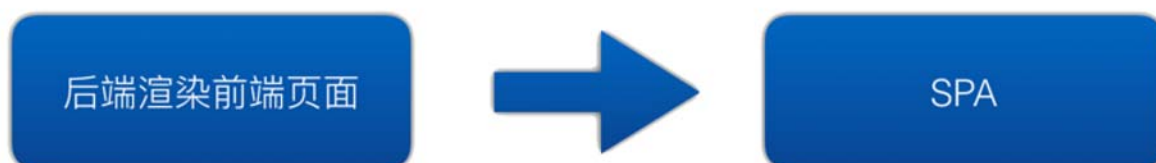


Node 在沪江的大规模实践

引入Node之前

在 Node 之前，沪江的业务经历了如下历程：



在 SPA 之前，页面由前端和后端共同维护，这样有以下弊端：

- 前后端共同维护相同的页面，维护成本高
- 不少页面和资源都在后端服务器，不便设置合理的缓存策略，影响性能

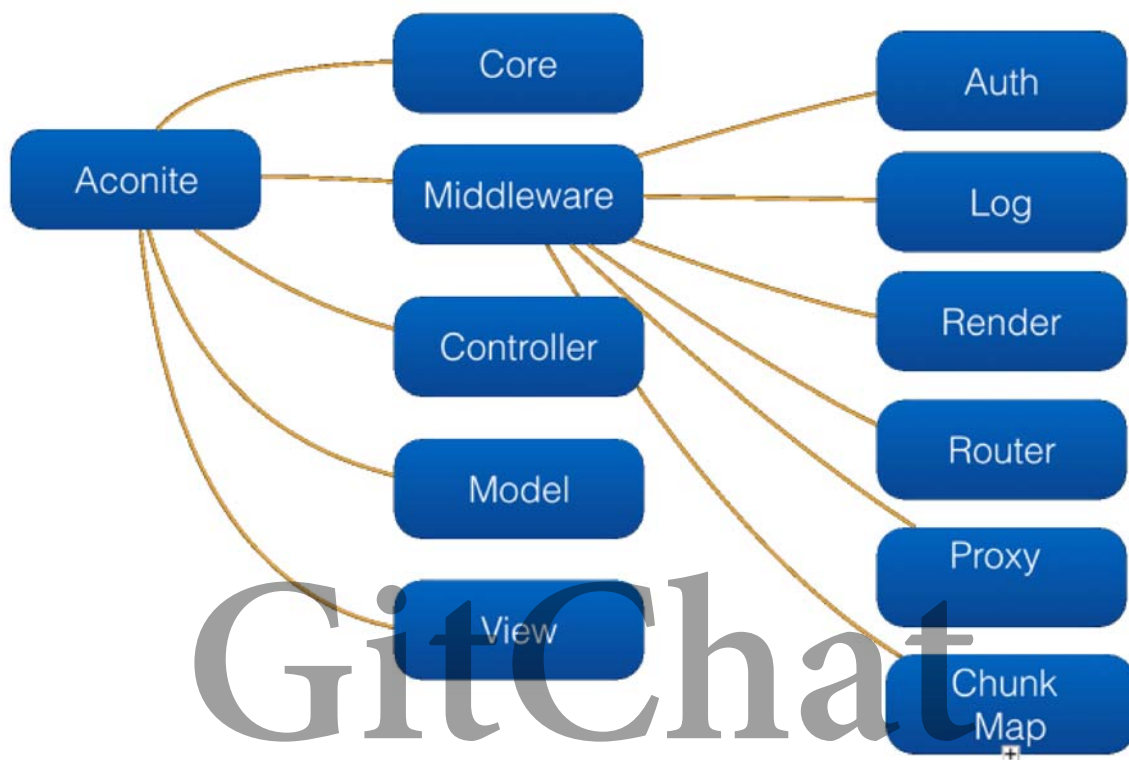
在移动端流量大幅增大，并且超过 PC 流量之后，SEO 对于流量的影响变得不是那么重要，在这样的背景下，不少的企业都选择了SPA的方式在客户端渲染页面。这样大量的渲染逻辑从后端页面移到了浏览器端。前后端有了第一次分离。

在这个过程中，我们做了如下的事情：

- 搭建静态资源服务器，将脚本等页面资源发布到静态资源服务器，前后端发布实现了分离
- 采用强缓存策略优化静态资源性能，采用版本号的方式避免静态资源更新中的缓存

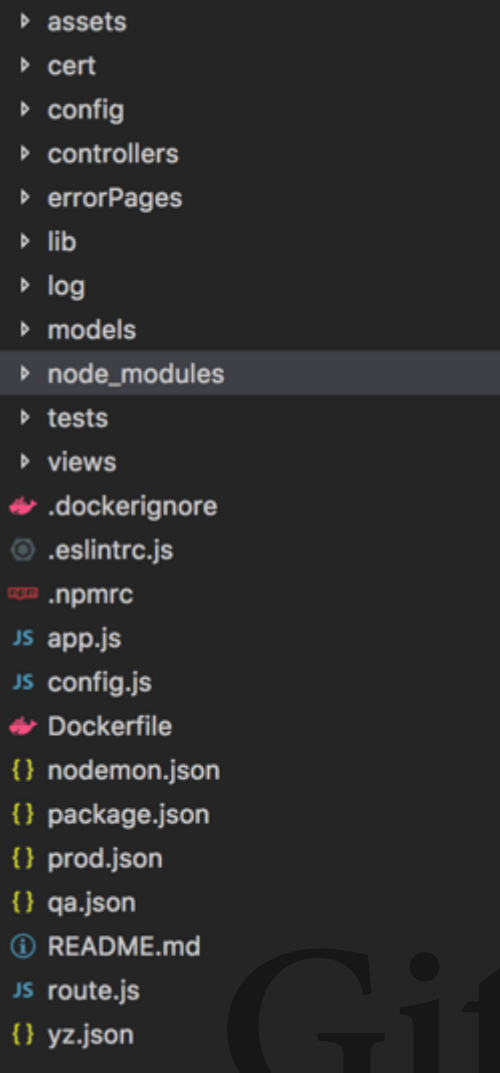
Node框架

由于前端开发，普遍没有后端开发的一些思维和经验，因此在引入 Node 之初，我们需要搭建一个框架来降低接入 Node 的成本。框架选用了业界流行的 Koa。我们基于 Koa 开发了一套 MVC 的框架。框架的结构如下：



框架做了如下事情：

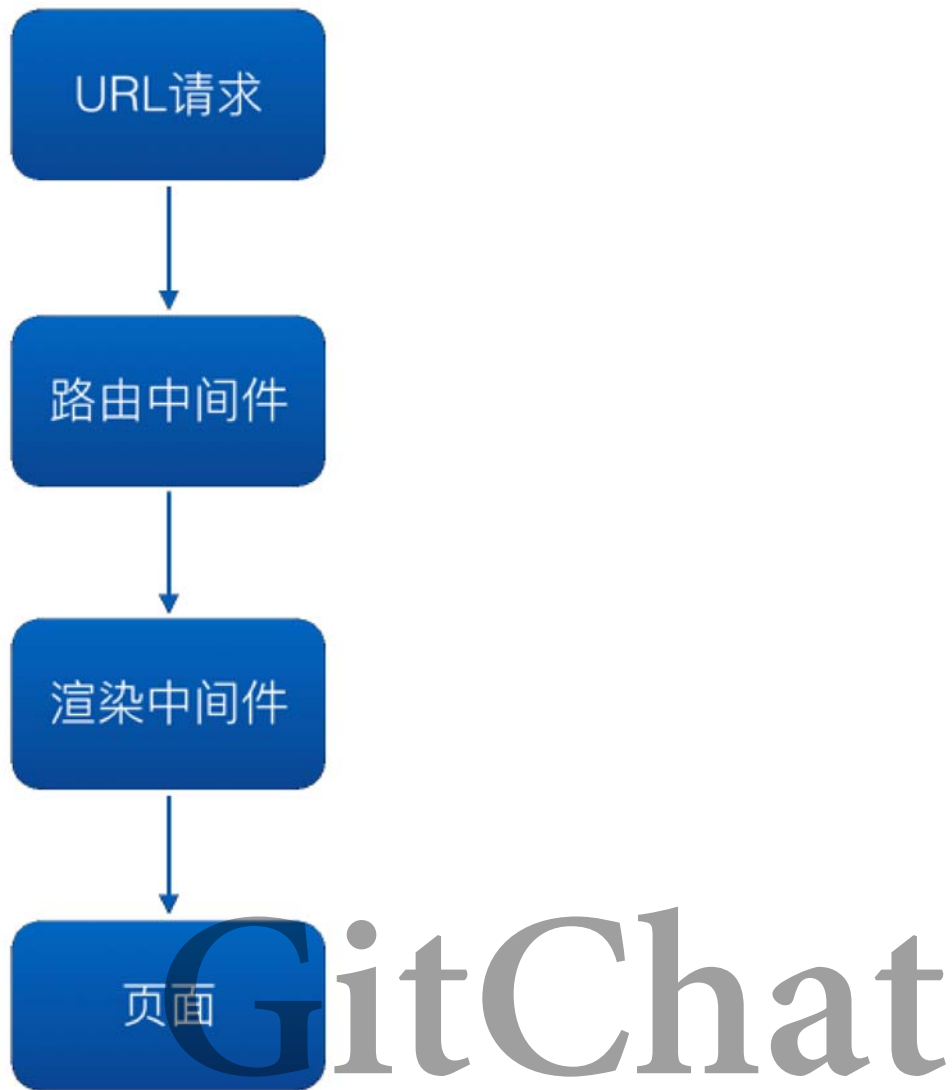
- 选择 Koa2 框架，选择了一些常用的第三方中间件，如：koa-router、body-parser、etag 等
- 封装封装路由和渲染组件，实现 MVC 结构



- assets
- cert
- config
- controllers
- errorPages
- lib
- log
- models
- node_modules
- tests
- views
- 🔥 .dockerignore
- ⦿ .eslintrc.js
- 📄 .npmrc
- JS app.js
- JS config.js
- 🔥 Dockerfile
- { } nodemon.json
- { } package.json
- { } prod.json
- { } qa.json
- 📄 README.md
- JS route.js
- { } yz.json

MVC 分别对应了上面的 models、views、controllers 目录。

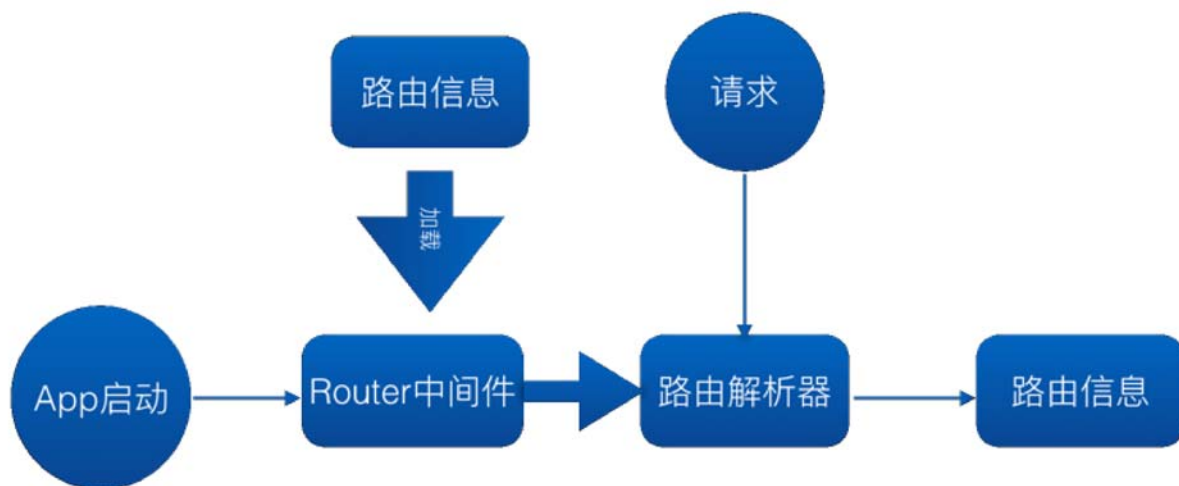
通过中间件，我们的渲染流程如下：



可以看到，渲染流程主要分为两部分：

- 采用路由中间件根据页面请求信息，得到路由信息
- 根据路由信息，加载对应的 controller，得到渲染数据，调用模版，渲染页面

路由中间件



在应用启动的时候，该中间件会加载路由信息到 koa-router 中，当接收页面请求时，会采用 koa-router 得到路由信息，将这些信息挂载到请求的 Context 对象上，以便后续的渲染中间件使用。路由信息包含渲染的 controller 信息，和页面跳转信息。

渲染中间件

在路由中间件得到的路由信息被挂载在请求的 Context 对象上，在渲染中间件需要根据路由信息做出处理，路由数据中存在页面跳转的，跳转到对应的页面。否则，按照下面的流程处理：



- 执行 action 得到渲染数据，在 action 中通过挂载在 Context 上的 render 方法渲染页面

规范、质量

在向团队推广的时候，需要保证质量，完善的文档以及低的接入成本。我们通过以下途径来保障。

文档

完整、准确的文档，可以大幅降低接入成本。我们通过一下方式来保障：

- 通过 JSDoc 生成文档，保证了修改代码的时候，就会更新文档
- 提供框架门户网站，提供统一文档和各种资源入口
- 框架的文档，通过网站的方式提供，中间件的文档，直接发布到 NPM 库，在 NPM 包中可以直接得到文档

单测

通过单测来提升代码的质量，主要采用如下方式：

- 采用 mocha 框架对中间件和框架做单元测试
- 采用 nyc 来报告单元测试覆盖率

在中间件的开发和维护的之后，通过单元测试，保障了代码的正确性，避免修改一些 bug 引起新的 bug。

中间件和框架都通过 NPM 包的方式发布，包的版本严格遵循 semver 规范。对于重大

- 如果不定义边界，可能造成后端团队和前端团队分工不明确的问题

为了规避这些风险，我们在 Node 项目推进过程中，会适时调整合适的边界，再推进的初期，我们给出了如下的边界：

- 严禁操作 DB
- 严禁使用缓存
- 严禁使用 Session
- Node只做渲染，可以做一些无业务依赖的API接口拼装，不做有彼此依赖的API拼装，因为这样会将具体的业务引入到Node项目中

有了这些约定，首先，可以降低由于前端开发没有后端经验带来的风险，另外，也将前端团队和后端团队的职责分开，避免职责不清的问题。随着Node项目在各产线的推进，我们放开了部分边界，比如使用缓存和接口拼装上。定义这样的边界，主要目的是让专人干专事。

发布、部署

首先，我们需要统一开发，测试，验证，线上环境的Node版本，避免运行环境的差异造成服务的不稳定。这里采用了 PM2 来处理守护 Node 服务和启用 Cluster 的利用服务器的多 CPU 核心。

我们的构建，发布流程如下图所示：



我们必须保证CI上的服务器环境、Node版本和线上一致。另外，NPM 包应当在构建服务器安装，而不应当在部署的时候安装，这样可以避免在安装NPM包的时候，可能会存在不同环境下的包的版本不一致的情况。当然，我们也推荐锁死包的版本。

对于不同环境可能会有不同的配置，比如调用后端的API接口在不同环境不同。我们可以通过环境变量的方式，将这些配置区分。例如

```
let API = 'http://test.com/a'
const env = process.env

if(env === 'qa'){
  API = 'http://qa.test.com/a'
}else if(env === 'yz'){
  API = 'http://yz.test.com/a'
}
module.exports = {
  API
}
```

这样，在不同环境下的代码都是一致的，不存在说在发布到了QA环境之后，后续的环境还需要修改代码之后再发布，保证代码在提交QA之后的纯净性，避免通过测试后，因为修改环境配置带来的未测试的代码上线风险。

日志

GitChat

由于Node项目是后端服务，和其他后端服务一样，我们需要通过日志的方式记录程序的行为。通过这些行为帮助开发团队评估线上的服务是否正常运行，同时也可以利用这些日志弄清楚服务产生异常行为的细节，以便开发团队能快速定位分析问题，修复问题。

在Node端，我们采用 `log4js` 来记录日志，定义了两类日志格式：

- 应用日志：按照级别记录程序的行为，此处的级别采用 `log4js` 中的那些级别


```

"s_name": "192.168.1.100",
"proj": "192.168.1.100",
"s_ip": "192.168.1.100",
"method": "GET",
"host": "192.168.1.100",
"path": "http://192.168.1.100/",
"query_string": "{}",
"c_ip": "192.168.1.100",
"user_id": 0,
"useragent": "YisouSpider",
"referer": "",
"status": 200,
"c_bytes": 0,
"s_bytes": 50358,
"time_taken": 1461,
"time_third_api": 0,
"request_id": "7b2acc0d56104b0db5f28f922ea1aa75",
"@version": "1",
"@timestamp": "2016-08-08 10:10:10",
"type": "node-web"

```

访问日志中主要包含UA，请求的request信息，以及请求的状态码，响应时间等信息

应用日志

应用日志是应用记录自身的行为，包含了的字段内容如下：

```

"s_name": "192.168.1.100",
"level": "warn",
"message": "WARN: APT | 192.168.1.100 | 192.168.1.100 | 192.168.1.100 |"

```

应用日志主要包含日志级别和日志消息内容。日志级别分为：trace、debug、info、warn、error、fatal。这些日志从左到右逐级增大。

通过日志中间件统一记录日志，这样日志格式和接入方式被统一，便于统一将日志输出到 ELK 系统中。后续也可以借助ELK来查看，分析日志。日志中间件也可以支持输出指定级别的日志，可以用于如下场景：

在业务新上线的时候，可以记录较低级别的日志（如 `debug`），这样在 ELK 系统中，可以查看到应用的明细日志，便于发现问题和分析问题。当服务稳定了之后，可以提高记录日志的级别（如 `warn`），从而减小记录日志的量，降低 ELK 的负载压力。

另外，记录了 ELK 日志之后，我们也应当去关注这些日志。我们可以拉取 ELK 的数据，生成各业务线的 Node 服务健康报表，促使各业务线改进提升线上服务质量。

监控

通过日志可以记录服务的运行行为，但服务的实时运行情况，需要通过登录到服务器上去查看，这非常不方便。为了更方便的查看服务的运行状况，我们开发了监控系统。沪江的监控系统基于 grafana + influxdb。Node 端通过 metrics 收集数据。

metrics 数据收集服务做了以下事情：

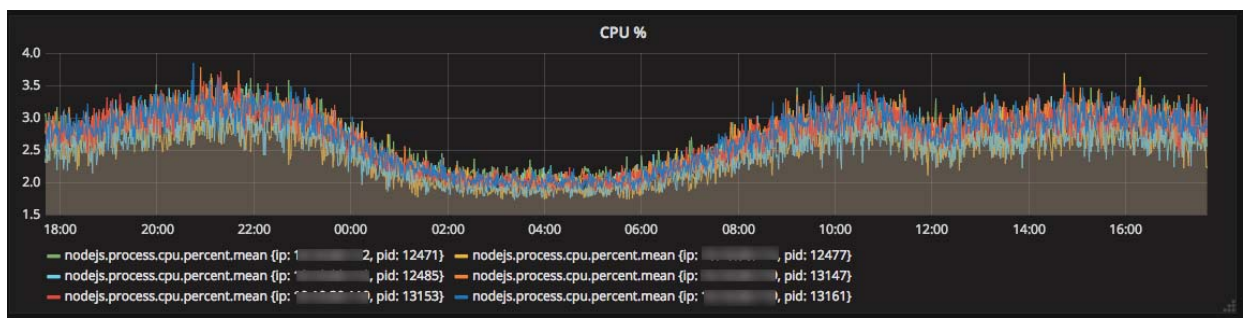
- 提供 histogram, counter 等 metric, 在本地做一些统计分析, 比如通过 counter 计数, 通过 histogram 来计算样本的 P99、P999、mean、max、min、qps 等数据。这些数据都在 Node 服务本地计算, 不必在监控服务中处理。
- 定期采集数据, 沪江采用每秒一次的频次
- 通过 kafka 消息队列定期上报数据

- 定期采集数据，沪江采用每秒一次的频次
- 通过 kafka 消息队列定期上报数据

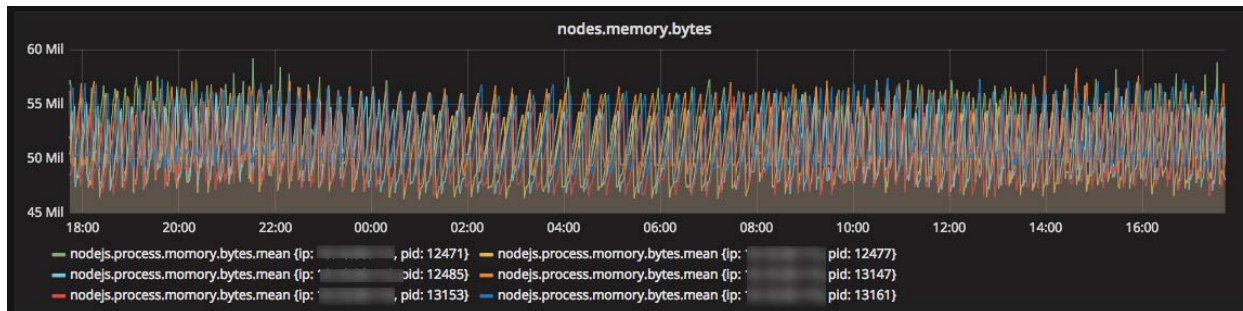
- 通过 kafka 消息队列定期上报数据

收集了两类数据：

NI 的性价比和半导体的



某服务内存监控



通过图表的方式，可以查看服务器的资源使用情况，从而来预估服务器需求。另外在 grafana 系统中，还可以设置预警，当出发报警时，给相关人通过邮件、微信等方式发送预警信息。

Node 服务的性能数据

对于服务本身，尤其是作为渲染页面的 Node 服务，我们关注的是服务器的处理能力，这里可以统计服务处理页面的处理时间、QPS 等信息。另外，对于 Node 服务调用的 API 的性能，我们也需要监控记录。这样当服务出现了性能瓶颈时，可以通过监控数据分析出主要的瓶颈在哪里，实在 Node 服务本身还是说在 API 服务。然后针对性做性能优化。

某服务某页面响应时间 P99



针对请求响应的延迟时间，统一也可以做出预警规则。

由于计算请求的时间和 QPS 等信息，要么给出统计 API，然后在应用中处理请求的时候，主动调用 API 的方式，来记录性能信息。但这样，对于业务代码具有侵入性。接入成本比较高。这里参考了 PM2 的监控代码，直接通过复写 Node 的 http 模块中的相关方法，来记录服务器端处理请求的性能，以及 Node 服务发起的 API 请求的性能。

经验

这里分享一些 Node 推进过程中的一些问题和经验。

压测

在 Node 压测的时候，我们碰到了一些坑，这里主要列出一些：

- 压测的时候，CPU 和内存负载未满载，但吞吐量上不去，检查 TCP 连接数，发现 TCP 连接爆了，这个一般见于直接连接 Node 服务做压测。一般采用在 Node 服务前面部署一个 Nginx 服务做连接复用来解决这个问题。
- 不在同一网络环境下压测，CPU、内存负载未满载，但服务吞吐量上不去，检查带宽限制，压测的时候，可以压测一个小页面，查看服务器的负载是否能上去。如果能上去，则说明带宽限制。
- Node 服务中，并行发起多个 API 接口，发现应用的吞吐量不高，API 接口负载未满载。可能的原因之一是 Node 应用和 API 服务的连接瓶颈造成。可以采用 Node 请求 API 中的 `forever` 参数来启用 `keep alive`，从而提升连接的复用率，降低 API 服务和 Node 应用的连接开销。

项目推进

为了降低沟通成本，并且保障服务的运行质量，我们制定了一些措施。