### 内置对象

#### Application对象

全局应用对象，只会实例化一个。

框架运行时，会在Application实例上触发一些事件，可以监听这些事件做一些操作。

一般会在app.js中进行监听：

* server: 该事件一个 worker 进程只会触发一次，在 HTTP 服务完成启动后，会将 HTTP server 通过这个事件暴露出来给开发者。
* error: 运行时有任何的异常被 onerror 插件捕获后，都会触发 error 事件，将错误对象和关联的上下文（如果有）暴露给开发者，可以进行自定义的日志记录上报等处理。
* request 和 response: 应用收到请求和响应请求时，分别会触发 request 和 response 事件，并将当前请求上下文暴露出来，开发者可以监听这两个事件来进行日志记录。

|  |
| --- |
| // app.js  module.exports = app => {  app.once('server', server => {  // websocket  });  app.on('error', (err, ctx) => {  // report error  });  app.on('request', ctx => {  // log receive request  });  app.on('response', ctx => {  // ctx.starttime is set by framework  const used = Date.now() - ctx.starttime;  // log total cost  });  }; |

获取方式：

1. 几乎所有被框架 Loader 加载的文件（Controller，Service，Schedule 等），都可以 export 一个函数，这个函数会被 Loader 调用，并使用 app 作为参数：

|  |
| --- |
| // app.js  module.exports = app => {  app.cache = new Cache();  }; |

1. 在 Context 对象上，可以通过 ctx.app 访问到 Application 对象
2. 在继承于 Controller, Service 基类的实例中，可以通过 this.app 访问到 Application 对象。

#### Context

在每一次收到用户请求时，框架会实例化一个 Context 对象，这个对象封装了这次用户请求的信息。

框架会将所有的 Service 挂载到 Context 实例上。

Middleware获取Context：

|  |
| --- |
| async function middleware(ctx, next) {  // ctx is instance of Context  console.log(ctx.query);  } |

除了用户请求时可以获取Context，非用户请求场景可以创建匿名Context实例：

Application.createAnonymousContext()；

#### Request & Response

可以通过ctx获取request和response：ctx.request 、ctx.response

ctx会代理一部分request和response上的属性和方法：

request:

ctx.query和ctx.request.query是等价的;

ctx.params 存在，而ctx.request.params不存在

ctx.body为undefined , ctx.request.body才能拿到body的值

response：

ctx.response.body和ctx.body是等价的。

注意：ctx.body是ctx.response.body的简写，不要和 ctx.request.body 混淆了

#### Controller

Controller基类有下列属性：

* + - ctx - 当前请求的 Context 实例。
    - app - 应用的 Application 实例。
    - config - 应用的配置。
    - service - 应用所有的 service。
    - logger - 为当前 controller 封装的 logger 对象。

在Controller文件中引用Controller基类：

|  |
| --- |
| // 从 egg 上获取（推荐）  const Controller = require('egg').Controller;  class UserController extends Controller {  // implement  }  module.exports = UserController; |

#### Service

继承Service基类：

|  |
| --- |
| // 从 egg 上获取（推荐）  const Service = require('egg').Service;  class UserService extends Service {  // implement  }  module.exports = UserService; |

Service有和 Controller 基类一样的属性；

#### Helper

Helper 自身是一个类，有和 Controller 基类一样的属性；

获取方式：

ctx.helper

#### Config

获取方式：

1、app.config

2、也可以在Controller, Service, Helper 的实例上通过 this.config 获取到 config 对象。因为他们实例上都有那5种属性。

#### Logger

框架中提供了多个logger对象：

* + 1. App Logger

获取方式：app.logger

作用：记录一些应用级别的日志，业务上与请求无关的日志。

* + 1. App CoreLogger

获取方式：app. coreLogger

作用：框架或插件用它来记录能够用级别的日志，一般应用不应该使用它记录日志

* + 1. Context Logger

获取方式：ctx.logger

作用：与请求有关，打印的日志前面都会带上请求相关的信息

* + 1. Context CoreLogger

获取方式：ctx.coreLogger

作用：Context Logger 的区别是一般只有插件和框架会通过它来记录日志。

* + 1. Controller Logger & Service Logger

在Controller或Service上可以使用 this.logger来获取，

本质上就是一个 Context Logger，相比Context Logger会多一个文件路径

### 运行环境

#### 指定运行环境

EGG\_SERVER\_ENV=prod npm start

#### 应用内获取运行环境

app.config.env

#### EGG\_SERVER\_ENV与NODE\_ENV的映射关系

| **NODE\_ENV** | **EGG\_SERVER\_ENV** | **说明** |
| --- | --- | --- |
|  | local | 本地开发环境 |
| test | unittest | 单元测试 |
| production | prod | 生产环境 |

如果未指定EGG\_SERVER\_ENV会根据NODE\_ENV来匹配

#### 自定义环境

比如，要为开发流程增加集成测试环境 SIT。将 EGG\_SERVER\_ENV 设置成 sit（并建议设置 NODE\_ENV = production），启动时会加载 config/config.sit.js，运行环境变量 app.config.env 会被设置成 sit。

#### 与Koa的区别

在 Koa 中我们通过 app.env 来进行环境判断，app.env 默认的值是 process.env.NODE\_ENV。但是在 Egg（和基于 Egg 的框架）中，配置统一都放置在 app.config 上，所以我们需要通过 app.config.env 来区分环境，app.env 不再使用。

### 配置

框架提供了强大且可扩展的配置功能，可以自动合并应用、插件、框架的配置，按顺序覆盖，且可以根据环境维护不同的配置。合并后的配置可直接从 app.config 获取。

#### 多环境配置

config.default.js

config.prod.js

config.unittest.js

config.local.js

config.default.js 是默认的配置文件，所有环境都会加载；

当指定env时，会加载对应的配置文件，并覆盖默认配置文件中的同名配置。

#### 配置写法

|  |
| --- |
| // 三种写法：  module.exports = {  logger: {  level: 'DEBUG',  },  };  或：  exports.logger = {  level: 'DEBUG',  };  或返回一个function，接受appInfo参数：  module.exports = appInfo => {  return {  logger: {  dir: path.join(appInfo.baseDir, 'logs'),  },  };  }; |

内置的appInfo有：

|  |  |
| --- | --- |
| appInfo | 说明 |
| pkg | package.json |
| name | 应用名，同 pkg.name |
| baseDir | 应用代码的目录 |
| HOME | 用户目录，"/Users/yangwenju" |
| root | 应用根目录，只有在 local 和 unittest 环境下为 baseDir，其他都为 HOME。 |
| env | local、prod |

appInfo.root比较优雅，比如在服务器环境我们会使用/home/admin/logs 作为日志目录，而本地开发时又不想污染用户目录，这样的适配就很好解决这个问题。

#### 配置加载顺序

优先级：应用 > 框架 > 插件

例如：

-> 插件 config.default.js

-> 框架 config.default.js

-> 应用 config.default.js

-> 插件 config.prod.js

-> 框架 config.prod.js

-> 应用 config.prod.js

注意：配置的合并，处理数组是会有差异，数组会直接覆盖而不是合并。

#### 配置结果

框架在启动的时候会把合并后的最终配置dump到run/application\_config.json（worker 进程）和 run/agent\_config.json（agent 进程）中，可以用来分析问题。

还会生成 run/application\_config\_meta.json（worker 进程）和 run/agent\_config\_meta.json（agent 进程）文件，用来排查属性的来源

### 中间件

#### 写法

|  |
| --- |
| // options === app.config.robot  module.exports = (options, app) => {  return async function robotMiddleware(ctx, next) {  const source = ctx.get('user-agent') || '';  const match = options.ua.some(ua => ua.test(source));  if (match) {  ctx.status = 403;  ctx.message = 'Go away, robot.';  } else {  console.log('ooooooooooooooooooo');  await next();  console.log('hhhhhhhhhhhhhhhhhhh');  }  };  }; |

await next()前的代码，会先执行，然后进入路由，等接口返回后会继续执行await next()后的代码。

#### 配置

一个中间件是一个放在app/middleware目录下的单独文件，需要

exports一个方法，有两个参数：

options：app.config[${middlewareName]

app: 当前应用 Application 的实例

可以在config里定义该中间件的单独配置，然后在中间件里通过options来获取。

#### 使用

中间件编写完成后，我们还需要手动挂载

* + 1. 全局生效

在config.default.js中：

|  |
| --- |
| module.exports = {  // 配置需要的中间件，数组顺序即为中间件的加载顺序  middleware: [ 'gzip' ],  // 配置 gzip 中间件的配置  gzip: {  threshold: 1024, // 小于 1k 的响应体不压缩  },  }; |

该配置会在启动时合并到app.config.appMiddleware，并挂载到app.middleware上

* + 1. router中使用

只针对某一个接口，可以直接在 app/router.js 中实例化和挂载，如下：

这个时候需要手动加载中间件（调用中间件的方法，作为参数传入路由，并且需要手动传入配置参数，不会默认读取中间件配置）

|  |
| --- |
| module.exports = app => {  const gzip = app.middleware.gzip({ threshold: 1024 });  app.router.get('/needgzip', gzip, app.controller.handler);  }; |

* + 1. 在框架或插件中使用

|  |
| --- |
| // app.js  module.exports = app => {  // 在中间件最前面统计请求时间  app.config.coreMiddleware.unshift('report');  }; |

应用层定义的中间件（app.config.appMiddleware）和框架默认的中间件（app.config.coreMiddleware）都会被加载器加载，并挂载到app.middleware上。

#### 框架默认中间件

除了应用层中间件，框架自身也会加载许多中间件，所有这些自带中间件的配置都可以在配置中修改中间件的同名配置项进行修改；

注意：仅支持同名配置项的修改，如果应用层有自定义同名中间件，会在启动时报错。

因为框架和插件加载的中间件会在应用层中间件之前加载；

#### 使用Koa中间件

|  |
| --- |
| // app/middleware/compress.js  module.exports = require('koa-compress');  // config/config.default.js  module.exports = {  middleware: [ 'compress' ],  compress: {  threshold: 2048,  },  }; |

#### 通用配置

无论是应用层加载的中间件还是框架自带中间件，都支持几个通用的配置项：

* enable：控制中间件是否开启。 true/false
* match：设置只有符合某些规则的请求才会经过这个中间件。
* ignore：设置符合某些规则的请求不经过这个中间件。

match 和ignore不能同时设置，match表示匹配，ignore表示忽略

都支持三种配置方式：

* 字符串：匹配的是url 的路径前缀
* 正则：满足正则匹配的url
* 函数：将ctx作为参数传入，最终返回true/false来判断是否匹配。

### 路由

#### 定义

router.verb(['router-name', ]'path-match'[, middleware1, ..., middlewareN], app.controller.action);

* verb：http方法：支持head,options,get,post,put,patch,del,redirect
* router-name: 别名（可选）
* path-match: 路由url地址(也可以是正则表达式)
* middleware1 ： 可配置多个中间件，按顺序加载
* controller： 将理由映射到具体的controller上，两种写法：

app.controller.user.fetch

'user.fetch'

#### 重定向

* + 1. 内部重定向

|  |
| --- |
| // app/router.js  module.exports = app => {  app.router.get('index', '/home/index', app.controller.home.index);  app.router.redirect('/', '/home/index', 302);  }; |

* + 1. 外部重定向

|  |
| --- |
| // app/router.js  module.exports = app => {  app.router.get('/search', app.controller.search.index);  };  // app/controller/search.js  exports.index = async ctx => {  const type = ctx.query.type;  const q = ctx.query.q || 'nodejs';  if (type === 'bing') {  ctx.redirect(`http://cn.bing.com/search?q=${q}`);  } else {  ctx.redirect(`https://www.google.co.kr/search?q=${q}`);  }  }; |

* + 1. 中间件使用

|  |
| --- |
| // app/controller/search.js  exports.index = async ctx => {  ctx.body = `search: ${ctx.query.name}`;  };  // app/middleware/uppercase.js  module.exports = () => {  return async function uppercase(ctx, next) {  ctx.query.name = ctx.query.name && ctx.query.name.toUpperCase();  await next();  };  };  // app/router.js  module.exports = app => {  // 中间件需要调用一下  app.router.get('s', '/search', app.middlewares.uppercase(), app.controller.search)  }; |

### 控制器

#### 属性

this.ctx:当前请求的上下文Context实例，有肯多便捷属性和方法

this.app:当前应用的Application实例，可拿全局对象和方法

this.service:可以访问所有service，等价于this.ctx.service

this.config: 访问应用运行时的配置项

this.logger:logger对象，有debug，info，warn，error四个方法

#### 获取http请求参数

* + 1. query

ctx.query 若有重复key，只获取第一个

* + 1. queries

ctx.queries 会获取所有的key,包括重复key,key的值是个数组

* + 1. param

ctx.params

* + 1. body

Web 开发中数据传递最常用的两类格式分别是 JSON 和 Form。

框架内置了 bodyParser 中间件来对这两类格式的请求 body 解析成 object 挂载到 ctx.request.body 上。

* 当请求的 Content-Type 为 application/json，application/json-patch+json，application/vnd.api+json 和 application/csp-report 时，会按照 json 格式对请求 body 进行解析，并限制 body 最大长度为 100kb。
* 当请求的 Content-Type 为 application/x-www-form-urlencoded 时，会按照 form 格式对请求 body 进行解析，并限制 body 最大长度为 100kb。
* 解析成功后，body会是个Object或Array
  + 1. header
* ctx.headers，ctx.header，ctx.request.headers，ctx.request.header都能获取整个header对象
* ctx.get(name)，ctx.request.get(name)：获取header对象中的一个字段，不存在会返回空字符串。
* 建议使用**ctx.get(name)**，因为它会自动处理大小写
* ctx.set(name, value) 设置一个请求头；

ctx.set(headers)设置多个请求头

* + 1. cookie

ctx.cookies.set(name, value) 设置cookie

ctx.cookies.get(name) 获取cookie值

* + 1. session

ctx.session.userId = null //设置session的值

ctx.session.userId // 获取session的值

#### 发送http响应

* + 1. 设置status

this.ctx.status = 201;

* + 1. 设置body

ctx.body = obj;

注意：ctx.body 是 ctx.response.body 的简写，不要和 ctx.request.body 混淆了。

### Service

#### 属性

同controller属性：this.ctx，this.app，this.service，this.config，this.logger

#### 使用

|  |
| --- |
| // app/service/user.js  const Service = require('egg').Service;  class UserService extends Service {  // 默认不需要提供构造函数。  // constructor(ctx) {  // super(ctx); 如果需要在构造函数做一些处理，一定要有这句话，才能保证后面 `this.ctx`的使用。  // // 就可以直接通过 this.ctx 获取 ctx 了  // // 还可以直接通过 this.app 获取 app 了  // }  async find(uid) {  const user = await this.ctx.db.query('select \* from user where uid = ?', uid);  return something;  } |

### 定时任务

#### 编写

所有的定时任务都统一存放在 app/schedule 目录下。

举例：

|  |
| --- |
| const Subscription = require('egg').Subscription;  class UpdateCache extends Subscription {  // 通过 schedule 属性来设置定时任务的执行间隔等配置  // 该属性名是固定的，不能改  static get schedule() {  return {  interval: '1m', // 1 分钟间隔  // cron: '\*/10 \* \* \* \*', crontab写法  type: 'all', // 指定所有的 worker 都需要执行  };  }  // subscribe 是真正定时任务执行时被运行的函数（该名称是固定的，不能改）  async subscribe() {  const res = await this.ctx.curl('http://www.api.com/cache', {  dataType: 'json',  });  this.ctx.app.cache = res.data;  }  }  module.exports = UpdateCache; |

还可以简写：

|  |
| --- |
| module.exports = {  schedule: {  interval: '1m', // 1 分钟间隔  type: 'all', // 指定所有的 worker 都需要执行  },  // task是固定的名称，不可更改  async task(ctx) {  const res = await ctx.curl('http://www.api.com/cache', {  dataType: 'json',  });  ctx.app.cache = res.data;  },  }; |

#### 其他参数

immediate：配置了该参数为 true 时，这个定时任务会在应用启动并 ready 后立刻执行一次这个定时任务。

disable：为true时，该任务不会启动

env：数组，仅在指定环境下执行

#### 手动执行定时任务

app.runSchedule(schedulePath)

schedulePath：app/schedule 目录下的相对路径或者完整的绝对路径

### 自定义启动

|  |
| --- |
| // app.js  class AppBootHook {  constructor(app) {  this.app = app;  }    configWillLoad() {  // 此时 config 文件已经被读取并合并，但是还并未生效  // 这是应用层修改配置的最后时机  // 注意：此函数只支持同步调用    // 例如：参数中的密码是加密的，在此处进行解密  this.app.config.mysql.password = decrypt(this.app.config.mysql.password);  }    async didLoad() {  // 所有的配置已经加载完毕  // 可以用来加载应用自定义的文件，启动自定义的服务    // 例如：创建自定义应用的示例  this.app.queue = new Queue(this.app.config.queue);  await this.app.queue.init();    // 例如：加载自定义的目录  this.app.loader.loadToContext(path.join(\_\_dirname, 'app/tasks'), 'tasks', {  fieldClass: 'tasksClasses',  });  }    async willReady() {  // 所有的插件都已启动完毕，但是应用整体还未 ready  // 可以做一些数据初始化等操作，这些操作成功才会启动应用    // 例如：从数据库加载数据到内存缓存  this.app.cacheData = await this.app.model.query(QUERY\_CACHE\_SQL);  }    async didReady() {  // 应用已经启动完毕    }    async serverDidReady() {  // http / https server 已启动，开始接受外部请求  // 此时可以从 app.server 拿到 server 的实例    this.app.server.on('timeout', socket => {  // handle socket timeout  });  }  }    module.exports = AppBootHook; |

### 框架扩展

#### Application

app对象指的是Koa的全局应用对象，全局只有一个，在应用启动时创建。

访问方式：

* ctx.app
* Controller，Middleware，Helper，Service 中用this.app访问
* 在app.js中 app对象会以第一个参数注入到入口函数

扩展：

框架会把 app/extend/application.js 中定义的对象与 Koa Application 的 prototype 对象进行合并，在应用启动时会基于扩展后的 prototype 生成 app 对象。

方法扩展：

|  |
| --- |
| // app/extend/application.js  module.exports = {  foo(param) {  // this 就是 app 对象，在其中可以调用 app 上的其他方法，或访问属性  },  }; |

属性扩展：

|  |
| --- |
| // app/extend/application.js  const BAR = Symbol('Application#bar');  module.exports = {  get bar() {  // this 就是 app 对象，在其中可以调用 app 上的其他方法，或访问属性  if (!this[BAR]) {  // 实际情况肯定更复杂  this[BAR] = this.config.xx + this.config.yy;  }  return this[BAR];  },  }; |

#### Context

每次请求生成一个Context实例

访问方式：

* middleware 中 this 就是 ctx，例如 this.cookies.get('foo')。
* controller有两种写法，类的写法通过this.ctx获取，方法的写法直接通过ctx入参
* helper、service中的this执行helper、service对象本身，使用this.ctx访问context；

扩展方式：

框架会把app/extend/context.js中定义的对象与Koa Context进行合并，生成新的ctx对象。

方法扩展：

|  |
| --- |
| // app/extend/context.js  module.exports = {  foo(param) {  // this 就是 ctx 对象，在其中可以调用 ctx 上的其他方法，或访问属性  },  }; |

属性扩展：

|  |
| --- |
| // app/extend/context.js  const BAR = Symbol('Context#bar');  module.exports = {  get bar() {  // this 就是 ctx 对象，在其中可以调用 ctx 上的其他方法，或访问属性  if (!this[BAR]) {  // 例如，从 header 中获取，实际情况肯定更复杂  this[BAR] = this.get('x-bar');  }  return this[BAR];  },  }; |

#### Request

访问方式: ctx.request

扩展方式：

|  |
| --- |
| // app/extend/request.js  module.exports = {  get foo() {  // this即request对象  return this.get('x-request-foo');  },  }; |

Koa内置的request属性和方法：https://koajs.com/#request-aliases

#### Response

访问方式：ctx.response

扩展方式：

|  |
| --- |
| // app/extend/response.js  module.exports = {  set foo(value) {  // this即response对象  this.set('x-response-foo', value);  },  }; |

Koa 内置的 response 的属性和方法列表：<https://koajs.com/#response-aliases>

#### Helper

访问方式：ctx.helper

扩展方式：

|  |
| --- |
| // app/extend/helper.js  module.exports = {  foo(param) {  // this 是 helper 对象，在其中可以调用其他 helper 方法  // this.ctx => context 对象  // this.app => application 对象  },  }; |

#### 按照环境扩展

|  |
| --- |
| // app/extend/application.unittest.js  module.exports = {  mockXX(k, v) {  }  }; |

只在unittest环境中提供mockXX方法；