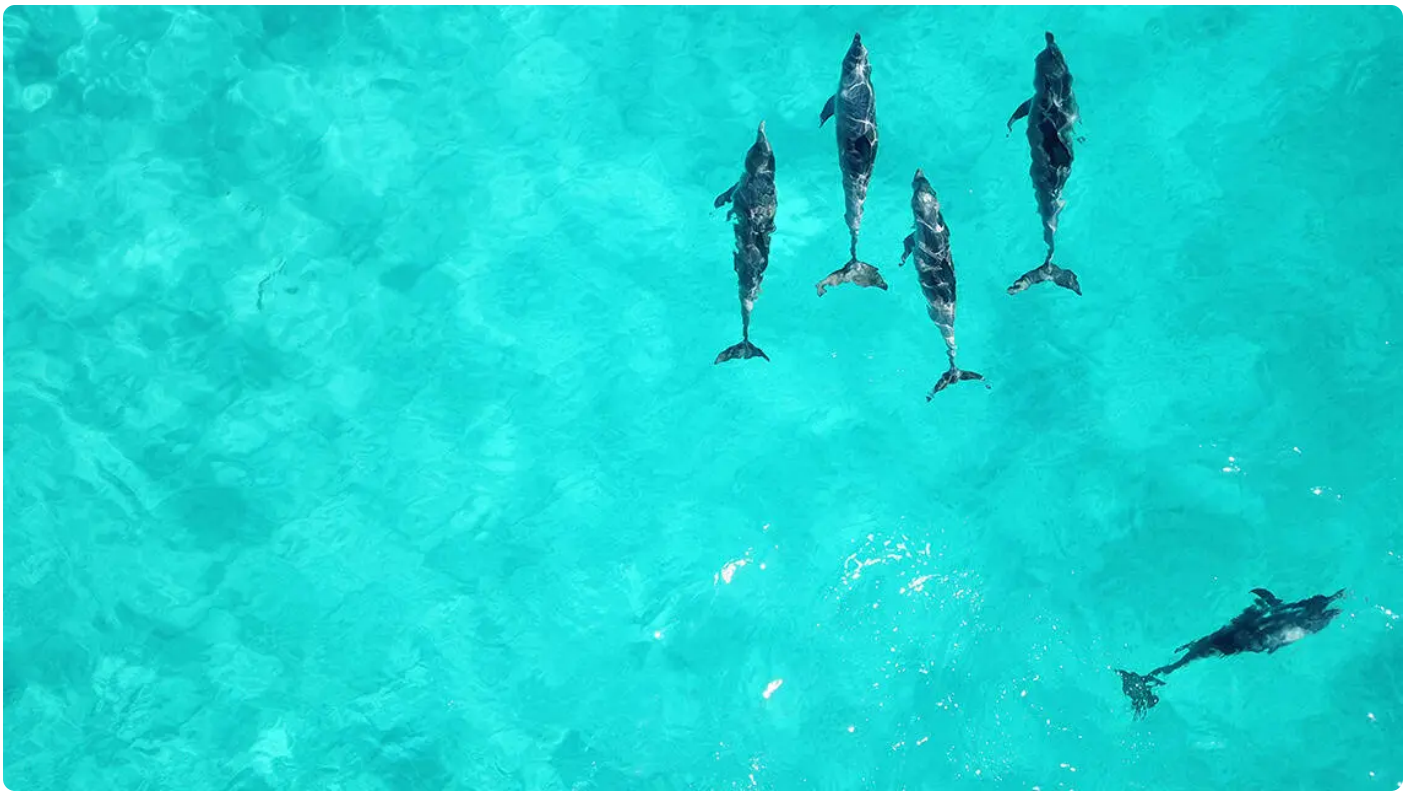
21 | 用户注册和登录：如何结合Vue 3和Koa.js实现注册登录？

2023-01-11 杨文坚 来自北京

《Vue 3 企业级项目实战课》

课程介绍 >



讲述：杨文坚

时长 15:15　大小 13.94M

你好，我是杨文坚。

上节课，我们围绕运营平台 Node.js 服务的功能特点，学习了数据库方案设计，以及如何优雅实现数据库的快速初始化。同时，也提到一个项目观点：数据库的设计等于是业务功能的设计，所以只要数据库设计好了，业务功能设计也就基本成型。

今天这节课，我们也秉承这个观点，基于上节课设计的“员工用户表”来实现用户的注册和登录的功能（为了描述方便，“员工用户”我们就都简称为“用户”）。

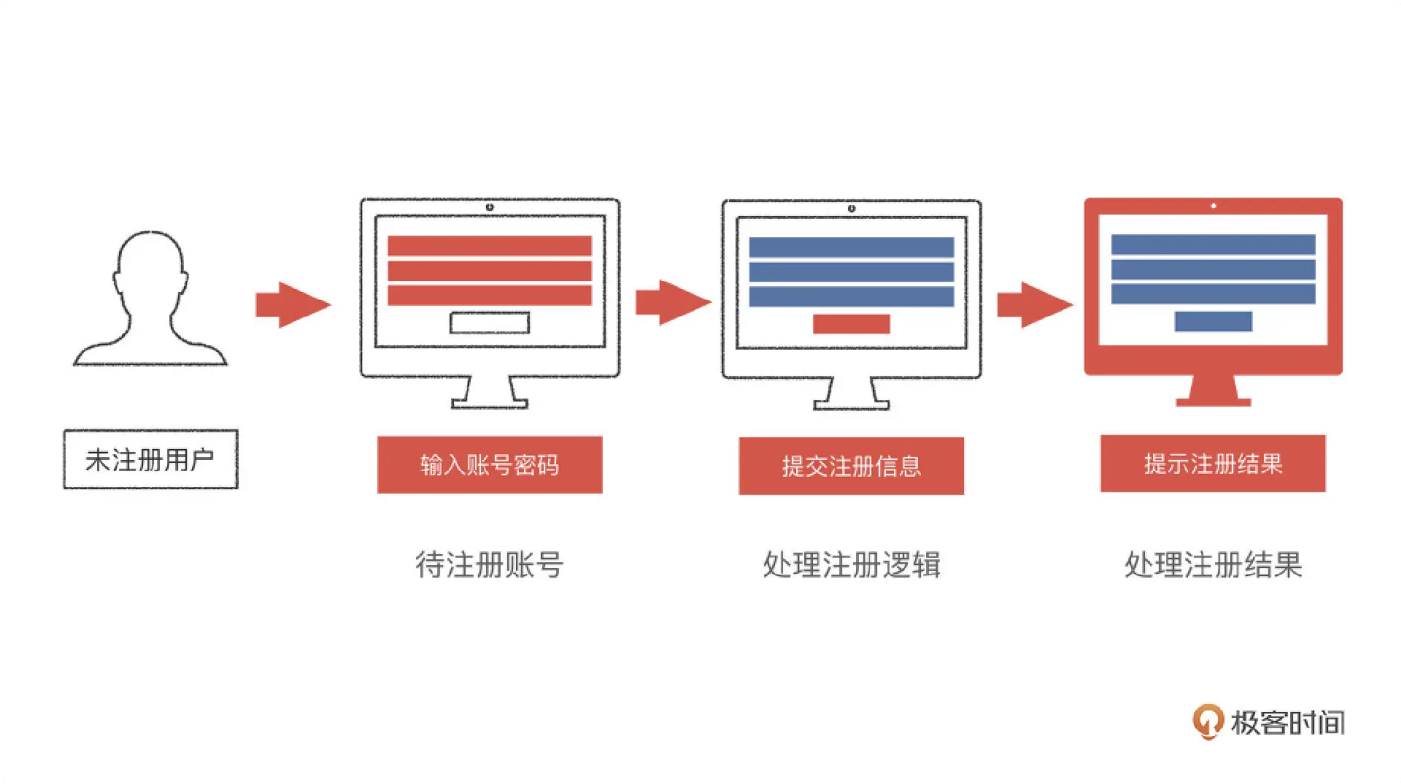
一般，Web 平台涉及用户操作的功能，都必须匹配相应的用户注册和登录的功能，才能规范用户的操作权限，避免产生业务数据污染，保证平台流程有条不紊地自动运行。那么如何实现运营平台的注册和登录功能呢？

只有数据库的员工用户表是远远不够的，我们需要基于员工用户表，设计“注册和登录的功能逻辑”。

如何设计注册和登录的功能逻辑？

用户注册和登录功能逻辑的设计，其实并不复杂，重点是理清功能点的每个步骤的逻辑。

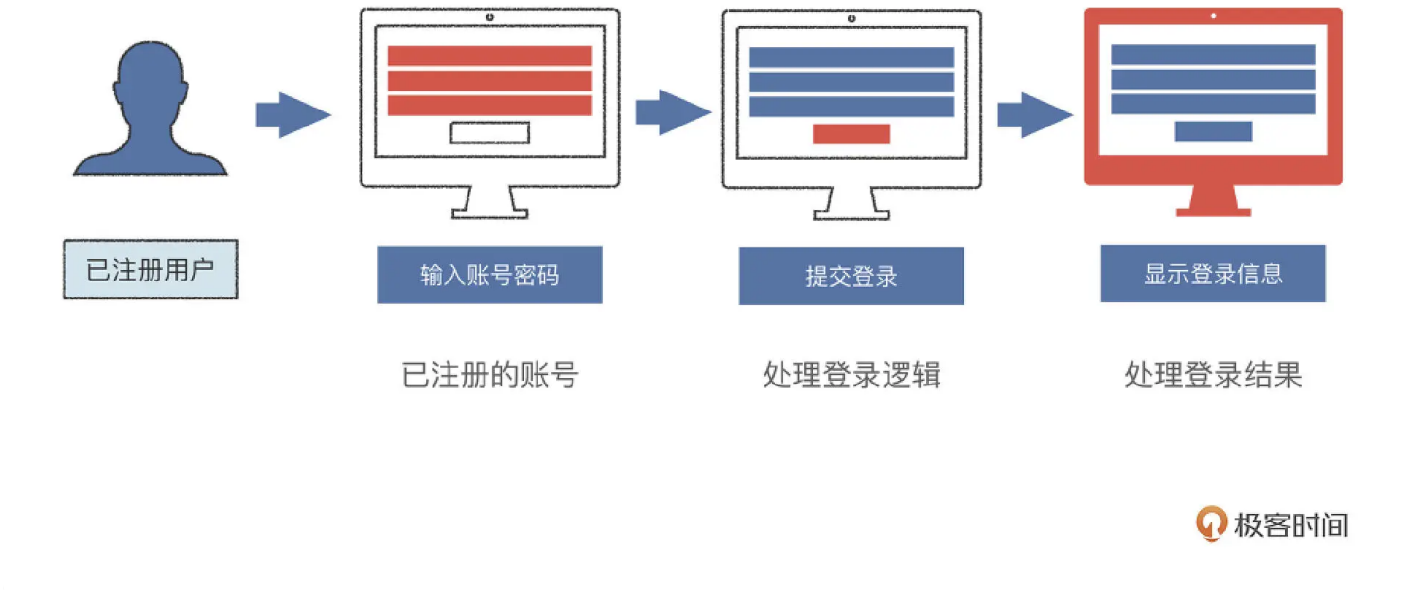
我们先看用户注册的功能逻辑链路，我画了一张图。



可以看出，注册功能的主要逻辑可以分成三步。

- 第一步，用户输入和提交“待注册”账号密码。这一步中，功能上要辅助用户校验用户名称是否符合规范，同时，要辅助用户校验两次密码是否一致。
- 第二步，用户提交注册信息，处理注册逻辑。这时候要平台判断账号是否存在，判断是否能注册，并处理注册结果。如果用户名称已存在，就准备好失败提示结果，如果用户名称不存在，就注册用户信息。
- 第三步，处理注册后的结果。如果用户注册成功，就自动跳转到登录页面，如果用户注册失败，显示失败原因，提醒用户如何改正。

设计完用户注册功能，我们来设计用户登录功能逻辑，我也同样画了一张图展示功能逻辑链路。



可以看出，用户登录的功能逻辑也可以分成三个步骤。

- 第一步，用户输入和提交“已注册”账号密码。
- 第二步，用户提交登录信息。这时候平台需要判断账号密码是否正确，并返回登录成功或者失败的结果。
- 第三步，处理登录后的结果。如果登录成功，自动跳转管理页面，如果登录失败，显示失败原因。

这些就是从功能逻辑层面来设计用户注册，是不是很简单？

功能的逻辑设计这类工作，一般在企业内部，是产品经理负责执行的。但是，如果项目是由技术人员来主导的，例如我们课程的企业内部运营搭建平台，那一般都需要程序员来自主设计功能逻辑流程的。

好，实现了功能逻辑的设计，接下来就是把“功能逻辑设计”“翻译”成“技术逻辑设计”，或者是“技术方案设计”。这个“翻译”过程就是程序员**根据功能逻辑的情况，选择合适的技术方案或者框架，来实现对应的功能点**。

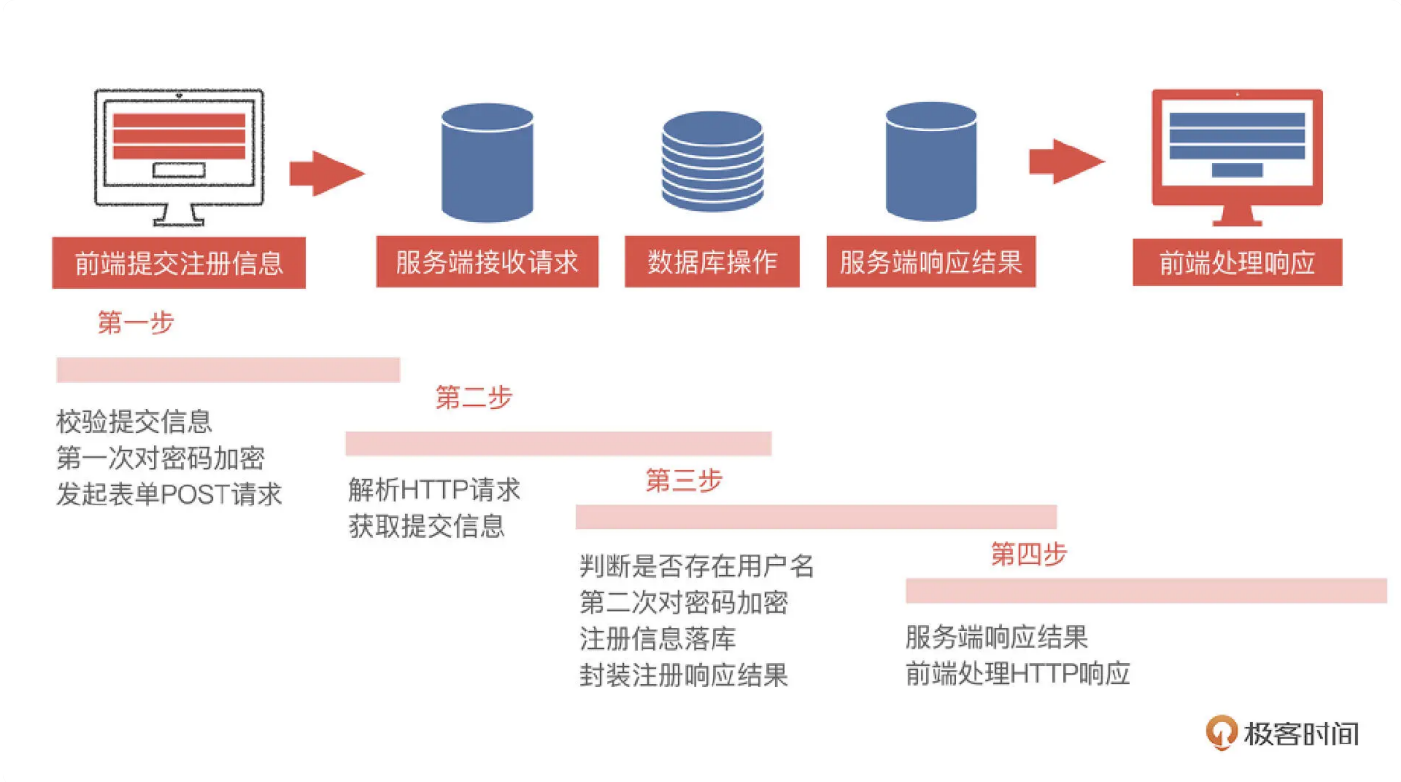
那么如何设计用户注册和登录的技术方案呢？

如何设计用户注册和登录的技术方案？

简单理解，技术方案 / 技术逻辑的设计，是把“功能逻辑”进行技术化的“翻译”。那么我们可以根据前面画好的功能逻辑的设计图，结合技术层面的理解来修改。

注册流程的技术方案

我先画出用户注册的技术方案设计图。



可以得知，用户注册的技术方案实现逻辑分成四步。

第一步，前端提交“待注册”的账号和密码。

在这一步中，前端可以通过动态表单实现注册表单功能，并且基于动态表单来做新校验。同时还要进行用户名符串校验，两次密码的字符串格式校验，以及检查两次密码是否一致。

信息校验完后，这里要将密码进行第一次 md5 加密，避免密码数据在 HTTP 传输过程中被明文暴露出来。虽然现在大多数网站都用 HTTPS 进行加密传输，但是仍然存在暴露明文数据的

风险，例如一些操作系统、一些 APP 网络代理，都有可能拿到明文的信息。所以我们这里就把密码做一次 md5 加密，这是非对称加密，无法解密出原来字符串。

在第一步末尾，前端用表单形式发起 POST 异步 HTTP 请求，把处理好的注册信息发送给服务端。

第二步，服务端接收到 HTTP 请求，解析出表单里的账号和密码。

我们是用 Koa.js 来搭建的 Web 服务，那么要解析出 HTTP 的表单数据，也就是注册用户的信息，可以用 Koa.js 的中间件“koa-bodyparser”来解析。

第三步，服务端处理注册的数据逻辑。

这时候服务端会查询数据库，检查账号是否已经存在，判断是否能进行注册流程。如果账号存在，返回对应失败结果，提供给浏览器提示用户失败原因；如果账号不存在，就可以进行注册的数据操作了，这里进行密码第二次 md5 加密，然后把账号、密码和数据插入到数据库里，最后返回注册成功的结果。

这里要注意，我们最终存在数据库的用户密码，是经过两次 md5 加密的结果（前端一次，服务端一次）。其实，只用前端的一次加密逻辑也可以，但是两次 md5 加密主要是想双重安全保障，毕竟没有绝对安全的加密算法的，md5 加密，在极端情况下还是能暴力破解的。

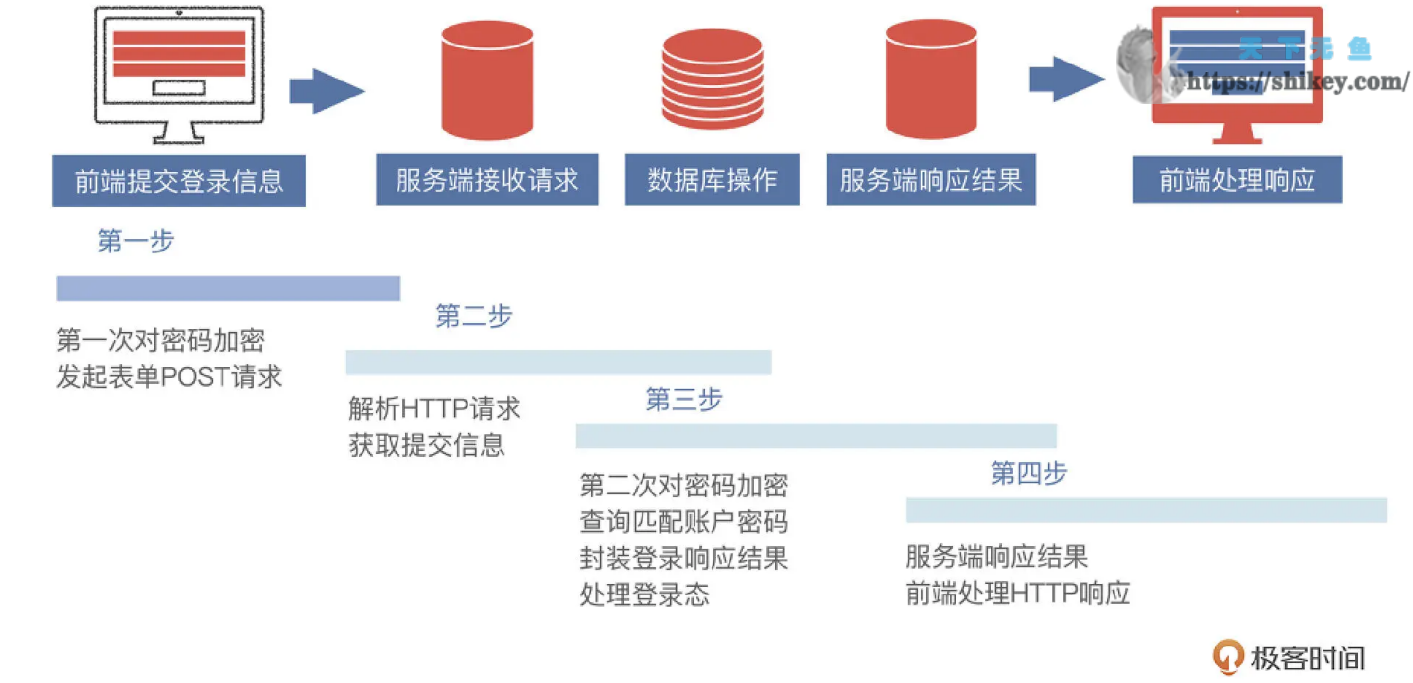
要记得，这个两次 md5 加密逻辑也要用到后面的登录操作流程中，才能保证匹配上数据库的用户密码数据。

第四步，浏览器接处理服务端 HTTP 响应结果。如果注册成功，就设置定时器自动跳转登录页面，如果注册失败，就提示失败原因。

到这里，注册流程的完整技术方案，我们就梳理好了。

登录流程的技术方案

接下来设计用户登录流程的技术方案。我也把登录实现技术方案画成一张图。



用户登录操作的技术实现，我也分解成了四个步骤。

第一步，前端提交“已注册”的账号和密码，这时候前端也可以用动态表单，来实现登录的页面交互功能。同时，前端要密码用 md5 做第一次加密，最后以表单形式发起 POST 异步 HTTP 请求给服务端。

第二步，服务端接收 HTTP 请求，解析出表单里的账号和密码，具体技术实现跟注册技术方案的第二步一样。

第三步，服务端校验账号和密码是否正确。这一步中，要对密码进行第二次 md5 加密，因为存在数据库的密码是二次 md5 加密的。

然后，我们用账号和二次加密后的密码进行数据库的查询。如果账号和密码正确，就在 Control 层设置 HTTP 的 Cookie 作为登录态，再返回成功结果。如果账号和密码匹配不上，就返回失败结果。

第四步，浏览器处理 HTTP 响应结果。如果注册成功，就设置定时器自动跳转管理页面，如果注册失败，就提示失败原因。

好，用户注册和登录的技术方案梳理完成，是不是可以直接进入代码实现的环节了呢？

其实还不行，在登录技术方案中，我们提到一个环节，用户登录成功后就跳转到管理页面。这之后就是后续用户操作流程了，所以，管理页面必须做好用户登录态的校验。**我们这里用了 Cookie 做为登录态，那么到底什么是用户登录态呢？**

“登录态”和相关技术原理，跟用户注册登录的技术功能是一体的，非常重要，我们需要理清相关概念才能进入代码实现阶段。

什么是“用户登录态”？

当用户登录成功后，标记在浏览器端、客户端或者服务端的数据，作为登录判断的唯一标识，就是“用户登录态”，一般简称为“登录态”。

在 Web 服务中，登录态通常用来在每个 HTTP 请求中，判断是否存在登录数据。虽然 HTTP 是无状态的网络协议，但是在 HTTP 的内容中，可以设置一些数据作为登录态的唯一标识。

不过，登录态在项目开发中，经常被认为是一种“业务概念”，不是技术概念，所以没有什么技术规范，都是根据业务需要，自行选择合适的技术来实现。这时候，我们就需要人为设计和约定一套“登录态的数据规范”。

在 Web 领域里，登录态的信息有两种标记方法。

- 基于 HTTP 的 Cookie 能力，标记在浏览器端。
- 标记在服务端，术语是 Session，实现技术一般都是服务端的存储技术，例如缓存或数据库。

单纯使用 Cookie，是最简单的实现方式。当用户登录成功后，在服务端中把登录数据设置在 HTTP 的 Cookie 里，响应给浏览器，这时候浏览器会记录对应网址的 Cookie。

也就是说，最后让浏览器端基于 Cookie 来存储登录态，每当浏览器发起 HTTP 请求后，就会携带对应的 Cookie 给服务端做校验。

不过，单纯使用 Session，很难依靠存储技术实现登录态，因为缺少“浏览器端”或者“客户端”的用户跟踪能力，所以需要结合 Cookie 来实现 Session 跟踪。当用户登录成功后，把登录态存储在数据库或者缓存中，然后把数据的 id，也就是 SessionId，放在 Cookie 里传给浏览

器。每次浏览器发起 HTTP 请求时，会携带对应的 Cookie 给服务端，服务端拿到 Cookie 里的 SessionId，去缓存或者数据库里，找到完整的登录信息进行校验。

我们的课程内容侧重整体平台设计和实现，所以就选择最简单的 Cookie 方式来做登录态管理。

这时候，Cookie 的安全性就尤为重要。我们需要把登录态的 Cookie 设置 HttpOnly，让 Cookie 的读写只能在服务端处理。同时，还要设置 Cookie 的有效时间。

而且我们知道，Cookie 是可以在浏览器的控制台看到明文内容的，那么 Cookie 里用户明文信息存在浏览器端很也很危险，**有没有低成本的办法，把 Cookie 的数据在浏览器里加密，然后在服务器里解密呢？**

答案是有的，就是 JSON Web Token 技术，简称“JWT”。

如何使用 JWT 来管理用户登录态？

首先来理解一下，“JWT”，全称 JSON Web Token，业界一般是这么描述的。

> “JSON Web Token 定义了一种紧凑的、自包含的方式，用于作为 JSON 对象在各方之间安全地传输信息。该信息可以被验证和信任，因为它是数字签名的。”

我们结合具体实现步骤来理解。

第一步，生成 JWT。当用户登录成功后，服务器把用户数据封装成 JSON 对象，然后进行加密，成一个“加密字符串”，响应返回给浏览器。

第二步，存储 JWT。这时候，浏览器可以把这个“加密字符串”存在 localStorage 里，或者服务器响应时存在 Cookie 里。

第三步，浏览器使用 JWT 和服务器校验 JWT。这时候，浏览器发送 HTTP 请求的过程中，可以把“加密字符串”加到 HTTP Header 里，或者放在 HTTP Body 里，或者自动跟随在 Cookie 里。当服务端接收到 HTTP 请求后，从 HTTP 的 Header、Body 或者 Cookie 里拿到 JWT，进行解密，把 JSON 数据拿出来检查是否符合要求。

这是 JWT 的一个最简单的流程。其实，我们只需要用到 JWT 的加密和解密的能力，来进行 Cookie 数据的加密和解密。

好，到现在，我们已经准备好了用户注册和登录的技术方案，结合登录态的技术方案，可以开始写代码实现了。

如何根据技术方案进行代码实现？

这里我用注册流程作为案例，演示代码实现。注册流程代码分成两大部分，前端部分和服务端部分。

首先是前端部分，前端的实现用之前课程里的动态表单进行封装，来实现注册的表单。

复制代码

```
<template>
  <DynamicForm
    class="sign-up-form"
    :model="model"
    :fieldList="fieldList"
    @finish="onFinish"
    @finishFail="onFinishFail"
  >
    <div class="btn-groups">
      <Button type="primary">注册</Button>
    </div>
  </DynamicForm>
</template>

<script setup lang="ts">
import { DynamicForm, Button, Message } from '@my/components';
import md5 from 'md5';
import type { DynamicFormField } from '@my/components';

interface SignUpFormData {
  username: string;
  password: string;
  confirmPassword: string;
}

const model: SignUpFormData = {
  username: 'admin001',
  password: '123456',
  confirmPassword: '123456'
};

const fieldList: DynamicFormField[] = [
```

```
  {
    label: '用户名称',
    name: 'username',
    fieldType: 'Input',
    rule: {
      validator: (val: unknown) => {
        const hasError = /^[0-9a-z\-._]{4,16}$/.test(`${val || ''}`) !== true;
        return {
          hasError,
          message: hasError ? '必须是4~16位的0-9a-z和.-_的组合' : ''
        };
      }
    }
  },
  {
    label: '密码',
    name: 'password',
    fieldType: 'InputPassword',
    rule: {
      validator: (val: unknown) => {
        const hasError = /^[0-9a-zA-Z]{6,16}$/gi.test(`${val || ''}`) !== true;
        return {
          hasError,
          message: hasError ? '密码必须是6~16位的数字和字母组合' : ''
        };
      }
    }
  },
  {
    label: '确认密码',
    name: 'confirmPassword',
    fieldType: 'InputPassword',
    rule: {
      validator: (val: unknown) => {
        const hasError = /^[0-9a-zA-Z]{6,16}$/gi.test(`${val || ''}`) !== true;
        return {
          hasError,
          message: hasError ? '密码必须是6~16位的数字和字母组合' : ''
        };
      }
    }
  }
];

const onFinish = (e: SignUpFormData) => {
  if (e.password !== e.confirmPassword) {
    Message.open({
      type: 'error',
      text: '两次密码不一致',
      duration: 2000
    });
    return;
```

```
  }
  fetch('/api/post/account/sign-up', {
    body: JSON.stringify({
      username: e.username,
      password: md5(e.password)
    }),
    headers: {
      'content-type': 'application/json'
    },
    method: 'POST'
  })
    .then((res) => res.json())
    .then((result: any) => {
      Message.open({
        type: result.success ? 'success' : 'error',
        text: result.message,
        duration: 2000
      });
      if (result.success) {
        setTimeout(() => {
          window.location.href = '/page/sign-in';
        }, 2000);
      }
    })
    .catch((err: Error) => {
      Message.open({
        type: 'error',
        text: `注册失败 [${err.toString()}]`,
        duration: 2000
      });
    });
};

const onFinishFail = (e: unknown) => {
  // eslint-disable-next-line no-console
  console.log('fail =', e);
};
```


接下来就是服务端部分。

还记得我们做过的 Node.js 服务端结构分层吗？代码的实现，其实就是在不同分层里，实现对应的分工代码就好。

服务端的注册功能，我们根据之前的服务端分层 Router、Controller、Service、Model 逐一实现。

首先是 Router，也就是路由层，要注册 HTTP 的注册 API。


```ts
// packages/work-server/src/router.ts
import Router from '@koa/router';
import {
  signUp,
} from './controller/user';
// 其它省略代码 ...

router.post('/api/post/account/sign-up', signUp);
// 其它省略代码 ...
const routers = router.routes();

export default routers;
```

注册了路由，接下来就是实现路由层 Router 上的控制层 Controller 的用户注册方法。


```ts
// packages/work-server/src/controller/user.ts
import type { Context, Next } from 'koa';
import { registerUser, queryAccount } from '../service/user';

// 其它省略代码 ...
export async function signUp(ctx: Context) {
  const params = ctx.request.body as {
    username?: string;
    password?: string;
  };
  const result = await registerUser(params);
  ctx.body = result;
}
// 其它省略代码 ...
```

实现了控制层 Controller 的注册方法，接下来就实现其所依赖的业务层，Service 的注册逻辑方法。


```ts
// packages/work-server/src/service/user.ts
import {
  checkUserIsUsernameExist,
  createUser
} from '../model/user';
import type { MyAPIResult } from './types';
```

```ts
import type { UserInfo } from '../model/user';

export async function registerUser(params: {
  username?: string;
  password?: string;
}): Promise<MyAPIResult> {
  const { username, password } = params;
  let result: MyAPIResult = {
    data: null,
    success: false,
    message: '注册失败'
  };
  if (!username || !password) {
    result.message = '信息不全';
    return result;
  }

  try {
    const isExist = await checkUserIsUsernameExist({ username });
    if (isExist === true) {
      result.message = '用户名已存在';
      return result;
    }
    const createResult = await createUser({ username, password });
    result = {
      data: createResult,
      success: true,
      message: '注册成功'
    };
  } catch (err: any) {
    // eslint-disable-next-line no-console
    console.log(err);
    result.message = err?.toString() || '出现异常';
  }
  return result;
}
```


在业务层 Service 的注册方法 registerUser 中，实现逻辑是先查询是否存在用户名，如果不存在就可以进行注册操作。所以，这里依赖了数据模型层 Model 中的两个方法。

那接下来，我们就实现数据库操作的数据库模型层。


```ts
import md5 from 'md5';
import { v4 } from 'uuid';
import { runSQL, tranformModelData } from '../util/db';

export interface UserInfo {
```

```
  id: number;
  uuid: string;
  username: string;
  password: string;
  info: unknown;
  extends: unknown;
  createTime: string;
  modifyTime: string;
}

export async function checkUserIsUsernameExist(params: {
  username: string;
}): Promise<boolean> {
  const { username } = params;
  const sql = `
    SELECT * FROM \`user_info\` WHERE username = ?;
  `;
  const values = [username];
  const results = await runSQL(sql, values);
  if (results && results.length > 0) {
    return true;
  }
  return false;
}

export async function createUser(params: {
  username: string;
  password: string;
}) {
  const sql = `
    INSERT INTO \`user_info\` SET ?;
  `;
  const uuid = v4();
  const { password, username } = params;
  const values = {
    uuid,
    password: md5(password),
    username,
    status: 1
  };
  const results = await runSQL(sql, values);
  if (results?.insertId > 0) {
    return { uuid, username };
  } else {
    return null;
  }
}
```


用户注册技术方案的代码实现过程就是这样，是不是很清晰，更多完整代码信息，你可以看对应章节的代码案例。

总结

经过今天的学习，相信你已经掌握了用户注册和登录的全栈功能实现。我们基于运营搭建平台这个案例，分析了 Web 平台常见的用户注册和登录功能，类似的从前端到服务端的全栈技术实现流程主要可以分成三步。

- 第一步，设计功能逻辑，抛开前端和服务端的界限，先理清楚功能逻辑，把功能逻辑的步骤给梳理出来。
- 第二步，设计技术方案，根据梳理出的功能逻辑步骤，进行技术层面的“翻译”，也就是拆解成对应的前端实现技术和服务端实现技术。至于哪些是要前端实现，哪些要在服务端实现，就看技术特点和个人选择了。
- 第三步，根据技术方案做代码实现。具体就是根据技术方案里的选型、分工等设计，选择对应的技术来逐一实现代码。

在这个全栈项目的常规开发流程中，第一步不一定是程序员的职责，**但是程序员必须要掌握第一步的功能逻辑的设计能力，有了这个能力，你可以突破技术职责的边界，锻炼出独当一面的项目管理软实力。**

分析了功能逻辑设计和技术方案设计，在“用户注册和登录”功能的开发中，我们要理解登录态的使用场景以及技术实现方式。

最后也学习了 JSON Web Token 这个低成本的权限校验技术方案。不过，关于平台的用户注册和登录的技术理解，其实，**没有最完美、最安全的技术方案，只有当下“较为安全的技术阀杆”**。毕竟，技术都有时效性，技术的安全性就也有时效性，所以任何技术都是要与时俱进，安全技术也一样，我们要根据技术的发展做持续的安全优化。

思考题

通过 Cookie 作为登录态唯一数据存在浏览器里，如果 Cookie 被其他人拿到，是否能伪造 Cookie 所属用户的登录行为？有办法避免出现这个问题吗？从 Web 服务平台提供方视角出发，一般怎么做登录态的保护？

欢迎在评论区留言参与讨论，我们下一节课再见。

完整的代码在这里

赞 2　提建议

上一篇 20丨数据库方案设计：如何设计运营搭建平台的数据库？

下一篇 22丨物料组件的编译和管理：如何处理组件的多种模块格式？

精选留言 (1)

写留言

蒙

2023-01-11 来自北京

加解密可以直接用node的加密函数不用jwt吗，
用jwt加解密有什么额外好处吗

作者回复: 可以把jwt当成是一种最佳实践，自己用加密函数实现其实是自己的一种实践。所以直接使用jwt会发现他考虑了各种场景的实际问题，为此提供了一些简洁的api，而自己实现就要自己去考虑了

2