Vue 全站服务器渲染 SSR 实践

> 掘金从 `0.12.X` 版本开始使用 Vue.js 开发产品，途经了掘金及 Vue.js 的高速发展，我们也不断迭代产品经历了 3 次大的重构。2017 过年后我们也经历了有史以来最大的一次重构。这篇文章我会着重讲述重构过程中对于 Vue.js 后端服务器渲染的掘金团队实践。

这里我就不再赘述什么是 Vue.js 了，请需要了解的同学前往：

1. vuejs.org / 中文文档
2. GitHub
3. 掘金

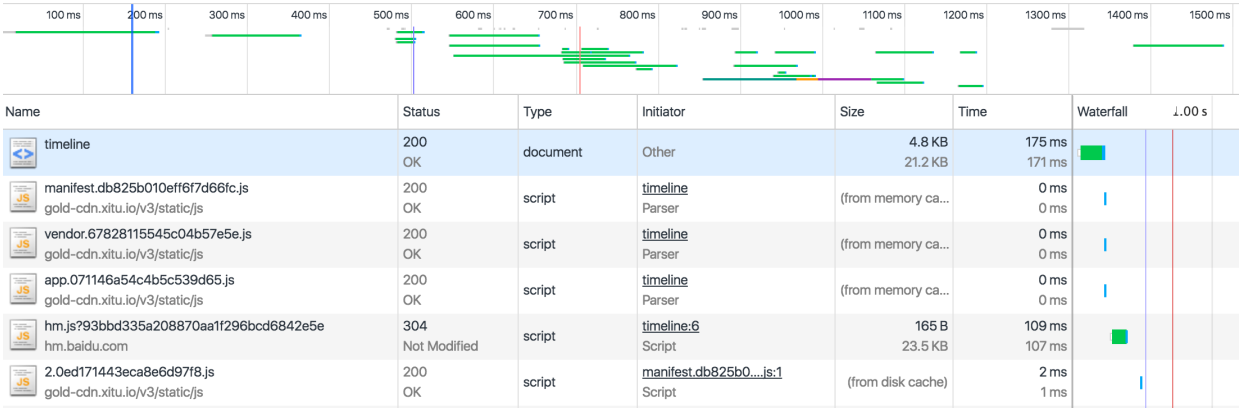
查看相关资源信息。

服务器端渲染定义

服务器端渲染 Server-Side Rendering 也常被人写作 SSR 是 Vue.js 2.0 版本发布的时候同时推出的功能，Virtual-DOM 的实现让 DOM 生成被 JavaScript 来描述也就给了前后端在 JavaScript 环境下都可以生成页面内容，而其最主要的业务 `vue-server-renderer` 也包含在了 `Vue.js` 核心库中。

Why SSR

1. SEO：除了 Google 和 Bing 比较完美地实现了对于 SPA（Single-Page Application）的爬虫渲染及内容抓取，大多数搜索引擎包括百度都没有支持。因而，包含丰富内容的产品并需要 SEO 流量的产品也就自然需要 SSR 实现。
2. 加载速度：Faster Time-to-Content



当网站是 SPA 时，渲染网站需要：第一次加载 HTML、加载 JavaScript、加载相应页面需要的 Vue Components（为了性能大多数组建都是异步取的）、渲染 DOM、加载数据、渲染 DOM、展示 … 而后端提前渲染好页面，可以快速展示到页面内容（其实加载数据部分的时间成本是省不下来的），因而还包含了使用缓存、renderToStream 等等优化速度的方法。

How SSR

由于 SSR 本身是 `Vue 2.+` 原生支持的功能，因而使用官方的文档来实现是最好的方式，当然也有其他的各种优化的方式：

1. 官方文档 & 教程
2. Nuxt.js 是一个参考了 React.js 栈下的 Next.js 的一个更高封装接口的 SSR 框架。它封装了一层和 `vue-router`、`vuex` 及 `webpack` 设置这一层需求，可以快速开发 SSR.

What SSR

由于掘金在这次重构优化当中，除了本身代码质量的提升外，SSR 实现主要是基于已经实现的 SPA 业务来分割文章等静态包含内容的页面的。因而，我们的整个网站是由 Vue 搭建的，因而在全站实现一个完整的后端渲染中有比较复杂的设置及文件结构。

在实现 SSR 的过程中，有如下几个部分是特别重要的，我在后文中会讲的比较清楚：

1. 如何设置开启 SSR.
2. 如何用尽可能相似的代码同时支持 Client 及 Server，不同的地方是什么。
3. 前后端拉取数据的异同。
4. 如果使用 `vue-loader` 的配置文件如何定义。

文件结构

```
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
```

上文是官方文件推荐的文件结构。

App.vue

包含了 Vue 项目的根目录及定义核心的 Vue 业务逻辑及需求，也绑定了相应的 Vue 依赖。

app.js

Universal Code，也就是前后端都会同步依赖的 Vue 定义，例如 `vue-router` 、 `vuex` 。这部分的业务及内部援引的其他组建会被前后端共同使用。

> 特别⚠
>
> 在后段渲染中，其实现模式与前端不同，如果每一个请求都生成一个新的 root vue instance，它都会存在在内存中并被其他的请求使用，进而产生错误。
>
> 因而我们应该将需求包裹在一个 Factory 函数里让每一次生成的 vue instance 都是全新的（这个问题同样要应用在 `vue-router` 和 `vuex` 等会不同请求调用的需求上）

entry-client.js

前端业务，依赖着 **app.js**.

entry-server.js

后端业务，依赖着 **app.js**.

components/*.vue

各种 Vue 组建、页面等。

我这里只包含了与 SSR 相关的掘金文件结构：

```
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
```


与官方给出的文件结构有雷同，单掘金因为本身的业务较为复杂因而文件结构也相对复杂一些。其中 `app.js` 我们这里用的是 `main.js`。而另外，配合不同的 production 需求是有不同的 NPM Scripts 来管理的。如下：

```
{
  ...
  "scripts": {
    ...
    "build:client": "cross-env NODE_ENV=production VUE_ENV=client
webpack --config build/webpack.client.conf.js --progress --hide-
modules",
    "build:server": "cross-env NODE_ENV=production VUE_ENV=server
webpack --config build/webpack.server.conf.js --progress --hide-
modules",
    ...
  },
  ...
}
```

其中与 build 相关的就是 `build:client` 和 `build:server` 脚本。其中与 SSR 相关的 `build:server` 业务调用流程如下：

1. `webpack.server.config.js` 被调用来 bundle 代码，引入第三方库，设置 entry 生成 `server-bundle.js` 文件。

2. 调用 `server-entry.js` 引用基础的 universal code 部分的 `main.js` 后端有一个 root Vue object，并根据具体情况实现是否后端渲染。在掘金里，我们有一个设定就是如果用户未登录，内容页面基本上都是静态渲染的。

3. `server.js` 运行起来并满足后端渲染条件时就会调用到 `renderer.js` 里面包含了具体的 `renderToStream` 后端生成内容的业务，当 SSR 触发时会被调用。

设置文件

核心的配置文件也就是 `renderer.js` ，里面 expose 了两个主要函数：

1. `setup(app)` 主要设置在 dev 还是 prod 下不同。prod 下会载入之前通过 Bundle Renderer 生成的 `server-bundle.js` bundle 文件来生成最终的 `renderer` 对象。

2. `render(req, res)` 具体的后端渲染逻辑。

在这里我们使用了简单的缓存 cache，具体逻辑如下

```
function cache (info) {
  return pageCache.setCache(info)
  .catch(err => {
    console.error('cache error:', err)
  })
}
```


而 render 函数里调用如下：

```
function render (req, res) {
  if (!renderer) {
    return res.end('Waiting for compilation, please refresh in a
moment.')
  }

  const user = req.currentUser
  const context = { url: req.url, code: 200, user }
  const renderStream = renderer.renderToStream(context)

  let shouldContinue = true
  let cachedHtml = ''

  renderStream.once('data', () => {
    if ([301, 302].indexOf(context.code) !== -1 &&
context.location) {
      shouldContinue = false
      if (!user) {
        cache({
```

```
          url: req.originalUrl,
          code: context.code,
          location: context.location,
          expire: 1000 * 60
        })
      }
      return res.redirect(context.code, context.location)
    }

    const { meta, title, link, seo } = context.meta.inject()
    const html = [
      htmlTemplate[0],
      meta.text(),
      title.text(),
      link.text(),
      htmlTemplate[1]
    ].join('')
    cachedHtml = cachedHtml + html

    res.setHeader('Content-Type', 'text/html')
    res.status(context.code).write(html)
  })

  renderStream.on('data', chunk => {
    if (!shouldContinue) { return }
    cachedHtml = cachedHtml + chunk
    res.write(chunk)
  })

  renderStream.on('end', () => {
    if (!shouldContinue) { return }
    const initialState = encodeURIComponent(context.initialState)
    const html = '...'
    cachedHtml = cachedHtml + html
    if (!user && context.code === 200) {
      cache({
        url: req.originalUrl,
        html: cachedHtml
      })
    }
    res.end(html)
  })

  renderStream.on('error', err => {
    console.error(`error during render: ${req.url}`)
    console.error(err)
    if (!shouldContinue) { return }
    res.status(500).end('Internal Error.')
  })
}
```

这里 cache 存了大概 60 秒并存在 redis 里，dev 状态下不启用 cache。

一些小心得

> 此部分由掘金团队的阿德提供。

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。

404 页面

当一个资源找不到时，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。

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：


```
{
  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
}
```

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
    })
  }
```

在顶层组件里，我们跳出了路由的盒子以实现不改变 URL 显示 404 页面：

```
component.error-view(v-if="errorView", :is="errorView")
router-view(v-if="!errorView")
```

但 SSR 很贵

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 welcome 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。

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。

数据请求也很贵

这里的数据请求特指对 LeanCloud 的请求。我们的前端服务器在青云上，直接在青云服务器上请求 LeanCloud 的代价相当大，所以我们会尽量把请求放到前端，让浏览器直接请求 LeanCloud ，对于服务器端渲染所必需的请求，则需要用缓存来提升性能，而这些请求是由 LeanCloud SDK 发出的，这带来的问题就是如何让它们走缓存这条路径，有两个解决方案，一是修改所有调用代码，二是劫持底层方法，我们毫无疑问地选择了第二个，劫持了 `http` 和 `https` 模块的 `request` 方法（ `cache/leancloud.js` ），将 LeanCloud 的数据请求转换为对缓存接口的请求。加上缓存之后，一个 LeanCLoud 的数据请求也至少要耗费三百多毫秒的时间。数据源请求性能不行是个比较悲催的问题，这不仅限制了设计，而且在人们看不到的地方也要做出一些努力才能达到勉强可接受的水平。

最后

以上就是掘金 SSR 实践中的一些心得体会，大家在使用掘金过程中遇到了相应的功能会体验到。更多的问题可以搜索哦！

比心
Ming