JavaWeb

网页 ↔ JavaWeb程序 ↔ 数据库

1. **数据库**

- MySQL
- JDBC
- Maven

2. **前端**

- HTML + CSS
- JavaScript
- Ajax + Vue + ElementUI

3. **Web核心**

- Tomcat + HTTP + Servlet
- Request + Response
- JSP
- Cookie + Session
- Fillter + Listener

一、数据库

1.1 MySQL

安装 MySQL 5.7.30

- 添加环境变量
- 配置my.ini文件

一些命令：

```
1. mysql -u root -p # 从命令行进入mysql

2. net start/stoo mysql # 命令行下输入，开启/关闭本机MySQL服务

3. show databases; # 查询所有数据库名称
```

1.2 SQL语句分类

- **DDL**（Data Definition Language）数据定义语言，用来定义数据库对象：数据库，表，列等
- **DML**（Data Manipulation Language）数据操作语言，用来对数据库中表的数据进行增删改
- **DQL**（Data Query Language）数据查询语言，用来查询数据库中表的记录（数据）
- **DCL**（Data Control Language）数据控制语言，用来定义数据库的访问权限和安全级别，及创建用户

1.2.1 DDL——操作数据库

1. 查询

```
SHOW DATABASES;
```

2. 创建

- 创建数据库

```
CREATE DATABASE 数据库名称;
```

- 创建数据库（判断，如果不存在则创建）

```
CREATE DATABASE IF NOT EXISTS 数据库名称;
```

3. 删除

- 删除数据库

```
DROP DATABASE 数据库名称;
```

- 删除数据库（判断，若存在则删除）

```
DROP DATABASE IF EXISTS 数据库名称;
```

4. 使用数据库

- 查看当前使用的数据库

```
SELECT DATABASE();
```

- 使用数据库

```
USE 数据库名称;# 可以从当前数据库直接转到另一个数据库
```

1.2.2 DDL——操作表

CRUD

- Create 创建
- Retrieve 查询
- Update 修改
- Delete 删除

1. 查询

- 查询当前数据库下所有表名称

```
SHOW TABLES;
```

- 查询表结构

```
DESC 表名称;
```

2. 创建

```
CREATE TABLE 表名(
                字段名1 数据类型1,
                字段名2 数据类型2,
                ...
                字段名n 数据类型n,

);  # 最后一行末尾不能加逗号
```

常用数据类型:

age **int**

score **double**(总长度，小数点后保留的位数)

birthda **date**

name **char(10)** # 定长字符串，直接分配10个字符空间，存储性能高，浪费空间（以空间换时间）

name **varchar(10)** # 变长字符串，根据实际分配字符空间，存储性能低，节约空间（以时间换空间）

3. 删除

- 删除表

```
DROP TABLE 表名;
```

- 删除表时判断表是否存在

```
DROP TABLE IF EXISTS 表名;
```

4. 修改

- 修改表名 RENAME

```
ALTER TABLE 表名 RENAME TO 新的表名;
```

- 添加一列 ADD

```
ALTER TABLE 表名 ADD 列名 数据类型;
```

- 修改数据类型 MODIFY

```
ALTER TABLE 表名 MODIFY 列名 新数据类型;
```

- 修改列名和数据类型 CHANGE

```
ALTER TABLE 表名 CHANGE 列名 新列名 新数据类型;
```

- 删除列 DROP

```
ALTER TABLE 表名 DROP 列名;
```

1.2.3 DML

1. 添加数据

- 给指定列添加数据

```
INSERT INTO 表名(列名1, 列名2, ...) VALUES(值1, 值2, ...);
```

- 给全部列添加数据

```
INSERT INTO 表名 VALUES(值1, 值2, ...);
```

- 批量添加数据

```
INSERT INTO 表名(列名1, 列名2, ...) VALUES(值1, 值2, ...),(值1, 值2, ...)...;
INSERT INTO 表名 VALUES(值1, 值2, ...),(值1, 值2, ...)...;
```

2. 修改数据

```
UPDATE 表名 SET 列名1=值1, 列名=值2, ... [WHERE 条件]; # 若不加条件，则修改所有数据
```

3. 删除数据

- 删除数据

```
DELETE FROM 表名 [WHERE 条件];   # 若不加条件，则删除所有数据
```

1.2.4 DQL

1. 查询语法

```
SELECT
    字段列表
FROM
    表名列表
WHERE
    条件列表
GROUP BY
    分组字段
HAVING
    分组后的条件
ORDER BY
    排序字段
LIMIT
    分页限定
```

2. 基础查询

- 查询所有字段

```
select * from stu; # 实际应用不建议使用*
```

- 去除重复记录

关键字：distinct/DISTINCT

```
select address from stu;  # 未去除
select distinct address from stu; # 去除
```

- 对列名起别名

```
select name, math as 数学成绩, english as 英语成绩, from stu; # as关键字可以省略，使用空格代替
```

3. 条件查询

- 常规查询

```
select * from stu where age > 20;

# and or 关键字
select * from stu where age > 20 and age <= 30 ;
select * from stu where age = 20 or age = 30 ;

# between 关键字
select * from stu where age between 20 and 30;

# = != <>
select * from stu where age = 20;
select * from stu where age != 20;
```

```
select * from stu where age <> 20;  # <> 不等于关键字

# in 关键字
select * from stu where age in (20,30,40);

!注意：null值的比较不能使用=或!=。需使用is或is not
```

- 模糊查询 LIKE

通配符：

“_”：代表单个任意字符

“%”：代表任意个数字符

```
select * from stu where name like '马%' # 查询姓马的记录
select * from stu where name like '_花%' # 查询第二个字是'花'的记录
select * from stu where name like '%德%' # 查询名字含'德'的记录
```

4. 排序查询

- 语法

排序方式

ASC：升序（默认）

DESC：降序

```
# 若有多个排序条件，当前面的条件值一样时，才会根据第二条件进行排序
select 字段列表 from 表名 order by 排序字段名1 [排序方式1], 排序字段名2 [排序方式2] ...;

# 例
select * from stu order by math desc, english asc;
```

5. 分组查询

- 聚合函数

（1）概念：将一列数据作为一个整体，进行纵向计算。

⚠ null不参与所有聚合函数运算

（2）分类：

函数名	作用
count(列名)	统计数量（一般选用不为null的列）
max(列名)	最大值
min(列名)	最小值
sum(列名)	求和
avg(列名)	平均值

(3) 语法

```
select 聚合函数名(列名) from 表名;
```

(4) 例

```
select count(id) from stu; # 统计班级一共有多少学生

select max(math) from stu; # 查询数学成绩最高分

select sum(math) from stu;
```

- 分组查询

(1) 语法

```
select from [where 分组前条件限定] group by 分组字段名 [having 分组后条件过滤];
```

⚠ 注意：分组之后，查询的字段为聚合函数和分组字段，查询其他字段无任何意义

where和having区别：

- 执行时机不一样：where是分组之前进行限定，不满足wher条件，则不参与分组，而having是分组之后对结果进行过滤。
- 可判断的条件不一样：where不能对聚合函数进行判断，having可以。

⚠ 执行顺序：where > 聚合函数 > having

(2) 例

```
# 查询男同学和女同学各自的数学平均分
select sex, avg(math) from stu group by sex;  # 查询的字段要跟分组的字段(order by)一致，

# 此处为sex，查询其他字段无任何意义

# 查询男同学和女同学各自的数学平均分，以及各自人数
select sex, avg(math), count(*) from stu group by sex;

# 查询男同学和女同学各自的数学平均分，以及各自人数，要求：分数低于70分的不参与分组
select sex, avg(math), count(*) from stu where math>70 group by sex;

# 查询男同学和女同学各自的数学平均分，以及各自人数，要求：分数低于70分的不参与分组，分组之后人数大于2
select sex, avg(math), count(*) from stu where math>70 group by sex having count(*) > 2;
```

6. 分页查询

（1）语法

```
select 字段列表 from 表名 limit 起始索引(start with 0), 查询条目数;
```

计算公式：起始索引 = (当前页码-1) * 每页显示的条数

⚠

- 分页查询limit仅在MySQL数据库使用
- Oracle使用的是rownumber
- SQL server使用的是top

（2）例

```
# 从0开始查询，查询3条数据
select * from stu limit 0, 3;

# 每页显示3条数据，查询第1页 (0 1 2)
select * from stu limit 0, 3;

# 每页显示3条数据，查询第2页 (3 4 5)
select * from stu limit 3, 3;

# 每页显示3条数据，查询第2页 (6 7 8)
select * from stu limit 6, 3;
```

1.3 约束

1.3.1 概念

- 是作用于表中列上的规则，用于限制加入表的数据
- 约束的存在保证了数据库中数据的正确性、有效性和完整性

1.3.2 分类

共6种

- 非空约束 NOT NULL
- 唯一约束 UNIQUE
- 主键约束 PRIMARY KEY
- 外键约束 FOREIGN KEY
- 默认约束 DEFAULT
- 检查约束 CHECK

1.3.3 外键约束

- 添加外键

```
# 部门表
create talbe dept(
    id int primary key,
    dep_name varchar(20),
    addr varchar(20)
);

# 员工表
create table emp(
    id int primary key auto_increment, # auto_increment自动增长
    name varchar(20),
    age int,
    dep_id int,  # 所属部门

    # 添加外键dep_id，关联dept表的id主键
    constraint fk_emp_dept foreign key(dep_id) references dept(id)
);

# 创建员工表之前需先创建部门表。
# 主表为dept，从表为emp
# 首次添加数据时，先添加主表，后添加从表
```

- 建完表后，添加外键

```
alter table emp add constraint fk_emp_dept foreign key(dep_id) references
dept(id);
```

- 删除外键

```
alter table emp drop foreign key fk_emp_dept; # 从emp中删除外键fk_emp_dept
```

1.4 数据库设计

1.4.1 表关系

1. 一对多

如部门表和员工表，一个部门对应对各员工，一个员工对应一个部门

实现方式：在多的一方建立外键，指向”一“的一方的主键

2. 多对多

如：订单和商品，一个商品对应多个订单，一个订单包含多个商品

实现方式：简历第三张中间表，中间表至少包含两个外键，分别关联两方主键

3. 一对一

如：用户和用户详情

一对一关系多用于表拆分，将一个实体中经常使用的字段放一张表，不经常使用的字段放另一张表，用于提升查询性能

实现方式：在任意一方加入外键，关联另一方主键，并且设置外键为唯一（UNIQUE）

1.5 多表查询

笛卡尔积：去A，B集合所有组合情况

多表查询：

- 连接查询
  - 内连接：相当于查询A、B交集数据
  - 外连接
    - 左外连接：相当于查询A表所有数据和交集部分数据
    - 右外连接：相当于查询B表所有数据和交集部分数据
- 子查询

1.5.1 内连接

- 隐式内连接

```
select 字段列表 from 表1,表2,... where 条件;

select * from emp, dept where emp.dep_id = dept.did;

# 选择查询的列
select emp.name, emp.gender, dept.dname from emp, dept where emp.dep_id =
dept.did;

# 给表起别名
select t1.name, t1.gender, t2.dname from emp t1, dept t2 where t1.dep_id =
t2.did;
```

- 显式内连接

```
select 字段列表 from 表1 [inner] join 表2 on 条件;

select * from emp inner join dept on emp.id = dept.did;
```

1.5.2 外连接

- 左外连接

```
select * from emp left join dept on emp.id = dept.did;
```

- 右外连接

```
select * from emp right join dept on emp.id = dept.did;
```

1.5.3 子查询

概念：查询中嵌套查询，称嵌套查询为字查询

子查询根据查询结果不同，作用不同：

- 单行单列

作为条件值，使用=、!=、<、>等进行条件判断

```
select 字段列表 from 表名 where 字段名=(子查询);
```

- 多行单列

作为条件值，使用in等关键字进行条件判断

```
select 字段列表 from 表名 where 字段名 in (子查询);
```

- 多行多列

作为虚拟表

```
select 字段列表 from (子查询) where 条件;
```

1.6 事务

1.6.1 简介

- 数据库的事务（Transaction）是一种机制、一个操作序列，包含了一组数据库操作命令
- 事务把所有的命令作为一个整体一起向系统提交或撤销操作请求，即这一组数据命令要么同时成功，要么同时失败
- 事务是一个不可分割的工作逻辑单元

1.6.2 命令

- 开启事务

```
START TRANSACTION;
# 或者
 BEGIN
```

- 提交事务

```
COMMIT;
```

- 回滚事务

```
ROLLBACK;
```

1.6.3. 四大特征 ACID

- 原子性（Atomicity）
- 一致性（Consistency）
- 隔离性（Isolation）
- 持久性（Durability）

⚠ MySQL里，事务默认情况下是自动提交的。Oracle默认不提交。

查看方式：

```
select @@autoncommit;  # 返回值是1时，代表默认提交，返回值是0时，默认不提交
```

修改事务的提交方式：改为手动提交

```
set @@autocommit;
```

1.7 JDBC

1.7.1 简介

JDBC概念：

- JDBC就是使用Java语言操作关系型数据库（MySQL、Oracle...）的一套API。
- 全称：（Java DataBase Connectivity）Java数据库连接

JDBC本质：

- 官方（Sun公司）定义的一套操作所有关系型数据库的规则，即接口（Interface）
- 各个数据库厂商（MySQL、Oracle等）去实现这套接口，提供数据库驱动jar包
- 我们可以使用这套接口（JDBC）编程，真正执行的代码是驱动jar包中的实现类

JDBC好处：

- 各数据库厂商使用相同的接口，Java代码不需要针对不同的数据库分别开发

- 可随时替换底层数据库，访问数据库的Java代码基本不变

1.7.2 入门

步骤

0. 创建工程，导入驱动jar包

1. 注册驱动

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

2. 获取连接

```
Connection conn = DriverManager.getConnection(url, username, password);
```

3. 定义SQL语句

```
String sql = "update...";
```

4. 获取执行SQL对象

```
Statement stmt = conn.createStatement();
```

5. 执行SQL

```
stmt.executeUpdate(sql);
```

6. 处理返回结果

7. 释放资源

1.7.3 JDBC API

1. DriverManager

DriverManager（驱动管理类）作用：

- 注册驱动

```
DriverManager.registerDriver(new Driver); # 取反射入口时自动运行。
```

⚠ 提示：

MySQL 5之后的驱动包，可以省略注册驱动步骤

自动加载jar包中META-INF/services./java.sql.Driver文件中的驱动类

- 获取数据库连接

```
Connection conn = DriverManager.getConnection(url, username, password);

// url语法：jdbc:mysql://ip地址:端口号/数据库名称?参数键值对1&参数键值对2...
// url可配置useSSL=false参数，禁用安全连接方式，消除警告提示
// jdbc:mysql://127.0.0.1:3306/db1?useSSL=false
```

2. Connection

Connection（数据库连接对象）作用：

1. **获取执行SQL对象**

- 普通执行SQL对象

```
Statement createStatement()
```

- 预编译SQL的执行SQL对象：防止SQL注入

```
PreparedStatement prepareStatement(sql)
```

- 执行存储过程的对象

```
CallableStatement prepareCall(sql)
```

2. **管理事务**

- MySQL事务管理

开始事务：BEGIN;/START TRANSACTION;
提交事务：COMMIT;
回滚事务：ROLLBACK;

MySQL默认自动提交事务

- JDBC事务管理：Connection接口中定义了3个对应的方法

开始事务：setAutoCommit(boolean autoCommit)：true为自动提交事务；false为手动提交事务，即为开启事务

提交事务：commit()

回滚事务：rollback();

3. Statement

作用：

1. 执行SQL语句

```
int executeUpdate(sql) // 执行DML、DDL语句。返回值：1）DML语句影响的行数 2）DDL语句执行后，执行成功也可能返回0

ResultSet executeQuery(sql) // 执行DQL语句。返回值：ResultSet 结果集对象
```

4. ResultSet

作用：

- 封装了DQL查询语句的结果

> ResultSet stmt.executeQuery(sql)：执行DQL语句，返回ResultSet对象

获取查询结果

> **boolean next()** : (1) 将光标从当前位置向前移动一行 (2) 判断当前行是否为有效行
>
> 返回值：
>
> - true：有效行，当前行有数据
> - false：无效行，当前行没有数据

> **xxx getXxx(参数)**：获取数据
>
> xxx：数据类型，如 int getInt(参数)；String getString(参数)
>
> 参数：
>
> - int：列的编号，**从1开始**
> - String：列的名称

5. PreparedStatement

(1) 作用

预编译SQL语句并执行：预防SQL注入问题（将敏感字符进行转义）

SQL注入：通过操作输入来修改事先定义好的SQL语句，用以达到执行代码对服务器进行**攻击**的方法。

(2) 语法

- 获取PrepareStatement对象

```
// SQL语句中的参数值，使用?占位符替代
String sql = "select * from user where username=? and password=?";

// 通过Connection对象获取，并且传入对应的sql语句
PrepareStatement pstmt = conn.getPrepartStatement(sql);
```

- 设置参数值

> PrepareStatement对象：setXxx(参数1, 参数2)：给?赋值
>
> Xxx：数据类型；如setInt(参数1, 参数2)
>
> 参数：
>
> - 参数1：?的位置编号，**从1开始**
> - 参数2：?的值

- 执行SQL

```
// 不需再传递sql
executeUpdate();
executeQuery();
```

(3) PreparedStatement好处

- 预编译SQL，性能更高
- 防止SQL注入：将敏感字符进行转义

(4) 开启

- PreparedStatement预编译功能开启：**useServerPrepStmts=true** (在url中添加)
- 配置MySQL执行日志（重启MySQL服务后生效）（my.ini文件）

> log-output=FILE
>
> general-log=1
>
> general_log_file="D:\mysql.log"
>
> slow-query-log=1
>
> slow_query_log_file="D:\mysql.slow.log"
>
> long_query_time=2

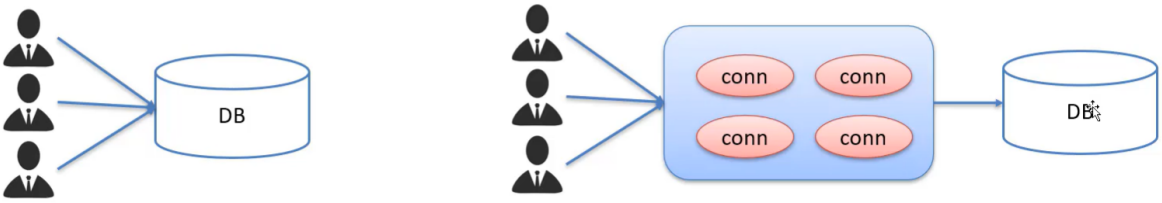
(5) 原理

- 在获取PreparedStatement对象时，将sql语句发送给MySQL服务器进行检查，编译（这些步骤很耗时）
- 执行时就不用再进行这些步骤了，速度更快
- 如果sql模板一样，则只进行一次检查、编译

1.7.4 数据库连接池

1. 数据库连接池简介

- 数据库连接池是一个容器，负责分配、管理数据库连接（Connection）
- 它允许应用程序重复使用一个现有的数据库连接，而不是再重新建立一个；
- 释放空闲时间超过最大空闲时间的数据库连接来避免因为没有释放数据库连接而引起的数据库连接遗漏
- 好处
    - 资源重用
    - 提升系统响应速度
    - 避免数据库连接遗漏



2. 数据库连接池实现

- 标准接口：**DataSource**
    - 官方（SUN）提供的数据库连接池标准接口，由第三方组织实现此接口。
    - 功能：获取连接

      > Connection getConnection()

- 常见数据库连接池
    - DBCP
    - C3P0
    - Druid
- Druid（德鲁伊）
    - Druid连接池是阿里巴巴开源的数据库连接池项目
    - 功能强大，性能优秀，是java语言最好的数据库连接池之一

3. Druid数据库连接池

使用步骤

(1) 导入jar包 druid-1.1.12.jar

(2) 定义配置文件

保存为druid.properties

```
driverClassName=com.mysql.jdbc.Driver
url=jdbc:mysql://127.0.0.1:3306/mydb1?useSSL=false&useServerPrepStmts=true
username=root
password=123789
# 初始化连接数量
initialSize=5
# 最大连接数
maxActive=10
# 最大等待时间
maxWait=3000
```

(3) 加载配置文件

```
Properties prop = new Properties();
prop.load(new FileInputStream("src/jdbctest1/druid.properties"));
```

(4) 获取数据库连接池对象

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

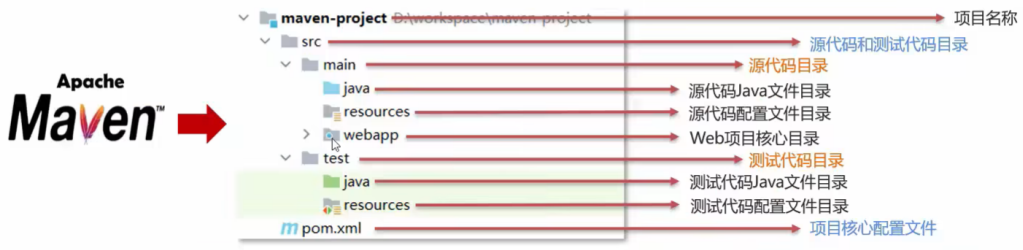
(5) 获取连接

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

1.8 Maven

- Maven是专门用于管理和构建Java项目的工具，它的主要功能有
  - 提供了一套标准化的项目结构

    所有IDE使用Maven构建的项目结构完全一样，所有IDE创建的Maven项目可以通用



  - 提供了一套标准化的构建流程（编译、测试、打包、发布...）

    Maven提供了一套简单的命令来完成项目构建

  - 提供了一套依赖管理机制

    依赖管理其实就是管理你项目所依赖的第三方资源（jar包、插件...）

1.8.1 Maven简介

Apache Maven是一个项目管理和构建工具，它基于**项目对象模型（POM）**的概念，通过一小段描述信息来管理项目的构建、报告和文档

1. Maven模型

2. 仓库分类

- 本地仓库：自己计算机上的一个目录
- 中央仓库：由Maven团队维护的全球唯一的仓库

  地址：https://repo1.maven.org/maven2/
- 远程仓库：一般由公司团队搭建的私有仓库

1.8.2 Maven配置

安装好Maven后

- 配置本地仓库

  修改conf/settings.xml中的为一个指定目录

```
<localRepository>D:\Program Files\apache-maven-
3.5.4\mvn_repos</localRepository>
```

- 配置阿里云私服

  修改conf/settings.xml中的标签，为其添加如下字标签：

```
<mirror>
      <id>alimaven</id>
      <name>aliyun maven</name>
      <url>https://maven.aliyun.com/repository/public/</url>
      <mirrorOf>central</mirrorOf>
</mirror>
```

1.8.3 Maven基本使用

1. Maven常用命令

- **complie**：编译
- **clean**：清理
- **test**：测试
- **package**：打包
- **install**：安装

2. Maven生命周期

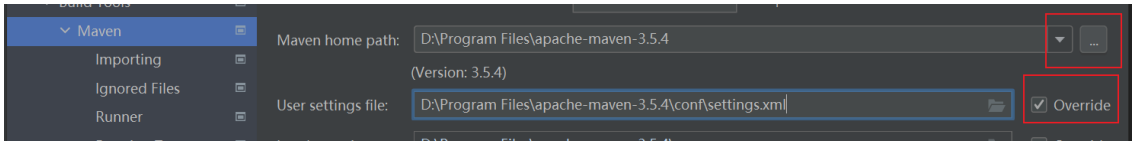
- Maven构建项目生命周期描述的是一次构建过程经历了多少事件
- Maven对项目构建的生命周期划分为3套
  - clean：清理工作
  - default：核心工作，例如编译、测试、打包、安装等
  - site：产生报告，发布站点等

  同一生命周期内，执行后边的命令，前边的所有命令会自动执行

1.8.4 IDEA配置Maven

1. IDEA配置Maven环境

- 选择IDEA中的 File --> Settings
- 搜索maven
- 设置IDEA使用本地安装的Maven，并修改配置文件路径



2. Maven坐标详解

- 什么是坐标
  - Maven中的坐标是资源的唯一标识符
  - 使用坐标来定义项目或引入项目中需要的依赖
- Maven坐标主要组成
  - groupId：定义当前Maven项目隶属组织名称（通常是域名反写，例如：com.company）
  - artifactId：定义当前Maven项目名称（通常是模块名称，例如order-service、good-service）
  - version：定义当前项目版本号

3. IDEA创建Maven项目

- 创建项目，选择Maven，点击next
- 填写模块名称，坐标信息等，点击finish，创建完成
- 编写HelloWorld，运行

4. IDEA导入Maven项目

- 选择右侧Maven面板，点击+号
- 选中对应项目的pom.xml文件，双击
- 如果没有Maven面板，选择View --> Appearance --> Tool Windows Bars

5. 依赖管理

使用坐标导入jar包

1. 在pom.xml中编写标签
2. 在标签中使用引入坐标
3. 定义坐标的groupId、artifactId、version
4. 点击刷新，使坐标生效

6. 依赖范围

- 通过设置坐标的依赖范围（scope），可以设置对应jar包的作用范围：编译环境、测试环境、运行环境
- 默认值：compile

依赖范围	编译classpath	测试classpath	运行classpath	例子
compile	Y	Y	Y	logback
test	-	Y	-	Junit
provided	Y	Y	-	servlet-api
runtime	-	Y	Y	jdbc驱动
system	Y	Y	-	存储在本地的jar包
import	引入DependencyManagement			

1.9 MyBtis

可以到mybatis官网查看文档

什么是MyBatis?

- 是一款优秀的**持久层框架**，用于简化JDBC开发
- MyBatis免除了几乎所有的JDBC代码，以及设置参数和获取结果集的工作
- 其他持久层框架
  - MyBatis-Plus
  - Spring Data JPA
  - Hibernate

持久层

- 负责将数据保存到数据库的那一层代码
- JavaEE三层框架：表现层、业务层、持久层

框架

- 就是一个半成品软件，是一套可重用的、通用的、软件基础代码模型
- 在框架的基础之上构建软件编写更加高效、规范、通用、可扩展

1.9.1 MyBatis快速入门

- 创建项目，修改pom.xml，导入坐标
- 编写**MyBati核心配置文件**--------替换连接信息，解决硬编码问题
- 编写**SQL映射文件**----------管理sql语句，解决硬编码问题

```
// 未使用代理情况

// 1. 加载mybatis核心配置文件，获取SqlSessionFactory
String resource = "mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);

// 2. 获取sqlSession对象，用它来执行SQL
SqlSession sqlSession = sqlSessionFactory.openSession();

// 3. 执行sql
List<User> users = sqlSession.selectList("test.selectAll");

System.out.println(users);

// 4. 释放资源
sqlSession.close();
```

- 存在问题：如果有很多地方需要SqlSessionFactory对象，则获取SqlSessionFactory的代码重复，且会生成多个连接池，造成资源浪费。解决：使用静态工厂类，以及static代码块（static代码块在加载类的时候在运行一次，static代码块不能抛出异常，需就地进行处理）

```
public class SqlSessionFactoryUtils {
    private static SqlSessionFactory sqlSessionFactory;

    static {
        try {
            String resource = "mybatis-config.xml";
            InputStream inputStream =
Resources.getResourceAsStream(resource);
            sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);
        } catch (IOException e) {
            e.printStackTrace();
        }
```

```
    }
    public static SqlSessionFactory getSqlSessionFactory(){
        return sqlSessionFactory;
    }
}
```

1.9.2 Mapper代理开发

1. 目的

- 解决原生方式中的硬编码
- 简化后期执行SQL

2. 规则

(1) 定义与SQL映射文件同名的Mapper接口，并且将Mapper接口和SQL映射文件放置在同一目录下

(2) 设置SQL映射文件的namespace属性为Mapper接口全限定名

(3) 在Mapper接口中定义方法，方法名就是SQL映射文件中SQL语句的id，并保持参数类型和返回值类型一致

(4) 编码

- 通过SqlSession 的getMapper方法获取Mapper接口的代理对象
- 调用对应方法完成SQL的执行

```
// 1. 加载mybatis核心配置文件，获取SqlSessionFactory
String resource = "mybatis-config.xml";
InputStream inputStream = Resources.getResourceAsStream(resource);
SqlSessionFactory sqlSessionFactory = new
SqlSessionFactoryBuilder().build(inputStream);

// 2. 获取sqlSession对象，用它来执行SQL
SqlSession sqlSession = sqlSessionFactory.openSession();


// 3.1 获取UserMapper接口的代理对象
UserMapper userMapper = sqlSession.getMapper(UserMapper.class);
List<User> users = userMapper.selectAll();

System.out.println(users);

// 4. 释放资源
sqlSession.close();
```

注：若Mapper接口名称和SQL映射文件名称相同，并在同一目录下，则可以使用包扫描的方法简化SQL映射文件的加载

```xml
<mappers>
    <!--加载sql的映射文件-->
    <!--<mapper resource="com/itheima/mapper2/UserMapper.xml"/>-->
    <package name="com.itheima.mapper"/>
</mappers>
```

1.9.3 Mybatis核心配置文件

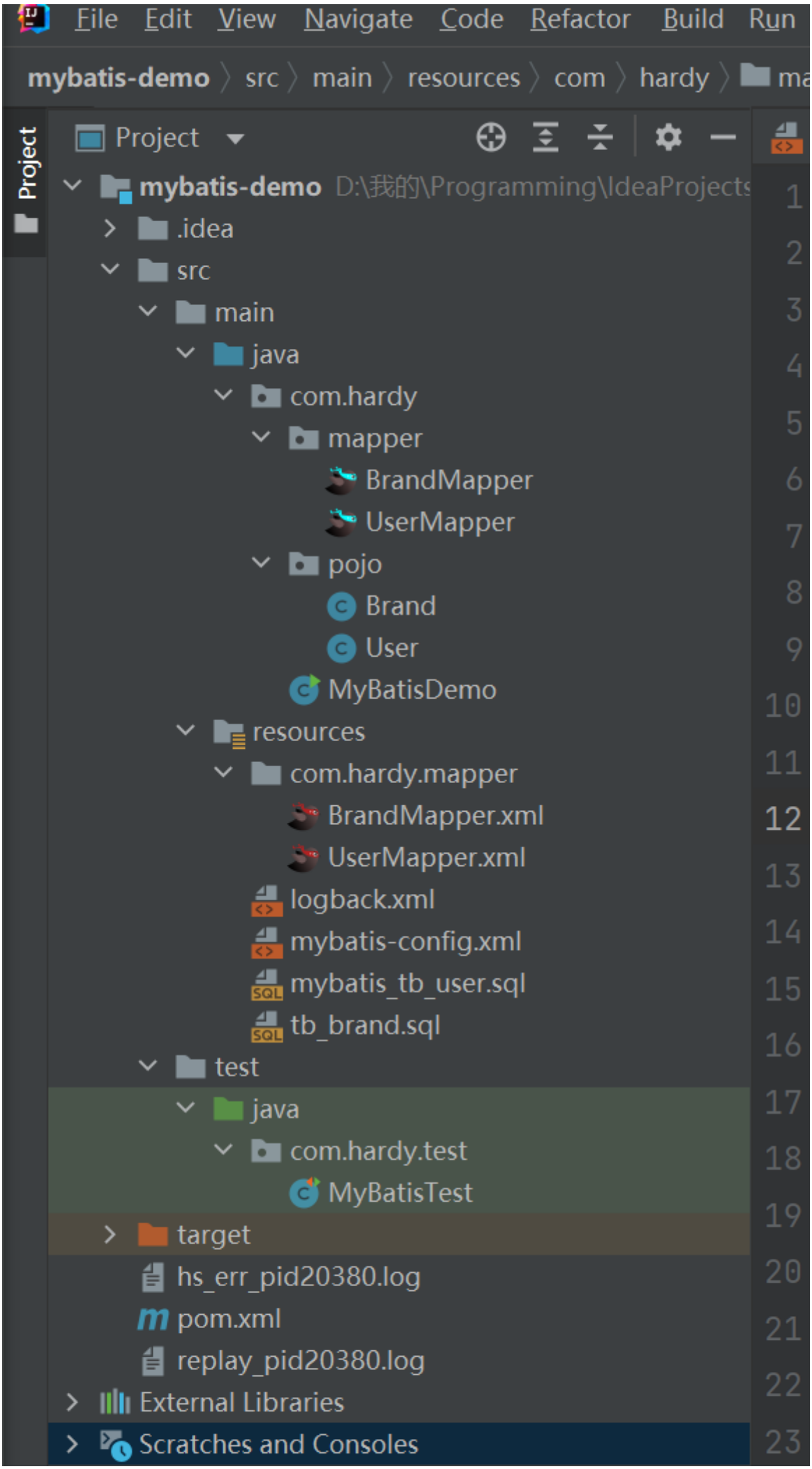
指的是👉 **mybatis_config.xml** 这个文件

- 该文件有多种属性（官网查看）
- 通过配置这些属性来使用mybatis
- 配置各个标签时，需要按照前后顺序进行配置

1.9.4 配置文件完成增删查改

1. 使用myabtis配置文件方式

项目结构


File Edit View Navigate Code Refactor Build Run
mybatis-demo src main resources com hardy
Project
mybatis-demo D:\我的\Programming\IdeaProjects
.idea
src
main
java
com.hardy
mapper
BrandMapper
UserMapper
pojo
Brand
User
MyBatisDemo
resources
com.hardy.mapper
BrandMapper.xml
UserMapper.xml
logback.xml
mybatis-config.xml
mybatis_tb_user.sql
tb_brand.sql
test
java
com.hardy.test
MyBatisTest
target
hs_err_pid20380.log
pom.xml
replay_pid20380.log
External Libraries
Scratches and Consoles

(1) 配置mybatis_config.xml

1. 配置要连接的数据库

```
<dataSource type="POOLED">
          <!-- 数据库连接信息 -->
          <property name="driver" value="com.mysql.jdbc.Driver"/>
          <property name="url" value="jdbc:mysql:///mybatis?useSSL=false"/>
          <property name="username" value="root"/>
          <property name="password" value="123789"/>
</dataSource>
```

2. 添加标签，标签内容：返回类型resultType对应的类所在的全限定包名

```
<typeAliases>
        <package name="com.hardy.pojo"/>
</typeAliases>
```

3. 配置mappers标签

```
<mappers>
        <!--加载sql映射文件-->
        <mapper resource="com/hardy/mapper/UserMapper.xml"/>
</mappers>
```

(2) 创建SQL映射文件UserMapper.xml

SQL映射文件即描述SQL语句的文件，将该文件存放在resources包内对应接口类（对应接口类**UserMapper.java**）中。

- **resultType="user"** ------ 当mybatis_config.xml配置了标签时，才可以这样写。否则要写全限定类名。如：**resultType="com.hardy.pojo.User"**

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
        PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
        "http://mybatis.org/dtd/mybatis-3-mapper.dtd">

<!--namespace：名称空间-->
<mapper namespace="com.hardy.mapper.UserMapper">
    <select id="selectAll" resultType="user">
        select * from tb_user;
    </select>
</mapper>
```

MyBatis中SQL语句占位符：

- #{}：会将其替换为?，为了防止SQL注入
- ${}：直接拼接sql，存在sql注入问题
- 使用时机
    - 参数传递时：#{}
    - 表名或列名不固定时：${tableName}，存在sql注入问题

```sql
select * from tb_brand where id = #{id};
// 替换为
select * from tb_brand where id = ?;

select * from ${} where id = #{id};
```

(3) 创建对应接口类

- 使用代理开发时，才需要此操作。
- 该类是一个接口，存放在mapper包中。
- 该接口中有一个方法，方法名就是SQL映射文件中QL语句的id，并保持参数类型和返回值类型一致

```java
package com.hardy.mapper;

import com.hardy.pojo.Brand;
import java.util.List;

public interface BrandMapper {
    List<Brand> selectAll();

    List<Brand> selectByCondition(@Param("status") int status,
                                  @Param("companyName") String companyName,
                                  @Param("brandName") String brandName
                                  );
}
```

(4) 创建DQL语句返回值类型

即新建一个类。如User类。存放于pojo包中

```
package com.hardy.pojo;

public class User {
    private Integer id      ;
    private String username ;
    private String password ;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    @Override
    public String toString() {
        return "User{" +
                "id=" + id +
                ", username='" + username + '\'' +
                ", password='" + password + '\'' +
                '}';
    }
}
```

2. 查询

- 查询所有

```
<select id="selectAll" resultType="user">
    select * from tb_user;
</select>
```

- 条件查询

```
<select id="selectByCondition" resultMap="brandResultMap">
    select *
    from tb_brand
    <where>
        <choose>
            <when test="status != null">
                status = #{status}
            </when>
            <when test="companyName != null and companyName != ''">
                companyName = #{companyName}
            </when>
            <when test="brandName != null and brandName != ''">
                brandName = #{conmpanyName}
            </when>
        </choose>
    </where>
</select>
```

3. 添加

```
    <insert id="add">
        insert into tb_brand(company_name, brand_name, ordered, description,
status)
        values(#{companyName},#{brandName},#{ordered},#{description},#{status})
    </insert>
```

记得提交事务

```
sqlSession.commit();
// 或者设置自动提交事务
SqlSession sqlSession = sqlSessionFactory.openSession(true);
```

主键返回

插入后返回主键

```
<insert id="add" useGeneratedKeys="true" keyProperty="id">
    insert into tb_brand(company_name, brand_name, ordered, description, status)
    values(#{companyName},#{brandName},#{ordered},#{description},#{status})
</insert>
```

```
Integer id = brand.getId();
```

4. 修改

- 修改所有字段

```
<update id="update">
    update tb_brand
    set
        brand_name = #{brandName},
        company_name = #{companyName},
        ordered = #{ordered},
        status = #{status},
        description = #{description}
    where id = #{id}
</update>
```

- 修改动态字段

  要修改的字段不可预知

```
<update id="update">
    update tb_brand
    <set>
        <if test="status != null">
            status = #{status}
        </if>
        <if test="companyName != null and companyName != ''">
            company_name = #{companyName}
        </if>
        <if test="brandName != null and brandName != ''">
            brand_name = #{brandName}
        </if>
        <if test="ordered != null">
            ordered = #{ordered}
        </if> <if test="description != null and description != ''">
        description = #{description}
        </if>
    </set>
    where id = #{id}

</update>
```

5. 删除

- 单个删除

```
<delete id="deleteById">
    delete from tb_brand
    where id = #{id};
</delete>
```

- 批量删除

```
<!--
        mybatis会将数组参数，封装为一个Map集合
```

```
            * 默认：array=数组(key=array, value=参数数组)
            * 使用@Param注解改变map集合的默认key名称
        -->
    <delete id="deleteByIds">
        delete from tb_brand
        where id
        in
            <foreach collection="array" item="id" separator="," open="("
close=")">
                #{id}
            </foreach>
        ;
    </delete>
```

```
int deleteByIds(Integer[] ids);
```

```
Integer[] ids = {9,10};
brandMapper.deleteByIds(ids);
```

1.9.5 动态SQL

定义：SQL语句会随着用户的输入或外部条件的变化而变化，称之为动态SQL

1. 动态多条件查询

if：用于判断参数是否有值，使用test属性进行条件判断

- 存在问题：第一个条件不需要逻辑运算符
- 解决方案
  - 使用恒等式
  - 标签替换where关键字

```
<!--当第一个if不满足第二个if满足时会出现"where and"。当所有if都不满足会出现"where;"结尾
    产生SQL语法错误
-->
<select id="selectByCondition" resultMap="brandResultMap">
        select *
        from tb_brand
        where
              <if test="status != null">
                  and status = #{status}
              </if>

              <if test="companyName != null and companyName != ''">
                  and company_name like #{companyName}
              </if>

              <if test="brandName != null and brandName != ''">
                  and brand_name like #{brandName};
              </if>
</select>
```

```
<!--解决方案1：使用恒等式-->
<select id="selectByCondition" resultMap="brandResultMap">
        select *
        from tb_brand
        where 1=1
              <if test="status != null">
                  and status = #{status}
              </if>

              <if test="companyName != null and companyName != ''">
                  and company_name like #{companyName}
              </if>
              <if test="brandName != null and brandName != ''">
                  and brand_name like #{brandName};
              </if>

    </select>


<!--解决方案2：使用<where>标签-->
<select id="selectByCondition" resultMap="brandResultMap">
        select *
        from tb_brand
            <where>
                <if test="status != null">
                    status = #{status}
                </if>

                <if test="companyName != null and companyName != ''">
                    company_name like #{companyName}
                </if>

                <if test="brandName != null and brandName != ''">
                    brand_name like #{brandName};
                </if>
            </where>
</select>
```

2. 动态单条件查询

只有一个条件或零个有效（使用默认的）

- 类似于switch
- 类似于switch
- 类似于default

```
<select id="selectByCondition" resultMap="brandResultMap">
        select *
        from tb_brand
           where
                <choose>
                    <when test="status != null">
                        status = #{status}
```

```
                    </when>

                    <when test="companyName != null and companyName != ''">
                        companyName = #{companyName}
                    </when>

                    <when test="brandName != null and brandName != ''">
                        brandName = #{conmpanyName}
                    </when>

                    <otherwise>
                        1 = 1
                    </otherwise>
                </choose>
    </select>

<!-- 等价于 -->
<select id="selectByCondition" resultMap="brandResultMap">
        select *
        from tb_brand
            <where>
                <choose>
                    <when test="status != null">
                        status = #{status}
                    </when>

                    <when test="companyName != null and companyName != ''">
                        companyName = #{companyName}
                    </when>

                    <when test="brandName != null and brandName != ''">
                        brandName = #{conmpanyName}
                    </when>

                </choose>
            </where>
    </select>
```

1.9.6 Mybatis参数传递

Mybatis接口方法中可以接受各种各样的参数，Mybatis底层对于这些参数进行不同的封装处理方式

MyBatis提供了**ParamNameResolver**类进行参数封装

1. 单个参数

- **POJO类型**：直接使用，属性名 和 参数占位符名称 一致
- **Map集合**：直接使用，键名 和 参数占位符名称 一致
- **Collection**：封装为Map集合，可以使用@Param注解，替换Map集合中默认的arg键名

  map.put("arg0", collection集合);

map.put("collection", collection集合);

- **List**：封装为Map集合，可以使用@Param注解，替换Map集合中默认的arg键名

  map.put("arg0", list集合);
  map.put("collection", list集合);
  map.put("list", list集合);

- **Array**：封装为Map集合，可以使用@Param注解，替换Map集合中默认的arg键名

  map.put("arg0", 数组);
  map.put("array", 数组);

- **其他**

建议：使用@Param注解来修改Map集合中默认的键名，并使用修改后的名称来获取键值，这样可读性更高

2. 多个参数

- 封装为Map集合

```
User select(@Param("username") String username, @Param("password") String
password);
```

- Mybatis底层做法

```
// 若不添加注解
map.put("arg0", 参数值1)
map.put("param1", 参数值1)
map.put("arg1", 参数值2)
map.put("param2", 参数值2)

// 此时可以使用arg0或param1指代参数。不建议，造成指代不明
<select id="selectAll" resultType="user">
    select * from tb_brand;
    where
        username = #{arg0}  // or "param1"
    and password = #{arg1}  // or "param2"
</select>

// 可以使用@Param注解，替换Map集合中默认的arg键名
User select(@Param("username") String username, @Param("password") String
password);
map.put("arg0", 参数值1)
map.put("username", 参数值1)
map.put("arg1", 参数值2)
map.put("password", 参数值2)
```

1.9.7 注解完成增删查改

- 查询：@Select
- 添加：@Insert
- 修改：@Update
- 删除：@Delete

> 注解完成简单功能
>
> 配置文件完成复杂功能

```
@Select("select * from tb_user where id = #{id}")
public User selectById(int id)
// 无需再写XML映射文件
```

1.9.8 ResultMap

```
public interface BrandMapper {
    @Select("select * from tb_brand")
    @ResultMap("brandResultMap")
    List<Brand> selectAll();

}
```

```
<mapper namespace="com.hardy.mapper.BrandMapper">
    <resultMap id="brandResultMap" type="brand">
        <result column="brand_name" property="brandName"></result>
        <result column="company_name" property="companyName"></result>
    </resultMap>
</mapper>
```

二、前端

2.1 HTML

2.1.1 HTML介绍

- HTML是一门语言，所有的网页都是用HTML这门语言编写的
- HTML，**HyperText Markup Language**：超文本标记语言
  - 超文本：超越了文本的限制，比普通文本更强大。除了文字信息，还可以定义图片、音频、视频等内容
  - 标记语言：**由标签构成的语言**
- HTML运行在浏览器上， HTML标签由浏览器来解析
- HTML标签都是预定义好的
- **W3C标准**：网页主要由三部分组成

- 结构：HTML
- 表现：CSS
- 行为：JavaScript

2.1.2 HTML结构标签

一个HTML页面最基本的结构

```
<html>
    <head>
        <title></title>
    </head>

    <body>
    </body>
</html>
```

2.2 CSS

2.2.1 HTML介绍

- CSS是一门语言，用于控制网页表现
- CSS（**Cascading Style Sheet**）：层叠样式表

2.2.2 CSS导入方式

1. 内联样式

在标签内部使用**style**属性， 属性值是css属性键值对

```
<div style="color:red">Hello CSS</div>
```

2. 内部样式

定义style标签，在标签内部定义css样式

```
<style type="text/css">
    div{
        color:red;
    }
</style>
```

3. 外部样式

定义link标签，引入外部css文件

```
<link rel="stylesheet" href="demo.css">
```

2.2.3 CSS选择器

CSS选择器有很多。

- 规则：**谁选择范围越小，谁（选择器）就生效**
- **id属性值不能重复，class属性值可以重复**

1. 元素选择器

> 元素名称{ color : red; }

```
div{color:red;}
```

2. id选择器

> #id属性值 { color: red; }

```
#name{color:red;}

<div id="name">hello css</div>
```

3. 类选择器

> .class属性值{color:red;}

```
.cls{color:red;}

<div class="cls">hello css</div>
```

2.2.4 CSS属性

查文档

2.3 JavaScript

2.3.1 JavaScript简介

- 是一门跨平台、面向对象的脚本语言。用来控制网页行为，它能使网页交互
- 与Java是完全不同的语言，不论是概念还是设计。但是基础语法类似。
- 简称**JS**
- ECMAScript 6（ES6）是最新的JavaScript版本（发布于2015年）

2.3.2 JavaScript引入方式

- 内部脚本：将JS代码定义在HTML页面中
    - 使用标签
    - 可以在HTML文档的任意地方，放置任意数量的
    - 一般把脚本置于元素的底部（内），可改善显示速度，因为脚本执行会拖慢显示
- 外部脚本：将JS代码定义在外部S文件中，然后引入到HTML页面中
    - 外部脚本不能包含标签
    - 标签不能自闭合

2.3.3 JavaScript基础语法

1. 书写语法

- 区分大小写
- 结尾分号可有可无
- 注释同Java
- 大括号代表代码块

2. 输出语句

- 使用**window.alert()** 写入警告框
- 使用**document.write()** 写入HTML页面输出
- 使用**console.log()** 写入浏览器控制台

3. 变量

- 使用**var**关键字声明变量
- JavaScript是弱类型语言，变量可以存放不同类型的值
- 命名规则
  - 字母、数字、下划线、美元符号$组成
  - 不能以数字开头
- ECMAScript6新增了**let**关键字来定义变量。与var类似，但所声明的变量，只在let关键字所在代码块内有效，且不允许重复
- ECMAScript6新增了**const**关键字，用来声明一个只读的常量。一旦声明，常量的值就不能改变

4. 数据类型

分为原始类型和引用类型

- 使用**typeof**可以获取数据类型

```
typeof a
```

5种**原始类型**：

- number：数字（整数、小数、NaN）
- string：字符、字符串、单双引都可以
- boolean
- null：对象为空
- undefined：当声明的变量未初始化时，该变量的默认值是undefined

5. 运算符

- == 与 ===
  - == ：判断类型是否一样， 若不一样，则进行**类型转换**，再去比较其值
  - ===：全等于。判断类型是否一样，若不一样，直接返回false，类型一样则继续比较。
- 其他运算符与Java一致

6. 类型转换

- 其他类型转为number
  - string：按照字面值转为数字，如果字面值不是数字，则转为NaN（Not a Number）
  - boolean： true=1，false=0
- 其他类型转为boolean
  - number：0与NaN = false，其他=true
  - string：空字符串=false，其他=true
  - null：false
  - undefined：false

7. 函数

- 使用**function**关键字定义

```
function functionName(para1, para2...) {
    ...
}
```

- 形参不需要类型，因为JS是弱类型的
- 返回值也不需要定义类型，直接使用return返回即可

2.3.4 JavaScript常用对象

1. Array

定义：

```
var 变量名 = new Array(元素列表);  // var arr = new Array(1,2,3);
var 变量名 = [元素列表];           // var arr = [1,2,3];
```

- js数组类似于Java集合，长度，类型可变

属性：

- arr.**length**：长度

方法：

- arr.**push(para)**：将para加到末尾
- arr.**splice(para2)**：从para1开始删除para2个元素

2. String

定义：

```
var 变量名 = new String(s);  // var str = new String("hello");
var 变量名 = s;              // var str = "hello";
```

属性：

- **length**：字符串长度

方法：

- **charAt()**
- **IndexOf()**
- **trim()**：去除字符串首尾空字符

3. 自定义对象

```
var 对象名称 = {
    属性1：属性值1,
    属性2：属性值2,
    ...
    函数名称：function(形参列表){},
    ...
}
```

2.3.5 BOM

- **B**rowser **O**bject **M**odel，浏览器对象模型
- JavaScript将浏览器的各个组成部分封装为对象
- 组成：
    - Windiow：浏览器窗口对象
    - Navigator：浏览器对象
    - Screen：屏幕对象
    - History：历史记录对象
    - Location：地址栏对象

1. Window

获取：直接使用window，其中window可以省略

```
window.alert("abc");
alert("abc");
```

属性：获取其他BOM对象

- history
- Navigator
- Screen
- location

方法：

- **alert**()：显示一段消息和一个确认按钮的警告框
- **confirm**()：显示带有一段消息以及确认按钮和取消按钮的对话框。有一个返回值，**确定返回true，取消返回false**
- **setInterval**(function, 毫秒值)：按照指定的周期（毫秒）来调用函数或计算表达式
- **setTimeout**(function, 毫秒值)：在指定的毫秒数后调用函数或计算表达式

2. History

方法：

- **back**()：加载history列表的前一个URL
- **forward**()：加载history列表的下一个URL

3. Location

属性：

- **href**：设置或返回完整的URL

2.3.6 DOM

- **D**ocument **O**bject **M**odel， 文档对象模型
- 将标记语言的各个组成部分封装为对象
  - Document：整个文档对象
  - Element：元素对象（标签对象）
  - Attribute：属性对象
  - Text：文本对象
  - Comment：注释对象
- JavaScript通过DOM，就能够对HTML进行操作了

2.3.7 事件监听

事件：HTML事件是发生在HTML元素上的“事情”。比如：

- 按钮被点击
- 鼠标移动到元素之上
- 按下键盘按键

事件监听：JavaScript可以在事件被侦测到时**执行代码**

1. 事件绑定

- 通过HTML标签中的事件属性进行绑定

```
<input type="button" onclcik="on()">   <!-- button被点击就调用on()方法 -->
```

- 通过DOM元素属性绑定

```
<input type="button" id="btn">

document.getElementById("btn").onclick = function(){...};
```

2. 常见事件

查看文档

2.3.8 正则表达式

- 概念：定义了字符串组成的规则
- 定义：
  - 直接量：不加引号

    ```
    var reg = /^\\w{6,12}$/;
    ```

  - 创建RegExp对象

    ```
    var reg = new RedExp("^\\w{6,12}$");
    ```

- 方法
  - **test**(str)：判断指定字符串是否符合规则

- 语法

字符	含义
^	开始
$	结束
[]	某个范围内的单个字符，e.g [0-9]
.	单个任意字符
\w	单个字母、数字、下划线字符
\d	单个数字字符
+	至少一个
*	0个或多个
?	0个或一个
{x}	x个
{m, }	至少m个
{m, n}	至少m个，最多n个

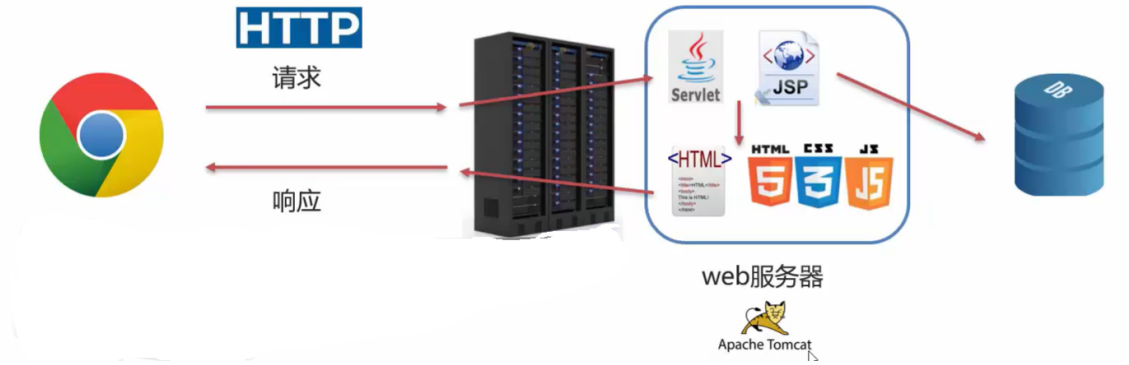
三、Web核心

- Web：全球广域网，也称万维网（www），能够通过浏览器访问的网站

JavaWeb技术栈：

- **B/S架构**

- Browser/Server，浏览器/服务器架构模式。
- 特点：客户端只需要浏览器，应用程序的逻辑和数据都存储在服务器。
- 好处：易于维护升级



- 静态资源：HTML、CSS、JavaScript、图片等。负责页面展现
- 动态资源：Servlet、JSP等， 负责逻辑处理
- 数据库：负责存储数据
- HTTP协议：定义通信协议
- Web服务器：负责解析HTTP协议，解析请求数据，并发送响应数据

3.1 HTTP

- **H**yper**T**ext **T**ransfer **P**rotocol，超文本传输协议，规定了浏览器于服务器之间数据传输的规则
- 特点：
  - 基于TCP
  - 基于请求-响应模型：一次请求对应一次响应
  - 无状态：对于事务处理没有记忆能力。每次请求-响应都是独立的
    - 缺点：多次请求间不能共享数据。Java使用会话技术（Cookie、Session）来解决这个问题
    - 优点：速度快

3.1.1 Request请求数据格式

```
POST / HTTP/1.1
Host: www.itcast.cn
Connection: keep-alive
Cache-Control: max-age=0 Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 Chrome/91.0.4472.106

username=superbaby&password=123456
```

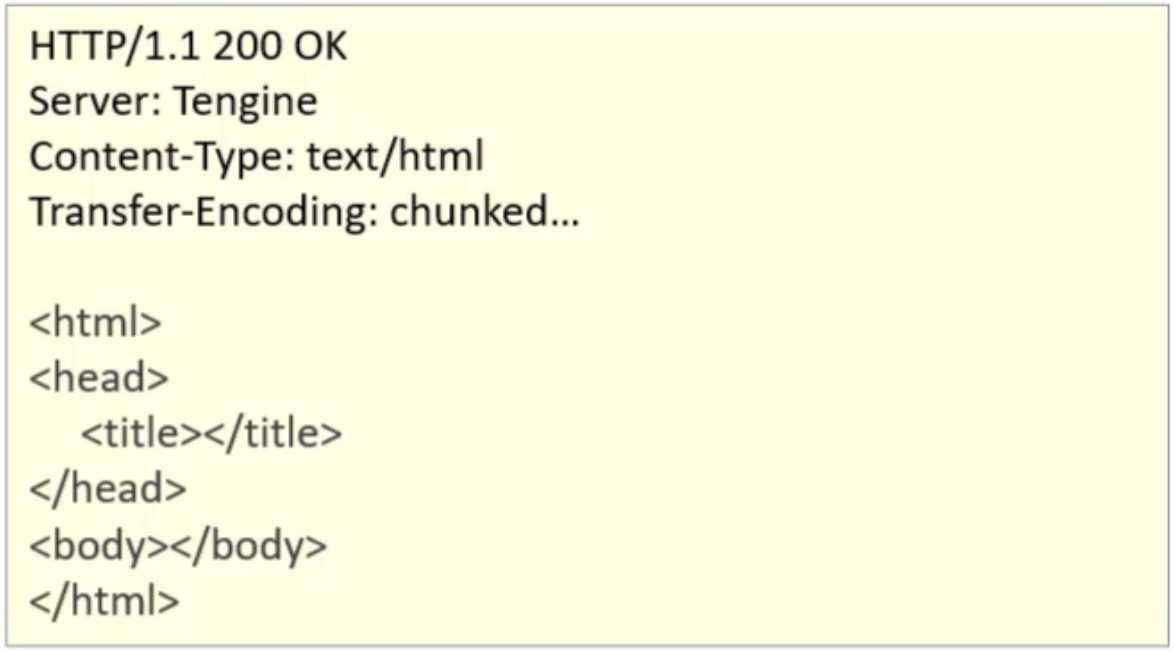
请求数据分为三个部分

- **请求行**：请求数据的第一行。其中GET表示请求方式，/表示请求资源路径，HTTP/1.1表示协议版本
- **请求头**：第二行开始，格式为key-value
- **请求体**：POST请求的最后一部分，存放请求数据

GET于POST的区别：

- GET的请求参数在请求行中，没有请求体。POST请求参数在请求体中
- GET请求参数大小有限制，POST没有

3.1.2 Response响应数据格式



响应数据分为3个部分：

- **响应行**：第一行。HTTP/1.1表示协议版本，200表示状态码，OK表示状态码描述
- **响应头**：第二行开始，格式为key:value
- **响应体**：最后一部分，存放响应数据

常见的HTTP响应头：

- Content-Tpye：表示该响应内容的类型，如：text/html，image/jepg
- Content-Length：表示该响应内容的长度 （字节数）
- Content-Encoding：表示该响应压缩算法，如：gzip
- Cache-Control：指示客户端应如何响应缓存，如：max-age=300，表示可以最多缓存300秒

3.2 Tomcat

3.2.1 Web服务器

Web服务器是一个应用程序（软件），对HTTP协议的操作进行封装，使得程序员不必直接对协议进行操作，让Web开发更加便捷。主要功能是：提供网上信息浏览服务

常见Web服务器

- tomcat
- jetty
- WebLogic
- WebSphere

Tomcat

- 是一个开源免费的轻量级Web服务器，支持Servlet / JSP 等少量JavaEE规范
    - JavaEE：值Java企业级开发的技术规范总和。包含13项技术规范：JDBC、JNDI、EJB、RMI、**JSP**、**Servlet**、XML、JMS、Java IDL，JTS、JTA、JavaMail、JAF
- Tomcat也被称为Web容器、Servlet容器。Servlet需要依赖于Tomcat才能运行

3.2.2 基本使用

下载、安装、卸载、启动、关闭略

- 控制台中文乱码：修改conf/logging.properties文件

> java.util.logging.ConsoleHandler.encoding = GBK

1. 配置

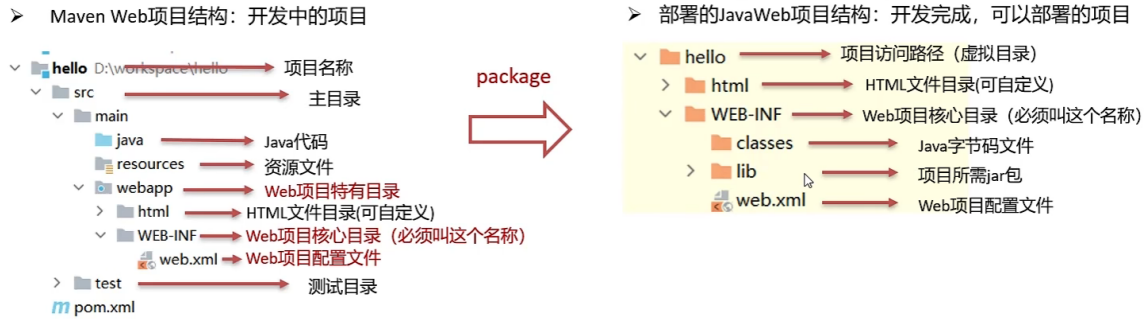
1. 修改启动端口号：conf/server.xml
    - 将端口修改为80
    - HTTP协议默认端口号为80，如果将Tomcat端口号改为80，则将来访问Tomcat时，将不用输入端口号
2. 启动时可能出现的问题
    - 端口号冲突：找到对应的程序，将其关闭
    - 启动窗口一闪而过：检查JAVA_HOME环境变量是否正确配置

2. 部署项目

- 将项目放置到webapps目录下，即完成部署
- 一般JavaWeb项目会被打成war包，然后将war包放到webapps目录下，Tomcat会自动解压缩war文件

3.2.3 IDEA中创建Maven Web项目

1. Web项目结构



- 编译后的字节码文件和resources的资源文件，放到WEB-INF的classes目录下
- pom.xml中依赖的坐标对应的jar包，放入WEB-INF下的lib目录下

3.2.4 IDEA中使用Tomcat

1. 集成本地Tomcat

1. 点击Configuration（在三角形运行按钮旁边）
2. 点击+号，选择TomEE里的Local
3. 在Application server里选择本地Tomcat，选最顶层目录即可
4. 将端口号改成和Tomcat配置的一样
5. 点击OK

2. Tomcat Maven插件

1. 在pom.xml里添加Tomcat插件

```
<!--注意这里是在<plugin>标签下添加-->
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.tomcat.maven</groupId>
      <artifactId>tomcat7-maven-plugin</artifactId>
      <version>2.2</version>
    </plugin>
  </plugins>
</build>
```

2. 使用Maven Helper插件快速启动项目，选中项目，右键--->Run Maven---->tomcat7:run

3.3 Servlet

- Servlet是Java提供的一门**动态**web资源开发技术
- 不同用户访问网站展示不同的内容
- Servlet是JavaEE规范之一，其实就是一个**接口**，将来我们**需要定义Servlet类实现Servlet接口**，并由web服务器运行Servlet

3.3.1 Servlet快速入门

1. 创建web项目，导入Servlet坐标

```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>3.1.0</version>
  <scope>provided</scope>    <!--此标签必须加，provided表示使用范围为编译、测试，运行时不起作用，因为Tomcat里已集成，再使用会造成冲突-->
</dependency>
```

2. 创建：定义一个类实现Servlet接口，并重写接口中所有方法 [，并在Service方法中输入一句话]

```
@WebServlet("/demo1")
public class ServletDemo1 implements Servlet {
    @Override
    public void init(ServletConfig servletConfig) throws ServletException

    @Override
    public ServletConfig getServletConfig() {
        return null;
    }

    @Override
    public void service(ServletRequest servletRequest, ServletResponse servletResponse) throws ServletException, IOException {
        System.out.println("servlet hello world~");
    }

    @Override
    public String getServletInfo() {
        return null;
    }

    @Override
    public void destroy() {}
}
```

3. 配置：在类上使用**@WebServlet**注解，配置该Servlet的访问路径

```
@WebServlet("/demo1")
```

4. 访问：启动Tomcat，浏览器输入URL访问该Servlet

URL: http ://localhost:8080 / servlet-demo01 / demo1

主机地址 项目名称 具体Servlet

3.3.2 Servlet执行流程

- 流程：调用Servlet对象，执行service方法
- Servlet对象由web服务器（Tomcat）创建，Servlet方法由web服务器调用
- 服务器如何知道Servlet中一定有service方法？ 因为我们自定的Servlet实现了Servlet接口，而Servlet接口中有service方法

3.3.3 Servlet生命周期

Servlet运行在Servlet容器（web服务器）中，其生命周期由容器来管理，分为4个阶段：

1. 加载与实例化

默认情况下，当Servlet第一次被访问时，由容器创建Servlet对象

非默认情况：

```
@WebServlet(urlPatterns="/demo1" loadOnStartup=1)
// 负整数：第一次被访问时创建Servlet对象
// 0或正整数：服务器启动时创建Servlet对象，数字越小优先级越高（创建时机提前，将耗时操作提前）
```

2. 初始化

在Servlet实例化之后，容器将调用Servlet的init()方法来初始化这个对象，完成一些如加载配置文件、创建连接等初始化的工作。该方法**只调用一次**

3. 请求处理

每次请求Servlet时，Servlet容器都会调用Servlet的service方法对请求进行处理

4. 服务终止

当需要释放内存或者容器关闭时，容器就会调用destroy()方法完成资源的释放。在destroy()方法调用之后，容器会释放这个Servlet实例，该实例随后会被Java的垃圾收集器所回收

3.3.4 Servlet体系结构

我们将来开发B/S结构的web项目，都是针对HTTP协议，所有我们自定义Servlet，会**继承HttpServlet**

步骤

- 继承HttpServlet
- 重写doGet和doPost方法

HttpServlet原理：

- 获取请求方式，并根据不同请求方式，调用不同的doXxx方法

```
@WebServlet("/demo2")
public class HttpServletDemo extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        System.out.println("get...");
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        System.out.println("post...");
    }
}
```

- HttpServlet中为什么要根据请求方法的不同，调用不同的方法?
    - 因为请求的方式不同
- 如何调用?

3.3.5 Servlet urlPattern配置

- Servlet 要想被访问，必须配置其访问路径（urlPattern）
- 一个Servlet ，可以配置多个urlPattern

```
@WebServlet(urlPatterns={"/demo1", "/demo2"})
```

- urlPattern配置规则
  - 精确匹配
    - 配置路径：@WebServlet("/user/select")
    - 访问路径：localhost:8080/web-demo//user/select
  - 目录匹配
    - 配置路径：@WebServlet("/user/*")
    - 访问路径：localhost:8080/web-demo//user/aaa
      localhost:8080/web-demo//user/bbb
    - 精确匹配优先级高于目录匹配
  - 扩展名匹配
    - 配置路径：@WebServlet("*.do")
    - 访问路径：localhost:8080/web-demo//user/aaa.do
      localhost:8080/web-demo//user/bbb.do
  - 任意匹配
    - 配置路径：@WebServlet("/")
      @WebServlet("/*")
    - 访问路径：localhost:8080/web-demo//hehe
      localhost:8080/web-demo//haha
- 当Servlet配置了"/"，会覆盖掉tomcat中的DefaultServlet，当其他url-pattern都匹配不上时最后会匹配这个Servlet
- 当项目中配置了"/*"，意味着匹配任意访问路径

3.3.6 XML配置Servlet

Servlet从3.0版本后开始支持使用注解配置，3.0之前只支持XML配置文件的配置方式

步骤：

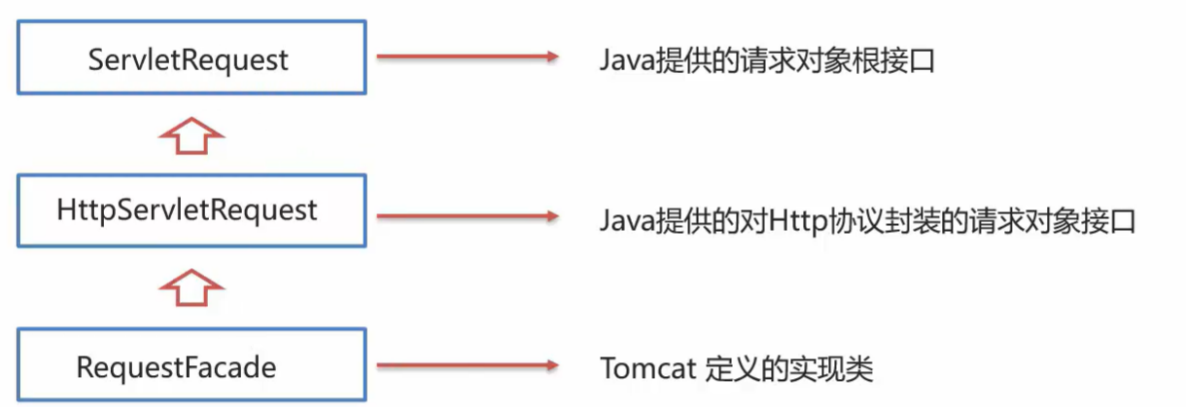
1. 编写Servlet类
2. 在web.xml中配置Servlet

```
<!-- Servlet全类名 -->
<servlet>
  <servlet-name>demo1</servlet-name>
  <servlet-class>com.hardy.web.ServletDemo1</servlet-class>
</servlet>

<servlet-mapping>
  <servlet-name>demo1</servlet-name>
  <url-pattern>/demo1</url-pattern>
</servlet-mapping>
```

3.4 Request与Response

3.4.1 Requst的继承体系



- Tomcat需要解析请求数据，封装为request对象，并且创建request对象传递到service方法中。

3.4.2 Request获取请求数据

1. 获取请求数据方法

请求数据分3个部分：

(1) 请求行

- getMethod()：获取请求方式：GET/POST
- getContextPath()：获取虚拟目录（项目访问路径）：/request-demo
- getRequestURL()：获取URL（统一资源定位符）：http://localhost:8080/request-demo/req1
- getRequestURI()：获取URI（统一资源标识符）：request-demo/req1
- getQueryString()：获取请求参数（GET方式）：usrname=zs&password=123

(2) 请求头

- getHeader(String name)：根据请求头名称，获取值

(3) 请求体

Get方法没有请求体

- ServletInputStream getInputStream()：获取字节输入流
- BufferedReader getReader()：获取字符输入流

2. Request通用方式获取请求数据

通用方式：不分GET和POST

- GET方式

```
String getQueryString()
```

- POST方式

```
BufferedReader getReader()
```

方法：

- ```
  Map<String, String[]> gerParameterMap()  // 获取所有参数的Map集合
  ```
- ```
  String[] getParameterValues(String name)  // 根据名称获取参数值（数组）
  ```
- ```
  String getParameter(String name)  // 根据名称获取参数值（单个值）
  ```

3. 请求参数中文乱码处理

Tomcat8之后，已将GET请求乱码问题解决，设置默认的解码方式为UTF-8

(1) POST

```
req.setCharacterEncoding("UTF-8");
```

(2) GET

```
username = new String(username.getBytes(StandardCharsets.ISO_8859_1),
StandardCharsets.UTF_8)
```

3.4.3 Request请求转发

- 请求转发（forward）：一种在服务器内部的资源跳转方式
- 实现方式：

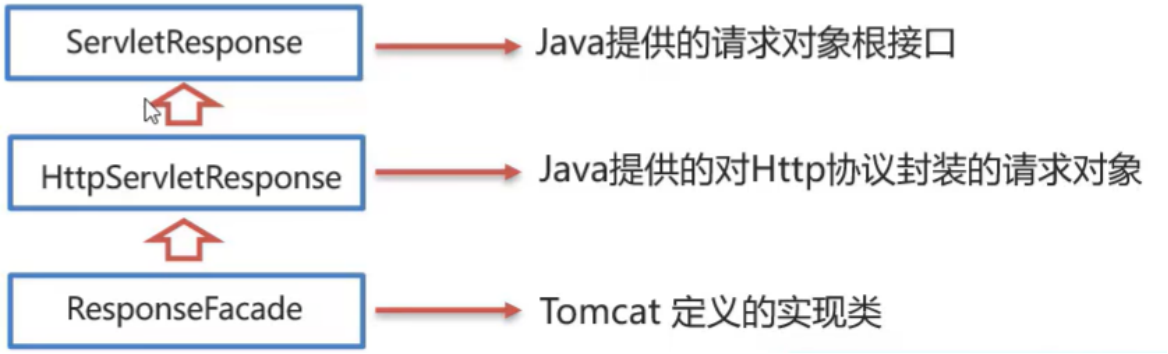
```
req.getRequestDispatcher("资源路径").forward(req, resp);
```

- 请求转发资源间共享数据：使用Request对象
  - void **setAttribute**(String name, Object o)：存储数据到request中
```

    - Object **getAttribute**(String name)：根据key，获取值
    - void **removeAttribute**(String name)：根据key，删除该键值对

- 请求转发特点
    - 浏览器地址不发生变化
    - 只能转发到当前服务器的内部资源
    - 一次请求，可以在转发的资源间使用request共享数据

3.4.4 Response继承体系



3.4.5 Response设置响应数据

响应数据分为3部分

1. 响应行

```
void setStatus(int sc)  // 设置响应状态码
```

2. 响应头

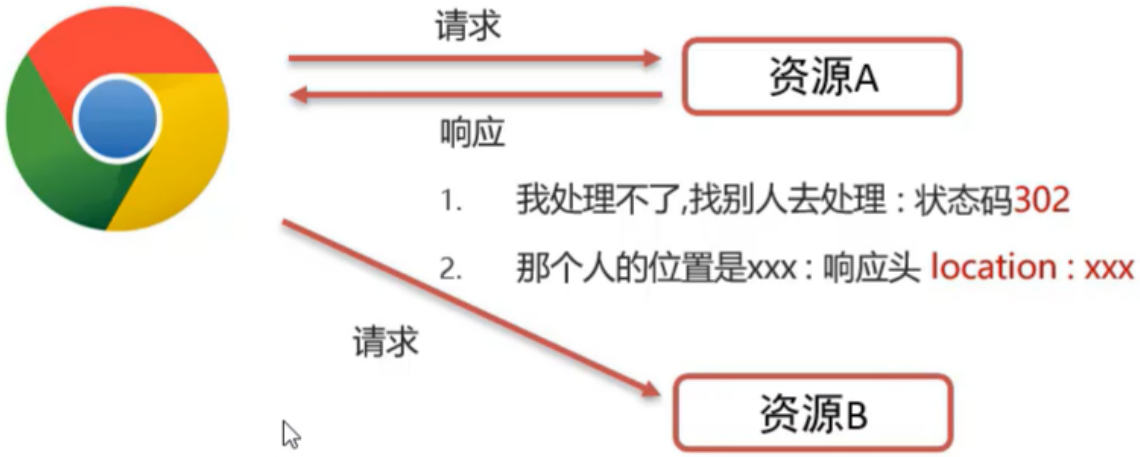
```
void setHeader(String name, String value)  // 设置响应键值对
```

3. 响应体

- ```
  PrintWriter getWriter() // 获取字符输出流
  ```
- ```
  ServletOutputStream getOutputStream() // 获取字节输出流
  ```

3.4.6 Response完成重定向

- 重定向（Redirect）：一种资源跳转方式



- 实现方式

```
resp.setStatus(302);
resp.setHeader("location", "资源B的路径")  // 资源的路径从虚拟路径开始写，
如"/requestandrespose/resp2"

// 简化方式
resp.sendRedirect("资源B的路径");
```

- 重定向特点
  - 浏览器地址路径发送变化
  - 可以重定向到任意位置的资源（服务器内部、外部均可）
  - 两次请求，不能在多个资源使用request共享数据

- 路径问题
  - 浏览器使用：需要加虚拟目录（项目访问路径）
  - 服务端使用：不需要加虚拟目录

- 动态获取虚拟目录

```
String contextPath = request.getContextPath();
```

3.4.7 Response响应字符数据

1. 使用

1. 通过Response对象获取字符输出流

```
PrintWriter writer = resp.getWriter();
```

2. 写数据

```
writer.writer("aaa");  // 不需要手动关闭流，随着响应结束，response对象销毁，由服务器关闭
```

3. 中文问题

```
// 方法一
resp.setCharacterEncoding("UTF-8");

// 方法二
resp.setContentType("text/html;charset=UTF-8");
```

3.4.8 Response响应字节数据

1. 一般做法

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
    FileInputStream inputStream = new FileInputStream("a.png");

    ServletOutputStream os = resp.getOutputStream();

    byte[] buff = new byte[1024];
    int len = 0;
    while( (len = inputStream.read(buff)) != -1){
        os.write(buff, 0, len);
    }
    inputStream.close();
}
```

2. 简化做法

- 使用依赖

```
<dependency>
      <groupId>commons-io</groupId>
      <artifactId>commons-io</artifactId>
      <version>2.6</version>
</dependency>
```

```
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
throws ServletException, IOException {
        FileInputStream inputStream = new FileInputStream("a.png");

        ServletOutputStream os = resp.getOutputStream();

        IOUtils.copy(inputStream, os); // 拷贝流

        inputStream.close();
    }
```

3.5 JSP

3.5.1 JSP简介

- **J**ava **S**erver **P**ages，Java服务端页面
- 一种动态的网页技术，其中既可以定义HTML、JS、CSS等静态内容， 还可以定义Java代码的动态内容
- JSP = Java + HTML
- 作用：简化开发，避免了在Servlet中直接输出HTML标签

3.5.2 JSP快速入门

1. 导入JSP坐标

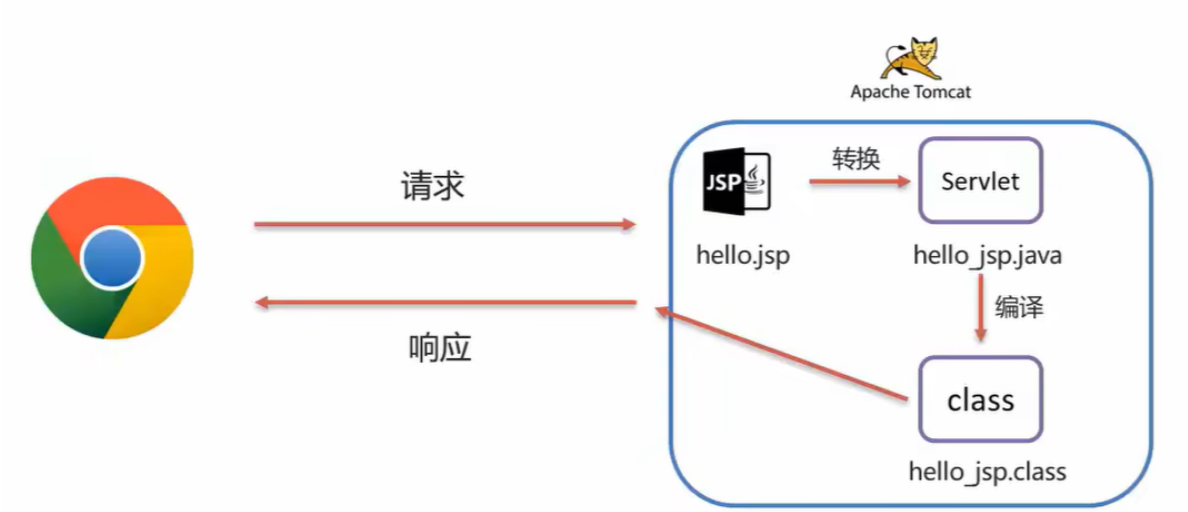
```
<dependency>
  <groupId>javax.servlet.jsp</groupId>
  <artifactId>jsp-api</artifactId>
  <version>2.2</version>
  <scope>provided</scope>
</dependency>
```

2. 创建JSP文件
3. 编写HTML标签和Java代码

```jsp
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>Title</title>
</head>
<body>
    <h1>JSP, hello world!</h1>
<%
    System.out.println("hello jsp~");
%>
</body>
</html>
```

3.5.3 JSP原理

- 本质就是一个Servlet
- JSP在被访问时，由JSP容器（Tomcat）将其转换为Java文件（Servlet），再由JSP容器将其编译，最终对外提供服务的就是这个字节码文件



3.5.4 JSP脚本

- JSP脚本用于在JSP页面定义Java代码
- 分类
    - <% ... %>：内容会直接放到_jspService()方法之中
    - <%= ... %>：内容会放到out.print()，作为out.print()的参数
    - <%! ... %>：内容会直接放到_jspService()方法之外，被类直接包含

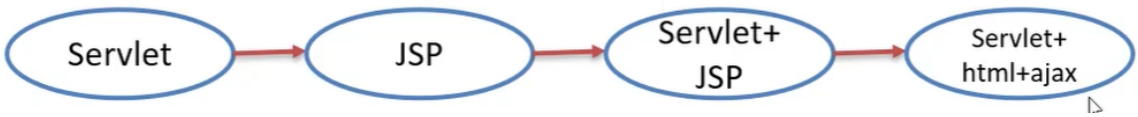
```jsp
<%
    System.out.println("hello jsp~");
    int i = 3;
%>

<%="hello"%>
<%=i%>
<%="hellllllllllo"%>

<%!
    void show(){}
    String name = "zs";
%>
```

3.5.5 JSP缺点与演进

1. 缺点

- 书写麻烦
- 阅读麻烦
- 复杂度高：运行需要依赖于各种环境，JRE，JSP容器，JavaEE
- 占内存和磁盘：JSP会自动生成.java和.class文件
- 调试困难
- 不利于团队合作

2. 演进



- 不要直接在JSP里写Java代码（不直接不是不写，而是不直接写）
- Servlet：逻辑处理，封装数据
- JSP：获取数据，遍历展现数据

3.5.6 EL表达式

- **E**xpression **L**anguage，表达式语言，用于简化JSP页面内的Java代码
- 主要功能：获取数据
- 语法：${expression}

```jsp
${brands}   // 获取域中存储的key为brands的数据
```

1. 使用

- 在Servlet中封装数据

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
    // 1. 准备数据
    List<Brand> brands = new ArrayList<>();
    brands.add(...);
    brands.add(...);
    brands.add(...);

    // 2. 存储到request域中
    req.setAttribute("brands", brands);

    // 3. 转发到el-demo.jsp
    req.getRequestDispatcher("/el-demo.jsp").forward(req, resp);
}
```

- 在JSP就可以使用数据了
- 记得加上 **<%@ page isELIgnored="false" %>**，表示不忽略EL表达式

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ page isELIgnored="false" %>
<html>
<head>
    <title>Title</title>
</head>

<body>

${brands}
</body>
</html>
```

2. 域对象

JavaWeb中的四大域对象：

- page：当前页面有效
- request：当前请求有效
- session：当前会话有效
- application：当前应用有效

EL表达式获取数据，会依次从这四个域中寻找，直到找到

3.5.7 JSTL标签

- **J**sp **S**tandard **T**ag **L**ibrary， JSP标准标签库，使用标签取代JSP页面上的Java代码

标签	描述
<c:out>	用于在JSP中显示数据，就像<%= ... >
<c:set>	用于保存数据
<c:remove>	用于删除数据
<c:catch>	用来处理产生错误的异常状况，并且将错误信息储存起来
<c:if>	与我们在一般程序中用的if一样
<c:choose>	本身只当做<c:when>和<c:otherwise>的父标签
<c:when>	<c:choose>的子标签，用来判断条件是否成立
<c:otherwise>	<c:choose>的子标签，接在<c:when>标签后，当<c:when>标签判断为false时被执行
<c:import>	检索一个绝对或相对 URL，然后将其内容暴露给页面
<c:forEach>	基础迭代标签，接受多种集合类型
<c:forTokens>	根据指定的分隔符来分隔内容并迭代输出
<c:param>	用来给包含或重定向的页面传递参数
<c:redirect>	重定向至一个新的URL.
<c:url>	使用可选的查询参数来创造一个URL

1. JSTL快速入门

1. 导入坐标

```
<dependency>
  <groupId>jstl</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
</dependency>
<dependency>
  <groupId>taglibs</groupId>
  <artifactId>standard</artifactId>
  <version>1.1.2</version>
</dependency>
```

2. 在JSP页面上引入JSTL标签库

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>

<%-- 引入JSTL标签库 --%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<html>
<head>
    <title>Title</title>
</head>
<body>

</body>
</html>
```

3. 使用（一般结合EL表达式使用）

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>

<%-- 引入JSTL标签库 --%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<html>

<head>
    <title>Title</title>
</head>

<body>
    <c:if test="${status == 1}">
        启用
    </c:if>
    <c:if test="${status == 0}">
        禁用
    </c:if>
</body>
</html>
```

2. 常用标签

1. <c:if> 标签

```
<c:if test="表达式">
    ...
</c:if>
```

2. <c:forEach> 标签
    - 相当于for循环
        - items属性：被遍历的容器
        - var属性：遍历产生的临时变量

```
<c:forEach items="${brands}" var="brand">
    <tr align="center">
        <td>${brand.id}</td>
        <td>${brand.brandName}</td>
        <td>${brand.companyName}</td>
        <td>${brand.description}</td>
    </tr>
</c:forEach>
```

3.5.8 MVC模式和三层架构

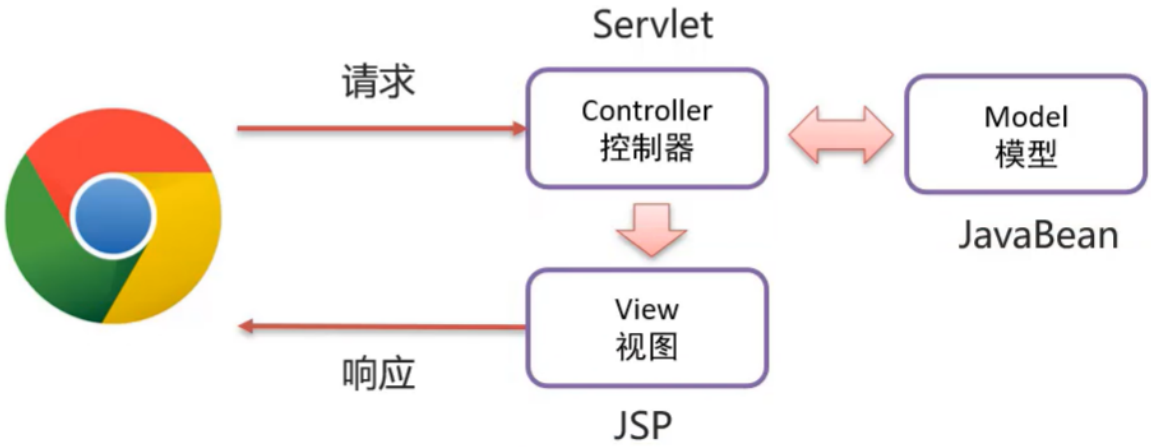
1. MVC模式

MVC模式是一种分层开发的模式，其中：

- M：Model，业务模型，处理业务
- V：View，视图，界面展示
- C：Controller，控制器，处理请求，调用模型和视图

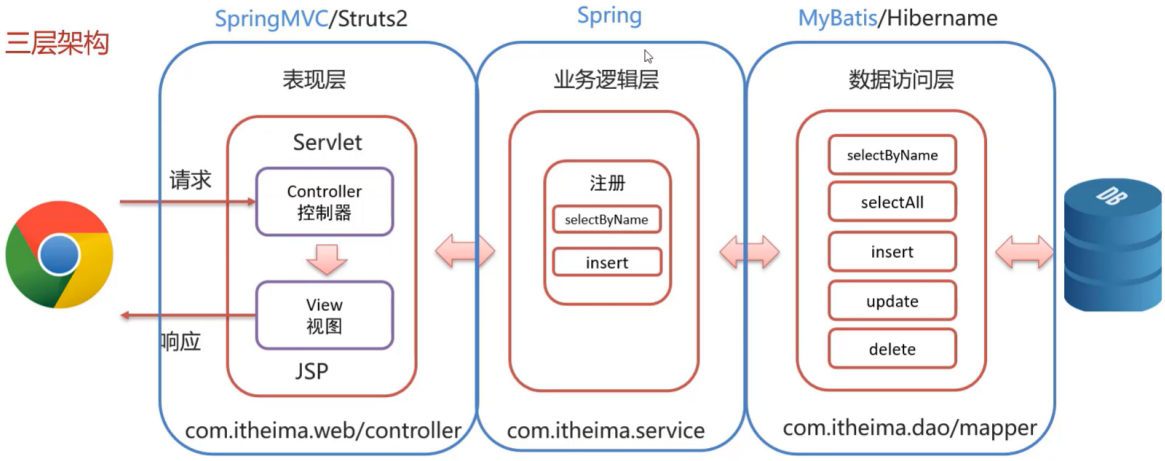
好处：

- 职责单一，互不影响
- 有利于分工协作
- 有利于组件重用

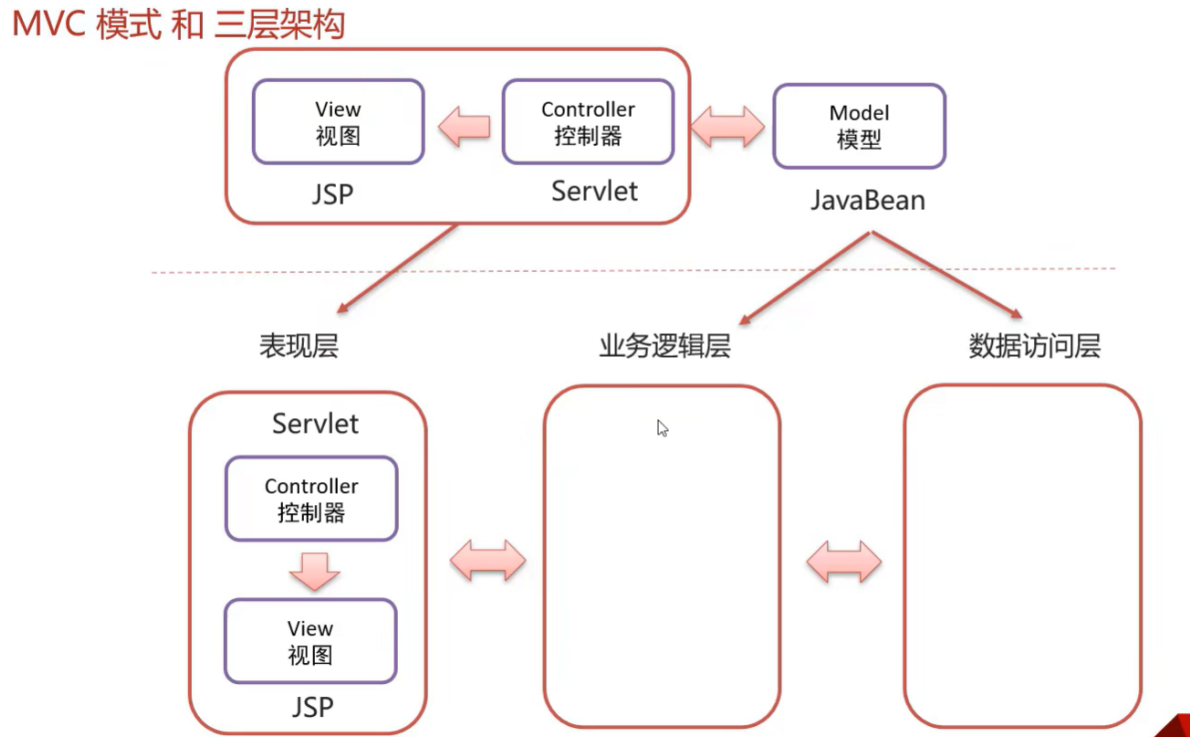


2. 三层架构

- 表现层 Controller：接受请求，封装数据，调用业务逻辑层，响应数据
- 业务逻辑层 Service：对业务逻辑进行封装，组合数据访问层中基本功能，形成复杂的业务逻辑功能
- 数据访问层 Dao：对数据库的CRUD操作



3. 关系



3.6 会话跟踪技术

- 会话：用户打开浏览器，访问web服务器的资源，会话建立，直到有一方断开连接，会话结束。在一次会话中可以包含**多次**请求和响应。
- 会话跟踪：一种维护浏览器状态的方法，服务器需要识别多次请求是否来自同意浏览器，以便在同一次会话的多次请求间**共享数据**。
- HTTP是**无状态**的，每次浏览器想服务器请求时，服务器都会将该请求视为新的请求，因此我们需要会话跟踪技术来实现会话内数据共享。

- 实现方式：
  - 客户端会话跟踪技术：**Cookie**
  - 服务端会话跟踪技术：**Session**

3.6.1 Cookie基本使用

- Cookie：客户端会话技术，将数据保存到客户端，以后每次请求都携带Cookie数据进行访问
- 服务器端视角

1. 发送Cookie

1. 创建Cookie对象，设置数据

```
Cookie cookie = new Cookie("username", "zs");
```

2. 发送Cookie到客户端：使用Response对象

```
response.addCookie(cookie);
```

2. 获取Cookie

1. 获取客户端携带的所有Cookie，使用Request对象

```
Cookie[] cookies = request.getCookies();
```

2. 遍历数组，获取每一个Cookie对象
3. 使用Cookie对象方法获取数据

```
cookie.getName();
cookie.getValue();
```

3.6.2 Cookie原理

1. 原理

- Cookie的实现是基于HTTP协议的
  - 响应头：set-cookie
  - 请求头：cookie

2. 使用细节

1. **存活时间**
   - 默认情况下，Cookie存储在浏览器内存中，当浏览器关闭，内存释放， 则Cookie被销毁
   - **setMaxAge**(int seconds)：设置Cookie存活时间
     - 正数：将Cookie写入浏览器所在电脑的硬盘，持久化存储。到时间自动删除。
     - 负数：默认值。浏览器关闭，则Cookie被销毁
     - 零：删除对应Cookie

2. **Cookie存储中文**
   - Cookie默认不支持存储中文
   - 如需存储，则需要进行转码：URL编码

```
// 发送
String value = "张三";
value = URLEncoder.encode(value, "UTF-8");
response.addCookie(new Cookie("username", value));

// 接收
value = cookie.getValue();
value = URLDecoder.decode(value, "UTF-8");
```