

link: null
title: 珠峰架构师成长计划
description: 代码
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=147 sentences=242, words=1315

1. TS中的工程化

- 前端工程化就是通过流程规范化、标准化提升团队协作效率
- 通过组件化、模块化提升代码质量
- 使用构建工具、自动化工具提升开发效率
- 编译 => 打包(合并) => 压缩 => 代码检查 => 测试 => 持续集成

2. 编译参数

- 编译参数 (<https://www.tslang.cn/docs/handbook/compiler-options.html>)

参数 功能 rootDir,outDir 配置输入输出目录 lib.jsx 添加基础库的能力 noImplicitAny,strict 更严格的模式 module,target,declaration,sourceMap 控制输出的内容 watch 观察模式 allowJs 允许编译javascript文件

3. 代码检查

3. 代码检查

- ESLint 是一款插件化的 JavaScript 静态代码检查工具，ESLint 通过规则来描述具体的检查行为
- [eslint](https://eslint.org) (<https://eslint.org>)

3.1 模块安装

```
npm i eslint typescript @typescript-eslint/parser @typescript-eslint/eslint-plugin --save-dev
```

3.2 eslintrc配置文件

- [英文rules](https://eslint.org/docs/rules/) (<https://eslint.org/docs/rules/>)
- [中文rules](https://cn.eslint.org/docs/rules/) (<https://cn.eslint.org/docs/rules/>)
- 需要添加 parserOptions以支持模块化的写法
- .eslintrc.js

```
module.exports = {  
  "parser": "@typescript-eslint/parser",  
  "plugins": ["@typescript-eslint"],  
  "rules": {  
    "no-var": "error",  
    "no-extra-semi": "error",  
    "@typescript-eslint/indent": ["error", 2]  
  },  
  "parserOptions": {  
    "ecmaVersion": 6,  
    "sourceType": "module",  
    "ecmaFeatures": {  
      "modules": true  
    }  
  }  
}
```

3.3 npm命令

```
"scripts": {  
  "start": "webpack",  
  "build": "tsc",  
  "eslint": "eslint src --ext .ts",  
  "eslint:fix": "eslint src --ext .ts --fix"  
}
```

代码

```
var name2 = 'zhuFeng';;  
if(true){  
  let a = 10;  
}
```

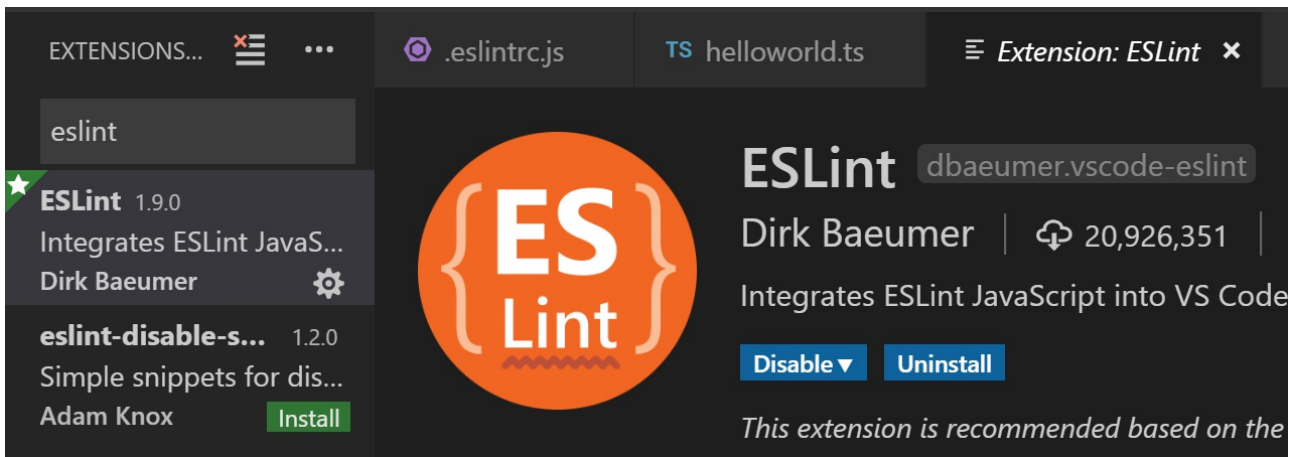
检查结果

```
src\helloworld.ts  
1:1  error  Unexpected var, use let or const instead  no-var  
1:23  error  Unnecessary semicolon                    no-extra-semi  
1:24  error  Unnecessary semicolon                    no-extra-semi  
3:1  error  Expected indentation of 2 spaces but found 4  @typescript-eslint/inden
```

3.4 配置自动修复

- 安装vscode的eslint插件
- 配置参数 .vscode/settings.json

```
{  
  "eslint.autoFixOnSave": true,  
  "eslint.validate": [  
    "javascript",  
    "javascriptreact",  
    {  
      "language": "typescript",  
      "autoFix": true  
    },  
    {  
      "language": "typescriptreact",  
      "autoFix": true  
    }  
  ]  
}
```



4. Git Hooks 检查

- Git 基本已经成为项目开发中默认的版本管理软件，在使用 Git 的项目中，我们可以为项目设置 Git Hooks 来帮我们在提交代码的各个阶段做一些代码检查等工作
- 钩子 (Hooks) 都被存储在 Git 目录下的 hooks 子目录中。也就是绝大部分项目中的 .git/hooks 目录
- 钩子 (<https://git-scm.com/book/zh/v2/%E8%87%AA%E5%AE%9A%E4%B9%89-Git-Git-%E9%92%A9%E5%AD%90>)分为两大类，客户端和服务器的
 - 客户端钩子主要被提交和合并这样的操作所调用
 - 而服务器端钩子作用于接收被推送的提交这样的联网操作，这里我们主要介绍客户端钩子

4.1 pre-commit

- pre-commit 就是在代码提交之前做些东西，比如代码打包，代码检测，称之为钩子 (hook)
- 在commit之前执行一个函数 (callback)。这个函数成功执行完之后，再继续commit，但是失败之后就阻止commit
- 在.git->hooks>下面有个pre-commit.sample*，这个里面就是默认的函数(脚本)样本

4.2 安装pre-commit

```
npm install pre-commit --save-dev
```

4.3 配置脚本

```
"scripts": {
  "build": "tsc",
  "eslint": "eslint src --ext .ts",
  "eslint:fix": "eslint src --ext .ts --fix"
},
"pre-commit": [
  "eslint"
]
```

如果没有在 .git->hooks目录下生成 pre-commit文件的话，则要手工创建 node ./node_modules/pre-commit/install.js

5. 单元测试

- Mocha是现在最流行的JavaScript测试框架之一，在浏览器和Node环境都可以使用。
- 所谓“测试框架”，就是运行测试的工具。通过它，可以为JavaScript应用添加测试，从而保证代码的质量

5.1 安装

```
cnpm i mocha @types/mocha chai @types/chai ts-node @types/node --save-dev
```

5.2 sum.ts

```
export default function sum(x:number,y:number):number{
  return x + y;
}
```

```
export default function sum(x:number,y:number) {
  return (x+y).toFixed(10);
}
```

5.3 sum.test.ts

- 通常，测试脚本与所要测试的源码脚本同名，但是后缀名为.test.js（表示测试）或者.spec.js（表示规格）。比如，add.js的测试脚本名字就是add.test.js
- 测试脚本里面应该包括一个或多个describe块，每个describe块应该包括一个或多个it块
- describe块称为“测试套件” (test suite)，表示一组相关的测试。它是一个函数，第一个参数是测试套件的名称（“加法函数的测试”），第二个参数是一个实际执行的函数。
- it块称为“测试用例” (test case)，表示一个单独的测试，是测试的最小单位。它也是一个函数，第一个参数是测试用例的名称（“1 加 1 应该等于 2”），第二个参数是一个实际执行的函数。

```
import sum from '../src/sum';
import * as assert from 'assert';
import * as chai from 'chai'
describe('test sum', ()=>{
  it('1+1=2', ()=>{
    assert.equal(sum(1,1),2);
  })
  it('2+2=4', ()=>{
    chai.expect(2+2).to.be.equal(4);
  })
});
```

```
it('0.1+0.2=0.3', ()=>{
  assert.equal(sum(.1, .2), .3);
})
```

5.4 断言库

- 所谓“断言”，就是判断源码的实际执行结果与预期结果是否一致，如果不一致就抛出一个错误
- 所有的测试用例（it块）都应该含有一句或多句的断言。它是编写测试用例的关键。断言功能由断言库来实现，Mocha本身不带断言库，所以必须先引入断言库

5.4.1 assert

- `assert.equal` 用于判断两个值是否相等

5.4.2 chai

```
expect(1 + 1).to.be.equal(2);
expect(1 + 1).to.be.not.equal(3);
let a = {name:'zhufeng'};let b = {name:'zhufeng'};
expect(a).to.be.deep.equal(b);

expect(true).to.be.ok;
expect(false).to.not.be.ok;

expect('zhufeng').to.be.a('string');
expect({name:'zhufeng'}).to.be.an('object');
function Person(){}
let person = new Person();
expect(person).to.be.an.instanceof(Person);

expect([1,2,3]).to.include(2);
expect('zhufeng').to.contain('feng');
expect({ name: 'zhufeng', age: 9 }).to.include.keys('name');

expect([]).to.be.empty;
expect('').to.be.empty;
expect({}).to.be.empty;

expect('zhufengnodejs@126.com').to.match(/^zhufeng/);
```

头部是expect方法，尾部是断言方法，比如equal、a/an、ok、match等。两者之间使用to或to.be连接

5.5 指定测试脚本文件

- mocha命令后面紧跟测试脚本的路径和文件名，可以指定多个测试脚本。
- Mocha默认运行test子目录里面的测试脚本。所以，一般都会把测试脚本放在test目录里面，然后执行mocha就不需要参数了
- 加上--recursive参数，就会执行test子目录下面所有的层的测试用例
- 命令行指定测试脚本时，可以使用通配符，同时指定多个文件

```
mocha spec/{a,b}.js      执行spec目录下面的a.js和b.js
mocha test*.js           执行test目录下面任何子目录中、文件后缀名为js的测试脚本
```

5.6 tsconfig.json

```
{
  "compilerOptions": {
    "module": "commonjs"
  }
}
```

5.7 package.json

- mocha命令后面紧跟测试脚本的路径和文件名，可以指定多个测试脚本
- Mocha默认运行 test子目录里面的测试脚本。所以，一般都会把测试脚本放在 test目录里面，然后执行mocha就不需要参数了
- 加上--recursive参数可以保证子目录下面所有的测试用例都会执行

```
"scripts": {
  "test": "set TS_NODE_COMPILER_OPTIONS='{\"module\": \"commonjs\"}' && mocha --require ts-node/register --watch --watch-extensions ts test/**/*",
},
```

如果"module": "es2015"的话则需要在命令行中配置module规范

5.8 配置文件mocha.opts

- Mocha允许在test目录下面，放置配置文件mocha.opts，把命令行参数写在里面

```
--require ts-node/register
--watch
--watch-extensions ts
test/sum.test.ts
```

5.9 方法调用

- Sinon 是用于 JavaScript 的测试框架，适用于任何单元测试框架。
- Sinon 将测试为三种类型：
 - Spies: 提供有关函数调用的信息，而不会影响其行为
 - Stubs: 类似于 Spies，但完全取代了功能。这样就可以使存根函数做任何你喜欢的事情 - 抛出异常，返回一个特定的值等等
 - Mocks: 通过组合 Spies 和 Stubs，可以更轻松地替换整个对象

监控函数执行 真正执行原函数 替换原有实现 提供行为描述 spy 是 stub 是 mock 是 是

```
npm install sinon @types/sinon -D
```

5.9.1 要测试的类

```
src\todo.ts
```

```
export default class Todos {
  private todos:string[]=[]
  private store:any
  constructor(store:any) {
    this.store=store;
  }
  add(todo:string) {
    this.todos.push(todo);
  }
  print() {
    console.log(this.todos);
  }
  save() {
    this.store.save(this.todos);
  }
}
```

5.9.2 spy

- 提供有关函数调用的信息，而不会影响其行为

```
import * as sinon from "sinon";
import * as chai from "chai";
import Todos from "../src/todos";
describe("测试 Todos", () => {
  it("spy print", () => {
    let store = {save() {}};
    const t = new Todos(store);

    sinon.spy(console, "log");
    t.add("eat");
    t.add("sleep");
    t.print();

    chai.expect(console.log.calledOnce).to.be.true;
  });
});
```

5.9.3 stub

- 类似于 spy，但完全取代了功能。这样就可以使存根函数做任何你喜欢的事情 - 抛出异常，返回一个特定的值等等

```
import * as sinon from "sinon";
import { expect } from "chai";
import Todos from "../src/todos";
describe("测试 Todos", () => {
  it("stub", () => {
    let store = {save() {}};
    const todos = new Todos(store);

    const stubAdd = sinon.stub(todos, "add").callsFake(() => {});

    stubAdd("eat");

    todos.print();

    expect(stubAdd.calledOnce).to.be.true;
  });
});
```

5.9.4 mock

- 通过组合 Spies 和 Stubs，可以更轻松地替换整个对象

```
import * as sinon from "sinon";
import { expect } from "chai";
import Todos from "../src/todos";
describe("测试 Todos", () => {
  it("mock", () => {
    let store = {save() {}};
    const todos = new Todos(store);

    const mock = sinon.mock(console);

    mock.expects("log").calledOnce;
    todos.add("eat");
    todos.print();

    mock.verify();
  });
});
```

6. 为JS添加TS支持

6.1 sum.js

src\sum.js

```
export default function sum(x,y) {
  return (x+y).toFixed(10);
}
```

6.3 sum.test.ts

test\sum.test.ts

```
import sum from '../src/sum';
import assert from 'assert';
describe('test sum', () => {
  it('1+2=3', () => {
    assert.equal(sum(1,2),3);
  })
  it('0.1+0.2=0.3', () => {
    sum(1,2);
    assert.equal(sum(0.1,0.2),0.3);
  })
});
```

6.3 tsconfig.json

```
{
  "compilerOptions": {
+   "checkJs": true, //允许代码中使用JS
+   "checkJs": true, //可以对JS进行类型检查
  }
```

7. 持续集成

- [Travis CI \(https://travis-ci.com\)](https://travis-ci.com) 提供的是持续集成服务（Continuous Integration，简称 CI）。它绑定 Github 上面的项目，只要有新的代码，就会自动抓取。然后，提供一个运行环境，执行测试，完成构建，还能部署到服务器
- 持续集成指的是只要代码有变更，就自动运行构建和测试，反馈运行结果。确保符合预期以后，再将新代码集成到主干
- 持续集成的好处在于，每次代码的小幅变更，就能看到运行结果，从而不断累积小的变更，而不是在开发周期结束时，一下子合并一大块代码

7.1 登录并创建项目

- [Travis CI \(https://travis-ci.com\)](https://travis-ci.com) 只支持 Github, 所以你要拥有 GitHub 帐号
- 该帐号下面有一个项目, 面有可运行的代码, 还包含构建或测试脚本
- 你需要激活了一个仓库, Travis 会监听这个仓库的所有变化

7.2 .travis.yml

- Travis 要求项目的根目录下, 必须有一个 .travis.yml 文件。这是配置文件, 指定了 Travis 的行为
- 该文件必须保存在 Github 仓库里面, 一旦代码仓库有新的 Commit, Travis 就会去找这个文件, 执行里面的命令
- 这个文件采用 YAML 格式。下面是一个最简单的 Node 项目的 .travis.yml 文件
- language 字段指定了默认运行环境, [所有的语言在此 \(https://docs.travis-ci.com/user/languages\)](https://docs.travis-ci.com/user/languages)
- node_js: "11" 表示不执行任何脚本, 状态直接设为成功

```
language: node_js
node_js:
  - "11"
install: npm install
script: npm test
```

7.3 运行流程

7.3.1 运行流程

- Travis 的运行流程很简单, 任何项目都会经过两个阶段
 - install 阶段: 安装依赖
 - script 阶段: 运行脚本

7.3.2 install

- install 字段用来指定安装脚本 install: npm install -g npm
- 如果不需要安装, 即跳过安装阶段, 就直接设为 true install: true

7.3.3 script

- script 字段用来指定构建或测试脚本

```
script: npm run build
```

7.3.4 钩子方法

Travis 为上面这些阶段提供了7个钩子

```
before_install  安装阶段之前执行
install         安装
before_script   脚本阶段之前执行
script         脚本阶段
after_success or after_failure 脚本成功或失败
[OPTIONAL] before_deploy 部署之前
[OPTIONAL] deploy 部署
[OPTIONAL] after_deploy 部署之后
after_script    脚本阶段之后
```

7.4 实战

7.4.1 安装 hexo

- [hexo \(https://hexo.io/zh-cn/docs/\)](https://hexo.io/zh-cn/docs/) 是一个快速、简洁且高效的博客框架

```
$ npm install -g hexo-cli
```

7.4.2 生成项目

```
hexo init zfblog
cd zfblog
npm install
```

7.4.3 启动项目

```
hexo generate
hexo server
hexo deploy
```

7.4.4 部署项目

```
$ cnpm install eslint hexo-deployer-git --save
cnpm generate
cnpm deploy
```

_config.yml

```
deploy:
  type: git
  repo: https://github.com/zhufengnodejs/zfblog2.git
  branch: gh-pages
  message:
```

<https://zhufengnodejs.github.io/zfblog2> (<https://zhufengnodejs.github.io/zfblog2>)



7.4.5 同步仓库

- 登录travis-ci.com (<https://travis-ci.com/>)选择同步仓库

7.4.6 设置仓库环境变量

变量名 含义 USERNAME git用户名 zhufengnodejs UESREMAIL git用户邮箱
zhufengnodejs@126.com (<mailto:zhufengnodejs@126.com>)

GH_TOKEN 用户生成的令牌 GH_REF 仓库地址 github.com/zhufengnodejs/zfblog3.git GH_BRANCH 推送的pages分支 gh-pages

 zhufengnodejs / zfblog2  build passing

[Current](#) [Branches](#) [Build History](#) [Pull Requests](#) [Settings](#)

General

☒ Build pushed branches ? ☐ Limit concurrent jobs ?

☒ Build pushed pull requests ?






Auto Cancellation

Auto Cancellation allows you to only run builds for the latest commits in the queue. This setting can be applied to builds for Branch builds and Pull Request builds separately. Builds will only be canceled if they are waiting.

☐ Auto cancel branch builds ☐ Auto cancel pull request builds

Environment Variables

Notice that the values are not escaped when your builds are executed. Special characters (for bash) should be escaped accordingly.

GH_BRANCH	 *****
GH_REF	 *****
GH_TOKEN	 *****
UESREMAIL	 *****
USERNAME	 *****

7.4.7 Github生成访问令牌 (即添加授权)

- 访问令牌的作用就是授权仓库操作权限
- Github>settings>Personal access tokens> Generate new token > Generate token> Copy Token

7.4.8 .travis.yml

```
language: node_js
node_js:
  - '11'
install:
  - npm install
script:
  - hexo g
after_script:
  - cd ./public
  - git init
  - git config user.name "${USERNAME}"
  - git config user.email "${UESREMAIL}"
  - git add -A
  - git commit -m "Update documents"
  - git push --force "https://${GH_TOKEN}@${GH_REF}" "master:${GH_BRANCH}"
branches:
  only:
    - master
```