**智能Web前端技术开发指南**

**前言**

南方智能创立三年，Web前端技术蓬勃发展，技术迭代更新频繁，在大环境下此指南希望能够让所有前端开发人员更加能够了解企业级开发，使得大家对开发的统一，协作友好都能够付出相应的努力，也希望能够从各方方面面提升技术水平能力。

**感谢**

感谢参与编写文档的伙伴们，分别是：曾沂轮、林润茏、马凤银、蒋思敏、严韵诗。

# **规范**

为什么需要规范?

1. 降低新成员融入项目的成本, 同时也一定程度避免挖坑
2. 提高开发效率，减少代码臃肿
3. 实现高度统一的代码风格，方便review

## **脚手架**

前端脚手架大部分作用是能够提供快速搭建工程化项目的基石，让开发快速进入业务应用开发中。

脚手架创建的工程具体目录结构、开发语言、开发框架、规范、构建、测试等基础内容，统一使用脚手架也是在开发前提环境下统一了环境，对企业协作开发来说至关重要。

### **vue-cli**

我们主框架Vue.JS的主基调中，vue-cli是我们一开始必备的脚手架，它里面包含了各种配置、开发/构建服务、预设内容。

相对团队而言过于开放的脚手架对团队维护成本较高，需要我们减少配置，比如对CSS预处理选择SCSS，构建使用Webpack，ESlint使用自定义扩展等等需求，这里就产生了团队特有的脚手架svi。

### **svi-cli**

SVI是我们智能前端的脚手架，基于Node.JS开发的工具，它的作用是为了统一前端的开发工程化，结合svi-templates仓库上的工程模板去使用。

SVI提供主要有两个命令：svi list (查看可用模板)、 svi init (创建工程命令)。

常用创建vue工程的命令：svi init vue-app my-project

SVI创建出来的工程，一定包含有ESlint/TSlint自主规范（详细看3检查规范），然后我们提供了各种业务上常用的模板（详细看2模板）。

其他脚手架都不在我们的体系中，像create-react-app、angular-cli都是各个JavaScript框架带有的一套解决方案，但我们选择的主框架是Vue所以他们不在我们使用范畴内。

首推使用是我们自主的svi开发工程，未来也不断维护完善让它的承担更多工作。

## **模板**

这里重点是介绍SVI上的模板，并解释包含什么内容可以提升到模板中。

SVI的模板大部分都和各种我们的场景及业务中出现频繁的，也有部分是我们基础平台应用的模板，比如有Smart Web和iScene的模板。



图2.1 已有模板列表

### **模板要求**

模板的特性应该拥有通用性或基础性。

基础性一般指模板具有提供开发基础环境，能够提供基础的环境配置，工程构建，代码规范检查等内容，或者能够提供平台基础模板。

通用性一般指在一定范围内普遍性存在的，能够提供给80%的开发者使用并达到满足需求，例如Smart Web的基础模板具有OA系统模板通用性。

### **如何创建模板**

该说明用途是让有需求创建模板的开发者理解。

#### **模板存放仓库**

因为模板是用在svi-cli脚手架的，它是固定从gitlab仓库svi-templates群组中拉取，所有模板是放在该群组中，地址是：<http://172.16.11.133/svi/Templates>。

#### **模板结构要求**

模板必须含有： template, README.md, meta.js, package.json

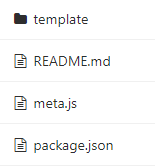


图2.2 模板内容结构

template包含我们模板工程内容最终创建给开发者使用的内容

README.md 是对该模板的描述，用法用途等信息

meta.js 是模板交互的文件配置，比如变量名、过滤文件、判断条件等作用

package.json 是模板仓库说明文件

#### **模板变量赋值**

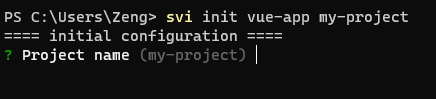
如果让创建模板时使用自定义项目名称呢？这时需要用到里面meta.js设计和template的作用。

例如，meta.js中配置



其中renderFiles字段是说明要渲染替换变量的文件列表，prompts是命令交互中的字段使用的是inquirer的prompt，比如里面的name就是我们定义的项目名称字段，并且它是必填项(required: true)。

字段name在命令的交互是这样的：

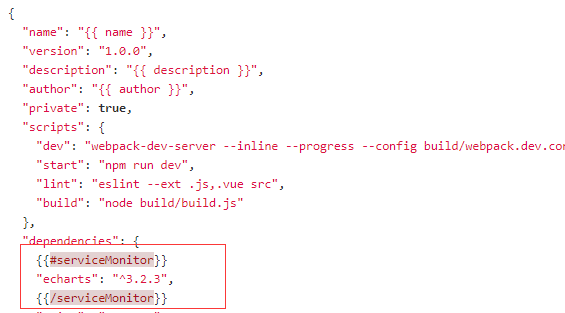


而template中要改变的地方就是renderFiles的文件列表，比如template/package.json使用的handlebars语言模板占位符{{xxx}}方式，如下图



在这样的配置下svi-cli程序会自动的寻找和替换相对应的变量，从而去赋值。

基本上handlebars的用法都是可用的，比如条件判断标签，如下图



## **检查规范**

规范即是开发的利器，也是审核代码的标准。

遵循代码的规范是每个人要做的。

### **代码检查工具**

#### **ESlint**

ESlint是JavaScript的语法静态检查工具，强大并有完善的检查工作，每个工程中都会去安装eslint依赖，并且通过各种配置去约束项目工程代码。

我们通过多个项目的实践经验得出自己的eslint配置 @smart/eslint-config-standard，是我们对代码唯一规范的检查配置，所有前端项目都应该包含。

* + 1. 使用@smart/eslint-config-standard

第一步安装：

npm install @smart/eslint-config-standard –registry http://172.16.11.131:4873

然后.eslintrc.js配置文件中extends字段添加即可

{

'extends': [

'@smart/standard'

],

}

如果要结合其他扩展插件使用，那么还需要像eslint-plugin-vue，eslint-plugin-prettier，@typescript-eslint/eslint-plugin等插件。

#### **TSlint**

TypeScript的检查之前一般是推荐使用TSlint，现在也是统一使用ESlint，可以参考@typescript-eslint/eslint-plugin和@typescript-eslint/parser插件工作。

{

"parser": "@typescript-eslint/parser",

"plugins": ["@typescript-eslint"],

}

推荐使用我们对typescript的规范扩展进行配置@smart/eslint-config-typescript，里面包含了对@typescript-eslint/eslint-plugin等相关配置进行扩展自定义的规范内容

extends: [

'@smart/eslint-config-typescript'

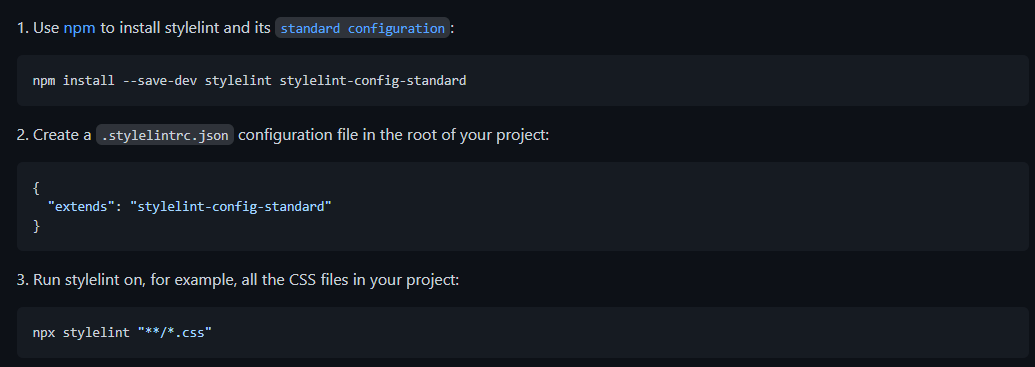
]

有兴趣可以打开仓库了解，地址：http://172.16.11.133/front-end/smart/eslint-config-typescript

#### **Stylelint**

样式的规范检查，主要需要作用于CSS/SCSS/LESS等样式的规范编写，使用是stylelint来检查，并且结合stylelint-config-standard或者stylelint-config-recommended扩展来检查。

使用方式：



详细请看，https://github.com/stylelint/stylelint/blob/master/docs/user-guide/get-started.md

暂时未有自定义自主的检查样式扩展。

#### **Prettier**

prettier是一个自动格式化代码风格的工具，它也可以结合eslint检查代码风格是否一致规范。

比如使用eslint-prettier-plugin可以在做eslint检查时同时对代码风格进行检查。

直接使用我们基于eslint-config-prettier扩展定义的@smart/eslint-config-prettier，使用配置方式特别简单，只需在.eslintrc.js配置文件中：

module.exports = {

extends: [

'@smart/prettier',

]

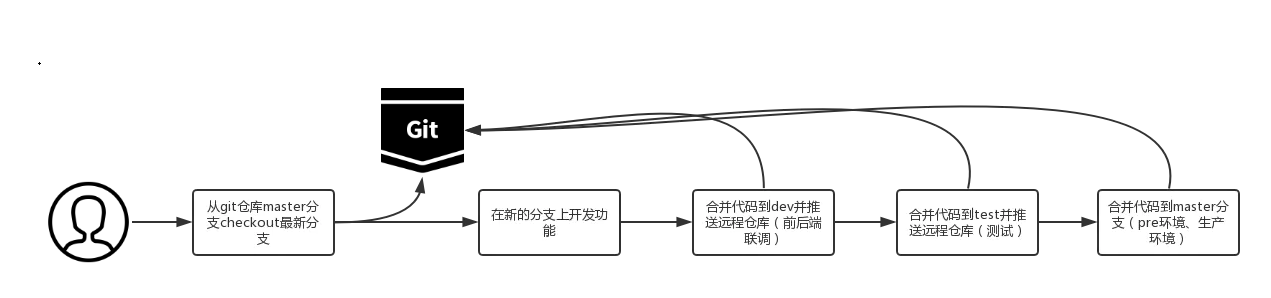
}

如果不满足再进行扩展自定义 .prettierrc.js 配置即可。

## **GIT规范**

公司使用的是Gitlab仓库管理代码，也是为了提高开发协作效率，git的操作和规范对团队也是比较重要的，在我们开发日常中都在不断的使用需要遵守规范。

一个比较完善git开发协作流程图



### **分支模型**

不同分支不同作用，能够快速响应各种各样状态。

规范的认识分支的作用，对于仓库的维护也是简单了许多，比如我们定义master(主干)分支是稳定版本分支，除了管理员其他开发人员都应无权限直接维护该分支，只能通过请求合并(MR/PR)维护的方式。

下面讲解有哪些分支和它们的作用：

#### **master分支**

master分支表示一个稳定的发布版本. 对应系统的大版本。

1. 代码会跟随工作版本迭代, 在dev分支测试稳定后, 会合并到master分支, 并使用tag标记应用版本和对应的工作版本。
2. 由项目负责人进行通过d分支审核合并迭代, 普通开发者应当没有权限修改该分支。

#### **dev分支**

开发者主要工作的分支, 最新的特性或bug修复都会提交到这个分支，dev分支是可变的，开发者如果在该分支进行了提交，在push到远程之前应该先pull一下， 并尽量使用rebase模式，保证分支的简洁。

1. 在dev分支中也可能会经历发布过程（发布到测试环境）, 例如bug修复版本，这里同样可以使用tag来标记这些发布，例如test\_v0.1.1。
2. 如果是直接在开发分支上进行开发，在推送到远程之前，应该使用git pull形式更新本地分支。
3. 如果是从其他分支开发再请求合并到dev分支，在推送到远程合并前，请先使用git rebase来更新本地分支。

#### **feature分支**

涉及多人协作或者大功能的开发, 应该从dev分支checkout出独立的feature分支, 避免干扰dev分支，完成后再请求合并(MR/PR)到dev分支。

1. 场景:

* 涉及多人协作: 团队多个成员在同一个项目下负责开发不同的功能, 这时候每个成员在自己的feature分支独立开发。
* 大功能开发: 大功能开发跨越周期比较长, 需要多次迭代才会稳定. 这时候应该在独立的分支上开发. 方便跟踪历史记录, 也免于干扰dev分支的迭代和发布。

1. 命名规范

* feature-name: name是功能名称
* feature-version: 这也是团队常见的模式, 当无法使用一个功能名称来描述时, 可以使用版本号作为功能。

1. 合并时机

* 当feature分支迭代稳定, 并通过测试后, 合并到dev分支. 合并到dev后, feature分支的生命周期就结束了(应当删除掉)， 后续bug修复和功能优化直接在dev开发
* 当多个feature分支需要合并对外发布临时版本时，合并到preview分支。⚠️这种情况不应该合并到dev分支, 因为feature分支可能还不稳定或未完成. 比如为了联调某些功能。

1. 合并方式

* 不建议使用fast-forward合并方式，这样可以在分支图上查看到分支历史。

#### **preview分支**

临时的预览分支， preview分支用于临时合并feature分支，这其中可能会修复某些bug或者冲突，可以选择性地将这些提交cherrypick回feature分支. 当预览结束后就可以销毁(删除)preview分支。

#### **release分支**

release分支主要是临时创建修改某个发布版本的bug修复分支。

1. 命名规范

* release-{version} 外部人员只会关注版本

1. 如何修复

* 如果对应bug可以在dev分支直接被修复, 可以先提交到dev分支(或者已经修复了), 然后再cherrypick到release分支
* 如果bug在新版本无法复现. 比如新版本升级了依赖. 那么在release分支直接修复即可。

1. 发布

* bug修复测试通过后，可以直接发布到生产环境，并且重新打包一个tag标记。

### **版本规范**

前端项目使用语义化版本进行发布管理。

版本格式：主版本号.次版本号.修订号，版本号递增规则如下：

* 主版本号：当你做了不兼容的 API 重大修改；
* 次版本号：当你做了向下兼容的功能性新增；
* 修订号：当你做了向下兼容的问题修正。

先行版本号及版本编译元数据可以加到“主版本号.次版本号.修订号”的后面，作为延伸。

### **提交规范**

#### **提交信息规范**

一个好的提交信息, 会帮助你提高项目的整体质量。

为什么这么说？

* 格式统一的提交信息可以帮助自动化生成changelog更新日志。
* 版本库不只是存放代码的仓库, 也记录项目的开发记录. 这些记录应该可以帮助后来者快速地学习和回顾代码. 也应该方便其他协作者review你的代码。

原则: 半年后, 你能看懂你的commit做了什么内容。

必要信息

* 为什么进行这次提交?
* 提交改变了什么, 让其他reviewer更容易审核代码和忽略无关的改变
* 如何解决的问题?
* 问题是什么导致的？
* 简短说明使用什么方式, 策略, 修复了问题。
* 变化可能影响哪些地方。
* 说明变动功能的细节。一个提交不应该做超过2个功能的变动。

#### **提交格式**

我们采用基于angular的提交规范, 简化和扩展了不同点。

格式内容包含：

<type>(<scope>): <subject> // header头部

<body>

<footer>

**header**

如果提交时使用Feat或者Fix，在已发布的版本里这些提交信息应该出现在CHANGELOG.md文件中。

* type: 说明commit的类别, 让阅读者更快地区分提交的类型,允许以下类型:
* Feat: 新功能/新特性
* Fix: 修复了bug，贴上issues或禅道bug号
* Docs: 文档更新
* Style: 优化项目结构或者代码格式
* Test: 测试更新
* Chore: 日常更新. 如升级依赖, 升级构建工具
* Perf: 性能优化
* Revert: revert之前的commit

git revert 命令用于撤销之前的一个提交, 并在为这个撤销操作生成一个提交

* Refactor: 代码重构
* Build或Release: 构建或发布版本
* CI: 持续集成
* Safe: 修复安全问题
* scope: 可选. 说明提交影响的范围. 例如样式, 后端接口, 逻辑层等等
* subject: 提交目的的简短描述, 动词开头, 不超过80个字符. 不要为了提交而提交

**body**

可选. 对本次提交的详细描述. 如果变动很简单, 可以省略

**footer**

可选. 只用于说明不兼容变动(break change)和关闭 Issue(如果使用gitlab上的Issuse，#1542)或禅道BUG号(#zentao203)

**模板和示例**

# 50-character subject line

#

# 100-character wrapped longer description. This should answer:

#

# \* Why was this change necessary?

# \* How does it address the problem?

# \* Are there any side effects?

#

# Include a link to the ticket, if any.

**跳过CI**

如果项目上配有gitlab-ci，可以在标题header上添加 [ci skip] 即可跳过本次提交CI。

**Commitlint 提交验证**

我们可以使用git钩子进行对我们的提交信息进行前置检查，使用工具是commitlint来进行检查。

使用@commitlint/cli和 @smart/commitlint-config 和husky配合来进行检查。

详细配置方式，查看http://172.16.11.133/front-end/document/blob/master/standard/Git.md#commitlint-提交验证

**生成更新日志**

安装conventional-changelog-cli 和 @smart/conventional-changelog-smart

使用命令，即可方便生成出CHANGLOG.md更新日志文件

conventional-changelog -n ./node\_modules/@smart/conventional-changelog-smart/index.js -i CHANGELOG.md -s -r 0

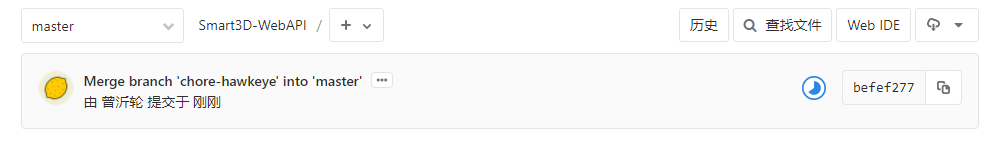
**Bug 处理**

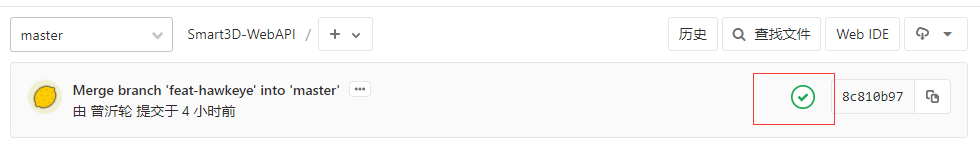
详细的Bug处理规则会在【测试指南】中说明，但这里说明是bug处理好在提交信息中可以记录，关闭禅道BUG可以在提交信息中追加 #zentao203 来联动bug，其中203是禅道bug号。

### **CI/CD**

#### **gitlab-ci**

测试环境在公司内部部署的gitlab-ci来进行持续集成部署。它能够实时的反馈在gitlab仓库中，让开发者都可以结果。





CI列表：



通过 .gitlab-ci.yml 配置文件来定义工作流，比如

before\_script:

- npm install --registry http://172.16.11.131:4873

stages:

- lint

- build

lint:

stage: lint

script:

- npm run lint

build:

stage: build

script:

- npm run release

only:

- dev

以上的配置包含lint和build两个阶段：

* lint检查阶段
* 所有push到版本库的代码都会跑这个阶段.
* build构建阶段
* 只有dev分支会跑构建阶段

#### **jenkins**

选择jenkins来做持续集成，它与gitlab没有强联动，只能监听仓库的变化无法把结果反馈到gitlab，建议开发的CI使用gitlab-ci，但它的好处是比如插件结合使用，比如我们部署使用docker打包或ssh连接发布等。

更详细内容请阅读[七. 部署指南 – Jenkins部署]

### **发布工作流**

#### **流程**

1. 合并到master主干，进行CI通过测试
2. 使用npm version命令提升package.json的版本号, 更新CHANGELOG.md
3. 打上tag, 提交推送
4. 可选. 合并到release分支
5. 构建，可以是CI上自动构建，也可以是本地构建

#### **交付**

更详细内容请阅读[七. 部署指南]

1. 把构建好的包跟随后端Jar/War包一起部署
2. 使用tomcat或nginx代理的http服务容器部署
3. 构建包含nginx的docker镜像方便部署

# **编码指南**

## **命名规范**

ECMAScript 规范中标识符采用驼峰大小写格式，驼峰命名法由小(大)写字母开始，后续每个单词首字母都大写。根据首字母是否大写，分为两种方式：

* Pascal Case 大驼峰式命名法：首字母大写。eg：StudentInfo、UserInfo、ProductInfo
* Camel Case 小驼峰式命名法：首字母小写。eg：studentInfo、userInfo、productInfo

标识符，则包括变量、函数名、类名、属性名和函数或类的参数，每个命名方法又略有不同。

1. **项目命名**

**命名规范：**推荐采用小写方式，以中划线分隔（因为中划线是搜索引擎更容易接受的方式）

**命名示例：**my-project-name

注：项目名称与产品一致，可以自己的特性比如iScene，Smart3D, SgeoC

1. **目录命名**

**命名规范：**采用小写方式，以驼峰或下划线分隔；有复数结构时，要采用复数命名法

**命名示例：** scripts、styles、images、dataModels/data\_Models

1. **文件命名**

**命名规范：**推荐采用名词性短语/动宾短语命名，如果是类使用大写开头驼峰形式，Angular组件使用用‘-’进行单词分割，Vue组件推荐使用大写开头驼峰形式。

**命名示例：** Profile.js, CreateVue.vue, disk-creation.html，add-disk-dialog.ts

1. **JS命名**

#### **变量**

**命名规范：**小驼峰式命名法，前缀应当是名词。(函数的名字前缀为动词，以此区分变量和函数)

**命名建议：**尽量在变量名字中体现所属类型，如：length、count等表示数字类型；而包含name、title表示为字符串类型。

**命名示例：**

// 好的命名方式

let maxCount = 10;

let tableTitle = ‘LoginTable’;

// 不好的命名方式

let setCount = 10;

let getTitle = ‘LoginTable’;

#### **常量**

**命名规范：**使用大写字母和下划线来组合命名，下划线用以分割单词.

**命名示例：**

const MAX\_COUNT = 10;

const URL = ‘https://3d.southsmart.com/#/home’;

#### **函数**

**命名规范：**小驼峰式命名法，前缀应当为动词。

**命名建议：**可使用常见动词约定

|  |  |  |
| --- | --- | --- |
| 动词 | 含义 | 返回值 |
| can | 判断是否可执行某个动作(权限) | 函数返回一个布尔值。true：可执行；false：不可执行 |
| has | 判断是否含有某个值 | 函数返回一个布尔值。true：含有此值；false：不含有此值 |
| is | 判断是否为某个值 | 函数返回一个布尔值。true：为某个值；false：不为某个值 |
| get | 获取某个值 | 函数返回一个非布尔值 |
| set | 设置某个值 | 无返回值、返回是否设置成功或者返回链式对象 |
| load | 加载某些数据 | 无返回值或者返回是否加载完成的结果 |

**命名示例：**

// 是否可阅读

function canRead(): boolean {

return true;

}

// 获取名称

function getName(): string {

return this.name;

}

* 给函数命名。这在做堆栈轨迹时很有帮助。

// bad

var log = function (msg) {

  console.log(msg);

};

// good

var log = function log(msg) {

  console.log(msg);

};

* 如果你的文件导出一个类，你的文件名应该与类名完全相同。

// file contents

class CheckBox {

  // ...

}

module.exports = CheckBox;

// in some other file

// bad

var CheckBox = require('./checkBox');

// bad

var CheckBox = require('./check\_box');

// good

var CheckBox = require('./CheckBox');

#### **类 & 构造函数**

**命名规范：**大驼峰式命名法，前缀为名称

**命名示例：**

class Person {

public name: string;

constructor(name) {

this.name = name;

}

}

const person = new Person(‘Marsy’);

* **类的成员**

类的成员包含：公共/私有属性和方法

**命名规范：**

公共属性和方法：跟变量和函数的命名一样。

私有属性和方法：前缀为\_(下划线)，后面跟【公共属性和方法】一样的命名方式。

**命名示例：**

class Person {

// 私有属性

private \_name: string;

constructor(name) { }

// 公共方法

getName() {

return this.\_name;

}

// 公共方法

setName(name) {

this.\_name = name;

}

}

const person = new Person();

person.setName(‘Marsy’);

person.getName(); // => Marsy’

#### **注释规范**

js支持三种不同类型的注释：行内注释、单行注释和多行注释。

* **行内注释**

**说明：**行内注释以两个斜线开始，以行尾结束。

**语法：**code // 这是行内注释（//与代码及注释文字之间保留一个空格）

**命名建议：**

* // 用来显示一个解释的评论
* // -> 用来显示表达式的结果
* // >用来显示 console 的输出结果

**命名示例：**

function test() { // 测试函数（显示一个解释的评论）

console.log('Hello World!'); // >Hello World!（显示打印输出的结果）

return 3 + 2; // ->5（显示表达式的结果）

}

* **单行注释**

**说明：**单行注释以两个斜线开始，以行尾结束。

**语法：**// 这是单行注释（//与注释文字之间保留一个空格）

**命名示例：**

// 调用了一个函数；1)单独在一行

setTitle();

* **多行注释**

**说明：**以 /\* 开头， \*/ 结尾

**语法：**/\* 注释说明 \*/（若开始/*和结束*/都在一行，推荐采用单行注释。若至少三行注释时，第一行为/*，最后行为*/，其他行以*开始，并且注释文字与*保留一个空格。）

**命名示例：**

/\*

\* 代码执行到这里后会调用setTitle()函数

\* setTitle()：设置title的值

\*/

setTitle();

* **函数(方法)注释**

**说明：**函数(方法)注释也是多行注释的一种，但是包含了特殊的注释要求，参照JSDoc

**语法：**

/\*\*

\* 函数说明

\* @关键字

\*/

|  |  |  |
| --- | --- | --- |
| 注释名 | 语法 | 示例 |
| @param | @param 参数名 {参数类型} 描述信息（参数信息） | @param name {String} 传入名称 |
| @return | @return {返回类型} 描述信息（返回值信息） | @return {Boolean} true:可执行;false:不可执行 |
| @author | @author 作者信息 [附属信息：如邮箱、日期]（此函数作者信息） | @author 张三 2015/07/21 |
| @version | @version XX.XX.XX（此函数版本号） | @version 1.0.3 |
| @example | @example 示例代码（演示函数的使用） | @example setTitle(‘测试’) |

常用注释关键字：(只列出一部分，并不是全部)

/\*\*

\* 合并Grid的行

\* @param grid {GridPanel} 需要合并的Grid

\* @param cols {Array} 需要合并列的Index(序号)数组；从0开始计数，序号也包含。

\* @param isAllSome {Boolean} ：是否2个tr的cols必须完成一样才能进行合并。true：完成一样；false(默认)：不完全一样

\* @return void

\* @author yourName 2020/01/18

\* @example

\* mergeCells(grid, cols)

\*/

function mergeCells(grid: GridPanel, cols: Number[], isAllSome: boolean = false) {

  // Do Something

}

#### **构造函数**

* 给对象原型分配方法，而不是使用一个新对象覆盖原型。覆盖原型将导致继承出现问题：重设原型将覆盖原有原型！

function Jedi() {

console.log('new jedi');

}

// bad

Jedi.prototype = {

  fight: function fight() {

    console.log('fighting');

  },

  block: function block() {

    console.log('blocking');

  }

};

// good

Jedi.prototype.fight = function fight() {

  console.log('fighting');

};

Jedi.prototype.block = function block() {

  console.log('blocking');

};

* 方法可以返回 this 来实现方法链式使用。

// bad

Jedi.prototype.jump = function jump() {

  this.jumping = true;

  return true;

};

Jedi.prototype.setHeight = function setHeight(height) {

  this.height = height;

};

var luke = new Jedi();

luke.jump(); // => true

luke.setHeight(20); // => undefined

// good

Jedi.prototype.jump = function jump() {

  this.jumping = true;

  return this;

};

Jedi.prototype.setHeight = function setHeight(height) {

  this.height = height;

  return this;

};

var luke = new Jedi();

 luke.jump().setHeight(20);

## **ES6指南**

ECMAScript 6.0（以下简称 ES6）是 JavaScript 语言的下一代标准，已经在 2015 年 6 月正式发布了。它的目标，是使得 JavaScript 语言可以用来编写复杂的大型应用程序，成为企业级开发语言。

1. **let和const**

* **let取代var**

ES6 提出了两个新的声明变量的命令：let和const，它们二者都是块作用域的。

// const and let only exist in the blocks they are defined in.

{

  let a = 1;

  const b = 1;

}

console.log(a); // ReferenceError

console.log(b); // ReferenceError

其中，let完全可以取代var，因为两者语义相同，而且let没有副作用（下面代码如果用var替代let，实际上就声明了两个全局变量，这显然不是本意。变量应该只在其声明的代码块内有效，var命令做不到这一点）。

'use strict';

if (true) {

let x = 'hello';

}

for (let i = 0; i < 10; i++) {

console.log(i);

}

var命令存在变量提升效用，let命令没有这个问题。（下面代码如果使用var替代let，console.log那一行就不会报错，而是会输出undefined，因为变量声明提升到代码块的头部。这违反了变量先声明后使用的原则）

'use strict';

if (true) {

console.log(x); // ReferenceError

let x = 'hello';

}

所以，建议不再使用var命令，而是使用let命令取代。

* **全局常量和线程安全**

在let和const之间，建议优先使用const，尤其是在全局环境，不应该设置变量，只应设置常量 (所有的函数都应该设置为常量) 。const优于let有以下几个原因：

1. const可以提醒阅读程序的人，这个变量不应该改变
2. const比较符合函数式编程思想，运算不改变值，只是新建值，而且这样也有利于将来的分布式运算
3. JavaScript 编译器会对const进行优化，所以多使用const，有利于提高程序的运行效率，也就是说let和const的本质区别，其实是编译器内部的处理不同。

// bad

var a = 1, b = 2, c = 3;

// good

const a = 1;

const b = 2;

const c = 3;

// best

const [a, b, c] = [1, 2, 3];

长远来看，JavaScript 可能会有多线程的实现（比如 Intel 公司的 River Trail 那一类的项目），这时let表示的变量，只应出现在单线程运行的代码中，不能是多线程共享的，这样有利于保证线程安全。

注意：在代码编写中，应养成习惯性将const变量放在一起，let变量放在一起

1. **字符串**

* 静态字符串一律使用单引号或反引号，不使用双引号。动态字符串使用反引号。

// bad

const a = "foobar";

const b = 'foo' + a + 'bar';

// acceptable 可接受的

const c = `foobar`;

// good

const a = 'foobar';

const b = `foo${a}bar`;

* 超过80个字符的字符串应该使用字符串连接换行 注：如果过度使用长字符串连接可能会对性能有影响。jsPerf & Discussion

// bad

var errorMessage = 'This is a super long error that was thrown because of Batman. When you stop to think about how Batman had anything to do with this, you would get nowhere fast.';

// bad

var errorMessage = 'This is a super long error that \

was thrown because of Batman. \

When you stop to think about \

how Batman had anything to do \

with this, you would get nowhere \

fast.';

// good

var errorMessage = 'This is a super long error that ' +

  'was thrown because of Batman.' +

  'When you stop to think about ' +

  'how Batman had anything to do ' +

  'with this, you would get nowhere ' +

  'fast.';

* 编程构建字符串时，使用字符串模板而不是字符串连接（模板给你一个可读的字符串，简洁的语法与适当的换行和字符串插值特性）。

// bad

function sayHi(name) {

  return 'How are you, ' + name + '?';

}

// bad

function sayHi(name) {

  return ['How are you, ', name, '?'].join();

}

// good

function sayHi(name) {

  return `How are you, ${name}?`;

}

1. **解构赋值**

* 使用数组成员对变量赋值时，优先使用解构赋值。

const arr = [1, 2, 3, 4];

// bad

const first = arr[0];

const second = arr[1];

// good

const [first, second] = arr;

* 函数的参数如果是对象的成员，优先使用解构赋值。

// bad

function getFullName(user) {

  const firstName = user.firstName;

  const lastName = user.lastName;

}

// good

function getFullName(obj) {

  const { firstName, lastName } = obj;

}

// best

function getFullName({ firstName, lastName }) {

}

* 如果函数返回多个值，优先使用对象的解构赋值，而不是数组的解构赋值。这样便于以后添加返回值，以及更改返回值的顺序。

// bad

function processInput(input) {

  return [left, right, top, bottom];

}

// good

function processInput(input) {

  return { left, right, top, bottom };

}

const { left, right } = processInput(input);

1. **对象**

* 使用对象字面量创建对象

// bad

var item = new Object();

// good

var item = {};

* 单行定义的对象，最后一个成员不以逗号结尾。多行定义的对象，最后一个成员以逗号结尾。

// bad

const a = { k1: v1, k2: v2, };

const b = {

  k1: v1,

  k2: v2

};

// good

const a = { k1: v1, k2: v2 };

const b = {

  k1: v1,

  k2: v2,

};

* 对象尽量静态化，一旦定义，就不得随意添加新的属性。如果添加属性不可避免，要使用Object.assign方法。

// bad

const a = {};

a.x = 3;

// if reshape unavoidable

const a = {};

Object.assign(a, { x: 3 });

// good

const a = { x: null };

a.x = 3;

* 如果对象的属性名是动态的，可以在创造对象的时候，使用属性表达式定义。

function getKey(k) {

  return `a key named ${k}`;

}

// bad

const obj = {

  id: 5,

  name: 'San Francisco',

};

obj[getKey('enabled')] = true;

// good

const obj = {

  id: 5,

  name: 'San Francisco',

  [getKey('enabled')]: true,

};

上面代码中，对象obj的最后一个属性名，需要计算得到。这时最好采用属性表达式，在新建obj的时候，将该属性与其他属性定义在一起。这样一来，所有属性就在一个地方定义了。

* 对象的属性和方法，尽量采用简洁表达法，这样易于描述和书写。

var ref = 'some value';

// bad

const atom = {

  ref: ref,

  value: 1,

  addValue: function (value) {

    return atom.value + value;

  },

};

// good

const atom = {

   ref,

value: 1,

addValue(value) {

     return atom.value + value;

  },

};

1. **数组**

* 使用字面量语法创建数组

// bad

const items = new Array();

// good

const items = [];

* 使用扩展运算符（...）拷贝数组，不要使用 Array.from、Array.of等数组的新的内置API，Array新api用于适合的场景。

// bad

const len = items.length;

const itemsCopy = [];

let i;

for (i = 0; i < len; i++) {

  itemsCopy[i] = items[i];

}

// good

const itemsCopy = [...items];

* 使用 Array.from 方法，将类似数组的对象转为数组。

const foo = document.querySelectorAll('.foo');

const nodes = Array.from(foo);

1. **函数**

* 使用函数声明而不是函数表达式(函数声明拥有函数名，在调用栈中更加容易识别。并且，函数声明会整体提升，而函数表达式只会提升变量本身。这条规则也可以这样描述，始终使用箭头函数来代替函数表达式)。

// bad

const foo = function () {

};

// good

function foo() {

}

注意：绝对不要在一个非函数块（if，while，等等）里声明一个函数，把那个函数赋给一个变量。浏览器允许你这么做，但是它们解析不同 注：ECMA-262 把 块 定义为一组语句，函数声明不是一个语句。

// bad

if (currentUser) {

  function test() {

    console.log('Nope.');

  }

}

// good

if (currentUser) {

  var test = function test() {

    console.log('Yup.');

  };

}

* 立即执行函数可以写成箭头函数的形式。

(() => {

  console.log('Welcome to the Internet.');

})();

* 那些使用匿名函数当作参数的场合，尽量用箭头函数代替。因为这样更简洁，而且绑定了 this。

// bad

[1, 2, 3].map(function (x) {

  return x \* x;

});

// good

[1, 2, 3].map((x) => {

  return x \* x;

});

// best

[1, 2, 3].map(x => x \* x);

* 箭头函数取代Function.prototype.bind，不应再用 self/\_this/that 绑定 this。

// bad

const self = this;

const boundMethod = function(...params) {

  return method.apply(self, params);

}

// acceptable

const boundMethod = method.bind(this);

// best

const boundMethod = (...params) => method.apply(this, params);

* 简单的、单行的、不会复用的函数，建议采用箭头函数。如果函数体较为复杂，行数较多，还是应该采用传统的函数写法。

所有配置项都应该集中在一个对象，放在最后一个参数，布尔值不可以直接作为参数。

// bad

function divide(a, b, option = false ) {

}

// good

function divide(a, b, { option = false } = {}) {

}

* 不要在函数体内使用 arguments 变量，使用 rest 运算符（...）代替。因为 rest 运算符显式表明你想要获取参数，而且 arguments 是一个类似数组的对象，而 rest 运算符可以提供一个真正的数组。

// bad

function concatenateAll() {

  const args = Array.prototype.slice.call(arguments);

  return args.join('');

}

// good

function concatenateAll(...args) {

  return args.join('');

}

* 使用默认值语法设置函数参数的默认值，而不是修改函数的实参。

// really bad

function handleThings(opts) {

  opts = opts || {};

}

// still bad

function handleThings(opts) {

  if (opts === void 0) {

    opts = {};

  }

}

// good

function handleThings(opts = {}) {

  // ...

}

1. **map + set + weakmap + weakset数据结构**

新加的集合类型，提供了更加方便的获取属性值的方法，可以检查某个属性是属于原型链上还是当前对象的，并用获取对象的set和get方法。但是，推荐使用weakmap和weakset，而不是map和set，除非必须使用。

* 普通集合会阻止垃圾回收器对这些作为属性键存在的对象的回收，有造成内存泄漏的危险。

// not good,Maps

var wm = new Map();

wm.set(key, { extra: 42 });

wm.size === 1

// not good, Sets

var ws = new Set();

ws.add({ data: 42 });

// good,Weak Maps

var wm = new WeakMap();

wm.set(key, { extra: 42 });

wm.size === undefined

// good,Weak Sets

var ws = new WeakSet();

ws.add({ data: 42 });//因为添加到ws的这个临时对象没有其他变量引用它，所以ws不会保存它的值，也就是说这次添加其实没有意思

// not good

let object = {},

object.hasOwnProperty(key)

// good

let object = new WeakSet();

object.has(key) === true;

* 当你的元素或者键值有可能不是字符串时，推荐使用WeakMap和WeakSet。

// bad

var obj = { 3:'value' };

// good

var ws = new WeakSet();

ws.add(3, 'value');

* 有移除操作的需求时，使用WeakMap和WeakSet

// bad

var obj = { 'key':'value' };

delete obj.key;

// good

var ws = new WeakSet();

ws.add('key', 'value');

ws.remove('key');

* 当仅需要一个不可重复的集合时，使用WeakSet优先于普通对象，而不要使用{foo: true}这样的对象。

// bad

var obj = { 'key':'value' };

// good

var ws = new WeakSet();

ws.add('key', 'value');

* 当需要遍历功能时，使用WeakMap和WeakSet，因为其可以简单地使用for..of进行遍历，性能更高

// bad

var obj = { key:'value', key1: 'value1' };

for(var key in obj){

}

// good

var ws = new WeakSet();

ws.add('key', 'value').add('key1', 'value1');

for(var key of ws){

}

注意区分 Object 和 Map，只有模拟现实世界的实体对象时，才使用 Object。如果只是需要key: value的数据结构，使用 Map 结构。因为 Map 有内建的遍历机制。

let map = new Map(arr);

for (let key of map.keys()) {

  console.log(key);

}

for (let value of map.values()) {

  console.log(value);

}

for (let item of map.entries()) {

  console.log(item[0], item[1]);

}

1. **Class**

* 总是用 Class，取代需要 prototype 的操作。因为 Class 的写法更简洁，更易于理解。

// bad

function Queue(contents = []) {

  this.\_queue = [...contents];

}

Queue.prototype.pop = function() {

  const value = this.\_queue[0];

  this.\_queue.splice(0, 1);

  return value;

}

// good

class Queue {

  constructor(contents = []) {

    this.\_queue = [...contents];

  }

  pop() {

    const value = this.\_queue[0];

    this.\_queue.splice(0, 1);

    return value;

  }

}

* 使用extends实现继承，因为这样更简单，不会有破坏instanceof运算的危险。

// bad

const inherits = require('inherits');

function PeekableQueue(contents) {

  Queue.apply(this, contents);

}

inherits(PeekableQueue, Queue);

PeekableQueue.prototype.peek = function() {

  return this.\_queue[0];

}

// good

class PeekableQueue extends Queue {

  peek() {

    return this.\_queue[0];

  }

}

* 如果不是class类，不使用new

// not good

function Foo() {

}

const foo = new Foo();

// good

class Foo {

}

const foo = new Foo();

* 定义类时，方法的顺序如下：

1. constructor
2. public get/set 公用访问器，set只能传一个参数
3. public methods 公用方法，公用相关命名使用小驼峰式写法(lowerCamelCase)
4. private get/set 私有访问器，私有相关命名应加上下划线 \_ 为前缀
5. private methods 私有方法

// good

class SomeClass {

  constructor() {

// constructor

  }

  get aval() {

    // public getter

  }

  set aval(val) {

    // public setter

  }

doSth() {

    // 公用方法

   }

get \_aval() {

    // private getter

  }

set \_aval() {

    // private setter

  }

 \_doSth() {

    // 私有方法

  }

}

1. **模块**

首先，Module 语法是 JavaScript 模块的标准写法，坚持使用这种写法。

* 使用import取代require。

// bad

const moduleA = require('moduleA');

const func1 = moduleA.func1;

const func2 = moduleA.func2;

// good

import { func1, func2 } from 'moduleA';

* 使用export取代module.exports。

// commonJS的写法

var React = require('react');

var Breadcrumbs = React.createClass({

  render() {

    return <nav />;

  }

});

module.exports = Breadcrumbs;

// ES6的写法

import React from 'react';

class Breadcrumbs extends React.Component {

  render() {

    return <nav />;

  }

};

export default Breadcrumbs;

如果模块只有一个输出值，就使用export default，如果模块有多个输出值，就不使用export default，export default与普通的export不要同时使用。

* 不要在模块输入中使用通配符\*。因为这样可以确保你的模块之中，有一个默认输出（export default）。

// bad

import \* as myObject from './importModule';

// good

import myObject from './importModule';

* 如果模块默认输出一个函数，函数名的首字母应该小写。

function makeStyleGuide() {

}

export default makeStyleGuide;

* 如果模块默认输出一个对象，对象名的首字母应该大写。

const StyleGuide = {

  es6: {

  }

};

export default StyleGuide;

* 导出单一一个类时，确保你的文件名就是你的类名

// file contents

class CheckBox {

  // ...

}

module.exports = CheckBox;

// in some other file

// bad

const CheckBox = require('./checkBox');

// bad

const CheckBox = require('./check\_box');

// good

const CheckBox = require('./CheckBox');

* 多变量要导出时应采用对象解构形式

// not good

export const a= 'a';

export const b= 'b';

// good

export const a= 'a';

export const b= 'b';

export default { a, b };

* 不要直接从一个 import 上 export

// bad

// filename es6.js

export default { es6 } from './airbnbStyleGuide';

// good

// filename es6.js

import { es6 } from './AirbnbStyleGuide';

export default es6;

1. **promise、symbols、proxy**

* promise是一种异步处理模式。发promise申明和调用分开，推荐异步方式使用Promise

// not good

(new Promise(resolve, reject){})

  .then(cunction(){},function(){})

  .then();

// good

var promise = new Primise(function(resolve, reject){});

promise

  .then(function(){},function(){]})

  .then();

* symbol用于对象的键和私有属性，使用过于复杂，没有使用必要，容易扰乱外层作用域。总之不要使用

// good

function MyClass(privateData) {

  let key = privateData;

}

// not good

const key = Symbol('key');

function MyClass(privateData) {

  this[key] = privateData;

}

const object = new MyClass("hello")

object['key'] === undefined //无法访问该属性，因为是私有的

* Proxy对象拦截器，访问对象时候的代理。任何访问该对象的操作之前都会通过这道“拦截”，即执行Proxy里面定义的方法。其中，ES6原生规定的Proxy的基本用法为：

let pro = new Proxy(target, handler);

## **TypeScript指南**

### **我们为什么需要TypeScript?**

我们知道，JavaScript 是一门非常灵活的编程语言：

* 它没有类型约束，一个变量可能初始化时是字符串，过一会儿又被赋值为数字。
* 由于隐式类型转换的存在，有的变量的类型很难在运行前就确定。
* 基于原型的面向对象编程，使得原型上的属性或方法可以在运行时被修改。
* 函数是 JavaScript 中的一等公民，可以赋值给变量，也可以当作参数或返回值。

这种灵活性就像一把双刃剑，一方面使得 JavaScript 蓬勃发展，无所不能，从 2013 年开始就一直蝉联最普遍使用的编程语言排行榜冠军；另一方面也使得它的代码质量参差不齐，维护成本高，运行时错误多。

而 TypeScript 的类型系统，在很大程度上弥补了 JavaScript 的缺点。因为TypeScript是静态类型语言。

类型系统按照「类型检查的时机」来分类，可以分为动态类型和静态类型。

**动态类型**是指在运行时才会进行类型检查，这种语言的类型错误往往会导致运行时错误。JavaScript 是一门解释型语言，没有编译阶段，所以它是动态类型，以下这段代码在运行时才会报错。

**静态类型**是指编译阶段就能确定每个变量的类型，这种语言的类型错误往往会导致语法错误。TypeScript 在运行前需要先编译为 JavaScript，而在编译阶段就会进行类型检查，所以 TypeScript 是静态类型，这段 TypeScript 代码在编译阶段就会报错了。

下面TypeScript快速开始：

npm install typescript -g // 全局安装ts

安装完以后可以直接编译.ts文件 ：tsc demo.ts 可在当前路径看到编译后的 demo.js文件

如果想要查看ts文件的运行结果：可安装ts-node工具直接在node环境中查看ts文件的运行结果

npm install ts-node -g； // ts-node demo.ts 可查看运行结果

### **3.1 基本数据类型**

let name: string = 'smart';

let age: number = 10;

let married: boolean = true;

let hobbies: string[] = ['1', '2', '3'];

let interests: Array<string> = ['4', '5', '6'];

// null 空 undefined 未定义

//它们都其它类型的子类型 你可以把它们赋给其它类型的变量

let myname1: string | null = null;

let myname2: string | undefined = undefined;

// void类型 空的 没有

function greeting(name: string): void {

console.log(`hello ${name}`);

}

greeting('smart');

### **3.2 复杂数据类型**

#### **3.2.1 元组**

类似一个数组 它是一个长度和类型都固定的数组

1 长度固定 2 类型可以不一样

let point: [number, number] = [100, 100];

point[0], point[1];

let person: [string, number] = ['smart', 10];

#### **3.2.2 枚举**

// ts 写法 -使用功能枚举类型

enum Status {

OFFLINE,

ONLINE,

DELETED,

}

console.log(Status.OFFLINE) // 0

console.log(Status.ONLINE) // 1

console.log(Status.DELETED) // 2

### **3.3 函数**

// 返回值为number 类型

function add(first: number, second: number): number {

return first + second

}

const total = add(1, 2)

// 无返回值

function sayHello(): void {

console.log('hello')

// return '' 报错

}

// 解构的类型指定

function adds({ first, second }: { first: number; second: number }): number {

return first + second

}

const to = adds({

first: 1,

second: 2,

})

### **3.4 类的定义与继承**

/\*\*

\* 子类继承父类后子类的实例上就拥有了父类中的属性和方法

\* 访问修饰符 public protected private

\* public 自己 自己的子类 和其它类都可能访问

\* protected 受保护的 自己和自己的子类能访问 ，其它 类不能访问

\* private 私有的 只能自己访问，自己的子类不能访问，其它更不行了

\*/

class Person {

public name: string;

protected age: number;

private amount: number;

constructor(name: string, age: number) {

this.name = name;

this.age = age;

this.amount = 100;

}

getName() {

return this.name;

}

setName(newName: string) {

this.name = newName;

}

}

class Student extends Person {

static type = 'Student'

stuNo: number;

constructor(name: string, age: number, stuNo: number) {

super(name, age);

this.stuNo = stuNo;

}

static getType() {

return Student.type;

}

getStuNo() {

return this.name + this.age + this.amount + this.stuNo;

}

setStuNo(newStuNo: number) {

this.stuNo = newStuNo;

}

}

### **3.5 类的装饰器**

interface Person {

xx: string;

yy: string

}

function enhancer(target: any) {

target.prototype.xx = 'xx';

target.prototype.yy = 'yy';

}

@enhancer

class Person {

constructor() { }

}

let p = new Person();

console.log(p.xx);

console.log(p.yy);

### **3.6 抽象类和重写重载继承多态**

abstract class Animal {

name: string;

abstract getName(): string

}

class Cat extends Animal {

getName(): string {

return this.name;

}

}

let cat = new Cat();

cat.name = '猫';

console.log(cat.getName());

### **3.7 接口**

接口可以在面向对象中表示 行为的抽象， 另外可以用来描述 对象的形状

接口就是把一些类中共有属性和方法抽象出来，可以用来约束实现此接口的类

#### **3.7.1对象的形状**

// 接口可以用来描述“对象的形状”，少属性或者多属性都会报错

Interface Speakable{

speak():void;

name?:string;//?表示可选属性

}

let speakman: Speakable{

speak(){}, // 少属性会报错

name,

age// 多属性也会报错

}

#### **3.7.2行为的抽象**

Interface Speakable{

speak():void;

}

Interface Eatable{

eat():void;

}

class Person implements Speakable, Eatable {

name:stringspeak() {

}

eat() {

}

}

### **3.8 基本泛型**

就是泛指的类型，也就是说，定义的时候不确定是什么类型，等到调用的时候采取指定类型：

实现一个函数createArray, 它可以创建一个指定长度的数组，同时将每一项都填充一个默认值

function createArray(length: number, value:any):Array<any> {

let result: any = [];

for(let i =0; i< length; i++){

result[i] = value

}

return result

}

let result = createArray(3, ‘x’)

使用泛型

function createArray<T>(length: number, value:T):Array<T> {

let result: any = [];

for(let i =0; i< length; i++){

result[i] = value

}

return result

}

let result = createArray<string>(3, ‘x’)

泛型的约束

在函数中使用泛型的时候，由于预先并不知道具体的类型，所以不能访问相应类型的方法

interface LengthWise {

length: number

}

function logger<T extends LengthWise>(val: T) {

console.log('aaaa', val.length);

}

logger('smart'); // > 5

### **3.9 compose泛型**

相信用过redux都知道compose函数[footnoteRef:1]

[1] 参考<https://github.com/reduxjs/redux/blob/d794c56f78eccb56ba3c67971c26df8ee34dacc1/src/compose.ts>

在github上把源码copy过来看，会看到多个函数重载 分别分析：不传参数，传入一个参数，传入两个参数（与多个参数情况一致）

type Func<T extends any[], R> = (...a: T) => R

// 传入零个方法

export default function compose(): <R>(a: R) => R

console.log(compose()(‘smart’))

// 传入一个方法 传入一个函数F 返回函数F

export default function compose<F extends Function>(f: F): F

function add1(str){return str +’1’}

console.log(compose(add1)(‘smart’))

// 传入二个方法 重点理解 按照下面例子分析：注意参数 A T R分别代表什么

function add1(str){return str +’1’}

function add2(str){return str +’2’}

console.log(compose(add1, add2)(‘smart’))

// 先看函数返回 Func<T, R> 在最开始定义了翻回一个返回值是R的函数

// 所以 add2(‘smart’) => T= “smart” A= ‘smart2’

// add1(add2(‘smart’)) => A= ‘smart2’ R= ‘smart21’

export default function compose<A, T extends any[], R>(

f1: (a: A) => R, A= ‘smart2’ R= ‘smart21’

f2: Func<T, A> T= “smart” A= ‘smart2’

): Func<T, R>

// 传入三个方法

export default function compose<A, B, T extends any[], R>(

f1: (b: B) => R,

f2: (a: A) => B,

f3: Func<T, A>

): Func<T, R>

/\* four functions \*/

export default function compose<A, B, C, T extends any[], R>(

f1: (c: C) => R,

f2: (b: B) => C,

f3: (a: A) => B,

f4: Func<T, A>

): Func<T, R>

/\* rest \*/

export default function compose<R>(

f1: (a: any) => R,

...funcs: Function[]

): (...args: any[]) => R

export default function compose<R>(...funcs: Function[]): (...args: any[]) => R

// 函数实现

export default function compose(...funcs: Function[]) {

if (funcs.length === 0) {

// infer the argument type so it is usable in inference down the line

return <T>(arg: T) => arg

}

if (funcs.length === 1) {

return funcs[0]

}

return funcs.reduce((a, b) => (...args: any) => a(b(...args)))

}

### **3.10类型保护**

TypeScript 能够在特定的区块中保护变量属于某种确定的类型，可以在此区块中放心的引用此类型的属性，或者调用此类型的方法。这就是TS的类型保护。日常编码过程中，我们通常使用类型断言、in语法、instanceof语法、typeof语法等方法指定某个区块中的变量类型。

class Bird {

// 独有方法

fly() {};

// 共有方法

layEggs() {};

}

class Fish {

// 独有方法

swim() {};

// 共有方法

layEggs() {};

}

function getSmallPet(): Fish | Bird {

// ...

}

let pet = getSmallPet();

// 使用类型断言

if ((pet as Bird).fly) {

(pet as Bird).fly()

} else {

(pet as Fish).swim()

}

// 使用 in 语法进行

if ('fly' in pet) {

pet.fly()

} else {

pet.swim()

}

//使用instanceof语法

if (pet instanceof Bird) {

pet.fly()

} else {

pet.swim()

}

// 使用typeof语法（只能识别"number"， "string"， "boolean"或 "symbol"）

if (typeof pet === 'number') {

pet++

} else {

pet = Number(pet) + 1

}

### **3.11类型推断**

在日常代码中，如果我们没有显示声明变量的类型，TS会使用类型推断来猜测类型，从而限制一些方法的使用，弥补了JS作为弱类型语言的弱点。这就是TS的类型推断。如：

var x = 3；// number

var arr = [3,’arr’]；//Array<number | string>

window.onmousedown = function(mouseEvent) {

console.log(mouseEvent.button); // error:mouseEvent 上不存在属性 button

};

实际编码过程中，经常出现TS推断的类型不是我们期望的类型的情况，这时候就需要明确指定类型。

### **3.12类型变换（类型断言）**

TS是类型声明式语言，如果我们需要在编码过程中，动态转换变量类型，TS将会提示错误。这时候就需要使用类型断言告知TS类型发生变化。如:

var a = ‘123’;

a as any = 123;

注意转换规则要遵循兼容性原则，从x转向y，则y必须具备x的所有属性。

### **3.13条件类型**

当初始状态下不能直接确定具体类型时，就可以使用条件类型声明。TS提供了预定义的条件类型关键字：

Exclude<T, U> -- 从T中剔除可以赋值给U的类型。

Extract<T, U> -- 提取T中可以赋值给U的类型。

NonNullable<T> -- 从T中剔除null和undefined。

ReturnType<T> -- 获取函数返回值类型。

InstanceType<T> -- 获取构造函数类型的实例类型。

示例

type T00 = Exclude<"a" | "b" | "c" | "d", "a" | "c" | "f">; // "b" | "d"

type T10 = ReturnType<() => string>; // string

type T20 = InstanceType<typeof C>; // C

还可以通过类型运算得到最终的变量类型：

//示例一

declare function f<T extends boolean>(x: T): T extends true ? string : number

const x = f(Math.random() < 0.5) // const x: string | number

const y = f(true) // const y: string

const z = f(false) // const z: number

//示例二

type TypeName<T> =

T extends string ? "string" :

T extends number ? "number" :

T extends boolean ? "boolean" :

T extends undefined ? "undefined" :

T extends Function ? "function" : "object";

type T0 = TypeName<string>; // "string"

type T1 = TypeName<"a">; // "string"

type T2 = TypeName<true>; // "boolean"

type T3 = TypeName<() => void>; // "function"

type T4 = TypeName<string[]>; // "object"

### **3.14内置工具类型**

TS内置了一些可以对已有类型进行处理，然后输出新的类型的工具类型。

Partical--为定义的属性都添加上?，表示允许属性为undefined

interface Person {

name: string;

age: number;

}

type NewPerson = Partial<Person>;

const axes: NewPerson = {

name: 'axes'

}

Require--为定义的属性都添加上-?，表示属性都是必须声明的

Pick<T, K extends keyof T>--从T中找出属性K

type NewPerson = Pick<Person, 'name'>; // { name: string; }

Record<K,U>--把K中定义的值的类型都设置为类型U

type T11 = Record<'a' | 'b' | 'c', Person>; // { a: Person; b: Person; c: Person; }

ThisType--指定上下文类型

const obj: ThisType<Person> = {

dosth() {

this.name // string

}

}

### **3.15模块和命名空间**

模块是一种创建一组相关变量、函数、类和接口等的方法。只能在其自身的作用域里执行，而不是在全局作用域里。TypeScript 与 ECMAScript 2015 一样，任何包含 import/export 的文件都被当成一个模块。

命名空间的作用是明确构造逻辑分组并防止命名冲突。可以把一个命名空间分割到多个文件进行实现。语法格式如下：

// Validation.ts

namespace Validation {

export interface StringValidator {

isAcceptable(s: string): boolean;

}

}

// LettersOnlyValidator.ts

/// <reference path="Validation.ts" />

namespace Validation {

const lettersRegexp = /^[A-Za-z]+$/;

export class LettersOnlyValidator implements StringValidator {

isAcceptable(s: string) {

return lettersRegexp.test(s);

}

}

}

调用命名空间：

/// <reference path=" Validation.d.ts" />

/// <reference path=" LettersOnlyValidator.d.ts" />

let validators: { [s: string]: Validation.StringValidator; } = {};

区别：命名空间可以跨多个文件使用，这既带来了便利，又带来了难以识别组件依赖性的缺点，实际使用过程中要注意相关注释。需要注意的是TS推荐使用namespace代替module，以区分JS概念中的module。

### **3.16类型声明**

基本类型声明：

// 变量

const num: number = 1;

const str: string = 'str';

const bool: boolean = true;

const nulls: null = null;

const undefine: undefined = undefined;

const symbols: symbol = Symbol('symbal');

const any: any = 'any types'; // typescript的any类型，相当于什么类型约束都没有

const nums: number[] = [1, 2, 3, 4];

接口声明（声明对象类型）：

interface Name {

 first: string;

 second: string;

}

let username: Name = {

 first: 'John',

 second: 'joe'

};

//接口合并

interface User {

 name: string

 age: number

}

interface User {

 sex: string

}

/\*

User 接口为 {

 name: string

 age: number

 sex: string

}

\*/

类型别名（可以声明组合类型、交叉类型、元组类型等）：

type User = {

 name: string

 age: number

};

// 组合类型

type Name = {

 name: string;

}

type User = Name & { age: number };

//交叉类型

interface Dog {

 wong();

}

interface Cat {

 miao();

}

type Pet = Dog | Cat

// 获取变量的类型

let div = document.createElement('div');

type B = typeof div;

### **3.17类型扩展**

在已声明好的类型上进行扩展，可以通过接口继承、声明交叉类型等方式

// 接口继承

interface Name {

  first: string;

  second: string;

}

Interface person extends Name{

age?:number;//可选类型

say():void;

}

// 声明交叉类型

type Fruit = 'Apple' | 'Banana';

const human: {

name: string,

age: number,

needFruit: {

[key in Fruit | 'Orange']?: number

}

} = {

name: '张三',

age: 12,

needFruit: {

Apple: 50,

Orange: 100,

}

}

### **3.18声明文件**

在使用第三方JS库时，我们可以安装第三方库对应的声明文件使用TS的类型检查等特性。声明文件以.d.ts为后缀。一般情况下，我们会通过@types/第三方库名去寻找第三方库对应的声明文件。如果是旧的JS库，往往没有对应的TS库。这时候就需要自己补全声明文件。

下面简要说明一下结合tsconfig.json的使用：

1. 第三方库

按照声明库：npm install @types/three

在tsconfig.js中引入

{

  "compilerOptions":{

  "types": [

+  "three",

]

}

1. 自建声明文件

以vue-cli创建的工程为例，在src文件夹内创建模块名.d.ts文件。

全局变量的声明文件主要有以下几种语法：

declare var 声明全局变量

declare function 声明全局方法

declare class 声明全局类

declare enum 声明全局枚举类型

declare namespace 声明命名空间（含有子属性）

declare module 声明模块

interface 和 type 声明全局类型

### **3.19 tsconfig.json配置**

tsconfig.json指定了用来编译这个项目的根文件和编译选项。

基础配置项：

{

"[extends](https://www.typescriptlang.org/tsconfig)": "@tsconfig/recommended/tsconfig.json",//扩展配置项

"[compilerOptions](https://www.typescriptlang.org/tsconfig)": {//编译配置项

"[preserveConstEnums](https://www.typescriptlang.org/tsconfig)": true

},

"[include](https://www.typescriptlang.org/tsconfig)": ["src/\*\*/\*"],//ts检查项

"[exclude](https://www.typescriptlang.org/tsconfig)": ["node\_modules", "\*\*/\*.spec.ts"] //忽略ts检查的文件

}

主要的配置项在compilerOptions内：

  "compilerOptions": {

    "target": "esnext",//指定编译后的版本目标

    "module": "esnext",//指定模块标准

    "strict": true,//开启严格模式

    "jsx": "preserve",// 指定jsx代码的编译方式。preserve指保留jsx，不做编译

    "importHelpers": true,// 是否引入tslib里的工具函数

    "moduleResolution": "node",//模块解析策略

    "experimentalDecorators": true,// 是否启用实验性的装饰器特性

    "skipLibCheck": true, //是否忽略声明文件的类型检查

    "esModuleInterop": true,//是否兼容模块导入，允许从无default导出的模块进行默认导入

    "allowSyntheticDefaultImports": true,//开启了esModuleInterop后默认开启

    "sourceMap": true,// 是否将map文件内容和js文件编译在同一个js文件中

    "baseUrl": ".",//设置解析非相对模块名称的基本目录

    "types": [//引入第三方库的types文件

      "webpack-env",

      "three",

    ],

    "paths": {//设置访问路径别名

      "@/\*": [

        "src/\*"

      ]

    },

    "lib": [//指定要包含在编译中的库文件

      "esnext",

      "dom",

      "dom.iterable",

      "scripthost"

    ]

  },

## **Vue指南**

### **4.1 vue源码核心原理分析**

#### **4.1.1 vue响应式原理**

**初始化数据**：首先会把我们在options api 中传入的data参数初始化，调用了observe方法

import { observe } from './observer/index.js'

function initData (vm) {

let data = vm.$options.data;

data = vm.\_data = typeof data === 'function' ? data.call(vm) : data;

observe(data);

}

**递归属性劫持**

class Observer { // 观测值

constructor (value) {

this.walk(value);

}

walk (data) { // 让对象上的所有属性依次进行观测

let keys = Object.keys(data);

for( let i = 0; i < keys.length; i++ ){

let key = keys[i];

let value = data[key];

defineReactive(data,key,value);

}

}

}

function defineReactive (data,key,value) {

observe(value); // 如果是对象中又包含了对象，需要递归处理

Object.defineProperty(data,key,{

get(){

return value

},

set(newValue){

if(newValue == value) return;

observe(newValue);

value = newValue

}

})

}

export function observe(data) {

if(typeof data !== 'object' && data != null){

return;

}

return new Observer(data);

}

**数组方法劫持**：如果把数组当作对象来劫持会把数组的索引也一并劫持，vue为了避免这种情况对数组做了特殊处理

import {arrayMethods} from './array';

class Observer {

constructor(value){

if(Array.isArray(value)){ // 判断如果是数组，对数组做了特殊处理 value.\_\_proto\_\_ = arrayMethods; // 重写数组原型方法

this.observeArray(value);

} else {

this.walk(value);

}

}

observeArray(value){

for(let i = 0 ; i < value.length ;i ++){

observe(value[i]);

}

}

}

**重写数组的原型方法:** 需要重写数组的方法，给新插入的数据加入数据劫持

// value.\_\_proto\_\_ = arrayMethods 原型链查找的问题， 会向上查找，先查找重写的，找不到就会继续向上查找

// arrayMethods.\_\_proto\_\_ = oldArrayMethods

let oldArrayProtoMethods = Array.prototype;

export let arrayMethods = Object.create(oldArrayProtoMethods);

let methods = [ 'push', 'pop', 'shift', 'unshift', 'reverse', 'sort', 'splice'];

methods.forEach(method => {

arrayMethods[method] = function (...args) {

const result = oldArrayProtoMethods[method].apply(this, args);

const ob = this.\_\_ob\_\_;

// push unshift 添加的元素可能还是一个对象

let inserted;

switch (method) {

case 'push':

case 'unshift':

inserted = args;

break;

case 'splice':

// 3个参数 新增的属性 splice 有删除 新增的的功能 arr.splice(0,1,{name:1})

inserted = args.slice(2)

default:

break;

}

if (inserted) ob.observeArray(inserted); // 对新增的每一项进行观测

return result

}

})

**增加\_\_ob\_\_属性**：给所有响应式数据增加标识，并且可以在响应式上获取Observer实例上的方法

class Observer {

constructor(value){

// 给每一个监控过的对象都增加一个\_\_ob\_\_属性

Object.defineProperty(value,'\_\_ob\_\_',{

enumerable:false,

configurable:false,

value: this

});

// ...

}

}

**数据代理**：为了让用户更好的使用 可以直接vm.xxx 对数据进行代理

function proxy(vm,source,key){

Object.defineProperty(vm, key, {

get() {

return vm[source][key];

},

set (newValue) {

vm[source][key] = newValue;

}

});

}

function initData(vm){

let data = vm.$options.data;

data = vm.\_data = typeof data === 'function' ? data.call(vm) : data;

for(let key in data){ // 将\_data上的属性全部代理给vm实例

proxy(vm, '\_data', key)

}

observe(data);

}

#### **4.1.2.模板编译**

核心思路：

1、将模板用正则匹配一段一段截取成ast 语法树

2、再将ast树 再次转化成js的语法。

3、然后通过模板引擎解析成虚拟dom,也就是Vnode。

4、最后把Vnode替换成dom节点。

**目标**：将template编译成render函数

Vue.prototype.\_init = function (options) {

const vm = this;

vm.$options = options;

// 初始化状态

initState(vm);

// 页面挂载 如果用户传入了el 就要实现挂载流程

if (vm.$options.el) {

vm.$mount(vm.$options.el);

}

}

Vue.prototype.$mount = function (el) {

const vm = this;

const options = vm.$options;

el = document.querySelector(el);

// 默认先会查找有没有render方法，没有render会采用template template也没有就用el中的内容

if (!options.render) {

let template = options.template;

// 如果没有模板但是有el

if (!template && el) {

template = el.outerHTML;

}

const render= compileToFunctions(template);

options.render = render;

}

}

1. 解析标签和内容

<div id=’app’>

<p>{{name}}</p>

</div>

用正则表达式把上面的template一段一段匹配截取，解析成ast语法树

{

tag: 'div',

attrs: [{name:'id',value:'app'}],

parent: null,

type: 1,

children: [ {

tag: 'p',

attrs: [],

parent: root,

type: 1,

children: [

{

text : '{{name}}',

type: 3

}

]

}]

}

// 用于匹配html的正则表达式

const ncname = `[a-zA-Z\_][\\-\\.0-9\_a-zA-Z]\*`;

const qnameCapture = `((?:${ncname}\\:)?${ncname})`;

const startTagOpen = new RegExp(`^<${qnameCapture}`); // 标签开头的正则 捕获的内容是标签名

const endTag = new RegExp(`^<\\/${qnameCapture}[^>]\*>`); // 匹配标签结尾的 </div>

const attribute = /^\s\*([^\s"'<>\/=]+)(?:\s\*(=)\s\*(?:"([^"]\*)"+|'([^']\*)'+|([^\s"'=<>`]+)))?/; // 匹配属性的

const startTagClose = /^\s\*(\/?)>/; // 匹配标签结束的 >

const defaultTagRE = /\{\{((?:.|\r?\n)+?)\}\}/g

function start(tagName,attrs){

console.log(tagName,attrs)

}

function end(tagName){

console.log(tagName)

}

function chars(text){

console.log(text);

}

function parseHTML(html){

// 循环解析html字符串

while(html){

let textEnd = html.indexOf('<');

if(textEnd == 0){

// 通过这个方法获取到匹配的结果 tagName,attrs

const startTagMatch = parseStartTag();

if(startTagMatch){

start(startTagMatch.tagName,startTagMatch.attrs);

continue;

}

const endTagMatch = html.match(endTag);

if(endTagMatch){

advance(endTagMatch[0].length);

end(endTagMatch[1]);

continue;

}

}

let text;

if(textEnd >= 0){

text = html.substring(0,textEnd);

}

if(text){

advance(text.length);

chars(text); // 3解析文本

}

}

function advance(n){

html = html.substring(n);

}

function parseStartTag(){

const start = html.match(startTagOpen);

if(start){

const match = {

tagName:start[1],

attrs:[]

}

advance(start[0].length);

let attr,end;

while(!(end = html.match(startTagClose)) && (attr = html.match(attribute))){

advance(attr[0].length);

match.attrs.push({name:attr[1],value:attr[3]});

}

if(end){

advance(end[0].length);

return match

}

}

}

}

export function compileToFunctions(template){

parseHTML(template);

return function(){}

}

1. 生成ast语法树

let root;

let currentParent;

let stack = [];

const ELEMENT\_TYPE = 1;

const TEXT\_TYPE = 3;

function createASTElement(tagName,attrs){

return {

tag:tagName,

type:ELEMENT\_TYPE,

children:[],

attrs,

parent:null

}

}

function start(tagName, attrs) {

// 遇到开始标签 就创建一个ast元素并推入栈中

let element = createASTElement(tagName,attrs);

if(!root){

root = element;

}

currentParent = element;

stack.push(element);

}

function end(tagName) {

let element = stack.pop();

currentParent = stack[stack.length-1];

if(currentParent){

element.parent = currentParent;

currentParent.children.push(element); // 实现了一个树的父子关系

}

}

// 文本节点

function chars(text) {

text = text.replace(/\s/g,'');

if(text){

currentParent.children.push({

type:TEXT\_TYPE,

text

})

}

}

1. 生成代码

将ast语法树 转化成render函数的结果

<div style="color:red">hello {{name}} <span></span></div>

render(){

return \_c('div',{style:{color:'red'}},\_v('hello'+\_s(name)),\_c('span',undefined,''))

}

实现代码生成

function gen(node) {

if (node.type == 1) {

// 元素标签

return generate(node);

} else {

let text = node.text

if(!defaultTagRE.test(text)){

return `\_v(${JSON.stringify(text)})`

}

let lastIndex = defaultTagRE.lastIndex = 0

let tokens = [];

let match,index;

while (match = defaultTagRE.exec(text)) {

index = match.index;

if(index > lastIndex){

tokens.push(JSON.stringify(text.slice(lastIndex,index)));

}

tokens.push(`\_s(${match[1].trim()})`)

lastIndex = index + match[0].length;

}

if(lastIndex < text.length){

tokens.push(JSON.stringify(text.slice(lastIndex)))

}

return `\_v(${tokens.join('+')})`;

}

}

function getChildren(el) { // 生成儿子节点

const children = el.children;

if (children) {

return `${children.map(c=>gen(c)).join(',')}`

} else {

return false;

}

}

function genProps(attrs){ // 生成属性

let str = '';

for(let i = 0; i<attrs.length; i++){

let attr = attrs[i];

if(attr.name === 'style'){

let obj = {}

attr.value.split(';').forEach(item=>{

let [key,value] = item.split(':');

obj[key] = value;

})

attr.value = obj;

}

str += `${attr.name}:${JSON.stringify(attr.value)},`;

}

return `{${str.slice(0,-1)}}`;

}

function generate(el) {

let children = getChildren(el);

let code = `\_c('${el.tag}',${

el.attrs.length?`${genProps(el.attrs)}`:'undefined'

}${

children? `,${children}`:''

})`;

return code;

}

let code = generate(root);

1. 生成render函数

export function compileToFunctions(template) {

// 解析html字符串 将html字符串 => ast语法树

// 需要将ast语法树生成最终的render函数 就是字符串拼接 供模板引擎解析

parseHTML(template);

// 将ast树 再次转化成js的语法

// \_c("div",{id:app},\_c("p",undefined,\_v('hello' + \_s(name) )),\_v('hello'))

let code = generate(root);

let render = `with(this){return ${code}}`;

let renderFn = new Function(render);

return renderFn

}

#### **4.1.3.渲染watcher**

初始化渲染Watcher

import {mountComponent} from './lifecycle'

Vue.prototype.$mount = function (el) {

const vm = this;

const options = vm.$options;

el = document.querySelector(el);

// 如果没有render方法

if (!options.render) {

let template = options.template;

// 如果没有模板但是有el

if (!template && el) {

template = el.outerHTML;

}

const render= compileToFunctions(template);

options.render = render;

}

mountComponent(vm,el);

}

lifecycle.js

先调用\_render方法生成虚拟dom,通过\_update方法将虚拟dom创建成真实的dom

export function lifecycleMixin() {

Vue.prototype.\_update = function (vnode) {}

}

export function mountComponent(vm, el) {

vm.$el = el;

let updateComponent = () => {

// 先调用\_render方法生成虚拟dom,通过\_update方法将虚拟dom创建成真实的dom

// 将虚拟节点 渲染到页面上

vm.\_update(vm.\_render());

}

new Watcher(vm, updateComponent, () => {}, true);

}

render.js

export function renderMixin(Vue){

Vue.prototype.\_render = function () {}

}

watcher.js

let id = 0;

class Watcher {

constructor(vm, exprOrFn, cb, options) {

this.vm = vm;

this.exprOrFn = exprOrFn;

if (typeof exprOrFn == 'function') {

this.getter = exprOrFn;

}

this.cb = cb;

this.options = options;

this.id = id++;

this.get();

}

get() {

this.getter();

}

}

export default Watcher;

生成虚拟DOM

import {createTextNode,createElement} from './vdom/create-element'

export function renderMixin(Vue){

Vue.prototype.\_v = function (text) { // 创建文本

return createTextNode(text);

}

Vue.prototype.\_c = function () { // 创建元素

return createElement(...arguments);

}

Vue.prototype.\_s = function (val) {

return val == null? '' : (typeof val === 'object'?JSON.stringify(val):val);

}

Vue.prototype.\_render = function () {

const vm = this;

const {render} = vm.$options;

let vnode = render.call(vm);

return vnode;

}

}

创建虚拟节点

export function createTextNode(text) {

return vnode(undefined,undefined,undefined,undefined,text)

}

export function createElement(tag,data={},...children){

let key = data.key;

if(key){

delete data.key;

}

return vnode(tag,data,key,children);

}

function vnode(tag,data,key,children,text){

return {

tag,

data,

key,

children,

text

}

}

生成真实的DOM元素

将虚拟节点渲染成真实节点

import {patch} './observer/patch'

export function lifecycleMixin(Vue){

Vue.prototype.\_update = function (vnode) {

const vm = this;

vm.$el = patch(vm.$el,vnode);

}

}

export function patch(oldVnode,vnode){

const isRealElement = oldVnode.nodeType;

if(isRealElement){

const oldElm = oldVnode;

const parentElm = oldElm.parentNode;

let el = createElm(vnode);

parentElm.insertBefore(el,oldElm.nextSibling);

parentElm.removeChild(oldVnode)

return el;

}

}

function createElm(vnode){

let {tag,children,key,data,text} = vnode;

if(typeof tag === 'string'){

vnode.el = document.createElement(tag);

updateProperties(vnode);

children.forEach(child => {

return vnode.el.appendChild(createElm(child));

});

}else{

vnode.el = document.createTextNode(text);

}

return vnode.el

}

function updateProperties(vnode){

let newProps = vnode.data || {}; // 获取当前老节点中的属性

let el = vnode.el; // 当前的真实节点

for(let key in newProps){

if(key === 'style'){

for(let styleName in newProps.style){

el.style[styleName] = newProps.style[styleName]

}

}else if(key === 'class'){

el.className= newProps.class

}else{ // 给这个元素添加属性 值就是对应的值

el.setAttribute(key,newProps[key]);

}

}

}

#### **4.1.4.生命周期合并**

Mixin原理

import {mergeOptions} from '../util/index.js'

export function initGlobalAPI(Vue){

Vue.options = {};

Vue.mixin = function (mixin) {

// 将属性合并到Vue.options上

this.options = mergeOptions(this.options,mixin);

return this;

}

}

合并生命周期

对生命周期的勾子进行特殊合并处理：如果合并成数组：[lifecyclefn, lifecyclefn, lifecyclefn]

例如：父组件中有 created，子组件也含有created 则合并成[created, created] 依次调用

export const LIFECYCLE\_HOOKS = [

'beforeCreate',

'created',

'beforeMount',

'mounted',

'beforeUpdate',

'updated',

'beforeDestroy',

'destroyed',

]

const strats = {};

function mergeHook(parentVal, childValue) {

if (childValue) {

if (parentVal) {

return parentVal.concat(childValue); // 合并生命周期逻辑

} else {

return [childValue]

}

} else {

return parentVal;

}

}

LIFECYCLE\_HOOKS.forEach(hook => {

strats[hook] = mergeHook

})

export function mergeOptions(parent, child) {

const options = {}

for (let key in parent) {

mergeField(key)

}

for (let key in child) {

if (!parent.hasOwnProperty(key)) {

mergeField(key);

}

}

function mergeField(key) {

if (strats[key]) {

options[key] = strats[key](parent[key], child[key]);

} else {

if (typeof parent[key] == 'object' && typeof child[key] == 'object') {

options[key] = {

...parent[key],

...child[key]

}

}else{

options[key] = child[key];

}

}

}

return options

}

调用生命周期

export function callHook(vm, hook) {

const handlers = vm.$options[hook];

if (handlers) {

for (let i = 0; i < handlers.length; i++) {

handlers[i].call(vm);

}

}

}

初始化流程中调用生命周期

Vue.prototype.\_init = function (options) {

const vm = this;

vm.$options = mergeOptions(vm.constructor.options,options);

// 初始化状态

callHook(vm,'beforeCreate');

initState(vm);

callHook(vm,'created');

if (vm.$options.el) {

vm.$mount(vm.$options.el);

}

}

#### **4.1.5.依赖收集**

每个属性都要有一个dep,每个dep中存放着watcher,同一个watcher会被多个dep所记录。

在渲染时存储watcher

class Watcher{

// ...

get(){

pushTarget(this);

this.getter();

popTarget();

}

}

定义Dep

let id = 0;

class Dep{

constructor(){

this.id = id++;

}

}

let stack = [];

export function pushTarget(watcher){

Dep.target = watcher;

stack.push(watcher);

}

export function popTarget(){

stack.pop();

Dep.target = stack[stack.length-1];

}

export default Dep;

**对象依赖收集**

let dep = new Dep();

Object.defineProperty(data, key, {

get() {

if(Dep.target){ // 如果取值时有watcher

dep.depend(); // 让watcher保存dep，并且让dep 保存watcher

}

return value

},

set(newValue) {

if (newValue == value) return;

observe(newValue);

value = newValue;

dep.notify(); // 通知渲染watcher去更新

}

});

Dep实现

class Dep{

constructor(){

this.id = id++;

this.subs = [];

}

depend(){

if(Dep.target){

Dep.target.addDep(this);// 让watcher,去存放dep

}

}

notify(){

this.subs.forEach(watcher=>watcher.update());

}

addSub(watcher){

this.subs.push(watcher);

}

}

Watcher

constructor(){

this.deps = [];

this.depsId = new Set();

}

addDep(dep){

let id = dep.id;

if(!this.depsId.has(id)){

this.depsId.add(id);

this.deps.push(dep);

dep.addSub(this);

}

}

update(){

this.get();

}

**数组的依赖收集**

this.dep = new Dep(); // 专门为数组设计的

if (Array.isArray(value)) {

value.\_\_proto\_\_ = arrayMethods;

this.observeArray(value);

} else {

this.walk(value);

}

function defineReactive(data, key, value) {

let childOb = observe(value);

let dep = new Dep();

Object.defineProperty(data, key, {

get() {

if(Dep.target){

dep.depend();

if(childOb){

childOb.dep.depend(); // 收集数组依赖

}

}

return value

},

set(newValue) {

if (newValue == value) return;

observe(newValue);

value = newValue;

dep.notify();

}

})

}

arrayMethods[method] = function (...args) {

// ...

ob.dep.notify()

return result;

}

递归收集数组依赖

if(Dep.target){

dep.depend();

if(childOb){

childOb.dep.depend(); // 收集数组依赖

if(Array.isArray(value)){ // 如果内部还是数组

dependArray(value);// 不停的进行依赖收集

}

}

}

function dependArray(value) {

for (let i = 0; i < value.length; i++) {

let current = value[i];

current.\_\_ob\_\_ && current.\_\_ob\_\_.dep.depend();

if (Array.isArray(current)) {

dependArray(current)

}

}

}

#### **4.1.6.Vue异步更新nextTick实现**

实现队列机制

update(){

queueWatcher(this);

}

Scheduler

import {

nextTick

} from '../util/next-tick'

let has = {};

let queue = [];

function flushSchedulerQueue() {

for (let i = 0; i < queue.length; i++) {

let watcher = queue[i];

watcher.run()

}

queue = [];

has = {}

}

let pending = false

export function queueWatcher(watcher) {

const id = watcher.id;

if (has[id] == null) {

has[id] = true;

queue.push(watcher);

if(!pending){

nextTick(flushSchedulerQueue)

pending = true;

}

}

}

#### **4.1.7.dom-diff算法[footnoteRef:2]**

要理解dom-diff算法，就要先理解virtual dom的概念。在一个大型的页面应用中，如果每次数据更新都使用真实dom节点，将会消耗巨大的性能。因此React、Vue等框架都会采用virtual dom，用一个相对简单的js结构去代表真实dom，一般会包含标签名、属性、子元素对象等基础属性。每次数据驱动更新时，都会涉及大量的virtual dom对比，找出需要更新的dom。这时候，我们就需要使用dom-diff算法，优化比较运算时的计算负担。

本章节我们简要分析dom-diff算法的原理[footnoteRef:3]。

[2] dom-diff源码简化版 <https://github.com/aooy/aoy/blob/master/src/vdom/diff.js>

[3] 参考<https://segmentfault.com/a/1190000020663531?utm_source=tag-newest>

1. virtual dom的基本结构（不同框架的属性名可能不一样）

// body下的 <div id="v" class="classA"><div> 对应的 oldVnode 就是

{

el: div //对真实的节点的引用，本例中就是document.querySelector('#id.classA')

tag: 'DIV', //节点的标签

sel: 'div#v.classA' //节点的选择器

data: null, // 一个存储节点属性的对象，对应节点的el[prop]属性，例如onclick , style

children: [], //存储子节点的数组，每个子节点也是vnode结构

text: null, //如果是文本节点，对应文本节点的textContent，否则为null

}

2）比较节点的方法

方法接收旧节点oldVnode和新节点vnode两个参数。如果节点相同，则执行patchVnode方法更新节点。如果节点不相同，则使用新节点替换救旧节点。

function patch (oldVnode, vnode) {

// some code……

if (sameVnode(oldVnode, vnode)) {

patchVnode(oldVnode, vnode)

} else {

const oEl = oldVnode.el // 当前oldVnode对应的真实元素节点

let parentEle = api.parentNode(oEl) // 父元素

createEle(vnode) // 根据Vnode生成新元素

if (parentEle !== null) {

api.insertBefore(parentEle, vnode.el, api.nextSibling(oEl)) // 将新元素添加进父元素

api.removeChild(parentEle, oldVnode.el) // 移除以前的旧元素节点

oldVnode = null

}

}

// some code

return vnode

}

3）判断节点是否相同的标志

可以看到key值是判断节点是否相同的标志，因此在循环定义组件的时候，定义key值可以优化节点比较的性能。

function sameVnode (a, b) {

return (

a.key === b.key && // key值

a.tag === b.tag && // 标签名

a.isComment === b.isComment && // 是否为注释节点

// 是否都定义了data，data包含一些具体信息，例如onclick , style

isDef(a.data) === isDef(b.data) &&

sameInputType(a, b) // 当标签是<input>的时候，type必须相同

)

}

4）更新节点

function patchVnode (oldVnode, vnode) {

const el = vnode.el = oldVnode.el

let i, oldCh = oldVnode.children, ch = vnode.children

if (oldVnode === vnode) return//相等，无须更新

if (oldVnode.text !== null && vnode.text !== null && oldVnode.text !== vnode.text) {

api.setTextContent(el, vnode.text);//更新文本节点

}else {

updateEle(el, vnode, oldVnode)

if (oldCh && ch && oldCh !== ch) {

updateChildren(el, oldCh, ch)//更新children节点

}else if (ch){

createEle(vnode) //只有新子节点，创建

}else if (oldCh){

api.removeChildren(el)//只有旧子节点，移除

}

}

}

5）更新子节点

遍历子节点通过二分逼近算法进行优化。oldCh和newCh各有两个头尾的变量StartIdx和EndIdx，它们的2个变量相互比较。

一共有4种比较方式。

* 头部尾部的同类型节点，更新前后位置没有发生变化，所以不用移动它们对应的DOM：

oldStartVnode === newStartVnode 新旧节点的起始节点相同，直接更新内容

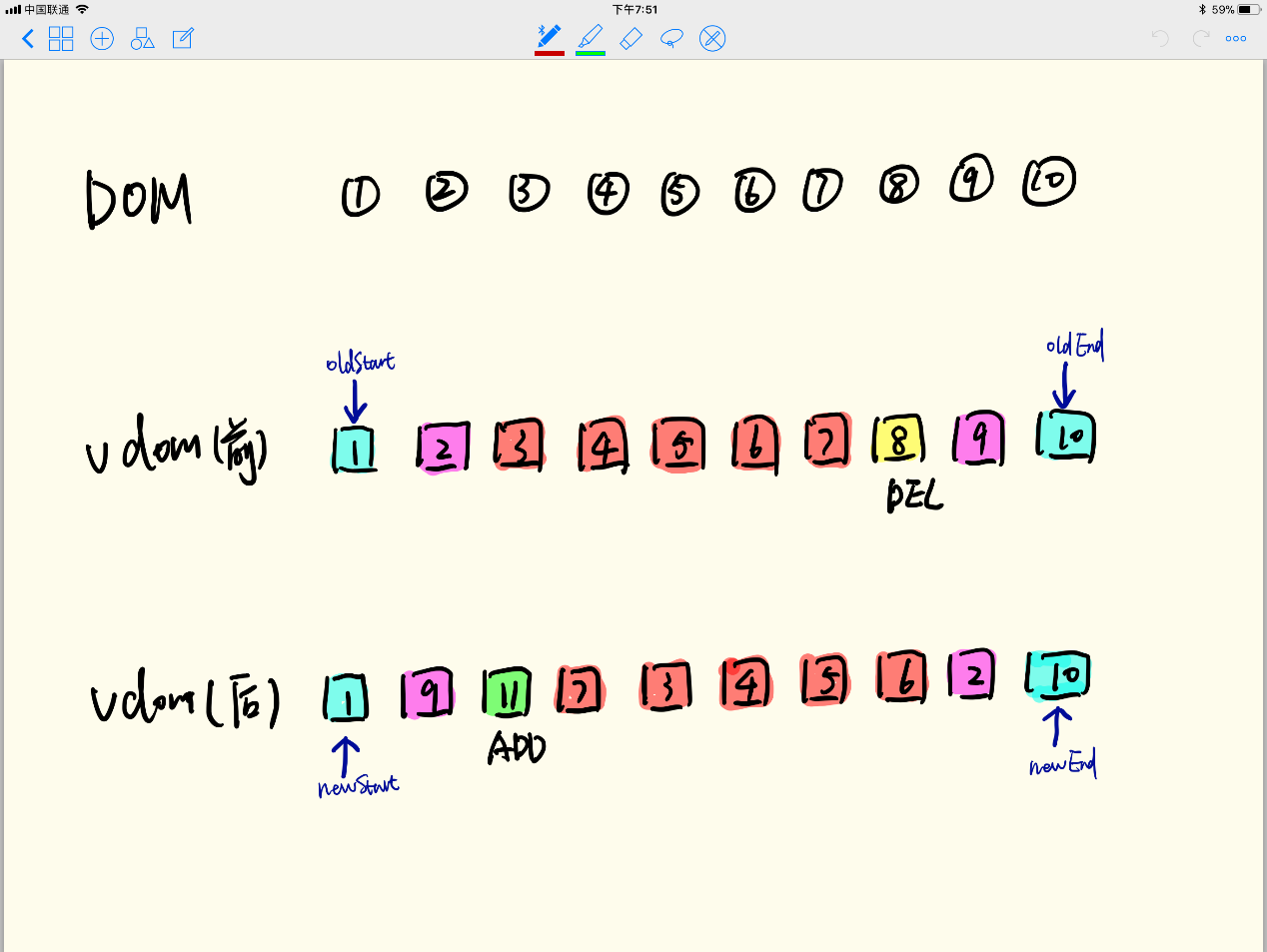
oldEndVnode === newEndVnode 新旧节点的末尾节点相同，直接更新内容

* 头尾、尾头的同类型节点，直接移动dom：

oldStartVnode === newEndVnode 旧节点的起始节点与新节点的末尾节点相同，更新节点内容并把oldStartVnode插入到oldEndVnode前

oldEndVnode === newStartVnode 比较旧节点的末尾节点与新节点的起始节点，更新节点内容并把oldEndVnode插入到oldStartVnode前

参考下图可以比较直观地理解



如果4种比较都没匹配，且设置了key，就会比较key值。

如果上述比较都没有匹配，则说明是新增或删除节点

在比较的过程中，变量会向中间逼近，一旦StartIdx>EndId，表明oldCh和newCh至少有一个已经遍历完了，则跳出循环[footnoteRef:4]。

[4]参考<https://segmentfault.com/a/1190000020663531?utm_source=tag-newest>

function updateChildren (parentElm, oldCh, newCh) {

let oldStartIdx = 0, newStartIdx = 0

let oldEndIdx = oldCh.length - 1

let oldStartVnode = oldCh[0]

let oldEndVnode = oldCh[oldEndIdx]

let newEndIdx = newCh.length - 1

let newStartVnode = newCh[0]

let newEndVnode = newCh[newEndIdx]

let oldKeyToIdx

let idxInOld

let elmToMove

let before

while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {

if (oldStartVnode == null) {

//对于vnode.key的比较，会把oldVnode = null

oldStartVnode = oldCh[++oldStartIdx]

}else if (oldEndVnode == null) {

oldEndVnode = oldCh[--oldEndIdx]

}else if (newStartVnode == null) {

newStartVnode = newCh[++newStartIdx]

}else if (newEndVnode == null) {

newEndVnode = newCh[--newEndIdx]

}else if (sameVnode(oldStartVnode, newStartVnode)) {

// 开始比较四种特殊情况

patchVnode(oldStartVnode, newStartVnode)

oldStartVnode = oldCh[++oldStartIdx]

newStartVnode = newCh[++newStartIdx]

}else if (sameVnode(oldEndVnode, newEndVnode)) {

patchVnode(oldEndVnode, newEndVnode)

oldEndVnode = oldCh[--oldEndIdx]

newEndVnode = newCh[--newEndIdx]

}else if (sameVnode(oldStartVnode, newEndVnode)) {

patchVnode(oldStartVnode, newEndVnode)

api.insertBefore(parentElm, oldStartVnode.el, api.nextSibling(oldEndVnode.el))

oldStartVnode = oldCh[++oldStartIdx]

newEndVnode = newCh[--newEndIdx]

}else if (sameVnode(oldEndVnode, newStartVnode)) {

patchVnode(oldEndVnode, newStartVnode)

api.insertBefore(parentElm, oldEndVnode.el, oldStartVnode.el)

oldEndVnode = oldCh[--oldEndIdx]

newStartVnode = newCh[++newStartIdx]

}else {

// 使用key比较

if (oldKeyToIdx === undefined) {

oldKeyToIdx = createKeyToOldIdx(oldCh, oldStartIdx, oldEndIdx) // 有key生成index表

}

idxInOld = oldKeyToIdx[newStartVnode.key]

if (!idxInOld) {

//key值不在index表里，新增节点

api.insertBefore(parentElm, createEle(newStartVnode).el, oldStartVnode.el)

newStartVnode = newCh[++newStartIdx]

}

else {

//key值在index表里

elmToMove = oldCh[idxInOld]

//新旧节点的选择器不一样，则新增节点

if (elmToMove.sel !== newStartVnode.sel) {

api.insertBefore(parentElm, createEle(newStartVnode).el, oldStartVnode.el)

}else {

//新旧节点的选择器相同，则更新节点内容并移动dom

patchVnode(elmToMove, newStartVnode)

oldCh[idxInOld] = null//标记节点已经被处理过

api.insertBefore(parentElm, elmToMove.el, oldStartVnode.el)//移动dom

}

newStartVnode = newCh[++newStartIdx] //更新指针

}

}

}

//跳出循环后

//旧数组的起始指针比末尾指针大

//说明新数组中还有需要新增的节点

//反之，旧数组中还有需删除的节点

if (oldStartIdx > oldEndIdx) {

before = newCh[newEndIdx + 1] == null ? null : newCh[newEndIdx + 1].el

addVnodes(parentElm, before, newCh, newStartIdx, newEndIdx)

}else if (newStartIdx > newEndIdx) {

removeVnodes(parentElm, oldCh, oldStartIdx, oldEndIdx)

}

}

### **4.2 Vue应用实操**

#### **4.2.1组件编写（仿element-ui表单组件el-form实现）**

注意：Provide, inject,$parent,$children在组件中的使用

* 组件基本结构

el-form.vue

<template>

<form><slot></slot></form>

</template>

<script>

export default {

name:'elForm'

}

</script>

el-form-item.vue

<template>

<div><slot></slot></div>

</template>

<script>

export default {

name:'elFormItem'

}

</script>

el-input.vue

<template>

<input type="text">

</template>

<script>

export default {

name:'elInput'

}

</script>

* 组件传值方式

使用**Provide**，接收用户属性，并将当前组件提供出去，因为el-form 和 el-form-item 不一定是父子关系，可能是父孙关系

<template>

<form><slot></slot></form>

</template>

<script>

export default {

name:'elForm',

provide(){

return {elForm:this}

},

props:{

model:{

type:Object,

default:()=>({})

},

rules:Object

}

}

</script>

使用**inject**,从而解决了跨组件通信（provide和inject不要在业务逻辑中使用）

<template>

<div><slot></slot></div>

</template>

<script>

export default {

name:'elFormItem',

inject:['elForm'],

props:{

label:{

type:String,

default:''

},

prop:String

},

mounted(){

console.log(this.elForm)

}

}

</script>

**v-model** 双向绑定响应更新，组件的v-model其实就是 input + value的语法糖

<template>

<input type="text" :value="value" @input="handleInput">

</template>

<script>

export default {

name:'el-input',

props:{

value:String

},

methods:{

handleInput(e){

this.$emit('input',e.target.value);

}

}

}

</script>

* 组件事件通信

使用**$parent**。Vue在组件初始化的过程中会创造父子关系，为了方便通信我们使用$parent属性来自己封装一个$dispatch方法并挂载到原型链上，用于触发对应的祖先组件中的方法

Vue.prototype.$dispatch = function (componentName,eventName) {

const parent = this.$parent;

while (parent) {

let name = parent.$options.name;

if (name == componentName) {

break;

} else {

parent = parent.$parent;

}

}

if(parent){

if(eventName){

return parent.$emit(eventName)

}

return parent

}

}

在组件中使用：

handleInput(e){

this.$emit('input',e.target.value);

// 找到对应的formItem进行检测

this.$dispatch('elFormItem','validate');

}

**$on**手动绑定事件

<template>

<div>

<label v-if="label">{{label}}</label>

<slot></slot>

{{errorMessage}}

</div>

</template>

<script>

import Schema from "async-validator";

export default {

name: "elFormItem",

inject: ["elForm"],

props: {

label: {

type: String,

default: ""

},

prop: String

},

data(){

return {errorMessage:''}

},

mounted() {

this.$on("validate", () => {

if (this.prop) {

let rule = this.elForm.rules[this.prop];

let newValue = this.elForm.model[this.prop];

let descriptor = {

[this.prop]: rule

};

let schema = new Schema(descriptor);

return schema.validate({[this.prop]:newValue},(err,res)=>{

if(err){

this.errorMessage = err[0].message;

}else{

this.errorMessage = ''

}

})

}

});

}

};

</script>

使用**$children**，同理：Vue在组件初始化的过程中会创造父子关系，为了方便通信我们使用$children属性来自己封装一个$broadcast方法，用于触发对应后代组件中的方法

Vue.prototype.$broadcast = function (componentName,eventName) {

let children = this.$children;

let arr = [];

function findFormItem(children){

children.forEach(child => {

if(child.$options.name === componentName){

if(eventName){

arr.push(child.$emit('eventName'))

}else{

arr.push(child)

}

}

if(child.$children){

findFormItem(child.$children);

}

});

}

findFormItem(children);

return arr;

}

在组件中调用校验方法：

<template>

<form @submit.prevent>

<slot></slot>

</form>

</template>

<script>

export default {

name: "elForm",

provide() {

return { elForm: this };

},

props: {

model: {

type: Object,

default: () => ({})

},

rules: Object

},

methods: {

async validate(cb) {

let children = this.$broadcast("elFormItem");

try{

await Promise.all(children.map(child=>child.validate()));

cb(true);

}catch{

cb(false)

}

}

}

};

</script>

el-form-item改写校验方法

methods: {

validate() {

if (this.prop) {

let rule = this.elForm.rules[this.prop];

let newValue = this.elForm.model[this.prop];

let descriptor = {

[this.prop]: rule

};

let schema = new Schema(descriptor);

return schema.validate({ [this.prop]: newValue }, (err, res) => {

if (err) {

this.errorMessage = err[0].message;

} else {

this.errorMessage = "";

}

});

}

}

},

mounted() {

this.$on("validate", () => {

this.validate();

});

}

#### **4.2.2路由拦截，权限鉴定**

export const ADD\_ROUTE = 'ADD\_ROUTE' // 添加路由动作

export const SET\_MENU\_PERMISSION = 'SET\_MENU\_PERMISSION' // 表示菜单权限已经拥有

export const menuPermission = async function(to, from, next) {

if (store.state.user.hasPermission) {

if (!store.state.user.menuPermission) {

store.dispatch(`user/${types.ADD\_ROUTE}`);

next({...to,replace:true});

} else {

next();

}

} else {

next();

}

}

根据用户返回的权限过滤需要的路由

import router from '@/router/index'

import per from '@/router/per';

async [types.ADD\_ROUTE]({ commit, state }) {

let authList = state.userInfo.authList;

if (authList) {

// 开始 规划路由

let routes = filterRouter(authList);

let route = router.options.routes.find(item => item.path === '/manager');

route.children = routes;

router.addRoutes([route]);

commit(types.SET\_MENU\_PERMISSION, true);

}

}

过滤的当前用户支持的路由

const filterRouter = (authList) => {

let auths = authList.map(item => item.auth);

const filter = (authRoutes) => {

let result = authRoutes.filter(route => {

if (auths.includes(route.meta.auth)) {

if (route.children) {

route.children = filter(route.children);

}

return route;

}

})

return result

}

return filter(per);

}

[types.SET\_MENU\_PERMISSION](state, has) {

state.menuPermission = has;

}

#### **4.2.3自定义指令、插件**

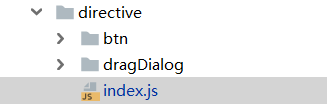
**自定义指令**：除了Vue内置的指令如v-model、v-show等，我们也可以根据业务需要定义自定义指令。如果需要对dom元素进行底层操作，且功能逻辑多次复用，那么就可以使用自定义指令提高开发效率。

**插件**：插件通常用来为Vue添加全局功能。开发插件的核心是实现install方法，install方法接收Vue构造器和可选项两个参数。方法内可实现全局混入方法、挂载全局资源、全局混入组件选项、扩展Vue原型链等操作。

* 推荐文件结构

在项目中需要扩展指令或者，全局组件和原型方法，可以在项目src下定义一个directive文件夹

把定义好的指令在index统一导出



import btn from './btn'

import dragDialog from '@/directive/dragDialog'

export default {

install (Vue) {

Vue.use(btn)

Vue.use(dragDialog)

}

}

在main.js中引入，即可在全局注册指令和方法

import directive from './directive'

Vue.use(directive)

* 自定义指令+插件鉴权实例：

Permission.js

export default {

  install(Vue, options = {}) {

    const defaultAuthMap = {

      'read': 2,

      'operation': 4,

      'management': 8

};

//全局混入权限指令

  Vue.directive('permission', function(el, binding) {

      if (binding.value !== binding.oldValue && binding.oldValue !== undefined) {

        init(el, binding);

      }

    });

//拓展原型链

 /\*\*

     \* 判断默认权限

     \* @param {String|Number} cdoe 功能编码

     \* @param {Stirng} codeName    功能名称

     \*

     \* @return Boolean

     \*/

    Vue.prototype.$hasDefaultPermission = (code, codeName) => {

     //业务实现…

    };

/\*\*

\* @param {Element } el 使用指令的dom节点

  \* @param {Object} binding 接收参数

 \*/

 function init(el, binding) {

      let checkCodes = {};

      let value = binding.value;///获取指令的绑定值

let arg = binding. arg; ///获取指令的传参

let modifiers = binding. modifiers; ///获取指令的修饰符

      let level = '';

      let levelCode = '';

      if (value === undefined) throw new Error('Permission directive must have parameters!');

//

      if (arg === 'default') {

        checkCodes = defaultAuth;

      }

      let access = false;

//计算参数的权限等级

      if (arg === 'default' && value !== null) {

        if (modifiers.read) {

          level = parseInt(value.trim(), 10);

          levelCode = 'read';

        } else if (modifiers.operation) {

       //其他业务逻辑

        }

     //默认权限遍历判断，等级高包括等级低

        for (const code in checkCodes) {

          if (level >= checkCodes[code] && levelCode === code) {

            access = true;

            break;

          }

        }

      }

      if (!access) {

   //使用nextTick才能正常找到parentNode

        Vue.nextTick(() => {

          el.parentNode && el.parentNode.removeChild(el);

        });

      }

    }

}

组件中使用：

  <el-button

type="primary"

plain icon="iconfont icon-tianjia"

     v-permission:default.operation="defaultAuthority"

      @click="handleAddWork">新建工作

</el-button>

#### **4.2.4通信：父子组件、兄弟组件、祖孙组件**

Vue组件通信方式有多种，以组件传值的层级进行分类，简要列举在实际开发过程中推荐使用的通信方式：

**父子组件**：

* props和$emit 父组件向子组件传递数据是通过prop传递的，子组件传递数据给父组件是通过$emit触发事件来做到的
* $parent,$children 获取当前组件的父组件和当前组件的子组件
* $refs 获取子组件实例

**祖孙组件（多级嵌套）：**

* $attrs和$listeners A->B->C。Vue 2.4 开始提供了$attrs和$listeners来解决这个问题（祖孙之间数据传递）
* 父组件中通过provide来提供变量，然后在子孙组件中通过inject来注入变量。

**兄弟组件：**

* eventBus平级组件数据传递 这种情况下可以使用中央事件总线的方式

**全局状态管理：**

* vuex状态管理（全局的数据管理）

## **Angular指南**

*（暂不编写）*

# **兼容指南**

众所周知，前端开发一直面临着兼容性的问题，不管是跨浏览器还是跨平台，浏览界面的介质不一，尺寸不一，也就会出现不同的渲染效果和问题。

因此，前端团队有必要根据针对应用所面对的用户情况、应用类型、开发成本等因素，来制定应用的浏览器兼容规范，并写入应用使用手册中。除非特殊要求，不符合浏览器兼容规范的浏览器，前端开发人员可以选择性忽略。

*本章所有内容均基于Web开发，不讨论App开发。*

## **浏览器兼容性**

浏览器生态丰富，不同厂商甚至不同版本之间差异性都很大，经历过IE6~8版本兼容处理的开发都懂得。Web开发很难做到所有版本的浏览器都兼容，而且不同的用户群体使用的浏览器版本趋势不同，所以在开发前有必要先收集用户使用浏览器的统计数据，再确定兼容策略。

*本节仅讨论PC端的浏览器兼容问题。*

### **确定浏览器分级**

除IE自身版本之间差异较大外，其他浏览器由于内核不一，对脚本和样式的处理方式也不一样，国内常见浏览器的兼容性如下表。

浏览器兼容性统计表[footnoteRef:5]

|  |  |  |  |
| --- | --- | --- | --- |
| 浏览器 | 兼容级别 | 划分 | 支持性 |
| IE6/7/8 | 极难兼容 | 老式浏览器 | 不支持HTML5、CSS3、SVG标准 |
| IE9 | 较难兼容 | 老式浏览器 | 不支持Flex、Web Socket、WebGL |
| IE10 | 较易兼容 | 准现代浏览器 | 部分支持Flex（-ms-flexbox）、Web Socket |
| IE11 | 较易兼容 | 准现代浏览器 | 部分支持Flex、WebGL |
| Chrome/Firefox  Safari/Opera/Edge | 易兼容 | 现代浏览器 |  |

[5] 参考：[Web前端兼容性指南](https://my.oschina.net/u/4286781/blog/4319039)（2020-06-22）

国内常见浏览器内核表[footnoteRef:6]

|  |  |  |
| --- | --- | --- |
| 浏览器 | 内核 | 引擎前缀 |
| IE/百度 | Trident（俗称IE内核） | -ms- |
| Edge | Chromium（俗称Chrome内核） | -ms- |
| Chrome | Blink | -webkit- |
| Firefox | Gecko（俗称FF内核） | -moz- |
| Safari | Webkit | -webkit- |
| Opera | Blink | -webkit- （旧版为 -o-） |
| 360/猎豹/2345 | IE+Chrome 双内核 |  |
| 搜狗/遨游/QQ | Trident（兼容模式）+Webkit（高速模式） |  |

注意：针对CSS3新属性，需要加浏览器前缀来兼容早期浏览器

[6] 参考：[浏览器引擎前缀（2021-04-15）](https://developer.mozilla.org/zh-CN/docs/Glossary/Vendor_Prefix)、[浏览器兼容常见问题（2020-12-11）](https://blog.csdn.net/MaybyYouLove/article/details/111055101)

在认识到各浏览器的特点之后，可对浏览器进行分级，以表示应用对不同浏览器的支持程度。比如以下三个等级[footnoteRef:7]，可作为应用的兼容规范说明：

* **完全兼容**: 保证百分百功能正常。（如：Chrome49+，Firefox44+）
* **部分兼容**: 只能保证功能、样式与需求大致一致，对于一些不影响主体需求和功能的bug，会做降低优先级处理或者不处理。（如：Chrome28+，Firefox37+，IE11）
* **不兼容**: 不考虑兼容性。（如：其他）

一般而言, 需根据浏览器市场分布情况、用户占比、开发成本等因素划分等级。

[7] 参考：[if 我是前端团队Leader，怎么制定前端协作规范?](https://bobi.ink/2019/07/19/frontend-standard/#32-%E7%A1%AE%E5%AE%9A%E6%B5%8F%E8%A7%88%E5%99%A8%E5%88%86%E7%BA%A7)

### **获取统计数据**

在需求整理阶段，可确定用户群体，借助人工手段或在线工具统计用户使用浏览器的情况，从而选择性进行兼容开发，节约开发成本。

**一般应用推荐兼容：Chrome、Firefox、Edge、IE11、360**

在线应用监测：

[百度统计](https://tongji.baidu.com/web/demo/visit/client?siteId=5503017)、[友盟](https://web.umeng.com/main.php?spm=a211g2.211692.0.0.3a437d23sjzEPv&c=user&a=index)、[Google Analytics （需要科学上网）](https://analytics.google.com/analytics/web/)等

通用数据统计：

[百度流量研究院（主要提供国内浏览器统计）](https://tongji.baidu.com/research/site?source=index)

[statcounter（国际浏览器统计）](http://gs.statcounter.com/)

[浏览器发布年份统计](https://en.wikipedia.org/wiki/Timeline_of_web_browsers)

### **确定兼容策略**

**渐进增强**还是**优雅降级**，这是两个不同方向策略。**渐进增强**保证低版本浏览器的体验，对于支持新特性的新浏览器提供稍好的体验；**优雅降级**则相反，为现代浏览器提供最好的体验，而旧浏览器则退而求之次，保证大概的功能.

选择不同的策略对前端开发的影响是比较大的，但是开发者没有选择权。**确定哪种兼容策略，应该取决于用户比重，如果大部分用户使用的是现代浏览器，就应该使用优雅降级，反之选择渐进增强**。

### **推荐做法**

1. **条件注释法**

在正常代码之外添加判别IE浏览器或对应版本的条件注释，符合条件的浏览器或者版本号才执行代码。

HTML条件注释关键字说明表

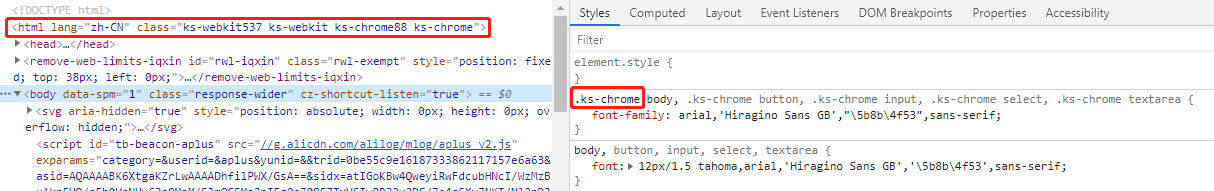
|  |  |  |
| --- | --- | --- |
| 关键字 | 对应运算 | 说明 |
| lt | < | 小于 |
| gt | > | 大于 |
| lte | <= | 小于或等于 |
| gte | >= | 大于或等于 |
| ! | != | 不等于 |

条件

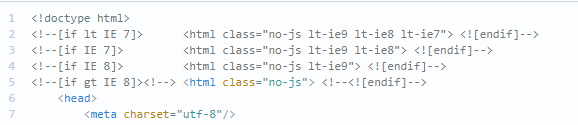
1. **HTML加上特殊类**

在<html>标签上加上操作系统、浏览器内核、浏览器类型、CSS3动画支持、IE各版本类，然后按照类名定义不同环境下的样式，可实现不同浏览器下的差异化体验，也便于快速定位并修复某个浏览器下的特定bug。

如淘宝首页HTML类：



也可用条件注释法为不同浏览器添加特定类：



1. **引入浏览器兼容框架**

* [normalize.css](https://github.com/necolas/normalize.css/)：让不同的浏览器在渲染网页元素的时候形式更统一
* [html5shiv.js](https://github.com/aFarkas/html5shiv)：让IE6~IE8识别HTML5标签，并且可以添加CSS样式
* [respond.js](https://github.com/scottjehl/Respond)：使IE6~IE8浏览器支持CSS3媒体查询

可采用条件注释法引入js：

<!-- WARNING: Respond.js doesn't work if you view the page via file:// -->

<!--[if lt IE 9]>

<script src="https://oss.maxcdn.com/libs/html5shiv/3.7.0/html5shiv.js"></script>

<script src="https://oss.maxcdn.com/libs/respond.js/1.3.0/respond.min.js"></script>

<![endif]-->

1. **引入样式前缀自动化插件**

上述有提到不同浏览器有不同的引擎前缀，针对css3新属性，需要加浏览器前缀来兼容早期浏览器。**Autoprefixer**是一款自动管理浏览器前缀的插件，它可以解析CSS文件并添加浏览器前缀到CSS内容中，使用Can I Use网站的数据来决定哪些前缀是需要的。

把Autoprefixer添加到资源构建工具（如webpack、gulp、grunt）后，只需按照最新的W3C规范来正常书写CSS即可。如果项目需要支持旧版浏览器，可修改browsers参数设置。

// 我们编写的样式

div {

transform: rotate(30deg);

}

// 自动补全的样式，具体补全哪些由要兼容的浏览器版本决定，可以自行设置

div {

-ms-transform: rotate(30deg);

-webkit-transform: rotate(30deg);

-o-transform: rotate(30deg);

-moz-transform: rotate(30deg);

transform: rotate(30deg);

}

1. **IE采用最新渲染模式**

在<header>标签中加入下面这段代码：

<meta http-equiv= "x-ua-compatible" content= "ie=edge">

1. **自查兼容性**

在编写代码过程中，针对性自查编码的兼容性。

在线自查工具：

[CanIuse](https://caniuse.com/)（主要为html5、css3兼容性检测）

[QuirksMode](http://www.quirksmode.org/compatibility.html)（英文网，有移动端兼容性参考表）

### **常见兼容方案**

* 1. **CSS相关**
* 不同浏览器的标签默认的外margin 和内padding不同

在css入口中添加以下通配符样式，也可引入上文提到的Normalize来清除默认样式

\* { margin: 0; padding: 0; }

* IE9以下版本浏览器不能使用opacity

opacity: 0.5; // normal

filter: alpha(opacity = 50); // for IE7~IE8

filter: progid:DXImageTransform.Microsoft.Alpha(style=0,opacity=60); // for IE6

* Firefox、Safari 不支持cursor: hand

// 统一使用

cursor: pointer;

* 清除浮动

.clearfix:after {

content: "";

display: block;

height: 0;

clear: both;

visibility: hidden;

}

* 超链接访问过后hover样式丢失

主要是被点击访问过的超链接样式不在具有hover和active，解决方法是改变CSS属性的排列顺序: **L-V-H-A**

a:link{…} a:visited{…} a:hover{…} a:active{…}

* chrome下默认会将小于12px的文本强制按照12px来解析

// css入口加入以下写法

-webkit-text-size-adjust: none;

* 消除ul、ol等列表的缩进

ul, ol {

list-style: none;

margin: 0; // for ie、firefox

padding: 0; // for firefox

}

* 1. **JS相关**
* Firefox不支持innerText

用textContent处理innerText的兼容性问题

function getInnerText(element) {

if (typeof element.innerText === 'string') {

return element.innerText;

} else {

return element.textContent

}

}

* 获取网页可见区域的宽高、整个区域的宽高、滚动的距离

// 可见区域的宽高

const winW = document.body.clientWidth || document.documentElement.clientWidth;

const winH = document.body.clientHeight || document.documentElement.clientHeight;

// 整个区域的宽高

const winW = document.body.scrollWidth || document.documentElement.scrollWidth;

const winH = document.body.scrollHeight || document.documentElement.scrollHeight;

// 网页被卷去的高

const scrollHeight = document.body.scrollTop || document.documentElement.scrollTop;

// 网页左卷的距离

const scrollLeft = document.body.scrollLeft || document.documentElement.scrollLeft;

* 事件绑定与解绑

// 添加事件监听，三个参数分别为：对象、事件类型、事件处理函数

function addEvent(dom, type, fn) {

if (dom.addEventListener) {

dom.addEventListener(type, fn ,false); // 非IE

} else {

dom.attachEvent('on' + type, fn); // IE

}

}

// 删除事件监听

function removeEvent(dom, type, fn) {

if (dom.removeEventListener) {

dom.removeEventListener(type, fn, false); // 非IE

} else{

dom.detachEvent('on' + type, fn); // IE

};

}

* Event事件对象的兼容写法

// Chrome、Firefox、IE9+支持，IE9以下不支持；

document.onclick=function(ev){

const e=ev;

console.log(e);

}

// Chrome和IE支持，Firefox不支持；

document.onclick=function(){

const e= window.event;

console.log(e);

}

// 兼容写法

document.onclick = function (ev) {

const e = ev || window.event;

}

* 停止事件冒泡

// 兼容写法

if (e.stopPropagation) { // 针对非ie浏览器

e.stopPropagation();

} else { // 针对ie浏览器

e.cancelBubbles = true;

}

* 阻止浏览器默认行为

// 兼容写法

if (e.preventDefault) { // 针对非ie浏览器

e.preventDefault();

} else { // 针对ie浏览器

window.event.returnValue = false;

}

* 获取dom事件的target对象

//兼容写法

// IE浏览器：e.srcElement;

// 其他浏览器：e.target;

dom.onclick = function(eve) {

const e = eve || window.event;

const target = e.target || e.srcElement;

console.log(target.innerHTML);

}

* 获取键盘事件

// 兼容写法

// IE浏览器：event.which

// 其他浏览器：event.keycode

const e = eve || window.event;

const code = e.keyCode || e.which

* 获取dom的外部样式

// 兼容写法

// dom.style.attr 无法获取外部样式

function getStyle(dom,attr) {

     if (dom.currentStyle) {

          return dom.currentStyle[attr]; // 兼容IE

     } else {

          return getComputedStyle(dom, null)[attr]; // 兼容Chrome、Firefox

     }

}

### **测试手段**

整体性**人肉测试**有三种解决方案：

1. 在线测试工具：提交应用地址，通过对方服务器，在各种机器或设备的浏览器上运行并测试网站，大部分是生成各浏览器对页面渲染的截图。
2. 本地安装虚拟机：实现各种操作系统、各种浏览器的测试，这种方式检测出来的结果真实，但非常消耗硬盘和CPU资源，搭建过程也较为复杂繁琐。
3. 公用测试服务器：公司搭建专门的兼容性测试服务器，虽然解决了本机安装的问题，但对服务器资源的利用率低消耗率高。（公司内部暂时没有）

除了人肉测试之外，也可借助云端解决方案实现完整测试并获取优化方案（如：F2etest阿里云镜像），但需要购买云服务器，通常提供的功能越多，价格也越高。

## **屏幕分辨率兼容性**

在不同的屏幕分辨率，浏览器页面展示差异很大。特别是屏幕分辨率较小时，容易发生布局错乱。为了解决这个问题，响应式设计方案应运而生。

### **相关概念**

按照设备类型，屏幕可划分为**桌面屏幕（PC）、平板屏幕（Pad）和移动屏幕（Mobile）**三种类型。

1. 手机屏幕分辨率说明

手机屏幕由于尺寸过小，使用原始分辨率会使得页面显示过小，因此使用了**逻辑分辨率（CSS中的PX）**，用倍数放大的方式（DPR）来保证兼容性。所以屏幕分辨率为360\*640的手机，系统查询出来的设备分辨率可能为1080\*1920甚至更高。

1. 桌面屏幕分辨率说明

移动设备一开始就考虑了DPR，而Windwos桌面的分辨率由于历史原因没有这一概念，于是引入了DPI和DPI比例（DPIR）。比如设置DPIR=125%，可以查询Chrome的window.devicePixelRatio，这时输出1.25，这说明DPIR=DPR。但需要注意的是与手机屏幕分辨率不同，**桌面分辨率要除以DPIR，才是逻辑分辨率**。如1920\*1080设置DPIR=1.25，逻辑分辨率实际为1536\*864。

屏幕分辨率基础概念说明表1 见上方

|  |  |  |
| --- | --- | --- |
| 缩写 | 全称 | 说明 |
| PX | Device Pixels | 设备像素，指设备的物理像素 |
| PX | CSS Pixels | CSS像素，指CSS样式代码中使用的逻辑像素 |
| DPR | Device Pixel Ratio | 设备像素比，指CSS逻辑像素对于物理像素的倍数 |
| PPI | Pixel Per Inch | 屏幕上每英寸(2.54厘米)的像素点个数 |
| DPI | Dots Per Inch | 屏幕或纸上每英寸(2.54厘米)的点个数，标准密度：传统打印=72；Windows=96；Android=160；iOS=163。 |
| DPIR | DPI Ratio | DPI缩放比例，指DPI对于Windows标准DPI的倍数=DPI/96，等同于DPR |

不同的设备类型有着不同的分辨率，但每种设备类型的分辨率都有着大致的分段区间，如Bootstrap5就对各类型的屏幕尺寸进行了如下分割定义：

Bootstrap5尺寸分割定义（参考系是逻辑分辨率）1 见上方

|  |  |  |
| --- | --- | --- |
| 分辨率 | 设备名 | 典型屏幕 |
| >=1400px | xxl 超超大屏设备 | 桌面屏幕 |
| >=1200px | xl 超大屏设备 | 便携屏幕 |
| >=992px | lg 大屏设备 | 竖屏桌面屏幕、横屏平板屏幕 |
| >=768px | md 中屏设备 | 竖屏平板屏幕 |
| >=576px | sm 小屏设备 | 横屏移动屏幕 |
| <576px | xs 超小屏（自动）设备 | 竖屏移动屏幕 |

传统的开发方式是PC端开发一套，手机端再开发一套，开发成本过高，代码复用性较弱。而如Bootstrap之类的响应式布局，能够实现同一页面在不同屏幕尺寸下有不同的布局，开发一套就可满足多端展示需求，缺点就是CSS比较重。

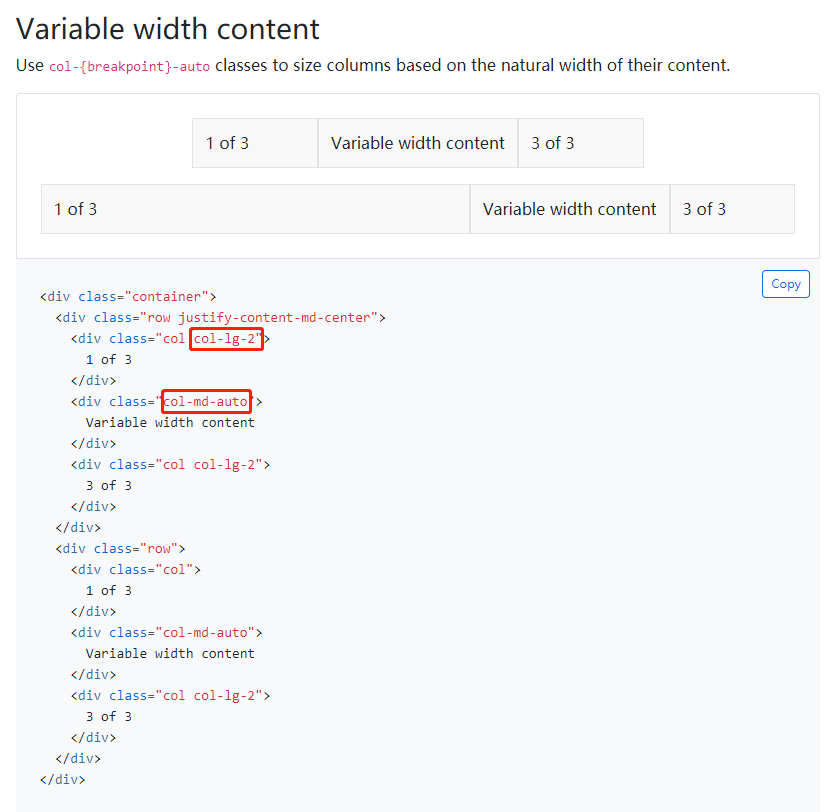
### **推荐做法**

确定屏幕分辨率兼容策略的方式和浏览器兼容性类似，首先要确定应用的使用场景，再考虑各分辨率下兼容方案。不管是何种方案，都可考虑以下几点：

1. **栅格布局（Grid layout）**

栅格布局是以行列交叉形成的固定格子来设计页面排版，使其风格工整简洁，这也是一种常见的平面设计方法。很多UI库具备该功能，如Bootstrap、Ant Design、Element。

其中Bootstrap的栅格系统，采用了**flexbox**设计，将整个页面的宽度分为1~12等分，作为栅格单位。其他框架类似。



1. **CSS3媒体查询（@media）**

CSS3媒体查询可以让我们针对不同的媒体类型定义不同的样式，当重置浏览器窗口大小的过程中，页面也会根据浏览器的宽度和高度重新渲染页面，因此能同时适应PC端和移动端。

Bootstrap的分段区间就是通过定义媒体查询实现的，但不管是哪种分割方案，都会面临移动优先还是PC优先的问题。

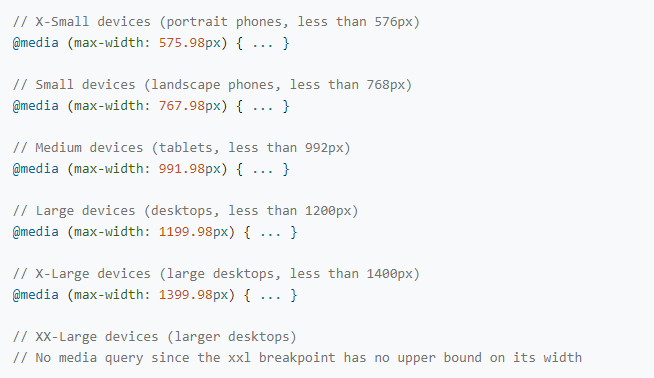
**移动优先**，默认是用手机浏览该网页的，当用PC浏览时，就会以**min-width**进行递增式媒体查询；**PC优先**与之相反，默认是用PC浏览该网页，当屏幕缩小时以**max-width**递减式进行媒体查询。

Bootstrap最开始是针对移动端的解决方案，因此主要采用的是移动优先，特殊情况下采用PC优先。Bootstrap的分割方案不一定能满足项目中的实际需求，建议先用跨度大的分割点进行分割，如果出现不适配的情况，再根据实际情况增加新的分割点。

移动优先的写法（以Bootstrap为例）：



PC优先的写法（以Bootstrap为例）：



1. **REM布局**

REM是CSS3新增的单位，并且移动端的支持度很高，Android2.x+、ios5+都支持。**rem单位相对于根元素html的font-size来决定大小**（**em相对于父元素**），根元素的font-size相当于提供了一个基准，当页面的size发生变化时，只需要改变font-size的值，那么以rem为固定单位的元素的大小也会发生响应的变化。

在实际运用中，所有设置的固定值都用rem做单位（首先在HTML中设置一个基准值：px和rem的对应比例，然后在效果图上获取px值，布局的时候转化为rem值)。可参考运用方案：大屏上的全屏页面的自适应适配方案（2018-05-15）

需要注意的是，在响应式布局中，必须通过js来动态控制根元素font-size的大小，也就是说css样式和js代码有一定的耦合性，且**必须将改变font-size的代码放在css样式之前**。

此处也可结合上述的媒体查询方案，设定不同设备下的基线字体大小：

/\* pc width > 1100px \*/

html { font-size: 100%; }

body {

background-color: yellow;

font-size: 1.5rem;

}

/\* ipad pro \*/

@media screen and (max-width: 1024px) {

body {

background-color: #FF00FF;

font-size: 1.4rem;

}

}

/\* ipad \*/

@media screen and (max-width: 768px) {

body {

background-color: green;

font-size: 1.3rem;

}

}

/\* iphone6 7 8 plus \*/

@media screen and (max-width: 414px) {

body {

background-color: blue;

font-size: 1.25rem;

}

}

/\* iphoneX \*/

@media screen and (max-width: 375px) and (-webkit-device-pixel-ratio: 3) {

body {

background-color: #0FF000;

font-size: 1.125rem;

}

}

/\* iphone6 7 8 \*/

@media screen and (max-width: 375px) and (-webkit-device-pixel-ratio: 2) {

body {

background-color: #0FF000;

font-size: 1rem;

}

}

/\* iphone5 \*/

@media screen and (max-width: 320px) {

body {

background-color: #0FF000;

font-size: 0.75rem;

}

}

1. **视口单位（vw/vh）**

css3中引入了一个新的单位vw/vh，与视图窗口有关，vw表示相对于视图窗口的宽度，vh表示相对于视图窗口高度。由此可见vw/vh很类似百分比单位，需要区别的是 **% 相对于祖先元素或自身**（如border-radius、translate等）而言，而**vw/vh相对于视窗**的尺寸。

视口单位说明表[footnoteRef:8]

[8] 参考：[前端响应式布局原理与方案（详细版）-4.视口单位（2019-04-08）](https://juejin.cn/post/6844903814332432397)

|  |  |
| --- | --- |
| 单位 | 含义 |
| vw | 相对于视窗的宽度，1vw 等于视口宽度的1%，即视窗宽度是100vw |
| vh | 相对于视窗的高度，1vh 等于视口高度的1%，即视窗高度是100vh |
| vmin | vw和vh中的较小值 |
| vmax | vw和vh中的较大值 |

// iPhone 6尺寸作为设计稿基准，使用Sass函数编译

$vm\_base: 375;

@function vw($px) {

@return ($px / 375) \* 100vw;

}

// 无论是文本还是布局宽度、间距等都使用vw作为单位

.mod\_nav {

background-color: #fff;

&\_list {

display: flex;

padding: vm(15) vm(10) vm(10);

&\_item {

flex: 1;

text-align: center;

font-size: vm(10);

&\_logo {

display: block;

margin: 0 auto;

width: vm(40);

height: vm(40);

}

&\_name {

margin-top: vm(2);

}

}

}

}

也可结合REM单位使用：

// rem 单位换算：定为 75px 只是方便运算，750px~75px、640~64px依此类推

$vm\_fontsize: 75; // iPhone 6尺寸的根元素大小基准值

@function rem($px) {

@return ($px / $vm\_fontsize ) \* 1rem;

}

// 根元素大小使用 vw 单位

$vm\_design: 750;

html {

font-size: ($vm\_fontsize / ($vm\_design / 2)) \* 100vw;

// 同时，通过Media Queries 限制根元素最大最小值

@media screen and (max-width: 320px) {

font-size: 64px;

}

@media screen and (min-width: 540px) {

font-size: 108px;

}

}

// body 也增加最大最小宽度限制，避免默认100%宽度的 block 元素跟随 body 而过大过小

body {

max-width: 540px;

min-width: 320px;

}

1. **图片响应式方案**

这里的图片响应式包括两个方面，一个是自适应大小，这样能够保证图片在不同的屏幕分辨率下出现压缩、拉伸的情况；一个是根据不同的屏幕分辨率和设备像素比来尽可能选择高分辨率的图片。

* **使用max-width**

使图片能随着容器的大小进行缩放

img {

display: inline-block;

max-width: 100%;

height: auto;

}

* **使用 background-image**

.banner{

background-image: url(/static/large.jpg);

}

@media screen and (max-width: 767px){

background-image: url(/static/small.jpg);

}

* **使用 picture 标签**

<picture>

<source srcset="banner\_w1000.jpg" media="(min-width: 801px)">

<source srcset="banner\_w800.jpg" media="(max-width: 800px)">

<img src="banner\_w800.jpg" alt="">

</picture>

<!-- picturefill.min.js 解决IE等浏览器不支持 <picture> 的问题 -->

<script type="text/javascript" src="js/vendor/picturefill.min.js"></script>

需要注意的是，picture必须要写img标签，否则无法显示，对picture的操作最后都是在img上面，例如onload事件是在img标签上触发的，picture和source不会进行layout，它们的宽和高都是0。

另外使用source，还可以对图片格式做一些兼容处理：

<picture>

<source type="image/webp" srcset="banner.webp">

<img src="banner.jpg" alt="">

</picture>

### **测试工具**

这里推荐的工具主要用于测试网站在页面展示上的响应能力

[**Mobile-Friendly**](https://search.google.com/test/mobile-friendly)：谷歌工具（需要梯子），可以分析网站的响应速度。输入页面URL就能了解页面在**移动平台**上是否兼容。

[**Am I Responsive**](http://ami.responsivedesign.is/)：此工具可获取网站在常见设备上的显示方式。只需在平台上输入网址，就能看到网站在笔记本电脑、平板电脑、手机和台式机上的显示效果，并且提供了**自定义屏幕分辨率**选项。

[**Responsive Design Checker**](https://www.responsivedesignchecker.com/)：与 Am I Responsive 工具类似，该工具还可测试页面在 Google Pixel 手机以及 Amazon Kindle 上的显示效果。

## **跨平台兼容性**

随着移动和平板市场的日益发展，Web在桌面、平板、移动平台上的兼容性问题日益突出。由于移动和平板是触摸式操作，与桌面的鼠标操作方式有很大差异，因此在不同平台上要做相应修改。为了解决这个问题，诞生了**跨平台框架**（如React Native、Flutter、Weex等），在不同平台上，外观、布局、操作都有差异化修改。

大型网站，手机网站与桌面网站是不同的入口，因此不存在兼容问题，是两个单独的应用程序。对于流量较小的网站，平台的兼容策略主要是应用响应式框架，加上移动端主菜单与导航栏即可，其次可以选用跨平台框架来实现在不同平台的差异化体验。没有这些框架对于Web网站来说不造成大的体验下降。而如果需要开发混合移动、桌面应用，则需要认真考虑这些框架，毕竟用户对本地应用的体验期待要高很多。

# **测试指南**

从现状统计，有很大部分前端开发从来没有做过前端测试工作，更没有写过前端单元测试用例，该不该写前端测试，还是得视项目情况而定，一般标准的项目/平台都要做单元测试。并且有好的测试能够很好地保证项目/平台的健壮，更能够找出代码的逻辑正确性，环境的兼容性等等。

**什么是前端测试？[footnoteRef:9]**

[9]引用美团前端测试方案 <https://juejin.cn/post/6844903983098626056>

* 单元测试：是指对软件中的最小可测试单元进行检查和验证，通常指的是独立测试单个函数。
* UI 测试：是对图形交互界面的测试。
* 集成测试：测试应用中不同模块如何集成，如何一起工作，这和它的名字一致。
* 端到端测试（e2e）：站在用户角度的测试，把程序看成是一个黑盒子，不关注程序内部是怎么实现的，只负责打开浏览器，把测试内容在页面上输入一遍，看是不是得到想要的结果。

前端测试的框架可谓是百花齐放。

* 单元测试（Unit Test）有 Mocha, Ava, Karma, Jest, Jasmine 等。
* 集成测试（Integration Test）和 UI 测试（UI Test）有 ReactTestUtils, Test Render, Enzyme, React-Testing-Library, Vue-Test-Utils 等。

## **单元测试**

主流测试框架对比

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 框架 | 断言 | 仿真 | 快照 | 异步测试 |
| Mocha | 默认不支持，可配置 | 默认不支持，可配置 | 默认不支持，可配置 | 友好 |
| Ava | 默认支持 | 不支持，需第三方配置 | 默认支持 | 友好 |
| Jasmine | 默认支持 | 默认支持 | 默认支持 | 不友好 |
| Jest | 默认支持 | 默认支持 | 默认支持 | 友好 |
| Karma | 不支持，需第三方配置 | 不支持，需第三方配置 | 不支持，需第三方配置 | 不支持，需第三方配置 |

### **Mocha**

Mocha 是生态最好，使用最广泛的单测框架，但是他需要较多的配置来实现它的高扩展性。

**特点：**

* 灵活(不包括断言和仿真，自己选对应工具)
* 流行断言的选择：chai，sinon
* 社区成熟用的人多，测试各种东西社区都有示例
* 需要较多配置
* 可以使用快照测试，但依然需要额外配置

首先选择Chai断言结合使用，Chai语法简单，使用方便，大部分都优先选择它。

进一步与我们的主框架Vue结合使用，结合Vue提供的 Vue-Test-Utils 库一起编写Vue的测试用例简单方便。

注：我们使用脚手架会包含了测试框架进去，一般首先都是mocha + chai做单元测试。

### **Karma**

Karma是由Google团队开发的一套前端测试运行框架。它不同于测试框架（Mocha，Jasmine等），运行在这些测试框架之上。Karma是一直作为一个Test Runner而存在的。该工具可用于测试所有主流Web浏览器，也可以集成到 CI工具，也可以和其他代码编辑器一起使用。

Karma 可以在不同的桌面或移动设备浏览器上，或在持续集成的服务器上测试 JavaScript 代码。Karma 支持 Chrome、ChromeCanary、 Safari、Firefox、 IE、Opera、 PhantomJS，知道如何捕获浏览器正使用的默认路径，这些路径可能在启动器配置文件被忽视（overridden）。

Karma一般会配合 Mocha 或 Jasmine 等一起使用。

推荐在一些独立 库/API 中使用较为方便。

### **Jest**

Jest 是 Facebook 开源的一款 JS 单元测试框架，它也是 React 目前使用的单元测试框架。目前除了 Facebook 外，Twitter、Nytimes、Airbnb 也在使用 Jest。Jest 除了基本的断言和 Mock 功能外，还有快照测试、实时监控模式、覆盖度报告等实用功能。 同时 Jest 几乎不需要做任何配置便可使用。

Jest 基于 Jasmine, 做了大量修改并添加了很多特性，同样开箱即用，异步测试支持良好，所以jasmine就用的少被jest所替代。

Vue和React脚手架都包含了该单元测试框架，React上选择它会较多些，Vue的选择还是Mocha + Chai比较多。

## **UI测试**

不少数人都将 UI 测试（UI Test）和端到端测试（E2E Test）混为一谈，认为是同一个测试类型。

事实上，UI 测试（UI Test）和端到端测试（E2E Test）是稍有区别的：

* UI 测试（UI Test）只是对于前端的测试，是脱离真实后端环境的，仅仅只是将前端放在真实环境中运行，而后端和数据都应该使用 Mock 的。
* 端到端测试（E2E Test）则是将整个应用放到真实的环境中运行，包括数据在内也是需要使用真实的。

就前端而言，UI 测试（UI Test）更贴近于我们的开发流程。在前后端分离的开发模式中，前端开发通常会使用到 Mock 的服务器和数据。因而我们需要在开发基本完成后进行相应的 UI 测试（UI Test）。

### **Storybook框架**

Storybook是用于UI开发的工具。它通过隔离组件使开发更快、更容易。这允许您一次只处理一个组件。您可以开发整个ui，而不需要启动复杂的开发堆栈、强制将某些数据放入数据库或在导航围绕着应用程序。

Storybook支持React/Vue/Angular框架，是UI测试中选择最丰富的一个框架。

*(待应用补充)*

## **集成测试**

把单元测试、UI测试集成部署在真实(浏览器)环境中测试，更能够体现应用程序在浏览器的测试结果。

## **端到端测试(e2e)**

框架对比

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 框架 | 是否跨浏览器支持 | 实现 | 优点 | 缺点 |
| Nightwatch | 是 | Selenium | 可以和其他框架一起使用。适用局部的功能测试场景 | 不支持 TypeScript, 社区文化稍弱于其他几个框架 |
| Cypress | 是 | Chrome | 容易调试和日志记录，使用 Mocha 作为它的测试结构，若单测使用的是 Mocha，所有测试使用的是相同的结构，看起来更“标准” | 缺乏高级功能 |
| Testcafe | 是 | Testcafe | 支持TS，可并行测试，开箱即用 | 不支持录屏，DOM 快照等高级功能 |
| Puppeteer | 否 | Chrome | 速度快，易调试 | Headless Chrome 不支持安装扩展 |

Puppeteer 是 Google Chrome 团队推出的库，尽管它相对其他 e2e 框架较新，但它同样也有一个庞大的社区。它拥有更简洁易用的 API，更快的运行速度，已逐渐成为业内自动化测试的标杆，俘获大量 Selenium 用户的心。

推荐最终测试组合

Vue: Mocha + Chai + Vue-Test-Units + Puppeteer

Cypress也是备选之一，因为我们框架Vue中自带。

测试框架的统一，将在脚手架中进行统一。

## **Bug处理规则**

* 冒烟测试：在对测试正式发版之前会要求对代码进行自测，及冒烟测试。
* 正式测试阶段：正式测试阶段测试人员会在禅道进行bug提交和管理，对BUG的处理规则如下：
* [确认]: 已大致定位原因, 需要较多时间处理的BUG, 可置为"确认"
* [提需求]: 无规律或只出现一次的BUG, 研发没找到原因, 加上必要排查日志后, 可提需求给测试; 复现后重新打开
* [不予解决]: 没有修改程序代码, 是由于其他原因(需求变更等)而解决的问题;
* [解决]：修复bug并提交请求合并到git dev分支下后，修改禅道上的bug状态，并且填写解决方案。

# **性能指南**

本章节仅从前端开发方面进行考虑，从页面资源加载优化、代码优化、webpack打包优化、性能调试工具的使用介绍等方面进行介绍。

## **页面资源加载优化**

### **资源预加载[footnoteRef:10]**

[10] 参考资料：

<https://juejin.cn/post/6844904195707895816>

<https://www.cnblogs.com/xiaohuochai/p/9178390.html>

推荐资源加载相关设置：

1. css放在head中，因其加载不阻塞页面进程
2. js脚本放到body的底端加载
3. 动态加载js模块

在html里的请求资源，可以使用相应的关键字优化资源加载的优先级。

简要列明常用的属性：

|  |  |  |
| --- | --- | --- |
| 关键字 | 作用 | 适用场景 |
| preload | 初始加载页面的时候，资源提前加载。 | 页面运行必须使用的脚本、样式、静态文件,如字体文件等。 |
| 示例 | <link rel="preload" href="next.css">  <link rel="preload" href="next.js"> | |
| prefetch | 浏览器空闲的时候自动下载该资源。 | 路由切换后需要用到的脚本和样式。 |
| 示例 | <link rel="prefetch" href="next.css">  <link rel="prefetch" href="next.js"> | |
| defer | 不阻止document的解析，异步下载资源。脚本执行顺序在DOMContentLoaded前。 | 异步执行的脚本 |
| 示例 | <script src="d.js" defer></script> | |
| async | 不阻止document的解析，异步下载资源。脚本下载完立即执行。 |  |
| 示例 | <script src="b.js" async></script> | |

\* vue-cli默认开启prefetch,可以在vue.config.js中优化配置。

\* 兼容性：IE11+/主流浏览器

### **HTTP请求优化**

* 与后台合理商讨接口划分。避免一个页面发起太多请求。
* 合理设置缓存过期时间，与后台协商合理的缓存机制，配合localStorage适当缓存资源。
* 资源合并压缩：

图标：iconfont字体图标、雪碧图

js脚本：分模块、分页面打包压缩首屏加载时间

图片压缩：webp

* 懒加载：图片懒加载、模块懒加载、路由懒加载

## **代码优化**

### **CSS减少重绘回流**

* 避免使用层级较深的选择器
* 避免使用CSS表达式，（计算频繁，页面滚动缩放都会触发）
* 元素适当地定义高度或最小高度，避免布局发生晃动
* 实现动画效果、拖拽效果时，可将元素position设置为absolute，脱离文档流
* 不要使用table布局，因为一个小改动可能会造成整个table重新布局。而且table渲染通常要3倍于同等元素时间
* 能够使用CSS实现的效果，尽量使用CSS而不使用JS实现。避免频繁访问元素的scrollTop、scrollHeight等属性。

### **函数防抖节流**

在遇到频繁触发的事件时，比如监听鼠标移动事件、鼠标拖拽事件、窗口resize事件、input框输入事件时，为了性能的考虑，我们可以把回调函数的执行控制在一个时间阈值里，优化响应速度。具体的响应逻辑可以自己优化，也可以直接使用lodash中提供的方法：debounce、throttle。

### **常见的代码优化**

* 1. 频繁的元素插入、删除操作，使用链表结构组织数据。
  2. 使用mvvm框架（vue）时，避免把嵌套层级深的复杂结构如地图实例等对象放到data里，以免频繁触发框架的响应监听。
  3. 使用闭包后，谨记释放内存。
  4. 耗时的数据处理工作，可以新开一个Web Worker的线程进行处理。
  5. 图形库的内存释放

应用中如果把结构复杂的对象，如地图库、图形库等挂载到组件实例（window全局对象同理）中，应该在组件卸载时及时释放内存。常见内存溢出场景：地图分屏功能、含地图实例的组件在路由切换时未释放内存。工程中使用如ArcGIS for js/ThreeJS/Cesium等webgl框架，除了调用框架本身提供的销毁方法，也要获取canvas中webgl的上下文，清除gpu内存。以Threejs内存释放为例：依次销毁几何、材质、贴图，清空挂载在this上的对象，清空webgl上下文。

  disposeScene() {

      this.anmiateId && cancelAnimationFrame(this.anmiateId);

//调用Threejs提供的dispose方法

      this.$scene.traverse((*son*) => {

        if (son instanceof Mesh) {

          for (let value of Object.values(son.material)) {

            if (value instanceof THREE.Texture) {

              value.dispose();

              value = {};

            }

          }

          son.material.dispose();

          son.geometry.dispose();

          son.material = {};

          son.geometry = {};

        }

      });

      this.$controls.dispose();

      this.$target.dispose();

      this.$dracoLoader.dispose();

//释放挂载到this实例的对象

     this.$model = null;

      this.$camera = null;

      this.$controls = null;

      this.$target = null;

      this.$dracoLoader = null;

      this.$composer = null;

      this.$scene = null;

//清空webgl上下文

      let domElement = (this.$renderer as THREE.WebGLRenderer).domElement;

      const gl = domElement.getContext('webgl2');

      if (!gl) return;

      const extension = gl.getExtension('WEBGL\_lose\_context');

      if (!extension) return;

      extension.loseContext();

      domElement = null;

      (this.$renderer as THREE.WebGLRenderer).dispose();

      this.$renderer = null;

    },

## **Webpack打包优化**

* JS加载会阻塞浏览器进程，通过webpack打包优化，把常用的第三方库单独打包，避免单个js文件体积太大。以vue.config.js配置为例：

module.exports = {

 chainWebpack(*config*) {

 config.optimization.splitChunks({

        chunks: 'all', // "initial"，"async"和"all"。

        cacheGroups: {

          libs: {

            name: 'chunk-libs',

            test: /[\\/]node\_modules[\\/]/,

            priority: 1, // 数字越大优先级越高

            chunks: 'initial',

          },

          threejs: {

            name: 'three',

            priority: 2,

            test: /[\\/]node\_modules[\\/]\_?three(.\*)/,

          },

          element: {

            name: 'element',

            priority: 2,

            test: /[\\/]node\_modules[\\/]\_?element-ui(.\*)/,

          },

        },

      });

}

}

* 去除生产环境sourceMap

module.exports = {

productionSourceMap: false,

}

* 外网项目的公共库可以使用cdn引入，以使用 externals 让 webpack 不打包某部分，然后在其他地方引入 cdn 上的 js 文件，利用缓存下载 cdn 文件达到减少打包时间的目的。
* 使用compression-webpack-plugin压缩体积较大的js/css/json（需要后端配合启用gzip）
* 使用image-webpack-loader压缩体积较大的图片

## **性能调试工具的使用（performance/memory）[footnoteRef:11]**

[11] 参考资料：[https://developer.chrome.com/docs/devtools/](https://developer.chrome.com/docs/devtools/evaluate-performance/)

配合浏览器提供的性能调试工具，可以快速排查问题。本章节以chrome提供的performance和memory面板为例，简要介绍功能的使用。进一步了解可以查看chrome提供的开发者工具指南。

### **Performance面板**

面板可分为控制窗格、总览窗格、火焰图窗格、内存信息、详细信息五个板块。

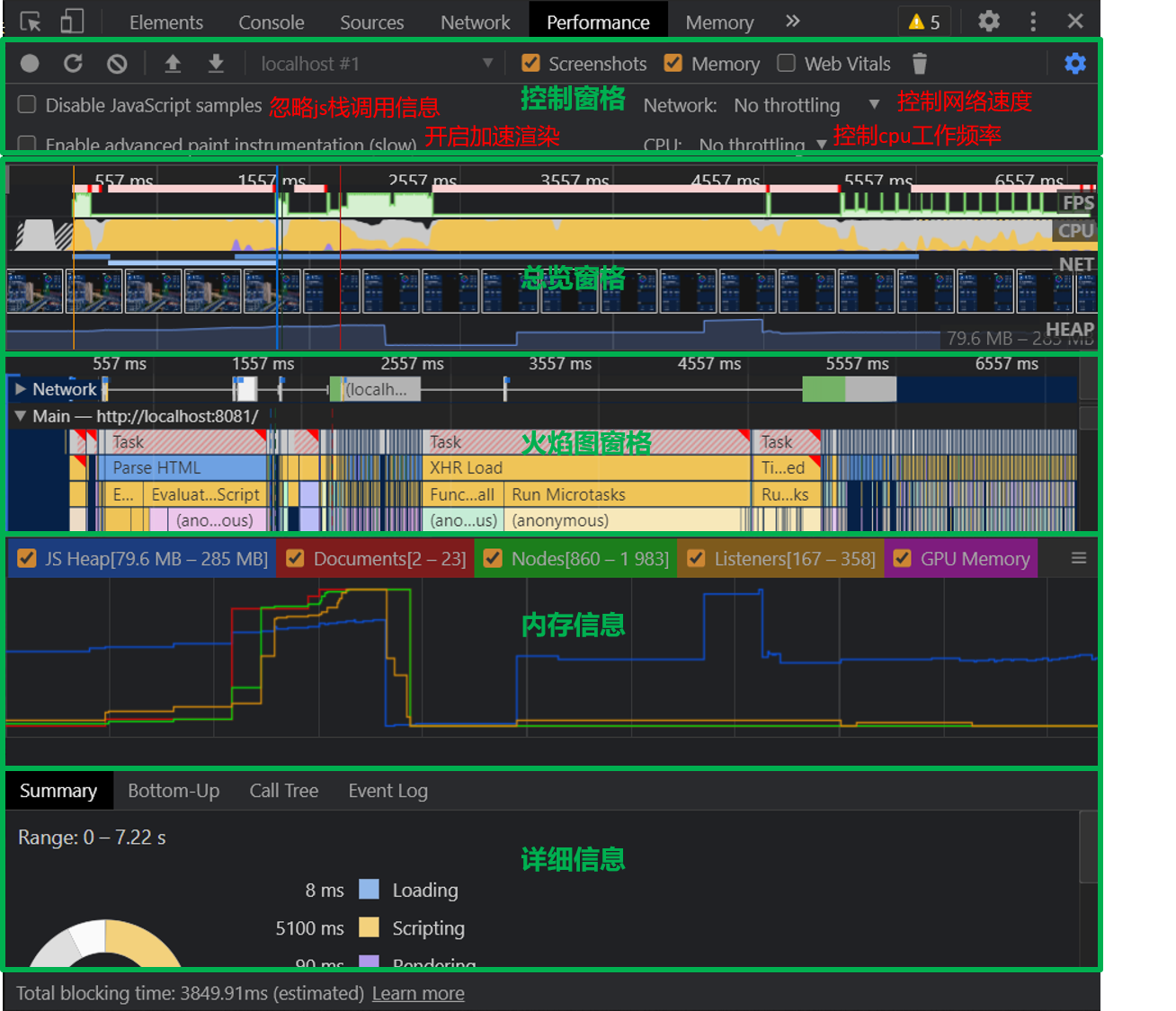
**控制窗格**可以对录制性能快照进行一些参数设置，模拟其他设备的运行情况。

**总览窗格**可以看到FPS/CPU/NET/快照/内存等信息。其中CPU曲线区域图中，黄色、蓝色、紫色、灰色标识分别对应函数执行、HTML操作、CSS渲染、其他操作所占用的内存。

**火焰图窗格**可以看到请求瀑布/主线程函数瀑布/WebWorker线程/GPU进程等各个模块的火焰图。右上角有红色三角形的任务是被浏览器标为LongTask的任务，表示带有性能隐患。

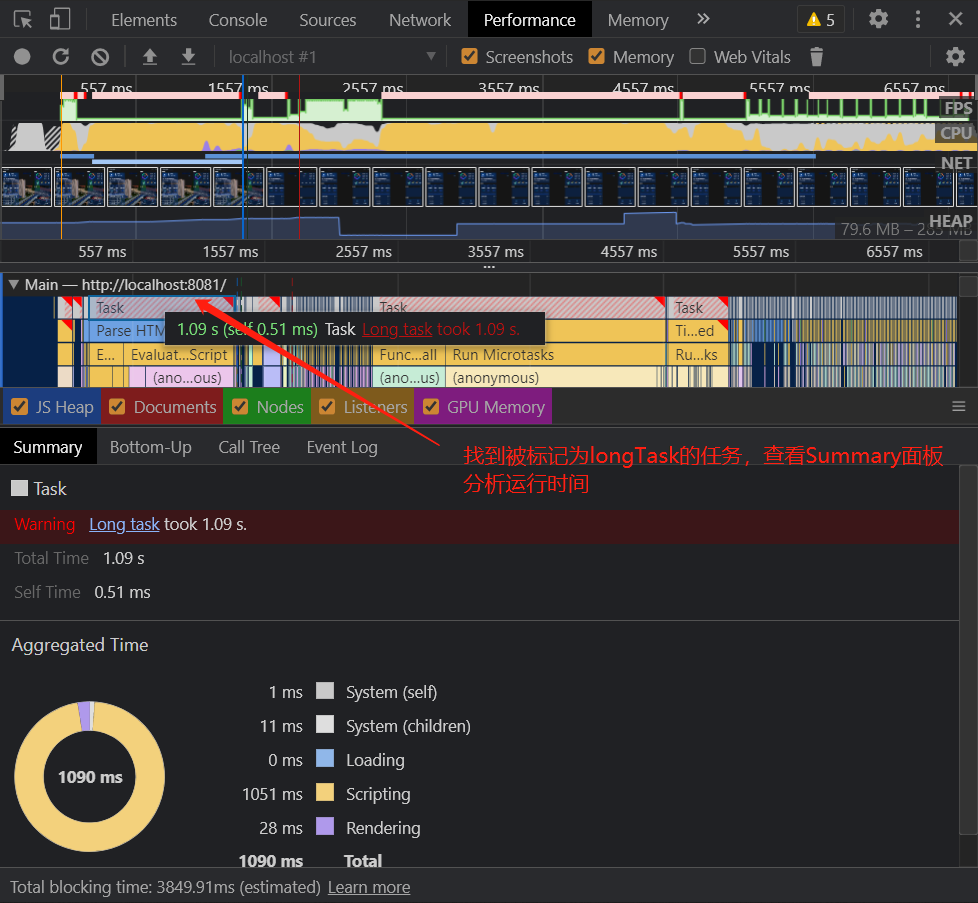
**内存信息**可以看到JS堆/文档数/节点数/事件监听/GPU内存等内存使用情况。 如果JS堆的曲线一直增长，则可能存在内存泄露、事件监听曲线一直增长，则可能存在重复绑定的问题。

**详细信息窗格**可以看到页面的耗时情况,选中具体任务后，可以看到任务的详细信息。

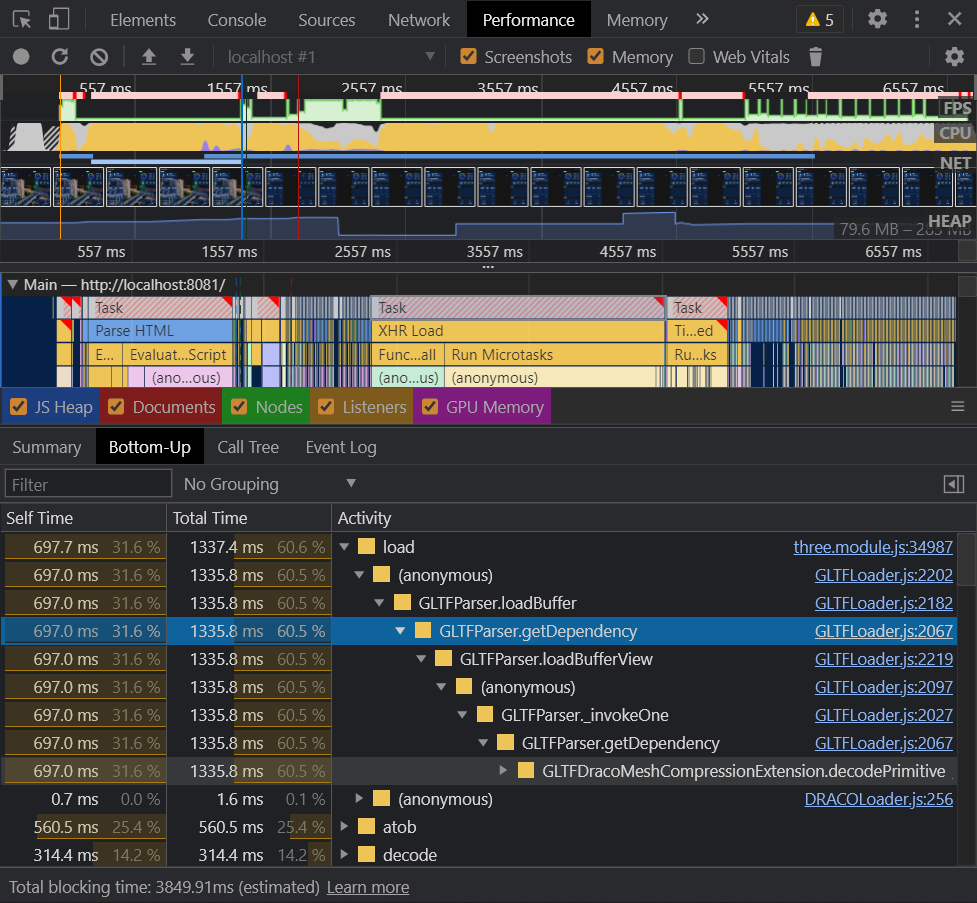


建议使用无痕模式进行记录，排除浏览器插件的影响。

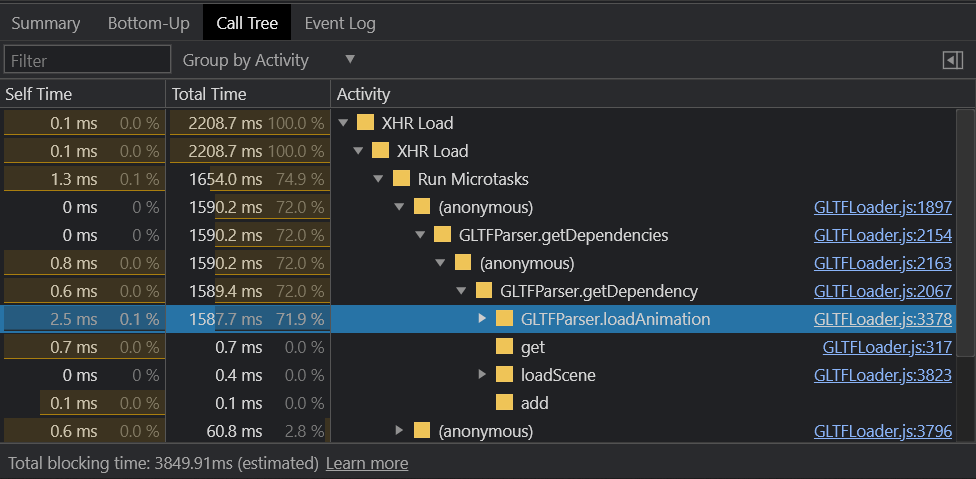
下面以一次简单的性能分析为例，简要说明性能排查的步骤：



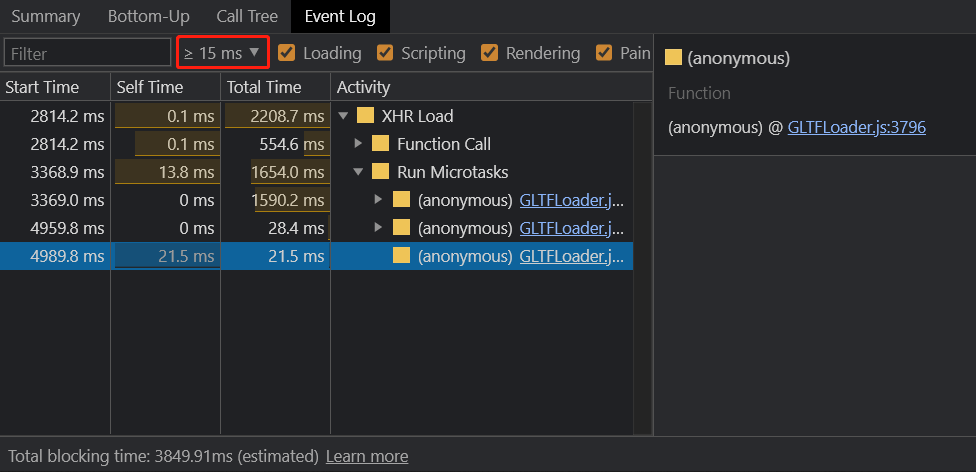
通过Bottom-U面板可以看到选中任务的函数调用情况，依据事件耗时长短排序，从而找出耗时长的函数，Self Time表示对应活动消耗的时间，Total Time表示对应活动以及其子活动共同消耗的时间。从这里可以看出，主要的性能问题是使用dracoLoader进行文件解析。



通过Call-Tree面板可以查看事件调用跟踪信息，比Bottom-Up更详细，追踪粒度更细致。



通过Event-Log面板可以查看任务过程中浏览器发生的活动，可以使用过滤器组合时间、活动类型等条件筛选出想了解的活动。



性能面板除了分析响应速度、找出运行时的性能问题、还可以分析每个时间线上事件的调用信息，对开发过程中的代码调试也很有作用。可以浏览官方文档作进一步了解。

### **Memory面板**

面板可分为内存快照列表、控制窗格、对象列表、详细信息四个板块。

**内存快照列表**展示点击录制按钮后生成的快照列表。

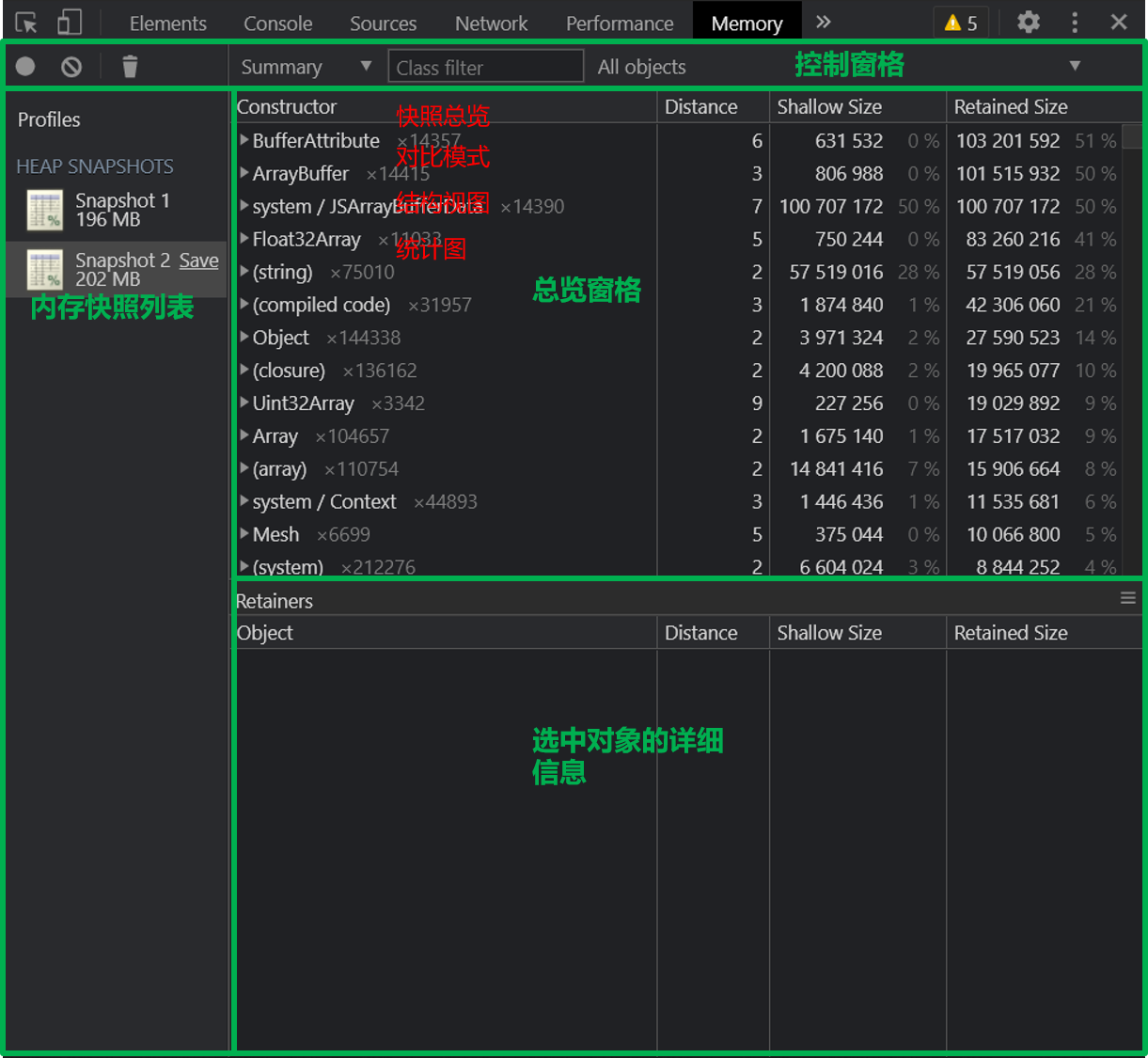
**控制窗格**包含控制录制快照的按钮、切换总览窗格的展示模式、名称过滤器等控件。

**总览窗格**有4种模式：在Summary状态下，展示的是构造函数名称分组的对象列表；在Comparison状态下，展示两个快照的对比信息；在Containment状态下，展示当前快照的对象结构视图；在Statistic上，展示内存使用的统计饼图；

**详细信息窗格**可以看到选中对象的内存占用情况、使用节点到根节点的距离等信息。

面板关键字介绍：

|  |  |
| --- | --- |
| Constructor | 使用该构造函数创建的所有对象 |
| Distance | 节点到GC根节点的最短层级 |
| Shallow Size | 显示通过该构造函数创建的所有对象的内存大小（不包含引用的对象） |
| Retained Size | 显示该组对象中最大的可释放内存（包含引用对象，不包含被GC根引用的对象）。 |
| #New | 对比模式下，该组对象新增项 |
| #Deleted | 对比模式下，该组对象删除项 |
| #Delta | 对比模式下，该组对象增量。#Delta = #New - #Deleted |
| Alloc. Size | 对比模式下，该组对象内存分配大小 |
| Freed Size | 对比模式下，该组对象内存释放大小 |
| Size Delta | 对比模式下，该组对象内存增量 Size Delta = Alloc. Size - Freed Size |



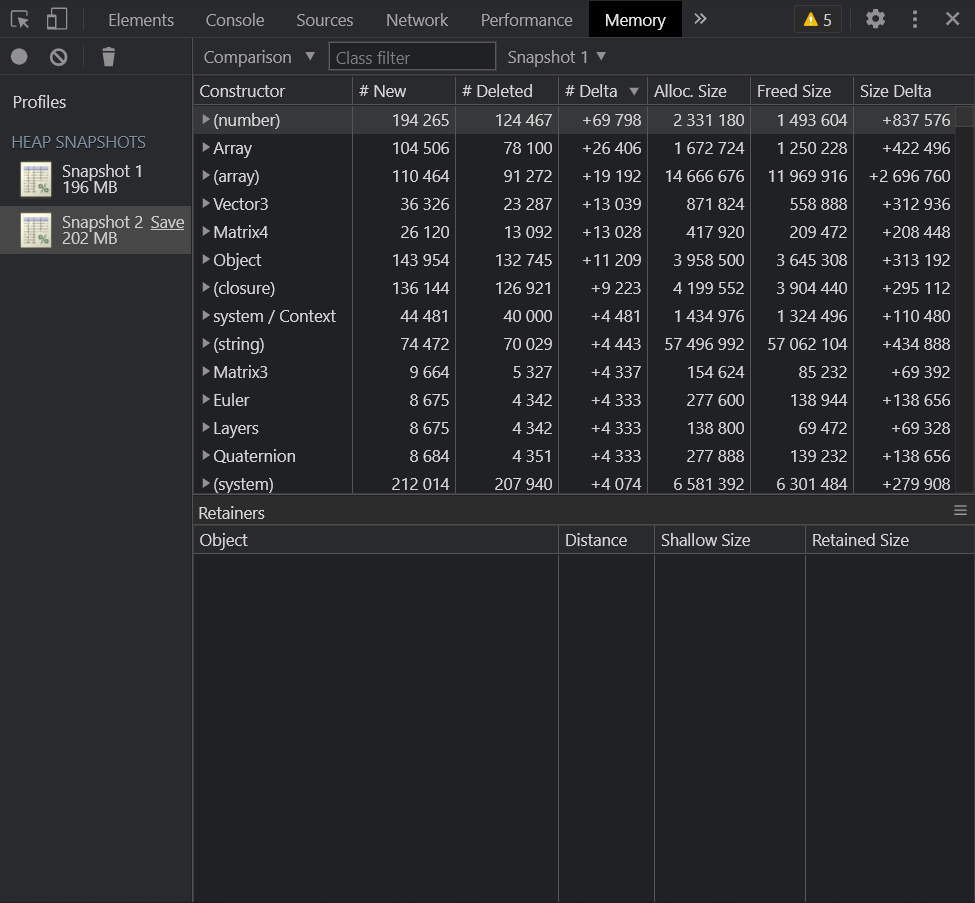
下面以一次简单的内存分析为例，简要说明内存排查的步骤：

目的：排查组件销毁函数是否存在内存释放不完全的问题。

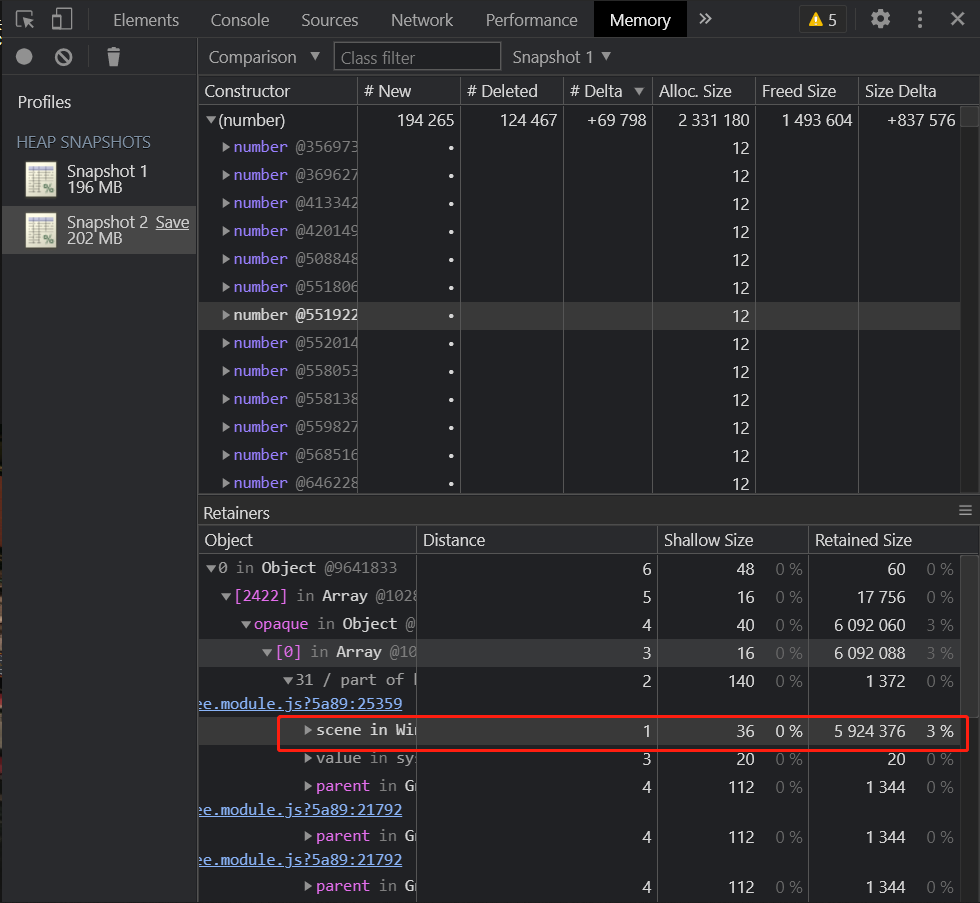
步骤一：页面首次加载完成后，通过Memory面板录制第一次内存快照，得到快照一。

步骤二：触发工程热更新，待页面二次加载完成后，录制第二次内存快照，得到快照二。

步骤三：在快照二的控制窗格选中对比模式，对比快照选择快照一。通过#Delta一列，我们可以看到number、Array等构造函数组的项数呈现增加的状态。



步骤四：点击number组下的某个实例，查看详细信息。可以排查一下Retained Size较大的对象。可以看到scene实例下Retained Size还存在未被释放的对象。



步骤五：找出问题后，调整自己的代码。查阅Three的文档可知，Geometry/Mateiral等实例需要手动调用dispose方法才可以释放内存。

# **安全指南**

浏览器本身具备同源策略，通过限制可能的恶意网站对不同源网站的访问，以确保用户数据的安全性。网站安全则是讲“黑客”如何通过各种方式获取用户隐私数据或者恶搞网站。方式主要包括：伪造网站、直接在目标网站上窃取用户数据(植入代码)或进行流量劫持、破解用户密码等。

在设计Web安全策略时，往往不是独立的，需要前端和后端（服务器、后台接口、甚至是操作系统）联合考虑，虽然大部分防范措施交由后端处理更为有效，但前端代码亦是引诱黑客入侵的大门，随时都有可能传送被篡改的数据给到后端，从而引发漏洞。所以**前端安全的关键是保障数据交互和传递的安全**。

这个世界上没有绝对的安全，即使极其严格的策略都有可能被绕过，前端开发中安全也需要考虑成本，实际项目中，需针对应用的具体使用环境，选用性价比高的安全策略。

*本章仅以前端相关注意事项为主。*

## **常见攻击类型**

前端攻击方式有很多种，如XSS攻击、CSRF攻击、SQL注入、点击劫持、上传文件攻击等，比较典型的是前两种。

### **XSS攻击**

XSS（跨站脚本攻击，Cross Site Script)，Web页面里插入恶意js代码，在已登录的用户页面上执行脚本，利用已信任用户的身份进行攻击甚至盗走用户身份信息。特点是攻击者的代码必须能获取用户浏览器端的执行权限。

XSS分为反射型、存储型、DOM型三种攻击类型。

反射型攻击主要是攻击者将恶意代码存放在URL中，当用户访问恶意URL时，网站服务端将恶意代码从 URL 中取出，拼接在 HTML 中返回给浏览器，浏览器渲染时执行恶意代码。该漏洞容易出现在网站搜索与页面跳转。

存储型攻击是指攻击者将恶意代码提交到目标网站的数据库中，属于最直接的危害类型，容易造成蠕虫，大量窃取cookie。常见于论坛发帖、发表文章或评论、用户私信等地方。

DOM型攻击是基于文档对象模型的一种漏洞，指受害者端的网页脚本在修改本地页面DOM环境时未进行合理的处置，而使得攻击脚本被执行。因为DOM中有很多对象，其中一些是用户可以操纵的，如URL、location、referer等。

反射型和存储型在服务器端可以通过过滤输出处理，对前端项目来说主要是针对**DOM型攻击**采取安全措施。

### **CSRF攻击**

CSRF（跨站点伪造请求，Cross Site Request Forgery)，在用户登陆目标网站后，诱导用户访问第三方网站，在第三方网站中向目标网站发起伪造请求，利用用户已登陆的权限，绕过后台用户验证。与XSS相比，XSS利用的是用户对指定网站的信任，而CSRF利用的是网站对用户网页浏览器的信任。

CSRF通常是跨域的，因为外域通常更容易被攻击者掌控。但是如果本域下有容易被利用的功能，比如可以发图和链接的论坛和评论区，攻击可以直接在本域下进行，而且这种攻击更加危险。被攻击的网站无法防止攻击发生，只能通过增强自己网站针对CSRF的防护能力来提升安全性。

## **推荐安全措施**

单纯的前端安全措施比较少见，毕竟前端防范能力有限。此处只简单介绍一些前端能够做到的措施，在实际生活中，还需要在后台系统进行进一步的过滤。

### **启用CSP**

有两种方法可启用CSP（内容安全策略），一种是可通过html中的<meta>标签，另一种是通过HTTP头部配置Content-Security-Policy字段（Nginx或Apache）

* meta标签启用法（具体用法请参考MDN）：

<meta http-equiv="Content-Security-Policy" content="default-src 'self' \*.trusted.com">

* 服务器配置法：
  1. Apache：在VirtualHost的httpd.conf文件中添加如下代码

Header set Content-Security-Policy "default-src 'self';"

* 1. Nginx：在 server 对象块中添加如下代码

add\_header Content-Security-Policy "default-src 'self';"

除此之外，还有其他HTTP头部配置参数能有效防范XSS攻击：

部分HTTP头部配置用途表[footnoteRef:12]

[12] 参考：[WEB前端安全自查和加固（2020-07-14）](https://insights.thoughtworks.cn/web-frontend-security/)、[http-header安全字段总结（2018-10-25）](https://xz.aliyun.com/t/3013)

|  |  |
| --- | --- |
| 配置名 | 用途 |
| HttpOnly | 对关键**Cookies**设置HttpOnly=true，前端js无法获取该cookies，即使前端被攻击，也无法盗取身份信息 |
| X-Frame-Options | 设置禁止页面在iframe元素中加载，可保护用户免受**点击劫持（ClickJacking）**的攻击 |
| X-XSS-Protection | 启用后可防止**跨站点脚本攻击（XSS）**，适用于现代浏览器使用的XSS过滤器 |
| X-Content-Type-Options | 用于防止MIME嗅探漏洞，禁止浏览器嗅探未设定的资源类型，能在一定程度上防止**文件上传攻击**。 |
| Strict-Transport-Security | 要求浏览器对所有资源使用**HTTPS**链接 |
| Access-Control-Allow-Origin | CORS（跨源资源共享）标头，可设置跨域请求的安全列表，设置妥当能有效防止攻击者**跨站获取数据** |
| Content-Security-Policy | 用于指示浏览器仅加载策略中定义允许的内容。如果正确实施，此策略可防止**XSS、ClickJacking和HTML注入攻击** |

### **避免框架的危险特性**

如Vue的v-html指令、React API的dangerouslySetInnerHTML、Angular API的bypassSecurityTrust、DOM API 的 innerHTML，都是允许用户插入任意html代码的入口，在项目中应尽量避免使用。此外，**切勿基于用户输入去设置 innerHTML 的值，而应该尽可能用 textContent 代替 innerHTML**，以防止完全生成 HTML 输出。

### **避免内联事件**

DOM 中的内联事件监听器，如 location、onclick、onerror、onload、onmouseover 等，能把字符串作为代码运行。如果不可信的数据拼接到字符串中传递给这些 API，很容易产生安全隐患，尽量不要使用。在 JavaScript 中通过 **.addEventlistener()** 事件绑定会更安全。

### **限制输入内容长度**

对于不受信任的输入，都应该限定一个合理的长度。虽然无法完全防止 XSS 发生，但可以增加 XSS 攻击的难度。

### **请求验证Token**

接口请求避免使用GET操作对数据资源进行修改，使用POST时增加一个一次性的token（需后台配合校验）。

### **过滤输入输出**

在将用户输入发送到后端或渲染后端数据之前，应该先对其进行验证或清理。可以通过删除或替换上下文相关的危险字符来对数据进行清理，例如使用**白名单**并对输入数据进行**转义**。清理和编码并不是一件容易的事，所以建议引入成熟的第三方库进行预处理：

* [DOMPurify](https://github.com/cure53/DOMPurify)：一款常用的HTML过滤库，可以处理（来自用户的）不受信任的HTML片段，删除可能导致XSS的所有元素及属性。（在使用时，建议传入RETURN\_DOM或者RETURN\_DOM\_FRAGMENT选项[footnoteRef:13]）

[13] 参考：[通过混淆命名空间绕过DOMPurify实现XSS（2020-10-09）](https://www.anquanke.com/post/id/219089)

* [secure-filters](https://github.com/salesforce/secure-filters)：提供了清理 HTML、JavaScript、内联 CSS 样式和其他上下文的方法。

# **部署指南**

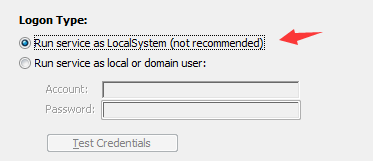
前端有3种部署方式，分别为自动化部署，手动部署和编译源码部署。

## **Jenkins自动化部署**

1. **配置Jenkins**
2. 需要提前安装 JDK，JDK 安装教程网上很多，请自行搜索。
3. 打开 [Jenkins](https://link.zhihu.com/?target=https%3A//www.jenkins.io/zh/download/) 下载页面。



1. 安装过程中遇到 Logon Type 时，选择第一个。



1. 端口默认为 8080，这里我填的是 8000。安装完会自动打开 http://localhost:8000 网站，这时需要等待一会，进行初始化。
2. 按照提示找到对应的文件（直接复制路径在我的电脑中打开），其中有管理员密码。



1. 安装插件，选择第一个。



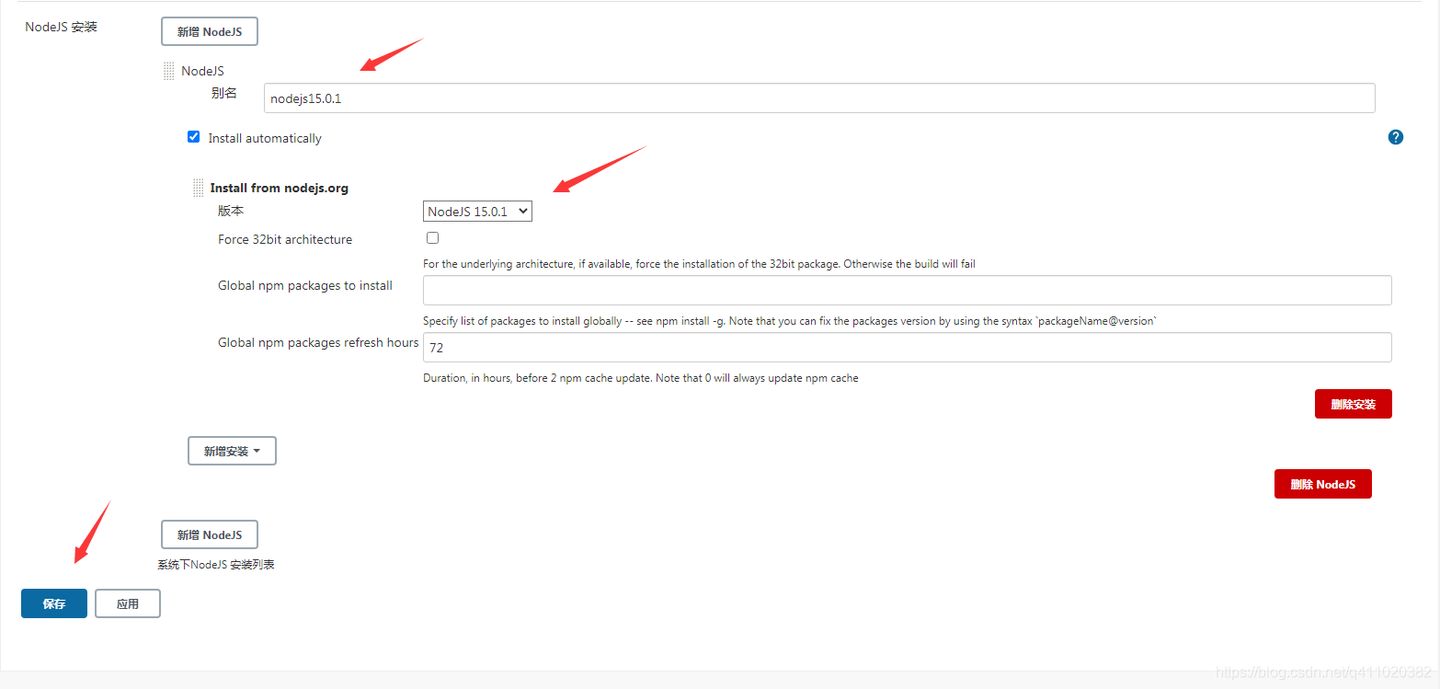
1. 创建管理员用户，点击完成并保存，然后一路下一步。



1. 配置完成后自动进入首页，这时点击 Manage Jenkins -> Manage plugins 安装插件。



1. 点击 可选插件，输入 nodejs，搜索插件，然后安装。
2. 安装完成后回到首页，点击 Manage Jenkins -> Global Tool Configuration 配置 nodejs。如果你的电脑是 win7 的话，nodejs 版本最好不要太高，选择 v12 左右的就行。



1. **创建静态服务器**
2. 建立一个空目录，在里面执行 npm init -y，初始化项目
3. 执行 npm i express 下载express
4. 建立一个server.js文件，代码如下：

const express = require('express')

const app = express()

const port = 8080

app.use(express.static('dist'))

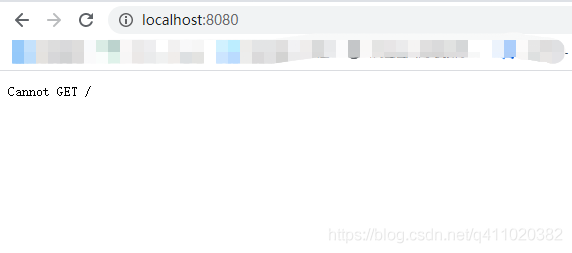
app.listen(port, () => {

console.log(`Example app listening at http://localhost:${port}`)

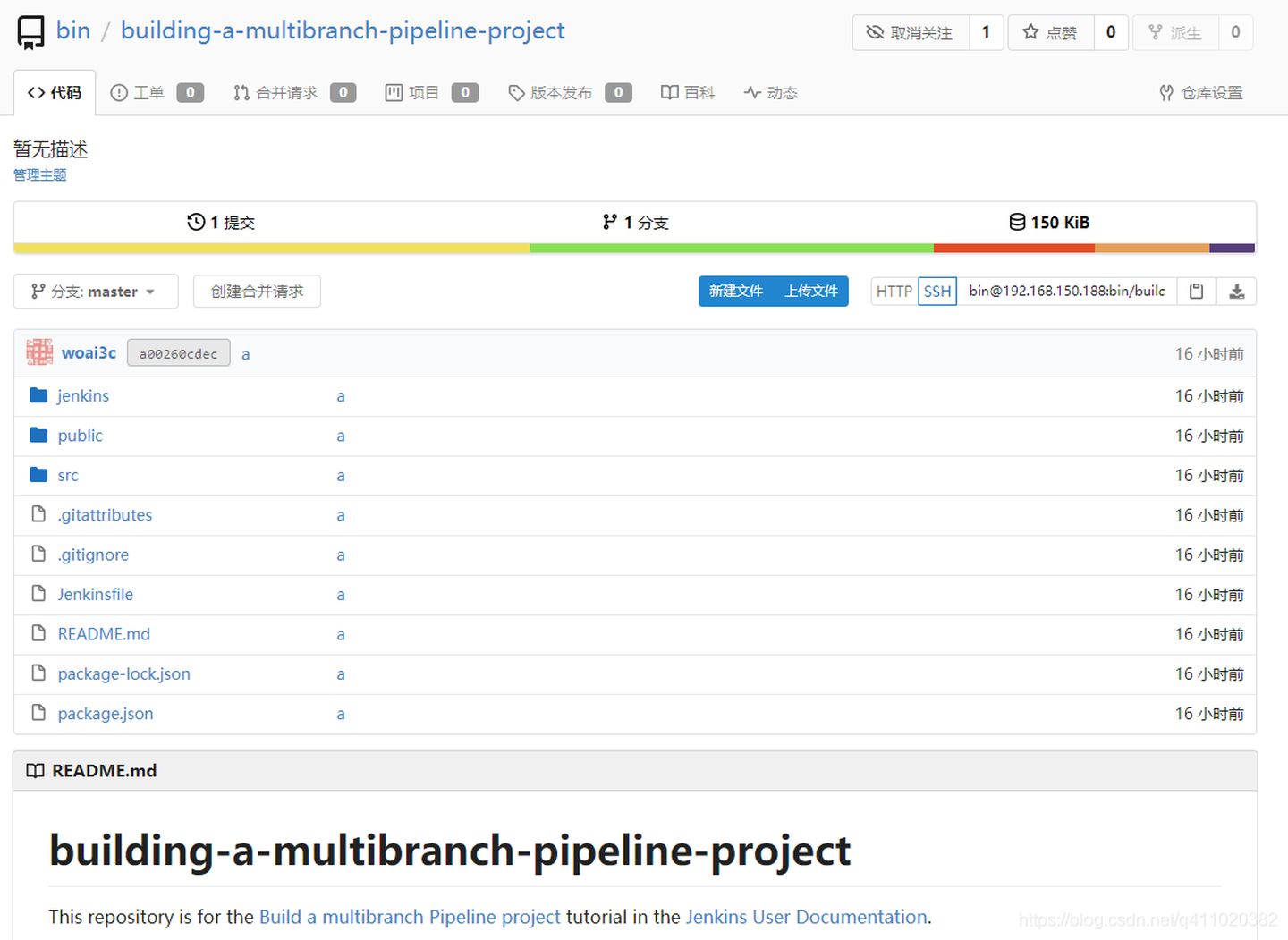
})

它将当前目录下的 dist 文件夹设为静态服务器资源目录，然后执行 node server.js 启动服务器。

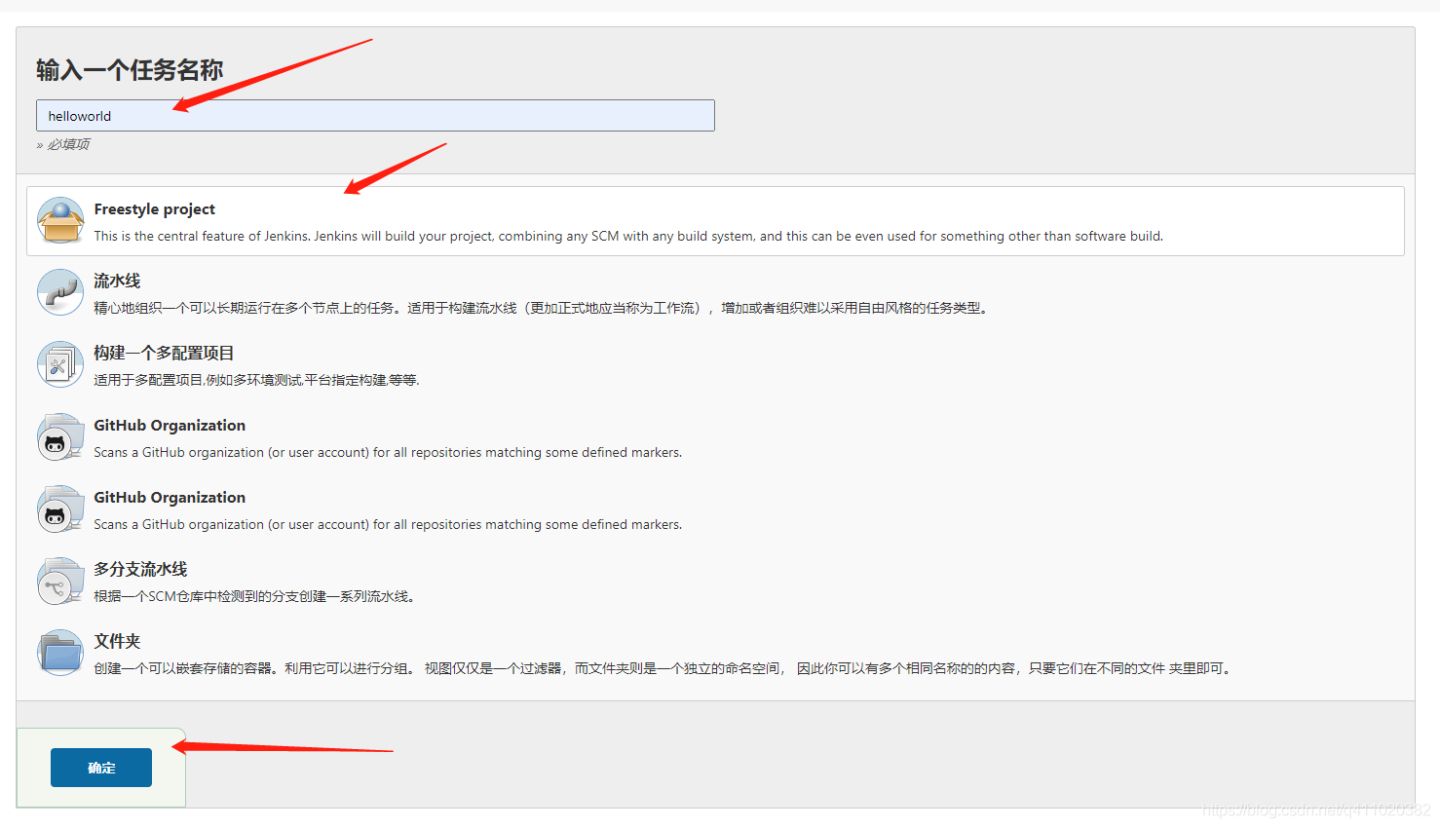
由于现在没有 dist 文件夹，所以访问网站是空页面。（不过不要着急，一会就能看到内容了。）



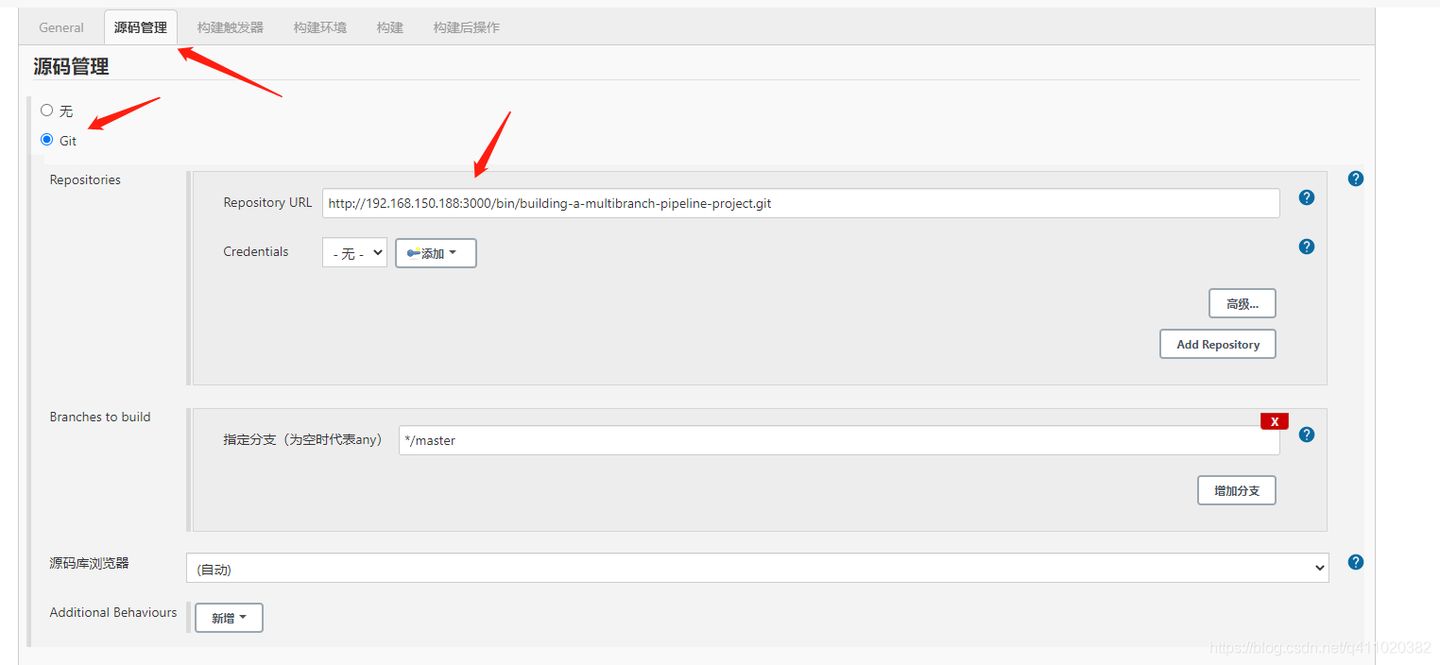
1. **自动构建+部署到服务器**
2. 下载 Jenkins 提供的 demo 项目 [building-a-multibranch-pipeline-project](https://link.zhihu.com/?target=https%3A//github.com/jenkins-docs/building-a-multibranch-pipeline-project)，然后在你的gitlab新建一个仓库，把内容克隆进去，并提交到gitlab服务器。



1. 打开 Jenkins 首页，点击 新建 Item 创建项目。



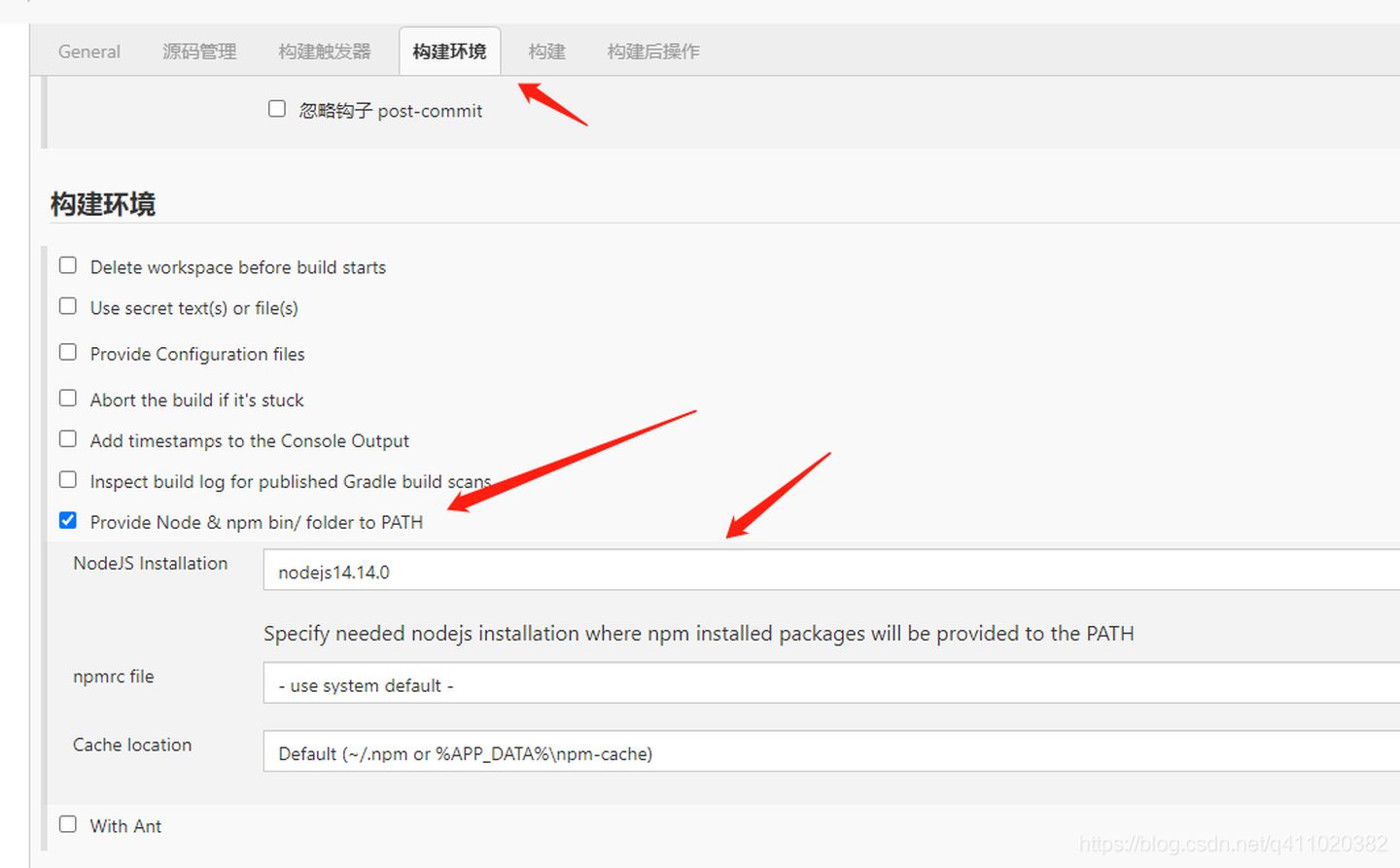
1. 选择源码管理，输入你的 gitlab上的仓库地址。



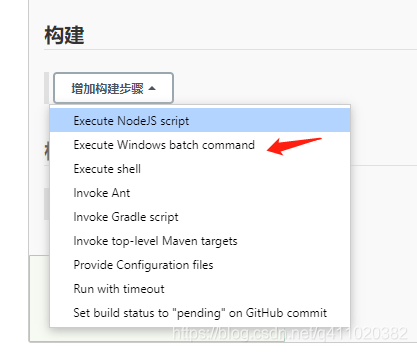
1. 你也可以尝试一下定时构建，下面这个代码表示每 5 分钟构建一次。



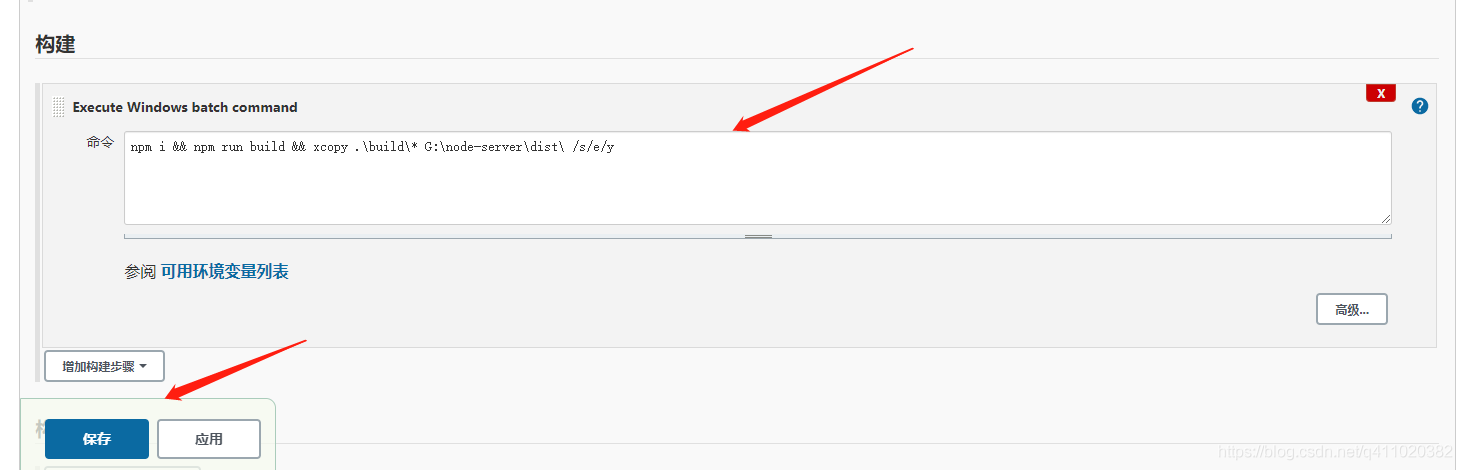
1. 选择你的构建环境，这里选择刚才配置的 nodejs。



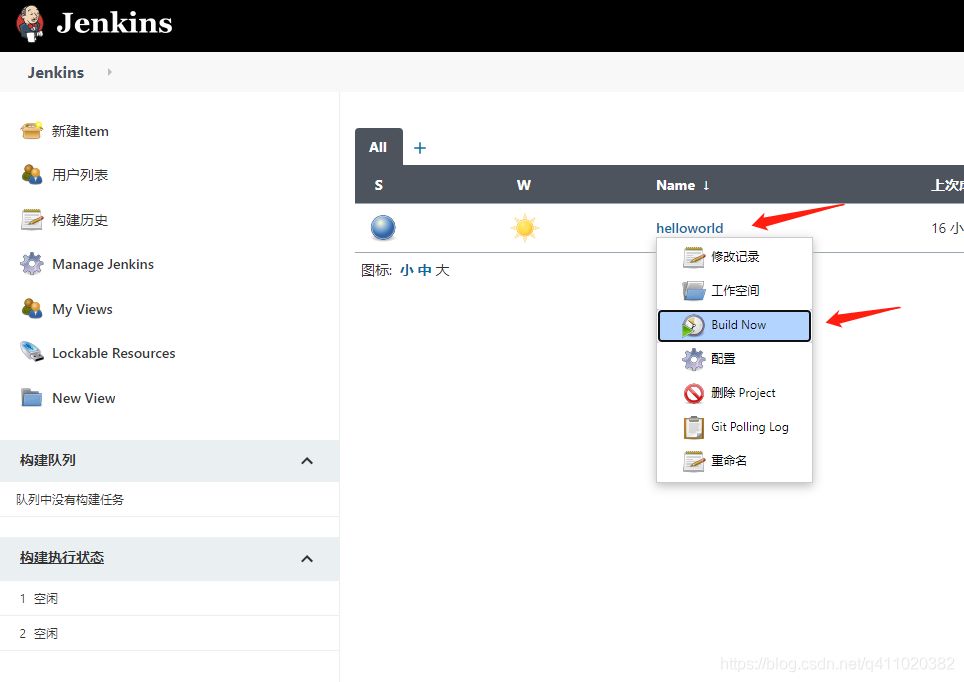
1. 点击增加构建步骤，windows 要选 execute windows batch command，linux 要选 execute shell。



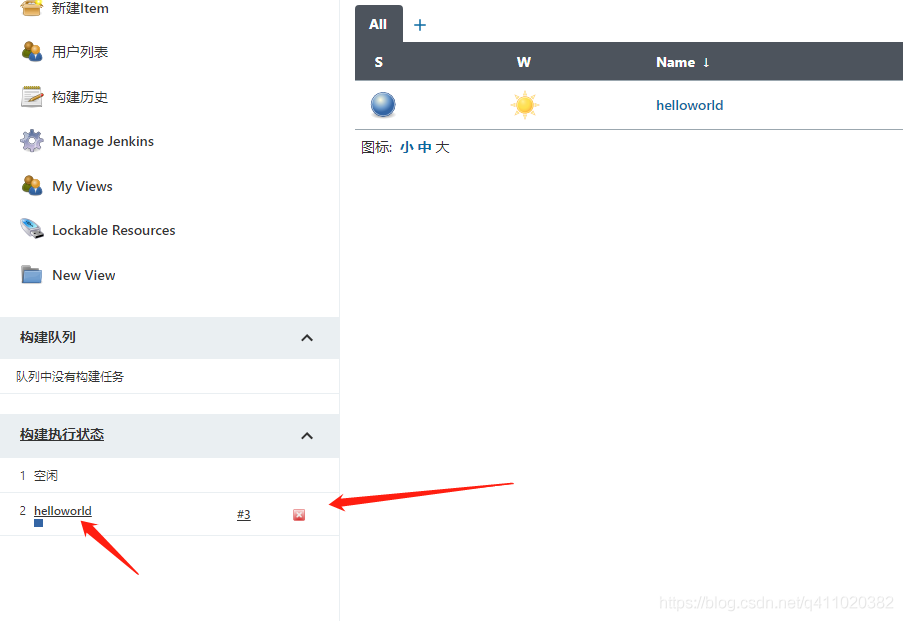
1. 输入 npm i && npm run build && xcopy .\build\\* G:\node-server\dist\ /s/e/y，这行命令的作用是安装依赖，构建项目，并将构建后的静态资源复制到指定目录 G:\node-server\dist\ 。这个目录是静态服务器资源目录。



1. 保存后，返回首页。点击项目旁边的小三角，选择 build now。

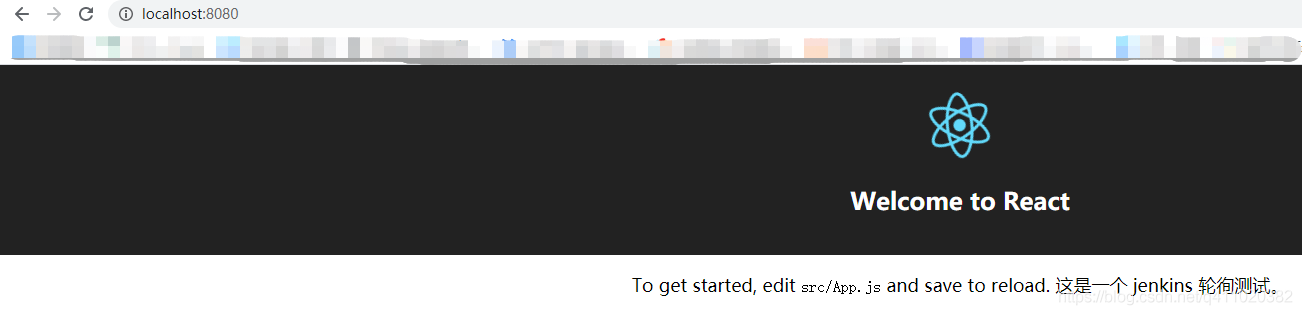


1. 开始构建项目，我们可以点击项目查看构建过程。

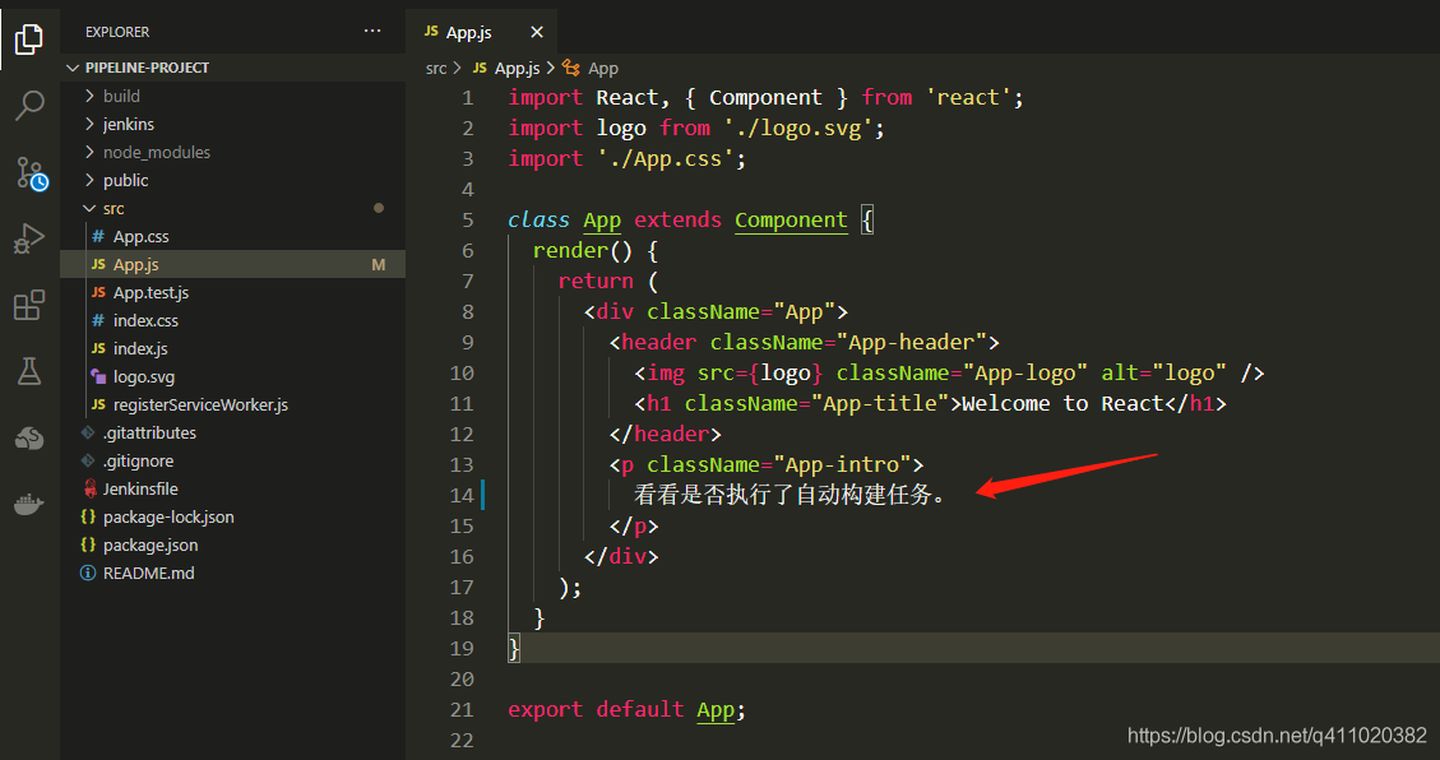


1. 构建成功，打开 http://localhost:8080/ 看一下结果。





1. 由于刚才设置了每 5 分钟构建一次，我们可以改变一下网站的内容，然后什么都不做，等待一会再打开网站看看。



1. 把修改的内容提交到 gitlab服务器，稍等一会。打开网站，发现内容已经发生了变化。



1. **使用pipeline构建项目**

使用流水线构建项目可以结合 gitlab的 webhook 钩子，以便在执行 git push 的时候，自动构建项目。

1. 点击首页右上角的用户名，选择设置。



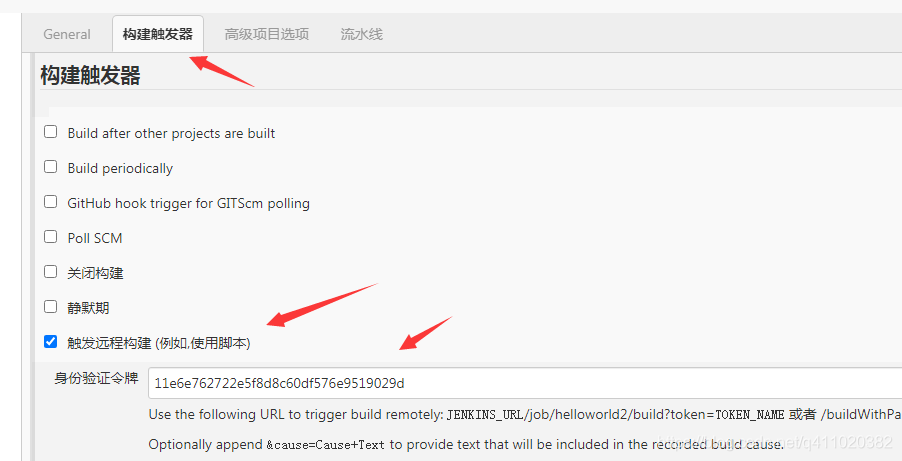
1. 添加 token，记得将 token 保存起来。



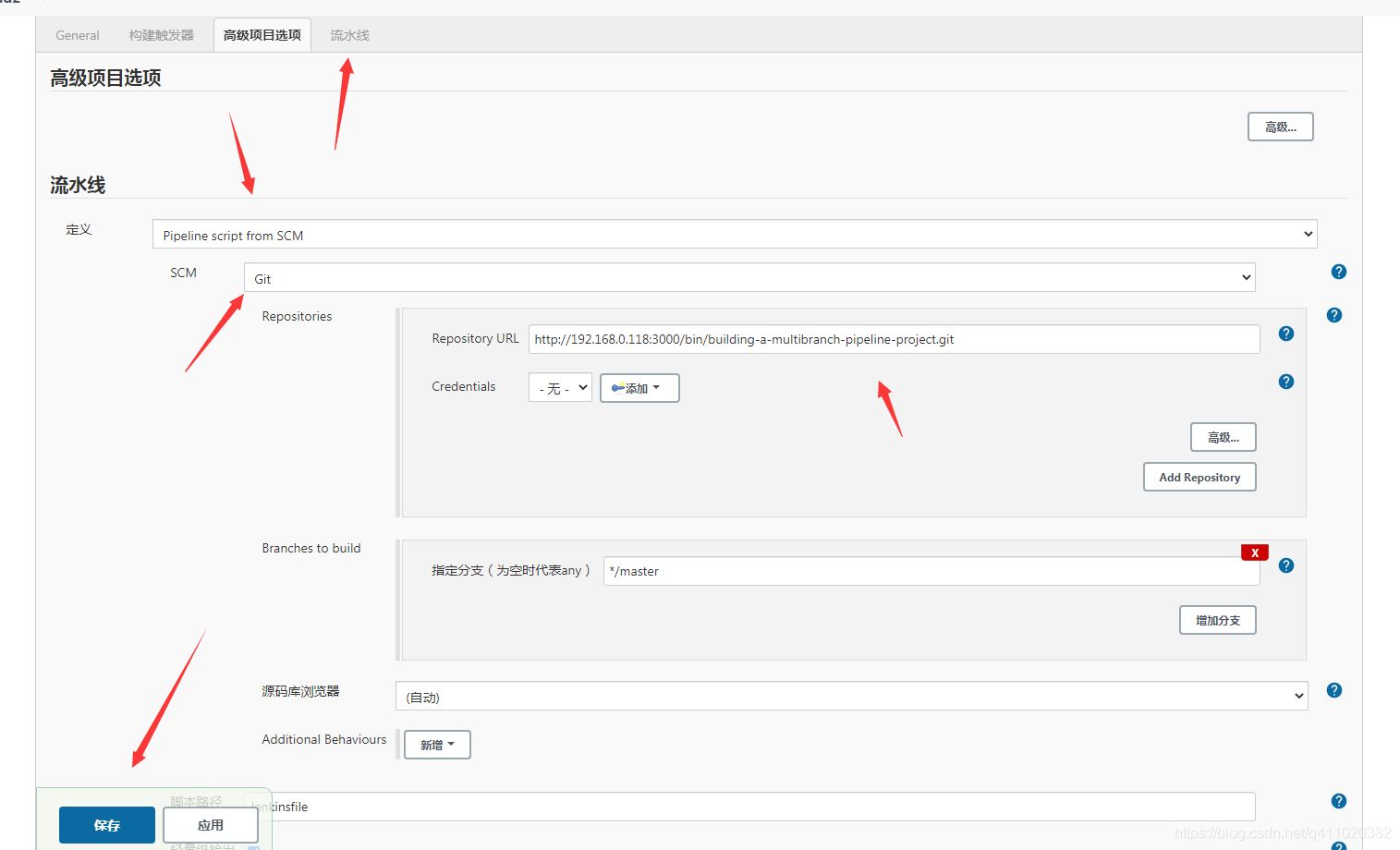
1. 打开 Jenkins 首页，点击 新建Item 创建项目。



1. 点击构建触发器，选择触发远程构建，填入刚才创建的 token。



1. 选择流水线，按照提示输入内容，然后点击保存。

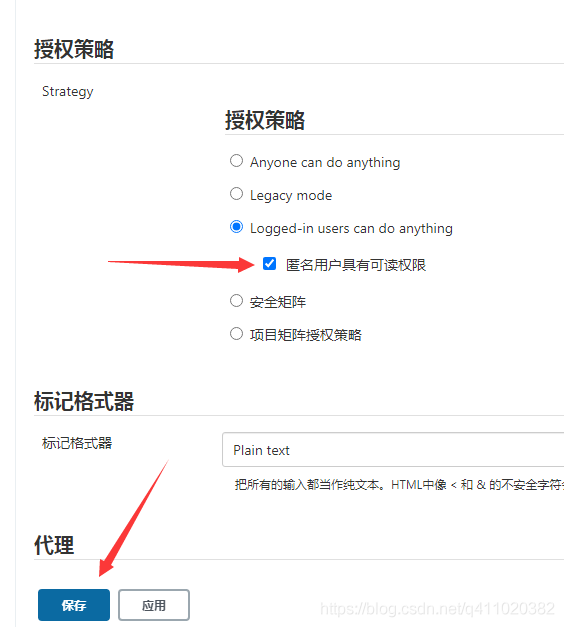


1. 打开 Jenkins 安装目录下的 jenkins.xml 文件，找到 <arguments> 标签，在里面加上 -Dhudson.security.csrf.GlobalCrumbIssuerConfiguration.DISABLE\_CSRF\_PROTECTION=true。它的作用是关闭 CSRF 验证，不关的话，gitlab的 webhook 会一直报 403 错误，无法使用。加好参数后，在该目录命令行下输入 jenkins.exe restart 重启 Jenkins。

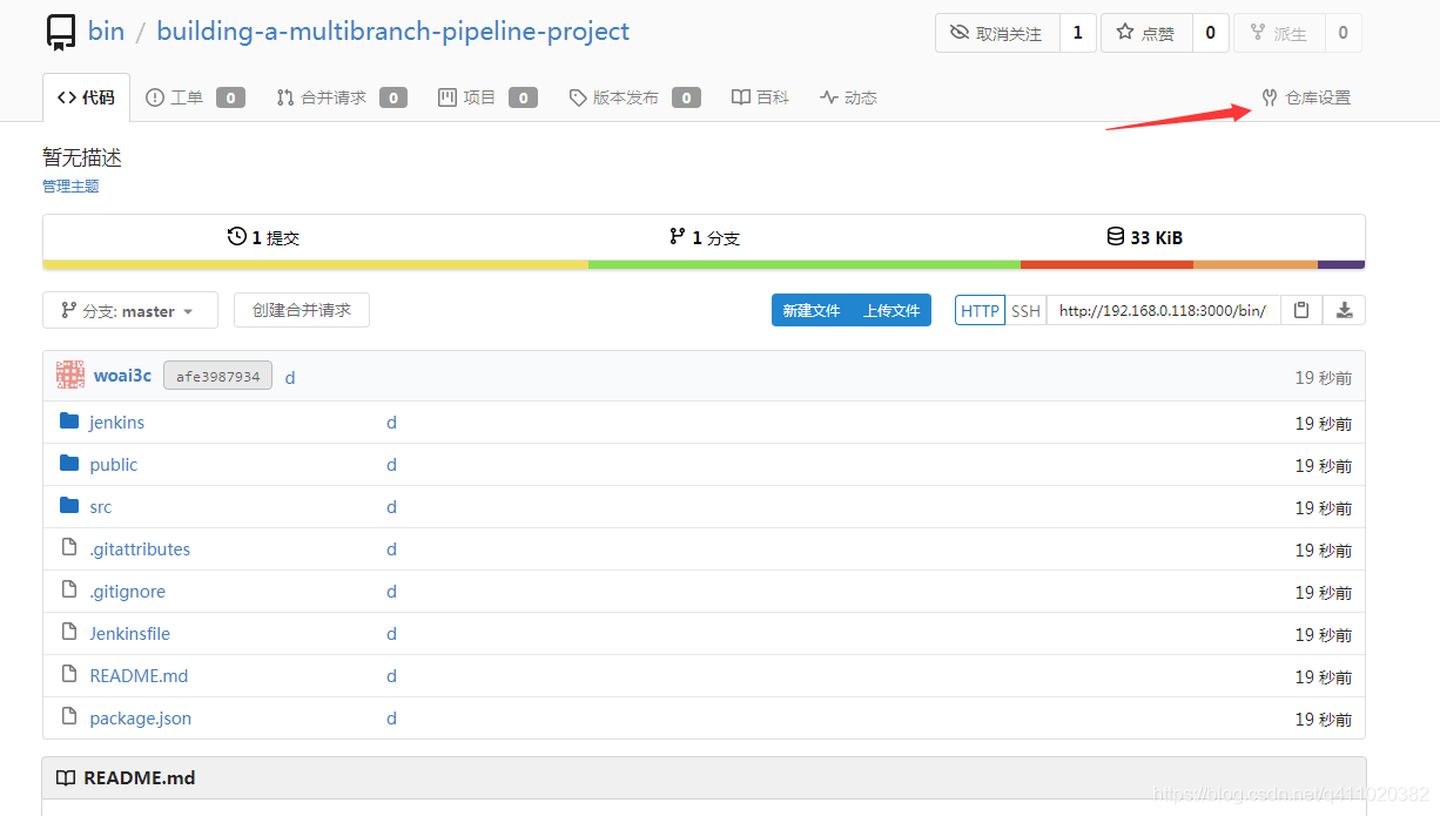


1. 回到首页，配置全局安全选项。勾上匿名用户具有可读权限，再保存。

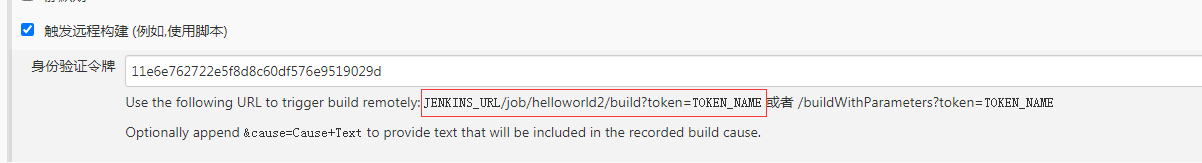


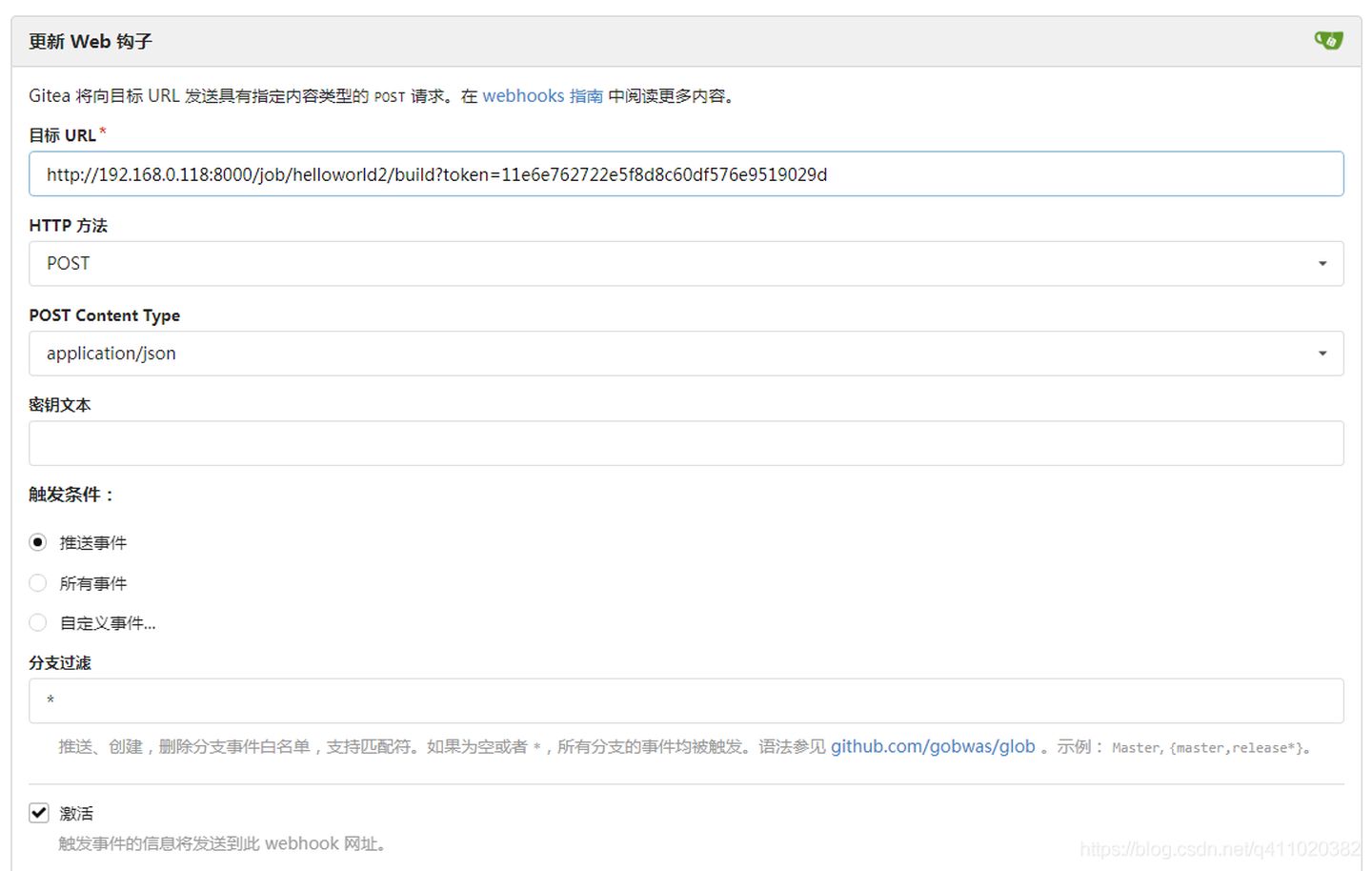


1. 打开你的 gitlab仓库页面，选择仓库设置。



1. 点击管理 web 钩子，添加 web 钩子，钩子选项选择 gitlab。
2. 目标 URL 按照 Jenkins 的提示输入内容。然后点击添加 web 钩子。





1. 点击创建好的 web 钩子，拉到下方，点击测试推送。不出意外，应该能看到推送成功的消息，此时回到 Jenkins 首页，发现已经在构建项目了。



1. 由于没有配置 Jenkinsfile 文件，此时构建是不会成功的。所以接下来需要配置一下 Jenkinsfile 文件。将以下代码复制到你 gitlab项目下的 Jenkinsfile 文件。jenkins 在构建时会自动读取文件的内容执行构建及部署操作。

pipeline {

    agent any

    stages {

        stage('Build') {

            steps {  // window 使用 bat， linux 使用 sh

                bat 'npm i'

                bat 'npm run build'

            }

        }

        stage('Deploy') {

            steps {

                bat 'xcopy .\\build\\\* D:\\node-server\\dist\\ /s/e/y' // 这里需要改成你的静态服务器资源目录

            }

        }

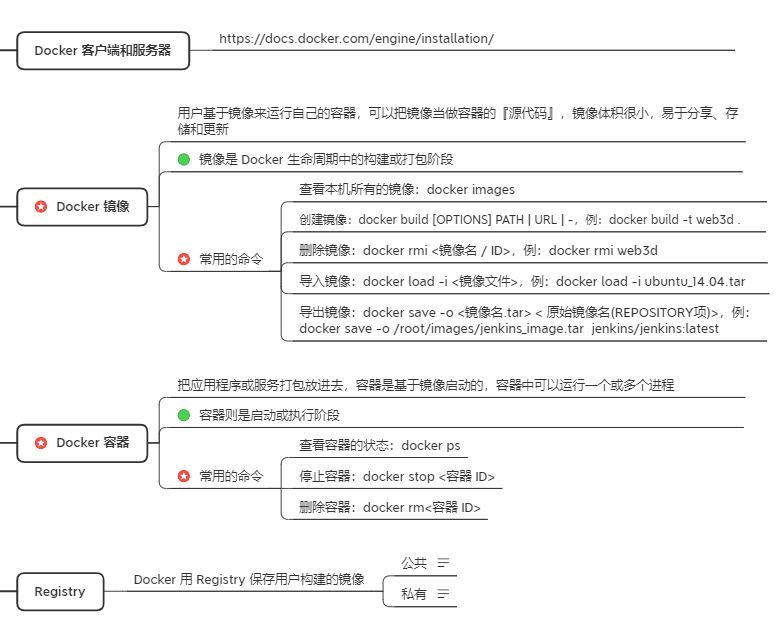
    }

}

1. 每当你的 Gitlab 项目执行 push 操作时，gitlab都会通过 webhook 发送一个 post 请求给 Jenkins，让它执行构建及部署操作。

## **前端docker部署**

1. **Docker基础介绍**

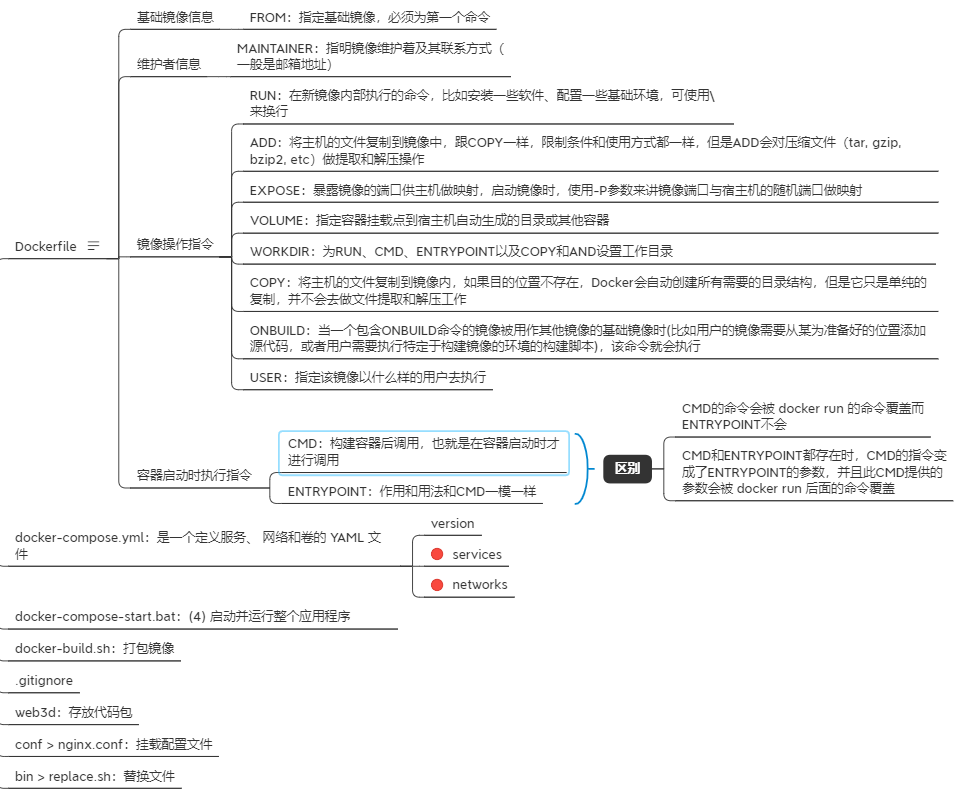


1. **配置文件说明**

**举例：三维平台docker前端部署**

Gitlab地址：http://172.16.11.133/SmartMap/web3d-frontend-docker.git

推荐使用分支：project



通过Dockerfile配置文件中



从中得出docker构建镜像时，是把nginx镜像包含进行，再运行容器时其实是直接使用nginx代理里面的静态文件。

1. **前端部署基本步骤**

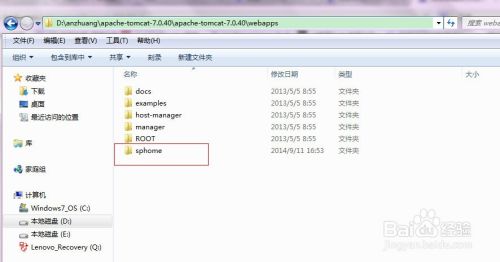
Docker下载安装地址：https://www.docker.com/products/docker-desktop

1. 代码打包，npm run build:docker
2. 将打好得包复制黏贴到对应的文件夹，如：web3d
3. 查看当前容器是否存在，docker ps
4. 删除容器，docker rm <容器id>（注意：如果容器在运行中，则需要先停止容器：docker stop <容器id>）
5. 删除镜像，docker rmi <镜像名>（注意：如果存在容器与镜像的情况，一定要先删除容器）
6. 构建镜像包，docker build -t web3d .（注意：镜像名后有: 空格 + 一点）
7. 启动并运行整个应用程序：docker-compose up -d

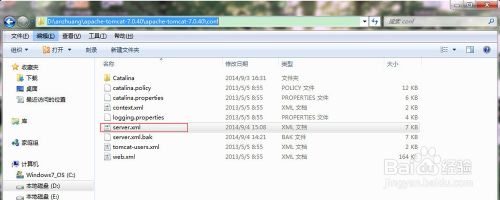
## **tomcat部署前端静态文件**

1. **直接放到webapps目录下**

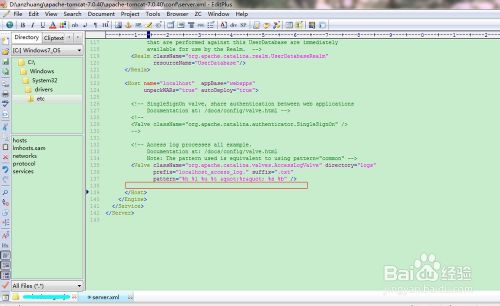
可以把文件直接放到webapps下面。当只是运行一个项目的时候，这种方法还好，但是当你涉及到两个以上项目的时候，就麻烦了（设定虚拟目录的方法，下面详细介绍。）。



1. **在server.xml中指定**
2. 找到Tomcat下的conf文件夹下的server.xml

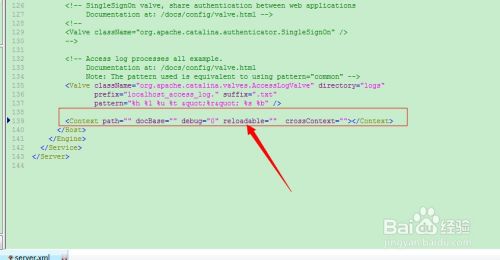


1. 通过习惯用的编辑器打开server.xml，可以看到Host标签，默认就有一个，一个Host代表一个站点，找到Host结束标签，我们在这里配置虚拟路径。



1. 在</Host>的上面添加<Context path="" docBase="" reloadable="" debug="" crossContext=""/>，这里属性值都没有填写，下面填写属性值。

其中path是虚拟路径，docBase是JSP应用程序的物理路径，workDir是这个应用的工作目录，存放运行是生成的于这个应用相关的文件。



1. **创建一个Context文件**

以上两种方法，Web应用被服务器加载后都会在Tomcat的conf\catalina\localhost目录下生成一个XML文件，其内容如下：

<Context path="/admin" docBase="${catalina.home}/server/webapps/admin" debug="0" privileged="true"></Context>

可以看出，文件中描述一个应用程序的Context信息，其内容和server.xml中的Context信息格式是一致的，文件名便是虚拟目录名。您可以直接建立这样的一个xml文件，放在Tomcat的conf\catalina\localhost目录下。例子如下：

注意：删除一个Web应用同时也要删除webapps下相应的文件夹祸server.xml中相应的Context，还要将Tomcat的conf\catalina\localhost目录下相应的xml文件删除。否则Tomcat仍会岸配置去加载。

## **Nginx代理**

下载Nginx网上教程很多，这里简单介绍配置内容。

server {

listen 8080;

server\_name localhost;

location / {

root D:\Webapps;

index index.html index.htm;

}

}

上面配置意思就是等于说把D:\Webapps的应用代理到localhost:8080服务下。

当然Nginx还有更多的配置，比如负载均衡，缓存，https等等相关服务配置。