

ThinkJS 2.2 Documentation

快速入门

介绍

ThinkJS 是一款使用 ES6/7 特性全新开发的 Node.js MVC 框架，使用 ES7 中 `async/await`，或者 ES6 中的 `*/yield` 特性彻底解决了 Node.js 中异步嵌套的问题。同时吸收了国内外众多框架的设计理念和思想，让开发 Node.js 项目更加简单、高效。

使用 ES6/7 特性来开发项目可以大大提高开发效率，是趋势所在。并且新版的 Node.js 对 ES6 特性也有了较好的支持，即使有些特性还没有支持，也可以借助 [Babel](#) 编译来支持。

特性

使用 ES6/7 特性来开发项目

借助 Babel 编译，可以在项目中大胆使用 ES6/7 所有的特性，无需担心哪些特性当前版本不支持。尤其是使用 `async/await` 或者 `*/yield` 来解决异步回调的问题。

```
//user controller, home/controller/user.js
export default class extends think.controller.base {
  //login action
  async loginAction(self){
    //如果是get请求，直接显示登录页面
    if(this.isGet()){
      return this.display();
    }
    //这里可以通过post方法获取所有的数据，数据已经在logic里做了校验
    let data = this.post();
    //用户名去匹配数据库中对应的条目
    let result = await this.model('user').where({name: data.name}).find();
    if(!validateLogin(result)){
      return this.fail('login fail');
    }
    //获取到用户信息后，将用户信息写入session
```

```
    await this.session('userInfo', result);
    return this.success();
  }
}
```

上面的代码我们使用了 ES6 里的 `class`, `export`, `let` 以及 ES7 里的 `async/await` 等特性, 虽然查询数据库和写入 `Session` 都是异步操作, 但借助 `async/await`, 代码都是同步书写的。最后使用 `Babel` 进行编译, 就可以稳定运行在 Node.js 的环境中了。

支持 TypeScript

[TypeScript](#) 是一种由微软开发的自由和开源的编程语言。它是 JavaScript 的一个超集, 向这个语言添加了可选的静态类型, 在大型项目里非常有用。

ThinkJS 2.1 开始支持了创建 TypeScript 类型的项目, 并且开发时会自动编译、自动更新, 无需手工编译等复杂的操作。具体请见[这里](#)。

断点调试

从 ThinkJS 2.2.0 版本开始, 支持对 ES2015+ 和 TypeScript 项目的断点调试, 并且报错信息也会定位到源代码下, 这样可以给开发和调试带来巨大的便利, 具体请见[断点调试](#)。

支持多种项目结构和多种项目环境

项目支持单模块模式、普通模式、分模块模式等多种项目结构, 可以满足各种项目复杂度的开发。

默认支持 `development`, `testing` 和 `production` 3 种项目环境, 可以在不同的项目环境下进行不同的配置, 满足在不同环境下的配置需求, 同时还可以基于项目需要进行扩展。

支持丰富的数据库

ThinkJS 支持 `mysql`, `mongodb`, `sqlite` 等常见的数据库, 并且封装了很多操作数据库的接口, 无需手动拼接 SQL 语句, 还可以自动防止 SQL 注入等安全漏洞。同时支持事务、关联模型等高级功能。

代码自动更新

ThinkJS 内置了一套代码自动更新的机制, 文件修改后立即生效, 不用重启 Node.js 服务, 也不用借助第三方模块。

自动创建 REST 接口

使用 `thinkjs` 命令可以自动创建 REST 接口，不用写任何的代码即可完成 REST API 的开发。如果想在 REST 接口中过滤字段或者进行权限校验，也很方便处理。

支持多种 WebSocket 库

ThinkJS 支持 `socket.io`，`sockjs` 等常见的 WebSocket 库，并且对这些库进行包装，抹平各个库之间接口调用上的差异，给开发者一致的体验。

丰富的测试用例

ThinkJS 含有 1500+ 的测试用例，代码覆盖率达到 95%，每一次修改都有对应的测试用例来保障框架功能的稳定。

支持命令行调用执行定时任务

ThinkJS 里的 `Action` 除了可以响应用户的请求，同时支持在命令行下访问，借助这套机制就可以很方便的执行定时任务。

Hook 和 Middleware

ThinkJS 使用 Hook 和 Middleware 机制，可以灵活的对访问请求进行拦截处理。

详细的日志

ThinkJS 内置了详细的日志功能，可以很方便的查看各种日志，方便追查问题。

HTTP 请求日志

```
[2015-10-12 14:10:03] [HTTP] GET /favicon.ico 200 5ms
[2015-10-12 14:10:11] [HTTP] GET /zh-cn/doc.html 200 11ms
[2015-10-12 14:10:11] [HTTP] GET /static/css/reset.css 200 3ms
```

Socket 连接日志

```
[2015-10-12 14:13:54] [SOCKET] Connect mysql with mysql://root:root@127.0.0.1:3306
```

错误日志

```
[2015-10-12 14:15:32] [Error] Error: ER_ACCESS_DENIED_ERROR: Access denied for user 'root3'@'localhost' (using password: YES)
```

```
[2015-10-12 14:16:12] [Error] Error: Address already in use, port:8360. http://www.thinkjs.org/doc/error.html#EADDRINUSE
```

丰富的路由机制

ThinkJS 支持正则路由、规则路由、静态路由等多种路由机制，并且可以基于模块来设置。可以让 URL 更加简洁的同时又不丢失性能。

支持国际化和多主题

ThinkJS 使用很简单的方法就可以支持国际化和多主题等功能。

与其他框架的对比

与 express/koa 对比

express/koa 是 2 个比较简单的框架，框架本身提供的功能比较简单，项目中需要借助大量的第三方插件才能完成项目的开发，所以灵活度比较高。但使用很多第三方组件一方面提高了项目的复杂度，另一方面第三方插件质量参差不齐，也会带来内存泄漏等风险。

koa 1.x 使用 ES6 里的 `*/yield` 解决了异步回调的问题，但 `*/yield` 只会是个过渡解决方案，会被 ES7 里的 `async/await` 所替代。

而 ThinkJS 提供了整套解决方案，每个功能都经过了严格的性能和内存泄漏等方面的测试，并且在项目中可以直接使用 ES6/7 所有的特性。

与 sails 对比

sails 也是一个提供整套解决方案的 Node.js 框架，对数据库、REST API、安全方面也很多封装，使用起来比较方便。

但 sails 对异步回调的问题还没有优化，还是使用 callback 的方式，给开发带来很大的不便，导致项目中无法较好的使用 ES6/7 特性。

ThinkJS 的不足

上面说了很多 ThinkJS 的优点，当然 ThinkJS 也有很多的不足。如：

- 框架还比较新，缺少社区等方面的支持
- 还没有经过超大型项目的检验

性能对比

评价一个框架是否出色，一方面看支持的功能，另一方面也要看性能。虽然 ThinkJS 更适合大型项目，功能和复杂度远远超过 Express 和 Koa，但性能上并不比 Express 和 Koa 逊色多少，具体的测试数据请见下图。

任务名称	QPS	RT(ms)	成功率	起止时间
Node框架测试_Sails_2016-01-13 19:02:58	515	355.2	100%	2016-01-13 19:03:03 - 2016-01-13 19:03:23
Node框架测试_ThinkJS_2016-01-13 19:00:09	2001	83.04	100%	2016-01-13 19:01:02 - 2016-01-13 19:01:23
Node框架测试_Express_2016-01-13 18:52:16	2896	57.92	100%	2016-01-13 18:53:02 - 2016-01-13 18:53:24
Node框架测试_Koa_2016-01-13 18:47:55	2482	62.33	100%	2016-01-13 18:48:02 - 2016-01-13 18:48:24
Node框架测试_Node_2016-01-13 18:42:55	6390	24.61	100%	2016-01-13 18:43:02 - 2016-01-13 18:43:25

注：以上数据使用分布式压力测试系统测试。

从上图中测试数据可以看到，虽然 ThinkJS 比 Express 和 Koa 性能要差一些，但差别并不大。ThinkJS 和 Sails.js 都更符合大型项目，但 ThinkJS 的性能要比 Sails.js 高很多。

具体测试代码请见：<https://github.com/thinkjs-team/thinkjs-performance-test>，可以下载代码在本机测试，如果使用 **ab** 测试工具，请注意该工具在 Mac 系统下很不稳定，多次测试结果会相差很大。

ES6/7 参考文档

关于 ES6/7 特性可以参考下面的文档：

- [JavaScript Promise 迷你书](#)
- [learn-es2015](#)
- [ECMAScript 6 入门](#)
- [给 JavaScript 初心者的 ES2015 实战](#)
- [ECMAScript 6 Features](#)
- [ECMAScript 6 compatibility table](#)
- [ECMAScript 7 Features](#)
- [ECMAScript 7 compatibility table](#)

创建项目

安装 Node.js

ThinkJS 是一款 Node.js 的 MVC 框架，所以安装 ThinkJS 之前，需要先安装 Node.js 环境，可以去[官方](#) 下载最新的安装包进行安装，也可以通过其他一些渠道安装。

安装完成后，在命令行执行 `node -v`，如果能看到对应的版本号输出，则表示安装成功。

ThinkJS 需要 Node.js 的版本 `>=0.12.0`，如果版本小于这个版本，需要升级 Node.js，否则无法启动服务。建议将 Node.js 版本升级到 `4.2.1` 或更高版本。

安装 ThinkJS

通过下面的命令即可安装 ThinkJS:

```
npm install thinkjs@2 -g --verbose
```

如果安装很慢的话，可以尝试使用 [taobao](#) 的源进行安装。具体如下:

```
npm install thinkjs@2 -g --registry=https://registry.npm.taobao.org --verbose
```

安装完成后，可以通过 `thinkjs --version` 或 `thinkjs -V` 命令查看安装的版本。

注：如果之前安装过 ThinkJS 1.x 的版本，可能需要将之前的版本删除掉，可以通过 `npm uninstall -g thinkjs-cmd` 命令删除。

更新 ThinkJS

更新全局的 ThinkJS

执行下面的命令即可更新全局的 ThinkJS:

```
npm install -g thinkjs@2
```

更新项目里的 ThinkJS

在项目目录下，执行下面的命令即可更新当前项目的 ThinkJS:

```
npm install thinkjs@2
```

使用命令创建项目

ThinkJS 安装完成后，就可以通过下面的命令创建项目：

```
thinkjs new project_path; #project_path为项目存放的目录
```

注：从 `2.2.12` 版本开始，创建的项目默认为 ES6 模式，不再需要加 `--es` 参数，如果想创建一个 ES5 模式项目，需要加参数 `--es5`。

如果能看见类似下面的输出(下面截图里的demo就是上面的project_path)，表示项目创建成功了：

```
create : demo/
create : demo/package.json
create : demo/.thinksrc
create : demo/nginx.conf
create : demo/README.md
create : demo/www/
create : demo/www/index.js
create : demo/app
create : demo/app/common/runtime
create : demo/app/common/config
create : demo/app/common/config/config.js
create : demo/app/common/config/view.js
create : demo/app/common/config/db.js
...
create : demo/app/home/logic
create : demo/app/home/logic/index.js
create : demo/app/home/view
create : demo/app/home/view/index_index.html

enter path:
$ cd demo/

install dependencies:
$ npm install

run the app:
$ npm start
```

关于创建项目命令的更多信息，请见 [扩展功能 -> ThinkJS 命令](#)。

安装依赖

项目安装后，进入项目目录，执行 `npm install` 安装依赖，可以使用 `taobao` 源进行安装。

```
npm install --registry=https://registry.npm.taobao.org --verbose
```

启动项目

在项目目录下执行命令 `npm start`，如果能看到类似下面的内容，表示服务启动成功。

```
[2015-09-21 20:21:09] [THINK] Server running at http://127.0.0.1:8360/  
[2015-09-21 20:21:09] [THINK] ThinkJS Version: 2.0.0  
[2015-09-21 20:21:09] [THINK] Cluster Status: closed  
[2015-09-21 20:21:09] [THINK] WebSocket Status: closed  
[2015-09-21 20:21:09] [THINK] File Auto Reload: true  
[2015-09-21 20:21:09] [THINK] App Enviroment: development
```

访问项目

打开浏览器，访问 `http://127.0.0.1:8360/` 即可。

如果是在远程机器，需要通过远程机器的 IP 访问，同时要保证 8360 端口可访问。

如果你还没有自己的服务器，可以到[这里体验](#)一下搭建安装Thinkjs

项目结构

通过 thinkjs 命令创建完项目后，项目目录结构类似如下：

```
|-- nginx.conf  
|-- package.json  
|-- src  
|   |-- common  
|   |   |-- bootstrap  
|   |   |   |-- generate_icon.js  
|   |   |   |-- middleware.js  
|   |   |-- config  
|   |   |   |-- config.js  
|   |   |   |-- env  
|   |   |   |   |-- development.js  
|   |   |   |   |-- production.js  
|   |   |   |-- hook.js  
|   |   |   |-- locale  
|   |   |   |   |-- en.js  
|   |   |   |   |-- zh-cn.js
```

```

|-- route.js
|-- controller
|-- error.js
|-- runtime
|-- home
|-- config
|-- controller
|-- base.js
|-- index.js
|-- logic
|-- doc.js
|-- model
|-- view
|-- zh-cn
|-- common
|-- error_400.html
|-- error_403.html
|-- error_404.html
|-- error_500.html
|-- error_503.html
|-- home
|-- doc_index.html
|-- doc_search.html
|-- inc
|-- footer.html
|-- header.html
|-- index_changelog.html
|-- index_demo.html
|-- index_index.html
|-- www
|-- favicon.ico
|-- index.js
|-- production.js
|-- static
|-- css
|-- img
|-- js

```

注：指定不同的模式创建的项目目录机构可能有细微的差别，但总体是类似的。

nginx.conf

nginx 的配置文件，建议线上使用 nginx 做反向代理。

src

源代码目录，使用 **ES6** 模式创建项目才有该目录。项目启动时会自动将 **src** 目录下的文件编译到 **app** 目录下。

如果没有使用 ES6 特性创建项目，则直接有 `app/` 目录。

src/common

通用模块目录，项目目录都是按模块来划分的，`common` 模块下存放一些通用的处理逻辑。

src/common/bootstrap

项目启动目录，该目录下的文件会自动加载，无需手动 `require`。

可以在这个目录下文件里定义一些全局函数、注册中间件等常用的功能。

定义全局函数

```
// src/common/bootstrap/fn.js
global.formatDate = obj => {
  ...
}
```

这里定义了一个全局函数 `formatDate`，那么项目里任何地方都可以直接使用该函数。

注册中间件

```
// src/common/bootstrap/middleware.js
think.middleware('replace_image', http => {
  ...
});
```

这里定义了一个中间件 `replace_image`，那么就可以在配置文件 `hook.js` 里将该中间件注册进去了。

注：bootstrap 只能放在 common 模块里。

src/common/config

配置文件，这里放一些通用的配置。

其中：路由配置、hook 配置、本地化配置等必须放在这里。

```
'use strict';
/**
 * config
```

```
*/  
export default {  
  //key: value  
};
```

src/common/controller

控制器，放一些通用的控制器。其中 `error.js` 里错误处理的不同行为，项目里可以根据需要进行修改。

src/common/runtime

项目运行时生成的一些目录，如：缓存文件目录，用户上传的文件临时存放的目录。

src/home

`home` 模块，项目默认模块。可以在 `src/common/config/config.js` 中修改配置 `default_module` 来重新定义默认模块。

src/home/logic

逻辑处理。每个操作执行前可以先进行逻辑校验，可以包含：参数是否合法、提交的数据是否正常、当前用户是否已经登录、当前用户是否有权限等。这样可以降低 `controller` 里的 `action` 的复杂度。

```
'use strict';  
/**  
 * logic  
 * @param {} []  
 * @return {} []  
 */  
export default class extends think.logic.base {  
  /**  
   * index action logic  
   * @return {} []  
   */  
  indexAction(){  
  
  }  
}
```

src/home/controller

控制器。一个 `url` 对应一个 `controller` 下的 `action`。

```
'use strict';

import Base from './base.js';

export default class extends Base {
  /**
   * index action
   * @return {Promise} []
   */
  indexAction(){
    //auto render template file index_index.html
    return this.display();
  }
}
```

src/home/model

模型。数据库相关操作。

view

视图目录，存放对应的模版文件。如果支持国际化和多主题，那么视图目录下需要有对应的子目录。

www

项目的可访问根目录，nginx 里的根目录会配置到此目录下。

www/development.js

开发模式下项目的入口文件，可以根据项目需要进行修改。`www/production.js` 为线上的入口文件。

入口文件的代码类似如下，可以根据项目需要进行修改。

```
var thinkjs = require('thinkjs');
var path = require('path');
```

```
var rootPath = path.dirname(__dirname);

var instance = new thinkjs({
  APP_PATH: rootPath + '/app',
  ROOT_PATH: rootPath,
  RESOURCE_PATH: __dirname,
  env: 'development'
});

instance.compile({retainLines: true, log: true});

instance.run();
```

www/static

存放一些静态资源文件。

代码规范

文件路径必须小写

很多时候是在 `Windows` 或者 `Mac OSX` 系统下开发项目，但一般都部署 `Linux` 系统下。

在 `Windows` 和 `Mac` 系统下，文件路径是不区分大小写的，而 `Linux` 下是区分大小写的。这样很容易出现文件大小写的问题导致开发环境下是好的，但上线后却报错了。

为了避免这种情况的发生，文件路径尽量都使用小写字符。并且在服务启动时，ThinkJS 会检测项目下文件路径，如果有大写字符则会告警，如：

```
[2015-10-13 10:36:59] [WARNING] filepath `admin/controller/apiBase.js` has uppercases.
```

缩进使用 2 个空格

在 Node.js 环境下开发，有时候逻辑比较复杂，有各种条件判断，或者有一些异步操作，这些都会增加代码的缩进。

为了不至于让缩进占用了太多的列宽，建议使用 2 个空格作为缩进。

使用 ES6 语法开发

ES6 中有大量的语法糖可以简化我们的代码，让代码更加简洁高效。

Node.js 最新版本已经较好的支持了 ES6 的语法，即使有些语法不支持，也可以通过 Babel 编译来支持。所以是时候使用 ES6 语法来开发项目了。

不要使用 constructor 方法

使用 ES6 里的 class 来创建类的时候，可以使用 `constructor` 方法达到类实例化的时候自动调用。如：

```
export default class think.base {
  constructor(){
    ...
  }
}
```

但如果不使用 ES6 里的 class，就没有 `constructor` 方法了。

为了统一处理，ThinkJS 提供了 `init` 方法来代替 `constructor` 方法，该方法不管是在 class 下还是动态创建类的情况下都可以做到类实例化的时候自动被调用。

```
export default class think.base {
  /**
   * 初始化方法，类实例化时自动被调用
   * @return {} []
   */
  init(){
    ...
  }
}
```

注： ThinkJS 里所有的类都会继承 `think.base` 基类。

使用 Babel 编译

虽然现在的 Node.js 版本已经支持了很多 ES6 的特性，但这些特性现在还只是实现了，V8 里还没有对这些特性进行优化。如：`*/yield` 等功能。

所以建议使用 Babel 来编译，一方面可以使用 ES6 和 ES7 几乎所有的特性，另一方面编译后的性能也比默认支持的要高。

使用 async/await 替代 */yield

`*/yield` 是 ES6 里提出一种解决异步执行的方法，它只是一个过渡的方案，ES7 里便提出了 `async/await` 来代替它。

相对 `async/await`，`*/yield` 有以下的缺陷：

- 1、`*/yield` 调用后返回一个迭代器，需要借助第三方模块来执行。如：`co`
- 2、`*/yield` 无法和 Arrow Function 一起使用。
- 3、`*/yield` 调用另一个 `*/yield` 时，需要使用 `yield *`，带来不便。
- 4、目前 V8 对 `*/yield` 还没有做优化，最好也通过 Babel 来编译。

所以完全可以使用 ES7 里的 `async/await` 来代替 `*/yield`，然后使用 Babel 编译来运行。

升级指南

本文档为 2.1 到 2.2 的升级指南，2.0 到 2.1 升级指南请见[这里](#)。

2.2 版本兼容 2.1 版本，只是添加了很多功能和微调了一些东西，具体的修改列表请见 [ChangeLog](#)。

升级依赖的 ThinkJS 版本

将 `package.json` 里依赖的 ThinkJS 版本修改为 `2.2.x`，然后重新安装。

添加新的依赖

在项目目录下执行 `npm install source-map-support --save` 安装依赖。

如果是 TypeScript 项目，还需要安装 `npm install source-map --save`。

修改编译脚本

如果是 ES2015+ 项目（不包括 TypeScript 项目），修改 `package.json` 里的 `compile` 命令，将值改为：

```
babel --presets es2015-loose,stage-1 --plugins transform-runtime src/ --out-dir ap
p/ --source-maps
```

删除 app/ 目录

删除项目下的 `app/` 目录，然后执行 `npm start` 重新启动项目。

断点调试

使用 ES2015+ 特性来开发 Node.js 项目可以带来巨大的便利，但同时由于有些特性现在还不支持，需要借助 Babel 编译，运行的代码实际上是编译后的代码，这样给调试带来很大的麻烦。

ThinkJS 从 2.2.0 版本开始支持断点调试源代码，同时如果运行时出现报错，错误也是定位到源代码下。

使用 node-inspector 断点调试

安装 node-inspector

可以通过 `npm install -g node-inspector` 来全局安装 node-inspector，如果是在 *unix 系统下，需要在命令前面添加 sudo 执行。

启动 node-inspector 服务

通过命令 `node-inspector &` 来启动 node-inspector 服务。

启动 Node.js 服务

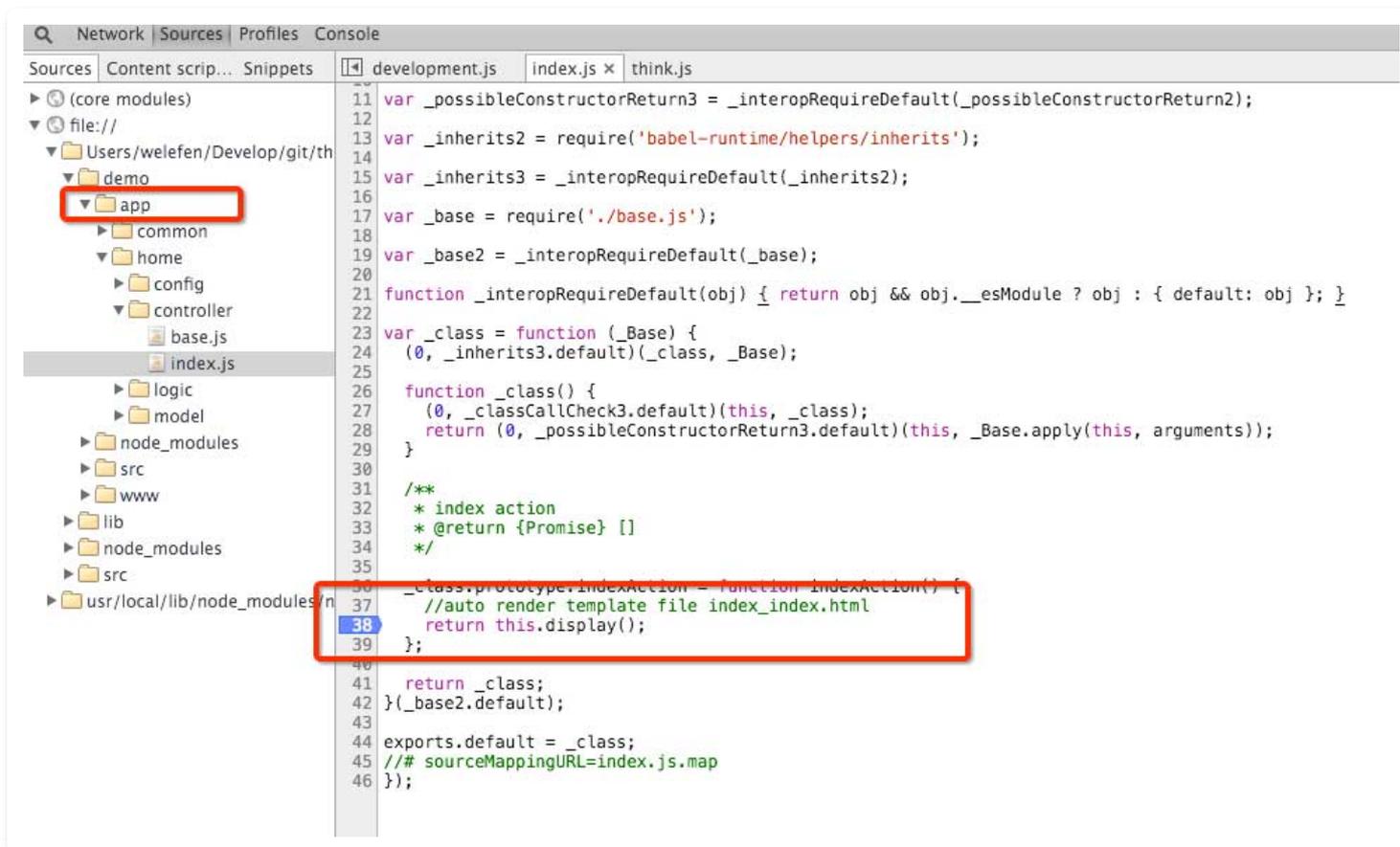
使用 `node --debug www/production.js` 来启动 Node.js 服务。

这里跟之前启动服务有些区别，由于启动时需要添加 `--debug` 参数，所以不能用 `npm start` 来执行启动了。

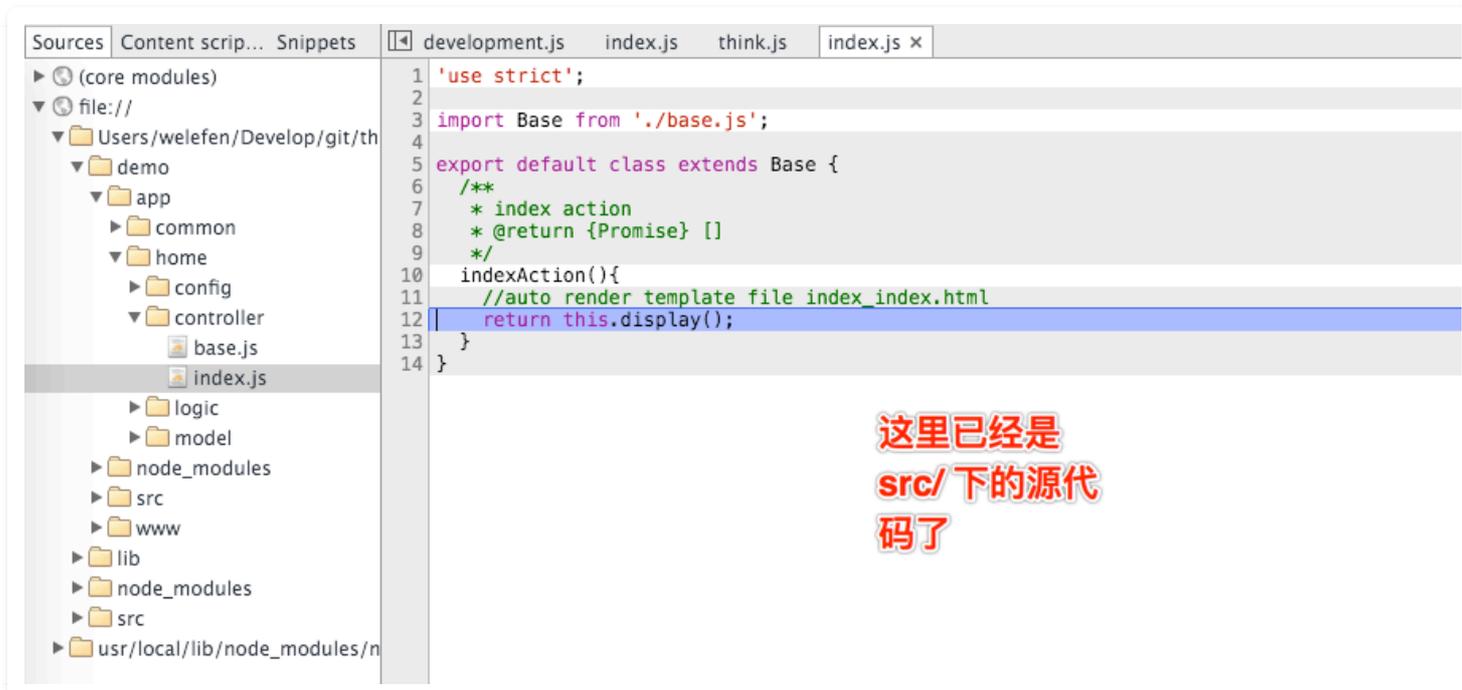
调试

访问 `http://127.0.0.1:8080/debug?port=5858`，会出现调试页面。

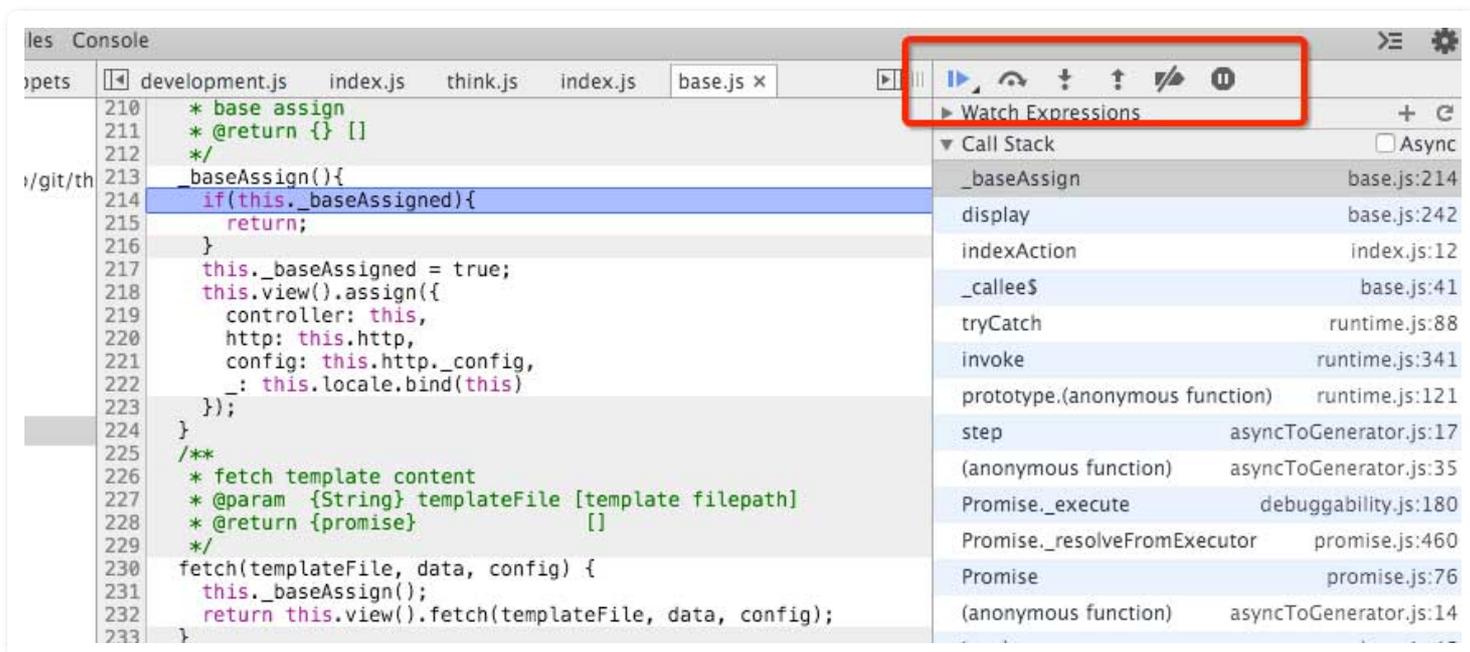
然后在 app 目录下找到对应的编译后的文件，在对应的地方加上断点（这里一定要是在 app/ 目录，不能是源代码 src/ 目录），如：



然后新建标签页，访问对应的接口。这时候页面会一直卡在那里。这时候返回 node-inspector 的标签页，会看到内容已经跳到 ES2015+ 的代码，如：



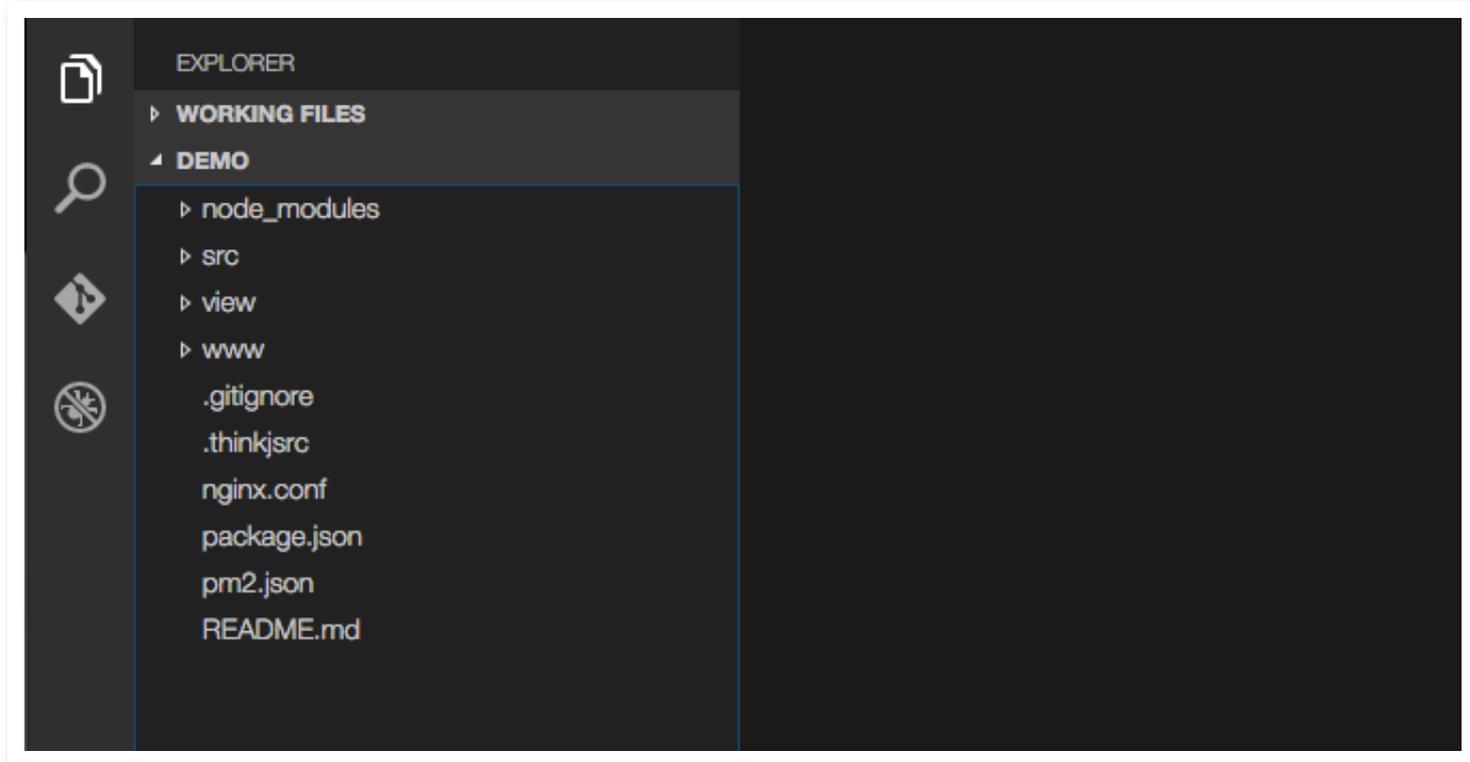
然后就可以利用后侧的断点工具进行调试了。



在 VS Code (v1.7+) 下断点调试

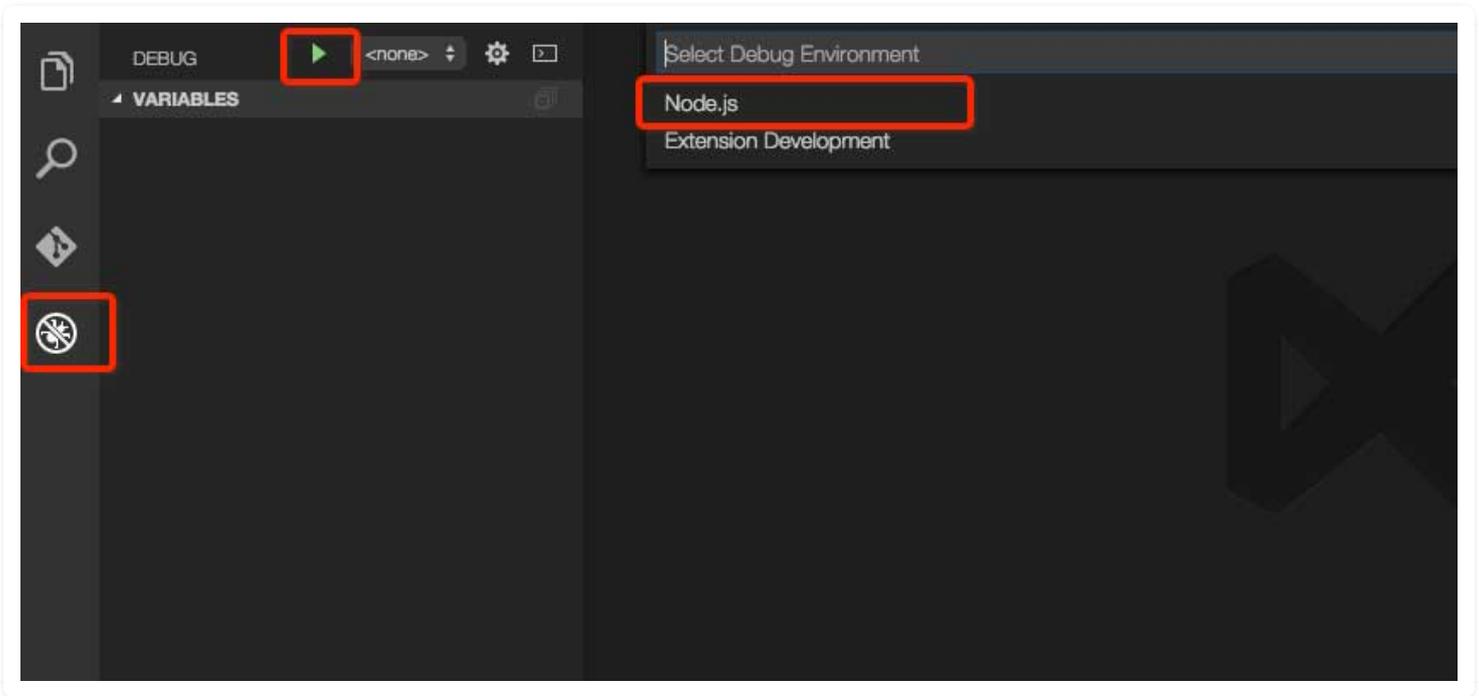
打开项目

通过 VS Code 菜单 File -> Open 来打开 ThinkJS 2015+ 项目，如：



设置调试配置

点击左侧的调试菜单，点击上面的调试按钮，会调试选择的环境，选择 Node.js。如：



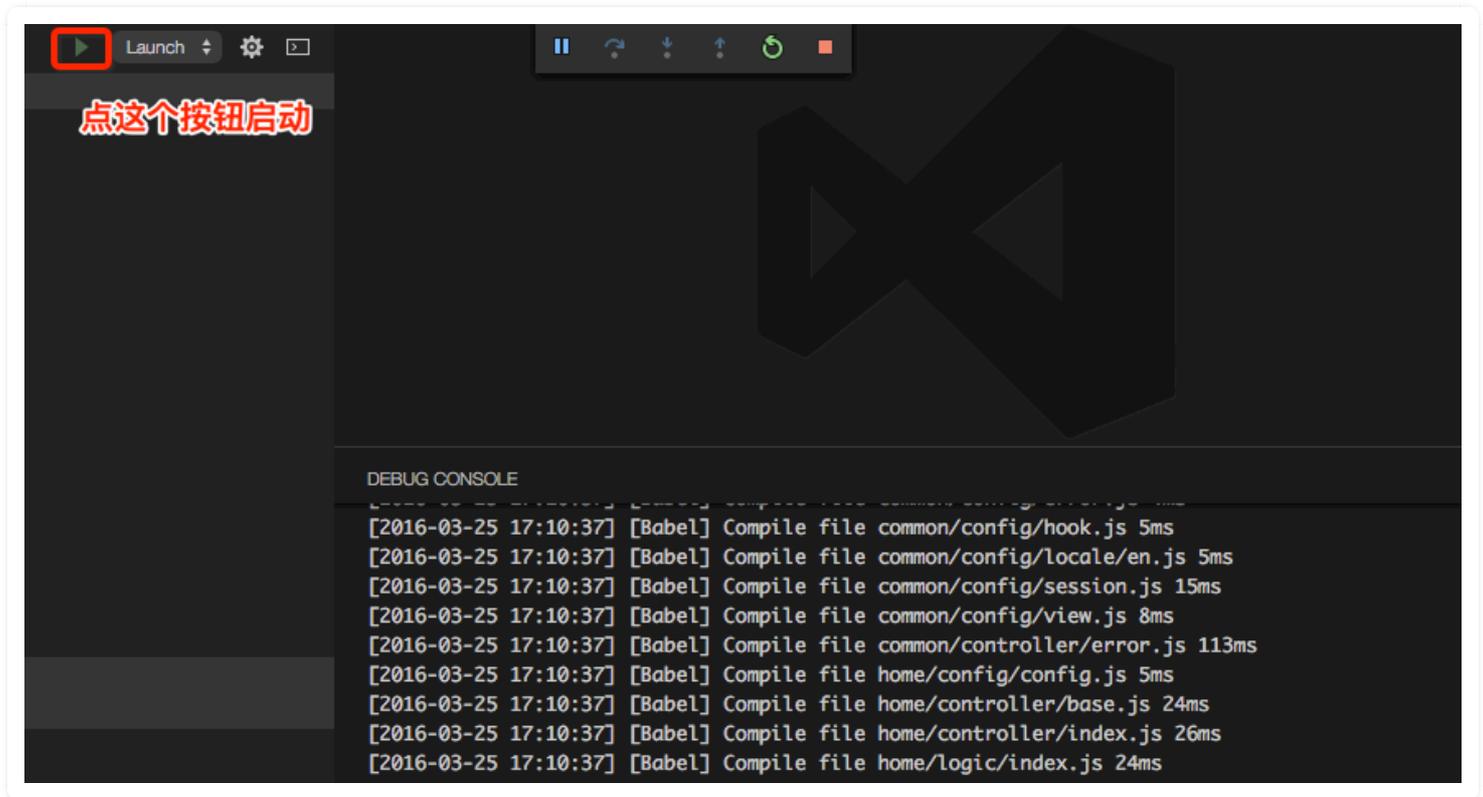
选择 Node.js 后，会生成一个 `launch.json` 文件。修改里面的配置，将 `sourceMaps` 值改为 `true`（注意：有 2 个 `sourceMaps` key，都修改）。
编辑配置为

```
{
  // Use IntelliSense to learn about possible Node.js debug attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "type": "node",
      "request": "launch",
      "name": "启动程序",
      "program": "${workspaceRoot}\\www\\development.js",
      "cwd": "${workspaceRoot}",
      "sourceMaps": true,
      "outFiles": [
        "${workspaceRoot}/app/**"
      ]
    },
    {
      "type": "node",
      "request": "attach",
      "name": "附加到进程",
      "port": 5858
    }
  ]
}
```

即：修改 `program` 配置，添加 `sourceMaps` 和 `outFiles` 配置。

启动服务

点击上面的调试按钮来启动服务。如果已经在命令行启动了 Node.js 服务，需要关掉，否则会因为端口被占用导致错误。

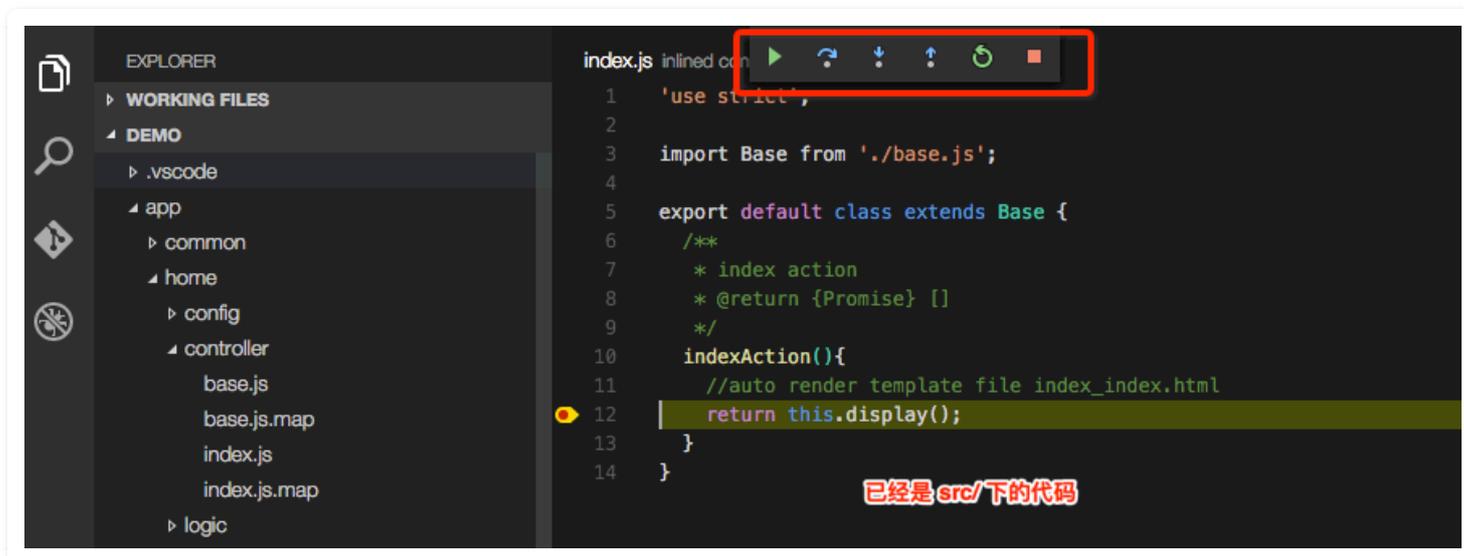


开始调试

回到代码模式，在 `app/` 目录下的文件里加上断点（一定要是在 `app/` 目录下的文件，不能是 `src/` 下的文件）。

在源码中直接添加断点即可调试。

访问对应的页面，就可以看到代码显示的已经是源代码了，然后利用顶部的调试按钮就可以调试了。如：

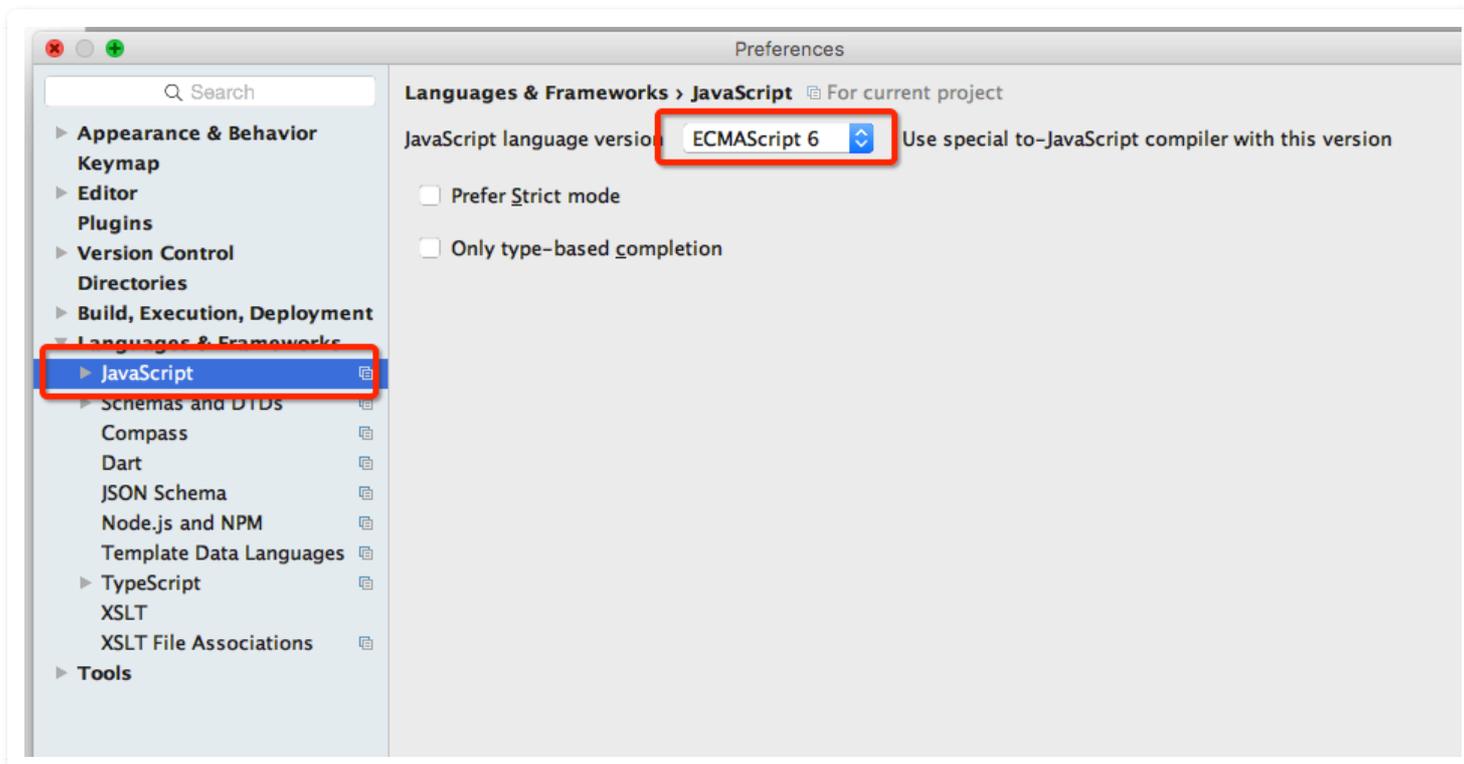


这样就可以很好的在 VS Code 下调试 ES2015+ 代码了。

在 WebStorm 下断点调试

配置 WebStorm

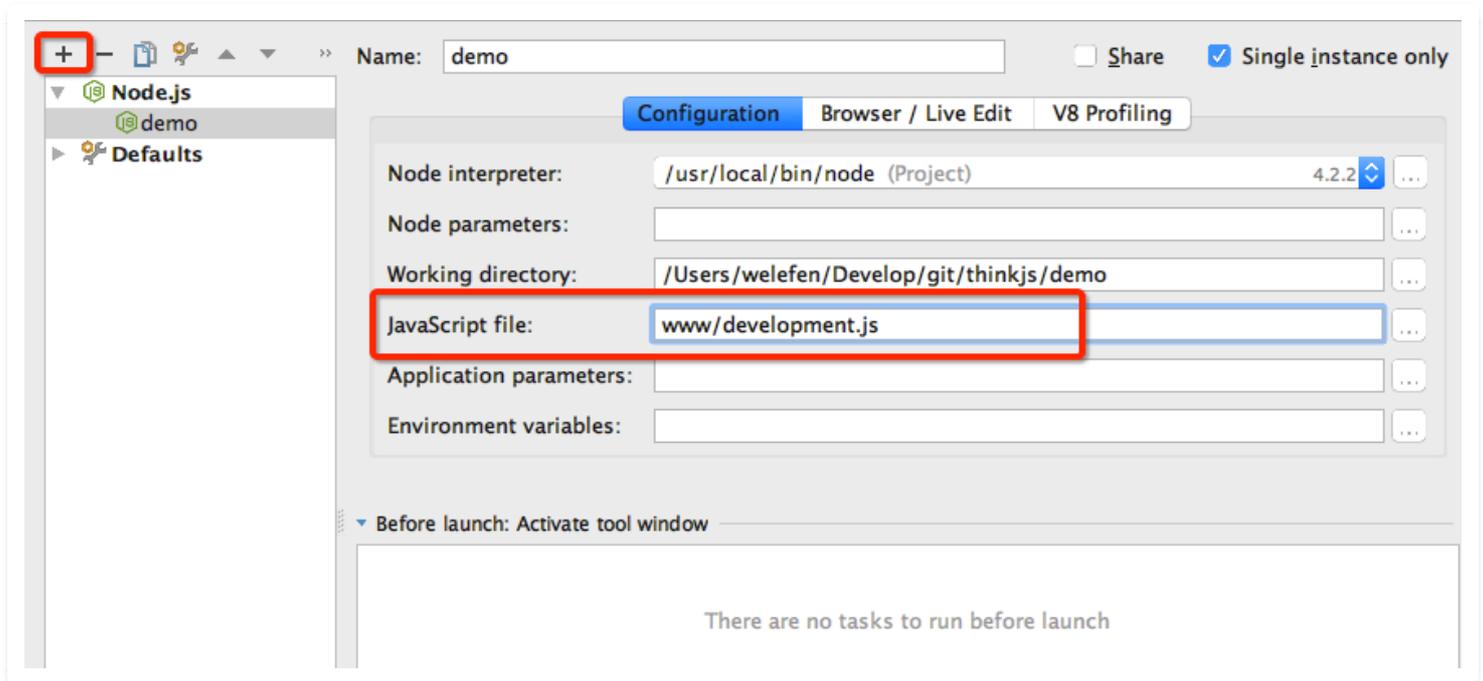
将新建的 ThinkJS 2015+ 项目导入到 WebStorm 中，然后在首选项的 JavaScript 版本设置为 ECMAScript 6。如：



点击右上角的 `Edit Configurations`，然后新建个项目，项目类型选择 Node.js。如：

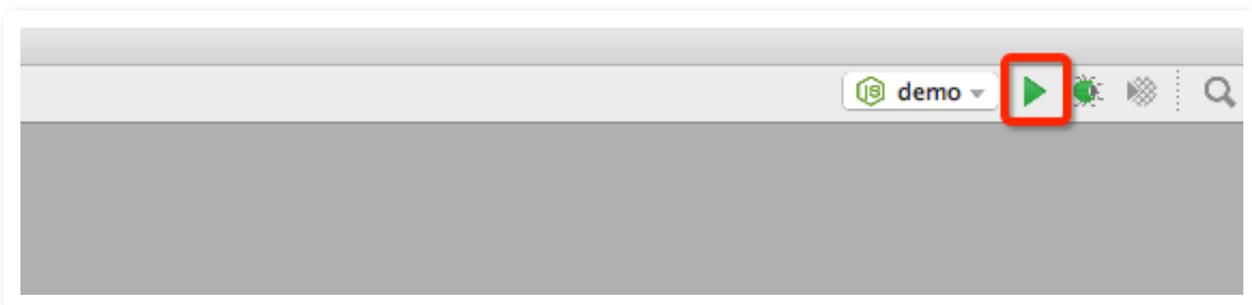


在右侧配置项 `JavaScript File` 里填入 `www/development.js`，或者通过右侧的按钮选择也可以。如：



调试

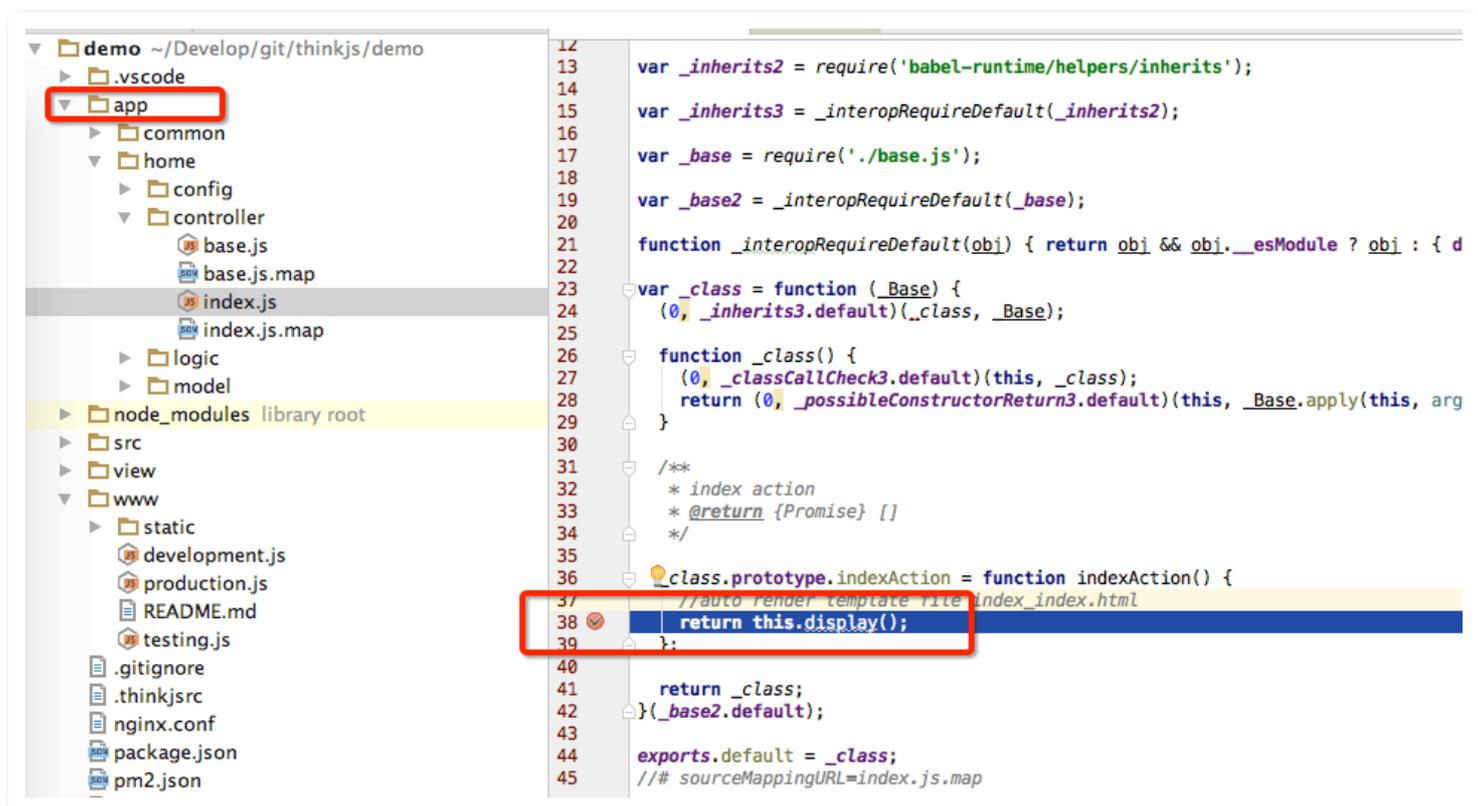
点击右上角的调试按钮，会启动 Node.js 服务。如：



```
Run demo
[2016-03-25 17:30:14] [Babel] Compile file common/config/error.js 7ms
[2016-03-25 17:30:14] [Babel] Compile file common/config/hook.js 3ms
[2016-03-25 17:30:14] [Babel] Compile file common/config/locale/en.js 5ms
[2016-03-25 17:30:14] [Babel] Compile file common/config/session.js 14ms
[2016-03-25 17:30:14] [Babel] Compile file common/config/view.js 7ms
[2016-03-25 17:30:14] [Babel] Compile file common/controller/error.js 115ms
[2016-03-25 17:30:14] [Babel] Compile file home/config/config.js 6ms
[2016-03-25 17:30:14] [Babel] Compile file home/controller/base.js 14ms
[2016-03-25 17:30:14] [Babel] Compile file home/controller/index.js 26ms
[2016-03-25 17:30:14] [Babel] Compile file home/logic/index.js 15ms
[2016-03-25 17:30:14] [Babel] Compile file home/model/index.js 14ms
[2016-03-25 17:30:14] [THINK] Server running at http://127.0.0.1:8360/
[2016-03-25 17:30:14] [THINK] ThinkJS Version: 2.1.8
[2016-03-25 17:30:14] [THINK] Cluster Status: closed
[2016-03-25 17:30:14] [THINK] WebSocket Status: closed
[2016-03-25 17:30:14] [THINK] File Auto Compile: true
[2016-03-25 17:30:14] [THINK] File Auto Reload: true
[2016-03-25 17:30:14] [THINK] App Environment: development
```

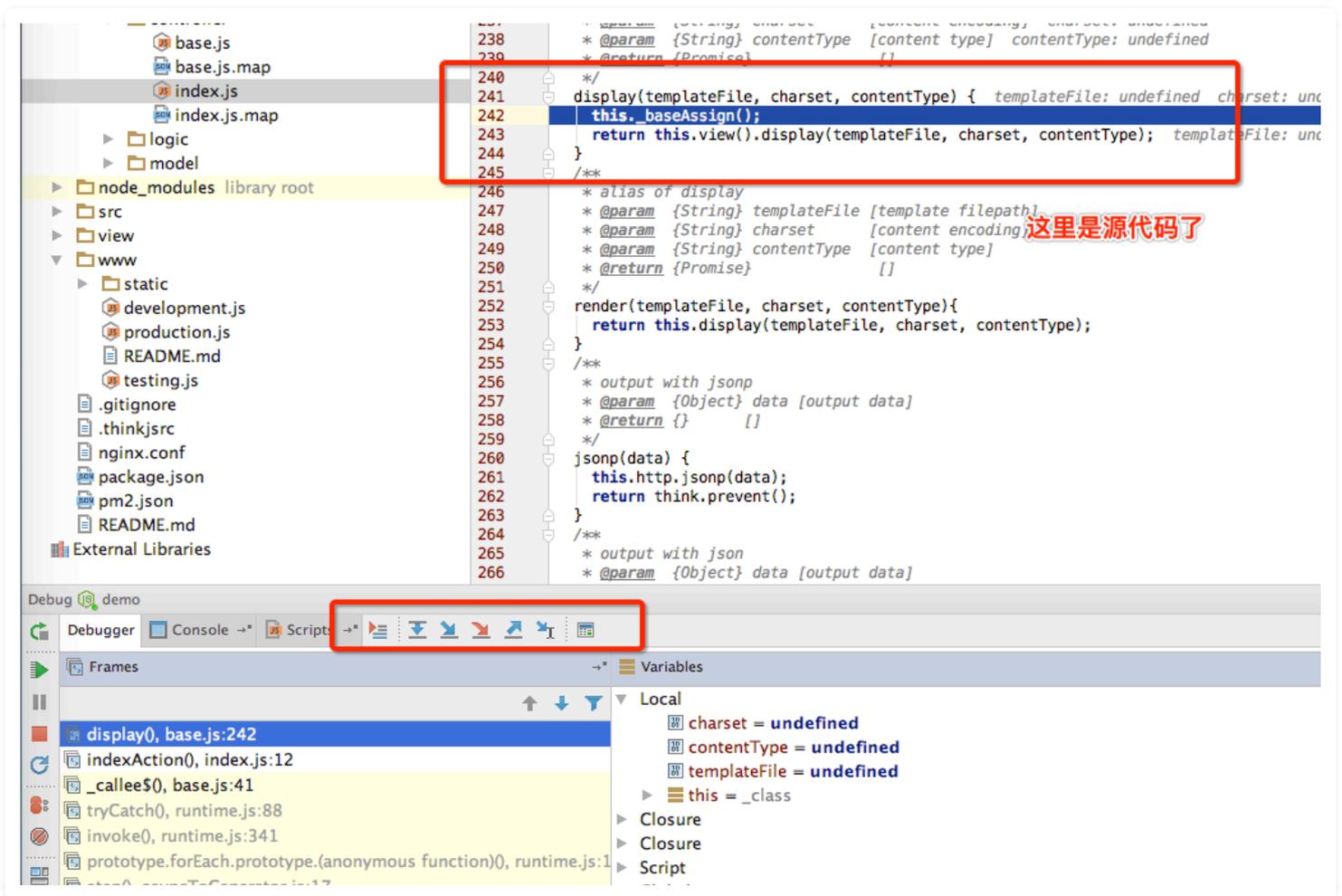
如果之前已经在命令行下启动了服务，需要关掉，否则会出现端口被占用导致报错的情况。

在 app/ 目录下的文件设置断点（一定要在 app/ 目录下，不能是 src/ 目录下），如：



```
12
13 var _inherits2 = require('babel-runtime/helpers/inherits');
14
15 var _inherits3 = _interopRequireDefault(_inherits2);
16
17 var _base = require('./base.js');
18
19 var _base2 = _interopRequireDefault(_base);
20
21 function _interopRequireDefault(obj) { return obj && obj.__esModule ? obj : { d
22
23 var _class = function (_Base) {
24   (0, _inherits3.default)(_class, _Base);
25
26   function _class() {
27     (0, _classCallCheck3.default)(this, _class);
28     return (0, _possibleConstructorReturn3.default)(this, _Base.apply(this, arg
29   }
30
31   /**
32    * index action
33    * @return {Promise} []
34    */
35
36   _class.prototype.indexAction = function indexAction() {
37     //auto render template file index_index.html
38     return this.display();
39   };
40
41   return _class;
42 }(_base2.default);
43
44 exports.default = _class;
45 //# sourceMappingURL=index.js.map
```

打开浏览器，访问对应的接口。返回 WebStorm，点击调试按钮就可以进行调试了，并且看到的是源代码。



常见问题

为什么推荐 ES6/7 语法开发项目

ES6/7 里提供了大量的新特性，这些特性会带来巨大的开发便利和效率上的提升。如：ES6 里的 `*/yield` 和 ES7 里的 `async/await` 特性解决异步回调的问题；箭头函数解决 `this` 作用域的问题；`class` 语法糖解决类继承的问题。

虽然现在 Node.js 环境还没有完全支持这些新的特性，但借助 Babel 编译，可以稳定运行在现在的 Node.js 环境中。所以我们尽可以享受这些新特性带来的便利。

开发时，修改文件需要重启服务么？

默认情况下，由于 Node.js 的机制，文件修改必须重启才能生效。

这种方式下给开发带来了很大的不便，ThinkJS 提供了一种文件自动更新的机制，文件修改后可以立即生效，无需重启服务。

自动更新的机制会消耗一定的性能，所以默认只在 `development` 项目环境下开启。线上代码更新

还是建议使用 `pm2` 模块来管理。

如何修改服务监听的端口

默认情况下，Node.js 服务监听的端口为 `8360`，如果需要修改的话，可以通过修改配置文件 `src/common/config/config.js` 来修改，如：

```
export default {  
  port: 1234 //将监听的端口修改为 1234  
}
```

怎么修改视图文件目录结构

默认情况下，视图文件路径为 `view/[module]/[controller]_[action].html`。其中控制器和操作之间是用 `_` 来连接的，如果想将连接符修改为 `/`，可以修改配置文件

`src/common/config/view.js`：

```
export default {  
  file_depr: '/', //将控制器和操作之间的连接符修改为 /  
}
```

如何开启 cluster

线上可以开启 cluster 功能达到利用多核 CPU 来提升性能，提高并发处理能力。

可以在配置文件 `src/common/config/env/production.js` 中加入如下的配置：

```
export default {  
  cluster_on: true //开启 cluster  
}
```

注：如果使用 PM2 管理服务且开启了 cluster，那么 ThinkJS 里就无需再开启 cluster 了。

修改请求超时时间

默认请求的超时时间是 120s，可以通过修改配置文件 `src/common/config/config.js` 里 `timeout` 配置值。

```
export default {
  timeout: 30, //将超时时间修改为 30s
}
```

如何捕获异常

JS 本身是无法通过 try/catch 来捕获异步异常的，但使用 async/await 后则可以通过 try/catch 来捕获异常，如：

```
export default class extends think.controller.base {
  async indexAction(){
    try{
      await this.getFromAPI1();
      await this.getFromAPI2();
      await this.getFromAPI3();
    }catch(err){
      //通过 err.message 拿到具体的错误信息
      return this.fail(err.message);
    }
  }
}
```

上面的方式虽然可以通过 try/catch 来捕获异常，但在 catch 里并不知道异常是哪个触发的。

实际项目中，经常要根据不同的错误返回不同的错误信息给用户，这时用整体的 try/catch 就不太方便了。

此时可以通过单个异步接口返回特定值来判断，如：

```
export default class extends think.controller.base {
  async indexAction(){
    //忽略该接口的错误（该接口的错误不重要，可以忽略）
    await this.getFromAPI1().catch(() => {});
    //异常时返回特定值 false 来判断
    let result = await this.getFromAPI2().catch(() => false);
    if(result === false){
      return this.fail('API2 ERROR');
    }
  }
}
```

如上面代码所述，通过返回特定值判断就可以方便的知道是哪个异步接口发生了错误，这样就可以针对不同的错误返回不同的错误信息。

如何忽略异常

使用 `async/await` 时，如果 `Promise` 返回了一个 `rejected Promise`，那么会抛出异常。如果这个异常不重要需要忽略的话，可以通过 `catch` 方法返回一个 `resolve Promise` 来完成。如：

```
export default class extends think.controller.base {
  async indexAction(){
    //通过在 catch 里返回 undefined 来忽略异常
    await this.getAPI().catch(() => {});
  }
}
```

PREVENT_NEXT_PROCESS

在调用有些方法后（如：`success`）后会发现有个 `message` 为 `PREVENT_NEXT_PROCESS` 的错误。这个错误是 `ThinkJS` 为了阻止后续执行添加的，如果要在 `catch` 里判断是否是该错误，可以通过 `think.isPrevent` 方法来判断。如：

```
module.exports = think.controller({
  indexAction(self){
    return self.getData().then(function(data){
      return self.success(data);
    }).catch(function(err){
      //忽略 PREVENT_NEXT_PROCESS 错误
      if(think.isPrevent(err)){
        return;
      }
      console.log(err.stack);
    })
  }
})
```

另一种处理方式：对于 `success` 之类的方法前面不要添加 `return`，这样 `catch` 里就不会有此类的错误了。

并行处理

使用 `async/await` 来处理异步时，是串行执行的。但很多场景下我们需要并行处理，这样可以大大提高执行效率，此时可以结合 `Promise.all` 来处理。

```
export default class extends think.controller.base {
  async indexAction(){
```

```
    let p1 = this.getServiceData1();
    let p2 = this.getAPIData2();
    let [p1Data, p2Data] = await Promise.all([p1, p2]);
  }
}
```

上面的代码 `p1` 和 `p2` 是并行处理的，然后用 `Promise.all` 来获取 2 个数据。这样一方面代码是同步书写的，同时又不失并行处理的性能。

如何输出图片

项目中有时候要输出图片等类型的数据，可以通过下面的方式进行：

```
export default class extends think.controller.base {
  imageAction(){
    //图片 buffer 数据，读取本地文件或者从远程获取
    let imageBuffer = new Buffer();
    this.type('image/png');
    this.end(imageBuffer);
  }
}
```

如何在不同的环境下使用不同的配置

我们经常在不同的环境下使用不同的配置，如：开发环境和线上环境使用不同的数据库配置。这时可以通过 `src/common/config/env/[env].js` 来配置，`[env]` 默认有 `development`，`testing` 和 `production` 3 个值，分别对应开发环境、测试环境和线上环境。这时可以在对应的配置文件设定配置来用在不同的环境下。

如：配置线上环境下的数据库，那么可以在 `src/common/config/env/production.js` 中配置：

```
export default {
  db: { //这里要有一级 db
    type: 'mysql',
    adapter: {
      mysql: {
        host: '',
        port: ''
      }
    }
  }
}
```

详细的数据库配置请见[这里](#)。

nunjucks 模板继承路径怎么写

使用 nunjucks 的模板继承时，由于设置了 `root_path`，所以路径需要使用相对路径。如：

```
{% extends "../parent.html" %} //表示同级别目录下的 parent.html 文件
{% extends "../../layout.html" %} //表示父级别下的 layout.html 文件
```

如何让 Action 只允许命令行调用

默认情况下，Action 既可以用户访问，也可以命令行调用。但有些 Action 我们希望只在命令行下调用，这时可以通过 `isCli` 来判断。如：

```
export default class extends think.controller.base {
  indexAction(){
    //禁止 URL 访问该 Action
    if(!this.isCli()){
      this.fail('only allow invoked in cli mode');
    }
    ...
  }
}
```

如何跨模块调用

当项目比较复杂时，会有一些跨模块调用的需求。

调用 controller

可以通过 `this.controller` 方法传递第二个参数模块名达到调用其他模块下 controller 的功能，如：

```
export default class extends think.controller.base {
  indexAction(){
    //获取 admin 模块下 user controller 的实例
    let controllerInstance = this.controller('user', 'admin');
    //获取 controller 的实例下就可以调用下面的方法了
    let bar = controllerInstance.foo();
  }
  index2Action(){
```

```
//也可以通过这种更简洁的方式获取
let controllerInstance = this.controller('admin/user');
let bar = controllerInstance.foo();
}
}
```

调用 action

可以通过 `this.action` 方法调用其他模块里 controller 下的 action 方法，如：

```
export default class extends think.controller.base {
  async indexAction(){
    //获取 admin 模块下 user controller 的实例
    let controllerInstance = this.controller('user', 'admin');
    //调用 controller 里的 test action, 会自动调用 __before 和 __after 魔术方法
    let data = await this.action(controllerInstance, 'test')
  }
  async index2Action(){
    //也可以通过字符串来指定 controller, 这样会自动找对应的 controller
    let data = await this.action('admin/user', 'test')
  }
}
```

注： action 调用返回的始终为 Promise，调用 action 时不会调用对应的 logic。

调用 model

可以通过 `this.model` 方法获取其他模块下的 model 实例，如：

```
export default class extends think.controller.base {
  indexAction(){
    //获取 admin 模块下的 user model 实例
    let modelInstance1 = this.model('user', {}, 'admin');
    //也可以通过这种更简洁的方式
    let modelInstance2 = this.model('admin/user');
  }
}
```

如何请求其他接口数据

在项目中，经常要请求其他接口的数据。这时候可以用内置的 `http` 模块来操作，但 `http` 模块提供的接口比较基础，写起来比较麻烦。推荐大家用基于 `http` 模块封装的 `request` 模块或者

`superagent` 模块。如：

```
import request from 'request';
/* 获取 API 接口数据 */
let getApiData = () => {
  let deferred = think.defer();
  request.get({
    url: 'http://www.example.com/api/user',
    headers: {
      'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) Chrome/47.0.2
526.111 Safari/537.36'
    }
  }, (err, response, body) => {
    if(err){
      deferred.reject(err);
    }else{
      deferred.resolve(body);
    }
  });
}
```

但这么写需要创建一个 `deferred` 对象，然后在回调函数里去根据 `err` 进行 `resolve` 或者 `reject`，写起来有些麻烦。ThinkJS 里提供了 `think.promisify` 方法来快速处理这一问题。

```
import request from 'request';
/* 获取 API 接口数据 */
let getApiData = () => {
  let fn = think.promisify(request.get);
  return fn({
    url: 'http://www.example.com/api/user',
    headers: {
      'User-Agent': 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10_11_2) Chrome/47.0.2
526.111 Safari/537.36'
    }
  });
}
```

开发环境好的，线上部署 502

有时候开发环境下是好的，到线上使用 `pm2` 和 `nginx` 部署时，访问出现 502 的情况，这个情况一般为 `node` 服务没有正常启动导致的。可以通过 `pm2 logs` 看对应的错误信息来分析排查，也可以先关闭服务，手动通过 `node www/production.js` 启动服务，然后访问看具体的错误信息。

设置跨域头信息

高级浏览器支持通过设置头信息达到跨域请求，ThinkJS 里可以通过下面的方式来设置：

```
export default class extends think.controller.base {
  indexAction(){
    let method = this.http.method.toLowerCase();
    if(method === 'options'){
      this.setCorsHeader();
      this.end();
      return;
    }
    this.setCorsHeader();
    this.success();
  }
  setCorsHeader(){
    this.header('Access-Control-Allow-Origin', this.header('origin') || '*');
    this.header('Access-Control-Allow-Headers', 'x-requested-with');
    this.header('Access-Control-Request-Method', 'GET,POST,PUT,DELETE');
    this.header('Access-Control-Allow-Credentials', 'true');
  }
}
```

更多头信息设置请见 <https://www.w3.org/TR/cors>。

如果是在 REST API，那么可以放在 `__call` 方法里判断，如：

```
export default class extends think.controller.base {
  __call(){
    let method = this.http.method.toLowerCase();
    if(method === 'options'){
      this.setCorsHeader();
      this.end();
      return;
    }
    this.setCorsHeader();
    return super.__call();
  }
  setCorsHeader(){
    this.header('Access-Control-Allow-Origin', this.header('origin') || '*');
    this.header('Access-Control-Allow-Headers', 'x-requested-with');
    this.header("Access-Control-Allow-Methods", "GET,POST,OPTIONS,PUT,DELETE");
    this.header('Access-Control-Allow-Credentials', 'true');
  }
}
```

current path is not thinkjs project.

使用 thinkjs 命令创建一些 adapter 或者 model 之类时，有时候会报

`current path is not thinkjs project` 的错误。

这是因为在用 `thinkjs new project` 来创建项目时，会在项目下创建一个名为 `.thinkjsrc` 的文件，这个文件里有对项目的一些描述。后续创建 adapter 等功能时需要读取这个文件，如果这个文件丢失，那么就会报 `current path is not thinkjs project` 错误。

解决方案也很简单，找个目录创建一个模式一样的项目，然后把 `.thinkjsrc` 文件拷贝到当前项目下即可。

注：`.thinkjsrc` 文件需要纳入到项目版本管理中，不然后续会持续出现这个问题。

编译时，如何追加 presets 和 plugins

使用 Babel 编译时，内置使用的 presets 为 `es2015-loose,stage-1`，plugins 为 `transform-runtime`。如果内置的不满足项目的需求，可以修改 `www/development.js` 来追加，如：

```
//compile src/ to app/
instance.compile({
  log: true,
  presets: ['stage-0'], //添加 stage-0 presets
  plugins: ['transform-decorators-legacy']
});
```

添加对应的 presets 和 plugins 后，需要在项目安装对应的依赖才可运行。

如何自定义创建服务

ThinkJS 默认会创建一个 HTTP 服务，如果不能满足项目的需求的话，可以自定义创建服务。

可以通过修改 `src/common/config/config.js` 文件，添加配置 `create_server` 来自定义服务，如：

```
import http from 'http';
export default {
  create_server: (callback, port, host, app) => {
    let server = http.createServer(callback);
    server.listen(port, host);
    return server;
  }
}
```

其中形参 `callback` 为：

```
let callback = (req, res) => {
  think.http(req, res).then(http => {
    new this(http).run();
  });
};
```

用户登录后才能访问

项目开发中，经常会有某些功能需要用户登录后才能访问，如：后台管理。对于此类需要可以放在一个通用的逻辑处理里，建议放在 base controller 里的 `__before` 进行处理。

```
/* base controller */
export default class extends think.controller.base {
  async __before(){
    //部分 action 下不检查
    let blankActions = ['login'];
    if(blankActions.indexOf(this.http.action) >= 0){
      return;
    }
    let userInfo = await this.session('userInfo');
    //判断 session 里的 userInfo
    if(think.isEmpty(userInfo)){
      return this.redirect('/user/login');
    }
  }
}
```

其他 controller 文件继承 base controller，如：

```
import Base from './base';

export default class extends Base {

}
```

这样继承了 Base 的 controller 都会校验用户信息了。

TypeError: Unkown charset 'utf8mb4'

升级到 2.2.2 版本后，如果数据库的编码之前填的是 utf8mb4，那么会报一个

`TypeError: Unkown charset 'utf8mb4'` 的错误。这是因为 2.2.2 版本将依赖的 mysql 库改为了 `node-mysql2`（该模块性能是 `node-mysql` 的 2 倍），这个模块不支持 utf8mb4 简写，所以需要将

配置文件里的数据库编码改为 `UTF8MB4_GENERAL_CI`，具体支持的编码列表见：<https://github.com/sidorares/node-mysql2/blob/master/lib/constants/charsets.js>

进阶应用

模块

ThinkJS 创建项目时支持多种项目模式，默认创建的项目是按模块来划分的，并且自动添加了 `common` 和 `home` 2 个模块。每个模块有独立的配置、控制器、视图、模型等文件。

使用模块的方式划分项目，可以让项目结构更加清晰。如：一般一个博客系统可分为前后台 2 个模块。

模块列表

进去 `src/` 目录就可以看到模块列表：

```
drwxr-xr-x  5 welefen  staff  170 Aug 18 15:55 common/  
drwxr-xr-x  6 welefen  staff  204 Sep  8 19:14 home/
```

common 模块

`common` 模块是个通用模块，该模块下存放一些通用的功能，如：通用的配置，`runtime` 目录，启动文件，错误处理控制器等。

注：该模块下的控制器不能响应用户的请求。

默认模块

默认模块为 `home` 模块。当解析用户的请求找不到模块时会自动对应到 `home` 下。

可以通过配置 `default_module` 来修改默认模块，修改配置文件 `src/common/config/config.js`：

```
//将默认模块名改为 blog  
export default {  
  default_module: 'blog'  
}
```

```
}
```

添加模块

添加模块直接通过 `thinkjs` 命令即可完成。

在当前项目目录下，执行 `thinkjs module xxx`，即可创建名为 `xxx` 的模块。

如果模块名已经存在，则无法创建。

禁用模块

ThinkJS 默认会自动查找和识别项目下的模块，并认为所有的模块都是可用的。

如果想禁用部分模块，可以修改配置文件 `src/common/config/config.js`，添加下面的配置：

```
export default {
  deny_module_list: ['xxx'] //禁用 xxx 模块
}
```

控制器

控制器是一类操作的集合，用来响应用户同一类的请求。

定义控制器

创建文件 `src/home/controller/article.js`，表示 `home` 模块下有名为 `article` 控制器，文件内容类似如下：

```
'use strict';

import Base from './base.js';

export default class extends Base {
  /**
   * index action
   * @return {Promise} []
   */
  indexAction(){
    //auto render template file index_index.html
    return this.display();
  }
}
```

```
}  
}
```

如果不想使用 ES6 语法，那么文件内容类似如下：

```
'use strict';  
  
var Base = require('./base.js');  
  
module.exports = think.controller(Base, {  
  /**  
   * index action  
   * @return {Promise} []  
   */  
  indexAction: function(self){  
    //auto render template file index_index.html  
    return self.display();  
  }  
});
```

注：上面的 `Base` 表示定义一个基类，其他的类都继承该基类，这样就可以在基类里做一些通用的处理。

多级控制器

对于很复杂的项目，一层控制器有时候不能满足需求。这个时候可以创建多级控制器，如：

`src/home/controller/group/article.js`，这时解析到的控制器为二级，具体为 `group/article`，`Logic` 和 `View` 的目录与此相同。

使用 async/await

借助 Babel 编译，还可以在控制器里使用 ES7 里的 `async/await`。

ES7 方式

```
'use strict';  
  
import Base from './base.js';  
  
export default class extends Base {  
  /**  
   * index action  
   * @return {Promise} []  
   */  
}
```

```
    async indexAction(){
      let model = this.model('user');
      let data = await model.select();
      return this.success(data);
    }
  }
}
```

动态创建类的方式

```
'use strict';

var Base = require('./base.js');

module.exports = think.controller(Base, {
  /**
   * index action
   * @return {Promise} []
   */
  indexAction: async function(){
    var model = this.model('user');
    var data = await model.select();
    return this.success(data);
  }
});
```

init 方法

ES6 里的 class 有 constructor 方法，但动态创建的类就没有该方法了，为了统一初始化执行的方法，将该方法统一定义为 `init`。

该方法在类实例化的时候自动调用，无需手工调用。

ES6 方式

```
```js
'use strict';

import Base from './base.js';

export default class extends Base {
 init(http){
 super.init(http); //调用父类的init方法
 }
}
...
```
```

```
}  
}  
``
```

动态创建类方式

```
'use strict';  
  
var Base = require('./base.js');  
  
module.exports = think.controller(Base, {  
  init: function(http){  
    this.super('init', http); //调用父类的init方法  
    ...  
  }  
});
```

`init` 方法里需要调用父类的 `init` 方法，并将参数 `http` 传递进去。

前置操作 `__before`

ThinkJS 支持前置操作，方法名为 `__before`，该方法会在具体的 Action 调用之前自动调用。如果前置操作里阻止了后续代码继续执行，则不会调用具体的 Action，这样可以提前结束请求。

ES6 方式

```
``js  
'use strict';  
  
import Base from './base.js';  
  
export default class extends Base {  
  /**  
   * 前置方法  
   * @return {Promise} []  
   */  
  __before(){  
    ...  
  }  
}  
}  
``
```

Action

一个 Action 代表一个要执行的操作。如：url 为 `/home/article/detail`，解析后的模块为 `/home`，控制器为 `article`，Action 为 `detail`，那么执行的 Action 就是文件 `src/home/controller/article` 里的 `detailAction` 方法。

```
'use strict';

import Base from './base.js';

export default class extends Base {
  /**
   * 获取详细信息
   * @return {Promise} []
   */
  detailAction(self){
    ...
  }
}
```

如果解析后的 Action 值里含有 `_`，会自动做转化，具体的转化策略见 [路由 -> 大小写转化](#)。

后置操作 `__after`

ThinkJS 支持后置操作，方法名为 `__after`，该方法会在具体的 Action 调用之后执行。如果具体的 Action 里阻止了后续的代码继续执行，则后置操作不会调用。

空操作 `__call`

当解析后的 url 对应的控制器存在，但 Action 不存在时，会试图调用控制器下的魔术方法 `__call`。这里可以对不存在的方法进行统一处理。

```
'use strict';

import Base from './base.js';

export default class extends Base {
  /**
   * @return {Promise} []
   */
  __call(){
    ...
  }
}
```

错误处理

当 url 不存在或者当前用户没权限等一些异常请求时，这时候会调用错误处理。ThinkJS 内置了一套详细的错误处理机制，具体请见 [扩展功能 -> 错误处理](#)。

数据校验

控制器里在使用用户提交的数据之前，需要对数据合法性进行校验。为了降低控制器里的逻辑复杂度，ThinkJS 提供了一层 Logic 专门用来处理数据校验和权限校验等相关操作。

详细信息请见 [扩展功能 -> Logic -> 数据校验](#)。

变量赋值和模版渲染

控制器里可以通过 `assign` 和 `display` 方法进行变量赋值和模版渲染，具体信息请见 [这里](#)。

模型实例化

在控制器中可以通过 `this.model` 方法快速获得一个模型的实例。

```
export default class extends think.controller.base {
  indexAction(){
    let model = this.model('user'); //实例化模型 user
    ...
  }
}
```

model 方法更多使用方式请见 [API -> think.model.base](#)。

http 对象

控制器在实例化时，会将 `http` 传递进去。该 `http` 对象是 ThinkJS 对 `req` 和 `res` 重新包装的一个对象，而非 Node.js 内置的 http 对象。

Action 里如果想获取该对象，可以通过 `this.http` 来获取。

```
'use strict';

import Base from './base.js';
```

```
export default class extends Base {
  indexAction(){
    let http = this.http;
  }
}
```

关于 `http` 对象包含的属性和方法请见 [API -> http](#)。

REST API

有时候，项目里需要提供一些 REST 接口给第三方使用，这些接口无外乎就是增删改查等操作。

如果手工去书写这些操作则比较麻烦，ThinkJS 提供了 REST Controller，该控制器会自动含有通用的增删改查等操作。如果这些操作不满足需求，也可以进行定制。具体请见 [这里](#)。

this 作用域的问题

Node.js 里经常有很多异步操作，而异步操作常见的处理方式是使用回调函数或者 Promise。这些处理方式都会增加一层作用域，导致在回调函数内无法直接使用 `this`，简单的处理办法是在顶部定义一个变量，将 `this` 赋值给这个变量，然后在回调函数内使用这个变量。如：

```
module.exports = think.controller({
  indexAction: function(){
    var self = this; //这里将 this 赋值给变量 self, 然后在后面的回调函数里都使用 self
    this.model('user').find().then(function(data){
      return self.model('article').where({user_id: data.id}).select();
    }).then(function(data){
      self.success(data);
    })
  }
})
```

如果每个 Action 里都要使用者手工写一个 `var self = this`，势必比较麻烦。为了解决这个问题，ThinkJS 在 Action 里直接提供了一个参数，这个参数等同于 `var self = this`，具体如下：

```
module.exports = think.controller({
  //参数 self 等同于 var self = this
  indexAction: function(self){
    this.model('user').find().then(function(data){
      return self.model('article').where({user_id: data.id}).select();
    }).then(function(data){
      self.success(data);
    })
  }
})
```

```
}  
})
```

当然更好的解决办法是推荐使用 ES6 里的 Generator Function 和 Arrow Function，这样就可以彻底解决 this 作用域的问题。

使用 Generator Function

```
export default class extends think.controller.base {  
  * indexAction(){  
    let data = yield this.model('user').find();  
    let result = yield this.model('article').where({user_id: data.id}).select();  
    this.success(result);  
  }  
}
```

使用 Arrow Function

```
module.exports = think.controller({  
  indexAction: function(){  
    this.model('user').find().then(data => {  
      return this.model('article').where({user_id: data.id}).select();  
    }).then(data => {  
      this.success(data);  
    })  
  }  
})
```

JSON 输出

项目中经常要提供一些接口，这些接口一般都是直接输出 JSON 格式的数据，并且会有标识表明当前接口是否正常。如果发生异常，需要将对应的错误信息随着接口一起输出。控制器里提供了

`this.success` 和 `this.fail` 方法来输出这样的接口数据。

输出正常的 JSON

可以通过 `this.success` 方法输出正常的接口数据，如：

```
export default class extends think.controller.base {  
  indexAction(){  
    let data = {name: "thinkjs"};  
    this.success(data);  
  }  
}
```

```
}  
}
```

输出结果为 `{errno: 0, errmsg: "", data: {"name": "thinkjs"}}`，客户端可以通过 `errno` 是否为 0 来判断当前接口是否有异常。

输出含有错误信息的 JSON

可以通过 `this.fail` 方法输出含有错误信息的接口数据，如：

```
export default class extends think.controller.base {  
  indexAction(){  
    this.fail(1000, 'connect error'); //指定错误号和错误信息  
  }  
}
```

输出结果为 `{errno: 1000, errmsg: "connect error"}`，客户端判断 `errno` 大于 0，就知道当前接口有异常，并且通过 `errmsg` 拿到具体的错误信息。

配置错误号和错误信息

如果每个地方输出错误的时候都要指定错误号和错误信息势必比较麻烦，比较好的方式是把错误号和错误信息在一个地方配置，然后输出的时候只要指定错误号，错误信息根据错误号自动读取。

错误信息支持国际化，所以配置放在 `src/common/config/locale/[lang].js` 文件中。如：

```
export default {  
  10001: 'get data error'  
}
```

通过上面的配置后，执行 `this.fail(10001)` 时会自动读取到对应的错误信息。

友好的错误号

在程序里执行 `this.fail(10001)` 虽然能输出正确的错误号和错误信息，但人不能直观的看出来错误号对应的错误信息是什么。

这时可以将 key 配置为大写字符串，值为错误号和错误信息。如：

```
export default {  
  GET_DATA_ERROR: [1234, 'get data error'] //key 必须为大写字符或者下划线才有效  
}
```

执行 `this.fail('GET_DATA_ERROR')` 时也会自动取到对应的错误号和错误信息。

格式配置

默认输出的错误号的 key 为 `errno`，错误信息的 key 为 `errmsg`。如果不满足需求的话，可以修改配置文件 `src/common/config/error.js`。

```
export default {
  key: 'errno', //error number
  msg: 'errmsg', //error message
}
```

输出不包含错误信息的 JSON

如果输出的 JSON 数据里不想包含 `errno` 和 `errmsg` 的话，可以通过 `this.json` 方法输出 JSON。如：

```
export default class extends think.controller.base {
  indexAction(){
    this.json({name: 'thinkjs'});
  }
}
```

常用功能

获取 GET 参数

可以通过 `get` 方法获取 GET 参数，如：

```
export default class extends think.controller.base {
  indexAction(){
    let name = this.get('name');
    let allParams = this.get(); //获取所有 GET 参数
  }
}
```

如果参数不存在，那么值为空字符串。

获取 POST 参数

可以通过 `post` 方法获取 POST 参数，如：

```
export default class extends think.controller.base {
  indexAction(){
    let name = this.post('name');
    let allParams = this.post(); //获取所有 POST 参数
  }
}
```

如果参数不存在，那么值为空字符串。

获取上传的文件

可以通过 `file` 方法获取上传的文件，如：

```
export default class extends think.controller.base {
  indexAction(){
    let file = this.file('image');
    let allFiles = this.file(); //获取所有上传的文件
  }
}
```

返回值是个对象，包含下面的属性：

```
{
  fieldName: 'file', //表单字段名称
  originalFilename: filename, //原始的文件名
  path: filepath, //文件保存的临时路径，使用时需要将其移动到项目里的目录，否则请求结束时会被删除
  size: 1000 //文件大小
}
```

如果文件不存在，那么值为一个空对象 `{}`。

JSONP 格式数据输出

可以通过 `this.jsonp` 方法输出 JSONP 格式的数据，`callback` 的请求参数名默认为 `callback`。如果需要修改请求参数名，可以通过修改配置 `callback_name` 来完成。

更多方法

- `isGet()` 当前是否是 GET 请求
- `isPost()` 当前是否是 POST 请求
- `isAjax()` 是否是 AJAX 请求
- `ip()` 获取请求用户的 ip
- `redirect(url)` 跳转到一个 url
- `write(data)` 输出数据, 会自动调用 `JSON.stringify`
- `end(data)` 结束当前的 http 请求
- `json(data)` 输出 JSON 数据, 自动发送 JSON 相关的 Content-Type
- `jsonp(data)` 输出 JSONP 数据, 请求参数名默认为 `callback`
- `success(data)` 输出一个正常的 JSON 数据, 数据格式为 `{errno: 0, errmsg: "", data: data}`
- `fail(errno, errmsg, data)` 输出一个错误的 JSON 数据, 数据格式为 `{errno: errno_value, errmsg: string, data: data}`
- `download(file)` 下载文件
- `assign(name, value)` 设置模版变量
- `display()` 输出一个模版
- `fetch()` 渲染模版并获取内容
- `cookie(name, value)` 获取或者设置 cookie
- `session(name, value)` 获取或者设置 session
- `header(name, value)` 获取或者设置 header
- `action(name, data)` 调用其他 Controller 的方法, 可以跨模块
- `model(name, options)` 获取模型实例

完整方法列表请见 [API -> Controller](#)。

视图

视图即模版, 默认的根本目录为 `view/`。

视图文件

视图文件默认的命名规则为 `模块/控制器_操作.html`。

假如 URL `home/article/detail` 解析后的模块是 `home`, 控制器是 `article`, 操作是 `detail`, 那么对应的视图文件为 `home/article_detail.html`。

视图配置

视图默认配置如下，可以在配置文件 `src/common/config/view.js` 中修改：

```
export default {
  type: 'ejs', //模版引擎
  content_type: 'text/html', //输出模版时发送的 Content-Type
  file_ext: '.html', //文件的扩展名
  file_depr: '_', //控制器和操作之间的连接符
  root_path: think.ROOT_PATH + '/view', //视图文件的根目录
  adapter: { //模版引擎需要的配置项
    ejs: {}, //使用 ejs 模板引擎时额外配置
    nunjucks: {} //使用 nunjucks 模板引擎时额外配置
  }
};
```

注：2.0.6 版本开始去除了 `options` 配置项，使用 `adapter` 代替。

视图默认根目录在 `view/`。如果想每个模块有独立的视图目录，将配置 `root_path` 修改为空即可。

修改连接符

默认控制器和操作之间的连接符是 `_`，文件名类似为 `index_index.html`，如果想将控制器作为一层目录的话，如：`index/index.html`，可以将连接符修改为 `/`。

```
export default {
  file_depr: '/'
}
```

修改模板引擎配置

如果想修改模板引擎的一些配置，可以修改配置 `adapter` 里对应字段。如：

```
export default {
  adapter: {
    ejs: {
      delimiter: '&' //将定界符修改为 <& 和 &>
    },
    nunjucks: {
      trimBlocks: false, //不转义
      prerender: function(nunjucks, env){} //针对nunjucks模板的过滤器
    }
  }
}
```

模版引擎

ThinkJS 默认支持的模版引擎有: `ejs`, `jade`, `swig` 和 `nunjucks`, 默认模版引擎为 `ejs`, 可以根据需要修改为其他的模版引擎。

ejs

定界符

ejs 默认的定界符是 `<%` 和 `%>`。如果想修改定界符, 可以通过配置 `adapter` 里的 `ejs` 来修改, 如:

```
export default {
  adapter: {
    ejs: {
      delimiter: '&' //将定界符修改为 <& 和 &>
    }
  }
}
```

变量输出

- 转义输出 `<%= data.name%>`
- 不转义输出 `<%- data.name%>`
- 注释 `<%# data.name%>`

条件判断

```
<%if(data.name === '1'){%>
  <p>...</p>
<%}else if(data.name === '2'){%>
  <p>...</p>
<%}else{%>
  <p>...</p>
<%}%>
```

循环

```
<%list.forEach(function(item){%>
  <li><%=item.name%></li>
<%})%>
```

过滤器

新版的 `ejs` 已经不再支持过滤器的功能了，如果需要一些过滤功能，可以在 `src/common/bootstrap/` 里定义一些全局函数，模板里可以直接使用这些函数。

引用文件

`ejs` 不支持模版继承。但可以将公用的模版独立成一个文件，然后通过 `include` 来引入。

```
<%include inc/header.html%>
```

注： `ejs` 模版使用的变量需要在控制器中赋值，否则会报错。

更多 `ejs` 使用文档请见 [这里](#)。

nunjucks

`nunjucks` 是一款类似于 `jinja2` 的模版引擎，功能异常强大，复杂项目建议使用该模版引擎。

定界符

块级定界符为 `{%` 和 `%}`，变量定界符为 `{{` 和 `}}`，注释定界符为 `<#` 和 `#>`。如：

```
{{ username }}

{% block header %}
This is the default content
{% endblock %}
```

变量输出

可以通过 `{{ username }}` 来输出变量，默认输出的变量会自动转义，如果不想被转义，可以通过 `{{ username | safe }}` 来处理。

模版继承

父级模版：

```
{% block header %}
This is the default content
{% endblock %}

<section class="left">
  {% block left %}{% endblock %}
</section>
```

```
<section class="right">
  {% block right %}
  This is more content
  {% endblock %}
</section>
```

子级模版:

```
{% extends "./parent.html" %}

{% block left %}
This is the left side!
{% endblock %}

{% block right %}
This is the right side!
{% endblock %}
```

注: nunjucks 默认设置了 `root_path`, 所以模板继承时需要使用相对路径。

条件判断

```
{% if hungry %}
  I am hungry
{% elif tired %}
  I am tired
{% else %}
  I am good!
{% endif %}
```

循环

```
<h1>Posts</h1>
<ul>
  {% for item in items %}
    <li>{{ item.title }}</li>
  {% else %}
    <li>This would display if the 'item' collection were empty</li>
  {% endfor %}
</ul>
```

具体使用文档请见 [这里](#)。

jade

jade 模版使用方式请见 [这里](#)。

swig

swig 模版使用方式请见 [这里](#)。

添加过滤器等功能

`swig`，`nunjucks` 等很多模板引擎都支持添加过滤器等功能，可以在模板配置文件 `src/common/config/view.js` 中对应的 `adapter` 添加 `prerender` 配置来完成。如：

```
export default {
  adapter: {
    nunjucks: {
      prerender: function(nunjucks, env){
        //添加一个过滤器，这样可以在模板里使用了
        env.addFilter('filter_foo', function(){

        });
      }
    }
  }
}
```

注：该功能是在版本 `2.0.5` 中新增加的。

扩展模版引擎

模版引擎使用 Adapter 实现。如果项目里需要使用其他模版引擎，可以通过 Adapter 进行扩展，具体请见 [这里](#)。

变量赋值

控制器中可以通过 `assign` 方法进行变量赋值。

赋值单个变量

```
export default class extends think.controller.base {
  indexAction(){
    this.assign('title', 'ThinkJS 官网');
  }
}
```

```
}
```

赋值多个变量

```
export default class extends think.controller.base {
  indexAction(){
    this.assign({
      title: 'ThinkJS 官网',
      author: 'thinkjs'
    });
  }
}
```

获取赋值

变量赋值后也可以通过 `assign` 来获取赋过的值。如：

```
export default class extends think.controller.base {
  indexAction(){
    this.assign('title', 'ThinkJS 官网');
    let title = this.assign('title');
  }
}
```

模版渲染

可以通过 `display` 方法进行模版渲染。如果不传具体的模版文件路径，会自动查找。如：

```
export default class extends think.controller.base {
  indexAction(){
    this.display();// render home/index_index.html
  }
}
```

也可以指定具体的模版文件进行渲染，关于 `display` 方法的详细使用请见 [这里](#)。

获取渲染后的内容

如果有时候不想支持输出模版，而是想获取渲染后的模版内容，那么可以通过 `fetch` 方法来获取。

ES6 方式

```
export default class extends think.controller.base {
  async indexAction(){
    let content = await this.fetch();
    ...
  }
}
```

动态创建类的方式

```
module.exports = think.controller({
  indexAction: function(){
    this.fetch().then(function(content){
      ...
    })
  }
})
```

关于 `fetch` 方法的详细使用方式请见 [这里](#)。

国际化

启动国际化后，视图路径会多一层国际化的目录。如：具体的视图路径变为

`view/zh-cn/home/index_index.html`，其中 `zh-cn` 为语言名。

关于如果使用国际化请见 [扩展功能 -> 国际化](#)。

多主题

设置多主题后，视图路径会多一层多主题的目录。如：具体的视图路径变为

`view/default/home/index_index.html`，其中 `default` 为主题名称。

可以通过 `http.theme` 方法来设置当前的主题，设置主题一般是通过 middleware 来实现。

关于 middleware 更多信息请见 [扩展功能 - Middleware](#)。

默认模版变量

为了可以在模版里很方便的获取一些通用的变量，框架自动向模版里注册了 `http`，`controller`，`config` 等变量，这些变量可以在模版里直接使用。

下面的代码示例都是基于 `ejs` 模版引擎的，其他的模版引擎下需要根据相应的语法进行修改。

http

模版里可以直接使用 `http` 对象下的属性和方法。

controller

模版里可以直接使用 `controller` 对象下的属性和方法。

```
export default class extends think.controller.base {
  indexAction(){
    this.navType = 'home';
  }
}
```

Action 里给当前控制器添加了属性 `navType`，那么模版里就可以直接通过 `controller.navType` 来使用。

```
<%if(controller.navType === 'home'){%>
  <li className="action">home</li>
<%}else{%>
  <li>home</li>
<%}%>
```

config

通过 `config` 对象可以在模版中直接对应的配置，如：

```
<%if(config.name === 'text'){%>

<%}%>
```

国际化方法 `_`

在模版中可以直接通过 `_` 方法获取对应本地化的值，这些值在 `src/common/config/locales/[lang].js` 中定义。

```
<%= _('title')%>
```

更多国际化相关的信息请见 [这里](#)。

配置

ThinkJS 提供了灵活的配置，可以在不同的模块和不同的项目环境下使用不同的配置，且这些配置在服务启动时就已经生效。

注意：不可将一个 http 请求中的私有值设置到配置中，这将会被下一个 http 设置的值给冲掉。

项目模块

ThinkJS 默认创建的项目是按模块来划分的，可以在每个模块下定义不同的配置。其中 `common` 模块下定义一些通用的配置，其他模块下配置会继承 `common` 下的配置。如：`home` 模块下的最终配置是将 `common` 和 `home` 模块下配置合并的结果。

项目环境

ThinkJS 默认支持 3 种项目环境，可以根据不同的环境进行配置，以满足不同情况下的配置需要。

- `development` 开发环境
- `testing` 测试环境
- `production` 线上环境

项目里也可以扩展其他的环境，当前使用哪种环境可以在 [入口文件](#) 中设置，设置 `env` 值即可。

定义配置文件

`config/config.js`

存放一些基本的配置，如：

```
export default {
  port: 8360,
  host: '',
  encoding: 'utf-8',
  ...
}
```

`config/[name].js`

存放具体某个独立功能的配置，如：`db.js` 为数据库配置，`redis` 为 redis 配置。

```
// db.js
export default {
  type: 'mysql',
  ...
};
```

config/env/[mode].js

不同项目环境的差异化配置，如：`env/development.js`，`env/testing.js`，`env/production.js`

```
// config/env/development.js
export default {
  port: 7777,
  db: { //开发模式下数据库配置
    type: 'mysql',
    adapter: {
      mysql: {
        host: '127.0.0.1',
        port: '',
      }
    }
  },
  ...
}
```

注：不同项目环境差异化配置一般不是很多，所以放在一个文件中定义。这时候如果要修改一个独立功能的配置，就需要将独立功能对应的 key 带上。如：上述代码里的修改数据库配置需要将数据库对应的名称 `db` 带上。

config/locale/[lang].js

国际化语言包配置，如：`locale/en.js`，`locale/zh-cn.js`。

配置格式采用 `key: value` 的形式，并且 `key` 不区分大小写。

加载配置文件

框架支持多种级别的配置文件，会按以下顺序进行读取：

框架默认的配置 -> 项目模式下框架配置 -> 项目公共配置 -> 项目模式下的公共配置 -> 模块下的配置

配置读取

通过 config 方法获取

在 Controller, Logic, Middleware 等地方可以通过 `this.config` 来获取。如：

```
let db = this.config('db'); //读取数据库的所有配置
let host = this.config('db.host'); //读取数据库的 host 配置，等同于 db.host
```

通过 http 对象上的 config 方法获取

http 对象也有 config 方法用来获取相关的配置，如：

```
let db = http.config('db');
```

其他地方配置读取

其他地方可以通过 `think.config` 来读取相关的配置：

```
let db = think.config('db'); //读取通用模块下的数据库配置
let db1 = think.config('db', undefined, 'home'); //获取 home 模块下数据库配置
```

注：路由解析前，无法通过 `config` 方法或者 http 对象上的 `config` 方法来获取非通用模块下的配置，所以路由解析前就使用的配置需要定义在通用模块里。

系统默认配置

env

项目模式下的配置，`config/env/development.js`。

```
export default {
  auto_reload: true,
  log_request: true,
  gc: {
    on: false
  },
  error: {
    detail: true
  }
}
```

```
}
```

`config/env/testing.js` 和 `config/env/production.js` 无默认配置。

locale

国际化语言包配置，默认的配置如下：

```
// config/locale/en.js
export default {
  CONTROLLER_NOT_FOUND: 'controller `%s` not found. url is `%s`.',
  CONTROLLER_INVALID: 'controller `%s` is not valid. url is `%s`',
  ACTION_NOT_FOUND: 'action `%s` not found. url is `%s`',
  ACTION_INVALID: 'action `%s` is not valid. url is `%s`',
  WORKER_DIED: 'worker `%d` died, it will auto restart.',
  MIDDLEWARE_NOT_FOUND: 'middleware `%s` not found',
  ADAPTER_NOT_FOUND: 'adapter `%s` not found',
  GCTYPE_MUST_SET: 'instance must have gcType property',
  CONFIG_NOT_FUNCTION: 'config `%s` is not a function',
  CONFIG_NOT_VALID: 'config `%s` is not valid',
  PATH_EMPTY: '`%s` path must be set',
  PATH_NOT_EXIST: '`%s` is not exist',
  TEMPLATE_NOT_EXIST: 'can\'t find template file `%s`',
  PARAMS_EMPTY: 'params `%s` value can\'t empty',
  PARAMS_NOT_VALID: 'params `{name}` value not valid',
  FIELD_KEY_NOT_VALID: 'field `%s` in where condition is not valid',
  DATA_EMPTY: 'data can not be empty',
  MISS_WHERE_CONDITION: 'miss where condition',
  INVALID_WHERE_CONDITION_KEY: 'where condition key is not valid',
  WHERE_CONDITION_INVALID: 'where condition `%s`: `%s` is not valid',
  TABLE_NO_COLUMNS: 'table `%s` has no columns',
  NOT_SUPPORT_TRANSACTION: 'table engine is not support transaction',
  DATA_MUST_BE_ARRAY: 'data is not an array list',
  PARAMS_TYPE_INVALID: 'params `{name}` type invalid',
  DISALLOW_PORT: 'proxy on, cannot visit with port',
  SERVICE_UNAVAILABLE: 'Service Unavailable',

  validate_required: '{name} can not be blank',
  validate_contains: '{name} need contains {args}',
  validate_equals: '{name} need match {args}',
  validate_different: '{name} need not match {args}',
  validate_after: '{name} need a date that\'s after the {args} (defaults to now)',
  validate_alpha: '{name} need contains only letters (a-zA-Z)',
  validate_alphaDash: '{name} need contains only letters and dashes(a-zA-Z_)',
  validate_alphaNumeric: '{name} need contains only letters and numeric(a-zA-Z0-9)',
  validate_alphaNumericDash: '{name} need contains only letters, numeric and dash(a-zA-Z0-9_)',
  validate_ascii: '{name} need contains ASCII chars only',
  validate_base64: '{name} need a valid base64 encoded',
```

```

    validate_before: '{name} need a date that\'s before the {args} (defaults to now)
',
    validate_byteLength: '{name} need length (in bytes) falls in {args}',
    validate_creditcard: '{name} need a valid credit card',
    validate_currency: '{name} need a valid currency amount',
    validate_date: '{name} need a date',
    validate_decimal: '{name} need a decimal number',
    validate_divisibleBy: '{name} need a number that\'s divisible by {args}',
    validate_email: '{name} need an email',
    validate_fqdn: '{name} need a fully qualified domain name',
    validate_float: '{name} need a float in {args}',
    validate_fullWidth: '{name} need contains any full-width chars',
    validate_halfWidth: '{name} need contains any half-width chars',
    validate_hexColor: '{name} need a hexadecimal color',
    validate_hex: '{name} need a hexadecimal number',
    validate_ip: '{name} need an IP (version 4 or 6)',
    validate_ip4: '{name} need an IP (version 4)',
    validate_ip6: '{name} need an IP (version 6)',
    validate_isbn: '{name} need an ISBN (version 10 or 13)',
    validate_isin: '{name} need an ISIN (stock/security identifier)',
    validate_iso8601: '{name} need a valid ISO 8601 date',
    validate_in: '{name} need in an array of {args}',
    validate_notIn: '{name} need not in an array of {args}',
    validate_int: '{name} need an integer',
    validate_min: '{name} need an integer greater than {args}',
    validate_max: '{name} need an integer less than {args}',
    validate_length: '{name} need length falls in {args}',
    validate_minLength: '{name} need length is max than {args}',
    validate_maxLength: '{name} need length is min than {args}',
    validate_lowercase: '{name} need is lowercase',
    validate_mobile: '{name} need is a mobile phone number',
    validate_mongoId: '{name} need is a valid hex-encoded representation of a MongoDB
B ObjectId',
    validate_multibyte: '{name} need contains one or more multibyte chars',
    validate_url: '{name} need an URL',
    validate_uppercase: '{name} need uppercase',
    validate_variableWidth: '{name} need contains a mixture of full and half-width c
hars',
    validate_order: '{name} need a valid sql order string',
    validate_field: '{name} need a valid sql field string',
    validate_image: '{name} need a valid image file',
    validate_startWith: '{name} need start with {args}',
    validate_endWidth: '{name} need end with {args}',
    validate_string: '{name} need a string',
    validate_array: '{name} need an array',
    validate_boolean: '{name} need a boolean',
    validate_object: '{name} need an object'
}

```

config

基本配置, `config/config.js`。

```
export default {
  port: 8360, //服务监听的端口
  host: '', //服务监听的 host
  encoding: 'utf-8', //项目编码
  pathname_prefix: '', //pathname 去除的前缀, 路由解析中使用
  pathname_suffix: '.html', //pathname 去除的后缀, 路由解析中使用
  hook_on: true, //是否开启 hook
  cluster_on: false, //是否开启 cluster, 值为具体的数值时可以配置 `cluster` 的个数
  timeout: 120, //120 seconds
  auto_reload: false, //自动重新加载修改的文件, development 模式下使用

  resource_on: true, // 是否处理静态资源请求, proxy_on 开启下可以关闭该配置
  resource_reg: /^(static\/|[\^\/]+\.(?!js|html)\w+$/), //静态资源的正则

  route_on: true, //是否开启自定义路由

  log_error: true, //是否打印错误日志
  log_request: false, //是否打印请求的日志

  create_server: undefined, //自定义启动服务
  output_content: undefined, //自定义输出内容处理方式, 可以进行 gzip 处理等
  deny_module_list: [], //禁用的模块列表
  default_module: 'home', //默认模块
  default_controller: 'index', //默认的控制器
  default_action: 'index', //默认的动作
  callback_name: 'callback', //jsonp 请求的 callback 名称
  json_content_type: 'application/json', //json 输出时设置的 Content-Type
}
```

cache

缓存配置, `config/cache.js`。

```
export default {
  type: 'file', //缓存方式
  adapter: {
    file: {
      timeout: 6 * 3600, //6 hours
      path: think.RUNTIME_PATH + '/cache', //文件缓存模式下缓存内容存放的目录
      path_depth: 2, //子目录深度
      file_ext: '.json' //缓存文件的扩展名
    },
    redis: {
      prefix: 'thinkjs_', //缓存名称前缀
    }
  }
};
```

cookie

cookie 配置, `config/cookie.js`。

```
export default {
  domain: '', // cookie domain
  path: '/', // cookie path
  httponly: false, //是否 httponly
  secure: false, //是否在 https 下使用
  timeout: 0 //cookie 有效时间
};
```

db

数据库配置, `config/db.js`。

```
export default {
  type: 'mysql', //数据库类型
  log_sql: true, //是否记录 sql 语句
  log_connect: true, // 是否记录连接数据库的信息
  adapter: {
    mysql: {
      host: '127.0.0.1', //数据库 host
      port: '', //端口
      database: '', //数据库名称
      user: '', //账号
      password: '', //密码
      prefix: 'think_', //数据表前缀
      encoding: 'utf8', //数据库编码
      nums_per_page: 10, //一页默认条数
    }
  }
};
```

error

错误信息配置, `config/error.js`。

```
export default {
  key: 'errno', //error number
  msg: 'errmsg', //error message
};
```

```
value: 1000 //default errno
};
```

gc

缓存、Session等垃圾处理配置, `config/gc.js`。

```
export default {
  on: true, //是否开启垃圾回收处理
  interval: 3600, // 处理时间间隔, 默认为一个小时
  filter: function(){ //如果返回 true, 则进行垃圾回收处理
    let hour = (new Date()).getHours();
    if(hour === 4){
      return true;
    }
  }
};
```

hook

hook 配置, `config/hook.js`。

```
export default {
  request_begin: [],
  payload_parse: ['parse_form_payload', 'parse_single_file_payload', 'parse_json_payload', 'parse_querystring_payload'],
  payload_validate: ['validate_payload'],
  resource: ['check_resource', 'output_resource'],
  route_parse: ['rewrite_pathname', 'parse_route'],
  logic_before: [],
  logic_after: [],
  controller_before: [],
  controller_after: [],
  view_before: [],
  view_template: ['locate_template'],
  view_parse: ['parse_template'],
  view_filter: [],
  view_after: [],
  response_end: []
};
```

post

post 请求时的配置, `config/post.js`。

```
export default {
  json_content_type: ['application/json'],
  max_file_size: 1024 * 1024 * 1024, //1G
  max_fields: 100,
  max_fields_size: 2 * 1024 * 1024, //2M,
  ajax_filename_header: 'x-filename',
  file_upload_path: think.RUNTIME_PATH + '/upload',
  file_auto_remove: true
};
```

redis

redis 配置, `config/redis.js`。

```
export default {
  host: '127.0.0.1',
  port: 6379,
  password: '',
  timeout: 0,
  log_connect: true
};
```

memcache

memcache 配置, `config/memcache.js`。

```
export default {
  host: '127.0.0.1', //memcache host
  port: 11211,
  username: '', //
  password: '',
  timeout: 0, //缓存失效时间
  log_connect: true
};
```

session

session 配置, `config/session.js`。

```
export default {
  name: 'thinkjs',
  type: 'file',
  path: think.RUNTIME_PATH + '/session',
  secret: '',
  timeout: 24 * 3600,
  cookie: { // cookie options
    length: 32
  }
};
```

view

视图配置, `config/view.js`。

```
export default {
  content_type: 'text/html',
  file_ext: '.html',
  file_depr: '_',
  root_path: '',
  type: 'ejs',
  adapter: {
    ejs: {

    }
  }
};
```

websocket

websocket 配置, `config/websocket.js`。

```
export default {
  on: false, //是否开启 websocket
  type: 'think', //websocket 使用的库
  allow_origin: '',
  sub_protocal: '',
  adp: undefined,
  path: '', //url path for websocket
  messages: {
    // open: 'home/websocket/open',
  }
};
```

扩展配置

项目里可以根据需要扩展配置，扩展配置只需在 `src/common/config/` 建立对应的文件即可，如：

```
// src/common/config/foo.js
export default {
  name: 'bar'
}
```

这样就可以通过 `think.config('foo')` 来获取对应的配置了。

路由

当用户访问一个 URL 时，最终执行哪个模块下哪个控制器的哪个操作，这是由路由来解析后决定的。

ThinkJS 提供了一套灵活的路由机制，除了默认的解析外，还可以支持多种形式的自定义路由，让 URL 更加简单友好。

URL 解析为 pathname

当用户访问服务时，服务端首先拿到的是一个完整的 URL，如：访问本页面，得到的 URL 为 `http://www.thinkjs.org/zh-cn/doc/2.0/route.html`。

将 URL 进行解析得到的 pathname 为 `/zh-cn/doc/2.0/route.html`。

pathname 过滤

有时候为了搜索引擎优化或者一些其他的原因，URL 上会多加一些东西。比如：当前页面是一个动态页面，但 URL 最后加了 `.html`，这样对搜索引擎更加友好。但这些在后续的路由解析中是无用的，需要去除。

ThinkJS 里提供了下面的配置可以去除 `pathname` 的前缀和后缀内容：

```
export default {
  pathname_prefix: '',
  pathname_suffix: '.html',
}
```

上面配置可以在 `src/common/config/config.js` 中进行修改。

pathname 过滤时会自动去除左右的 `/`，该逻辑不受上面的配置影响。对 pathname 进行过滤后，拿到干净的 pathname 为 `zh-cn/doc/2.0/route`。

注：如果访问的 URL 是 `http://www.thinkjs.org/`，那么最后拿到干净的 pathname 则为空字符串。

子域名部署

子域名部署请见[Middleware -> 子域名部署](#)。

路由识别

路由解析

路由识别默认根据 `模块/控制器/操作/参数1/参数1值/参数2/参数2值` 来识别过滤后的 pathname，如：pathname 为 `admin/group/detail`，那么识别后的结果为：

- module 为 `admin`
- controller 为 `group`
- action 为 `detail`，对应的方法名为 `detailAction`

如果项目里并没有 `admin` 这个模块或者这个模块被禁用了，那么识别后的结果为：

- module 为默认模块 `home`
- controller 为 `admin`
- action 为 `group`，对应的方法名为 `groupAction`
- 参数为 `{detail: ''}`

如果有多级控制器，那么会进行多级控制器的识别，然后才是 action 的识别。

大小写转化

路由识别后，`module`、`controller` 和 `Action` 值都会自动转为小写。如果 Action 值里有 `_`，会作一些转化，如：假设识别后的 Controller 值为 `index`，Action 值为 `user_add`，那么对应的 Action 方法名为 `userAddAction`，但模版名还是 `index_user_add.html`。

路由默认值

当解析 pathname 没有对应的值时，此时便使用对应的默认值。其中 module 默认值为 `home`，controller 默认值为 `index`，action 默认值为 `index`。

这些值可以通过下面的配置进行修改，配置文件 `src/common/config/config.js`：

```
export default {
  default_module: 'home',
  default_controller: 'index',
  default_action: 'index',
}
```

自定义路由

默认的路由虽然看起来清晰明了，解析起来也很简单，但看起来不够简洁。

有时候需要更加简洁的路由，这时候就需要使用自定义路由解析了。如：文章的详细页面，默认路由可能是：`article/detail/id/10`，但我们想要的 URL 是 `article/10` 这种更简洁的方式。

开启配置

开启自定义路由，需要在 `src/common/config/config.js` 开启如下的配置：

```
export default {
  route_on: true
}
```

路由规则

开启自定义路由后，就可以通过路由规则来定义具体的路由了，路由配置文件为：

`src/common/config/route.js`，格式如下：

```
export default [
  ["规则1", "需要识别成的pathname"],
  ["规则2", {
    get: "get请求下需要识别成的pathname",
    post: "post请求下需要识别成的pathname"
  }]
];
```

注：自定义路由每一条规则都是一个数组。（至于为什么不用对象，是因为正则路由下，正则不能作为对象的 key 直接使用）

识别方式

自定义路由的匹配规则为：从前向后逐一匹配，如果命中到了该项规则，则不在向后匹配。

ThinkJS 支持 3 种类型的自定义路由，即：正则路由，规则路由和静态路由。

正则路由

正则路由是采用正则表示式来定义路由的一种方式，依靠强大的正则表达式，能够定义非常灵活的路由规则。

```
export default [
  [/^article\/(\d+)$/, "home/article/detail?id=:1"]
];
```

上面的正则会匹配类似 `article/10` 这样的 pathname，识别后新的 pathname 为 `home/article/detail`，并且将捕获到的值赋值给参数 `id`，这样在控制器里就可以通过 `this.get` 方法来获取该值。

```
export default class extends think.controller.base {
  detailAction(){
    let id = this.get('id');
  }
}
```

如果正则里含有多个子分组，那么可以通过 `:1`，`:2`，`:3` 来获取对应的值。

```
export default [
  [/^article\/(\d+)$/, {
    get: "home/article/detail?id=:1",
    delete: "home/article/delete?id=:1",
    post: "home/article/save?id=:1"
  }]
];
```

规则路由

规则路由是一种字符串匹配方式，但支持含有一些动态的值。如：

```
export default [
  ['group/:year/:month', "home/group/list"]
]
```

假如访问的 URL 为 `http://www.example.com/group/2015/10`，那么会命中该项规则，得到的 pathname 为 `home/group/list`，同时会添加 2 个参数 `year` 和 `month`，这 2 个参数可以在控制器里通过 `this.get` 方法来获取。

```
export default class extends think.controller.base {
  listAction(){
    let year = this.get('year');
    let month = this.get('month');
  }
}
```

静态路由

静态路由是一种纯字符串的完全匹配方式，写法和识别都很简单，功能也相对要弱很多。

```
export default [
  ["list", "home/article/list"]
]
```

假如访问的 URL 为 `http://www.example.com/list`，那么替换后的 pathname 为 `home/article/list`。

优化路由性能

上面已经说到，自定义路由是个数组，数组每一项是个具体的路由规则，匹配时是从前向后逐一进行匹配。如果这个路由表比较大的话，可能会有性能问题。

为了避免有性能问题，ThinkJS 提供了一种更加高效的自定义路由方式，按模块来配置路由。使用这种方式，路由配置格式跟上面稍微有所不同。

common/config/route.js

使用这种方式后，通用模块里的路由配置不再配置具体的路由规则，而是配置哪些规则命中到哪个模块。如：

```
export default {
  admin: {
    reg: /^admin/ //命中 admin 模块的正则
  },
  home: { //默认走 home 模块
  }
}
```

注: 此方式如果需要多个路由规则匹配同一个模块, 只需要修改正则即可, 对象的属性必须是 `reg: your reg` 格式且只有一条.

example:

```
export default {
  admin: {
    reg: /^(account|admin)/ // account和admin 都命中admin模块
  }
}
```

错误的写法 example:

```
export default {
  admin: {
    account: /^account/,
    admin : /^admin/
  }
}
```

admin/config/route.js

admin 模块配置 admin 下的具体路由规则。

```
export default [
  [/^admin\/(?!api).*$/, 'admin/index'],
  [/^admin\/api\/(\w+)?(?:\/([\d,]*))?$/, 'admin/:1?id=:2&resource=:1'],
];
```

假设访问的 URL 为 `http://www.example.com/admin/api`, 那么解析后的 pathname 为 `admin/api`, 匹配 `common` 里的规则时会命中 `admin` 模块, 然后再对 `admin` 模块下的路由规则进行逐一匹配。通过这种方式后就可以大大减少路由规则匹配的数量, 提高匹配效率。

自定义首页路由

首页默认执行的是 index controller 里的 indexAction。有些项目里希望对首页路由自定义, 但配置 `['', 'index/list']` 并不管用。

ThinkJS 为了性能上的考虑不支持对首页进行自定义路由, 因为更多情况下首页是不用自定义的, 并且访问的量比较大。如果支持自定义, 每次都要把自定义路由过一遍, 比较费性能。

模型

模型介绍

项目开发中，经常要操作数据库，如：增删改查等操作。模型就是为了方便操作数据库进行的封装，一个模型对应数据库中的一个数据表。

目前支持的数据库有：`MySQL`，`MongoDB`，`PostgreSQL` 和 `SQLite`。

创建模型

可以在项目目录下通过命令 `thinkjs model [name]` 来创建模型：

```
thinkjs model user;
```

执行完成后，会创建文件 `src/common/model/user.js`。

默认情况下模型文件会创建在 `common` 模块下，如果想创建在其他的模块下，创建时需要指定模块名：

```
thinkjs model home/user
```

注：模型文件不是必须存在，如果没有自定义方法可以不创建模型文件，实例化时会取模型基类的实例。

模型属性

model.pk

主键 key，默认为 `id`。MongoDB 下为 `_id`。

model.schema

数据表字段定义，默认会从数据库中读取，读到的信息类似如下：

```

{
  id: {
    name: 'id',
    type: 'int', //类型
    required: true, //是否必填
    primary: true, //是否是主键
    unique: true, //是否唯一
    auto_increment: true //是否自增
  }
}

```

可以在模型添加额外的属性，如：默认值和是否只读，如：

```

export default class extends think.model.base {
  /**
   * 数据表字段定义
   * @type {Object}
   */
  schema = {
    view_nums: { //阅读数
      default: 0 //默认为 0
    },
    fullname: { //全名
      default: function() { //first_name 和 last_name 的组合, 这里不能用 Arrows Function
        return this.first_name + this.last_name;
      }
    },
    create_time: { //创建时间
      default: () => { //获取当前时间
        return moment().format('YYYY-MM-DD HH:mm:ss')
      },
      readonly: true //只读, 添加后不可修改
    }
  }
}

```

`default` 默认只在添加数据时有效。如果希望在更新数据时也有效，需要添加属性 `update: true`，该属性在 2.1.4 版本中开始支持。

`readonly` 只在更新时有效。

注：如果设置了 `readonly`，那么会忽略 `update` 属性。

更多属性请见 [API -> Model](#)。

模型实例化

模型实例化在不同的地方使用的方式有所差别，如果当前类含有 `model` 方法，那可以直接通过该方法实例化，如：

```
export default class extends think.controller.base {
  indexAction(){
    let model = this.model('user');
  }
}
```

否则可以通过调用 `think.model` 方法获取实例化，如：

```
let getModelInstance = function(){
  let model = think.model('user', think.config('db'), 'home');
}
```

使用 `think.model` 获取模型的实例化时，需要带上模型的配置。

链式调用

模型中提供了很多链式调用的方法（类似 jQuery 里的链式调用），通过链式调用方法可以方便的进行数据操作。链式调用是通过返回 `this` 来实现的。

```
export default class extends think.model.base {
  /**
   * 获取列表数据
   */
  async getList(){
    let data = await this.field('title, content').where({
      id: ['>', 100]
    }).order('id DESC').select();
    ...
  }
}
```

模型中支持链式调用的方法有：

- `where`，用于查询或者更新条件的定义
- `table`，用于定义要操作的数据表名称
- `alias`，用于给当前数据表定义别名

- `data`, 用于新增或者更新数据之前的数据对象赋值
- `field`, 用于定义要查询的字段, 也支持字段排除
- `order`, 用于对结果进行排序
- `limit`, 用于限制查询结果数据
- `page`, 用于查询分页, 生成 sql 语句时会自动转换为 limit
- `group`, 用于对查询的 group 支持
- `having`, 用于对查询的 having 支持
- `join`, 用于对查询的 join 支持
- `union`, 用于对查询的 union 支持
- `distinct`, 用于对查询的 distinct 支持
- `cache` 用于查询缓存

链式调用方法具体使用方式请见 [API -> Model](#)。

数据库配置

数据库配置

数据库默认配置如下, 可以在 `src/common/config/db.js` 中进行修改:

```
export default {
  type: 'mysql',
  log_sql: true,
  log_connect: true,
  adapter: {
    mysql: {
      host: '127.0.0.1',
      port: '',
      database: '', //数据库名称
      user: '', //数据库帐号
      password: '', //数据库密码
      prefix: 'think_',
      encoding: 'utf8'
    },
    mongo: {
    }
  }
};
```

也可以在其他模块下配置, 这样请求该模块时就会使用对应的配置。

数据表定义

默认情况下，模型名和数据表名都是一一对应的。假设数据表前缀是 `think_`，那么 `user` 模型对应的数据表为 `think_user`，`user_group` 模型对应的数据表为 `think_user_group`。

如果需要修改，可以通过下面 2 个属性进行：

- `tablePrefix` 表前缀
- `tableName` 表名，不包含前缀

ES6 方式

```
export default class extends think.model.base {
  init(...args){
    super.init(...args);
    this.tablePrefix = ''; //将数据表前缀设置为空
    this.tableName = 'user2'; //将对应的数据表名设置为 user2
  }
}
```

动态创建类方式

```
module.exports = think.model({
  tablePrefix: '', //直接通过属性来设置表前缀和表名
  tableName: 'user2',
  init: function(){
    this.super('init', arguments);
  }
})
```

修改主键

模型默认的主键为 `id`，如果数据表里的 Primary Key 设置不是 `id`，那么需要在模型中设置主键。

```
export default class extends think.model.base {
  init(...args){
    super.init(...args);
    this.pk = 'user_id'; //将主键字段设置为 user_id
  }
}
```

`count`，`sum`，`min`，`max` 等很多查询操作都会用到主键，用到这些操作时需要修改主键。

配置多个数据库

如果项目中有连接多个数据库的需求，可以通过下面的方式连接多个数据库。

```
//src/common/config/db.js
export default {
  type: 'mysql',
  mysql: {
    host: '127.0.0.1',
    port: '',
    database: 'test1',
    user: 'root1',
    password: 'root1',
    prefix: '',
    encoding: 'utf8'
  },
  mysql2: {
    type: 'mysql', //这里需要将 type 重新设置为 mysql
    host: '127.0.0.1',
    port: '',
    database: 'test2',
    user: 'root2',
    password: 'root2',
    prefix: '',
    encoding: 'utf8'
  }
}
```

注意：`mysql2` 的配置中需要额外增加 `type` 字段将类型设置为 `mysql`。

配置完成后，调用的地方可以通过下面的方式调用。

```
export default class extends think.controller.base {
  indexAction(){
    let model1 = this.model('test'); //
    let model2 = this.model('test', 'mysql2'); //指定使用 mysql2 的配置连接数据库
  }
}
```

分布式数据库

大的系统中，经常有多个数据库用来做读写分离，从而提高数据库的操作性能。ThinkJS 里可以通过 `parser` 来自定义解析，可以在文件 `src/common/config/db.js` 中修改：

```
//读配置
```

```
const MYSQL_READ = {
  host: '10.0.10.1',
}
//写配置
const MYSQL_WRITE = {
  host: '10.0.10.2'
}
export default {
  host: '127.0.0.1',
  adapter: {
    mysql: {
      parser: function(options){ //mysql 的配置解析方法
        let sql = options.sql; //接下来要执行的 SQL 语句
        if(sql.indexOf('SELECT') === 0){ //SELECT 查询
          return MYSQL_READ;
        }
        return MYSQL_WRITE;
      }
    }
  }
}
```

parser 解析的参数 `options` 里会包含接下来要执行的 SQL 语句，这样就很方便的在 parser 里返回对应的数据库配置。

CRUD 操作

添加数据

添加一条数据

使用 `add` 方法可以添加一条数据，返回值为插入数据的 id。如：

```
export default class extends think.controller.base {
  async addAction(){
    let model = this.model('user');
    let insertId = await model.add({name: 'xxx', pwd: 'yyy'});
  }
}
```

添加多条数据

使用 `addMany` 方法可以添加多条数据，如：

```
export default class extends think.controller.base {
  async addAction(){
    let model = this.model('user');
    let insertId = await model.addMany([
      {name: 'xxx', pwd: 'yyy'},
      {name: 'xxx1', pwd: 'yyy1'}
    ]);
  }
}
```

thenAdd

数据库设计时，我们经常需要把某个字段设为唯一，表示这个字段值不能重复。这样添加数据的时候只能先去查询下这个数据值是否存在，如果不存在才进行插入操作。

模型中提供了 `thenAdd` 方法简化这一操作。

```
export default class extends think.controller.base {
  async addAction(){
    let model = this.model('user');
    //第一个参数为要添加的数据，第二个参数为添加的条件，根据第二个参数的条件查询无相关记录时才会
    添加
    let result = await model.thenAdd({name: 'xxx', pwd: 'yyy'}, {name: 'xxx'});
    // result returns {id: 1000, type: 'add'} or {id: 1000, type: 'exist'}
  }
}
```

更新数据

update

更新数据使用 `update` 方法，返回值为影响的行数。如：

```
export default class extends think.controller.base {
  async updateAction(){
    let model = this.model('user');
    let affectedRows = await model.where({name: 'thinkjs'}).update({email: 'admin@thinkjs.org'});
  }
}
```

默认情况下更新数据必须添加 where 条件，以防止误操作导致所有数据被错误的更新。如果确认是更新所有数据的需求，可以添加 `1=1` 的 where 条件进行，如：

```
export default class extends think.controlle.base {
  async updateAction(){
    let model = this.model('user');
    let affectedRows = await model.where('1=1').update({email: 'admin@thinkjs.org'
});
  }
}
```

increment

可以通过 `increment` 方法给符合条件的字段增加特定的值，如：

```
export default class extends think.model.base {
  updateViewNums(id){
    return this.where({id: id}).increment('view_nums', 1); //将阅读数加 1
  }
}
```

decrement

可以通过 `decrement` 方法给符合条件的字段减少特定的值，如：

```
export default class extends think.model.base {
  updateViewNums(id){
    return this.where({id: id}).decrement('coins', 10); //将金币减 10
  }
}
```

查询数据

模型中提供了多种方式来查询数据，如：查询单条数据，查询多条数据，读取字段值，读取最大值，读取总条数等。

查询单条数据

可以使用 `find` 方法查询单条数据，返回值为对象。如：

```
export default class extends think.controller.base {
  async listAction(){
    let model = this.model('user');
    let data = await model.where({name: 'thinkjs'}).find();
    //data returns {name: 'thinkjs', email: 'admin@thinkjs.org', ...}
  }
}
```

如果数据表没有对应的数据，那么返回值为空对象 `{}`，可以通过 `think.isEmpty` 方法来判断返回值是否为空。

查询多条数据

可以使用 `select` 方法查询多条数据，返回值为数据。如：

```
export default class extends think.controller.base {
  async listAction(){
    let model = this.model('user');
    let data = await model.limit(2).select();
    //data returns [{name: 'thinkjs', email: 'admin@thinkjs.org'}, ...]
  }
}
```

如果数据表中没有对应的数据，那么返回值为空数组 `[]`，可以通过 `think.isEmpty` 方法来判断返回值是否为空。

分页查询数据

页面中经常遇到按分页来展现某些数据，这种情况下就需要先查询总的条数，然后在查询当前分页下的数据。查询完数据后还要计算有多少页。模型中提供了 `countSelect` 方法来方便这一操作，会自动进行总条数的查询。

```
export default class extends think.controller.base {
  async listAction(){
    let model = this.model('user');
    let data = await model.page(this.get('page'), 10).countSelect();
  }
}
```

返回值格式如下：

```
{
  numsPerPage: 10, //每页显示的条数
  currentPage: 1, //当前页
  count: 100, //总条数
  totalPages: 10, //总页数
  data: [{ //当前页下的数据列表
    name: 'thinkjs',
    email: 'admin@thinkjs.org'
  }, ...]
}
```

如果传递的当前页数超过了页数范围，可以通过传递参数进行修正。`true` 为修正到第一页，`false` 为修正到最后一页，即：`countSelect(true)` 或 `countSelect(false)`。

如果总条数无法直接查询，可以将总条数作为参数传递进去，如：`countSelect(1000)`，表示总条数有1000条。

count

可以通过 `count` 方法查询符合条件的总条数，如：

```
export default class extends think.model.base {
  getMin(){
    //查询 status 为 publish 的总条数
    return this.where({status: 'publish'}).count();
  }
}
```

sum

可以通过 `sum` 方法查询符合条件的字段总和，如：

```
export default class extends think.model.base {
  getMin(){
    //查询 status 为 publish 字段 view_nums 的总和
    return this.where({status: 'publish'}).sum('view_nums');
  }
}
```

max

可以通过 `max` 方法查询符合条件的最大值，如：

```
export default class extends think.model.base {
  getMin(){
    //查询 status 为 publish, 字段 comments 的最大值
    return this.where({status: 'publish'}).max('comments');
  }
}
```

min

可以通过 `min` 方法查询符合条件的最小值，如：

```
export default class extends think.model.base {
  getMin(){
    //查询 status 为 publish, 字段 comments 的最小值
    return this.where({status: 'publish'}).min('comments');
  }
}
```

查询缓存

为了性能优化，项目中经常要对一些从数据库中查询的数据进行缓存。如果手工将查询的数据进行缓存，势必比较麻烦，模型中直接提供了 `cache` 方法来设置查询缓存。如：

```
export default class extends think.model.base {
  getList(){
    //设定缓存 key 和缓存时间
    return this.cache('get_list', 3600).where({id: {'>': 100}}).select();
  }
}
```

上面的代码为对查询结果进行缓存，如果已经有了缓存，直接从缓存里读取，没有的话才从数据库里查询。缓存保存的 key 为 `get_list`，缓存时间为一个小时。

也可以不指定缓存 key，这样会自动根据 SQL 语句生成一个缓存 key。如：

```
export default class extends think.model.base {
  getList(){
    //只设定缓存时间
    return this.cache(3600).where({id: {'>': 100}}).select();
  }
}
```

```
}
```

缓存配置

缓存配置为模型配置中的 `cache` 字段（配置文件在 `src/common/config/db.js`），如：

```
export default {  
  cache: {  
    on: true,  
    type: '',  
    timeout: 3600  
  }  
}
```

- `on` 数据库缓存配置的总开关，关闭后即使程序中调用 `cache` 方法也无效。
- `type` 缓存配置类型，默认为内存，支持的缓存类型请见 [Adapter -> Cache](#)。
- `timeout` 默认缓存时间。

删除数据

可以使用 `delete` 方法来删除数据，返回值为影响的行数。如：

```
export default class extends think.controller.base {  
  async deleteAction(){  
    let model = this.model('user');  
    let affectedRows = await model.where({id: ['>', 100]}).delete();  
  }  
}
```

模型中更多的操作方式请见相关的 [API -> model](#)。

事务

模型中提供了对事务操作的支持，但前提需要数据库支持事务。

`Mysql` 中的 `InnoDB` 和 `BDB` 存储引擎支持事务，如果在 `Mysql` 下使用事务的话，需要将数据库的存储引擎设置为 `InnoDB` 或 `BDB`。

`SQLite` 直接支持事务。

使用事务

模型中提供了 `startTrans` , `commit` 和 `rollback` 3 种方法来操作事务。

- `startTrans` 开启事务
- `commit` 正常操作后, 提交事务
- `rollback` 操作异常后进行回滚

ES6 方式

```
export default class extends think.controller.base {
  async indexAction(){
    let model = this.model('user');
    try{
      await model.startTrans();
      let userId = await model.add({name: 'xxx'});
      let insertId = await this.model('user_group').add({user_id: userId, group_id
: 1000});
      await model.commit();
    }catch(e){
      await model.rollback();
    }
  }
}
```

动态创建类的方式

```
module.exports = think.controller({
  indexAction: function(self){
    var model = this.model('user');
    return model.startTrans().then(function(){
      return model.add({name: 'xxx'});
    }).then(function(userId){
      return self.model('user_group').add({user_id: userId, group_id: 1000})
    }).then(function(){
      return self.commit();
    }).catch(function(err){
      return self.rollback();
    });
  }
})
```

transaction 方法

使用事务时，要一直使用 `startTrans`，`commit` 和 `rollback` 这 3 个方法进行操作，使用起来有一些不便。为了简化这一操作，模型中提供了 `transaction` 方法来更加方便的处理事务。

ES6 方式

```
export default class extends think.controller.base {
  async indexAction(self){
    let model = this.model('user');
    let insertId = await model.transaction( function * (){
      let userId = await model.add({name: 'xxx'});
      return await self.model('user_group').add({user_id: userId, group_id: 1000})
    });
  }
}
```

注：Arrow Function 无法和 `*/yield` 一起写，所以上面为 `function *`。如果想使用 Arrow Function，可以使用 `async`，如：`async () => {}`。

使用动态创建类的方式

```
module.exports = think.controller({
  indexAction: function(self){
    var model = this.model('user');
    return model.transaction(function(){
      return model.add({name: 'xxx'}).then(function(userId){
        return self.model('user_group').add({user_id: userId, group_id: 1000});
      });
    }).then(function(insertId){

    }).catch(function(err){

    })
  }
})
```

`transaction` 接收一个回调函数，这个回调函数中处理真正的逻辑，并需要返回。

操作多个模型

如果同一个事务下要操作多个数据表数据，那么要复用同一个数据库连接（开启事务后，每次事务操作会开启一个独立的数据库连接）。可以通过 `db` 方法进行数据库连接复用。

```
indexAction(){
  let model = this.model('user');
  await model.transaction(async () => {
    //通过 db 方法将 user 模型的数据库连接传递给 article 模型
    let model2 = this.model('article').db(model.db());
    // do something
  })
}
```

关联模型

数据库中表经常会跟其他数据表有关联，数据操作时需要连同关联的表一起操作。如：一个博客文章会有分类、标签、评论，以及属于哪个用户。

ThinkJS 中支持关联模型，让处理这类操作非常简单。

支持的类型

关联模型中支持常见的 4 类关联关系。如：

- `think.model.HAS_ONE` 一对一模型
- `think.model.BELONG_TO` 一对一属于
- `think.model.HAS_MANY` 一对多
- `think.model.MANY_TO_MANY` 多对多

创建关联模型

可以通过命令 `thinkjs model [name] --relation` 来创建关联模型。如：

```
thinkjs model home/post --relation
```

会创建模型文件 `src/home/model/post.js`。

指定关联关系

可以通过 `relation` 属性来指定关联关系。如：

```

export default class extends think.model.relation {
  init(...args){
    super.init(...args);
    //通过 relation 属性指定关联关系，可以指定多个关联关系
    this.relation = {
      cate: {},
      comment: {}
    };
  }
}

```

也可以直接使用 ES7 里的语法直接定义 `relation` 属性。如：

```

export default class extends think.model.relation {

  //直接定义 relation 属性
  relation = {
    cate: {},
    comment: {}
  };

  init(...args){
    super.init(...args);
  }
}

```

单个关系模型的数据格式

```

export default class extends think.model.relation {
  init(...args){
    super.init(...args);
    this.relation = {
      cate: {
        type: think.model.MANY_TO_MANY, //relation type
        model: '', //model name
        name: 'profile', //data name
        key: 'id',
        fKey: 'user_id', //foreign key
        field: 'id,name',
        where: 'name=xx',
        order: '',
        limit: '',
        rModel: '',
        rfKey: ''
      },
    };
  }
}

```

```
}
```

各个字段含义如下：

- `type` 关联关系类型
- `model` 关联表的模型名，默认为配置的 `key`，这里为 `cate`
- `name` 对应的数据字段名，默认为配置的 `key`，这里为 `cate`
- `key` 当前模型的关联 `key`
- `fKey` 关联表与只对应的 `key`
- `field` 关联表查询时设置的 `field`，如果需要设置，必须包含 `fKey` 对应的值
- `where` 关联表查询时设置的 `where` 条件
- `order` 关联表查询时设置的 `order`
- `limit` 关联表查询时设置的 `limit`
- `page` 关联表查询时设置的 `page`
- `rModel` 多对多下，对应的关联关系模型名
- `rfKey` 多对多下，对应里的关系关系表对应的 `key`

如果只用设置关联类型，不用设置其他字段信息，可以通过下面简单的方式：

```
export default class extends think.model.relation {
  init(...args){
    super.init(...args);
    this.relation = {
      cate: think.model.MANY_TO_MANY
    };
  }
}
```

HAS_ONE

一对一关联，表示当前表含有一个附属表。

假设当前表的模型名为 `user`，关联表的模型名为 `info`，那么配置中字段 `key` 的默认值为 `id`，字段 `fKey` 的默认值为 `user_id`。

```
export default class extends think.model.relation {
  init(..args){
    super.init(...args);
    this.relation = {
      info: think.model.HAS_ONE
    };
  }
}
```

执行查询操作时，可以得到类似如下的数据：

```
[
  {
    id: 1,
    name: '111',
    info: { //关联表里的数据信息
      user_id: 1,
      desc: 'info'
    }
  }, ...]
```

BELONG_TO

一对一关联，属于某个关联表，和 HAS_ONE 是相反的关系。

假设当前模型名为 `info`，关联表的模型名为 `user`，那么配置字段 `key` 的默认值为 `user_id`，配置字段 `fKey` 的默认值为 `id`。

```
export default class extends think.model.relation {
  init(..args){
    super.init(...args);
    this.relation = {
      user: think.model.BELONG_TO
    };
  }
}
```

执行查询操作时，可以得到类似下面的数据：

```
[
  {
    id: 1,
    user_id: 1,
    desc: 'info',
    user: {
      name: 'thinkjs'
    }
  }, ...
]
```

HAS_MANY

一对多的关系。

加入当前模型名为 `post`，关联表的模型名为 `comment`，那么配置字段 `key` 默认值为 `id`，配置字段 `fKey` 默认值为 `post_id`。

```
'use strict';
/**
 * relation model
 */
export default class extends think.model.relation {
  init(...args){
    super.init(...args);

    this.relation = {
      comment: {
        type: think.model.HAS_MANY
      }
    };
  }
}
```

执行查询数据时，可以得到类似下面的数据：

```
[{
  id: 1,
  title: 'first post',
  content: 'content',
  comment: [{
    id: 1,
    post_id: 1,
    name: 'welefen',
    content: 'first comment'
  }, ...]
}, ...]
```

如果关联表的数据需要分页查询，可以通过 `page` 参数进行，如：

```
'use strict';
/**
 * relation model
 */
export default class extends think.model.relation {
  init(...args){
    super.init(...args);
```

```

    this.relation = {
      comment: {
        type: think.model.HAS_MANY
      }
    };
  }
  getList(page){
    return this.setRelation('comment', {page: page}).select();
  }
}

```

除了用 `setRelation` 来合并参数外，可以将参数设置为函数，合并参数时会自动执行该函数。

MANY_TO_MANY

多对多关系。

假设当前模型名为 `post`，关联模型名为 `cate`，那么需要一个对应的关联关系表。配置字段 `rModel` 默认值为 `post_cate`，配置字段 `rfKey` 默认值为 `cate_id`。

```

'use strict';
/**
 * relation model
 */
export default class extends think.model.relation {
  init(...args){
    super.init(...args);

    this.relation = {
      cate: {
        type: think.model.MANY_TO_MANY,
        rModel: 'post_cate',
        rfKey: 'cate_id'
      }
    };
  }
}

```

查询出来的数据结构为：

```

[ {
  id: 1,
  title: 'first post',
  cate: [ {
    id: 1,
    name: 'cate1',

```

```
    post_id: 1
  }, ...]
}, ...]
```

关联死循环

如果 2 个关联表，一个设置对方为 HAS_ONE，另一个设置对方为 BELONG_TO，这样在查询关联表的数据时会将当前表又查询了一遍，并且会再次查询关联表，最终导致死循环。

可以在配置里设置 `relation` 字段关闭关联表的关联查询功能，从而避免死循环。如：

```
export default class extends think.model.relation {
  init(..args){
    super.init(...args);
    this.relation = {
      user: {
        type: think.model.BELONG_TO,
        relation: false //关联表 user 查询时关闭关联查询
      }
    };
  }
}
```

也可以设置只关闭当前模型的关联关系，如：

```
export default class extends think.model.relation {
  init(..args){
    super.init(...args);
    this.relation = {
      user: {
        type: think.model.BELONG_TO,
        relation: 'info' //关联表 user 查询时关闭对 info 模型的关联关系
      }
    };
  }
}
```

临时关闭关联关系

设置关联关系后，查询等操作都会自动查询关联表的数据。如果某些情况下不需要查询关联表的数据，可以通过 `setRelation` 方法临时关闭关联关系查询。

全部关闭

通过 `setRelation(false)` 关闭所有的关联关系查询。

```
export default class extends think.model.relation {
  init(...args){
    super.init(...args);
    this.relation = {
      comment: think.model.HAS_MANY,
      cate: think.model.MANY_TO_MANY
    };
  }
  getList(){
    return this.setRelation(false).select();
  }
}
```

部分启用

通过 `setRelation('comment')` 只查询 `comment` 的关联数据，不查询其他的关联关系数据。

```
export default class extends think.model.relation {
  init(...args){
    super.init(...args);
    this.relation = {
      comment: think.model.HAS_MANY,
      cate: think.model.MANY_TO_MANY
    };
  }
  getList2(){
    return this.setRelation('comment').select();
  }
}
```

部分关闭

通过 `setRelation('comment', false)` 关闭 `comment` 的关联关系数据查询。

```
export default class extends think.model.relation {
  init(...args){
    super.init(...args);
    this.relation = {
      comment: think.model.HAS_MANY,
      cate: think.model.MANY_TO_MANY
    };
  }
}
```

```
};
}
getList2(){
    return this.setRelation('comment', false).select();
}
}
```

重新全部启用

通过 `setRelation(true)` 重新启用所有的关联关系数据查询。

```
export default class extends think.model.relation {
    init(...args){
        super.init(...args);
        this.relation = {
            comment: think.model.HAS_MANY,
            cate: think.model.MANY_TO_MANY
        };
    }
    getList2(){
        return this.setRelation(true).select();
    }
}
```

设置查询条件

field

设置 field 可以控制查询关联表时数据字段，这样可以减少查询的数据量，提高查询效率。默认情况会查询所有数据。

如果设置了查询的字段，那么必须包含关联字段，否则查询出来的数据无法和之前的数据关联。

```
export default class extends think.model.relation {
    init(...args){
        super.init(...args);
        this.relation = {
            user: {
                type: think.model.BELONG_TO,
                field: 'id,name,email' //必须要包含关联字段 id
            }
        };
    }
}
```

如果某些情况下必须动态的设置的话，可以将 field 设置为一个函数，执行函数时返回对应的字段。如：

```
export default class extends think.model.relation {
  init(...args){
    super.init(...args);
    this.relation = {
      user: {
        type: think.model.BELONG_TO,
        field: model => {
          return model._relationField;
        }
      }
    };
  }
  selectData(relationfield){
    //将要查询的关联字段设置到一个私有属性中，便于动态设置 field 里获取
    this._relationField = relationfield;
    return this.select();
  }
}
```

形参 `model` 指向当前模型类。

where

设置 where 可以控制查询关联表时的查询条件，如：

```
export default class extends think.model.relation {
  init(...args){
    super.init(...args);
    this.relation = {
      user: {
        type: think.model.BELONG_TO,
        where: {
          grade: 1 //只查询关联表里 grade = 1 的数据
        }
      }
    };
  }
}
```

也可以动态的设置 where 条件，如：

```

export default class extends think.model.relation {
  init(...args){
    super.init(...args);
    this.relation = {
      user: {
        type: think.model.BELONG_TO,
        where: model => {
          return model._relationWhere;
        }
      }
    };
  }
  selectData(relationWhere){
    this._relationWhere = relationWhere;
    return this.select();
  }
}

```

形参 `model` 指向当前模型类。

page

可以通过设置 `page` 进行分页查询，`page` 参数会被解析为 `limit` 数据。

```

export default class extends think.model.relation {
  init(...args){
    super.init(...args);
    this.relation = {
      user: {
        type: think.model.BELONG_TO,
        page: [1, 15] //第一页，每页 15 条
      }
    };
  }
}

```

也可以动态设置分页，如：

```

export default class extends think.model.relation {
  init(...args){
    super.init(...args);
    this.relation = {
      user: {
        type: think.model.BELONG_TO,
        page: model => {
          return model._relationPage;
        }
      }
    };
  }
}

```

```

    }
  }
};
}
selectData(page){
  this._relationPage = [page, 15];
  return this.select();
}
}

```

形参 `model` 指向当前模型类。

limit

可以通过 `limit` 设置查询的条数，如：

```

export default class extends think.model.relation {
  init(...args){
    super.init(...args);
    this.relation = {
      user: {
        type: think.model.BELONG_TO,
        limit: [10] //限制 10 条
      }
    };
  }
}

```

也可以动态设置 `limit`，如：

```

export default class extends think.model.relation {
  init(...args){
    super.init(...args);
    this.relation = {
      user: {
        type: think.model.BELONG_TO,
        limit: model => {
          return model._relationLimit;
        }
      }
    };
  }
  selectData(){
    this._relationLimit = [1, 15];
    return this.select();
  }
}

```

形参 `model` 指向当前模型类。

注：如果设置 `page`，那么 `limit` 会被忽略，因为 `page` 会转为 `limit`。

order

通过 `order` 可以设置关联表的查询排序方式，如：

```
export default class extends think.model.relation {
  init(...args){
    super.init(...args);
    this.relation = {
      user: {
        type: think.model.BELONG_TO,
        order: 'level DESC'
      }
    };
  }
}
```

也可以动态的设置 `order`，如：

```
export default class extends think.model.relation {
  init(...args){
    super.init(...args);
    this.relation = {
      user: {
        type: think.model.BELONG_TO,
        order: model => {
          return model._relationOrder;
        }
      }
    };
  }
  selectData(){
    this._relationOrder= 'level DESC';
    return this.select();
  }
}
```

形参 `model` 指向当前模型类。

注：动态配置值从 `2.2.3` 版本开始支持。

注意事项

关联字段的数据类型要一致

比如：数据表里的字段 `id` 的类型为 `int`，那么关联表里的关联字段 `user_id` 也必须为 `int` 相关的类型，否则无法匹配数据。这是因为匹配的时候使用绝对等于进行判断的。

mongo 关联模型

该关联模型的操作不适合 mongo 模型，mongo 的关联模型请见 <https://docs.mongodb.org/manual/tutorial/model-embedded-one-to-one-relationships-between-documents/>。

Mysql

ThinkJS 对 Mysql 操作有很好的支持，底层使用的库为 <https://www.npmjs.com/package/mysql>。

连接池

默认连接 Mysql 始终只有一个连接，如果想要多个连接，可以使用连接池的功能。修改配置 `src/common/config/db.js`，如：

```
export default {
  connectionLimit: 10 //建立 10 个连接
}
```

socketPath

默认情况下是通过 host 和 port 来连接 Mysql 的，如果想通过 unix domain socket 来连接 Mysql，可以设置下面的配置：

```
export default {
  socketPath: '/tmp/mysql.socket'
}
```

SSL options

可以通过下面的配置来指定通过 SSL 来连接：

```
export default {
  ssl: {
    ca: fs.readFileSync(__dirname + '/mysql-ca.crt')
  }
}
```

数据库支持 emoji 表情

数据库的编码一般会设置为 `utf8`，但 `utf8` 并不支持 emoji 表情，如果需要数据库支持 emoji 表情，需要将数据库编码设置为 `utf8mb4`。

同时需要将 `src/common/config/db.js` 里的 `encoding` 配置值修改为 `utf8mb4`。如：

```
export default {
  encoding: 'utf8mb4'
}
```

Error: Handshake inactivity timeout

在某些 Node.js 版本下（如：4.2.0）连接 Mysql 时会出现下面的错误：

```
Error: Handshake inactivity timeout
at Handshake.sequence.on.on.on.on.on.self._connection._startTLS.err.code (/home/***/node_modules/mysql/lib/protocol/sequences/Handshake.js:104:21)
at Handshake.emit (events.js:92:17)
at Handshake._onTimeout (/home/***/node_modules/mysql/lib/protocol/sequences/Handshake.js:104:21)
at Timer.listOnTimeout [as ontimeout] (timers.js:112:15)
-----
at Protocol._enqueue (/home/***/node_modules/mysql/lib/protocol/Protocol.js:135:48)
at Protocol.handshake (/home/***/node_modules/mysql/lib/protocol/Protocol.js:52:41)
at PoolConnection.connect (/home/***/node_modules/mysql/lib/Connection.js:119:18)
at Pool.getConnection (/home/***/node_modules/mysql/lib/Pool.js:45:23)
at Object.exports.register (/home/***/node_modules/hapi-plugin-mysql/lib/index.js:10:10)
at /home/***/node_modules/hapi/lib/plugin.js:242:14
at iterate (/home/***/node_modules/hapi/node_modules/items/lib/index.js:35:13)
at done (/home/***/node_modules/hapi/node_modules/items/lib/index.js:27:25)
at Object.exports.register (/home/***/node_modules/lout/lib/index.js:95:5)
at /home/***/node_modules/hapi/lib/plugin.js:242:14
```

解决方案： 将 Node.js 升级到最新版本即可解决。

MongoDB

ThinkJS 支持使用 MongoDB 数据库，底层使用 `mongodb` 模块。

如果想在项目中使用 Mongoose 来代替默认的 Mongo 模型，可以参见：<http://welefen.com/post/use-mongoose-in-thinkjs.html>

配置

使用 MongoDB 数据库，需要将模型中的配置 `type` 改为 `mongo`，修改配置文件

`src/common/config/db.js`：

```
export default {
  type: 'mongo'
}
```

多 HOST

可以将配置里的 `host` 字段设置为数据支持多 host 的功能，如：

```
export default {
  type: 'mongo',
  adapter: {
    mongo: {
      host: ['10.0.0.1', '10.0.0.2'],
      port: ['1234', '5678']
    }
  }
}
```

注：此配置项在 `2.0.14` 版本中支持。

配置选项

如果要在连接 MongoDB 服务的时候添加额外的参数，可以通过在 `options` 里追加，如：

```
export default {
  type: 'mongo',
  adapter: {
    mongo: {
      options: {
```

```
    authSource: 'admin',
    replicaSet: 'xxx'
  }
}
}
```

上面的配置后，连接 MongoDB 的 URL 变成类似于

```
mongodb://127.0.0.1:27017/?authSource=admin&replicaSet=xxx。
```

更多额外的配置请见 <http://mongodb.github.io/node-mongodb-native/2.0/reference/connecting/connection-settings/>。

创建模型

可以通过命令 `thinkjs model [name] --mongo` 来创建模型，如：

```
thinkjs model user --mongo
```

执行完成后，会创建文件 `src/common/model/user.js`。如果想创建在其他模块下，需要带上具体的模块名。如：

```
thinkjs model home/user --mongo
```

会在 `home` 模块下创建模型文件，文件为 `src/home/model/user.js`。

模型继承

模型需要继承 `think.model.mongo` 类，如果当前类不是继承该类，需要手动修改。

ES6 语法

```
export default class extends think.model.mongo {
}
```

动态创建类的方式

```
module.exports = think.model('mongo', {  
  
})
```

CRUD 操作

CRUD 操作和 Mysql 中接口相同，具体请见 [模型 -> 介绍](#)。

创建索引

mongo 模型可以配置索引，在增删改查操作之前模型会自动去创建索引，配置放在 `indexes` 属性里。如：

```
export default class extends think.model.mongo {  
  init(...args){  
    super.init(...args);  
    //配置索引  
    this.indexes = {  
  
    }  
  }  
}
```

单字段索引

```
export default class extends think.model.mongo {  
  init(...args){  
    super.init(...args);  
    //配置索引  
    this.indexes = {  
      name: 1  
    }  
  }  
}
```

唯一索引

通过 `$unique` 来指定为唯一索引，如：

```
export default class extends think.model.mongo {
  init(...args){
    super.init(...args);
    //配置索引
    this.indexes = {
      name: {$unique: 1}
    }
  }
}
```

多字段索引

可以将多个字段联合索引，如：

```
export default class extends think.model.mongo {
  init(...args){
    super.init(...args);
    //配置索引
    this.indexes = {
      email: 1
      test: {
        name: 1,
        title: 1,
        $unique: 1
      }
    }
  }
}
```

获取索引

可以通过方法 `getIndexes` 获取创建的索引。如：

```
export default class extends think.controller.base {
  async indexAction(){
    let model = this.model('user');
    let indexes = await model.getIndexes();
  }
}
```

aggregate

可以通过 `aggregate` 方法进行混合操作。如：

```
export default class extends think.model.mongo {
  match(){
    return this.aggregate([
      {$match: {status: 'A'}},
      {$group: {_id: "$cust_id", total: {$sum: "$amount"}}}
    ]);
  }
}
```

具体请见 <https://docs.mongodb.org/manual/core/aggregation-introduction/>。

MapReduce

可以通过 `mapReduce` 方法进行 MapReduce 操作。如：

```
export default class extends think.model.mongo {
  execMapReduce(){
    let map = () => {
      emit(this.cust_id, this.amount);
    }
    let reduce = (key, values) => {
      return Array.sum(values);
    }
    return this.mapReduce(map, reduce, {
      query: {status: "A"},
      out: "order_totals"
    })
  }
}
```

具体请见 <https://docs.mongodb.org/manual/core/aggregation-introduction/#map-reduce>。

SQLite

ThinkJS 中支持使用 SQLite 数据库，底层使用 `sqlite3` 模块。

配置

使用 SQLite，需要将模型中的配置 `type` 改为 `sqlite`，修改配置文件

```
src/common/config/db.js:
```

```
export default {
  type: 'sqlite'
}
```

存储方式

SQLite 支持使用内存或者文件 2 种方式来存放数据，需要设置配置 `path`。

内存方式

```
export default {
  type: 'sqlite',
  adapter: {
    sqlite: {
      path: true, //使用内存来存储数据
    }
  }
}
```

文件方式

文件方式需要设置存储 SQLite 数据的目录，默认为 `src/common/runtime/sqlite`。

```
export default {
  type: 'sqlite',
  adapter: {
    sqlite: {
      path: '/path/to/store/sqlite' //设置存储数据文件的目录
    }
  }
}
```

对应的数据表文件路径为 `path + /[name].sqlite`，默认情况下数据库 `demo` 对应的文件路径为 `src/common/runtime/sqlite/demo.sqlite`。

CRUD 操作

CRUD 操作和 Mysql 相同，具体请见 [模型 -> 介绍](#)。

PostgreSQL

ThinkJS 支持 PostgreSQL，底层使用 `pg` 模块。

配置

使用 PostgreSQL，需要将模型中的配置 `type` 改为 `postgresql`，修改配置文件

`src/common/config/db.js`：

```
export default {
  type: 'postgresql',
  adapter: {
    postgresql: {

    }
  }
}
```

CRUD 操作

CRUD 操作和 Mysql 相同，具体请见 [模型 -> 介绍](#)。

Adapter

Adapter

Adapter 是用来解决一类功能的多种实现，如：支持多种数据库，支持多种模版引擎等。系统默认支持的 Adapter 有：`Cache`，`Session`，`WebSocket`，`Db`，`Store`，`Template` 和 `Socket`。

创建 Adapter

可以通过命令 `thinkjs adapter [type]/[name]` 来创建 Adapter，如：

```
thinkjs adapter template/dot
```

创建一个名为 `dot` 的 Template Adapter，创建的文件路径为 `src/common/adapter/template/dot.js`。文件内容类似如下：

```
export default class extends think.adapter.template {
  /**
   * init
   * @return {[[]]}
   */
  init(...args){
    super.init(...args);
  }
}
```

如果创建的类型之前不存在，会自动创建一个 Base 类，其他类会继承该类。

加载 Adapter

可以通过 `think.adapter` 方法加载对应的 Adapter，如：

```
let Template = think.adapter('template', 'dot'); //加载名为 dot 的 Template Adapter
let instance = new Template(...args); //实例化 Adapter
```

使用第三方 Adapter

加载 Adapter 时，系统会自动从 `src/common/adapter` 目录和系统目录查找对应的 Adapter，如果引入第三方的 Adapter，需要将 Adapter 注册进去，否则系统无法找到该 Adapter。

可以通过 `think.adapter` 方法注册第三方的 Adapter，如：

```
let DotTemplate = require('think-template-dot');
think.adapter('template', 'dot', DotTemplate);
```

将文件存放在 `src/common/bootstrap/` 目录下，这样服务启动时就会自动加载。

Cache

在项目中，合理使用缓存对性能有很大的帮助。ThinkJS 提供了多种的缓存方式，包括：内存缓存、文件缓存、Memcache 缓存、Redis 缓存等。

缓存类型

系统默认支持的缓存类型如下：

- `memory` 内存缓存
- `file` 文件缓存
- `memcache` Memcache 缓存
- `redis` Redis 缓存

如果使用 `memcache` 缓存，需要设置 Memcache 配置信息，见 [配置](#)。

如果使用 `redis` 缓存，需要设置 Redis 配置信息，见 [配置](#)。

缓存配置

默认缓存配置如下，可以在配置文件 `src/common/config/cache.js` 中进行修改：

```
export default {
  type: 'file', //缓存类型
  timeout: 6 * 3600, //失效时间，单位：秒
  adapter: { //不同 adapter 下的配置
    file: {
      path: think.RUNTIME_PATH + '/cache', //缓存文件的根目录
      path_depth: 2, //缓存文件生成子目录的深度
      file_ext: '.json' //缓存文件的扩展名
    },
    redis: {
      prefix: 'thinkjs_'
    },
    memcache: {
      prefix: 'thinkjs_'
    }
  }
};
```

注：`2.0.6` 版本开始添加了 `adapter` 配置。

其中 `prefix` 在 `memcache` 和 `redis` 类型中使用，存储时会将缓存 `key + prefix` 作为新的 `key` 来存储，用于防止跟其他地方使用的缓存 `key` 冲突。如果不想设置 `prefix`，可以将 `prefix` 设置为空字符串，如：

```
export default {
  prefix: '' //将缓存 key 前缀设置为空
}
```

使用缓存

可以通过 `think.cache` 方法对缓存进行增删改查操作，具体请见 [API -> think](#)。

如果当前使用场景在继承自 `think.http.base` 的类下，可以通过 `this.cache` 方法来操作缓存，具体请见 [API -> think.http.base](#)。

扩展缓存

可以通过下面的命令创建一个名为 `foo` 缓存类：

```
thinkjs adapter cache/foo
```

执行完成后，会创建文件 `src/common/adapter/cache/foo.js`。扩展缓存类需要实现如下的方法：

```
export default class extends think.cache.base {
  /**
   * 初始化方法
   * @param {Object} options []
   * @return {} []
   */
  init(options){
    //set gc type & start gc
    this.gcType = 'cache_foo';
    think.gc(this);
  }
  /**
   * 获取缓存
   * @param {String} name []
   * @return {Promise} []
   */
  get(name){

  }
  /**
   * 设置缓存
   * @param {String} name []
   * @param {Mixed} value []
   * @param {Number} timeout []
   * @return {Promise}
   */
  set(name, value, timeout){

  }
  /**
```

```
* 删除缓存
* @param {String} name []
* @return {Promise} []
*/
delete(name){

}
/**
 * 缓存垃圾回收
 * @return {Promise} []
 */
gc(){

}
}
```

框架里的 Cache 实现请见 <https://github.com/75team/thinkjs/tree/master/src/adapter/cache>。

使用第三方缓存 Adapter

如何使用第三方的缓存 Adapter 请参见 [Adapter -> 介绍](#)。

Session

需要用户登录的网站基本上都离不开 Session，ThinkJS 里默认支持多种类型的 Session，如：`file`，`db`，`redis` 等。

支持的 Session 类型

- `memory` 内存方式
- `file` 文件类型
- `db` 数据库类型
- `redis` Redis 类型

db Session

使用 `db` 类型的 Session 需要创建对应的数据表（如果是 MongoDB 则无需创建），可以用下面的 SQL 语句创建：

```
DROP TABLE IF EXISTS `think_session`;
CREATE TABLE `think_session` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `cookie` varchar(255) NOT NULL DEFAULT '',
  `data` text,
```

```
`expire` bigint(11) NOT NULL,  
PRIMARY KEY (`id`),  
UNIQUE KEY `cookie` (`cookie`),  
KEY `expire` (`expire`)  
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

需要将 `think_` 改为 db 配置中的数据表前缀。

redis Session

使用 `redis` 类型的 Session 需要配置 Redis，具体见 [配置](#)。

Session 配置

Session 默认配置如下，可以在 `src/common/config/session.js` 中进行修改：

```
export default {  
  type: 'file',  
  name: 'thinkjs', //对应 cookie 的名称  
  secret: '', //Session 对应的 cookie 是否需要加密  
  timeout: 24 * 3600, //过期时间，默认为一天  
  cookie: { // cookie options  
    length: 32  
  },  
  adapter: {  
    file: {  
      path: think.RUNTIME_PATH + '/session'  
    }  
  }  
};
```

注：`2.0.6` 版本开始添加了 adapter 配置。

关于 Cookie 的配置请见 [配置](#)。

Session 读写

Controller 或 Logic 里可以通过下面的方式读写 Session：

读取 Session

```
export default class extends think.controller.base {  
  async indexAction(){  
    //获取session  
    let value = await this.session('userInfo');
```

```
}  
}
```

设置 Session

```
export default class extends think.controller.base {  
  async indexAction(){  
    //设置 session  
    await this.session('userInfo', data);  
  }  
}
```

清除 Session

```
export default class extends think.controller.base {  
  async indexAction(){  
    //清除当前用户的 session  
    await this.session();  
  }  
}
```

http 对象上可以通过 `http.session` 方法读写 Session, 具体请见 [API -> http](#)。

扩展 Session

可以通过下面的命令创建 Session Adapter:

```
thinkjs adapter session/foo
```

会创建文件 `src/common/adapter/session/foo.js`, 需要实现下面的方法:

```
export default class extends think.adapter.session {  
  /**  
   * init  
   * @param {Object} options []  
   * @return {} []  
   */  
  init(options){  
  
  }  
  /**
```

```

    * 获取 Session
    * @param {String} name []
    * @return {Promise} []
    */
    get(name){

    }
    /**
    * 设置 Session
    * @param {String} name []
    * @param {Mixed} value []
    */
    set(name, value){

    }
    /**
    * 删除 Session
    * @param {String} name []
    * @return {Promise} []
    */
    delete(name){

    }
    /**
    * 更新 Session
    * @return {Promise} []
    */
    flush(){

    }
    /**
    * 清除过期的 Session
    * @return {Promise} []
    */
    gc(){

    }
}

```

框架里的 Session 实现请见 <https://github.com/75team/thinkjs/tree/master/src/adapter/session>。

使用第三方 Session Adapter

如何使用第三方的缓存 Adapter 请参见 [Adapter -> 介绍](#)。

WebSocket

项目里经常要使用 WebSocket 来实现聊天等功能，ThinkJS 支持多种 WebSocket 库，如：

`socket.io`, `sockjs` 等, 并对这些库进行了一些简单的包装, 让使用的接口一致。

开启 WebSocket

WebSocket 功能默认是关闭的, 项目如果需要开启, 可以修改配置文件

`src/common/config/websocket.js`:

```
export default {
  on: false, //是否开启 WebSocket
  type: 'socket.io', //使用的 WebSocket 库类型, 默认为 socket.io
  allow_origin: '', //允许的 origin
  adp: undefined, // socket 存储的 adapter, socket.io 下使用
  path: '', //url path for websocket
  messages: {
    // open: 'home/websocket/open',
  }
};
```

需要将配置 `on` 的值修改为 `true`, 并重启 Node.js 服务。

事件到 Action 的映射

ThinkJS 里对 WebSocket 的包装遵循了 `socket.io` 的机制, 服务端和客户端之间通过事件来交互, 这样服务端需要将事件名映射到对应的 Action, 才能响应具体的事件。配置在 `messages` 字段, 具体如下:

```
export default {
  messages: {
    open: 'home/socketio/open', // WebSocket 建立连接时处理的 Action
    close: 'home/socketio/close', // WebSocket 关闭时处理的 Action
    adduser: 'home/socketio/adduser', //adduser 事件处理的 Action
  }
};
```

其中 `open` 和 `close` 事件名固定, 表示建立连接和断开连接的事件, 其他事件均为自定义, 项目里可以根据需要添加。

Action 处理

通过上面配置事件到 Action 的映射后, 就可以在对应的 Action 作相应的处理。如:

```

export default class extends think.controller.base {
  /**
   * WebSocket 建立连接时处理
   * @param {} self []
   * @return {} []
   */
  openAction(self){
    var socket = self.http.socket;
    this.broadcast('new message', {
      username: socket.username,
      message: self.http.data
    });
  }
}

```

emit

Action 里可以通过 `this.emit` 方法给当前 socket 发送事件，如：

```

export default class extends think.controller.base {
  /**
   * WebSocket 建立连接时处理
   * @param {} self []
   * @return {} []
   */
  openAction(self){
    var socket = self.http.socket;
    this.emit('new message', 'connected');
  }
}

```

broadcast

Action 里可以通过 `this.broadcast` 方法给所有的 socket 广播事件，如：

```

export default class extends think.controller.base {
  chatAction(self){
    var socket = self.http.socket;
    //广播给除当前 socket 之外的所有 sockets
    this.broadcast('new message', {msg: 'message', username: 'xxx'});
  }
}

```

注：broadcast 方法默认是给除去当前 socket 的所有 sockets 发送事件，如果想包含当前的 socket，可以设置第三个参数值为 `true`。

```
export default class extends think.controller.base {
  chatAction(self){
    var socket = self.http.socket;
    //广播给所有的 sockets, 包含当前的 socket
    this.broadcast('new message', {msg: 'message', username: 'xxx'}, true);
  }
}
```

socket 对象

Action 里可以通过 `this.http.socket` 拿到当前的 socket 对象。

事件数据

Action 里可以通过 `this.http.data` 拿到发送过来事件的数据。

socket.io

`socket.io` 对 WebSocket 前后端都有封装，使用起来非常方便。

io 对象

在 Action 里可以通过 `this.http.io` 来获取 `io` 对象，该对象为 socket.io 的一个实例。

io 对象包含的方法请见 [http://socket.io/docs/server-api/#server\(\)](http://socket.io/docs/server-api/#server())。

设置 path

设置被 socket.io 处理的路径，默认为 `/socket.io`。如果需要修改，可以修改下面的配置：

```
/* src/common/config/websocket.js */
export default {
  type: 'socket.io',
  adapter: {
    'socket.io': {
      path: '/other_path'
    }
  }
}
```

注：服务端修改了处理的路径后，客户端也要作对应的修改。

设置 adapter

使用多节点来部署 WebSocket 时，多节点之间可以借助 Redis 进行通信，这时可以设置 adapter 来实现。

```
/* src/common/config/websocket.js */

import redis from 'socket.io-redis';
export default {
  type: 'socket.io',
  adapter: {
    'socket.io': {
      adp: function(){
        return redis({ host: 'localhost', port: 6379 });
      }
    }
  }
}
```

如果 redis 需要设置密码，那么需要稍微改动一下配置。

```
``js
/* src/common/config/websocket.js */

import adapter from "socket.io-redis";
import redis from "redis";

const pub = redis.createClient(port, host, { auth_pass: pwd });
const sub = redis.createClient(port, host, { return_buffers: true, auth_pass: pwd });

export default {
  on: true,
  type: "socket.io",
  adapter: {
    "socket.io": {
      adp: function(){
        return adapter({
          pubClient: pub, subClient: sub
        });
      }
    }
  }
}
```

...

具体请见 <http://socket.io/docs/using-multiple-nodes/>。

socket.io client

浏览器端需要引入 socket.io client, 下载地址为: <http://socket.io/download/>。

```
var socket = io('http://localhost:8360');
//发送事件
socket.emit('name', 'data');
//监听事件
socket.on('name', function(data){

})
```

也可以直接引入一个 CDN 地址: <http://s4.qhimg.com/static/535dde855bc726e2/socket.io-1.2.0.js>。

校验用户登录

WebSocket 建立连接时可以拿到 cookie, 所以可以在 `open` 对应的 Action 里校验用户是否登录。如:

```
export default class extends think.controller.base {
  async openAction(){
    let userInfo = await this.session('userInfo');
    if(think.isEmpty(userInfo)){

    }
  }
}
```

聊天代码示例

聊天示例代码请见: <https://github.com/75team/thinkjs2-demos/tree/master/websocket-socket.io>。

SockJS

配置

使用 SockJS 库，需要将配置里的 type 修改为 `sockjs`，如：

```
export default {
  type: 'sockjs'
}
```

sockjs 对象

Action 里可以通过 `this.http.sockjs` 拿到 sockjs 对象，该对象为 SockJS 类的一个实例。

设置 path

设置被 SockJS 处理的路径，默认为 `/sockjs`，可以通过下面的配置修改：

```
export default {
  path: '/websocket'
}
```

SockJS client

浏览器端需要引入 SockJS client，下载地址为：<https://github.com/sockjs/sockjs-client>。

SockJS client 并没有做什么封装，所以需要额外做一层包装，变成事件的方式，以便跟包装后的服务端对应。包装方式参考如下：

```
SockJS.prototype.emit = function(event, data){
  this.send(JSON.stringify({event: event, data: data}));
}
SockJS.prototype.events = {};
SockJS.prototype.on = function(event, callback){
  if(!(event in this.events)){
    this.events[event] = [];
  }
  this.events[event].push(callback);
}
SockJS.prototype.onmessage = function(e) {
  var data = JSON.parse(e.data);
  var callbacks = this.events[data.event] || [];
  callbacks.forEach(function(callback){
    callback && callback(data.data);
  })
};
SockJS.prototype.onopen = function() {
```

```
this.onmessage(JSON.stringify({data: {event: 'open'}}));
};
SockJS.prototype.onclose = function() {
  this.onmessage(JSON.stringify({data: {event: 'close'}}));
};
```

通过上面的包装后就可以通过事件的方式来接收和发送消息了，如：

```
var socket = new SockJS('/sockjs'); //这里的路径必须和配置里相同，如果没有配置则为 /sockj
s
//监听事件
socket.on('add user', function(data){

});
//发送事件
socket.emit('new message', 'xxx');
```

校验用户登录

SockJS 为了安全，在建立连接时不提供相关的 cookie，所以无法通过 cookie 来校验用户是否登录。可以先在页面里输出一个 token，建立连接时将该 token 发送用来校验是否已经登录。具体请见：<https://github.com/sockjs/sockjs-node#authorisation>。

聊天代码示例

聊天示例代码请见：<https://github.com/75team/thinkjs2-demos/tree/master/websocket-sockjs>。

nginx 反向代理

nginx 从 `1.3.13` 版本开始支持反向代理 WebSocket 请求，如果在项目中使用，需要在 nginx 配置文件中添加如下的配置：

```
proxy_set_header Upgrade $http_upgrade;
proxy_set_header Connection "upgrade";
```

注：使用 `thinkjs` 命令创建项目时，会自动创建 nginx 配置文件，并且配置文件已经包含了上面 2 个配置，可以直接使用。

nginx 代理 WebSocket 请求的文档请见 <http://nginx.org/en/docs/http/websocket.html>。

获取当前所有的 WebSocket 连接对象

可以通过 `thinkCache(thinkCache.WEBSOCKET)` 来获取所有的 WebSocket 连接对象，数组格式。

如何实现私聊

ThinkJS 目前还没有私聊的机制，项目里可以通过获取所有的 WebSocket 连接然后找到对应的连接进行。

Template

Template Adapter 用来实现支持多种类型的模版引擎，如：`ejs`，`swig` 等。

支持模版引擎类型

- `base`
- `ejs` ejs 模版引擎
- `jade` jade 模板引擎
- `swig` 一种支持模版继承的模版引擎
- `nunjucks` 一种类似 jinja2 的模版引擎，功能非常强大

模版引擎配置

模版引擎配置如下，可以在 `src/common/config/view.js` 中修改：

```
export default {
  type: 'ejs',
  adapter: {
    ejs: { //额外的配置
  }
}
};
```

使用模版引擎

模版引擎会在视图里自动调用，默认情况不需要手工调用使用。如果在有些场景非要使用的话，可以通过下面的方式加载对应的模版引擎：

```
let EjsTemplate = think.adapter('template', 'ejs');
let instance = new EjsTemplate(...args);
```

扩展模版引擎类型

可以通过下面的命令创建一个名为 `foo` Template 类:

```
thinkjs adapter template/foo
```

执行完成后, 会创建文件 `src/common/adapter/template/foo.js`。扩展缓存类需要实现如下的方法:

```
export default class extends think.adapter.base {
  /**
   * get compiled content
   * @params {String} templateFile 模版文件目录
   * @params {Object} tVar 模版变量
   * @params {Object} config 模版引擎配置
   * @return {Promise} []
   */
  run(templateFile, tVar, config){

  }
}
```

框架里的 Template 实现请见

<https://github.com/75team/thinkjs/tree/master/src/adapter/template>。

使用第三方模版 Adapter

如何使用第三方的模版 Adapter 请参见 [Adapter -> 介绍](#)。

Middleware

Middleware

当处理用户的请求时，需要经过很多处理，如：解析参数，判断是否静态资源访问，路由解析，页面静态化判断，执行操作，查找模版，渲染模版等。项目里根据需要可能还会增加其他的一些处理，如：判断 IP 是否在黑名单中，CSRF 检测等。

ThinkJS 里通过 middleware 来处理这些逻辑，每个逻辑都是一个独立的 middleware。在请求处理中埋很多 hook，每个 hook 串行执行一系列的 middleware，最终完成一个请求的逻辑处理。

hook 列表

框架里包含的 hook 列表如下：

- `request_begin` 请求开始
- `payload_parse` 解析提交上来的数据
- `payload_validate` 验证提交的数据
- `resource` 静态资源请求处理
- `route_parse` 路由解析
- `logic_before` logic 处理之前
- `logic_after` logic 处理之后
- `controller_before` controller 处理之前
- `controller_after` controller 处理之后
- `view_before` 视图处理之前
- `view_template` 视图文件处理
- `view_parse` 视图解析
- `view_filter` 视图内容过滤
- `view_after` 视图处理之后
- `response_end` 请求响应结束

每个 hook 里调用多个 middleware 来完成处理，具体包含的 middleware 如下：

```
export default {
  request_begin: [],
  payload_parse: ['parse_form_payload', 'parse_single_file_payload', 'parse_json_payload', 'parse_querystring_payload'],
  payload_validate: ['validate_payload'],
  resource: ['check_resource', 'output_resource'],
  route_parse: ['rewrite_pathname', 'parse_route'],
  logic_before: [],
  logic_after: [],
  controller_before: [],
  controller_after: [],
  view_before: [],
  view_template: ['locate_template'],
  view_parse: ['parse_template'],
  view_filter: [],
  view_after: [],
```

```
response_end: []
};
```

配置 hook

hook 默认执行的 middleware 往往不能满足项目的需求，可以通过配置修改 hook 对应要执行的 middleware 来完成，hook 的配置文件为 `src/common/config/hook.js`。

```
export default {
  payload_parse: ['parse_xml'], //解析 xml
}
```

上面的配置会覆盖掉默认的配置值。如果在原有配置上增加的话，可以通过下面的方式：

在前面追加

可以通过配置 `prepend` 让 middleware 作为前置追加：

```
export default {
  payload_parse: ['prepend', 'parse_xml'], //在前面追加解析 xml
}
```

在后面追加

可以通过配置 `append` 让 middleware 作为后置追加：

```
export default {
  payload_parse: ['append', 'parse_xml'], //在后面追加解析 xml
}
```

注：建议使用追加的方式配置 middleware，系统的 middleware 名称可能在后续的版本中有所修改。

执行 hook

可以通过 `think.hook` 方法执行一个对应的 hook，如：

```
await think.hook('payload_parse', http, data); //返回的是一个 Promise
```

在含有 `http` 对象的类中可以直接使用 `this.hook` 来执行 hook，如：

```
await this.hook('payload_parse', data);
```

创建 middleware

ThinkJS 支持 2 种方式的 middleware，即：class 方式和 function 方式。可以根据 middleware 复杂度决定使用哪种方式。

class 方式

如果 middleware 需要执行的逻辑比较复杂，需要定义为 class 的方式。可以通过 `thinkjs` 命令来创建 middleware，在项目目录下执行如下的命令：

```
thinkjs middleware xxx
```

执行完成后，会看到对应的文件 `src/common/middleware/xxx.js`。

ES6 方式

```
'use strict';
/**
 * middleware
 */
export default class extends think.middleware.base {
  /**
   * run
   * @return {} []
   */
  run(){

  }
}
```

动态创建类的方式

```
'use strict';
```

```

/**
 * middleware
 */
module.exports = think.middleware({
  /**
   * run
   * @return {} []
   */
  run: function(){

  }
})

```

middleware 里会将 `http` 传递进去，可以通过 `this.http` 属性来获取。逻辑代码放在 `run` 方法执行，如果含有异步操作，需要返回一个 `Promise` 或者使用 `*/yield`。

function 方式

如果 middleware 要处理的逻辑比较简单，可以直接创建为函数的形式。这种 middleware 不建议创建成一个独立的文件，而是放在一起统一处理。

可以建立文件 `src/common/bootstrap/middleware.js`，该文件在服务启动时会自动被加载。可以在这个文件添加多个函数式的 middleware。如：

```

think.middleware('parse_xml', async http => {
  let payload = await http.getPayload();
  if (!payload) {
    return;
  }
  ...
});

```

函数式的 middleware 会将 `http` 对象作为一个参数传递进去，如果 middleware 里含有异步操作，需要返回一个 `Promise` 或者使用 Generator Function。

以下是框架里解析 json payload 的实现：

```

think.middleware('parse_json_payload', http => {
  let types = http.config('post.json_content_type');
  if (types.indexOf(http.type()) === -1) {
    return;
  }
  return http.getPayload().then(payload => {
    try{
      http._post = JSON.parse(payload);
    }
  });
});

```

```
    }catch(e){}
  });
});
```

解析后赋值

有些 middleware 可能会解析相关的数据，然后希望重新赋值到 `http` 对象上，如：解析传递过来的 xml 数据，但后续希望可以通过 `http.get` 方法来获取。

- `http._get` 用来存放 GET 参数值，`http.get(xxx)` 从该对象获取数据
- `http._post` 用来存放 POST 参数值，`http.post(xxx)` 从该对象获取数据
- `http._file` 用来存放上传的文件值，`http.file(xxx)` 从该对象获取数据

```
think.middleware('parse_xml', async http => {
  let payload = await http.getPayload();
  if (!payload) {
    return;
  }
  return parseXML(payload).then(data => {
    http._post = data; //将解析后的数据赋值给 http._post, 后续可以通过 http.post 方法来获取
  });
});
```

关于 `http` 对象更多信息请见 [API -> http](#)。

阻止后续执行

有些 middleware 执行到一定条件时，可能希望阻止后面的逻辑继续执行。如：IP 黑名单判断，如果命中了黑名单，那么直接拒绝当前请求，不再执行后续的逻辑。

ThinkJS 提供了 `think.prevent` 方法用来阻止后续的逻辑执行，该方法是通过返回一个特定类型的 `Reject Promise` 来实现的。

```
think.middleware('parse_xml', async http => {
  let payload = await http.getPayload();
  if (!payload) {
    return;
  }
  var ip = http.ip();
  var blackIPs = ['123.456.789.100', ...];
  if(blackIPs.indexOf(ip) > -1){
    http.end();//直接结束当前请求
    return think.prevent();//阻止后续的代码继续执行
  }
});
```

```
}  
});
```

除了使用 `think.prevent` 方法来阻止后续逻辑继续执行，也可以通过 `think.defer().promise` 返回一个 Pending Promise 来实现。

如果不想直接结束当前请求，而是返回一个错误页面，ThinkJS 提供了 `think.statusAction` 方法来实现，具体使用方式请见 [扩展功能 -> 错误处理](#)。

使用第三方 middleware

在项目里使用第三方 middleware 可以通过 `think.middleware` 方法来实现，相关代码存放在 `src/common/bootstrap/middleware.js` 里。如：

```
var parseXML = require('think-parsexml');  
  
think.middleware('parseXML', parseXML);
```

然后将 `parseXML` 配置到 hook 里即可。

项目里的一些通用 middleware 也推荐发布到 npm 仓库中，middleware 名称推荐使用 `think-xxx`。

第三方 middleware 列表

第三方 middleware 列表请见 [插件 -> middleware](#)。

CSRF

ThinkJS 提供了 CSRF 处理的 middleware，但默认并没有开启。

开启 CSRF

配置 hook 文件 `src/common/config/hook.js`，添加如下的配置：

```
export default {  
  logic_before: ['prepend', 'csrf']  
}
```

配置

CSRF 默认的配置如下，可以在配置文件 `src/common/config/csrf.js` 中修改：

```
export default {
  session_name: '__CSRF__', // Token 值存在 session 的名字
  form_name: '__CSRF__', // CSRF 字段名字，从该字段获取值校验
  errno: 400, // 错误号
  errmsg: 'token error' // 错误信息
};
```

子域名部署

当项目比较复杂时，可能希望将不同的功能部署在不同的域名下，但代码还是在一个项目下。如：域名 `admin.example.com` 部署后台管理的功能，希望映射到 `admin` 模块下。ThinkJS 提供子域名的 middleware 来处理这个需求。

配置

可以修改 `src/common/config/hook.js` 来开启：

```
export default {
  route_parse: ['prepend', 'subdomain']
}
```

然后设定子域名部署的相关配置，该配置可以在 `config/config.js` 里设置：

```
export default {
  subdomain: {
    admin: 'admin', // 表示将 admin.example.com 映射到 admin 模块下
    ...
  }
}
```

假如原来的 pathname 为 `group/detail`，命中了 `admin.example.com` 这个子域名后，pathname 变为 `admin/group/detail`，后续路由解析就会根据新的 pathname 进行。

禁止端口访问

介绍

代码上线后一般会用 nginx 做一层反向代理，这时用户的请求会落到 nginx 上，然后通过 nginx 转发到 Node.js 服务上，这样可以很方便的做负载均衡。

通过 nginx 代理后就不希望用户直接访问到 Node.js 服务了，一种方案是让 Node.js 启动的端口只允许内部访问，外部无法直接访问到。另一种方案是在应用层判断。

ThinkJS 提供禁止端口访问的 Middleware，这样如果不方便直接在机器上配置禁止端口访问的话，就可以使用该 Middleware 来禁止。

middleware 配置

修改 hook 配置文件 `src/common/config/hook.js`，添加如下的配置：

```
export default {
  request_begin: ['prepend', 'force_proxy']
}
```

然后在配置文件 `src/common/config/env/production.js` 里配置：

```
export default {
  proxy_on: true
}
```

这样只在线上环境开启了禁止端口访问的功能，开发环境不受影响。

只监听内网 host

Node.js 启动服务时默认监听的端口是 `0.0.0.0`，这样服务既可以内网访问，也可以外网访问。可以将 host 设置为 `127.0.0.1` 限制为内网访问。

可以通过修改配置为 `src/common/config/config.js` 来完成，如：

```
export default {
  host: '127.0.0.1'
}
```

扩展功能

TypeScript

[TypeScript](#) 是一种由微软开发的自由和开源的编程语言。它是 JavaScript 的一个超集，向这个语言添加了可选的静态类型，在大型项目里非常有用。

ThinkJS 2.1 开始支持了创建 TypeScript 类型的项目，并且开发时会自动编译、自动更新，无需手工编译等复杂的操作。

创建 TypeScript 项目

可以通过指定 `--ts` 参数来创建 TypeScript 项目：

```
thinkjs new thinkjs_demo --ts
```

TypeScript 项目的文件后缀是 `.ts`。如果手工建立一些文件，后缀名也要是 `.ts`，否则调用 `tsc` 编译时会报错。

.d.ts 文件

`.d.ts` 文件为第三方类库的描述文件。创建项目时，会创建文件 `typings/thinkjs/think.d.ts`，该文件为 ThinkJS 的描述文件。项目里的文件可以通过下面的方式引入这个描述文件：

```
/// <reference path="../../../../typings/thinkjs/think.d.ts" />
```

该代码必须放在文件的最前面，同时保持相对路径正确。如果文件有 `use strict` 也要放在这个后面，否则不会识别。

如果项目里还引入了其他第三方库，那么就需要安装对应的描述文件。可以通过 [tsd](#) 工具来安装。

第三方类库的描述文件列表可以从 <https://github.com/DefinitelyTyped/DefinitelyTyped> 找到，基本覆盖了一些比较热门类库。

TypeScript 编译

由于 TypeScript 的编译功能有很多缺陷，所以现在的方案是通过 TypeScript 将 `.ts` 代码编译为 ES6 代码，然后使用 Babel 6 编译为 ES5 代码。

如果发现 TypeScript 有问题，可以给 TypeScript 提 issue，帮助完善，地址为：<https://github.com/Microsoft/TypeScript>。

已有项目升级为 TypeScript 项目

对于已有用 ES6/7 特性开发的项目可以很方便的升级为 TypeScript 项目，具体如下：

修改入口文件

修改入口文件 `www/development.js`，将之前 `compile` 相关的代码改为：

```
//compile src/ to app/  
instance.compile({  
  log: true,  
  type: 'ts' //TypeScript  
});
```

修改 package.json

修改配置文件 `package.json`，删除之前 `Babel` 和 `ThinkJS` 相关模块的依赖，添加如下的依赖：

```
{  
  "dependencies": {  
    "thinkjs": "2.1.x",  
    "babel-runtime": "6.x.x"  
  },  
  "devDependencies": {  
    "typescript": "next",  
    "babel-cli": "6.x.x",  
    "babel-preset-es2015-loose": "6.x.x",  
    "babel-preset-stage-1": "6.x.x",  
    "babel-plugin-transform-runtime": "6.x.x",  
    "babel-core": "6.x.x"  
  }  
}
```

如果 `dependencies` 和 `devDependencies` 里已经有一些项目里要用的模块依赖，需要合并在一起。

修改完成后，执行 `npm install` 安装对应的依赖。

修改 `.thinkjsrc`

修改项目配置文件 `.thinkjsrc`，修改为类似如下的配置：

```
{
  "createAt": "2016-01-13 17:27:19",
  "mode": "module",
  "ts": true
}
```

下载 `think.d.ts` 描述文件

下载文件 <https://github.com/75team/thinkjs/blob/master/template/think.d.ts>，保存为 `typings/thinkjs/think.d.ts`。

修改文件后缀

将 `src/` 目录下所有的 `.js` 文件修改为 `.ts` 文件。

添加 `bin/compile.js` 文件

下载文件 <https://github.com/75team/thinkjs/blob/master/template/bin/compile.ts>，保存为 `bin/compile.js`。

修改 `compile` 命令

将 `package.json` 里原有的 `compile` 命令修改为 `node bin/compile.js`。

项目文件里添加描述文件

在 `src/` 目录下所有文件内容顶部加上如下的代码，要注意相对路径是否正确：

```
/// <reference path="../../../../typings/thinkjs/think.d.ts" />
```

全部修改后，执行 `npm start` 就可以启动服务了。

Logic

当在 Action 里处理用户的请求时，经常要先获取用户提交过来的数据，然后对其校验，如果校验没问题后才能进行后续的操作。当参数校验完成后，有时候还要进行权限判断，等这些都判断无误后才能进行真正的逻辑处理。如果将这些代码都放在一个 Action 里，势必让 Action 的代码非常复杂且冗长。

为了解决这个问题，ThinkJS 在控制器前面增加了一层 `Logic`，Logic 里的 Action 和控制器里的 Action 一一对应，系统在调用控制器里的 Action 之前会自动调用 Logic 里的 Action。

Logic 层

Logic 目录在 `src/[module]/logic`，在通过命令 `thinkjs controller [name]` 创建 Controller 时会自动创建对应的 Logic。

Logic 代码类似如下：

```
'use strict';
/**
 * logic
 * @param {} []
 * @return {} []
 */
export default class extends think.logic.base {
  /**
   * index action logic
   * @return {} []
   */
  indexAction(){

  }
}
```

注：若自己手工创建时，Logic 文件名和 controller 文件名相同

其中，Logic 里的 Action 和 Controller 里的 Action 一一对应。Logic 里也支持 `__before` 和 `__after` 等魔术方法。

请求类型校验配置

对应一个特定的 Action，有时候只需要一种或者二三种请求类型，需要将其他类型的请求给拒绝

掉。可以通过配置特定的请求类型来完成校验。

```
export default class extends think.logic.base {
  indexAction(){
    this.allowMethods = 'post'; //只允许 POST 请求类型
  }
  testAction(){
    this.allowMethods = 'get,post'; //只允许 GET 和 POST 请求类型
  }
}
```

数据校验配置

数据校验的配置如下：

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      doc: "string|default:index",
      version: "string|in:1.2,2.0|default:2.0"
    }
  }
}
```

配置格式

配置格式为 `字段名` -> `配置`，每个字段的配置支持多个校验类型，校验类型之间用 `|` 隔开，校验类型和参数之间用 `:` 隔开，参数之间用 `,` 隔开来支持多个参数。

参数格式

校验类型后面可以接参数，除了支持用逗号隔开的简单参数外，还可以支持 JSON 格式的复杂参数。如：

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      field1: "array|default:[1,2]", //参数为数组
      field2: 'object|default:{"name":"thinkjs"}' //参数为对象
    }
  }
}
```

除了配置为字符串，也可以配置对象的方式，如：

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      field1: {required: true, array: true, default: [1, 2]}, //参数为数组
      field2: {object: true, default: {name: "thinkjs"}} //参数为对象
    }
  }
}
```

支持的数据类型

支持的数据类型有：`boolean`、`string`、`int`、`float`、`array`、`object`，默认为 `string`。

默认值

使用 `default:value` 来定义字段的默认值，如果当前字段值为空，会将默认值覆盖过去，后续获取到的值为该默认值。

获取数据的方式

默认根据当前请求的类型来获取字段对应的值，如果当前请求类型是 GET，那么会通过 `this.get('version')` 来获取 `version` 字段的值。如果请求类型是 POST，那么会通过 `this.post` 来获取字段的值。

但有时候在 POST 类型下，可能会获取上传的文件或者获取 URL 上的参数，这时候就需要指定获取数据的方式了。支持的获取数据方式为 `get`，`post` 和 `file`。

```
export default class extends think.logic.base {
  /**
   * 保存数据, POST 请求
   * @return {} []
   */
  saveAction(){
    let rules = {
      name: "required",
      image: "object|file|required",
      version: "string|get|in:1.2,2.0|default:2.0"
    }
  }
}
```

上面示例指定了字段 `name` 通过 `post` 方法来获取值，字段 `image` 通过 `file` 方式来获取值，字段 `version` 通过 `get` 方式来获取值。

手动设置数据值

如果有时候不能自动获取值的话（如：从 header 里取值），那么可以手动获取值后配置进去。如：

```
export default class extends think.logic.base {
  /**
   * 保存数据, POST 请求
   * @return {} []
   */
  saveAction(){
    let rules = {
      name: {
        required: true,
        value: this.header('x-name') //从 header 中获取值
      }
    }
  }
}
```

手动获取并设置值的时候只能通过对象的方式设置规则。

注：该方式从 `2.2.3` 版本开始支持。

错误信息

上面的配置只是指定了具体的校验规则，并没有指定校验出错后给出的错误信息。错误信息支持国际化，需要在配置文件 `src/common/config/locale/[lang].js` 中定义。如：

```
// src/common/config/locale/en.js
export default {
  validate_required: '{name} can not be blank',
  validate_contains: '{name} need contains {args}',
}
```

其中 key 为 `validate_` + 校验类型名称，值里面支持 `{name}` 和 `{args}` 2个参数，分别代表字段名称和传递的参数。

如果想定义个特定字段某个错误类型的具体信息，可以通过在后面加上字段名。如：

```
// src/common/config/locale/en.js
export default {
  validate_required: '{name} can not be blank',
  validate_required_email: 'email can not be blank', //指定字段 email 的 required 错误信息
}
```

数据校验方法

配置好校验规则后，可以通过 `this.validate` 方法进行校验。如：

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      doc: "string|default:index",
      version: "string|in:1.2,2.0|default:2.0"
    }
    let flag = this.validate(rules);
    if(!flag){
      return this.fail('validate error', this.errors());
      //如果出错，返回值格式为: {"errno":1000,"errmsg":"validate error","data":{"mobile_number":"mobile_number need is a mobile phone number"}}
    }
  }
}
```

如果你在controller的action中使用了 `this.isGet()` 或者 `this.isPost()` 来判断请求的话，在上面的代码中也需要加入对应的 `this.isGet()` 或者 `this.isPost()`，如：

```
export default class extends think.logic.base {
  indexAction(){
    //和controller中的action保持一致
    if(this.isPost()) {
      let rules = {
        doc: "string|default:index",
        version: "string|in:1.2,2.0|default:2.0"
      }
      let flag = this.validate(rules);
      if(!flag){
        return this.fail('validate error', this.errors());
        //如果出错，返回值格式为: {"errno":1000,"errmsg":"validate error","data":{"mobile_number":"mobile_number need is a mobile phone number"}}
      }
    }
  }
}
```

```
}  
}
```

如果返回值为 `false`，那么可以通过 `this.errors` 方法获取详细的错误信息。拿到错误信息后，可以通过 `this.fail` 方法把错误信息以 JSON 格式输出，也可以通过 `this.display` 方法输出一个页面。

错误信息通过 `errors` 字段赋值到模版里，模版里通过下面的方式显示错误信息（以 ejs 模版为例）：

```
<%for(var field in errors){%>  
  <%=field%>:<%=errors[field]%>  
<%}%>
```

自动校验

一般情况下，都是校验有问题后，输出一个 JSON 信息。如果每次都要在 Logic 的 Action 手动调用 `this.validate` 进行校验，势必比较麻烦。可以通过将校验规则赋值给 `this.rules` 属性进行自动校验。如：

```
export default class extends think.logic.base {  
  indexAction(){  
    this.rules = {  
      doc: "string|default:index",  
      version: "string|in:1.2,2.0|default:2.0"  
    }  
  }  
}
```

将校验规则赋值给 `this.rules` 属性后，会在这个 Action 执行完成后自动校验，如果有错误则直接输出 JSON 格式的错误信息。自动校验是通过魔术方法 `__after` 来完成的。

支持的校验类型

required

必填项。对于布尔值 `false`，数字 `0`，空字符串，空数组，空对象等值用 `required` 校验时都不能通过。

```
export default class extends think.logic.base {  
  indexAction(){
```

```
let rules = {
  name: 'required' //name 的值必填
}
}
```

requiredIf

当另一个项的值为某些值其中一项时，该项必填。如：

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      name: 'requiredIf:email,admin@example.com,admin1@example.com'
    }
  }
}
```

当 `email` 的值为 `admin@example.com`，`admin1@example.com` 等其中一项时，`name` 的值必填。

requiredNotIf

当另一个项的值不在某些值中时，该项必填。如：

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      name: 'requiredNotIf:email,admin@example.com,admin1@example.com'
    }
  }
}
```

当 `email` 的值不为 `admin@example.com`，`admin1@example.com` 等其中一项时，`name` 的值必填。

requiredWith

当其他几项有一项值存在时，该项必填。

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
```

```
    name: 'requiredWith:email,title'
  }
}
```

当 `email`, `title` 等项有一项值存在时, `name` 的值必填。

requiredWithAll

当其他几项值都存在时, 该项必填。

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      name: 'requiredWithAll:email,title'
    }
  }
}
```

当 `email`, `title` 等项值都存在时, `name` 的值必填。

requiredWithout

当其他几项有一项值不存在时, 该项必填。

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      name: 'requiredWithout:email,title'
    }
  }
}
```

当 `email`, `title` 等项其中有一项值不存在时, `name` 的值必填。

requiredWithoutAll

当其他几项值都不存在时, 该项必填。

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
```

```
    name: 'requiredWithoutAll:email,title'
  }
}
```

当 `email`, `title` 等项值都不存在时, `name` 的值必填。

contains

值需要包含某个特定的值。

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      name: 'contains:thinkjs' //需要包含字符串 thinkjs。
    }
  }
}
```

equals

和另一项的值相等。

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      name: 'equals:firstname'
    }
  }
}
```

`name` 的值需要和 `firstname` 的值相等。

different

和另一项的值不等。

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      name: 'different:firstname'
    }
  }
}
```

```
}  
}
```

`name` 的值不能和 `firstname` 的值相等。

before

值需要在在一个日期之前，默认为需要在当前日期之前。

```
export default class extends think.logic.base {  
  indexAction(){  
    let rules = {  
      start_time: 'before', //需要在当前日期之前。  
      start_time1: 'before:2015/10/12 10:10:10' //需要在 2015/10/12 10:10:10 之前。  
    }  
  }  
}
```

after

值需要在在一个日期之后，默认为需要在当前日期之后。

```
export default class extends think.logic.base {  
  indexAction(){  
    let rules = {  
      end_time: 'after', //需要在当前日期之后。  
      end_time1: 'after:2015/10/10' //需要在 2015/10/10 之后。  
    }  
  }  
}
```

alpha

值只能是 [a-zA-Z] 组成。

```
export default class extends think.logic.base {  
  indexAction(){  
    let rules = {  
      en_name: 'alpha'  
    }  
  }  
}
```

`en_name` 的值只能是 [a-zA-Z] 组成。

alphaDash

值只能是 [a-zA-Z_] 组成。

alphaNumeric

值只能是 [a-zA-Z0-9] 组成。

alphaNumericDash

值只能是 [a-zA-Z0-9_] 组成。

ascii

值只能是 ascii 字符组成。

base64

值必须是 base64 编码。

byteLength

字节长度需要在 一个区间内。

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      name: 'byteLength:10' //字节长度不能小于 10
      name1: 'byteLength:10,100' //字节长度需要在 10 - 100 之间
    }
  }
}
```

creditcard

需要是信用卡数字。

currency

需要是货币。

date

需要是个日期。

decimal

需要是个小数。

divisibleBy

需要被一个数整除。

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      count: 'divisibleBy:3' //可以被 3 整除
    }
  }
}
```

email

需要是个 email 格式。

fqdn

需要是个合格的域名。

float

需要是个浮点数。

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      money: 'float' //需要是个浮点数
      money1: 'float:3.2' //需要是个浮点数，且最小值为 3.2
      money2: 'float:3.2,10.5' //需要是个浮点数，且最小值为 3.2，最大值为 10.5
    }
  }
}
```

```
}  
}  
}
```

fullWidth

包含宽字节字符。

halfWidth

包含半字节字符。

hexColor

需要是个十六进制颜色值。

hex

需要是十六进制。

ip

需要是 ip 格式。

ip4

需要是 ip4 格式。

ip6

需要是 ip6 格式。

isbn

需要是图书编码。

isin

需要是证券识别编码。

iso8601

需要是 iso8601 日期格式。

in

在某些值中。

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      version: 'in:1.2,2.0' //需要是 1.2, 2.0 其中一个
    }
  }
}
```

noin

不能在某些值中。

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      version: 'noin:1.2,2.0' //不能是 1.2, 2.0 其中一个
    }
  }
}
```

int

需要是 int 型。

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      value: 'int' //需要是 int 型
      value1: 'int:1' //不能小于1
      value2: 'int:10,100' //需要在 10 - 100 之间
    }
  }
}
```

min

不能小于某个值。

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      value: 'min:10' //不能小于10
    }
  }
}
```

max

不能大于某个值。

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      value: 'max:10' //不能大于10
    }
  }
}
```

length

长度需要在某个范围。

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      name: 'length:10' //长度不能小于10
      name1: 'length:10,100' //长度需要在 10 - 100 之间
    }
  }
}
```

minLength

长度不能小于最小长度。

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      name: 'minLength:10' //长度不能小于10
    }
  }
}
```

maxLength

长度不能大于最大长度。

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      name: 'maxLength:10' //长度不能大于10
    }
  }
}
```

lowercase

需要都是小写字母。

uppercase

需要都是大写字母。

mobile

需要手机号。

```
export default class extends think.logic.base {
  indexAction(){
    let rules = {
      mobile: 'mobile:zh-CN' //必须为中国的手机号
    }
  }
}
```

mongoid

是 MongoDB 的 ObjectID。

multibyte

包含多字节字符。

url

是个 url。

order

数据库查询 order，如：name DESC。

field

数据库查询的字段，如：name,title。

image

上传的文件是否是个图片。

startWith

以某些字符打头。

endWith

以某些字符结束。

string

值为字符串。

array

值为数组。

boolean

值为布尔类型。对于字符串 `yes`, `on`, `1`, `true` 会自动转为布尔 `true`。

object

值为对象。

regexp

正则，如：

```
export default class extends think.logic.base {
  indexAction(){
    this.rules = {
      number: {
        required: true,
        regexp: /^d{6}$/
      }
    }
  }
}
```

扩展校验类型

如果默认支持的校验类型不能满足需求，可以通过 `think.validate` 方法对校验类型进行扩展。如：

```
// src/common/bootstrap/validate.js
think.validate('validate_name', (value, ...args) => {
  //需要返回 true 或者 false
  //true 表示校验成功，false 表示校验失败
})
```

上面注册了一个名为 `validate_name` 的校验类型，这样在 Logic 里就可以直接使用该校验类型了。

参数解析

如果要解析后面的 `args`，如：该字段值跟其他字段值进行比较，这时拿到的参数是其他字段名称，但比较的时候肯定需要拿到这个字段值，所以需要将字段名称解析为对应的字段值。

可以通过注册一个解析参数函数来完成。如：上面的校验类型名称为 `validate_name`，那么对应的解析参数的名称必须为 `_validate_name`，即： `_` + `校验类型`。

```
think.validate('_validate_name', (args, data) => {
  let arg0 = args[0];
  args[0] = data[arg0].value; //将第一个参数字段名称解析为对应的参数值
  return args;
})
```

Service

有时候项目里需要调用一些第三方的服务，如：调用 Github 相关接口。如果直接在 controller 里直接调用这些接口，一方面导致 controller 代码比较复杂，另一方面也不能更多进行代码复用。

对于这些情况，可以包装成 service 供 controller 里调用。

创建 service

可以通过命令 `thinkjs service [name]` 来创建命令，具体使用请见 [扩展功能 -> ThinkJS 命令 -> 添加 service](#)。

默认生成的 service 是一个 class，但有些 service 直接提供一些静态方法即可，这时候可以把 class 改为对象即可。

加载 service

可以通过 `think.service` 加载一个 service，如：

```
export default class extends think.controller.base {
  indexAction(){
    let GithubService = think.service('github');
    let instance = new GithubService();
  }
}
```

如果想跨模块加载 service，可以通过下面的方式：

```
export default class extends think.controller.base {
  indexAction(){
    let GithubService = think.service('github', 'admin'); //加载 admin 模块下的 github service
  }
}
```

```
    let instance = new GithubService();
  }
}
```

注：如果项目不是特别复杂，建议把 service 放在 `common` 模块下，就都可以方便的加载了。

Cookie

cookie 配置

cookie 默认配置如下：

```
export default {
  domain: '',
  path: '/',
  httponly: false, //是否 http only
  secure: false,
  timeout: 0 //有效时间, 0 为浏览器进程, 单位为秒
};
```

默认 cookie 是随着浏览器进程关闭而失效，可以在配置文件 `src/common/config/cookie.js` 中进行修改。如：

```
export default {
  timeout: 7 * 24 * 3600 //将 cookie 有效时间设置为 7 天
};
```

获取 cookie

controller 或者 logic 中，可以通过 `this.cookie` 方法来获取。如：

```
export default class extends think.controller.base {
  indexAction(){
    let cookie = this.cookie('theme'); //获取名为 theme 的 cookie
  }
}
```

http 对象里也提供了 `cookie` 方法来获取 cookie。如：

```
let cookie = http.cookie('theme');
```

设置 cookie

controller 或者 logic 中，可以通过 `this.cookie` 方法来设置。如：

```
export default class extends think.controller.base {
  indexAction(){
    this.cookie('theme', 'default'); //将 cookie theme 值设置为 default
  }
}
```

http 对象里也提供了 `cookie` 方法来设置 cookie。如：

```
http.cookie('theme', 'default');
```

如果设置 cookie 时想修改一些参数，可以通过第三个参数来控制，如：

```
export default class extends think.controller.base {
  indexAction(){
    this.cookie('theme', 'default', {
      timeout: 7 * 24 * 3600 //设置 cookie 有效期为 7 天
    }); //将 cookie theme 值设置为 default
  }
}
```

删除 cookie

controller 或者 logic 中，可以通过 `this.cookie` 方法来删除。如：

```
export default class extends think.controller.base {
  indexAction(){
    this.cookie('theme', null); //删除名为 theme 的 cookie
  }
}
```

http 对象里也提供了 `cookie` 方法来删除 cookie。如：

```
http.cookie('theme', null);
```

REST API

项目中，经常要提供一个 API 供第三方使用，一个通用的 API 设计规范就是使用 REST API。REST API 是使用 HTTP 中的请求类型来标识对资源的操作。如：

- GET /ticket #获取ticket列表
- GET /ticket/12 #查看某个具体的ticket
- POST /ticket #新建一个ticket
- PUT /ticket/12 #更新ticket 12
- DELETE /ticket/12 #删除ticket 12

ThinkJS 中提供了很便捷的方式来创建 REST API，创建后无需额外的代码即可响应 REST API 的处理，同时也可以通过定制响应额外的需求。

创建 REST API

通过 `thinkjs controller [name] --rest` 即可创建一个 REST API。如：

```
thinkjs controller home/ticket --rest
```

上面的命令表示在 `home` 模块下创建了一个 `ticket` 的 Rest Controller，该 Controller 用来处理资源 `ticket` 的请求。

处理 REST API 请求

Rest Controller 创建完成后，无需写任何的代码，即可完成对 REST API 的处理。资源名称和数据表名称是一一对应的，如：资源名为 `ticket`，那么对应的数据表为 `数据表前缀` + `ticket`。

请求类型

REST API 默认是从 HTTP METHOD 里获取当前的请求类型的，如：当前请求类型是 `DELETE`，表示对资源进行删除操作。

如果有些客户端不支持发送 `DELETE` 请求类型，那么可以通过属性 `_method` 指定一个参数用来接收请求类型。如：

```
export default class extends think.controller.rest {
  init(http){
    super.init(http);
    this._method = '_method'; //指定请求类型从 GET 参数 _method 里获取
  }
}
```

字段过滤

默认情况下，获取资源信息时，会将资源的所有字段都返回。有时候需要隐藏部分字段，可以通过在 `__before` 魔术方法里完成此类操作。

```
export default class extends think.controller.rest {
  __before(){
    this.modelInstance.fieldReverse('password,score'); //隐藏 password 和 score 字段
  }
}
```

权限管理

有些 REST API 需要进行权限验证，验证完成后才能获取对应的信息，可以通过在 `__before` 魔术方法里进行验证。

```
export default class extends think.controller.rest {
  async __before(){
    let auth = await this.checkAuth();
    if(!auth){
      return this.fail('no permissions'); //没权限时直接返回
    }
  }
}
```

更多定制

更多定制方式请参见 [API -> controller.rest](#)。

Babel

ThinkJS 2.1 中，将依赖的 Babel 版本从 5 升级到 6。由于 Babel 6 是个彻底重构的版本，完全插件化了，所以很多模块在不同的插件都会有依赖，这样会导致一些问题，如：

- 安装后的目录很大，并且首次运行很慢
- 目录层级过深，windows 可能会报错

推荐的解决方案为将 npm 升级到 3，可以通过下面的命令升级：

```
npm install -g npm@3
```

修改编译参数

Babel 6 默认的编译参数为：

```
{
  presets: ['es2015-loose', 'stage-1'],
  plugins: ['transform-runtime']
}
```

如果编译参数不能满足你的需求的话，可以在入口文件 `www/development.js` 里进行修改：

```
instance.compile({
  retainLines: true,
  log: true,
  presets: [], //追加的 presets 列表
  plugins: [] //追加的 plugins 列表
});
```

后续上线前编译执行 `npm run compile` 实际上是调用 `package.json` 里对应的 `compile` 命令，所以若果有 `presets` 或者 `plugins` 修改的话，`compile` 命令也要对应改下。

thinkjs 命令

以全局模式安装 thinkjs 模块后，系统下就会有 thinkjs 命令，在终端执行 `thinkjs -h` 可以看到详细介绍。

```
Usage: thinkjs [command] <options ...>
```

```
Commands:
```

```
new <projectPath>          create project
module <moduleName>       add module
controller <controllerName> add controller
service <serviceName>     add service
model <modelName>         add model
middleware <middlewareName> add middleware
adapter <adapterName>     add adapter
plugin <pluginPath>       create ThinkJS plugin
```

Options:

```
-h, --help          output usage information
-v, --version       output the version number
-V                 output the version number
-t, --ts           use TypeScript for project, used in `new` command
-T, --test        add test dirs when create project, used in `new` command
-r, --rest        create rest controller, used in `controller` command
-M, --mongo       create mongo model, used in `model` command
-R, --relation    create relation model, used in `model` command
-m, --mode <mode> project mode type(normal, module), default is module, used in `
```

创建项目

创建项目可以通过 `thinkjs new <projectPath>` 来执行，如：

```
thinkjs new thinkjs_demo
```

创建 ES6/7 项目

如果想使用 ES6/7 特性开发项目，那么创建项目时需要加上 `--es` 参数，这样生成文件的代码都是 ES6/7 语法的。如：

```
thinkjs new thinkjs_demo --es
```

创建 TypeScript 项目

如果想使用 TypeScript 来开发项目，那么创建项目时需要加上 `--ts` 参数，这样生成文件的代码都是 TypeScript 语法的。如：

```
thinkjs new thinkjs_demo --ts
```

注：TypeScript 项目文件后缀都是 `.ts`。

设置项目模式

默认创建的项目是按模块来划分的。如果项目比较小，不想按模块来划分的话，可以创建项目时指定 `--mode` 参数。如：

```
thinkjs new thinkjs_demo --mode=normal
```

支持的模式列表如下：

- `normal` 普通项目，模块在功能下划分。
- `module` 按模块划分，大型项目或者想严格按模块划分的项目。

注：创建项目后，会在项目下创建一个名为 `.thinkjsrc` 的隐藏文件，里面标识了当前项目的一些配置，该配置会影响后续创建文件，所以需要将该文件需要纳入到版本库中。

添加模块

创建项目时会自动创建模块 `common` 和 `home`，如果还需要创建其他的模块，可以在项目目录下通过 `thinkjs module [name]` 命令来创建。如：

```
thinkjs module admin
```

执行完成后，会创建目录 `src/admin`，以及在该目录下创建对应的文件。

添加 middleware

可以在项目目录下通过 `thinkjs middleware [name]` 命令来添加 middleware。如：

```
thinkjs middleware test;
```

执行完成后，会创建 `src/common/middleware/test.js` 文件。

添加 model

可以在项目目录下通过 `thinkjs model [name]` 命令来添加 model。如：

```
thinkjs model user;
```

执行完成后，会创建 `src/common/model/user.js` 文件。

默认会在 `common` 模块下创建，如果想在其他模块下创建，可以通过指定模块创建。如：

```
thinkjs model home/user;
```

指定模块为 `home` 后，会创建 `src/home/model/user.js` 文件。

添加 Mongo Model

默认添加的 Model 是关系数据库的模型，如果想创建 Mongo Model，可以通过指定 `--mongo` 参数来添加。如：

```
thinkjs model home/user --mongo
```

添加 Relation Model

添加关联模型可以通过指定 `--relation` 参数。如：

```
thinkjs model home/user --relation
```

添加 controller

可以在项目目录下通过 `thinkjs controller [name]` 命令来添加 controller。如：

```
thinkjs controller user;
```

执行完成后，会创建 `src/common/controller/user.js` 文件，同时会创建 `src/common/logic/user.js` 文件。

默认会在 `common` 模块下创建，如果想在其他模块下创建，可以通过指定模块创建。如：

```
thinkjs controller home/user;
```

指定模块为 `home` 后，会创建 `src/home/controller/user.js` 文件。

添加 Rest Controller

如果想提供 Rest API，可以带上 `--rest` 参数来创建。如：

```
thinkjs controller home/user --rest;
```

添加 service

可以在项目目录下通过 `thinkjs service [name]` 命令来添加 service。如：

```
thinkjs service github; #创建调用 github 接口的 service
```

执行完成后，会创建 `src/common/service/github.js` 文件。

默认会在 `common` 模块下创建，如果想在其他模块下创建，可以通过指定模块创建。如：

```
thinkjs service home/github;
```

指定模块为 `home` 后，会创建 `src/home/service/github.js` 文件。

添加 adapter

可以通过 `thinkjs adapter [type]/[name]` 来创建 adapter。如：

```
thinkjs adapter template/dot
```

执行后会创建文件 `src/common/adapter/template/dot.js`，表示创建一个名为 `dot` 的模版类型 adapter。

创建 plugin

ThinkJS 支持 `middleware` 和 `adapter` 2 种插件，可以通过 `thinkjs plugin <pluginName>` 来初始化一个插件，然后进行开发。

```
thinkjs plugin think-template-dot
```

插件名称建议使用 `think-` 打头，这样发布到 npm 仓库后，方便其他用户搜索。

静态资源访问

项目开发时，一般都需要在模版里引用静态资源。

使用 `thinkjs` 命令创建项目时，会自动创建 `www/static` 目录，该目录下专门用来存放 JS、CSS、图片等静态资源。

访问静态资源

静态资源放在 `www/static` 目录后，模版里就可以通过下面的方式引入静态资源。

模版里引用 JS 文件

```
<script src="/static/js/foo.js"></script>
```

模版里引用 CSS 文件

```
<link href="/static/css/foo.css" rel="stylesheet" />
```

模版里引用图片文件

```

```

静态资源访问配置

对于一个请求是否是静态资源请求，是通过正则来判断的。默认配置如下：

```
export default {
  resource_on: true, //是否开启静态资源解析功能
  resource_reg: /^(static\/|[\^\/]+\.(?!js|html)\w+$)/, //判断为静态资源请求的正则
}
```

项目里可以根据需要在配置文件里 `src/common/config/config.js` 进行修改。

线上关闭静态资源访问

项目上线后，一般会使用 nginx 等 WEB 服务器做一层代理，这时候就可以将静态资源的请求直接让 nginx 来处理，项目里就可以关闭对静态资源请求的处理来提高性能。

可以在配置文件 `src/common/config/env/production.js` 里修改配置来关闭，如：

```
export default {
  resource_on: false
}
```

错误处理

系统在处理用户请求时，会遇到各种各样的错误情况。如：系统内部错误，url 不存在，没有权限，服务不可用等，这些情况下需要给用户显示对应的错误页面。

错误页面

通过 `thinkjs` 命令创建项目时，会自动添加错误处理的逻辑文件以及相应的错误页面。

错误逻辑文件路径为 `src/common/controller/error.js`，该文件内容大致如下：

```
'use strict';
/**
 * error controller
 */
export default class extends think.controller.base {
  /**
   * display error page
   * @param {Number} status []
   * @return {Promise}      []
   */
  displayErrorPage(status){
    let module = 'common';
    if(think.mode !== think.mode_module){
      module = this.config('default_module');
    }
    let file = `${module}/error/${status}.html`;
    let options = this.config('tpl');
    options = think.extend({}, options, {type: 'ejs'});
```

```
    return this.display(file, options);
}
/**
 * Bad Request
 * @return {Promise} []
 */
_400Action(){
    return this.displayErrorPage(400);
}
/**
 * Forbidden
 * @return {Promise} []
 */
_403Action(){
    return this.displayErrorPage(403);
}
/**
 * Not Found
 * @return {Promise} []
 */
_404Action(){
    return this.displayErrorPage(404);
}
/**
 * Internal Server Error
 * @return {Promise} []
 */
_500Action(){
    return this.displayErrorPage(500);
}
/**
 * Service Unavailable
 * @return {Promise} []
 */
_503Action(){
    return this.displayErrorPage(503);
}
}
```

对应的模版文件路径为 `view/common/error_{Number}.html`。

错误类型

系统默认支持的错误类型有 `400`，`403`，`404`，`500` 和 `503`。

400

错误的请求，如：恶意构造一些非法的数据访问、访问的 url 不合法等。

403

当前访问没有权限。

404

访问的 url 不存在。

500

系统内部出现错误，导致当前请求不可用。

503

服务不可用，需要等到恢复后才能访问。

扩展错误类型

项目里可以根据需要扩展错误类型，假如添加一个项目特有的错误 `600`，那么可以通过下面步骤进行：

1、添加 `_600Action`

在 `src/common/controller/error.js` 文件中，合适的位置添加如下的代码：

```
_600Action(){  
  return this.displayErrorPage(600);  
}
```

2、添加错误页面

添加文件 `view/common/error_600.html`，并在文件里添加显示的错误内容。

3、显示错误页面

添加完错误后，需要在对应地方调用显示错误才能让用户看到，可以通过 `think.statusAction` 方法实现。如：

```
export default class extends think.controller.base {  
  indexAction(){  
    if(someError){  
      return think.statusAction(600, this.http); //显示 600 错误，需要将 http 对象传递  
      进去
```

```
}  
}  
}
```

修改错误页面样式

修改错误页面样式，只需要修改对应的模版文件即可，如：修改 `404` 错误则修改模版文件 `view/common/error_404.html`。

错误信息

EPERM

Operation Not Permitted

尝试执行某些需要特殊权限的操作。

ENOENT

No Such File Or Directory

通常由文件系统操作引起，比如路径中某个组件指定的路径或文件并不存在。

EACCES

Permission Denied

拒绝访问。

EEXIST

File Exists

要求目标不存在的操作遇到了目标存在的情况。

ENOTDIR

Not a Directory

给定的路径存在，但不是想要的文件夹。通常由 `fs.readdir` 引起。

EISDIR

Is a Directory

操作的目标是文件，但给定的却是文件夹。

EMFILE

Too Many Open Files In System

系统打开的文件数量已经达到上限，至少关闭一个才能打开请求的文件。

通常不允许过多文件同时打开的系统（如OS X）中出现，要提高限制，可以在运行Node.js进程的同一个sh中运行 `ulimit -n 2048`。

EPIPE

Broken Pipe

对管道、套接字或FIFO只有写而没有读。通常在 `net` 或 `http` 层出现，意味着正向其中写入数据的远程服务已关闭。

EADDRINUSE

Address Already In Use

尝试把服务器绑定到一个本地地址，但该地址已经被占用。

ECONNRESET

Connection Reset By Peer

连接被对端强制关闭。通常在远程套接字通信中由于对端超时或重启而丢失连接时导致。常见于 `http` 和 `net` 模块。

ECONNREFUSED

Connection Refused

目标机器频繁拒绝导致无法建立连接。通常在尝试连接外部非活动主机时发生。

ENOTEMPTY

Directory Not Empty

操作目标是空文件夹，但当前文件夹非空，常见于调用 `fs.unlink`。

ETIMEDOUT

Operation Timed Out

目标超过既定时间未响应造成连接或请求失败。通常在 `http` 或 `net` 中出现，一般表明连接的套接字没有适当 `.end()`。

国际化

获取语言

可以通过 `http.lang` 方法从 cookie 或者 header 里获取当前用户的语言。如：

```
let lang = http.lang();
```

如果要支持从 cookie 里获取用户选择的语言，那么需要设置语言在 cookie 里的名称。可以在配置文件 `src/common/config/locale.js` 里修改，如：

```
export default {  
  cookie_name: 'think_locale', //存放语言的 cookie 名称  
  default: 'en' //默认语言  
};
```

在 Controller 里可以直接通过 `this.lang` 方法获取对应的语言。

设置语言

在 Controller 里通过 `this.lang` 方法获取到语言后，有时候需要对语言名格式化下，如：将 `en_US` 改为 `en`。格式化后需要将语言明写回去，以便于后续使用，这时候还可以借助 `this.lang` 来设置。如：

```
export default class extends think.controller.base {
  testAction(){
    let lang = this.lang();
    if(lang === 'en_US'){
      this.lang('en');
    }
  }
}
```

如果是在 middleware 中操作，那么可以借助 http 对象上的 lang 方法进行。

从 URL 中解析语言

有些情况下，语言是要从 URL 中解析。比如：当前页面的地址是

`https://www.thinkjs.org/zh-cn/doc/2.0/i18n.html`，就包含了语言 `zh-cn`。

这种情况下需要在项目里通过 middleware 来解析其中的语言，如：

```
think.middleware('get_lang', http => {
  let supportLangs = think.config('locale.support');
  let lang = http.pathname.split('/')[0]; //从 URL 中获取语言

  if(supportLangs.indexOf(lang) > -1){
    http.pathname = http.pathname.substr(lang.length + 1);
  }else{
    lang = http.lang(); //从 cookie 或者 header 中获取语言
    if(supportLangs.indexOf(lang) === -1){
      lang = http.config('locale.default'); //默认支持的语言
    }
  }
  http.lang(lang, true); //设置语言，并设置模版路径中添加语言目录
});
```

从 URL 中解析到语言后，通过 `http.lang` 方法设置语言，后续在 Controller 里可以直接通过 `http.lang` 来获取语言了。

定义 middleware `get_lang` 后，添加到对应的 hook 里。如：

```
export default {
  route_parse: ['prepend', 'get_lang'], //将 get_lang 前置添加到 route_parse hook 里
}
```

语言变量配置

支持国际化的项目需要配置变量在不同语言下的值，配置文件在

`src/common/config/locale/[lang].js`，配置格式如下：

```
// src/common/config/locale/zh-cn.js
export default {
  'title-home': 'ThinkJS官网 - A Node.js MVC Framework Support All Of ES6/7 Features',
  'title-changelog': '更新日志 - ThinkJS官网',
}
```

```
// src/common/config/locale/en.js
export default {
  'title-home': 'ThinkJS - A Node.js MVC Framework Support All Of ES6/7 Features',
  'title-changelog': 'Changelog - ThinkJS'
}
```

当然值里也支持变量，格式为 `util.format`，如：

```
export default {
  'test': 'test %s',
}
```

获取语言变量

配置语言变量后，可以通过 `http.locale` 方法来获取当前语言对应的变量值。如：

```
let homeTitle = http.locale('title-home');
```

如果在 Controller 中，可以直接通过 `this.locale` 方法来获取，如：

```
export default class extends think.controller.base {
  indexAction(){
    let homeTitle = this.locale('title-home');
  }
}
```

模版里使用语言变量

模版里可以通过 `_` 函数来获取对应的语言值。下面以 `ejs` 模版引擎为例：

```
<%- _('title-home') %>
```

也可以使用变量，如：

```
<%- _('title-home', 'haha', 'test') %>
```

设置模版语言路径

有些项目中，需要根据不同的语言定制不同的模版，这时模版路径里加一层语言目录来处理就比较合适了。如：`view/zh-cn/home/index_index.html`，路径中添加了一层 `zh-cn` 语言目录。

可以通过 `http.lang` 方法设置语言并设置在模版路径里添加一层语言目录。如：

```
http.lang(lang, true); // true 表示在模版路径里添加一层语言目录
```

在 Controller 里可以通过 `this.lang` 方法来设定。如：

```
export default class extends think.controller.base {
  indexAction(){
    let lang = getFromUrl();
    this.lang(lang, true);
    ...
  }
}
```

路径常量

系统提供了很多常量供项目里使用，利用这些常量可以方便的访问对应的文件。

think.ROOT_PATH

项目的根目录。

think.RESOURCE_PATH

静态资源根目录，路径为 `think.ROOT_PATH` + `/www/`。

think.APP_PATH

APP 代码目录，路径为 `think.ROOT_PATH` + `/app/`。

think.THINK_PATH

ThinkJS 框架的根目录。

think.THINK_LIB_PATH

ThinkJS 框架 `lib` 目录。

think.getPath(module, type)

对于 `model`, `controller`, `view` 等目录，由于每个模块下都有这些目录，所以无法给出一个固定的路径值。可以通过 `think.getPath` 来获取模块下的路径。

```
let path1 = think.getPath('common', 'model'); //获取 common 下 model 的目录
let path2 = think.getPath('home', 'controller'); //获取 home 模块下 controller 的目录
```

自定义路径常量

除了通过系统给的属性或者方法来获取路径，还可以在项目里定义额外的路径常量。

入口文件里定义

项目的入口文件为 `src/index.js` 或者 `src/production.js` 等，可以在这些入口文件里定义一些路径常量。如：

```
var thinkjs = require('thinkjs');
var path = require('path');

var rootPath = path.dirname(__dirname);

var instance = new thinkjs({
```

```
APP_PATH: rootPath + '/app',
ROOT_PATH: rootPath,
RESOURCE_PATH: __dirname,
UPLOAD_PATH: __dirname + '/upload', // 定义文件上传的目录
env: 'development'
});

instance.run();
```

启动文件里定义

定义在 `src/common/bootstrap` 里的文件在项目启动时会自动加载，所以也可以在这些文件里定义路径常量。如：

```
// src/common/bootstrap/common.js
think.UPLOAD_PATH = think.RESOURCE_PATH + '/upload'; //定义文件上传的目录
```

定时任务

项目在线上运行时，经常要定时去执行某个功能，这时候就需要使用定时任务来处理了。ThinkJS 支持命令行方式调用，结合系统的 `crontab` 功能可以很好的支持定时任务。

命令行执行

ThinkJS 除了支持通过 URL 访问来执行外，还可以通过命令行的方式调用执行。使用方式如下：

```
node www/production.js home/index/index
```

上面的命令表示执行 `home` 模块下 `index` Controller 里的 `indexAction`。

携带参数

如果需要加参数，只要在后面加上对应的参数即可：

```
node www/production.js home/index/index?name=thinkjs
```

Action 里就可以通过 `this.get` 方法来获取参数 `name` 了。

修改请求方法

命令行执行默认的请求类型是 GET，如果想改为其他的类型，可以用下面的方法：

```
node www/production.js url=home/index/index&method=post
```

这样就把请求类型改为了 post。但这种方式下，参数 url 的值里就不能包含 & 字符了（可以通过 / 的方式指定参数，如 `node www/production.js url=home/index/index/foo/bar&method=post`）。

除了修改请求类型，还可以修改下面的参数：

- `host` 修改请求的 host 默认为 127.0.0.1
- `ip` 修改请求的 ip 默认为 127.0.0.1

修改 header

有时候如果想修改更多的 headers，可以传一个完整的 json 数据，如：

```
node www/production.js {"url":"/index/index","ip":"127.0.0.1","method":"POST","headers":{"xxx":"yyy"}}
```

禁止 URL 访问

默认情况下，命令行执行的 Action 通过 URL 也可以访问到。如果禁止 URL 访问到该 Action，可以通过 `this.isCli` 来判断。如：

```
export default class extends think.controller.base {
  indexAction(){
    //禁止 URL 访问该 Action
    if(!this.isCli()){
      this.fail('only invoked in cli mode');
    }
    ...
  }
}
```

执行脚本

可以创建一个简单的执行脚本来调用命令行执行，如：

```
cd project_path;
node www/production.js home/index/index;
```

在项目目录下创建目录 `crontab`，将上面执行脚本存为一个文件放在该目录下。

定时执行

借助系统里的 `crontab` 可以做到定时执行，通过命令 `crontab -e` 来编辑定时任务，如：

```
0 */1 * * * /bin/sh project_path/crontab/a.sh # 1 小时执行一次
```

使用 `node-crontab` 模块执行定时任务

除了使用 `crontab` 和命令行联合执行定时任务外，也可以使用 `node-crontab` 模块执行定时任务。如：

```
import crontab from 'node-crontab';
// 1 小时执行一次
let jobId = crontab.scheduleJob('0 */1 * * *', () => {

});
```

将上面代码文件存放在 `src/common/bootstrap` 目录下，这样可以在服务启动时自动执行。

如果希望在开发环境下能立即看下执行的效果，可以用类似下面的方式：

```
import crontab from 'node-crontab';

let fn = () => {
  //定时任务具体逻辑
  //调用一个 Action
  think.http('/home/image/spider', true); //模拟访问 /home/image/spier
}
// 1 小时执行一次
let jobId = crontab.scheduleJob('0 */1 * * *', fn);
//开发环境下立即执行一次看效果
if(think.env === 'development'){
  fn();
}
```

线上部署

代码编译

开发环境下，代码会自动编译、自动更新，但这种机制时间长了会有一些的内存泄露，所以线上不可使用这种方式。

需要在代码上线前执行 `npm run compile` 命令，将 `src/` 目录编译到 `app/` 目录，然后将 `app/` 目录下的文件上线。

使用 PM2 管理服务

PM2 是一款专业管理 Node.js 服务的模块，非常建议在线上使用。使用 PM2 需要以全局的方式安装，如：`sudo npm install -g pm2`。安装完成后，命令行下会有 `pm2` 命令。

创建项目时，会在项目目录下创建名为 `pm2.json` 的配置文件，内容类似如下：

```
{
  "apps": [{
    "name": "demo",
    "script": "www/production.js",
    "cwd": "/Users/welefen/Develop/git/thinkjs/demo",
    "max_memory_restart": "1G",
    "autorestart": true,
    "node_args": [],
    "args": [],
    "env": {

    }
  }
]}
}
```

将 `cwd` 配置值改为线上真实的项目路径，然后在项目目录下使用下面的命令来启动/重启服务：

```
pm2 startOrReload pm2.json
```

如果进程重启之前想进行一些操作，如：保存一些临时数据，那么可以使用 `GracefulReload`，具体请见：<http://pm2.keymetrics.io/docs/usage/pm2-doc-single-page/#graceful-reload>。

PM2 详细的配置请见 <http://pm2.keymetrics.io/docs/usage/application-declaration/>。

注：如果线上不使用 PM2 来管理 Node.js 服务的话，启动服务需要使用命令 `node www/production.js`。

使用 nginx 做反向代理

创建项目时，会在项目目录创建一个名为 `nginx.conf` 的 nginx 配置文件。配置文件内容类似如下：

```
server {
    listen 80;
    server_name localhost;
    root /Users/welefen/Develop/git/thinkjs/demo/www;
    set $node_port 8360;

    index index.js index.html index.htm;
    if ( -f $request_filename/index.html ){
        rewrite (.*) $1/index.html break;
    }
    if ( !-f $request_filename ){
        rewrite (.*) /index.js;
    }
    location = /index.js {
        proxy_http_version 1.1;
        proxy_set_header Connection "";
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_set_header X-NginX-Proxy true;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
        proxy_pass http://127.0.0.1:$node_port$request_uri;
        proxy_redirect off;
    }
    location = /production.js {
        deny all;
    }

    location = /testing.js {
        deny all;
    }
    location ~ /static/ {
        etag on;
        expires max;
    }
}
```

将 `server_name localhost` 里的 `localhost` 修改为对应的域名，将 `set $node_port 8360` 里的 `8360` 修改和项目里监听的端口一致。

修改完成后，将该配置文件拷贝到 nginx 的配置文件目录中，然后通过 `nginx -s reload` 命令 reload 配置，这样就可以通过域名访问了。

线上建议开启配置 `proxy_on`，具体请见[禁止端口访问](#)。

关闭静态资源处理的配置

为了方便开发，ThinkJS 是支持处理静态资源请求的。但代码部署在线上时，是用 nginx 来处理静态资源请求的，这时可以关闭 ThinkJS 里处理静态资源请求的功能来提高性能。

可以在配置文件 `src/common/config/env/production.js` 中加入如下的配置：

```
export default {
  resource_on: false
}
```

使用 cluster

线上可以开启 cluster 功能达到利用多核 CPU 来提升性能，提高并发处理能力。

可以在配置文件 `src/common/config/env/production.js` 中加入如下的配置：

```
export default {
  cluster_on: true
}
```

设置 `cluster_on` 值为具体的数值可以配置 `cluster` 的个数。

注：如果使用 PM2 来部署，并且开启了 cluster 模式，那么就无需再开启 ThinkJS 里的 cluster。

单元测试

项目中写一些接口时有时候希望进行单元测试，ThinkJS 从 2.1.4 版本开始创建项目时支持创建单元测试相关的目录。

创建单元测试项目

创建项目时带上 `--test` 参数后就会创建测试相关的目录，如：

```
thinkjs new demo --es --test
```

创建完成后，项目下会有 `test/` 目录，该目录有单元测试的示例代码，自己写的单元测试代码也都放在该目录下。

注：如果 ThinkJS 版本低于 `2.1.4`，需要先将全局的 ThinkJS 升级。

测试框架

默认使用的测试框架是 `mocha`，代码覆盖率框架为 `istanbul`。

加载 thinkjs

测试项目里的代码，有些代码依赖了 `think` 这个全局对象，那么这时就要引入 `thinkjs`，可以通过下面的方式引入：

```
var instance = new require('thinkjs');
instance.load();
```

书写单元测试

可以用 `describe` 和 `it` 的方式来书写测试代码，如：

```
var assert = require('assert');
describe('unit test', function(){
  it('test controller', function(){
    var data = getFromFn();
    assert.equal(data, 1); //测试 data 是否等于 1，不等于则会测试失败
  })
})
```

有些接口是异步的，`mocha` 提供了一个参数来完成，如：

```
var assert = require('assert');
describe('unit test', function(){
  it('test controller', function(done){
    getFromFn().then(function(data){
      assert.equal(data, 1); //测试 data 是否等于 1，不等于则会测试失败
      done(); //这里必须执行下 done，告知接口已经拿到数据并校验
    })
  })
})
```

更多的测试用户请参考生成的文件 `test/index.js`，也可以参考 ThinkJS 框架的测试用例 <https://github.com/75team/thinkjs/tree/master/test>。

执行单元测试

单元测试写完后，执行 `npm test` 就可以进行单元测试。如果代码需要编译，可以在另一个标签页面里执行 `npm start`，这样代码改变后就会自动编译。

老项目支持

如果是之前创建的项目，那么通过下面的方式来支持单元测试：

- 拷贝 <https://github.com/75team/thinkjs/blob/master/template/test/index.js> 内容到 `test/index.js` 文件下
- 修改 `package.json` 文件，添加：在 `devDependencies` 里添加：

```
{
  "devDependencies": {
    "mocha": "1.20.1",
    "istanbul": "0.4.0"
  }
}
```

开发插件

ThinkJS 2.0 里支持 2 种类型的插件，一种是 [Middleware](#)，另一种是 [Adapter](#)。

创建插件

可以通过下面的命令创建一个插件，插件命令建议使用 `think-` 打头。

```
thinkjs plugin think-xxx
```

执行后，会自动创建 `think-xxx` 目录，并可以看到类似下面的信息：

```
create : think-xxx/
create : think-xxx/src
create : think-xxx/src/index.js
create : think-xxx/test
create : think-xxx/test/index.js
```

```
create : think-xxx/.eslintrc
create : think-xxx/.npmignore
create : think-xxx/.travis.yml
create : think-xxx/package.json
create : think-xxx/README.md
```

```
enter path:
$ cd think-xxx/
```

```
install dependencies:
$ npm install
```

```
watch compile:
$ npm run watch-compile
```

```
run test:
$ npm run test-cov
```

目录结构

- `src/` 存放源代码，使用 ES6/7 特性开发
- `test/` 单元测试目录
- `.eslintrc` eslint 检查配置文件
- `.npmignore` npm 发布时忽略的文件
- `.travis.yml` travis 持续集成配置文件
- `package.json` npm 配置文件
- `README.md` 说明文件

安装依赖

```
npm install --verbose
```

开发

代码文件为 `src/index.js`，默认生成的文件只是一个基本的类输出，没有继承任何类。

如果是 Middleware，需要继承 `think.middleware.base` 类。如果是 Adapter，需要继承 `think.adapter.base` 类。

开发过程中，可以在命令行下执行 `npm run watch-compile`，这样文件修改后就会立即编译。

单元测试

在 `test/index.js` 文件书写相关的单元测试，测试框架使用的是 mocha，可以通过下面的命令查看单元测试结果。

```
npm run test-cov
```

说明文档

代码开发和单元测试完成后，需要在 `README.md` 里书写详细的说明文档。

发布

可以通过 `npm publish` 发布模块到 npm 仓库里（如果之前没发布过，会提示创建帐号和密码）。

发布完成后，可以联系 ThinkJS-Team，经确认无误后，可以添加到官方的插件列表中，并领取相关的奖励。

推荐模块

网络请求

- superagent
- request

日志

- log4js

日期处理

- moment

编码转化

- iconv-lite

图像处理

- gm

框架

- thinkjs
- express
- koa
- sails

调试

- node-inspector

单元测试

- mocha
- istanbul
- muk

服务管理

- pm2

邮件

- nodemailer

定时任务

- node-crontab

更多功能

如何将 callback 包装成 Promise

Node.js 本身提供的很多接口都是 callback 形式的，并且很多第三方库提供的接口也是 callback 形式的。但 ThinkJS 需要接口是 Promise 形式的，所以需要将 callback 形式的接口包装成 Promise。

ThinkJS 提供了 `think.promisify` 方法可以快速将接口包装为 Promise 方式，具体请见[这里](#)。

任务队列

Node.js 一个很大的优点就是异步 I/O，这样就可以方便的做并行处理，如：并行去请求一些接口，并行去处理一些文件。但操作系统本身对文件句柄是有限制的，如果并行处理的数目不限制，可能会导致报错。

这时候一般都是通过任务队列来处理，ThinkJS 里提供了 `think.parallelLimit` 方法来处理此类需求，具体见[这里](#)。

API

think

`think` 是一个全局对象，该对象里包含了大量有用的属性和方法。这些方法在应用的任何地方都可以直接使用，无需再 require。

属性

think.startTime

服务启动时间，是个 `unix` 时间戳。

think.env

当前项目运行的环境，默认支持下面 3 个值，可以在项目启动时指定：

- `development` 开发环境，会自动更新修改的文件
- `testing` 测试环境
- `production` 线上环境，代码上线时使用

think.dirname

项目的文件夹名称，可以在项目启动时指定，默认值如下：

```
think.dirname = {
  config: 'config', //配置文件目录
  controller: 'controller', //控制器目录
  model: 'model', //模型目录
  adapter: 'adapter', //适配器目录
  logic: 'logic', //逻辑目录
  service: 'service', //服务目录
  view: 'view', //视图目录
  middleware: 'middleware', //中间件目录
  common: 'common', //通用目录
  bootstrap: 'bootstrap', //启动目录
  locale: 'locale' //本土化目录
}
```

think.port

项目运行的端口，可以在项目启动时指定。如果指定，则忽略配置文件里的端口。

think.sep

目录分隔符，等同于 `path.sep`。

think.isMaster

是否是 master 进程。

think.cli

是否是命令行模式在运行项目，默认为 `false`。如果是命令行模式，则该值为传递的参数，可以通过下面的方式启动命令行模式。

```
node www/index.js /home/index/test
```

think.lang

系统当前的语言，从环境变量中读取，在 `Windows` 下可能为空。

think.mode

项目当前的模式，框架支持 2 中项目模式：

- `think.mode_normal` 多模块模式，目录结构只有 `Controller`，`View`，`Logic` 等分模块
- `think.mode_module` 多模块模式，严格按照模块来划分目录结构

think.version

ThinkJS当前的版本

think.module

当前项目下的模块列表。

think.THINK_PATH

ThinkJS代码的路径

think.THINK_LIB_PATH

ThinkJS代码 `lib/` 的具体路径

think.ROOT_PATH

项目的根目录，在 `www/index.js` 中定义

think.APP_PATH

项目的 `app` 目录，在 `www/index.js` 中定义

think.RESOURCE_PATH

项目的静态资源根目录，在 `www/index.js` 中定义

think.RUNTIME_PATH

Runtime 目录，默认为当前项目下的 `runtime/` 目录。

方法

think.Class(methods, clean)

动态的创建一个类，默认继承自 think.base。如果使用 ES6 特性进行开发的话，可以直接使用 ES6 里的 class 来创建类。

```
//继承自 think.base
var Cls1 = think.Class({
  getName: function(){

  }
})
```

不继承 think.base

```
var Cls2 = think.Class({
  getName: function(){

  }
}, true);
```

继承一个类

```
//继承自 Cls2
var Cls3 = think.Class(Cls2, {
  init: function(name){
    this.name = name;
  },
  getName: function(){

  }
})
```

实例化类

```
//获取类的实例，自动调用 init 方法
var instance = new Cls3('thinkjs');
```

think.extend(target, source1, source2, ...)

- `target` {Object} 目录对象

- `source1` {Mixed} 源对象1
- `return` {Object} 目录对象

将 `source1`, `source2` 等对象上的属性或方法复制到 `target` 对象上，类似于 jQuery 里的 `$.extend` 方法。

默认为深度复制，可以将第一个参数传 `false` 进行浅度复制。

```
think.extend({}, {name: 'foo'}, {value: 'bar'});  
// returns  
{name: 'foo', value: 'bar'}
```

think.isBoolean(obj)

- `obj` {Mixed} 要检测的对象
- `return` {Boolean}

检测一个对象是否是布尔值。

```
think.isBoolean(true); //true  
think.isBoolean(false); //true  
think.isBoolean('string'); //false
```

think.isNumber(obj)

- `obj` {Mixed} 要检测的对象
- `return` {Boolean}

检测一个对象是否是数字。

```
think.isNumber(1); //true  
think.isNumber(1.21); //true
```

think.isObject(obj)

- `obj` {Mixed} 要检测的对象
- `return` {Boolean}

检测是否是对象

```
think.isObject({}); //true
think.isObject({name: "welefen"}); //true
think.isObject(new Buffer('welefen')); //false
```

think.isString(obj)

- `obj` {Mixed} 要检测的对象
- `return` {Boolean}

检测是否是字符串

```
think.isString("xxx"); // true
think.isString(new String("xxx")); //true
```

think.isFunction(obj)

- `obj` {Mixed} 要检测的对象
- `return` {Boolean}

检测是否是函数

```
think.isFunction(function(){}); //true
think.isFunction(new Function("")); //true
```

think.isDate(obj)

- `obj` {Mixed} 要检测的对象
- `return` {Boolean}

检测是否是日期对象

```
think.isDate(new Date()); //true
```

think.isRegExp(obj)

- `obj` {Mixed} 要检测的对象
- `return` {Boolean}

检测是否是正则

```
think.isRegExp(/\w+/); //true
think.isRegExp(new RegExp("\w+")); //true
```

think.isError(obj)

- `obj` {Mixed} 要检测的对象
- `return` {Boolean}

检测是否是个错误

```
think.isError(new Error("xxx")); //true
```

think.isEmpty(obj)

- `obj` {Mixed} 要检测的对象
- `return` {Boolean}

检测是否为空

```
// 检测是否为空
think.isEmpty({}); //true
think.isEmpty([]); //true
think.isEmpty(""); //true
think.isEmpty(0); //true
think.isEmpty(null); //true
think.isEmpty(undefined); //true
think.isEmpty(false); //true
```

think.isArray(obj)

- `obj` {Mixed} 要检测的对象
- `return` {Boolean}

检测是否是数组

```
think.isArray([]); //true
think.isArray([1, 2]); //true
think.isArray(new Array(10)); //true
```

think.isIP4(obj)

- `obj` {Mixed} 要检测的对象
- `return` {Boolean}

检测是否是 IP4

```
think.isIP4("10.0.0.1"); //true  
think.isIP4("192.168.1.1"); //true
```

think.isIP6(obj)

- `obj` {Mixed} 要检测的对象
- `return` {Boolean}

检测是否是 IP6

```
think.isIP6("2031:0000:130f:0000:0000:09c0:876a:130b"); //true  
think.isIP6("2031:0000:130f::09c0:876a:130b"); //true
```

think.isIP(obj)

- `obj` {Mixed} 要检测的对象
- `return` {Boolean}

检测是否是 IP

```
think.isIP("10.0.0.1"); //true  
think.isIP("192.168.1.1"); //true  
think.isIP("2031:0000:130f:0000:0000:09c0:876a:130b"); //true ip6
```

think.isFile(file)

- `file` {Mixed} 要检测的文件路径
- `return` {Boolean}

检测是否是文件， 如果不存在或者不是文件则返回 false

```
think.isFile("/home/welefen/a.txt"); //true
think.isFile("/home/welefen/dirname"); //false
```

think.isFileAsync(file)

- `file` {Mixed} 要检测的文件路径
- `return` {Promise}

异步检测是否是文件，返回一个 Promise。该方法在 2.1.5 版本中添加。

think.isDir(dir)

- `dir` {Mixed} 要检测的路径
- `return` {Boolean}

检测是否是目录，如果不存在则返回 false

```
think.isDir("/home/welefen/dirname"); //true
```

think.isDirAsync(dir)

- `dir` {Mixed} 要检测的路径
- `return` {Promise}

异步检测是否是目录，返回 Promise。该方法在 2.1.5 版本中添加。

think.datetime(date, format)

- `date` {Date}
- `format` {String} 日期格式，默认为 YYYY-MM-DD HH:mm:ss
- `return` {String}

返回一个格式化的日期，格式为：YYYY-MM-DD HH:mm:ss，如：

```
let str = think.datetime();
//str is 2016-02-01 10:00:00
//
let str1 = think.datetime(new Date, 'YYYY-MM-DD');
// str1 is 2016-02-01
```

该方法在 `2.1.5` 版本中添加。

think.isBuffer(obj)

- `obj` {Mixed} 要检测的对象
- `return` {Boolean}

检测是否是 Buffer

```
think.isBuffer(new Buffer(20)); //true
```

think.isNumberString(obj)

- `obj` {Mixed} 要检测的对象
- `return` {Boolean}

是否是字符串类型的数字

```
think.isNumberString(1); //true  
think.isNumberString("1"); //true  
think.isNumberString("1.23"); //true
```

think.isPromise(obj)

- `obj` {Mixed} 要检测的对象
- `return` {Boolean}

检测是否是个 Promise

```
think.isPromise(new Promise(function(){})); //true  
think.isPromise(getPromise()); //true
```

think.isHttp(obj)

- `obj` {Mixed} 要检测的对象
- `return` {Boolean}

检测是否是包装的 http 对象

```
think.isHttp(http); // true
```

think.isWritable(path)

- `path` {String} 要写的目录
- `return` {Boolean}

判断文件或者目录是否可写，如果不存在则返回 false

think.isPrevent(obj)

- `obj` {Mixed}
- `return` {Boolean}

判断是否是个阻止类型的 Promise。通过 `think.prevent()` 会生成该 Promise。

think.mkdir(p, mode)

- `p` {String} 要创建的目录
- `mode` {Number} 要创建的目录权限，默认为 `0777`

递归的创建目录，如果目录已经存在，那么修改目录的权限。

```
// 假设 /home/welefen/a/b/ 不存在  
think.mkdir("/home/welefen/a/b");  
think.mkdir("home/welefne/a/b/c/d/e"); // 递归创建子目录
```

think.rmdir(p, reserve)

- `p` {String} 要删除的目录
- `reserve` {Boolean} 是否保留该目录。如果为 true，则只删除子目录
- `return` {Promise}

递归的删除目录，如果目录不存在则直接返回。返回是个 Promise，后续操作要在 `then` 里执行

```
function rmTmp(){  
  think.rmdir('/foo/bar').then(function(){  
    //后续其他操作  
  })  
}
```

如果使用 `Generator Function`，则可以使用 `yield`

```
function * rmTmp(){
  yield think.rmdir('/foo/bar');
  //后续其他操作
}
```

think.chmod(p, mode)

- `p` {String} 要修改的目录
- `mode` {Number} 目录权限，默认为 `0777`

修改目录权限，如果目录不存在则直接返回

```
think.chmod("/home/welefen/a", 0777);
```

think.md5(str)

- `str` {String} 要计算的字符串
- `return` {String} 返回字符串的 md5 值

计算字符串的 md5 值

```
think.md5('thinkjs');
// returns 7821eb623e0b1138a47db6a88c3f56bc
```

think.camelCase(sr)

- `str` {String} 要转换的字符串
- `return` {String}

转换为驼峰方式

```
think.camelCase('a_bbb_ccc');
//returns aBbbCcc
```

think.defer()

- `return {Object}` Deferred对象

创建一个 `Deferred` 对象, `new Promise` 的一种快捷方式。虽然不建议使用 `Deferred` 这种方式, 但有时候不得不使用。如: `setTimeout`, `event`。

```
//使用Deferred的方式
var fn = function(){
  var deferred = think.defer();
  process.nextTick(function(){
    if(xxx){
      deferred.resolve(data);
    }else{
      deferred.reject(err);
    }
  })
  return deferred.promise;
}
```

使用 `Deferred` 方式比直接使用 `new Promise` 的方法代码更加简洁。

```
//直接使用new Promise的方式
var fn = function(){
  return new Promise(function(resolve, reject){
    process.nextTick(function(){
      if(xxx){
        resolve(data);
      }else{
        reject(err);
      }
    })
  })
}
```

注: 异步 `callback` 的操作不要使用 `Deferred` 方式, 可以用 `think.promisify` 方法快速把 `callback` 包装成 `Promise`。

think.promisify(fn, receiver)

- `fn` {Function} 要转化的函数
- `receiver` {Object} this指向

将异步方法快速包装成 `Promise`, 异步方法必须符合最后一个参数为回调函数, 且回调函数的第一个参数为 `err` 的原则。

```
var fs = require('fs');
```

```
//获取文件内容
var getContent = function(filePath){
  //将readFile方法包装成Promise
  var readFilePromise = think.promisify(fs.readFile, fs);
  //读取文件内容
  return readFilePromise(filePath, 'utf8');
}

//获取具体的文件内容
getContent('/foo/bar/file.txt').then(function(content){
  console.log(content);
}).catch(function(err){
  console.error(err.stack);
})
```

think.reject(err)

- `err` {Error} Error对象
- `return` {Promise} reject promise

返回一个 reject promise, 与 `Promise.reject` 不同的是, 该方法会自动打印错误信息。避免需要调用 `catch` 方法手工打印错误信息。

```
//使用Promise.reject
var fn = function(){
  return Promise.reject(new Error('xxx'));
}
//需要手工调用catch方法打印错误信息
fn().catch(function(err){
  console.error(err.stack);
})
```

```
//使用think.reject
var fn = function(){
  return think.reject(new Error('xxx'));
}
//会自动打印格式化后的错误信息
fn();
```

think.co

`co` 模块的别名 <https://github.com/tj/co>

think.lookClass(name, type, module, base)

- `name` {String} 类名
- `type` {String} 类型 (controller | model | logic ...)
- `module` {String} 模块名
- `base` {String} 找不到时找对应的基类

根据类型，名称来查找类。如果找不到会到 common 模块下查找，如果还是找不到，则查找对应类型的基类。

```
//查找 home 模块下 user controller
//如果找不到，会找 common 模块下 user controller
//如果还是找不到，会找 base controller
think.lookClass('user', 'controller', 'home');

//查找 admin 模块下 user controller
think.lookClass('admin/user', 'controller');
```

think.getPath(module, type, prefix)

- `module` {String} 模块名
- `type` {String} 类型，如： controller, model, logic
- `prefix` {String} 前缀

根据当前项目类型获取对应类型的目录。

```
let path = think.getPath('home', 'controller');
```

假如当前项目的根目录是 `/foo/bar`，那么获取到的目录为：

- 项目模式 `think.mode_normal` 下路径为 `/foo/bar/app/controller/home`
- 项目模式 `think.mode_module` 下路径为 `/foo/bar/app/home/controller`

think.require(name, flag)

- `name` {String}
- `flag` {Boolean}

think.safeRequire(file)

- `file` {String} 要加载的文件

安全的加载一个文件，如果文件不存在，则返回 `null`，并打印错误信息。

`think.prevent()`

返回一个特殊的 `reject promise`。该 `Promise` 可以阻止后续的行为且不会报错。

`think.log(msg, type, showTime)`

- `msg` {String | Error} 信息
- `type` {String} 类型
- `showTime` {Number | Boolean} 是否显示时间

打印日志，该方法打印出来的日志会有时间，类型等信息，方便查看和后续处理。

```
think.log('WebSocket Status: closed', 'THINK');  
//writes '[2015-09-23 17:43:00] [THINK] WebSocket Status: closed'
```

打印错误信息

```
think.log(new Error('error'), 'ERROR');  
//writes '[2015-09-23 17:50:17] [Error] Error: error'
```

显示执行时间

```
think.log('/static/module/jquery/1.9.1/jquery.js', 'HTTP', startTime);  
//writes '[2015-09-23 17:52:13] [HTTP] /static/module/jquery/1.9.1/jquery.js 10ms'
```

不显示时间

```
think.log('/static/module/jquery/1.9.1/jquery.js', 'HTTP', null);  
//writes '[HTTP] /static/module/jquery/1.9.1/jquery.js'
```

自定义

```
think.log(function(colors){  
  return colors.yellow('[WARNING]') + ' test';  
});
```

```
//writes '[WARNING] test'
```

其中 `colors` 为 npm 模块 colors, <https://github.com/Marak/colors.js>。

think.config(name, value, data)

- `name` {String} 配置名称
- `value` {Mixed} 配置值
- `data` {Object} 配置对象

读取或者设置配置，可以指定总的配置对象。

```
//获取配置
let value = think.config('name');
//获取 admin 模块下的配置
let value = think.config('name', undefined, 'admin');

// 写入配置
think.config('name', 'value');
```

think.getModuleConfig(module)

- `module` {String} 模块名称
- `return` {Object}

获取模块的所有配置。该配置包含模块的配置，通用模块的配置，框架默认的配置。

```
//获取 admin 模块的所有配置
let configs = think.getModuleConfig('admin');
```

think.hook()

注册、获取和执行 hook，项目中可以根据需要追加或者修改。

获取事件对应的 middleware 列表

```
think.hook('view_template');
//returns
['locate_template']
```

设置 hook

```
//替换原有的 hook
think.hook('view_template', ['locate_template1']);

//将原有的之前追加
think.hook('view_template', ['locate_template1'], 'prepend');

//将原有的之后追加
think.hook('view_template', ['locate_template1'], 'append');
```

删除 hook

```
think.hook('view_template', null);
```

执行 hook

```
let result = think.hook('view_template', http, data);
//result is a promise
```

think.middleware()

注册、创建、获取和执行 middleware。

创建 middleware

```
//解析 XML 示例
var ParseXML = think.middleware({
  run: function(){
    var http = this.http;
    return http.getPayload().then(function(payload){
      var data = xmlParse.parse(payload); //使用一个xml解析, 这里 xmlParse 是示例
      http._post = data; //将解析后的数据赋值给 http._post, 后续可以通过 http.post('xxx'
    ) 获取
  });
});
```

使用 ES6 创建 middleware。

```
let Cls1 = class extends think.middleware.base {
  run(){
    let http = this.http;
  }
}
```

注册 middleware

middleware 可以是简单的 function，也可以是较为复杂的 class。

```
//注册 middleware 为 function
think.middleware('parse_xml', http => {

})
```

```
//注册 middleware 为 class
//会自动调用 run 执行
let Cls = think.middleware({
  run: function(){
    let http = this.http;

  }
});
think.middleware('parse_xml', Cls);
```

获取 middleware

```
let middleware = think.middleware('parse_xml');
```

执行 middleware

```
let result = think.middleware('parse_xml', http);
//result is a promise
```

think.adapter()

创建、注册、获取和执行 adapter。

创建 adapter

```
//创建一个 adapter
var Cls = think.adapter({

});

//创建一个 session adapter, 继承自 session base 类
var Cls = think.adapter('session', 'base', {

})
```

```
//使用 ES6 创建一个 session adapter
let Cls = class extends think.adapter.session {

}
```

注册 adapter

```
//注册一个 xxx 类型的 session adapter
think.adapter('session', 'xxx', Cls);
```

获取 adapter

```
//获取 file 类型的 session adapter
let Cls = think.adapter('session', 'file');
```

执行 adapter

```
let Adapter = think.adapter('session', 'file');
let instance = new Adapter(options);
```

think.gc(instance)

- `instance` {Object} 类的实例

注册实例到 gc 队列中。instance 必须含有属性 `gcType` 和方法 `gc`。

像 cache, session 这些功能一般都是有过期时间，过期后需要进行清除工作。框架提供了一套机制方便清除过期的文件等。

```
let Cls = class extends think.adapter.cache {
  init(options){
    super.init(options);
    this.gcType = 'xFileCache';
    think.gc(this);
  }
  gc(){
    //寻找过期的内容并清除
  }
}
```

think.http(req, res)

- `req` {Object} request 对象
- `res` {Object} response 对象
- `return` {Promise}

根据 `req` 和 `res` 包装成 `http` 对象。`req` 和 `res` 可以自定义。

```
//根据一个 url 生成一个 http 对象，方便命令行下调用
think.http('/index/test').then(http => {

});
```

think.uuid(length)

- `length` {Number} 生成字符串的长度，默认为 32

生成一个随机字符串。

think.parseConfig(...args)

解析配置里的 `adapter` 和 `parser`，如：

```
let config = think.parseConfig({prefix: 'think_'}, {
  type: 'mysql',
  adapter: {
    mysql: {
      prefix: 'test_',
      host: ['10.0.0.1', '10.0.0.2'],
      parser: options => {
        return {
          host: '10.0.0.1'
        }
      }
    }
  }
});
```

```
    }
  }
}
});
// config value is {prefix: 'test_', host: '10.0.0.1'}
```

如果只想解析 `adapter`，而不解析 `parser`，可以通过传递第一个参数为 `true`，如：

```
let config = think.parseConfig(true, {prefix: 'think_'}, {
  type: 'mysql',
  adapter: {
    mysql: {
      prefix: 'test_',
      host: ['10.0.0.1', '10.0.0.2'],
      parser: options => {
        return {
          host: '10.0.0.1'
        }
      }
    }
  }
});
// config value is {prefix: 'test_', ['10.0.0.1', '10.0.0.2'], parser: function(){
...}}
```

think.session(http)

- `http` {Object} http对象

生成 session，并写到 http 对象上。如果已经存在，则直接返回。

think.controller()

创建、执行 controller

创建 controller

```
//创建 controller, 继承 think.controller.base
let Cls = think.controller({
})
//创建 controller, 继承 think.controller.rest
let Cls = think.controller('rest', {
})
```

```
//使用 ES6 创建 controller
let Cls1 = class extends think.controller.base {

}
```

实例化 controller

```
//实例化 home 模块下 user controller
let instance = think.controller('user', http, 'home');
```

think.logic()

创建、执行 logic

创建 logic

```
//创建 logic, 继承 think.logic.base
let Cls = think.logic({

})
```

```
//使用 ES6 创建 logic
let Cls1 = class extends think.logic.base {

}
```

实例化 logic

```
//实例化 home 模块下 user logic
let instance = think.logic('user', http, 'home');
```

think.model()

创建或者获取 model

创建 model

```
//创建一个 model
let model = think.model({
  getList: function(){

  }
});

//ES6 里直接继承 think.model.base 类
let model = class extends think.model.base {
  getList(){

  }
}

//创建一个 model 继承自 mongo model
let model = think.model('mongo', {
  getList: function(){

  }
});
//ES6 里直接继承 think.model.mongo 类
let model = class extends think.model.mongo {
  getList(){

  }
}
```

获取 model 实例

```
let configs = {
  host: '127.0.0.1',
  name: 'user'
}
//获取 home 模块下 user model
let instance = think.model('user', configs, 'home');
```

think.service()

创建或者获取 service

创建 service

```
//创建一个 service 类
```

```
let service = think.service({
})

//ES6 里直接继承 think.service.base 类
let service = class extends think.service.base {
}
```

service 基类继承自 [think.base](#)，所以可以用 think.base 里的方法。

如果 service 不想写成类，那就没必要通过这种方法创建。

获取 service

```
//获取 home 模块下 post service, 并传递参数 {}
//如果获取到的 service 是个类, 则自动实例化
think.service('post', {}, 'home');
```

think.cache(name, value, options)

- `name` {String} 缓存 key
- `value` {Mixed} 缓存值
- `options` {Object} 缓存选项
- `return` {Promise} 操作都是返回 Promise

获取、设置或者删除缓存，value 是 `undefined` 表示读取缓存。value 是 `null` 时删除缓存。

value 为 `Function` 时表示获取缓存，如果获取不到，则调用该函数，然后将返回值设置到缓存中并返回。

```
//获取缓存
think.cache('name').then(data => {});

//指定缓存类型获取, 从 redis 里获取缓存
think.cache('name', undefined, {type: 'redis'});

//如果缓存 userList 不存在, 则查询数据库, 并将值设置到缓存中
think.cache('userList', () => {
  return think.model('user').select();
});

//设置缓存
think.cache('name', 'value');

//删除缓存
```

```
think.cache('name', null);
```

think.locale(key, ...data)

- `key` {String} 要获取的 key
- `data` {Array} 参数

根据语言获取对应的值，当前语言通过 `think.lang` 方法来获取，可以在系统启动时指定。

```
think.locale('CONTROLLER_NOT_FOUND', 'test', '/index/test');  
//returns  
'controller `test` not found. url is `/index/test`.'
```

think.validate()

注册、获取或执行检测。

注册检测方法

```
//注册检测类型为 not_number  
think.validate('not_number', value => {  
  return !(/^\d+$/.test(value));  
})
```

获取检测方法

```
let fn = think.validate('not_number');
```

检测数据

```
let result = think.validate({  
  name: {  
    value: 'name',  
    required: true,  
    not_number: true  
  },  
  pwd: {  
    value: 'xxx',  
    required: true,  
    minLength: 6  
  }  
})
```

```
    }
  });
  //如果 result 是 isEmpty, 表示数据都正常
  if(think.isEmpty(result)){
  }
}
```

think.await(key, callback)

- `key` {String}
- `callback` {Function}

执行等待，避免一个耗时的操作多次被执行。callback 需要返回一个 Promise。

如：用户访问时，要请求一个远程的接口数据。如果不处理，每个用户请求都会触发这个远程接口的访问，导致有很大的资源浪费。可以让这些用户公用一个远程接口的请求。

```
import superagent from 'superagent';

export default class extends think.controller.base {
  * indexAction(){
    let result = yield think.await('get_xxx_data', () => {
      let req = superagent.post('xxxx');
      let fn = think.promisify(req.end, req);
      return fn();
    });
    this.success(result);
  }
}
```

think.npm(pkg)

- `pkg` {String} 模块名

加载模块。如果模块不存在，则自动安装。这样可以做到动态安装模块。

```
//如果mysql模块，则通过npm安装
let mysql = think.npm('mysql');
```

```
//指定版本加载一个模块
let mysql = think.npm('mysql@2.0.0')
```

think.error(err, addon)

- `err` {Error | Promise | String} 错误信息
- `addon` {Error | String} 追加的错误信息

格式化错误信息，将部分系统的错误信息描述完整化。

```
let error = think.error(new Error('xxx'));
```

捕获 promise 的错误信息

```
let promise = Project.reject(new Error('xxx'));  
promise = think.error(promise)
```

自动给 promise 追加 catch，捕获错误信息。

think.statusAction(status, http, log)

- `status` {Number} 状态码
- `http` {Object} 包装的http对象
- `log` {Boolean} 是否打印错误信息

当系统出现异常时（系统错误，页面找不到，没权限等），显示对应的错误页面。

创建项目时，会在 common 模块下生成文件 `src/common/controller/error.js`，专门用来处理错误情况。

默认支持的错误类型有：`400`，`403`，`404`，`500`，`503`。

项目里可以根据需要修改错误页面或者扩展。

```
export default class extends think.controller.base {  
  indexAction(){  
    if(xxxx){  
      let error = new Error('not found');  
      //将错误信息写到 http 对象上，用于模版里显示  
      this.http.error = error;  
      return think.statusAction(404, this.http);  
    }  
  }  
}
```

think.parallelLimit(dataList, callback, options)

- `dataList` {Array} 要处理的数据列表
- `callback` {Function} 处理函数，会将每条数据传递进去，需要返回 Promise
- `options` {Object} 额外选项
- `return` {Promise}

`options` 包含一下选项：

- `limit` {Number} 并发限制数，默认为 10 条
- `ignoreError` {Boolean} 是否忽略错误，默认情况下一个错误后会停止后续执行

并发限制处理方法。如：有 10000 条网络数据需要处理，如果同时处理会会网络 IO 错误，此时可以对并发处理进行限制。该方法在 `2.0.6` 版本中添加。

一个请求下多条数据同时处理场景

```
import superagent from 'superagent';

export default class extends think.controller.base {
  async indexAction(){
    let dataList = [...];
    //result 为每条处理结果集合
    //如果某些条数据处理异常，那么对应的数据为 undefined，处理时需要过滤下
    let result = await think.parallelLimit(dataList, item => {
      let url = item.url;
      let req = superagent.get(url);
      let fn = think.promisify(req.end, req); //将 end 方法包装成 Promise
      return fn();
    }, {
      limit: 20, //一次执行 20 条
      ignoreError: true
    })
  }
}
```

单条数据在多个请求下处理场景

有些数据处理虽在一个情况下只用处理一次，但单次处理比较耗时，如果同时请求很多的话可能会导致报错。这个时候也要进行限制，如果当前同时处理数目较多，后续请求则进行等待。

这个需求可以通过传入一个相同的 key 将任务分组，如：

```
import gm from 'gm';

export default class extends think.controller.base {
  async indexAction(){
```

```
let result = await think.parallelLimit('clip_image', () => {
  let imageFile = this.file('image').path;
  let instance = gm(imageFile).resize(240, 240).noProfile();
  let fn = think.promisify(instance.write, instance);
  return fn('/path/to/save/image.png');
}, {
  limit: 20 //一次执行 20 条
})
}
```

类

think.base

think.base 详细介绍请见 [这里](#)

think.http.base

think.http.base 详细介绍请见 [这里](#)

think.base

`think.base` 是基类，所有的类都会继承该类，该类提供了一些基本的方法。

使用 ES6 语法继承该类：

```
export default class extends think.base {
  /**
   * init method
   * @return {} []
   */
  init(){

  }
}
```

注： 使用 ES6 里的类时不要写 `constructor`，把初始化的一些操作放在 `init` 方法里，该方法在类实例化时自动被调用，效果等同于 `constructor`。

使用普通的方式继承该类：

```
module.exports = think.Class(think.base, {
  /**
   * init method
   * @return {} []
   */
  init: function(){

  }
})
```

init(...args)

- `args` {Array}

初始化方法，这里可以进行一些赋值等操作。

```
class a extends think.base {
  init(name, value){
    this.name = name;
    this.value = value;
  }
}
```

注：与 1.x 版本不同的是，2.x 版本 `init` 方法不再支持返回一个 `Promise`，一些通用操作放在 `__before` 魔术方法里进行。

__before()

前置魔术方法，可以将一些通用的行为放在这里进行，如：controller 里检测用户是否登录

```
export default class think.controller.base {
  /**
   * 前置魔术方法
   * @return {Promise} []
   */
  * __before(){
    let userInfo = yield this.session('userInfo');
    //如果没有登录，则跳转到登录页面
    if(think.isEmpty(userInfo)){
      return this.redirect('/login');
    }
    this.assign('userInfo', userInfo)
  }
}
```

__after()

后置魔术方法，在方法执行完成后在执行。

basename()

- `return` {String} 返回当前类文件的名称

获取当前类文件的名称，不包含文件具体路径和扩展名。

```
//假设当前类文件具体路径为 /home/xxx/project/app/controller/user.js
class a extends think.base {
  test(){
    var filename = this.basename();
    //returns 'user'
  }
}
```

parseModuleFromPath()

从当前类所在的 `filepath` 解析出所在对应的模块。

invoke(method, ...data)

- `method` {String} 要调用的方法名称
- `data` {Array} 传递的参数
- `return` {Promise}

调用一个方法，自动调用 `__before` 和 `__after` 魔术方法。不管方法本身是否返回 `Promise`，该方法始终返回 `Promise`。

方法本身支持是 `*/yield` 和 `async/await`。

```
//使用 async/await
class Cls extends think.base {
  async getValue(){
    let value = await this.getValue();
    return value;
  }
}
let instance = new Cls();
```

```
instance.invoke('getValue').then(data => {  
  
});
```

```
//使用 */yield  
class Cls extends think.base {  
  * getValue(){  
    let value = yield this.getValue();  
    return value;  
  }  
}  
let instance = new Cls();  
instance.invoke('getValue').then(data => {  
  
});
```

think.http.base

`think.http.base` 继承自 [think.base](#) 类，该类为含有 http 对象处理时的基类。middleware, controller, view 类都继承自该类。

使用 ES6 语法继承该类

```
export default class extends think.http.base {  
  /**  
   * 初始化方法，实例化时自动被调用，不要写 constructor  
   * @return {}  
   */  
  init(){  
  
  }  
}
```

使用普通的方式继承该类

```
module.exports = think.Class(think.http.base, {  
  init: function(){  
  
  }  
});
```

属性

http

封装的 http 对象，包含的属性和方法请见 [API -> http](#)。

方法

config(name, value)

- `name` {String} 配置名称
- `value` {Mixed} 配置值

读取或者设置配置，value 为 `undefined` 时为读取配置，否则为设置配置。

该方法不仅可以读取系统预设值的配置，也可以读取项目里定义的配置。

注：不可将当前请求的用户信息作为配置来设置，会被其他用户给冲掉。

```
export default class extends think.controller.base {
  indexAction(){
    //获取配置值
    let value = this.config('name');
  }
}
```

action(controller, action)

- `controller` {Object | String} controller实例
- `action` {String} action名称
- `return` {Promise}

调用 controller 下的 action，返回一个 Promise。自动调用 `__before` 和 `__after` 魔术方法。

如果 controller 是字符串，则自动去寻找对应的 controller。

```
//调用当前模块下controller里的action
export default class extends think.controller.base {
  async indexAction(){
    //调用user controller下的detail方法
    let value = await this.action('user', 'detail');
  }
}
```

```
}
```

```
//跨模块调用controller里的action
export default class extends think.controller.base {
  async indexAction(){
    //调用admin模块user controller下的detail方法
    let value = await this.action('admin/user', 'detail');
  }
}
```

cache(name, value, options)

- `name` {String} 缓存名称
- `value` {Mixed | Function} 缓存值
- `options` {Object} 缓存配置，具体见缓存配置

读取或者设置缓存，`value` 为 `undefined` 时是读取缓存，否则为设置缓存。默认缓存类型为 `file`。

```
export default class extends think.controller.base {
  async indexAction(){
    //获取缓存
    let value = await this.cache('name');
  }
}
```

当参数 `value` 为 function 时，表示获取缓存，如果缓存值不存在，则调用该 function，将返回值设置缓存并返回。这样避免在项目开发时要先判断缓存是否存在，然后再从相关地方读取值然后设置缓存的麻烦。

```
export default class extends think.controller.base {
  async indexAction(){
    //获取缓存，缓存不存在时自动调用 function，并设置缓存
    let value = await this.cache('name', () => {
      return this.model('user').select();
    });
  }
}
```

设置缓存并修改缓存类型：

```
export default class extends think.controller.base {
  async indexAction(){
    //设置缓存，缓存类型使用redis
    await this.cache('name', 'value', {
      type: 'redis'
    });
  }
}
```

hook(event, data)

- `event` {String} 事件名称
- `data` {Mixed} 参数
- `return` {Promise}

执行对应的事件，一个事件包含若干个 middleware，会按顺序执行这些 middleware。

事件可以在配置 `src/common/config/hook.js` 里定义，也可以通过 `think.hook` 来注册。

```
export default class extends think.controller.base {
  async indexAction(){
    let result = await this.hook('parse_data');
  }
}
```

model(name, options)

- `name` {String} 模型名称
- `options` {Object} 配置，具体见数据库配置
- `return` {Object} model实例

获取模型实例，默认获取当前模块下对应模型的实例，也可以跨模块获取模型的实例。

```
export default class extends think.controller.base {
  indexAction(){
    //获取当前模块下的 user model 的实例
    let model = this.model('user');
    //获取admin模块下 article model 的实例
    let model1 = this.model('admin/article');
    //获取当前模块下的 test model 的实例，并且是 sqlite 的数据库
    let model2 = this.model('test', {
      type: 'sqlite' //设置数据库类型为sqlite，更多参数见数据库配置
    })
  }
}
```

```
}  
}
```

controller(name)

- `name` {String} controller名称
- `return` {Object} controller实例

获取 Controller 的实例，如果 Controller 找不到，则报错。

```
export default class extends think.controller.base {  
  indexAction(){  
    //获取当前模块下 user controller 的实例  
    let controller = this.controller('user');  
    //获取admin模块下 user controller 的实例  
    let controller1 = this.controller('admin/user');  
  }  
}
```

service(name)

- `name` {String} service 名称
- `return` {Class}

获取对应的 service。service 返回的可能是 class，也可能直接是个对象，所以不会直接实例化。

```
export default class extends think.controller.base {  
  indexAction(){  
    //获取对应的service  
    let service = this.service('user');  
    //获取service的实例  
    let instance = new service(...args);  
    //获取admin模块下的user service  
    let service1 = this.service('admin/user');  
  }  
}
```

http

这里的 http 对象并不是 Node.js 里的 http 模块，而是对 request 和 response 对象包装后一个新的

对象。

```
var http = require('http');

http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('Hello World\n');
}).listen(8124);
```

如上面的代码所示，Node.js 创建服务时，会传递 request 和 response 2个对象给回调函数。为了后续调用方便，ThinkJS 对这2个对象进行了包装，包装成了 http 对象，并且提供很多有用的方法。

http 对象会在 middleware, logic, controller, view 中传递。

注：http 对象是 EventEmitter 的一个实例，所以可以对其进行事件监听和执行。

属性

http.req

系统原生的 request 对象

http.res

系统原生的 response 对象

http.startTime

请求的开始时间，是个 **unix** 时间戳。

http.url

当前请求的 url 。

http.version

当前请求的 http 版本。

http.method

当前请求的类型。

http.headers

当前请求的所有头信息。

http.pathname

当前请求的 pathname，路由识别会依赖该值，会在后续的处理中对其进行改变。所以在 action 拿到值可能跟初始解析出来的值不一致。

http.query

当前请求的所有 query 数据。

http.host

当前请求的 host，包含端口。

http.hostname

当前请求的 hostname，不包含端口。

http.payload

当前请求的 payload 数据，提交型的请求才含有该值。

注：该属性后续会废弃，建议使用 `http.getPayload` 方法。

http._get

存放 GET 参数值。

http._post

存放 POST 参数值

http._file

存放上传的文件数据

http._cookie

存放 cookie 数据。

http.module

当前请求解析后对应的模块名。

http.controller

当前请求解析后对应的控制器名。

http.action

当前请求解析后对应的操作名。

方法

http.getPayload()

- `return` {Promise} payload 内容

获取 payload。

http.config(name)

- `name` {String} 参数名
- `return` {Mixed} 返回对应的参数值

获取当前请求下对应的参数值。

http.referrer()

- `return` {String} 请求的 referrer

返回当前请求的 referrer。

http.userAgent()

- `return` {String} 请求的 userAgent

返回当前请求的 userAgent。

http.isGet()

- `return` {Boolean}

返回当前请求是否是 GET 请求。

http.isPost()

- `return` {Boolean}

返回当前请求是否是 POST 请求。

http.isAjax(method)

- `method` {String} 请求类型
- `return` {Boolean}

返回当前请求是否是 Ajax 请求。

```
http.isAjax(); //判断是否是Ajax请求
http.isAjax('GET'); //判断是否是Ajax请求，且请求类型是GET
```

http.isJsonp(name)

- `name` {String} callback 参数名称，默认为 callback
- `return` {Boolean}

返回当前请求是否是 jsonp 请求。

```
//url is /index/test?callback=testxxx
http.isJsonp(); //true
http.isJsonp('cb'); //false
```

http.get(name, value)

- `name` {String} 参数名称
- `value` {Mixed} 参数值

获取或者设置 GET 参数值。可以通过该方法设置 GET 参数值，方便后续的逻辑里获取。

```
// url is /index/test?name=thinkjs
http.get('name'); // returns 'thinkjs'
http.get('name', 'other value');
http.get('name'); // returns 'other value'
```

http.post(name, value)

- `name` {String} 参数名称
- `value` {Mixed} 参数值

获取或者设置 POST 值。可以通过该方法设置 POST 值，方便后续的逻辑里获取。

```
http.post('email'); //获取提交的email
```

http.param(name)

- `name` {String} 参数名称
- `return` {Mixed}

获取参数值，优先从 POST 里获取，如果值为空，则从 URL 参数里获取。

http.file(name)

- `name` {String} 文件对应的字段名称
- `return` {Object}

获取上传的文件。

```
http.file('image');
//returns
{
  fieldName: 'image', //表单里的字段名
  originalFilename: filename, //原始文件名
  path: filepath, //文件临时存放的路径
  size: size //文件大小
}
```

http.header(name, value)

- `name` {String} header 名称
- `value` {String} header 值

获取或者设置 header 信息。

```
http.header('accept'); //获取accept  
http.header('X-NAME', 'thinkjs'); //设置header
```

http.expires(time)

- `time` {Number} 过期时间，单位为秒

强缓存，设置 `Cache-Control` 和 `Expires` 头信息。

```
http.header(86400); //设置过期时间为 1 天。
```

http.status(status)

设置状态码。如果头信息已经发送，则无法设置状态码。

```
http.status(400); //设置状态码为400
```

http.ip()

获取用户的 ip 。如果使用了代理，获取的值可能不准。

http.lang(lang, asViewPath)

- `lang` {String} 要设置的语言
- `asViewPath` {Boolean} 是否添加一层模版语言目录

获取或者设置国际化的语言，可以支持模版路径要多一层语言的目录。

获取语言

```
let lang = http.lang();
```

获取语言的循序为 `http._lang` -> 从 cookie 中获取 -> 从 header 中获取，如果需从 url 中解析语言，可以获取后通过 `http.lang(lang)` 方法设置到属性 `http._lang` 中。

设置语言

```
let lang = getFromUrl();  
http.lang(lang, true); //设置语言，并指定模版路径中添加一层语言目录
```

http.theme(theme)

获取或者设置主题，设置后模版路径要多一层主题的目录。

http.cookie(name, value)

- `name` {String} cookie 名称
- `value` {String} cookie 值

读取或者设置 cookie 值。

```
http.cookie('think_test'); //获取名为 think_test 的 cookie  
http.cookie('name', 'value'); //设置 cookie，如果头信息已经发送则设置无效  
http.cookie('name', null); //删除 cookie
```

http.session(name, value)

- `name` {String} session 名
- `value` {Mixed} session 值
- `return` {Promise}

读取、设置和清除 session。

读取 Session

```
let value = await http.session('userInfo');
```

设置 Session

```
await http.session('userInfo', data);
```

清除 Session

```
await http.session();
```

http.redirect(url, status)

- `url` {String} 要跳转的 url
- `status` {Number} 状态码, 301 或者 302, 默认为302

页面跳转。

```
http.redirect('/login'); //跳转到登录页面
```

http.type(contentType, encoding)

- `contentType` {String} 要设置的 contentType
- `encoding` {String} 要设置的编码

获取或者设置 Content-Type。

```
http.type(); //获取Content-Type  
http.type('text/html'); //设置Content-Type, 会自动加上charset  
http.type('audio/mpeg', false); //设置Content-Type, 不追加charset
```

http.write(content, encoding)

- `content` {Mixed} 要输出的内容
- `encoding` {String} 编码

输出内容, 要调用 `http.end` 才能结束当前请求。

http.end(content, encoding)

- `content` {Mixed} 要输出的内容

- `encoding` {String} 编码

输出内容并结束当前请求。

http.success(data, message)

- `data` {Mixed} 要输出的数据
- `message` {String} 追加的message

格式化输出一个正常的的数据，一般是操作成功后输出。

```
http.success({name: 'thinkjs'});
//writes
{
  errno: 0,
  errmsg: '',
  data: {
    name: 'thinkjs'
  }
}
```

这样客户端就可以根据 `errno` 是否为 `0` 为判断当前请求是否正常。

http.fail(errno, errmsg, data)

- `errno` {Number} 错误号
- `errmsg` {String} 错误信息
- `data` {Mixed} 额外的数据

格式化输出一个异常的数据，一般是操作失败后输出。

注：字段名 `errno` 和 `errmsg` 可以在配置里进行修改。

```
http.fail(100, 'fail')
//writes
{
  errno: 100,
  errmsg: 'fail',
  data: ''
}
```

这样客户端就可以拿到具体的错误号和错误信息，然后根据需要显示了。

注：字段名 `errno` 和 `errmsg` 可以在配置里进行修改。

http.json(data)

- `data` {Object}

json 方式输出数据，会设置 Content-Type 为 `application/json`，该值对应的配置为 `json_content_type`。

controller

`think.controller.base` 继承自 [think.http.base](#) 类。项目里的控制器需要继承该类。

使用 ES6 的语法继承该类

```
export default class extends think.controller.base {
  indexAction(){

  }
}
```

使用普通方式继承该类 #####

```
module.exports = think.controller({
  indexAction(){

  }
})
```

属性

controller.http

传递进来的 [http](#) 对象。

方法

controller.ip()

- `return` {String}

获取当前请求用户的 ip, 等同与 http.ip 方法。

```
export default class extends think.controller.base {
  indexAction(){
    let ip = this.ip();
  }
}
```

controller.method()

- `return` {String}

获取当前请求的类型, 转化为小写。

```
export default class extends think.controller.base {
  indexAction(){
    let method = this.method(); //get or post ...
  }
}
```

controller.isMethod(method)

- `method` {String} 类型
- `return` {Boolean}

判断当前的请求类型是否是指定的类型。

controller.isGet()

- `return` {Boolean}

判断是否是 GET 请求。

controller.isPost()

- `return` {Boolean}

判断是否是 POST 请求。

controller.isAjax(method)

- `method` {String}
- `return` {Boolean}

判断是否是 Ajax 请求。如果指定了 `method`，那么请求类型也要相同。

```
export default class extends think.controller.base {
  indexAction(){
    //是ajax 且请求类型是 POST
    let isAjax = this.isAjax('post');
  }
}
```

controller.isWebSocket()

- `return` {Boolean}

是否是 websocket 请求。

controller.isCli()

- `return` {Boolean}

是否是命令行下调用。

controller.isJsonp(callback)

- `callback` {String} callback 名称
- `return` {Boolean}

是否是 jsonp 请求。

controller.get(name)

- `name` {String} 参数名

获取 GET 参数值。

```
export default class extends think.controller.base {
  indexAction(){
    //获取一个参数值
    let value = this.get('xxx');
    //获取所有的参数值
    let values = this.get();
  }
}
```

```
}
```

controller.post(name)

- `name` {String} 参数名

获取 POST 提交的参数。

```
export default class extends think.controller.base {
  indexAction(){
    //获取一个参数值
    let value = this.post('xxx');
    //获取所有的 POST 参数值
    let values = this.post();
  }
}
```

controller.param(name)

- `name` {String} 参数名

获取参数值，优先从 POST 里获取，如果取不到再从 GET 里获取。

controller.file(name)

- `name` {String} 上传文件对应的字段名

获取上传的文件，返回值是个对象，包含下面的属性：

```
{
  fieldName: 'file', //表单字段名称
  originalFilename: filename, //原始的文件名
  path: filepath, //文件保存的临时路径，使用时需要将其移动到项目里的目录，否则请求结束时会被删除
  size: 1000 //文件大小
}
```

如果文件不存在，那么值为一个空对象 `{}`。

controller.header(name, value)

- `name` {String} header 名
- `value` {String} header 值

获取或者设置 header。

```
export default class extends think.controller.base {
  indexAction(){
    let accept = this.header('accept'); //获取 header
    this.header('X-NAME', 'thinks'); //设置 header
  }
}
```

controller.expires(time)

- `time` {Number} 过期时间，单位为秒

强缓存，设置 `Cache-Control` 和 `Expires` 头信息。

```
export default class extends think.controller.base {
  indexAction(){
    this.expires(86400); //设置过期时间为 1 天。
  }
}
```

controller.userAgent()

获取 userAgent。

controller.referrer(onlyHost)

- `referrer` {Boolean} 是否只需要 host

获取 referrer。

controller.cookie(name, value, options)

- `name` {String} cookie 名
- `value` {String} cookie 值
- `options` {Object}

获取、设置或者删除 cookie。

```
export default class extends think.controller.base {
  indexAction(){
    //获取 cookie 值
    let value = this.cookie('think_name');
  }
}
```

```
export default class extends think.controller.base {
  indexAction(){
    //设置 cookie 值
    this.cookie('think_name', value, {
      timeout: 3600 * 24 * 7 //有效期为一周
    });
  }
}
```

```
export default class extends think.controller.base {
  indexAction(){
    //删除 cookie
    this.cookie('think_name', null);
  }
}
```

controller.session(name, value)

- `name` {String} session 名
- `value` {Mixed} session 值
- `return` {Promise}

读取、设置和清除 session。

读取 Session

```
export default class extends think.controller.base {
  async indexAction(){
    //获取session
    let value = await this.session('userInfo');
  }
}
```

设置 Session

```
export default class extends think.controller.base {
  async indexAction(){
    //设置 session
    await this.session('userInfo', data);
  }
}
```

清除 Session

```
export default class extends think.controller.base {
  async indexAction(){
    //清除当前用户的 session
    await this.session();
  }
}
```

controller.lang(lang, asViewPath)

- `lang` {String} 要设置的语言
- `asViewPath` {Boolean} 是否在模版目录添加一层语言目录

读取或者设置语言。

controller.locale(key)

- `key` {String}

根据 language 获取对应的语言文本。

controller.redirect(url, statusCode)

- `url` {String} 要跳转的 url
- `statusCode` {Number} 状态码, 默认为 302

页面跳转。

controller.assign(name, value)

- `name` {String | Object} 变量名
- `value` {Mixed} 变量值

将变量赋值到模版中。

```
export default class extends think.controller.base {
  indexAction(){
    //单个赋值
    this.assign('title', 'thinkjs');
    //批量赋值
    this.assign({
      name: 'xxx',
      desc: 'yyy'
    })
  }
}
```

controller.fetch(templateFile)

- `templateFile` {String} 模版文件地址
- `return` {Promise}

获取解析后的模版内容。

直接获取

```
// 假设文件路径为 /foo/bar/app/home/controller/index.js
export default class extends think.controller.base {
  async indexAction(){
    // home/index_index.html
    let content = await this.fetch();
  }
}
```

改变 action

```
// 假设文件路径为 /foo/bar/app/home/controller/index.js
export default class extends think.controller.base {
  async indexAction(){
    // home/index_detail.html
    let content = await this.fetch('detail');
  }
}
```

改变 controller 和 action

```
// 假设文件路径为 /foo/bar/app/home/controller/index.js
export default class extends think.controller.base {
```

```
async indexAction(){
  // home/user_detail.html
  let content = await this.fetch('user/detail');
}
}
```

改变 module, controller 和 action

```
// 假设文件路径为 /foo/bar/app/home/controller/index.js
export default class extends think.controller.base {
  async indexAction(){
    // admin/user_detail.html
    let content = await this.fetch('admin/user/detail');
  }
}
```

改变文件后缀名

```
// 假设文件路径为 /foo/bar/app/home/controller/index.js
export default class extends think.controller.base {
  async indexAction(){
    // home/index_detail.xml
    let content = await this.fetch('detail.xml');
  }
}
```

获取绝对路径文件

```
// 假设文件路径为 /foo/bar/app/home/controller/index.js
export default class extends think.controller.base {
  async indexAction(){
    // /home/xxx/aaa/bbb/c.html
    let content = await this.fetch('/home/xxx/aaa/bbb/c.html');
  }
}
```

controller.display(templateFile)

- `templateFile` {String} 模版文件路径

输出模版内容到浏览器端。查找模版文件策略和 `controller.fetch` 相同。

controller.jsonp(data)

- `data` {Mixed} 要输出的内容

jsonp 的方法输出内容，获取 callback 名称安全过滤后输出。

```
export default class extends think.controller.base {
  indexAction(){
    this.jsonp({name: 'thinkjs'});
    //writes
    'callback_fn_name({name: "thinkjs"})'
  }
}
```

controller.json(data)

- `data` {Mixed} 要输出的内容

json 的方式输出内容。

controller.status(status)

- `status` {Number} 状态码，默认为 404

设置状态码。

controller.deny(status)

- `status` {String} 状态码，默认为 403

拒绝当前请求。

controller.write(data, encoding)

- `data` {mixed} 要输出的内容
- `encoding` {String} 编码

输出内容

controller.end(data, encoding)

- `data` {mixed} 要输出的内容

- `encoding` {String} 编码

输出内容后结束当前请求。

controller.type(type, charset)

- `type` {String} Content-Type
- `charset` {Boolean} 是否自动追加 charset

设置 Content-Type。

controller.download(filePath, contentType, fileName)

- `filePath` {String} 下载文件的具体路径
- `content-Type` {String} Content-Type
- `fileName` {String} 保存的文件名

下载文件。

```
export default class extends think.controller.base {
  indexAction(){
    let filePath = think.RESOUCE_PATH + '/a.txt';
    //自动识别 Content-Type, 保存的文件名为 a.txt
    this.download(filePath);
  }
}
```

```
export default class extends think.controller.base {
  indexAction(){
    let filePath = think.RESOUCE_PATH + '/a.log';
    //自动识别 Content-Type, 保存的文件名为 b.txt
    this.download(filePath, 'b.txt');
  }
}
```

```
export default class extends think.controller.base {
  indexAction(){
    let filePath = think.RESOUCE_PATH + '/a.log';
    //指定 Content-Type 为 text/html, 保存的文件名为 b.txt
    this.download(filePath, 'text/html', 'b.txt');
  }
}
```

controller.success(data, message)

- `data` {Mixed} 要输出的数据
- `message` {String} 追加的message

格式化输出一个正常的的数据，一般是操作成功后输出。

```
http.success({name: 'thinkjs'});
//writes
{
  errno: 0,
  errmsg: '',
  data: {
    name: 'thinkjs'
  }
}
```

这样客户端就可以根据 `errno` 是否为 `0` 为判断当前请求是否正常。

controller.fail(errno, errmsg, data)

- `errno` {Number} 错误号
- `errmsg` {String} 错误信息
- `data` {Mixed} 额外的数据

格式化输出一个异常的数据，一般是操作失败后输出。

注：字段名 `errno` 和 `errmsg` 可以在配置里进行修改。

```
http.fail(100, 'fail')
//writes
{
  errno: 100,
  errmsg: 'fail',
  data: ''
}
```

这样客户端就可以拿到具体的错误号和错误信息，然后根据需要显示了。

注：字段名 `errno` 和 `errmsg` 可以在配置里进行修改。

controller.sendTime(name)

- `name` {String} header key

发送请求的执行时间，使用 header 的方式发出。

rest controller

`think.controller.rest` 继承自 [think.controller.base](#)，用来处理 Rest 接口。

使用 ES6 的语法继承该类

```
export default class extends think.controller.rest {  
  
}
```

使用普通方式继承该类

```
module.exports = think.controller('rest', {  
  
})
```

属性

controller._isRest

标识此 controller 对应的是 Rest 接口。如果在 `init` 方法里将该属性设置为 `false`，那么该 controller 不再是一个 Rest 接口。

controller._method

获取 method 方式。默认从 http method 中获取，但有些客户端不支持发送 DELETE, PUT 类型的请求，所以可以设置为从 GET 参数里获取。

```
export default class extends think.controller.rest {  
  init(http){  
    super.init(http);  
    //设置 _method, 表示从 GET 参数获取 _method 字段的值  
    //如果没有取到, 则从 http method 中获取  
    this._method = '_method';  
  }  
}
```

controller.resource

当前 Rest 对应的 Resource 名称。

controller.id

资源 ID

controller.modelInstance

资源对应 model 的实例。

方法

controller.__before()

可以在魔术方法 `__before` 中进行字段过滤、分页、权限校验等功能。

```
export default class extends think.controller.rest{
  __before(){
    //过滤 password 字段
    this.modelInstance.field('password', true);
  }
}
```

controller.getAction()

获取资源数据，如果有 id，拉取一条，否则拉取列表。

```
//方法实现，可以根据需要修改
export default class extends think.controller.rest {
  async getAction(){
    let data;
    if (this.id) {
      let pk = await this.modelInstance.getPk();
      data = await this.modelInstance.where({[pk]: this.id}).find();
      return this.success(data);
    }
    data = await this.modelInstance.select();
    return this.success(data);
  }
}
```

```
}  
}
```

controller.postAction()

添加数据

```
//方法实现，可以根据需要修改  
export default class extends think.controller.rest {  
  
  async postAction(){  
    let pk = await this.modelInstance.getPk();  
    let data = this.get();  
    delete data[pk];  
    if(think.isEmpty(data)){  
      return this.fail('data is empty');  
    }  
    let insertId = await this.modelInstance.add(data);  
    return this.success({id: insertId});  
  }  
}
```

controller.deleteAction()

删除数据

```
//方法实现，可以根据需要修改  
export default class extends think.controller.rest {  
  async deleteAction(){  
    if (!this.id) {  
      return this.fail('params error');  
    }  
    let pk = await this.modelInstance.getPk();  
    let rows = await this.modelInstance.where({[pk]: this.id}).delete();  
    return this.success({affectedRows: rows});  
  }  
}
```

controller.putAction()

更新数据

```
//方法实现，可以根据需要修改
export default class extends think.controller.rest {
  async putAction(){
    if (!this.id) {
      return this.fail('params error');
    }
    let pk = await this.modelInstance.getPk();
    let data = this.get();
    delete data[pk];
    if (think.isEmpty(data)) {
      return this.fail('data is empty');
    }
    let rows = await this.modelInstance.where({[pk]: this.id}).update(data);
    return this.success({affectedRows: rows});
  }
}
```

controller.__call()

找不到方法时调用

```
export default class extends think.controller.rest {
  __call(){
    return this.fail(think.locale('ACTION_INVALID', this.http.action, this.http.url));
  }
}
```

model

`think.model.base` 继承自 [think.base](#) 类。

使用 ES6 的语法继承该类

```
export default class extends think.model.base {
  getList(){
  }
}
```

使用普通方式继承该类

```
module.exports = think.model({
  getList: function(){

  }
})
```

属性

model.pk

数据表主键，默认为 `id`。

model.name

模型名，默认从当前文件名中解析。

当前文件路径为 `for/bar/app/home/model/user.js`，那么解析的模型名为 `user`。

model.tablePrefix

数据表名称前缀，默认为 `think_`。

model.tableName

数据表名称，不包含前缀。默认等于模型名。

model.schema

数据表字段，关系型数据库默认自动从数据表分析。

model.indexes

数据表索引，关系数据库会自动从数据表分析。

model.config

配置，实例化的时候指定。

model_db

连接数据库句柄。

model_data

操作的数据。

model_options

操作选项。

方法

model.model(name, options, module)

- `name` {String} 模型名称
- `options` {Object} 配置项
- `module` {String} 模块名
- `return` {Object}

获取模型实例，可以跨模块获取。

```
export default class extends think.model.base {
  async getList(){
    //获取 user 模型实例
    let instance = this.model('user');
    let list = await instance.select();
    let ids = list.map(item => {
      return item.id;
    });
    let data = await this.where({id: ['IN', ids]}).select();
    return data;
  }
}
```

model.getTablePrefix()

- `return` {string}

获取表名前缀。

model.getConfigKey()

- `return` {String}

获取配置对应的 key，缓存 db 句柄时使用。

model.db()

- `return` {Object}

根据当前的配置获取 db 实例，如果已经存在则直接返回。

model.getModelName()

- `return` {String} 模型名称

如果已经配置则直接返回，否则解析当前的文件名。

model.getTableName()

- `return` {String} 获取表名，包含表前缀

获取表名，包含表前缀。

model.cache(key, timeout)

- `key` {String} 缓存 key
- `timeout` {Number} 缓存有效时间，单位为秒
- `return` {this}

设置缓存选项。

设置缓存 key 和时间

```
export default class extends think.model.base {
  getList(){
    return this.cache('getList', 1000).where({id: {'>': 100}}).select();
  }
}
```

只设置缓存时间，缓存 key 自动生成

```
export default class extends think.model.base {
  getList(){
    return this.cache(1000).where({id: {'>': 100}}).select();
  }
}
```

设置更多的缓存信息

```
export default class extends think.model.base {
  getList(){
    return this.cache({
      key: 'getList',
      timeout: 1000,
      type: 'file' //使用文件方式缓存
    }).where({id: {'>': 100}}).select();
  }
}
```

model.limit(offset, length)

- `offset` {Number} 设置查询的起始位置
- `length` {Number} 设置查询的数据长度
- `return` {this}

设置查询结果的限制条件。

限制数据长度

```
export default class extends think.model.base {
  getList(){
    //查询20条数据
    return this.limit(20).where({id: {'>': 100}}).select();
  }
}
```

限制数据起始位置和长度

```
export default class extends think.model.base {
  getList(){
    //从起始位置100开始查询20调数据
    return this.limit(100, 20).where({id: {'>': 100}}).select();
  }
}
```

也可以直接传入一个数组，如：

```
export default class extends think.model.base {
  getList(){
    //从起始位置100开始查询20调数据
    return this.limit([100, 20]).where({id: {'>': 100}}).select();
  }
}
```

model.page(page, listRows)

- `page` {Number} 当前页，从 1 开始
- `listRows` {Number} 每页的条数
- `return` {this}

设置查询分页数据，自动转化为 `limit` 数据。

```
export default class extends think.model.base {
  getList(){
    //查询第 2 页数据，每页 10 条数据
    return this.page(2, 10).where({id: {'>': 100}}).select();
  }
}
```

也可以直接设置一个参数为数组，关联模型等情况下可能会有用。如：

```
export default class extends think.model.base {
  getList(){
    //查询第 2 页数据，每页 10 条数据
    return this.page([2, 10]).where({id: {'>': 100}}).select();
  }
}
```

model.where(where)

- `where` {String | Object} where 条件
- `return` {this}

设置 where 查询条件。可以通过属性 `_logic` 设置逻辑，默认为 `AND`。可以通过属性 `_complex`

设置复合查询。

注：1、以下示例不适合 mongo model，mongo 中设置 where 条件请见 model.mongo 里的 where 条件设定。2、where 条件中的值需要在 Logic 里做数据校验，否则可能会有漏洞。

普通条件

```
export default class extends think.model.base {
  where1(){
    //SELECT * FROM `think_user`
    return this.where().select();
  }
  where2(){
    //SELECT * FROM `think_user` WHERE ( `id` = 10 )
    return this.where({id: 10}).select();
  }
  where3(){
    //SELECT * FROM `think_user` WHERE ( id = 10 OR id < 2 )
    return this.where('id = 10 OR id < 2').select();
  }
  where4(){
    //SELECT * FROM `think_user` WHERE ( `id` != 10 )
    return this.where({id: ['!=', 10]}).select();
  }
}
```

null 条件

```
export default class extends think.model.base {
  where1(){
    //SELECT * FROM `think_user` where ( title IS NULL );
    return this.where({title: null}).select();
  }
  where2(){
    //SELECT * FROM `think_user` where ( title IS NOT NULL );
    return this.where({title: ['!=', null]}).select();
  }
}
```

EXP 条件

ThinkJS 默认会对字段和值进行转义，防止安全漏洞。有时候一些特殊的情况不希望被转义，可以使用 EXP 的方式，如：

```
export default class extends think.model.base {
  where1(){
    //SELECT * FROM `think_user` WHERE ( (`name` ='name') )
```

```

return this.where({name: ['EXP', "=\\\"name\\\""]}).select();
}
}

```

LIKE 条件

```

export default class extends think.model.base {
  where1(){
    //SELECT * FROM `think_user` WHERE ( `title` NOT LIKE 'welefen' )
    return this.where({title: ['NOTLIKE', 'welefen']}).select();
  }
  where2(){
    //SELECT * FROM `think_user` WHERE ( `title` LIKE '%welefen%' )
    return this.where({title: ['like', '%welefen%']}).select();
  }
  //like 多个值
  where3(){
    //SELECT * FROM `think_user` WHERE ( (`title` LIKE 'welefen' OR `title` LIKE '
    suredy' ) )
    return this.where({title: ['like', ['welefen', 'suredy']]}).select();
  }
  //多个字段或的关系 like 一个值
  where4(){
    //SELECT * FROM `think_user` WHERE ( (`title` LIKE '%welefen%') OR (`content`
    LIKE '%welefen%' ) )
    return this.where({'title|content': ['like', '%welefen%']}).select();
  }
  //多个字段与的关系 Like 一个值
  where5(){
    //SELECT * FROM `think_user` WHERE ( (`title` LIKE '%welefen%') AND (`content`
    LIKE '%welefen%' ) )
    return this.where({'title&content': ['like', '%welefen%']}).select();
  }
}

```

IN 条件

```

export default class extends think.model.base {
  where1(){
    //SELECT * FROM `think_user` WHERE ( `id` IN ('10','20') )
    return this.where({id: ['IN', '10,20']}).select();
  }
  where2(){
    //SELECT * FROM `think_user` WHERE ( `id` IN (10,20) )
    return this.where({id: ['IN', [10, 20]]}).select();
  }
  where3(){
    //SELECT * FROM `think_user` WHERE ( `id` NOT IN (10,20) )
  }
}

```

```
    return this.where({id: ['NOTIN', [10, 20]]}).select();
  }
}
```

BETWEEN 查询

```
export default class extends think.model.base {
  where1(){
    //SELECT * FROM `think_user` WHERE ( (`id` BETWEEN 1 AND 2) )
    return this.where({id: ['BETWEEN', 1, 2]}).select();
  }
  where2(){
    //SELECT * FROM `think_user` WHERE ( (`id` BETWEEN '1' AND '2') )
    return this.where({id: ['between', '1,2']}).select();
  }
}
```

多字段查询

```
export default class extends think.model.base {
  where1(){
    //SELECT * FROM `think_user` WHERE ( `id` = 10 ) AND ( `title` = 'www' )
    return this.where({id: 10, title: "www"}).select();
  }
  //修改逻辑为 OR
  where2(){
    //SELECT * FROM `think_user` WHERE ( `id` = 10 ) OR ( `title` = 'www' )
    return this.where({id: 10, title: "www", _logic: 'OR'}).select();
  }
  //修改逻辑为 XOR
  where2(){
    //SELECT * FROM `think_user` WHERE ( `id` = 10 ) XOR ( `title` = 'www' )
    return this.where({id: 10, title: "www", _logic: 'XOR'}).select();
  }
}
```

多条件查询

```
export default class extends think.model.base {
  where1(){
    //SELECT * FROM `think_user` WHERE ( `id` > 10 AND `id` < 20 )
    return this.where({id: {'>': 10, '<': 20}}).select();
  }
  //修改逻辑为 OR
  where2(){
    //SELECT * FROM `think_user` WHERE ( `id` < 10 OR `id` > 20 )
  }
}
```

```
return this.where({id: {'<': 10, '>': 20, _logic: 'OR'}}).select()
}
}
```

复合查询

```
export default class extends think.model.base {
  where1(){
    //SELECT * FROM `think_user` WHERE ( `title` = 'test' ) AND ( ( `id` IN (1,2,
3) ) OR ( `content` = 'www' ) )
    return this.where({
      title: 'test',
      _complex: {id: ['IN', [1, 2, 3]],
        content: 'www',
        _logic: 'or'
      }
    }).select()
  }
}
```

model.field(field)

- `field` {String | Array} 设置要查询的字段，可以是字符串，也可以是数组
- `return` {this}

设置要查询的字段。

字符串方式

```
export default class extends think.controller.base {
  async indexAction(){
    let model = this.model('user');
    //设置要查询的字符串，字符串方式，多个用逗号隔开
    let data = await model.field('name,title').select();
  }
}
```

调用 SQL 函数

```
export default class extends think.controller.base {
  //字段里调用 SQL 函数
  async listAction(){
    let model = this.model('user');
    let data = await model.field('id, INSTR(\`'30,35,31,\`,`id + \`,` as d').selec
```

```
t();  
  }  
}
```

数组方式

```
export default class extends think.controller.base {  
  async indexAction(){  
    let model = this.model('user');  
    //设置要查询的字符串，数组方式  
    let data = await model.field(['name','title']).select();  
  }  
}
```

model.fieldReverse(field)

- `field` {String | Array} 反选字段，即查询的时候不包含这些字段
- `return` {this}

设置反选字段，查询的时候会过滤这些字段，支持字符串和数组 2 种方式。

model.table(table, hasPrefix)

- `table` {String} 表名
- `hasPrefix` {Boolean} 是否已经有了表前缀，如果 `table` 值含有空格，则不在添加表前缀
- `return` {this}

设置表名，可以将一个 SQL 语句设置为表名。

设置当前表名

```
export default class extends think.model.base {  
  getList(){  
    return this.table('test', true).select();  
  }  
}
```

SQL 语句作为表名

```
export default class extends think.model.base {  
  async getList(){  
    let sql = await this.model('group').group('name').buildSql();
```

```
    let data = await this.table(sql).select();
    return data;
  }
}
```

model.union(union, all)

- `union` {String | Object} 联合查询 SQL 或者表名
- `all` {Boolean} 是否是 UNION ALL 方式
- `return` {this}

联合查询。

SQL 联合查询

```
export default class extends think.model.base {
  getList(){
    //SELECT * FROM `think_user` UNION (SELECT * FROM think_pic2)
    return this.union('SELECT * FROM think_pic2').select();
  }
}
```

表名联合查询

```
export default class extends think.model.base {
  getList(){
    //SELECT * FROM `think_user` UNION ALL (SELECT * FROM `think_pic2`)
    return this.union({table: 'think_pic2'}, true).select();
  }
}
```

model.join(join)

- `join` {String | Object | Array} 要组合的查询语句，默认为 `LEFT JOIN`
- `return` {this}

组合查询，支持字符串、数组和对象等多种方式。

字符串

```
export default class extends think.model.base {
  getList(){
```

```

//SELECT * FROM `think_user` LEFT JOIN think_cate ON think_group.cate_id=think
_cate.id
return this.join('think_cate ON think_group.cate_id=think_cate.id').select();
}
}

```

数组

```

export default class extends think.model.base {
  getList(){
    //SELECT * FROM `think_user` LEFT JOIN think_cate ON think_group.cate_id=think
    _cate.id RIGHT JOIN think_tag ON think_group.tag_id=think_tag.id
    return this.join([
      'think_cate ON think_group.cate_id=think_cate.id',
      'RIGHT JOIN think_tag ON think_group.tag_id=think_tag.id'
    ]).select();
  }
}

```

对象：单个表

```

export default class extends think.model.base {
  getList(){
    //SELECT * FROM `think_user` INNER JOIN `think_cate` AS c ON think_user.`cate_
    id`=c.`id`
    return this.join({
      table: 'cate',
      join: 'inner', //join 方式, 有 left, right, inner 3 种方式
      as: 'c', // 表别名
      on: ['cate_id', 'id'] //ON 条件
    }).select();
  }
}

```

对象：多次 JOIN

```

export default class extends think.model.base {
  getList(){
    //SELECT * FROM think_user AS a LEFT JOIN `think_cate` AS c ON a.`cate_id`=c.`
    id` LEFT JOIN `think_group_tag` AS d ON a.`id`=d.`group_id`
    return this.alias('a').join({
      table: 'cate',
      join: 'left',
      as: 'c',
      on: ['cate_id', 'id']
    }).join({

```

```

    table: 'group_tag',
    join: 'left',
    as: 'd',
    on: ['id', 'group_id']
  }).select()
}
}

```

对象：多个表

```

export default class extends think.model.base {
  getList(){
    //SELECT * FROM `think_user` LEFT JOIN `think_cate` ON think_user.`id`=think_cate.`id` LEFT JOIN `think_group_tag` ON think_user.`id`=think_group_tag.`group_id`
    return this.join({
      cate: {
        on: ['id', 'id']
      },
      group_tag: {
        on: ['id', 'group_id']
      }
    }).select();
  }
}

```

```

export default class extends think.model.base {
  getList(){
    //SELECT * FROM think_user AS a LEFT JOIN `think_cate` AS c ON a.`id`=c.`id` LEFT JOIN `think_group_tag` AS d ON a.`id`=d.`group_id`
    return this.alias('a').join({
      cate: {
        join: 'left', // 有 left,right,inner 3 个值
        as: 'c',
        on: ['id', 'id']
      },
      group_tag: {
        join: 'left',
        as: 'd',
        on: ['id', 'group_id']
      }
    }).select()
  }
}

```

对象：ON 条件含有多个字段

```

export default class extends think.model.base {
  getList(){
    //SELECT * FROM `think_user` LEFT JOIN `think_cate` ON think_user.`id`=think_c
    ate.`id` LEFT JOIN `think_group_tag` ON think_user.`id`=think_group_tag.`group_id`
    LEFT JOIN `think_tag` ON (think_user.`id`=think_tag.`id` AND think_user.`title`=t
    hink_tag.`name`)
    return this.join({
      cate: {on: 'id, id'},
      group_tag: {on: ['id', 'group_id']},
      tag: {
        on: { // 多个字段的 ON
          id: 'id',
          title: 'name'
        }
      }
    }).select()
  }
}

```

对象: table 值为 SQL 语句

```

export default class extends think.model.base {
  async getList(){
    let sql = await this.model('group').buildSql();
    //SELECT * FROM `think_user` LEFT JOIN ( SELECT * FROM `think_group` ) ON thin
    k_user.`gid`=( SELECT * FROM `think_group` ).`id`
    return this.join({
      table: sql,
      on: ['gid', 'id']
    }).select();
  }
}

```

model.order(order)

- `order` {String | Array | Object} 排序方式
- `return` {this}

设置排序方式。

字符串

```

export default class extends think.model.base {
  getList(){
    //SELECT * FROM `think_user` ORDER BY id DESC, name ASC
    return this.order('id DESC, name ASC').select();
  }
}

```

```
}
getList1(){
  //SELECT * FROM `think_user` ORDER BY count(num) DESC
  return this.order('count(num) DESC').select();
}
}
```

数组

```
export default class extends think.model.base {
  getList(){
    //SELECT * FROM `think_user` ORDER BY id DESC,name ASC
    return this.order(['id DESC', 'name ASC']).select();
  }
}
```

对象

```
export default class extends think.model.base {
  getList(){
    //SELECT * FROM `think_user` ORDER BY `id` DESC,`name` ASC
    return this.order({
      id: 'DESC',
      name: 'ASC'
    }).select();
  }
}
```

model.alias(tableAlias)

- `tableAlias` {String} 表别名
- `return` {this}

设置表别名。

```
export default class extends think.model.base {
  getList(){
    //SELECT * FROM think_user AS a;
    return this.alias('a').select();
  }
}
```

model.having(having)

- `having` {String} having 查询的字符串
- `return` {this}

设置 having 查询。

```
export default class extends think.model.base {
  getList(){
    //SELECT * FROM `think_user` HAVING view_nums > 1000 AND view_nums < 2000
    return this.having('view_nums > 1000 AND view_nums < 2000').select();
  }
}
```

model.group(group)

- `group` {String} 分组查询的字段
- `return` {this}

设定分组查询。

```
export default class extends think.model.base {
  getList(){
    //SELECT * FROM `think_user` GROUP BY `name`
    return this.group('name').select();
  }
}
```

model.distinct(distinct)

- `distinct` {String} 去重的字段
- `return` {this}

去重查询。

```
export default class extends think.model.base {
  getList(){
    //SELECT DISTINCT `name` FROM `think_user`
    return this.distinct('name').select();
  }
}
```

model.explain(explain)

- `explain` {Boolean} 是否添加 explain 执行
- `return` {this}

是否在 SQL 之前添加 explain 执行，用来查看 SQL 的性能。

model.optionsFilter(options)

操作选项过滤。

model.dataFilter(data)

- `data` {Object | Array} 要操作的数据

数据过滤。

model.beforeAdd(data)

- `data` {Object} 要添加的数据

添加前置操作。

model.afterAdd(data)

- `data` {Object} 要添加的数据

添加后置操作。

model.afterDelete(data)

删除后置操作。

model.beforeUpdate(data)

- `data` {Object} 要更新的数据

更新前置操作。

model.afterUpdate(data)

- `data` {Object} 要更新的数据

更新后置操作。

model.afterFind(data)

- `data` {Object} 查询的单条数据
- `return` {Object | Promise}

`find` 查询后置操作。

model.afterSelect(data)

- `data` [Array] 查询的数据数据
- `return` {Array | Promise}

`select` 查询后置操作。

model.data(data)

- `data` {Object}

添加和更新操作时设置操作的数据。

model.options(options)

- `options` {Object}

设置操作选项。如：

```
export default class extends think.model.base {
  getList(){
    return this.options({
      where: 'id = 1',
      limit: [10, 1]
    }).select();
  }
}
```

model.close()

关闭数据库连接，一般情况下不要直接调用。

model.getSchema(table)

- `table` {String} 表名
- `return` {Promise}

获取表的字段信息，自动从数据库中读取。

model.getTableFields(table)

`已废弃`，使用 `model.getSchema` 方法替代。

model.getLastSql()

- `return` {String}

获取最后执行的 SQL 语句。

model.buildSql()

- `return` {Promise}

将当前的查询条件生成一个 SQL 语句。

model.parseOptions(oriOpts, extraOptions)

- `oriOpts` {Object}
- `extraOptions` {Object}
- `return` {Promise}

根据已经设定的一些条件解析当前的操作选项。

model.getPk()

- `return` {Promise}

返回 `pk` 的值，返回一个 Promise。

model.parseType(field, value)

- `field` {String} 数据表中的字段名称
- `value` {Mixed}
- `return` {Mixed}

根据数据表中的字段类型解析 value。

model.parseData(data)

- `data` {Object} 要解析的数据
- `return` {Object}

调用 `parseType` 方法解析数据。

model.add(data, options, replace)

- `data` {Object} 要添加的数据
- `options` {Object} 操作选项
- `replace` {Boolean} 是否是替换操作
- `return` {Promise} 返回插入的 ID

添加一条数据。

model.thenAdd(data, where)

- `data` {Object} 要添加的数据
- `where` {Object} where 条件
- `return` {Promise}

当 where 条件未命中到任何数据时才添加数据。

model.addMany(dataList, options, replace)

- `dataList` {Array} 要添加的数据列表
- `options` {Object} 操作选项
- `replace` {Boolean} 是否是替换操作
- `return` {Promise} 返回插入的 ID

一次添加多条数据。

model.delete(options)

- `options` {Object} 操作选项
- `return` {Promise} 返回影响的行数

删除数据。

删除 id 为 7 的数据。

```
model.delete({
```

```
where: {id: 7}
});
```

model.update(data, options)

- `data` {Object} 要更新的数据
- `options` {Object} 操作选项
- `return` {Promise} 返回影响的行数

更新数据。

updateMany(dataList, options)

- `dataList` {Array} 要更新的数据列表
- `options` {Object} 操作选项
- `return` {Promise}

更新多条数据，`dataList` 里必须包含主键的值，会自动设置为更新条件。

model.increment(field, step)

- `field` {String} 字段名
- `step` {Number} 增加的值，默认为 1
- `return` {Promise}

字段值增加。

model.decrement(field, step)

- `field` {String} 字段名
- `step` {Number} 增加的值，默认为 1
- `return` {Promise}

字段值减少。

model.find(options)

- `options` {Object} 操作选项
- `return` {Promise} 返回单条数据

查询单条数据，返回的数据类型为对象。如果未查询到相关数据，返回值为 `{}`。

model.select(options)

- `options` {Object} 操作选项
- `return` {Promise} 返回多条数据

查询多条数据，返回的数据类型为数组。如果未查询到相关数据，返回值为 `[]`。

model.countSelect(options, pageFlag)

- `options` {Object} 操作选项
- `pageFlag` {Boolean} 当页数不合法时处理，true 为修正到第一页，false 为修正到最后一页，默认不修正
- `return` {Promise}

分页查询，一般需要结合 `page` 方法一起使用。如：

```
export default class extends think.controller.base {
  async listAction(){
    let model = this.model('user');
    let data = await model.page(this.get('page')).countSelect();
  }
}
```

返回值数据结构如下：

```
{
  numsPerPage: 10, //每页显示的条数
  currentPage: 1, //当前页
  count: 100, //总条数
  totalPages: 10, //总页数
  data: [{ //当前页下的数据列表
    name: "thinkjs",
    email: "admin@thinkjs.org"
  }, ...]
}
```

model.getField(field, one)

- `field` {String} 字段名，多个字段用逗号隔开
- `one` {Boolean | Number} 获取的条数
- `return` {Promise}

获取特定字段的值。

model.count(field)

- `field` {String} 字段名
- `return` {Promise} 返回总条数

获取总条数。

model.sum(field)

- `field` {String} 字段名
- `return` {Promise}

对字段值进行求和。

model.min(field)

- `field` {String} 字段名
- `return` {Promise}

求字段的最小值。

model.max(field)

- `field` {String} 字段名
- `return` {Promise}

求字段的最大值。

model.avg(field)

- `field` {String} 字段名
- `return` {Promise}

求字段的平均值。

model.query(...args)

- `return` {Promise}

指定 SQL 语句执行查询。

model.execute(...args)

- `return` {Promise}

执行 SQL 语句。

model.parseSql(sql, ...args)

- `sql` {String} 要解析的 SQL 语句
- `return` {String}

解析 SQL 语句，调用 `util.format` 方法解析 SQL 语句，并将 SQL 语句中的 `__TABLENAME__` 解析为对应的表名。

```
export default class extends think.model.base {
  getSql(){
    let sql = 'SELECT * FROM __GROUP__ WHERE id=%d';
    sql = this.parseSql(sql, 10);
    //sql is SELECT * FROM think_group WHERE id=10
  }
}
```

model.startTrans()

- `return` {Promise}

开启事务。

model.commit()

- `return` {Promise}

提交事务。

model.rollback()

- `return` {Promise}

回滚事务。

model.transaction(fn)

- `fn` {Function} 要执行的函数
- `return` {Promise}

使用事务来执行传递的函数，函数要返回 Promise。

```
export default class extends think.model.base {
  updateData(data){
    return this.transaction(async () => {
      let insertId = await this.add(data);
      let result = await this.model('user_cate').add({user_id: insertId, cate_id:
100});
      return result;
    })
  }
}
```

MongoDB

`think.model.mongo` 继承类 [think.model.base](#)。

使用 ES6 的语法继承该类

```
export default class extends think.model.mongo {
  getList(){
  }
}
```

使用普通方式继承该类

```
module.exports = think.model('mongo', {
  getList: function(){
  }
})
```

属性

`model.schema`

设置字段，如：

```
export default class extends think.model.mongo {
```

```
init(...args){
  super.init(...args);
  //设置字段
  this.schema = {
    name: {
      type: 'string'
    },
    pwd: {
      type: 'string'
    }
  }
}
```

注：目前框架并不会对字段进行检查。

model.indexes

设置字段索引，数据操作之前会自动创建索引。

```
export default class extends think.model.mongo {
  init(...args){
    super.init(...args);
    //配置索引
    this.indexes = {

    }
  }
}
```

单字段索引

```
export default class extends think.model.mongo {
  init(...args){
    super.init(...args);
    //配置索引
    this.indexes = {
      name: 1
    }
  }
}
```

唯一索引

通过 `$unique` 来指定为唯一索引，如：

```
export default class extends think.model.mongo {
  init(...args){
    super.init(...args);
    //配置索引
    this.indexes = {
      name: {$unique: 1}
    }
  }
}
```

多字段索引

可以将多个字段联合索引，如：

```
export default class extends think.model.mongo {
  init(...args){
    super.init(...args);
    //配置索引
    this.indexes = {
      email: 1
      test: {
        name: 1,
        title: 1,
        $unique: 1
      }
    }
  }
}
```

model.pk

主键名，默认为 `_id`，可以通过 `this.getPk` 方法获取。

方法

model.where(where)

mongo 模型中的 where 条件设置和关系数据库中不太一样。

等于判断

```
export default class extends think.model.mongo {
  where1(){
```

```
    return this.where({ type: "snacks" }).select();
  }
}
```

AND 条件

```
export default class extends think.model.mongo {
  where1(){
    return this.where({ type: 'food', price: { $lt: 9.95 } }).select();
  }
}
```

OR 条件

```
export default class extends think.model.mongo {
  where1(){
    return this.where({
      $or: [ { qty: { $gt: 100 } }, { price: { $lt: 9.95 } } ]
    }).select();
  }
  where2(){
    return this.where({
      type: 'food',
      $or: [ { qty: { $gt: 100 } }, { price: { $lt: 9.95 } } ]
    }).select();
  }
}
```

内嵌文档

```
export default class extends think.model.mongo {
  where1(){
    return this.where( {
      producer:
        {
          company: 'ABC123',
          address: '123 Street'
        }
    }).select();
  }
  where2(){
    return this.where({ 'producer.company': 'ABC123' }).select();
  }
}
```

IN 条件

```
export default class extends think.model.mongo {
  where1(){
    return this.where({ type: { $in: [ 'food', 'snacks' ] } }).select();
  }
}
```

更多文档请见 <https://docs.mongodb.org/manual/reference/operator/query/#query-selectors>。

model.collection()

- `return` {Promise}

获取操作当前表的句柄。

```
export default class extends think.model.mongo {
  async getIndexes(){
    let collection = await this.collection();
    return collection.indexes();
  }
}
```

model.aggregate(options)

聚合查询。具体请见 <https://docs.mongodb.org/manual/core/aggregation-introduction/>。

model.mapReduce(map, reduce, out)

mapReduce 操作，具体请见 <https://docs.mongodb.org/manual/core/map-reduce/>。

model.createIndex(indexes, options)

- `indexes` {Object} 索引配置
- `options` {Object}

创建索引。

model.getIndexes()

- `return` {Promise}

获取索引。

middleware

`think.middleware.base` 类继承自 [think.http.base](#)。

ES6 方式

```
export default class extends think.middleware.base {
  run(){

  }
}
```

动态创建类的方式

```
module.exports = think.middleware({
  run: function(){

  }
})
```

方法

middleware.run()

- `return` {Promise}

middleware 对外的方法入口，调用 middleware 时会自动调用该方法。