SimpleFileServer 软件工程化（自动化、协作化）说明文档

一、引言

1.1 编写目的

本说明文档旨在阐述 SimpleFileServer 项目在软件工程化方面所采用的自动化和协作化手段，帮助开发团队成员、项目管理人员以及其他相关人员了解项目的开发流程、工具使用和协作方式，以提高开发效率、保证软件质量，并促进团队成员之间的有效沟通与协作。

1.2 项目背景

SimpleFileServer 是一个允许用户浏览、上传、下载和管理文件的简单文件服务器。随着项目的不断发展，为了提高开发效率、保证代码质量以及促进团队协作，采用了一系列软件工程化手段，包括自动化构建、测试、部署以及协作化开发工具和流程。

二、自动化手段

2.1 自动化构建

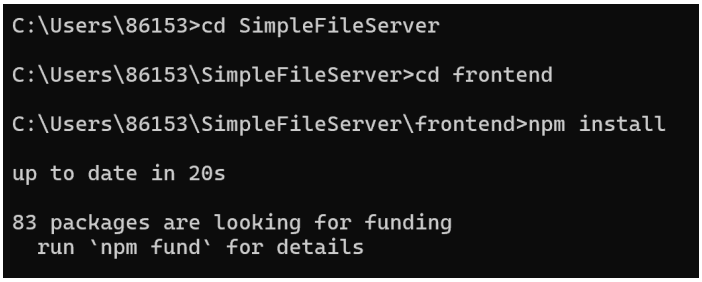
2.1.1 工具选择

项目采用 npm 作为包管理工具，结合 Node.js 实现自动化构建。npm 可以方便地管理项目的依赖项，并执行各种脚本命令。

安装依赖项：

```
# Install backend dependencies
cd backend
npm install

# Install frontend dependencies
cd ../frontend
npm install
```



```
C:\Users\86153>cd SimpleFileServer

C:\Users\86153\SimpleFileServer>cd backend

C:\Users\86153\SimpleFileServer\backend>npm install

up to date in 971ms

33 packages are looking for funding
  run `npm fund` for details
```

运行服务器：

```
# Run the backend server (development mode) (Default port: 11073)
cd backend
npm run dev

# Or run the backend server (production mode) (Default port: 11073)
cd backend
npm start

# In another terminal, run the frontend server (Default port: 2711)
cd frontend
npm run dev

# Or run the frontend server (production mode) (Default port: 2711)
cd frontend
npm run build
npm start
```

```
C:\Users\86153\SimpleFileServer\frontend>npm run dev

> frontend@0.1.0 dev
> next dev --turbopack --port 2712 -H localhost

   ▲ Next.js 15.3.2 (Turbopack)
   - Local:        http://localhost:2712
   - Network:      http://localhost:2712

 ✓ Starting...
 ✓ Ready in 9.4s
```

```
C:\Users\86153\SimpleFileServer\backend>npm run dev

> simple-file-server-backend@1.0.0 dev
> nodemon app.js

[nodemon] 3.1.10
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node app.js`
WARNING: No configurations found in configuration directory:C:\Users\86153\SimpleFileServer\backend\config
WARNING: To disable this warning set SUPPRESS_NO_CONFIG_WARNING in the environment.
[2025-06-14T10:55:47.849Z] Server running on port 11073
```

可以看到文件服务器前后端代码成功运行。

2.1.2 构建脚本

在项目的 `package.json` 文件中定义了一系列构建脚本，用于自动化执行不同环境下的构建任务。以下是一些主要的构建脚本：

```
// backend/package.json
{
  "scripts": {
    "dev": "nodemon app.js",
    "start": "node app.js",
    "test": "jest"
```

```
  }
}

// frontend/package.json
{
  "scripts": {
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "test": "playwright test"
  }
}
```

- **开发环境**：在开发环境中，使用 `npm run dev` 命令启动开发服务器，支持热更新，提高开发效率。
- **生产环境**：在生产环境中，先使用 `npm run build` 命令进行项目构建，然后使用 `npm start` 命令启动生产服务器。

2.2 自动化测试

2.2.1 测试框架选择

1. Jest

- **用途**：作为主要的测试运行器，用于编写和执行单元测试和集成测试。它提供了丰富的断言库和测试生命周期钩子，方便进行各种测试场景的编写。
- **配置**：在项目中，通过 `jest.setTimeout` 来设置测试的超时时间，确保长时间运行的测试不会无限期阻塞。
- **实际代码分析**：在 `tests/comic_images/api.comic.spec.test.js` 文件中，Jest 用于测试漫画文件的 API 接口。例如：

```
jest.setTimeout(20000); // 设置超时时间为20秒

describe('GET /api/comic', () => {
  // 测试解析 test.cbz 文件成功
  test('TC-1: 解析 test.cbz 成功', async () => {
    const res = await request(server)
      .get('/api/comic')
      .auth('admin', '123456')
      .query({ path: 'comic/test.cbz' });

    expect(res.status).toBe(200);
    expect(Array.isArray(res.body.pages)).toBe(true);
  });
});
```

这个测试用例验证了漫画文件解析 API 的正确性，包括状态码和返回数据格式的检查。

2. Supertest

- **用途**：用于测试 Express 应用的 HTTP 接口。可以方便地发起 HTTP 请求，并对响应进行断言。
- **使用场景**：在 API 接口测试中，通过 Supertest 模拟各种请求，验证接口的正确性。
- **实际代码分析**：在 `tests/comic_images/api.image.spec.test.js` 文件中，Supertest 用于测试图片文件的 API 接口。例如：

```
describe('GET /api/images', () => {
  // 测试正常获取图片列表
  test('TI-1: 正常获取图片列表', async () => {
    const res = await request(server)
      .get('/api/images')
      .auth('admin', 'admin123')
      .query({ dir: 'images' });

    expect(res.status).toBe(200);
    expect(Array.isArray(res.body.images)).toBe(true);
  });
});
```

这个测试用例使用 Supertest 发起 HTTP 请求，验证了图片列表 API 的正确性。

3. Playwright

- **用途**：用于编写端到端（E2E）测试，模拟用户在浏览器中的操作。可以进行页面元素的查找、点击、输入等操作，并验证页面的状态和响应。
- **使用场景**：在图像预览功能测试中，使用 Playwright 模拟用户登录、目录和文件的点击操作，验证图像预览功能的正确性。
- **实际代码分析**：在 `tests/comic_images/comic-preview.spec.ts` 文件中，Playwright 用于测试漫画预览功能。例如：

```
test('TF-05:CBZ 漫画预览功能测试', async ({ page }) => {
    await login(page);
    await page.waitForLoadState('networkidle');
    await clickDirectory(page, 'comic');
    await page.waitForLoadState('networkidle');
    await clickFile(page, 'test.cbz');

    // 确认打开了图片预览
    await expect(page.getByRole('img', { name: 'Page 1' })).toBeVisible();

    // 浏览图片测试
    await page.keyboard.press('ArrowRight');
    await page.waitForTimeout(300);
    await page.keyboard.press('ArrowLeft');
});
```

这个测试用例模拟了用户登录、进入漫画目录、打开漫画文件以及浏览漫画页面的操作，验证了漫画预览功能的正确性。

2.3 自动化部署

2.3.1 部署流程

项目采用持续集成/持续部署（CI/CD）流程，使用 GitHub Actions 实现自动化部署。部署流程如下：

1. 当开发人员将代码推送到 GitHub 仓库的特定分支时，触发 GitHub Actions 工作流。
2. GitHub Actions 自动拉取代码，并在虚拟环境中安装项目依赖。
3. 执行自动化测试，确保代码质量。
4. 如果测试通过，自动部署到目标环境。

2.3.2 配置文件

GitHub Actions 的配置如下：

```yaml
name: CI/CD Pipeline

on:
  push:
    branches:
      - main

jobs:
  build-and-test:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Set up Node.js
        uses: actions/setup-node@v2
        with:
          node-version: 18

      - name: Install backend dependencies
        run: |
          cd backend
          npm install

      - name: Install frontend dependencies
        run: |
          cd frontend
          npm install

      - name: Run backend tests
        run: |
          cd backend
          npm test

      - name: Run frontend tests
        run: |
          cd frontend
          npm test

  deploy:
    needs: build-and-test
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v2

      - name: Deploy to production
        run: |
          # 部署脚本，根据实际情况进行配置
          echo "Deploying to production..."
```

这个配置文件定义了一个 CI/CD 工作流，当代码推送到 main 分支时，会自动执行测试和部署操作。

三、协作化手段

3.1 版本控制

3.1.1 工具选择

项目使用 Git 作为版本控制系统，结合 GitHub 作为代码托管平台。

3.1.2 Git 操作流程

开发人员在进行代码开发时，应遵循以下 Git 操作流程：

1. **克隆仓库**：将远程仓库克隆到本地开发环境。

```
git clone https://github.com/Kobayashi2003/SimpleFileServer.git
cd SimpleFileServer
```

2. **创建分支**：在开始新的开发任务之前，创建一个新的分支，避免直接在主分支上进行修改。

```
git checkout -b feature/new-feature  # 创建并切换到新分支
```

3. **提交代码**：在完成一部分工作后，将修改提交到本地仓库。

```
git add .  # 将所有修改添加到暂存区
git commit -m "Add new feature: upload files"  # 提交修改并添加描述信息
```

4. **推送分支**：将本地分支推送到远程仓库。

```
git push origin feature/new-feature  # 将本地分支推送到远程仓库
```

5. **拉取更新**：在开始新的工作之前，确保本地分支是最新的。

```
git pull origin main  # 从远程仓库拉取最新的主分支代码
```

6. **合并分支**：当开发任务完成并通过测试后，将分支合并到主分支。

```
git checkout main  # 切换到主分支
git merge feature/new-feature  # 将新功能分支合并到主分支
git push origin main  # 将合并后的主分支推送到远程仓库
```

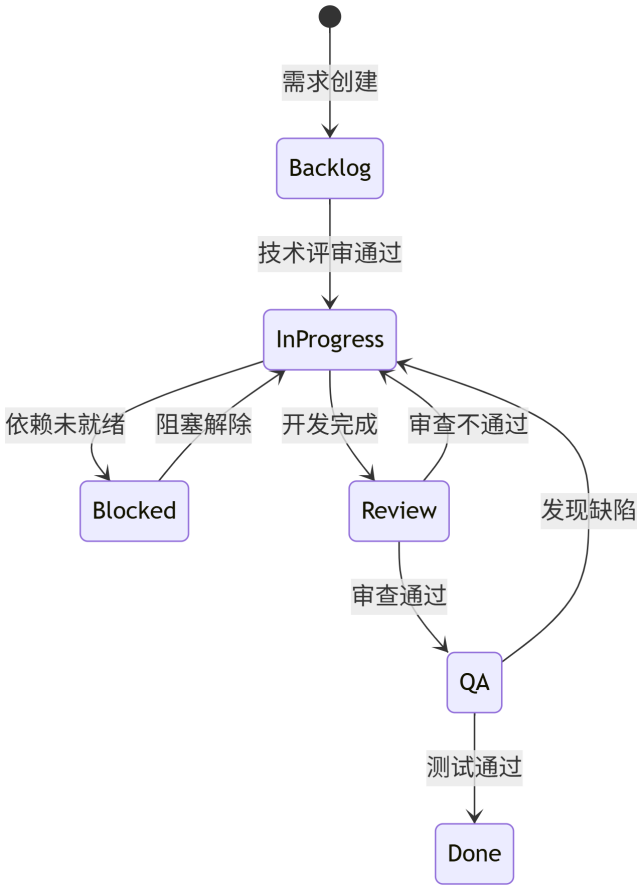
3.2 代码审查

3.2.1 审查工具

使用 GitHub 的 Pull Request（PR）功能进行代码审查。当开发人员完成一个功能或修复一个问题后，创建一个 PR 请求将代码合并到目标分支。

3.2.2 审查流程

1. **创建 PR**：开发人员在完成代码开发并推送到远程分支后，在 GitHub 上创建一个新的 PR。
2. **描述变更**：在 PR 中详细描述本次变更的内容、目的和实现方式。
3. **分配审查人员**：将 PR 分配给团队中的其他成员进行审查。
4. **代码审查**：审查人员对代码进行审查，检查代码质量、功能实现和潜在问题，并提出意见和建议。
5. **修改代码**：开发人员根据审查意见对代码进行修改，并提交更新。
6. **重新审查**：审查人员对修改后的代码进行再次审查，直到满意为止。
7. **合并 PR**：当审查通过后，由指定的人员将 PR 合并到目标分支。



3.3 项目管理与协作工具

3.3.1 工具选择

使用 GitHub Issues 进行项目任务管理和缺陷跟踪。团队成员可以在 Issues 中创建任务、分配任务、跟踪任务进度和记录缺陷信息。

3.3.2 协作流程

1. **任务创建**：项目负责人或团队成员在 GitHub Issues 中创建任务，描述任务的详细信息，如任务描述、优先级、截止日期等。
2. **任务分配**：将任务分配给具体的开发人员。
3. **任务跟踪**：开发人员在开发过程中更新任务的状态，如进行中、已完成等。
4. **缺陷管理**：当发现缺陷时，创建一个新的 Issue 记录缺陷信息，包括缺陷描述、复现步骤、严重程度等，然后分配给相关人员进行修复。

四、结论

通过采用上述软件工程化手段，SimpleFileServer 项目在自动化和协作化方面取得了一定的成效。自动化构建、测试和部署提高了开发效率和软件质量，减少了人为错误；而版本控制、代码审查和项目管理工具的使用促进了团队成员之间的有效沟通与协作，确保项目的顺利进行。在未来的开发过程中，可以根据项目的实际需求进一步优化和完善这些软件工程化手段。