# 外文翻译

Egg.js <https://eggjs.org/en/intro/index.html>

译文：

# Egg.js 是什么?

**Egg.js 为企业级框架和应用而生**，我们希望由 Egg.js 孕育出更多上层框架，帮助开发团队和开发人员降低开发和维护成本。

注：Egg.js 缩写为 Egg

## 设计原则

我们深知企业级应用在追求规范和共建的同时，还需要考虑如何平衡不同团队之间的差异，求同存异。所以我们没有选择社区常见框架的大集市模式（集成如数据库、模板引擎、前端框架等功能），而是专注于提供 Web 开发的核心功能和一套灵活可扩展的插件机制。我们不会做出技术选型，因为固定的技术选型会使框架的扩展性变差，无法满足各种定制需求。通过 Egg，团队的架构师和技术负责人可以非常容易地基于自身的技术架构在 Egg 基础上扩展出适合自身业务场景的框架。

Egg 的插件机制有很高的可扩展性，**一个插件只做一件事**（比如 [Nunjucks](https://mozilla.github.io/nunjucks" \t "_blank) 模板封装成了 [egg-view-nunjucks](https://github.com/eggjs/egg-view-nunjucks)、MySQL 数据库封装成了 [egg-mysql](https://github.com/eggjs/egg-mysql)）。Egg 通过框架聚合这些插件，并根据自己的业务场景定制配置，这样应用的开发成本就变得很低。

Egg 奉行『**约定优于配置**』，按照[一套统一的约定](https://eggjs.org/zh-cn/advanced/loader.html)进行应用开发，团队内部采用这种方式可以减少开发人员的学习成本，开发人员不再是『钉子』，可以流动起来。没有约定的团队，沟通成本是非常高的，比如有人会按目录分栈而其他人按目录分功能，开发者认知不一致很容易犯错。但约定不等于扩展性差，相反 Egg 有很高的扩展性，可以按照团队的约定定制框架。使用 [Loader](https://eggjs.org/zh-cn/advanced/loader.html) 可以让框架根据不同环境定义默认配置，还可以覆盖 Egg 的默认约定。

## 与社区框架的差异

[Express](http://expressjs.com/) 是 Node.js 社区广泛使用的框架，简单且扩展性强，非常适合做个人项目。但框架本身缺少约定，标准的 MVC 模型会有各种千奇百怪的写法。Egg 按照约定进行开发，奉行『约定优于配置』，团队协作成本低。

[Sails](http://sailsjs.com/) 是和 Egg 一样奉行『约定优于配置』的框架，扩展性也非常好。但是相比 Egg，[Sails](http://sailsjs.com/" \t "_blank) 支持 Blueprint REST API、[WaterLine](https://github.com/balderdashy/waterline" \t "_blank) 这样可扩展的 ORM、前端集成、WebSocket 等，但这些功能都是由 [Sails](http://sailsjs.com/) 提供的。而 Egg 不直接提供功能，只是集成各种功能插件，比如实现 egg-blueprint，egg-waterline 等这样的插件，再使用 sails-egg 框架整合这些插件就可以替代 [Sails](http://sailsjs.com/) 了。

# Egg.js 与 Koa

## 异步编程模型

Node.js 是一个异步的世界，官方 API 支持的都是 callback 形式的异步编程模型，这会带来许多问题，例如

* [callback hell](http://callbackhell.com/): 最臭名昭著的 callback 嵌套问题。
* [release zalgo](https://oren.github.io/#/articles/zalgo/): 异步函数中可能同步调用 callback 返回数据，带来不一致性。

因此社区提供了各种异步的解决方案，最终胜出的是 Promise，它也内置到了 ECMAScript 2015 中。而在 Promise 的基础上，结合 Generator 提供的切换上下文能力，出现了 [co](https://github.com/tj/co) 等第三方类库来让我们用同步写法编写异步代码。同时，[async function](https://github.com/tc39/ecmascript-asyncawait) 这个官方解决方案也于 ECMAScript 2017 中发布，并在 Node.js 8 中实现。

### async function

[async function](https://github.com/tc39/ecmascript-asyncawait) 是语言层面提供的语法糖，在 async function 中，我们可以通过 await 关键字来等待一个 Promise 被 resolve（或者 reject，此时会抛出异常）， Node.js 现在的 LTS 版本（8.x）已原生支持。

|  |
| --- |
| **const** fn = **async** **function**() {  **const** user = **await** getUser();  **const** posts = **await** fetchPosts(user.id);  **return** { user, posts }; }; fn().then(res => console.log(res)).catch(err => console.error(err.stack)); |

## Koa

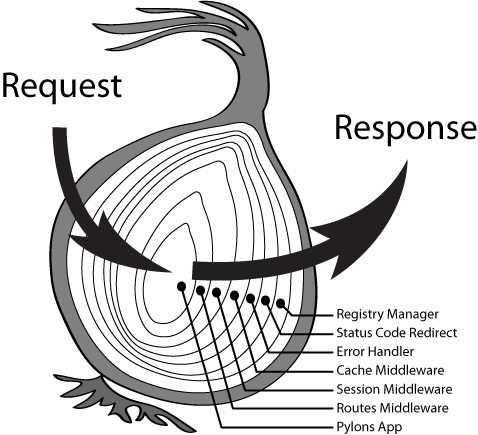
[Koa](https://koajs.com/) 是一个新的 web 框架，由 Express 幕后的原班人马打造， 致力于成为 web 应用和 API 开发领域中的一个更小、更富有表现力、更健壮的基石。

Koa 和 Express 的设计风格非常类似，底层也都是共用的[同一套 HTTP 基础库](https://github.com/jshttp)，但是有几个显著的区别，除了上面提到的默认异步解决方案之外，主要的特点还有下面几个。

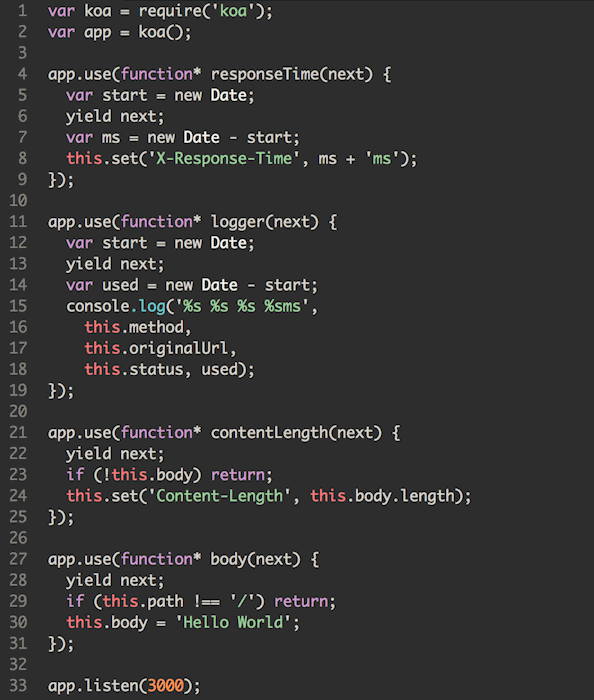
### Middleware

Koa 的中间件和 Express 不同，Koa 选择了洋葱圈模型。

* 中间件洋葱图：



* 中间件执行顺序图：



所有的请求经过一个中间件的时候都会执行两次，对比 Express 形式的中间件，Koa 的模型可以非常方便的实现后置处理逻辑，对比 Koa 和 Express 的 Compress 中间件就可以明显的感受到 Koa 中间件模型的优势。

* [koa-compress](https://github.com/koajs/compress/blob/master/index.js) for Koa.
* [compression](https://github.com/expressjs/compression/blob/master/index.js) for Express.

### Context

和 Express 只有 Request 和 Response 两个对象不同，Koa 增加了一个 Context 的对象，作为这次请求的上下文对象（在 Koa 1 中为中间件的 this，在 Koa 2 中作为中间件的第一个参数传入）。我们可以将一次请求相关的上下文都挂载到这个对象上。类似 [traceId](https://github.com/eggjs/egg-tracer/blob/1.0.0/lib/tracer.js" \l "L12" \t "_blank) 这种需要贯穿整个请求（在后续任何一个地方进行其他调用都需要用到）的属性就可以挂载上去。相较于 request 和 response 而言更加符合语义。

同时 Context 上也挂载了 Request 和 Response 两个对象。和 Express 类似，这两个对象都提供了大量的便捷方法辅助开发，例如

* get request.query
* get request.hostname
* set response.body
* set response.status

### 异常处理

通过同步方式编写异步代码带来的另外一个非常大的好处就是异常处理非常自然，使用 try catch 就可以将按照规范编写的代码中的所有错误都捕获到。这样我们可以很便捷的编写一个自定义的错误处理中间件。

|  |
| --- |
| **async** **function** **onerror**(ctx, next) {  **try** {  **await** next();  } **catch** (err) {  ctx.app.emit('error', err);  ctx.body = 'server error';  ctx.status = err.status || 500;  } } |

只需要将这个中间件放在其他中间件之前，就可以捕获它们所有的同步或者异步代码中抛出的异常了。

## Egg 继承于 Koa

如上述，Koa 是一个非常优秀的框架，然而对于企业级应用来说，它还比较基础。

而 Egg 选择了 Koa 作为其基础框架，在它的模型基础上，进一步对它进行了一些增强。

### 扩展

在基于 Egg 的框架或者应用中，我们可以通过定义 app/extend/{application,context,request,response}.js 来扩展 Koa 中对应的四个对象的原型，通过这个功能，我们可以快速的增加更多的辅助方法，例如我们在 app/extend/context.js 中写入下列代码：

|  |
| --- |
| *// app/extend/context.js* module.exports = {  **get** isIOS() {  **const** iosReg = /iphone|ipad|ipod/i;  **return** iosReg.test(**this**.get('user-agent'));  }, }; |

在 Controller 中，我们就可以使用到刚才定义的这个便捷属性了：

|  |
| --- |
| *// app/controller/home.js* exports.handler = ctx => {  ctx.body = ctx.isIOS  ? 'Your operating system is iOS.'  : 'Your operating system is not iOS.'; }; |

更多关于扩展的内容，请查看[扩展](https://eggjs.org/zh-cn/basics/extend.html)章节。

### 插件

众所周知，在 Express 和 Koa 中，经常会引入许许多多的中间件来提供各种各样的功能，例如引入 [koa-session](https://github.com/koajs/session" \t "_blank) 提供 Session 的支持，引入 [koa-bodyparser](https://github.com/koajs/bodyparser" \t "_blank) 来解析请求 body。而 Egg 提供了一个更加强大的插件机制，让这些独立领域的功能模块可以更加容易编写。

一个插件可以包含

* extend：扩展基础对象的上下文，提供各种工具类、属性。
* middleware：增加一个或多个中间件，提供请求的前置、后置处理逻辑。
* config：配置各个环境下插件自身的默认配置项。

一个独立领域下的插件实现，可以在代码维护性非常高的情况下实现非常完善的功能，而插件也支持配置各个环境下的默认（最佳）配置，让我们使用插件的时候几乎可以不需要修改配置项。

[egg-security](https://github.com/eggjs/egg-security) 插件就是一个典型的例子。

更多关于插件的内容，请查看[插件](https://eggjs.org/zh-cn/basics/plugin.html)章节。

### Egg 与 Koa 的版本关系

#### Egg 1.x

Egg 1.x 发布时，Node.js 的 LTS 版本尚不支持 async function，所以 Egg 1.x 仍然基于 Koa 1.x 开发，但是在此基础上，Egg 全面增加了 async function 的支持，再加上 Egg 对 Koa 2.x 的中间件也完全兼容，应用层代码可以完全基于 async function 来开发。

* 底层基于 Koa 1.x，异步解决方案基于 [co](https://github.com/tj/co) 封装的 generator function。
* 官方插件以及 Egg 核心使用 generator function 编写，保持对 Node.js LTS 版本的支持，在必要处通过 co 包装以兼容在 async function 中的使用。
* 应用开发者可以选择 async function（Node.js 8.x+） 或者 generator function（Node.js 6.x+）进行编写。

#### Egg 2.x

Node.js 8 正式进入 LTS 后，async function 可以在 Node.js 中使用并且没有任何性能问题了，Egg 2.x 基于 Koa 2.x，框架底层以及所有内置插件都使用 async function 编写，并保持了对 Egg 1.x 以及 generator function 的完全兼容，应用层只需要升级到 Node.js 8 即可从 Egg 1.x 迁移到 Egg 2.x。

* 底层基于 Koa 2.x，异步解决方案基于 async function。
* 官方插件以及 Egg 核心使用 async function 编写。
* 建议业务层迁移到 async function 方案。
* 只支持 Node.js 8 及以上的版本。

# 渐进式开发

在 Egg 里面，有[插件](https://eggjs.org/zh-cn/basics/plugin.html)，也有[框架](https://eggjs.org/zh-cn/advanced/framework.html)，前者还包括了 path 和 package 两种加载模式，那我们应该如何选择呢？

本文将以实例的方式，一步步给大家演示下，如何渐进式地进行代码演进。

全部的示例代码可以参见 [eggjs/examples/progressive](https://github.com/eggjs/examples/tree/master/progressive" \t "_blank)。

## 最初始的状态

假设我们有一段分析 UA 的代码，实现以下功能：

* ctx.isAndroid
* ctx.isIOS

通过之前的教程，大家一定可以很快地写出来，我们快速回顾下：

对应的代码参见 [step1](https://github.com/eggjs/examples/tree/master/progressive/step1)。

目录结构：

|  |
| --- |
| example-app ├── app │ ├── extend │ │ └── context.js │ └── router.js ├── test │ └── index.test.js └── package.json |

核心代码：

|  |
| --- |
| *// app/extend/context.js* module.exports = {  **get** isIOS() {  **const** iosReg = /iphone|ipad|ipod/i;  **return** iosReg.test(**this**.get('user-agent'));  }, }; |

## 插件的雏形

我们很明显能感知到，这段逻辑是具备通用性的，可以写成插件。

但一开始的时候，功能还没完善，直接独立插件，维护起来比较麻烦。

此时，我们可以把代码写成插件的形式，但并不独立出去。

对应的代码参见 [step2](https://github.com/eggjs/examples/tree/master/progressive/step2)。

新的目录结构：

|  |
| --- |
| example-app ├── app │ └── router.js ├── config │ └── plugin.js ├── lib │ └── plugin │ └── egg-ua │ ├── app │ │ └── extend │ │ └── context.js │ └── package.json ├── test │ └── index.test.js └── package.json |

核心代码：

* app/extend/context.js 移动到 lib/plugin/egg-ua/app/extend/context.js。
* lib/plugin/egg-ua/package.json 声明插件。

|  |
| --- |
| {  "eggPlugin": {  "name": "ua"  } } |

* config/plugin.js 中通过 path 来挂载插件。

|  |
| --- |
| *// config/plugin.js* **const** path = require('path'); exports.ua = {  enable: true,  path: path.join(\_\_dirname, '../lib/plugin/egg-ua'), }; |

## 抽成独立插件

经过一段时间开发后，该模块的功能成熟，此时可以考虑抽出来成为独立的插件。

首先，我们抽出一个 egg-ua 插件，看过[插件文档](https://eggjs.org/zh-cn/advanced/plugin.html)的同学应该都比较熟悉，我们这里只简单过一下：

目录结构：

|  |
| --- |
| egg-ua ├── app │ └── extend │ └── context.js ├── test │ ├── fixtures │ │ └── test-app │ │ ├── app │ │ │ └── router.js │ │ └── package.json │ └── ua.test.js └── package.json |

对应的代码参见 [step3/egg-ua](https://github.com/eggjs/examples/tree/master/progressive/step3/egg-ua)。

然后改造原有的应用，对应的代码参见 [step3/example-app](https://github.com/eggjs/examples/tree/master/progressive/step3/example-app)。

* 移除 lib/plugin/egg-ua 目录。
* package.json 中声明对 egg-ua 的依赖。
* config/plugin.js 中修改依赖声明为 package 方式。

|  |
| --- |
| *// config/plugin.js* exports.ua = {  enable: true,  package: 'egg-ua', }; |

**注意：在插件还没发布前，可以通过 npm link 的方式进行本地测试，具体参见 [npm-link](https://docs.npmjs.com/cli/link" \t "_blank)。**

|  |
| --- |
| $ cd example-app $ npm link ../egg-ua $ npm i $ npm test |

## 沉淀到框架

重复上述的过程，很快我们会积累了好几个插件和配置，并且我们会发现，在团队的大部分项目中，都会用到这些插件。

此时，就可以考虑抽象出一个适合团队业务场景的框架。

首先，抽象出 example-framework 框架，如上看过[框架文档](https://eggjs.org/zh-cn/advanced/framework.html)的同学应该都比较熟悉，我们这里只简单过一下：

目录结构：

|  |
| --- |
| example-framework ├── config │ ├── config.default.js │ └── plugin.js ├── lib │ ├── agent.js │ └── application.js ├── test │ ├── fixtures │ │ └── test-app │ └── framework.test.js ├── README.md ├── index.js └── package.json |

* 对应的代码参见 [example-framework](https://github.com/eggjs/examples/tree/master/progressive/step4/example-framework)。
* 把原来的 egg-ua 等插件的依赖，从 example-app 中移除，配置到该框架的 package.json 和 config/plugin.js 中。

然后改造原有的应用，对应的代码参见 [step4/example-app](https://github.com/eggjs/examples/tree/master/progressive/step4/example-app)。

* 移除 config/plugin.js 中对 egg-ua 的依赖。
* package.json 中移除对 egg-ua 的依赖。
* package.json 中声明对 example-framework 的依赖，并配置 egg.framework。

|  |
| --- |
| {  "name": "progressive",  "version": "1.0.0",  "private": true,  "egg": {  "framework": "example-framework"  },  "dependencies": {  "example-framework": "\*"  } } |

**注意：在框架还没发布前，可以通过 npm link 的方式进行本地测试，具体参见 [npm-link](https://docs.npmjs.com/cli/link" \t "_blank)。**

|  |
| --- |
| $ cd example-app $ npm link ../egg-framework $ npm i $ npm test |

## 写在最后

综上所述，大家可以看到我们是如何一步步渐进地去进行框架演进，这得益于 Egg 强大的插件机制、代码的共建，以及复用和下沉，这些步骤竟然可以这么地无痛来得以完成！

* 一般来说，当应用中有可能会复用到的代码时，直接放到 lib/plugin 目录去，如例子中的 egg-ua。
* 当该插件功能稳定后，即可独立出来作为一个 node module 。
* 如此以往，应用中相对复用性较强的代码都会逐渐独立为单独的插件。
* 当你的应用逐渐进化到针对某类业务场景的解决方案时，将其抽象为独立的 framework 进行发布。
* 当在新项目中抽象出的插件，下沉集成到框架后，其他项目只需要简单的重新 npm install 下就可以使用上，对整个团队的效率有极大的提升。
* **注意：不管是应用/插件/框架，都必须编写单元测试，并尽量实现 100% 覆盖率。**

# Egg@2 升级指南

## 背景

随着 Node.js 8 LTS 的发布， 内建了对 ES2017 Async Function 的支持。

在这之前，TJ 的 [co](https://github.com/tj/co) 使我们可以提前享受到 async/await 的编程体验，但同时它不可避免的也带来一些问题：

* 性能损失
* [错误堆栈不友好](https://github.com/eggjs/egg/wiki/co-vs-async)

现在 Egg 正式发布了 2.x 版本：

* 保持了对 Egg 1.x 以及 generator function 的**完全兼容**。
* 基于 Koa 2.x，异步解决方案基于 async function。
* 只支持 Node.js 8 及以上版本。
* 去除 [co](https://github.com/tj/co) 后堆栈信息更清晰，带来 30% 左右的性能提升（不含 Node 带来的性能提升），详细参见：[benchmark](https://eggjs.github.io/benchmark/plot/)。

Egg 的理念之一是渐进式增强，故我们为开发者提供渐进升级的体验。

* [快速升级](https://eggjs.org/zh-cn/migration.html#%E5%BF%AB%E9%80%9F%E5%8D%87%E7%BA%A7)
* [插件变更说明](https://eggjs.org/zh-cn/migration.html#%E6%8F%92%E4%BB%B6%E5%8F%98%E6%9B%B4%E8%AF%B4%E6%98%8E)
* [进一步升级](https://eggjs.org/zh-cn/migration.html#%E8%BF%9B%E4%B8%80%E6%AD%A5%E5%8D%87%E7%BA%A7)
* [针对插件开发者的升级指南](https://eggjs.org/zh-cn/migration.html#%E6%8F%92%E4%BB%B6%E5%8D%87%E7%BA%A7)

## 快速升级

* Node.js 使用最新的 LTS 版本（>=8.9.0）。
* 修改 package.json 中 egg 的依赖为 ^2.0.0。
* 检查相关插件是否发布新版本（可选）。
* 重新安装依赖，跑单元测试。

**搞定！几乎不需要修改任何一行代码，就已经完成了升级。**

## 插件变更说明

### egg-multipart

yield parts 需修改为 await parts() 或 yield parts()

|  |
| --- |
| *// old* **const** parts = ctx.multipart(); **while** ((part = **yield** parts) != null) {  *// do something* }  *// yield parts() also work* **while** ((part = **yield** parts()) != null) {  *// do something* }  *// new* **const** parts = ctx.multipart(); **while** ((part = **await** parts()) != null) {  *// do something* } |

* [egg-multipart#upload-multiple-files](https://github.com/eggjs/egg-multipart#upload-multiple-files)

### egg-userrole

不再兼容 1.x 形式的 role 定义，因为 koa-roles 已经无法兼容了。 请求上下文 Context 从 this 传入改成了第一个参数 ctx 传入，原有的 scope 变成了第二个参数。

|  |
| --- |
| *// old* app.role.use('user', **function**() {  **return** !!**this**.user; });  *// new* app.role.use((ctx, scope) => {  **return** !!ctx.user });  app.role.use('user', ctx => {  **return** !!ctx.user; }); |

* [koajs/koa-roles#13](https://github.com/koajs/koa-roles/pull/13)
* [eggjs/egg-userrole#9](https://github.com/eggjs/egg-userrole/pull/9)

## 进一步升级

得益于 Egg 对 1.x 的**完全兼容**，我们可以如何非常快速的完成升级。

不过，为了更好的统一代码风格，以及更佳的性能和错误堆栈，我们建议开发者进一步升级：

* 修改为推荐的代码风格，传送门：[代码风格指南](https://eggjs.org/zh-cn/style-guide.html)
* [中间件使用 Koa2 风格](https://eggjs.org/zh-cn/migration.html#%E4%B8%AD%E9%97%B4%E4%BB%B6%E4%BD%BF%E7%94%A8-Koa2-%E9%A3%8E%E6%A0%BC)
* [函数调用的 yieldable 转为 awaitable](https://eggjs.org/zh-cn/migration.html#yieldable-To-awaitable)

### 中间件使用 Koa2 风格

2.x 仍然保持对 1.x 风格的中间件的兼容，故不修改也能继续使用。

* 返回的函数入参改为 Koa 2 的 (ctx, next) 风格。
  + 第一个参数为 ctx，代表当前请求的上下文，是 [Context](https://eggjs.org/zh-cn/basics/extend.html#Context) 的实例。
  + 第二个参数为 next，用 await 执行它来执行后续中间件的逻辑。
* 不建议使用 async (ctx, next) => {} 格式，避免错误堆栈丢失函数名。
* yield next 改为函数调用 await next() 的方式。

|  |
| --- |
| *// 1.x* module.exports = () => {  **return** **function**\* **responseTime**(next) {  **const** start = Date.now();  **yield** next;  **const** delta = Math.ceil(Date.now() - start);  **this**.set('X-Response-Time', delta + 'ms');  }; };  *// 2.x* module.exports = () => {  **return** **async** **function** **responseTime**(ctx, next) {  **const** start = Date.now();  *// 注意，和 generator function 格式的中间件不同，此时 next 是一个方法，必须要调用它*  **await** next();  **const** delta = Math.ceil(Date.now() - start);  ctx.set('X-Response-Time', delta + 'ms');  }; }; |

### yieldable to awaitable

我们早在 Egg 1.x 时就已经支持 async，故若应用层已经是 async-base 的，就可以跳过本小节内容了。

[co](https://github.com/tj/co) 支持了 yieldable 兼容类型：

* promises
* array (parallel execution)
* objects (parallel execution)
* thunks (functions)
* generators (delegation)
* generator functions (delegation)

尽管 generator 和 async 两者的编程模型基本一模一样，但由于上述的 co 的一些特殊处理，导致在移除 co 后，我们需要根据不同场景自行处理：

#### promise

直接替换即可：

|  |
| --- |
| **function** **echo**(msg) {  **return** Promise.resolve(msg); }  **yield** echo('hi egg'); *// change to* **await** echo('hi egg'); |

#### array - yield []

yield [] 常用于并发请求，如：

|  |
| --- |
| **const** [ news, user ] = **yield** [  ctx.service.news.list(topic),  ctx.service.user.get(uid), ]; |

这种修改起来比较简单，用 Promise.all() 包装下即可：

|  |
| --- |
| **const** [ news, user ] = **await** Promise.all([  ctx.service.news.list(topic),  ctx.service.user.get(uid), ]); |

#### object - yield {}

yield {} 和 yield map 的方式也常用于并发请求，但由于 Promise.all 不支持 Object，会稍微有点复杂。

|  |
| --- |
| *// app/service/biz.js* **class** **BizService** **extends** **Service** {  \* list(topic, uid) {  **return** {  news: ctx.service.news.list(topic),  user: ctx.service.user.get(uid),  };  } }  *// app/controller/home.js* **const** { news, user } = **yield** ctx.service.biz.list(topic, uid); |

建议修改为 await Promise.all([]) 的方式：

|  |
| --- |
| *// app/service/biz.js* **class** **BizService** **extends** **Service** {  list(topic, uid) {  **return** Promise.all([  ctx.service.news.list(topic),  ctx.service.user.get(uid),  ]);  } }  *// app/controller/home.js* **const** [ news, user ] = **await** ctx.service.biz.list(topic, uid); |

如果无法修改对应的接口，可以临时兼容下：

* 使用我们提供的 Utils 方法 [app.toPromise](https://github.com/eggjs/egg-core/blob/da4ba1784175c43217125f3d5cd7f0be3d5396bf/lib/egg.js" \l "L353" \t "_blank)。
* **建议尽量改掉，因为实际上就是丢给 co，会带回对应的性能损失和堆栈问题。**

|  |
| --- |
| **const** { news, user } = **await** app.toPromise(ctx.service.biz.list(topic, uid)); |

#### 其他

* thunks (functions)
* generators (delegation)
* generator functions (delegation)

修改为对应的 async function 即可，如果不能修改，则可以用 [app.toAsyncFunction](https://github.com/eggjs/egg-core/blob/da4ba1784175c43217125f3d5cd7f0be3d5396bf/lib/egg.js" \l "L344" \t "_blank) 简单包装下。

**注意**

* [toAsyncFunction](https://github.com/eggjs/egg-core/blob/da4ba1784175c43217125f3d5cd7f0be3d5396bf/lib/egg.js#L344) 和 [toPromise](https://github.com/eggjs/egg-core/blob/da4ba1784175c43217125f3d5cd7f0be3d5396bf/lib/egg.js" \l "L353" \t "_blank) 实际使用的是 [co](https://github.com/tj/co) 包装，因此会带回对应的性能损失和堆栈问题，建议开发者还是尽量全链路升级。
* [toAsyncFunction](https://github.com/eggjs/egg-core/blob/da4ba1784175c43217125f3d5cd7f0be3d5396bf/lib/egg.js#L344) 在调用 async function 时不会有损失。

@sindresorhus 编写了许多[基于 promise 的 helper 方法](https://github.com/sindresorhus/promise-fun)，灵活的运用它们配合 async function 能让代码更加具有可读性。

## 插件升级

应用开发者只需升级插件开发者修改后的依赖版本即可，也可以用我们提供的命令 egg-bin autod 快速更新。

以下内容针对插件开发者，指导如何升级插件：

### 升级事项

* 完成上面章节提到的升级项。
  + 所有的 generator function 改为 async function 格式。
  + 升级中间件风格。
* 接口兼容（可选），如下。
* 发布大版本。

### 接口兼容

某些场景下，插件开发者提供给应用开发者的接口是同时支持 generator 和 async 的，一般是会用 co 包装一层。

* 在 2.x 里为了更好的性能和错误堆栈，我们建议修改为 async-first。
* 如有需要，使用 [toAsyncFunction](https://github.com/eggjs/egg-core/blob/da4ba1784175c43217125f3d5cd7f0be3d5396bf/lib/egg.js" \l "L344" \t "_blank) 和 [toPromise](https://github.com/eggjs/egg-core/blob/da4ba1784175c43217125f3d5cd7f0be3d5396bf/lib/egg.js" \l "L353" \t "_blank) 来兼容。

譬如 [egg-schedule](https://github.com/eggjs/egg-schedule) 插件，支持应用层使用 generator 或 async 定义 task。

|  |
| --- |
| *// {app\_root}/app/schedule/cleandb.js* exports.task = **function**\* (ctx) {  **yield** ctx.service.db.clean(); };  *// {app\_root}/app/schedule/log.js* exports.task = **async** **function** **splitLog**(ctx) {  **await** ctx.service.log.split(); }; |

插件开发者可以简单包装下原始函数：

|  |
| --- |
| *// https://github.com/eggjs/egg-schedule/blob/80252ef/lib/load\_schedule.js#L38* task = app.toAsyncFunction(schedule.task); |

### 插件发布规则

* **需要发布大版本**
  + 除非插件提供的接口都是 promise 的，且代码里面不存在 async，如 [egg-view-nunjucks](https://github.com/eggjs/egg-view-nunjucks)。
* 修改 package.json
  + 修改 devDependencies 依赖的 egg 为 ^2.0.0。
  + 修改 engines.node 为 >=8.0.0。
  + 修改 ci.version 为 8, 9， 并重新安装依赖以便生成新的 travis 配置文件。
* 修改 README.md 的示例为 async function。
* 编写升级指引。
* 修改 test/fixtures 为 async function，可选，建议分开另一个 PR 方便 Review。

一般还会需要继续维护上一个版本，故需要：

* 对上一个版本建立一个 1.x 这类的 branch 分支
* 修改上一个版本的 package.json 的 publishConfig.tag 为 release-1.x
* 这样如果上一个版本有 BugFix 时，npm 版本时就会发布为 release-1.x 这个 tag，用户通过 npm i egg-xx@release-1.x 来引入旧版本。
* 参见 [npm 文档](https://docs.npmjs.com/cli/dist-tag" \t "_blank)。