- 介绍
- 特性
- 环境搭建、创建、运行
- 目录结构介绍
- 主要内容介绍
  - 什么是MVC
  - 控制器(controller)
  - 服务(service)
  - 路由器(routes)
- 项目实战演示
  - 使用插件 `egg-mongoose` 链接数据库
  - 创建用户模型
  - 创建用户
  - 获取所有用户
  - 根据id获取用户详情
  - 更新用户
  - 删除用户
- 中间件
  - 配置
  - 使用中间件
- 插件
  - 为什么要使用插件
  - 中间件、插件、应用的关系
  - 使用插件
- 框架扩展
- 定时任务
    - 编写定时任务
    - 定时方式
    - interval
    - cron
    - 类型
    - 其他参数
    - 动态配置定时任务

介绍

Eggjs是一个基于Koajs的框架，所以它应当属于框架之上的框架，它继承了Koa的高性能优点，同时又加入了一些约束与开发规范，来规避Koajs框架本身的开发自由度太高的问题。

约定大于配置

Koajs是一个nodejs中比较基层的框架，它本身没有太多约束与规范，自由度非常高，每一个开发者实现自己的服务的时候，都有自己的“骚操作”。而egg为了适应企业开发，加了一些开发时的规范与约束，从而解决Koajs这种自由度过高而导致不适合企业内使用的缺点，Egg便在这种背景下诞生。

Egg是由阿里巴巴团队开源出来的一个“蛋”，为什么是个蛋？蛋是有无限可能的，鸡孵出的蛋生小鸡，恐龙孵出来的蛋就是恐龙，这也正更好的体现了egg最大的一个亮点“插件机制”，每个公司每个团队甚至单个开发者都可以在这之上孵化出最适合自己的框架。像阿里内部不同的部门之间都孵化出了合适自己的egg框架，如蚂蚁的chair，UC的Nut，阿里云的aliyun-egg等，可以看下面这张图。

特性

- 提供基于 Egg 定制上层框架的能力
- 高度可扩展的插件机制
- 内置多进程管理
- 基于 Koa 开发，性能优异
- 框架稳定，测试覆盖率高
- 渐进式开发

环境搭建、创建、运行

```
$ npm i egg-init -g
$ mkdir egg-example && cd egg-example
$ npm init egg --type=simple
$ npm i
```

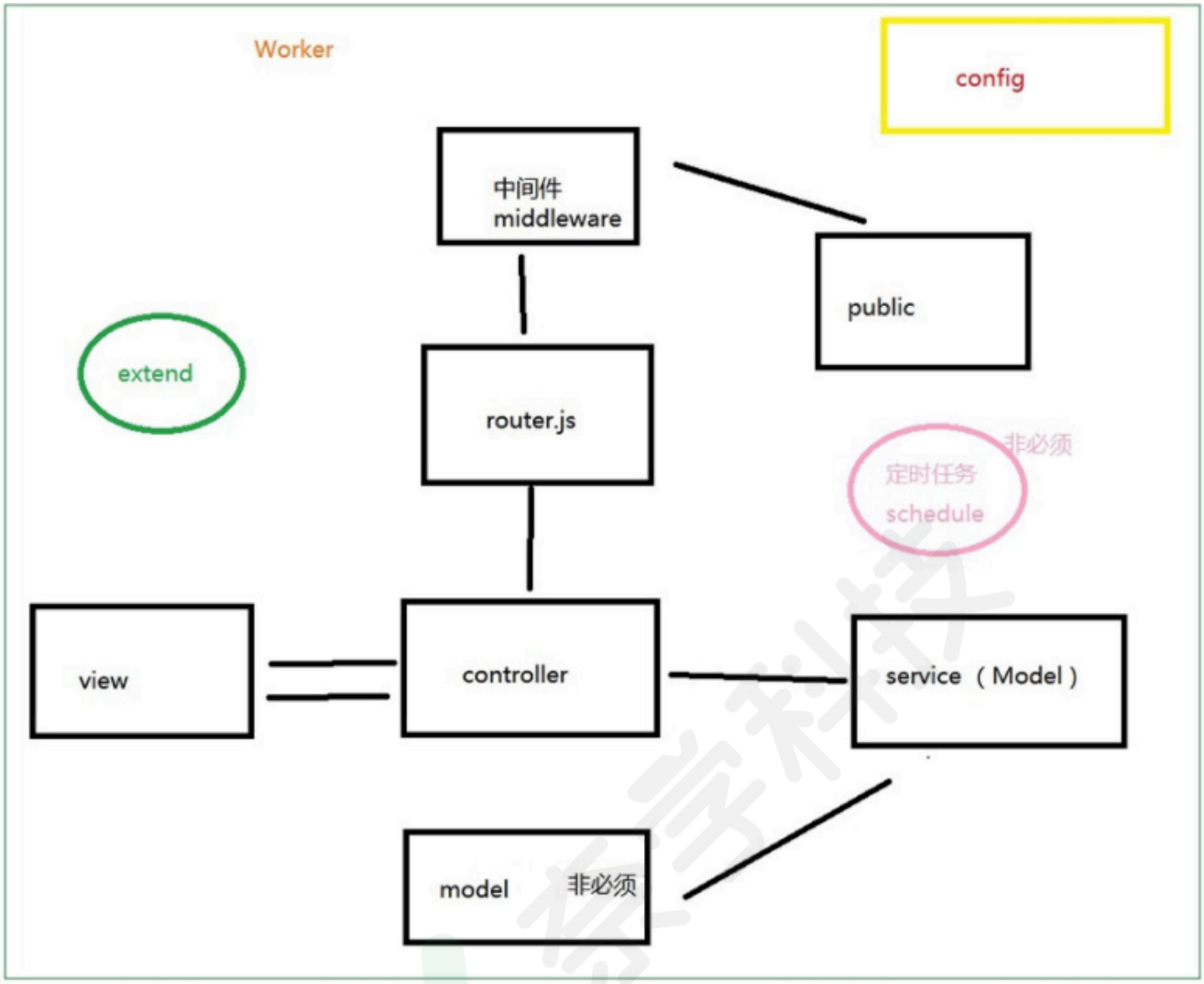
启动项目

```
$ npm run dev
$ gooopen http://localhost:7001
```

目录结构介绍

```
egg-project
├── package.json
├── app.js (可选)
├── agent.js (可选)
├── app(项目开发目录)
|   ├── router.js (用于配置 URL 路由规则)
|   ├── controller (用于解析用户的输入，处理后返回相应的结果)
|   |   └── home.js
|   ├── service (用于编写业务逻辑层)
|   |   └── user.js
|   ├── middleware (用于编写中间件)
|   |   └── response_time.js
|   ├── schedule (可选)
|   |   └── my_task.js
|   ├── public (用于放置静态资源)
|   |   └── reset.css
|   ├── view (可选)
```

```
|   |   └── home.tpl
|   └── extend (用于框架的扩展)
|       ├── helper.js (可选)
|       ├── request.js (可选)
|       ├── response.js (可选)
|       ├── context.js (可选)
|       ├── application.js (可选)
|       └── agent.js (可选)
├── config (用于编写配置文件)
|   ├── plugin.js(用于配置需要加载的插件)
|   ├── config.default.js
|   ├── config.prod.js
|   ├── config.test.js (可选)
|   ├── config.local.js (可选)
|   └── config.unittest.js (可选)
└── test (用于单元测试)
    ├── middleware
    |   └── response_time.test.js
    └── controller
        └── home.test.js
```

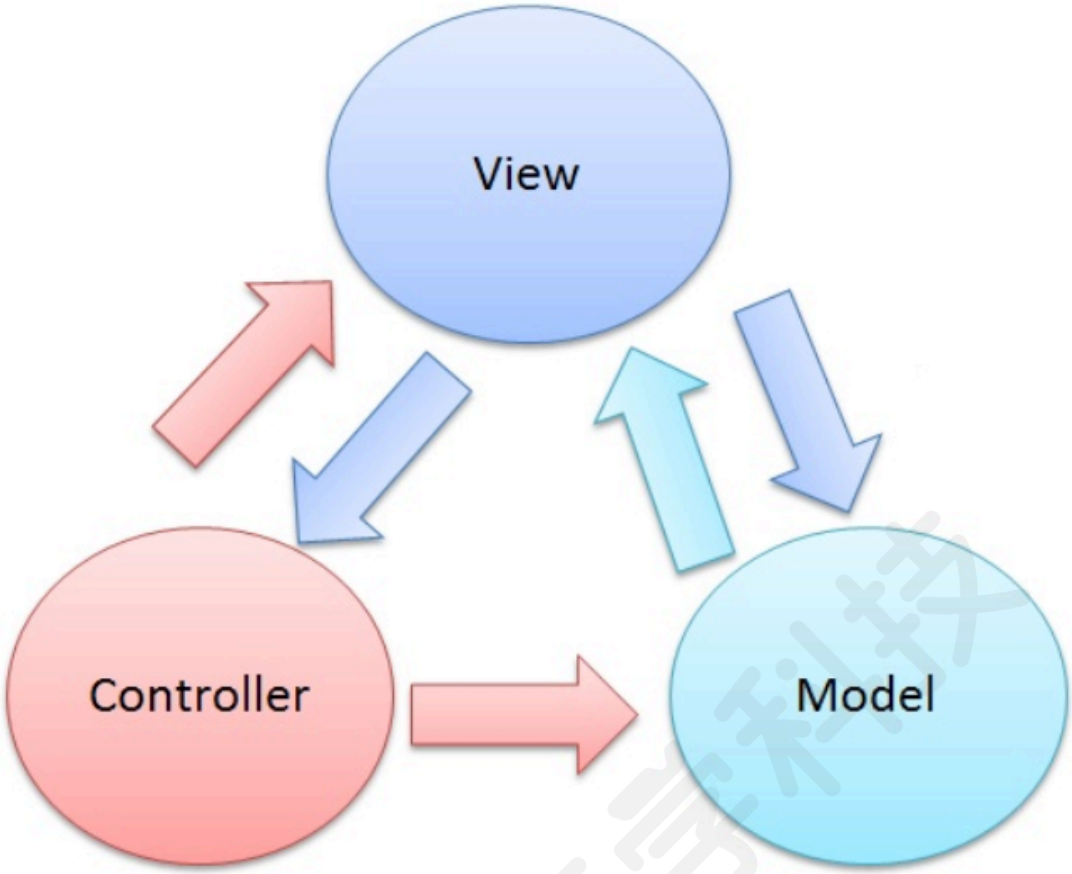


约定大于配置

主要内容介绍

什么是MVC

egg的设计完全符合比较好的mvc的设计模式



- **Model（模型）** - 模型代表一个存取数据的对象。它也可以带有逻辑，在数据变化时更新控制器。
- **View（视图）** - 视图代表模型包含的数据的可视化。
- **Controller（控制器）** - 控制器作用于模型和视图上。它控制数据流向模型对象，并在数据变化时更新视图。它使视图与模型分离开。

控制器(controller)

`app/controller`目录下面实现Controller

```
'use strict';

const Controller = require('egg').Controller;


class HomeController extends Controller {
  async index() {
    const { ctx } = this;
    ctx.body = 'hi, egg';
  }
}

module.exports = HomeController;
```

服务(service)

```
'use strict';

const Service = require('egg').Service;

class HomeService extends Service {
  async index() {
    return {ok:1}
  }
}

module.exports = HomeService;
```

修改 `controller/home.js`

```
'use strict';

const Controller = require('egg').Controller;

class HomeController extends Controller {
  async index() {
    const {
      ctx,
      service
    } = this;
    const res = await service.home.index();
    ctx.body = res
```

```
  }
}

module.exports = HomeController;
```

路由器(routes)

```
'use strict';

/**
 * @param {Egg.Application} app - egg application
 */
module.exports = app => {
  const { router, controller } = app;
  router.get('/', controller.home.index);
};
```

访问：http://locoalhost:7001

```
hi, egg
```

项目实战演示

针对用户表的增删改查操作

案例基于mongoose非关系型数据库

使用插件egg-mongoose链接数据库

下载

```
npm i egg-mongoose -S
```

配置

`config/plugin.js`

```
exports.mongoose = {
  enable: true,
  package: 'egg-mongoose',
};
```

config/config.default.js

```
config.mongoose = {
    url: "mongodb://127.0.0.1:27017/egg-test",
    options:{
        useUnifiedTopology: true,
        useCreateIndex:true
    }
}
```

创建用户模型

model/user.js

```
module.exports = app => {
  const mongoose = app.mongoose;
  const UserSchema = new mongoose.Schema({
    username: {
      type: String,
      required: true
    },
```

```
    password: {
      type: String,
      required: true
    },
    avatar: {
      type: String,
      default:
'https://1.gravatar.com/avatar/a3e54af3cb6e157e
496ae430aed4f4a3?s=96&d=mm'
    },
    createdAt: {
      type: Date,
      default: Date.now
    }
  })
  return mongoose.model('User', UserSchema);
}
```

创建用户

router.js

```
// 用户创建
 router.post('/api/user',controller.user.create
);
```

controller/user.js

```
//创建用户
async create() {
    const {
      ctx,
      service
    } = this;
    const payLoad = ctx.request.body || {};
    const res = await
service.user.create(payLoad);
    ctx.body = {res};
 }
```

service/user.js

```
async create(payload) {
    const {
        ctx
    } = this;
    return ctx.model.User.create(payload);
}
```

获取所有用户

router.js

```
router.get('/api/user',controller.user.index);
```

controller/user.js

```
// 获取所有用户
async index() {
    const {
        ctx,
        service
    } = this;
    const res = await service.user.index();
    ctx.body = res;
}
```

service/user.js

```
async index() {
    const {
        ctx
    } = this;
    return ctx.model.User.find();
}
```

根据id获取用户详情

router.js

```
// 根据id获取用户详情
router.get('/api/user/:id',controller.user.detail);
```

controller/user.js

```
async detail() {
    const id = this.ctx.params.id;
    const res = await this.service.user.detail(id);
    ctx.body = res;
}
```

service/user.js

```
async detail(id){
   return this.ctx.model.User.findById({_id:id})
}
```

更新用户

router.js

```
// 修改用户
router.put('/api/user/:id',controller.user.update);
```

controller/user.js

```
async update() {
    const id = this.ctx.params.id;
    const payLoad = this.ctx.request.body;
    // 调用 Service 进行业务处理
    await this.service.user.update(id, payLoad);
    // 设置响应内容和响应状态码
    ctx.body = {msg:'修改用户成功'};
}
```

service/user.js

```
async update(_id, payLoad) {
    return this.ctx.model.User.findByIdAndUpdate(_id,payLoad);
}
```

删除用户

router.js

```
// 删除用户
router.delete('/api/user/:id',controller.user.delete);
```

controller/user.js

```
  async delete() {
    const id = this.ctx.params.id;
     // 调用 Service 进行业务处理
    await this.service.user.delete(id);
     // 设置响应内容和响应状态码
    ctx.body = {msg:"删除用户成功"};
  }
```

service/user.js

```
async delete(_id){
    return this.ctx.model.User.findByIdAndDelete(_id);
}
```

中间件

配置

一般来说中间件也会有自己的配置。在框架中，一个完整的中间件是包含了配置处理的。我们**约定**一个中间件是一个放置在 `app/middleware` 目录下的单独文件，它需要 `exports` 一个普通的 `function`，接受两个参数：

- `options`：中间件的配置项，框架会将 `app.config[${middlewareName}]` 传递进来。
- `app`：当前应用 `Application` 的实例。

```
module.exports = (option, app) => {
  return async function (ctx, next) {
    try {
      await next();
    } catch (err) {
      // 所有的异常都在app上触发一个error事件，框架会记录一条错误日志
      app.emit('error', err, this);
      const status = err.status || 500;
      // 生成环境下 500 错误的详细错误内容不返回给客户端，因为可能包含敏感信息
      const error = status === 500 && app.config.env === 'prod' ? 'Internal Server Error' : err.message
      // 从error对象上读出各个属性，设置到响应中
      ctx.body = {
```

```
13          code: status, // 服务端自身的处理逻辑错误(包含框架错误500 及 自定义业务逻辑错误533开始 ) 客户端请求参数导致的错误(4xx开始), 设置不同的状态码
14          error:error
15        }
16        if(status === 422){
17          ctx.body.detail = err.errors;
18        }
19        ctx.status = 200
20      }
21    }
22  }
```

使用中间件

中间件编写完成后，我们还需要**手动挂载**，支持以下方式：

在应用中，我们可以完全通过配置来加载自定义的中间件，并决定它们的顺序。

如果我们需要加载上面的 `error_handler` 中间件，在 `config.default.js` 中加入下面的配置就完成了中间件的开启和配置：

```
1 // add your middleware config here
2 config.middleware = ['errorHandler'];
```

插件

插件机制是我们框架的一大特色。它不但可以保证框架核心的足够精简、稳定、高效，还可以促进业务逻辑的复用，生态圈的形成。有人可能会问了

- Koa 已经有了中间件的机制，为啥还要插件呢?
- 中间件、插件、应用它们之间是什么关系，有什么区别?
- 我该怎么使用一个插件?
- 如何编写一个插件?
- ...

接下来我们就来逐一讨论

为什么要使用插件

我们在使用 Koa 中间件过程中发现了下面一些问题：

1. 中间件加载其实是有先后顺序的，但是中间件自身却无法管理这种顺序，只能交给使用者。这样其实非常不友好，一旦顺序不对，结果可能有天壤之别。
2. 中间件的定位是拦截用户请求，并在它前后做一些事情，例如：鉴权、安全检查、访问日志等等。但实际情况是，有些功能是和请求无关的，例如：定时任务、消息订阅、后台逻辑等等。
3. 有些功能包含非常复杂的初始化逻辑，需要在应用启动的时候完成。这显然也不适合放到中间件中去实现。

综上所述，我们需要一套更加强大的机制，来管理、编排那些相对独立的业务逻辑。

中间件、插件、应用的关系

一个插件其实就是一个『迷你的应用』，和应用（app）几乎一样：

- 它包含了 Service、中间件、配置、框架扩展等等。
- 它没有独立的 Router 和 Controller。
- 它没有 `plugin.js`，只能声明跟其他插件的依赖，而**不能决定**其他插件的开启与否。

他们的关系是：

- 应用可以直接引入 Koa 的中间件。
- 当遇到定时任务、消息订阅、后台逻辑这些场景时，则应用需引入插件。
- 插件本身可以包含中间件。
- 多个插件可以包装为一个上层框架。

使用插件

上面我们使用的 `egg-mongoose` 就是一个插件。

插件一般通过 npm 模块的方式进行复用：

```
npm i egg-validate -S
```

然后需要在应用或框架的 `config/plugin.js` 中声明：

```
exports.validate = {
  enable: true,
  package: 'egg-validate',
};
```

就可以直接使用插件提供的功能：

`controller/user.js`

```
'use strict';

const Controller = require('egg').Controller;

```

```
class UserController extends Controller {
  constructor(props) {
    super(props);
    this.UserCreateRule = {
      username: {
        type: 'string',
        required: true,
        allowEmpty: false,
        // 用户名必须是3-10位之间的字母、下划线、@、. 并且不能以数字开头
        format: /^[A-Za-z_@.]{3,10}/
      },
      password: {
        type: 'password',
        require: true,
        allowEmpty: false,
        min: 6
      }
    }
  }

  async create() {
    const {
      ctx,
      service
    } = this;
```

```
30      // 校验参数
31      ctx.validate(this.UserCreateRule)
32      const payLoad = ctx.request.body || {};
33      const res = await
    service.user.create(payLoad);
34      this.ctx.helper.success({
35        ctx: this.ctx,
36        res
37      });
38    }
39
40  }
41
42  module.exports = UserController;
```

框架扩展

Helper 函数用来提供一些实用的 utility 函数。

它的作用在于我们可以将一些常用的动作抽离在 helper.js 里面成为一个独立的函数，这样可以用 JavaScript 来写复杂的逻辑，避免逻辑分散各处。另外还有一个好处是 Helper 这样一个简单的函数，可以让我们更容易编写测试用例。

框架内置了一些常用的 Helper 函数。我们也可以编写自定义的 Helper 函数。

框架会把 `app/extend/helper.js` 中定义的对象与内置 `helper` 的 prototype 对象进行合并，在处理请求时会基于扩展后的 prototype 生成 `helper` 对象。

例如，增加一个 `helper.success()` 方法：

`extend/helper.js`

```
module.exports = {
    success:function({res=null,msg='请求成功'}){
        // this是helper对象，在其中可以调用其他的helper方法
        // this.ctx =>context对象
        // this.app =>application对象
        this.ctx.body = {
            code:200,
            data:res,
            msg
        }
        this.ctx.status = 200;
    }
}
```

`controller/user.js`

```
async index() {
    const res = await this.service.user.index();
    this.ctx.helper.success({
        res
    });

}
```

定时任务

虽然我们通过框架开发的 HTTP Server 是请求响应模型的，但是仍然还会有许多场景需要执行一些定时任务，例如：

1. 定时上报应用状态。（订单超时反馈，订单详情处理等）
2. 定时从远程接口更新本地缓存。
3. 定时进行文件切割、临时文件删除。

框架提供了一套机制来让定时任务的编写和维护更加优雅

编写定时任务

所有的定时任务都统一存放在 `app/schedule` 目录下，每一个文件都是一个独立的定时任务，可以配置定时任务的属性和要执行的方法。

一个简单的例子，我们定义一个更新远程数据到内存缓存的定时任务，就可以在 `app/schedule` 目录下创建一个 `update_cache.js` 文件

```
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
  static get schedule() {
    return {
      interval: '5s', // 1 分钟间隔
      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
  async subscribe() {
    console.log("任务执行 : " + new Date().toString());

    // const res = await this.ctx.curl('https://free-api.heweather.net/s6/weather/now?location=beijing&key=4693ff5ea653469f8bb0c29638035976', {
    //   dataType: 'json',
    // })
    // this.ctx.app.cache = res.data;
  }
}
module.exports = UpdateCache;

```

```
22
```

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
  },
  async task(ctx) {
    const res = await ctx.curl('https://free-api.heweather.net/s6/weather/now?location=beijing&key=4693ff5ea653469f8bb0c29638035976', {
      dataType: 'json',
    });
    ctx.app.cache = res.data;
  },
};

```

这个定时任务会在每一个 Worker 进程上每 1 分钟执行一次，将远程数据请求回来挂载到 app.cache 上。

定时方式

定时任务可以指定 interval 或者 cron 两种不同的定时方式。

interval

通过 `schedule.interval` 参数来配置定时任务的执行时机，定时任务将会每间隔指定的时间执行一次。interval 可以配置成

- 数字类型，单位为毫秒数，例如 `5000`。
- 字符类型，会通过 `ms` 转换成毫秒数，例如 `5s`。

```
module.exports = {
  schedule: {
    // 每 10 秒执行一次
    interval: '10s',
  },
};
```

cron

通过 `schedule.cron` 参数来配置定时任务的执行时机，定时任务将会按照 cron 表达式在特定的时间点执行。cron 表达式通过 `cron-parser` 进行解析。

注意：cron-parser 支持可选的秒（linux crontab 不支持）。

```
*    *    *    *    *    *
T    T    T    T    T    T
|    |    |    |    |    |


|    |    |    |    |    └ day of week (0 - 7) (0 or 7 is Sun)
|    |    |    |    └──── month (1 - 12)
|    |    |    └─────────── day of month (1 - 31)
|    |    └──────────────── hour (0 - 23)
|    └───────────────────── minute (0 - 59)
└────────────────────────── second (0 - 59, optional)
```

```
module.exports = {
  schedule: {
    // 每三小时准点执行一次
    cron: '0 0 */3 * * *',
  },
};
```

类型

框架提供的定时任务默认支持两种类型，`worker` 和 `all`。`worker` 和 `all` 都支持上面的两种定时方式，只是当到执行时机时，会执行定时任务的 `worker` 不同：

- `worker` 类型：每台机器上只有一个 worker 会执行这个定时任务，每次执行定时任务的 worker 的选择是随机的。
- `all` 类型：每台机器上的每个 worker 都会执行这个定时任务。

其他参数

除了刚才介绍到的几个参数之外，定时任务还支持这些参数：

- `cronOptions`：配置 cron 的时区等，参见 cron-parser 文档
- `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`：配置该参数为 true 时，这个定时任务不会被启动。
- `env`：数组，仅在指定的环境下才启动该定时任务。

动态配置定时任务

`config/config.default.js`

```
config.cacheTick = {
    interval: '5s', // 1 分钟间隔
    type: 'all', // 指定所有的 worker 都需要执行
    immediate: true, //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: true, //为true表示定时任务不会被启动
  };
```

`schedule/update_cache.js`

```
module.exports = app => {
  return {
    schedule: app.config.cacheTick,
    async task(ctx) {
      console.log("任务执行 : " + new Date().toString());
    },
  }
};
```

启动项目，查看控制台输出。