Vue 全站服务器渲染 SSR 实践

**问：请问最佳实践是什么？ 要注意什么？ 性能怎么优化？**

掘金上一次 Web 改版实现了全站的 SSR，这其中包含了很多实践中的思考和体会。本场Chat我将与大家分享。

<http://gitbook.cn/m/mazi/article/5900675d2fde0e5078d4ed5e>

* 1. 如果我现在开启一个新的项目，怎么从头开始支持 SSR？

如果业务不复杂，建议使用 Nuxt.js 有比较完整的整合 vue 全家桶并持 SSR 的方案 <https://github.com/nuxt/nuxt.js>及<https://nuxtjs.org/>

它是一个在 Vue 之上套了一层的的解决方案，也有一些 starter 模版易于开始。如果你安装了 vue-cli，可以用 $ vue init nuxt/starter <project-name> 来直接开启一个 starter，现在支持：starter/express/koa/adonuxt（adonis.js 另一个后端框架）这 4 个基本设置模版。

而 nuxt 本身也包含了一些简单的 cli 命令，用 npm install -g nuxt 之后就可以：

$ nuxt dev

$ nuxt build

$ nuxt start

$ nuxt generate

来开始项目。

因为 Nuxt 本身是完全基于 Vue 的，因而这个库的基础就是设定了一些结构，帮助你更好地写适合 SSR 的 Vue 应用。



这个是 Nuxt.js 官方推荐的一个文件格式，其中最最重要的部分就是 nuxt.config.js。这里把相应的 webpack 下的配置文件定义都包裹进去了，不需要完全自己实现。

这是一个 nuxt.config.js 文件。基础的部分设置了这个 Vue 整体项目的 title／meta／... 属性，但是每一个 page 在 pages/ 文件夹里还可以独立设置。整个 Nuxt 是一个集成度很高的 VUE／SSR 解决方案，因此比较适合一个从 0 开始的项目，如果是一个现有业务再整合到 Nuxt 里成本比较高，不建议

* 1. 1.2 如果是一个已经比较成熟的 Vue.js 项目，想要开始支持 SSR

https://ssr.vuejs.org/en/ 已经给出了一个比较完整的产品结构方案。

基本的文件结构在文档里也讲的很清楚：

src

├── components

│ ├── Foo.vue

│ ├── Bar.vue

│ └── Baz.vue

├── App.vue

├── app.js # universal entry

├── entry-client.js # runs in browser only

└── entry-server.js # runs on server only

这里就要特别：你现有的 Vue 项目结构如果特别不符合标准（我是从 0.12.X 开始写 Vue 的，那个时候 Vue 项目是没有官方推荐的文件结构的）。

那么接入 SSR 甚至是接入整个 Vue 的开发生态的组件成本都很高

因此，你的 Vue 项目应该遵循 Vue 产品的基本结构，App.vue 和 /components 是项目的具体业务，而最初的 Vue 设置（包括 vue-router/vuex 及其他 plugin 的设置）会在这里，这样你再去配置 SSR 就相对比较简单，如果你本身的项目结构比较混乱在 SSR 设置上也会出现问题。

**1.3 一些我们自己的最佳实践**

1.3.1 掘金的 SSR 文件结构

我们的项目承载了掘金整个 web 端 SPA 业务需求从 0.12.x -> 1.0 -> 1.X -> 2.0，每次改版都有重构，其中包含了用户登录认证、多层级分页、多状态管理等需求，之前也被 Evan 评价过很多次，因而算是一个比较完整的 Vue 项目。

我抽出来了整个与 SSR 相关的我们产品的文件结构出来：

juejin/

├── backend/ # 后端业务

├── build/ # build 脚本

├── cache/ # 页面 Redis 缓存

├── src/

│ ├── api/ # 业务 API

│ ├── App/ # 网站外层 \*.vue

│ ├── component/ # 各个组件

│ ├── model/ # 数据层封装

│ ├── router/ # 路由

│ ├── state/ # 状态管理

│ ├── util/ # 工具函数

│ ├── view/ # 具体各个页面 \*.vue

│ ├── ...

│ ├── main.js # root Vue

│ ├── client-entry.js # 前端业务

│ └── client-server.js # 后端渲染业务

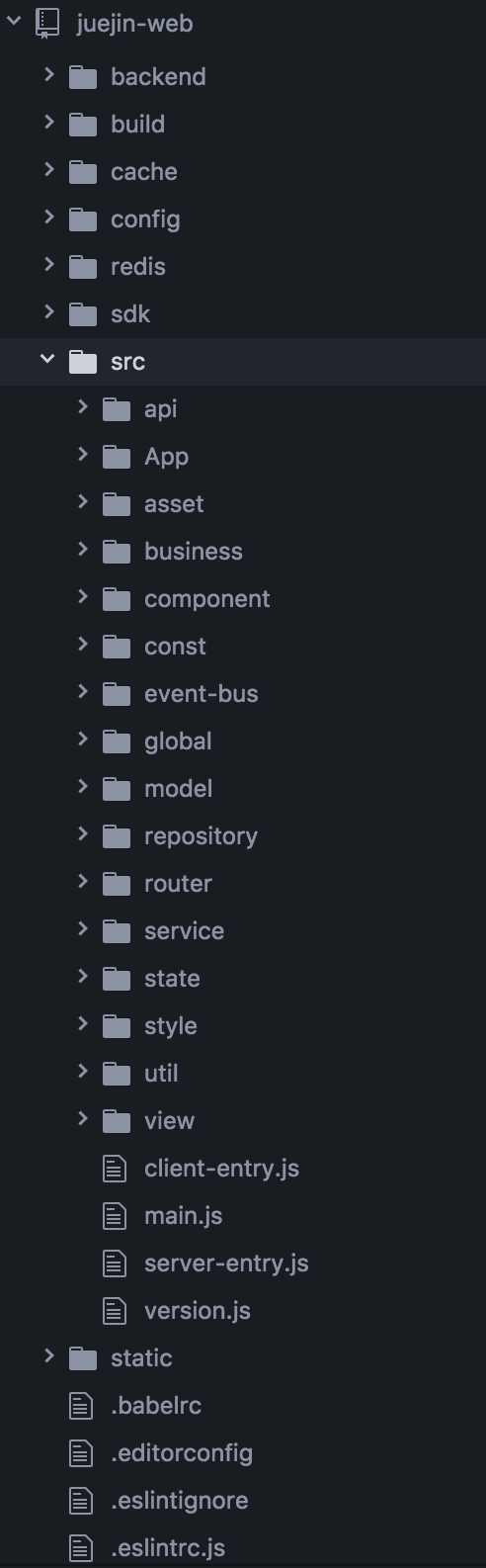
├── ...

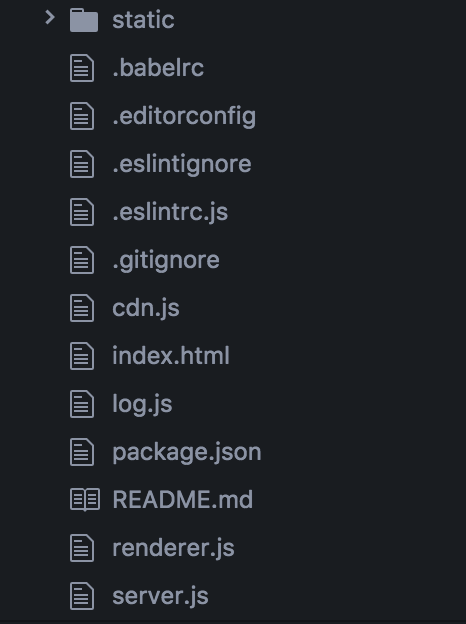
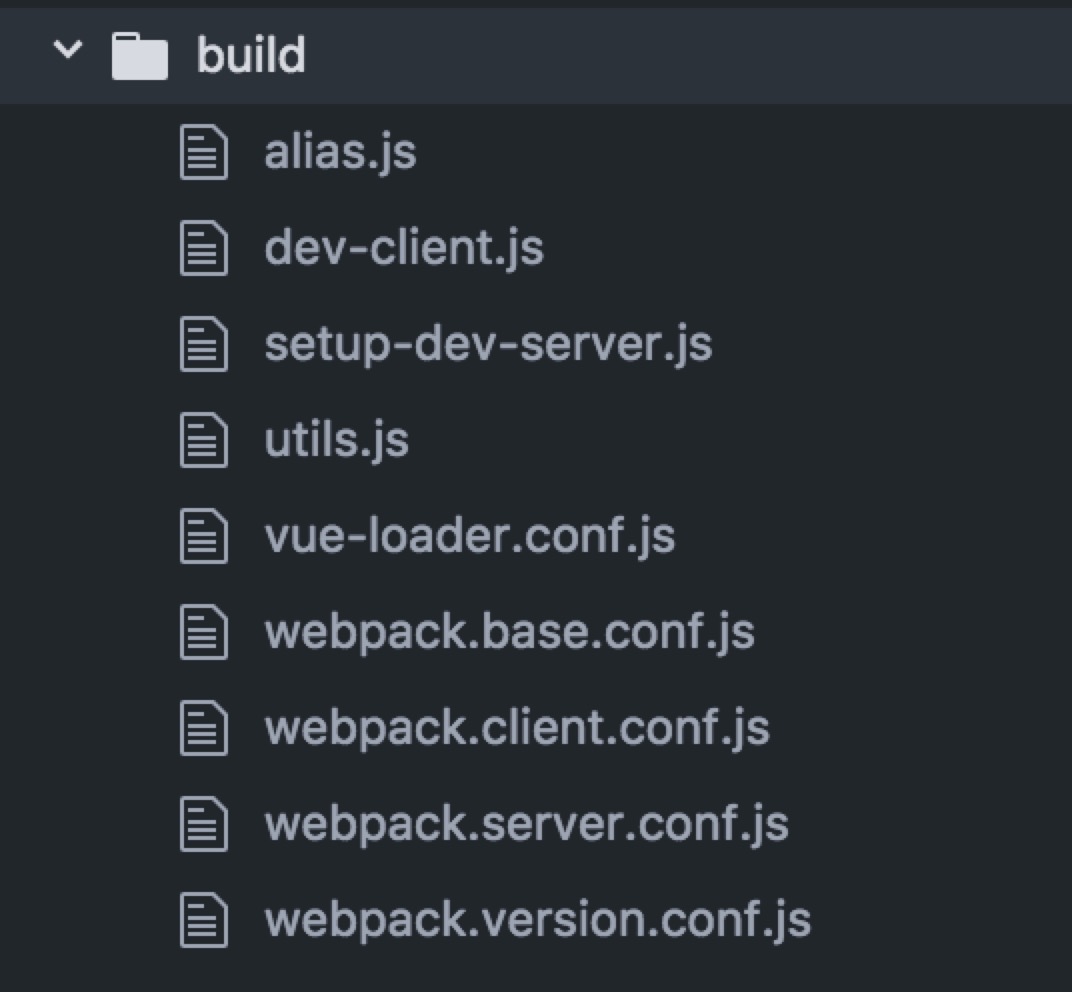
├── renderer.js # SSR 脚本

└── server.js # 后台开启脚本

这里面特币需要注意，server.js 是系统正式开始运作时调用的，它会定义后端业务层的路由，也会引入 renderer 来实现相关的 SSR 业务逻辑。

但是，其实在开启 server 前，项目会通过 build/ 文件夹下的 webpack 定义好如何提前 build 前端业务组件和 SSR 的 bundle 文件



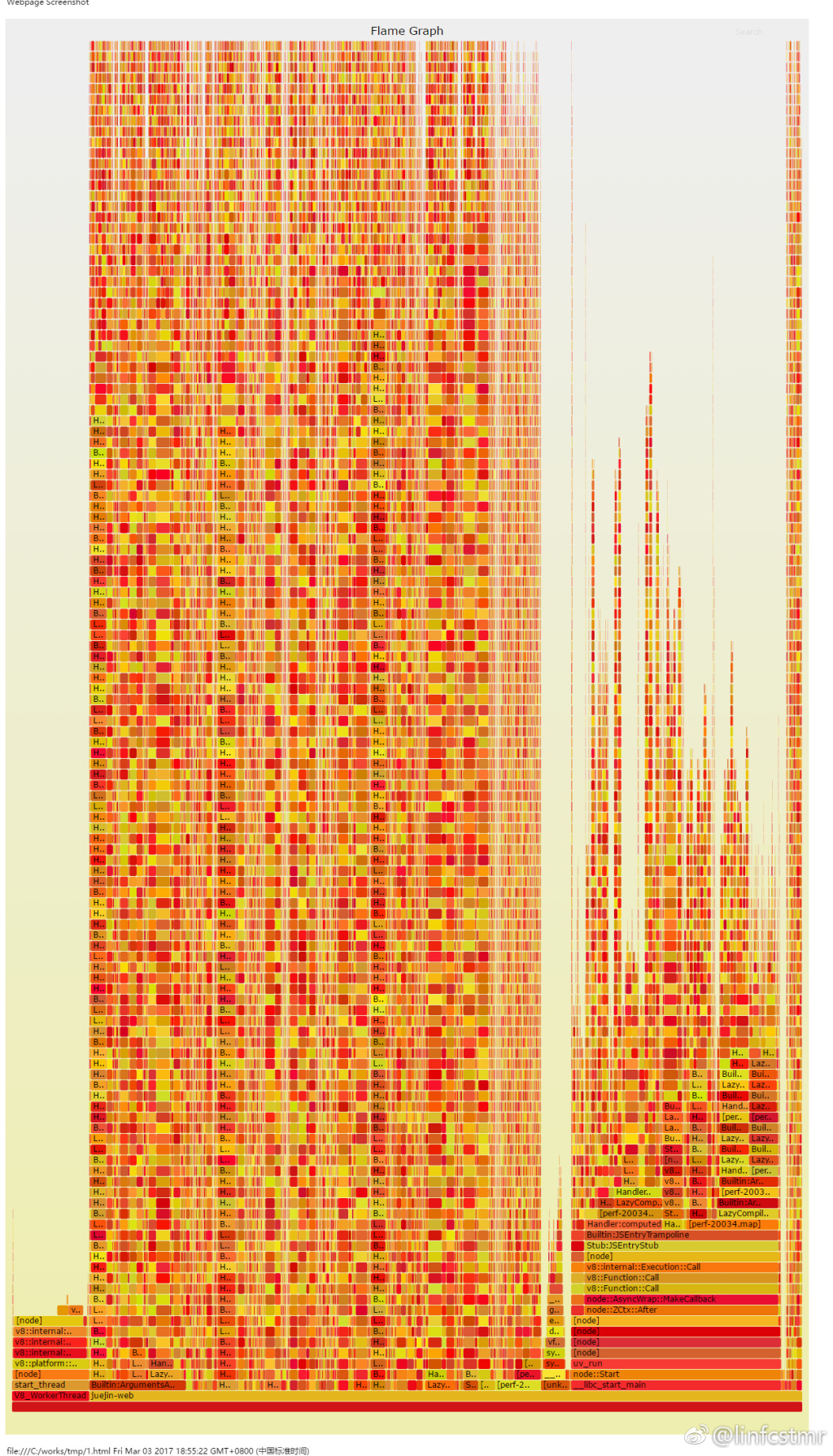
除了因为 Vue 项目比较大我们会把项目本身的业务存到更明确的文件夹内（如 router/, state/, view/, util/ 等等）我们也一定会有具体的 renderer.js 脚本来管理 SSR 的具体业务。webpack 的 build 脚本放到 /build 里并分成了 webpack.client.config.js 和 webpack.server.config.js 两个文件分别生成 Vue 前端业务和后端 SSR server-bundle 文件。

1.3.2 SSR 性能不好

（听 Evan 说新版会 10x faster），一定需要 cache 缓存

SSR 需要先在服务器端生成虚拟 DOM ，然后再序列化为 HTML 文本，而且为了请求之间的隔离性，每次请求都会创建一个新的 context ，这种做法使得它与直接使用模板引擎相比性能差了数十倍。我们在重构上线之前对我们的首页做了压测，几个并发直接就把一个 CPU 打满了，根本达不到上线水平。对于掘金这种多组件页面来说，CPU 是 SSR 的性能瓶颈，在压测时所记录的 CPU Profile 中可以明显地看到相关的操作耗费了大量的 CPU 资源。

这是我们在还没有上缓存前，压测的时候后端 SSR 的火箭图.



最终为了提高并发，对未登录用户我们添加了长达数分钟的页面缓存，对登录用户只直出根组件（组件少不加缓存性能也可以接受）。这里有一个问题，既然要尽可能减少服务器端渲染，而服务器端渲染的目的是做 SEO ，那为什么不只对爬虫请求做服务器端渲染呢？原因是怕被判定为作弊。

每一次后端 SSR 从拉取数据，后端引入 Vue 渲染出 DOM 再输出到 HTML 文件本身的性能是有限的，因而对于现在掘金的业务来说缓存层非常重要。本身和文档中相似，我们用 lru-cache（一个简单的 Node 下的简单 Object 缓存）可以自动生成组件缓存。但是我们对要吐数据的层做了 redis 的页面缓存，这部分请求反馈其实是在 Nginx 层就调用了 redis 来吐数据，这样速度更快。当页面的缓存过期后，才有用我们的 Node 后台的 SSR 渲染。

另一方面数据请求也会影响渲染时间，因而数据层最好也要有缓存。

综上所述，我们做了

1. 数据层的缓存

2. 渲染的页面层的缓存，不是每一个需求都会出发 SSR 后端渲染

3. 组建层的缓存。用 router.getMatchedComponents()

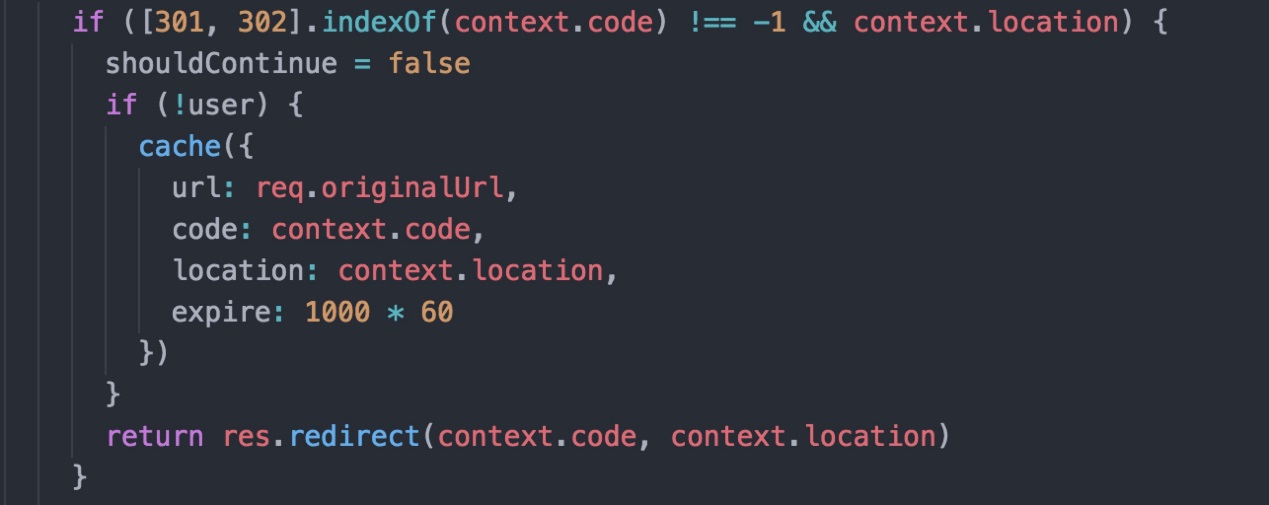
来解决高并发需求，现在 SSR 渲染过程有很多字符串操作，很吃 CPU，并发高必须加缓存

1.3.3 404页面的特别处理（这和 Vue／SSR 都有关系）

状态码机制

我们通过异常处理机制来实现常见的 301 、 404 等状态，在服务器端渲染的数据预取阶段，如果判定目标资源找不到或需要跳转，则抛出一个带有规范化信息的异常，终止后继步骤结束渲染，在外层捕获异常，然后通过解析其携带的信息来返回相应的结果。

例如，我们 SSR renderer.js 里有这样一段业务.



重定向触发的时候，用户未登录情况下，我们会缓存本身的重定向（这个是 redis 层的缓存），并完成重定向.

但是当一个内容找不到的时候：，我们需要返回 404 状态码，显示 404 页面，但是 URL 不会改变，这也就是说，一个 URL 会对应两个页面，一是正常页面，二是 404 页面.

这和 SSR 之前说的每一个 URL 又一个相应的渲染逻辑就不一样了

我们在路由表的最后配置了 404 路由，如果当前 URL 没有匹配前面的任意一条规则，就会显示 404 页面：

routes.js 定义里，这里 process.BROWSER 是在 webpack.client.config.js 里设置的的

{

path: '\*',

name: 'notFound',

component: process.BROWSER

? () => System.import(' view/NotFoundView ')

: require('view/NotFoundView')

}

以上做法可以应对 URL 不匹配的情况，但如果是匹配的情况下找不到资源呢？比如说给 `/entry/:entryId` 一个不存在的 `entryId` ，这就需要在预取时判断并做出响应了。

状态树中有一个 error module 用来存储当前的异常相关状态：

{

location: null,

errorView: null,

statusCode: 200

}

在 Vuex 里定义整体的 ERROR 事件的时候，就可以对每一个 ERROR state 特殊设置和处理

判定资源不存在时，在抛出异常之前会 dispatch 一个 action 将其设为 404 页面的状态：

[SHOW\_NOT\_FOUND\_VIEW] ({ commit }) {

commit(UPDATE\_STATE, {

errorView: 'NotFoundView',

statusCode: 404

})

}

在掘金网站的基础 template 里，对于任何 errorView 出现都有相应的展示



总结： SSR 本身依照文档，项目基本架构符合要求，并不是一个难的东西，现在最大的一个坑就是性能，需要缓存。

没有问题的话我们进入下一题：

**问：怎么区分开服务器渲染必须请求和浏览器端请求的？看了掘金未登录状态下的SSR很好，可是登陆后基本就没内容，有办法解决么？**

你在 server.js（express里）解析不同路由的时候就要设置清楚啊，需要后端渲染的 renderer 生成，不需要的就都抛给前端 vue-router 来处理。掘金的设置里优先 vue-router 后 server 渲染。

现在掘金的业务逻辑就是登陆后不做 SSR，因为 SSR 主要是为了 SEO，登陆与否不重要。而 SSR 本身现在有缓存／性能问题，因此能不用的时候反而是尽可能先不用。

注：这里 setupBackend(app) 里的定义主要是用户 auth 相关的。

**问题：问：异步请求获得的内容数据怎么等它在服务器抓完，再发回前端？（当然我用的是react，但原理差不多。为了对同一component不同请求得到的内容的ssr，目前我只能在rendertostring前持续检查props的是否准备好，好了在render，实际上还是很麻烦，因为没有async life cycle。不知道你们怎么处理的，有没有更好的办法。）@candy**

答：可看官方文档，最重要的就是 router.getMatchedComponents() 来拉取不同页面需要的组件，组件被 lru-cache 缓存了一层。

这里官方文档写的很清楚了：

// entry-server.js

import { createApp } from './app'

export default context => {

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

const { app, router, store } = createApp()

router.push(context.url)

router.onReady(() => {

const matchedComponents = router.getMatchedComponents()

if (!matchedComponents.length) {

reject({ code: 404 })

}

// call asyncData() on all matched route components

Promise.all(matchedComponents.map(Component => {

if (Component.asyncData) {

return Component.asyncData({

store,

route: router.currentRoute

})

}

})).then(() => {

// After all preFetch hooks are resolved, our store is now

// filled with the state needed to render the app.

// When we attach the state to the context, and the `template` option

// is used for the renderer, the state will automatically be

// serialized and injected into the HTML as window.\_\_INITIAL\_STATE\_\_.

context.state = store.state

resolve(app)

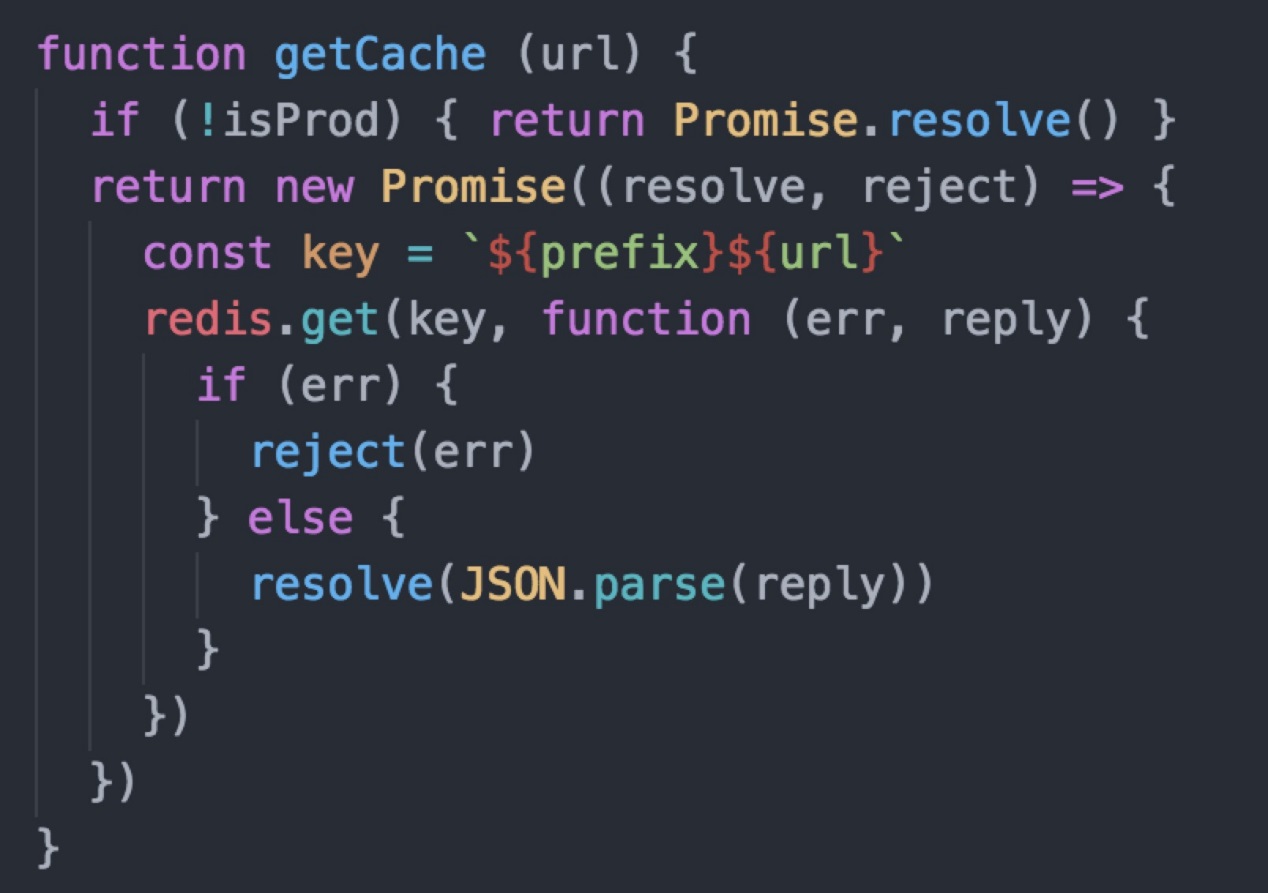
}).catch(reject)

}, reject)

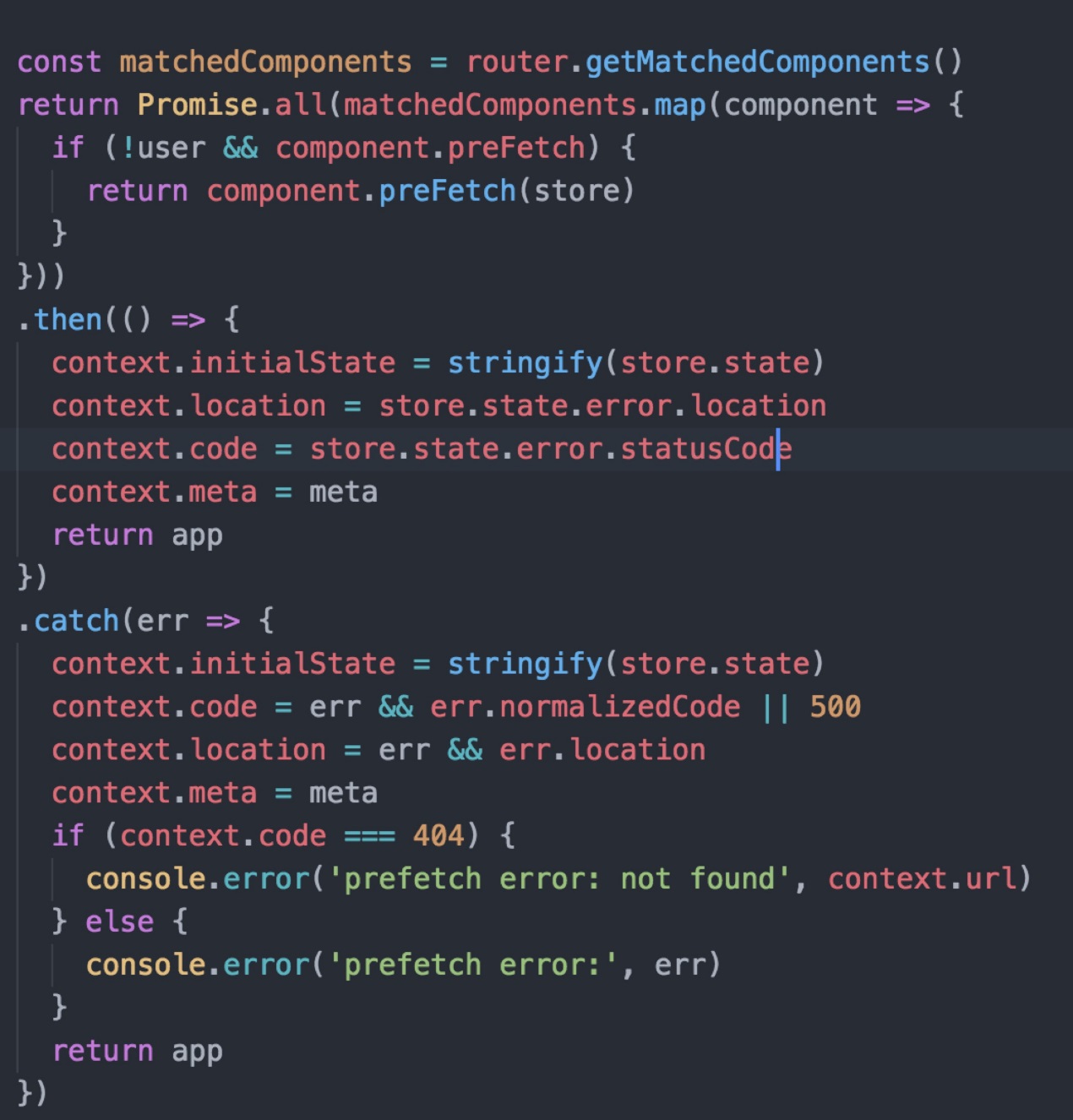
})

}

//注释：router.getMatchedComponents 用来抓组件，加了 lru-cache 可以多一层组件缓存, 抓下来组件之后开始根据 Component.asyncData 来抓数据, Promise.all 表示，需要把所有相关的组件下的所有数据都 fetch 下来才会 resolve 到下一个动作, asyncData 则是在相应的 ComponentName.vue 里面定义的. 如果想是我么掘金里又加了一层 SSR 本身的缓存，就会像是这样：



如果 redis 里直接拿出来了，就直接渲染 reply 了，如果没有就去重拉数据.



**答疑：**

(1) 网站公共数据也是放在then加载对吗？

基本上，如果 SSR 的页面没有数据层缓存，每一次渲染都要拉一遍所有以来数据;

(2) 比如做一个博客系统，博客的标题和导航菜单配置之类的数据。每个页面都要这个数据。

嗯，如果这个数据是在数据库或者远端，每次都要拉取。如果这个数据你本身放在了后端 Node 的内存里，自然就可以快一些, 不过这样的被依赖但是不是每次都要会变的数据，最好都缓存下.

(3) 问：l ssr 的性能高并发情况如何解决？现在能做到并发多少(有什么指标)？有 ssr 脚手架吗？

这个可能和之前第一个问题我的回复又重复内容,现在我们的解决方案就是多层缓存,我们做了:

1. 数据层的缓存

2. 渲染的页面层的缓存，不是每一个需求都会出发 SSR 后端渲染

3. 组建层的缓存

来解决高并发需求.

现在 SSR 渲染有非常多的字符串操作，非常吃 CPU 计算资源，无法支持高并发。如果页面内容数据／渲染逻辑都比较复杂，就很难做了

大家选择对一个 Vue 项目中的部分页面 SSR 一定是因为这个页面本身承载的是内容且交互不复杂，因而这部分页面才适合做 SSR 渲染，缓存了也是可以支持的

如果一个页面全部都是复杂交互，需要拉很多层数据，那本身就不太适合做 SSR 了.

Beta:

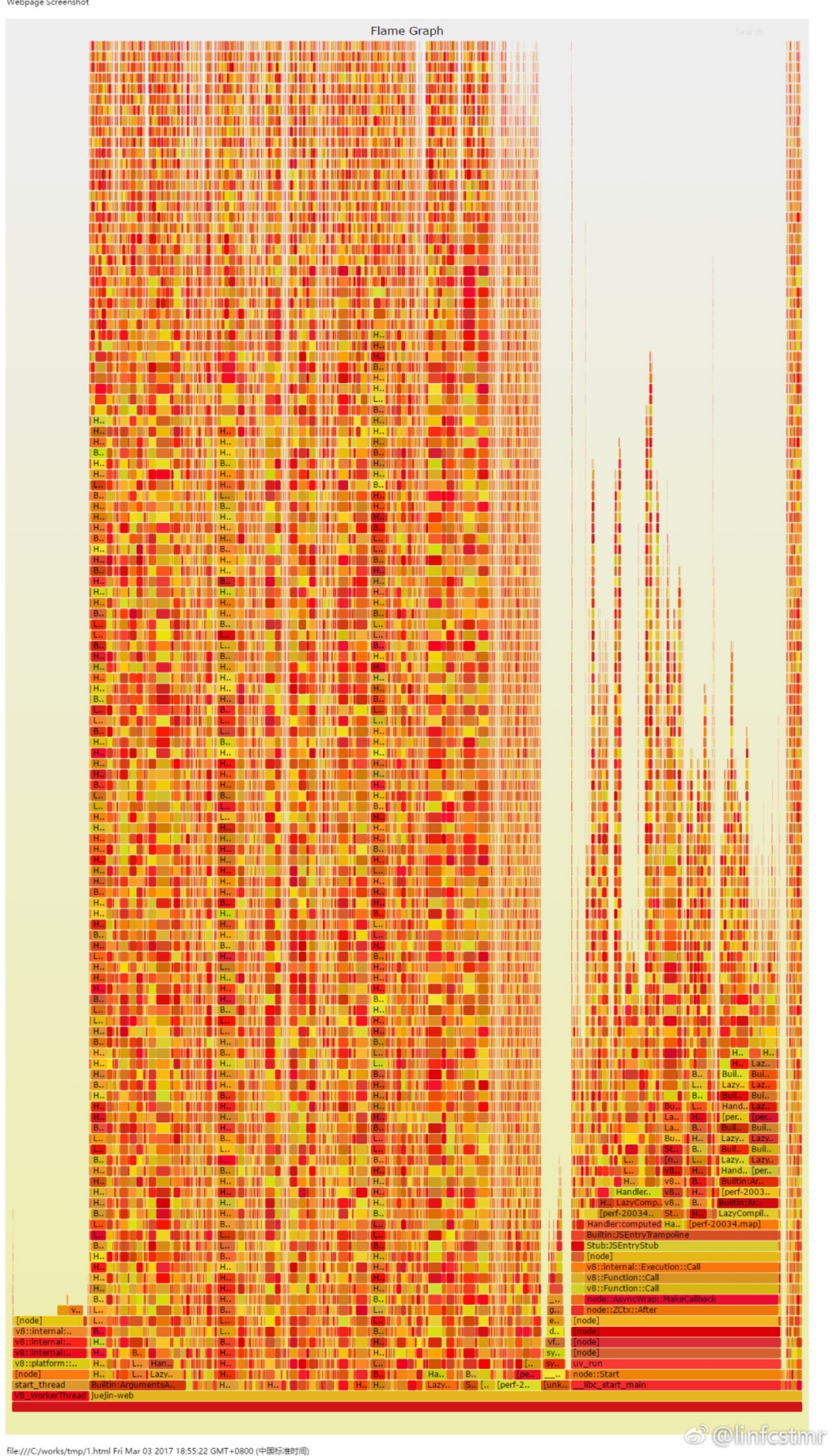
（3）最后一个问题：想了解一下既然使用vue进行SSR的性能代价那么大，为何仍旧还在掘金的部分页面中采用；如果是因为搜索引擎的原因，为什么不直接通过服务器端的进行渲染，而要用vue；是因为要实现前后端分离，还是有别的原因存在，这里的考量是怎么样的？

好问题.你说的没错，如果单纯是为了 SEO 后端渲染效果不差，缓存层的逻辑和 Vue SSR 上面说的其实是一样的。

这里我要介绍一下团队的结构,我们的后端是全套的微服务分解各个业务，而系统运维只到配置层。其实一个“前端”项目是从 Web 框架（如 Express）到前端业务都算在 Web 端的责任之重的,因而,我们非常相信也很支持 Vue 的发展，相信这样的完整 JS 框架会成为业务开发的主流，因而我们暂时将产品的整体业务按照 Vue.js 的框架实现，这部分的决策架在了完全性能优先之上。另外，随着 Vue／React 等框架 SSR 越来越成熟，彼此的竞争与共同发展一定会触发 SSR 业务优化的大进步，到那个时候性能应该不是问题。

前后端分离，这是架构基础，我们用了全套的微服务构建三断业务，和是否是 SSR 无关。

最后再给大家看一下 SSR 的 Flame Graph 真的很吓人(当然，掘金的页面逻辑确实也比较复杂了).



**答疑环节：**

卿本佳人: 大神的生命周期讲的挺厉害，能详细解释下吗

阴明@xitu.io:再说一次，SSR 本身的实现难度很大层面被 Vue 框架本身解决了，因此，不是很难实现。

阴明@xitu.io:

@卿本佳人 生命周期？

阴明@xitu.io:

你是说 Vue 的，还是 SSR 的额？

卿本佳人:

可以都讲下吗

卿本佳人:

我之前在一个会上听过大神的讲座讲的是vue的生命周期

阴明@xitu.io:

SSR 其实如果没有缓存的话，其实每一次都是

request 到了 server

server 处理，触发后端渲染

引入 renderer 和相应的 Vue 应用

renderer 开始正常运行 Vue，根据具体的 route 开始抓出相应的组件，和依赖的数据

开始加载组件／拉取数据（这部分是异步 Promise 来实现），先拉组件，再拉数据

数据都拉下来，组件都加载好了，Node 环境下开始以来 VDOM 逻辑生产 DOM

DOM 生产出来后就可以输出 HTML 了

然后用 renderToString 或者 renderToStream 吐给 response

阴明@xitu.io:

@卿本佳人 这里主要将 SSR 的生命周期，这个算事解答。

朱元雯 Nina:请问一下：是前后端分离的项目，为什么在掘金的web项目里有backend文件夹？里面是放的后端代码？

阴明@xitu.io:@Nina 我上面解释了，我们的“前端”包含了处理 request 这一层，而后端是开发微服务／数据库管理的，我们的 backend/ 文件夹里包含了 Express 的基本后端操作

后端不处理真正面向用户层的 request。

阴明@xitu.io:和大家分享一个 http://stackoverflow.com/trends 可以看到各个技术在 Stack Overflow 上的热门程度，Vue 确实很不错（并不只是大家平时转转微博，互相吐槽，而是有越来越多的人在真真的使用）

朱元雯 Nina:那接口呢？express的基本后端操作包括写接口么？

阴明@xitu.io: @Nina 是的。

阴明@xitu.io:

不过这些操作不包含具体数据的 CRUD 和业务的，因为整个业务都套在了 Vue 下，那些具体的数据／业务请求，都在 Vue 框架里，端操作其实非常非常简单，这个 Express 只负责 juejin.im/ 下的业务，完全不负责我们真正的具体业务。

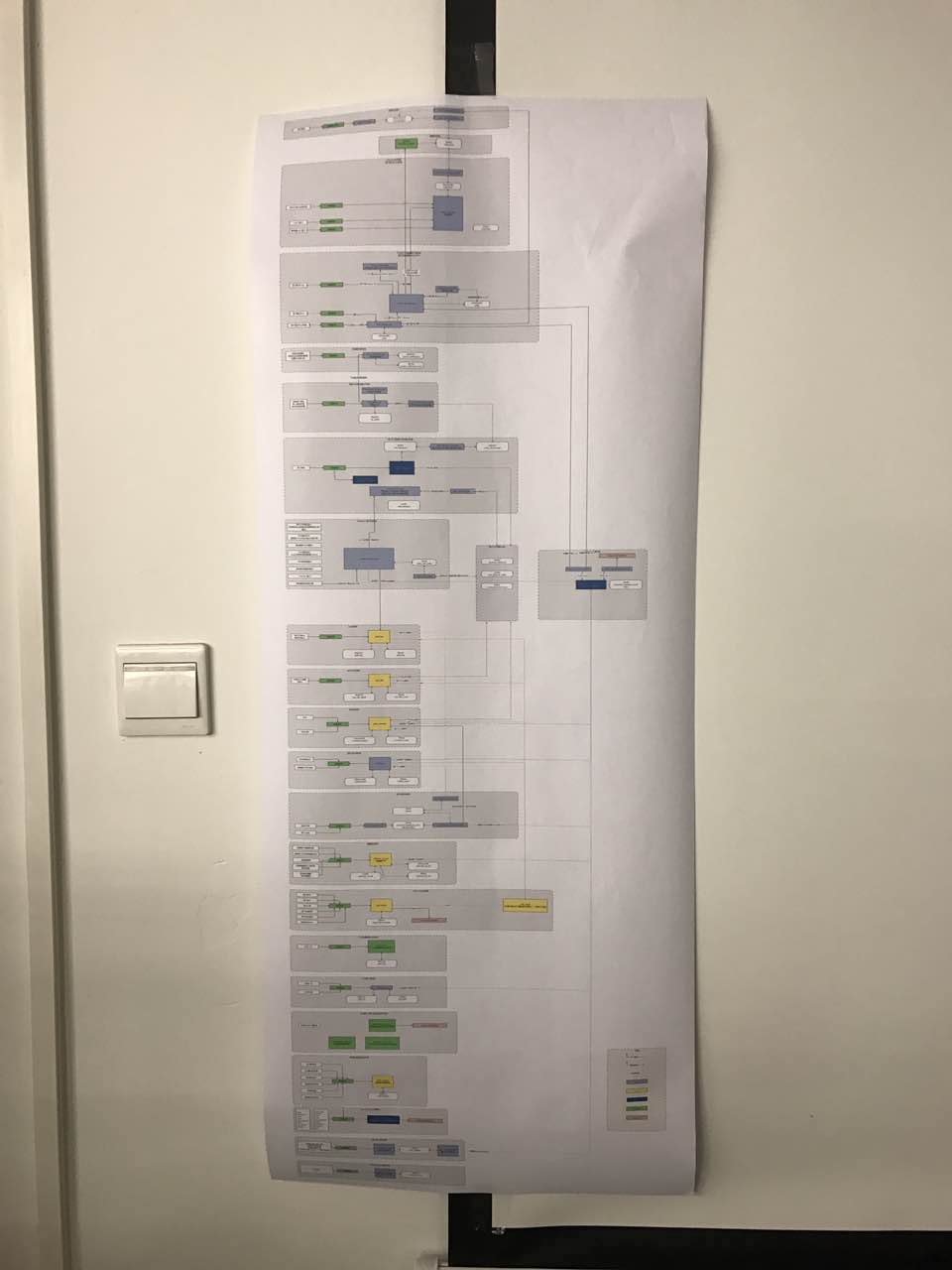
咸柠沙士:能介绍一下针对SSR，掘金在测试这块是怎么样一个做法？

阴明@xitu.io:@咸柠沙士 必须诚实的说，我们现在 SSR 这里还没有写测试。

羊犄角：前端这块相当于独立出来一个中间层 是么

@羊犄角 是的，你可以这么理解。Vue 业务，Express 处理最基础的几个请求。

boucy bear:能不能简单讲下整个掘金的软件架构。



阴明@xitu.io:@boucy bear因为包含技术隐私，这里可能不能深入到每一个模块给大家详解。

我们把整体业务都拆解到了完整的微服务系统里。就是处理 user 的就是一个单独的微服务，处理文章 post 也是一个单独的业务 。CRUD 发生的时候，是否要写入大数据系统、计算操作、推送等，也都分别拆解到了内部的微服务立了。我们技术负责人的逻辑是：原子化每一个业务，减少彼此间的完全依赖，让开发团队更加灵活。掘金表面业务很简单，其实背后是很复杂的。

咸柠沙士:请教我可以理解成在掘金里，目前是否就是把VUE作为解决 视图层的一个统一结构？

阴明@xitu.io:@咸柠沙士 并不只是 View 层的，而是整体端业务的逻辑。

BoBo：vue+express 做类似饿了么APP的手机网站，数据库选型mongodb和MySQL ，比较推荐什么呢？

业务如果简单 mongo 就好，如果未来请求超高并发，我觉得 mongo 可能药丸。

@BoBo MySQL 经久不衰，PostgreSQL 也可以。其实业务量级不大的时候，用啥都不太重要。量级起来问题就会出来。

羊犄角: vue打包部署，是直接放到静态文件里的么 还是通过路由访问的？

阴明@xitu.io:@羊犄角 前端业务肯定是之前就 build 好了呀，router 也都是 vue 定义的。SSR 的话，正式环境也是提前用 createBundleRenderer 在 webpack 里 build 好了 bundle 文件。详见：<https://ssr.vuejs.org/en/build-config.html>

链接：

阴明@xitu.io:大家无聊的话，可以去我个人主页／twitter／微博上和我互动。

<http://ming.today>

<https://twitter.com/kalasoo>

<https://weibo.com/kalasoo>

大家别忘了关注本期作者阴明，并为作者打分哈！

<http://gitbook.cn/m/mazi/rating/5900675d2fde0e5078d4ed5e>