# node技术进阶与实战

马全华

美团点评前端技术专家

TABLE OF

# CONTENTS 大纲

- Web基础
- 应用场景
- 优化&运维

# Web基础

- http
- koa

# 故事的开始

```
const http = require('http')

http.createServer((req, res) => {
    res.end('hello node')
}).listen(3000)
```

```
const http = require('http')

http.createServer((req, res) => {
    res.end('hello node')
}).listen(3000)
```

# Code 即 Sever

**req:**

http.IncomingMessage?

Readable Stream

**res:**

http.ServerResponse

Writable Stream

## req:

- .url
- .method
- .headers
- event:data
- event:end

## res:

- .setHeader()
- .statusCode
- .write()
- .end()

# server

CODE

**request**  **response**

# 小结

- 9个常用属性/API
- 非常原始

# 从头开始搭Server

以一篇文章页面做为例子

/article?id=1234

```
const http = require('http')

http.createServer((req, res) => {
    let article = fetchArticleFromDB()

    res.end(`<html lang="en">
    <head>
        <meta charset="UTF-8">
        <meta name="viewport" content="width=device-width, initial-scale=1.0">
        <title>${article.title}</title>
    </head>
    <body>
        <div class="article">
            ${article.content}
        </div>
    </body>
    </html>`)

}).listen(3000)
```

# 缺点什么？

- 路由解析
- 参数解析
- cookie解析
- response header设置

# 路由解析

url: /article?id=123456



Controller

route.js

```
const Url = require('url')
const routes = {}

exports.handleRequest = (req, res) => {
    let url = Url.parse(req.url, true)

    if (url.pathname in routes) {
        routes[url.pathname](req, res)
            .then(content => res.end(content))
    } else {
        res.statusCode = 404
        res.end('not found')
    }
}

exports.get = (route, controller) => {
    routes[route] = controller
}
```

```
const http = require('http')
const router = require('./route')

router.get('/article', async (req) => {
    let article = fetchArticleFromDB()
    return `<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>${article.title}</title>
    </head>
    <body>
        <div class="article">
            ${article.content}
        </div>
    </body>
    </html>`
})

http.createServer((req, res) => {
    router.handleRequest(req, res)
}).listen(3000)
```

# 参数解析

url: /article?id=123456

query

body: form data

# query

```
exports.handleRequest = async (req, res) => {
    let url = Url.parse(req.url, true)
    req.query = url.query
    if (url.pathname in routes) {
        routes[url.pathname](req, res)
            .then(content => res.end(content))
    } else {
        res.statusCode = 404
        res.end('not found')
    }
}
```

```
router.get('/article', async (req) => {
    let article = fetchArticleFromDB(req.query.id)
    return `<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>${article.title}</title>
    </head>
    <body>
        <div class="article">
            ${article.content}
        </div>
    </body>
    </html>`
})
```

# Post的参数怎么处理?

# Readable Stream

event: data

event: end

remember?

# form.js

```
const qs = require('querystring')

exports.parseForm = (req) => {
    return new Promise((resolve, reject) => {
        let buffers = []
        req.on('data', chunk => buffers.push(chunk))
        req.on('end', () => {
            resolve(qs.parse(Buffer.concat(buffers).toString()))
        })
    })
}
```

route.js

```
exports.handleRequest = async (req, res) => {
    let url = Url.parse(req.url, true)
    req.query = url.query
    req.body = await form.parseForm(req)
    if (url.pathname in routes) {
        routes[url.pathname](req, res)
            .then(content => res.end(content))
    } else {
        res.statusCode = 404
        res.end('not found')
    }
}
```

# Cookie解析

header: cookie string

object

# cookie.js

```
const qs = require('querystring')

exports.parseCookie = (req) => qs.parse(req.headers['cookie'] || '', '; ')
```

# response header设置

- Content-Type
- Cache-Control
- Set-Cookie
- ...

res.setHeader()

```
exports.handleRequest = async (req, res) => {
    let url = Url.parse(req.url, true)
    req.query = url.query
    req.body = await form.parseForm(req)
    req.cookies = cookie.parseCookie(req)
    if (url.pathname in routes) {
        routes[url.pathname](req, res)
            .then(content => res.end(content))
    } else {
        res.statusCode = 404
        res.end('not found')
    }
}

exports.get = (route, controller) => {
    routes[route] = controller
}
```

```
router.get('/article', async (req, res) => {
    let article = fetchArticleFromDB(req.query.id)
    res.setHeader('content-type', 'text/html')
    return `<html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>${article.title}</title>
    </head>
    <body>
        <div class="article">
          ${req.query.id} : ${article.content}
        </div>
    </body>
    </html>`
})

http.createServer((req, res) => {
    router.handleRequest(req, res)
}).listen(3000)
```

# 静态资源

- 文件读取
- 缓存
- etag
- last modified
- …

# 小结

- 封装原始读取

# 思考题

如果实现文件上传

# koa详解

# koa

- Express　　callback
- koa1　　co & yield
- koa2　　async & await

# koa核心

- 中间件框架
- context

# 使用koa改写

```
const Koa = require('koa')
const router = require('koa-router')()
const app = new Koa()

router.get('/article', ctx => {
    let article = fetchArticleFromDB(ctx.query.id)
    ctx.body = `<!DOCTYPE html>
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>${article.title}</title>
    </head>
    <body>
        <div class="article">
            ${article.content}
        </div>
    </body>
    </html>`
})
app.use(router.routes())
app.listen(3000)
```

# koa主线

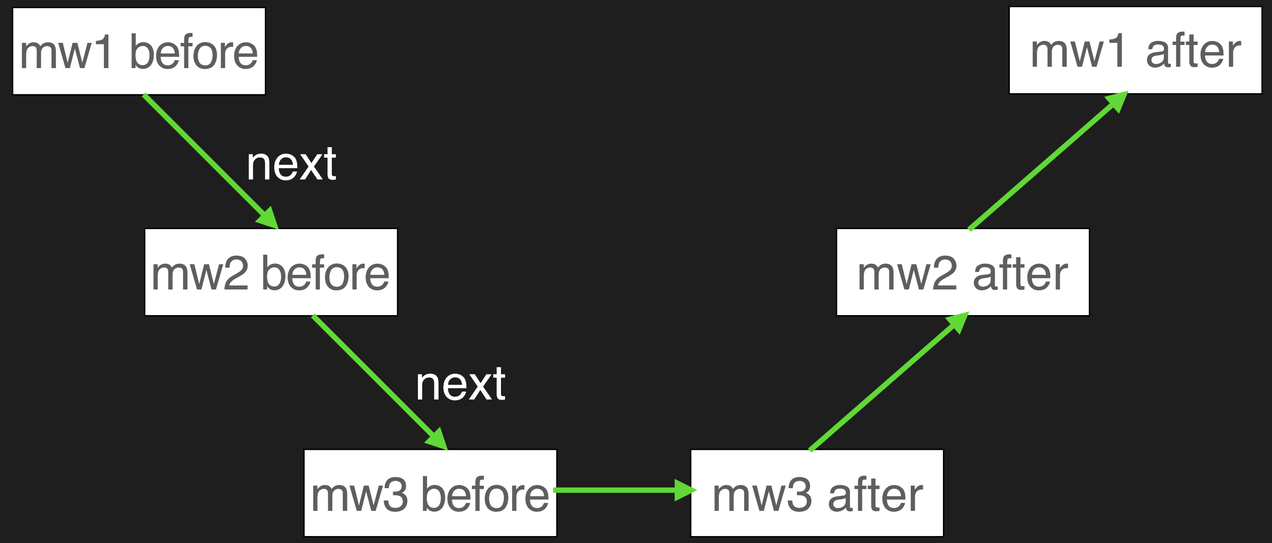
```
listen(...args) {
  const server = http.createServer(this.callback());
  return server.listen(...args);
}

callback() {
  const fn = compose(this.middleware);

  const handleRequest = (req, res) => {
    const ctx = this.createContext(req, res);
    return this.handleRequest(ctx, fn);
  };

  return handleRequest;
}

handleRequest(ctx, fnMiddleware) {
  const onerror = err => ctx.onerror(err);
  const handleResponse = () => respond(ctx);
  return fnMiddleware(ctx).then(handleResponse).catch(onerror);
}
```

compose = middleware => ctx => Promise

# koa主线

```
use(fn) {
  this.middleware.push(fn);
  return this;
}
```

```
async (ctx, next) => {
    // before

    await next()

    // after
}

app.use(middleware1)
app.use(middleware2)
app.use(middleware3)
```



# koa主线

```
listen(...args) {
  const server = http.createServer(this.callback());
  return server.listen(...args);
}

callback() {
  const fn = compose(this.middleware);

  const handleRequest = (req, res) => {
    const ctx = this.createContext(req, res);
    return this.handleRequest(ctx, fn);
  };

  return handleRequest;
}

handleRequest(ctx, fnMiddleware) {
  const onerror = err => ctx.onerror(err);
  const handleResponse = () => respond(ctx);
  return fnMiddleware(ctx).then(handleResponse).catch(onerror);
}
```

# middleware机制

```
function compose (middleware) {
  return function (context, next) {
    // last called middleware #
    let index = -1
    return dispatch(0)
    function dispatch (i) {
      if (i <= index) return Promise.reject(new Error('next() called multiple times'))
      index = i
      let fn = middleware[i]
      if (i === middleware.length) fn = next
      if (!fn) return Promise.resolve()
      try {
        return Promise.resolve(fn(context, dispatch.bind(null, i + 1)));
      } catch (err) {
        return Promise.reject(err)
      }
    }
  }
}
```

# koa主线

```
listen(...args) {
  const server = http.createServer(this.callback());
  return server.listen(...args);
}

callback() {
  const fn = compose(this.middleware);

  const handleRequest = (req, res) => {
    const ctx = this.createContext(req, res);
    return this.handleRequest(ctx, fn);
  };

  return handleRequest;
}

handleRequest(ctx, fnMiddleware) {
  const onerror = err => ctx.onerror(err);
  const handleResponse = () => respond(ctx);
  return fnMiddleware(ctx).then(handleResponse).catch(onerror);
}
```

# 中间件

- router
- bodyparser
- static file
- ...

# context

- request
- response

# createContext

```
createContext(req, res) {
  const context = Object.create(this.context);
  const request = context.request = Object.create(this.request);
  const response = context.response = Object.create(this.response);
  context.app = request.app = response.app = this;
  context.req = request.req = response.req = req;
  context.res = request.res = response.res = res;
  request.ctx = response.ctx = context;
  request.response = response;
  response.request = request;
  context.originalUrl = request.originalUrl = req.url;
  context.cookies = new Cookies(req, res, {
    keys: this.keys,
    secure: request.secure
  });
  request.ip = request.ips[0] || req.socket.remoteAddress || '';
  context.accept = request.accept = accepts(req);
  context.state = {};
  return context;
}
```

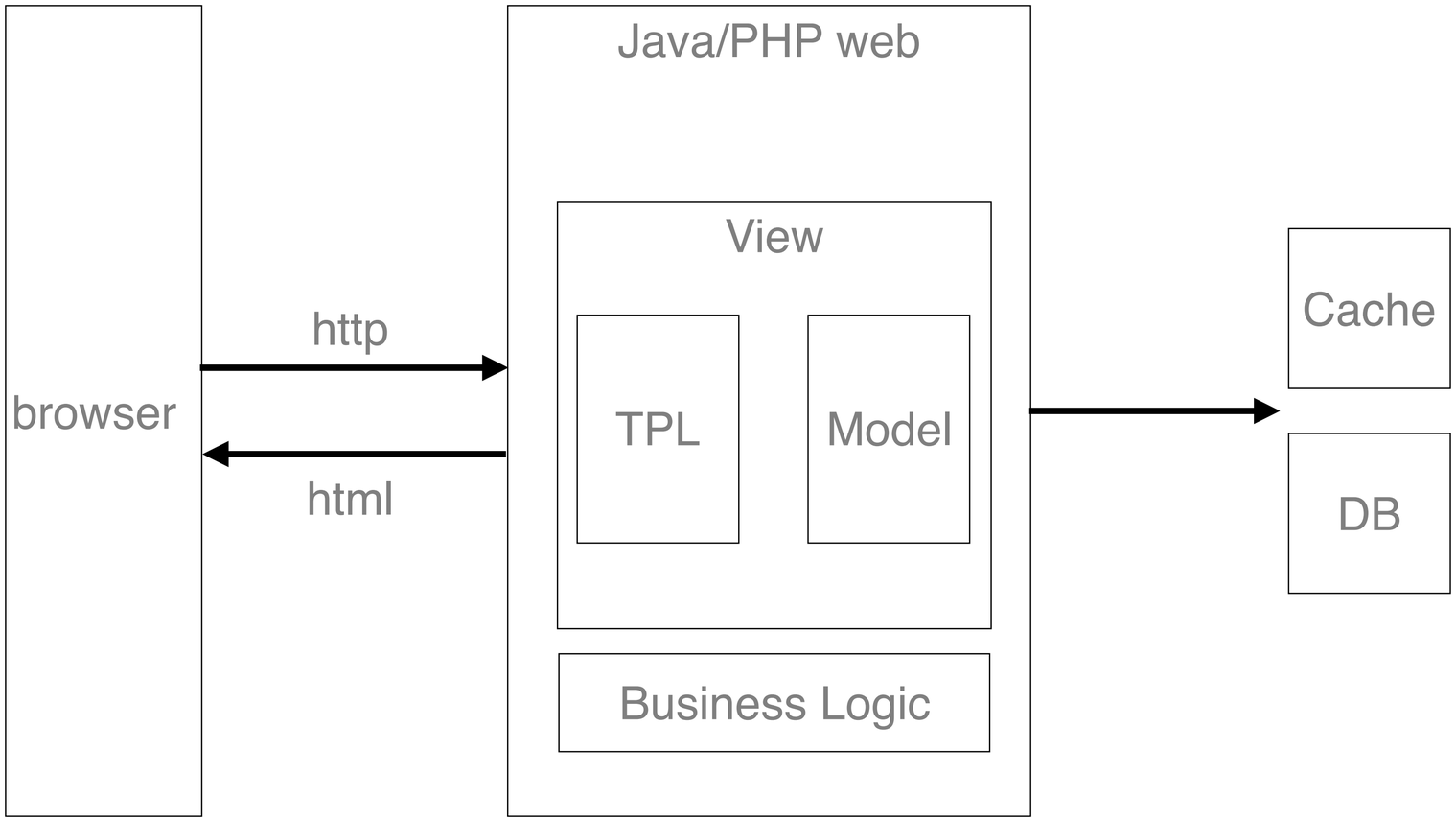
# 小结

- koa中间件机制
- context封装

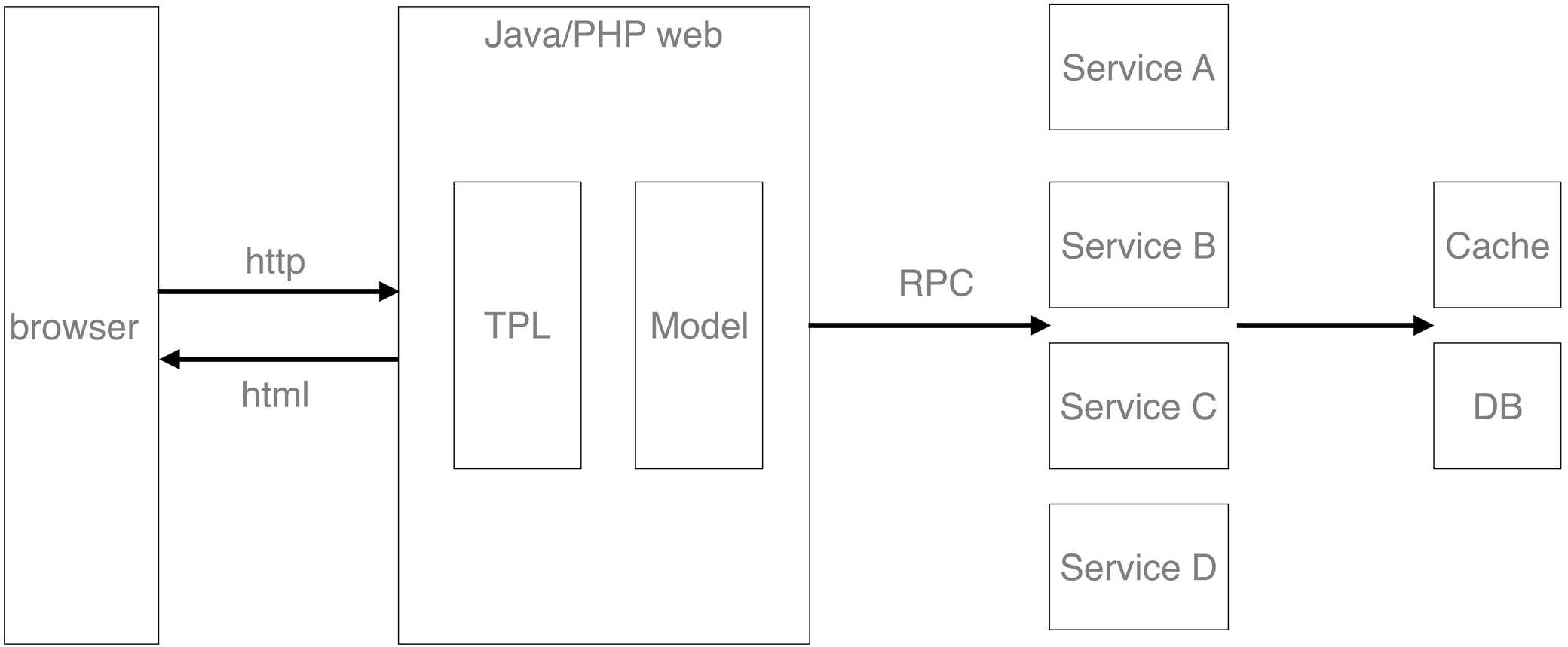
# 应用场景

- 直出
- 前后端分离
- SSR（React）

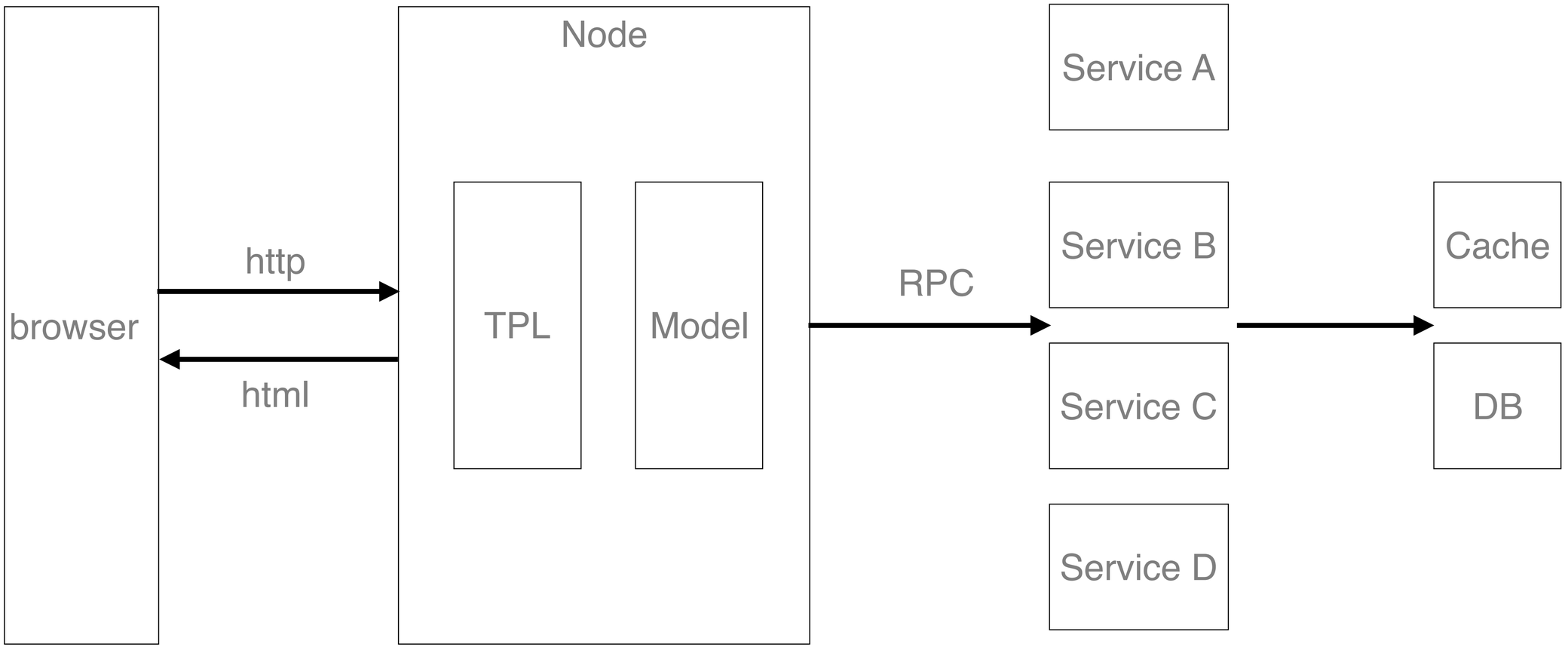
# 传统直出


Java/PHP web
View
TPL
Model
Business Logic
browser
http
html
Cache
DB

# 传统直出


Java/PHP web
Service A
Service B
Service C
Service D
Cache
DB
browser
http
html
TPL
Model
RPC

# 传统直出


Node
Service A
Service B
Service C
Service D
Cache
DB
browser
http
html
TPL
Model
RPC

# 直出

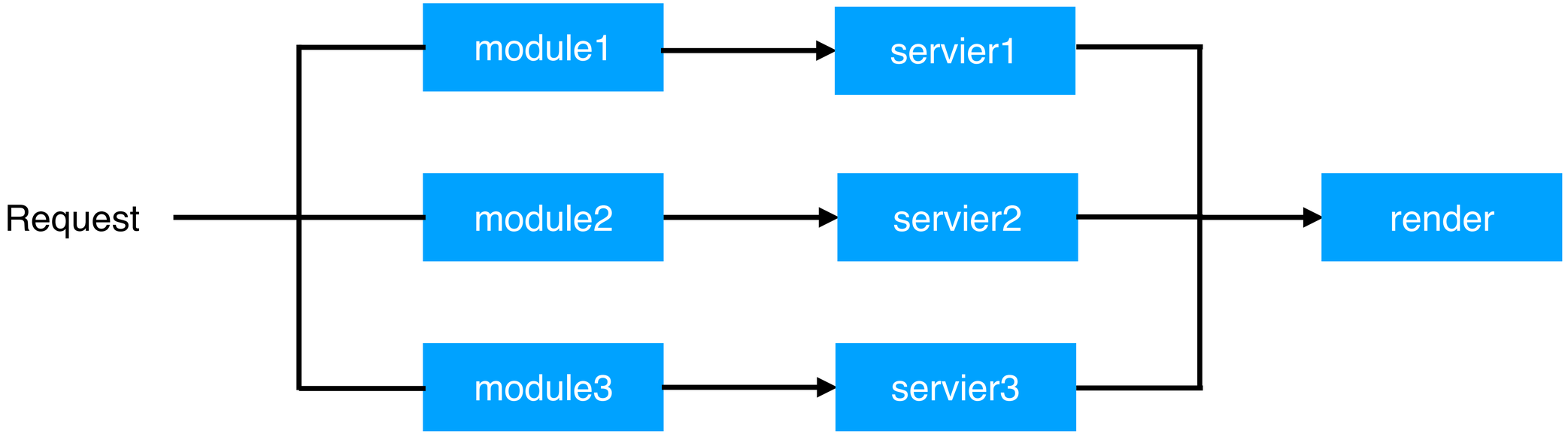
- 拉取数据
- 渲染模板
- 前端交互对接

# 代码示例

```
router.get('/article', async ctx => {
    let article = await fetchArticleFromService(ctx.query.id)
    ctx.body = `<!DOCTYPE html>
    <html lang="en">
    <head>
        <meta charset="UTF-8">
        <title>${article.title}</title>
    </head>
    <body>
        <div class="article">
            ${article.content}
        </div>
        <script src="/public/xxxx.js"></script>
    </body>
    </html>`
})
```

# 应对复杂度

- 分模块
- 并发


Request
module1
module2
module3
servier1
servier2
servier3
render

# 代码示例

```
async ctx => {
    let data = await Promise.all([
        fetchModule1(),
        fetchModule2(),
        fetchModule3()
    ])

    ctx.body = `
        ...
    `
}
```

# 可选

- 模块化框架
- 模板引擎及其中间件
- 缓存

# 优缺点

√ SEO

√ 性能好

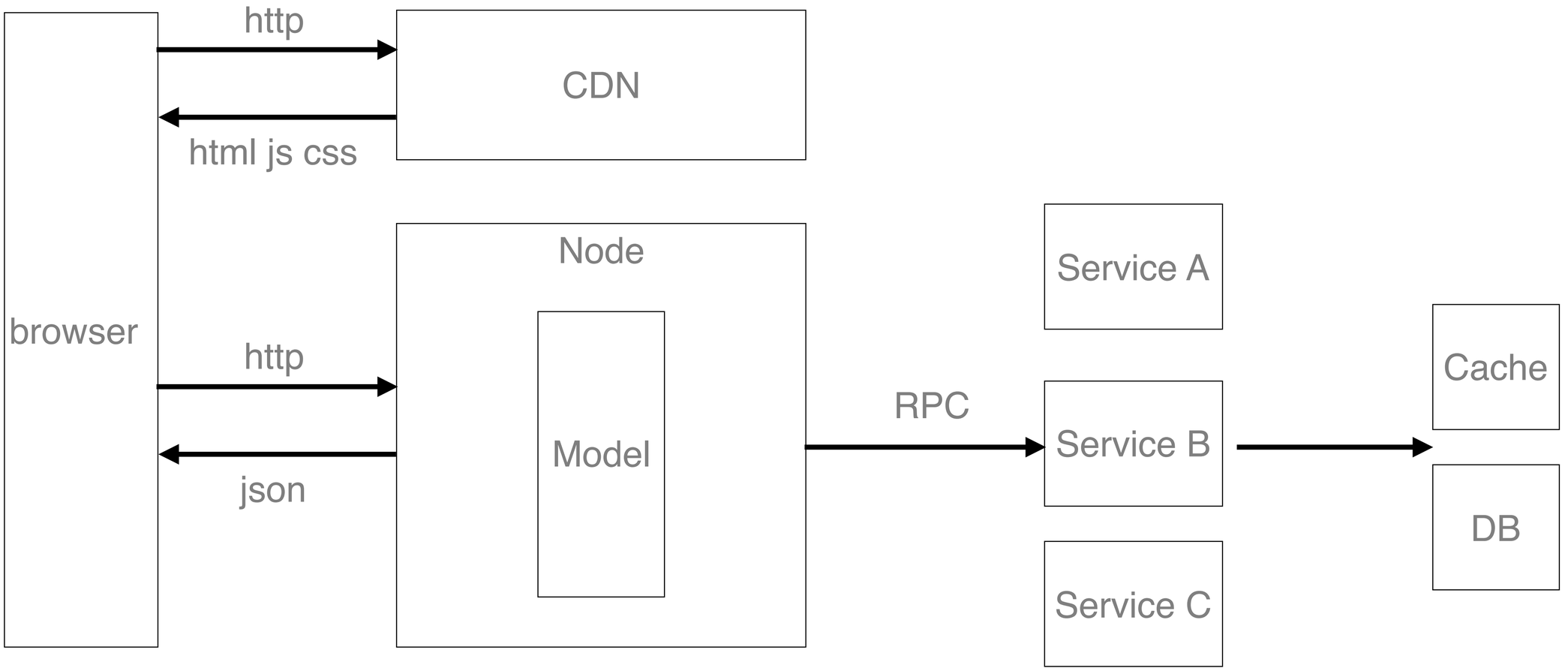
✗ 耦合重

# 适合页面

1.重SEO

2.浏览型页面

# 前后端分离


http
CDN
html js css
Node
Service A
browser
http
Model
RPC
Service B
Cache
json
DB
Service C

# http API

- jsonp支持
- 跨域ajax支持

# 优缺点

√ 解耦

√ qps高

✗ SEO

✗ 首屏慢

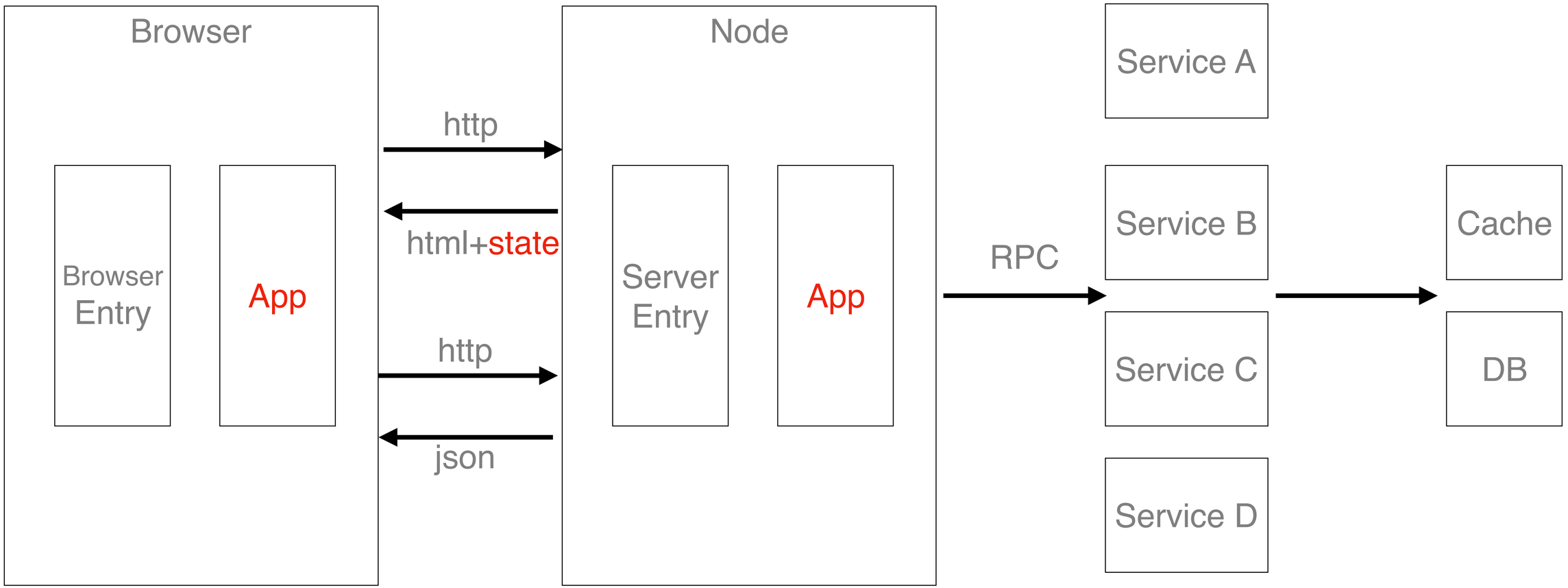
# 适合页面

1.无SEO

2.重交互

3. hybrid

# React SSR(同构)


Browser
Browser Entry
App
http
html+state
http
json
Node
Server Entry
App
RPC
Service A
Service B
Service C
Service D
Cache
DB

# 核心

- App 逻辑不依赖平台
- State 透传

# 以文章页面为例子

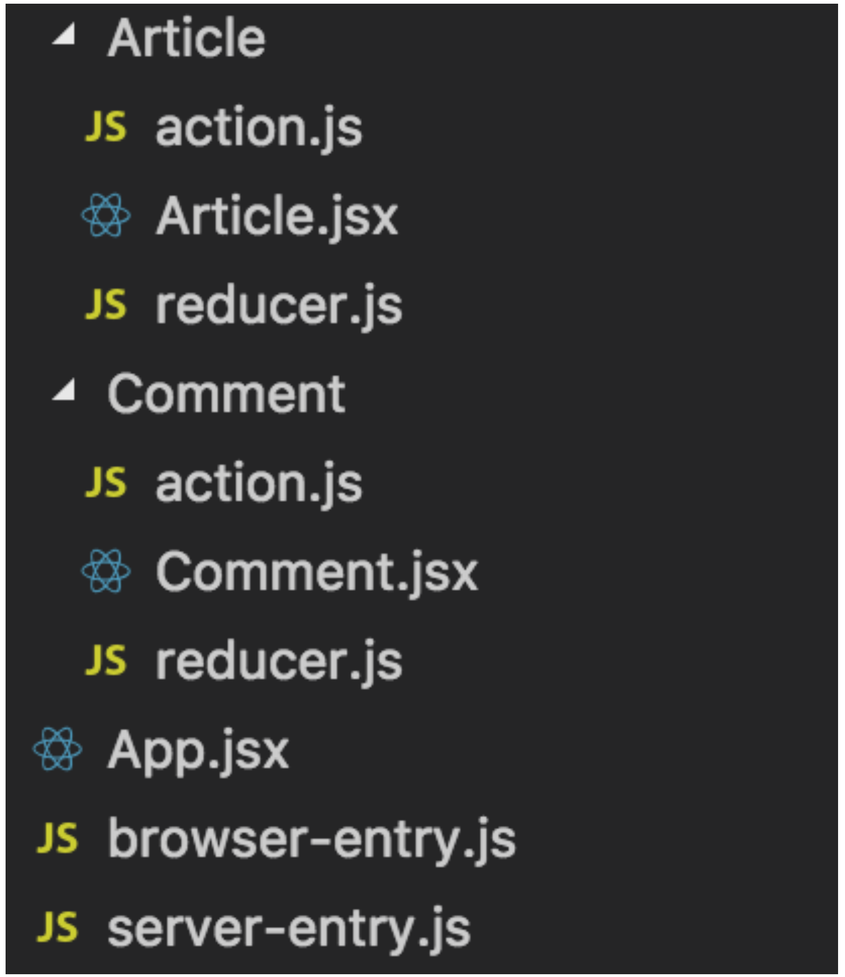
```
<div>
    <Article />
    <Comment />
</div>
```

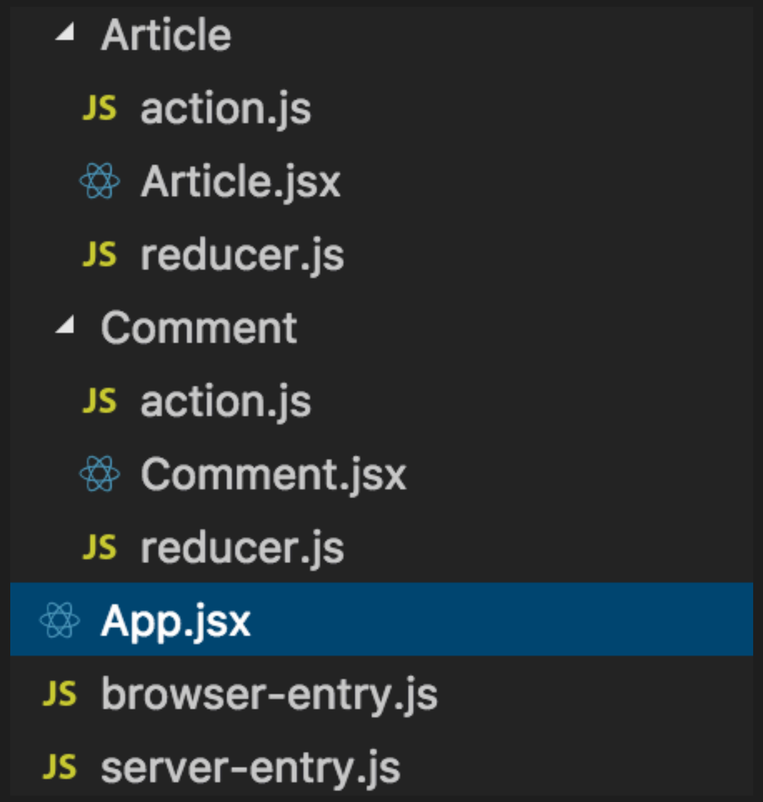
# 同构结构

server-entry

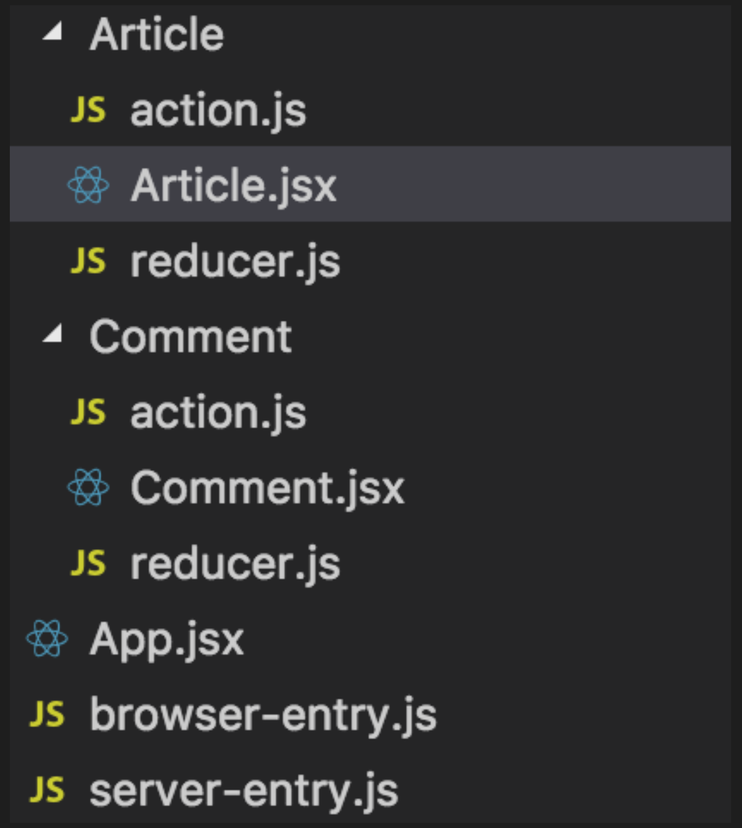
browser-entry

React App

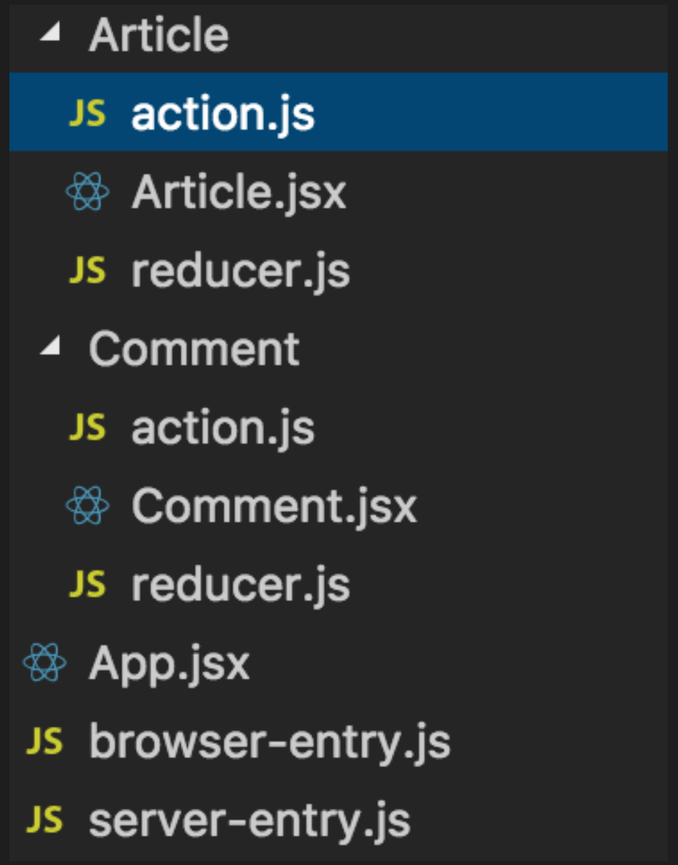




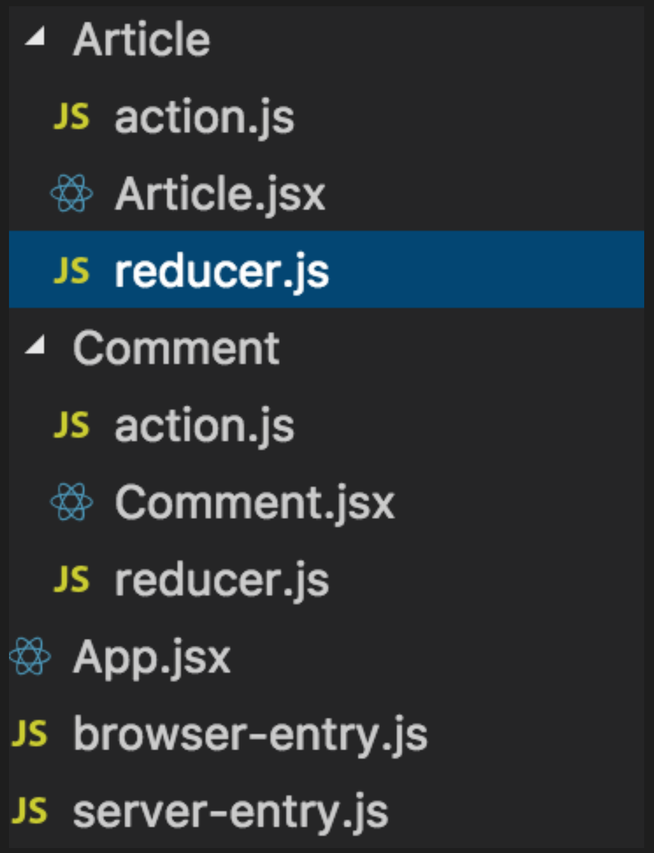

```
module.exports = (initState) => {
    let store = createStore(combineReducers({
        id: (state = 0) => state,
        article: articleReducer,
        comment: commentReducer
    }), initState)

    return {
        app: <Provider store={store}>
            <div>
                <Article />
                <Comment />
            </div>
        </Provider>,
        store: store
    }
}
```




```
class Article extends React.Component {
    render() {
        return <div className="article">
            <div className="title" >{this.props.title}</div>
            <div className="content">{this.props.content}</div>
        </div>
    }
    componentDidMount() {
        console.log('article mount')
    }
}

module.exports = connect(state => ({
    title: state.article.title,
    content: state.article.content
}))(Article)
```




```
exports.init = async (dispatch, id) => {

    let article = await fetch('/api/article', {
        id: id
    })

    dispatch({
        type: 'INIT_ARTICLE',
        data: article
    })
}
```




```
module.exports = (preState = {
    title: '',
    content: ''
}, action) => {
    if (action.type === 'INIT_ARTICLE') {
        return action.data
    }
    return preState
}
```

# Server主流程

- 壳子模板
- server-entry
  - renderToString
  - json state

# 壳子模板

```
router.get('/article', async ctx => {
    let pageData = await preparePageData(ctx)
    let reactData = await createReactApp(ctx)
    ctx.body = `<!DOCTYPE html>
    <html>
    <head>
        <meta charset="UTF-8">
        <title>${pageData.title}</title>
    </head>
    <body>
        <div id="react-app">${reactData.domString}</div>
        <script>
            var REACT_PAGE_STATE = ${reactData.state}
        </script>
        <script src="browser-entry.js"></script>
    </body>
    </html>`
})
```

# server-entry

```
exports.createReactApp = async ctx => {
    const initState = {
        id: ctx.query.id
    }
    let { app, store } = App(initState)

    await articleAction.init(store.dispatch, ctx.query.id)

    return {
        domString: renderToString(app),
        state: JSON.stringify(store.getState())
    }
}
```

# Browser主流程

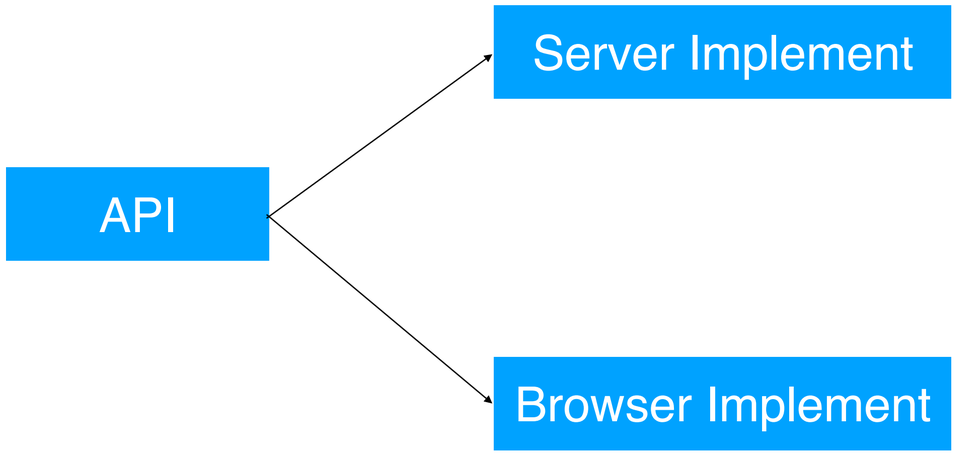
- browser-entry
  - state
  - hydrate
  - client biz

# browser-entry

```
let { app, store } = App(window.REACT_PAGE_STATE)
hydrate(app, document.getElementById('react-app'))

commentAction.init(store.dispatch, store.getState().id)
```

# 平台无关

- fetch
- cookie
- ua
- logger
- ...



# 渲染控制

- 灵活配置
- 自动降级

# 构建发布

- 同步
- API接口兼容

# 优缺点

- √ SEO
- √ 首屏快
- √ 解耦

- ✗ 服务端开销大
- ✗ QPS低
- ✗ 复杂度高

# 适合页面

1.重SEO

2.重交互

3.机器资源够

# 小结

- 3种主要场景
- 业务选型

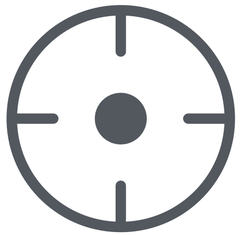
# 思考题

重新审视业务中的选型

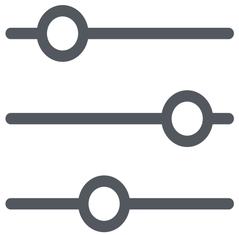
# 优化&运维

- 核心模型
- 优化指导
- 运维监控

# 核心模型



单线程



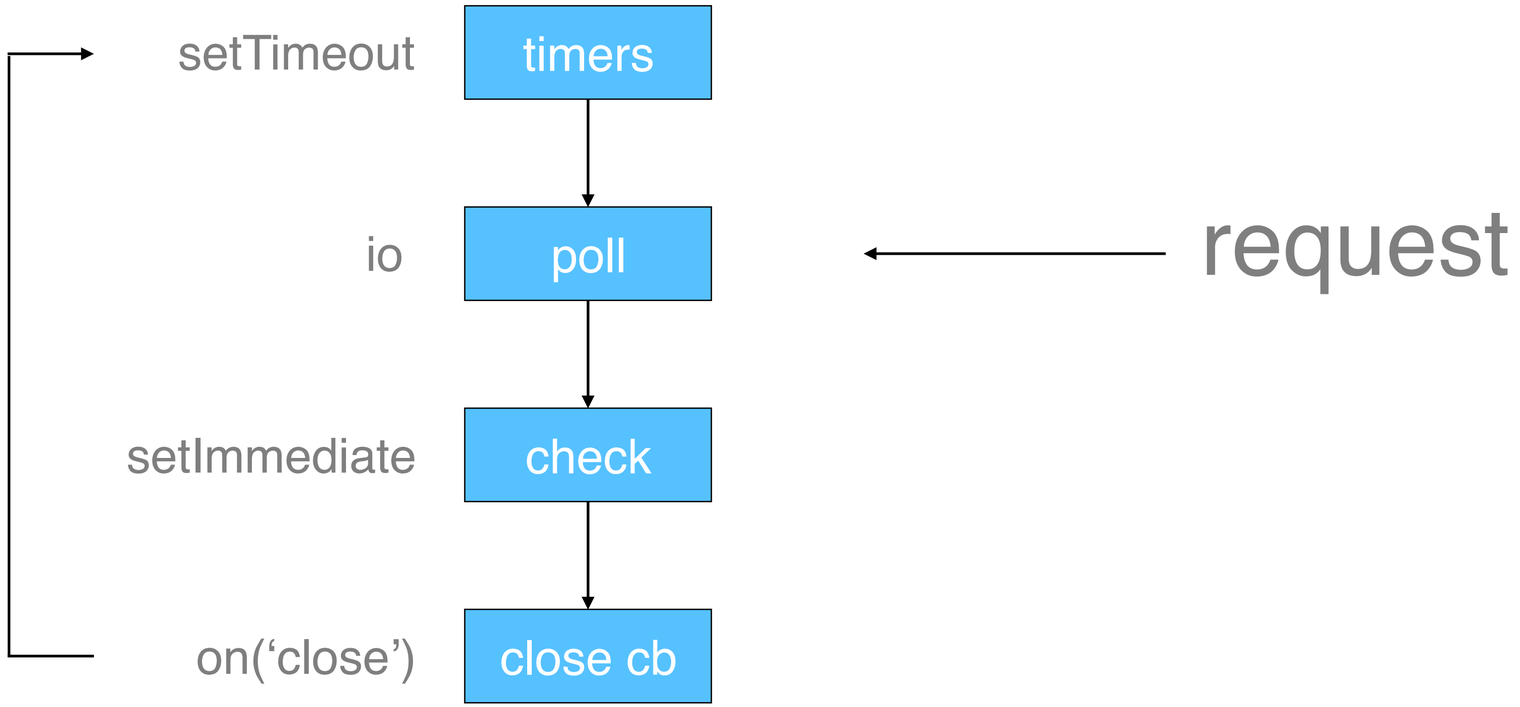
异步IO

事件循环

# libuv

- 主线程
- 工作线程池

# Event Loop


setTimeout
timers
io
poll
request
setImmediate
check
on('close')
close cb

# Node版本

- LTS 偶数大版本
- 保持统一

# 优化目标

⇩ 请求耗时

⇧ TPS

# 瓶颈

- CPU计算
- event loop阻塞

# 通用优化

- 去除co&yield —> async await

# 通用优化

- 去babel

# 通用优化

- JSON.parse & JSON.stringify
- 正则表达式

# 通用优化

- 减少Buffer copy
- 尽可能使用Buffer.allocUnsafe代替Buffer.alloc

# 通用优化

- 使用模板字符串代替模板引擎
- 简化模板中的逻辑代码

# 通用优化

- 串行 -> 并行

```
let a = await funcA()
let b = await funcB()
```

```
await Promise.all([funcA(), funcB()])
```

# 通用优化

- C & C++ 改写

# 通用优化

- 坑分享

# 场景优化

- 传统直出
- 前后端分离
- 同构SSR

# 杂谈

- 跪舔

# profile

- node 自带
- v8-profiler

# node prof

- node --prof app.js
- 压测
- node --prof-process isolate-0x102801600-v8.log > profile.txt

# node prof

```
[JavaScript]:
  ticks  total  nonlib   name
   549    5.2%    5.2%  Builtin: StringPrototypeCharCodeAt
   297    2.8%    2.8%  LazyCompile: *escapeTextForBrowser :496:34
   102    1.0%    1.0%  Builtin: InterpreterEntryTrampoline
    71    0.7%    0.7%  Builtin: CallFunction_ReceiverIsAny
    69    0.7%    0.7%  Builtin: KeyedLoadIC_Megamorphic
    66    0.6%    0.6%  StoreIC: A store IC from the snapshot
```

# v8-profiler

- npm i v8-profiler
- 代码集成
- cpu profile & mem dump

```
v8Profiler.startProfiling('web', true)
setTimeout(() => {
    let profile = v8Profiler.stopProfiling('web')
    profile.export(function(error, result) {
        fs.writeFileSync('profile.cpuprofile', result);
        profile.delete();
    });
}, 20000)
```

# 监控

- 机器监控
- Node 监控
- 业务（代码）监控

# 业务代码监控

- koa中间件

```
const monitor = async (ctx, next) => {
    let start = Date.now()
    let error = null
    try {
        await next()
    } catch (e) {
        error = e
    }
    report({
        path: ctx.path,
        cost: Date.now() - start,
        code: ctx.statusCode,
        error: error
    })
    if (error) {
        throw error
    }
}
```

# 业务代码监控

- 自定义事务 侵入代码

# 日志

- 文件日志
- 日志云

# 部署运维

- 构建
- 容器
- 负载均衡

# 其他

- 单元测试
- RPC框架
- …

# THANKS