前端工程化

下

1. 技术选型
2. 统一规范
3. 测试
4. 部署
5. 监控
6. 性能优化
7. 重构

4.部署

手动部署：

1. 执行测试 npm run test。
2. 构建项目 npm run build。
3. 将打包好的文件放到静态服务器。

自动部署：

**Gitea + Jenkins** 自动构建前端项目并部署到服务器

环境准备

**1.**下载安装**Gitea**。打开 **Gitea** 的安装目录，找到 **custom\conf\app.ini**，在里面加上一行代码 **START\_SSH\_SERVER = true**。这时就可以使用 **ssh** 进行 **push** 操作了

取消 **git** 代理

git config --global --unset http.proxy

git config --global --unset https.proxy

**2. push**项目到**Gtiea**上

**3.** 下载安装**Jenkins**

**4.** 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，**Gitea** 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**。

**5.** 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。

使用 **pipeline** 构建项目

**1.** 在**Jenkins**点击首页右上角的用户名，选择设置

**2.** 添加 **Api token**，记得将 **token** 保存起来。

**3.** 打开 **Jenkins** 首页，点击新建 **Item** 创建项目。

**4.** 点击构建触发器，选择触发远程构建，填入刚才创建的 **token**。

**5.** 选择流水线，按照提示输入内容，然后点击保存

**6.** 打开你的 **Gitea** 仓库页面，选择仓库设置。

**7.**点击管理 web 钩子，添加 **web** 钩子，钩子选项选择Gitea。

**8.**目标 **URL** 按照 **Jenkins** 的提示输入内容。然后点击添加 web 钩子。

**9.** 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。

**10.** 配置**Gitea** 项目下的 Jenkinsfile 文件

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

}

}

}

}

**11.** 每当你的 **Gitea** 项目执行 push 操作时，**Gitea** 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。

5.监控

性能监控

一般利用 window.performance 来进行数据采集

使用方法（以Chrome为例）：

1.打开F12开发者工具

2.选择Console

3.在命令行处输入上述命令

数据诊断：

Performance 接口可以获取到当前页面中与性能相关的信息，它是 High Resolution Time API 的一部分，同时也融合了 Performance Timeline API、Navigation Timing API、 User Timing API 和 Resource Timing API。

timing: {

*//* 同一个浏览器上一个页面卸载*(unload)*结束时的时间戳。如果没有上一个页面，这个值会和*fetchStart*相同。

navigationStart: 1543806782096,

*//* 上一个页面*unload*事件抛出时的时间戳。如果没有上一个页面，这个值会返回*0*。

unloadEventStart: 1543806782523,

*//* 和 *unloadEventStart* 相对应，*unload*事件处理完成时的时间戳。如果没有上一个页面*,*这个值会返回*0*。

unloadEventEnd: 1543806782523,

*//* 第一个*HTTP*重定向开始时的时间戳。如果没有重定向，或者重定向中的一个不同源，这个值会返回*0*。

redirectStart: 0,

*//* 最后一个*HTTP*重定向完成时（也就是说是*HTTP*响应的最后一个比特直接被收到的时间）的时间戳。

*//* 如果没有重定向，或者重定向中的一个不同源，这个值会返回*0.*

redirectEnd: 0,

*//* 浏览器准备好使用*HTTP*请求来获取*(fetch)*文档的时间戳。这个时间点会在检查任何应用缓存之前。

fetchStart: 1543806782096,

*// DNS* 域名查询开始的*UNIX*时间戳。

*//*如果使用了持续连接*(persistent connection)*，或者这个信息存储到了缓存或者本地资源上，这个值将和*fetchStart*一致。

domainLookupStart: 1543806782096,

*// DNS* 域名查询完成的时间*.*

*//*如果使用了本地缓存（即无 *DNS* 查询）或持久连接，则与 *fetchStart* 值相等

domainLookupEnd: 1543806782096,

*// HTTP*（*TCP*）域名查询结束的时间戳。

*//*如果使用了持续连接*(persistent connection)*，或者这个信息存储到了缓存或者本地资源上，这个值将和 *fetchStart*一致。

connectStart: 1543806782099,

*// HTTP*（*TCP*）返回浏览器与服务器之间的连接建立时的时间戳。

*//* 如果建立的是持久连接，则返回值等同于*fetchStart*属性的值。连接建立指的是所有握手和认证过程全部结束。

connectEnd: 1543806782227,

*// HTTPS* 返回浏览器与服务器开始安全链接的握手时的时间戳。如果当前网页不要求安全连接，则返回*0*。

secureConnectionStart: 1543806782162,

*//* 返回浏览器向服务器发出*HTTP*请求时（或开始读取本地缓存时）的时间戳。

requestStart: 1543806782241,

*//* 返回浏览器从服务器收到（或从本地缓存读取）第一个字节时的时间戳。

*//*如果传输层在开始请求之后失败并且连接被重开，该属性将会被数制成新的请求的相对应的发起时间。

responseStart: 1543806782516,

*//* 返回浏览器从服务器收到（或从本地缓存读取，或从本地资源读取）最后一个字节时

*//*（如果在此之前*HTTP*连接已经关闭，则返回关闭时）的时间戳。

responseEnd: 1543806782537,

*//* 当前网页*DOM*结构开始解析时（即*Document.readyState*属性变为*“loading”*、相应的 *readystatechange*事件触发时）的时间戳。

domLoading: 1543806782573,

*//* 当前网页*DOM*结构结束解析、开始加载内嵌资源时（即*Document.readyState*属性变为*“interactive”*、相应的*readystatechange*事件触发时）的时间戳。

domInteractive: 1543806783203,

*//* 当解析器发送*DOMContentLoaded* 事件，即所有需要被执行的脚本已经被解析时的时间戳。

domContentLoadedEventStart: 1543806783203,

*//* 当所有需要立即执行的脚本已经被执行（不论执行顺序）时的时间戳。

domContentLoadedEventEnd: 1543806783216,

*//* 当前文档解析完成，即*Document.readyState* 变为 *'complete'*且相对应的*readystatechange* 被触发时的时间戳

domComplete: 1543806783796,

*// load*事件被发送时的时间戳。如果这个事件还未被发送，它的值将会是*0*。

loadEventStart: 1543806783796,

*//* 当*load*事件结束，即加载事件完成时的时间戳。如果这个事件还未被发送，或者尚未完成，它的值将会是*0.*

loadEventEnd: 1543806783802

}

可以计算得出

*//* 重定向耗时

redirect: timing.redirectEnd - timing.redirectStart,

*// DOM* 渲染耗时

dom: timing.domComplete - timing.domLoading,

*//* 页面加载耗时

load: timing.loadEventEnd - timing.navigationStart,

*//* 页面卸载耗时

unload: timing.unloadEventEnd - timing.unloadEventStart,

*//* 请求耗时

request: timing.responseEnd - timing.requestStart,

*//* 获取性能信息时当前时间

time: **new** Date().getTime(),

白屏时间（从输入网址，到页面开始显示内容的时间）

将以下脚本放在 </head> 前面就能获取白屏时间。

<script>

whiteScreen = **new** Date() - performance.timing.navigationStart

</script>

相关资源加载时间

**window.performance.getEntriesByType(‘resource')**

关键数据

*//* 资源的名称

name: item.name,

*//* 资源加载耗时

duration: item.duration.toFixed(2),

*//* 资源大小

size: item.transferSize,

*//* 资源所用协议

protocol: item.nextHopProtocol,

可能影响页面性能的因素

1. 资源过多
2. 网速过慢
3. DOM元素过多

**SPA** （**Single Page Application**）的监控

对于SPA，在 SPA 切换路由时，**window.performance.timing** 的数据不会更新。

Vue场景对策

一个可行的办法就是切换路由时，在路由的全局前置守卫 beforeEach 里获取开始时间，在组件的 mounted 钩子里执行 vm.$nextTick 函数来获取组件的渲染完毕时间。

router.beforeEach((to, **from**, next) => {

store.commit('setPageLoadedStartTime', **new** Date())

})

mounted() {

**this**.$nextTick(() => {

**this**.$store.commit('setPageLoadedTime', **new** Date() - **this**.$store.state.pageLoadedStartTime)

})

}

性能数据上报

window.onload = () => {

*//* 在浏览器空闲时间获取性能及资源信息

*// https://developer.mozilla.org/zh-CN/docs/Web/API/Window/requestIdleCallback*

**if** (window.requestIdleCallback) {

window.requestIdleCallback(() => {

monitor.performance = getPerformance()

monitor.resources = getResources()

})

} **else** {

setTimeout(() => {

monitor.performance = getPerformance()

monitor.resources = getResources()

}, 0)

}

}

错误监控

1. 资源加载错误，通过 addEventListener('error', callback, true) 在捕获阶段捕捉资源加载失败错误。
2. js 执行错误，通过 window.onerror 捕捉 js 错误。
3. promise 错误，通过 addEventListener('unhandledrejection', callback)捕捉 promise 错误，但是没有发生错误的行数，列数等信息，只能手动抛出相关错误信息。

错误数据上报

*//* 监听 *js* 错误

window.onerror = **function**(msg, url, row, col, error) {

**const** data = {

type: 'javascript',

row: row,

col: col,

msg: error && error.stack? error.stack : msg,

url: url,

*//* 错误发生的时间

time: **new** Date().getTime(),

}

*//* 即时上报

axios.post({ url: 'xxx', data, })

}

用户信息收集

**navigator**

使用 window.navigator 可以收集到用户的设备信息，操作系统，浏览器信息...

浏览深度

通过 document.documentElement.scrollTop 属性以及屏幕高度，可以判断用户是否浏览完网站内容。

页面跳转来源

通过 document.referrer 属性，可以知道用户是从哪个网站跳转而来。

6.性能优化

检查加载性能

* 白屏时间
* 首屏时间：指从输入网址，到页面完全渲染的时间。在 window.onload 事件里执行 new Date() - performance.timing.navigationStart 即可获取首屏时间。

检查运行性能

以Chrome为例

1.打开网站，按 F12 选择 performance

2.点击左上角的灰色圆点，变成红色就代表开始记录了

3.模仿用户使用网站

4.在使用完毕后，点击 stop获取性能报告。

如果有红色的块，代表有掉帧的情况；如果是绿色，则代表 FPS 很好

利用工具检查

**chrome** 工具 **Lighthouse**

**<https://developers.google.com/web/tools/lighthouse>**

7.重构

[《重构2》](https://book.douban.com/subject/30468597/)一书中对重构进行了定义：

所谓重构（refactoring）是这样一个过程：在不改变代码外在行为的前提下，对代码做出修改，以改进程序的内部结构。重构是一种经千锤百炼形成的有条不紊的程序整理方法，可以最大限度地减小整理过程中引入错误的概率。本质上说，重构就是在代码写好之后改进它的设计。

重构的原则

1. 事不过三，三则重构。即不能重复写同样的代码，在这种情况下要去重构。
2. 如果一段代码让人很难看懂，那就该考虑重构了。
3. 如果已经理解了代码，但是非常繁琐或者不够好，也可以重构。
4. 过长的函数，需要重构。
5. 一个函数最好对应一个功能，如果一个函数被塞入多个功能，那就要对它进行重构了。

重构手法

在[《重构2》](https://book.douban.com/subject/30468597/)这本书中，介绍了多达上百个重构手法。比较常用的：

1. 提取重复代码，封装成函数
2. 拆分太长或功能太多的函数

强烈推荐阅读[《重构2》](https://book.douban.com/subject/30468597/)这本书。

附录

错误收集DEMO

**<!DOCTYPE html>**

**<html>**

**<head>**

**<meta charset="UTF-8">**

**<meta name="viewport" content="width=device-width, initial-scale=1.0">**

**<meta http-equiv="X-UA-Compatible" content="ie=edge">**

**<script>**

**function monitorInit() {**

**const monitor = {**

**//** 数据上传地址

**url: '',**

**//** 性能信息

**performance: {},**

**//** 资源信息

**resources: {},**

**//** 错误信息

**errors: [],**

**//** 用户信息

**user: {**

**//** 屏幕宽度

**screen: screen.width,**

**//** 屏幕高度

**height: screen.height,**

**//** 浏览器平台

**platform: navigator.platform,**

**//** 浏览器的用户代理信息

**userAgent: navigator.userAgent,**

**//** 浏览器用户界面的语言

**language: navigator.language,**

**},**

**//** 手动添加错误

**addError(error) {**

**const obj = {}**

**const { type, msg, url, row, col } = error**

**if (type) obj.type = type**

**if (msg) obj.msg = msg**

**if (url) obj.url = url**

**if (row) obj.row = row**

**if (col) obj.col = col**

**obj.time = new Date().getTime()**

**monitor.errors.push(obj)**

**},**

**//** 重置 **monitor** 对象

**reset() {**

**window.performance && window.performance.clearResourceTimings()**

**monitor.performance = getPerformance()**

**monitor.resources = getResources()**

**monitor.errors = []**

**},**

**//** 清空 **error** 信息

**clearError() {**

**monitor.errors = []**

**},**

**//** 上传监控数据

**upload() {**

**//** 自定义上传

**// axios.post({**

**// url: monitor.url,**

**// data: {**

**// performance,**

**// resources,**

**// errors,**

**// user,**

**// }**

**// })**

**},**

**//** 设置数据上传地址

**setURL(url) {**

**monitor.url = url**

**},**

**}**

**//** 获取性能信息

**const getPerformance = () => {**

**if (!window.performance) return**

**const timing = window.performance.timing**

**const performance = {**

**//** 重定向耗时

**redirect: timing.redirectEnd - timing.redirectStart,**

**//** 白屏时间

**whiteScreen: whiteScreen,**

**// DOM** 渲染耗时

**dom: timing.domComplete - timing.domLoading,**

**//** 页面加载耗时

**load: timing.loadEventEnd - timing.navigationStart,**

**//** 页面卸载耗时

**unload: timing.unloadEventEnd - timing.unloadEventStart,**

**//** 请求耗时

**request: timing.responseEnd - timing.requestStart,**

**//** 获取性能信息时当前时间

**time: new Date().getTime(),**

**}**

**return performance**

**}**

**//** 获取资源信息

**const getResources = () => {**

**if (!window.performance) return**

**const data = window.performance.getEntriesByType('resource')**

**const resource = {**

**xmlhttprequest: [],**

**css: [],**

**other: [],**

**script: [],**

**img: [],**

**link: [],**

**fetch: [],**

**//** 获取资源信息时当前时间

**time: new Date().getTime(),**

**}**

**data.forEach(item => {**

**const arry = resource[item.initiatorType]**

**arry && arry.push({**

**//** 资源的名称

**name: item.name,**

**//** 资源加载耗时

**duration: item.duration.toFixed(2),**

**//** 资源大小

**size: item.transferSize,**

**//** 资源所用协议

**protocol: item.nextHopProtocol,**

**})**

**})**

**return resource**

**}**

**window.onload = () => {**

**//** 在浏览器空闲时间获取性能及资源信息 **https://developer.mozilla.org/zh-CN/docs/Web/API/Window/requestIdleCallback**

**if (window.requestIdleCallback) {**

**window.requestIdleCallback(() => {**

**monitor.performance = getPerformance()**

**monitor.resources = getResources()**

**console.log('**页面性能信息**')**

**console.log(monitor.performance)**

**console.log('**页面资源信息**')**

**console.log(monitor.resources)**

**})**

**} else {**

**setTimeout(() => {**

**monitor.performance = getPerformance()**

**monitor.resources = getResources()**

**console.log('**页面性能信息**')**

**console.log(monitor.performance)**

**console.log('**页面资源信息**')**

**console.log(monitor.resources)**

**}, 0)**

**}**

**}**

**//** 捕获资源加载失败错误 **js css img...**

**addEventListener('error', e => {**

**const target = e.target**

**if (target != window) {**

**monitor.errors.push({**

**type: target.localName,**

**url: target.src || target.href,**

**msg: (target.src || target.href) + ' is load error',**

**//** 错误发生的时间

**time: new Date().getTime(),**

**})**

**console.log('**所有的错误信息**')**

**console.log(monitor.errors)**

**}**

**}, true)**

**//** 监听 **js** 错误

**window.onerror = function(msg, url, row, col, error) {**

**monitor.errors.push({**

**type: 'javascript', //** 错误类型

**row: row, //** 发生错误时的代码行数

**col: col, //** 发生错误时的代码列数

**msg: error && error.stack? error.stack : msg, //** 错误信息

**url: url, //** 错误文件

**time: new Date().getTime(), //** 错误发生的时间

**})**

**console.log('**所有的错误信息**')**

**console.log(monitor.errors)**

**}**

**//** 监听 **promise** 错误缺点是获取不到行数数据

**addEventListener('unhandledrejection', e => {**

**monitor.errors.push({**

**type: 'promise',**

**msg: (e.reason && e.reason.msg) || e.reason || '',**

**//** 错误发生的时间

**time: new Date().getTime(),**

**})**

**console.log('**所有的错误信息**')**

**console.log(monitor.errors)**

**})**

**return monitor**

**}**

**const monitor = monitorInit()**

**</script>**

**<link rel="stylesheet" href="test.css">**

**<title>Document</title>**

**</head>**

**<body>**

**<button class="btn1">**错误测试按钮**1</button>**

**<button class="btn2">**错误测试按钮**2</button>**

**<button class="btn3">**错误测试按钮**3</button>**

**<img src="https://avatars3.githubusercontent.com/u/22117876?s=460&v=4" alt="">**

**<img src="test.png" alt="">**

**<script src="192.168.10.15/test.js"></script>**

**<script>**

**document.querySelector('.btn1').onclick = () => {**

**setTimeout(() => {**

**console.log(button)**

**}, 0)**

**}**

**document.querySelector('.btn2').onclick = () => {**

**new Promise((resolve, reject) => {**

**reject({**

**msg: 'test.js promise is error'**

**})**

**})**

**}**

**document.querySelector('.btn3').onclick = () => {**

**throw ('**这是一个手动扔出的错误**')**

**}**

**</script>**

**</body>**

**</html>**

参考：

**<https://segmentfault.com/a/1190000037752931>**

**<https://juejin.cn/post/6887751398499287054>**

**<https://zhuanlan.zhihu.com/p/30329705>**

**<https://juejin.cn/post/6844903998412029959>**