### 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。

#### 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