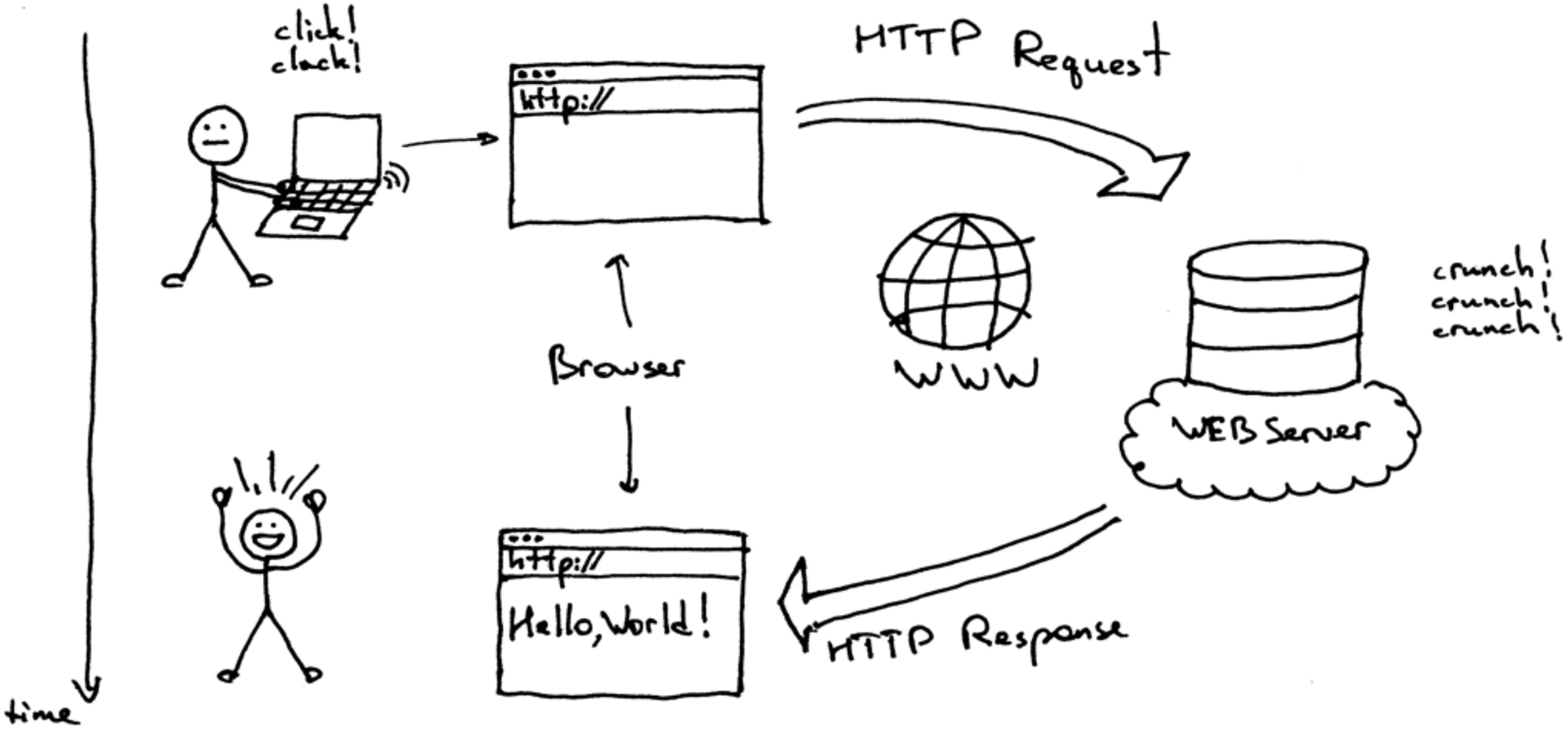
百万级应用的服务端渲染实践

Request To Server
&&
Response To Client



服务器响应

静态 HTML

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Document</title>
  <link rel="stylesheet" href="/path/to/stylesheets" />
</head>
<body>
  <p>Hello world</p>
  <script src="/path/to/script"></script>
</body>
</html>
```

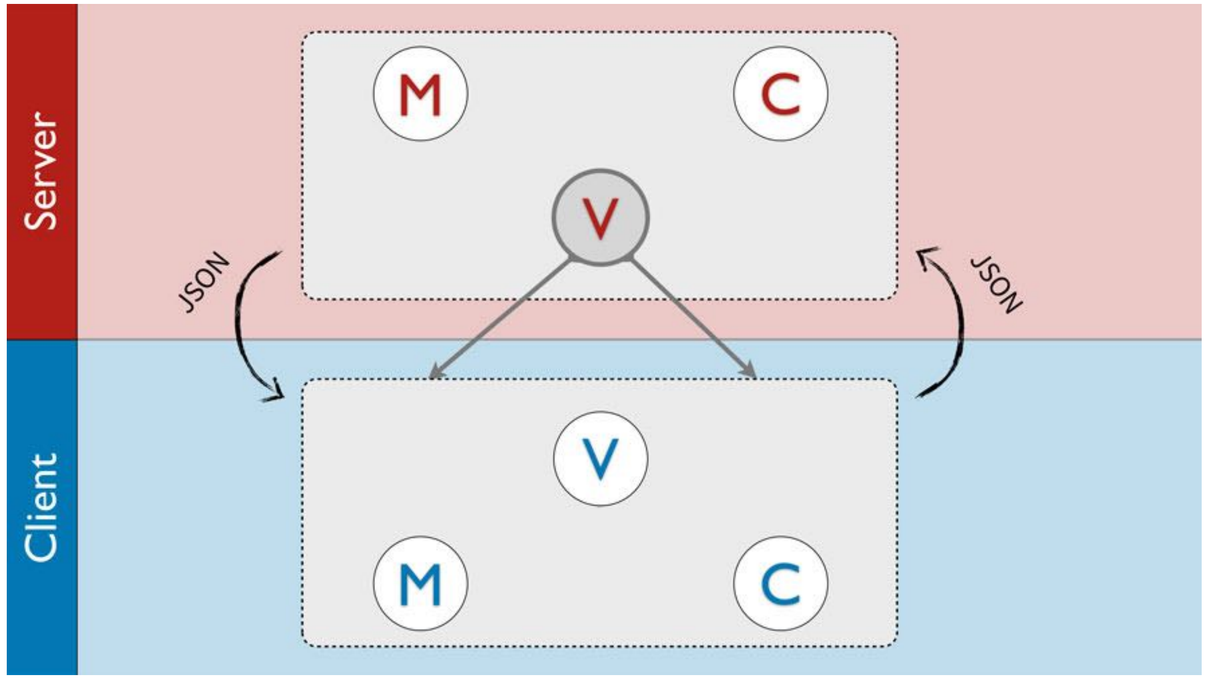
动态模版

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Document</title>
  <link rel="stylesheet" href="/path/to/stylesheets" />
</head>
<body>
  <?php echo '<p>Hello world</p>'; ?>
  <script src="/path/to/script"></script>
</body>
</html>
```

过分依赖后端模版带来的问题

- 逻辑过分集中于后端，MVC 难以分离
- 前端开发需要服务端环境，沟通成本加大
- 前后端没有清晰界限，职责难以明确

前后端分离



- 后端专注于数据与业务逻辑
- 前端专注于数据获取与渲染

前后端分离带来的问题

- 很多首屏能够渲染的逻辑最终必须得客户端执行
- SEO 不足，导致网站权重下降
- 前端逻辑容易过分操作 DOM ，影响开发体验

前端组件化的时代的降临

服务端返回 HTML

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Document</title>
</head>
<body>
  <div id="main"></div>
  <script src="./react.bundle.js"></script>
</body>
</html>
```

客户端运行的 JS

```gitignore
import React from 'react'
import ReactDOM from 'react-dom'

import App from './App'

ReactDOM.render(
  <App />,
  document.getElementById('main')
)
```

- 渲染逻辑只能在客户端进行
- 不利于 SEO
- 相对渲染的速度慢

新的时代
依托于 node 的前后端分离

- Express， koa 等新时代 node 框架，越来越多的首屏逻辑可以通过 jade， ejs 等模版引擎进行渲染

```
app.set('view engine', 'pug')
app.set('views', __dirname + '/views')

app.get('/', (req, res) => {
  res.render('index', {
    info: 'Hello world',
  })
})
```

```
p= info
```

- node 可以对后端 API 进行再封装，将部分业务逻辑在 node 端完成

```
app.get('/comments', (req, res) => {
  fetchIdListByUser(userid)
    .then(handleResponse)
    .then(idList => Promise.all(idList.map(id => fetchCommentsById(id))))
    .then(comments => {
      res.json({
        comments,
      })
    })
    .catch(error => console.error(error.stack))
})
```

- Server + JS context 使得 React 和 Vue 的服务端渲染变得可能

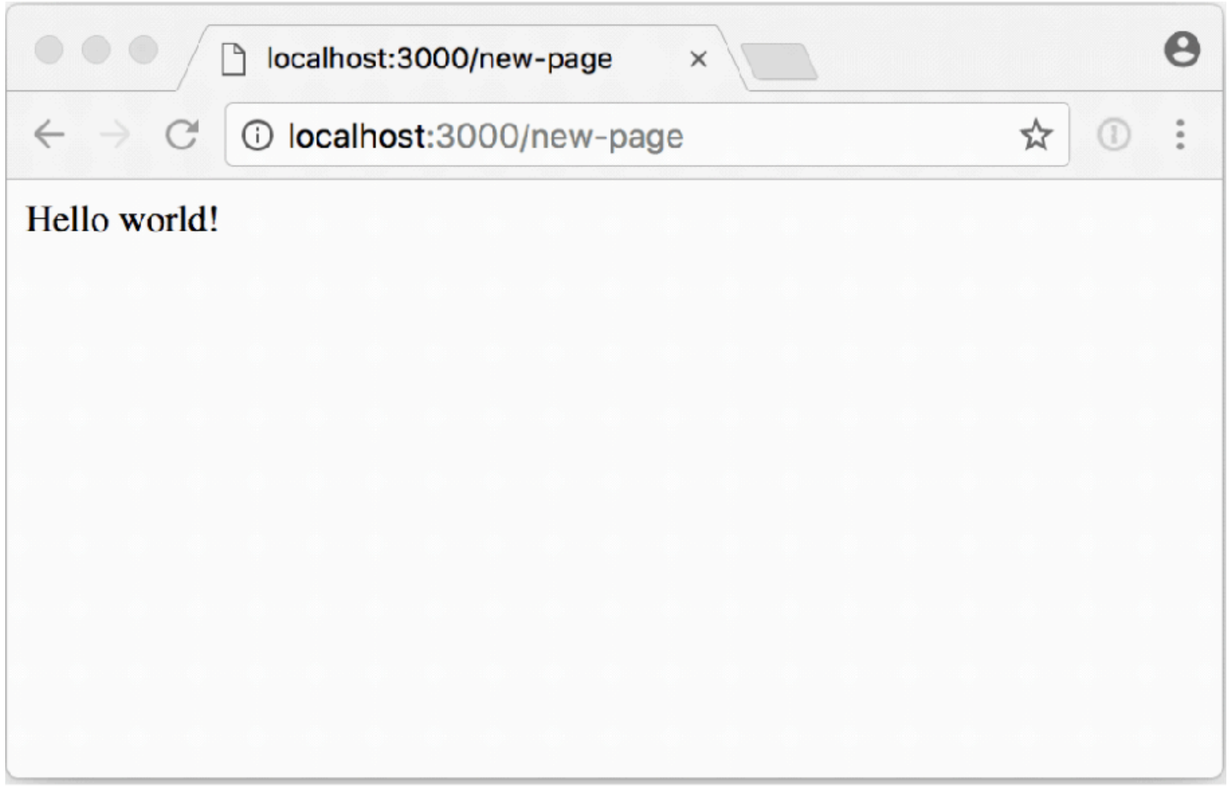
```
const ReactDOMServer = require('react-dom/server')

app.get('/', (req, res) => {
  return ReactDOMServer.renderToString(element)
})
```

react 的服务端渲染实践

	Next.js	Hypernova
star	16.6k	3.4k
open issues / issues	167/1648	12/37
contributors	243	13

```
export default() => {
  return (
    <div> Hello world! </div>
  )
}
```



优势

- 内置 webpack 和多个插件，快速上手
- 社区活跃，问题能及时得到关注

劣势

- 老代码迁移较困难，有一套新的“规范”
- webpack 的脚手架等配置容易冲突

使用 hypernova 进行服务端渲染

- 丰富的插件支持

1. Hypernova-react
2. Hypernova-ruby
3. Hypernova-node
4. …

- 最小化的损伤组件，只需要在入口组件进行包裹

```
import { renderReact } from 'hypernova-react'
import MyComponent from './component'

export default renderReact(
  'MyComponent.hypernova.js',
  MyComponent,
)
```

- 使用简单，发送 POST 请求，即可触发渲染

```
curl -X POST -d @payload.json http://localhost:3000/batch --header
"Content-Type:application/json"
```

与服务端模版配合

与其他客户端配合

hypernova-ruby

hypernova-ruby build passing

> A Ruby client for the Hypernova service

Getting Started

Add this line to your application's Gemfile:

```
gem 'hypernova'
```

And then execute:

```
$ bundle
```

Or install it yourself as:

```
$ gem install hypernova
```

hypernova-node

hypernova-client

> A node client for sending requests to Hypernova.

class Renderer

Renderer.prototype.addPlugin

> (plugin: HypernovaPlugin)

Adds a plugin to the renderer.

Renderer.prototype.render

> (data: Jobs): Promise

Sends a request to Hypernova for the provided payload and returns a promise which will fulfill with the HTML string you can pass down to the client.

使用 hypernova-ruby 作为 hypernova client 发送请求

```
<%=
  render_react_component(
    'MyComponent.js',
    :name => 'Person',
    :color => 'Blue',
    :shape => 'Triangle'
  )
%>
```

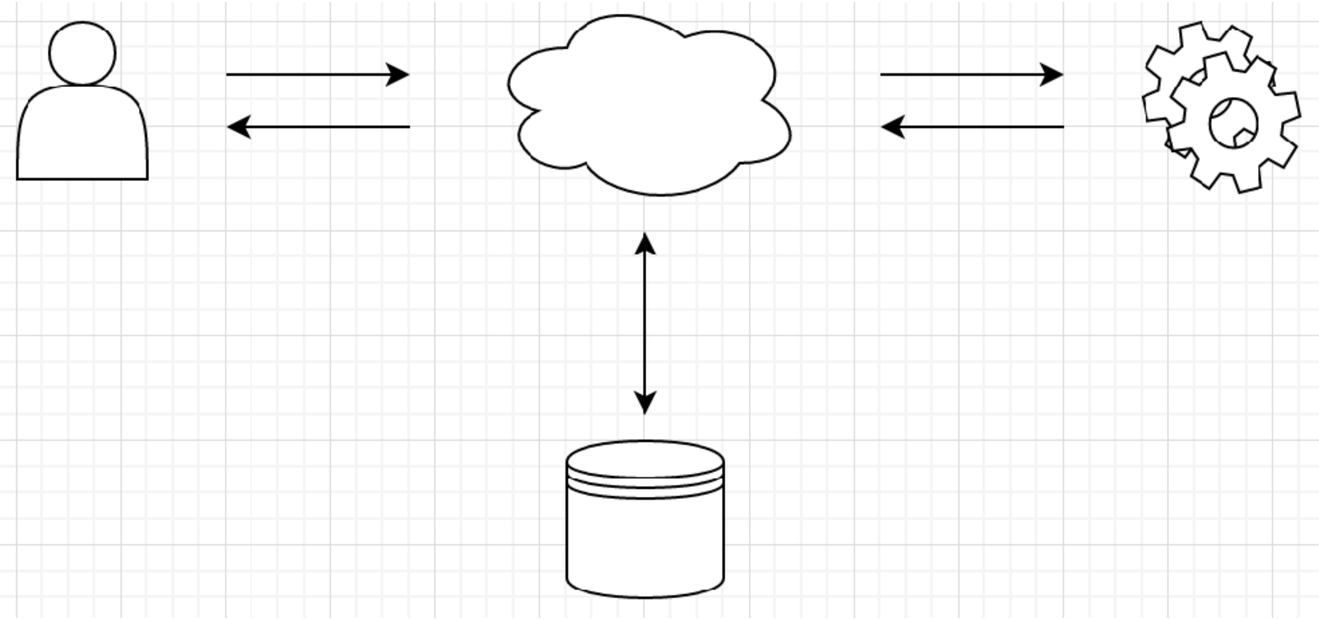
```ruby
batch = Hypernova::Batch.new(service)

# each job in a hypernova render batch is identified by a token
# this allows retrieval of unordered jobs
token = batch.render(
  :name => 'some_bundle.bundle.js',
  :data => {foo: 1, bar: 2}
)

# now we can submit the batch job and await its results
# this blocks, and takes a significant time in round trips, so try to only
# use it once per request!
result = batch.submit!

# ok now we can access our rendered strings.
foo1 = result[token].html_safe
```

hypernova 如何工作



1. 用户通过发送请求进行网站的访问
2. 服务端获取到当前页面的所有数据并准备进行渲染
3. 这个时候，hypernova client (ruby, node) 开始发送到 hypernova server 的请求进行请求渲染
4. hypernova server 进行渲染
5. 然后您的服务器将通过 hypernova 渲染的标记加入渲染的结果
6. 在客户端，JS 会继续进行客户端渲染来实现渐进增强

hypernova 技术实现细节

- 顶层 hypernova ， 通过不同环境执行不同的代码
  使得同一套代码能够跑在不同环境

```
export default function hypernova(runner) {
  return typeof window === 'undefined'
    ? runner.server()
    : runner.client();
}
```

- hypernova-react，进一步包裹组件，通过配置，在不同环境跑不同逻辑

```
import hypernova, { serialize, load } from 'hypernova';

export const renderReact = (name, component) => hypernova({
  server() {
    return (props) => ReactDOMServer.renderToString(React.createElement(component, props))
  },

  client() {
    // client render
  },
});
```

- 多种 hypernova-wrapper 配合，适合不同的业务场景

```
const renderFunction = (name, configureStore) => hypernova({
  server: () => props => serialize(name, configureStore.server(props)),
  client: () => {},
});
```

```
export const renderMorearty = (name, component, configureStore) => {
  return hypernova({
    server() {
      return (props) => {
        const wrappedComponent = configureStore.server(props);
        return ReactDOMServer.renderToString(React.createElement(wrappedComponent.bootstrap()));
      };
    },

    client() {
      return configureStore.client();
    },
  });
};
```

hypernova 的多种组件获取方式

1. 直接通过静态文件获取，会在 hypernova 内部 wrap 成为 commonJS 组件
2. 直接通过 require 函数，直接传入组件名称
3. 直接通过 promise fetch，获取远端组件

为什么 Strikingly 使用 hypernova

张钊 @loatheb

Strikingly，把互联网的力量带给所有人

我们的任务是让任何人拥有能力抒发自己的个性。

我们是由 Y Combinator 孵化的国际化团队，落户上海

并获得了 SV Angel, Index Ventures, FundersClub, 创新工场等的投资

THANK YOU