

Koa 入门

Node.js Web 系列课程

版权声明

本内容版权属杭州饥人谷教育科技有限公司（简称饥人谷）所有。

任何媒体、网站或个人未经本网协议授权不得转载、链接、转贴，或以其他方式复制、发布和发表。

已获得饥人谷授权的媒体、网站或个人在使用时须注明「资料来源：饥人谷」。

对于违反者，饥人谷将依法追究 responsibility。

联系方式

如果你想要购买本课程

请微信联系 [xiedaimala02](#) 或 [xiedaimala03](#)

如果你发现有人盗用本课程

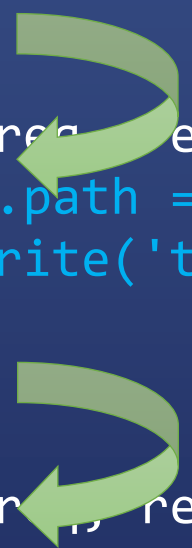
请微信联系 [xiedaimala02](#) 或 [xiedaimala03](#)

Koa 读作抠尔/抠阿/抠A

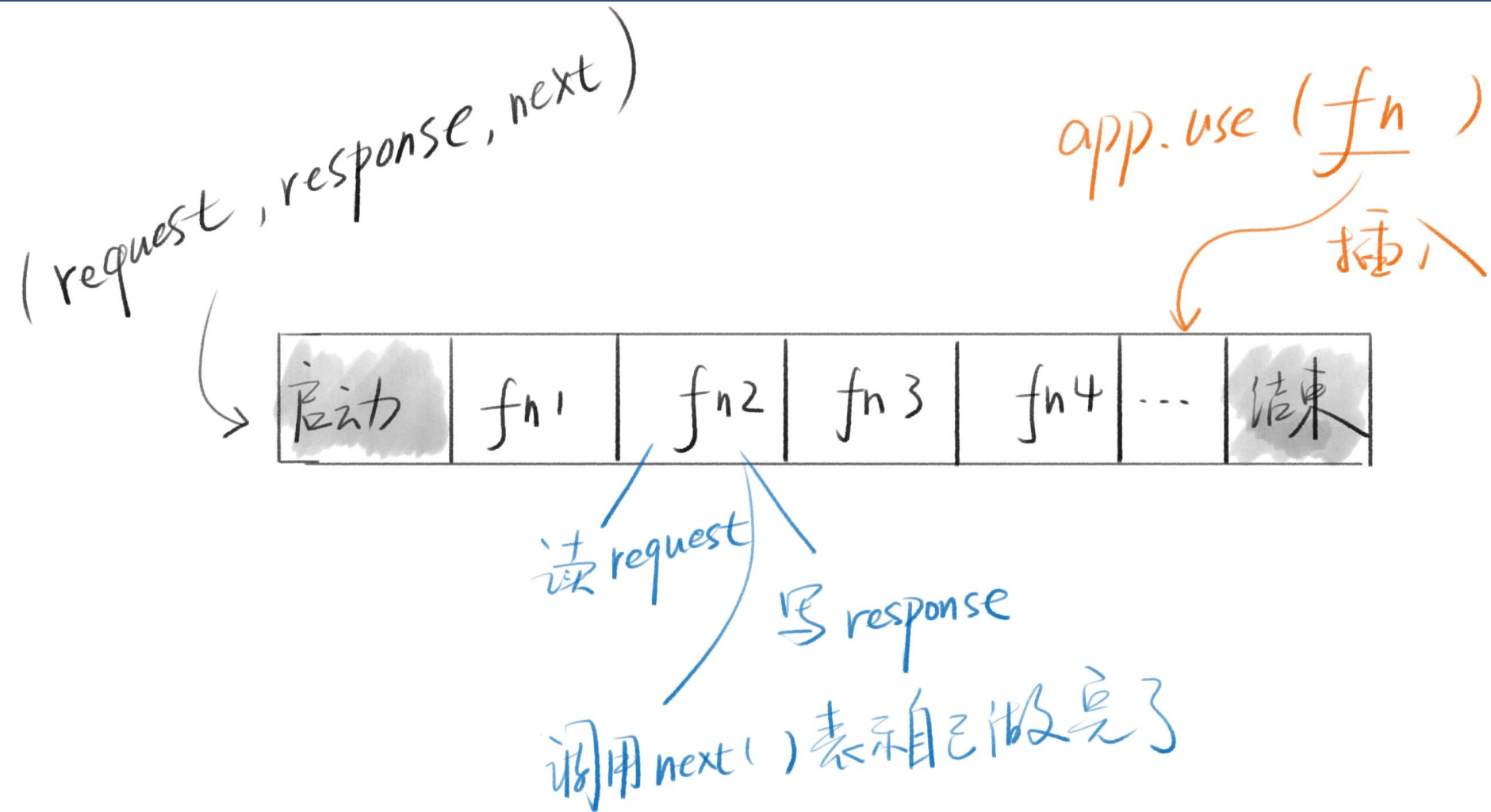
跟中国人交流你就读 K-O-A 就行了

前情提要 - Express.js

```
app.use((req, res, next) => {  
  if (req.path === '/' && req.method === 'get') {  
    res.write('this is index');  
  }  
  next();  
});  
app.use((req, res, next) => {  
  if (req.path === '/about' && req.method === 'get') {  
    res.write('this is about');  
  }  
  next();  
});  
app.use((req, res, next) => {  
  if (req.path === '/others' && req.method === 'get') {  
    res.write('this is others');  
  }  
  next();  
});
```



express 的中间件模型



Koa 的时间线

- Express

- ✓ 2010 年 6 月，TJ 开始编写 Express
- ✓ 2014 年发展到 v0.12，基本成熟，移交 StrongLoop

- Koa

- ✓ 2013 年 8 月，TJ 开始编写 Koa
- ✓ 2015 年 8 月，Koa 发布 v1.0.0 版本

- Node.js

- ✓ 2013 年 3 月，Node.js v0.10 发布
- ✓ 2014 年 12 月，io.js 不满 Node.js 的管理发起分裂
- ✓ 2015 年 2 月，Node.js v0.12 发布
- ✓ 2015 年 9 月，Node.js 与 io.js 合并为 Node.js v4.0

Koa 的时间线 2

- Koa 对 Node.js 的支持

- ✓ 2015 年 2 月，Koa 放弃对 Node v0.11 以下的支持，并开始支持 io.js
- ✓ 2015 年 10 月，Koa 放弃对 Node v4.0 以下的支持，并用 ES6 重写的所有代码，发布 v2.0.0 内测版

Koa 对比 Express

- 编程模型不同

- ✓ Express 的中间件是线型的
- ✓ Koa 的中间件是 U 型的（后面会讲）

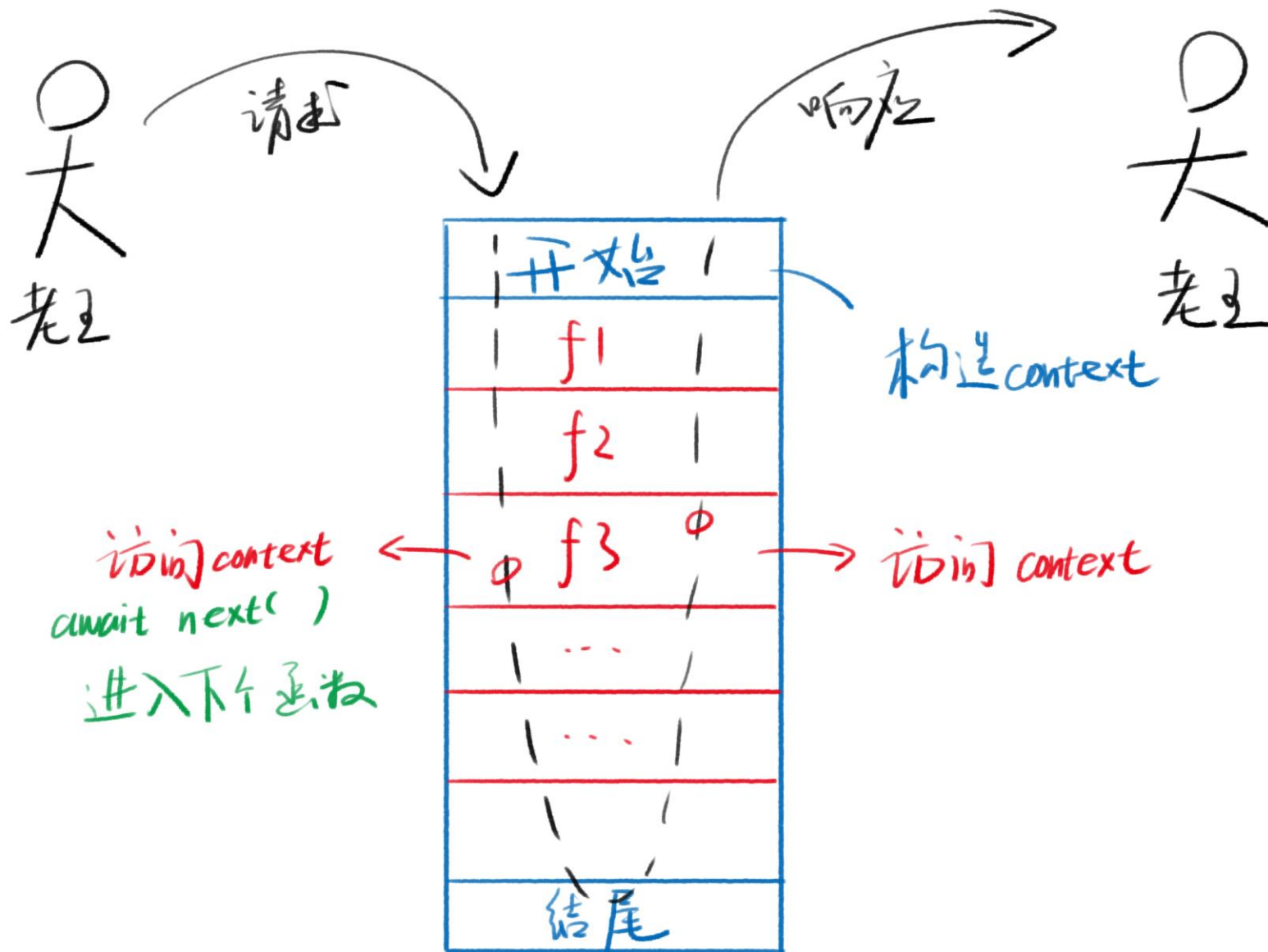
- 对语言特性的使用不同

- ✓ Express 使用回调函数 `next()`
- ✓ Koa v1.x 使用 generator 语法
- ✓ Koa v2.x 使用 `async / await` 语法

- 请问，你会使用哪一个

- ✓ 2011 ~ 2016 年，你大概率会使用 Express
- ✓ 2017 年之后，你可能会使用 Koa

Koa 的中间件模型



工具安装

- 全局安装

- ✓ 安装 node-dev 代替 node，代码一更新就自动重启
- ✓ 安装 ts-node-dev 代替 ts-node，支持 TS 语言

- 代码

- ✓ `npm i -g node-dev ts-deno-dev`
- ✓ 或者 `yarn global add node-dev ts-deno-dev`

- 注意

- ✓ 如果用 TS，需要运行 `tsc --init` 初始化 `tsconfig.json`
- ✓ `tsc` 的安装方法是全局安装 [typescript@3.8.3](#)
- ✓ 你还需要安装 `@types/koa`

示例代码

```
app.use(async (ctx, next) => {  
  await next();  
  const time = ctx.response.get('X-Response-Time');  
  console.log(`${ctx.url} - ${time}`);  
});
```

```
app.use(async (ctx, next) => {  
  const start = Date.now();  
  await next();  
  const time = Date.now() - start;  
  ctx.set('X-Response-Time', `${time}ms`);  
});
```

```
app.use(async ctx => {  
  ctx.body = 'Hello World';  
  // 最后一个中间件可以不写 await next()  
});
```

运行顺序

① → ② → ③ → ④ → ⑤

```
app.use(async (ctx, next) => {
```

① → `await next();`

```
  const time = ctx.response.get('X-Response-Time');  
  console.log(`${ctx.url} - ${time}`);  
});
```

⑤

```
app.use(async (ctx, next) => {
```

② → `const start = Date.now();`

`await next();`

```
  const time = Date.now() - start;  
  ctx.set('X-Response-Time', `${time}ms`);
```

④

```
});
```

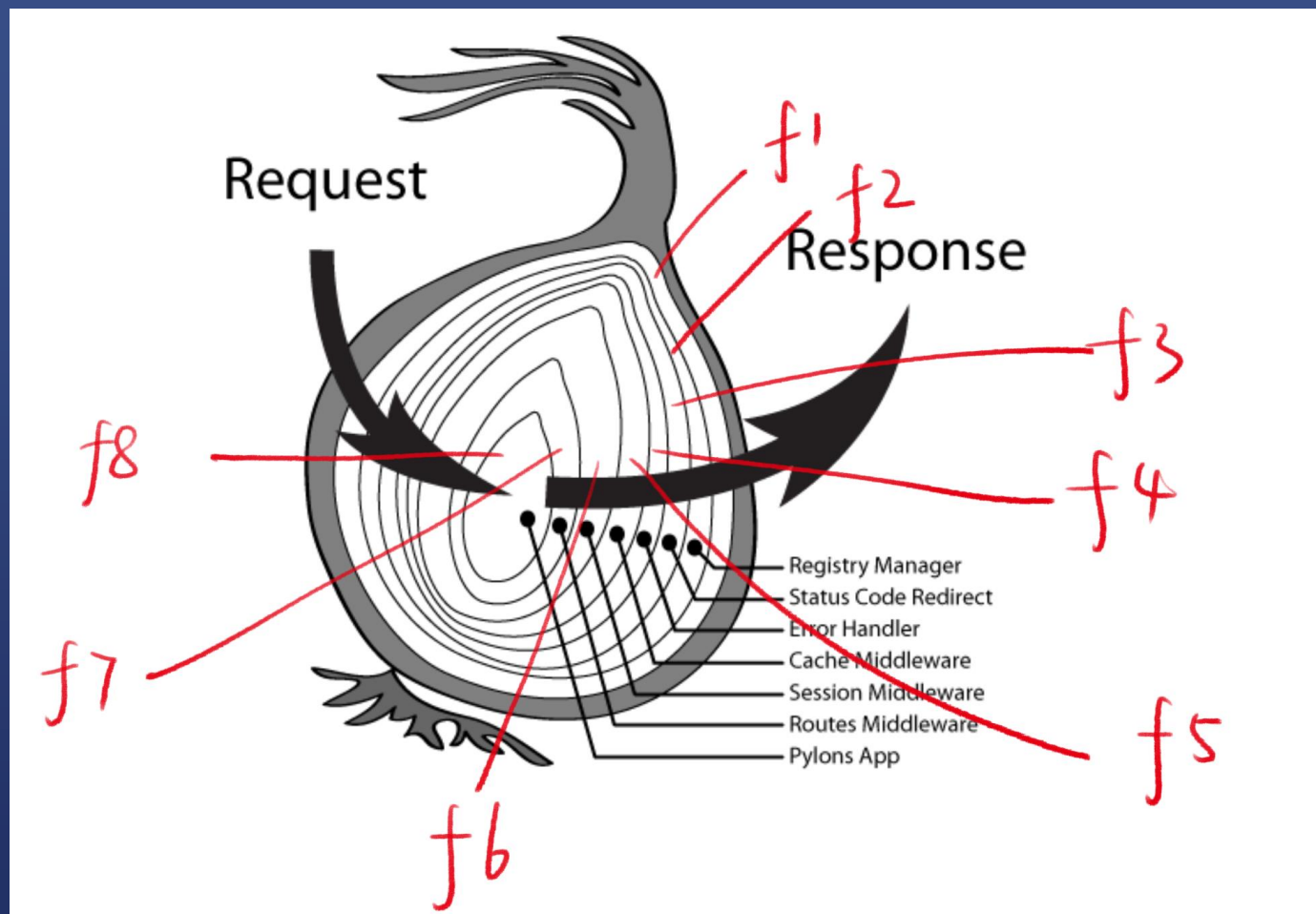
```
app.use(async ctx => {
```

③ → `ctx.body = 'Hello World';`

`// 最后一个中间件可以不写 await next()`

```
});
```

也有人将其描述为洋葱模型



实际上这个图来自 [Python Web 框架 Pylons](#)（2010 年）的官网

await next() 是什么意思

• 释义

```
app.use(async (ctx, next) => {  
  const start = Date.now();  
  await next(); // 等待 下一个中间件()  
  const time = Date.now() - start;  
  ctx.set('X-Response-Time', `${time}ms`);  
});
```

- ✓ next() 表示进入下一个函数
- ✓ 下一个函数会返回一个 Promise 对象，称为 p
- ✓ 下一个函数所有代码执行完毕后，将 p 置为成功
- ✓ await 会等待 p 成功后，再回头执行剩余的代码

await next() 释义

① → ② → ③ → ④ → ⑤

① → `app.use(async (ctx, next) => {
 await next();
 const time = ctx.response.get('X-Response-Time');
 console.log(`${ctx.url} - ${time}`);
});`

② → `app.use(async (ctx, next) => {
 const start = Date.now();
 await next();
 const time = Date.now() - start;
 ctx.set('X-Response-Time', `${time}ms`);
});`

③ → `app.use(async ctx => {
 ctx.body = 'Hello World';
 // 最后一个中间件可以不写 await next()
});`

- ✓ 1 后面的 `await next()` 会等待 4 执行完毕
- ✓ 2 后面的 `await next()` 会等待 3 执行完毕

await next() 改写

- 改写成 Promise 写法

```
app.use(async (ctx, next) => {  
  const start = Date.now();  
  return next().then(()=>{  
    const time = Date.now() - start;  
    ctx.set('X-Response-Time', `${time}ms`);  
  });  
});
```

- ✓ 一定要写 return, 因为中间件必返回 Promise 对象
- ✓ 错误处理在这里写有点反模式, 用 app.on('error') 更方便一点

思考题

- Express 里如何计算 response time
 - ✓ 思路一：用两个 app.use 加 res.locals
 - ✓ 思路二：搜业界(TJ)方案，看源码
- 我推荐的源码学习方式
 - ✓ 带着问题读源码

Koa API 全览

所有功能，了然于胸

目录

- `app.xxx` // application
- `ctx.xxx` // context
- `ctx.request.xxx`
- `ctx.response.xxx`

app.xxx

- 文档在此

- API

- ✓ app.env
- ✓ app.proxy
- ✓ app.subdomainOffset
- ✓ app.listen()
- ✓ app.callback()
- ✓ app.use(fn) —— 插入中间件 fn
- ✓ app.keys
- ✓ app.context 见下一章
- ✓ app.on('error', fn) —— 错误处理
- ✓ app.emit —— 触发事件

ctx.xxx

- 文档在此

- API

- ✓ ctx.req // Node.js 封装的请求
- ✓ ctx.res
- ✓ ctx.request // Koa 封装请求
- ✓ ctx.response
- ✓ ctx.state——跨中间件分享数据
- ✓ ctx.app
- ✓ ctx.cookies.get / set
- ✓ ctx.throw
- ✓ ctx.assert
- ✓ ctx.respond 不推荐使用

- ✓ Request 委托
- ✓ Response 委托

ctx.request.xxx

- 文档在此

- API

- ✓ request.header
- ✓ request.headers
- ✓ request.method
- ✓ request.length
- ✓ request.url
- ✓ request.origin
- ✓ request.href
- ✓ request.path
- ✓ request.querystring
- ✓ request.search

- ✓ request.host
- ✓ request.hostname
- ✓ request.URL
- ✓ request.type
- ✓ request.charset
- ✓ request.query
- ✓ request.fresh
- ✓ request.stale
- ✓ request.protocol
- ✓ ...
- ✓ request.idempotent
- ✓ request.get(field)

ctx.response.xxx

- 文档在此

- API

- ✓ response.header
- ✓ response.headers
- ✓ response.socket
- ✓ response.status
- ✓ response.message
- ✓ response.length
- ✓ response.body x 5
- ✓ response.get()
- ✓ response.set() x 2
- ✓ response.append()
- ✓ response.type
- ✓ response.is()
- ✓ response.redirect(url, [alt])
- ✓ response.attachment()
- ✓ response.headerSent
- ✓ response.flushHeaders()
- ✓ ...

总结

- Koa 原理

- ✓ 封装请求和响应
- ✓ 通过 U 型模型 / 洋葱模型构造中间件

- 跟 Express 的区别

- ✓ 模型不同
- ✓ 语法特性不同（对 Node.js 7.6.0 版本的要求不同）
- ✓ 没有内置中间件

- Koa API

- ✓ 平平无奇，几乎和 Express 一样

谁在用 Koa

- 框架

- ✓ egg.js

- 但我推荐另一个框架

- ✓ <https://2019.stateofjs.com/back-end/>

- ✓ Next.js / Nuxt / Nest

再见

下节课开始了解基于 Express / Koa 的框架