的功能----代理模式，需要使用反射。

2、包装类

`BufferedReader bufferedReader = new BufferedReader(new FileReader(""));`

`BufferedReader` 就是 `FileReader` 的一个包装类。

3、包装类的实现

- 需要有一个接口
- 被包装类，和包装类都要实现这个接口
- 被包装类里要有真实对象的一个成员变量
- 不需要增强的方法，直接调用真实对象的方法即可
- 需要增强的方法，调用真实对象的方法，还可以额外增加一些代码对功能进行增强

4、示例

普通人变身钢铁侠的包装类

接口：`Person`

```
package com.itheima.ds.wrapper;
```

```
public interface Person {  
    public void eat();  
    public void fitg();  
}
```

普通人类: Man

```
package com.itheima.ds.wrapper;

public class Man implements Person {

    @Override
    public void eat() {
        System.out.println("吃 2 碗");
    }

    @Override
    public void fight() {
        System.out.println("战斗力: 5");
    }
}
```

包装类: IronMan

```
package com.itheima.ds.wrapper;

public class IronMan implements Person {

    private Man man;

    public IronMan(Man man) {
        this.man = man;
    }

    @Override
    public void eat() {
        man.eat();
    }

    @Override
    public void fight() {
        man.fithg();
        System.out.println("盔甲增强战斗力: 95");
    }
}
```

使用包装类对象:

```
package com.itheima.ds.wrapper;  
public class WrapperTest {  
  
    public static void main(String[] args) {  
        Man man = new Man();  
  
        IronMan ironMan = new IronMan(man);  
  
        ironMan.eat();  
        ironMan.fithg();  
    }  
}
```

5-3-3_c3p0 连接池

1、什么是 c3p0 连接池

c3p0：是一个市面上比较流行的开源免费的连接池工具。功能比较强大，能够对连接池进行管理相关的 jar 包：

c3p0-* .jar
mchange-commons-* .jar

注意：在使用连接池的时候，除了连接池本身的 jar 包之外，还必须有数据库驱动包

2、连接池的通用 使用步骤

导入数据库驱动包和连接池的 jar 包

代码：

创建连接池对象：使用哪个连接池，就需要创建哪个连接池的对象

从连接池获取连接：getConnection();

3、c3p0 的配置项

必须有的四个基本配置项

driverClass
jdbcUrl
user
password

其它配置项

initialPoolSize: 连接池的初始化大小。创建连接池的时候，需要往池子里初始化几个连接对象
maxPoolSize: 连接池最大容量。当连接不够用时，连接池会自动增加连接。**maxPoolSize** 表示最多可以有多少个
maxIdleTime: 连接池里最大空闲连接时间。连接池里的一个连接，多长时间不使用的时候，把这个连接释放掉
minPoolSize: 连接池里最小连接数。

c3p0 的配置文件

配置文件的名称必须是：c3p0-config.xml

配置文件必须要放在：src 目录下

配置示例：

```
<c3p0-config>
  <default-config>
    <property name="driverClass">com.mysql.jdbc.Driver</property>
    <property name="jdbcUrl">jdbc:mysql:///heima41</property>
    <property name="user">root</property>
    <property name="password">root</property>

    <property name="checkoutTimeout">30000</property>
    <property name="initialPoolSize">10</property>
    <property name="maxIdleTime">30000</property>
    <property name="maxPoolSize">100</property>
    <property name="minPoolSize">10</property>

  </default-config>
</c3p0-config>
```

4、c3p0 连接池的使用

导入 jar 包：数据库驱动包 和 c3p0 连接池的 jar 包

准备配置文件：名称必须是 c3p0-config.xml，位置必须要放在 src 下

代码部分：

```
//创建连接池对象：类的构造方法里有写好的代码，会自动从 src 下加载 c3p0-config.xml 配置文件
ComboPooledDataSource dataSource = new ComboPooledDataSource();

//从连接池中获取连接
Connection connection = dataSource.getConnection();
```

5-3-4_druid 连接池

1、什么是 druid 连接池

DRUID：是 Alibaba 提供的免费开源的连接池工具。无论是功能、性能、可扩展性都要比其它连接池要强，据说是最强的 Java 数据库连接池。提供的日志监控功能，来监控连接池里连接的信息。

相关的 jar 包：druid-* .jar

2、druid 的使用

```
static DataSource createDataSource(Map properties)  
static DataSource createDataSource(Properties properties)
```

导入 jar 包：数据库驱动包 和 druid 的 jar 包

准备配置文件：druid.properties，放在 src 下

```
url=jdbc:mysql://heimai41  
username=root  
password=root  
driverClassName=com.mysql.jdbc.Driver  
initialSize=10  
maxActive=20  
minIdle=5
```

代码：

读取 properties 配置文件

```
Properties properties = new Properties();  
InputStream is = DruidDemo.class.getClassLoader().getResourceAsStream("druid.properties");  
properties.load(is);
```

创建连接池对象：DruidDataSourceFactory

```
DataSource dataSource = DruidDataSourceFactory.createDataSource(properties);
```

从连接池里获取数据库连接

```
Connection connection = dataSource.getConnection();
```

5-4_作业

准备

创建一个数据库 demo

创建一张表 student:

学号，主键

姓名

年龄

性别

入学日期

向 student 表里插入一些数据

张三

李四

使用 **JdbcUtils** 工具类（连接池优化过的工具）

查询 student 表里所有的数据，显示到控制台

向 student 表里插入一条数据：王五

修改 student 表里 张三 年龄

删除 student 表里的 张三 的数据

所有操作都要使用 **PreparedStatement** 对象

1. 直接使用 c3p0 连接池对象进行操作
2. 直接使用 druid 连接池对象进行操作
3. 使用 **JdbcUtils** 工具类进行操作

6-JDBCTemplate

重点知识

ParameterMetaData 元数据

ResultSetMetaData 元数据

JDBCTemplate 应用

执行 DML 语句：使用 JDBCTemplate 的 update 方法

执行 DQL 语句：查询一个值，queryForObject ---查询数据的条数

执行 DQL 语句：查询一条数据，封装成 Map，queryForMap

执行 DQL 语句：查询多条数据，封装成 List，queryForList

执行 DQL 语句：查询多条数据，封装成 List<JavaBean>， query(sql, rowMapper, params)

执行 DQL 语句：查询一条数据，封装成 JavaBean 对象，queryForObject(sql, rowMapper, params)

6-1_上节课回顾

6-1-1_预编译对象

1、作用

防止 SQL 注册

可以循环使用数据库连接

提高数据库操作效率

2、使用方法

- 获取连接
- 准备 SQL 语句：使用占位符?来代替参数值
- 预编译 SQL，得到一个预编译对象
- 设置 SQL 的参数值
- 执行 SQL 语句：`execute()`执行 DDL 和 DCL；`executeQuery()`执行 DQL 语句，得到 `ResultSet`。`executeUpdate()`执行 DML，得到 int 表示 SQL 影响的行数
- 处理结果
- 释放资源

6-1-2_c3p 连接池

1、相关 jar 包

0.9.2 版本：c3p0 mchange

数据库的驱动包

2、使用步骤

- 导入 jar 包
- 准备配置文件：名称必须是 `c3p0-config.xml`，必须放在 `src` 下。
最少要有四个基本配置：数据库驱动、地址、用户名、密码
- 代码部分：

//创建连接池对象：会自动从 `src` 下读取 `c3p0-config.xml`

```
ComboPooledDataSource dataSource = new ComboPooledDataSource();
```

//从连接池里获取一个连接

```
Connection connection = dataSource.getConnection();
```

6-1-3_druid 连接池

1、相关的 jar 包

- druid 本身的 jar 包：
- 数据库的驱动包

2、使用方法

- 导入 jar 包
- 准备配置文件：是 `properties` 格式的配置文件，名称随意。最有要有四个基本配置：驱动、地址、用户名、密码

- 代码部分：

//1. 读取 properties 配置文件

```
Properties prop = new Properties();
```

```
prop.load(properties 的文件的输入流对象);
```

//2. 根据配置文件创建 druid 的连接池对象

```
DataSource dataSource = DruidDataSourceFactory.createDataSource(prop);
```

//3. 从连接池里获取连接

```
Connection connection = dataSource.getConnection();
```

6-2 数据库的元数据

6-2-1 什么是元数据

按照传统的说明，定义数据结构的数据。通俗一点说：数据库里某些结构的定义信息，比如：字段的元数据，数据库的元数据。

比如：字段的元数据：字段的名称、字段的类型等等定义的信息

数据库的元数据： database 的名称、类型等等信息

JDBC 的元数据：

DatabaseMetaData: 是数据库的元数据，从 Connection 对象里获取的，从这个元数据对象里可以获取到 database 的名称、数据库的版本号等等

ParameterMetaData: 是参数的元数据，从预编译对象中获取的，预编译的 SQL 的参数元数据：SQL 里有几个？，每一个参数？的类型

ResultSetMetaData: 是查询结果集的元数据，从 ResultSet 里获取的。可以从中获取到结果集里列的个数、每一个列的名称、每一列的类型等等。

6-2-2 ParameterMetaData

1、什么是 ParameterMetaData

从 PreparedStatement 对象里获取参数元数据对象，可以从中获取到预编译的 SQL 语句中，参数？的个数、类型信息。

2、怎样获取 ParameterMetaData

在 PreparedStatement 对象里有一个方法：

```
ParameterMetaData getParameterMetaData()
```

3、ParameterMetaData 的 API

获取 SQL 中参数的个数：**int getParameterCount()**

获取 SQL 中指定参数的类型：**int getParameterType(int 参数的序号)**

注意：不是所有的数据库，都可以获取到预编译 SQL 的中参数的类型。 MySql 就会有问题

6-2-3 _ResultSetMetaData

1、什么是 ResultSetMetaData

是从 `ResultSet` 对象中获取的结果集的元数据对象，可以从中获取到结果集里列的个数，以及每个列的名称、类型等等。

2、怎样获取 ResultSetMetaData

`ResultSetMetaData getMetaData() throws SQLException`

Retrieves the number, types and properties of this `ResultSet` object's columns.

Returns:

the description of this `ResultSet` object's columns

3、ResultSetMetaData 的 API

- 获取 `ResultSet` 中字段的数量: `int getColumnCount()`
- 获取指定字段的名称: `String getColumnName(int 列序号)`
- 获取指定字段的类型: `int getColumnType(int 列序号)`

6-2-4 使用元数据自定义一个 JDBCTemplate 类

1、使用原始方式操作数据库的弊端

每次操作数据库，都必须要写固定的代码。如果操作数据库非常频繁，程序里就会有大量的冗余的重复的代码。

2、解决方式

自定义框架: `MyJdbcTemplate`, 实现了 DML 语句的封装。操作起来更简单。

6-3_JDBCTemplate

6-3-1_什么是 JDBCTemplate

使用原始的 JDBC+连接池操作数据库，已经可以满足大部的数据库的基本操作，但是有大量的重复代码。`JDBCTemplate` 就实现了对数据库操作的进一步封装，再操作数据库就更加简便、易于使用了。

JDBCTemplate 是 Spring 框架的一部分，把 JDBC 操作的核心部分封装起来了，比如: Statement 对象的创建与关闭、结果集的获取与关闭。我们在使用 `JDBCTemplate` 的时候，只需要提供要执行的 SQL 语句和参数就可以，调用 `JDBCTemplate` 获取执行的结果。

6-3-2_JDBCTemplate 相关的 API

操作数据库的核心类: `JDBCTemplate`。里边提供了执行各种 SQL 语句和方法

构造方法: **JdbcTemplate(DataSource dataSource)**

参数:

dataSource: 任意一种实现了 `javax.sql.DataSource` 接口的连接池都可以, 例如: c3p0, druid

常用方法:

`void execute(String sql)`: 执行任意 SQL 语句, 但是没有返回值, 所以通常用于执行 DDL 和 DCL

`int update(String sql, Object... params)`: 执行 DML 语句

参数:

SQL: 是 DML 语句。如果有参数, 要使用带占位符?的 SQL

params: 可变参数。是 SQL 语句的参数值

返回值:

`int`, 表示 SQL 语句影响的行数

`queryXXX()`: 执行 DQL 语句的方法。类里有很多方法的重载, 参数不同, 返回值也不同

6-3-3 使用 JDBCTemplate 的方式

1、导入 jar 包

JDBCTemplate 相关的 jar 包: 5 个

连接池的 jar 包: c3p0 相关的 jar 包 `c3p0-*jar, mchange-commons-*jar`

数据库驱动包

2、执行 DML 语句

使用的方法 API

```
int execute(String sql, Object... params)
```

可以执行的 SQL 语句类型

insert	update	delete
--------	--------	--------

3、执行 DQL 语句

相关的 API

- `query()`: 通用的执行 DQL 的方法, 我们可自定义把查询结果集转换成任意对象。通常是封装成 Java 对象
- `queryForObject()`: 一般是查询单值的结果, 比如聚合函数的查询: 查询数据库里的总条数; 查询一条数据, 封装成一个 Java 对象
- `queryForMap()`: 查询一条数据, 封装成 Map。Map 的 key 是字段名称, value 是字段的值
- `queryForList()`: 查询多条数据, 封装成 `List<Map<字段名, 字段值>>`

queryForObject

查询单值：查询 student 表的总条数

查询总条数：select count(*) from student，得到的结果是一个值。

单值查询，通常使用 queryForObject 方法

```
queryForObject(String sql, Long.class, Object... params)
```

参数：

sql：要执行的 SQL 语句。如果有参数，需要使用占位符？

typeClass：要把查询结果转换成什么类型的 Java 值。

如果写成：Long.class，结果就是一个 Long 对象

如果写成：Integer.class，结果就是一个 Integer 对象

params：执行 SQL 语句需要的参数值

返回值：

返回结果的类型，是由第二个参数决定的。

queryForMap

查询一条数据封装成 Map：查询 sid 值为 5 的 student 数据

查询一条数据，使用 queryForMap 来处理

```
queryForMap(String sql, Object... params)
```

参数：

sql：要执行的 SQL 语句。如果有参数，需要使用占位符？

params：执行 SQL 需要的参数值

返回值：

Map<字段名, 字段值> 是一行数据封装成的 Map

queryForList

查询多条数据封装成 List：查询 student 表里所有的数据

```
queryForList(String sql, Object... params)
```

参数：

sql：要执行的 SQL 语句

params：SQL 语句需要的参数值

返回值：

List<Map<字段名, 字段值>>。List 里每一个元素是一行数据的 map，Map 的 key 是字段名，value 是字段值

query()

查询 student 表里的所有数据，封装成一个 List<Student>

```
query(String sql, RowMapper rowMapper, Object... params)
```

参数：

sql：要执行的 SQL 语句

rowMapper：是一个接口。通常是 new 匿名内部类，重写 mapRow 方法：

把 ResultSet 里的一行数据，转换成你想要的结果对象

比如：把 ResultSet 里的一行数据，转换成一个 Student 对象

params：SQL 语句需要的参数

返回值：

List 集合，集合里的每一个元素，就是 RowMapper 里自定义封装的 Java 对象

```
query(String sql, new BeanPropertyRowMapper(Student.class), Object... params)
```

参数:

SQL: 要执行的 SQL

BeanPropertyRowMapper 对象: 是 RowMapper 接口的一个实现类, 是 JDBCTemplate 提供好的, 已经封装过的类。

new BeanPropertyRowMapper(JavaBean.class)

params: 执行 SQL 需要的参数值

queryForObject

查询一条数据, 封装成 Student 对象

```
queryForObject(String sql, RowMapper rowMapper, Object... params)
```

参数:

sql: 要执行的 SQL 语句

rowMapper: 把 ResultSet 封装的代码逻辑, 直接使用 BeanPropertyRowMapper 对象即可

new BeanPropertyRowMapper(Student.class);

params: SQL 需要的参数值

返回值:

在 rowMapper 指定的 JavaBean 对象

7-HTML

7-1_上节课回顾

7-1-1_元数据

怎么获取

```
ParameterMetaData metadata = preparedStatement.getParameterMetaData();
```

作用

可以获取参数的个数: int getParameterCount();

可以获取参数的类型: int getParameterType(int 参数序号)

2、ResultSetMetaData

怎么获取

```
ResultSetMetaData metadata = resultSet.getMetaData();
```

作用

获取结果集里列的个数: int getColumnCount()

获取结果集里列的名称: `String getColumnName(int 列序号)`

获取结果集里列的类型: `int getColumnType(int 列序号)`

7-1-2_JDBCTemplate

1、导入 jar 包

`JDBCTemplate` 本身的 jar

连接池的 jar 包

数据库的驱动包

2、应用方式

第一步：创建 `JDBCTemplate` 对象

```
new JDBCTemplate(连接池对象)
```

第二步：执行 DML 语句

```
int update(String sql, Object... params);
```

第三步：执行 DQL 语句-查询单个值：`queryForObject`

```
queryForObject(String sql, Class typeClass, Object... params)
```

第四步：执行 DQL 语句-查询一条数据，封装成 Map：`queryForMap`

```
queryForMap(String sql, Object... params)
```

第五步：执行 DQL 语句-查询多条数据，封装成 List：`queryForList`

```
queryForList(String sql, Object... params)
```

返回结果：`List<Map<字段名, 字段值>>`

第六步：执行 DQL 语句-查询多条数据，封装成 JavaBean 的集合：`query`

```
query(String sql, RowMapper rowMapper, Object... params)
```

```
query(String sql, new BeanPropertyRowMapper(JavaBean.class), Object... params)
```

第七步：执行 DQL 语句-查询一条数据，封装成一个 JavaBean 对象：`queryForObject`

```
queryForObject(String sql, new BeanPropertyRowMapper(JavaBean.class), Object... params)
```

返回：第二个参数指定的 一个 JavaBean 对象

7-2_HTML 简介

7-2-1_什么是 HTML

HTML: HyperText Markup Language, 超文本标记语言。

7-2-2_HTML 的作用

是用来编写网页的

7-2-3_HTML 的语法

HTML 文件的后缀名是: .html 或者 .htm

HTML 由一堆标签组成的

标签: <关键字>标签的内容</关键字> <关键字/> 自闭合标签

标签上可以加属性: <关键字 属性名="值" 属性名="值" ></关键字>

一个标签上可以加任意多个属性, 多个属性之间使用空格分开

标签可以嵌套:

正确的嵌套: <关键字> <子标签></子标签> </关键字>

错误的: <关键字> <子标签> </关键字></子标签>

HTML 不区分大小写

HTML 不需要编译, 可以使用浏览器直接打开

7-2-4_HTML 的结构

必须根标签: html

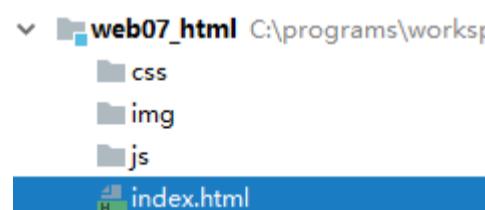
根标签里必须有两个子标签:

head: 对当前 HTML 页面进行配置

body: 显示到浏览器页面上的内容

注意: 如果不按照标签的结构写 HTML, 浏览器也可以显示, 但是不符合规范。

7-2-5_使用 idea 创建 static web 应用的目录结构



7-3_HTML 的常见标签

7-3-1_文字排版类标签

1、标题标签: **hn**

01. 标题标签: h1~h6

特点:

默认加粗

内置字号: h1 最大, h6 最小

独占一行

2、横线标签: **hr**

02. 横线标签: hr 是自闭合标签

属性:

color: 横线的颜色

size: 横线的粗细。 50px, px 是像素

width: 横线的长度。 500px

align: 横线的水平位置。常用值: left, center (默认值), right

3、段落标签和换行标签

03. 段落标签: p 一个 p 标签是一个段落, 把内容写在 p 标签的标签体里

特点:

段落之间有空白的间隔。

04. 换行标签: br 自闭合标签

特点:

换行后, 行之间没有空白的间隔

HTML 里空格的表示: no-break space

4、文字标签

05. 文字标签: font。

font 标签在 HTML5 里不建议使用了, 推荐是 span 标签+css 来进行文字的样式控制

常用属性:

color: 文字的颜色

size: 文字的字号大小。最小 1, 最大 7, 默认 3

face: 文字的字体。

06. 粗体字标签: b bold

07. 斜体字标签: i italic

08. 下划线标签: u underline

7-3-2_图片显示标签

01. 图片标签: img 自闭合标签 可以显示任意的图片。jpg, png, gif

属性:

src: source 资源的路径

width: 图片显示的宽度

height: 图片显示的高度

title: 鼠标悬浮提示

alt: 图片加载不出来的时候, 显示的文字信息

注意: 设置图片的宽高时, 只需要设置一个 width, 或者设置一个 height, 可以实现图片的等比例缩放。

如果设置了 width 和 height, 图片显示就可能变形

7-3-3_清单标签和超链接标签

1、清单标签:

无序清单: ul 配合子标签 li

ul: unordered list

属性:

type: 每一个列表项前边的标记类型。常用的值有:

disc: 小圆点, 默认值

circle: 小圆圈

square: 小方块

li: list item

有序清单: ol 配合子标签 li

ol: ordered list

属性:

type: 每一个列表项前边的序号类型

1: 阿拉伯数字序号

a: 英文字母序号

A: 大写字母序号

i: 罗马数字序号

I: 大写罗马数字序号

li: list item

2、超链接标签

超链接标签: **a**

属性:

href: 打开的页面地址

target: 在哪显示新页面

_self: 在当前窗口显示新页面, 默认

_blank: 在新窗口显示新页面

a 标签: 可以作为锚点来进行页面定位

1. 设置一个锚点: **a** 不需要有标签体, 只要有一个 **id** 属性
2. 设置一个链接, 点击这个链接的时候, 页面滚动到锚点所在的位置

href 属性值写成: #锚点的 **id**

7-3-4 表格标签: 创建与合并

1、表格的创建

table: 表格

属性:

border: 0 无边框, 1 有边框

width: 表格的宽度

align: 表格的水平位置: left/center/right

bgcolor: 背景颜色

cellspacing: 单元格之间的间距

cellpadding: 单元格边框线条和内容之间的距离

tr: 表格的行, 一个 **table** 里可以有多个 **tr** 标签

属性:

align: 行里所有单元格内容的水平位置

height: 行的高度

td: 表单的单元格, 一个 **tr** 里可以有多个 **td** 标签

属性:

align: 单元格内容的水平位置

rowspan: 跨行合并的, 值是合并了几个单元格

colspan: 跨列合并的, 值是合并了几个单元格

th: 表头单元格。是一种特殊的 **td**, 用法和 **td** 完全一样。仅仅是自带样式

caption: 表格的标题, 是 **table** 的子标签通常加在第一个 **tr** 之前

标签名称>子标签名称*个数>子标签名称*个数 按 Tab 键

2、合并单元格

要合并 1-1 和 1-2

合并单元格的步骤:

1. 确定是跨行合并还是跨列合并: 跨行合并使用 **rowspan**, 跨列合并使用 **colspan**

跨列, 使用 **colspan**

2. 确定合并了几个单元格: 合并几个单元格, 属性的值就是几。**rowspan="合并的个数"**, **colspan="合并的个数"**

合并了 2 个单元格, **colspan="2"**

3. 确定属性加在哪个单元格上: 要合并的单元格里最前边的那个单元格上

要合并的是 1-1 和 1-2, 1-1 在前, 所以 **colspan** 属性应该加在 1-1 单元格上

4. 删除被合并掉的单元格

1-2 已经被合并掉, 把 1-2 删除掉

7-3-5_语义化标签和 HTML 实体字符

1、语义化标签

什么是语义化标签

看到标签名称, 就知道是什么标签。

用处:

1.对搜索引擎有用

2.利于后续的维护—看到页面的代码, 就知道哪些是文档的头、尾、体

常见的语义化标签

<header>: 用于网页的页眉, 页眉通常用于设置网站标志、主导航、全站链接以及搜索框。

<nav>: 用于页面的导航, 仅对文档中重要的链接群使用。

<main>: 页面主要内容, 一个页面只能使用一次。

<section>: 具有相似主题的一组内容, 比如网站的主页可以分成介绍、新闻条目、联系信息等条块。

<footer>: 用于网页的页脚, 只有当父级是 **body** 时, 才是整个页面的页脚。

标签的本质是 **div** 标签

2、实体字符

什么是实体字符

HTML 是由一堆标签组成的<关键字></关键字>。那么如果我要在页面上显示： 3<a b>5 页面显示效果不正常。

这个时候就需要使用 HTML 提供的实体字符来代替这些特殊字符。

常见的实体字符

&nbsp 空格

< <

> >

& &

' 英文的单引号

" 英文的双引号

7-4 HTML 其它

7-4-1 文档声明

```
<!DOCTYPE html>
```

声明给浏览器，当前 HTML 页面使用的是哪个版本的 HTML 规范写成的。

7-4-2 head 标签

主要是对当前 HTML 进行配置的，在 head 标签里增加子标签进行配置

网页的标题配置：<title>网页标题</title>

网页的字符集配置：<meta charset="UTF-8">

8-Form&Css

8-1 上节课回顾

8-1-1 文字排版

标题标签：h1~h6

8-1-2 图片

src: 图片的地址

width: 图片的宽度

height: 图片的高度

title: 鼠标悬浮提示

alt: 图片显示不出来，文字提示信息

8-1-3_清单和超链接

1、清单

有序清单: **ol li**

无序清单: **ul li**

2、超链接

href: 跳转的地址

target: 新页面在哪显示。**_self/_blank**

8-1-4_表格标签

table: 表示表格整体

tr: 表格的行

td: 单元格

th: 表头单元格。是一种特殊的 **td**

caption: 表格的标题

合并单元格的步骤

确定跨行 **rowspan** 还是跨列 **colspan**

确定要合并几个单元格：合并几个单元格，**rowspan/colspan** 的值就是几

确定 **rowspan/colspan** 加在哪个单元格上：加在要合并的单元格里，最前边的那个单元格上

删除被合并掉的单元格

8-2_表单 form(重点掌握)

8-2-1_form 标签

01. 表单标签: **form**

属性:

name: 给表单起一个名称

action: 表单数据提交的路径

method: 表单数据提交的方式。常用的表单提交方式: **get** 和 **post**

HTTP 协议

get 和 **post** 提交的区别 (面试题):

get 提交: 数据会显示到地址栏的; **post** 提交: 数据不会显示到地址栏

get 提交不安全; **post** 提交相对安全

get 提交长度有限制(IE 限制 2083 个字符); **post** 理论上没有长度限制

表单数据提交的格式: name=value&name=value

username=tom&password=111&sex=female&hobby=music&hobby=ball

真正的数据提交格式: 表单提交路径?name=value&name=value...

http://localhost:63342/workspace41/web08_form_css/01_1form
签.html?username=zhangsan&password=12345&sex=female&hobby=code&hobby=ball

标

8-2-2_input 标签

02.input 标签

属性:

type: input 表单项的类型

text: 文本框, 默认值

password: 密码框。显示的是掩码

radio: 单选按钮。相同 name 的 radio 属于同一组, 同组 radio 有一个特性: 选择互斥

默认值设置: 在需要设置选中的选项上, 增加属性: checked="checked"

checkbox: 复选框

默认值设置: 在需要设置选中的选项上, 增加属性: checked="checked"

button: 普通按钮。没有任何功能的按钮, 但是可以和 js 配合, 自定义任意的功能

submit: 提交按钮。把按钮所在的表单提交

reset: 重置按钮。把按钮据的表单重置为默认值

image: 图片提交按钮。功能和 submit 完全一样, 只是用一张图片作为提交按钮来使用

file: 文件选择框。默认提交的是文件名称, 而不是上传文件。

hidden: 隐藏域。不会显示到页面上的表单项。但是如果 name 属性, 隐藏域的值也会被提交。

如果数据不想让用户看到, 就把数据放在隐藏域里, 加上 name 属性

name: 表单项的名称。

任意一个表单项, 如果表单项的数据需要提交的话, 就必须有 name 属性

value: 表单项的值, 在不同表单项的作用不同

在 text 里: 表示默认值

在 password 里: 表示默认值, 但是很少使用

在 radio 里: 表示每一个选项的值。选择了哪个选项, 就提交哪个选项的 value 值

在 checkbox 里: 表示每一个选项的值。选择了哪个选项, 就提交哪个选项的 value 值 name=value

在 button 里: 表示按钮上的提示文字

在 `submit` 里: 表示按钮上的提示文字

在 `reset` 里: 表示按钮上的提示文字

在 `image` 里: 没有实际意义

在 `file` 里: `value` 是无效的

readonly: 固定值 `readonly`。表示表单项的值不可更改，但是数据会被提交

disabled: 固定值 `disabled`。表示表单项不可用：值不可更改，并且数据也不会提交

8-2-3 `select` 标签-下拉框标签

03. 下拉框标签: `select`

属性:

name: 如果下拉框的值需要提交，就必须有 `name` 属性

下拉选项标签: `option` 是 `select` 的子标签

属性:

value: 表示每一个下拉选项的值。选中哪个选项，就提交哪个选项的值

默认值设置:

没有任何设置的时候，默认选中第一个选项

手动设置默认值: 在需要选中的选项上增加属性: `selected="selected"`

8-2-4 `textarea` 标签-文本域标签

04. 文本域标签: `textarea`

属性:

name: 名称，如果文本域的数据要提交，就必须有 `name` 属性

cols: 宽度可以显示几列

rows: 高度可以显示几行 通常是被 `css` 代替

默认值设置:

`value` 属性对 `textarea` 是无效的，

默认值设置: 把默认值写在标签体（开始标签和结束标签中间）里

8-3 Css

8-3-1 css 概述

1、什么是 CSS

CSS: cascade stylesheet 层叠样式表

2、CSS 的作用

用来美化页面的

8-3-2_css 的 3 种引入方式

1、行内样式

行内样式:

在 html 标签上增加属性 style，把样式写 style 的属性值里

缺点:

没有复用性

```
<span style="color:red;font-size:20px;">jack</span>
```

2、内部样式

内部样式:

在 head 标签里增加子标签 style，把样式写在 style 标签体里

```
<style type="text/css">  
    span{  
        color:blue;  
        font-size:20px;  
    }  
</style>
```

3、外部样式

外部样式。

在 head 标签里增加子标签 link，引入外部的 css 文件

link 标签:

rel: relation，表示引入的文件和当前页面的关系。

引入 css 的时候，值是: stylesheet，引入的 css 文件是当前的样式表文件

type: 引入的文件类型。可以不加

href: 外部 css 文件 路径

```
<link rel="stylesheet" type="text/css" href="css/my.css">
```

8-3-3_css 选择器

CSS 的选择器:

基本选择器:

标签选择器: 标签名称{样式名:值;样式名:值;...}

ID 选择器: #id 值{样式名:值;样式名:值;...}

类选择器: .类名{样式名:值;样式名:值;...}

优先级: ID 选择器 > 类选择器 > 标签选择器 优先级一样: 后加载覆盖先加载

层级选择器：

父选择器 后代选择器 大后代选择器 {样式名:值;样式名:值;...}

注意：层级选择器里的空格，表示的是后代，而不是父子关系

属性选择器：

选择器[属性名="值"]{样式名:值;样式名:值;...}

8-3-4 _css 修饰 HTML 标签

看 w3c 手册

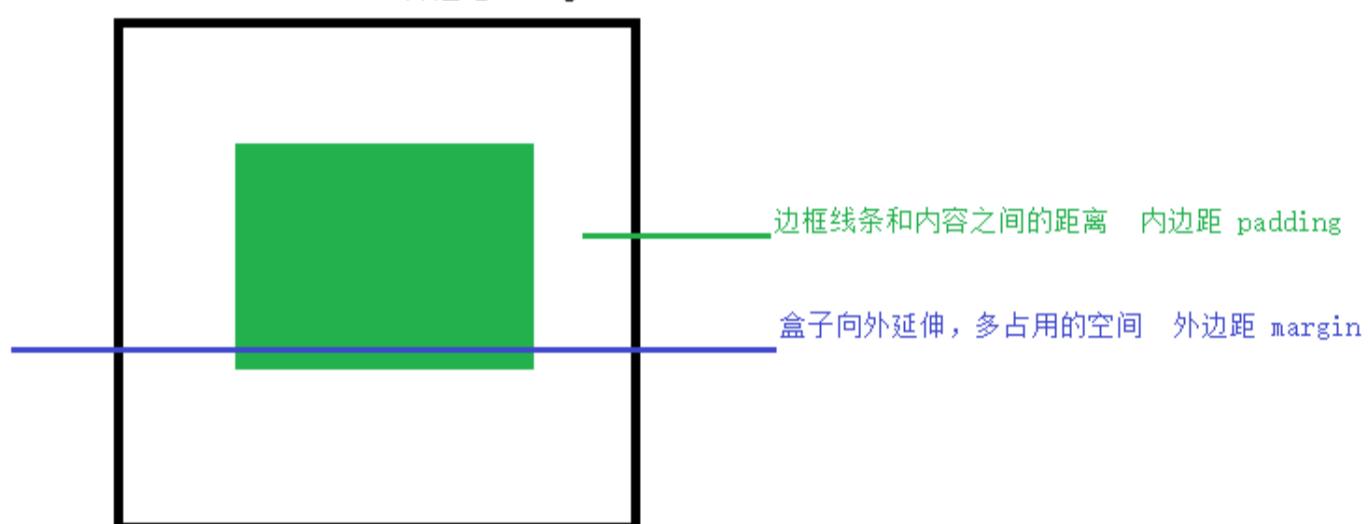
8-3-5 盒子模型

盒子模型指的样式：宽width高height

边框线条：border

内边距：padding

外边距：margin



9-JavaScript 上

总结

- 了解 js 的作用
- 能说出 js 的 5 种原始数据类型
- 能使用 js 常用的运算符：+-*等
- 能使用 js 常用的逻辑语句：if, else, for 等等
- **会定义和调用函数**
- **说出 4 个常用的事件：onclick, onsubmit, onchange, onsubmit**
- **会绑定事件**
- 能说出 js 的 5 个 BOM 对象

9-1_js 的概述

9-1-1_什么是 js

JavaScript: 简称是 **js**, 是 **web** 开发中不可缺少的脚本语言, 不需要编译就可以运行 (解释性语言)。**js** “寄生”在 **HTML** 体内, 随网络传输到客户端, 在浏览器内存中运行。

9-1-2_js 的作用

- 可以操作浏览器
- 可以操作 **HTML** 文档 (更改的是浏览器内存中的这一份 **HTML** 文档)

9-1-3_js 的组成

- **ECMAScript:** 是 **js** 的核心, 是 **js** 的基本语法规范
- **BOM:** **Browser Object Model**, 浏览器对象模型。我们的 **js** 代码可以调用 **BOM** 对象的属性和方法, 来操作浏览器。比如: 前进、后退、刷新当前页、网址跳转
- **DOM:** **Document Object Model**, 文档对象模型。我们的 **js** 代码可以调用 **DOM** 对象的属性和方法, 来操作 **HTML** 文档。比如: 修改样式、修改 **HTML** 里的文本、创建标签、删除标签、操作标签上的属性等等

9-1-4_js 的引入方式

1、内部 js

```
<script type="text/javascript">  
    alert();  
</script>
```

2、外部 js

```
<script type="text/javascript" src="js/my.js"></script>
```

3、引入 js 的注意事项

1. 一个 **script** 标签, 要么没有 **src** 属性, 在标签体里写 **js** 代码; 要么有 **src** 属性没有标签体, 引入外部 **js** 文件。
两种方式不能混用
2. **js** 代码在 **HTML** 里任意地方都可以生效, 但是有一个原则:
js 越晚加载越好 (越靠后越好), 通常是写在 **body** 结束标签之前
css 要写到页面的前边, 通常是写在 **head** 标签里

9-2_js 的基本语法(ECSAScript)

9-2-1_变量定义

js 里所有变量定义都使用关键字: var

js 是弱类型语言, Java 是强类型语言

9-2-2_数据类型

js 是动态类型语言

原始数据类型:

boolean: 布尔类型。true, false

number: 数字类型。

string: 字符串类型。

undefined: 未定义类型。只有一个值: undefined

object: 对象类型。null 也属于 object 类型

引用数据类型(下节课讲)

js 只有 9 个引用数据类型(内置类): Array, Boolean, Date, Events, Functions, Math, Number, RegExp, String

创建 js 对象, 也使用关键字: new

```
var obj = new Date();
```

9-2-3_运算符

js 的运算符和 Java 的运算符基本一样, 也有: +,-,*,/=,!=,+=,-=,*=,/=%,%=,>,<,>=,<=等等。

但是有一些是 js 比较特殊的:

== 和 **==:**

==: 只比较值, 只要值一样, 就是 true

==: 全等, 只有值和类型都一样, 才是 true

+/-*/

+: 加法运算。如果有字符串的加法是拼接字符串

-: 减法运算。如果有字符串, js 会尝试把字符串转换为数字之后, 再进行减法运算

*****: 乘法运算。如果有字符串, js 会尝试把字符串转换为数字之后, 再进行运算

/: 除法运算。如果有字符串, js 会尝试把字符串转换为数字之后, 再进行运算

&& 和 **||:** js 进行 boolean 类型运算时, 只有**&&** 和 **||**, 没有**&** 和 **|**

9-2-4_逻辑语句

js 的逻辑语句和 Java 逻辑语句非常相似, 也有: if, else, for, switch, while..., do..while

但是其中 if 和 Java 的 if 判断不同：

Java 的 if: 条件必须是 boolean

js 的 if: 条件可以是任意类型。 false, 0, "", null, undefined 是 false, 其它都是 true

9-2-5_函数(重点)

1、普通函数

定义:

```
function 函数名称(形参 1, 形参 2,...){  
    //函数的代码  
    //return 在定义函数时如果需要返回值就 return, 否则就不加 return  
}  
  
function add(a, b){  
    return a+b;  
}
```

注意: js 里的函数没有重载。如果多个函数的名称一样, 最后一个函数有效, 前边的都被覆盖掉了

调用:

```
函数名(实参 1, 实参 2,...);
```

```
var result = add(3, 5);
```

如果函数有返回值, result 就是返回值; 如果函数没有返回值, result 就是 undefined

2、匿名函数

定义:

```
var fn = function(形参 1, 形参 2,...){  
    //函数的代码  
    //return 在定义函数时如果需要返回值就 return, 否则就不加 return  
}
```

调用:

- 把匿名函数赋值给一个变量, 通过变量名调用函数
- 把匿名函数作为另外一个函数的实参, 传递进去

9-2-6_事件(重点)

1、事件的相关概念

- 事件源: 被监听的对象, 通常是 HTML 代码, 或者 js 对象
- 事件: 用来监听事件源状态变化的 (相当于一个监听器), js 提供了事件供开发者使用

- 响应行为：一旦监听到事件源状态变化，需要执行的代码----需要我们写代码

2、常用的事件(监听器)

事件(监听器)	监听的状态
onclick	监听单击事件
ondblclick	监听双击事件
onsubmit	监听表单提交，用于 form 标签上
onchange	监听域内容改变，例如：文本框的值变化了，下拉框的值变化。通常用于下拉框 select 标签的值变化
onload	监听加载完成事件，通常用于监听页面加载完成
onfocus	监听获取焦点（光标）
onblur	监听失去焦点（光标）
onkeydown	监听键盘按键被按下：可以监听所有按键
onkeyup	监听键盘按键被弹起
onkeypress	监听键盘按键被按下或按住：只能监听有显示字符的按键
onmouseover	监听鼠标移入
onmouseout	监听鼠标移出
onmousedown	监听鼠标按键被按下
onmouseup	监听鼠标按键被弹起
onmousemove	监听鼠标移动

3、事件的绑定方式

例如：一个按钮，单击弹窗。 事件源：按钮。 事件： onclick。 响应行为：弹窗 alert()

<!--事件绑定方式一： 事件和响应行为 完全 嵌入到事件源标签中-->

```
<input type="button" value="按钮 1" onclick="alert()">
```

<!--事件绑定方式二： 响应行为封装成函数， 事件嵌入到事件源中，调用函数-->

```
<input type="button" value="按钮 2" onclick="showWin()">
```

```
<script>
```

```
function showWin() {
```

```
    alert("showWin");
}
</script>
```

<!--事件绑定方式三： 事件和响应行为 与 事件源完全分离， 使用 js 代码动态绑定事件-->

```
<input type="button" value="按钮 3" id="btn3">
<script>
    //使用 js 给按钮 3 动态绑定事件：js 相关的框架里用的比较多
    //获取到了事件源标签对象
    var btn = document.getElementById("btn3");
    //设置事件的属性
    btn.onclick = function(){
        alert("动态绑定事件");
    }
</script>
```

9-3_js 的 BOM 对象

9-3-1_js 的 BOM 对象

1、什么是 BOM 对象

BOM: Browser Object Model, 浏览器对象模型。是 js 提供的一些对象，通过这些对象可以用来操作浏览器。

window 对象：代表整个浏览器窗口的对象，是所有 BOM 对象的顶级对象。**window.location**

location 对象：是浏览器地址信息对象，可以用来操作当前网址：获取当前网址，网址跳转

history 对象：是浏览器历史记录信息对象，可以用来进行历史记录切换。前进一步、后退一步

navigator 对象：浏览器导航信息对象（浏览器本身信息），只读，可以获取浏览器的内核、版本、操作系统等信息

screen 对象：浏览器屏幕信息对象，可以用来获取浏览器窗口显示区域的大小、位置颜色信息等等

9-3-2_window

1、提供了弹窗的方法

- 普通弹窗： **alert(string)** 没有返回值
- 确认弹窗： **confirm(string)** 返回 **boolean** 的结果
- 输入弹窗： **prompt(string)** 返回输入的内容。如果点击取消的话返回 **null**

2、提供了定时器

1、执行多次的定时器

设置: `var timer = setInterval(函数对象, 间隔毫秒值);`

清除: `clearInterval(timer);`

2、执行一次的定时器（延时器）

设置: `var timer = setTimeout(函数对象, 延迟毫秒值);`

清除: `clearTimeout(timer)`

3、提供了一些全局函数

`parseInt(string):` 转换成整数

`parseFloat(string):` 转换成小数

`eval(string):` 把字符串作为 js 代码执行

10-JavaScript 下

总结:

- 会使用 `location` 进行网址跳转
- DOM-能够使用获取标签的 4 个方法
- DOM-能够使用 `innerHTML` 操作标签体
- DOM-能够获取标签的属性值 和 设置标签的属性值
- 会创建数组对象，并进行遍历
- 会使用提供好的正则表达式校验字符串

10-1_上节课回顾

10-1-1_ECMAScript

1、定义变量: `var`

2、数据类型

原始数据类型

`boolean, number, string, undefined, object`

引用数据类型: 9 个内置类

3、运算符

- `==`和`===`: `==`只比较值, `==`比较值和类型

- `+*/`: 加法有字符串是拼接字符串, `-*/`有字符串 js 会尝试转换成数字再运算
- `&&`和`||`: js 的 boolean 运算里只有`&&`和`||`, 没有`&`和`|`

4、逻辑语句

`if` 判断: 可以是任何类型的条件。`false, 0, "", null, undefined` 是 `false`, 其它是 `true`

5、函数

普通函数

```
function 函数名(形参 1, 形参 2,...){
    //函数的代码
    //return: 如果需要有返回值就 return
}
var result = 函数名(实参 1, 实参 2,...)
```

匿名函数

```
var fn = function 函数名(形参 1, 形参 2,...){
    //函数的代码
    //return: 如果需要有返回值就 return
}
```

把匿名函数赋值给一个变量, 通过变量名调用匿名函数

把匿名函数作为另外一个实参传递进去

6、事件

常用的 4 个事件

`onclick, onsubmit, onchange, onload`

事件的绑定方式

```
<input type="button" onclick="alert()" value="按钮">

<input type="button" onclick="show()" value="按钮">
function show(){
    alert("....");
}

<input type="button" id="btn3" value="按钮">
var btn = document.getElementById("btn3");
```

```
btn.onclick = function(){
    alert();
}
```

10-1-2_BOM

1、5个BOM对象

window, location, history, navigator, screen

2、window对象

提供了弹窗的方法

普通弹窗: `alert(string)`

确认弹窗: `confirm(string)` 返回 boolean

输入弹窗: `prompt(string)` 返回的是输入的内容, 如果取消返回 null

提供了定时器

执行多次的定时器

`var timer = setInterval(函数对象, 间隔毫秒值);`

`clearInterval(timer)`

执行一次的延时器

`var timer = setTimeout(函数对象, 延迟毫秒值);`

`clearTimeout(timer);`

提供了一些全局函数

`parseInt()`: 转换成整数

`parseFloat()`: 转换成小数

`eval()`: 把字符串作为js代码执行

10-2_BOM对象

10-2-1_location对象

是浏览器地址信息对象, 可以用来获取当前网址, 或者是设置当前网址(页面跳转)

- 获取当前网址: `var url = location.href;`
- **设置当前网址: `location.href = "http://www.itcast.cn";`**
- 刷新当前页: `location.reload();`

10-2-2_history对象

是浏览历史记录信息对象, 可以用来进行历史记录切换: 前进一步、后退一步

- 前进一步: `history.forward()`
- 后退一步: `history.back()`
- 切换 n 步: `history.go(n)`: n 为正数, 表示前进 n 步; n 为负数, 表示后退 n 步

10-3_DOM 对象

10-3-1_什么是 DOM 对象

DOM: Document Object Model, 文档对象模型。可以让我们通过 js 代码, 调用 dom 对象的属性和方法, 来操作 HTML 文档。比如: 操作标签、操作标签的属性、操作标签体 (开始标签和结束标签中间的内容)。

一个网页要想显示到浏览器上, 就必须要被浏览器加载到内存中, 这个页面在浏览器内存中会按照 HTML 标签的结构, 形成一棵 dom 树。HTML 里所有的标签、标签上的属性、文本全部都会转换成 dom 树上的节点。

10-3-2_操作标签

1、获取标签

- `document.getElementById(id)`: 获取到 id 对应的标签对象
- `document.getElementsByName(name 属性值)`: 根据 name 属性值获取多个标签对象
在获取 select 下拉列表时, 获取的值是 option 选中的值。
- `document.getElementsByTagName(标签名称)`: 根据标签名称获取多个标签对象。
如果 option 没有 value 属性, 那么获取的是 option 标签的文本内容
- `document.getElementsByClassName(类名)`: 根据类名获取多个标签对象

这些方法, 除了 document 对象可以用, 表示从整个 HTML 文档里查找标签;

Element 对象也可以使用, 表示从指定的标签里查找标签

2、创建标签

```
var tag = document.createElement("div");
```

注意: 创建出来的标签, 不会显示到页面上, 需要插入到 dom 树才会显示

3、插入标签

父标签对象.`appendChild(子标签对象)`

注意: 插入的方法 `appendChild` 是剪切移动的方式

4、删除标签

标签对象.`remove()`

5、操作标签体

标签体：开始标签和结束标签中间的全部内容

- 获取标签体：`var content = 标签对象.innerHTML`
- 设置标签体：`标签对象.innerHTML = "新的标签体内容";`(覆盖原来的内容)
- 获取标签体内容：`var content = 标签对象.innerText;`
- 设置标签体内容：`标签对象.innerText = “新的标签体内容”;`(html 代码也会显示在页面)

练习：把 city 里所有的下拉选项清除，替换显示成提示的选项

```
<select name="city" id="city">  
    <option value="">--请选择--</option>  
    <option>天津</option>  
    <option>北京</option>  
    <option value="sh" id="sh">上海</option>  
    <option value="cq" id="cq">重庆</option>  
</select>  
<script>  
    var city = document.getElementById("city");  
    city.innerHTML = "<option value="">--大帅哥--</option>"  
</script>
```

10-3-3_操作属性

1、获取属性

方式一：`var attrValue = 标签对象.属性名`

方式二：`var attrValue = 标签对象.getAttribute("属性名")`

2、设置属性

方式一：`标签对象.属性名 = 值;`

方式二：`标签对象.setAttribute("属性名", 属性值)`

3、删除属性

`标签对象.removeAttribute("属性名")`, 不能删除自定义的属性。

4、操作属性的注意事项

如果操作属性名称是保留关键字，通过“`.属性名`”的方式操作不了，要使用 `getAttribute/setAttribute`

如果操作的是自定义的属性，而不是 HTML 标签提供的属性，要使用 `getAttribute/setAttribute`

操作 `class` 属性，特殊的操作方式：`标签对象.className` 进行操作的

10-4_js 的引用数据类型

10-4-1_引用数据类型(内置类)

JavaScript 对象
JS Array
JS Boolean
JS Date
JS Math
JS Number
JS String
JS RegExp
JS Functions
JS Events

10-4-2_Array

定义数组

```
//int[] arr = new int[3];  
  
var arr1 = new Array(); //长度为 0 的数组  
  
var arr2 = new Array(3); //长度为 3 的数组  
  
var arr3 = new Array("a", "b", "c"); //创建数组并初始化数据
```

常用属性

arr.length: 获取数组的长度

arr.length = 5 设置数组的长度

数组循环遍历

for 循环

数组常用方法

var myarr = arr.concat(arr1, arr2,...): 把多个数组合并成一个

var str = arr.join(","): 把数组拼接成字符串

arr.reverse(): 颠倒数组里元素的顺序, 直接在原数组基础上颠倒, 没有返回值

10-4-3_Date

创建 Date 对象

```
var date = new Date(); //创建日期对象, 当前日期  
  
var date = new Date(year, month, date); //创建指定日期对象, 注意月从 0 开始  
  
var date = new Date(year, month, date, hour, minute, second)
```

常用方法

var ms = date.getTime(); 获得毫秒值

10-4-4_Math

不需要创建对象，直接使用 Math 的属性和方法

```
var v = Math.PI;  
v = Math.abs(-3);  
v = Math.ceil(3.0000001);  
v = Math.floor(3.9999999);  
v = Math.round(3.50);  
v = Math.max(3, 5);  
v = Math.min(3, 5);  
v = Math.pow(2, 10);  
v = Math.random(); // [0, 1]
```

10-4-5_RegExp

创建正则表达式对象

```
var reg = new RegExp("正则表达式");  
var reg = /正则表达式/;
```

常用方法

```
varisOk = reg.test(str);  
参数 str: 要校验的字符串  
返回值: boolean, true 表示校验通过, false 表示校验不通过
```

11-Bootstrap

11-1_上节课回顾

11-1-1_BOM

1、location 对象

- 获取网址: var url = location.href;
- 设置网址: location.href = "网址";
- 刷新页面: location.reload()

2、history 对象

- history.forward()
- history.back()
- history.go(n) 正数前进, 负数后退

11-1-2_DOM

1、操作标签

获取标签
<code>document.getElementById</code>
<code>document.getElementsByName</code>
<code>document.getElementsByTagName</code>
<code>document.getElementsByClassName</code>
创建标签
<code>document.createElement("option")</code>
插入标签
父标签对象. <code>appendChild</code> (子标签对象)
删除标签
标签对象. <code>remove()</code>
操作标签体
获取标签体： <code>var inner = 标签对象.innerHTML</code>
设置标签体： <code>标签对象.innerHTML = "<h1>HTML 代码会生效</h1>"</code> ;

2、操作属性

获取属性
<code>var v = 标签对象.属性名</code>
<code>var v = 标签对象.getAttribute("标签名称")</code>
设置属性
<code>标签对象.属性名 = 值;</code>
<code>标签对象.setAttribute("标签名称", 值)</code>
删除属性
<code>标签对象.removeAttribute("标签名称")</code>

11-1-3_引用数据类型

1、Array

定义数组:

```
var arr = new Array();  
var arr = new Array(5);  
var arr = new Array("a", 1, true);  
  
var arr = ["a", 1, true];
```

数组常用的属性:

length: 获取数组的长度, 设置数组的长度

数组常用的方法:

arr.concat(arr1, arr2,...) 把多个数组合并成一个, 返回结果

arr.reverse() 数组顺序反转, 没有返回值, 在原数组基础上反转

arr.sort() 数组排序, 没有返回值, 在原数组基础上排序

2、Date

定义对象

```
var date = new Date();  
var date = new Date(年,月,日);    注意: 月是从 0 开始  
var date = new Date(年,月,日, 时,分,秒);
```

常用的方法

getTime() 获取毫秒值

toLocaleString() 转换成本地日期格式的字符串

3、Math

不需要创建对象, 直接使用 **Math** 的属性和方法

常用的方法:

abs()求绝对值

ceil()向上取整

floor()向下取整

round()四舍五入

random()取随机数 [0, 1)

4、RegExp

创建对象:

```
var reg = new RegExp("表达式字符串");
var reg = /表达式/;

常用方法:

test(string): 校验字符串是否符合正则表达式。返回 boolean
```

11-2_Bootstrap 简介

11-2-1_什么是 Bootstrap

Bootstrap: 是一个基于 HTML、CSS、JavaScript 的前端框架。它预定义好了一套 CSS 样式、和与 CSS 样式对应的 JavaScript (jQuery) 代码，我们只要提供固定的 HTML 结构，添加固定的 class 就可以完成指定的效果。

Bootstrap 使得 web 前端开发更加快捷、代码优雅、美观大方。

优雅的代码：

逻辑简洁

结构清晰：格式化

注释明了

命名规范

11-2-2_响应式布局设计

Bootstrap 最大的优点是可以实现响应式布局。

响应式布局：是随着移动互联网的出现而产生的一个概念，指：一套页面能够兼容不同的终端设备，而不需要为每种设备做特定的版本。

Bootstrap 通过 jQuery 解决了不同浏览器的兼容问题，通过媒体查询 (@media) 解决了不同终端的适配问题。

Bootstrap 框架是移动设备优先的，但是 **Bootstrap** 是响应式布局里最成功的一个框架了

响应式页面示例：苹果官网 <http://www.apple.com.cn>

11-2-3_Bootstrap 下载与目录结构

下载地址：www.bootcss.com

目录结构：

js: Bootstrap 框架里写好的 JavaScript 代码，实现了页面动态效果或者特效

fonts: Bootstrap 的字体图标

css: Bootstrap 框架里写好的样式代码

web 应用引入 Bootstrap：

把三个文件夹拷贝到 web 应用里，注意三个文件夹的目录结构层次不要变

11-2-4 _Bootstrap 页面模板

在使用 **Bootstrap** 开发前端页面时，绝大多数时候是不需要自己去写 **HTML**、**CSS** 和 **JS** 的。**Bootstrap** 提供了一套写好的页面模板，可以直接拿来使用。在 **Bootstrap** 的文档：起步-->基本模板里，如下：

```
<!DOCTYPE html>
<html lang="zh-CN">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Bootstrap 101 Template</title>

    <link href="css/bootstrap.min.css" rel="stylesheet">
</head>
<body>

<script src="js/jquery-3.2.1.min.js"></script>
<script src="js/bootstrap.min.js"></script>
</body>
</html>
```

11-3_栅格系统

在开发前端页面时，页面布局是基础的、重要的一步操作，也是非常复杂的。特别是对于非前端开发人员（例如我们）来说，就更为困难了。

Bootstrap 提供了一种页面布局的方式，它综合了 **table** 表格布局的简单易用、和 **div+css** 布局的高性能形成了一种新的布局方式，叫：栅格系统。

栅格系统的本质仍然是 **div+css**，但是在使用时更接近 **table** 表格。

对应 **table** 标签：栅格系统里提供了容器 **container**, **container-fluid**

对应 **tr** 标签：栅格系统里提供了行 **row**

对应 **td** 标签：栅格系统里提供了单元格 **col-*-***

11-3-1_栅格布局的容器

类似于表格的 **table** 标签。

Bootstrap 提供了两种容器：**container** 和 **container-fluid**。

1、container 容器

容器的宽度是阶段变化的固定宽度。它的宽度会随着设备类型而改变。所谓设备类型，其实就是浏览器页面的宽度

在 lg 类型设备中，container 容器宽 1170px

lg: 浏览器宽[1200px, ∞)时，是 lg 类型的设备

在 md 类型设备中，container 容器宽 970px

md: 浏览器宽[992px, 1200px)时，是 md 类型的设备

在 sm 类型设备中，container 容器宽 750px

sm: 浏览器宽[768px, 992px)时，是 sm 类型的设备

在 xs 类型设备中，container 容器宽 100%

xs: 浏览器宽[0, 768px)时，是 xs 类型的设备

2、container-fluid 容器

无论浏览器是什么类型的设备（即：无论浏览器的宽度是多少），container-fluid 的宽度始终是 100%

11-3-2_栅格布局的行

栅格系统的行，类似于表格的 tr 标签。一个容器中可以有多个行

1、一行分为 12 份

栅格系统把一行分为 12 份，即：一行最多有 12 个单元格

11-3-3_栅格布局的单元格 col-*-*

栅格系统的单元格类似于表格的 td，一行可以有多个单元格（一行最多有 12 个单元格）。

col-*-*:

第一个*: 表示设备类型

第二个*: 表示单元格占几份

例如：col-md-3 表示单元格在 md 类型的设备时，要占 3 份的大小，即：3/12 的大小

11-3-4_单元格样式

hidden-*: 单元格在指定设备类型里隐藏，其它设备类型不隐藏

visible-*: 单元格在指定设备的类型里显示，其它设备类型不显示

11-4_Bootstrap 常用样式

11-4-1_按钮

<!--

01.按钮

-->

```
<a class="btn btn-default" href="#" role="button">链接按钮</a>
<button class="btn btn-default" type="submit" onclick="alert('button 标签按钮')">Button 标签按钮</button>
<input class="btn btn-default" type="button" value="Input 按钮">
<input class="btn btn-default" type="submit" value="Submit 按钮">
<!-- Standard button -->
<button type="button" class="btn btn-default"> (默认样式) Default</button>

<!-- Provides extra visual weight and identifies the primary action in a set of buttons -->
<button type="button" class="btn btn-primary"> (首选项) Primary</button>

<!-- Indicates a successful or positive action -->
<button type="button" class="btn btn-success"> (成功) Success</button>

<!-- Contextual button for informational alert messages -->
<button type="button" class="btn btn-info"> (一般信息) Info</button>

<!-- Indicates caution should be taken with this action -->
<button type="button" class="btn btn-warning"> (警告) Warning</button>

<!-- Indicates a dangerous or potentially negative action -->
<button type="button" class="btn btn-danger"> (危险) Danger</button>

<!-- Deemphasize a button by making it look like a link while maintaining button behavior -->
<button type="button" class="btn btn-link"> (链接) Link</button>
```

11-4-2_表单

<!--

02. 表单

-->

```
<div class="container">
  <div class="row">
    <div class="col-sm-4"></div>
    <div class="col-sm-4">
      <form action="#" method="get">
```

```
<div class="form-group">
    <label for="exampleInputEmail1">Email</label>
    <input name="email" type="email" class="form-control" id="exampleInputEmail1"
placeholder="Email">
</div>

<div class="form-group">
    <label for="exampleInputPassword1">密码</label>
    <input name="password" type="password" class="form-control"
id="exampleInputPassword1" placeholder="Password">
</div>

<div class="form-group">
    <label for="exampleInputFile">选择文件</label>
    <input name="file" type="file" id="exampleInputFile">
    <p class="help-block">Example block-level help text here.</p>
</div>

<div class="checkbox">
    <label>
        <input type="checkbox" name="agree" value="yes">同意我们的许可协议
    </label>
</div>

<div class="form-group">
    <label for="jieshao">个人介绍</label>
    <textarea class="form-control" rows="3" id="jieshao"></textarea>
</div>

<div class="form-group">
    <label>性别</label>
    <label class="radio-inline">
        <input type="radio" name="sex" id="inlineRadio1" value="male"> 男
    </label>
    <label class="radio-inline">
        <input type="radio" name="sex" id="inlineRadio2" value="female"> 女
    </label>
</div>

<select class="form-control">
    <option>1</option>
    <option>2</option>
    <option>3</option>

```

```
        <option>4</option>
        <option>5</option>
    </select>
    <button type="submit" class="btn btn-default">Submit</button>
</form>
</div>
<div class="col-sm-4"></div>
</div>
</div>
```

11-4-3 表格

```
<!--
03 表格
-->
<div style="width:500px;">
    <table class="table table-striped table-bordered table-hover">
        <tr >
            <th>学号</th>
            <th>姓名</th>
            <th>年龄</th>
            <th>性别</th>
        </tr>
        <tr class="danger">
            <td>1</td>
            <td>张三</td>
            <td>19</td>
            <td>男</td>
        </tr>
        <tr>
            <td>2</td>
            <td></td>
            <td></td>
            <td></td>
        </tr>
        <tr>
            <td>3</td>
```

```
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</table></div>
```

11-4-4_图片

```
<!--
04. 图片
-->
![...](img/mm.jpg)
![...](img/mm.jpg)
![...](img/mm.jpg)
```

11-5_Bootstrap 常用组件

11-5-1_字体图标

```
<!--
01. 字体图标。通常是使用 i 标签显示字体图片。i: icon
-->
</span>


</div>
</i>


```

11-5-2_导航条

```
<!--
02. 导航条
-->

```

```
<!-- Brand and toggle get grouped for better mobile display -->

<div class="navbar-header">

    <!--三明治按钮-->

    <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#bs-example-navbar-collapse-1" aria-expanded="false">

        <!--sr: screen read-->

        <span class="icon-bar"></span>
        <span class="icon-bar"></span>
        <span class="icon-bar"></span>

    </button>

    <a class="navbar-brand" href="#">首页</a>

</div>

<!-- Collect the nav links, forms, and other content for toggling -->

<div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">

    <ul class="nav navbar-nav">

        <li class="active"><a href="#">电脑办公</a></li>
        <li><a href="#">手机数码</a></li>
        <li class="dropdown">

            <a href="#" class="dropdown-toggle" data-toggle="dropdown" role="button" aria-haspopup="true" aria-expanded="false">下拉菜单 <span class="caret"></span></a>

            <ul class="dropdown-menu">

                <li><a href="#">肠粉</a></li>
                <li><a href="#">奶粉</a></li>
                <li><a href="#">奶粉</a></li>
                <li><a href="#">奶粉</a></li>
                <li><a href="#">奶粉</a></li>
                <li><a href="#">Something else here</a></li>
                <li role="separator" class="divider"></li>
                <li><a href="#">饺子</a></li>
                <li role="separator" class="divider"></li>
                <li><a href="#">One more separated link</a></li>

            </ul>
        </li>
    </ul>
</div>

<form class="navbar-form navbar-left" role="search">

    <div class="form-group">
```

```
<input type="text" class="form-control" placeholder="Search">
</div>
<button type="submit" class="btn btn-default">搜索</button>
</form>
</div><!-- /.navbar-collapse -->
</div><!-- /.container-fluid -->
</nav>

<nav class="navbar navbar-default">
<div class="container-fluid">
    <!-- Brand and toggle get grouped for better mobile display -->
    <div class="navbar-header">
        <!--三明治按钮-->
        <button type="button" class="navbar-toggle collapsed" data-toggle="collapse" data-target="#bs-example-navbar-collapse-1" aria-expanded="false">
            <!--sr: screen read-->
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
            <span class="icon-bar"></span>
        </button>
        <a class="navbar-brand" href="#">首页</a>
    </div>
    <!-- Collect the nav links, forms, and other content for toggling -->
    <div class="collapse navbar-collapse" id="bs-example-navbar-collapse-1">
        <ul class="nav navbar-nav">
            <li class="active"><a href="#">电脑办公</a></li>
            <li><a href="#">手机数码</a></li>
            <li><a href="#">手机数码</a></li>
        </ul>
    </div>
</div>
```

```
</ul>

</div><!-- /.navbar-collapse -->

</div><!-- /.container-fluid -->

</nav>
```

11-5-3_分页条

```
<!--

03. 分页条

-->

<nav>

  <ul class="pagination">

    <li class="disabled">
      <a href="#" aria-label="Previous">
        <span aria-hidden="true">&laquo;</span>
      </a>
    </li>

    <li><a href="#">1</a></li>

    <li><a href="#">2</a></li>

    <li class="active"><a href="#">3</a></li>

    <li><a href="#">4</a></li>

    <li><a href="#">5</a></li>

    <li>
      <a href="#" aria-label="Next">
        <span aria-hidden="true">&raquo;</span>
      </a>
    </li>

  </ul>

</nav>
```

11-6_Bootstrap 常用插件

11-6-1_模态窗口

```
<!--

01. 模态窗口

-->

<!-- Button trigger modal -->
```

```

<button type="button" class="btn btn-primary btn-lg" data-toggle="modal" data-target="#myModal">
    弹出模态窗口
</button>

<!-- Modal -->

<div class="modal fade" id="myModal" tabindex="-1" role="dialog" aria-labelledby="myModalLabel">
    <div class="modal-dialog" role="document">
        <div class="modal-content">
            <div class="modal-header">
                <button type="button" class="close" data-dismiss="modal" aria-label="Close"><span
aria-hidden="true">&times;</span></button>
                <h4 class="modal-title" id="myModalLabel">容器标题</h4>
            </div>
            <div class="modal-body">
                窗口内容
            </div>
            <div class="modal-footer">
                <button type="button" class="btn btn-default" data-dismiss="modal">关闭</button>
                <button type="button" class="btn btn-primary">保存</button>
            </div>
        </div>
    </div>
</div>

```

11-6-2_轮播图 Carousel

```

<!--
03. 轮播图
-->

<div id="carousel-example-generic" class="carousel slide" data-ride="carousel">
    <!-- Indicators -->
    <ol class="carousel-indicators">
        <li data-target="#carousel-example-generic" data-slide-to="0" class="active"></li>
        <li data-target="#carousel-example-generic" data-slide-to="1"></li>
        <li data-target="#carousel-example-generic" data-slide-to="2"></li>
    </ol>

```

```

<!-- Wrapper for slides -->

<div class="carousel-inner" role="listbox">

    <div class="item active">
        
        <div class="carousel-caption">
            兵马俑
        </div>
    </div>

    <div class="item">
        
        <div class="carousel-caption">
            北京一日游
        </div>
    </div>

    <div class="item">
        
    </div>
</div>

<!-- Controls -->

<a class="left carousel-control" href="#carousel-example-generic" role="button" data-slide="prev">
    <span class="glyphicon glyphicon-chevron-left" aria-hidden="true"></span>
    <span class="sr-only">Previous</span>
</a>

<a class="right carousel-control" href="#carousel-example-generic" role="button" data-slide="next">
    <span class="glyphicon glyphicon-chevron-right" aria-hidden="true"></span>
    <span class="sr-only">Next</span>
</a>
</div>

```

12-注解和反射

总结：

反射-类加载器的获取方式和作用

获取类的字节码对象 **Class** 的三种方式

获取并操作构造方法对象 **Constructor**

获取并操作方法对象 Method

获取并操作属性对象 Field

注解-JDK 提供的常用注解: @Override, @Deprecated, @SupressWarning

注解-JDK 提供的元注解: @Target, @Retention

尝试自己定义一个注解

反射时关于注解解析的两个方法: isAnnotationPresent, getAnnotation

注解的应用

12-1_反射

12-1-1_反射概述

1、什么是反射

反射是一种机制，利用该机制可以在程序运行过程中对类进行解剖并操作类中的方法，属性，构造方法等成员。

调用方法的方式:

正常的方式： 对象.方法名()

反射的方式： 另外一种调用某一个对象的方法的方式

反射： 是另外一种调用方法或者属性的方式。我们都知道正常调用一个方法，可以使用以下方式： 对象名.方法名(实参...); 那么除了这种正常的方式之外，还有其它的方式也能调用方法，就是： 反射方式。

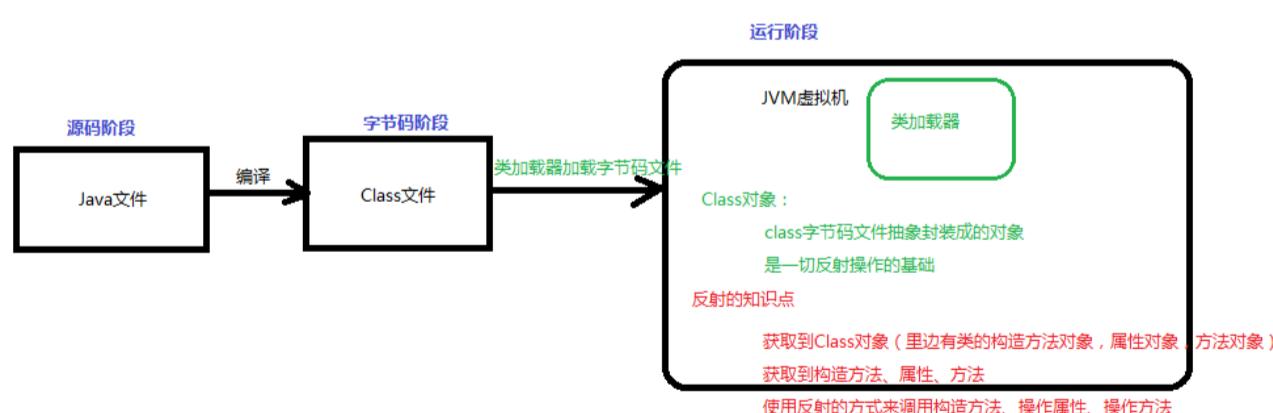
2、反射的原理

一个 Java 类要运行，需要经历三个阶段

源码阶段： Java 文件

字节码阶段： class 文件

运行阶段： 在 jvm 中运行



3、类加载器

什么是类加载器：一个 class 文件要想运行，就必须要把 class 文件加载到 JVM 内存中，这个加载的工作就是由类加载器负责完成的。

JVM 提供的三种类加载器：

Bootstrap 类加载器：引导类加载器，是整个 JVM 的核心类加载器。

加载 Java 的核心类: rt.jar

Extension 类加载器: 扩展类加载器

加载 Java 的一些扩展类: JAVA_HOME\jre\lib\ext*.jar

System 类加载器: 系统类加载器/应用类加载器

加载第三方 jar 包, 或者是我们写的 class 类

Bootstrap 类加载器先执行--> 扩展类加载器 --> 应用类加载器

怎样获取类加载器:

ClassLoader classLoader = 类名.class.getClassLoader();

ClassLoader classLoader = 对象.getClass().getClassLoader();

作用:

用来加载 class 到 JVM 内存中

类加载器对象提供了一个方法, 可以从 src 下读取文件:

InputStream is = getResourceAsStream(String filename)

反射的原理

Java 里一切皆对象。当一个 class 文件被加载到 JVM 内存之后:

class 文件本身会被封装成一个对象: **java.lang.Class 对象**。

Class 对象里维护了这个类的信息, 例如: 类的名称、有哪些方法、有哪些属性等等。

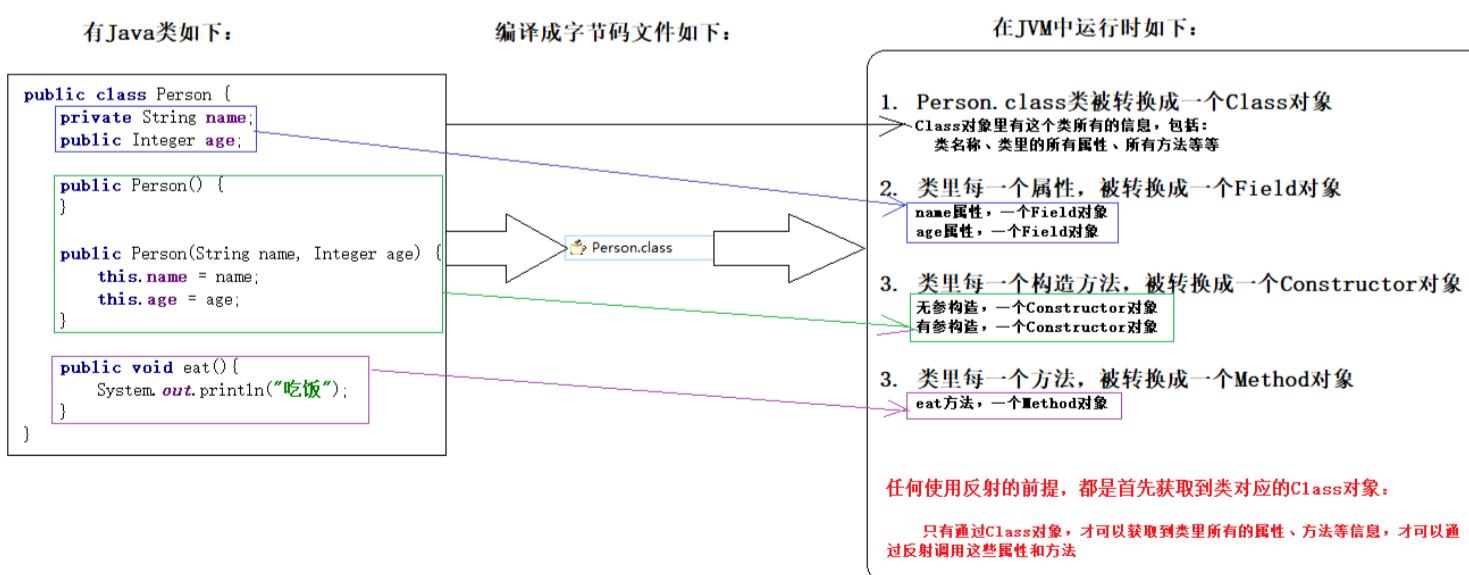
类里的每一个方法, 会被封装成一个对象: **java.lang.reflect.Method**

类里的每一个属性, 会被封装成一个对象: **java.lang.reflect.Field**

类里的每一个构造方法, 会被封装成一个对象: **java.lang.reflect.Constructor**

使用反射的前提就是: 首先获取到类对应的 Class 对象

我们可以通过操作 Class、Method、Field、Constructor 对象, 来操作一个类、类的方法、类的属性、类的构造方法。这种操作方式就是反射操作。



4、反射的用途

在 eclipse 或者 idea 里写 Java 代码时, 有自动提示功能, 列出对象可用的方法。就是使用反射的方式, 获取到了这个对象里的所有方法对象。

在后边要学习的框架底层，也大量的运用了反射。（框架底层应用了反射，但是我们在使用框架时很少使用反射）

5、反射演示的 Java 类准备

```
package com.itheima.reflect;

public class Person {
    private String name;
    public Integer age;

    public Person() {
    }

    public Person(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    public void eat(){
        System.out.println("吃饭");
    }

    private String sing(String song){
        return "唱歌:" + song;
    }
}
```

12-1-2_获取 Class 对象

1、Class 相关的 API

获取 Class 对象的三种方法：

返回值	方法名称	参数说明	方法描述
Class<Person>	类名.class		根据类名得到对应的 Class 对象
Class<? extends Person>	对象实例.getClass()		根据对象获取对应的 Class 对象
Class<?>	Class.forName(String className)	className: 类的全路径	根据类全路径获取对应的 Class 对象

Class 常用的 API

返回值	方法名称	参数说明	方法描述
String	getName()		获取类的全路径
Object	newInstance()		调用类的无参构造，创建实例对象

Class 和 Object 涉及到的 API

获取类对象的 3 种方式：

- 1、类名.class
- 2、对象实例.getClass()
- 3、Class.forName("类的全路径");

java.lang.Class 类

public String getName()：以 String 形式返回此 Class 对象表示的实体名称（类，接口，数组类，基本类型或 void）。

java.lang.Object 类

public final Class<?> getClass()：返回对象的运行时类，返回的类对象是由类的静态同步方法修饰

public static Class<?> forName(String className) throws ClassNotFoundException

返回类的对象通过被给出的类名称和接口名称，等价于 **Class.forName(className, true, currentLoader)**

currentLoader 就是当前类的类加载器：CurrentClass.class.getClassLoader()

@Deprecated(since="9")

public T newInstance() throws InstantiationException, IllegalAccessException 调用类的无参构造，创建实例对象

被 **clazz.getDeclaredConstructor().newInstance()** 替换

2、通过类名.class 获取 Class

```
//获取 Person 字节码对象 Class
Class clazz = Person.class;
//获取到类的全路径名称
String clazzName = clazz.getName();
System.out.println(clazzName);
```

3、通过对象.getClass()获取 Class

```
Person p = new Person();
//获取 Person 字节码对象 Class
Class clazz = p.getClass();
```

```
//获取到类的全路径名称
String clazzName = clazz.getName();
System.out.println(clazzName);
```

4、通过 Class.forName() 获取 Class

```
//获取 Person 字节码对象 Class
Class clazz = Class.forName("com.itheima.reflect.Person");
System.out.println(clazz);
```

12-1-3_获取 Constructor 对象

1、Constructor 相关的 API

从 Class 对象获取 Constructor 对象的方法 API:

返回值	方法名称	参数说明	方法描述
Constructor	getConstructor(Class... parameterTypes)	parameterTypes: 构造方法的参数类型 Class 对象	根据参数类型获取对应的构造方法

通过反射调用 Constructor 的 API

返回值	方法名称	参数说明	方法描述
Object	newInstance(Object... initargs)	initargs: 调用构造方法时需要的实参	调用有参构造方法，创建对象实例
T	newInstance()		调用无参构造方法，创建对象实例

通过类对象获取 Constructor 对象:

public Constructor<T> getConstructor(Class<?>... parameterTypes)

throws NoSuchMethodException,SecurityException

返回一个 **Constructor** 对象，该对象反映此 Class 对象所表示的类的指定公共构造函数。

public Constructor<?>[] getConstructors() throws SecurityException

返回一个包含 **Constructor** 对象的数组，这些对象反映此 Class 对象所表示的类的所有公共构造函数

通过 Constructor 对象实例化类:

public T newInstance()

被下面的方式替代

clazz.getDeclaredConstructor().newInstance()

2、获取并操作无参构造 Constructor 对象

```
//获取 Person 字节码对象 Class  
Class clazz = Class.forName("com.itheima.reflect.Person");  
  
//获取到 Person 类的无参构造方法  
Constructor constructor = clazz.getConstructor();  
  
//通过反射调用无参构造，创建对象实例  
Object obj = constructor.newInstance();  
  
System.out.println(obj);
```

3、获取并操作有参构造 Constructor 对象

```
//获取 Person 字节码对象 Class  
Class clazz = Class.forName("com.itheima.reflect.Person");  
  
//获取到 Person 类的有参构造方法  
Constructor constructor = clazz.getConstructor(String.class, Integer.class);  
  
//通过反射调用有参构造，传递实参，创建对象实例  
Object obj = constructor.newInstance("xiaoming", 25);  
  
Person p = (Person) obj;  
  
System.out.println("对象 p 的年龄是: " + p.age);
```

12-1-4_获取 Method 对象

1、Method 相关的 API

从 Class 对象获取 Method 对象的方法 API:

返回值	方法名称	参数说明	方法描述
Method	getMethod(String name, Class...parameterTypes)	name: 方法名称 parameterTypes: 方法的参数类型 Class 对象	根据方法名称、方法参数类型获取方法 Method 对象 只能获取 public 类型的方法
Method	getDeclaredMethod(String name, Class... parameterTypes)	name: 方法名称 parameterTypes: 方法的参数类型 Class 对象	根据方法名称、方法参数类型获取方法 Method 对象 可以获取任意类型的方法

通过反射调用方法的 API

返回值	方法名称	参数说明	方法描述
void	setAccessible(boolean flag)	flag: 是否允许暴力执行 Method 对象	private 类型的方法不能直接调用，必须要先设置允许暴力执行后，才可以反射调用
Object	invoke(Object obj, Object... args)	obj: 类对象的实例，执行哪个对象的方法 args: 执行方法需要的实参	通过反射执行方法，相当于： obj.方法(Object... args)

获取类对象方法:

```
java.lang.Class:  
    public Method getMethod(String name, Class<?>... parameterTypes)  
    public Method getDeclaredMethod(String name, Class<?>... parameterTypes)  
    public Method[] getDeclaredMethods()
```

获取类对象私有方法的前提:

```
java.lang.reflect.Method:  
    public void setAccessible(boolean flag)
```

2、获取并操作 **public** 方法 **Method** 对象

```
//获取 Person 字节码对象 Class  
Class clazz = Class.forName("com.itheima.reflect.Person");  
//获取 Person 的 eat 方法 (eat 是 public 类型的方法, 无参, 无返回值)  
Method eatMethod = clazz.getMethod("eat");  
//创建一个 Person 对象  
Object obj = clazz.newInstance();  
//通过反射调用 obj 对象的 eat 方法  
eatMethod.invoke(obj);
```

3、获取并操作任意方法 **Method** 对象

```
//获取 Person 字节码对象 Class  
Class clazz = Class.forName("com.itheima.reflect.Person");  
//获取 Person 的 sing 方法 (sing 是 private 类型的方法, 有 String 参数, 有 String 返回值)  
Method singMethod = clazz.getDeclaredMethod("sing", String.class);  
//设置方法允许暴力执行 (如果不设置, 不能反射调用 private 方法)  
singMethod.setAccessible(true);  
//创建一个 Person 对象  
Object obj = clazz.newInstance();  
//通过反射调用 obj 对象的 sing 方法, 得到返回值 result  
Object result = singMethod.invoke(obj, "凉凉");  
System.out.println(result);
```

12-1-5_获取 Field 对象

1、Field 相关的 API

从 Class 对象获取 Field 对象的方法 API:

返回值	方法名称	参数说明	方法描述
Field	getField(String name)	name: 属性名称	根据属性名称获取属性对象 Field 获取 public 类型的属性
Field	getDeclaredField(String name)	name: 属性名称	根据属性名称获取属性对象 Field 获取任意类型的属性, private 也可以获取

通过反射调用 Field 的 API (更多 API 参考手册):

返回值	方法名称	参数说明	方法描述
void	setAccessible(boolean flag)	flag: 设置是否允许暴力调用该属性	如果是 private 类型的属性, 必须要设置为 true 之后, 才可以通过反射调用
Object	get(Object obj)	obj: 类的对象实例	获取 obj 对象上 Field 属性的值
void	set(Object obj, Object value)	obj: 类的对象实例 value: 要设置给这个属性的值	把 value 值设置给 obj 对象的 Field 属性

获取类对象里面的属性 `java.lang.Class`:

获取 **public** 类型的属性

```
public Field getField(String name)
public Field[] getFields()
```

获取任意类型的属性

```
public Field getDeclaredField(String name)
public Field[] getDeclaredFields()
```

获取和设置属性值 `java.lang.reflect.Field`:

```
public Object get(Object obj)
public void set(Object obj, Object value)
```

2、获取并操作 **public** 类型的 Field

```
//获取 Person 字节码对象 Class
Class clazz = Class.forName("com.itheima.reflect.Person");
//获取 Person 类的 age 属性 (age 是 public 类型的属性)
Field ageField = clazz.getField("age");
//创建一个 Person 对象
Object obj = clazz.newInstance();
//设置 age 属性值为: 18
```

```
ageField.set(obj, 18);
//获取 age 属性值
Object result = ageField.get(obj);
System.out.println(result);
```

3、获取并操作任意类型的 Field

```
//获取 Person 字节码对象 Class
Class clazz = Class.forName("com.itheima.reflect.Person");
//获取 Person 类的 name 属性（name 是 private 类型的属性）
Field nameField = clazz.getDeclaredField("name");
//设置 name 属性允许暴力访问
nameField.setAccessible(true);
//创建一个 Person 对象
Object obj = clazz.newInstance();
//设置 name 属性值为: tom
nameField.set(obj, "tom");
//获取 name 属性值
Object result = nameField.get(obj);
System.out.println(result);
```

12-2_注解

12-2-1_注解概述

1、什么是注解

注解：Annotation，是一种代码级别的说明。它是 JDK1.5 之后引入的一个特性，和类、接口是同一级别的
注意：注解不同于注释。注释是给开发人员看的、没有功能，注解是给计算机使用的、可以实现某些功能

2、注解有什么用

编译检查：例如@Override
代替 XML：例如@WebServlet 是用来代替低版本中 web 应用的 web.xml
辅助生成文档：例如@Documented

3、常见的 JDK 注解

@Override

添加到方法上，表示方法是重写父类的方法，或者是实现了抽象方法

@Deprecated

添加到方法或类上，表示方法或类已经过时，不建议使用

```
@Deprecated  
public void demo11() {  
    System.out.println("不建议使用的方法");  
}
```

```
public static void main(String[] args) {  
    PersonTest t = new PersonTest();  
    t.demo11(); 调用了被@Deprecated标的方法  
}
```

@SupressWarning

压制警告。

定义了但未使用: unused

使用了过时的 API: deprecated

压制所有警告: **@SupressWarning("all")**

```
public void demo10() {  
    String str = "hello"; Variable 'str' is never used  
}
```

```
@SuppressWarnings("unused")  
public void demo10() {  
    @SuppressWarnings("unused")  
    String str = "hello";  
}
```

@author

用于注释里，表示作者是谁

```
/*  
 * @author James Gosling  
 * @author Arthur van Hoff  
 * @author Alan Liu  
 * @see java.text.DateFormat  
 * @see java.util.Calendar  
 * @see java.util.TimeZone  
 * @since JDK1.0  
 */  
  
public class Date implements java.io.Serializable, Cloneable, Comparable<Date>
```

{.....}

12-2-2_怎么自定义注解

1、定义注解的基本语法

```
@元注解  
public @interface 注解名{  
    public 属性类型 属性名称() [default 默认值];  
    ...  
    public 属性类型 属性名称() [default 默认值];  
}
```

2、使用 JDK 的元注解约束自定义注解

什么是元注解:

元注解: **JDK 提供的, 用来约束注解的注解**

常用的 JDK 元注解有两个: **@Target 和@Retention**

@Target 元注解

用来约束注解可以使用在什么地方。

常用的值有:

ElementType.TYPE: 表示自定义注解可以用在类上

ElementType.METHOD: 表示自定义注解可以用在方法上

ElementType.FIELD: 表示自定义注解可以用在属性字段上

用法示例:

```
@Target(ElementType.TYPE)  
public @interface MyTest1 {  
}
```

自定义的注解@MyTest1 就可以加在类上了

@Retention 元注解

用来约束注解可以保留到什么阶段。一个 Java 程序运行有三个阶段: 源码阶段, 字节码阶段, 运行阶段。当我们把自定义注解加在了 Java 代码中, 那么这个注解会保留到哪个阶段呢? 可以通过**@Retention** 元注解来进行配置

常用的值有:

RetentionPolicy.SOURCE: 保留到源码阶段。注解默认就是源码阶段可见

RetentionPolicy.CLASS: 保留到字节码阶段

RetentionPolicy.RUNTIME: 保留到运行阶段

应用示例:

```
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
public @interface MyTest1 {  
}
```

表示这个注解不仅在源码阶段存在，在字节码阶段存在，在运行阶段也会存在

3、注解的属性

设置注解属性的语法

```
public 属性类型 属性名称() [default 默认值];
```

注意：属性名称后边有括号；默认值可有可无

注解属性的类型

注解的属性可用的类型有：

8 种基本数据类型

String 类型

Enum 枚举类型

Class 类型

Annotation 类型

以上类型的数组形式

注解的属性示例

```
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.TYPE)  
public @interface MyTest1 {  
    public String aaa();  
    public int age() default 80;  
}
```

其中定义了两个属性：

aaa 属性：字符串类型的，没有默认值

age 属性：整型，默认值 80

4、注解的应用

注解里有属性需要设置值：

```
@注解名(属性名=值,属性名=值,...)
```

注意：

属性值必须要符合类型

如果属性有默认值，在应用注解时可以不再指定这个属性值

如果注解里只有一个属性值要设置，并且这个属性名称是 **value**，那么可以省略掉 “**value=**” 不写
注解里没有属性需要设置值：

@注解名

12-2-3_怎么解析注解

1、什么是解析注解

注解本身是没有任何功能的，如果我们自定义了一个注解，就必须要在解析注解的过程中，赋予注解一定的功能。

自定义注解的解析，需要配合反射才可以

2、解析注解相关的 API

Method 对象上有注解相关的 API

返回值	方法名称	参数说明	方法描述
boolean	isAnnotationPresent(Class annotationClass)	annotationClass: 注解的 Class 对象	判断 Method 方法上是否有指定的注解存在
Object	getAnnotation(Class annotationClass)	annotationClass: 注解的 Class 对象	获取 Method 方法上指定的注解对象

创建注解 MyTest

```
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
```

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface MyTest { }
```

创建类 MyTestDemo，在类的方法上添加@MyTest

```
public class MyTestDemo {
    @MyTest
    public void demo1(){}
```

```
        System.out.println("MyTestDemo.demo1.....");  
    }  
}
```

创建 MyTestParser，解析注解

```
import java.lang.reflect.Method;  
  
public class MyTestParser {  
    public static void main(String[] args) throws Exception {  
        //获取 MyTestDemo 的字节码对象  
        Class clazz = Class.forName("com.itheima.anno.MyTestDemo");  
        //获取 demo1 方法对象  
        Method demo1Method = clazz.getMethod("demo1");  
        //判断方法上是否有@MyTest 注解存在  
        boolean present = demo1Method.isAnnotationPresent(MyTest.class);  
        //如果有@MyTest 注解，就执行这个方法；否则不执行  
        if (present) {  
            demo1Method.invoke(clazz.newInstance());  
        } else {  
            System.out.println("方法上没有@MyTest 注解， 不执行方法");  
        }  
    }  
}
```

12-3_练习

12-3-1_根据配置文件，使用反射创建一个对象出来

有配置文件 student.properties 保存在 src 目录，内容如下：

```
class=com.jack.reflect.Person  
country=中国  
name=jack  
age=18
```

要求：根据配置文件的内容，创建出来一个 Student 对象

```
public class CreateClassObjectByProp {  
    @Test  
    public void test() throws Exception {
```

```
// 创建加载配置文件的类对象  
Properties prop = new Properties();  
  
// 读取配置文件  
InputStream stream = CreateClassObjectByProp.class.getClassLoader().getResourceAsStream("student.properties");  
  
// 加载配置文件  
prop.load(stream);  
  
// 获取类的全名称  
String aClass = prop.getProperty("class");  
  
// 获取类对象  
Class<?> personClass = Class.forName(aClass);  
  
// 获取方法  
Method eat = personClass.getDeclaredMethod("eat");  
eat.invoke(personClass.newInstance());  
  
}  
}
```

12-3-2_模拟@Test 注解

13-XML 和动态代理

13-1_XML

13-1-1_XML 概述

1、什么是 XML

XML: Extension Markup Language, 可扩展标记语言。和 HTML 类似, 都是由一堆标签组成的, 但是 XML 和 HTML 不同。

XML 和 HTML 的相同点:

都是由一堆标签组成的

标签上都可以增加属性

标签都可以嵌套

XML 和 HTML 的不同点:

HTML 的标签是预定义好的; 而 XML 的标签是自定义的

HTML 的标签有功能; 而 XML 的标签没有功能, 重点是标签里的数据

HTML 和 XML 的语法有区别: XML 的语法规范比 HTML 更严格

2、XML 的作用

作为软件的配置文件：

properties 格式的配置文件，语法简单，解析方便；但不能表示复杂的数据关系

XML 格式的配置文件，语法和解析麻烦；可以表示复杂的层级关系

作为数据传输的格式

XML 作为数据格式，可以被任意程序来进行解析-----已经被 json 代替了

3、XML 的基本语法

一个 XML，包含以下内容：

XML 的文档声明

XML 的注释

XML 的元素

XML 的属性

XML 的特殊字符和 CDATA 区

13-1-2_XML 的基本语法-怎么写一个 XML(重点)

```
<?xml version="1.0" encoding="UTF-8"?>

<!-- XML 的注释内容 --&gt;

&lt;books&gt;

    &lt;book id="book1" 出版社="清华大学出版社"&gt;

        &lt;name&gt;Java 编程思想&lt;/name&gt;

        &lt;price&gt;109.00&lt;/price&gt;

        &lt;author&gt;James&lt;/author&gt;

    &lt;/book&gt;

    &lt;book id="book2"&gt;

        &lt;name&gt;金瓶梅&lt;/name&gt;

        &lt;price&gt;199.00&lt;/price&gt;

        &lt;author&gt;兰陵笑笑生&lt;/author&gt;

    &lt;/book&gt;

&lt;/books&gt;</pre>
```

1、XML 的文档声明

格式： <?xml 属性名="值"?>

文档声明的属性有：

version： 必须。 XML 的版本号，通常是 1.0

encoding: 非必须。XML 文档的字符集编码，通常是 utf-8
standalone: 非必须。XML 文档是否是独立的文档。如果值为 no，表示 XML 还需要依赖其它文档
注意：XML 的文档声明必须在第一行第一列

2、XML 的注释

格式: <!-- 注释内容 -->

3、XML 的元素

XML 的元素和 HTML 的写法比较类型，但语法要更严格：

区分大小写 root。标签和 Root 标签不同

标签必须正确的嵌套。<a>是错误的

有且必须只有一个根标签

标签必须闭合。如果标签里没有内容，就自闭合

标签的命名规范如图：

XML 命名规则

XML 元素必须遵循以下命名规则

- 名称可以包含字母、数字以及其他字符
- 名称不能以数字或者标点符号开始
- 名称不能以字母 xml（或者 XML、Xml 等等）开始
- 名称不能包含空格

可使用任何名称，没有保留的字词

4、XML 的属性

属性的值必须要使用引号括起来

属性区分大小写

属性的命名规范，和元素的命名规范一样

5、特殊字符和 CDATA 区

特殊字符

如果 XML 里写了一些特殊字符，不符合语法规范的话，就会报错。比如：>, <, &, ', " 等等，这些特殊字符需要使用对应的实体字符来代替。常用的实体字符有：

实体字符	特殊字符	说明
>	>	大于号
<	<	小于号

&	&	and 符
'	'	单引号
"	"	双引号

CDATA 区

如果 XML 里有大量的特殊字符，每个实体字符都使用实体字符代替的话，XML 的可维护性非常差。可以把这些带有大量特殊字符的内容写在 CDATA 区里。

CDATA: Character Data, 字符数据区，写在这种区域里的所有数据，都是普通的字符串，没有斜体特殊含义

格式: <![CDATA[在这里可以写任意内容]]>

13-1-3 XML 解析(重点)

1、什么是 XML 解析

使用一段程序代码，来读取 XML 中的数据

2、XML 的解析方式(面试题)

XML 有两种解析方式：DOM 解析和 SAX 解析。两种解析方式各有优缺点：

DOM 解析：Document Object Model，文档对象模型。把整个 XML 读取到内存中形成一棵 dom 树

优点：可以对 DOM 树上的节点进行增、删、改操作

缺点：不利于读取大型 XML 文档

SAX 解析：Simple API for XML，解析 XML 的简单 API。事件驱动型解析，即：逐行读取 XML 的内容

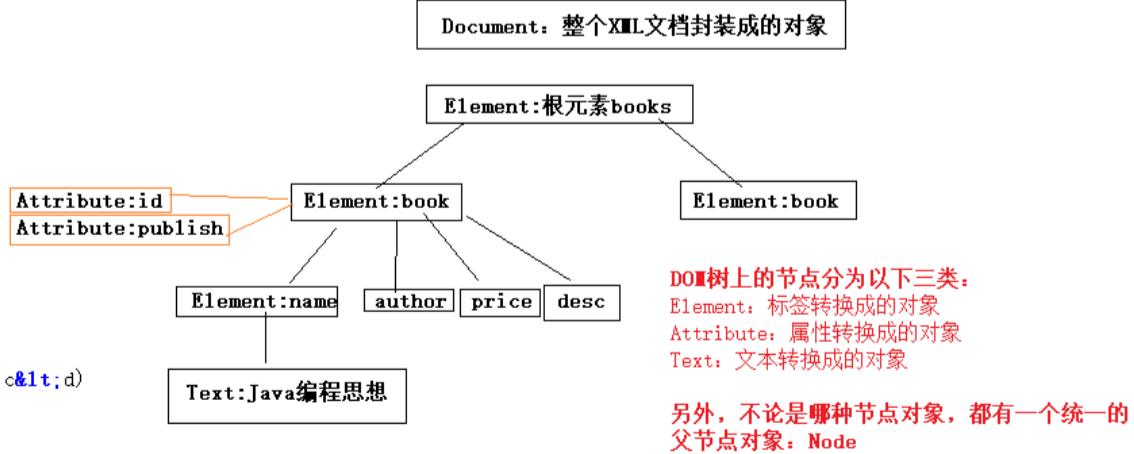
优点：可以很方便的读取大型 XML 文档

缺点：只能往下读，不能往回读。不能进行增、删、改操作

DOM 解析：首先要把 XML 文档加载到内存中

```
<!-- 注释内容 -->
<books>
  <book id="book1" publish="清华大学出版社">
    <name>Java编程思想</name>
    <author>James</author>
    <price>95</price>
    <desc><!CDATA[if (a>b&&c>d)
{System.out.println("aaa");}]></desc>
  </book>
  <book id="book2" publish="清华大学出版社">
    <name>Java核心技术</name>
    <author>jon</author>
    <price>50</price>
    <desc>Java核心技术 if (a>b && c>d)
{ System.out.println("aaa"); }</desc>
  </book>
</books>
```

其次，再把内存里的 XML 转换成一棵 DOM 树



3、常用的解析 XML 的工具包

针对解析 XML 的两种思路，不同组织/企业提供了对应的实现方案。比较常见的有：

JAXP: sun 公司提供的 XML 解析工具

jsoup: 一种处理 HTML 特定的解析开发包

JDOM: JDOM 组织提供的解析工具包

DOM4J: DOM4J 组织提供的解析工具包

4、使用 DOM4J 解析 XML

创建 XML 解析器对象		
DOM4J 解析器类	构造方法	说明
SAXReader	无参构造	使用 SAX 方式读取解析； 逐行读取 XML 文档 ，读完完成也会构造形成一棵 DOM 树
DOMReader	无参构造	使用 DOM 方式读取解析； 把整个 XML 文档加载到内存 ，之后再形成 DOM 树
解析器 org.dom4j.io.SAXReader		
创建解析器： SAXReader saxReader = new SAXReader();		
DOM4J 提供的 SAXReader 解析器对象功能很强，它不仅可以以 SAX 的方式逐行读取 XML 的内容，还会把读取到的所有内容在内存中形成一棵 DOM 树，从而进行增、删、改的操作。		
使用 SAXReader 读取 XML 文档，得到 Document 对象		
API 方法 1: read(InputStream is)		
参数：		
is: XML 文档的输入流对象		
返回值：		
Document 对象， XML 文档对象		
API 方法 2: read(File xmlFile)		
API 方法 3: read(String xmlPath)		
使用 Document 获取 XML 的根元素		
API 方法: getRootElement()		
返回值：		
Element 对象，是 XML 文档的根节点对象		
Element 对象的常用操作		

Element 对象的方法 API:

返回值	方法名称	参数说明	方法描述
List	elements(String name)	name: 子元素名称	获取指定名称的子元素集合
List	elements()		获取所有子元素的集合
Element	element(String name)	name: 子元素名称	获取指定名称的第一个子元素
String	getText()		获取元素里的文本
String	attributeValue(String attrName)	attrName: 属性名称	获取元素上指定属性的值

5、DOM4J 使用 xpath 简化 XML 解析

xpath			
xpath: XML 路径表达式，是一种用来在 XML 中查找信息的语言。			
常见的 xpath 表达式			
参考《XPathTutorial.chm》 实例 1~实例 6			
DOM4J 使用 xpath 的 jar 包			
jaxen-1.1-beta-6.jar			
DOM4J 中 xpath 相关的 API			
Document 对象中 Xpath 相关的方法 API			
返回值	方法名称	参数说明	方法描述
List<Node>	selectNodes(String xpath)	xpath: Xpath 表达式	获取符合 Xpath 的所有节点集合
Node	selectSingleNode(String xpath)	xpath: Xpath 表达式	获取符合 Xpath 的第一个节点对象
应用示例			
//1. 创建解析器 SAXReader saxReader = new SAXReader(); //2. 读取 XML 文档，得到 Document 对象 Document document = saxReader.read("src/books.xml"); //获取符合表达式的所有节点对象 List<Element> nodeList = document.selectNodes("//book"); for (Element element : nodeList) { String name = element.getName();			

```
System.out.println("标签名称: " + name);
}
```

13-1-4 XML 约束(了解)

1、什么是 XML 约束

XML 是可扩展标记语言，可以任意的自定义标签而不报错。但是如果你作为一个框架的开发者，在定义框架时对配置文件一定是有要求，而不能让别人随意乱写的，这个时候就需要对 XML 进行约束，例如：限制 XML 里可以出现什么元素、元素可以出现多少次、元素上可以有什么属性等等

2、XML 的约束方式有哪些(面试)

XML 有两种约束方式：DTD 约束和 Schema 约束。

DTD 语法自成一体；Schema 本身就是一个 XML

Schema 是 XML，可以方便的进行解析

Schema 有比 DTD 更严格的语义和更多的数据类型

Schema 支持名称空间

3、Schema 的名称空间

什么是名称空间

如果一个 XML 中使用多个 Schema 约束，而这些 Schema 中定义了相同的元素名时就会出现名字冲突。就像在一个 Java 类中使用了：import java.sql.* 和 import java.util.*，并且在代码中使用了 Date 类，那么就不能确定到底是 java.util.Date 还是 java.sql.Date 了。

总之，名称空间就是用来处理元素和属性名称冲突问题的，和 Java 中的 package 是同一用途。如果每个元素和属性都有自己的名称空间，就不会有冲突问题了。类似于每个类都有自己的包一样，类名就不会出现冲突了

Schema 的名称空间示例

```
<?xml version="1.0" encoding="UTF-8"?>

<!--

xmlns: xml namespace XML 名称空间.

targetNamespace: 目标名称空间.相当于定义一个包名.

-->

<schema xmlns="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.itheima.com/book"
    elementFormDefault="qualified"

    .....

</schema>
```

4、在 XML 中引入 DTD 约束

语法

在 XML 文档声明之后，写：

<!DOCTYPE 根标签名称 SYSTEM “DTD 文件路径”>：引入本地的 dtd 约束文档

<!DOCTYPE 根标签名称 PUBLIC “DTD 文件路径”>：引入网络的 DTD 约束文档

示例

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!-- 引入 DTD 约束文档 -->
```

```
<!-- DOCTYPE 根标签名称 SYSTEM dtd 文件路径 -->
```

```
<!DOCTYPE books SYSTEM "mybook.dtd">
```

```
<!-- XML 的注释内容 -->
```

```
<books>
```

```
  <book id="book1" 出版社="清华大学出版社">
```

```
    <name>Java 编程思想</name>
```

```
    <author>James</author>
```

```
    <price>109.00</price>
```

```
  </book>
```

```
  <book id="book2">
```

```
    <name>金瓶梅</name>
```

```
    <author>兰陵笑笑生</author>
```

```
    <price>199.00</price>
```

```
  </book>
```

```
</books>
```

5、在 XML 中引入 Schema 约束

语法

在 XML 的根标签上增加属性：

`xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"`

`xmlns="Schema 约束的名称空间"`

`xsi:schemaLoc="Schema 约束的名称空间 Schema 约束文件的路径"`

示例

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!-- XML 的注释内容 -->  
<!--  
在 XML 中引入 Schema 约束的方法：  
在根标签上增加以下内容：  
  
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
xmlns="约束文件的名称空间"  
xsi:schemaLocation="约束文件的名称空间 约束文件的路径"  
-->  
  
<books xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xmlns="http://www.itheima.com/book"  
       xsi:schemaLocation="http://www.itheima.com/book mybook.xsd"  
>  
  
<book>  
    <name>Java 编程思想</name>  
    <price>109.00</price>  
    <author>James</author>  
</book>  
  
<book>  
    <name>金瓶梅</name>  
    <price>199.00</price>  
    <author>兰陵笑笑生</author>  
</book>  
  
<abc></abc>  
  
</books>
```

13-2 动态代理

13-2-1 什么是代理

租房 ---> 房东 租房子

租房 ---> 中介 ---> 房东 租房子

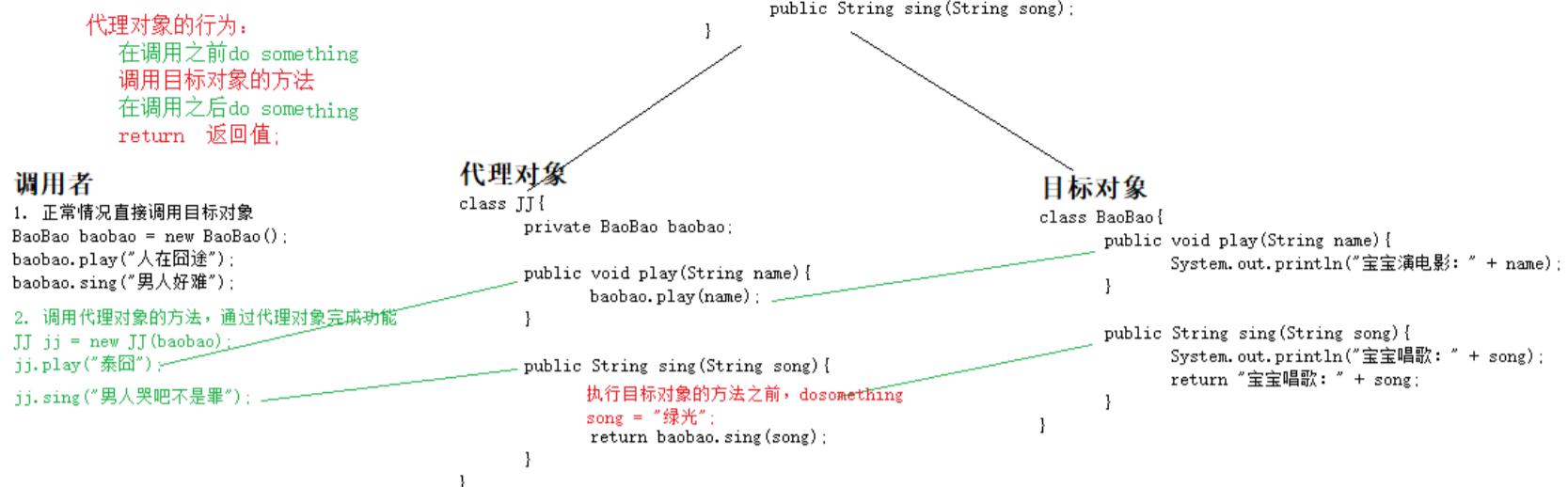
举办活动 ---> 明星 唱歌、跳舞、演电影

举办活动 ---> 经纪人 ---> 明星 唱歌、跳舞、演电影

调用者 ---> 目标对象.doSomething()

调用者 ---> 代理对象.doSomething() ----> 目标对象.doSomething()

静态代理：代理类是提前创建好的，可以直接使用
动态代理：代理类不存在，在代码中动态生成的代理类



13-2-2_代理的分类

静态代理：代理类是提前创建好的，只要 new 一个代理对象，调用代理对象的方法即可

动态代理：代理类不存在，是在代码运行过程中，动态生成的代理类，得到代理对象后，再调用代理对象的方法

13-2-3_动态代理

1、什么是动态代理

通过 JDK 提供的 API，可以根据我们的要求，动态的生成一个代理类的字节码数据（并非真实存在的字节码文件，而是在内存中的字节码数据）。只需要把这个字节码数据加载到 JVM 内存中，就能得到这个代理类的对象，然后调用代理对象的方法

2、相关的 API-java.lang.reflect.Proxy(会写)

类：java.lang.reflect.Proxy

方法：

```
public static Object newProxyInstance(
    ClassLoader loader,
    Class<?>[] interfaces,
    InvocationHandler h)
```

方法的原理：

方法的底层，会根据传入的参数，生成符合我们要求的一个代理类的字节码数据。使用我们提供的类加载器对象，把字节码数据加载到内存中，创建一个代理对象返回

注意：这个方法是基于接口的代理模式

参数：

loader：类加载器。用来加载方法底层生成的代理类字节码数据的

interfaces：代理类要实现的接口。方法的底层会根据这些接口创建对应的方法

h：指定代理对象的行为，即：要代理对象做什么事，一般使用匿名内部类的方式创建行为对象

返回值：

实现了指定接口的代理类对象

3、应用示例

提供接口

```
package com.itheima.proxy;

public interface Star {

    public void sing(String song);

    public String dance(String danceName);

}
```

提供目标对象

```
package com.itheima.proxy;

public class SuperStar implements Star {

    @Override
    public void sing(String song) {
        System.out.println("SuperStar.sing... ");
        System.out.println("唱：" + song);
    }

    @Override
    public String dance(String danceName) {
        System.out.println("SuperStar.dance... ");
        return "跳：" + danceName;
    }
}
```

使用 Proxy 动态创建代理对象

```
//需要增强的目标对象
SuperStar superStar = new SuperStar();

//生成代理对象

Star starProxy = (Star) Proxy.newProxyInstance(superStar.getClass().getClassLoader(), superStar.getClass().getInterfaces(), new InvocationHandler() {

    /**
     * 指定代理对象的行为
     * @param proxy 代理对象
     * @param method 要执行的方法
     * @param args 执行方法需要的参数
    
```

```
* @return  
* @throws Throwable  
*/  
  
@Override  
  
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
  
    //调用目标对象的方法之前, do something  
  
    System.out.println("调用目标对象的方法之前...");  
  
    //调用目标对象的方法  
  
    Object result = method.invoke(superStar, args);  
  
    //调用目标对象方法之后, do something  
  
    System.out.println("调用目标对象的方法之后...");  
  
    return result;  
  
}  
});  
  
//调用代理对象的方法  
  
starProxy.sing("凉凉");  
  
String dance = starProxy.dance("海草舞");
```

14-tomcat& servlet

14-1_HTTP 协议

14-1-1_HTTP 协议概述

1、什么是 HTTP 协议

HTTP: HyperText Transfer Protocol, 超文本传输协议, 是互联网上应用最为广泛的一种网络协议。

HTTPS: secure 安全的 HTTP 协议

HTTP 协议：客户端和服务端之间进行数据交互时，数据的格式规范。

HTTP 协议的默认端口是 80。意思是：

在浏览器输入网址时，如果服务器软件的端口是 80 的话，那么地址栏的 url 就可以不写 80 端口号了。

原本：<http://localhost:80/day14/index.html>

可以写成：<http://localhost/day14/index.html>

2、HTTP 协议的组成

HTTP 协议分为两部分：HTTP 请求和 HTTP 响应。

当在浏览器地址栏输入网址并回车时，浏览器会向服务端发送一个请求，是 HTTP 请求；

服务端接收并处理完成请求之后，服务器软件会向客户端发送一个响应，是 HTTP 响应。

有请求才有响应，没有请求就没有响应

HTTP 请求又分为三部分：请求行、请求头、请求体

HTTP 响应也分为三部分：响应行、响应头、响应体

3、学习 HTTP 的目的

学习 HTTP 协议，是为了了解客户端和服务端之间进行数据交互的时候，传递了哪些数据，这些数据的格式是什么样的。

如果不了解 HTTP 协议，完全不影响 web 应用的开发

但是 HTTP 协议是 web 应用的基石，了解 HTTP 协议，可以把 web 应用学习的更为深入，理解的更为透彻

4、HTTP 数据格式示例

HTTP 请求的数据格式示例



HTTP 响应的数据格式示例

HTTP响应

http响应是服务器软件发送给客户端的数据格式。

在发送之前，服务器软件会创建一个 response 对象，我们可以向 response 设置一些数据。

在发送的时候把 response 对象里的数据转换成 HTTP 响应，发送给客户端。

响应行：服务器软件发给客户端的基本信息
格式：协议版本 响应状态码 响应状态描述

```
HTTP/1.1 200
Accept-Ranges: bytes
ETag: w/"330-1531617272751"
Last-Modified: Sun, 15 Jul 2018 01:14:32 GMT
Content-Type: text/html
Content-Length: 330
Date: Sun, 15 Jul 2018 01:14:48 GMT
```

响应头：服务器发送给客户端的附加信息
格式：一行一个键值对，一个键值对是一个响应头，一次响应可以有多个响应头

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>登录</title>
</head>
<body>
<form action="#" 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>
</body>
</html>
```

响应体：本次响应的正文内容
没有固定格式的，客户端请求什么内容，响应体就是什么内容
比如：
客户端请求一个 html 页面，响应体里就是 HTML 页面
客户端请求一张图片，响应体里就是图片的数据
客户端请求一个 zip，响应体里就是 zip 的数据

14-1-2_HTTP 请求

1、请求行的格式

一次请求的基本信息

格式：请求方式 请求资源 协议版本

例如：POST /web14/login HTTP/1.1

2、请求头的格式

一次请求的附加信息

格式：一行一个键值对，一个键值对是一个请求头，一次请求可以有多个请求头。 key:value

例如： Host:localhost:8080

3、请求体的格式

请求的正文内容，即请求参数

格式：name=value&name=value

例如：username=lisi&password=111

注意：

请求体的内容，其实就是客户端提交的参数。比如提交的表单数据

请求体里不是任何时候都有数据的，只有符合以下条件时，请求体里才会有数据：

必须是 post 方式提交的参数

表单里必须有带 name 属性的表单项

14-1-3 HTTP 响应

1、响应行的格式

一次响应的基本信息

格式：协议版本 响应状态码 响应状态描述

例如：HTTP/1.1 200 OK

了解一些常见的响应状态码：

200：一切正常

302：重定向

304：取本地缓存

404：找不到资源

500：服务器内部错误

2、响应行的格式

一次响应的附加信息

格式：一行一个键值对，一个键值对是一个响应头，一次响应可以有多个响应头。key:value

例如：Content-Type:text/html

3、响应体的格式

一次响应的正文内容

格式：响应体没有固定的格式

响应体是服务端响应给客户端的正文内容，是没有固定格式的。

如果客户端请求了一个页面，那么服务端就会把一个页面放在响应体里发给客户端

如果客户端请求了一张图片，那么服务端就会把一张图片放在响应体里发送给客户端

14-2 web 开发中一些常见的概念

14-2-1 软件架构

C: Client 客户端软件

S: Server 服务器

例如：QQ、微信、大型网游等

优点：

显示效果炫

安全性高

服务器压力小

缺点：

需要安装客户端软件，安装软件时可能还有依赖 .netframework

更新维护不方便

2、B/S 架构

B: Browser 浏览器

S: Server 服务器

例如：京东、淘宝、12306、网银等

优点：

不需要额外安装软件，操作系统自带浏览器

更新维护方便

缺点：

显示效果略差----但是目前已经有了 HTML5 和 CSS3，可以实现非常漂亮的显示效果和特效

安全性不够高----可以使用 HTTPS 协议、U 盾之类的外设，或者其它安全措施

服务器压力大----服务器集群，实现负载均衡

14-2-2_web 资源

通过互联网可以访问的所有资源都是互联网资源。这些资源又分为静态资源和动态资源

1、静态资源

所有人访问时，数据没有任何变化的资源是静态资源。例如：HTML、CSS、js、图片、音频、视频等等

2、动态资源

不同人访问同一个资源，数据可能会有所不同的资源是动态资源。例如：JSP、Servlet、PHP、asp 等等

14-2-3_web 应用服务器

我们开发出来的 web 应用，要想让其它人通过网络能够访问，就需要把 web 应用部署在 web 应用服务器上。常见的 web 应用服务器软件有：

Tomcat: apache 提供的免费开源的小型服务器软件，小巧灵活，应用广泛，支持 JSP 规范和 Servlet 规范

WebLogic: Oracle 提供的收费的大型服务器软件，支持 JavaEE 所有规范。bea

WebSphere: IBM 提供的收费的大型服务器软件，支持 JavaEE 所有规范

14-3 Tomcat

14-3-1 下载与目录结构

bin: 可执行命令在这个文件夹里。例如：启动和关闭的命令
conf: 配置文件目录
lib: Tomcat 的核心 jar 包（Tomcat 也是用 Java 写的）
logs: 日志文件所在的文件夹
temp: 临时文件夹
webapps: web 应用的部署目录。我们写好的 web 应用需要放在这个文件夹里，Tomcat 启动时会自动部署
work: 工作文件夹，主要是给 JSP 使用的

14-3-2 启动与关闭 Tomcat

1、启动 Tomcat

Windows 操作系统里启动：tomcat\bin\startup.bat
Linux 操作系统里启动：tomcat\bin\startup.sh

2、关闭 Tomcat

Windows 操作系统里关闭：tomcat\bin\shutdown.bat
Linux 操作系统里关闭：tomcat\bin\shutdown.sh

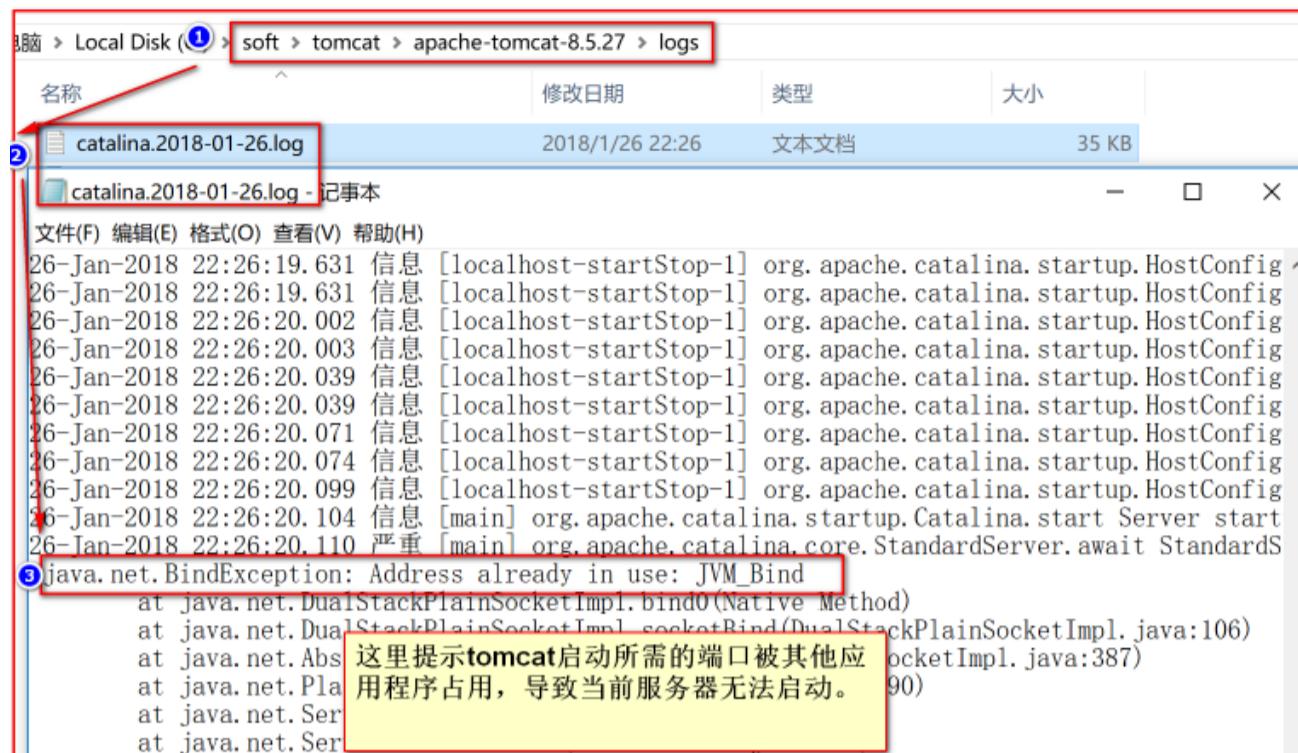
14-3-3 Tomcat 常见问题

1、启动闪退

现象：启动时 Tomcat 的黑窗口一闪而退
原因：可能是 JDK 环境变量配置不正确（Tomcat 本身是 Java 写的，运行 Tomcat 依赖于 JDK 环境）
解决：正确配置 JDK 环境变量

2、端口冲突导致启动失败

现象：启动时 Tomcat 的黑窗口运行几秒后自动退出，查看日志（tomcat\logs\catalina*.log）中有如下内容，如下图所示：



原因：端口冲突导致 Tomcat 无法成功启动。Tomcat 启动时默认要占用 3 个商品：8080,8005,8009 任意一个被占用就会导致无法启动

解决方案一：

杀掉占用端口的程序，然后再启动 Tomcat。具体操作步骤如下：

第一步：打开 cmd，输入命令：netstat -ano|findstr ":8080"

```
C:\Windows\system32\cmd.exe
C:\Users\Thinkpad>netstat -ano|findstr ":8080"
TCP    0.0.0.0:8080          0.0.0.0:0              LISTENING      8724
TCP    [::]:8080            [::]:0                  LISTENING      8724
进程的pid
C:\Users\Thinkpad>
```

第二步：打开任务管理器，找到 pid 对应的进程，结束掉



第三步：启动 Tomcat，如果还有相同的现象，就重复第一步和第二步，分别查找 8005 端口和 8009 商品的进程，如果有就结束掉

第四步：正常启动 Tomcat

解决方案二：

修改 Tomcat 的端口，使用其它端口。修改 Tomcat 端口的步骤如下：

打开 tomcat\conf\server.xml 文件，搜索：port=

找到 8080，修改成其它数值 1025~65535 之间

重启 Tomcat，如果还有相同的现象，就：

找到 8005，修改成其它数值 1025~65535 之间

找到 8009，修改成其它数值 1025~65535 之间

再重启

14-3-4 使用 Tomcat 部署一个 web 应用

1. 在 tomcat\webapps 下创建一个文件夹：web14
2. 在 web14 文件夹里，创建一个 HTML 页面：index.html
3. 使用编辑器打开 index.html，在文件里编写一些内容，例如：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>hello</title>
  </head>
  <body>
    hello, tomcat
  </body>
</html>
```

4. 启动 Tomcat
5. 打开浏览器，输入网址：<http://localhost:8080/web14/index.html>

如果你插上了网线，并且已经关闭了防火墙，可以在你同桌的电脑上访问你的这个页面，只要你同桌在他浏览器上输入网址：<http://你自己的 ip:8080/web14/index.html>

14-3-5 把 Tomcat 集成到 idea 里

14-4 Servlet 入门(重点)

14-4-1 Servlet 概述

1、什么是 Servlet

Servlet: Server Applet，运行在服务端的程序。是 Sun 公司提供的 JavaEE 规范之一。Servlet 可以实现在客户端，以 url 的形式，远程调用服务端的 Java 程序。

狭义的 Servlet，指 `javax.servlet.Servlet` 接口

广义的 Servlet，指所有实现了 `Servlet` 接口的类

Servlet 规范包含三门技术：Servlet 技术、Filter 技术、Listener 技术

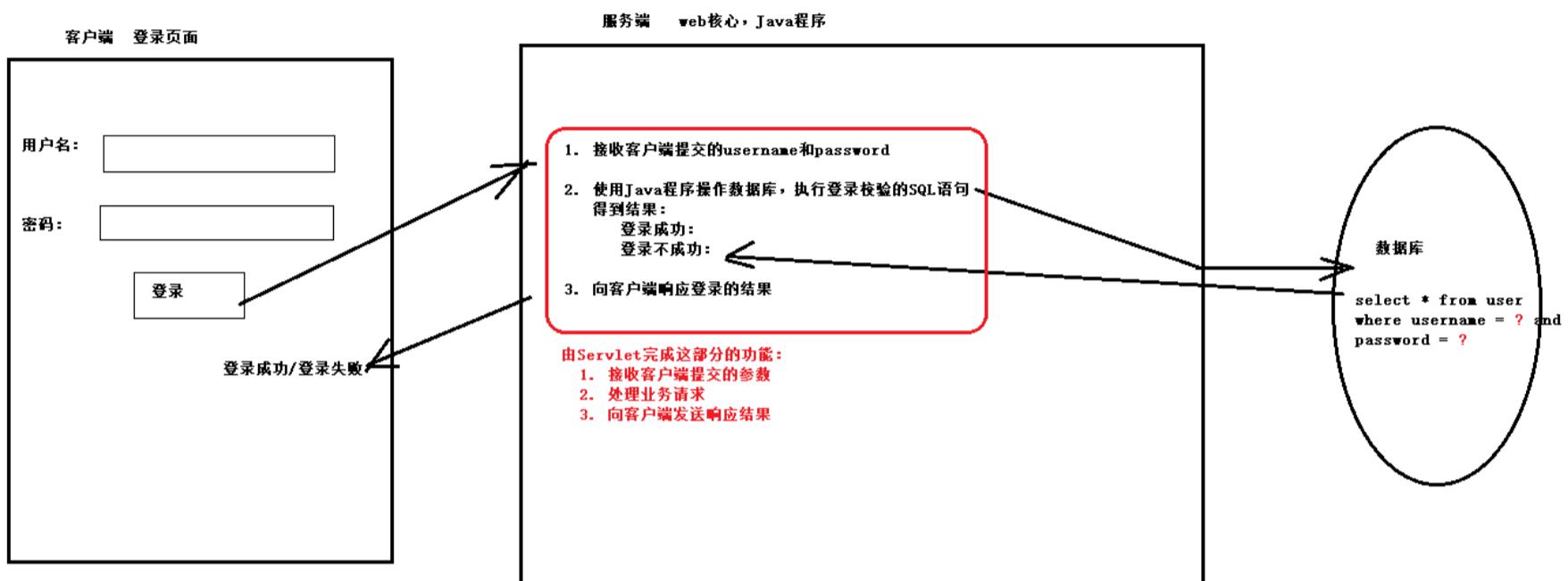
2、Servlet 的作用

Servlet 是服务端里负责和客户端进行交互的，详细的作用包括：

接收客户端提交的参数

处理客户端的请求

向客户端动态响应一些内容



3、Servlet 快速入门开发步骤

创建一个 Java 类，实现 Servlet 接口

实现接口的方法（接口里共 5 个方法，我们需要学习 3 个，重点关注 1 个 service 方法）

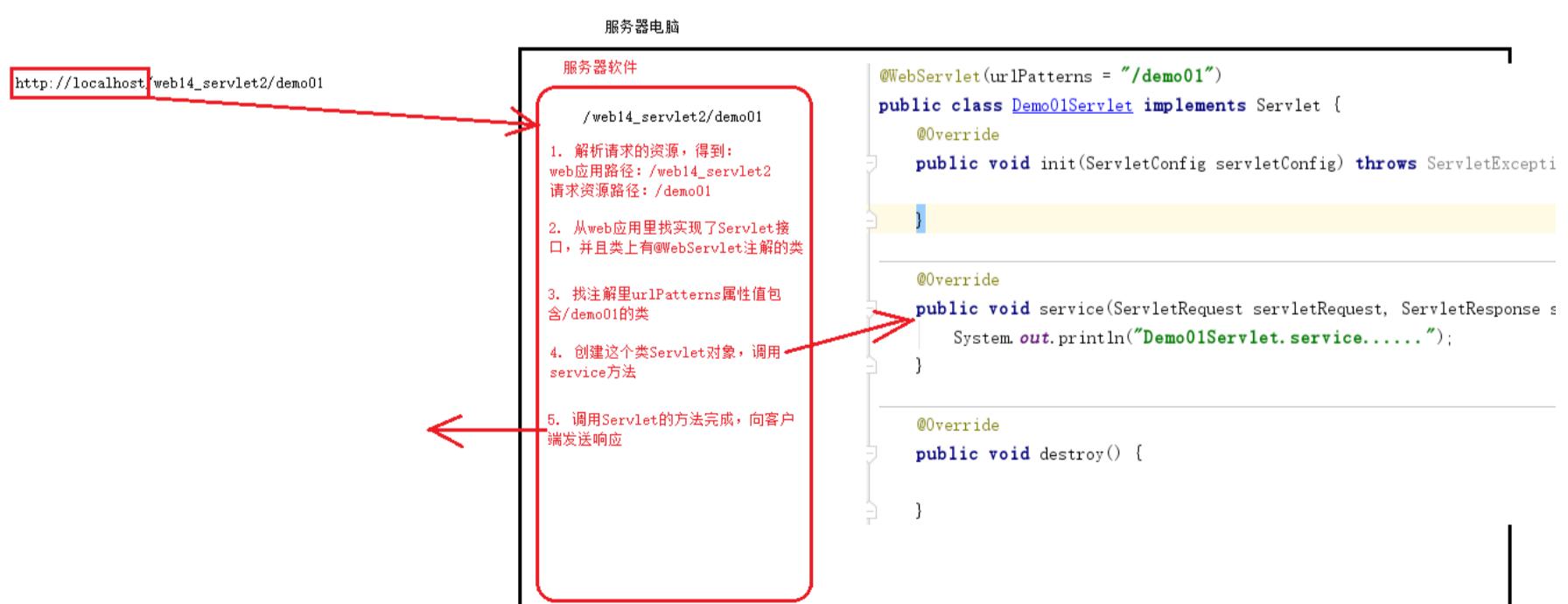
配置 Servlet 的访问路径

2.5 版本的是配置到 web.xml 中

3.0 及以后版本要配置到 Servlet 类的注解@WebServlet 中

14-4-2_Servlet 的 API

servlet 执行原理



1、Servlet 的生命周期(面试题)

Servlet 对象何时创建
默认第一次访问 Servlet 时，服务器软件会创建 Servlet 对象
Servlet 对象被创建时，服务器软件必定会执行其 init 方法
一个 Servlet 类只有一个对象
Servlet 对象何时销毁
服务器软件关闭时 Servlet 会被销毁
Servlet 被销毁时，会执行其 destroy 方法
Servlet 对象每次被访问必须执行的方法是哪个
每次请求到 Servlet，都必须会执行其 service 方法

2、Servlet 的 API

init(ServletConfig servletConfig)
初始化方法，当 Servlet 对象被创建时，服务器软件会调用 init 方法 参数： ServletConfig，当前 Servlet 的配置信息对象，是由服务器软件创建并传递进来的。 有三个作用（了解）： 获取当前 Servlet 的名称----注解中设置的 name，而不是类名 servletConfig.getServletName() 获取当前 Servlet 的初始化参数---注解中设置的 initParams servletConfig.getInitParameter("aa") 获取 ServletContext 对象----关于 ServletContext 对象的内容，下节课详细讲 servletConfig.getServletContext()
service(ServletRequest request, ServletResponse response)
业务处理方法，把业务逻辑代码要写在 service 方法里 参数： ServletRequest：代表 HTTP 请求的 request 对象，用来获取客户端提交的数据 获取客户端提交的参数：String value = request.getParameter(String name) ServletResponse：代表 HTTP 响应的 response 对象，用来向客户端响应数据 向客户端页面上输出内容：response.getWriter().print(String str);
destroy()
销毁方法，当 Servlet 对象被销毁时，服务器软件会执行这个方法

3、创建 **Servlet** 的简化方式

实际上 Sun 公司提供了多种创建 **Servlet** 的方法: 实现 **Servlet** 接口, 继承 **GenericServlet**, 继承 **HttpServlet**。

实现 **Servlet** 接口:

必须要实现接口的所有方法, 哪怕用不到, 导致冗余代码比较多

继承 **GenericServlet** 父类:

Sun 公司为了简化 **Servlet** 的创建而提供的一种方式, 比 **Servlet** 接口要简单一些, 但是它和 HTTP 协议无关

继承 **HttpServlet** 父类:

Sun 公司专门为 HTTP 协议提供的 **Servlet** 创建方式, 简单方便。我们在开发中使用的就是这种方式。

javax.servlet.http.HttpServlet

sun 公司专门为 HTTP 协议提供的 **Servlet** 创建方式: 继承 **HttpServlet**。这个类的 **service** 方法中, 有如下操作逻辑 (可以参考文档结尾附的 **HttpServlet** 部分源码):

```
String method = 请求方式;  
if(method 是 GET){  
    doGet(request, response);  
}else if(method 是 POST){  
    doPost(request, response);  
}else if(method 是 PUT){  
    doPut(request, response);  
}.....
```

我们在实际开发中, 通常只需要重写 **doGet** 和 **doPost** 方法即可

简化创建方式

创建一个 Java 类, 继承 **javax.servlet.http.HttpServlet**

重写 **doGet** 和 **doPost** 方法

增加@**WebServlet** 注解配置 **Servlet**

14-4-3_Servlet 的配置

一个 **Servlet** 创建完成, 如果想要让别人访问得到, 就必须要配置 **Servlet** 的访问路径等信息。

Servlet3.0 及以后的版本, 这些信息是使用注解@**WebServlet** 进行配置的, 常用的配置项有:

name: **Servlet** 的名称

urlPatterns: Servlet 的访问路径

loadOnStartup: 配置 Servlet 的创建时机

initParams: 配置 Servlet 的初始化参数

配置示例：

```
@WebServlet(name="", urlPatterns="", loadOnStartup=1, initParams={})  
public class DemoServlet implements Servlet{  
    .....  
}
```

1、**urlPatterns**-必须配置项

Servlet 的访问路径配置，即：哪些路径可以访问到 Servlet。必须

常用的配置方式有三处：

1. 完全匹配：/demo01 /login
2. 目录匹配：以/开头，以*结尾，例如 /aa/*
3. 扩展名匹配：以*开头，以扩展名结尾，例如 *.action

注意

1. 一次请求，只能到达一个 Servlet 里，即：一次请求只能有一个 Servlet 来处理。如果有多个 urlPattern 都匹配这次请求，那么优先级高的生效。

优先级：完全匹配 > 目录匹配 > 扩展名匹配

2. 目录匹配和扩展名匹配不能混用，即不能写成类似：/aa/*.action

2、**name**-非必须项

Servlet 的名称，非必须

3、**loadOnStartup**-非必须项

Servlet 的创建时机，整数值。非必须

默认情况下，Servlet 是第一次访问时，由服务器软件创建 Servlet 对象。

可以使用 loadOnStartup 属性来更改 Servlet 的创建时机：

如果 loadOnStartup 配置成了正整数，那么服务器一启动，就会创建 Servlet 对象

值越小，优先级越高，创建的越早。但是最好不要把 1 占用

4、**initParams**-非必须

Servlet 的初始化参数。非必须

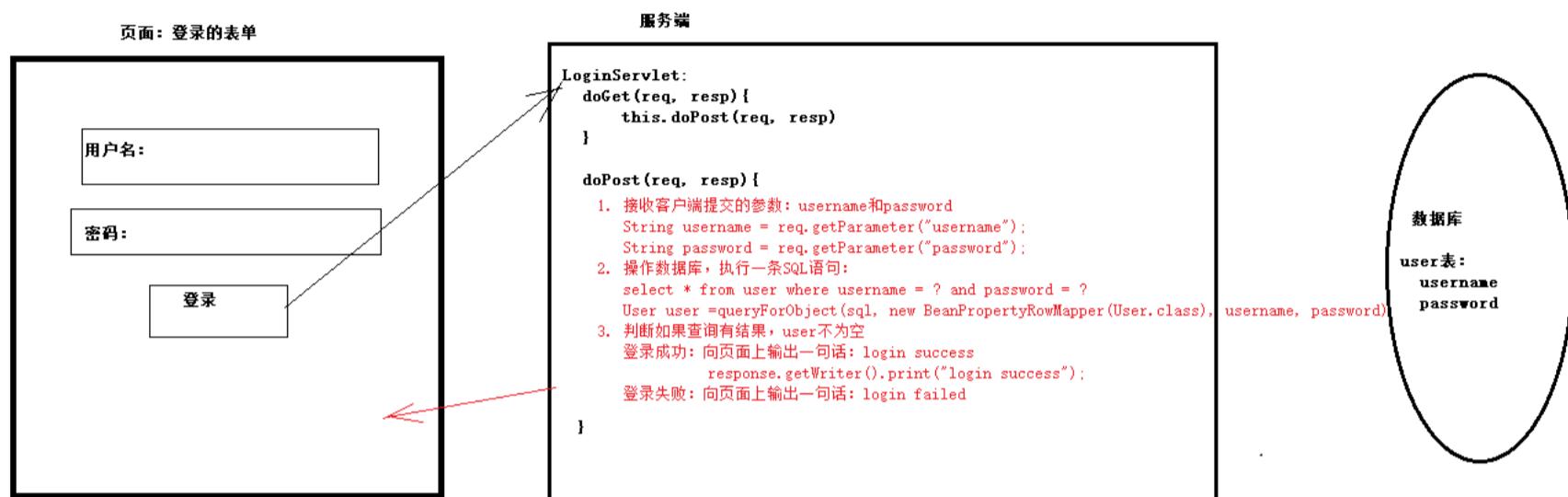
一个 initParams 是注解数组类型的值，即：它的值是一个注解@WebInitParam 的数组

配置方式如下：

```
initParams = {  
    @WebInitParam(name="aa", value="AA"),  
    @WebInitParam(name="bb", value="BB")  
}
```

14-4-4 登录功能案例

登录功能



登录功能实现步骤：

1. 创建一个web应用
2. 在web里创建文件夹：WEB-INF/lib，把数据库操作相关的jar包放在lib文件夹里，然后add as library
数据库驱动包，c3p0连接池的2个包，JDBCTemplate的5个包
3. 把c3p0-config.xml放在src下
4. 把JdbcUtils工具类放在src下 com.itheima.utils包里

代码部分：

1. 创建一个页面login.html，放在web文件夹下
页面里需要提供一个登录表单
2. 在src的com.itheima.servlet包里创建一个Servlet： LoginServlet
在doGet里调用doPost
在doPost里写登录的代码逻辑
 1. 使用request.getParameter()方法获取到页面提交的两个参数：username和password
 2. 操作数据库，执行SQL语句，查询这个username和password对应的数据是否存在
 3. 如果查询到结果了，说明登录成功：

```
response.getWriter().print("login success")
```

如果没有查到结果，或者查询出现异常，说明登录失败：

```
response.getWriter().print("login failed")
```

14-5_附： HttpServlet 中部分源码

注：源码比较复杂，这是删减过的，方便阅读

```
//从 Servlet 接口中实现的 service 方法，每次调用 Servlet，都必定会执行这个方法  
public void service(ServletRequest req, ServletResponse res) {  
    //把参数强转成 HttpServletRequest 和 HttpServletResponse  
    HttpServletRequest request = (HttpServletRequest)req;  
    HttpServletResponse response = (HttpServletResponse)res;  
    //然后调用 HttpServlet 自己定义的 service()方法  
    this.service(request, response);  
}  
  
//HttpServlet 自己定义的 service 方法重载  
protected void service(HttpServletRequest req, HttpServletResponse resp) {
```

```
//获取请求方式
String method = req.getMethod();

//根据请求方式做了分发： Get 方式调 doGet 方法； Post 方式调 doPost 方法； ...
if (method.equals("GET")) {
    .....
    this.doGet(req, resp);
} else if (method.equals("HEAD")) {
    .....
    this.doHead(req, resp);
} else if (method.equals("POST")) {
    this.doPost(req, resp);
} else if (method.equals("PUT")) {
    this.doPut(req, resp);
} else if (method.equals("DELETE")) {
    this.doDelete(req, resp);
} else if (method.equals("OPTIONS")) {
    this.doOptions(req, resp);
} else if (method.equals("TRACE")) {
    this.doTrace(req, resp);
} else {
    .....
    resp.sendError(501, errMsg);
}
}
```

15-ServletContext&Response

15-1_ServletContext(理解)

15-1-1_什么是 ServletContext

ServletContext: 直译：**Servlet上下文对象**。可以理解为是服务器软件把一个 web 应用封装成的一个对象。从这个对象中可以获取 web 应用的一些信息。
ServletContext 对象可以存取数据
一个 web 应用只有一个 ServletContext 对象，在服务器启动时，被服务器软件创建出来的。

15-1-2_怎么获取 ServletContext

在 Servlet 里任意位置使用如下 API:

```
ServletContext context = this.getServletContext();
```

15-1-3_ServletContext 的作用

1、是一个域对象，可以存取数据(**面试**)

什么是域对象

由 **Servlet** 规范提供的，可以用来存取数据的对象，并且保存在这些对象里的数据，在其作用范围里可以进行共享。**Servlet** 规范总共提供了 4 个域对象，其中 **ServletContext** 就是第一个域对象。

面试的问题：有哪些域对象，每个域对象的作用范围是什么

域对象的作用

只要是域对象，必定是可以存取数据的，必定会有以下三个方法：

setAttribute(String name, Object value);

getAttribute(String name);

removeAttribute(String name);

ServletContext 域对象的生命周期

何时创建：服务器启动时创建

何时销毁：服务器关闭时销毁

作用范围：整个 web 应用里都可以共享

2、可以获取 web 应用任意资源

String getRealPath(String path)

这个方法可以获取 web 应用里所有的资源，包括 **src** 下的，以及 **web** 目录中的所有内容。

参数：

path: 资源在 web 应用里的路径

返回值：

资源的实际路径，包含盘符的实际路径

InputStream getResourceAsStream(String path)

这个方法可以获取 web 应用里所有资源的输入流对象

参数：

path: 资源在 web 应用里的路径

返回值：

资源的 **InputStream** 输入流对象

web 应用里的有效的资源：

只有 src 和 web 文件夹里的资源是有效的，其它目录里的所有文件都无效，是获取不到的。因为 idea 在编译输入的时候，只会处理 **src** 和 **web** 文件夹里的内容，其它文件全部舍弃了。

如果我们需要用到一些资源文件，这些资源文件就必须放在 **src** 或者 **web** 目录里。

src: 通常是 Java 程序和 Java 的配置文件

web: 通常是放 html、js、css、图片、音频、视频、jar 包

15-2_Response(核心)

15-2-1_Response 概述

1、简介

response: 代表 HTTP 响应的 response 对象。由服务器软件创建并传递给 Servlet，我们在 Servlet 里可以向 response 里写一些数据，这些数据最终会被发送到客户端浏览器上。

ServletResponse 和 HttpServletResponse

在创建 Servlet 的时候，无论是实现 Servlet 接口的 service 方法，还是继承 HttpServlet 父类重写 doGet/doPost 方法。这些方法都有两个参数：一个代表 http 请求的 request 对象，一个代表 http 响应的 response 对象。

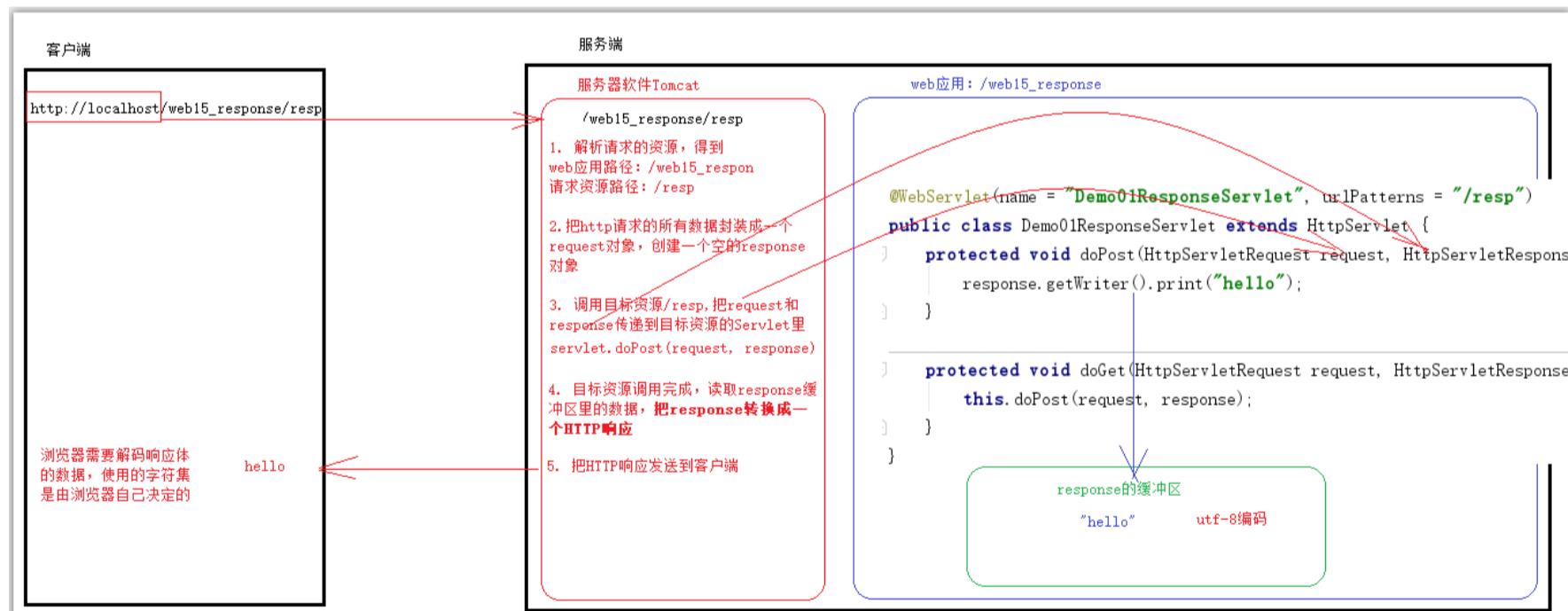
其中 Servlet 接口的 service 方法中，参数的类型是 ServletResponse；而 HttpServlet 的 doGet/doPost 方法中，参数的类型是 HttpServletResponse。

HttpServletResponse 是 ServletResponse 的子接口，功能更强，应用更广泛，所以我们实际使用的、和学习的就是 HttpServletResponse。

2、作用

用来向客户端发送数据的

3、Request 和 Response 的原理



15-2-2_Response API 介绍

The diagram illustrates the components of a response message:

- HTTP/1.1 200**: Status line.
- Accept-Ranges: bytes**, **ETag: W/"330-1531617272751"**, **Last-Modified: sun, 15 Jul 2018 01:14:32 GMT**, **Content-Type: text/html**, **Content-Length: 330**, **Date: Sun, 15 Jul 2018 01:14:48 GMT**: Headers.
- <!DOCTYPE html>**, **<html lang="en">**, **<head>**, **<meta charset="UTF-8">**, **<title>登录</title>**, **</head>**, **<body>**, **<form action="#" method="post">**, **用户名 : <input type="text" name="username">
**, **密码 : <input type="password" name="password">
, **<input type="submit" value="登录">, **</form>**, **</body>**, **</html>**: Body content.
- 设置响应行的数据: 协议版本 响应状态码 响应状态描述**: `response.setStatus(int code)`.
- 设置响应头的数据:** `response.setHeader(String name, String value)`.
- 设置响应体的数据: -----响应体的数据会显示到浏览器页面上**:
 - 设置字符串形式的响应体:** `response.getWriter().print("字符串响应体的内容");`
 - 设置字节码形式的响应体:** `ServletOutputStream os = response.getOutputStream();`
- 使用输入流读取一个文件的数据, 写入到os输出流里**:
这个数据就会被发送到客户端: 把文件传输到客户端了.
- 5个常见的响应状态码:**
 - 200: 一切正常
 - 302: 重定向
 - 304: 取本地缓存
 - 404: 找不到资源
 - 500: 服务器内部错误

15-2-3_Response 的常用功能

1、重定向功能

```
response.sendRedirect("/web 应用的 context 路径/资源路径")
```

2、延迟跳转功能

```
response.setHeader("refresh", "延迟秒数;url=跳转地址");
```

3、向页面输出内容

```
PrintWriter writer = response.getWriter();
writer.print("这些内容会被发送到浏览器页面上, 但是中文会乱码");
```

设置中文响应体乱码的解决方案:

```
//先执行一行代码 注意: 一定要在获取 writer 对象之前先执行, 否则无效
response.setContentType("text/html;charset=utf-8");
//再获取 writer 对象, 输出中文数据就正常了
response.getWriter().print("中文正常显示了");
```

4、文件下载-超链接实现文件下载(了解)

使用超链接方式实现文件下载, 简单方便易用, 但是不能进行权限控制。

实现步骤如下:

```
提供下载资源
```

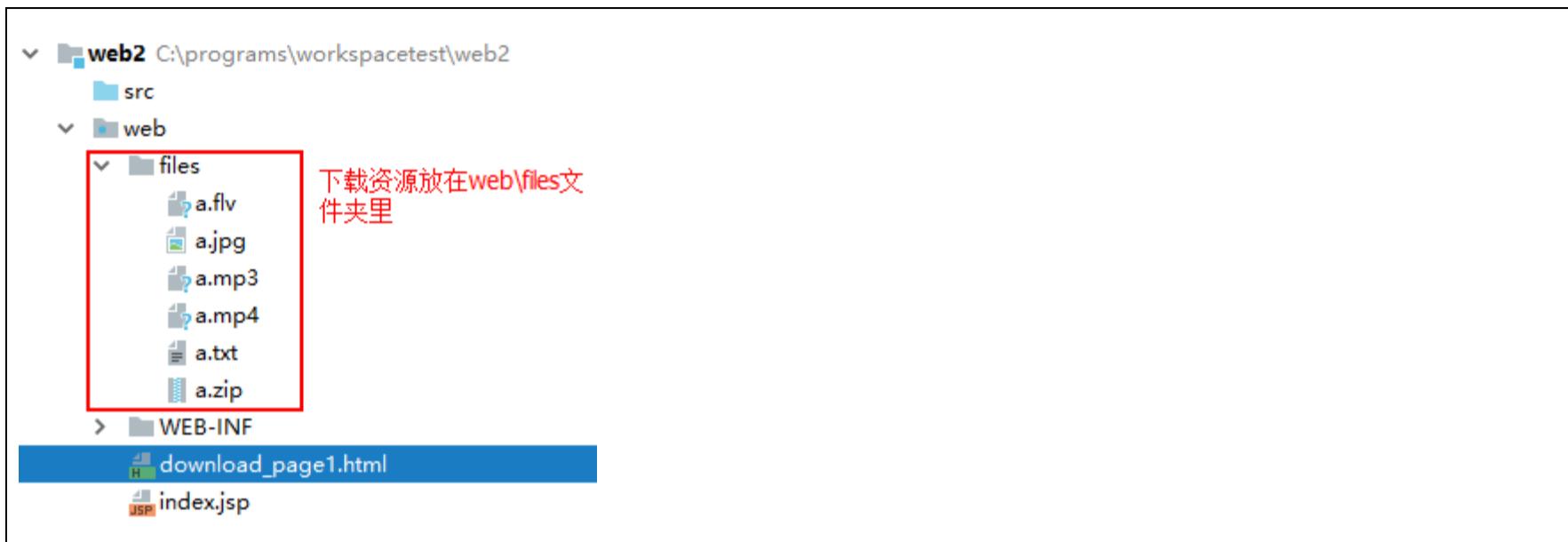
提供下载页面

在 web 下提供一个页面 download_page1.html, 页面内容如下:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>超链接方式实现文件下载</title>
</head>
<body>
<a href="files/a.flv">a.flv</a><br>
<a href="files/a.jpg">a.jpg</a><br>
<a href="files/a.mp3">a.mp3</a><br>
<a href="files/a.mp4">a.mp4</a><br>
<a href="files/a.txt">a.txt</a><br>
<a href="files/a.zip">a.zip</a><br>
</body>
</html>
```

打开页面下载文件

在页面下载链接上 右键-->链接另存为-->在弹出窗口中点击确定



下载资源放在 web\files 文件夹里

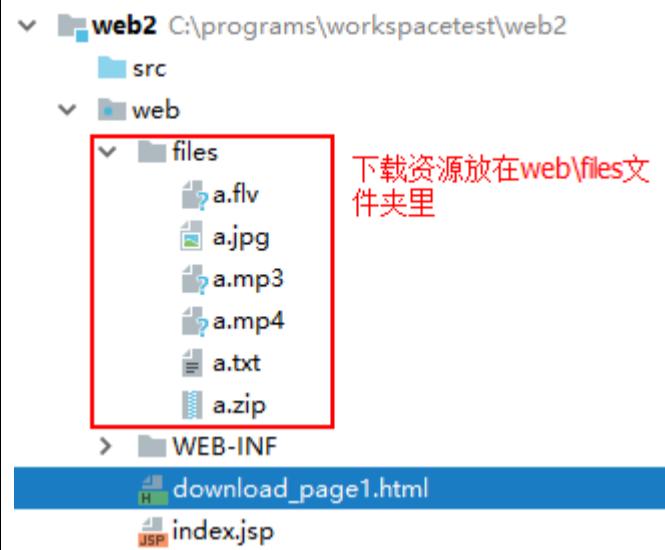
5、文件下载-Servlet 代码实现文件下载(理解)

使用 **Servlet** 提供下载功能，略复杂，但是可以进行权限控制。
实现步骤如下：

提供下载资源

把资源文件放在 web\WEB-INF\files1 文件夹里。

注：WEB-INF 是受保护的文件夹，只能通过服务端程序访问，客户端访问不到。如图：



提供下载页面

创建 DownloadServlet 提供下载功能

```
@WebServlet(name = "DownloadServlet", urlPatterns = "/download")  
public class DownloadServlet extends HttpServlet {  
  
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws  
ServletException, IOException {  
  
    ServletContext servletContext = this.getServletContext();  
  
    //获取客户端传递的参数 filename，即：想要下载的文件名称  
    String filename = request.getParameter("filename");  
  
    //下载文件需要的两步固定代码：1. 设置下载的文件的类型  
    response.setContentType(servletContext.getMimeType(filename));  
  
    //设置弹出下载框中的文件名称，注意：使用 DownloadUtils 工具类处理文件名称后再设置  
    response.setHeader("Content-Disposition", "attachment;filename=" +  
    DownloadUtils.encodeFilename(request, filename));  
  
    InputStream is = servletContext.getResourceAsStream("/WEB-INF/files1/" + filename);  
    ServletOutputStream os = response.getOutputStream();  
    byte[] buffer = new byte[1024];  
    int len = -1;  
    while((len = is.read(buffer))!=-1){  
        os.write(buffer, 0 , len);  
    }  
    is.close();  
    os.close();  
}
```

```
protected void doGet(HttpServletRequest request, HttpServletResponse response) throws  
ServletException, IOException {  
    this.doPost(request, response);  
}  
}
```

DownloadUtils 工具类附件



<---选中这个文件，Ctrl+C，然后到桌面上 Ctrl+V 就能拿到这个文件

提供下载页面

```
<!DOCTYPE html>  
<html lang="en">  
<head>  
    <meta charset="UTF-8">  
    <title>Servlet 实现文件下载</title>  
</head>  
<body>  
    <a href="download?filename=a.flv">a.flv</a><br>  
    <a href="download?filename=a.jpg">a.jpg</a><br>  
    <a href="download?filename=a.mp3">a.mp3</a><br>  
    <a href="download?filename=a.mp4">a.mp4</a><br>  
    <a href="download?filename=a.txt">a.txt</a><br>  
    <a href="download?filename=a.zip">a.zip</a><br>  
</body>  
</html>
```

15-3_JavaEE 里的路径的写法(重点)

15-3-1_相对路径写法

如果是在 HTML 里写的路径，建议使用相对路径

./开头，表示从当前文件夹里查找。（./可以省略）例如：

./files/a.jpg 表示从当前文件夹里找 files 下的 a.jpg

../开头，表示从上级文件夹里查找

../a/b.jpg 表示从上级文件夹里找 a 下的 b.jpg

注意：./可以省略，但是../不可以省略

15-3-2 绝对路径写法

1、HTML 页面里写的路径

要访问 Servlet

http://localhost/web15_response/checkCode

可以把前边的 http://ip:port 省略掉

页面是 http://localhost/web15_response/index.html,

页面里有一个 a 标签需要跳转到 http://localhost/web15_response/checkCode

a 的 href 属性值: /web15_response/checkCode

a 的 href 属性值: ./checkcode

2、服务端 Servlet 的路径的写法

有一个 Servlet 路径是: http://localhost/web15_response/servlet

Servlet 里需要重定向到: http://localhost/web15_response/servlet2

重定向的地址是: /web 应用的 context 路径/资源路径, 即: /web15_response/servlet2

response.sendRedirect("/web15_response/servlet2")

绝对路径: /web 应用的 context 路径/资源路径

16-Request

16-1 Request 概述

16-1-1 Request 简介

1、什么是 Request

request: 代表 HTTP 请求的 request 对象, [javax.servlet.http.HttpServletRequest](#)。

在创建 Servlet 时, 无论是实现 Servlet 接口重写 service 方法, 还是继承 HttpServlet 重写 doGet/doPost 方法, 都有两个参数: 一个代表 http 请求的 request 对象, 一个代表 http 响应的 response 对象。

在 Servlet 接口的 service 方法中, request 的类型是 ServletRequest; 而 HttpServlet 的 doGet/doPost 方法中, request 的类型是 HttpServletRequest。

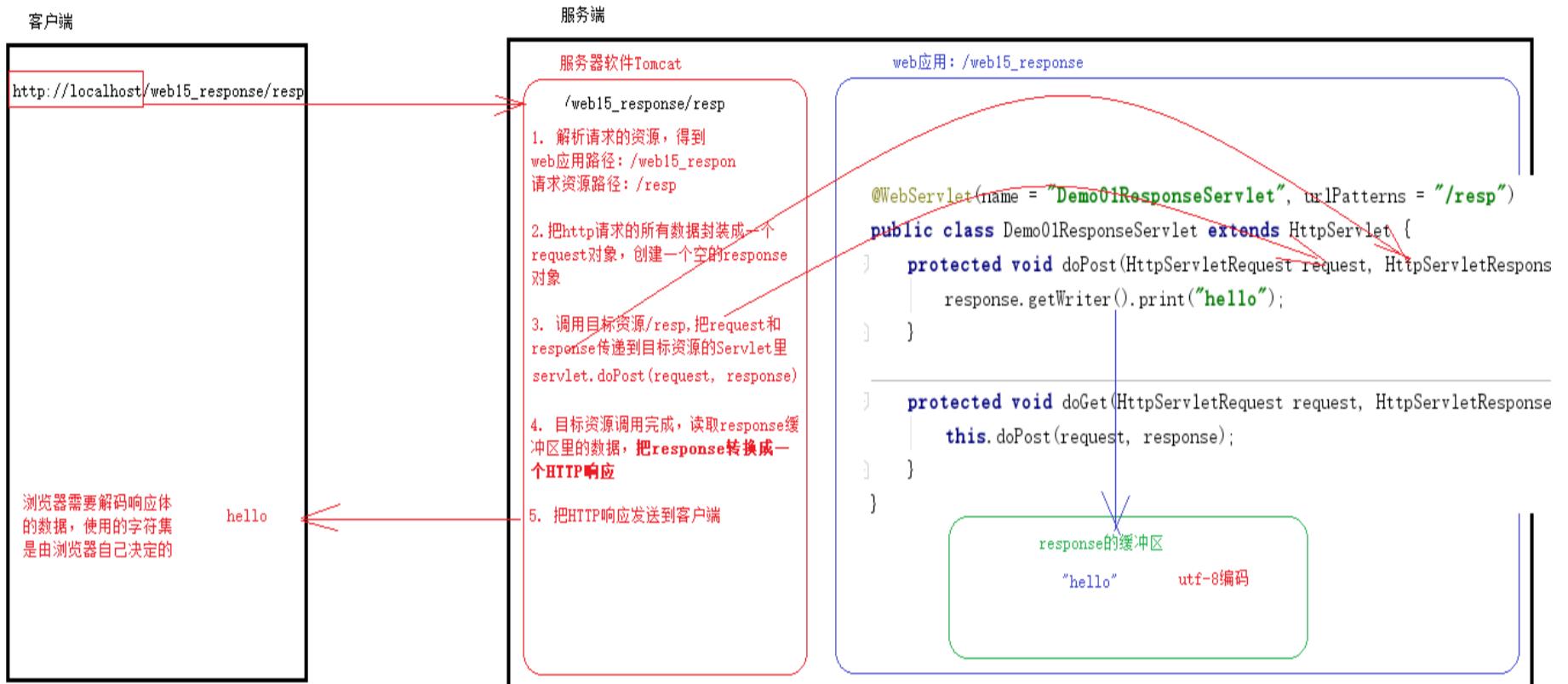
HttpServletRequest 是 ServletRequest 的子接口, 功能更强, 应用更多, 我们学习的就是 HttpServletRequest。

2、Request 的作用

使用 request 对象获取客户端提交的数据

16-1-2 Request 的原理

- 当客户端发起请求时，请求到了服务端软件。
- 服务器软件把 HTTP 请求的所有数据封装成一个 request 对象；然后创建一个空的 response 对象
- 服务器软件调用目标资源，把 request 和 response 传递给目标资源（例如 Servlet）
- 在 Servlet 里：我们可以写代码从 request 对象里获取 http 请求的所有数据。



16-2 Request 的 API



16-2-1 获取请求行的数据

1、获取请求方式

```
String method = request.getMethod();
```

2、获取请求资源

```
String requestURI = request.getRequestURI();
```

```
StringBuffer requestURL = request.getRequestURL();
```

3、获得 ContextPath(重要)

```
request.getContextPath();
```

16-2-2_获取请求头的数据

```
String headerValue = request.getHeader(String headerName);
```

16-2-3_获取请求参数

获取单值参数: **String value = request.getParameter(String name)**

获取多值参数: **String[] values = request.getParameterValues(String name)**

获取所有参数: **Map<String, String[]> map = request.getParameterMap();**

接收中文参数乱码的解决方案:

```
//先执行一行代码 (注意: 这一行代码必须要在所有接收参数之前 先执行, 否则是无效的)
```

```
request.setCharacterEncoding("utf-8");
```

//然后再接收参数, 中文就正常了

```
String username = request.getParameter("username")
```

16-3_Request 的其他功能

16-3-1_Request 是域对象

request 作为域对象, 存取数据的方法:

保存数据: **setAttribute(String name, Object value)**

获取数据: **getAttribute(String name)**

删除数据: **removeAttribute(String name)**

request 域对象的生命周期:

何时创建: 一次请求开始

何时销毁: 一次请求结束

作用范围: 一次请求链中

16-3-2_Request 实现请求转发

```
request.getRequestDispatcher("资源路径").forward(request, response);
```

请求转发跳转和重定向跳转的区别 (如下图):

请求转发只有一次请求; 重定向有 2 次请求

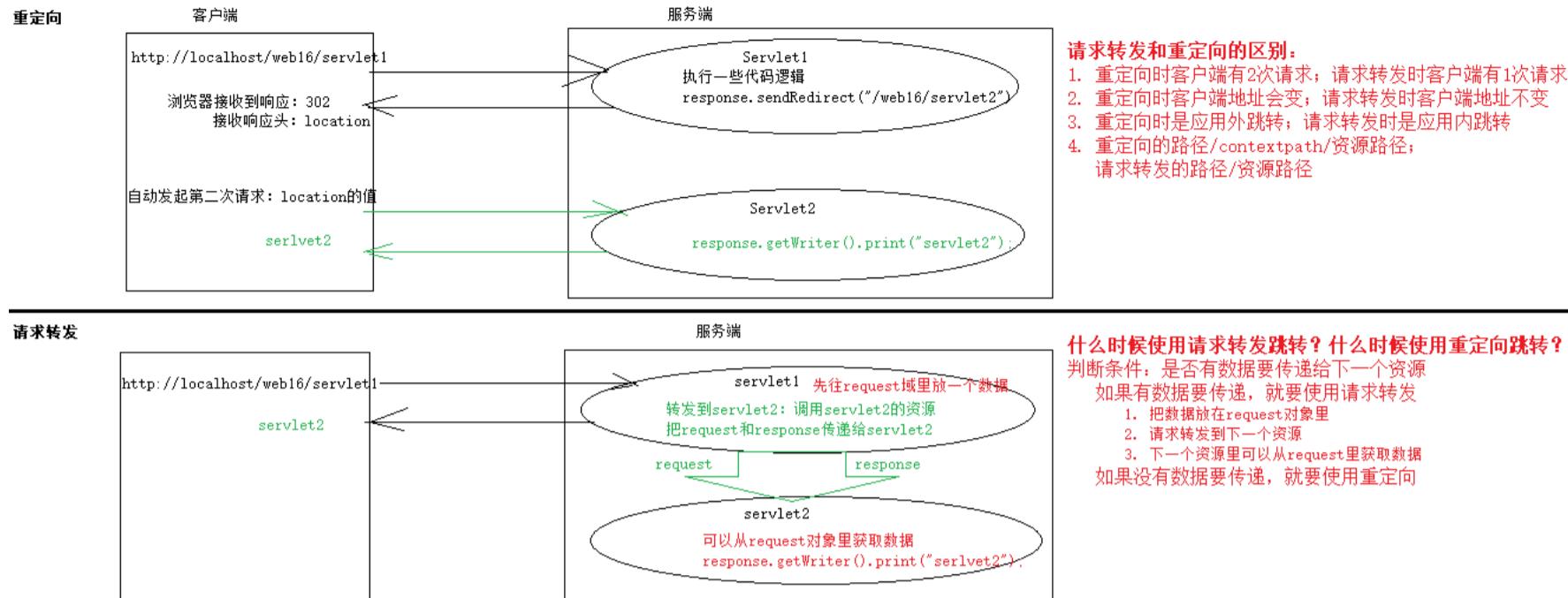
请求转发时地址栏不变; 重定向时地址栏会变

请求转发是应用内跳转; 重定向是应用外跳转

什么时候使用重定向跳转? 什么时候使用请求转发跳转?

如果在跳转时，有数据要传递给下一个资源，就使用请求转发；否则使用重定向

注意：请求转发和重定向不要混用



16-4_域对象

16-4-1_4个域对象必定有的方法

```
setAttribute(String name, Object value);  
  
getAttribute(String name);  
  
removeAttribute(String name);
```

16-4-2_ServletConext 域对象

何时创建：服务器一启动创建，服务器软件创建的

何时销毁：服务器关闭时销毁

作用范围：整个 web 应用

16-4-3_Request 域对象

何时创建：一次请求开始

何时销毁：一次请求结束

作用范围：一次请求转发链中

16-4-4_Session 域对象

16-4-5_pageContext 域对象

17-会话技术 cookie&session

17-1_概述

会话:

一次会话开始: 在浏览器里输入地址回车, 客户端和服务端之间握手建立连接, 会话开启

一次会话中, 可以有 n 次请求和响应

一次会话结束: 关闭浏览器时, 会话结束

会话技术:

存储会话数据的技术。在一次会话的多次请求中, 共享数据技术。

有两种:

Cookie: 把会话数据保存在客户端的技术。

服务器压力小

不够安全, 只能保存字符串

session: 把会话数据保存在服务端的技术, session 技术依赖于 Cookie 技术。

安全, 可以保存复杂数据

服务器压力大

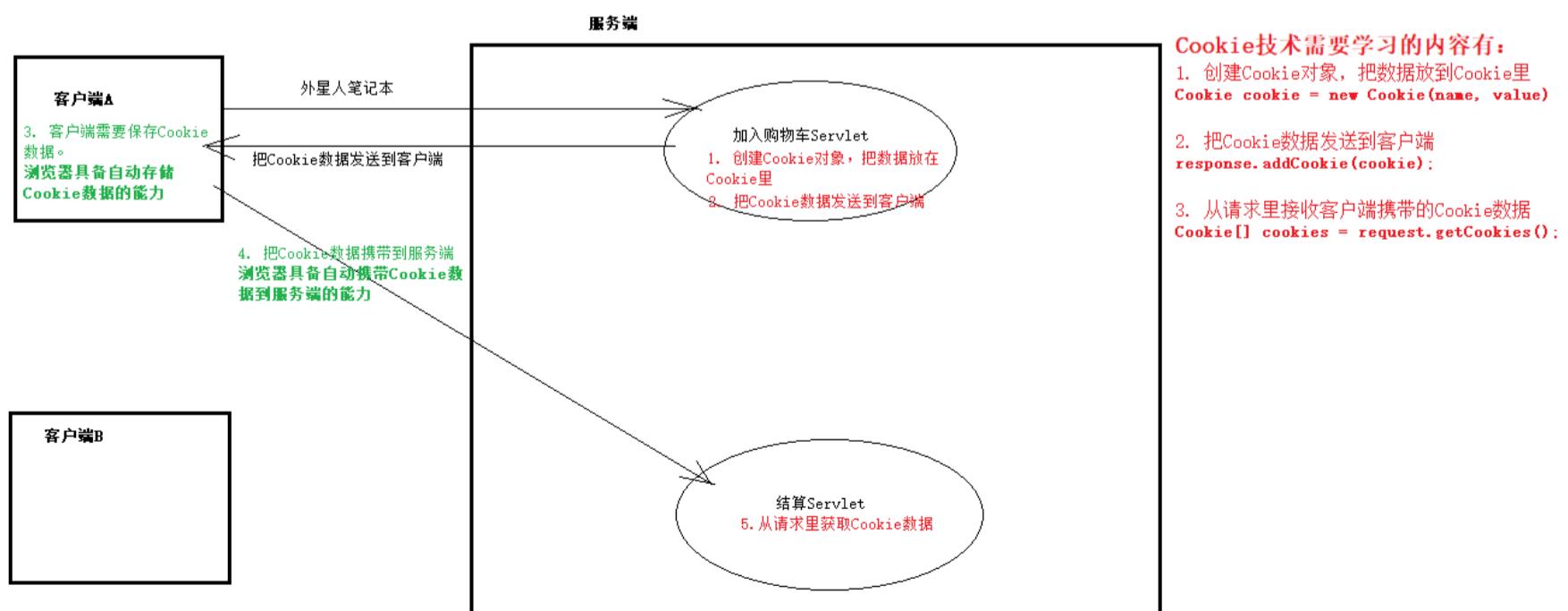
保存在 Cookie 和 session 中的数据, 最大的特点是:

多个会话之间互不干扰,

一个会话的多次请求中可以进行数据共享

17-2_cookie 技术

把会话数据保存在客户端的技术。



17-2-1_怎么创建 Cookie

```
Cookie cookie = new Cookie(String name, String value);
```

Cookie 的有效期:

默认有效期是: 一次会话

因为 Cookie 默认是在浏览器内存中保存的, 浏览器一关闭, 会话结束, Cookie 被清除

设置 Cookie 的有效期: `cookie.setMaxAge(int seconds);`

在 Cookie 有效期内, 无论浏览器是否关闭, Cookie 都是有效的。

直到: Cookie 过期自动失效, 或者手动强制删除浏览器的 Cookie 数据 (`ctrl+shift+delete`)

Cookie 的有效路径:

只有在访问有效路径内的资源时, 客户端才会自动携带 Cookie 到服务端。

生成 Cookie 的资源: /web17_session/aa/create

Cookie 的有效路径: /web17_session/aa

在访问有效路径内的资源时, 会携带 Cookie: /web17_session/aa/bb.jsp

在访问有效路径之外的资源时, 不会携带 Cookie: /web17_session/recieve02

例如:

生成 Cookie 的资源是/web17_session/createSession

有效路径是: /web17_session

生成 Cookie 的资源是/web17_session/aa/bb/createSession

有效路径是: /web17_session/aa/bb

设置 Cookie 的有效路径: `cookie.setPath(String path);`

17-2-2_怎么发送 cookie 到客户端

```
response.addCookie(cookie);
```

17-2-3_怎么接收客户端携带的 cookie

```
Cookie[] cookies = request.getCookies();
```

封装工具类: `String cookieValue = CookieUtils.getCookieValue(cookieName, request);`

17-2-4_怎么删除客户端的 cookie

服务端不能直接删除客户端的 Cookie 数据, 但是可以通过其它方式, 把客户端的 Cookie 覆盖掉。

实现方法:

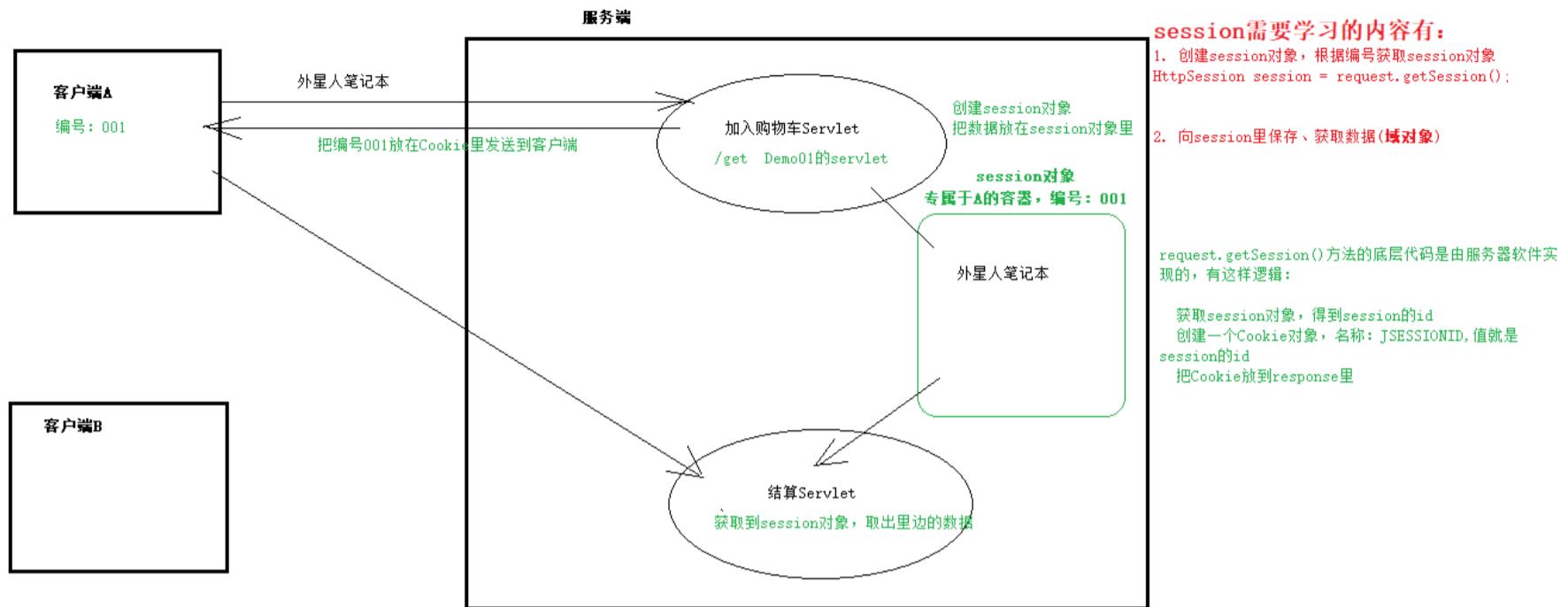
在服务端创建一个和客户端同 name, 同 path 的 Cookie

把这个 Cookie 的有效期设置为 0

把这个 cookie 发送到客户端，它会覆盖掉客户端同 name, 同 path 的 Cookie

17-3_session 技术

把会话数据保存在服务端的技术。依赖于 **Cookie** 技术。



17-3-1_怎么获取一个 session 对象

`HttpSession session = request.getSession();`

什么时候会创建 session 对象

客户端没有 JSESSIONID

客户端有 JSESSIONID, 但是服务端没有对应的 session 对象

什么时候会获取 session 对象:

客户端有 JSESSIONID, 并且服务端有对应的 session 对象

17-3-2_怎么向 session 中存取数据(域对象)

域对象必定有的三个方法:

`setAttribute(String name, Object value);`

`getAttribute(String name);`

`removeAttribute(String name);`

17-3-3_域对象总结

ServletContext:

何时创建: 服务器启动时创建的 (代表整个 web 应用)

何时销毁: 服务器关闭时

作用范围: 整个 web 应用中

request

何时创建: 一次请求开始

何时销毁: 一次请求结束

作用范围：一次请求转发链中

session:

作用范围：一次会话中，一次会话的多次请求之间进行数据共享；不同会话是不会混淆的

何时创建：

执行了 `request.getSession()` 方法，并且：

客户端没有 `JSESSIONID`；或者 客户端有 `JSESSIONID`，但是服务端没有 `session` 对象

何时销毁：

服务器非正常关闭。（如果是正常关闭，服务器会把 `session` 对象保存成文件，启动时读取恢复成 `session`）

会话超时，`session` 自动销毁。默认 30 分钟。

手动销毁。`session.invalidate();`

17-4 应用场景

主要是用来：一次会话的多次请求响应中，有一些临时数据，需要进行存储、共享

会话技术的常见应用场景：

1. 验证码真实值保存。

一次会话里，生成验证码的 `Servlet` 的一次请求有验证码真实值；表单提交时获取到真实值和表单提交的验证码做比较。就一个会话的多次请求中，要共享验证码真实值。可以使用会话技术保存。

验证码真实值保存在 `session` 里

2. 登录状态保存。

一个会话里，一次请求进行登录验证，验证成功，就是已登录状态。在退出或者关闭浏览器之前（即会话结束之前），可以有任意多次请求，应该是已登录状态。把登录状态，使用会话技术保存，通常做法是保存登录的 `User` 对象

登录状态，把登录成功的 `User` 对象保存到 `session` 里。

3. 购物车数据。

不同会话之间的购物车数据，不能混淆，要使用会话技术保存购物车数据。

可以保存在 `Cookie` 里：

缺点：只能保存字符串，不方便操作；不够安全

优点：服务器压力小，可以长期保存数据

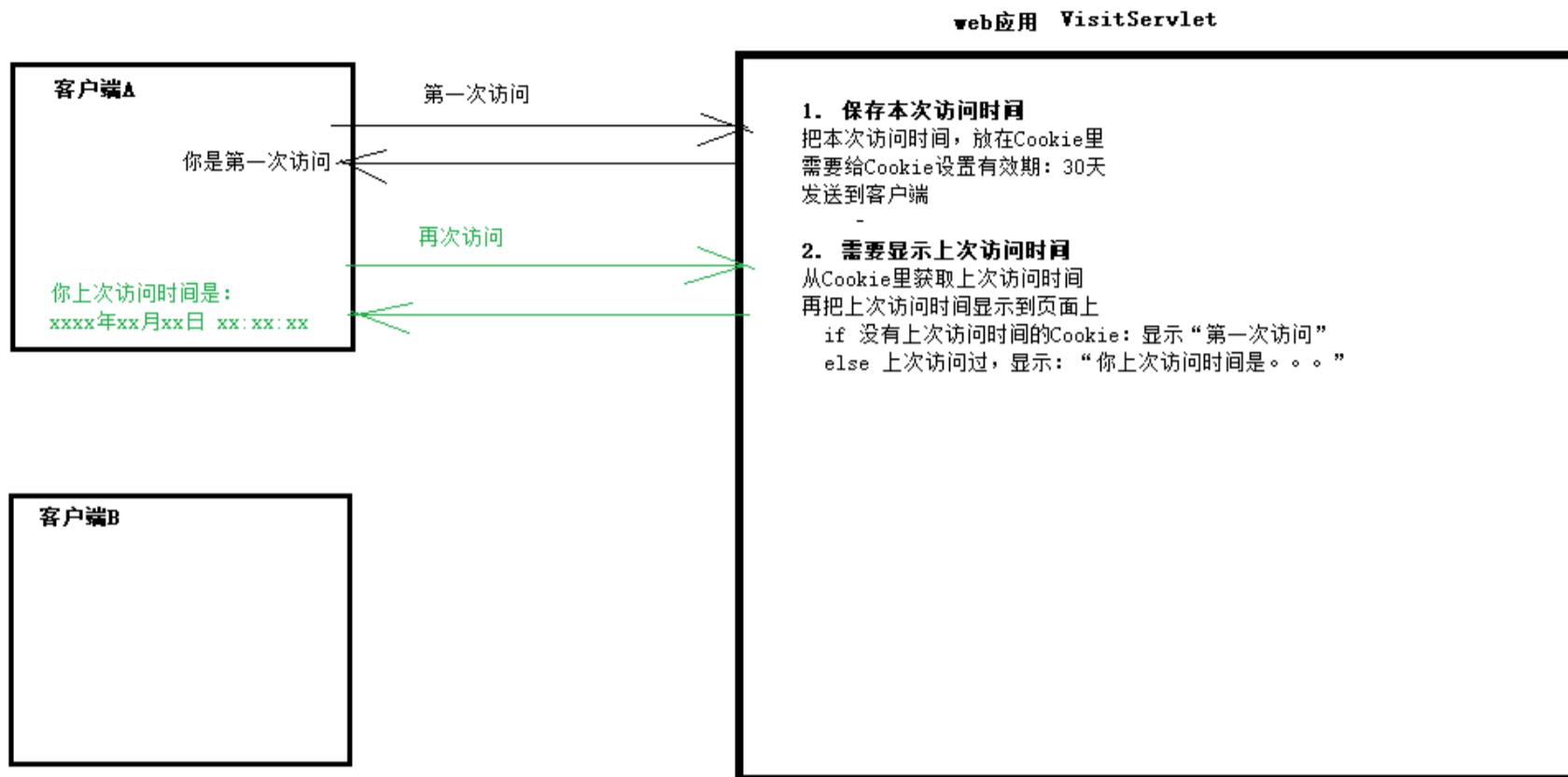
可以保存在 `session` 里：

缺点：服务器压力大，不能长期保存

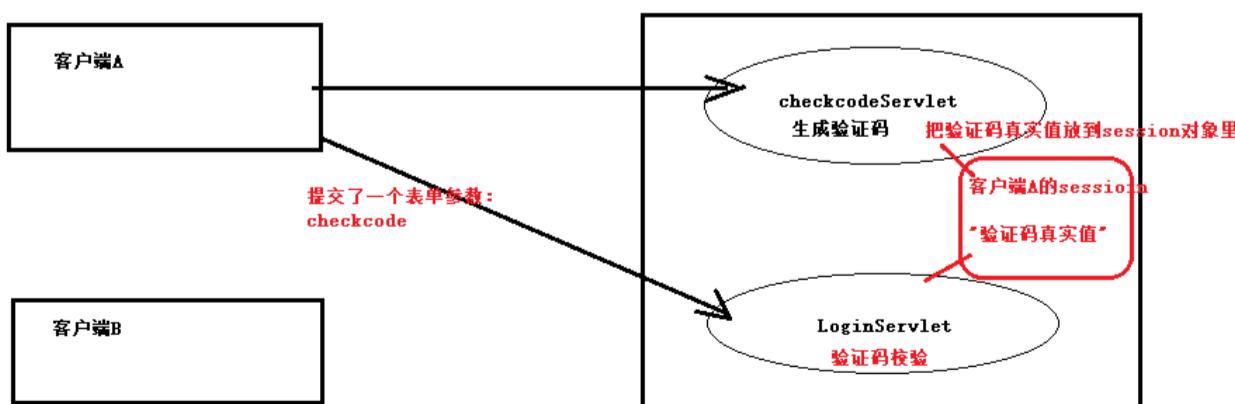
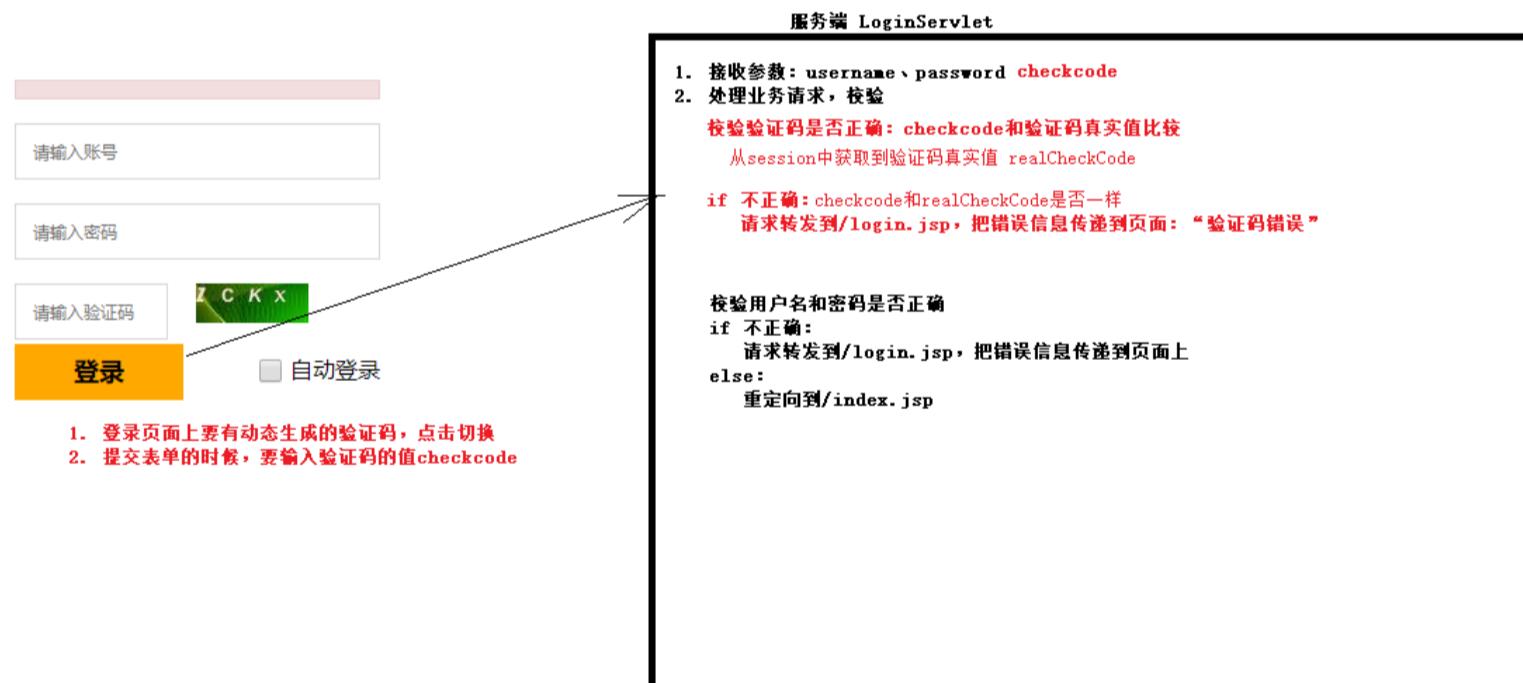
优点：可以保存 `Java` 对象，操作方便；安全

17-5_案例

1、显示上次访问的时间



2、完善登录功能，增加一次性验证码校验功能



18-JSP&EL&JSTL

18-1_JSP

18-1-1_什么是 JSP

JSP: Java Server Page, Java 服务端页面, 是一种动态页面技术。表面上看, JSP 就是 HTML+Java, 但是 JSP 的本质是 Servlet。

Servlet: 擅长处理业务逻辑, 但是不擅长向页面输出内容

JSP: 不擅长处理业务逻辑, 但是擅长向页面输出内容

实际开发过程中, 是 Servlet 和 JSP 配合开发的。

JSP 里页面显示相关的是重点。

18-1-2_JSP 的原理

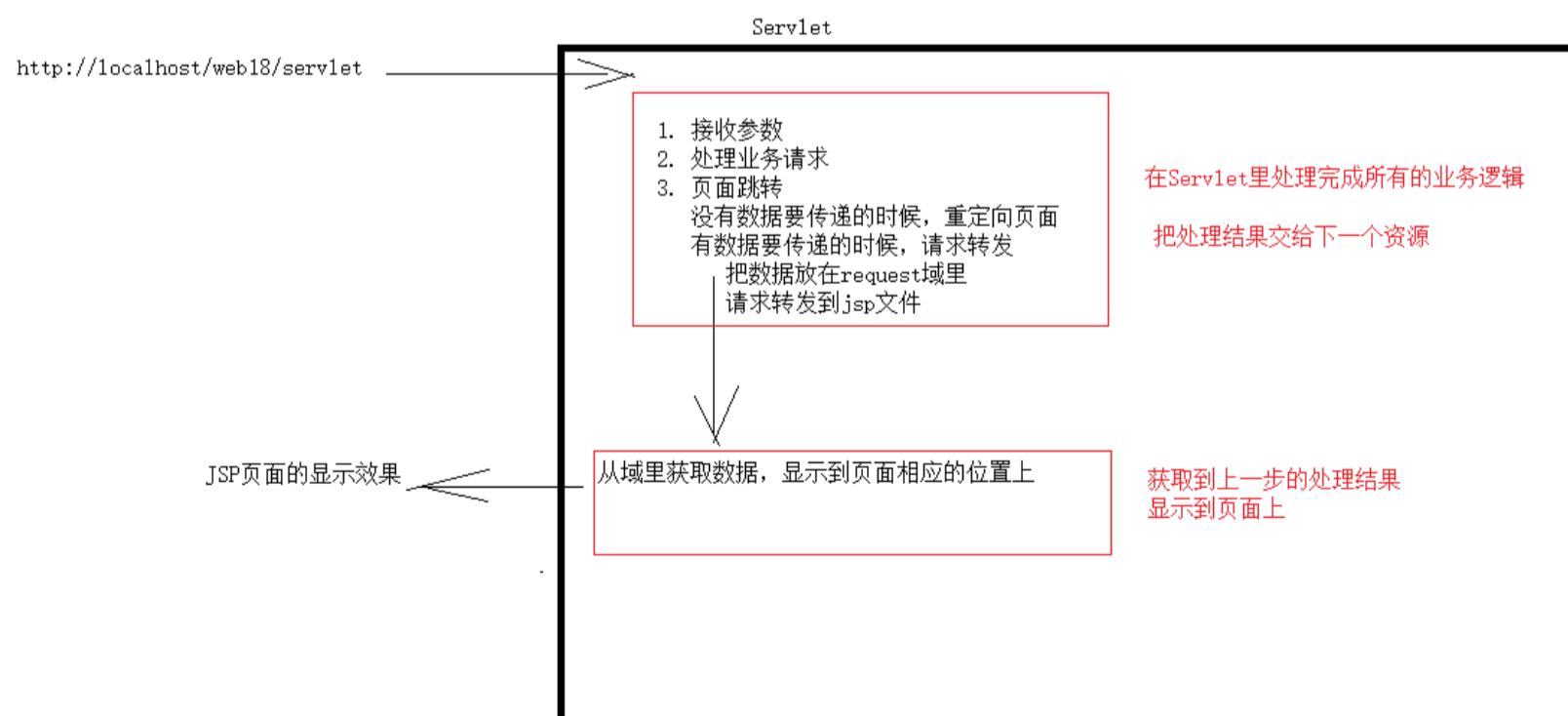
JSP 不能使用浏览器直接打开, 必须要部署到服务器软件里, 由服务器软件把 JSP 文件翻译成 Servlet, 之后实际执行的是 Servlet 的代码。

JSP 文件—翻译-->Java 文件 (Servlet) --编译-->class 文件—运行-->在页面上显示内容

JSP 被翻译之后的文件位置:

C:\Users\用户名\AppData\IntelliJ\Idea2018.1\system\tomcat\Tomcat_8_5_32_workspace41_4\work\Catalina\localhost\model 名称\org\apache\jsp

一个 JSP 文件, 只有在第一次被访问时, 服务器软件才会翻译成 Java 文件, 而不是每次访问都重新翻译。



18-1-3_基本语法

JSP 里可以写: HTML、css、js、以及嵌套 Java 代码。

JSP 里嵌套 Java 代码的语法

<%=Java 表达式 %>

把 Java 表达式的结果输出显示到页面上。会被转换成 service 方法里 out.print(Java 表达式);

<% Java 代码块 %>

把代码块的内容原封不动的拷贝到 service 方法里

<%! Java 声明 %>

Java 声明的代码，会被原封不动的拷贝成为 Servlet 的成员内容。

JSP 里的注释的语法

JSP 的注释：<%-- JSP 的注释 --%> 可以注释掉 JSP 里任意内容

18-2_域对象(面试题)

18-2-1_域对象共同都有的三个方法

setAttribute(String name, Object value)

getAttribute(String name)

removeAttribute(String name)

18-2-2_四个域对象

ServletContext (在 jsp 里叫： application)

何时创建：服务器启动时

何时销毁：服务器关闭时

作用范围：整个 web 应用----一个 web 应用里只有一个 ServletContext 对象

所有会话 所有请求 访问 web 应用的所有资源， 都是同一个 ServletContext 对象

session

何时创建：一次会话开始： request.getSession()， 并且客户端没有 JSESSIONID， 或者服务端没有 session 对象

何时销毁：服务器非正常关闭；会话超时自动销毁；手动销毁 session.invalidate()

作用范围：一次会话中

一个会话中 多次请求 访问 web 应用中的所有资源， 都是同一个 session 对象

request

何时创建：一次请求开始

何时销毁：一次请求结束

作用范围：一次请求转发链中

一个会话中 一次请求 访问 web 应用中的资源， 逻辑上是同一个 request 对象

pageContext

何时创建： jsp 被访问时

何时销毁： jsp 被访问完成

作用范围： 在当前 jsp 页面中

一个会话 一次请求 访问 jsp 文件，才是一个 pageContext 对象

18-3_EL 表达式

18-3-1_简介

概念：

EL: Expression Language, EL 表达式。是 JSP 规范提供的一项技术，用来代替 JSP 中的`<%=Java 表达式%>`。

作用：

用来从域中取出数据，向页面输出显示数据

语法：

`${requestScope.key} 或者 ${requestScope["key"]}`

18-3-2_EL 表达式的作用

从域里取数据，显示到页面上

从 application 域里取数据并显示： `${applicationScope.key}`

从 session 域里取数据并显示： `${sessionScope.key}`

从 request 域里取数据并显示： `${requestScope.key}`

从 pageContext 域里取数据并显示： `${pageScope.key}`

`${key}` 等价于 `<%=pageContext.getAttribute("key")%>`

表示依次从 `pageContext, request, session, ServletContext` 四个域中去查找数据。找到就返回结果

从域里取数据，运算后再显示

基本的数学运算： + - * /

逻辑运算： >, <, >=, <=, ==, !=, !

可以判空:`empty`

如果是 `List, Map` 容器，容器里有内容就不是空的；否则是空的

`""` 是空的

`${empty list ? "empty" : "is not empty"}`

可以进行三元运算： `${username == "jack" ? "ture" : "false"}`

18-4_JSTL 标签库

18-4-1_简介

什么是 JSTL

JSTL: JSP Standard Tag Library, JSP 标准标签库。由 sun 公司提供的一套额外的标签，可以引入到 JSP 里，对 JSP 的功能进行增强。

JSTL 包括 5 个子库，目前还有价值的只有一个 Core 核心库，它提供了 if 判断和 forEach 循环遍历。

I18N: internationalization, I18N

标签库	标签库的 URI	前缀
Core	http://java.sun.com/jsp/jstl/core	c
I18N	http://java.sun.com/jsp/jstl/fmt	fmt
SQL	http://java.sun.com/jsp/jstl/sql	sql
XML	http://java.sun.com/jsp/jstl/xml	x
Functions	http://java.sun.com/jsp/jstl/functions	fn

引入标签库到 jsp 中

1. 引入相关的 jar 包

2. 在 JSP 页面里引入一个标签库，把下面这行代码放在%@page...%>之后

```
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
```

JSTL 的标签本身不具备从域里取数据，并进行逻辑运算的能力。需要和 EL 表达式配合使用。

18-4-2_if 判断

if 标签：进行 if 判断

属性：

test: 判断条件。需要使用 EL 表达式进行判断，把判断结果给 JSTL 的 test 属性值

标签体：

如果判断条件为 true，标签体里的内容会生效

示例代码：

```
<c:if test="${number>3}">  
    <span style="color:red">number>3</span>  
</c:if>
```

18-4-3_forEach 循环遍历

基本 for 循环

forEach 基本 for 循环：

属性：

var: 循环的变量，这个变量的值会被 JSTL 底层代码放到 pageScope 域里

begin: 从几开始循环

end: 循环到几结束
step: 每次循环加几，步长

标签体：

要想取出来循环的变量的值，需要从 **pageScope** 域里取出来

```
<c:forEach var="i" begin="1" end="10" step="1">  
    <span style="color:red;">${i }</span>  
</c:forEach>
```

增强 for 循环

forEach 增强 for 循环：

属性：

items: 被循环遍历的对象。需要使用 EL 表达式获取

var: 循环的变量。变量的值会被保存到 **pageScope** 域里

varStatus: 当前循环的状态。比如：现在是第几次循环 **count**，这次循环的索引是多少 **index**，现在是否是第一次 **first**，是否是最后一次 **last**

标签体：

从 **pageScope** 里取出变量的数据

```
<br/>  
<c:forEach items="${list }" var="pro" varStatus="vs">  
    索引是: ${vs.index }， 第${vs.count }次: <span style="color:red;">${pro }</span>， 是否最后一次:  
    ${vs.last }， 是否第一次: ${vs.first }<br/>  
</c:forEach>
```

18-4-4_choose...when

类似于 Java 的 switch...case...

例如：**request** 域里有一个变量 **day**，值是 1~7 之间的整数，表示星期。页面上把数字变成星期显示出来，如下：

```
<c:choose>  
    <c:when test="${day == 1}">星期日</c:when>  
    <c:when test="${day == 2}">星期一</c:when>  
    <c:when test="${day == 3}">星期二</c:when>  
    <c:when test="${day == 4}">星期三</c:when>  
    <c:when test="${day == 5}">星期四</c:when>  
    <c:when test="${day == 6}">星期五</c:when>  
    <c:when test="${day == 7}">星期六</c:when>
```

```
<c:otherwise>未知</c:otherwise>  
</c:choose>
```

18-5_JSP 开发模式

18-5-1_MVC 模式

JSP 开发架构的发展过程

Model1 阶段：

第一代：纯 JSP 开发

所有代码全部写在 JSP 里，JSP 承担的职责：操作数据库、处理业务逻辑、页面展示

第二代：JSP+JavaBean 架构

JSP：负责页面展示，业务逻辑

JavaBean：负责封装实体，业务逻辑

另一种说法：JSP+Servlet 架构

JSP：页面展示，业务逻辑

Servlet：业务逻辑处理，操作数据库

Model2 阶段： JSP+JavaBean+Servlet 架构

JSP：负责页面展示

JavaBean：封装实体

Servlet：业务逻辑处理

MVC 模式

Model2 阶段，就符合 MVC 的思想。

M: Model 层，模型层，封装实体----JavaBean

V: View 层，视图层，页面展示----JSP

C: Controller 层，控制层，业务逻辑处理----Servlet

18-5-2 JavaEE 的三层架构

JavaEE 的开发分为三层：

web 层：表示层，负责和客户端交互

service 层：业务逻辑层，负责业务逻辑处理

dao 层：数据库访问层，负责操作数据库

在开发过程中，三层主要体现在 package 上

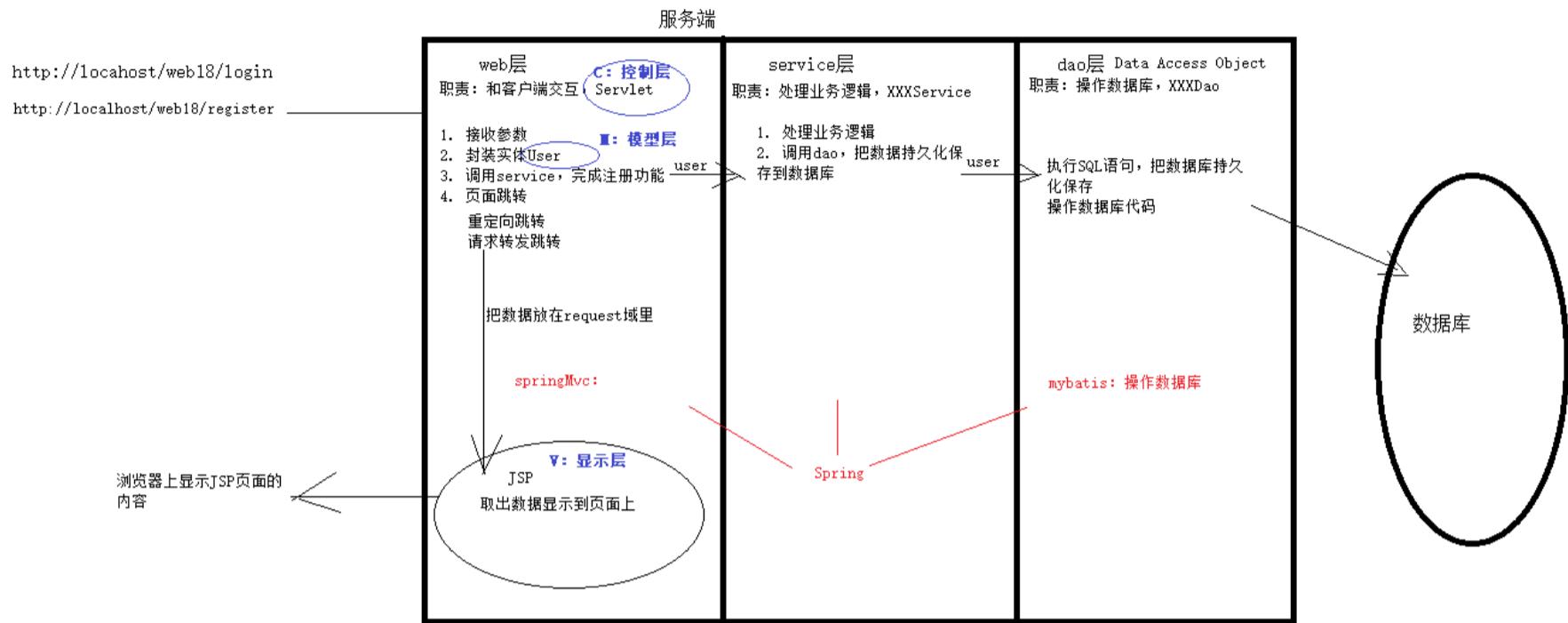
web：放 Servlet

service: 放 Service 类

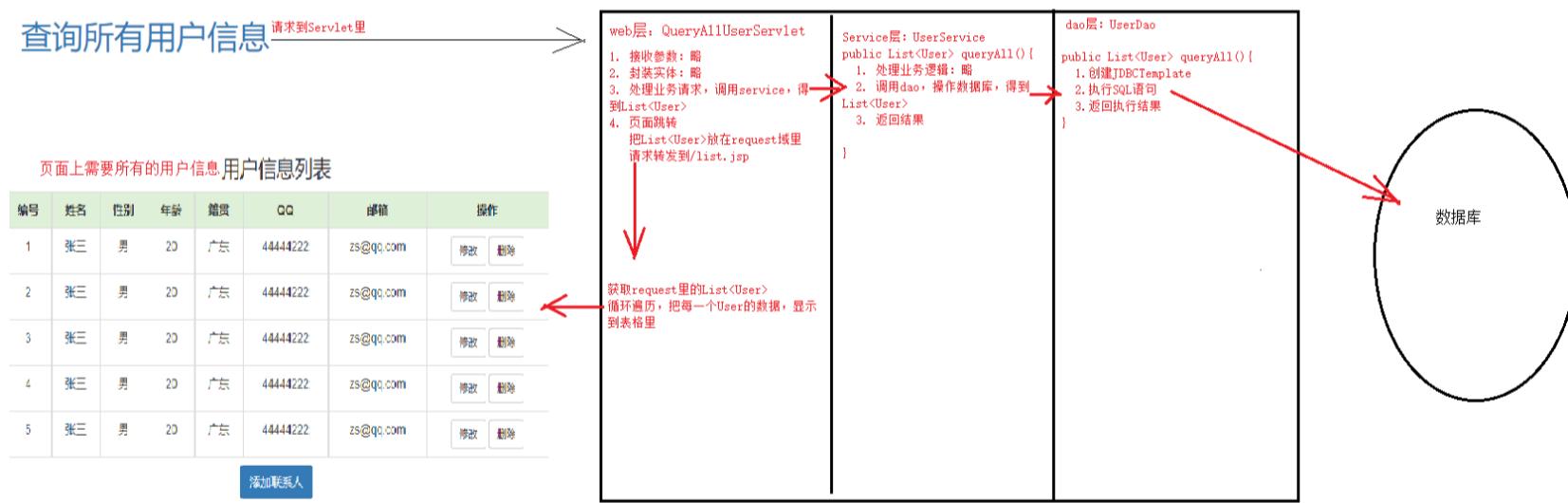
dao: 放 Dao 类

utils: 放工具类

domain: 放 JavaBean



18-6_案例：查询所有用户



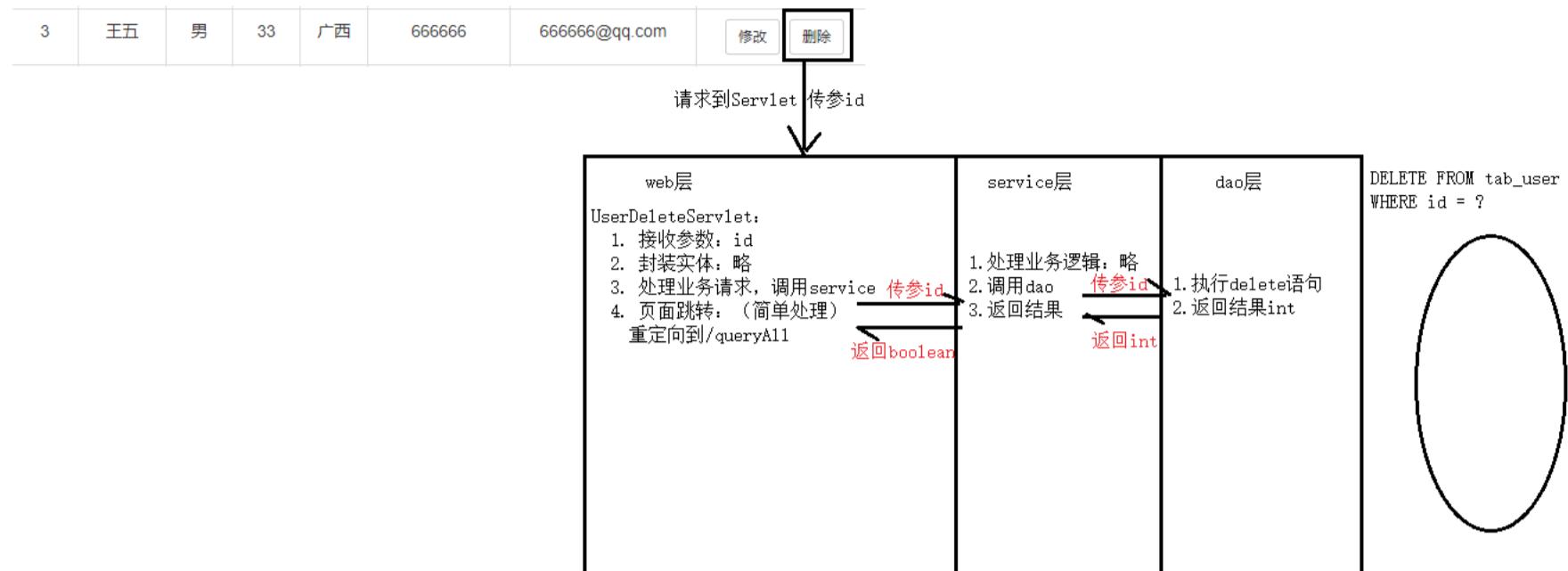
1. 准备开发环境
1. 数据库准备: 执行建表脚本
2. 创建web应用, 把页面放到web文件夹里
3. 创建package: web, service, dao, domain, util
4. 添加jar包、配置文件、工具类
2. 写代码实现功能

19-案例：用户信息的添加、删除、修改和分页

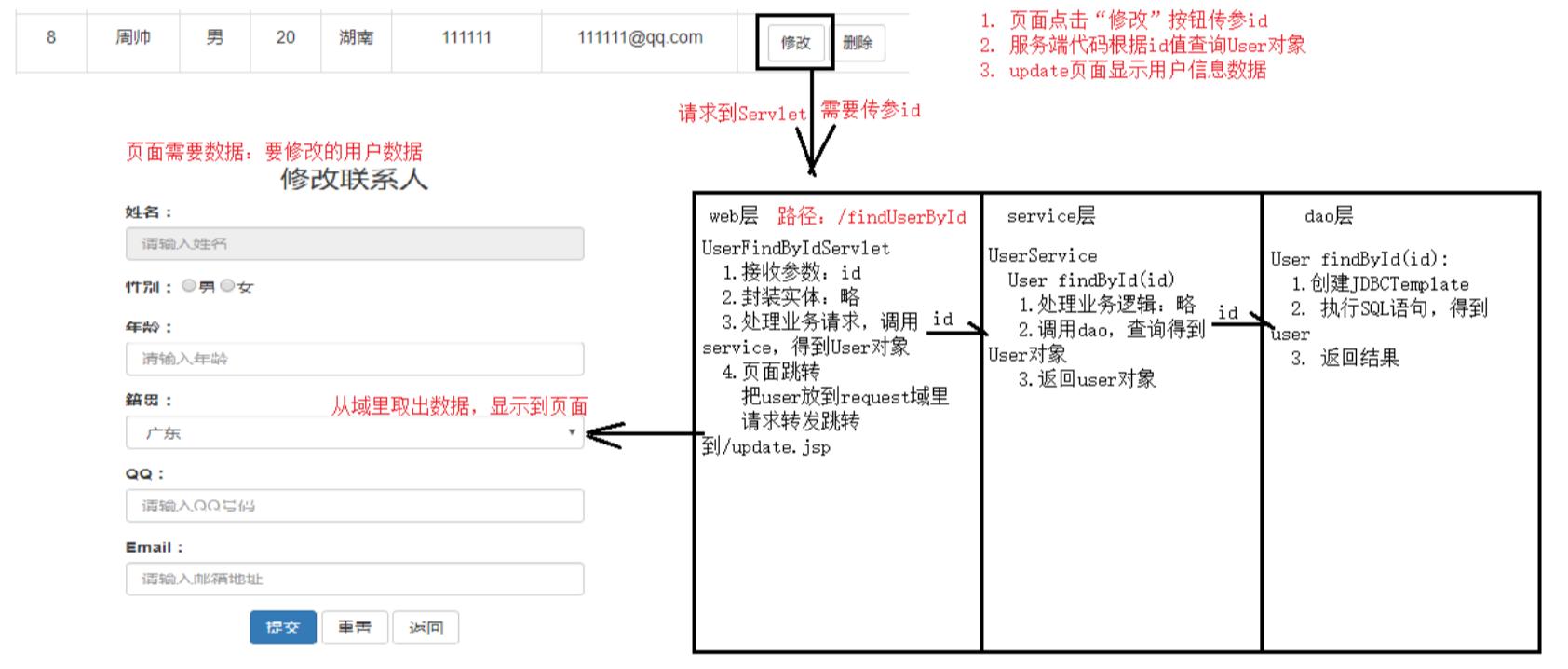
19-1_用户信息的添加



19-2_用户信息的删除



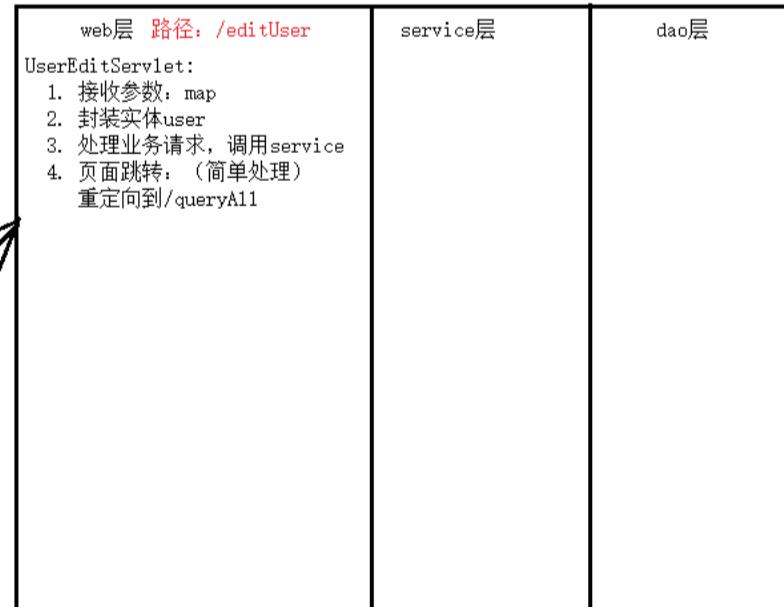
19-3_用户信息的修改



修改联系人

姓名:	王五
性别:	♂男 ♀女
年龄:	23
籍贯:	广东
QQ:	474574574
Email:	ww@qq.com

提交 重置 返回



19-4 分页显示用户信息

显示指定页码的数据: http://localhost/web19_page/queryByPage?pageNumber=页码



20-Filter(重点)和 Listener

20-1 过滤器概述

20-1-1 什么是过滤器

`javax.servlet.Filter`: 是 Servlet 规范的一种 (Servlet、Listener、Filter 技术)。用来在请求到达目标资源之前, 过滤器可以拦截这次请求, 对请求做一些处理之后 (可以对 `request` 对象和 `response` 对象做预处理), 再放行请求到达目标资源。

20-1-2 过滤器的作用

过滤敏感词, ip 过滤, 处理请求乱码, 压缩响应等等

20-1-3 快速入门

创建 Java 类, 实现 Filter 接口

重写接口的方法 (共有 3 个方法, 重点关注 `doFilter` 方法)

给过滤器类增加注解, 配置过滤器的拦截信息

20-2_Filter API

20-2-1_Filter 的生命周期(了解)

何时创建:

服务器启动时, 服务器软件创建过滤器对象。对象被创建时会执行 `init` 方法

过滤器对象被创建了一次

何时销毁:

服务器关闭时会销毁过滤器对象。但是只有正常关闭服务器软件时，才会执行 `destroy` 方法
每次拦截到请求必定会执行的方法是：

`doFilter` 方法

20-2-2_Filter 的 API

1、FilterConfig 对象

是过滤器的配置信息对象，是由服务器软件创建并传递进来的实参。

Method Summary

<code>String</code>	<code>getFilterName()</code> Returns the filter-name of this filter as defined in the deployment descriptor.
<code>String</code>	<code>getInitParameter(String name)</code> Returns a String containing the value of the named initialization parameter, or null if the parameter does not exist.
<code>Enumeration</code>	<code>getInitParameterNames()</code> Returns the names of the filter's initialization parameters as an Enumeration of String objects, or an empty Enumeration if the filter has no initialization parameters.
<code>ServletContext</code>	<code>getServletContext()</code> Returns a reference to the <code>ServletContext</code> in which the caller is executing.

2、FilterChain 过滤链对象(重点)

是过滤器链对象，对象里维护了拦截本次请求的所有过滤器的队列。

只有一个方法：`doFilter(request, response)`

作用：用来放行请求到下一个过滤器。如果下一个过滤器不存在，请求到达目标资源

Method Summary

<code>void</code>	<code>doFilter(ServletRequest request, ServletResponse response)</code> Causes the next filter in the chain to be invoked, or if the calling filter is the last filter in the chain, causes the resource at the end of the chain to be invoked.
-------------------	--

20-3_Filter 的配置

web3 版本开始，过滤器可以使用`@WebFilter`注解进行配置。常用的配置项(`@WebFilter`注解的属性)有：

`urlPatterns`: 必须。过滤器的拦截路径

`filterName`: 非必须。过滤器的名称。如果没有配置就是当前过滤器类的全包名

`dispatcherTypes`: 非必须。过滤器的拦截方式

`initParams`: 非必须。过滤器的初始化参数配置。

20-3-1_urlPattern 配置(拦截路径配置：重点)

配置语法：

`@WebFilter(urlPatterns="拦截路径配置")`

`@WebFilter(urlPatterns={"拦截路径配置 1","拦截路径配置 2",...})`

一个过滤器上可以配置多个 urlPattern 拦截路径

过滤器的拦截路径有三种配置方式：

拦截完全匹配的请求：/target

请求 `http://localhost/web20_filter/target` 时，请求的资源路径是：/target
和过滤器的 urlPattern 完全匹配，这个请求会被拦截

拦截目录匹配的请求：以/开头，以*结尾。比较常用的是/*

请求 `http://localhost/web20_filter/a.jsp` 时，请求的资源路径是：/a.jsp
符合/*，这个请求会被拦截

拦截扩展名匹配的请求：以*开头，以扩展名结尾。比如：*.jsp

请求 `http://localhost/web20_filter/a.jsp` 时，请求的路径是：/a.jsp
符合*.jsp，这个请求会被拦截

注意：

只要过滤器的 urlPattern 匹配得上，就会拦截这个请求

问题：一次请求，可以有几个过滤器拦截？多个
根据 urlPatterns 是否匹配。有几个匹配，就有几个过滤器拦截请求。

问题：一个过滤器，可以拦截几种请求？多种
过滤器的 urlPatterns 里边可以有多个值，每个值都是一种拦截路径
任意一个 urlPattern 匹配的上，就会拦截请求

20-3-2_dispatcherType 配置(拦截方式配置)

配置语法：

```
@WebFilter(urlPatterns="拦截路径", dispatcherTypes=拦截方式)  
@WebFilter(urlPatterns="拦截路径", dispatcherTypes={拦截方式 1,拦截方式 2,...})
```

一个过滤器可以配置多种拦截方式

拦截方式。表示过滤器要拦截哪种方式的请求。常用的有两种拦截方式：

DispatcherType.REQUEST: 默认配置。指客户端发出的请求

DispatcherType.FORWARD: 指请求转发的请求。
比如 `servlet1` 里请求转发到 `Servlet2`。
当客户端访问 `servlet1` 的请求，过滤器不会拦截。因为是客户端的请求
`servlet1` 里请求转发调用 `servlet2` 这次请求，过滤器会拦截

一个过滤器可以配置多种拦截方式。如下：

```
@WebFilter(urlPatterns="/*", dispatcherTypes={DispatcherType.REQUEST, DispatcherType.FORWARD})
```

20-3-3_initParams 配置(初始化参数配置)

配置 Filter 的初始化参数。参数值是另外一个注解 WebInitParam

```
@WebFilter(urlPatterns="", initParms={@WebInitParam(name="",value=""),@WebInitParam(name="",value="")})
```

20-4_过滤器的执行顺序

20-4-1_注解配置的过滤器

如果有多个过滤器都能拦截一个请求，那么这多个过滤器的拦截顺序是：
由过滤器的类名决定的。

20-4-2_web.xml 配置的过滤器

如果有多个过滤器都能拦截一个请求，那么多个过滤器的拦截顺序是：
由 web.xml 中 filter-mapping 出现的顺序决定的。

如果创建的 web 应用里没有 web.xml，可以从其它地方拷贝一个，放在 web/WEB-INF/目录里

```
<!--配置过滤器 Order01Filter-->
<filter>
    <filter-name>order01</filter-name>
    <filter-class>com.itheima.filter.Order01Filter</filter-class>
</filter>
<filter-mapping>
    <filter-name>order01</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
<!--配置过滤器 Order02Filter-->
<filter>
    <filter-name>order02</filter-name>
    <filter-class>com.itheima.filter.Order02Filter</filter-class>
</filter>
<filter-mapping>
    <filter-name>order02</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

20-5_过滤器的应用

提取公共代码到过滤器

比如：敏感词过滤，代码可以放在过滤器里，处理完敏感词之后再放行请求到目标资源。目标资源就不需要再单独处理了

预处理 request 和 response 对象

在过滤器里，先执行以下代码：

```
request.setCharacterEncoding("utf-8");  
response.setContentType("text/html;charset=utf-8");
```

执行完之后，再放行请求到目标资源

可以进行权限控制

比如：web 应用里有一些资源，是必须要管理员才可以访问的。

就可以把这些资源的路径，放到/admin 路径下边

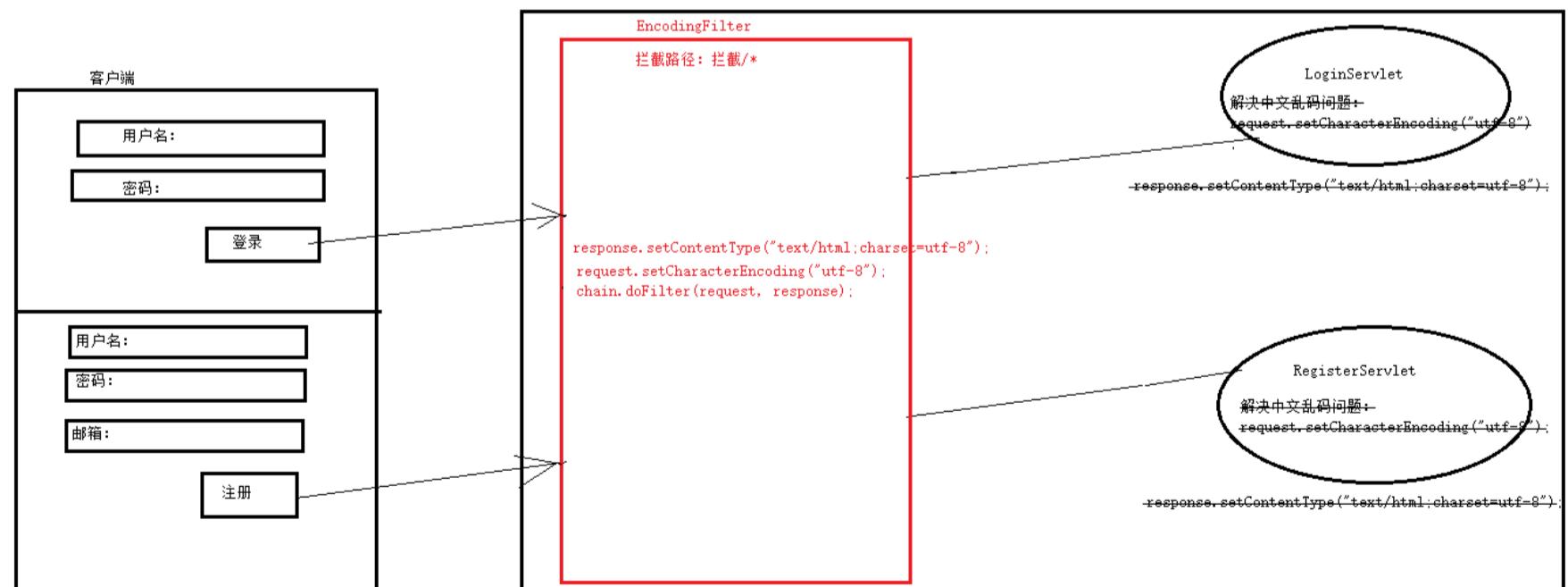
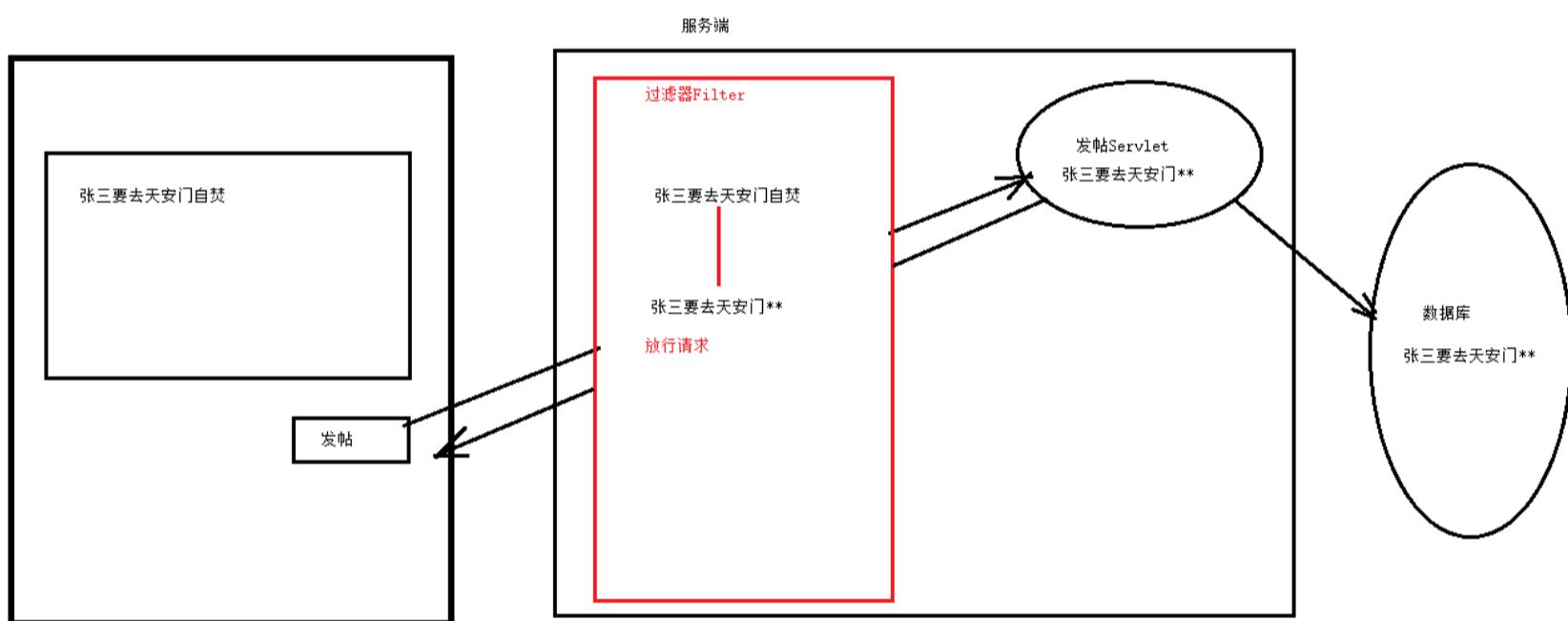
/admin/manager.jsp

/admin/manager

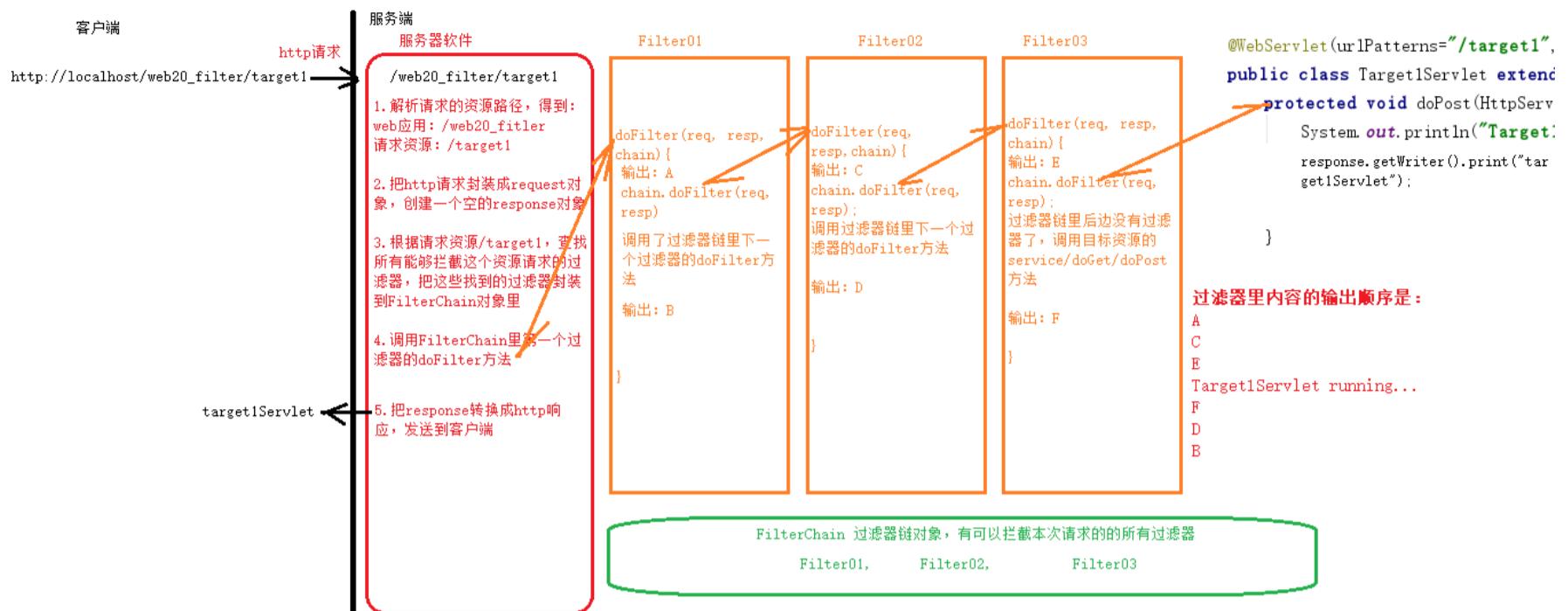
增加过滤器，拦截/admin/*，在过滤器里判断当前用户是否是管理员，

如果是，就放行：执行 chain.doFilter(request, response)

否则不放行



20-6_web 应用的运行流程



20-7_监听器

20-7-1_概述

什么是监听器
监听器 Listener，是 Servlet 规范的一种（ Servlet、Filter、Listener ）。是 JavaEE 提供的，用来监听某个对象状态变化的组件。
监听器和 js 的事件比较类似，也有相关的三个概念：
事件源：被监听的对象，一般是 3 域对象
监听器（事件）：用来监听事件源状态变化的组件，有 6+2 个监听器
响应行为：监听到事件源状态变化之后要执行的代码---我们写代码来完成

有哪些监听器

第一类：6 个监听域对象的监听器

	ServletContext 域	HttpSession 域	ServletRequest 域
监听域对象创建与销毁	ServletContextListener	HttpSessionListener	ServletRequestListener
监听域对象 Attribute 变化	ServletContextAttributeListener	HttpSessionAttributeListener	ServletRequestAttributeListener

第二类：2 个特殊的 JavaBean 的监听器

监听器	描述
HttpSessionBindingListener	监听 JavaBean 对象和 session 的绑定与解绑状态
HttpSessionActivationListener	监听 session 中 JavaBean 对象的钝化(把一个对象保存在文件里)与活化(反过来)状态

20-7-2 域对象的监听器

开发步骤

创建 Java 类，实现监听器接口

重写接口的方法

把监听器配置到 web.xml 中

20-7-3 特殊的 JavaBean 监听器(2 个，对象感知监听器)

特殊的监听器

给 JavaBean 使用的监听器，需要由 JavaBean 实现监听器接口，监听自己的状态变化

不需要配置到 web.xml 中

监听的状态：

HttpSessionBindingListener:

绑定：把 JavaBean 对象放到 session 域中

解绑：把 JavaBean 从 session 中移除

HttpSessionActivationListener:

钝化：session 中的 JavaBean 对象被序列化保存成了文件

活化：把文件里的数据恢复成为内存中的 session 中 JavaBean 对象

HttpSessionBindingListener

开发步骤：

创建一个 JavaBean，实现 HttpSessionBindingListener 接口

重写接口的方法

验证步骤：

创建一个 JavaBean 对象

把 JavaBean 对象放到 session 域中----触发绑定

把 JavaBean 从 session 中移除----触发解绑

HttpSessionActivationListener

开发步骤：

创建一个 JavaBean，实现 HttpSessionActivationListener，Serializable 接口

重写接口的方法

验证步骤：

创建一个 JavaBean 对象，把 JavaBean 对象放到 session 中

服务器正常关闭----触发钝化

服务器重新启动----触发活化

20-8_定时器

定时器相关 API

定时器类: `java.util.Timer`

构造方法: `Timer()` Creates a new timer.

常用的方法:

给定时器指派任务: `schedule(TimerTask task, Date firstTime, long period)`

参数:

`task`: 让定时器执行的任务。类型是 `TimerTask`, 是一个接口, 一般使用匿名内部类

```
new TimerTask(){  
    public void run(){  
        要执行的任务代码  
    }  
}
```

`firstTime`: 第一次执行的时间

`period`: 间隔多长时间执行一次, 单位是毫秒

20-9_邮件服务器

20-9-1_相关概念

邮件客户端

网页客户端: 使用浏览器登录邮件, 收发邮件

软件客户端: 腾讯的免费邮件客户端软件 `foxmail`, Microsoft 收费的邮件客户端软件 `outlook`

邮件服务器

数据库服务器: 提供数据存取服务的软件。 `MySQL`

`web` 应用服务器: 提供 `web` 访问服务的软件。 `Tomcat`

邮件服务器: 提供邮件收发服务的软件。

20-9-2_邮件收发原理

邮件收发协议

发件协议:

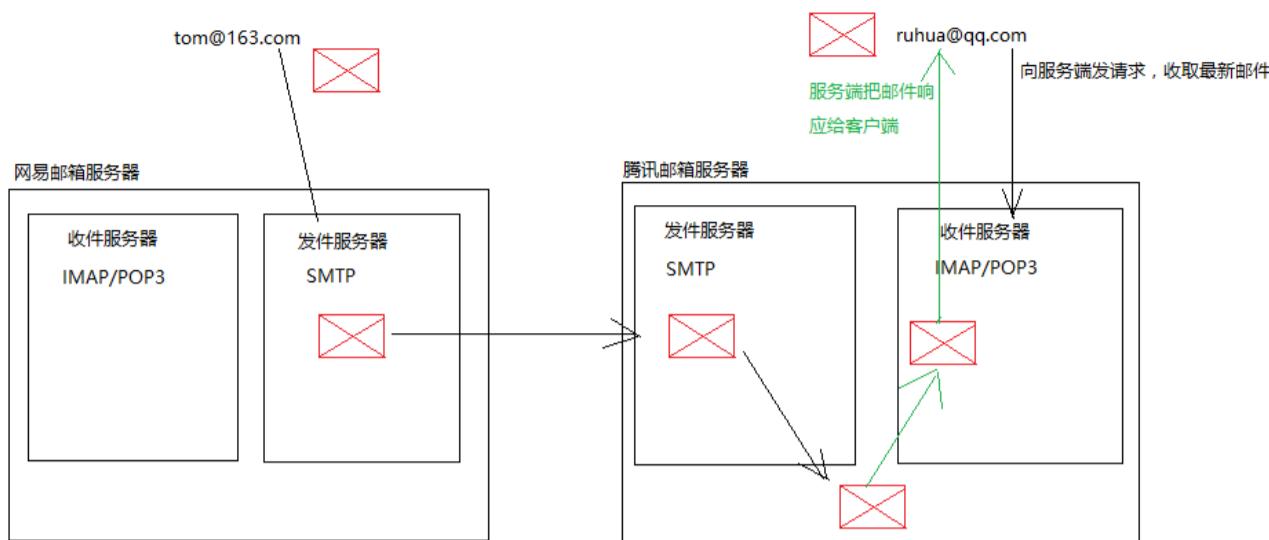
`SMTP`: Simple Mail Transfer Protocol, 简单邮件发送协议

收件协议:

`IMAP`: Internet Mail Access Protocol, 互联网邮件访问协议

`POP3`: Post Office Protocol - Version 3, 邮局协议版本 v3

邮件收发过程



20-9-3 Java 程序发送邮件

相关的 jar 包



mail.jar

<--点击选中，然后 ctrl+C 复制，到桌面上 ctrl+v 可保存到本地电脑上

使用工具类发送邮件，工具类代码如下

```
public class MailUtil {  
    /**  
     * 发送邮件  
     * @param email 收件人的邮箱地址  
     * @param subject 邮件主题  
     * @param emailMsg 邮件内容  
     */  
  
    public static void sendMail(String email, String subject, String emailMsg)  
        throws AddressException, MessagingException {  
        // 1. 创建一个程序与发件人的 发送邮件服务器会话对象 Session  
        Properties props = new Properties();  
        props.setProperty("mail.transport.protocol", "SMTP");//邮件发送协议  
        props.setProperty("mail.host", "smtp.163.com");//邮件发送服务器的地址  
        props.setProperty("mail.smtp.auth", "true");//指定验证为 true  
  
        // 创建验证器  
        Authenticator auth = new Authenticator() {  
            public PasswordAuthentication getPasswordAuthentication() {
```

```
//发件人的用户名（不带后缀的）和授权码（如果没有授权码，就使用密码）
return new PasswordAuthentication("liuyp_itcast", "heima100");
}

};

Session session = Session.getInstance(props, auth);

// 2.创建一个 Message, 它相当于是邮件内容
Message message = new MimeMessage(session);
//设置发送者的邮箱地址
message.setFrom(new InternetAddress("liuyp_itcast@163.com"));
//设置发送方式与接收者
message.setRecipient(RecipientType.TO, new InternetAddress(email));
//邮件主题
message.setSubject(subject);
//设置邮件的内容
message.setContent(emailMsg, "text/html;charset=utf-8");

// 3.创建 Transport 用于将邮件发送
Transport.send(message);
}

}
```

21-jQuery 上

21-1_概述

21-1-1_简介

jQuery，是一个优秀的 JavaScript 轻量级框架，它兼容 css3 和各大浏览器。jQuery 提供了 dom、event、animate、ajax 等的简易操作。jQuery 有非常丰富的插件，大多数需求都有对应的插件解决方案。jQuery 的宗旨：write less, do more.

21-1-2_下载与引入

www.jquery.com

21-1-3_js 和 jQuery 的区别和联系

jQuery 的本质是 js，是 js 的再封装

js 对象和 jQuery 对象不同但可以相互转换

js 对象转 jQuery 对象: \$(js 对象)

jQuery 对象转 js 对象: jQuery 对象[索引] 或者 jQuery 对象.get(索引)

js 和 jQuery 的事件写法不同，但意义相同

js 事件名称带 on; jQuery 事件名称不带 on

js 事件是属性; jQuery 事件是方法

js 响应行为是事件属性的值; jQuery 响应行为是事件方法的实参

js 和 jQuery 的页面加载完成事件不同

jQuery 页面加载完成有简写形式; 而 js 没有

\$(function(){});

\$(document).ready(function(){});

jQuery 的页面加载完成可以有多个; 而 js 只有一个有效的（最后一个有效）

21-2_dom 操作-获取标签-选择器（**重点**）

js 的获取标签:

getElementById

getElementsByName

getElementsByTagName

getElementsByClassName

21-2-1_基本选择器

- 标签选择器 **\$("div")**
- id 选择器 **\$("#id 值")**
- 类选择器 **\$(".class 名称")**

21-2-2_层级选择器

假定: A 和 B 是选择器字符串

- **\$("A B")** 获取 A 的后代 B 元素-祖孙选择器
- **\$("A>B")** 获取 A 元素的子元素 B-父子选择器
- **\$("A+B")** 获取 A 元素后边相邻的 B 元素-兄弟选择器
- **\$("A~B")** 获取 A 元素后边同级所有的 B 元素-弟弟选择器

21-2-3_属性选择器

- **\$("A[attr]")**: 获取包含有 attr 属性的 A 元素
- **\$("A[attr='v']")**: 获取 attr 属性值是 v 的 A 元素

- `$("#A[attr^='v'])`: 获取 attr 属性值以 v 开头的 A 元素
- `$("#A[attr$='v'])`: 获取 attr 属性值以 v 结尾的 A 元素
- `$("#A[attr*='v'])`: 获取 attr 属性值包含 v 的 A 元素
- `$("#A[attr!= 'v'])`: 获取 attr 属性值不等于 v 的 A 元素
- `$("#A[attr='v'][attr2])`: 复合属性选择器

21-2-4_基本过滤选择器

一般是和其它选择器配合使用，对其它选择器的结果进行再次过滤

- `:first` 获取第一个 `$("#div:first")`: 原理是先获取所有 div，然后从中获取第一个
- `:last` 获取最后一个 `($("#span:last")`
- `:not(selector)` 不要 selector 的结果 `($("#div:not(.cls)")`
- `:odd` 获取索引为奇数的元素 `($("#div:odd")`: 原理是先获取所有 div，得到数组，再对这个结果数据的索引进行过滤
- `:even` 获取索引为偶数的元素
- `:gt(n)` 获取索引大于 n 的元素 `($("#div:gt(2)")`: 先获取所有 div，然后对结果数据进行索引过滤
- `:lt(n)` 获取索引小于 n 的元素
- `:eq(n)` 获取索引等于 n 的元素
- `:animated` 获取动起来的动画
- `:header` 获取所有标题元素 h1~h6

21-2-5_表单属性选择器

- `:disabled` 获取不可用的
- `:enabled` 获取可用的
- **`:selected` 获取被选中的下拉框选项 option 标签**
- **`:checked` 获取被选中的单选或者多选项**

21-3_dom 操作-其它操作

21-3-1_操作标签体

相当于 js 的 innerHTML，开始标签和结束标签里的内容
获取标签体：jQuery 对象.html()
设置标签体：jQuery 对象.html("<h1>代码会生效</h1>");

21-3-2_操作标签体里的文本

相当于 js 里的 innerText
获取文本：jQuery 对象.text();
设置文本：jQuery 对象.text("<h1>代码会生效</h1>");

21-3-3_操作表单项的值 val (重点)

可以操作任意表单项的值，比如：文本框、密码框、单选、多选、下拉框的值、文本域的值
获取值：jQuery 对象.val();
设置值：jQuery 对象.val(值);

21-3-4_操作样式

获取样式：jQuery 对象.css("样式名称");
设置样式：jQuery 对象.css("样式名称", "样式值");

21-3-5_操作标签的属性

获取属性：
jQuery 对象.attr("属性名");
jQuery 对象.prop("属性名");
设置属性：
jQuery 对象.attr("属性名", 值);
jQuery 对象.prop("属性名", 值);
删除属性：
jQuery 对象.removeAttr("属性名");
jQuery 对象.removeProp("属性名");
如果操作的是 checked 和 selected 属性，必须使用 prop 相关的方法。其它属性建议优先尝试 attr 的方法，如果不行再使用 prop 的方法。

21-3-6_操作 class 属性

添加 class：jQuery 对象.addClass("class 名称");
删除 class：jQuery 对象.removeClass("class 名称");
切换 class：jQuery 对象.toggleClass("class 名称");
 如果标签有指定的 class，删除掉
 如果标签没有指定的 class，添加上

21-4_jQuery 简易动画（了解）

21-4-1_显示/隐藏

显示：jQuery 对象.show(毫秒值, 函数对象);
毫秒值：动画需要花费多长时间执行完成
函数对象：动画执行完成后，jQuery 框架会自动调用的函数
隐藏：jQuery 对象.hide(毫秒值, 函数对象)
切换：jQuery 对象.toggle(毫秒值, 函数对象)

21-4-2_滑动显示/隐藏

滑动显示: `jQuery 对象.slideDown(毫秒值, 函数对象);`

滑动隐藏: `jQuery 对象.slideUp(毫秒值, 函数对象);`

滑动切换: `jQuery 对象.slideToggle(毫秒值, 函数对象);`

21-4-3_淡入显示/淡出隐藏

淡入显示: `jQuery 对象.fadeIn(毫秒值, 函数对象);`

淡出隐藏: `jQuery 对象.fadeOut(毫秒值, 函数对象);`

淡入淡出切换: `jQuery 对象.fadeToggle(毫秒值, 函数对象);`

21-5_操作 jQuery 的核心思想

使用 **jQuery** 来操作 **html**, 通常只需要两步:

1. 操作谁, 用选择器找到
2. 要怎么操作, 用 **jQuery** 的方法执行

22-jQuery 下

22-1_dom 操作-操作标签

22-1-1_创建标签

```
document.createElement( "li" ); // js 方式
```

```
$( "<li id=' a' >标签体</li>" )
```

22-1-2_插入标签

正向插入

`A.append(B):` 把 B 插入 A 内部的最后

`A.prepend(B):` 把 B 插入到 A 内部最前 pre:previous

`A.before(B):` 把 B 插入 A 的前边

`A.after(B):` 把 B 插入到 A 的后边

反向插入

`A.appendTo(B):` 把 A 插入到 B 内部最后

`A.prependTo(B):` 把 A 插入到 B 内部最前

`A.insertBefore(B):` 把 A 插入到 B 前边

`A.insertAfter(B):` 把 A 插入到 B 后边

22-1-3_删除标签

jQuery 对象.remove(); 删除自己和所有的子标签
jQuery 对象.empty(); 删除所有子标签，但是自己保留

22-2_jQuery 的事件

22-2-1_常用事件写法

事件名称	描述
click(fn)	监听单击事件
dblclick(fn);	监听双击事件
change(fn)	监听域内容改变事件，一般监听下拉框的选项发生变化
submit(fn)	监听表单提交事件，用在 form 上
\$(fn), \$(document).ready(fn)	监听页面加载完成事件事件
focus(fn)	
blur(fn)	
keydown(fn)	
keyup(fn)	
keypress(fn)	
mouseover(fn)	
mouseout(fn)	
mousemove(fn)	
mousedown(fn)	
mouseup(fn)	

22-2-2_事件绑定与解绑（了解）

1、事件绑定 **on** 函数—常用于给未来元素绑定事件

未来元素：js 或者 jQuery 事件绑定时，事件源还不存在。事件绑定之后增加的元素叫未来元素
给未来元素绑定事件：
现在元素的 jQuery 对象.on(“事件名称”，“选择器字符串”，function(){

```
//响应行为代码
});

事件名称: click, dblclick, submit 等等不带 on 的字符串
选择器字符串: 现在元素内部, 根据这个字符串查找的结果, 绑定事件
```

示例:

```
//使用 on 函数, 给 ul 里所有的 li 绑定单击事件
$("ul").on("click", "li", function(){
    alert(this.innerHTML);
});

//事件绑定之后, 又增加了一个 li 标签(未来元素), 未来元素被点击时也会弹窗
$("<li>d</li>").appendTo("ul");
```

2、事件解绑 off 函数

jQuery 对象.off("事件名称");

示例:

```
//给 li 解绑 click 事件
$("li").off("click");

//如果是使用 on 绑定的事件, 要使用下面的方式来解绑
$("#itcast").off("click", "li");
```

22-2-3_鼠标悬停事件函数: hover

鼠标悬停并不是一个真正的事件, 而是指鼠标移入和鼠标移出这一对事件

```
jQuery 对象.hover(fn1, fn2): fn1 是鼠标移入的响应行为函数, fn2 是鼠标移出的响应行为函数
jQuery 对象.hover(fn): fn 表示鼠标移入和鼠标移出共同的响应行为
```

22-3_jQuery 的循环遍历

22-3-1_原生 JavaScript 提供的 for 循环

1、基本 for 循环

```
var items = $("option");
for (var item=0; item<items.length; item++) {...}
```

2、增强 for 循环：for...of..

语法：是 JavaScript 的语法规规 ECMAScript 第 6 版提供的

```
for(变量 of 被循环遍历的对象){  
    alert(变量);  
}
```

例如：

```
var arr = ["a", "b", "c"];  
for (item of arr) {  
    alert(item);  
}
```

22-3-2_jQuery 框架的全局 each 方法

```
$.each(被循环遍历的对象, function(index, element){  
    //index: 索引  
    //element: 被循环遍历对象中的每一个元素对象  
    //this: 和 element 是一样的作用  
});
```

22-3-3_jQuery 对象的 each 方法

```
jQuery 对象.each(function(index, element){  
    //index: 索引  
    //element: 被循环遍历对象中的每一个元素对象  
    //this: 和 element 是一样的作用  
});
```

22-3-4_对 jQuery 对象进行循环遍历注意

jQuery 对象类似数组，里边每一个元素是 js 对象。

22-4_表单校验插件（重点）

22-4-1_引入的文件

```
jQuery 类库  
表单校验插件的 js 文件
```

22-4-2_基本校验语法

```
表单的 jQuery 对象.validate({
    rules:{    //校验规则
        表单项的 name:{    //校验规则
            规则名称:规则值,
            ...
            规则名称:规则值
        }
    },
    messages:{    //提示信息
        表单项的 name:{    //提示信息
            规则名称:” 提示信息 ” ,
            ...
            规则名称:” 提示信息 ”
        }
    }
});
```

22-4-3_自定义错误信息的位置

把下面的标签加入到需要显示错误信息的地方，插件会自动在这个标签里显示对应的提示信息

```
<label class=" error"  for=" 表单项的 name 值 " ></label>
```

注意：

必须有 class=“ error ”， 插件会根据 error 的 class 值来找 label 标签

必须有 for 属性，属性值表单项的 name 值：表示这个 label 标签是为哪个表单项服务的

22-4-4_自定义校验规则

```
$.validator.addMethod( “规则名称” , function(value, element, params){
    //value:要校验的值
    //element: 要校验的表单项标签对象
    //params: 配置规则时，规则的值

    //如果校验通过 return true; 如果校验不通过， return false;
}, “默认提示信息” );
```

22-5_jQuery 的文档筛选函数（扩展：了解）

用来获取 html 标签对象的，是对 jQuery 选择器的补充。可以获取到父标签、兄弟标签等等

jQuery 对象.find(selector): 从 jQuery 对象内部查找 selector 选择器符合的标签元素
jQuery 对象.parent([selector]): 获取 jQuery 标签对象的父标签
jQuery 对象.parents([selector]): 获取 jQuery 标签对象的所有上级标签
jQuery 对象.siblings([selector]): 获取 jQuery 标签对象的兄弟姐妹标签
jQuery 对象.children([selector]): 获取 jQuery 标签对象的子标签
jQuery 对象.next([selector]): 获取 jQuery 标签对象后边的标签
jQuery 对象.nextAll([selector]): 获取 jQuery 标签对象后边所有的标签
jQuery 对象.prev([selector]): 获取 jQuery 标签对象前边的标签
jQuery 对象.prevAll([selector]): 获取 jQuery 标签对象前边的所有标签

22-6_js 和 jQuery 中的 this

1. 把 this 写在 html 标签里，表示当前标签对象

```
<input type="checkbox" onclick="showWin(this)" id="chk" checked="checked"/>
```

2. 动态绑定事件时，在响应行为函数里，this 表示当前事件源标签对象

```
document.getElementById("btn").onclick = function(){
    this: 表示当前事件源标签对象， btn
}
$("#btn").click(function(){
    this: 表示当前事件源标签对象
});
```

3. 在 jQuery 的循环遍历的 function 里，this 表示被循环遍历对象中的每一个元素对象

```
var arr = ["a", "b", "c"];
$.each(arr, function(){
    this: 表示被循环遍历对象中的每一个元素对象
});
```

22-7_总结

- 会使用 jQuery 创建、插入、删除标签
- **jQuery 的基本事件绑定方式**
- 了解 jQuery 的 on 和 off 函数-使用 on 给未来元素绑定事件
- 了解 jQuery 的 hover 函数
- **会使用几种循环遍历的方式**
- **会使用表单校验插件—引入文件**
- **会使用表单校验插件—通过基本语法使用常用的校验规则**

- 会使用表单校验插件—自定义错误信息的位置
- 会使用表单校验插件—自定义校验规则

23-Ajax&Json

23-1_Ajax

23-1-1_概述

1、简介

Ajax: Asynchronous JavaScript and XML，异步的 JS 和 XML。

JavaScript: Ajax 是 JavaScript 提供的技术

XML: Ajax 原生是使用 XML 进行客户端和服务端之间的数据交互的。目前 XML 已经被 json 取代了。

2、作用

局部刷新：在不刷新整体页面的情况下，向服务端发送请求，接收服务端的响应数据，对当前页面的某一部分进行更新

异步加载：

同步：客户端发送请求之后，浏览器处于等待服务器响应的"假死"状态，浏览器不能进行其它操作

异步：客户端发送请求之后，可以继续执行其它操作，而不会处于"假死"状态。等到服务器响应完成之后，可以继续后边的操作

3、Ajax 原理--js 的 Ajax（了解）

js 的 Ajax-基本步骤

1. 创建 Ajax 引擎对象
2. 给 Ajax 引擎对象绑定事件，监听它的状态变化，设置响应行为：主要监听 Ajax 接收到服务端响应之后，我们要做的事情
3. 给 Ajax 引擎配置请求相关的参数
4. 发送请求

js 的 Ajax-示例

```
<input type="button" value="ajax " onclick="jsAjax()">

function jsAjax(){

    //1. 创建 Ajax 引擎对象
    var xmlhttp = new XMLHttpRequest();

    //2. 给 Ajax 引擎对象绑定事件，监听它的状态变化，设置响应行为：主要监听 Ajax 接收到服务端响应之后，我们要做的事情
    //window.onload = function(){}
    xmlhttp.onreadystatechange = function(){

        if(xmlhttp.readyState == 4 && xmlhttp.status == 200){
            document.getElementById("result").innerHTML = xmlhttp.responseText;
        }
    }
}
```

```

//判断一下 xmlhttp 状态为 4 的话，才需要处理后边的内容
if(xmlhttp.readyState==4){

    //只有服务端正常响应完成时，才需要处理后边的内容

    if(xmlhttp.status==200){

        var result = xmlhttp.responseText;

        //把 result 设置成为 d1 的标签体

        document.getElementById("d1").innerHTML = result;

    }

}

//3. 给 Ajax 引擎配置请求相关的参数：请求方式 请求地址 是否异步
xmlhttp.open("GET",      "${pageContext.request.contextPath}/jsAjaxServlet?username=张三
&password=111", true);

//4. 发送请求
xmlhttp.send();

}

```

23-1-2_jQuery 的 Ajax (重点)

js 的 Ajax 每次操作时，都需要写固定的代码，重复代码过多。每次在操作时，不太方便。jQuery 对 js 的 Ajax 进行了二次封装，封装后提供了实现 Ajax 功能的一些方法。只需要传递 ajax 请求必须的参数，就能够实现 ajax 的效果。

jQuery 的 Ajax 解决了原生 js 的 Ajax 的兼容问题。

jQuery 提供了很多 Ajax 相关的方法，但是常用的有三个：

`jQuery.post(url, [params], [callback], [type])`

`jQuery.get(url, [params], [callback], [type])`

`jQuery.ajax(url,[settings])`

1、`$.post(url, [params], [callback], [type])`

参数：

`url:` 请求的地址，必须

`params:` 请求的参数，格式：`name=value&name=value…`

name=value：等号“=”两边不能有空格

`callback:` 回调函数。表示当服务端响应完成以后，jQuery 框架会执行这个回调函数。

`function(result){`

`//result 形参，谁调用谁传实参。jQuery 框架传递的实参进来，值是服务端响应回来的内容`

```
}
```

type: 服务端响应回来的数据内容的类型。text/json

应用示例：

```
var url = "${pageContext.request.contextPath }/jqAjaxServlet";  
var params = {username:"jerry", password:"jerry"};  
$.get(url, params, function(result){  
    alert(result);  
}, "text");
```

2、\$.get(url, [params], [callback], [type])

这个方法的应用和 post 方法的应用是完全一样的。不同的是：这个方法发送的是 get 请求

3、\$.ajax({setting})

比 post 和 get 方法更接近于 ajax 的底层，配置项更多，应用更灵活，但是不是很方便。

配置项：格式：{key:value, key:value, …}，可以配置多个配置项，以 key-value 的形式。

url: 请求的地址。必须

data: 请求参数。格式：name=value&name=value… 或者是 {name:value, name:value, …}

type: 请求方式。get/post

dataType: 服务端返回的数据类型。常用的 text/json

success: 回调函数，Ajax 引擎接收到服务端响应之后，jQuery 框架会执行的回调函数对象。

async: 是否异步。默认是 true，表示是异步请求

应用示例：

```
$.ajax({  
    url:"${pageContext.request.contextPath }/jqAjaxServlet", //请求地址，必须  
    data:"username=zhangsan&password=lisi", // 请求参数，也可以写成 {username:"zhangsan",  
    password:"lisi"}  
    type:"post", //请求方式，默认 get。  
    dataType:"text", //服务端响应回来的数据类型。text/json  
    success:function(result){//服务端响应完成后，jQuery 框架会执行回调函数对象  
        alert(result);  
    },  
    async:true //是否异步，默认是 true  
});
```

23-2_JSON

23-2-1_简介

JSON: JavaScript Object Notation, JS 对象标记法。是 JS 提供的一种轻量级的数据格式，主要是用来代替 XML 的。

定义 JSON:

对象定义方式: `var json = {"key":value, "key":value, ...};`

数组定义方式: `var json = [obj1, obj2, ...];`

第三种方式: 两种定义方式可以混用

解析 JSON:

`json.key;`

`json[索引]`

23-2-2_JavaBean 转换成 JSON 格式字符串

服务端向客户端传递数据的时候，可能会有一些比较复杂的数据，比如: JavaBean 对象，或者是 JavaBean 对象的集合。使用文本形式传递太麻烦，可以使用 JSON 的格式。

服务端把 JavaBean 对象转换成 JSON 格式的字符串，把这个字符串响应给客户端。客户端的 jQuery 在接收到这个字符串之后，会把它转换成一个 `json` 对象，传递给回调函数内部。

问题是: JavaBean 对象 (JavaBean 集合) 转换成 JSON 格式字符串太繁琐，可以使用工具包来帮我们完成这个工作。

1、常用的 JSON 转换工具包

`jsonlib`: 依赖的 jar 包过多，操作比较麻烦

`gson`: Google 提供的工具包，操作简单

`fastjson`: Alibaba 提供的工具包，操作简单，而且效率高

`Jackson`: 开源免费的工具包，springMVC 框架默认采用的转换方式

2、`jsonlib` 应用示例

```
//创建 JavaBean 对象
```

```
Person p1 = new Person("张三", 20);
```

```
Person p2 = new Person("李四", 22);
```

```
//创建 JavaBean 集合
```

```
List<Person> list = new ArrayList<Person>();
```

```
list.add(p1);
```

```
list.add(p2);

//把 JavaBean 对象转换成 json 格式字符串
JSONObject jsonObject = JSONObject.fromObject(p1);
String json = jsonObject.toString();
System.out.println(json);

//把 JavaBean 集合转换成 json 格式字符串
JSONArray jsonArray = JSONArray.fromObject(list);
System.out.println(jsonArray.toString());
```

3、gson 应用示例

```
// 创建 JavaBean 对象
Person p1 = new Person("张三", 20);
Person p2 = new Person("李四", 22);

// 创建 JavaBean 集合
List<Person> list = new ArrayList<Person>();
list.add(p1);
list.add(p2);

Gson gson = new Gson();
// 把 JavaBean 对象转换成 json 格式字符串
String json = gson.toJson(p1);
System.out.println(json);

// 把 JavaBean 集合转换成 json 格式字符串
json = gson.toJson(list);
System.out.println(json);
```

4、fastjson 应用示例

```
// 创建 JavaBean 对象
Person p1 = new Person("张三", 20);
Person p2 = new Person("李四", 22);

// 创建 JavaBean 集合
```

```
List<Person> list = new ArrayList<Person>();  
list.add(p1);  
list.add(p2);  
  
// 把 JavaBean 对象转换成 json 格式字符串  
String json = JSONObject.toJSONString(p1);  
System.out.println(json);  
  
// 把 JavaBean 集合转换成 json 格式字符串  
json = JSONObject.toJSONString(list);  
System.out.println(json);
```

5、Jackson 应用示例

```
// 创建 JavaBean 对象  
Person p1 = new Person("张三", 20);  
Person p2 = new Person("李四", 22);  
  
// 创建 JavaBean 集合  
List<Person> list = new ArrayList<Person>();  
list.add(p1);  
list.add(p2);  
  
ObjectMapper mapper = new ObjectMapper();  
// 把 JavaBean 对象转换成 json 格式字符串  
String json = mapper.writeValueAsString(p1);  
System.out.println(json);  
// 把 JavaBean 集合转换成 json 格式字符串  
json = mapper.writeValueAsString(list);  
System.out.println(json);
```

23-3_总结

- .会使用 jQuery 的 get()方法发送 Ajax GET 请求
- .会使用 jQuery 的 post()方法发送 Ajax POST 请求
- .会使用 JavaScript 获取 JSON 里的数据
- .会使用任意一种工具包把 JavaBean 转换成 JSON 字符串

24-linux

24-1_概述

24-1-1_什么是 Linux

Linux 是一个类 Unix 的开源免费的操作系统，继承了 Unix 的设计思想，是一个性能稳定的多用户网络操作系统，由 Linus Torvalds 开发，并于 1991 年公布的。现在已经衍生出了成百上千各不同的 Linux 分支。

由于出色的稳定性和安全性，Linux 几乎成为程序运行的最佳环境，它的应用范围非常广泛：不仅可以运行我们的程序代码，还广泛的应用于各种计算机设备中。例如：手机、平板、路由器等等，尤其需要提及的是，我们熟知的 Android 就是运行在 Linux 系统上的。



图 1-Linus Torvalds

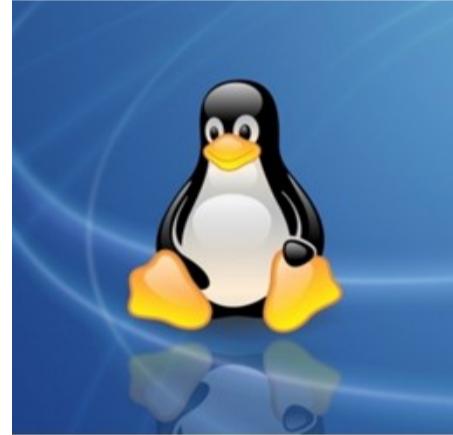


图 2-Linux 的 LOGO

24-1-2_Linux 的分类

严格来讲，Linux 这个词本身只表示 Linux 系统的内核，但是实际上人们已经习惯了用 Linux 来形容整个基于 Linux 内核的操作系统。

1、按照市场需求分类

服务器版：没有图形化界面，使用命令行的方式进行操作

桌面版：有类似于 Windows 的图形化界面，但是不够成熟稳定

2、按照原生程度分类

内核版：Linus 领导的开发小组所开发维护的 Linux 内核版本

发行版：一些企业、组织、社区在 Linux 内核基础上，进行再次开发，重新发行的版本



24-2_安装 Linux

24-2-1_安装虚拟机

常见的虚拟机软件

VirtualBox: Oracle 公司的免费的虚拟机软件

VMWare: 威睿公司的收费的虚拟机软件—市面上应用的比较多

VMWare 的安装步骤

参考文档: 《01-虚拟机 vmware 安装.doc》

24-2-2_在虚拟机安装 CentOS

CentOS 是 Linux 的一个发行版, 是目前企业中用来作为服务器的主要版本。我们通过 WMWare 虚拟出来一台计算机, 把 CentOS 安装到这个虚拟机上。

参考文档: 《02-centOS6.7 安装.doc》

24-3_Linux 常用命令

Linux 中的命令成千上百, 课程中没有必要也不可以一一涉及, 根据程序员在日常工作中的需求, 总结出几种常用的命令进行学习。

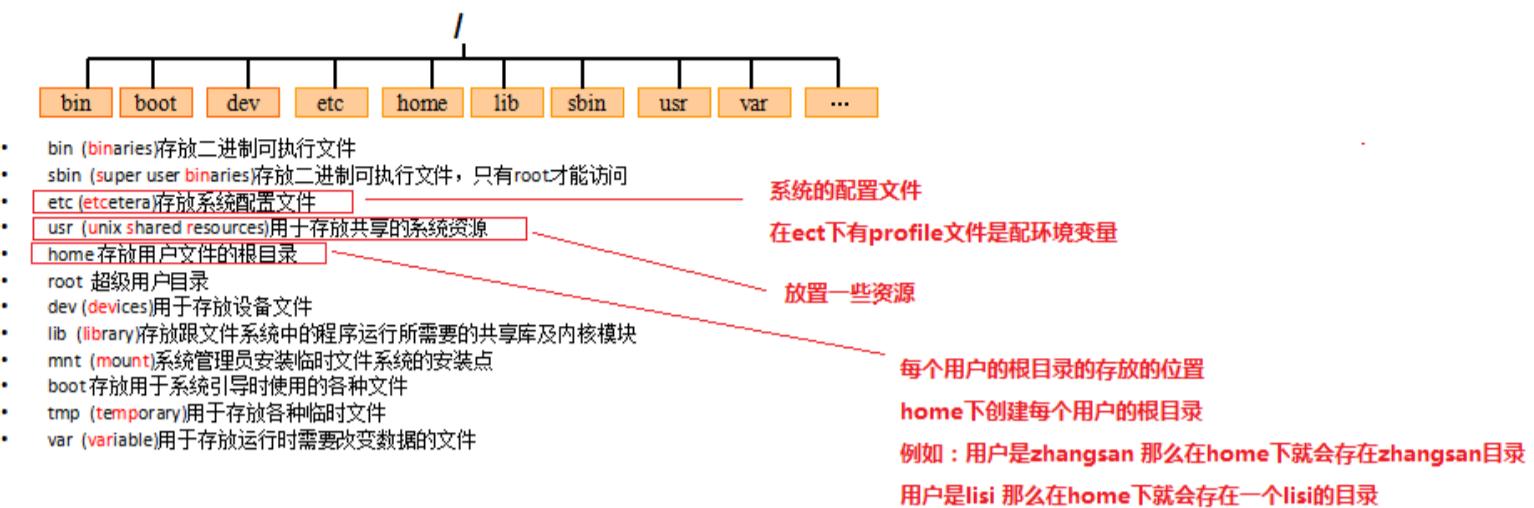
Linux 命令的基础语法:

命令 [-参数 [参数值]]

24-3-1_Linux 的目录结构

Linux 的目录结构

Linux 目录结构



Linux 里切换目录的命令

基本命令：**cd 目录**

cd: change directory

应用示例：

cd / → 切换到根目录

cd /usr → 切换到根目录下的 usr 目录中

cd .. → 切换到上级目录

cd ./a → 切换到当前目录的 a 文件夹中，可以省略掉./不写

cd ~ → 切换到当前用户的 home 目录

cd - → 切换到刚才所在的目录

显示当前在哪个文件夹里

命令：**pwd**

print working directory

24-3-2_操作目录(文件夹)的命令

常见的操作文件夹的命令

mkdir: make directory, 创建文件夹

ls → list, 列出文件夹(或文件)

mv → move, 移动文件夹(或文件)

cp → copy, 拷贝文件夹(或文件)

rm → remove, 删除文件夹(或文件)

find → 查找文件夹(或文件)

创建文件夹 **mkdir**

基本语法: **mkdir 文件夹名称**

应用示例

mkdir test → 在当前目录里创建文件夹 **test**

mkdir /usr/test → 在 /usr 目录里创建文件夹 **test**

查看文件夹 **ls**

基本语法: **ls [-al]**

参数:

a: all → 表示列出全部, 包括隐藏文件也会显示出来

l: list → 表示以列表形式显示详细信息

应用示例

ls -a → 列出当前目录里的所有文件(夹)

ls -l → 以列表形式列出当前目录里的文件(夹)

ls -al → 以列表形式列出当前目录里的所有文件(夹)

ll → 是 **ls -l** 的缩写

搜索文件夹 **find**

基本语法: **find 搜索位置 -name “文件名称”**

参数:

name: 文件名称表达式, 可以使用通配符: * 任意多个字符, ? 一个任意字符

应用示例

find ./ -name “*log” → 从当前目录里查找名称包含“log”的文件

移动(剪切)文件夹 **mv**

基本语法: **mv 文件夹名称 目标位置**

应用示例

mv test / → 把 **test** 文件夹移到到根目录下(移动即剪切)

说明: 目录位置的文件夹存在, 就是移动文件到目录文件夹

重命名文件夹 **mv**

基本语法: **mv 文件夹名称 新名称**

应用示例

mv test test2 → 把 test 文件夹重命名为 test2

mv 第二个参数文件不存在，就会命名成这个名称的文件夹

```
drwxr-xr-x. 2 root root 4096 Jul 30 18:11 heima.log
[root@heima heima]# mv heima.log/ heima41
[root@heima heima]# ll
total 4
drwxr-xr-x. 2 root root 4096 Jul 30 18:11 heima41
[root@heima heima]# mv heima41/ ../
[root@heima heima]# ll
total 0
[root@heima heima]# cd ../
[root@heima ~]# ll
total 52
-rw----- 1 root root 1430 Jul 30 17:33 anaconda-ks.cfg
drwxr-xr-x. 2 root root 4096 Jul 30 18:15 heima
drwxr-xr-x. 2 root root 4096 Jul 30 18:11 heima41
-rw-r--r--. 1 root root 26150 Jul 30 17:33 install.log
-rw-r--r--. 1 root root 7572 Jul 30 17:32 install.log.syslog
[root@heima ~]#
```

拷贝文件夹 cp

基本语法: **cp -r** 文件夹 目标位置

参数:

r: 表示执行迭代操作

应用示例:

cp -r heima41 heima 把 heima41 文件夹拷贝一份为 heima (文件夹里的内容也会一同拷贝)

删除文件夹 rm

基本语法: **rm -rf** 文件夹

参数:

r: 表示执行迭代操作

f: 表示强制执行不提示

应用示例:

rm -r heima 删除文件夹 heima, 删除每一个文件之前都需要确认 (文件夹的内容也会一并删除)

rm -rf heima 删除文件夹 heima, 强制删除不提示确认信息 (文件夹的内容也会一并删除)

24-3-3_操作文件的命令

常用的操作文件的命令

touch: 创建文件 (文本类文件)

more/less/cat/tail: 查看文件

vi/vim: 修改文件 (vi 和 vim 是文本编辑器)

rm: remove, 删除文件 (夹)

mv: move, 移动文件、重命名文件

cp: copy, 拷贝文件 (夹)

grep: Globally search a Regular Expression and Print, 搜索文件内容

创建文件 touch

基本语法: **touch** 文件名称

应用示例:

```
touch heima.txt      创建一个文件 heima.txt
```

查看文件 cat/more/less/tail

基本语法:

cat 文件名称: 查看文件内容, 不能翻页

more 文件名称: 查看文件内容, 空格键往后翻页; 但只能往后看, 不能往回看。按 Q 退出查看

less 文件名称: 查看文件内容, 空格键往后翻页, 或者翻页键往前、往后翻页看, 也可以↑↓向前向后一行行看. 按 Q 退出查看

tail 文件名称: 查看文件尾部内容, 通常用于查看日志文件

应用示例:

```
cat install.log
```

```
more install.log
```

```
less install.log
```

```
tail install.log
```

编辑文件 vi/vim

基本语法: **vim** 文件名称

在 Windows 中可以使用记事本、或者其它文本编辑器来编辑文件内容。而 Linux 中没有记事本, 但有类似的文本编辑器: vi, 或者 vim。

vi/vim 的基本应用方法是相同的, 都有三种模式: 普通模式、编辑模式、底行模式

普通模式: 使用 vi/vim 默认进入普通模式。在这个模式按"**i/a/o**"进入编辑模式, 或者按":"进入底行模式

编辑模式: 对文件内容进行编辑修改。

底行模式: 执行保存、退出等操作

使用步骤如下:

vim 文件名称-->默认进入普通模式

在普通模式按"**i**"键-->进入编辑模式: 可以修改文件内容 (不要 **ctrl+s**)

按"**ESC**"退出编辑模式-->进入普通模式

在普通模式按"**:**"键-->进入底行模式: 可以操作文件

输入"**wq**"保存并退出编辑器

输入"**q!**"强制退出不保存

搜索文件内容 grep

基本语法: **grep 表达式 文件 [--color]**

在指定文件里搜索符合表达式的内容

参数:

--color: 注意前边 2 个横杠, 表示要把符合表达式的内容高亮标示

应用示例

```
grep "java" ./instal.log --color
```

复制文件 cp

基本语法: **cp 原文件 文件名**

应用示例:

```
cp heima36.txt heima.txt      把 heima36.txt 拷贝一份成为 heima.txt
```

移动文件 mv

基本语法: **mv 文件名 目标位置**

应用示例

```
mv heima.txt /      把 heima.txt 移动到根目录下
```

重命名文件 mv

基本语法: **mv 文件名 新文件名**

应用示例

```
mv heima.txt heima111.txt      把 heima.txt 重命名为 heima111.txt
```

删除文件 rm

基本语法: **rm -f 文件名**

参数:

f: 表示强制删除不提示 force

应用示例

```
rm -f heima111.txt
```

24-3-4_压缩与解压缩命令

Linux 中压缩与解压缩相关的术语

打包: 把多个文件打包归档到一起, 形成一个新文件, 但并没有进行压缩。打包文件后缀名通常是: **.tar**

压缩: 压缩文件。后缀名通常是: **.gz**

打包并压缩: 把多个文件打包归档并压缩。后缀名通常是: **.tar.gz**

压缩文件 tar

基本语法: **tar -zcvf 压缩包名 要压缩的文件 1 要压缩的文件 2**

参数:

- z**: 调用压缩命令进行压缩
- c**: 要创建压缩包文件
- v**: 显示压缩过程
- f**: 指定要创建的压缩包文件名称

应用示例

```
tar -zcvf log.tar.gz install.log install.log.syslog
```

把 `install.log` 和 `install.log.syslog` 两个文件，打包并压缩到 `log.tar.gz` 中

解压缩文件 tar(重点)

基本语法: **tar -xvf 压缩包名 -C 解压位置**

参数:

- x**: `extract`, 表示提取文件
- v**: 显示解压过程
- f**: 压缩包的名称
- C**: 解压到哪个位置。注意: 解压位置的文件夹必须是已经创建好的, 或者是已经存在的
如果不指定 **C** 参数, 表示解压到当前文件夹里

应用示例:

```
tar -xvf log.tar.gz -C ./log/
```

把压缩包 `log.tar.gz` 解压到当前文件夹的 `log` 文件夹里

24-3-5_其他常用命令

查看进程 ps progress show

基本语法: **ps -ef**

参数:

- e**: 显示所有程序
- f**: 显示进程详细信息, 包括 (了解):
 - uid**: 哪个用户的进程
 - pid**: 进程 id
 - ppid**: 父进程的 id
 - c**: 进程占用 CPU 的百分比
 - stime**: 系统启动时间
 - time**: CPU 使用时间
 - cmd**: 是哪个命令的进程

结束进程 kill

基本语法: **kill -9 pid** 强制结束 pid 对应的进程

应用示例:

`kill -9 132` 强制结束 pid 为 132 的进程

管道命令

有时候一个操作，涉及到多个命令才可以完成，可以使用|来把多个命令连接起来。表示把前边命令执行的结果交给后边的命令来处理。

应用示例:

ps -ef|less 查找所有进程信息，使用 less 命令来查看

前边: `ps -ef` 表示查找所有进程信息（信息太多，不方便看）

中间: | 表示把前边命令的结果，交给后边的命令来处理

后边: `less` 使用 less 命令查看内容

ps -ef | grep "java" --color 查找所有进程信息，然后获取其中包含"java"字符串的内容

前边: `ps -ef` 表示查看所有进程。（可能查看到很多进程信息，要从中查找某个进程不方便）

中间: | 表示把前边命令的结果，交给后边的命令来处理

后边: `grep "java" --color` 表示从中查找包含"java"的内容，并高亮显示出来

网络通信命令

ifconfig: 查看网络配置信息

ping: 测试网络

netstat -anp: 查看网络连接情况（这个命令可以查询占用某端口的进程 pid，然后用 kill 命令结束进程）

参数:

a: 显示所有网络连接

n: 显示 ip 地址

p: 显示程序的 pid

关机命令

关机: **halt**

重启: **reboot**

结束占用 8080 端口的进程(应用)

查找占用 8080 端口的进程的 pid

`netstat -anp | grep ":8080" --color` 在结果 cmd 之前，有 pid 的值

结束掉 pid 对应进程

`kill -9 pid`

24-3-6 权限管理命令

Linux 的权限

Linux 系统的安全性很高，有很大一部分原因是权限系统非常完善。每个文件都有自己的权限设置：谁可以操作，可以怎样操作。可以使用 `ll` 命令，查看文件的权限，其中前边 10 个字符就是文件的权限。如图：

```
drwxr-xr-x. 3 root root 4096 6月 10 02:24 heima
```

文件的权限说明

第 1 位：表示文件的类型。`d`: 表示是一个文件夹，`-`: 表示是一个文件，`l`: 表示是一个链接文件（类似于 Windows 的快捷方式）

第 2~4 位：文件拥有者，对当前文件的权限。其中：`r` read 可读，`w` write 可写，`x` execute 可执行，`-` 无权限

第 5~7 位：拥有者同组用户，对当前文件的权限。

第 8~10 位：其它用户，对当前文件的权限

权限管理命令(授权命名) `chmod`

基本语法：`chmod 权限 文件`

`chmod:change mode`

应用示例：

`chmod u=rwx, g=rwx, o=rx heima.txt:`

给 `heima.txt` 授权，拥有者可读写执行、同组用户可读写执行、其它用户可读执行

`chmod a=rwx heima.txt`

给 `heima.txt` 授权，所有用户都可读写执行

`chmod 754 heima.txt`

给 `heima.txt` 授权，拥有者可读写执行、同组用户可读执行、其它用户可读

权限的增量设置模式：`+`表示增加授权，`-`表示取消授权

`chmod u+r heima.txt`

给 `heima.txt` 授权，给拥有者增加可读权限

`chmod o-x heima.txt`

`chmod a+r heima.txt`

拥有者 (user)			同组用户 (group)			其他用户(other)		
r	w	x	r	w	x	r	w	x
4	2	1	4	2	1	4	2	1

24-4 远程连接工具

在实际运维中，Linux 通常是作为服务器的操作系统。而服务器一般是在机房里维护，需要通过远程连接工具，连接上机房里的服务器，进行操作。

我们在工具里输入 Linux 命令，命令会被工具发送到服务器上执行，服务器执行完成，再把结果返回给工具，显示到工具的界面上。工具的作用仅仅是：发送命令到 Linux，接收 Linux 的执行结果。

Linux 里远程连接工具很多，xshell，putty，secureCRT 等等

24-5_附：kill 命令的通讯信号编号

linux signals

Signal Name	Number	Description
SIGHUP	1	Hangup (POSIX)
SIGINT	2	Terminal interrupt (ANSI)
SIGQUIT	3	Terminal quit (POSIX)
SIGILL	4	Illegal instruction (ANSI)
SIGTRAP	5	Trace trap (POSIX)
SIGIOT	6	IOT Trap (4.2 BSD)
SIGBUS	7	BUS error (4.2 BSD)
SIGFPE	8	Floating point exception (ANSI)
SIGKILL	9	Kill(can't be caught or ignored) (POSIX)
SIGUSR1	10	User defined signal 1 (POSIX)
SIGSEGV	11	Invalid memory segment access (ANSI)
SIGUSR2	12	User defined signal 2 (POSIX)
SIGPIPE	13	Write on a pipe with no reader, Broken pipe (POSIX)
SIGALRM	14	Alarm clock (POSIX)
SIGTERM	15	Termination (ANSI)
SIGSTKFLT	16	Stack fault
SIGCHLD	17	Child process has stopped or exited, changed (POSIX)
SIGCONT	18	Continue executing, if stopped (POSIX)
SIGSTOP	19	Stop executing(can't be caught or ignored) (POSIX)
SIGTSTP	20	Terminal stop signal (POSIX)
SIGTTIN	21	Background process trying to read, from TTY (POSIX)
SIGTTOU	22	Background process trying to write, to TTY (POSIX)
SIGURG	23	Urgent condition on socket (4.2 BSD)
SIGXCPU	24	CPU limit exceeded (4.2 BSD)
SIGXFSZ	25	File size limit exceeded (4.2 BSD)
SIGVTALRM	26	Virtual alarm clock (4.2 BSD)
SIGPROF	27	Profiling alarm clock (4.2 BSD)
SIGWINCH	28	Window size change (4.3 BSD, Sun)
SIGIO	29	I/O now possible (4.2 BSD)
SIGPWR	30	Power failure restart (System V)

25-Linux&nginx

25-1_Linux 中软件安装相关的命令

25-1-1_rpm 命令

rpm: Redhat Package Manager, 红帽软件包管理套件。

应用示例：

安装：**rpm -ivh 安装包**

参数：

i: install, 安装
v: 显示执行过程
h: 安装时列出标记

查询: **rpm -qa**

参数:

q: 使用查询模式 query
a: 查询所有软件

卸载: **rpm -e --nodeps 软件名**

参数:

e: erase, 删除软件
nodeps: 忽略软件包之间的关联性

25-1-2_yum 命令

简介

yum: yellow dog updater、modifier, 是 Redhat 的软件包管理器。它基于 rpm 命令, 但是比 rpm 功能更强: 它能自动处理多个软件包之间的依赖关系, 能够从指定服务器上自动下载 RPM 包并安装。不需要人工一次次下载安装依赖软件。

注意: 使用 yum 命令, 必须要联网。

应用示例:

安装: **yum install 软件名**
卸载: **yum remove 软件名**

25-2_Nginx

25-2-1_Nginx 概述

1、简介

Nginx (engine x) 是一款轻量级的 Web 服务器/反向代理服务器及电子邮件(IMAP/POP3/SMTP)服务器。Nginx 是由伊戈尔·赛索耶夫为俄罗斯访问量第二的 Rambler.ru 站点(俄文: Рамблер)开发的, 第一个公开版本 0.1.0 发布于 2004 年 10 月 4 日。

2、作用

作为 web 服务器, 可以用于部署静态 web 应用

作为负载均衡服务器, 可以实现负载均衡---反向代理

作为邮件服务器, 可以实现收发邮件等功能

3、负载均衡

当我们的 web 应用需要承担海量并发请求时, 通常需要使用服务器集群, 即: n 多个服务器实例, 共同提供服务。但是要实现集群, 有一些问题需要解决:

- 假如我们有 10 个服务器实例组成的集群，但是不能让用户记 10 个访问地址，对外访问路径只能有一个
 - 用户访问量大的时候，需要把用户的请求分配到压力较轻的服务器实例，实现各服务器的压力是均衡的--负载均衡
- 以上两个问题，都可以通过 Nginx 来解决

4、反向代理

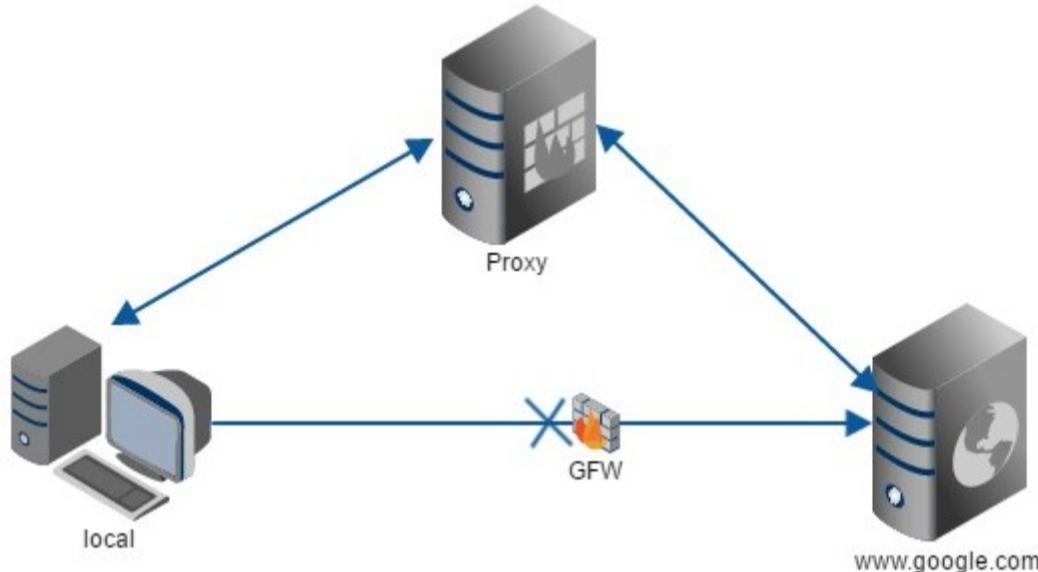
正向代理

正向代理，是**客户端代理**。

例如：我们访问不了 google，但是香港的电脑可以访问。我们可以把访问请求交给香港的某台服务器电脑，让这台服务器电脑帮我们转达至 google；然后得到 google 的响应内容，再把响应内容转交给我们。

这台服务器电脑，就是我们的代理，是客户端的代理，称为正向代理。

VPN

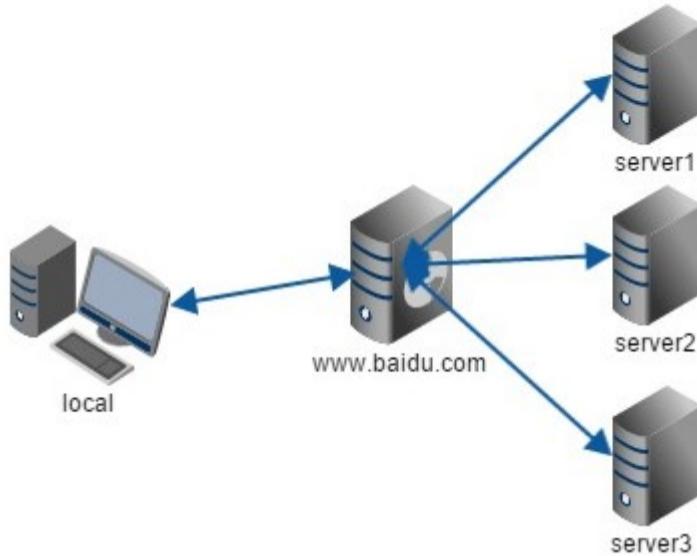


反向代理

反向代理，是**服务端代理**。

例如：我们有 10 台服务器集群，共同提供了一个 web 应用服务。为了实现负载均衡或者安全原因，为这 10 台服务器提供一个代理服务器。当有客户端访问 web 应用时，请求到代理服务器，由代理服务器把请求转发给某一个服务器处理，然后再由代理服务器把响应结果交给客户端。

这台代理服务器，是服务端的代理，称为反向代理。



25-2-2_下载与目录结构

下载地址: <http://nginx.org>

└── conf	存放配置文件，主要是 nginx.conf
└── contrib	存放一些 nginx 的工具
└── docs	存放一些说明文件及许可声明
└── html	存放默认的网页文件
└── logs	存放 Nginx 的运行日志
└── temp	存放 Nginx 运行中的临时文件
└── nginx.exe	Nginx 启动程序

25-2-3_Windows 版 Nginx 的基本操作

Windows 版的 Nginx 免安装，直接解压即可。

常用的 Nginx 操作有（在 dos 窗口中执行命令）：

启动 Nginx

重新加载 Nginx 配置

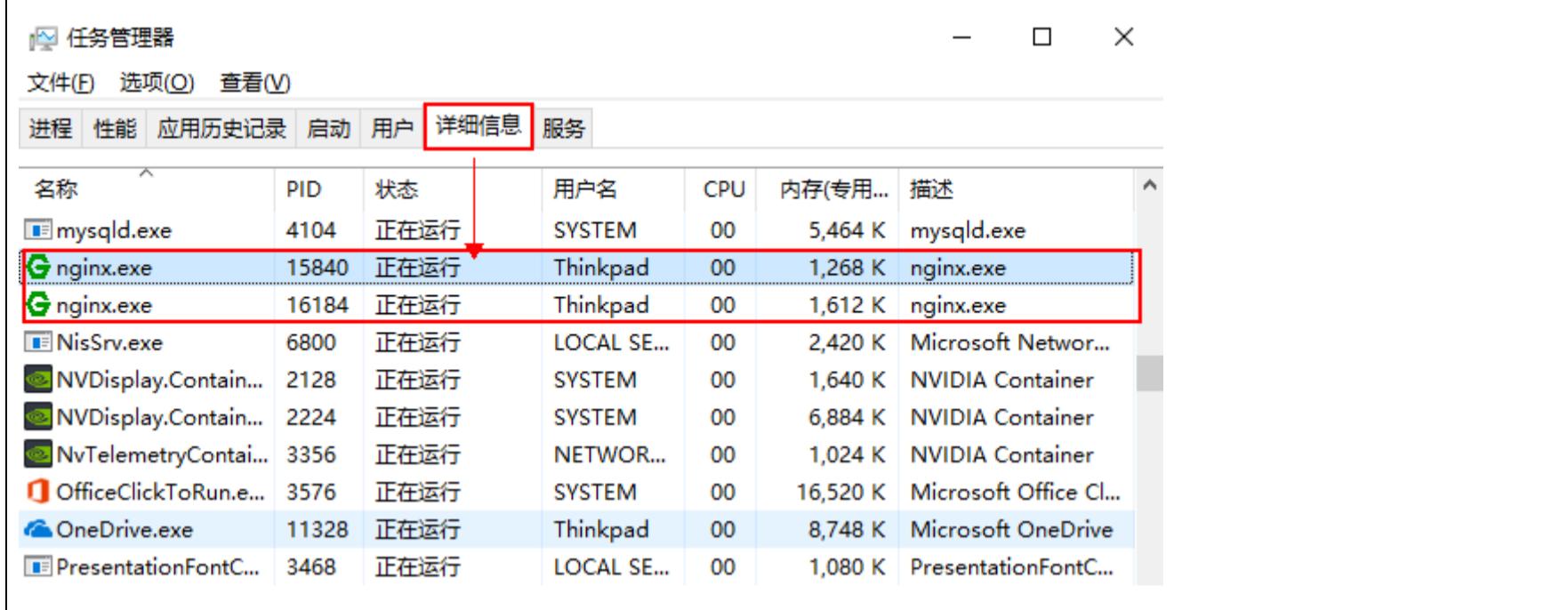
关闭 Nginx

1、启动 Nginx

命令： **start nginx.exe**

```
C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.15063]
(c) 2017 Microsoft Corporation。保留所有权利。
C:\programs\nginx-1.13.8>start nginx.exe
C:\programs\nginx-1.13.8>
```

如果启动成功，在任务管理器里可以看到 Nginx 的两个进程，如图：

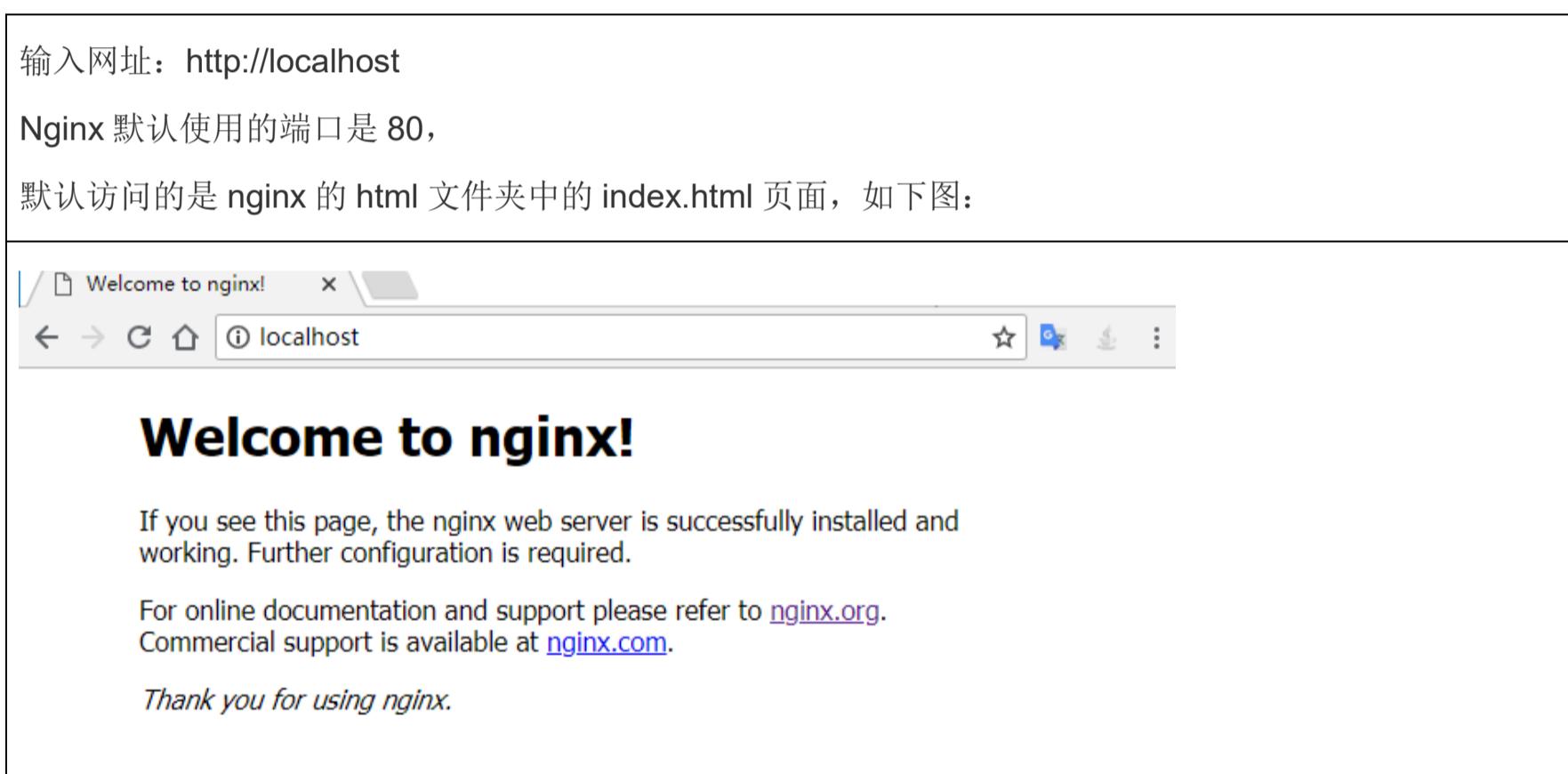


2、浏览器访问 Nginx

输入网址： <http://localhost>

Nginx 默认使用的端口是 80，

默认访问的是 nginx 的 html 文件夹中的 index.html 页面，如下图：



3、重新加载配置文件

命令： **nginx -s reload**

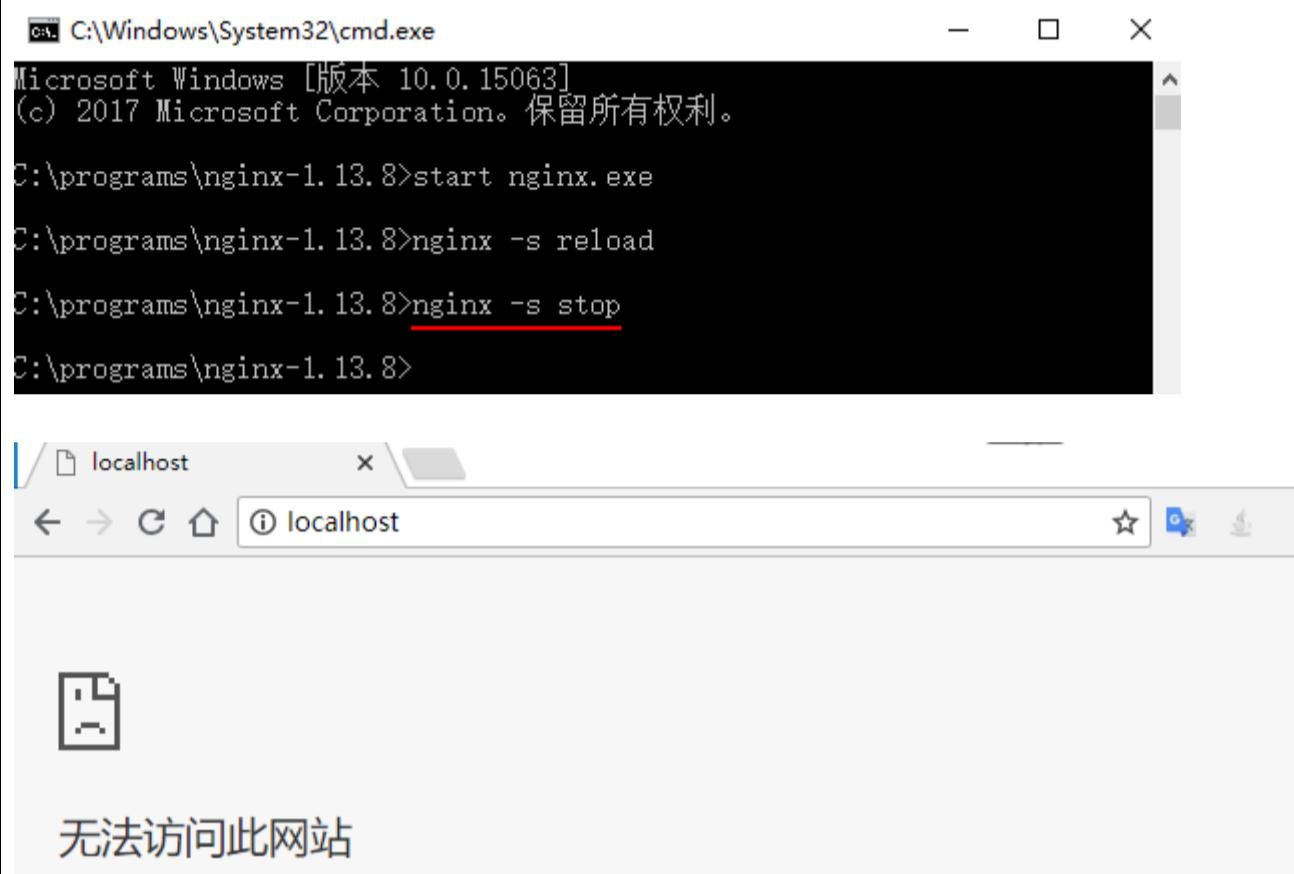
如果修改了配置文件 nginx.conf，就需要重新加载配置文件才会生效（重启 Nginx 也可以）



4、关闭 Nginx

命令: **nginx -s stop**

关闭后再访问 nginx, 就访问不到了



25-2-4 使用 Nginx 部署静态 web 应用

1、准备静态 web 应用

例如: 静态 web 应用是 E:\41\web00_bootstrap

注意:

应用里文件(文件夹)名称中不能包含中文、空格、特殊字符, 建议使用字母、数字和下划线
也不要把应用放在中文文件夹里

2、配置 nginx.conf

nginx 默认访问的是安装目录里 html 文件夹的内容, 如果要访问指定 web 应用, 需要修改 config/nginx.conf 配置文件, 如下:

前边有“#”的是注释内容, 不会生效

```
server {  
    listen      80;  
    server_name localhost;  
  
    #charset koi8-r;  
  
    #access_log  logs/host.access.log  main;  
  
    location / {  
        静态web应用的位置  
        root   E:\41\web00_bootstrap;  
        index  index.html;  
        配置默认首页  
    }  
}
```

3、启动 Nginx，使用浏览器访问 web 应用

访问地址: <http://localhost>

页面如下:



25-2-5 使用 Nginx 实现负载均衡(了解)

1、准备多个服务器实例

准备第一个服务器 Tomcat7，地址是:

<http://localhost:8070>

准备第二个服务器 Tomcat8，地址是:

<http://localhost:8080>

2、修改 nginx.conf 配置服务器池

在 Nginx 里增加服务器池，池子里有所有需要进行代理 负载均衡的服务器地址:

下面的代码，应该写在 http{}里边，server{}前边:

```
upstream localtomcat{  
    server localhost:8080 weight=1;  
    server localhost:8070 weight=1;  
}
```

服务器池子的名称: localtomcat

每一个服务器实例配置:

server: 固定写法，表示是一个服务器实例。后边跟空格

服务器 ip: 服务器软件所在电脑的 ip

服务器商品: 服务器软件的访问端口

weight: 服务器实例的权重，访问的几率

最后需要加上英文的分号;

```
upstream localtomcat{
    server localhost:8080 weight=1;
    server localhost:8070 weight=1;
}
```

3、修改 nginx.conf 配置访问请求分发地址

在 server{}里边，修改 location /{}时的内容：

proxy_pass: 配置服务器池子。http://服务器池子名称;

index: 默认访问的首页

服务器池

```
upstream localtomcat{
    server localhost:8080 weight=1;
    server localhost:8070 weight=1;
}

server {
    listen      80;
    server_name localhost;

    #charset koi8-r;

    #access_log  logs/host.access.log  main;

    location / {
        #root   html;
        #root   E:\41\01bootstrap;
        proxy_pass http://localtomcat;  proxy_pass http://服务器池子名称;
        index  index.html index.htm;
    }
}
```

格式：

```
upstream 池子名称{
    server 服务器1的ip:服务器1的端口;
    server 服务器2的ip:服务器2的端口;
    ...
}
```

有请求进来，把请求分发到服务器池中的某个服务器。

格式：

```
proxy_pass http://localtomcat; proxy_pass http://服务器池子名称;
```

25-2-6 Linux 版 Nginx 安装

参考安装文档《配置 JavaEE 运行环境(Linux).doc》

26-Redis

26-1 Redis

26-1-1 概述

1、NoSQL 数据库

简介

NoSQL: Not only SQL, 泛指所有非关系型数据库。

关系型数据库和非关系型数据库

关系型数据库：以表的形式来存取数据，数据之间的关系使用键来维护。例如： MySql、 Oracle 等等

非关系型数据库：泛指所有不以表形式存储数据的数据库。

为什么要有非关系型数据库

随着互联网的发展，越来越多的超大规模并发访问和海量数据，导致关系型数据库暴露了很多难以克服的问题：

* 高并发要求：瞬间的海量请求造成的并发问题

* 高性能要求：从海量数据里瞬间查询或者操作某一条数据的效率问题

* 高扩展要求：多个数据库节点，需要方便的进行扩展移植的问题

非关系型数据库的优点

易扩展：非关系型数据库共同的特点是 数据之间无关系，就非常容易进行扩展

大数据量，高性能：非关系型数据结构简单，可以更快速的对数据进行读写操作

灵活的数据模型：存储数据时不需要字段，可以随意的自定义自己的数据格式。比如：保存成 json 格式字符串

2、Redis

Redis 是一个开源免费、高性能的 **key-value** 数据库，它把数据保存在内存中，因此有着极高的读写性能（读 110000 次/s，写 81000 次/s）。

Redis 的端口：6379 merz

在 web 应用开发中，通常是把 Redis 作为缓存数据库，可以有效的提升 web 应用的访问效率：可以把一些访问特别频繁，但是又不经常变更的数据，放到 Redis 里；当客户端需要这些数据时，就可以直接从 Redis 里快速获取，而不需要再操作关系型数据库查询得到了。

26-1-2 安装 Redis

1、在 Linux 上安装 Redis

安装 c 编译环境 (gcc): **yum install gcc-c++**

上节课已经安装过，不再重新安装。安装方法参考《配置 JavaEE 运行环境(Linux).doc》第五章第一部分。

把 Redis 安装包上传到 Linus

把 Redis 安装包上传到 linux 的 /root/softwares 目录下

解压 Redis

把 Redis 解压到 /usr/local:

切换目录： **cd /usr/local/softwares**

解压 Redis 软件： **tar -xvf redis-3.0.0.tar.gz -C /usr/local**

```
[root@centos1 softwares]# tar -xvf redis-3.0.0.tar.gz -C /usr/local/
```

编译并安装 Redis

切换到解压目录： **cd /usr/local/redis-3.0.0**

```
[root@centos1 softwares]# cd /usr/local/redis-3.0.0/
[root@centos1 redis-3.0.0]#
```

编译 redis，命令： **make**

```
[root@centos1 redis-3.0.0]# make
```

安装 redis，命令：make PREFIX=/usr/local/redis install

把 redis 安装到了/usr/local/redis 文件夹中

```
[root@centos1 redis-3.0.0]# make PREFIX=/usr/local/redis install  
cd src && make install  
make[1]: Entering directory `/usr/local/redis-3.0.0/src'  
  
Hint: It's a good idea to run 'make test' ;)  
  
INSTALL install  
INSTALL install  
INSTALL install  
INSTALL install  
INSTALL install  
make[1]: Leaving directory `/usr/local/redis-3.0.0/src'  
[root@centos1 redis-3.0.0]#
```

2、配置 Redis

拷贝 Redis 的配置文件到安装目录

解压目录中有一个配置文件 redis.conf，拷贝到 Redis 的安装目录的 bin 文件夹下：

```
cp redis.conf /usr/local/redis/bin/
```

修改配置文件，让 Redis 可以在后台运行

切换到安装目录的 bin 文件夹下

使用 vim 修改 redis.conf 配置文件，把 daemonize 的值设置为 yes，保存并退出

```
[root@centos1 redis-3.0.0]# cd /usr/local/redis/bin/  
[root@centos1 bin]# vim redis.conf
```

```
##### GENERAL #####  
  
# By default Redis does not run as a daemon. Use 'yes' if you need it.  
# Note that Redis will write a pid file in /var/run/redis.pid when daemonized.  
daemonize yes 把值修改成yes  
  
# When running daemonized, Redis writes a pid file in /var/run/redis.pid by  
# default. You can specify a custom pid file location here.  
pidfile /var/run/redis.pid  
  
# Accept connections on the specified port, default is 6379.  
# If port 0 is specified Redis will not listen on a TCP socket.  
port 6379  
  
# TCP listen() backlog.  
#  
# In high requests-per-second environments you need an high backlog in order  
# to avoid slow clients connections issues. Note that the Linux kernel  
# will silently truncate it to the value of /proc/sys/net/core/somaxconn so  
# make sure to raise both the value of somaxconn and tcp_max_syn_backlog  
:wq
```

配置 Linux 的防火墙，开放 6379 端口

命令：/sbin/iptables -I INPUT -p tcp --dport 6379 -j ACCEPT

命令：/etc/rc.d/init.d/iptables save

```
[root@centos1 bin]# /sbin/iptables -I INPUT -p tcp --dport 6379 -j ACCEPT  
[root@centos1 bin]# /etc/rc.d/init.d/iptables save [确定]  
iptables: 将防火墙规则保存到 /etc/sysconfig/iptables:  
[root@centos1 bin]#
```

切换到 Redis 安装目录 bin 下，启动 Redis

切换目录: cd /usr/local/redis/bin

启动 Redis 服务: ./redis-server redis.conf

```
[root@centos1 bin]# ./redis-server redis.conf
[root@centos1 bin]#
```

使用客户端连接数据库

命令: ./redis-cli [-h Redis 的 ip 地址 -p Redis 的端口]

```
[root@centos1 bin]# ./redis-cli
127.0.0.1:6379> 看到这个，表示已经连接上了本机6379端口的Redis
```

退出客户端

在 Redis 界面, 输入命令: exit

```
127.0.0.1:6379> exit
[root@centos1 bin]#
```

关闭 Redis 服务

输入 Linux 命令: ./redis-cli shutdown

```
[root@centos1 bin]# ./redis-cli shutdown 关闭Redis服务之后，再使用客户端登录，提示连接被拒绝，不能登录了
[root@centos1 bin]# ./redis-cli
Could not connect to Redis at 127.0.0.1:6379: Connection refused
not connected>
```

26-2 Redis 的数据操作

26-2-1 Redis 的数据类型介绍

Redis 以 key-value 形式存储数据, 存储在内存中:

它的 **key** 始终是字符串: 但是最好不要超过 1024 字节, 否则会影响性能。一般企业会对 Redis 的 **key** 命名制定规范

它的 **value** 有五种数据类型, 分别是:

string 类型: 字符串类型

hash 类型: 哈希类型, 类似于 **map** 的键值对形式

list 类型: 列表类型, 类似于队列

set 类型: 无序集合, 类似于 Java 的 **Set**

zset 类型 (**sortedset**): 有序集合

它的key始终是字符串；但是最好不要超过1024字节，否则会影响性能。

它的value有五种数据类型，分别是：

string类型：字符串类型
hash类型：哈希类型，类似于map的键值对形式
list类型：列表类型，类似于队列
set类型：无序集合，类似于Java的Set
zset类型（sortedset）：有序集合

1. string类型的数据

key: string value: 字符串类型

name	张三
age	20
sex	male

2. list类型的数据

key: string value: 列表类型list



通常用于一些消息队列，生产者消费者模型

3. hash类型的数据

key: string value: hash类型，类似于一个Map

field value	
name	tom
age	20
email	tom@163.com

通常用于存储有关联关系的数据

4. set类型的数据

key: string value: set 无序不重复的集合

friends	tom lilei hanmeimei polly

通常用于集合的运算：交集、并集、差集
查询两个人共同的好友

5. zset类型的数据

key: string value: zset (sortedset) 有序的集合，内容不重复

lolrank	100 tom 95 jerry 40 lilei 20 hanmeimei 150 polly

通常用于排行榜，可以实时的进行数据排序
比如：游戏的排行榜

26-2-2 Redis 中不同类型数据类型的操作

1、操作 string 类型的数据(重点)

注意：一个 string 类型最多可以存储 512M 数据

设置数据： **set key value**

获取数据： **get key**

删除数据： **del key**

查看所有 key: **keys ***

2、操作 hash 类型的数据

注意：一个 hash 是一个键值对集合，每个 hash 可以存储 $2^{32} - 1$ 个键值对（40 多亿个）

设置一个 hash:

hset key field value

获取一个 hash 里某个 field:

hget key field

删除一个 hash 里某个 field:

hdel key field

查看一个 hash 里所有数据:

hgetall key

3、操作 list 类型的数据

redis 的 list 类型，数据存储结构类似于 Java 的 LinkedList，元素的增、删非常快。一个 list 可以存储 $2^{32} - 1$ 个元素

添加数据 **lpush/rpush**

左边添加: **lpush key value1 value2**

右边添加: **rpush key value1 value2**

查看全部数据 **lrange key 0 -1**

获取数据 lpop/rpop

lpop key: 弹出（获取）左边第一个

rpop key: 弹出（获取）右边第一个

4、操作 set 类型的数据

Redis 时的集合 set，无序不重复的，在操作上更类似于数据的集合。一个 set 可以存储 2^32-1 个数据

添加成员： sadd key member1 member2

获取所有成员： smembers key

随机获取一个成员： srandmember key

删除成员： srem key member1 member2

计算交集： sinter key1 key2

你有我也有

计算并集： sunion key1 key2

你有或者我有

计算差集： sdiff key1 key2

你没有我有，或者你有我没有

26-2-3_通用 key 操作

查询 key

keys pattern pattern 中可以出现*和？， *表示任意个任意字符， ?表示一个任意字符

keys my*

删除 key

del key

判断 key 是否存在

exists key

查看 key 的类型

type key

26-3_Jedis(重点)

26-3-1_什么是 Jedis

Jedis=Java Redis

Servlet=Server Applet

在实际开发中，需要使用程序代码来操作 Redis。Redis 数据库官方提供了 Java 程序操作 Redis 的方式：Jedis。

Java 要操作 Redis，需要引入相关的 jar 包如下：

 commons-pool2-2.3.jar
 jedis-2.7.0.jar

26-3-2_Jedis 的应用

1、Jedis 相关的 API

Redis 的连接对象：Jedis

构造方法：Jedis(String ip, int port)

Jedis 常用的操作方法：jedis 对象操作 Redis 数据库的方法名，和命令名称完全一样。

设置字符串：

命令：set key value

方法：jedis.set(String key, String value);

获取字符串：

命令：get key

方法：jedis.get(String key);

删除字符串：

命令：del key

方法：jedis.del(String key);

向 list 里左边添加一个数据：

命令：lpush key value

方法：jedis.lpush(String key, String value);

释放资源：jedis.close();

2、Jedis 的基本使用

```
//获取连接
Jedis jedis = new Jedis(String ip, int port);
//操作 redis 数据库，用什么方法操作：想执行什么命令，就使用什么方式。方法名称和命令的名称是一样的
jedis.set("name","tom");
String name = jedis.get("name");
//释放资源
jedis.close();
```

3、Jedis 连接池

连接池类: `JedisPool`

使用默认配置的连接池-构造方法:

```
JedisPool(String host, int port);
```

使用自定义配置的连接池-构造方法

```
JedisPool(JedisPoolConfig config, String host, int port);
```

`jedisPoolConfig`: 连接池的配置信息对象

无参构造: `JedisPoolConfig()`

设置参数:

```
setMaxTotal(int maxTotal)
```

```
setMaxIdle(int maxIdle)
```

`host`: Redis 的地址

`port`: Redis 的端口

从连接池中获取连接-方法:

```
Jedis jedis = pool.getResource();
```

操作步骤:

```
//创建连接池对象: JedisPool
```

```
//从连接池里获取连接:Jedis
```

```
//操作 redis 数据库
```

```
//释放资源
```

26-3-3 使用 `JedisUtils` 工具类操作 Redis

26-4 Redis 持久化(了解)

Redis 的数据是保存在内存中的，但是也提供了持久化机制，可以把内存中的数据持久化保存到磁盘文件里。

Redis 提供了两种持久化机制:

RDB 模式--快照模式

AOF 模式

其中 RDB 是默认开启的，而 AOF 需要手动配置开启。

26-4-1 RDB 模式-快照模式

Redis 会定时把内存中的数据备份，生成“快照文件”。文件名称是 `dump.rdb`，默认被保存在了 Redis 的安装目录里。

手动配置 RDB: 可以通过修改 `redis.conf` 来修改 RDB 的持久化配置

```
save 900 1: 1 次更改, 900 秒存一次
```

```
save 300 10: 10 次更改, 300 秒存一次  
save 60 10000: 10000 次更改, 60 秒存一次
```

RDB 模式的特点

优点:

性能略高一些

缺点:

可能会丢失数据, 不够安全

26-4-2_AOF 模式-append only file

每次对 Redis 进行变更操作时, Redis 都会把执行的命令保存到文件中。当 Redis 重启时, 会自动读取这个文件, 重新执行文件里的所有命令, 重构数据库。

AOF 需要手动开启, 可以通过 `redis.conf` 更改配置:

```
appendonly yes          #默认是 no, 未开启 AOF。设置为 yes 表示开启 AOF 模式。  
appendfilename "appendonly.aof" #默认生成的 AOF 文件名称
```

AOF 持久化有三种配置方式:

```
appendfsync always      #总是保存。执行的每一次数据变更命令, 都会被保存到文件中  
appendfsync everysec    #每秒保存一次, 默认  
appendfsync no          #不保存
```

优点:

数据比较安全, 丢失数据的可能性比较低

缺点:

性能低

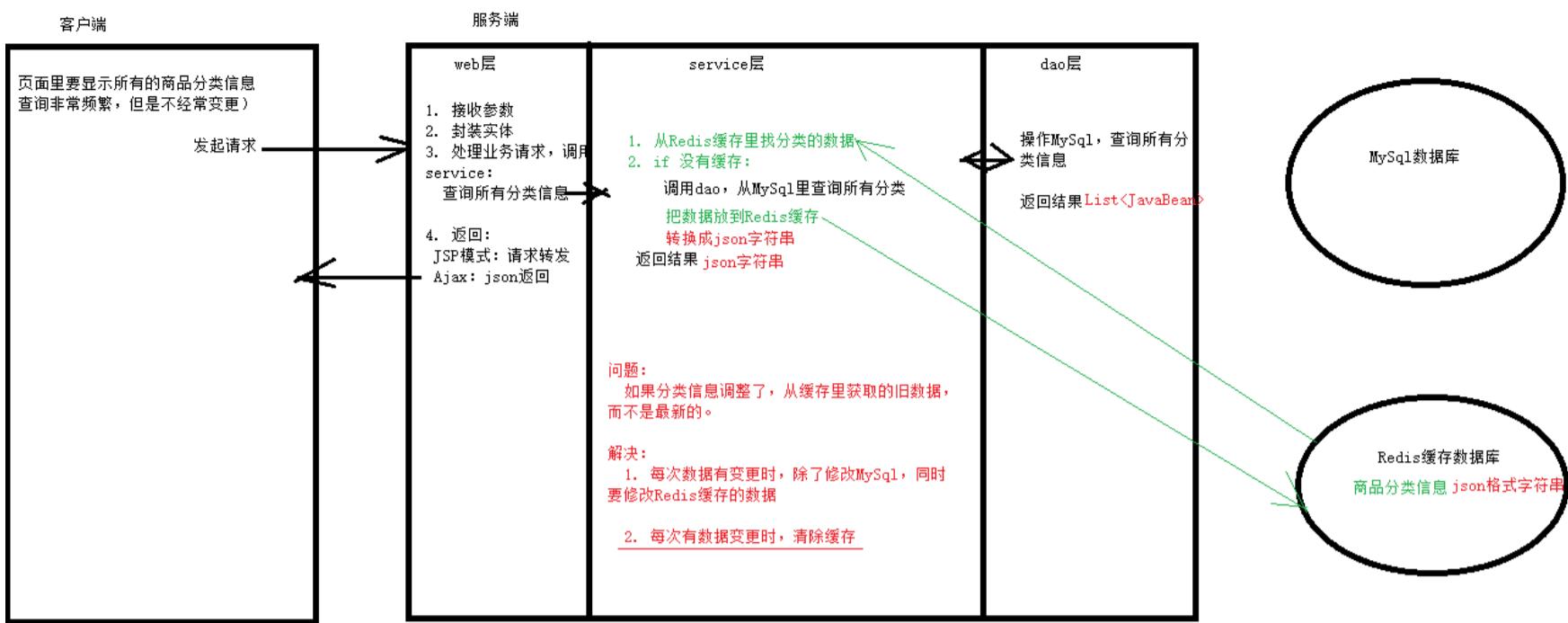
可能保存很多无用的操作命令

26-4-3_RDB 和 AOF

```
RDB 模式, 性能更高, 但是有可能会丢失数据。默认开启的  
AOF 模式, 数据不容易丢失, 但是性能略低
```

26-5 Redis 在 web 里的应用(重点)

在 web 应用里, 使用关系型数据库可以结构化保存数据与数据之间的关系, 使用 Redis 提高数据访问性能。通常是把读取频繁, 但不常变更的数据缓存起来, 放到 Redis 中, 读取效率更高。但是要注意: 如果数据有变更的话, 必须要更新缓存。否则会导致缓存和数据不一致而出现问题。



27-Maven

27-1_概述

27-1-1_简介

1、什么是 Maven

Maven，一个比较正式的定义，是 Apache 提供的一个项目管理工具，它包含了一个项目对象模型(**POM: Project Object Model**)、一组标准集合、一个项目生命周期、一个依赖管理系统、和用来运行在生命周期阶段中插件(plugin)目标(goal)的逻辑。

用更通俗的方式来说明，在程序开发过程中，除了写代码，还有很多其它必不可少的事情要做，比如：

- 引入 jar 包。尤其是大型工程，往往需要引入几十上百个 jar 包。每个 jar 包都需要手动引入，并且要注意 jar 包之间的版本冲突问题
- 程序编译问题。写好的程序是 Java 文件，必须要编译成 class 文件才能运行。现在有各种集成工具例如 idea 可以帮我们完成
- 单元测试。世界上不存在没有 bug 的程序，因此写完了程序有时候要写一些单元测试，然后一个个来运行检查代码的质量
- 开发好的程序代码，必须要打包：把所有的程序代码、资源、配置文件等等整合到一起，还要发布到 web 应用服务器软件进行部署。war

这些工作很麻烦，使用 Maven 可以帮我们从这些繁琐的工作中解脱出来：帮我们构建项目、管理 jar 包、编译代码，还能帮我们运行单元测试、打包，甚至能帮我们部署项目。

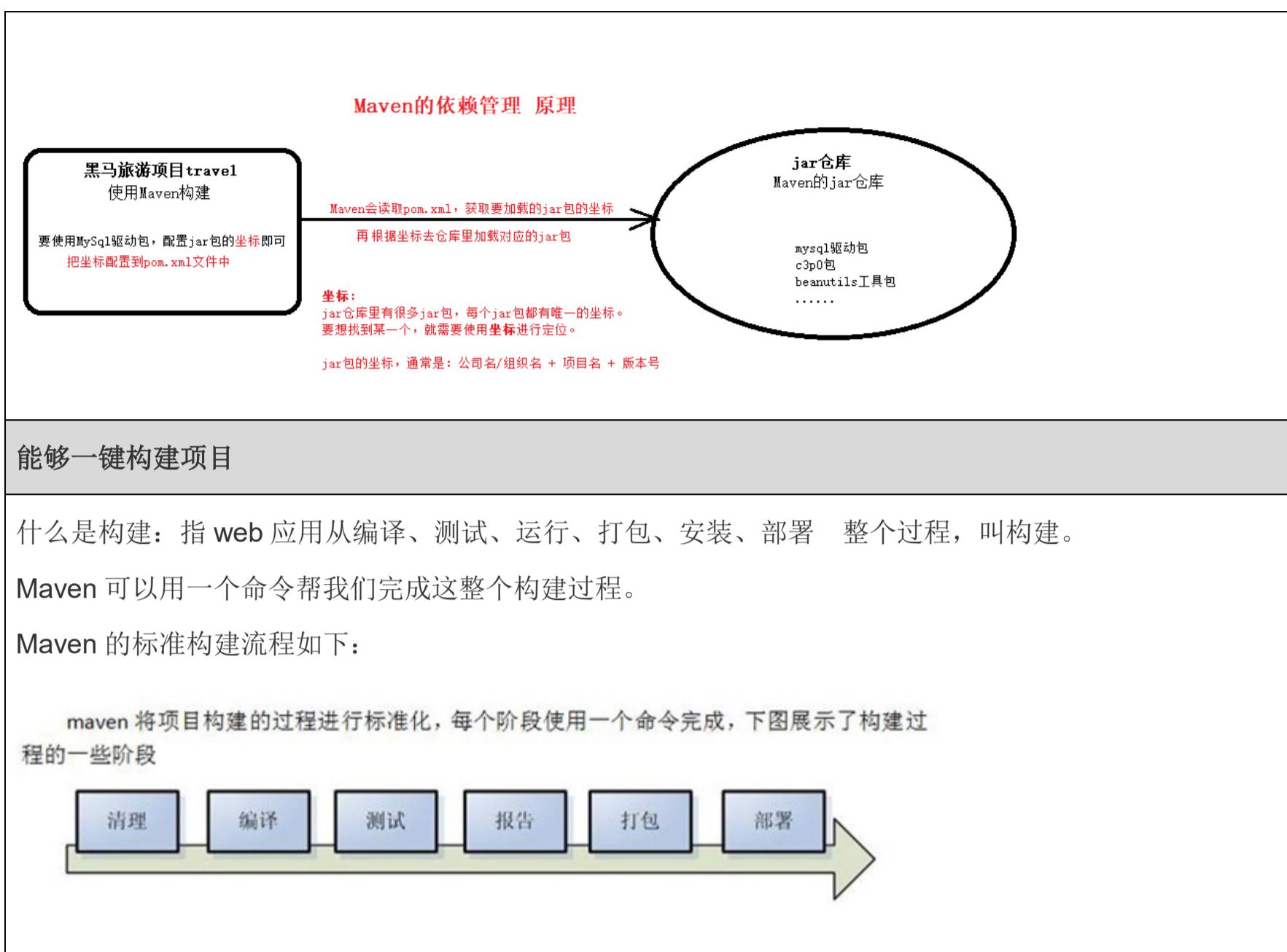
2、Maven 有什么作用

依赖管理：可以统一管理 jar 包

使用 Maven 之前，要使用某个 jar 包，我们需要去下载、拷贝到应用里、添加依赖。

使用 Maven 之后，只需要给 Maven 配置 jar 包的坐标到 pom.xml，Maven 会自动帮我们去加载 jar 包

Maven 的依赖管理原理：



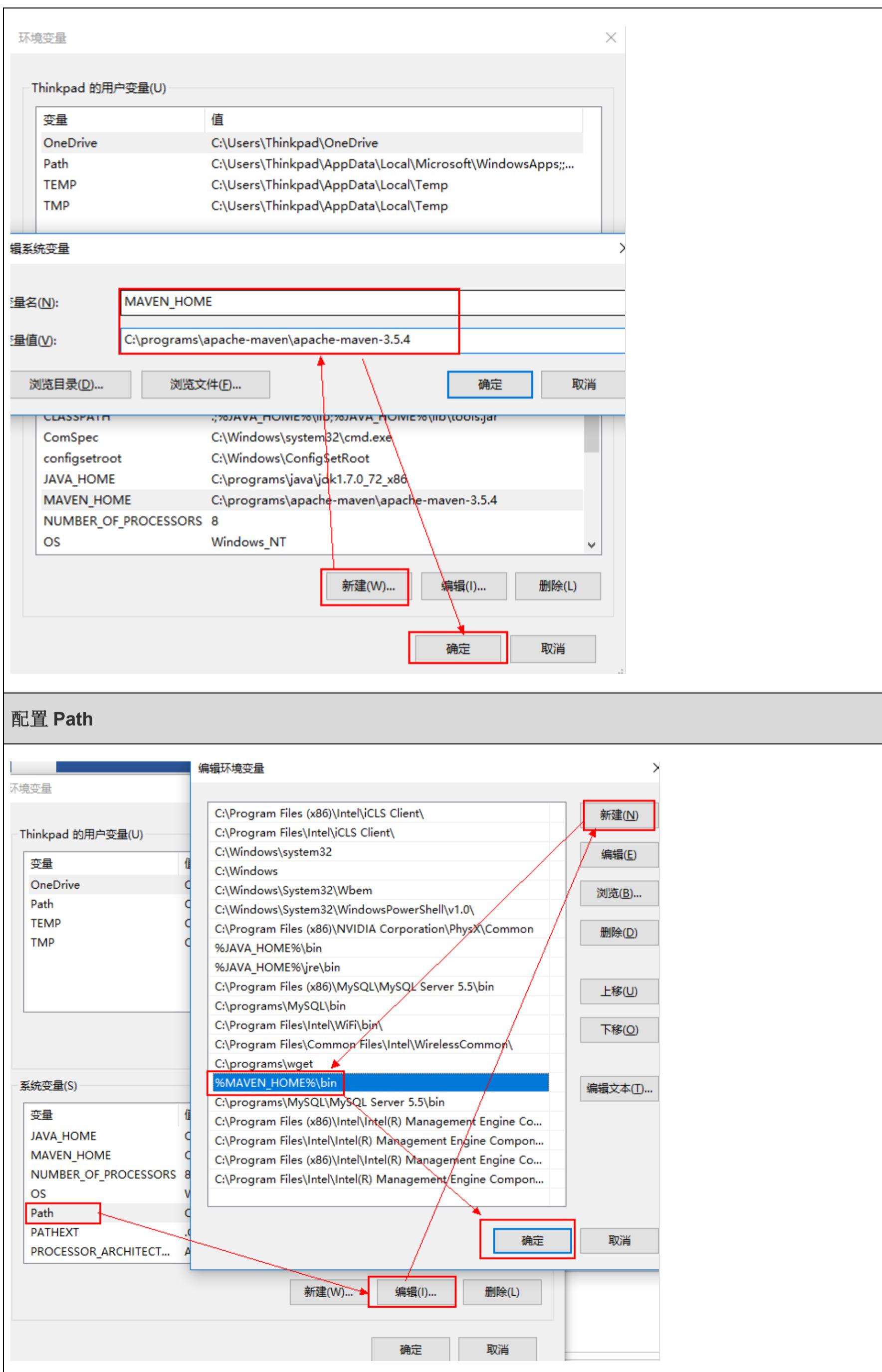
27-1-2_Maven 的安装与本地仓库配置

1、下载与目录结构

下载
http://maven.apache.org/
目录结构
<ul style="list-style-type: none"> bin: 可执行命令文件夹 boot: 类加载器（Maven 是 Java 程序，要使用类加载器） conf: 配置文件 lib: Maven 的核心程序

2、Maven 安装配置

<p>Maven 是免安装版的，直接解压即可。注意：不要放在有空格和中文的路径里 解压后但为了使用方便，我们要把 Maven 配置到环境变量中。</p>
配置 MAVEN_HOME



验证 Maven 配置

```
C:\Windows\system32\cmd.exe
Microsoft Windows [版本 10.0.15063]
(c) 2017 Microsoft Corporation。保留所有权利。

C:\Users\Thinkpad>mvn -v
Apache Maven 3.5.4 (1eedded0933998edf8bf061f1ceb3cfdec443fe; 2018-06-18T02:33:14+08:00)
Maven home: C:\programs\apache-maven\apache-maven-3.5.4\bin\
Java version: 1.8.0_181, vendor: Oracle Corporation, runtime: C:\programs\java\jdk1.8.0_181_x86\jre
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 10", version: "10.0", arch: "x86", family: "windows"
```

3、Maven 的仓库

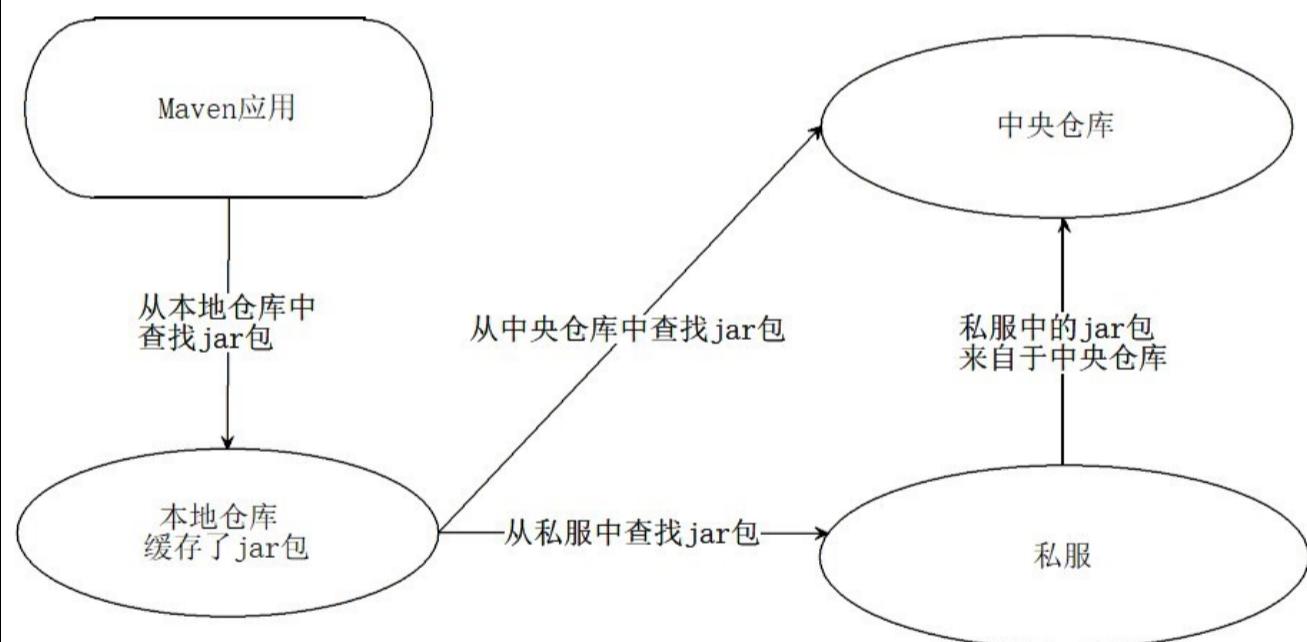
仓库分类

Maven 的仓库有三种：本地仓库，远程仓库（私服），中央仓库。每种仓库里都是各种 jar 包

本地仓库：用来存储从远程仓库/中央仓库下载的 jar 包。Maven 项目要使用 jar 包的话，优先从本地仓库中查找。本地仓库的默认地址是：\${user.dir}/.m2/repository 其中\${user.dir}是 Windows 当前用户的目录

远程仓库：如果 Maven 项目使用的 jar 包 在本地仓库中找不到，就会去远程仓库中查找并加载。远程仓库通常是由某些公司或者组织搭建提供的，例如 alibaba 提供了 Maven 仓库，一些企业也会在自己的局域网内搭建 Maven 仓库。这些仓库也称为私服。

中央仓库：是 Maven 软件内置的一个远程仓库，地址：<http://repo1.maven.org/maven2>。它是中央仓库，服务于整个互联网，由 Maven 自己的团队维护，里边存储了非常全的 jar 包，包含了世界上绝大部分流行的开源项目构件。

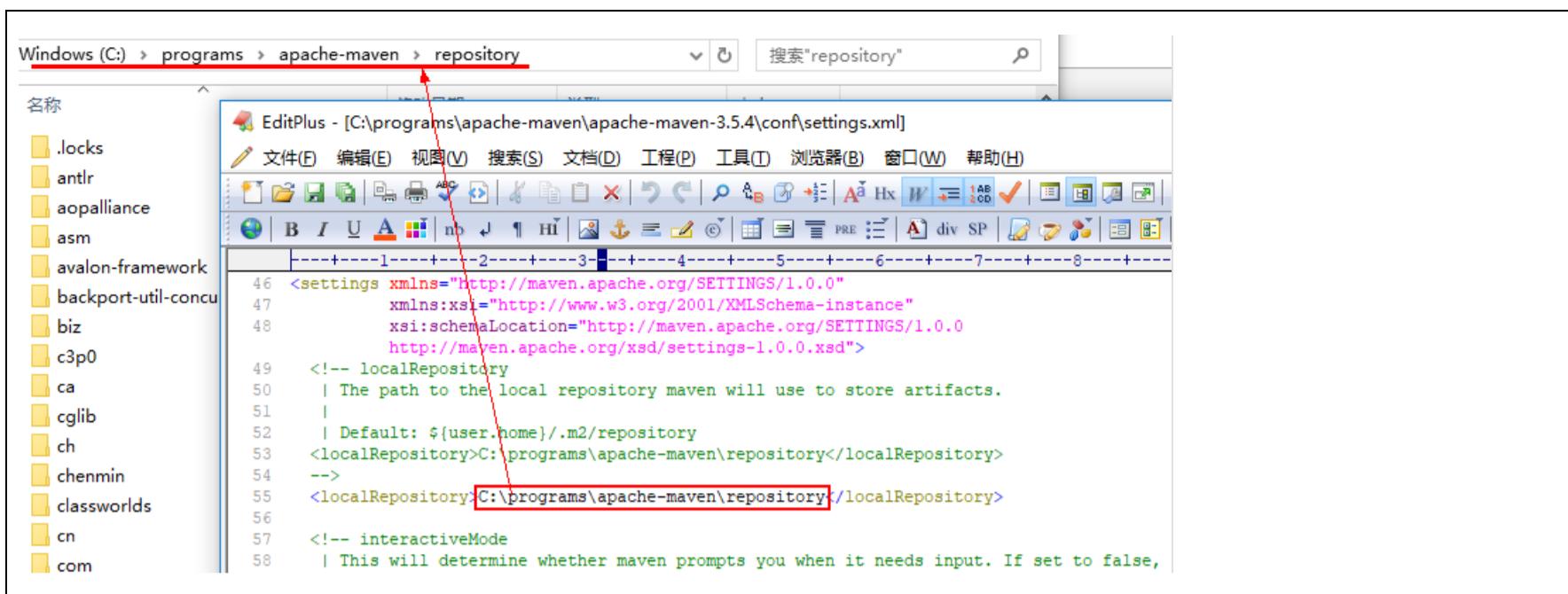


配置本地仓库

把本地仓库压缩包解压，例如，我解压到了 C:\programs\apache-maven 目录下。

注意：放在没有中文和空格的路径下。

打开 maven\conf\setting.xml，配置本地仓库的路径，如下：



27-2_Maven 的标准目录结构

使用 Maven 创建项目时，必须要遵循 Maven 的标准目录结构。

项目名称

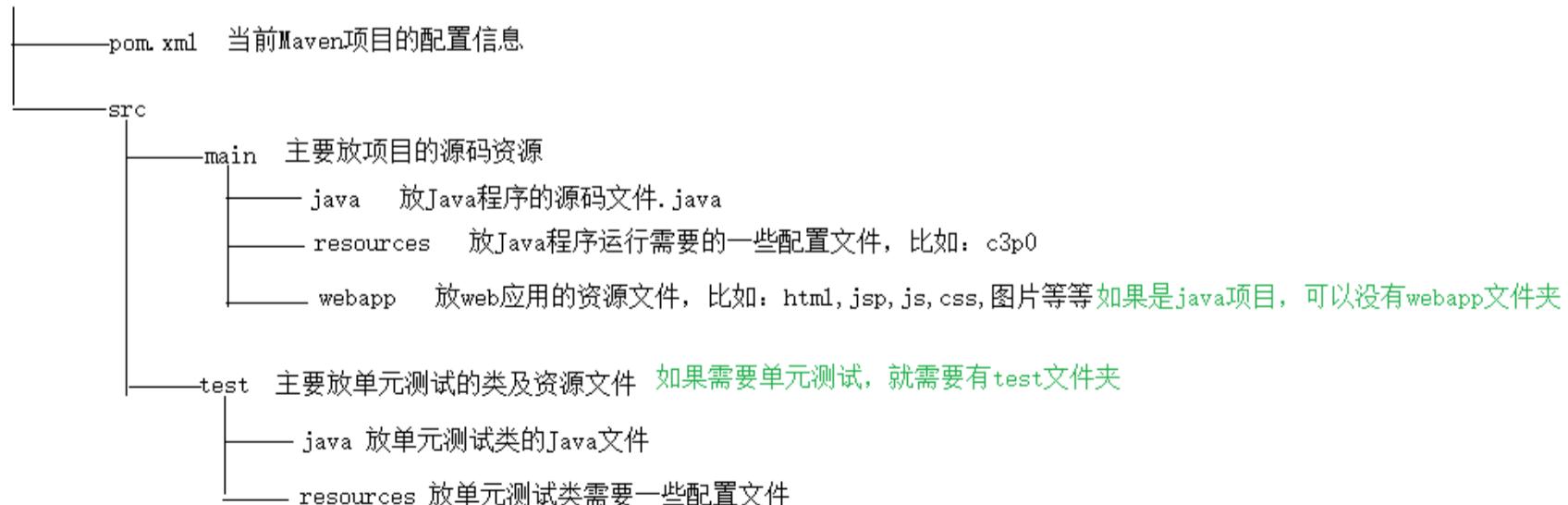
```

|-src
  |-main
    |-java
    |-resources
    |-webapps
    |-WEB-INF
    |-其它 web 资源
  |-test
    |-java
    |-resources
|-pom.xml

```

Maven项目的目录结构

项目名称 (java项目 / web项目)



27-3_Maven 的常用命令(goal)

我们可以在 cmd 里，使用 Maven 的一系列命令，来对 Maven 工程进行编译、测试、运行、打包、部署。

Maven 命令的统一格式： mvn 命令

27-3-1_clean-清理

清除项目根目录下的 target (之前编译的内容)

27-3-2_compile-编译

将项目中 src/main/java 里的.java 编译成.class

27-3-3_单元测试

编译并执行根目录中 src/test/java 中的测试类。

要求：单元测试类名必须是 XXXTest.java，否则不能执行

27-3-4_package-打包

将项目打包，然后放在根目录的 target 目录下

Java 项目，打包成 jar 包

web 项目，打包成 war 包

27-3-5_install-安装

把项目打包安装到本地仓库

Java 项目-->jar 包

web 项目-->war 包

27-3-6_Maven 命令的生命周期(了解)

Maven 有三套生命周期，三套生命周期之间互相独立的：

cleanLifeCycle：清理生命周期

clean

defaultLifeCycle：默认生命周期（构建生命周期）

compile, test, package, install, deploy

siteLifeCycle：站点生命周期

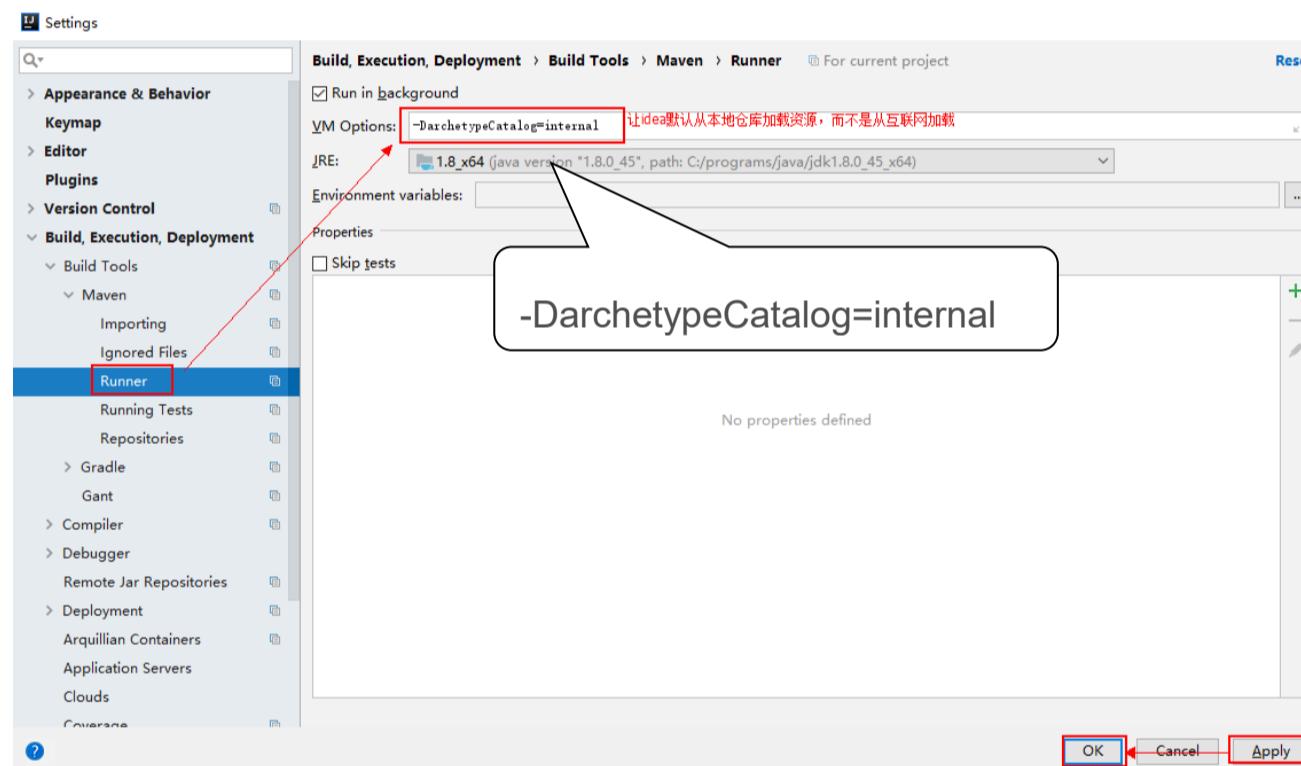
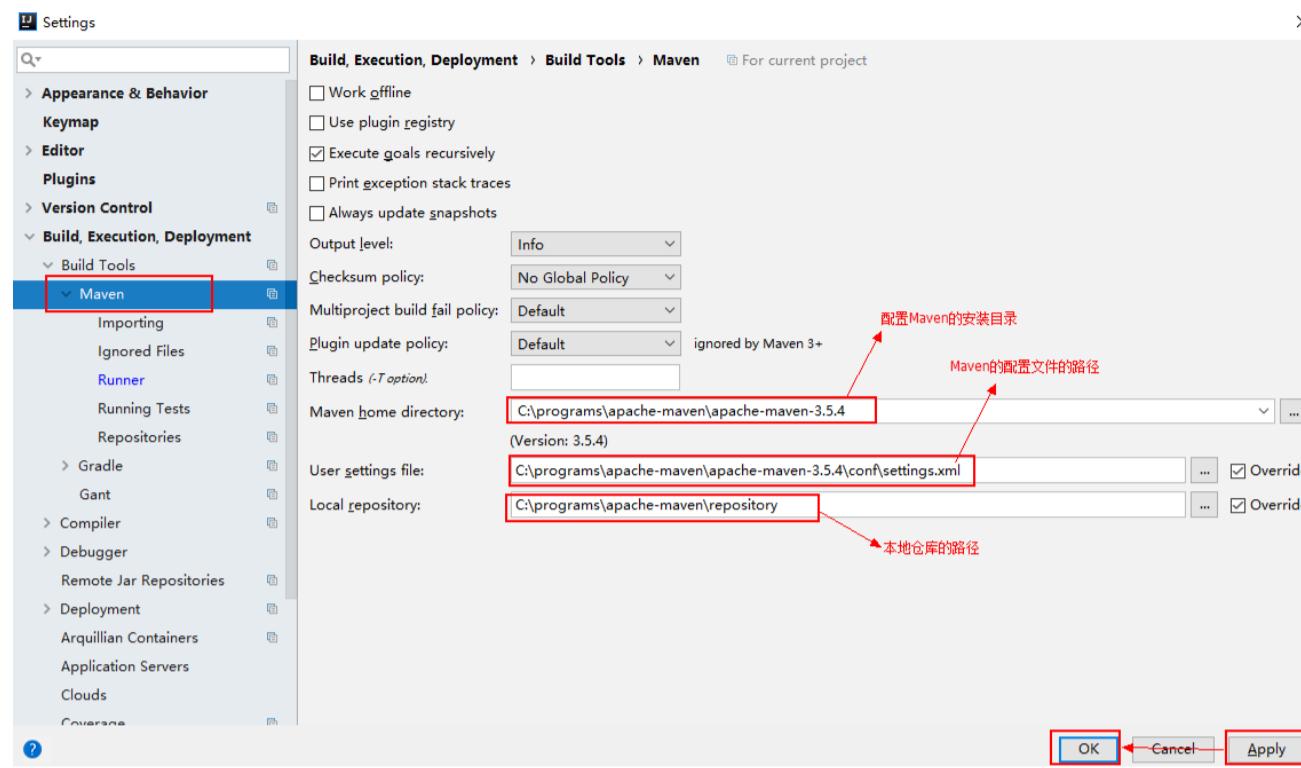
site

生成一些 HTML 文档，描述当前项目信息

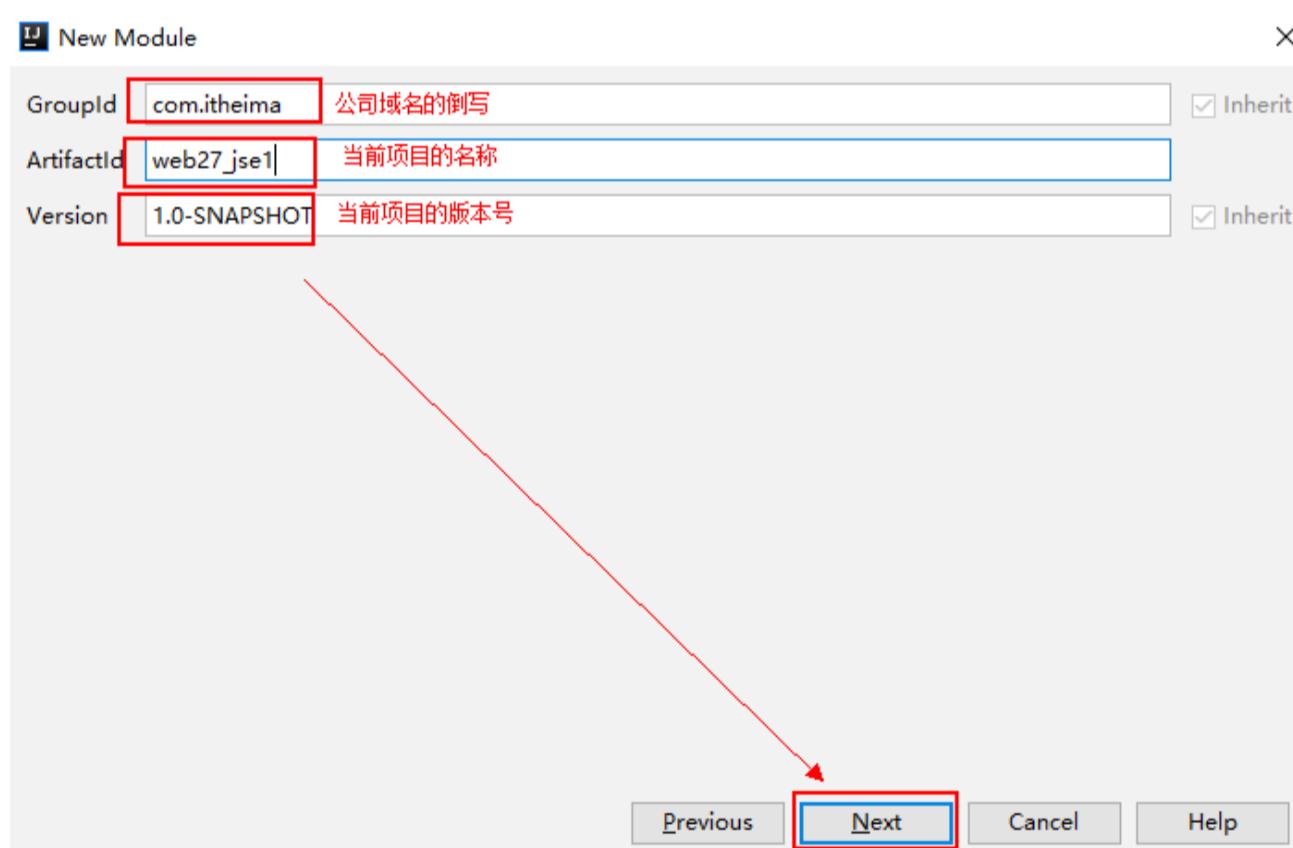
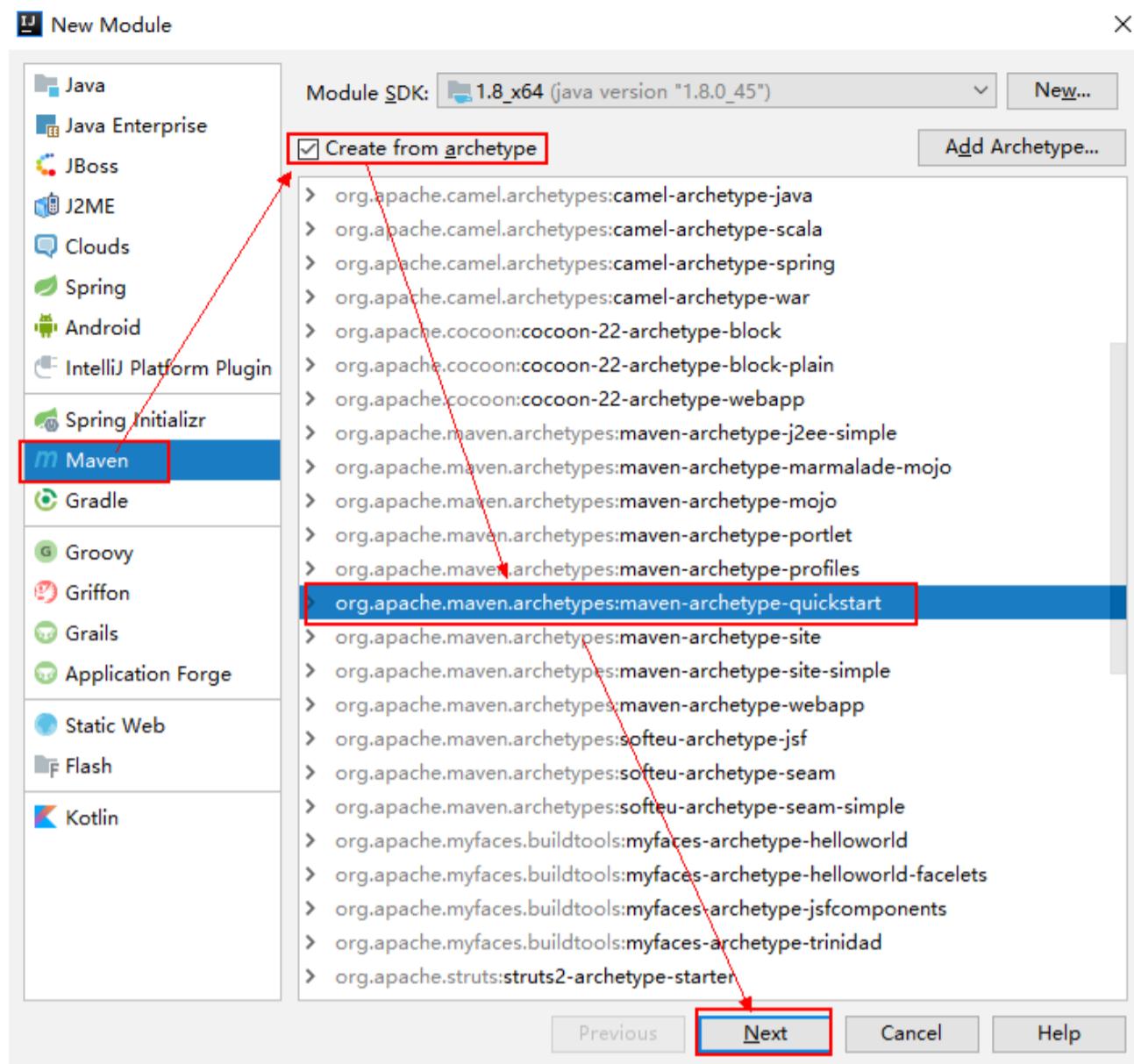
注意：在一套生命周期内，执行后边命令，前边命令会自动执行

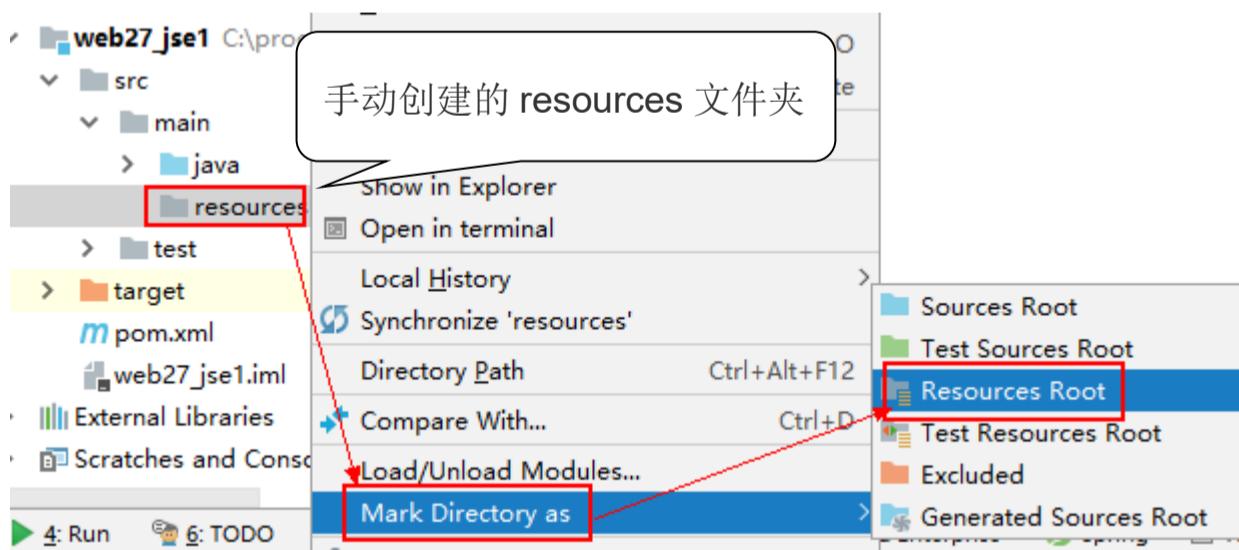
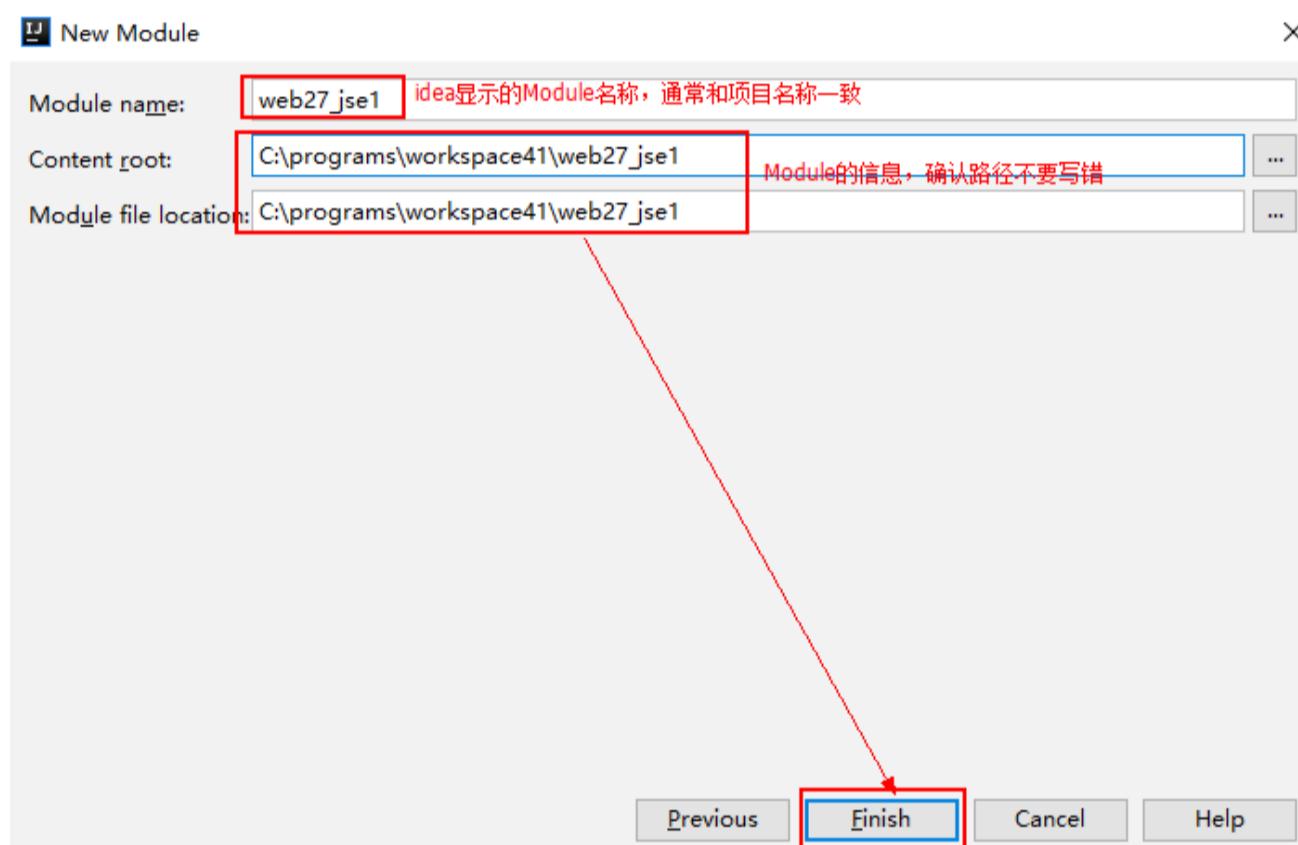
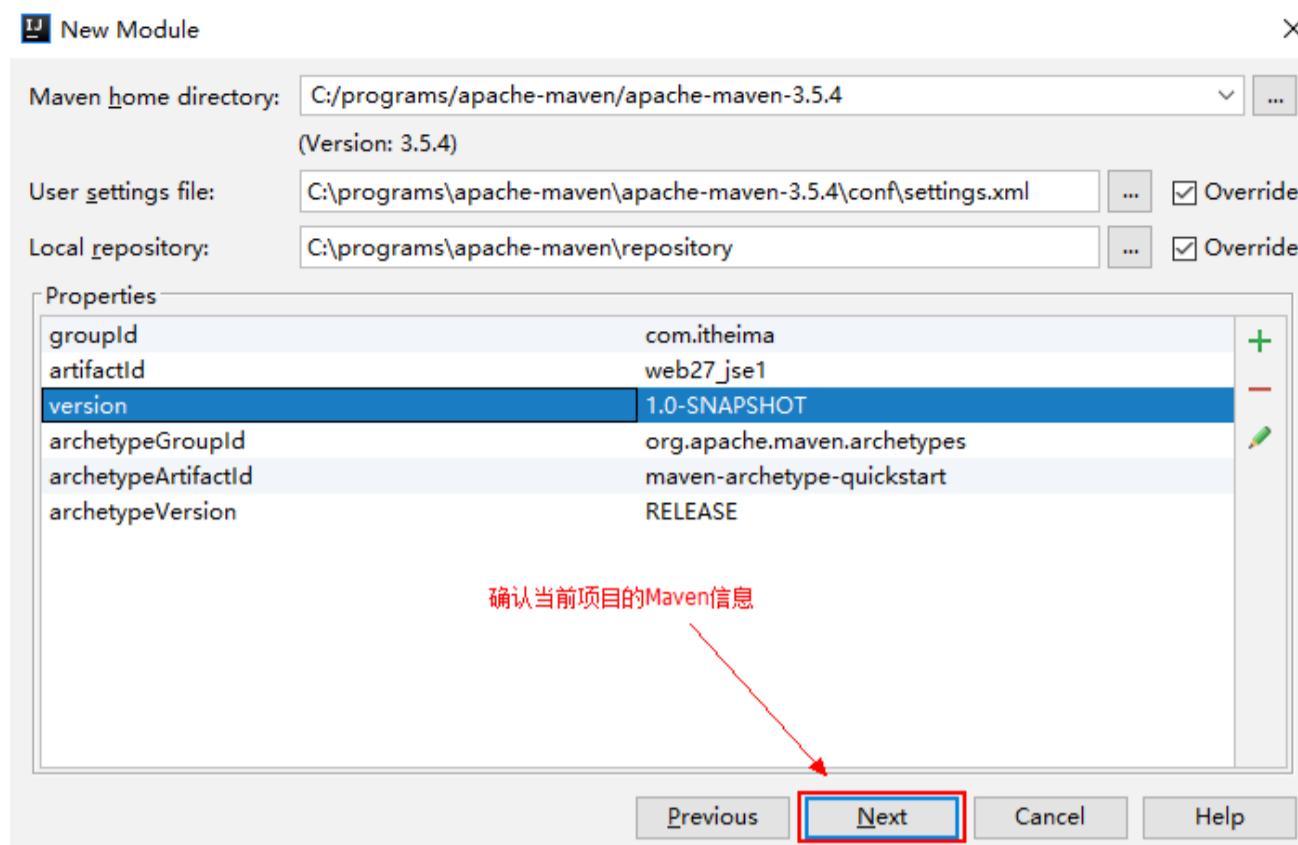
27-4 使用 idea 开发 Maven 项目

27-4-1 idea 配置 Maven

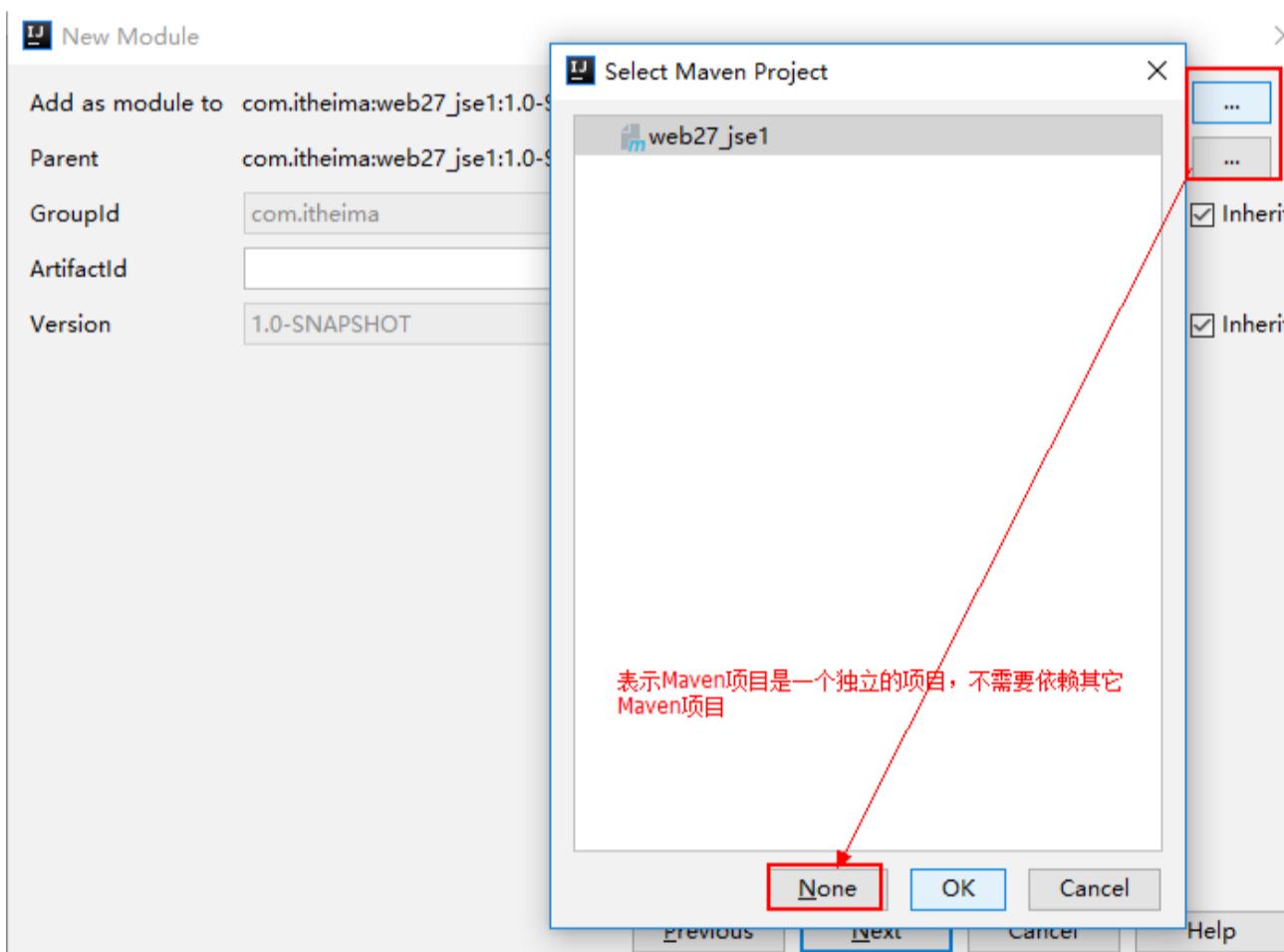
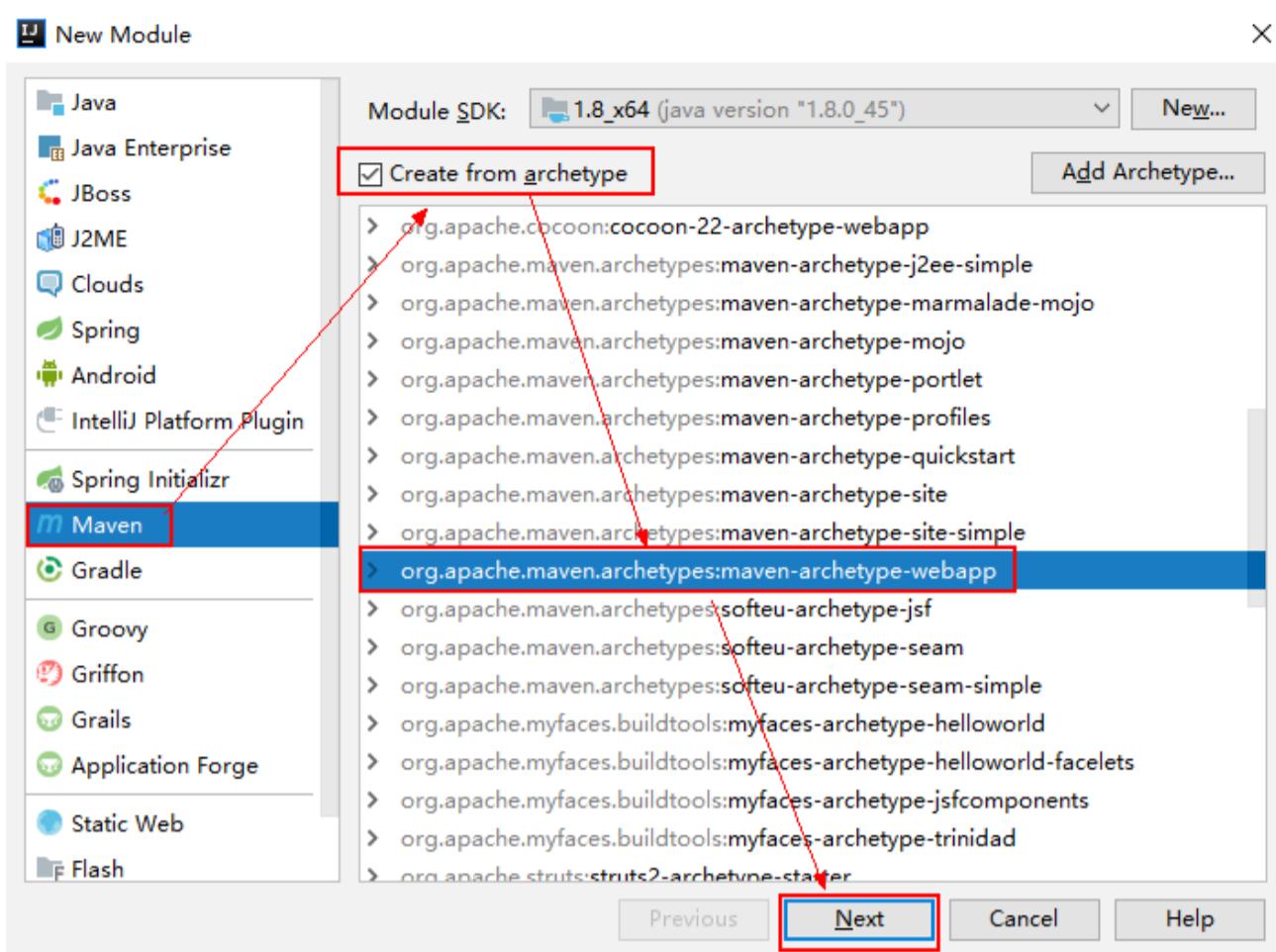


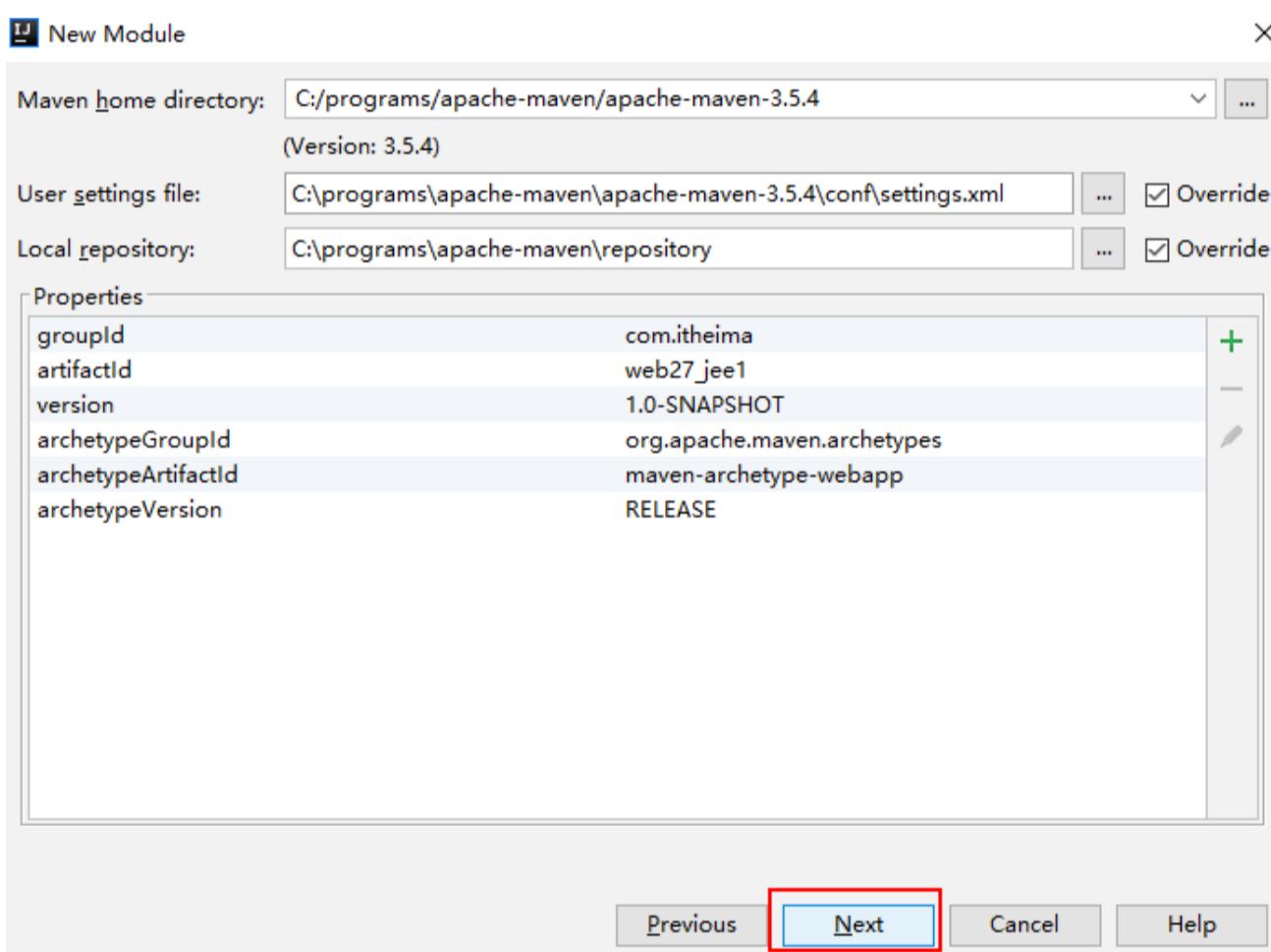
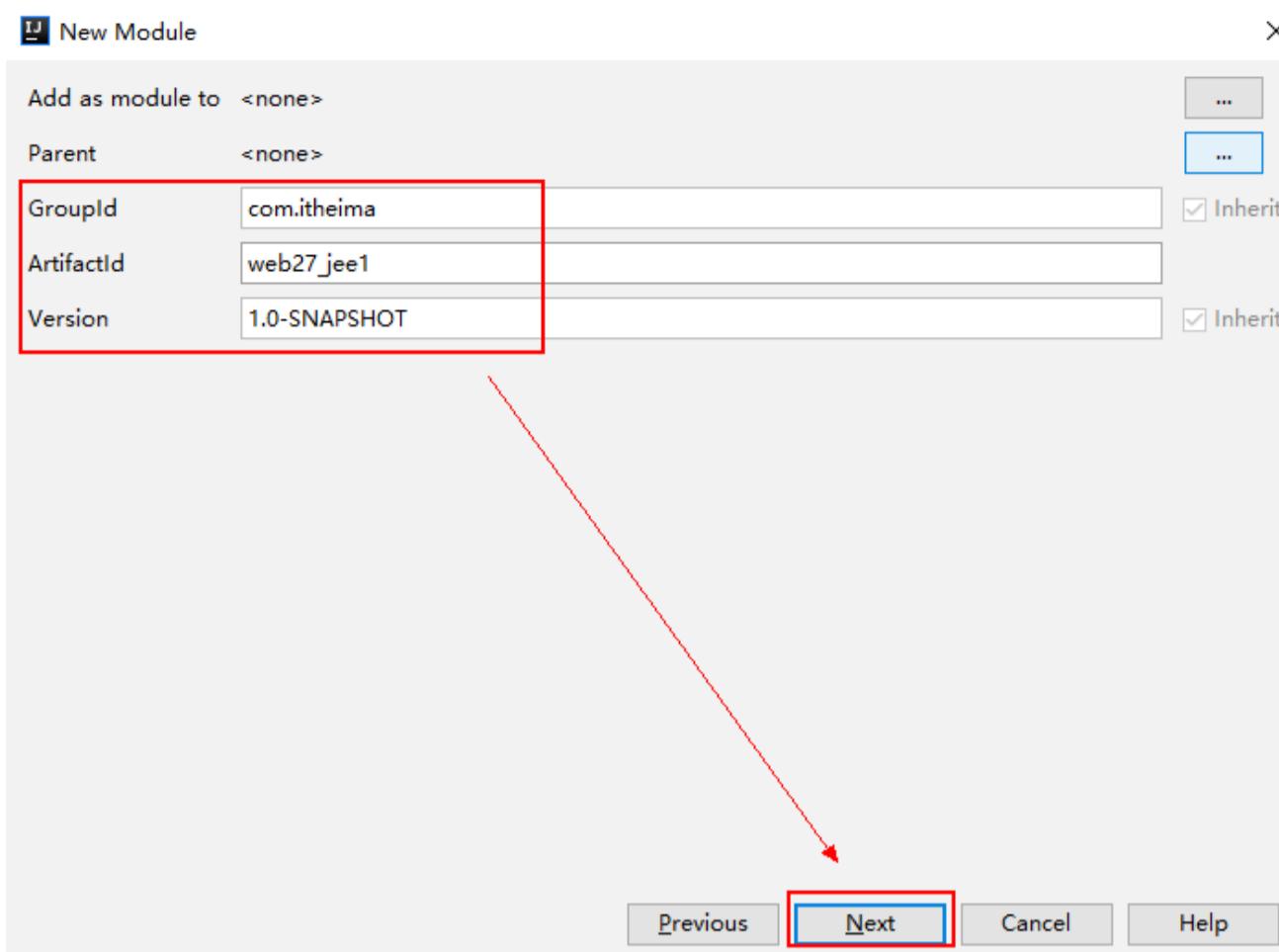
27-4-2_创建 Java 项目

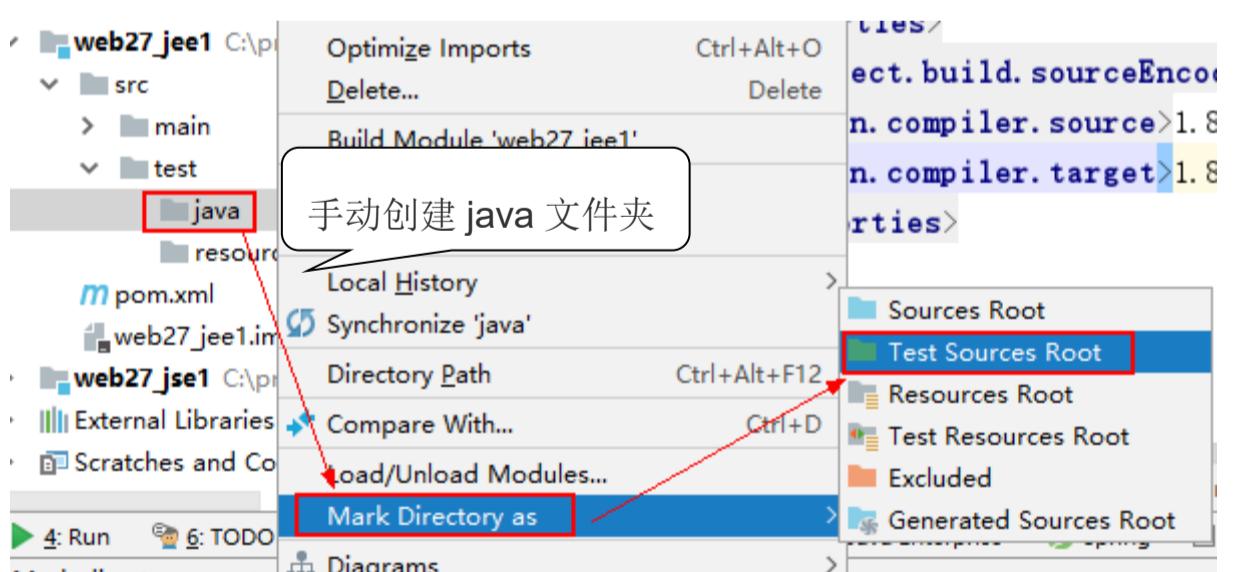
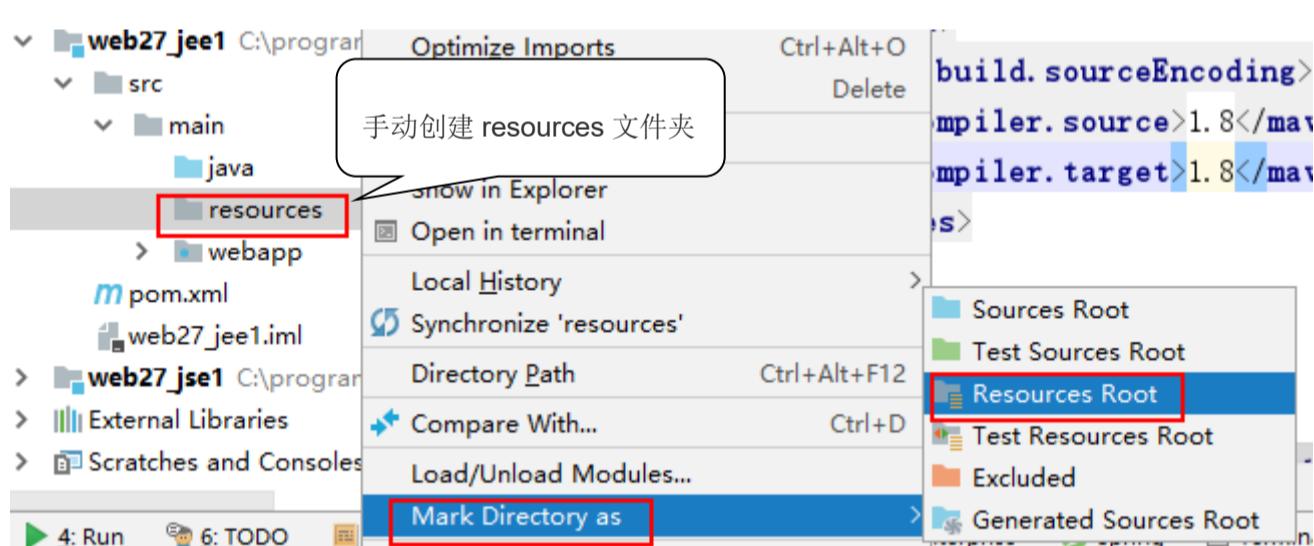
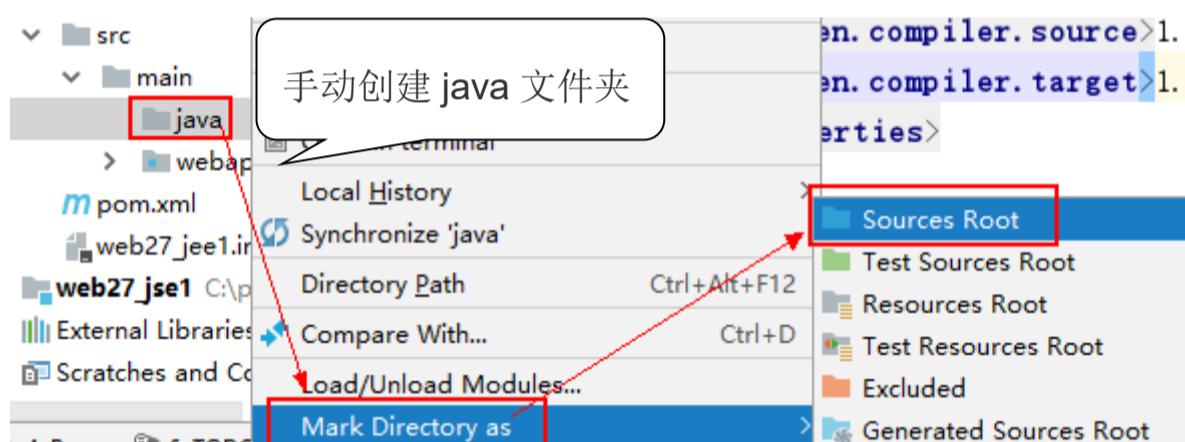
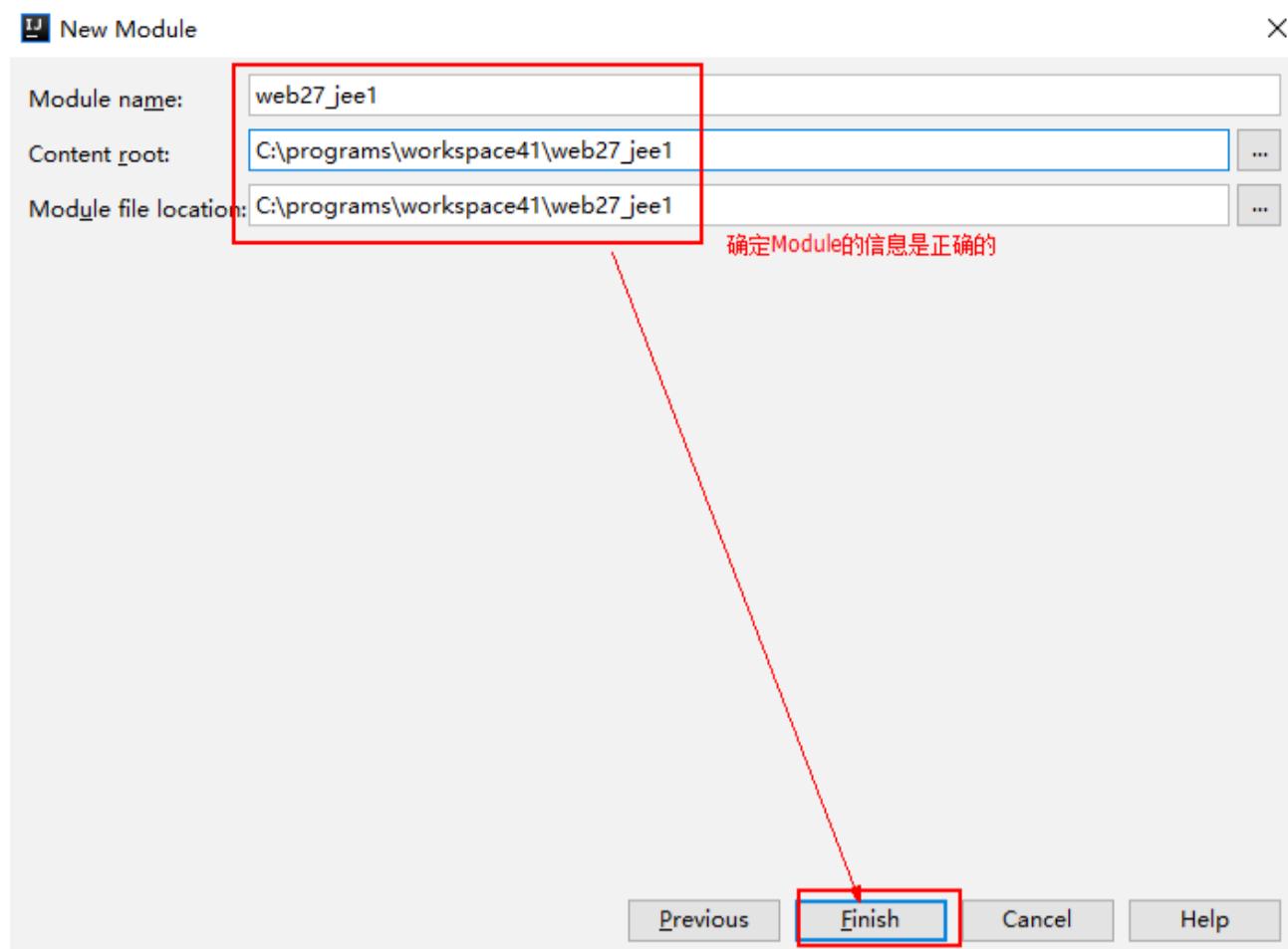


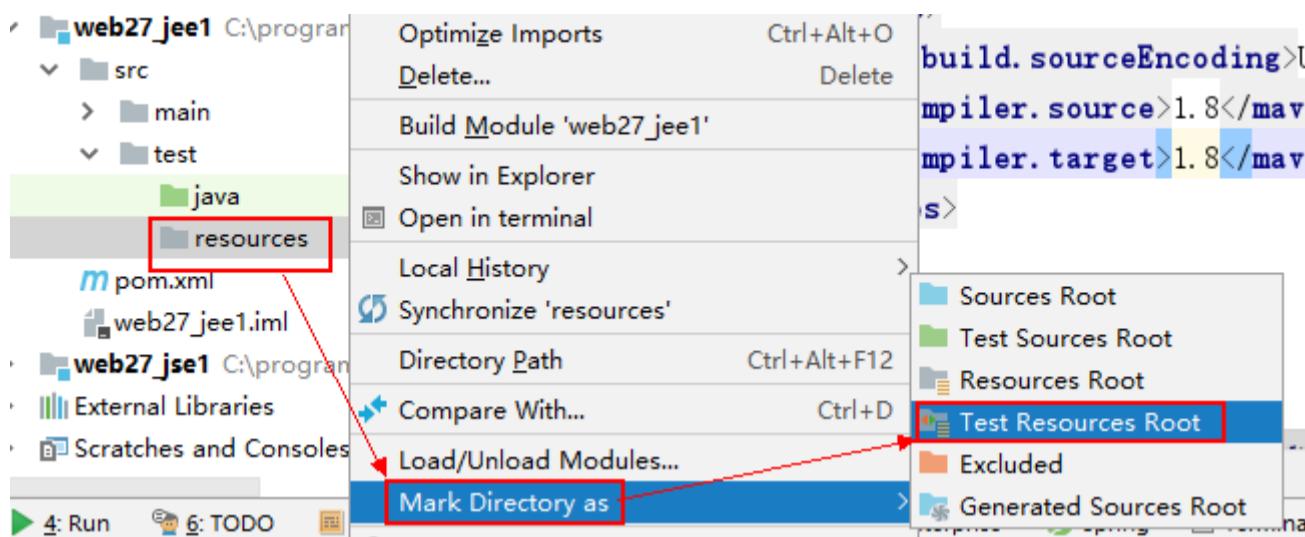


27-4-3_创建 web 项目(重点)









27-4-4_jar 包的依赖范围(适用范围)

项目中的 jar 包如果很多，就可能出现 jar 包冲突。比如：web 应用里有 servlet-api 的 jar 包，web 在 Tomcat 里运行也有 servlet-api 的 jar 包，那么在 Tomcat 启动时，就可能出现 jar 包冲突的问题

为了避免这种情况的发生，Maven 为每个依赖提供了作用范围，即：依赖范围。

compile: 默认的依赖范围，在编译、测试、运行时都有效---最终会出现在 jar 包/war 包里

test: 单元测试有效，其它无效---最后生成的 jar 包/war 包里没有这一类 jar 包

provided: jar 包在其它地方已经提供了，比如：jdk 已经提供了，或者 Tomcat/WebLogic/WebSphere 里边提供了----编译以及测试有效，最终不会出现在 war 包/jar 包里

runtime: 编译无效，测试有效，运行有效。比如：数据库驱动包。

常用 jar 包的依赖范围：

默认引入 的 jar 包 ----- **compile** 【默认范围 可以不写】(编译、测试、运行 都有效)

servlet-api 、**jsp-api** ----- **provided** (编译、测试 有效， 运行时无效 防止和 tomcat 下 jar 冲突)

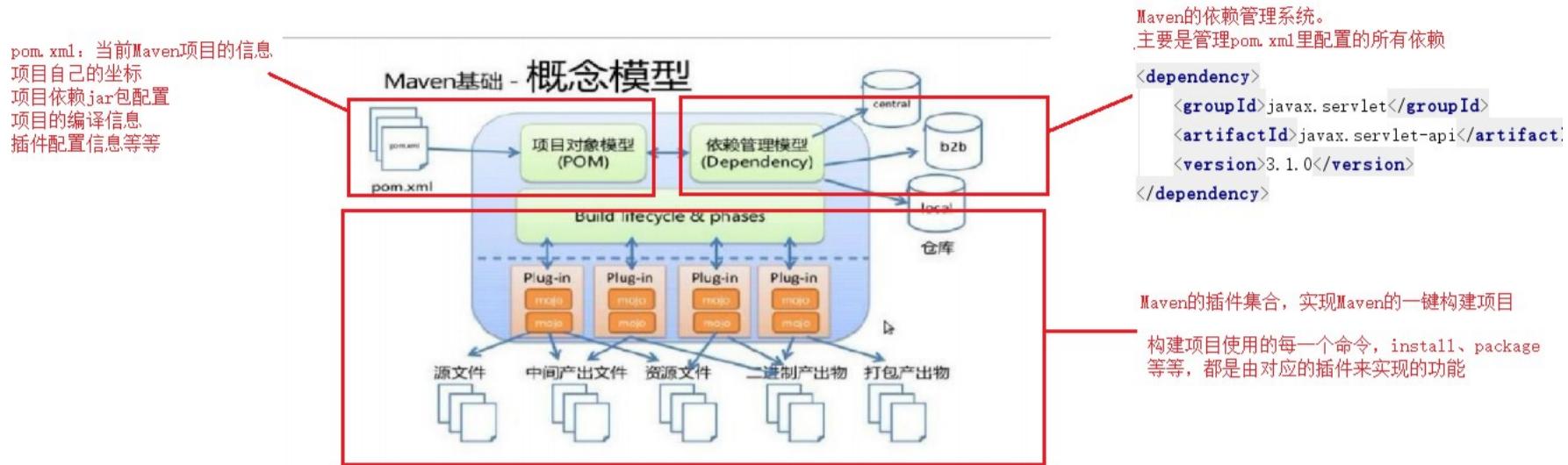
jdbc 驱动 jar 包 ---- **runtime** (测试、运行 有效)

junit ----- **test** (测试有效)

依赖范围	对于编译 classpath 有效	对于测试 classpath 有效	对于运行时 classpath 有效	例子
compile	Y	Y	Y	spring-core
test	-	Y	-	Junit
provided	Y	Y	-	servlet-api
runtime	-	Y	Y	JDBC驱动
system	Y	Y	-	本地的， Maven仓库之 外的类库

27-5_概念模型(了解)

Maven 包含了一个项目对象模型 (Project Object Model, POM.xml)，一组标准集合，一个项目生命周期 (Project Lifecycle)，一个依赖管理系统(Dependency Management System)，和用来运行定义在生命周期阶段(phase)中插件(plugin)目标(goal)的逻辑。



27-5-1_项目对象模型

项目对象模型：Project Object Model，指 Maven 工程的 pom.xml 文件

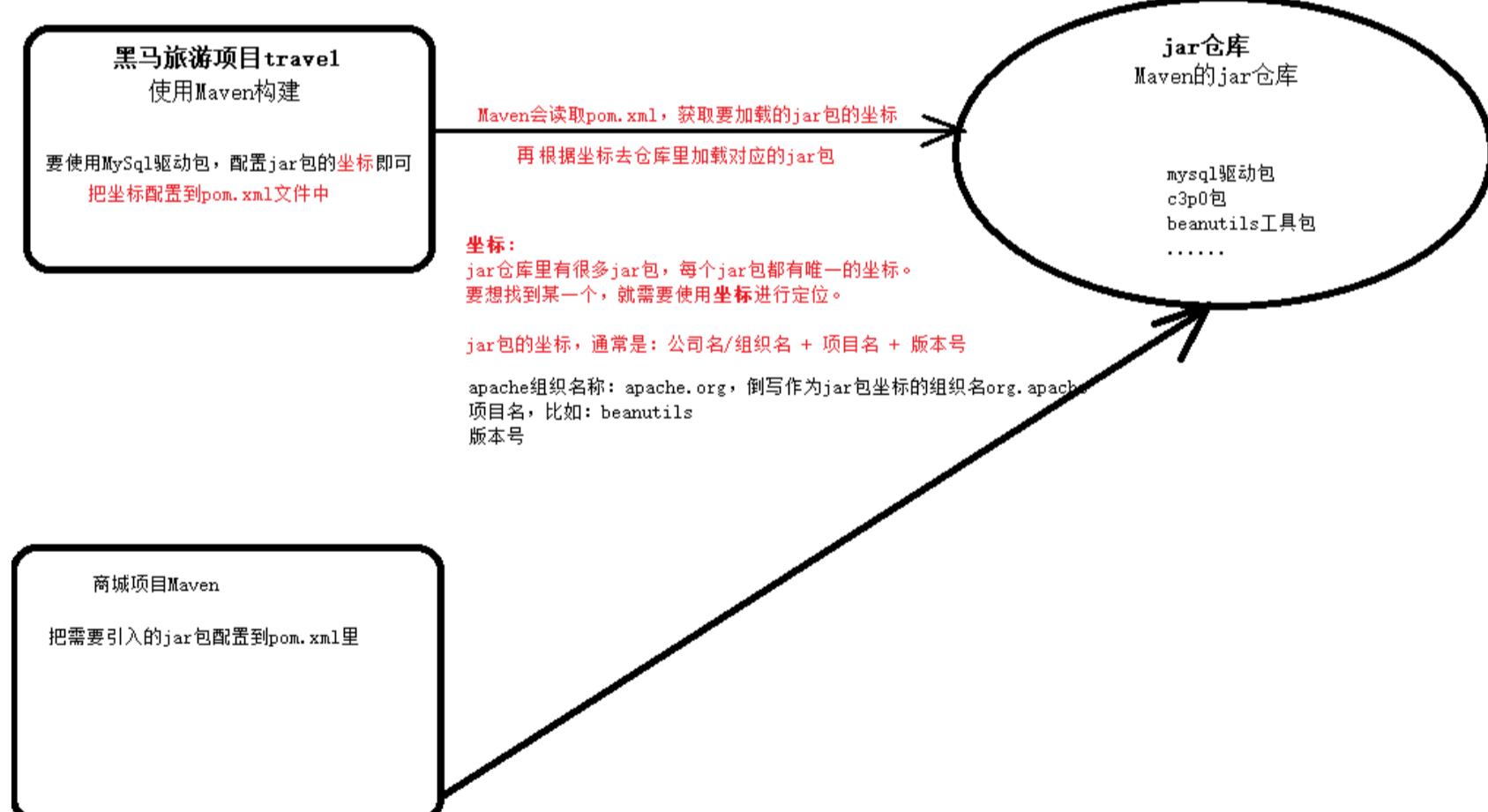
一个 Maven 工程一定有一个 pom.xml 文件，通过 pom.xml 文件来定义项目本身的坐标、项目依赖、项目信息、插件目标等

27-5-2_依赖管理系统

依赖管理系统：Dependency Management System

Maven 通过依赖管理系统，来统一管理项目所依赖的 jar 包

Maven的依赖管理 原理



27-5-3_一个 Maven 项目的生命周期

项目生命周期：Project LifeCycle

使用 **Maven** 完成项目的构建，包括：清理、编译、测试部署等过程。**Maven** 将这些过程规范为一个生命周期，包括以下阶段：



Maven 通过执行一些简单命令，即可实现上边生命周期的各个过程，比如：`mvn compile` 执行编译，`mvn package` 执行打包

27-5-4_一组标准集合

Maven 将整个项目管理过程定义了一组标准，比如：**Maven** 工程的目录结构标准、依赖管理中的坐标的标准等等

27-5-6_插件目标

插件：plugin

目标：goal

Maven 管理项目生命周期的过程，都是基于插件完成的。

27-6_总结

了解 **Maven** 的两大经典作用：依赖管理，构建项目

能够安装并配置 **Maven**，配置本地仓库

了解 **Maven** 仓库的类型

知道 **Maven** 项目的标准目录结构

了解 **Maven** 构建项目时常用的命令

了解 `maven` 命令的生命周期：`clean` 生命周期，`default` 生命周期，`site` 生命周期：执行一个生命周期里的某一个命令，前边的命令会先执行一遍

把 **Maven** 集成到 `idea` 里边

能够在 `idea` 里创建 Java 的 **Maven** 项目

能够在 idea 里创建 web 的 Maven 项目

使用 Tomcat 部署 Maven 项目

会给 Maven 项目增加依赖

了解 jar 包的依赖范围--- **servlet-api 和 jsp-api 设置成 provided**

了解 **Maven** 的概念模型