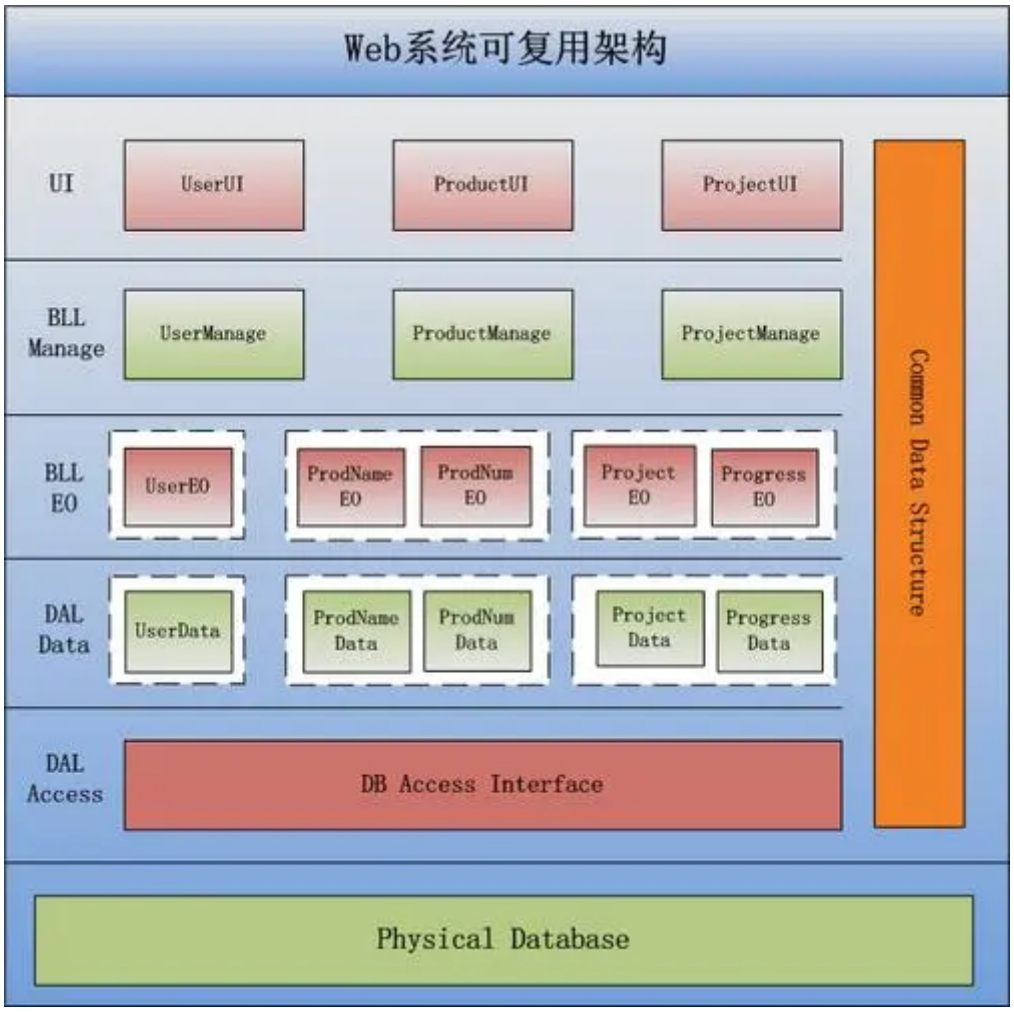
网络编程技术

- Java Web 开发技术
- Websocket 编程
- Socket 编程
- Docker

0x01 Jave Web

1. 组件和架构

- 组件
  - 浏览器
    - Chrome
  - 数据库：静态文件解析、应用服务器
  - 数据库
- **架构**



- UI - User Interface
- BLL - Business Logic Layer
- EO - Entity Object
- DAL - Database Access Layer

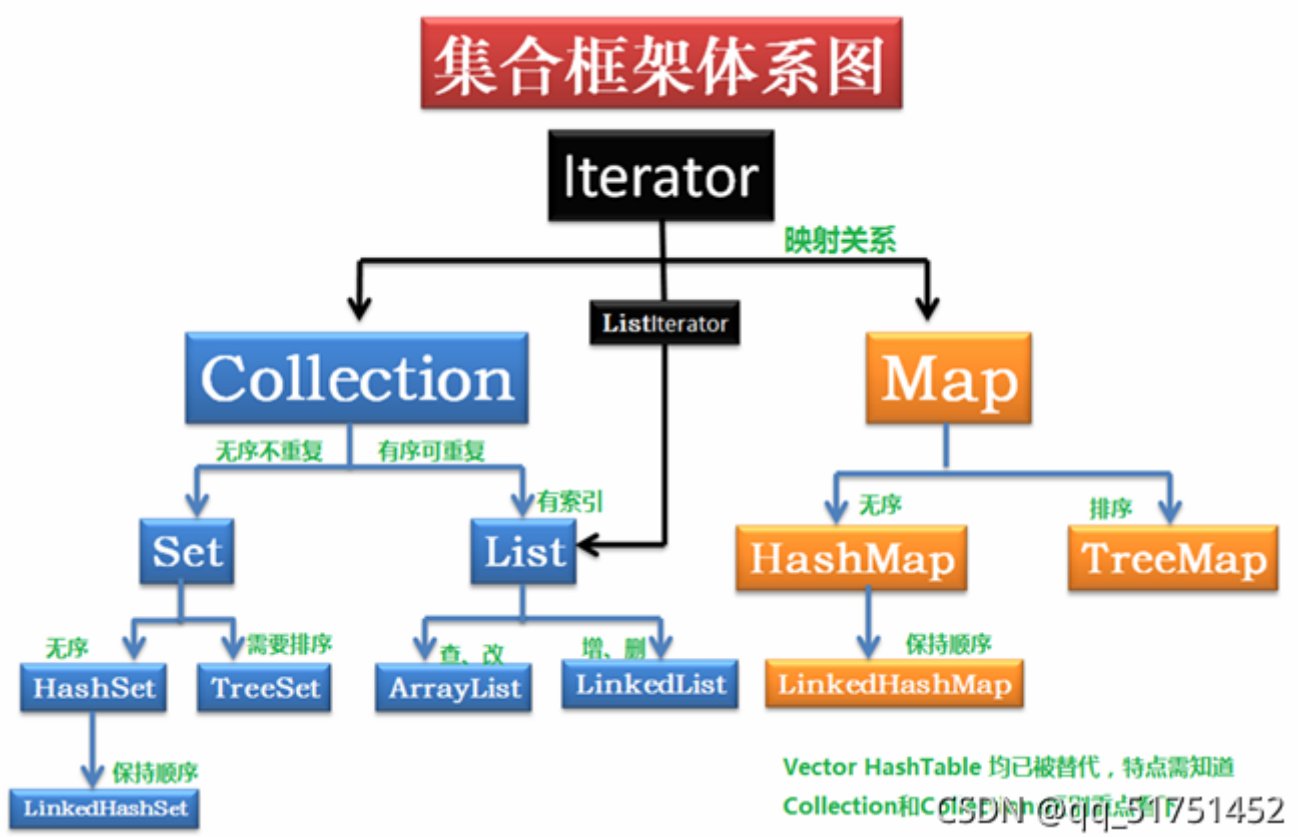
2. DNS、TCP/IP

- 普通顶级域名
- 国家代码顶级域名
- IP及端口号
- 通信方式：P2P、C/S、B/S
- **DNS的优先级**：hosts → 缓存

3. Java 基础

- 历史
  - sun
  - oracle
- 后台定位： `System.out.println(new Throwable().getStackTrace()[0]);`

- 命名规则：驼峰
- 基本数据类型： `int` ， `long` ， `float` ， `double` ， `boolean` ， `String`
- 高级数据类型： `int[]` ， `{Hash}Map` ， `{Hash}Set` ， `{Array}List` ， `JSONObject` ， `JSONArray`
    - `Map` 是**映射**接口，实现类有 `HashMap` 等
    - `Set` 是**集合**接口，实现类有 `HashSet` 等
    - `List` 是数组接口，实现类有 `ArrayList` 等
    - 上述三接口都是 `Collection` 的子接口



- **集合类型方法**
    - `add(元素)` - 添加元素
    - `addAll(集合)` - 并集
    - `remove(元素)` - 删除元素
    - `removeAll(集合)` - 差集
    - `A.retainAll(B)` - 得到A交B
- 类与对象
    - 属性
    - 方法
    - 构造函数
    - 包
    - 继承
    - 接口与实现
- 程序结构：顺序、选择、循环

4. HTTP、HTML

- URI与URL
  - URI和URL
    - URI：一般指要完成的操作。比如，http://localhost/addUser
    - URL：一般指某个入口，比如，http://localhost/admin/
  - URI(统一资源**标识**符)：相当于“接口”，由"URL"或"URN"实现
  - **URL**(统一资源**定位**符)：协议 + `://` + 主机名 (+ : 端口) + 路径 + 参数 + hash

    - 协议除 `http(s)` ，还有 `fpt` 、 `file` 等
    - 主机名可以是 IP 或域名
    - 参数就是请求体(Request)的"Param"，用 `&` 连接
    - hash 用于提供标题快速链接，如锚点 `#`
  - *URN*(统一资源**名称**)：网络上很少用，用名称来表示，比如 ISBN。
- HTTP 请求
  - `GET` - 获取
  - `POST` - 上传
  - `PUT` - （全局）更新
  - * `PATCH` - 局部更新
  - `DELETE` - 删除
  - `HEAD` - 请求头
  - `OPTIONS` - 支持什么请求
  - `TRACE` - 回显（测试）
  - `CONNECT` - 建立隧道
- HTML
  - 定义：

    > HTML（HyperText Markup Language，超文本标记语言）是一种用于创建网页的标记语言。它定义了网页的结构和内容，并使用标签（tag）将文本、图像、链接和其他媒体元素包裹起来，以便在Web浏览器中正确显示和呈现。

  - 注释 - `<!-- 内容 -->`
  - 标签： `<h1>` 、 `<p>` 、表单(`<text>` , `<button>`)、 `<span>` 、 `<img>` 、 `<a>` 、表格(`<tr>` 、 `<td>`)
- CSS
  - 选择器：
    - 元素 - `p`
    - 类 - `.class`
    - ID - `#id`

        - 属性 - `[attr="value"]`
        - 搭配前面的
            - 后代 - 空格
            - 子代 - `>`
            - 相邻兄弟 - `+`
            - 伪类 - `:`
    - 样式
        - 字体大小 - `font-size`
        - 颜色 - `color`

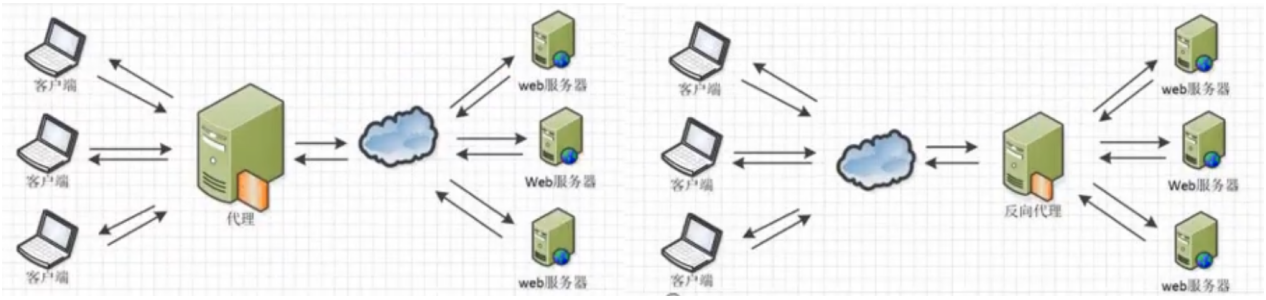
5. 开发环境

- JDK
    - 环境变量
- 数据库
    - 常用 DBMS

DBMS	厂商	默认超级用户名	客户端	端口
MySQL	Oracle Corporation	root	MySQL Shell, MySQL Workbench	3306
Oracle Database	Oracle Corporation	SYS (SYSDBA 角色)	SQL*Plus, Oracle SQL Developer	1521
Microsoft SQL Server	Microsoft Corporation	sa	SQL Server Management Studio	1433
PostgreSQL	The PostgreSQL Global Development Group	postgres	pgAdmin, psql	5432
SQLite	D. Richard Hipp	（无）	SQLite Shell, SQLiteStudio	（无）
MongoDB	MongoDB, Inc.	（无）	MongoDB Shell, MongoDB	27017

DBMS	厂商	默认超级用户名	客户端	端口
			Compass	

    - 主键、外键
        - 主键： `id INT PRIMARY KEY`
        - 外键： `customer_id INT, FOREIGN KEY (customer_id) REFERENCES customers(id)`
    - CRUD(Create, Read, Update, Delete)
- Tomcat
    - 端口：用于标识通信的逻辑终点，指定Tomcat服务器与客户端之间的通信端口。
    - 虚拟主机：在一个物理服务器上通过配置实现多个域名或多个网站共享同一个IP地址的技术
    - 虚拟目录：用于将请求的URL路径映射到实际文件系统中的不同位置，改变URL与文件位置之间的对应关系。
- Nginx
    - 特点：高性能 Web 和反向代理服务 HTTP 代理
    - 功能
        - 反向代理

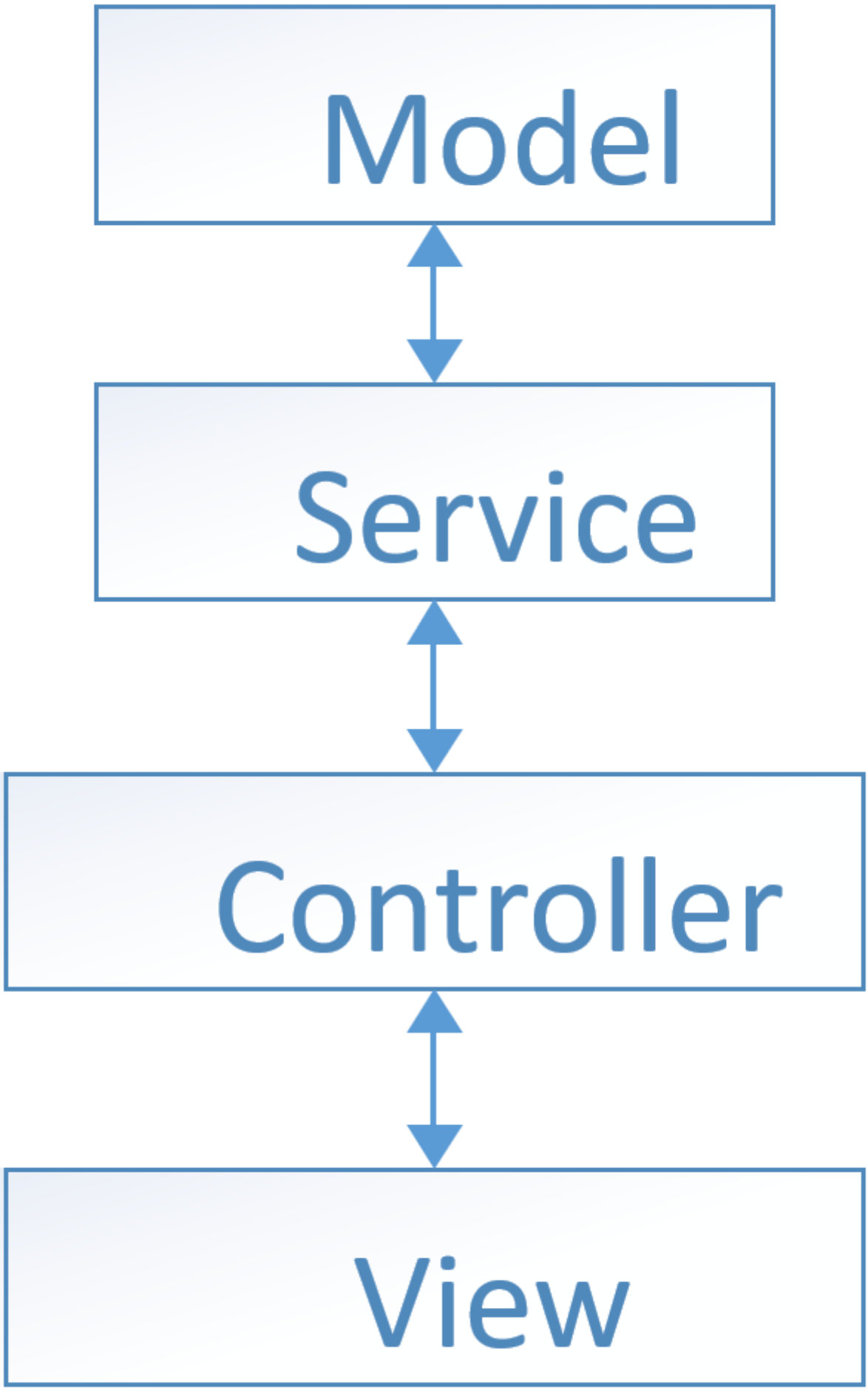


        - 负载均衡：业务集群、将负载重定向到多个 Web 服务器（如 Tomcat）
        - Web缓存：常用于对静态资源文件作缓存，如：js、图片、css
    - 静态资源代理和容器代理的**配置**

```
service {
    listen 80;
    server_name 1;
    location / {
        proxy_pass http://localhost:8888
        // 如果是集群: proxy_pass http://...
    }
    location /sgccfiles { // 静态页面
        alias C:/.../sgccfiles
        index index.html
    }
    location /saasfiles {
        alias C:/...
        index index.html
    }
}
```

- Redis
  提供非经常更新内容的缓存，如：字典数据、登录信息(token)，
  可避免数据库连接高并发
  - 配置
    - Windows: `redis.windows-service.conf`
    - Linux: `/etc/redis.conf`
    - 配置项
      - `requirepass` - 密码
      - `bind` - 绑定在哪个网卡（`0.0.0.0` 代表所有接口都开 redis）
      - `port` - 服务端口

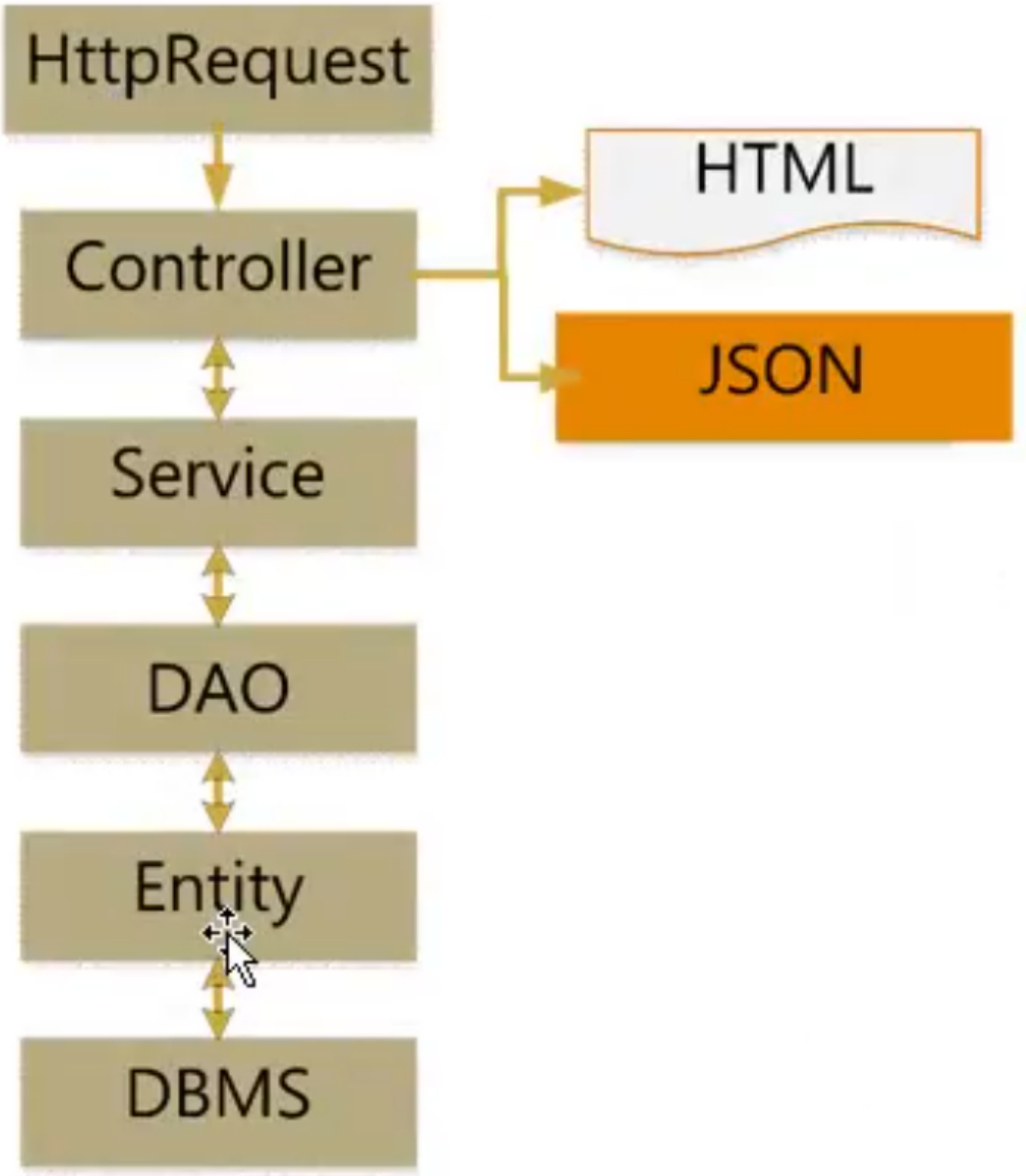
⭐6. 开发模式 - MVC


Model
Service
Controller
View

- Web 容器内置对象
  - `request`
  - `session`： 保存在浏览器关闭之前的所有数据
  - `response`
- *MVC 执行流程：开始 → 打开 View 层 → 向 Servlet 提出请求 → Servlet 处理请求 → 结果返回 View 层 → 结束
- MVC 模式概念（分层）
  - View - 视图，串行化纯文本
  - Controller - 控制器
  - Service - 服务，实现业务逻辑
  - Model(Domain, Entity) - 实体，关联数据库表
- 客户端跳转与服务端跳转
  - 客户端跳转：客户端跳转是通过在服务器返回的响应中包含特定的重定向指令，让客户端（通常是Web浏览器）负责执行跳转动作。客户端接收到重定向指令后，会向指定的目标URL发起新的请求，并在浏览器中展示新页面。
  - 服务端跳转：服务端跳转是在服务器端处理跳转逻辑，服务器收到请求后，在后台进行跳转操作，然后将跳转后的内容直接返回给客户端。

7. Spring Boot 架构

- 方法
  - 增加 - `POST` - `add` / `insert`
  - 删除 - `DELETE` - `delete`
  - 修改 - `PUT` - `update`
  - 查所有 - `GET` - `list` / `page`
  - 查单个 - `GET` - `getOne`
- RESTful
- 架构

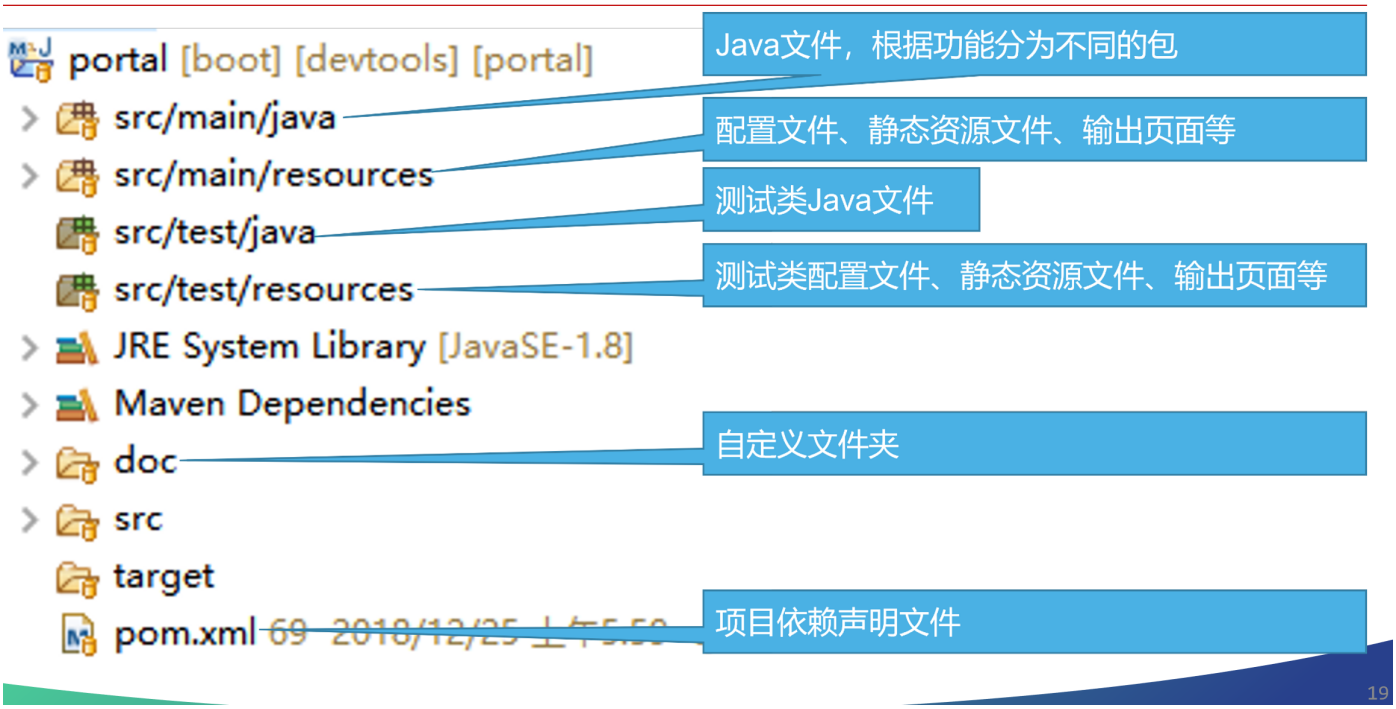


- 交互
  - HttpRequest 与 Controller 层交互
  - Controller 调用 Service 提供的服务
  - Service 调用 DAO 层定于的基本 CRUD 操作
  - DAO 层（持久层）对实体 Entity 进行持久化

- DAO 层说明

  这层是 MVC 开发规范上多出来的一层，用来定义业务系统针对数据库的存取，重用性强，忠实执行 Service 层的操作，与业务逻辑无关

- 目录结构



  - `resources/static` - 存放静态资源文件(HTML 相关)
  - `resources/application.yml` - 项目核心配置

建表

```
CREATE TABLE `t_person` (
  `id` varchar(100) NOT NULL COMMENT 'ID',
  `name` varchar(100) NOT NULL COMMENT '名称',
  `remark` varchar(500) DEFAULT NULL COMMENT '备注',
  `username` varchar(100) NOT NULL COMMENT '用户名',
  `password` varchar(100) DEFAULT '123456' COMMENT '密码',
  `creatorid` varchar(100) DEFAULT NULL COMMENT '创建人',
  `updaterid` varchar(100) DEFAULT NULL COMMENT '更新人',
  `createtime` datetime DEFAULT CURRENT_TIMESTAMP COMMENT '创建时间',
  `updatetime` timestamp NULL DEFAULT CURRENT_TIMESTAMP COMMENT '更新时间',
  PRIMARY KEY (`id`),
  UNIQUE KEY `uk_person_username` (`username`),
  KEY `asdfas` (`creatorid`),
  CONSTRAINT `asdfas` FOREIGN KEY (`creatorid`) REFERENCES `sys_dict_type` (`dict_type`) ON DELE
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

实体层开发

需要有数据注解 `@Data` 。

时间格式串行化：

```
@JsonFormat(pattern = "yyyy-MM-dd HH:mm:ss")

/* 示例 */
@Data
public class BaseDomain implements Serializable {
    // ...
    @JsonFormat(pattern = "yyyy年MM月dd日 HH:mm:ss.SSS")
    private Date createtime;//创建时间
}
```

e.g. Person 实体类：

```
@Data
public class Person extends BaseDomain {
    private static final long serialVersionUID = 1L;

    private String username; // 用户名
    private String password; // 密码
}
```

持久层开发

- 编写 CURD 方法，如 `getOne(id)` 、 `delete(id)` 、注解、嵌入脚本
- 编写 `mapper.xml` 持久层脚本
  很多 DBA 喜欢用下划线对字段命名，注意 XML 中语法

简单的 CURD，直接写：

```
public interface PersonDao {
    // 查所有
    @Select("select * from t_person")
    List<Person> listAll();

    // 进行登陆
    @Select("select count(1) from t_person t where t.username = #{username} and t.password = #{p
    int login(Person vo);

    // 删除
    @Delete("delete from t_person where id = #{id}")
    int delete(String id);

    // 根据 ID 或用户名查
    @Select("select t.* from t_person t where t.id = #{param} or t.username = #{param}")
    Person getOne(String param);

    // 根据用户名查
    @Select("select t.* from t_person t where t.username = #{username}")
    Person findByUsername(String username);
}
```

复杂的 CURD，先在 DAO 申明方法，然后在 `mapper.xml` 中写：

DAO：

```
public interface PersonDao {
    //在mapper.xml中实现
    int add(Person vo);

    //在mapper.xml中实现
    int update(Person vo);
}
```

mapper.xml：

```
<!-- ... -->

<!-- 命名空间对应 -->
<mapper namespace="com.ruoyi.test.dao.PersonDao">
    <!-- add 方法 -->
    <insert id="add" parameterType="com.ruoyi.test.domain.Person">
        INSERT INTO t_person(
            <if test="id != null and id != 0">id,</if>
            <if test="name != null and name != ''">name,</if>
            <if test="username != null and username != ''">username,</if>
            <if test="password != null and password != ''">password,</if>
            <if test="remark != null and remark != ''">remark</if>
        ) VALUES (
            <if test="id != null and id != 0">#{id},</if>
            <if test="name != null and name != ''">#{name},</if>
            <if test="username != null and username != ''">#{username},</if>
            <if test="password != null and password != ''">#{password},</if>
            <if test="remark != null">#{remark}</if>
        )
    </insert>

    <!-- update 方法 -->
    <update id="update" parameterType="com.ruoyi.test.domain.Person">
        UPDATE t_person
        <set>
            <if test="name != null and name != ''">name = #{name},</if>
            <if test="username != null and username != ''">username = #{username},</if>
            <if test="password != null and password != ''">password = #{password},</if>
            <if test="remark != null">remark = #{remark}</if>
        </set>
        WHERE id = #{id}
    </update>
</mapper>
```

服务层开发

- 接口和实现
- 注入持久层 `@Autowired` / `@Resource`
  指在服务层可以通过 `@Autowired` / `@Resource` 进行持久层操作的依赖注入。
- 命名 DAO 时可以直接命名 `dao`， 方便复制更改：

  ```
  @resource
  private PersonDao dao;
  ```

- 事务 - 涉及到**写操作**时都要加 `@Transactional` （直接在服务类前加）
  类似于 .NET 的 `.SaveChange()`

举例：

```
@Service
@Transactional
public class PersonServImpl implements IPersonServ { // 实现接口
    @Resource // 使用这个数据注释，进行持久层操作的依赖注入
    private PersonDao dao;
}
```

控制层开发

- 控制器类注解 `@RestController`， `@RequestMapping("path")`
- 方法注解 `@{Get/Post/...}Mapping`
  - 用 Body 体的(PUT, POST)变量注解 `@RequestBody`
  - 占位符参数： `@GetMapping("/getOnePerson/{id}") + ...(@PathVariable String id)`

```
@PutMapping("/updatePerson")
public Object updatePerson(@RequestBody Person vo)

@GetMapping("/getPerson/{id}")
public Object getPerson(@PathVariavle String id)
```

举例：

```java
@RestController
@RequestMapping("/hello")
public class HelloController {
    @Resource // 服务依赖注入，与 @Autowired 一样
    private IPersonServ personServ;

    @GetMapping("/")
    public String index() {
        System.out.println(new Throwable().getStackTrace()[0]);
        return "这是控制层返回的信息，调用了index()方法，全路径为【/hello/】";
    }

    @GetMapping("/getOnePerson/{id}")
    public Person getPerson(@PathVariable String id) {
        return personServ.getOne(id);
    }

    @GetMapping("/findPersonById")
    public Person findPersonById(String id) {
        return personServ.getOne(id);
    }

    @PostMapping("/loginPerson")
    public JSONObject loginPerson(@RequestBody Person vo, HttpSession session) {
        JSONObject o = personServ.login(vo);
        if(o.getBoolean("success")) {
            session.setAttribute("currentPerson", o.getObject("vo", Person.class));
        }
        return o;
    }
}
```

8. 前端开发

脚本引用

```html
<!-- 域外脚本 -->
<script src="https:// ... /jquery.3.2.1.min.js"></script>

<!-- 域内脚本 -->
<script src="../js/jquery.3.2.1.min.js"></script>
```

基础知识

- `` + ${} 实现引用变量字符串，如：

  ```
  alert(`学生${id}`);
  ```

- jQuery 相关 `$('')`
  - 内部用 CSS 选择器
  - 方法
    取值的话，则去掉最后一个参数
    - `$().css('color', red)` - 改变 CSS
    - `$().html('')` - 改变 HTML 内容
    - `$().val('')` - 改值（应当是改属性）
  - 掌握 DOM 的 `append()` 方法

```
var tableDom = $('#table-1');

var tableHtml = '<tr></tr>';
tableHtml.append(`<td>${id}</td> <td>${name}</td>`);

table.append(tableHtml);
```

⭐sessionStorage

浏览器缓存。

可以直接相当于对象使用。

```
sessionStorage['key'] = val;
var userName = sessionStorage['userName'];
sessionStorage['conf'] = JSON.stringify(conf);
var conf = JSON.parse(sessionStorage['conf']);
```

Ajax 相关

Ajax 中通过 `data` 指定传输数据，有两类情况：

- `data` 直接为 JS 对象：为 Param 参数，每一个属性用 `&` 分割
- `data` 使用 `JSON.stringify(Object)` ： 为 Body 参数
  ⚠同时还需要指定 `contentType: "application/json;charset=utf-8"` ！

(1) GET/DELETE 方法

当为 GET/DELETE 方法是，Ajax中的 `data` 一般是是 Param 参数（放在 URL 中）。

前端：

```
// 编写 findPersonById 方法，调用后端 GET API
function findPersonById(id) {
    $.ajax({
        type: 'get', // GET 方法
        url: '/hello/findPersonById', // API 的 URL
        data: {id: id}, // JS 对象数据，为 Param 参数
        async: true, // 是否异步（默认为 true）如为false，前端会等url请求完毕才能执行其它任务
        success: function (data) { // 成功回调函数
            console.log(data);
        },
        error: function (xhr, textStatus, errorThrown) { // 请求失败回调函数（应该是非200）
            console.log(xhr);
            console.log(textStatus);
            console.log(errorThrown);
        }
    });
} // 上面的 success 和 error 是 HTTP 请求成功或错误，非业务上的成功或错误
```

后端：

```
/* 控制层 - Person 控制器的 findPersonById API */
@GetMapping("/findPersonById")
public Person findPersonById(String id) { // 为 Param 参数
    System.out.println(new Throwable().getStackTrace()[0] + ", id = " + id); // 调试信息
    return personServ.getOne(id); // 直接调用服务层提供的服务
}

/* 服务层 - personServ 服务的 getOne 方法 */
@Override
public Person getOne(String id) {
    // System.out.println(new Throwable().getStackTrace()[0]);
    return dao.getOne(id); // 直接调用持久化层的 CURD 操作
}

// 持久层
@Select("SELECT t.* FROM t_person t WHERE t.id = #{param} OR t.username = #{param}")
Person getOne(String param);
// 以上代码的不足在于没有用JSONObject来封装业务，应该在服务层判断是否查出来为空
```

(2) POST/PUT 方法

当为 GET/DELETE 方法时，Ajax中的 `data` 一般是 Body 参数，需要注意处理。

前端：

```
// 编写 loginPerson 方法，调用后端 POST API
function loginPerson() {
    // 取得页面上的 username 和 password ，构建一个 JSON 对象
    let data = {
        username: $('#username').val(),
        password: $('#password').val()
    };
    $.ajax({
        type: 'post',
        url: '/hello/loginPerson',
        data: JSON.stringify(data), // Body 参数必须用 JSON.stringify
        contentType: "application/json;charset=utf-8", // Body 参数必须指定
        success: function (data) {
            console.log(data);
        },
        error: function (xhr, textStatus, errorThrown) {
            // ...
        }
    });
}
```

后端：

```java
/* 控制层 */
// @RequestBody 和前端 `application/json;charset=utf-8` 结合才能拿到传递过来的 Body 参数，并转换为 :
@PostMapping("/loginPerson")
public JSONObject loginPerson(@RequestBody Person vo, HttpSession session) { // session 为容器内
    JSONObject o = personServ.login(vo); // 调用登陆方法
    if (o.getBoolean("success"))
        session.setAttribute("currentPerson", o.getObject("vo", Person.class));
    return o;
}

/* 服务层 */
@Override
public JSONObject login(Person vo) {
    JSONObject o = new JSONObject();

    int result = dao.login(vo);
    if (result == 1) {
        o.put("success", true);
        o.put("message", "登录成功! ");
        o.put("vo", dao.findByUsername(vo.getUsername()));
    }
    else {
        o.put("success", false);
        o.put("message", "登录失败! ");
    }
    return o;
}
```

表单串行化

给 jQuery 扩展方法:

```javascript
// 扩展一个 serializeObject 方法
$.fn.serializeObject = function() {
    var o = {};
    var a = this.serializeArray(); // 将对象序列化为数组
    $.each(a, function () { // 对序列化数组的每个元素（即属性）进行转换
        if (o[this.name] !== undefined) { // 如果这个属性名已存在
            if (!o[this.name].push) // 这个属性名值 不为数组（即没有 push 方法）
                o[this.name] = [o[this.name]]; // 转换为数组
            o[this.name].push(this.value || ''); // 数组 push
        }
        else // 属性名不存在，新增
            o[this.name] = this.value || '';
    });
    return o; // 返回序列化对象
};
```

调用示例：

```
alert(JSON.stringify($("#insert-form").serializeObject()));
```

0x02 Websocket 网络编程

与 HTTP 1.1 区别

- HTTP 1.1 是非持续连接
- Websocket 是持续连接，相当于 HTTP 1.1 的补充协议
  通过 HTTP Request 建立连接，之后不需要再发送 Request，保持与另一端的 TCP 连接。

通信过程

1. websocket server跟随web容器启动，端口和web容器端口一样
   `@ServerEndpoint("/websocket/{cid}")`
   `/websocket/` 相当于控制层的路径，而 `/{cid}` 是客户端声明的昵称
2. 客户端与websocket server建立连接
   `url = ws://host:端口/ServereEndpoint/昵称`
   如 `ws://localhost:8080/websocket/bobydog`
   - 连接建立后，服务端拿到 `sessionId` ， 里面包括了 `cid` 和 `ip` 、 `port`
   - 其中客户端的ip和端口总是不一样的，即便是同一个cid（即便是同一个客户端主机，端口也会不一样）
   - `cid` 相当于昵称，可以为 `null` ， 但不推荐
3. 交换数据
   1. 客户端把**消息发送**到服务端
   2. 服务端**回应消息**
   3. 服务端**广播消息**

关键参数

- wsUrl - 客户端连接 Websocket 服务器的完整 URL
  格式： `ws(s)://主机:端口号/ServerEndpoint/{cid}` (wss类似于https)
- `ServerEndpoint` - 类似于控制层，即定义的 Websocket 服务入口
- `{cid}` - 客户端的名称
  可让多个客户端采用同样的名称，如：多个终端监控同一网络元素、弹出广告等

代码部分

前端

- 一个 Websocket 服务器地址输入框
- 一个个人昵称（作为 `cid`）
- 断开连接状态
  - 一个建立连接按钮，在建立连接后隐藏
- 建立连接状态
  - 一个消息输入框
  - 一个消息发送按钮
- 消息栏

JavaScript：

```
// 打开 Websocket 连接
let ws = null; // 先提前定义

// 打开连接
function openWebsocket() {
    let wsUrl = $('#serverUrl').val() + $('#nickname').val(); // 生成 WS 地址
    ws = new WebSocket(wsUrl); // 生成 WS 服务端
    // 以下定义各个回调函数
    ws.onopen = function(evt) { // 连接打开（建立）事件
        _appendMessage("连接已建立...");
        $(断开连接状态).hide();
        $(显示连接状态).show();
        ws.send("Hello WebSockets!"); // 向服务端发送消息
    }
    ws.onmessage = function(evt) { // 收到消息事件
        _appendMessage(`${evt.data}`);
    }
    ws.onclose = function(evt) { // 连接关闭事件
        _appendMessage("连接已关闭");
        $(断开连接状态).show();
        $(显示连接状态).hide();
    }
    ws.onerror = function(evt) { // 连接打开出错事件
        // ...
    }
}

// 关闭连接
function closeWebsocket() {
    ws.close();
}

// 发送消息输入框中消息
function sendMessage() {
    ws.send($(消息输入框).val());
}

// 显示消息
function _appendMessage(s) {
    消息栏.append(`<br>${new Date()}: ${s}`); // 时间: 消息内容
}
```

三、Socket 网络编程

Socket(套接字)，比较底层，只做了解。

1. B/S 套接字通信简单示意模型

要会画

四、Docker

- Docker 与虚拟机的区别
  Docker是**基于容器化**的虚拟化技术，**共享主机**操作系统内核，具有轻量级、快速启动和灵活的部署方式；虚拟机是**基于硬件**虚拟化技术，每个虚拟机需要**独立**的操作系统和资源，具有更高的隔离性和独立性。
- 镜像、容器、仓库的基本概念
  - 仓库
    仓库是用于存储和管理Docker镜像的地方。仓库可以分为两种类型：公共仓库和私有仓库。
    - 公共仓库(Public Repository)：公共仓库是由Docker官方或其他组织提供的存储镜像的地方，其中最著名的是Docker Hub。在公共仓库中，可以找到大量的预构建镜像供用户下载和使用。
    - 私有仓库(Private Repository)：私有仓库用于组织或个人存储和管理自己的镜像。私有仓库可以部署在本地或云上，提供更高的安全性和定制性。
  - 容器
    容器是**从镜像创建的运行实例**。可以将容器看作是一个**独立且隔离的运行环境**，其中可以运行

应用程序及其依赖项，而不会与主机环境或其他容器相互影响。容器是可运行、可启动、可停止和可删除的。

- 镜像
  镜像是Docker的基本构建块，它是一个轻量级、独立的可执行软件包，包含运行应用程序所需的一切，包括代码、运行时环境、库文件、配置等。镜像是只读的，可以用作创建容器的模板。
  拓展 - 层(Layer)：
  Docker镜像可以通过一个或多个层（Layers）来构建，每个层包含了对镜像的修改或添加。这种基于层的构建方式使得镜像可以进行高效的共享和复用，同时也方便了镜像的版本管理和更新。

- 镜像操作
  - 列出 - `docker images`
  - 查找 - `docker search <镜像名称>` （在 Docker Hub 上查找）
  - 拉取 - `docker pull <镜像名称>:<标签(版本)>`
  - 更新 - `docker pull <镜像名称>:<标签(版本)>`
  - 标签 - `docker tag <源镜像名称>:<源标签> <目的镜像名称>:<目的标签>`
  - 删除 - `docker rmi <镜像名称>:<标签>`
- 容器命令
  - 首次启动： `run` + 参数(`-itd` ， `-v` ， `-p` ， `--name`)
    - `-itd` - i交互式、t中断、d后台
    - `-v` - volume，本机文件夹映射到 Docker 文件夹
    - `-p 80:5000` - 将容器的5000端口映射到本机的80端口
    - `--name` - 容器名称