

ThinkJS 2.2 Documentation

Getting Started

Introduction

ThinkJS is the first Node.js MVC framework that supporting use full ES6/7 features to develop Node.js application. By using `async/await` in ES7 or `*/yield` in ES6, ThinkJS totally resolved the hard problem of asynchronous callbacks nesting hell. It absorbs the design and ideas of lots of other frameworks around the world, and makes develop Node.js projects faster and efficient than ever before.

Using ES6/7 features to build projects is very efficient, that must be the trend. The latest version of Node.js has supported the features of ES6 more friendly, though not all features. At this time, [Babel](#) can help us to compile ES6 code to ES5 code.

Features

Using ES6/7 features

Babel compile our code to ES5 so we don't need worry about the browser compatibility. So we can resolve the asynchronous callbacks problem by using `async/await` or `*/yield` features.

```
//user controller, home/controller/user.js
export default class extends think.controller.base {
  //login action
  async loginAction(self){
    //if it's a get request, then display them
    if(this.isGet()){
      return this.display();
    }
    //here you can use post() method to get all request data which has checked in logic
    let data = this.post();
    let md5 = think.md5('think_' + data.pwd);
    //take username and encrypted password to match data in database
    let result = await this.model('user').where({name: data.name, pwd: md5}).find();
    //if no result found, it means username or password error
    if(think.isEmpty(result)){
      return this.fail('login fail');
    }
    //write user info into session after receiving user information
    await this.session('userInfo', result);
    return this.success();
  }
}
```

JavaScript

We've used ES6 features like `class`, `export`, `let` and ES7 features like `async/await` in this example. Database queries and `Session` writing were all asynchronous actions, but here we are writing sync code to handle them with `async/await`. Last, it can run in Node.js environment stably after [Babel](#) compiling.

Supports TypeScript

[TypeScript](#) is a free and open source programming language designed by Microsoft. TypeScript is a typed superset of JavaScript that it has some useful function in large project such as optional static type.

ThinkJS 2.1 has supported TypeScript, the code will be auto compiled and updated during the develop process. You can know more from [here](#).

Supports variant project structures and environments

When using ThinkJS, you can apply single module mode, general mode or multiple modules mode, and to develop projects with its complexity range from very low to very high.

By default, there are three kinds of project environments: `development`, `testing` and `production`, you can use different configuration in different environment to support different requests. You can also custom and extend them in your projects.

Supports Abundant Database Type

ThinkJS supports `MySQL`, `MongoDB` and `SQLite`. It encapsulates many APIs of the database operations, without having to manually stitching SQL statements. You can automatically prevent SQL injection and other vulnerabilities. It also supports transaction and association and other advanced features.

Automatic Updating

ThinkJS has a mechanism that could automatically update codes after source files being modified without resort to restart Node.js server and other third party modules.

Automatic REST API Creation

You can use `thinkjs` command to create REST API automatically without writing any extra code. Meanwhile, you can also easily add filter or auth check if you want.

Supports multiple WebSocket libraries

ThinkJS supports some common WebSocket libraries like `socket.io` and `sockjs`, and packages them to provide the consistent APIs to developers.

Plentiful Test Cases

ThinkJS includes 1500+ test cases with the code coverage at 95%. Every change has its test case to insure the framework functions well.

Supports CLI to run cron job

`Action` in ThinkJS can both response to user request and the CLI invoke. With this feature, we can excute cron job more easily.

Hooks and Middlewares

ThinkJS supports Hooks and Middlewares, they make the requests handling much more flexible.

Detailed log

ThinkJS builds-in the detailed log function, it makes us read log and track problems easily.

HTTP request log

```
[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 connection log

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

Error log

```
[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
```

Configurable Routers

The routers ThinkJS supported include regex router, rule router and static router, and router can be set based on modules. That's very helpful for us to make URLs more simple and reserve their high performance at the same time.

Supports international and custom themes

ThinkJS provides us very simple methods to implement i18n and custom themes.

Comparing With Other Frameworks

Express/Koa

Express and koa are simple frameworks, they all only provide the very basic functions. So for developing complex projects, one must introduces the third party plugins. Though small cores often mean big flexibility, the introducing of other plugins would increases the project's complexity. Besides, no one can ensure all the third party plugins are safety and efficient.

Koa 1.x solved asynchronous callbacks problem by using `*/yield` feature. But the newer `async/await` feature will replace `*/yield` at last. ThinkJS supports both features well.

On the other hand, ThinkJS choosed to provide the full set of solutions. But not only that, in ThinkJS, every function has been strictly tested for performance optimazition and prevent mermory leaks. And the important thing is that we can use all ES6/7 feature in the project directly.

Sails

Sails is another Node.js framework that also provides complete solution. It's convinient because of the encapsulation of databases, REST APIs and security features.

But Sails still uses callbacks in asynchronous code. That's too hard to develop, and can't use ES6/7 features naturally in the projects.

Disadvantages

Even though ThinkJS has many advantages, it has also a few disadvantages too, for example:

- ThinkJS is a relatively new framework, the community is not strong enough.
- ThinkJS is short of large scale applications.

Performance Comparison

To evaluate whether a framework is good or not, the features it provided and the performance it could reach are equally important. Although ThinkJS more suits for large projects, features and complexity far exceeds Express and Koa, but its performance is not much less than them.

Task	QPS	RT(ms)	Success Rate	Times
Sails_2016-01-13 19:02:58	515	355.2	100%	2016-01-13 19:03:03 - 2016-01-13 19:03:23
ThinkJS_2016-01-13 19:00:09	2001	83.04	100%	2016-01-13 19:01:02 - 2016-01-13 19:01:23
Express_2016-01-13 18:52:16	2896	57.92	100%	2016-01-13 18:53:02 - 2016-01-13 18:53:24
Koa_2016-01-13 18:47:55	2482	62.33	100%	2016-01-13 18:48:02 - 2016-01-13 18:48:24
Node_2016-01-13 18:42:55	6390	24.61	100%	2016-01-13 18:43:02 - 2016-01-13 18:43:25

tips : The above data using distributed stress testing system to test.

All we can see is that there has just little distance between ThinkJS and Express/Koa. ThinkJS and Sails.js both more suit for large projects, but ThinkJS has higher performance than Sails.js.

You can go <https://github.com/thinkjs-team/thinkjs-performance-test> to clone all testing code and run it locally. If you use **ab** testing tool, note that it is instability on Mac.

ES6/7 reference documentation

You can read more about ES6/7 features here:

- [learn-es2015](#)
- [ECMAScript 6 Guide](#)
- [ECMAScript 6 Features](#)
- [ECMAScript 6 compatibility table](#)
- [ECMAScript 7 Features](#)
- [ECMAScript 7 compatibility table](#)

Create project

Install Node.js

ThinkJS is a Node.js MVC framework, it requires Node.js before you run it. You can install Node.js by go to <https://nodejs.org> to download the latest installation.

After installation, type `node -v` in your terminal. If it outputs version number, it installs success.

ThinkJS requires the version of Node.js `>=0.12.0`, if your version lower than it, you need update your Node.js, or you can't start the service. we recommend use Node.js `4.2.1`.

Install ThinkJS

Install ThinkJS by following command:

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

Bash

After installation, run `thinkjs --version` or `thinkjs -V` to check version number.

Tips: If you have installed ThinkJS 1.x before, you need remove it by `npm uninstall -g thinkjs-cmd` first of all.

Update ThinkJS

Update ThinkJS globally by run the following command:

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

Bash

Update ThinkJS in you current project by run the following command:

```
npm install thinkjs@2
```

Bash

User thinkjs command to create Project

After installation, you can create a new ThinkJS project by run the following command:

```
thinkjs new project_path; #project_path is the path you want store your project
```

Bash

If you want to use ES6 features in the development, you may want to create the ES6 mode project by use following command:

```
thinkjs new project_path --es; #project_path is the path you want store your project
```

Bash

If the output is like the following, that means you have created the project successfully:

```
create : demo/
create : demo/package.json
create : demo/.thinkjsrc
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
```

For more details about creating project, go to [extension function -> ThinkJS command](#).

Install dependencies

After project creation, go to the project directory and run `npm install` to install dependencies.

```
npm install
```

Bash

Compile Project

Since v2.0.6, ThinkJS has built-in the automatical compiling feature, so you don't need run `npm run watch-compile` for real time compile anymore. What you only need to do, is just start your service by run `npm start`.

Start Project

Run `npm start`, if terminal returns output like following, it means the service run success.

```
[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
```

Access Project

Open your browser and go to `http://127.0.0.1:8360` . If you are in a remote machine, you must replace `127.0.0.1` with your remote machine's IP.

Project Structure

After creating ThinkJS project, you will get the directory structure something like the following:

```
|-- 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
```

tips: Different mode used when creating the project, may result in the slightly different structure.

nginx.conf

This is the nginx's configuration file. When deploy your project to the production environment, we recommend you using nginx as the reverse proxy.

src

`src` folder holds all of the source files, but you can only have it by using `--es6` option when create projects. After start project, the source files in `src/` will be compiled automatically into the `app/` folder with same name.

src/common

You should place the common module files into this folder, the so-called common files used to store the code logic that could be used all around the project.

src/common/bootstrap

Files in this folder will be autoload when project bootstrapping, so you don't need to `require` them manually. You can define global functions, register middlewares by using this folder.

Defining global functions

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

JavaScript

We defined a global function `formatDate` here, you can call it anywhere in the project after define it here.

Register Middlewares

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

JavaScript

We defined a middleware `replace_image` here, then you can register it in the configure file `hook.js` .

tips: bootstrap can only stay in common module.

src/common/config

You can place the common config files here. Bear in mind, the route.js, hook.js and locale.js must stay within this folder.

```
'use strict';
/**
 * config
 */
export default {
  //key: value
};
```

JavaScript

src/common/controller

Within this folder, you should put the common controller files. For example, the `error.js` has designed some different error handling behaviors, you can modify it or add other controller according to the project requirements.

src/common/runtime

This is a temp folder to store for example cache files, upload files and other files at the runtime.

src/home

`home` module is a default module for your project. You can change your default module to other directory by add `default_module` option and assign a value to it in `src/common/config/config.js` .

src/home/logic

Before every operation execution, it is possible to validate something in advance here, so as to decrease the complexity of the actions in the controllers. For example, we can validate whether the parameters meet the requirements, the input data are acceptability, or current user have the access to do something.

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

src/home/controller

Controller folder. Each `url` has an matched `action` within the matched `controller` .

```
'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

Models to handle database operations.

view

The `view` folder used to store template files. If you need support i18n or multiple themes, you should create the sub-folders respectively.

www

Our project's root which we have to access, nginx's configure root will be set here.

www/index.js

Our project's entry file in development mode, it can be modified as the project's need. When in production environment, the entry file will be `www/production.js` .

The content of index.js is something like this:

```
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.run();
```

www/static

Holding the static files.

Specification

File Path Must Be Lowercased

Generally, ThinkJS projects would be deployed in Linux environment, although they are developed in Windows or Mac OSX environment.

In Windows and Mac, file paths are not case-sensitive, but in Linux they are case-sensitive. This may result in errors after deployed projects online.

To avoid this happen, it's recommended that all file paths use lowercase. This way, ThinkJS will scan your project paths after service started, and return warning messages like this if it found uppercase paths:

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

Indent Two Spaces

Sometimes, complicated logic will result in multi-levels indent in Node.js. We advice each line intent two spaces to prevent indent too deep.

Use ES6 Grammars

ES6 has lots of new features that can make our code simple and effective. Node.js has supported much of ES6 features in the latest version. You can use Babel compile your code to support all features.

Do Not Use constructor Method

If you use ES6's `class`, the `constructor` method can be used to make something auto run when it's instantiated. For example:

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

JavaScript

But if you are not using ES6's `class` grammar, you should not use `constructor` method.

ThinkJS provide `init` method to replace `constructor`. It will called automatically whether you using `class` or not.

```
export default class think.base {
  /**
   * Initial method, called when class instanced
   * @return {} []
   */
  init(){
    ...
  }
}
```

JavaScript

Tips: All ThinkJS class will extend the base class `think.base`.

Compile by Babel

The latest version of Node.js has supported most of ES6 features, but some of these features (e.g. `*/yield`) have not optimized in V8.

We advise you to compile your project code with Babel. Babel can identify almost all ES6 and ES7 grammar, and the performance of compiled code is higher than native-supporting.

Replace `*/yield` with `async/await`

`*/yield` is an ES6 feature to resolve async callback issue, and ES7 replace it with `async/await`.

Compared to `async/await`, `*/yield` has four shortcomings:

1. `*/yield` return a generator that need a third module such as `co` to run.
2. `*/yield` can't use with `Arrow Function` together.
3. When one `*/yield` need call another `*/yield`, we need use `yield *` command
4. V8 has not made optimization for `*/yield`, so we recommend you to use Babel. With Babel, you can use ES7 `async/await` to replace `*/yield`.

Upgrade Guide

This documentation is a guide about how to update ThinkJS from 2.1 to 2.2, look at [here](#) if you want to update from 2.0 to 2.1.

The version 2.2 is compatible with version 2.01. The new version has appended more functions and changed something lightly, you can reference it's [ChangeLog](#) to know what has changed.

Update ThinkJS Dependency Versions

Change Thinkjs's version to 2.2.x in package.json.

Add new dependency

Run `npm install source-map-support --save` to install dependency in the project folder.

If it's typescript project, it needs run `npm install source-map --save`.

Modify compile script

If your project is ES2015+ (not TypeScript), modify `compile` command as following in `package.json` :

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

Delete `app/` folder

delete `app/` folder in the project and run `npm start` to restart project.

Breakpoint Debugging

It takes to you high speed by using ES2015+ in Node.js project. Because of some feature's lack, we need use Babel to compile our codes and run compiled code. It's terrible that debug will return code position in compiled file while I really want to know code position in source file.

The good news is that ThinkJS has been support breakpoint debug source code after 2.2.0 version. That means ThinkJS will tell you the exactly code position in source file when you have error.

Breakpoint debugging with `node-inspector`

Install node-inspector

You can global install node-inspector by typing command `npm install -g node-inspector`. Sometimes you should use `sudo` in *unix system.

Start node-inspector service

Start node-inspector service by using `node-inspector &`.

Start Node.js service

Start Node.js service by using `node --debug www/production.js`.

What difference is `--debug` parameter than before. `npm start` command has no this parameter so we can't use it.

Debug

Visit `http://127.0.0.1:8080/debug?port=5858`, it will display debugging page.

Then find compiled file in `app` folder, insert breakpoint in position you want. It's important that insert in `app` folder not in `src` source folder.

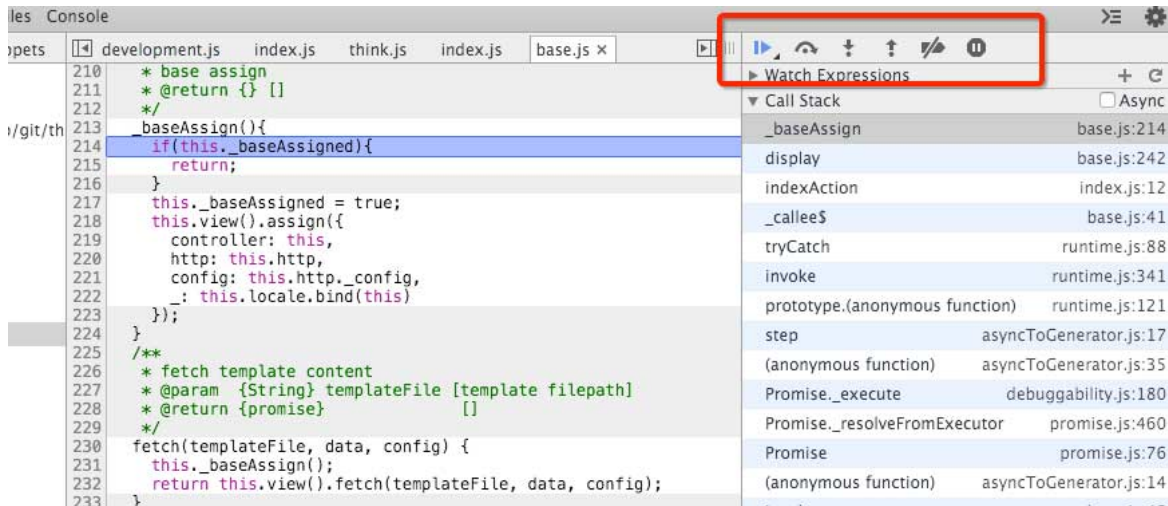
```
11 var _possibleConstructorReturn3 = _interopRequireDefault(_possibleConstructorReturn2);
12
13 var _inherits2 = require('babel-runtime/helpers/inherits');
14
15 var _inherits3 = _interopRequireDefault(_inherits2);
16
17 var _base = require('./base.js');
18
19 var _base2 = _interopRequireDefault(_base);
20
21 function _interopRequireDefault(obj) { return obj && obj.__esModule ? obj : { default: obj }; }
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, arguments));
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
46 */;
```

Open new tab and visit url you want debug. At this time page will be blocked, and come back to node-inspector page, you will see debug position has skipped to source file.

```
1 'use strict';
2
3 import Base from './base.js';
4
5 export default class extends Base {
6   /**
7    * index action
8    * @return {Promise} []
9    */
10   indexAction(){
11     //auto render template file index_index.html
12     return this.display();
13   }
14 }
```

这里已经是
src/ 下的源代码了

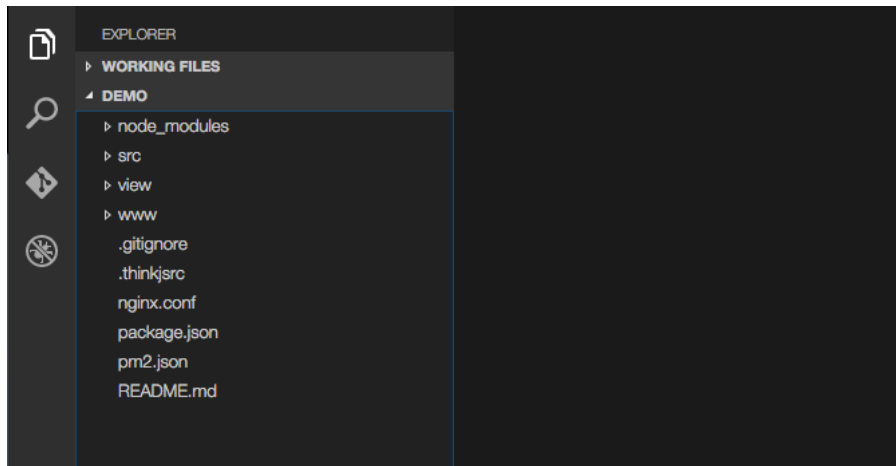
Then you can use this tools to debug.



Breakpoint debugging in VS Code

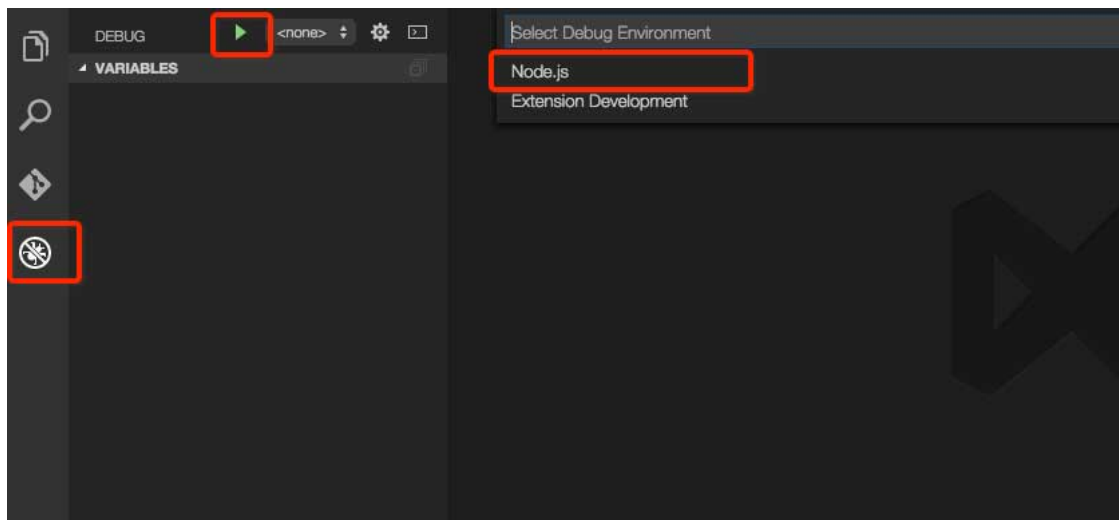
Open Project

click menu **File - Open** to open ThinkJS project, like:



Setting debug

Click debug menu on the left and debug button on menu. It will make you choose environment, and we choose Node.js.

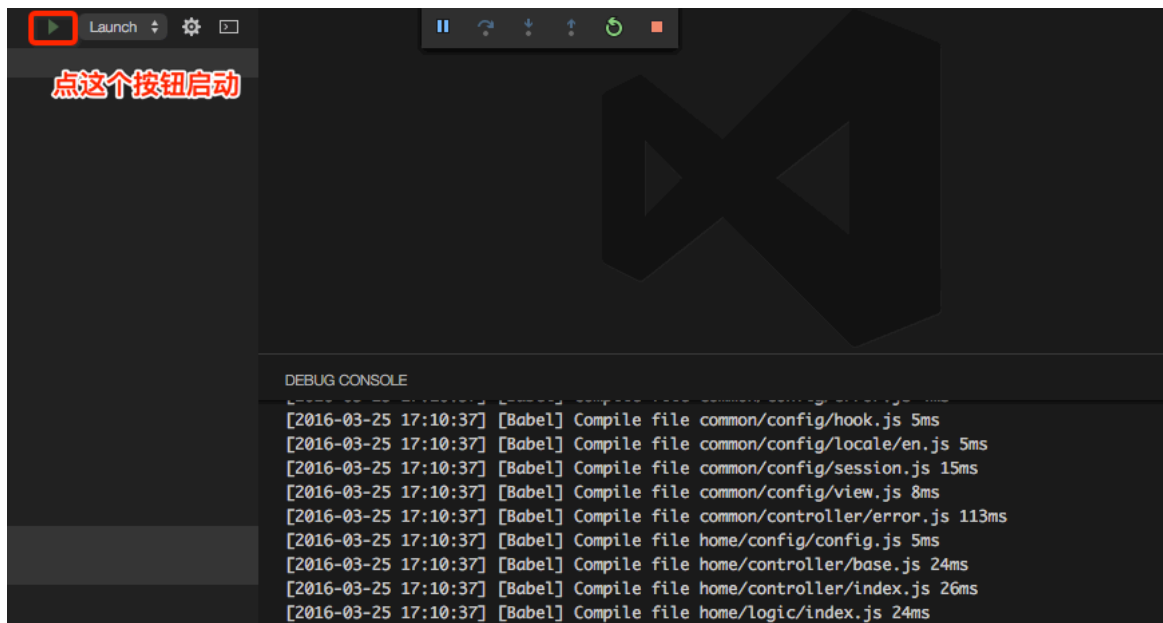


After that, **launch.json** will be created in project. Modify all **sourceMaps** value to **true** in this file.

```
launch.json .vscode
1  {
2    "version": "0.2.0",
3    "configurations": [
4      {
5        "name": "Launch",
6        "type": "node",
7        "request": "launch",
8        "program": "${workspaceRoot}/www/development.js",
9        "stopOnEntry": false,
10       "args": [],
11       "cwd": "${workspaceRoot}",
12       "preLaunchTask": null,
13       "runtimeExecutable": null,
14       "runtimeArgs": [
15         "--no-lazy"
16       ],
17       "env": {
18         "NODE_ENV": "development"
19       },
20       "externalConsole": false,
21       "sourceMaps": true,
22       "outDir": null
23     }
24   ]
25 }
```

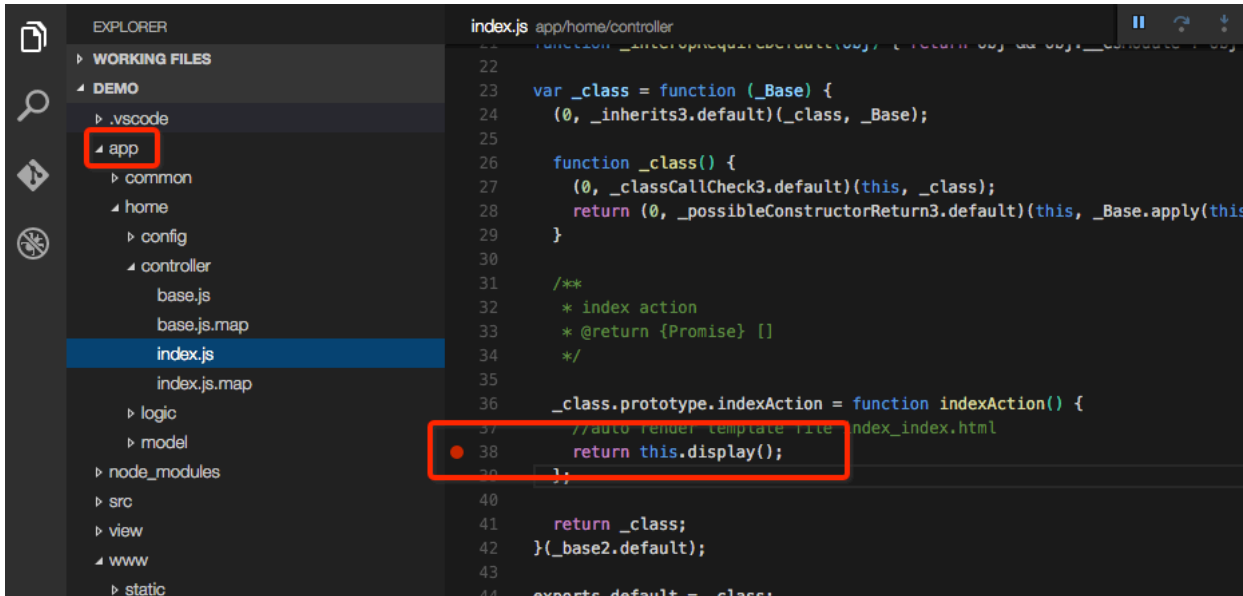
Start service

Click debug button to start service. If you have boot Node.js service, you need close it to avoid error of port using.

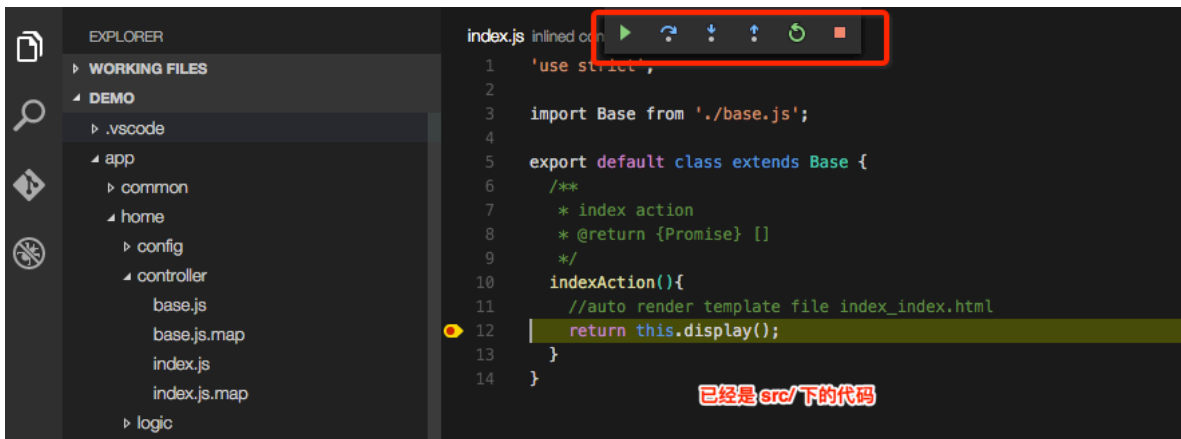


Start debug

Back to code view and insert breakpoint in `app` folder. It's important that insert in `app` folder not in `src` source folder.



Visit url you want debug, and we can use debug button to debug on the top.

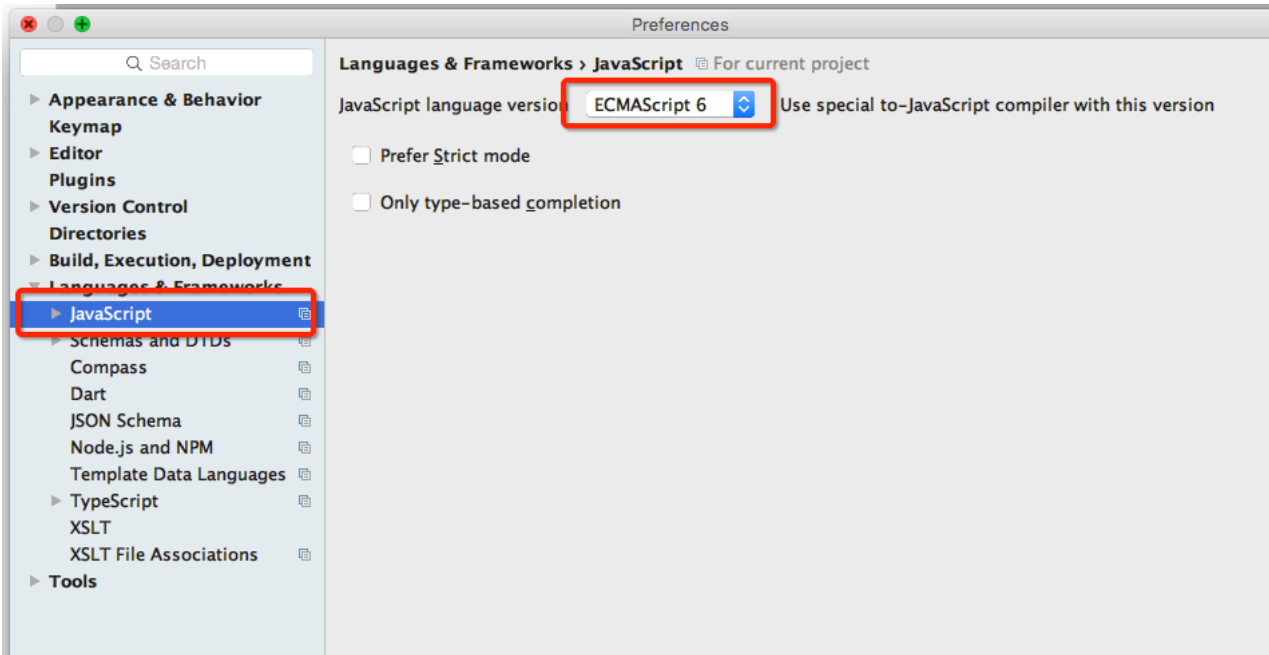


That's all for debugging ES2015+ in VS Code.

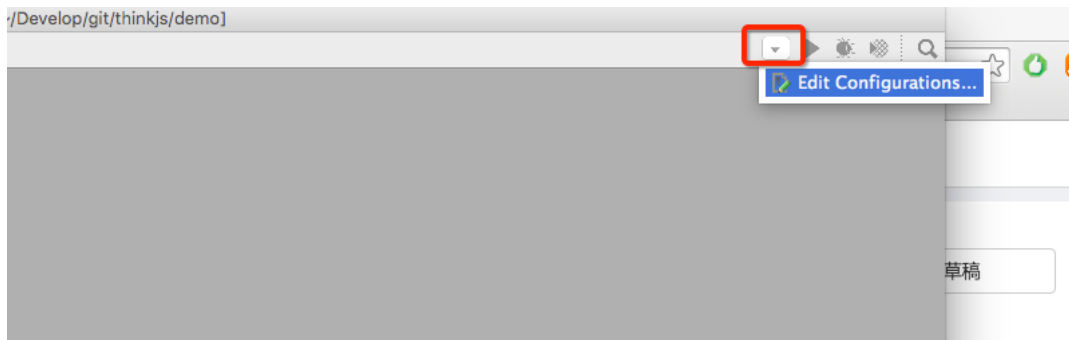
Breakpoint debugging in WebStorm

Configure WebStorm

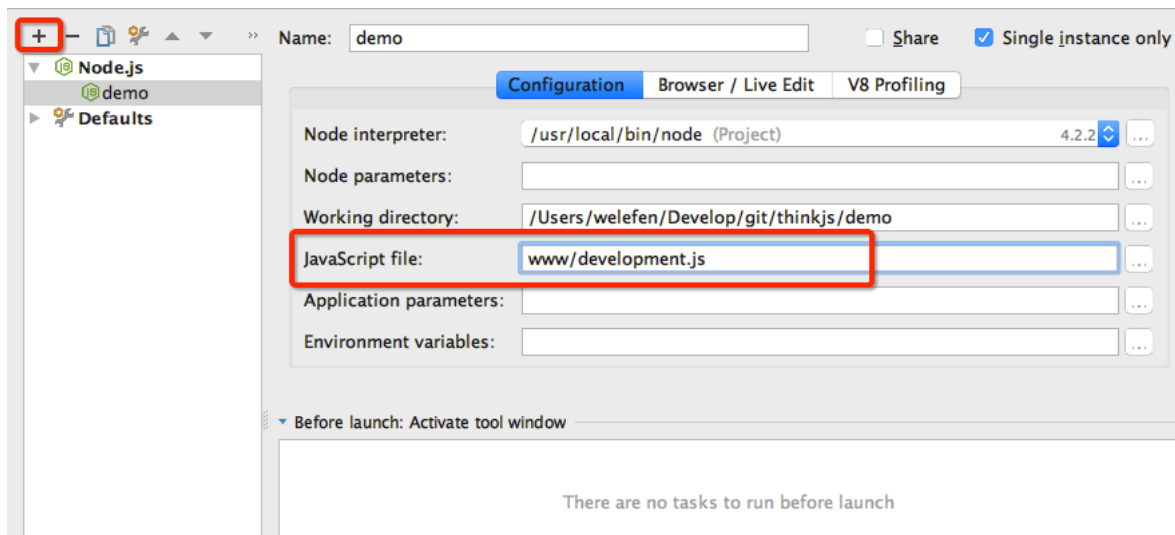
Import ThinkJS project into WebStorm and modify JavaScript version to ECMASCRIPT 6 in preference.



Click **Edit Configurations** and then create a new project and set project type to Node.js.

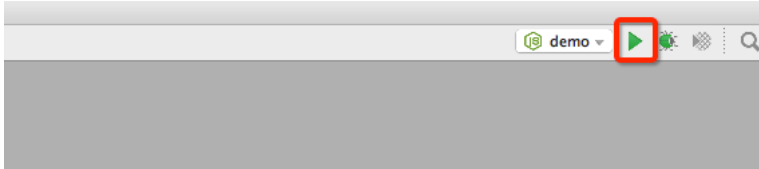


Insert `www/development.js` into **JavaScript File** on the right, or choose right file by using button.



Debug

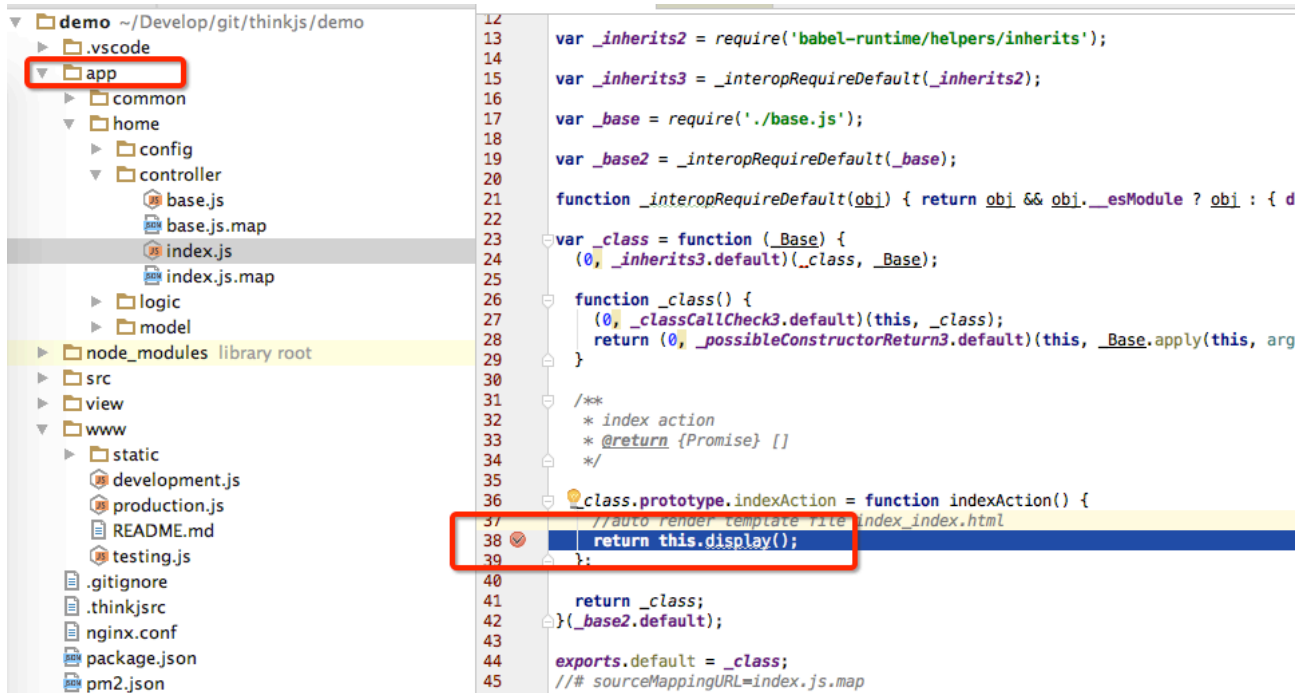
Click debug button will open Node.js service.



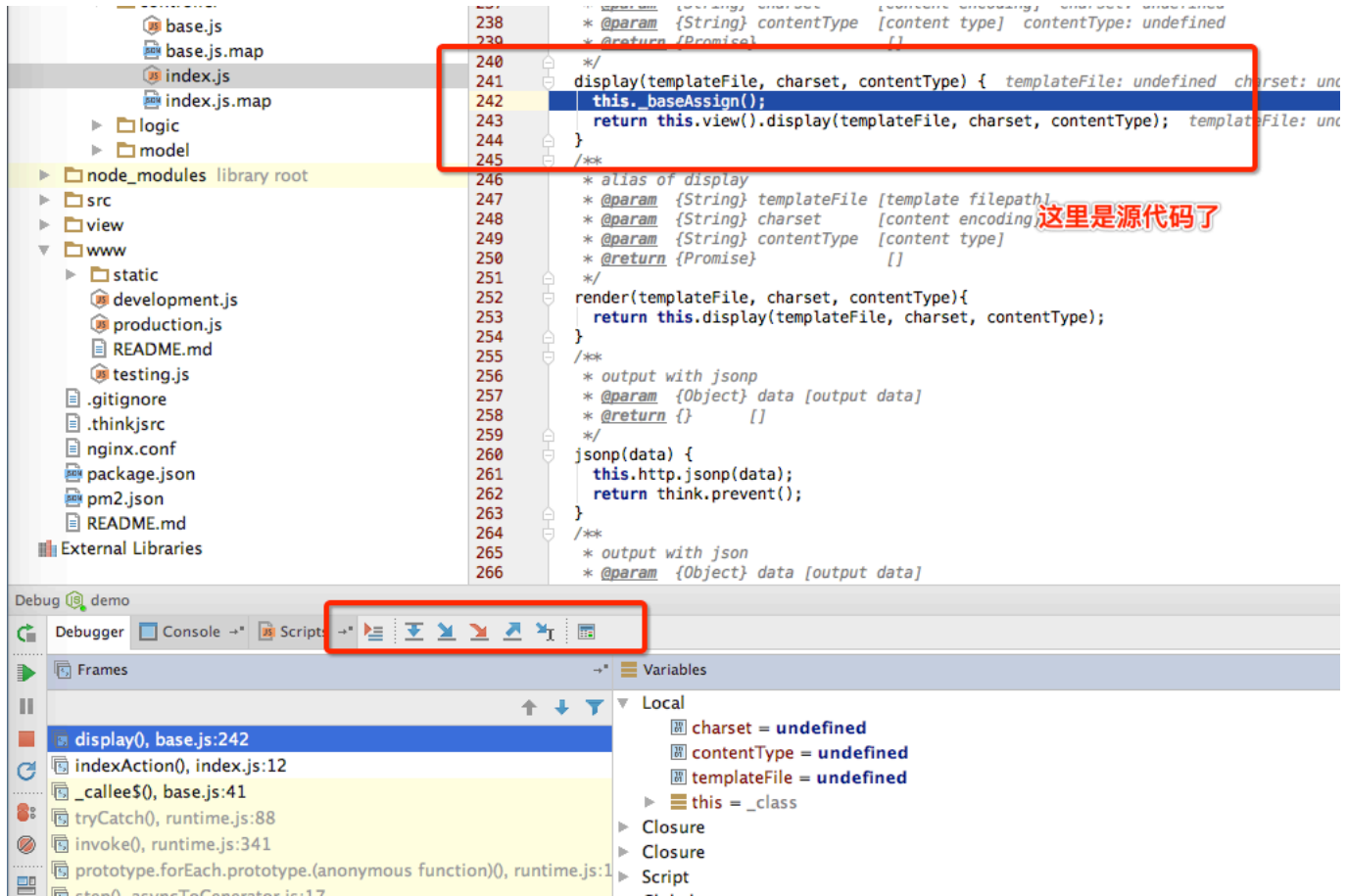
```
Run demo
[2016-03-25 17:30:14] [Babel] Compile file common/config/error.js 7ms
[2016-03-25 17:30:14] [Babel] Compile file common/config/hook.js 3ms
[2016-03-25 17:30:14] [Babel] Compile file common/config/locale/en.js 5ms
[2016-03-25 17:30:14] [Babel] Compile file common/config/session.js 14ms
[2016-03-25 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 Enviroment: development
```

If you have boot Node.js service, you need close it to avoid error of port using.

Insert breakpoint in `app` folder. It's important that insert in `app` folder not in `src` source folder.



Open browser and visit url you want debug. Back to WebStorm and click debug button to start debug in source code.



Common question

Why We Recommend You Use ES6/7 Grammar

ES6/7 support a mass of new features that bring us great convenience and efficiency. For example, we use ES6 `*/yield` and ES7 `async/await` feature to resolve async callback hell problem. And use arrow function to resolve `this` scope problem. Or use `class` grammar to resolve class inherit problem.

Although Node.js hasn't support all of those features, we can use them in Node.js stable environment in advance with the help of Babel. It's so good that we can enjoy the convenience and efficiency because of those new features.

Why Run `npm run watch-compile` Can't Stop the Process

Version 2.0.6 has removed this command, because this version has supported auto-compile feature, so all you need to do is to start the service by run `npm start`.

How to change server port

By default, Node.js server port is `8360`, you can change it in file `src/common/config/config.js`.

```
export default {
  port: 1234 //Change server port to 1234
}
```

Do We Need Restart Service After We Modified Something

Due to the working manner of Node.js, you must restart the service to make the modification to ta effect by default. It's so inconvenience to us. New version of ThinkJS supports auto update file mechanism to apply modification without restart.

Auto update may influence performance, so this feature turns on only in `development` mode. For online code, we advise you use `pm2` module.

How to Change the Structure of View Folder

By default, view files' path is `view/[module]/[controller]_[action].html`. In this example, controller and action was join by `_`. If you want change joiner to `/`, you can change configuration file `src/common/config/view.js` like this:


```
export default {
  file_depr: '/', //change joiner to /
}
```

How To Open Multiple Threads

For online code, you can improve its performance by make use of multi-core CPU to heighten concurrence computing.

You can open `src/common/config/env/production.js`, and add the following option to it:

```
export default {
  cluster_on: true //turn on cluster
}
```

How To Modify Request Timeout

The default timeout in ThinkJS is 120 seconds, you can modify it by open `src/common/config/config.js`, and add the following option:

```
export default {
  timeout: 30, // Change timeout to 30 seconds
}
```

How To Catch Exception

Using JS's `try/catch` can't catching exceptions come out of asynchronous code, but after using `async/await` you can use it:

```
export default class extends think.controller.base {
  async indexAction(){
    try{
      await this.getFromAPI1();
      await this.getFromAPI2();
      await this.getFromAPI3();
    }catch(err){
      //err.message is error message
      return this.fail(err.message);
    }
  }
}
```

Although you can catch the exception, but you can't know which code has triggered it.

In practice, this is not convenient because you often need to give error messages to users based on the errors.

At this time, you can check the specific error through single async request:

```
export default class extends think.controller.base {
  async indexAction(){
    //ignore this error
    await this.getFromAPI1().catch(() => {});
    //return false when exception occurs
    let result = await this.getFromAPI2().catch(() => false);
    if(result === false){
      return this.fail('API2 ERROR');
    }
  }
}
```

By returning different values for each exception, you can then return different error messages to users.

Ignore Exception

When using `async/await`, if Promise returned a rejected Promise, an exception will be thrown. If it is not important and you want to ignore it, you can make the catch method to return a resolve Promise:

```
export default class extends think.controller.base {
  async indexAction(){
    //returns undefined mean this exception will be ignore
    await this.getAPI().catch(() => {});
  }
}
```

PREVENTNEXTPROCESS

After calling some methods such as `success` , you may find an error message named `PREVENT_NEXT_PROCESS` in console. This error is introduced by ThinkJS to prevent from the running of subsequent processes. If you want to check `PREVENT_NEXT_PROCESS` in catch method, you can use `think.isPrevent` :

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

Another handling method: don't use `return` before method such as `success` , then there has no this error in `catch` .

Parallel Processing

While using `sync/await` , your code is executed serially. But mostly you want to execute parallelly to have higher execution efficiency. For this, you can use `Promise.all` to implement it.

```
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]);
  }
}
```

`p1` and `p2` are processed parallelly and then get both data by using `Promise.all` .

Output Images

If your projects need to output data like images and other file types, you can do it as following:

```
export default class extends think.controller.base {
  imageAction(){
    //image buffer data, you can read it locally or remotely
    let imageBuffer = new Buffer();
    this.type('image/png');
    this.end(imageBuffer);
  }
}
```

Using Different Configuration In Different Environments

Environments varies, the configuration may be vary too. For example, development environment and production environments should use different database configurations. You can modify `src/common/config/env/[env].js` to implement this. The option `[env]` has three default values: `development` , `testing` and `production` .

If you want to config production environment database, you can modify `src/common/config/env/production.js` :

```
export default {
  db: { //there has one level named db
    type: 'mysql',
    adapter: {
      mysql: {
        host: '',
        port: ''
      }
    }
  }
}
```

You can know more about configuration in [here](#).

To Extend Template In nunjucks

Because of `root_path` setting in nunjucks, you should using relative path when you want to extend template:

```
{% extends "../parent.html" %} //same level parent.html file
{% extends "../layout.html" %} //parent level layout.html file
```

To Allow Action Calls Only In CLI

Action can be executed by user requests or CLI calls. You can use `isCli` to judge when you want to allow Action call in CLI only:

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

Invokes Controller/Action/Model Across Modules

You may have some requests about calling function across modules when your projects is complicated.

Call Controllers

You can call controller in other modules by using `this.controller` and pass the second parameter in:

```
export default class extends think.controller.base {
  indexAction(){
    //get user controller instance in admin module
    let controllerInstance = this.controller('user', 'admin');
    //then you can use user controller function after getting instance
    let bar = controllerInstance.foo();
  }
  index2Action(){
    // or use this simple way
    let controllerInstance = this.controller('admin/user');
    let bar = controllerInstance.foo();
  }
}
```

Call Actions

You can call actions in other modules by using `this.action` :

```
export default class extends think.controller.base {
  async indexAction(){
    //get user controller instance in admin module
    let controllerInstance = this.controller('user', 'admin');
    //call test action in controller, it will also call `__before` and `__after` magic function automatically.
    let data = await this.action(controllerInstance, 'test')
  }
  async index2Action(){
    //you can also assign controller by string
    let data = await this.action('admin/user', 'test')
  }
}
```

tip : All actions will return Promise object, and it won't call logic when you call action alone.

Call Models

You can call models in other modules by using `this.model` :

```
export default class extends think.controller.base {
  indexAction(){
    //get user model instance in admin module
    let modelInstance1 = this.model('user', {}, 'admin');
    // or use this simple type
    let modelInstance2 = this.model('admin/user');
  }
}
```

How to request other api's data

It's common to request other api's data in our project. Of course you can use `http` module to implement it. But `http` module only provides some basic function. We recommend to use `request` module or `superagent` module:

```
import request from 'request';
/* Get API DATA */
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/47.0.2526.111 Safari/537.36'
    }
  }, (err, response, body) => {
    if(err){
      deferred.reject(err);
    }else{
      deferred.resolve(body);
    }
  });
}
```

To convert it to promise style, you should create an deferred object to resolve or reject in callback function. ThinkJS provides `think.promisify` function to convert function to promise quickly.

```
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/47.0.2526.111 Safari/537.36'
    }
  });
}
```

Normal in development environment and 502 error in production environment

Sometimes it's normal in development, but it comes 502 error in production by using pm2 and nginx. Think error is caused by abnormal with starting node service. You can see error info to check by `pm2 logs` command. Or you can close service and start service handly by `node www/production.js`, then watch error info.

CORS setting

Morden browsers support cors by setting some request header. ThinkJS can setting quickly like following:

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

JavaScript

More CORS info can watch <https://www.w3.org/TR/cors>.

If your project is REST API, you can add these codes in `__call` method.

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

JavaScript

current path is not thinkjs project.

When you use thinkjs command to create adapter or model, cli will return error like `current path is not thinkjs project` sometime.

When we use `thinkjs new project` command to create project, it will auto create a file named `.thinkjsrc` to store some description for this project. And creating adapter and model are depends on this file. If you miss this file, ThinkJS will return `current path is not thinkjs project` error.

To resolve this problem, you can copy `.thinkjsrc` file from anohter same project.

tip : `.thinkjsrc` file needs version control or will display this error someday.

Advanced

Module

ThinkJS supports a variety of programming modes when creating a project. By default, the new project is consist of modules, and has added the `common` and `home` modules automatically. Each module has itself a separate configuration, controller, view, model and other documents.

Modularization programming makes project structure much clearer. Such as a typical blog system can be divided into front and back modules in general.

Module List

Goes into `src/` directory, you can see a list of modules:

```
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 Module

`common` module is a universal module that will be commonly used by other modules of the project, it stored some common features, such as general configuration, runtime directory, startup files, error handling controllers.

NOTE : The controllers under the module does not respond to the user's request.

Default Module

Default module is the `home` module. Any requests that could not find corresponding module to process will be handed over to this module to process, so it is a catch all module.

If you want to modify the default module, open `src/common/config/config.js`, and modify the value of `default_module`:

```
//The default module's name is changed to blog
export default {
  default_module: 'blog'
}
```

JavaScript

Add Module

Add new module can be done by using `thinkjs` command.

In current project directory, execute `thinkjs module xxx`, you can create a module named `xxx`.

If the module's name already exists, you can not create it.

Disable Module

ThinkJS will automatically find and identify modules under the project and assume that all modules are available.

If you want to disable some modules, you can modify the configuration file `src/common/config/config.js`, add the following configuration:

```
export default {
  deny_module_list: ['xxx'] //Disable xxx module
}
```

JavaScript

Controller

Controller is a collection of same type operations, they respond to same type user requests.

The Definition of Controller

Creating a file `src/home/controller/article.js`, means that there's a controller called `article` in the `home` module, and the content of each controller is similar to the following:

```
'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();
  }
}
```

JavaScript

If you do not use ES6 syntax, then the content is similar to the following:

```
'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();
  }
});
```

JavaScript

NOTE: The `Base` above represents the definition of a base class, other classes inherit it, so that you can do some general thing in it.

Multi-levels Controller

For complicated projects, only have one level controller may not work well. Right now you can create multiple levels controller, the file `src/home/controller/group/article.js` for example will points to the second level controller `group/article`. Logic and View are the same.

Use Generator Function

You can easily use the generator function to handle asynchronous nesting problems in the `controller`.

The ES6 Way

```
'use strict';

import Base from './base.js';

export default class extends Base {
  /**
   * index action
   * @return {Promise} []
   */
  * indexAction(){
    let model = this.model('user');
    let data = yield model.select();
    return this.success(data);
  }
}
```

JavaScript

Dynamically Create Classes

```
'use strict';

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

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

JavaScript

Use async/await

With the Babel compilation, you can also use ES7's `async/await`.

The ES6 Way

```
'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);
  }
}
```

JavaScript

Dynamic Creation

```
'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 Method

The class in ES6 has a constructor method, but the classes that dynamically created do not, in order to perform the initialization uniformly, ThinkJS redefined it as `init`.

This method is automatically called when the class is instantiated, without manually call needed.

The ES6 Way

```
“js
'use strict';

import Base from './base.js';

export default class extends Base {
  init(http){
    super.init(http); //call super-class's `init` method
    ...
  }
}
”
```

Dynamically Create Classes

```
'use strict';

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

module.exports = think.controller(Base, {
  init: function(http){
    this.super('init', http); //call super-class's `init` method
    ...
  }
});
```

When using `init` method, don't forget to call call super-class's `init` method and make sure pass the `http` in.

Pre-Operation `__before`

ThinkJS supports pre-operation with the method called `__before`, it will be automatically called before a specific Action execution. If the pre-operation prevents subsequent code continuing to execute, it does not call the specific Action, so you can end request in advance.

The ES6 Way

```
“js
'use strict';

import Base from './base.js';

export default class extends Base {
  /**
   * Pre-Operation
   * @return {Promise} []
   */

```



```
__before(){
...
}
}
"
```

Action

An action represents an operation to be performed in response to a user request. Such as if the URL is `/home/article/detail`, the module is `/home`, the controller is `article`, and the action is `detail`, so the action to be executed is the `detailAction` method in the file `src/home/controller/article`.

```
JavaScript
'use strict';

import Base from './base.js';

export default class extends Base {
  /**
   * obtain detailed information
   * @return {Promise} []
   */
  detailAction(self){
    ...
  }
}
```

If the action name parsed contains `_`, it will automatically do the conversion, for the details of specific strategies of the conversion, see [Routing -> case](#).

Post-Operation `__after`

ThinkJS supports post-operation called `__after`, it will be executed after a specific action execution. If a specific action prevents subsequent code from continuing to execute, the post-operation will not be invoked.

No-operation `__call`

If one controller is found to exist after parsing the URL, but the action does not exist, it will attempt to call the `__call` magic method of the controller. This way, we can uniformly treat the missing actions.

```
JavaScript
'use strict';

import Base from './base.js';

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

Error Handling

If the URL does not exist, the current user has no permission to do some operations or there are other unusual requests, it will enter the error handling process. ThinkJS itself has built a complete error handling mechanism, for details see [extensions -> error](#).

Data Validation

Before using the user-submitted data in the controller, it is needed to verify its legitimacy. In order to reduce the logic complexity, ThinkJS provides a logic layer that is designed to handle data and permission validation and other related operations.

For more information, please see [Extended Functions -> Data Validation](#).

Variable Assignment and Template Rendering

Controller can do variable assignment and template rendering through `assign` and `display` methods, specific information can be found [here](#).

Model Instantiation

In controllers, you can quickly get an instance of a model by calling the `this.model` method.

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

More usage of `model` method can be found at [API -> think.http.base](#).

http Object

When a controller is instantiated, the `http` will be passed in. The `http` is a object that ThinkJS repacked for the `req` and `res`, it is not built in Node.js.

In Action, it can be obtained by `this.http`.

```
'use strict';

import Base from './base.js';

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

Details about the properties and methods of `http` object can be found at [API -> http](#).

REST API

Sometimes, the project has to provide some `REST` interfaces for third party to use, these interfaces are nothing more than the CRUD operations.

If you feel writing these operations by hand is very trouble, ThinkJS provides a REST Controller, that will automatically contains generic CRUD operations. If these actions do not satisfy your demand, it can also be customized. Specifically, [see here](#).

The this Scoping Issue

There are often many asynchronous operations in Node.js development, and the common approach is to use a callback function or `Promise`. These treatments will increase a level of scope, making it impossible to use `this` directly in the callback function, the simple approach to solve it is to define a variable at the top, `this` will be assigned to the variable, and then use the variable in the callback function. Such as:

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

Writing `var self = this` in each Action must be very trouble. To solve this problem, ThinkJS provides a parameter directly in Action, which is equivalent to `var self = this`, as follows:

```
module.exports = think.controller({
  // here, self is equivalent to 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);
    })
  }
})
```

Of course, the recommended and better solution is to use the Generator Function and Arrow Function of ES6.

Use 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);
  }
}
```

Use 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);
    })
  }
})
```

Output JSON

Many projects need provide interfaces that output data in JSON format, and there also must be a flag to indicate whether the interface is normal or not. If an exception occurs, the corresponding error message needs to be output together. The controller provides the `this.success` and `this.fail` methods to output interface data.

Output Normal JSON

The normal interface data can be output through `this.success` method, such as:

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

In this example, the output is `{errno: 0, errmsg: "", data: {"name": "thinkjs"}}`, the client can determine whether there is an exception with the current interface through `errno` is 0 or not.

Output JSON Contained the Error Message

Interface data contained error messages may output by the `this.fail` method, such as:

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

In this example, the output is `{errno: 1000, errmsg: "connect error"}`. When clients found `errno` is greater than zero, then it know there are exceptions with the current interface, so it can in turn to get specific error information through `errmsg`.

Configure Error Number and Error Message

It's recommended to configurate the error numbers and error messages in one place, then as long as specify error number when outputting, error information based on the error number will be automatically read out.

Error messages support internationalization, and the configuration is in the file `src/common/config/locale/[lang].js`. Such as:

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

Whit the above configuration, performing `this.fail(10001)` will automatically get corresponding error message, "get data error" in this case.

Friendly Error Number

Although it can output the correct error number and error message when performing the `this.fail(10001)`, but we can not intuitively see what error message corresponding it.

We recommend you to configure the keys using uppercase strings, and the value is an array with the error number and error message as its elements. Such as:

```
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
}
```

JavaScript

This way, when you calling `this.fail('GETDATA ERROR')`, you will automatically get the corresponding error number and error message.

Format Configuration

The keys of the default error number and error message are `errno` and `errmsg` respectively. If needed, you can modify the configuration file `src/common/config/error.js` to reset them.

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

JavaScript

Output The JSON That Does Not Contain The Error Message

If you don't want the outputed JSON data contained `errno` and `errmsg`, you can output JSON by `this.json` method. Such as:

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

JavaScript

Common Functions

Get GET Parameters

You can obtain GET parameters through the `get` method, such as:

```
export default class extends think.controller.base {
  indexAction(){
    let name = this.get('name');
    let allParams = this.get(); // obtain all GET parameters
  }
}
```

JavaScript

If the parameter does not exist, the value will be an empty string.

Get POST Parameters

You can obtain POST parameters through the `post` method, such as:

```
export default class extends think.controller.base {
  indexAction(){
    let name = this.post('name');
    let allParams = this.post(); // obtain all POST parameters
  }
}
```

JavaScript

If the parameter does not exist, then the value will be an empty string.

Get Uploaded Files

You can obtain the uploaded files by using `file` methods, such as:

```
export default class extends think.controller.base {
  indexAction(){
    let file = this.file('image');
    let allFiles = this.file(); // obtain all uploaded files
  }
}
```

JavaScript

The return value is an object that contains the following attributes:

```
{
  fieldName: 'file', // form field's name
  originalFilename: filename, // original file's name
  path: filepath, // file's temporary path, the file will be deleted when request end
  size: 1000 // file size
}
```

If the file does not exist, then the value will be an empty object `{}`.

JSONP Format Data Output

You can output data in JSONP format by `this.jsonp` method, the name of the callback request parameter defaults to `callback`. If you need to modify its name, you can modifying the configuration `callback_name`.

More Methods

- `isGet()` Used for check is it currently a GET request
- `isPost()` Used for check is it currently a POST request
- `isAjax()` Used for check is it currently a AJAX request
- `ip()` Used for get requesting user's ip
- `redirect(url)` Used for jump to an URL
- `write(data)` Output data, automatically call `JSON.stringify`
- `end(data)` End the current HTTP request
- `json(data)` Output JSON data, automatically send content-type Headers that related to JSON
- `jsonp(data)` Output JSONP data, the request parameter name defaults to the `callback`
- `success(data)` Output success JSON data with error info, such as `{errno: 0, errmsg: "", data: data}`
- `fail(errno, errmsg, data)` Output error JSON data with error info, such as `{errno: errno_value, errmsg: string, data: data}`
- `download(file)` Used for download a file
- `assign(name, value)` Set a variable so that we can use it in the template
- `display()` Output a template
- `fetch()` Rendering the template and get the result
- `cookie(name, value)` Get or set the cookie
- `session(name, value)` Get or set the session
- `header(name, value)` Get or set the header
- `action(name, data)` Call other Controller's method, included those in other modules
- `model(name, options)` Initiated a model instance

A complete list of methods please see [API -> Controller](#).

View

View is template, its default root directory is `view/`.

View Files

The default naming rule of view file is `module/controller_operation.html`.

For URL `home/article/detail`, after parsed, the module is `home`, the controller is `article`, the operation is `detail`, then the corresponding view file is `home/article_detail.html`.

View Configuration

Default view configuration is as follows, you can modify it in the configuration file `src/common/config/view.js`:

```
export default {
  type: 'ejs', // template engine
  content_type: 'text/html', // the Content-Type send with outputed template
  file_ext: '.html', // the extension name
  file_depr: '_', // the seperator between controller and action
  root_path: think.ROOT_PATH + '/view', // the root directory of view files
  prerender: undefined, // whether execution custom process logic before rendering template
  adapter: { // the configuration options needed by template engine
    ejs: {}, // the extra configuration options when using ejs as template engine
    nunjucks: {} // the extra configuration options when using nunjucks as template engine
  }
};
```

Note: Since `2.0.6` version, `options` configuration item was removed, and `adapter` is the replacement.

The default root directory of view is `view/`. If you want each module to own a separate view directory, just reset `root_path` configuration to empty.

Modifying Separator

The separator between the default controller and operation is `_`, so the file name is similar to `index_index.html`. If you want the controller to be as a layer directory, such as: `index/index.html`, you can modify the separator to `/`.

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

JavaScript

Modify The Template Engine

If you want to modify some configurations of the template engines, you can modify the corresponding field of configuration. Such as:

```
export default {
  options: {
    delimiter: '&' // modify as <& and &>
  }
}
```

JavaScript

Template Engine

ThinkJS support `ejs`, `jade`, `swig` and `nunjucks` as template engine, and the default template engine is `ejs`, you can modify the default template engine based on need.

ejs

Delimiter

The default delimiters of `ejs` are `<%` and `%>`. If you want to change them, you can modify the `options` field of the configuration, such as:

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

JavaScript

Variable Output

- Escape output `<%= data.name %>`
- Not escape output `<%- data.name %>`
- Comment `<## data.name %>`

Conditional

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

Loop

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

Filter

The new version of `ejs` no longer support the filter function, and if you need some filter functions, you can define some global function in `src/common/bootstrap/`, then you can use these functions directly in the template.

Reference File

`ejs` does not support template inheritance. But it can make a public template as an independent file, and then introduce it using `include` directive, such as:

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

Note : Variable that used by ejs template needs to be assigned in the controller, otherwise it will produce an error.

More ejs document please see [here](#).

nunjucks

Nunjucks is a another template engine ThinkJS supported, it similar to the jinja2, whose functions is unusually powerful, if your project is complex, we suggest you use it.

Delimiter

Block-level delimiters are `{% %}`, variable delimiters are `{{ and }}`, comment delimiters are `<# and #>`. Such as:

```
{{ username }}

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

Markup

Variable Output

You can use `{{username}}` to output variables, the default output variables will automatically be escaped, if you don't want to escape variables, use `{{username | safe}}` instead.

Template Inheritance

The parent template:

```
{% 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>
```

Markup

The child template:

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

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

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

Markup

Conditional

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

Markup

Loop

```
<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>
```

For complete documentation please see [here](#).

jade

The documentation of jade template can be found [here](#).

swig

The documentation of swig template can be found [here](#).

Add Filters and Other Functions

`Swig`, `nunjucks` and many other template engines support adding filters, and other functions, it can be done by finding the corresponding adapter in the template configuration file `src/common/config/view.js` and adding `prerender` configuration. Such as:

```
export default {
  prerender: function(nunjucks, env){
    // add a filter, then you can use it in the template
    env.addFilter('filter_foo', function(){

    })
  }
}
```

Note : This function is introduced since ThinkJS `2.0.5` .

Extend The Template Engine

Template engine is implemented by Adapter. If your project needs to use other template engines, it can be extended through Adapter, more details please see [here](#).

Variable Assignment

You can assigning value to template variable by using `assign` method in the controller.

Assignment of Single Variable

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

Assignment of Multiple Variables

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

Get The Values

You can get assigned values by `assign` after variable assignment. Such as:


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

Template Rendering

You can render the template by call the `display` method. If no specific template file path was passed, ThinkJS will search on for you automatically. Such as:

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

You could also specify a specific template file for rendering, more about the `display` method's using please see [here](#).

Get Rendered Content

If you don't want to outputting template, and only want to get the rendered content, you can use the `fetch` method.

The ES6 Way

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

Dynamically Creation

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

More details about the `fetch` method please see [here](#).

Internationalization

After starting the internationalization, the view path will has an extra layer of internationalization of directory. Such as specific view path becomes into the `view/zh-cn/home/index_index.html`, and `zh-cn` means language.

More about how to implementing internationalization, please see [extensions -> internationalization](#).

Multiple Themes

After setting the multiple theme, view path will be much more than a layer theme directory. Such as specific view path will becomes into the `view/default/home/index_index.html`, the `default` is the theme name.

You can set the current theme by `http.theme` method, setting theme is usually done by middleware.

More information on middleware please see [extensions - middleware](#).

Default Template Variables

In order to get some common variables easily in the template, ThinkJS will automatically register `http`, `controller`, `config` and other variables in the template, and these variables can be read directly in the template.

The following code examples are based on `ejs`, if you are using other template engine, you need to modify it to use the correct syntax.

http

In the template, the properties and methods under `http` object can be used directly.

controller

In the template, the properties and methods under `controller` object can be used directly.

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

JavaScript

Add property `navType` to the current controller in the Action, then you can use `controller.navType` in template directly.

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

config

You can get the configuration in the template through the `config` object, such as:

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

<%}%>
```

Get Localization Using `_`

In templates, you can obtain the value of the corresponding localization by `_`, these values are defined in the `src/common/config/locales/[lang].js`.

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

More information on internationalization please see [here](#).

Configuration

ThinkJS provides a flexible configuration mechanism, it can use different configuration in different modules and project environments, and these configurations will take effective after service started.

Note: Do not set the private value of an http request in the configuration, because other http setting may overriding these values.

The Project Module

The projects that created default by ThinkJS are divided according to the module, you can define different configuration under each module. General configuration can be defined under `common` modules, other modules will inherit the `common` configuration.

Project Environment

ThinkJS default support three kinds of project environments, it can be configured according to the different environment, in order to meet the needs of the different situations of configuration.

- `development` development
- `testing` testing
- `production` production

It can also be extended to other environment in project, which kind of environment to use at present can be set in the [entrance file](#), and set the `env` value.

Defining Configuration Files

config/config.js

For some basic configuration, such as:

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

config/[name].js

For a specific independent function configuration, such as `db.js` is the database configuration, `redis` is redis configuration.

```
// db.js
export default {
  type: 'mysql',
  adapter: {
    mysql: {
      host: '127.0.0.1', // database host
      port: '', // datagbase port
      database: '', // database name
      user: '', // user name
      ...
    }
  }
};
```

config/env/[mode].js

Differentiation configuration in different project environment, such as `env/development.js`, `env/testing.js`, `env/production.js`.

```
// config/env/development.js
export default {
  port: 7777,
  db: {
    type: 'mysql',
    adapter: {
      mysql: {
        host: '127.0.0.1',
        port: '',
        database: '',
        user: '',
        ...
      }
    }
  }
};
```

Note: The differences of different environments generally is not too much, so we defined them in a single file. At this time, if you want to modify an independent function configuration, you need to add a key corresponding to the independent function. Such as you need to add the the name of the `db` corresponding to the database when modifying the database configuration, as shown above.

config/locale/[lang].js

International language pack configuration, such as `locale/en.js`, `locale/zh-cn.js`.

Configuration format uses the form of `key: value`, and the `key` is case-insensitive.

Loading Configuration Files

ThinkJS supports multiple levels of the configuration file, it reads in the following order:

default configuration of the framework -> framework configuration under project mode -> project common configuration -> common configuration under

Reading Configuration

Using config

In Controller, Logic, Middleware, you can using `this.config`. Such as:

```
let db = this.config('db'); // reading all of the configurations about db
let host = this.config('db.host'); // reading the host configuration about the host of db
```

Using http.config

`http` objects also have the `config` method used to obtain the relevant configuration, such as:

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

Reading Configuration From Other Places

In other places, we can read the relevant configuration through `think.config`:

```
let db = think.config('db'); // reading the configuration about db under the common configuration
let db1 = think.config('db', undefined, 'home'); // get the db configuration under the home module
```

Note: Before parsing route, we can not get the general module configuration through the `config` method or `http.config` method, so the configuration which is used before route parsing must be defined in the general module.

The Default Configuration

env

Project configuration mode, the `config/env/development.js`.

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

The `config/env/testing.js` and `config/env/production.js` have no default configuration.

locale

International language pack configuration, the default configuration is as follows:

```
// 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 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 chars',
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

The basic configuration, [config/config.js](#) .

```

export default {
  port: 8360, // the port server is listening
  host: '', // host
  encoding: 'utf-8', // encoding
  pathname_prefix: '', // the prefix that will be remove when parsing routers
  pathname_suffix: '.html', // the suffix that will be remove when parsing routers

  hook_on: true, // turns hook on
  cluster_on: false, //turns cluster on

  timeout: 120, //120 seconds
  auto_reload: false, //auto reload the changed files

  resource_on: true, // turns resource on
  resource_reg: /^(static\/[^\v]+\.(?!js|html)\w+)$/ , //

  route_on: true, //turns routing on

  log_error: true, //log error
  log_request: false, //log http request

  create_server: undefined, //create server
  output_content: undefined, //output content function
  deny_module_list: [], //deny module list
  default_module: 'home', //default module
  default_controller: 'index', //default controller
  default_action: 'index', //default action
  callback_name: 'callback', //callback name for JSONP request
  json_content_type: 'application/json', //content-type for output json data
}

```

cache

Cache configuration, `config/cache.js` .

```

export default {
  type: 'file', //cache type
  timeout: 6 * 3600,
  adapter: {
    file: {
      path: think.RUNTIME_PATH + '/cache', // the folder to store the caching content in file caching type
      path_depth: 2, // the levels of subfolders
      file_ext: '.json' // the suffix of the caching files
    },
    redis: {
      prefix: 'thinkjs_', //cache key prefix
    }
  }
};

```

cookie

Cookie configuration, `config/cookie.js` .

```

export default {
  domain: '', // cookie domain
  path: '/', // cookie path
  httponly: false, //httponly
  secure: false, //secure
  timeout: 0 //cookie time
};

```

db

Database configuration, `config/db.js` .

```
export default {
  type: 'mysql', //database type
  log_sql: true, //log sql
  log_connect: true, // log database connection
  adapter: {
    mysql: {
      host: '127.0.0.1', //database host
      port: '', //database port
      database: '', //database name
      user: '', //database account
      password: '', //database account password
      prefix: 'think_', //table prefix
      encoding: 'utf8', //table encoding
      nums_per_page: 10, //nums per page
    }
  }
};
```

error

The error information configuration, `config/error.js` .

```
export default {
  key: 'errno', //error number
  msg: 'errmsg', //error message
  value: 1000 //default errno
};
```

gc

The cache, the session, and garbage disposal configuration, `config/gc.js` .

```
export default {
  on: true, //turn on gc
  interval: 3600, // gc interval
  filter: function(){ //
    let hour = (new Date()).getHours();
    if(hour === 4){
      return true;
    }
  }
};
```

hook

Hook configuration, `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

The post request configuration, `config/post.js` .

JavaScript

```
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 configuration, `config/redis.js` .

JavaScript

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

memcache

memcache configuration, `config/memcache.js` .

JavaScript

```
export default {
  host: '127.0.0.1', //memcache host
  port: 11211,
  username: '', //
  password: '',
  timeout: 0, //cache timeout
  log_connect: true
};
```

session

Session configuration, `config/session.js` .

JavaScript

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

view

View configuration, `config/view.js` .

JavaScript

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

    }
  }
};
```

websocket

Websocket configuration, `config/websocket.js` .

```
export default {
  on: false, //use websocket
  type: 'socket.io', //websocket type
  allow_origin: '',
  sub_protocol: '',
  adapter: undefined,
  path: '', //url path for websocket
  messages: {
    // open: 'home/websocket/open',
  }
};
```

JavaScript

The Extension Configuration

Projects configuration can be extended according to the need, extending configuration only need to set up the corresponding files in `src/common/config/` , such as:

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

JavaScript

So you can obtain the corresponding configuration through `think.config('foo')` .

Route

When an user visit an URL, eventually which module, controller and operation will be performed is decided by the parsed route.

ThinkJS provides a flexible route mechanism, in addition to the default resolution, it also support a variety forms of custom route, let the URLs more simple and friendly.

Resolving URL to pathname

When a user accesses to the service, the server first of all, will get a full URL, such as `http://www.thinkjs.org/zh-cn/doc/2.0/route.html` .

The pathname resolved by URL is `/zh-cn/doc/2.0/route.html` .

pathname Filter

Sometimes for the sake of SEO or other reasons, the URL will be added a few more things. Say the current page is a dynamic page, but the URL ended with suffix `.html` is more friendly to search engines. But the suffix is useless in the subsequent route resolution, it needs to be removed.

ThinkJS offer the following configuration, it can remove the prefix and postfix content of `pathname` :

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

JavaScript

Above configuration can be modified in the `src/common/config/config.js` .

When filtering, the `/` before and after pathname will be removed, and this logic is not affected by the configuration above. After filtering the pathname, the clean pathname you get is `zh-cn/doc/2.0/route` .

Note : If the URL is `http://www.thinkjs.org/` , then the clean pathname you get is an empty string.

Subdomain Deployment

For complex projects, we may want to deploy different function under the different domain, but the code is still in a single project. For example thought the domain name `admin.example.com` was deployed to host the adminstration functions, we still hope that it can be mapped to the `admin` module.

ThinkJS provides the following configuration that it can undertake subdomain deployment, the configuration can be set in the `config/config.js` :

```
export default {
  subdomain: {
    admin: 'admin', // means map admin.example.com to the admin module
    ...
  }
}
```

JavaScript

If the filtered pathname is `group/detail`, and the URL hit the subdomain `admin.example.com`, the pathname will become to `admin/group/detail` internally.

Routing Identification

Routing Resolving

By default, routing identification identify the filtered pathname according to the

`module/controller/action/parameter1/value-of-parameter1/parameter2/value-of-parameter2`. For example, if the pathname is `admin/group/detail`, the results of identification is:

- module is `admin`
- controller is `group`
- action is `detail`, the corresponding method is `detailAction`

If the project doesn't have `admin` or the module is disabled, then the results of identification is:

- module is the default module `home`
- controller is `admin`
- action is `group`, the corresponding method is `groupAction`
- parameter is `{detail: ''}`

Case Transformation

After route identification, `module`, `controller` and the `action` value will automatically convert to lowercase. If there are `_` in the Action value, it will do some transformation, for example the value of Controller is `index` after identification, the Action value is `user_add`, then the corresponding Action method called `userAddAction`, but the template name is still `index_user_add.html`.

The Default Route

Once there is no corresponding value when parsing the pathname, the default values are used. The module's default value is `home`, the controller's default value is `index`, and the action's default value is `index`.

These values can be modified through the following configuration, in the configuration file `src/common/config/config.js`:

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

JavaScript

Custom Route

Although the default route looks clear, it's also simple to parse, but looks not enough concise.

Sometimes we need more compact routes scheme, in this case we need to use a custom route. Such as the detail page of an article, the default route might be `article/detail/id/10`, but the URL we wanted is `article/10`.

Enable The Custom Configuration

To enable the custom route, open `src/common/config/config.js`, and set `route_on` as `true`.

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

JavaScript

Route Rules

After enabling the custom route, the next thing is to define the route rules in the route configuration file `src/common/config/route.js`, the format are as following:

```
export default [
  ["rule1", "the-pathname-you-wanted-to-be-identified-to"],
  ["rule2", {
    get: "the-pathname-you-wanted-to-be-identified-to-when-GET",
    post: "the-pathname-you-wanted-to-be-identified-to-when-POST"
  }]
];
```

JavaScript

Note: Each rule is an array.(The reason why we do not use object literal is regular expressions cannot be used as object's key.)

Identify Order

The match rule of custom route is : matching one by one from the front to end, if hit one rule, it will not match forward.

ThinkJS supports three types of custom route: regular route, rules route and static route.

Regular Route

Regular route using regular expressions to define routes, relying on the powerful regular expression, it can define very flexible route rules.

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

JavaScript

The above regular expression will match pathname like `article/10`, the resolved pathname will be `home/article/detail`, and the value of parameter `id` then can obtain through `this.get` method in the controller.

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

JavaScript

If regular route contains multiple child catch groups, then can obtain the corresponding values by `:1`, `:2`, `:3`:

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

JavaScript

Rules Route

Rules route is a way of string matching, but supports some dynamic values. Such as:

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

JavaScript

If URL is `http://www.example.com/group/2015/10`, then it will hit the rule, the pathname we get will be `home/group/list`, at the same time, it will add two parameters `year` and `month`, and they can be gotten through `this.get` method in the controller.

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

JavaScript

Static Route

Static route is a way of pure string exactly match, its writing and identification are very simple, of course the function is relatively weaker.

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

JavaScript

If the URL is `http://www.example.com/list`, then the pathname is replaced with `home/article/list`.

Optimizing The Route Performance

Above has said that the custom route is an array, each item of the array is a specific route rule, and it matches one by one from the front to end when matching. If the route table is large, there may be a performance issue.

In order to avoid performance issues, ThinkJS provides a more efficient way to custom route, configuring route according to the module. This way, the route configuration format is

slightly different from the above.

common/config/route.js

This time, the route configuration in general module no longer define specific route rules, but configures which rules hit which module. Such as:

```
export default {
  admin: {
    reg: /^admin/ // hit admin module
  },
  home: { // home module as default
  }
}
```

JavaScript

admin/config/route.js

The admin module configures specific route rules belongs to it.

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

JavaScript

Assuming the URL is `http://www.example.com/admin/api`, then the parsed pathname is `admin/api`, it will hit the `admin` module when matching the rules in the `common`, and then match the route rules one by one under the `admin` module. This way, it can greatly reduce the number of route rules need to match every time, makes route more efficient.

Custom Home Page Router

By default, home page router execute the `indexAction` of index controller. If you want custom home router in your projects, you will find it doesn't work by writing

```
['', 'index/list'] .
```

In consideration of performance, ThinkJS don't support customizing of home router. Because of the traffic of Home page are usually large, and there are less chances to modify the configuration of its router. If we support the customizing of Home page router, we would run through all the customized routers for every request to home page, that will be a huge wasting on performance.

s

Model

Model Introduction

During project development, you always need to manipulate data tables, thus involves CRUD operations. The model is just an incapsulation in order to facilitate database manipulation. A model maps to a data table in database.

ThinkJS currently supports MySQL, MongoDB and SQLite.

Create Model

You can use command `thinkjs model [name]` in project directory to create model:

```
thinkjs model user;
```

Bash

This will create file `src/common/model/user.js`.

Model file will be placed in `common` module by default, if you want to use other modules, you need to specify module name when creating:

```
thinkjs model home/user
```

Bash

Note: Model file is not required, you don't need to create it when there is no custom method, in this case the instance of base class will be used.

Model Properties

model.pk

Default primary key is `id`, and `_id` in MongoDB.

model.schema

The table fields definition, will be readed from database by default, and the result like this:

```
{
  id: {
    name: 'id',
    type: 'int', //type
    required: true, //is it required
    primary: true, //is it primary
    unique: true, //is it unique
    auto_increment: true //is it autoincrement?
  }
}
```

JavaScript

Additional properties can be added in the model, such as default value and does it read only:

```
export default class extends think.model.base {
  /**
   * Table fields definition
   * @type {Object}
   */
  schema = {
    view_nums: { //read count
      default: 0 //default is 0
    },
    fullname: { //fullname
      default: function() { //combination of first_name and last_name, can not use arrows function
        return this.first_name + this.last_name;
      }
    }
    create_time: { //create time
      default: () => { //get current time
        return moment().format('YYYY-MM-DD HH:mm:ss');
      },
      readonly: true //read only, you can't change it after added
    }
  }
}
```

JavaScript

`default` is only valid in adding, and `readonly` is only valid in updating.

You can see [API-> Model](#) to get more details.

Model Instantiation

Model instantiation is different depend on use cases. If current class has `model` method, it will be used directly to instantiate:

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

JavaScript

You can also use `think.model` to instantiate:

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

JavaScript

You need to pass in configuration when using `think.model`.

Chaining Invoke

Model provides many chaining invoke methods(like JQuery does) which can facilitate data manipulation. Chaining invoke is implemented by returning `this`:

```
export default class extends think.model.base {
  /**
   * get list data
   */
  * getList(){
    let data = yield this.field("title, content").where({
      id: [ ">", 100 ]
    }).order("id DESC").select();
    ...
  }
}
```

Model supports chaining invoke the following methods:

- `where` , define query or update conditions
- `table` , define table name
- `alias` , define alias of current table
- `data` , assign value before creating or updating data
- `field` , define field for querying, support exclude
- `order` , sort results
- `limit` , limit results number
- `page` , results pagination, will be translated to limit when generate sql commands
- `group` , querying group support
- `having` , querying having support
- `join` , querying join support
- `union` , querying union support
- `distinct` , querying distinct support
- `cache` , query cache

This doc stays at https://github.com/75team/www.thinkjs.org/tree/master/view/zh-cn/doc/2.0/model_intro.md.

Config Database

Config Database

Here is the model configuration, you can modify it in `src/common/config/db.js` :

```
```js
export default {
 type: 'mysql',
 log_sql: true,
 log_connect: true,
 adapter: {
 mysql: {
 host: '127.0.0.1',
 port: '',
 database: '', //database name
 user: '', //database account
 password: '', //database account password
 prefix: 'think_',
 encoding: 'utf8'
 },
 mongo: {

 }
}
};
```

You can use different configuration in different module, just config `src/[module]/config/db.js`.

## Define Data Table

By default, model name maps to table name. If your table's prefix is `think_` , `user` model will map to table `think_user` and `user_group` model will map to table `think_user_group` .

You can modify these by config the following two properties:

- `tablePrefix` table prefix
- `tableName` table name without prefix

## ES6 Way

```
export default class extends think.model.base {
 init(...args){
 super.init(...args);
 this.tablePrefix = ""; //set the prefix to blank
 this.tableName = "user2"; //set the data table name to user2
 }
}
```

## Dynamic Class Creation

```
module.exports = think.model({
 tablePrefix: "", //use property to set prefix and table name
 tableName: "user2",
 init: function(){
 this.super("init", arguments);
 }
})
```

## Modify Primary Key

Model fault primary key is `id`, if it is not the primary key setting in data table, you need to reset it:

```
export default class extends think.model.base {
 init(...args){
 super.init(...args);
 this.pk = "user_id"; // set primary key as user_id
 }
}
```

Operations like `count`, `sum`, `min` and `max` all will use primary key, when you need these operations, please reset the primary key.

## Distributed Database

In large-scale systems, there are often multiple databases to separate the reading and writing operations. ThinkJS supports custom parsing through parser, you can modify it in

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

```
// reading configuration
const MYSQL_READ = {
 host: "10.0.10.1",
}

// writing configuration
const MYSQL_WRITE = {
 host: "10.0.10.2"
}

export default {
 host: "127.0.0.1",
 adapter: {
 mysql: {
 parser: function(options){ // parsing method for mysql
 let sql = options.sql; // the SQL need to execute
 if(sql.indexOf("SELECT") === 0){ // SELECT query
 return MYSQL_READ;
 }
 return MYSQL_WRITE;
 }
 }
 }
}
```

The `options` of `parser` contains the SQL sentences that need to execute next, thus parser can return corresponding database configuration conveniently.

This doc stays at: [https://github.com/75team/www.thinkjs.org/tree/master/view/zh-cn/doc/2.0/model\\_config.md](https://github.com/75team/www.thinkjs.org/tree/master/view/zh-cn/doc/2.0/model_config.md).

## CRUD Operations

## Create Data

### add

Use `add` method to add a new record, return value is the id of inserted record:

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

JavaScript

### addMany

Use `addMany` create many records:

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

JavaScript

### thenAdd

We often need to prevent a field from duplication when designing database. So it's common to query whether data exists before inserting and just insert if it doesn't exist.

Model provides `thenAdd` to support this manipulation:

```
export default class extends think.controller.base {
 * addAction(){
 let model = this.model("user");
 //first param is the data need to add, second param is the condition, if there is no result when query use second param, the data will be added
 let result = yield model.thenAdd({name: "xxx", pwd: "yyy"}, {name: "xxx"});
 // result returns {id: 1000, type: "add"} or {id: 1000, type: "exist"}
 }
}
```

JavaScript

## Update Data

### update

Use `update` method to update data, return value is the influenced records:

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

JavaScript

### increment

Use `increment` method to increase one field's value:

```
export default class extends think.model.base {
 updateViewNums(id){
 return this.where({id: id}).increment("view_nums", 1); // increase one to reading number
 }
}
```

### decrement



Use `decrement` method to decrease one field's value:

```
export default class extends think.model.base {
 updateViewNums(id){
 return this.where({id: id}).decrement("coins", 10); // decrease ten coins
 }
}
```

## Query Data

Model provides many ways to query data, you can: query one line data, query multiple lines data, read the field value, read max value, read results count and so on.

### Query One Line Data

Use `find` to query one line data, return value is the object:

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

JavaScript

If there doesn't exist the data you need, return value is blank object `{}`. You can use `think.isEmpty` to check whether it is blank.

### Query Multiple Lines Data

Use `select` query many lines data, return value is results:

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

JavaScript

If there doesn't exist the data you need, return value is blank array `[]`. You can use `think.isEmpty` to check whether it is blank.

## Result Pagination

It's common to show paginated data in page. You need to query the total counts first, then calculate the number of pagination. Model provides `countSelect` method to facilitate this operation, it can query total counts automatically.

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

JavaScript

Return value's format:

```
{
 numsPerPage: 10, //number per page
 currentPage: 1, //current page
 count: 100, //total counts
 totalPages: 10, //total page number
 data: [{ //data of current page
 name: "thinkjs",
 email: "admin@thinkjs.org"
 }, ...]
}
```

JavaScript

If current page number exceeds page range, you can fix it through parameters. `true` means fix to first page, `false` means fix to last page: `countSelect(true)`, `countSelect(false)`.

If total count cannot be queried, you can pass it as a parameter like `countSelect(1000)`, means total count is 1000.

## count

Use `count` method to query total number of records that match the conditions:

```
export default class extends think.model.base {
 getMin(){
 // the total number where status = 'publish'
 return this.where({status: "publish"}).count();
 }
}
```

## sum

Use `sum` method to compute the sum of values of the same fields that match the conditions:

```
export default class extends think.model.base {
 getMin(){
 // the sum of values of view_nums where status = 'publish'
 return this.where({status: "publish"}).sum("view_nums");
 }
}
```

## max

Use `max` to find the largest value of the selected column:

```
export default class extends think.model.base {
 getMin(){
 // find the largest value of comments where status = 'publish'
 return this.where({status: "publish"}).max("comments");
 }
}
```

## min

Use `min` to find the smallest value of the selected column:

```
export default class extends think.model.base {
 getMin(){
 // find the smallest value of comments where status = 'publish'
 return this.where({status: "publish"}).min("comments");
 }
}
```

## Query Cache

Considering performance, querying data from cache is common. Doing it manually is difficult, so model provides `cache` method to set query cache:

```
export default class extends think.model.base {
 getList(){
 //set cache key and expire time
 return this.cache("get_list", 3600).where({id: { ">": 100 }}).select();
 }
}
```

JavaScript

These codes will cache query results. If cache matches, results will be returned directly from cache. Otherwise, database will be used. The key of cache is `get_list`, will expire after one hour.

Key is optional, model will generate a cache key from sql command:

```
export default class extends think.model.base {
 getList(){
 //only set cache time
 return this.cache(3600).where({id: {>: 100}}).select();
 }
}
```

## Cache Configuration

Config cache in model configuration's `cache` field:

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

- `on` controls the whole database cache configurations, `cache` will be disabled if it is off
- `type` type of cache, default is memory, supported types can be found at [Adapter -> Cache](#)
- `timeout` default expire time

## Delete Data

Use `delete` method to remove data, return the count of influenced row:

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

More operations in model can be found at [API -> model](#).

This doc stays at [https://github.com/75team/www.thinkjs.org/tree/master/view/zh-cn/doc/2.0/model\\_intro.md](https://github.com/75team/www.thinkjs.org/tree/master/view/zh-cn/doc/2.0/model_intro.md).

## Transaction

Model supports transaction operation provided the database you are using supports transaction too.

`InnoDB` and `BDB` engine of `Mysql` support transaction, if you need to use transaction in Mysql, must set the engine to InnoDB or BDB.

`SQLite` supports transaction.

### Use Transaction

Model provides `startTrans`, `commit` and `rollback` to operate transaction.

- `startTrans` start a transaction
- `commit` is used for commit transaction after your operations
- `rollback` is used for roll back if operation has exception

### ES6 Way

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

## Dynamic Class Creation Way

```
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();
 });
 }
})
```

JavaScript

## Transaction method

`startTrans`, `commit` and `rollback` need to be used when you use transaction. In order to simple this operation, model provides `transaction` method.

## ES6 Way

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

JavaScript

Note: Arrow function cannot used with `*/yield`, so we use `function *`. If you want to use arrow function, you can use async, like `async () => {}`.

## Dynamic Class Creation Way

```
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){

 })
 }
})
```

JavaScript

Transaction accepts a callback function which contains real operation logic and need to return.

This doc stays at [https://github.com/75team/www.thinkjs.org/tree/master/view/zh-cn/doc/2.0/model\\_transaction.md](https://github.com/75team/www.thinkjs.org/tree/master/view/zh-cn/doc/2.0/model_transaction.md).

## Relational Model

Tables in database often related to other tables and need to be operated with related tables together. For example, an article can have category, tag, comment and author, and these information often store in other related tables.

ThinkJS supports relational model which can simplify these operations.

## Supported Type

ThinkJS supports four relationships:

- `think.model.HAS_ONE` one to one model
- `think.model.BELONG_TO` one to one belong to
- `think.model.HAS_MANY` one to many
- `think.model.MANY_TO_MANY` many to many

## Create Relational Model

Use `thinkjs model [name] --relation` to create relational model:

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

JavaScript

This will create model file `src/home/model/post.js`.

## Set Relationship

Use `relation` property to set relationship:

```
export default class extends think.model.relation {
 init(...args){
 super.init(...args);
 //use relation property to set relationship, can set many relationships
 this.relation = {
 cate: {},
 comment: {}
 };
 }
}
```

JavaScript

You can also use ES7 syntax to define `relation` property:

```
export default class extends think.model.relation {
 //define relation property directly
 relation = {
 cate: {},
 comment: {}
 };

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

JavaScript

## Data Format of Single Relational Model

```
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: ""
 },
 };
 }
}
```

JavaScript

Each field's means:

- `type` type of relation
- `model` model name of relation table, default is `key`, here is `cate`
- `name` data field name, default is `key`, here is `cate`
- `key` related key of current model
- `fKey` related key of related table
- `field` field used to query related table, fKey must be included if you set this field
- `where` where condition used to query related table
- `order` order used to query related table

- `limit` limit used to query related table
- `page` page used to query related table
- `rModel` related model name in many to many type
- `rfKey` key in related table in many to many type

If you just want to set related type without other fields, you can use this simple way:

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

JavaScript

## HAS\_ONE

One to one relation, means current table has one additional table.

Suppose current model name is `user` and related table model name is `info`, then the default value of `key` field in configuration is `id`, and the default value of `fKey` is `user_id`.

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

JavaScript

Execute querying operation will get below data:

```
[
 {
 id: 1,
 name: "111",
 info: { // data from related table
 user_id: 1,
 desc: "info"
 }
 }, ...]
```

JavaScript

## BELONG\_TO

One to one relation, and one belong to another one, as opposed to HAS\_ONE.

Suppose current model name is `info` and related table model name is `user`, then the default value of `key` field in configuration is `user_id`, and the default value of `fKey` is `id`.

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

JavaScript

Execute querying operation will get below data:

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

## HAS\_MANY

One to many relation.

Suppose current model name is `post`, related table model name is `comment`, then the default value of `key` field in configuration is `id` and the default value of `fKey` is `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
 }
 };
 }
}
```

Execute querying operation will get below data:

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

If data in related table needs pagination, use `page` parameter:

```
"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();
 }
}
```

Besides using `setRelation`, you can also pass in a function, this function will be executed during parameter merging.

## MANYTOMANY

Many to many relation.

Suppose current model name is `post`, related table model name is `cate`, then we need a relationship table. The default value of `rModel` and `rfKey` fields in configuration are `post_cate` and `cate_id`.

```
JavaScript
"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"
 }
 };
 }
}
```

Querying results will be:

```
JavaScript
[
 {
 id: 1,
 title: "first post",
 cate: [
 {
 id: 1,
 name: "cate1",
 post_id: 1
 }, ...
], ...
 }, ...
]
```

### Dead Cycle

Suppose we have two tables, one set the other as HASONE and the other set this as BELONGTO, this will cause cycle querying during querying and result to dead cycle.

You can set `relation` field in config to close related querying and prevent dead cycle:

```
JavaScript
export default class extends think.model.relation {
 init(...args){
 super.init(...args);
 this.relation = {
 user: {
 type: think.model.BELONG_TO,
 relation: false //close related querying when query user
 }
 };
 }
}
```

You can also only close current model's relationship:

```
JavaScript
export default class extends think.model.relation {
 init(...args){
 super.init(...args);
 this.relation = {
 user: {
 type: think.model.BELONG_TO,
 relation: "info" //close info model's relationship when query user
 }
 };
 }
}
```

### Close Relationship Temporarily

After set relationship, operations like query will query related table automatically. If you don't want to query related table, just use `setRelation` method to close relationship temporarily.



## Close All

Use `setRelation(false)` to close all relationship query.

```
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();
 }
}
```

JavaScript

## Open Part

Use `setRelation('comment')` to query data from `comment`, other table won't be queried.

```
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();
 }
}
```

JavaScript

## Close Part

Use `setRelation('comment', false)` to close `comment` querying.

```
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();
 }
}
```

JavaScript

## Reopen All

Use `setRelation(true)` to reopen all related querying.

```
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();
 }
}
```

JavaScript

## mongo Relational Model

This relational model doesn't work for mongo model, mongo relational model stays here <https://docs.mongodb.org/manual/tutorial/model-embedded-one-to-one-relationships->

[between-documents/](#).

This doc stays at [https://github.com/75team/www.thinkjs.org/tree/master/view/zh-cn/doc/2.0/model\\_relation.md](https://github.com/75team/www.thinkjs.org/tree/master/view/zh-cn/doc/2.0/model_relation.md).

## Mysql

ThinkJS supports Mysql well, the underlying library is <https://www.npmjs.com/package/mysql>.

### Connections Pool

Mysql default has only one connection, if you want to use multiple connections, you can use connections pool. Modify `src/common/config/db.js`, such as:

```
export default {
 connectionLimit: 10 //create 10 connections
}
```

JavaScript

### socketPath

Default host and port will used to connect Mysql, if you want to use unix domain socket, see the below configuration:

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

JavaScript

### SSL options

Use below configuration to set SSL connection:

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

JavaScript

### Database Support Emoji

The encoding of database usually is `utf8`, but it doesn't support emoji. If you want database to support emoji, set database encoding to `utf8mb4`.

Besides, you have to modify `encoding` in `src/common/config/db.js` to `utf8mb4`:

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

JavaScript

### Error: Handshake inactivity timeout

In some Node.js version (like 4.2.0), connect Mysql will throw this error:

```
Error: Handshake inactivity timeout
at Handshake.sequence.on.on.on.on.on.self._connection._startTLS.err.code (/home/***/node_modules/mysql/lib/protocol/Protocol.js:154:17)
at Handshake.emit (events.js:92:17)
at Handshake._onTimeout (/home/***/node_modules/mysql/lib/protocol/sequences/Sequence.js:116:8)
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:40:27)
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
```

JavaScript

To solve this, just update Node.js to the latest version.

This doc stays at [https://github.com/75team/www.thinkjs.org/tree/master/view/zh-cn/doc/2.0/model\\_mysql.md](https://github.com/75team/www.thinkjs.org/tree/master/view/zh-cn/doc/2.0/model_mysql.md).

# MongoDB

ThinkJS supports MongoDB database, underlying module is [mongodb](#).

## Config

Change `type` in model configuration to `mongo` to use MongoDB database:

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

JavaScript

## Config options

In order to add additional params when connecting MongoDB service, add them to `options` :

```
export default {
 type: "mongo",
 adapter: {
 mongo: {
 options: {
 authSource: 'admin',
 replicaSet: 'xxx'
 }
 }
 }
}
```

JavaScript

Based on this config, the connection URL will become to `mongodb://127.0.0.1:27017/?authSource=admin` .

For more additional options, please read <http://mongodb.github.io/node-mongodb-native/2.0/reference/connecting/connection-settings/>.

## Create Model

Use command `thinkjs model [name] --mongo` to create model:

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

JavaScript

After executing, `src/common/model/user.js` will be created. If you want to place it within other module, add the specific module name:

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

JavaScript

This will create model file within `home` module, file name is `src/home/model/user.js` .

## Model Inheritance

Model has to inherit `think.model.mongo` class. If current class doesn't inherit it, you have to modify it:

### ES6 Way

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

JavaScript

## Dynamically Creating

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

JavaScript

## CURD Operations

CURD operations are same as Mysql, just read [Model -> Introduction](#).

## Create Index

mongo model can config index, model will create index automatically before CRUD operations. Configurations are placed in `indexes` property:

```
export default class extends think.model.mongo {
 init(...args){
 super.init(...args);
 //config index
 this.indexes = {

 }
 }
}
```

JavaScript

## Single Index

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

JavaScript

## Unique Index

Use `$unique` to set unique index:

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

JavaScript

## Multiple Fields Index

You can combine multiple fields to create index:

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

JavaScript

## Get Index

Use `getIndexes` to get created indexes:

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

JavaScript

## aggregate

Use `aggregate` method to do aggregation:

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

JavaScript

Details stay at <https://docs.mongodb.org/manual/core/aggregation-introduction/>.

## MapReduce

Use `mapReduce` method to do MapReduce operations:

```
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"
 })
 }
}
```

JavaScript

Details stay at <https://docs.mongodb.org/manual/core/aggregation-introduction/#map-reduce>.

This doc stays at [https://github.com/75team/www.thinkjs.org/tree/master/view/zh-cn/doc/2.0/model\\_mongodb.md](https://github.com/75team/www.thinkjs.org/tree/master/view/zh-cn/doc/2.0/model_mongodb.md).

## SQLite

ThinkJS supports SQLite database, underlying uses [sqlite3](#) module.

### Config

Change `type` property to `sqlite` to use SQLite. Modify `src/common/config/db.js` :

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

JavaScript

### Store Type

SQLite supports two ways to store data: memory and file, config `path` to `true` to use memory store:

#### Memory

```
export default {
 type: "sqlite",
 adapter: {
 sqlite: {
 path: true, // use memory to store data
 }
 }
}
```

JavaScript

#### File

Use file need to set the path of SQLite data, default is `src/common/runtime/sqlite` .

JavaScript

```
export default {
 type: "sqlite",
 adapter: {
 sqlite: {
 path: "/path/to/store/sqlite" //use file to store data
 }
 }
}
```

The path of data file is `path` + `/[name].sqlite`, default database `demo`'s file path is `src/common/runtime/sqlite/demo.sqlite`.

## CURD Operations

CURD operations are same as MySQL, just read [Model -> Introduction](#).

This doc stays at [https://github.com/75team/www.thinkjs.org/tree/master/view/zh-cn/doc/2.0/model\\_sqlite.md](https://github.com/75team/www.thinkjs.org/tree/master/view/zh-cn/doc/2.0/model_sqlite.md).

## PostgreSQL

ThinkJS supports `PostgreSQL` by using `pg` module in underlayer.

### Configuration

If you want to use PostgreSQL, you should change modle `type` to `postgresql` by modify `src/common/config/db.js`:

JavaScript

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

 }
 }
}
```

### CURD handler

PostgreSQL has same APIs with MySQL, you can know more in [Model -> Description](#).

## Adapter

### Adapter

Adapters are sorts of implementations which implements a same kind of function. In ThinkJS, the framework provides these adapters by default. Such as Cache, Session, WebSocket, Db, Store, Template, Socket and so on.

### Create An Adapter

You can create an adapter with console command like this:

Bash

```
thinkjs adapter template/dot
```

It creates a Template Adapter named `dot` in `src/common/adapter/template/dot.js`. The code probably likes the following:

JavaScript

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

The framework creates a `Base` class if the type you created doesn't exist, then other classes inherit from the `Base` class.

### Introduce an Adapter

You can introduce an Adapter by using `think.adapter` . For example:

```
let Template = think.adapter("template", "dot"); // introduce Template Adapter named dot
let instance = new Template(...args); // introduce an Adapter and instantiate it.
```

JavaScript

## Use third part Adapter

The framework searches Adapters from `src/common/adapter` and system path automatically, when it loads Adapters. You should regist third part Adapters if you need, otherwise the framework can't find them.

You can regist third part Adapters by using `think.adapter` , For example:

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

JavaScript

Then, the Adaptor files in `src/common/bootstrap/` can be loaded automatically when the service started.

## Cache

It is very helpful to use caches proper correctly in projects. So, ThinkJS provide a variety of caching methods,includes: Memory cache, file cache, Memcache and redis.

### Cache Types

ThinkJS supports the following cache types:

- `memory` Cache stored in Memory
- `file` Cache stored in file system
- `memcache` Cache stored in memcache
- `redis` Cache stored in redis

If you use Memcache or redis, you need set configuration information. See also [configuration of memcache](#) [configuration of redis](#)

### Configure Cache

The default cache configuration likes following. You can edit `src/common/config/cache.js` to change the configuration.

```
export default {
 type: "file", // the cache type
 timeout: 6 * 3600, // when the cache will expired , default is 6 hours.
 adapter: { // configurations of different type adaptor
 file: {
 path: think.RUNTIME_PATH + '/cache', // the path cache files put in
 path_depth: 2, // max depth generated by cache files
 file_ext: '.json' // cache files extend name
 },
 redis: {
 prefix: 'thinkjs_'
 },
 memcache: {
 prefix: 'thinkjs_'
 }
 }
};
```

JavaScript

**Note** : ThinkJS supports adaptor configuration from the version `2.0.6` .

In memcache or redis cache type, the `prefix` field is used. In this case, ThinkJS uses key + prefix as the storage key to prevent the conflict with other projects. If you don't want to set prefix, you can set it to empty string, like this:

```
export default {
 prefix: "" // it set the prefix of cache key to empty.
}
```

JavaScript

### Use Cache

You can add, delete, update and search the cache by using method `think.cache` , see also [API->think](#) for more details.

You can use method `this.cache` to operate cache, if your class is inherited from `think.http.base` , see also [API->think.http.base](#) for more details.

## Extend Cache

You can create a cache class named `foo` by using following command:

```
thinkjs adapter cache/foo
```

Bash

After the completion of the executing, ThinkJS will create the file `src/common/adapter/cache/foo.js`. Then you need to implement the following methods to extend cache class:

```
export default class extends think.cache.base {
 /**
 * init
 * @param {Object} options []
 * @return {} []
 */
 init(options){
 //set gc type & start gc
 this.gcType = 'cache_foo';
 think.gc(this);
 }
 /**
 * get cache
 * @param {String} name []
 * @return {Promise} []
 */
 get(name){
 }
 /**
 * set cache
 * @param {String} name []
 * @param {Mixed} value []
 * @param {Number} timeout []
 * @return {Promise}
 */
 set(name, value, timeout){
 }
 /**
 * delete cache
 * @param {String} name []
 * @return {Promise} []
 */
 delete(name){
 }
 /**
 * gc function
 * @return {Promise} []
 */
 gc(){
 }
}
```

JavaScript

To know the implementation of cache in ThinkJS, please see also (<https://github.com/75team/thinkjs/tree/master/src/adapter/cache>)

## Use Third Party Cache Adapter

To know how to use third party cache Adapter, please see also [Adapter -> intro](#)

## Session

Session is always needed for user login. ThinkJS supports a variety of session adapters, Such as `file`, `db`, `redis` and so on.

### Supported Session Types

- `memory` session stored in memory
- `file` session stored in file system
- `db` session stored in database
- `redis` session stored in redis



## db Session

You need create a table in your database except MongoDB, if you want to use db session. You can use the following SQL statement to create:

```
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;
```

SQL

The `think_` should be replaced with the table prefix in database configuration.

## redis Session

The redis session needs to configurate redis, see [configuration](#) for more details.

## Configure Session

Session configuration is as follows, you can edit it in the file `src/common/config/session.js`:

```
export default {
 type: 'file',
 name: 'thinkjs', // cookie name
 secret: '', // if the session cookie needs encrypt.
 timeout: 24 * 3600, // session expire time, defaults to one day
 cookie: { // cookie options
 length: 32
 },
 adapter: {
 file: {
 path: think.getPath('common', 'runtime') + '/session'
 }
 }
};
```

JavaScript

**Note**: The framework supports adaptor configuration from the version `2.0.6`.

This [configuration](#) is about Cookie.

## Read/Write Session

Controller or Logic can read/write session.

### Read Session

```
export default class extends think.controller.base {
 * indexAction(){
 //get session
 let value = yield this.session('userInfo');
 }
}
```

JavaScript

### Write Session

```
export default class extends think.controller.base {
 * indexAction(){
 //set session
 yield this.session('userInfo', data);
 }
}
```

JavaScript

### Clear Session

```
export default class extends think.controller.base {
 * indexAction(){
 //clear session of current user
 yield this.session();
 }
}
```

`http.session` method of http object can read and write Session. See also [API -> http](#) for more details.

## Extend Session

You can create a Session Adapter by using the command:

```
`thinkjs adapter session/foo`
```

this command will create file `src/common/adapter/session/foo.js`, then you need implement these methods:

```
export default class extends think.adapter.session {
 /**
 * init
 * @param {Object} options []
 * @return {} []
 */
 init(options){
 }
 /**
 * get Session
 * @param {String} name []
 * @return {Promise} []
 */
 get(name){
 }
 /**
 * set Session
 * @param {String} name []
 * @param {Mixed} value []
 */
 set(name, value){
 }
 /**
 * delete Session
 * @param {String} name []
 * @return {Promise} []
 */
 delete(name){
 }
 /**
 * update Session
 * @return {Promise} []
 */
 flush(){
 }
 /**
 * clear expired Session
 * @return {Promise} []
 */
 gc(){
 }
}
```

To know the implement of Session in ThinkJS, please see also <https://github.com/75team/thinkjs/tree/master/src/adapter/session>.

## Use Third Party Session Adapter

To know how to use third party session Adapter, please see also [Adapter -> intro](#)

# WebSocket

WebSocket is usually used to implement various functions such as chatroom. ThinkJS supports a lot of WebSocket libraries, for instance, `socket.io`, `sockjs` etc. Furthermore, by giving a simple encapsulation to these libraries, ThinkJS provided us consistent interfaces.

## Open WebSocket

WebSocket is closed by default. You can edit `src/common/config/websocket.js` to open it:

```
export default {
 on: false, // whether open WebSocket
 type: 'socket.io', // the WebSocket library name, defaults to socket.io
 allow_origin: '', // origin allowed
 adapter: undefined, // store adapter for socket, used in socket.io
 path: '', // url path for websocket
 messages: {
 // open: 'home/websocket/open',
 }
};
```

JavaScript

Change the `on` field to `true`, and restart Node.js.

## Map Event to Action

The encapsulation to WebSocket obeyed to the `socket.io` mechanism. The server and client communicate each other through events. So the server needs to map events to actions in order to respond correctly. The configuration is specified in `messages` field as following:

```
export default {
 messages: {
 open: 'home/socketio/open', // works on WebSocket connected.
 close: 'home/socketio/close', // works on WebSocket closed.
 adduser: 'home/socketio/adduser', // works when adduser.
 }
}
```

JavaScript

The event names `open`, `close` are immutable, representing a connection or disconnection. Others can be custom, you can add according to your need.

## Work With Action

Then, you can work with action like following code after finishing above configuration.

```
export default class extends think.controller.base {
 /**
 * works on WebSocket connected
 * @param {} self []
 * @return {} []
 */
 openAction(self){
 var socket = self.http.socket;
 this.broadcast('new message', {
 username: socket.username,
 message: self.http.data
 });
 }
}
```

JavaScript

## emit

You can emit event to the current socket in Action through `this.emit`:

```
export default class extends think.controller.base {
 /**
 * works on Websocket connected
 * @param {} self []
 * @return {} []
 */
 openAction(self){
 var socket = self.http.socket;
 this.emit('new message', 'connected');
 }
}
```

## broadcast

You can broadcast event to all sockets in Action through method `this.broadcast` :

```
export default class extends think.controller.base {
 chatAction(self){
 var socket = self.http.socket;
 // broadcast to all sockets excepting the current.
 this.broadcast('new message', {msg: 'message', username: 'xxx'});
 }
}
```

**Note** : the broadcast method broadcast to all sockets except current one. You can set the third parameter to `true` to include current one.

```
export default class extends think.controller.base {
 chatAction(self){
 var socket = self.http.socket;
 // broadcast to all sockets including the current.
 this.broadcast('new message', {msg: 'message', username: 'xxx'}, true);
 }
}
```

## Socket Object

You can get socket object in Action through `this.http.socket` .

## Event Data

You can get the event data in Action through `this.http.data` .

## socket.io

The `socket.io` library encapsulates socket both front end and back end, it is very convenient to use.

## io Object

You can get the `io` object in Action through `this.http.io` .It is an instance of `socket.io`

To know methods in io object, please see also [http://socket.io/docs/server-api/#server\(\)](http://socket.io/docs/server-api/#server()).

## Set Path

The socket.io process path is `/socket.io` by default. You can edit the following configuration if you need.

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

**Note** : After the server has modified the path, the client also should make the corresponding modification

## Set Adapter

When using multiple nodes to deploy WebSocket, multiple nodes can communicate with Redis. You can get things done by set up adapter.

```
import redis from 'socket.io-redis';

export default {
 adapter: function(){
 return redis({ host: 'localhost', port: 6379 });
 }
}
```

See also <http://socket.io/docs/using-multiple-nodes/> for more detail.

## socket.io Client

In browser end, you should introduce socket.io client. The download path is: <http://socket.io/download/>.

```
var socket = io('http://localhost:8360');
// emit event
socket.emit('name', 'data');
// listen event
socket.on('name', function(data){

})
```

This CDN url is available: <http://s4.qhimg.com/static/535dde855bc726e2/socket.io-1.2.0.js>.

## Check User Login

Websocket can get cookie when connected. So, you can check if the user is logged in in the `open` Action. For example:

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

 }
 }
}
```

## Code Sample: Chat

See also <https://github.com/75team/thinkjs2-demos/tree/master/websocket-socket.io> for more detailed chat code.

## SockJS

### Configuration

You should edit the `type` field in the configuration to `sockjs` :

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

### Sockjs Object

You can get sockjs object through `this.http.sockjs` in Action. The object is an instance of SocketJS.

### Set path

The SocketJS process path is `/sockjs` by default. You can edit the following configuration if you need change.

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

## SockJS Client

In Browser end, you should introduce SockJS client. The download path is: <https://github.com/sockjs/sockjs-client>.

SockJS client does not do too much encapsulation, so you need encapsulate it by yourself, change it to the event way, in order to follow the server side. The encapsulate method is

as follows:

```
JavaScript
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'}}));
};
```

After do above, we can receive and emit message, for example:

```
JavaScript
var socket = new SockJS('/sockjs'); // this path must be same with configuration.Defaults to /sockjs
// listen event
socket.on('add user', function(data){
});
// emit event
socket.emit('new message', 'xxx');
```

## Check User Login

For the safety reason, the SockJS doesn't supply cookie. So you can't check if the user is logged through cookie. You can output a token in your page, then send the token when connected to check. See also <https://github.com/sockjs/sockjs-node#authorisation> for more details.

## Code Sample: Chat

See also <https://github.com/75team/thinkjs2-demos/tree/master/websocket-sockjs> for more detailed chat code

## Nginx Reverse Proxy Setting

From the `1.3.13` version, Nginx supports reverse proxy WebSocket request, if used in the project, you need to add the following configuration in the nginx configuration file:

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

**Note** : when using `thinkjs` command to create project, ThinkJS creates nginx configuration file , including these two configuration fields. You can use it directly.

Please visit <http://nginx.org/en/docs/http/websocket.html> to read the reverse proxy WebSocket request document.

## How to get current WebSocket connecting object

You can get an array of WebSocket connecting objects by using `thinkCache(thinkCache.WEBSOCKET)` .

## How to achieve private chat

ThinkJS has not supported private chat yet, you can implement it by first find all of the WebSocket connections, and then match the corresponding connections.

## Template

Template Adapter used to support a variety of types of template engines, such as `ejs` , `swig` , etc.

## Supported Template Engines

- `base`
- `ejs` ejs template engine
- `jade` jade template engine
- `swig` a template engine supports template inheritance
- `nunjucks` a powerful template engine like jinja2

## Template Engine Configuration

To configure template engine, edit `src/common/config/view.js`:

```
export default {
 type: 'ejs',
 options: { // Additional configuration of the specific template engine

 }
};
```

JavaScript

## Use Template Engines

The template engine can be loaded automatically in the View. If you want to specify a template engine, then do it this way:

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

JavaScript

## Extend Template Engine Type

You can create an Template class named `foo` using the following command:

```
thinkjs adapter template/foo
```

JavaScript

The command creates file `src/common/adapter/template/foo.js`. Then, you should implement the following methods:

```
export default class extends think.adapter.base {
 /**
 * get compiled content
 * @params {String} templateFile the template files directory
 * @params {Object} tVar variables in template
 * @params {Object} config the configuration of template engine
 * @return {Promise} []
 */
 run(templateFile, tVar, config){

 }
}
```

JavaScript

To know the implement of Template in ThinkJS, please see also <https://github.com/75team/thinkjs/tree/master/src/adapter/template>.

## Use Third Part Template Adapter

To know how to use third part template adaptor, please see also [Adapter -> intro](#).

# Middleware

## Middleware

Handling user requests needs to take a lot of processes, such as parsing parameters, determining whether it is a static resource access or not, route parsing, page staticize judgment, executing actions, searching templates, rendering templates and so on. The project may also increase some other processes according to the requirements, like determining whether the IP is in the blacklist, CSRF detection and so on.

ThinkJS uses middlewares to handle these logics, each logic is an independent middleware. Many hooks are buried in the request process, each hook executes a series of middleware serially. And finally, one request logic process is completed.

## Hook List

ThinkJS contains the following hooks.

- `request_begin` request start

- `payload_parse` parse the data submitted
- `payload_validate` verify the data submitted
- `resource` static resource request process
- `route_parse` route parse
- `logic_before` before logic process
- `logic_after` after logic process
- `controller_before` before controller process
- `controller_after` after controller process
- `view_before` before view process
- `view_template` view process
- `view_parse` view parse
- `view_filter` view content filter
- `view_after` after view process
- `response_end` response end

Each hook calls one or more middleware to complete processing. The included middlewares are as the following:

```
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', 'subdomain_deploy', 'parse_route'],
 logic_before: ['check_csrf'],
 logic_after: [],
 controller_before: [],
 controller_after: [],
 view_before: [],
 view_template: ['locate_template'],
 view_parse: ['parse_template'],
 view_filter: [],
 view_after: [],
 response_end: []
};
```

JavaScript

## Config Hook

The middlewares executed default by hook usually can not meet the needs of the project. By this time, you can modify the middleware corresponding to the hook. The config file of hook is `src/common/config/hook.js`.

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

JavaScript

The above config will override the default config. If you want to add them in the original config, you can use the following ways.

### Append in Front

```
export default {
 payload_parse: ['prepend', 'parse_xml'], //append parse xml in front
}
```

JavaScript

### Append in End

```
export default {
 payload_parse: ['append', 'parse_xml'], //append parse xml in end
}
```

JavaScript

**Note** : It is recommended to use the way of append to config middleware, the name of system middleware may be modified in subsequent versions.

## Execute Hook

Use the method `think.hook` to execute the corresponding hook. eg.

```
await think.hook('payload_parse', http, data); //return a Promise
```

JavaScript

Use `this.hook` to execute hook directly in the class containing `http` object. eg.



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

## Create Middlewares

ThinkJS supports two modes of middleware, they are class mode and function mode. You can determine which mode to use depending on the complexity of middleware.

### Class Mode

If middleware needs to execute complex logic, you need to define it as class mode. Use the command `thinkjs` to create middleware, execute the following command in the project directory.

```
thinkjs middleware xxx
```

Bash

After execution, you will see the corresponding file `src/common/middleware/xxx.js`.

### ES6 Mode

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

JavaScript

### Dynamic Creation Class Mode

```
'use strict';
/**
 * middleware
 */
module.exports = think.middleware({
 /**
 * run
 * @return {} []
 */
 run: function(){
 }
});
```

JavaScript

Middlewares will be passed in `http`, you can use `this.http` to get it. The logic codes are executed in the method `run`. If they contain asynchronous operation, you need to return a `Promise` or use `*/yield`.

### Function Mode

If middlewares need to execute simple logic, you could define it as function mode. This middleware is not recommended to be created as a separate file, but to put together instead.

You could create the file `src/common/bootstrap/middleware.js`, which will be loaded automatically when service starts. And you can add one or more function mode middleware in this file. eg.

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

JavaScript

Function mode middlewares will be passed `http` object as a param. If the middleware has asynchronous operation, it need to return a `Promise` or use Generator Function.

The following is the implementation of parsing json payload in framework.

```

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){}
 });
});

```

## Set Value After Parsed

Some middlewares may parse the corresponding datas, and want to reassign `http` object. Such as parse the xml data passed, but hope to use the method `http.get` to get later.

- `http._get` store the value of GET params, `http.get(xxx)` to get data from this object
- `http._post` store the value of POST params, `http.post(xxx)` to get data from this object
- `http._file` store the value of uploaded file, `http.file(xxx)` to get data from this object

```

think.middleware('parse_xml', async http => {
 let payload = await http.getPayload();
 if (!payload) {
 return;
 }
 return parseXML(payload).then(data => {
 http._post = data; //assign the parsed data to http._post, use http.post to get value later
 });
});

```

See [API -> http](#) for more information about `http`.

## Prevent the Subsequent Execution

When executing the certain conditions, some middlewares may want to prevent the subsequent logic to execute. such as IP blacklist judgement, if hit the blacklist, then directly refuse the current request and no longer execute the subsequent logic.

ThinkJS provides the method `think.prevent` for preventing the subsequent logic to execute. This method returns a specific type of Reject Promise.

```

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();// directly end the current request
 return think.prevent(); // prevent the subsequent codes to execute
 }
});

```

In order to prevent the subsequent logic to execute, beside using the method `think.prevent`, you can also use `think.defer().promise` to return a Pending Promise.

If you don't want to end the current request directly, but return an error page instead, ThinkJS provides the method `think.statusAction`. See [Extend Function -> Error Handle](#) for detailed usage.

## Use Third-Party Middlewares

You can use third-party middlewares by use `think.middleware`. The corresponding code is in `src/common/bootstrap/middleware.js`. eg.

```

var parseXML = require('think-parsexml');

think.middleware('parseXML', parseXML);

```

Then just put `parseXML` config into hook.

It is recommended to release the common middlewares of project to npm repository. And the name of middleware is suggested to use `think-xxx`.

## Third-party Middleware List

See [plugin -> middleware](#) for the third-party middleware list.

## CSRF

ThinkJS provides a middleware to handle CSRF, but it's off by default.

### Turn CSRF On

Modify `src/common/config/hook.js` like this:

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

JavaScript

### Configure

The default configure of CSRF is as following, you can modify them in `src/common/config/csrf.js`:

```
export default {
 session_name: '__CSRF__', // Token value saved in session
 form_name: '__CSRF__', // CSRF key name, we can get value by this key and check the value
 errno: 400, //error number
 errmsg: 'token error' // error message
};
```

JavaScript

## Subdomains Deploy

When your projects become complicated, you may want to deploy different modules to different domains, but their code is still placed in one project folder. For example, the URL `admin.example.com` is deployed with admin dashboard function, and that's mapped to the `admin` module. ThinkJS provides subdomain middleware to handle this demand.

### Configuration

You can start this middleware by changing `src/common/config/hook.js`:

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

JavaScript

Then add the configuration about subdomains deploy in `config/config.js`:

```
export default {
 subdomain: {
 admin: 'admin', //It will map `admin.example.com` to admin module.
 ...
 }
}
```

JavaScript

If original pathname is `group/detail`, if your domain host is `admin.example.com`, the pathname received in ThinkJS will change to `admin/group/detail`.

## Ban Port Access

### Description

Online applications usually have Nginx as the reverse proxy. Nginx will redirect user requests to Node.js. Have reverse proxy will be easy to do load balance.

Once you have the reverse proxy, you usually want to ban the requests that directly accessed to the Node.js's port. At this time, you can set the Node.js port could only be accessed internally, or you must taking a judge in application level.

ThinkJS provides middleware to ban port access. If it's trouble to ban port access in host directly, you can take replace of Middleware to do it.

### Middleware Configuration

Modify `src/common/config/hook.js` as following:

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

JavaScript

Then modify `src/common/config/env/production.js` :

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

JavaScript

This will only ban port access in production environment and there is no effect in development environment.

## Listen Local Port Only

The default host Node.js listening when it starts service is `0.0.0.0` . It allows requests both coming from interior and outside. You can change it to `127.0.0.1` to allow local requests only.

You can modify `src/common/config/config.js` as following:

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

JavaScript

## More Features

### TypeScript

[TypeScript](#) is a free and open source programming language designed by Microsoft. TypeScript is a typed superset of JavaScript that it has some useful function in large project such as optional static type.

ThinkJS 2.1 has support TypeScript, you can auto-compile and auto-update in your projects.

### Create TypeScript Projects

We can create Typescript projects by using `--ts` :

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

Bash

TypeScript's file extension is `.ts` . If you want to create Typescript file manually, the postfix should be `.ts` too, or it will trigger error when calling `tsc` to compile.

### .d.ts file

`.d.ts` files are the description file of third party libraries. When we create our projects, the file named `typings/thinkjs/think.d.ts` will be created as ThinkJS's description file. You can include this description file in your project files as the following:

```
///
```

JavaScript

This code must be placed on the first line in the file, and the path must be correct. If your file has `use strict` , this code should set after `use strict` .

If your projects rely on other third party libraries, you should install their description files respectively. You can finish installation by use the [tsd](#).

You can find description file of some third party libraries in <https://github.com/DefinitelyTyped/DefinitelyTyped>.

### TypeScript Compiling

Because of there are some faults in compiling TypeScript, we have chosen to compile TypeScript `.ts` files to ES6 code first, then we compile the ES6 code to ES5 code using Babel 6.

If you find any problem about TypeScript, you can open an issue on <https://github.com/Microsoft/TypeScript>.

### Change Existing Projects To Using TypeScript

For projects that have used the ES6/7 features, ThinkJS could also supports to update them to use TypeScript.

### Change Entry File

Change entry file `www/development.js` as following:

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

JavaScript

## Modify package.json

Clear all dependencies related to `Babel` and `ThinkJS` and append following dependencies:

```
{
 "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"
 }
}
```

If some modules in `dependencies` and `devDependencies` have been used, they should be merged together.

After modifying, run `npm install` to install all dependencies.

## Modify .thinksrc

Change project's configuration file `.thinksrc` to:

```
{
 "createAt": "2016-01-13 17:27:19",
 "mode": "module",
 "ts": true
}
```

JavaScript

## Downloading `think.d.ts` Description File

Downloads file <https://github.com/75team/thinkjs/blob/master/template/think.d.ts> and save it as `typings/thinkjs/think.d.ts` .

## Modify File Extension

Modify all `.js` files in `src` to `.ts` files.

## Add `bin/compile.js` File

Downloads file <https://github.com/75team/thinkjs/blob/master/template/bin/compile.ts> and save it as `bin/compile.js` .

## Modify Compiling Command

Change compiling command in `package.json` to `node bin/compile.js` .

## Add Description File In Project's Discription File

All files in `src/` must add the following code in its first line, relative path should be correct:

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

JavaScript

Run `npm start` to start service after finished all modifying.

## Logic

When handling user requests in Action, you often need to get the submitted datas firstly, and then validate them. Only passing the data validation can do the subsequent operations. After the param validation, sometimes, you also need to judge permission. After all of these are correct, it is time to do the real logic process. If these codes are all placed in one

Action, it will must make the codes of Action very complex and redundant.

In order to solve this problem, ThinkJS add a layer of `Logic` before Controller. The Action in Logic and the Action in Controller are one-to-one correspondence. System will call the Action in Logic automatically before calling the Action in Controller.

## Logic Layer

The directory of Logic is `src/[module]/logic`. When using command `thinkjs controller [name]` to create Controller, there will automatically create the corresponding Logic. The codes of the Logic are roughly like the followings.

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

The Action in Logic and the Action in Controller are one-to-one correspondence. The Action in Logic also supports `__before` and `__after` and other magic methods.

## Request Type Validation

Only a couple of request types will be used in one specific Action. You can config some specific request types to validate them.

```
JavaScript
export default class extends think.logic.base {
 indexAction(){
 this.allowMethods = 'post'; //allow post request only
 }
 testAction(){
 this.allowMethods = 'get,post'; //allow get and post requests only
 }
}
```

## Data Validation Config

The config of data validation is as follows.

```
JavaScript
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 doc: "string|default:index",
 version: "string|in:1.2,2.0|default:2.0"
 }
 }
}
```

## Config Format

The config format is `field name` -> `config`, each field config supports multiple validation types. The multiple validation types are separated by `|`, the validation type and param are separated by `:`, param and param are separated by `,`.

## Param Format

Params could follow the end of validation type. Besides supporting simply string params separated by commas, it also supports the complex param in JSON format.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 field1: "array|default:[1,2]", // param is an array
 field2: 'object|default:{\name\":"thinkjs\"}' //param is an object
 }
 }
}
```

ThinkJS also allow you config it with object like:

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 field1: {required: true, array: true, default: [1, 2]}, //parameter is array
 field2: {object: true, default: {name: "thinkjs"}} //parameter is object
 }
 }
}
```

## Supported Data Type

The supported data types include `boolean` , `string` , `int` , `float` , `array` , `object` . And the default type is `string` .

## Default Value

Use `default: value` to define the default value of field. If the value of current field is empty, it will be overridden by the default one. What you get subsequently will be the default value.

## The Way to Get Data

By default, get the field value according to the current request type. If the type of current request is GET, use `this.get('version')` to get the value of `version` field. If the type of current request is POST, use `this.post` to get the field value.

But sometimes in the POST type, you may want to get the params from uploaded file or URL. By this time, you need to specify the way to get data. The supported ways to get data are `get` , `post` and `file` .

```
export default class extends think.logic.base {
 /**
 * save data, POST request
 * @return {} []
 */
 saveAction(){
 let rules = {
 name: "required",
 image: "object|file|required",
 version: "string|get|in:1.2,2.0|default:2.0"
 }
 }
}
```

The above demo specifies to use `post` method to get the value of the field `name` , use `file` method to get the value of the field `image` , use `get` method to get the value of the field `version` .

## Error Message

The above config only specifies the certain validation rules but not the error message when validation failure. Error messages support internationalization, you need to define it in the config file `src/common/config/locale/[lang].js` . eg.

```
// src/common/config/locale/en.js
export default {
 validate_required: '{name} can not be blank',
 validate_contains: '{name} need contains {args}',
}
```

The key is `validate_` + `validation type name` . The value supports two params: `{name}` and `{args}` , which respectively indicate the field name and the passed param.

If you want to define the detailed message of a certain error type for a specific field, you could add a field name to the end. eg.

```
// src/common/config/locale/en.js
export default {
 validate_required: '{name} can not be blank',
 validate_required_email: 'email can not be blank', //specify the error message of required for email field
}
```

## Data Validation Method

After configuring the validation rules, you can use the method `this.validate` to validate. eg.

```
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());
 }
 }
}
```

If the return value is `false`, you could use method `this.errors` to get the detailed error message. After getting the error message, you could use method `this.fail` to output it in JSON format, or use method `this.display` to output a page.

In template, you can get the error message by `errors` field. The following is the way to show error message (taking ejs template as an example).

```
<%for(var field in errors){%>
 <%-field%>:<%errors[field]%>
<%}%>
```

## Validate Automatically

In generally, there will output a JSON message after validation failure. If `this.validate` needs to be called manually everytime to validate in Action of Logic, it must be inconvenient. You can make the validation automatically by assigning validation rules to `this.rules`. eg.

```
export default class extends think.logic.base {
 indexAction(){
 this.rules = {
 doc: "string|default:index",
 version: "string|in:1.2,2.0|default:2.0"
 }
 }
}
```

After assigning validation rules to `this.rules`, the validation will be automatically done after Action execution. If there are errors, it will directly output error messages in JSON format. Automatical validation uses the magic method `__after` to complete.

## Supported Validation Type

### required

Required Item.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 name: 'required' //the value of name is required
 }
 }
}
```

### requiredIf

When the value of the other certain item is one of the specified values, this item is required. eg.



JavaScript

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 name: 'requiredIf:email,admin@example.com,admin1@example.com'
 }
 }
}
```

When the value of `email` is one of `admin@example.com` and `admin1@example.com`, the value of `name` is required.

### requiredNotIf

When the value of the other certain item is not one of the specified values, this item is required. eg.

JavaScript

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 name: 'requiredNotIf:email,admin@example.com,admin1@example.com'
 }
 }
}
```

When the value of `email` is not one of `admin@example.com` or `admin1@example.com`, the value of `name` is required.

### requiredWith

When one of the values of some other certain items does exist, this item is required. eg.

JavaScript

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 name: 'requiredWith:email,title'
 }
 }
}
```

When one of the values of `email` and `title` does exist, the value of `name` is required.

### requiredWithAll

When all of the values of some other certain items do exist, this item is required. eg.

JavaScript

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 name: 'requiredWithAll:email,title'
 }
 }
}
```

When all of the values of `email` and `title` do exist, the value of `name` is required.

### requiredWithout

When one of the values of some other certain items does not exist, this item is required. eg.

JavaScript

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 name: 'requiredWithout:email,title'
 }
 }
}
```

When one of the values of `email` and `title` does not exist, the value of `name` is required.

### requiredWithoutAll

When all of the values of some other certain items do not exist, this item is required. eg.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 name: 'requiredWithoutAll:email,title'
 }
 }
}
```

When all of the values of `email` and `title` do not exist, the value of `name` is required.

### contains

The value needs to contain the certain value.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 name: 'contains:thinkjs' //need to contain string 'thinkjs'.
 }
 }
}
```

### equals

Be equal to the value of the other item.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 name: 'equals:firstname'
 }
 }
}
```

The value of `name` needs to be equal to the value of `firstname`.

### different

Be different to the value of the other item.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 name: 'different:firstname'
 }
 }
}
```

The value of `name` can't be equal to the value of `firstname`.

### before

The value needs to be before a certain date. By default, it needs to be before the current date.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 start_time: 'before', //need to be before the current date
 start_time1: 'before:2015/10/12 10:10:10' //need to be before 2015/10/12 10:10:10
 }
 }
}
```

### after

The value needs to be after a certain date. By default, it needs to be after the current date.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 end_time: 'after', //need to be after the current date
 end_time1: 'after:2015/10/10' //need to be after 2015/10/10
 }
 }
}
```

### alpha

The value must only consist of [a-zA-Z].

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 en_name: 'alpha'
 }
 }
}
```

The value of `en_name` must only consist of [a-zA-Z].

### alphaDash

The value must only consist of [a-zA-Z\_].

### alphaNumeric

The value must only consist of [a-zA-Z0-9].

### alphaNumericDash

The value must only consist of [a-zA-Z0-9\_].

### ascii

The value must only consist of ascii.

### base64

The value must only consist of base64.

### byteLength

The length of bytes needs to be in a certain range.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 name: 'byteLength:10' // the length of bytes can not less than 10
 name1: 'byteLength:10,100' //the length of bytes must be in the range of 10 to 100
 }
 }
}
```

### creditcard

The value needs to be a credit card number.

### currency

The value needs to be a currency.

### date

The value needs to be a date.

### decimal

The value needs to be a decimal.

### **divisibleBy**

The value needs to be divisible by a number.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 count: 'divisibleBy:3' //could to be divisible by 3
 }
 }
}
```

JavaScript

### **email**

The value needs to be email format.

### **fqdn**

The value needs to be a qualified domain name.

### **float**

The value needs to be a float.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 money: 'float' //need to be a float
 money1: 'float:3.2' //need to be a float, and the minimum is 3.2
 money2: 'float:3.2,10.5' //need to be a float, and the minimum is 3.2, the maximum is 10.5
 }
 }
}
```

JavaScript

### **fullWidth**

The value needs contain full width char.

### **halfWidth**

The value needs contain half width char.

### **hexColor**

The value needs to be a hex color value.

### **hex**

The value needs to be hex.

### **ip**

The value needs to be ip format.

### **ip4**

The value needs to be ip4 format.

### **ip6**

The value needs to be ip6 format.

### **isbn**

The value needs to be a book serial number.

### **isin**

The value needs to be ISIN (International Securities Identification Numbers).

## iso8601

The value needs to be iso8601 date format.

## in

The value needs to be in some certain values.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 version: 'in:1.2,2.0' //need to be one of 1.2, 2.0
 }
 }
}
```

JavaScript

## noin

The value needs to be not in some certain values.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 version: 'noin:1.2,2.0' //need to be not in 1.2, 2.0
 }
 }
}
```

JavaScript

## int

The value needs to be int.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 value: 'int' //int
 value1: 'int:1' //can not less than 1
 value2: 'int:10,100' //need to be in the range of 10 to 100
 }
 }
}
```

JavaScript

## min

The value can not less than the certain value.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 value: 'min:10' //can not less than 10
 }
 }
}
```

JavaScript

## max

The value can not great than the certain value.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 value: 'max:10' //can not great than 10
 }
 }
}
```

JavaScript

## length

The length needs to be in the certain range.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 name: 'length:10' //the length can not less than 10
 name1: 'length:10,100' //the length need to be in the range of 10 to 100
 }
 }
}
```

### minLength

The length can not to be less than the min-length.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 name: 'minLength:10' //the length can not to be less than 10
 }
 }
}
```

### maxLength

The length can not to be great than the max-length.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 name: 'maxLength:10' //the length can not to be great than 10
 }
 }
}
```

### lowercase

The value needs to be all lowercase.

### uppercase

The value needs to be all uppercase.

### mobile

The value needs to be a mobile phone.

```
export default class extends think.logic.base {
 indexAction(){
 let rules = {
 mobile: 'mobile:zh-cn' //must be a chinese mobile phone
 }
 }
}
```

### mongoid

The value is the ObjectID of MongoDB.

### multibyte

Include multibyte char.

### url

The value is url.

### order

Database query order, like name DESC.

## field

Database query field, like name,title.

## image

Whether the file uploaded is a pic

## startWith

The value starts with some certain chars.

## endWith

The value ends with some certain chars.

## string

The value is string.

## array

The value is array.

## boolean

The value is boolean.

## object

The value is object.

## regexp

RegExp, such as:

```
export default class extends think.logic.base {
 indexAction(){
 this.rules = {
 number: {
 required: true,
 regexp: /^\\d{6}$/
 }
 }
 }
}
```

JavaScript

## Extending Validation Types

If the default supported validation types can not meet the demand, you can use the method `think.validate` to extend the validation types. eg.

```
// src/common/bootstrap/validate.js
think.validate('validate_name', (value, ...args) => {
 //need to return true or false
 //true-validate success, false-validate fail
})
```

JavaScript

The above registers a validation type named `validate_name`, thus, you can directly use this validation type in Logic.

## Param Parse

If you want to parse `args`, you can register a function. eg. the name of the above validation type is `validate_name`, then the corresponding name of parse param is `_validate_name`, that is `_` + `validation type`.

```
think.validate('_validate_name', (args, data) => {
 let arg0 = args[0];
 args[0] = data[arg0].value; //parse the first param field name to the corresponding param value
 return args;
})
```

JavaScript

# Thinkjs Command

After installing thinkjs module globally, there should be the thinkjs commands in your system. Run the command `thinkjs -h` in your terminal to get more detailed introduction.

```
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
```

Options:

```
-h, --help output usage information
-V, --version output the version number
-e, --es6 use es6 for 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(mini, normal, module), default is module, used in `new` command
```

## Create Project

You can create a project by run command `thinkjs new <projectPath>` :

```
thinkjs new thinkjs_demo;
```

Bash

## ES6 Way

If you want to create an ES6 mode project, `--es6` option is required. Thus, codes in the generated files are all in ES6 syntax.

```
thinkjs new thinkjs_demo --es6
```

Bash

## Set Project Mode

By default, new created project is divided by modules. If the project is small and you don't want to have it divided by modules, you can specify `--mode` option when creating project. eg.

```
thinkjs new thinkjs_demo --mode=mini
```

Bash

The following is the supported modules list:

- `mini` single-module project, for a very simple project.
- `normal` general project, which modules are divided according to the function.
- `module` divided by modules, for large projects or the project strictly divided by modules.

**Note** : After the project created, a hidden file named `.thinkjsrc` will be created in the project directory, which contains some configuration of the current project. And this hidden file will affect subsequent creating files, so you need to put it into the version repository.

## Add Module

The module `common` and `home` will be automatically created when creating projects. If you need to create other modules, you can execute the command `thinkjs module [name]` in the project directory. eg.

```
thinkjs module admin
```

Bash

After execution, the directory `src/admin` and the corresponding files in it will be created.

## Add Middleware

You can add middleware by run the command `thinkjs middleware [name]` in the project directory. eg.



```
thinkjs middleware test;
```

Bash

Execute it will create the file `src/common/middleware/test.js` .

## Add Model

You can add model by run the command `thinkjs model [name]` in the project directory. eg.

```
thinkjs model user;
```

Bash

Execute it will create the file `src/common/model/user.js` .

This file is in the `common` module by default. If you want to create it in other module, just specify the module. eg.

```
thinkjs model home/user;
```

Bash

Thus, it will create the file `src/home/model/user.js` , which is in the `home` module.

## Add Mongo Model

By default, the added model is relational database model. If you want to create Mongo Model, specify `--mongo` option. eg.

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

Bash

## Add Relation Model

Specify `--relation` option to create Relation Model. eg.

```
thinkjs model home/user --relation
```

Bash

## Add Controller

You can add controller by run the command `thinkjs controller [name]` in the project directory. eg.

```
thinkjs controller user;
```

Bash

After execution, there will create the file `src/common/controller/user.js` , and the file `src/common/logic/user.js` will be also created at the same.

These created files are in the `common` module by default. If you want to create them in other module, specify the module. eg.

```
thinkjs controller home/user;
```

Bash

Thus, it will create the file `src/home/controller/user.js` , which is in the `home` module.

## Add Rest Controller

If you want to provide Rest API, specify `--rest` option. eg.

```
thinkjs controller home/user --rest;
```

Bash

## Add service

You can add service by the command `thinkjs service [name]` in the project directory. eg.

```
thinkjs service github; #create the service that calls github interface
```

Bash

After execution, there will create the file `src/common/service/github.js` .

This created file is in the `common` module by default. If you want to create it in other module, specify the module. eg.

```
thinkjs service home/github;
```

Bash

Thus, it will create the file `src/home/service/github.js` , which is in the `home` module.

## Add adapter

You can add adapter by the command `thinkjs adapter [type]/[name]` in the project directory. eg.

```
thinkjs adapter template/dot
```

Bash

After execution, there will create the file `src/common/adapter/template/dot.js`, which means a template type adapter named dot.

## Add Plugin

Two kinds of plugins ThinkJS supported are middleware and adapter. You can initialize a plugin by the command `thinkjs plugin <pluginName>`, and then to develop.

```
thinkjs plugin think-template-dot
```

Bash

It suggested that the name of plugin could start with `think-`, so that it's convenient to search for other users after the plugin release to npm repository.

## Static Resources Access

We generally need to reference static resources in a template when developing projects.

When using the command `thinkjs` to create a project, it will automatically create the directory `www/static`, where specially stores JS, CSS, images and other static resources.

### Access Static Resources

After putting the static resources in `www/static`, you can reference them in a template by the following approaches.

#### Reference JS files in template

```
<script src="/static/js/foo.js"></script>
```

Markup

#### Reference CSS files in template

```
<link href="/static/css/foo.css" rel="stylesheet" />
```

Markup

#### Reference Image files in template

```

```

Markup

## Static Resources Access Configuration

Judging whether the request is a static resource request, we use regular expression. The default configuration is as follows.

```
export default {
 resource_on: true, // enable static resources resolution function
 resource_reg: /^(static\/|[\^\/]+\.(?!js|html)\w+$/), // regular expression for judging static resource request
}
```

JavaScript

You can modify the configuration file `src/common/config/config.js` according to your project requirements.

### Close Online Static Resources Access

After the project is online, it generally uses nginx or other WEB server as a agent. At this time, you can let nginx to directly handle the static resource requests. Thus, you could close the static resources access to improve performance.

Set the option `resource_on` to `false` in the configuration file `src/common/config/env/production.js` to close it. eg.

```
export default {
 resource_on: false
}
```

JavaScript

## Service

Some projects need to call some third-party services like Github related interfaces. If codes in the controller directly call these interfaces, on the one hand it will lead to code

complexity, on the other hand it could not do more code reuse.

For these cases, you can encapsulate some services for controllers to call.

## Create Services

Use the command `thinkjs service [name]` to create service. See [Extend functions -> ThinkJS Command -> Add Service](#) for more detailed usage.

The default generated service is a class. But some services only need to provide some static methods, at that time you could just change class to object.

## Load Services

Use `think.service` to load services. eg.

```
export default class extends think.controller.base {
 indexAction(){
 let GithubService = think.service('github');
 let instance = new GithubService();
 }
}
```

JavaScript

If you want to load service across-modules, use the following approaches.

```
export default class extends think.controller.base {
 indexAction(){
 let GithubService = think.service('github', 'admin'); //load github service in admin
 let instance = new GithubService();
 }
}
```

JavaScript

**Note** : If the project is not very complex, it's suggested that put service in the module `common`. Thus, they are both convenient to load.

## Cookie

### Get Cookie

In controller or logic, you can get cookie by use `this.cookie`. eg.

```
export default class extends think.controller.base {
 indexAction(){
 let cookie = this.cookie('theme'); //get the cookie 'theme'
 }
}
```

JavaScript

Http object also provides the method `cookie` to get cookie. eg.

```
let cookie = http.cookie('theme');
```

JavaScript

### Cookie Config

The cookie default config is as follows.

```
export default {
 domain: '',
 path: '/',
 httponly: false, // whether http only
 secure: false,
 timeout: 0 // valid time, 0-browser process, unit is second
};
```

JavaScript

The default cookie is invalid along with the closing of browser process, and you can modify it in the config file `src/common/config/cookie.js`. eg.

```
export default {
 timeout: 7 * 24 * 3600 //set cookie valid time to 7 days
};
```

JavaScript

### Set Cookie

In controller or logic, you can set cookie by use `this.cookie` . eg.

```
export default class extends think.controller.base {
 indexAction(){
 this.cookie('theme', 'default'); //set cookie theme to default
 }
}
```

JavaScript

Http object also provides the method `cookie` to set cookie. eg.

```
http.cookie('theme', 'default');
```

JavaScript

If you want to change some params when setting cookie, you can use these three params like the followings.

```
export default class extends think.controller.base {
 indexAction(){
 this.cookie('theme', 'default', {
 timeout: 7 * 24 * 3600 //set cookie valid time to 7 days
 });
 }
}
```

JavaScript

## Babel

We upgraded Babel from 5 to 6 in ThinkJS 2.1. Because Babel 6 is a refactored version that all functions build by plug-ins, and many modules of it relied on different plug-ins. That will leads to some problems:

- The node\_modules directory will be too big, and be very slow at the first run.
- On Windows hosts, some errors may be reported because of the deep path.

The best solution is to upgrade your npm to version 3. Use following command to upgrade:

```
npm install -g npm@3
```

Bash

## Change compile parameters

The default compile parameters of Babel 6 are:

```
{
 presets: ['es2015-loose', 'stage-1'],
 plugins: ['transform-runtime']
}
```

JavaScript

If the configure here doesn't satisfy with your requirements, you can modify it in file `www/development.js` :

```
instance.compile({
 retainLines: true,
 log: true,
 presets: [], //appended presets list
 plugins: [] //appended plugins list
});
```

JavaScript

At some point, before your deploy, you want to run `npm run compile` , this will actually call the `compile` script in `package.json` . So, if you have set `presets` or `plugins` list, then you should change the `compile` command accordingly.

## Error Handling

The application will encounter all kinds of errors when handling user requests. Such as system internal error, url not exist, permission denied, service unavailable and so on. In these cases, it needs to show the corresponding error page for users.

### Error Page

When using the command `thinkjs` to create project, it will automatically add the error handle logic file and the corresponding error page.

The error logic file is located in `src/common/controller/error.js` , and its content is roughly as follows.

```

'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);
 }
}

```

The path of the corresponding error template page is `view/common/error_{Number}.html` .

## Error Type

System default supported error types are `400` , `403` , `404` , `500` and `503` .

### 400

Error request, like maliciously construct some illegal data access, url accessed is illegal and so on.

### 403

The current access has no permission.

### 404

The requested url is not found.

## 500

System internal happened error, which leads to the current request is unavailable.

## 503

Service is unavailable until it is recovered.

## Extend Error Type

You can extend error type in your project depending on the practical requirement. such as adding the specific `600` error, and you can do as the following steps.

### 1. add `_600Action`

Add the following codes into `src/common/controller/error.js` file in the appropriate place.

```
_600Action(){
 return this.displayErrorPage(600);
}
```

JavaScript

### 2. Add Error Page

Add the file `view/common/error_600.html`, and write the corresponding error information into it.

### 3. Show Error Page

After added the error, you need to call it correspondingly in order to show it for users. It can be achieved by `think.statusAction` method. eg.

```
export default class extends think.controller.base {
 indexAction(){
 if(someError){
 return think.statusAction(600, this.http); //show 600 error, need to pass http object
 }
 }
}
```

JavaScript

## Modify Error Page Style

In order to modify the error page style, you just need to modify the corresponding template file. Eg. edit the template file `view/common/error_404.html` to modify `404` error page style.

## Error Message

---

### EPERM

#### Operation Not Permitted

An attempt was made to perform an operation that requires appropriate privileges.

### ENOENT

#### No Such File Or Directory

Commonly raised by fs operations; a component of the specified pathname does not exist – no entity (file or directory) could be found by the given path.

### EACCES

#### Permission Denied

An attempt was made to access a file in a way forbidden by its file access permissions.

### EEXIST

#### File Exists

An existing file was the target of an operation that required that the target not exist.

### ENOTDIR

### Not a directory

A component of the given pathname existed, but was not a directory as expected. Commonly raised by `fs.readdir`.

### EISDIR

#### Is a directory #####

An operation expected a file, but the given pathname was a directory.

### EMFILE

#### Too many open files in system #####

Maximum number of file descriptors allowable on the system has been reached, and requests for another descriptor cannot be fulfilled until at least one has been closed.

Commonly encountered when opening many files at once in parallel, especially on systems (in particular, OS X) where there is a low file descriptor limit for processes. To remedy a low limit, run `ulimit -n 2048` in the same sh that will run the Node.js process.

### EPIPE

#### Broken pipe

A write on a pipe, socket, or FIFO for which there is no process to read the data. Commonly encountered at the net and http layers, indicative that the remote side of the stream being written to has been closed.

### EADDRINUSE

#### Address already in use

An attempt to bind a server (net, http, or https) to a local address failed due to another server on the local system already occupying that address.

### ECONNRESET

#### Connection reset by peer #####

A connection was forcibly closed by a peer. This normally results from a loss of the connection on the remote socket due to a timeout or reboot. Commonly encountered via the http and net modules.

### ECONNREFUSED

#### Connection refused #####

No connection could be made because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the foreign host.

### ENOTEMPTY

#### Directory not empty

A directory with entries was the target of an operation that requires an empty directory – usually `fs.unlink`.

### ETIMEDOUT

#### Operation timed out

A connect or send request failed because the connected party did not properly respond after a period of time. Usually encountered by http or net – often a sign that a connected socket was not `.end()`d appropriately.

## Internationalization

---

### Get Language

Use the method `http.lang` to get the language of current user from cookie or header. eg.

```
let lang = http.lang();
```

JavaScript

If you want to support getting the user selected language from cookie, you need to set the language name in the cookie. You can modify it in the config file

```
src/common/config/locale.js . eg.
```

JavaScript

```
export default {
 cookie_name: 'think_locale', // the cookie name to store language
 default: 'en' // default language
};
```

Use the method `this.lang` to get the corresponding language directly in Controller.

## Parse the Language from URL

In some cases, the language is parsed from the URL. eg. the url of current page is `https://www.thinkjs.org/zh-cn/doc/2.0/i18n.html`, which contains the language `zh-cn`.

In this case, you need to use middleware to parse the language in your project. eg.

JavaScript

```
think.middleware('get_lang', http => {
 let supportLangs = think.config('locale.support');
 let lang = http.pathname.split('/')[0]; // get the language from URL

 if(supportLangs.indexOf(lang) > -1){
 http.pathname = http.pathname.substr(lang.length + 1);
 }else{
 lang = http.lang(); // get the language from cookie or header
 if(supportLangs.indexOf(lang) === -1){
 lang = http.config('locale.default'); //default supported language
 }
 }
 http.lang(lang, true); //set the language, and allow to add language directory into the template path
});
```

After parsing the language from URL, you could use method `http.lang` to set the language. And later, you can directly use `http.lang` to get the language in Controller.

After defining middleware `get_lang`, add it into the corresponding hook. eg.

JavaScript

```
export default {
 route_parse: ['prepend', 'get_lang'], //add get_lang prepend into route_parse hook
}
```

## Language Variable Config

Projects that support international need to config the different variable values in different languages. The config file is located in `src/common/config/locale/[lang].js`, format is as follows.

JavaScript

```
// src/common/config/locale/zh-cn.js
export default {
 'title-home': 'ThinkJS Official Website - A Node.js MVC Framework Support All Of ES6/7 Features',
 'title-changelog': 'Update logs - ThinkJS Official Website',
}
```

JavaScript

```
// 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'
}
```

## Get the Language Variable

After config the language variable, we can get the value of current language by `http.locale` method. eg.

JavaScript

```
let homeTitle = http.locale('title-home');
```

If in Controller, we can get it directly by `this.locale` method. eg.

JavaScript

```
export default class extends think.controller.base {
 indexAction(){
 let homeTitle = this.locale('title-home');
 }
}
```



## Use the Language Variable in Template

In template, use the function `__` to get the value of corresponding language. The following is `ejs` template as an example.

```
<%- __('title-home') %>
```

Markup

## Set the Template Language Path

In some projects, we need to customize different templates depending on the different languages. By this time, it is appropriate that adding a layer of language directory to the template path. eg. `view/zh-cn/home/index_index.html`, adds a layer of language directory `zh-cn` to the path.

Use the method `http.lang` to set language and add a layer of language directory in the template path. eg.

```
http.lang(lang, true); // true indicates that you can add a layer of language directory in the template path
```

JavaScript

In Controller, use the method `this.lang` to set. eg.

```
export default class extends think.controller.base {
 indexAction(){
 let lang = getFromUrl();
 this.lang(lang, true);
 ...
 }
}
```

JavaScript

## Path Const

System provides many consts for project, and you can use them to access corresponding files conveniently.

### think.ROOT\_PATH

The root of project.

### think.RESOURCE\_PATH

The root of static resources, path is `think.ROOT_PATH` + `/www/`.

### think.APP\_PATH

The directory of APP code, path is `think.ROOT_PATH` + `/app/`.

### think.THINK\_PATH

The root directory of ThinkJS framework.

### think.THINKLIBPATH

ThinkJS framework `lib` directory.

### think.getPath(module, type)

For the model,controller,view directory and so on, because each module has these directories, so we can't give a fixed path value. You can get path value of module by

```
think.getPath .
```

```
let path1 = think.getPath('common', 'model'); //get the directory of common module
let path2 = think.getPath('home', 'controller'); //get the directory of home module
```

JavaScript

## User-defined Path Consts

Besides the system properties and methods to get path, you can also define extra path consts in project.

### Define in Entrance File

The entrance file of project is `src/index.js` or `src/production.js`, you can define path const in it. eg.

```

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', // define the directory of file uploading
 env: 'development'
});

instance.run();

```

### Define in Startup File

The files defined in `src/common/bootstrap` will be loaded automatically, so you can also define path const in this file. eg.

```

// src/common/bootstrap/common.js
think.UPLOAD_PATH = think.RESOURCE_PATH + '/upload'; // define the directory of file uploading

```

## REST APIs

In projects, we often need to provide APIs for third party to call. A common API design specification is using REST API, which uses HTTP request type to identify resource operation.

- `GET /ticket` # get ticket list
- `GET /ticket/12` # view the specific ticket
- `POST /ticket` # new a ticket
- `PUT /ticket/12` # update ticket 12
- `DELETE /ticket/12` # delete ticket 12

ThinkJS provides a very convenient way to create REST API. After created, it can response REST API process without writing any code, and it can also response additional demand by customization.

### Create REST APIs

Use `thinkjs controller [name] --rest` to create REST API. eg.

```
thinkjs controller home/ticket --rest
```

The above command means that a Rest Controller named `ticket` is created in `home` module. And this Controller is used to handle the request for resource `ticket`.

### Process REST API Requests

After Rest Controller created, you can complete REST API process without writing any code. Resource name and data table name is one-to-one. eg. resource name is `ticket`, then the data table name is `data-table-prefix + ticket`.

### Request Type

REST API gets the current request type from HTTP METHOD by default. eg.the current request type is `DELETE`, which means to delete the resource.

If some clients do not support sending `DELETE` request, you can set the property `_method` to receive request type. eg.

```

export default class extends think.controller.rest {
 init(http){
 super.init(http);
 this._method = '_method'; //specify to get request type from _method in GET params
 }
}

```

### Field Filter

By default, all fields of resource are all returned when accessing it. Sometimes we need to hide part of fields, and we can complete such operations in magic method `__before`.

JavaScript

```
export default class extends think.controller.rest {
 __before(){
 this.modelInstance.fieldReverse('password,score'); //hide password and score fields
 }
}
```

## Authority Management

Some REST APIs require authentication. Only after passing the validation can it obtain the corresponding information. The validation can be carried out in the magic method

`__before`.

JavaScript

```
export default class extends think.controller.rest {
 * __before(){
 let auth = yield this.checkAuth();
 if(!auth){
 return this.fail('no permissions'); // return directly when no permission
 }
 }
}
```

## More Customization

See [API -> controller.rest](#) for more customization ways.

## Crontab

Online projects often need to be timed to execute certain function. By this time, you can use crontab. ThinkJS supports command line calls, combined with the system's crontab function, let you perfectly achieve this type of task.

## Command Line Execution

Besides supporting URL access, ThinkJS also supports command line calls. The usage is as follows.

```
node www/production.js home/index/index
```

Bash

The above command means to execute indexAction of `index` Controller in `home` module.

## Carry Params

If you need to add some params, just put the corresponding params at the end of the line:

```
node www/production.js home/index/index?name=thinkjs
```

Bash

In Action, you can use method `this.get` to get param `name`.

## Modify Request Method

In command line execution, the default request type is GET. If you want to modify it to use other type, you can do it this way:

```
node www/production.js url=home/index/index&method=post
```

Bash

Thus, the request type is changed to post. But in this way, the value of params in url can not include "&" anymore (but you can use "/" to specify params, such as

```
node www/production.js url=home/index/index/foo/bar&method=post).
```

Besides modifying request type, you can also modify the following params.

- `host` modify the request host, default is 127.0.0.1
- `ip` modify request ip, default is 127.0.0.1

## Modify Header

Sometimes, if you want to modify more headers, you can pass a complete json. eg.

```
node www/production.js {"url":"/index/index","ip":"127.0.0.1","method":"POST","headers":{"xxx":"yyyy"}}
```

Bash

## Forbid URL Access

By default, you can access Action that is executed in command line by URL. If forbid URL to access to the Action, you can use `think.cli` to judge. eg.

```
export default class extends think.controller.base {
 indexAction(){
 // forbid URL access to the Action
 if(!think.cli){
 this.fail('only invoked in cli mode');
 }
 ...
 }
}
```

JavaScript

## Executable Script

You can create a simple executable script to call command line to execute. eg.

```
cd project_path;
node www/production.js home/index/index;
```

Bash

This would create the directory `crontab` in the project, and put the above executable script as a file in this directory.

## Timed Execution

Using system crontab can do timed execution. Use command `crontab -e` to edit crontab. eg.

```
0 */1 * * * /bin/sh project_path/crontab/a.sh # execute once per 1 hour
```

Bash

## Use node-crontab Module to Execute Crontab

Besides combining crontab with command line, you can also use `node-crontab` module to execute crontab. eg.

```
import crontab from 'node-crontab';
// execute once per 1 hour
let jobId = crontab.scheduleJob('0 */1 * * *', () => {
 think.http('/home/image/spider', true); //visit /home/image/spier
});
```

JavaScript

Put the above code file in directory `src/common/bootstrap`, so it can be executed automatically when server startup.

## Online Deploy

### Use pm2 to Manage Services

pm2 is a Node.js module used for professionally manage Node.js services, it is highly recommended to use it online. It needs to be installed globally. eg.

`sudo npm install -g pm2`. After installation, the pm2 commands will be available in command line.

When creating project, the config file `pm2.json` will be created in the project directory. And its content is roughly as follows.

```
{
 "apps": [{
 "name": "demo",
 "script": "www/production.js",
 "cwd": "/Users/welefen/Develop/git/thinkjs/demo",
 "max_memory_restart": "1G",
 "autorestart": true,
 "node_args": [],
 "args": [],
 "env": {
 }
 }
]}
```

JavaScript

Modify the `cwd` config value into the real project path, then use the following command to start/restart the service in the project directory.

```
pm2 startOrGracefulReload pm2.json
```

Bash

See <http://pm2.keymetrics.io/docs/usage/application-declaration/> for the detailed config of pm2.

## Use Nginx As a Reverse Proxy

A nginx config file named `nginx.conf` in the project directory will be created when creating project, which content is roughly as follows.

```
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;
 }
}
```

Modify the localhost in `server_name localhost` into the corresponding domain name. Modify the 8360 in `set $node_port 8360` into the one you are using.

After the modification is complete, copy the config file to the config file directory of nginx, then reload the config by the command `nginx -s reload`. So you can access the application through the domain name.

It is recommended to open the config `proxy_on` online, so that you can forbid to access directly by IP + port. Modify the config file `src/common/config/env/production.js`, eg.

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

## The Config of Closing the Static Resource Process

To facilitate the development, ThinkJS supports to process the static resource request. But when code is deployed online, it uses nginx to process the static resource request. By this time, you can close the function of process static resource request to improve performance.

Add the following configuration in the config file `src/common/config/env/production.js`.

```
export default {
 resource_on: false
}
```

## Use Cluster

Enable cluster function online could make the good use of multicore CPU to improve the performance and concurrent processing capability.

You can add the following configuration in the config file `src/common/config/env/production.js`.

```
export default {
 cluster_on: true
}
```

## Developing Plugins

ThinkJS 2.0 supports two kinds of plugins: Middleware and Adapter.

### Creating Plugins

Run the following command to create a plugin, and the plugin name begin with `think-` is suggested:

```
thinkjs plugin think-xxx
```

After running, the `think-xxx` directory will be created, and it may includes the following content:

```
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
```

### Directory Structure

- `src/` place the source code, using ES6/7 features
- `test/` for unit testing
- `.eslintrc` configuration file eslint needed
- `.npmignore` files to ignore when npm launching
- `.travis.yml` configuration file for travis continuous integration
- `package.json` npm configuration file
- `README.md` ReadMe

### Installing Dependencies

```
npm install --verbose
```

### Developing

The code file is `src/index.js`, the generated file by default only contain a basic class, has not inherited any other class yet.

If it is Middleware, you may want to inherit `think.middleware.base`, if it is Adapter, you may want to inherit `think.adapter.base`.

Before developing, run `npm run watch-compile` to make the edited files be compiled in real time.

### Unit Testing

Unit Testing code should be written in `test/index.js`, the test framework is mocha, and run the following command to view the unit testing result:

```
npm run test-cov
```

## About README

After developing and testing, please write the notes in the README.md.

## Publishing

Run `npm publish` to publish the plugin into the npm repository (you may be asked to create account and password if it is your very first publish).

Then, you can inform ThinkJS team. After confirmed, your plugins will be added into the official plugin list and you can get the gift.

## Recommended Modules

---

### Network Request

- superagent
- request

### Log

- log4js

### Date Processing

- moment

### Code Transform

- iconv-lite

### Image Processing

- gm

### Framework

- thinkjs
- express
- koa
- sails

### Debug

- node-inspector

### Unit Testing

- mocha
- istanbul
- muk

### Service Management

- pm2

### Mailing

- nodemailer

### Timed Task

- node-crontab

## More Features

---

### How Encapsulating callback to Promise

Many interfaces Node.js provided are in callback style, and many third party interfaces also do the same. The interfaces provided by ThinkJS are in Promise style instead. So you need encapsulating interfaces in callback style to the ones in Promise style.

Using ThinkJS provided `think.promisify` method can quickly encapsulate interface as Promise, more detail please see [here](#).

## Tasks Queue

Asynchronous I/O is one of the main advantages of Node.js, it make parallel processing very easy, for example we can parallely process multiple files. But OS generally limit the number of opened files, otherwise will result in errors.

In this case, we can resort to tasks queue, and ThinkJS also provide the `think.parallelLimit` method to help us to handle this. More detail please see [here](#).

# API

## think

`think` is a global object that contains lots of useful methods and functions which can use anywhere of your application without `require`.

### Properties

#### think.startTime

The start time of service, a unix timestamp.

#### think.env

The current environment of application, it may be one of the three possable values, you can set it when application start:

- `development` The development environment, application will automatically update itself with modified files.
- `testing` The testing environment.
- `production` The production environment, when application deploy online.

#### think.dirname

The name of project directory, you can set it when application start, the default value is:

```
think.dirname = {
 config: 'config', // The config file directory
 controller: 'controller', // Directory of controller
 model: 'model', // Directory of model
 adapter: 'adapter', // Directory of adapter
 logic: 'logic', // Directory of logic
 service: 'service', // Directory of service
 view: 'view', // Directory of view
 middleware: 'middleware', // Directory of middleware
 runtime: 'runtime', // Directory of runtime
 common: 'common', // Directory of common functions
 bootstrap: 'bootstrap', // the start directory of application
 locale: 'locale' // Directory of locale
}
```

JavaScript

#### think.port

The port of application runs, which can assign before application start, if assigned to some value, application will ignore the port value in the config file.

#### think.cli

Whether application is run under command line mode, false by default. if it is under command line mode, this variable return command arguments. You start command line mode by run:

```
node www/index.js /home/index/test
```

#### think.lang

The current language of system, it read from the environment, which maybe empty in windows system.

#### think.mode

The current mode of application, framework support three mode in project:

- `think.mode_mini` single module mode, all of project is one module.
- `think.mode_normal` multiple modules mode, directory of project is separated to Controller, View, Logic and some other modules.



- `think.mode_module` multiple modules mode, but more stringent separate project with modules than normal mode.

### think.version

The current version of ThinkJS.

### think.module

The list of modules of project, if current mode is `mode_mini`, this variable is an empty array.

### think.THINK\_PATH

The path of ThinkJS code.

### think.THINKLIBPATH

The path where `lib/` of ThinkJS is.

### think.ROOT\_PATH

The root path of project, which is defined in `www/index.js`

### think.APP\_PATH

The path of app directory, which is defined in `www/index.js`

### think.RESOURCE\_PATH

The path of static resource directory, which is defined in `www/index.js`

## Methods

### think.Class(methods, clean)

Create a class dynamically, which inherit from `think.base` by default. you can use `class` to create class in ES6 if project is using ES6.

```
// inherit from think.base
var Cls1 = think.Class({
 getName: function(){
 }
});
```

JavaScript

### Did Not Inherit think.base

```
var Cls2 = think.Class({
 getName: function(){
 }
}, true);
```

JavaScript

### Inherit Other classes

```
// inherit from Cls2
var Cls3 = think.Class(Cls2, {
 init: function(name){
 this.name = name;
 },
 getName: function(){
 }
});
```

JavaScript

### Instantiate a Class

```
// instantiate a class which will call `init` function automatically
var instance = new Cls3('thinkjs');
```

JavaScript

### think.extend(target, source1, source2, ...)

- `target` {Object} directory object
- `source1` {Mixed} source object
- `return` {Object} directory object

It will copy methods or functions from source1, source2 and some other object to `target` object, it is similar to the `$.extend` in `jQuery`.

Deep copy by default, you can assign the first argument to `false` if you want shallow copy.

```
think.extend({}, {name: 'foo'}, {value: 'bar'});
// returns
{name: 'foo', value: 'bar'}
```

JavaScript

### think.isBoolean(obj)

- `obj` {Mixed} object which need to check
- `return` {Boolean}

Check whether this object is Boolean type or not.

```
think.isBoolean(true); //true
think.isBoolean(false); //true
think.isBoolean('string'); //false
```

JavaScript

### think.isNumber(obj)

- `obj` {Mixed} object which need to check
- `return` {Boolean}

Check whether this object is Number type or not.

```
think.isNumber(1); //true
think.isNumber(1.21); //true
```

JavaScript

### think.isObject(obj)

- `obj` {Mixed} object which need to check
- `return` {Boolean}

Check whether this object is object type or not.

```
think.isObject({}); //true
think.isObject({name: "welefen"}); //true
think.isObject(new Buffer('welefen')); //false
```

JavaScript

### think.isString(obj)

- `obj` {Mixed} object which need to check
- `return` {Boolean}

Check whether this object is string type or not.

```
think.isString("xxx"); // true
think.isString(new String("xxx")); //true
```

JavaScript

### think.isFunction(obj)

- `obj` {Mixed} object which need to check
- `return` {Boolean}

Check whether this object is function type or not.

```
think.isFunction(function(){}); //true
think.isFunction(new Function("")); //true
```

JavaScript

### think.isDate(obj)

- `obj` {Mixed} object which need to check

- `return` {Boolean}

Check whether this object is date type or not.

```
think.isDate(new Date()); //true
```

JavaScript

### think.isRegExp(obj)

- `obj` {Mixed} object which need to check
- `return` {Boolean}

Check whether this object is regular expression or not.

```
think.isRegExp(/w+/); //true
think.isRegExp(new RegExp("/w+/")); //true
```

JavaScript

### think.isError(obj)

- `obj` {Mixed} object which need to check
- `return` {Boolean}

Check whether whether this object is error type or not.

```
think.isError(new Error("xxx")); //true
```

JavaScript

### think.isEmpty(obj)

- `obj` {Mixed} object which need to check
- `return` {Boolean}

Check whether this object is empty or not.

```
// check is empty or not
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
```

JavaScript

### think.isArray(obj)

- `obj` {Mixed} object which need to check
- `return` {Boolean}

Check whether this object is array or not.

```
think.isArray([]); //true
think.isArray([1, 2]); //true
think.isArray(new Array(10)); //true
```

JavaScript

### think.isIP4(obj)

- `obj` {Mixed} object which need to check
- `return` {Boolean}

Check whether this object is IP4 or not.

```
think.isIP4("10.0.0.1"); //true
think.isIP4("192.168.1.1"); //true
```

JavaScript

### think.isIP6(obj)

- `obj` {Mixed} object which need to check
- `return` {Boolean}

Check whether this object is IP6 or not.

```
think.isIP6("2031:0000:130f:0000:0000:09c0:876a:130b"); //true
think.isIP6("2031:0000:130f::09c0:876a:130b"); //true
```

JavaScript

### think.isIP(obj)

- **obj** {Mixed} object which need to check
- **return** {Boolean}

Check whether this object is IP or not.

```
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
```

JavaScript

### think.isFile(file)

- **file** {Mixed} object which need to check
- **return** {Boolean}

Check whether this object is IP or not, if file didn't exist, return false.

```
think.isFile("/home/welefen/a.txt"); //true
think.isFile("/home/welefen/dirname"); //false
```

JavaScript

### think.isDir(dir)

- **dir** {Mixed} the path to check
- **return** {Boolean}

Check whether this path is directory or not. if not, return false.

```
think.isDir("/home/welefen/dirname"); //true
```

JavaScript

### think.isBuffer(obj)

- **obj** {Mixed} object which need to check
- **return** {Boolean}

Check whether this object is buffer object or not.

```
think.isBuffer(new Buffer(20)); //true
```

JavaScript

### think.isNumberString(obj)

- **obj** {Mixed} object which need to check
- **return** {Boolean}

Check whether this object is string type of number or not.

```
think.isNumberString(1); //true
think.isNumberString("1"); //true
think.isNumberString("1.23"); //true
```

JavaScript

### think.isPromise(obj)

- **obj** {Mixed} object which need to check
- **return** {Boolean}

Check whether this object is promise object or not.

```
think.isPromise(new Promise(function(){})); //true
think.isPromise(getPromise()); //true
```

JavaScript

### think.isHttp(obj)

- `obj` {Mixed} object which need to check
- `return` {Boolean}

Check whether this object is http object or not.

```
think.isHttp(http); // true
```

JavaScript

### think.isWritable(path)

- `path` {String} the path of directory or file
- `return` {Boolean}

Check whether this file or directory can write or not. if not, return false.

### think.isPrevent(obj)

- `obj` {Mixed} object which need to check
- `return` {Boolean}

Check whether this object is prevent type of promise or not, through `think.prevent()` will generate this type of promise.

### think.mkdir(p, mode)

- `p` {String} the name of directory
- `mode` {Number} the permission of directory , 0777 by default.

Function will create directory recursively, if directory is exist. this function will modify the permission of the directory.

```
// if /home/welefen/a/b/ didn't exist
think.mkdir("/home/welefen/a/b");
think.mkdir("home/welefen/a/b/c/d/e"); // create directory recursively
```

JavaScript

### think.rmdir(p, reserve)

- `p` {String} the path of directory to delete.
- `reserve` {Boolean} whether to keep this directory or not, if value is true, this function will only delete subdirectory.
- `return` {Promise}

Function will delete directory recursively, if directory is not exist, this function will return directly. or this function will return a promise object, then you can use its `then` to operate.

```
function rmTmp(){
 think.rmdir('/foo/bar').then(function(){
 // some operation
 })
}
```

JavaScript

if use `Generator Function`, you can use `yield` :

```
function * rmTmp(){
 yield think.rmdir('/foo/bar');
 // some operation
}
```

JavaScript

### think.chmod(p, mode)

- `p` {String} the path of directory
- `mode` {Number} the permission of directory , 0777 by default.

Change the permission of directory, if directory didn't exist, function will return null directly.

```
think.chmod("/home/welefen/a", 0777);
```

JavaScript

### think.md5(str)

- `str` {String} the string which need to generate md5.
- `return` {String} md5 value

Generate md5 value.

```
think.md5('thinkjs');
// returns
7821eb623e0b1138a47db6a88c3f56bc
```

## think.defer()

- `return` {Object} Deferred object

Create a `Deferred` object, which is a shortcut of `Promise`. Sometimes have to use this function with some operation like `setTimeout`, `event`, though this is not a recommend way.

```
// the way using Deferred
var fn = function(){
var deferred = think.defer();
process.nextTick(function(){
 if(xxx){
 deferred.resolve(data);
 }else{
 deferred.reject(err);
 }
})
return deferred.promise;
}
```

The way using `Deferred` is much cleaner than the way using `new Promise`.

```
// the way using new Promise
var fn = function(){
return new Promise(function(resolve, reject){
 process.nextTick(function(){
 if(xxx){
 resolve(data);
 }else{
 reject(err);
 }
 })
})
}
```

Notice: asynchronous callback operations DONT use the `Deferred` way, instead of encapsulate `callback` to `Promise` with using `think.promisify`.

## think.promisify(fn, receiver)

- `fn` {Function} function which to be promisify
- `receiver` {Object} where is `this` point to.

Encapsulate asynchronous functions to Promise quickly, the last argument of asynchronous functions must be a callback, which has an error handler of first argument.

```
var fs = require('fs');

// function which to get file content
var getContent = function(filePath){
 // encapsulate readFile to Promise
 var readFilePromise = think.promisify(fs.readFile, fs);
 // read file content
 return readFilePromise(filePath, 'utf8');
}

// get file content
getContent('/foo/bar/file.txt').then(function(content){
 console.log(content);
}).catch(function(err){
 console.error(err.stack);
})
```

## think.reject(err)

- `err` {Error} Error object
- `return` {Promise} reject promise

Return a reject promise, and the difference between this and `Promise.reject` is this function will print error message automatically, which can avoid calling `catch` function to

print error message by hand.

```
// use Promise.reject
var fn = function(){
 return Promise.reject(new Error('xxx'));
}
//but need to print error message with `catch` by hand.
fn().catch(function(err){
 console.error(err.stack);
})
```

JavaScript

```
// use think.reject
var fn = function(){
 return think.reject(new Error("xxx"));
}
// will print formatted error message automatically.
fn();
```

JavaScript

## think.co

The alias of modules is [co](#)

## think.lookClass(name, type, module, base)

- **name** {String} class name
- **type** {String} type (controller | model | logic ...)
- **module** {String} module name
- **base** {String} find base class if cannot find module

Find class with type or name of class. if cannot find module, program will find module in common module, if still cannot find module, program will the baseclass.

```
// find user controller in home module
// if cannot find, will find in common module
// if still cannot find, will find in base controller
think.lookClass("user", "controller", "home");

// find user controller in admin module
think.lookClass("admin/user", "controller");
```

JavaScript

## think.getPath(module, type, prefix)

- **module** {String} module name
- **type** {String} type, like controller, model, logic
- **prefix** {String} prefix

Get the path of module based on current project mode.

```
let path = think.getPath('home', 'controller');
```

JavaScript

If root path of current project is `/foo/bar`, then the return path is:

- project mode is `think.mode_mini` then the path is `/foo/bar/app/controller`
- project mode is `think.mode_normal` then the path is `/foo/bar/app/controller/home`
- project mode is `think.mode_module` then the path is `/foo/bar/app/home/controller`

## think.require(name, flag)

- **name** {String}
- **flag** {Boolean}

## think.safeRequire(file)

- **file** {String} the file to load

To load a file safely, if file didn't exist, function will return null, and print error message at the same time.

## think.prevent()

return a special `reject promise`, this promise can stop follow-up work, and not report error.

## think.log(msg, type, showTime)

- `msg` {String | Error} message
- `type` {String} type
- `showTime` {Number | Boolean} whether show time or not.

Print logs, which contains time, type and some other information.

```
think.log('WebSocket Status: closed', 'THINK');
//writes '[2015-09-23 17:43:00] [THINK] WebSocket Status: closed'
```

JavaScript

## Print error message

```
think.log(new Error('error'), 'ERROR');
//writes '[2015-09-23 17:50:17] [Error] Error: error'
```

JavaScript

## Print execute time

```
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'
```

JavaScript

## Don't show log time

```
think.log('/static/module/jquery/1.9.1/jquery.js', 'HTTP', null);
//writes '[HTTP] /static/module/jquery/1.9.1/jquery.js'
```

JavaScript

## Log by custom #####

```
think.log(function(colors){
 return colors.yellow('[WARNING]') + ' test';
});
//writes '[WARNING] test'
```

JavaScript

By the way, `colors` is a [module](#) named `colors` in npm modules.

## think.config(name, value, data)

- `name` {String} config name
- `value` {Mixed} config value
- `data` {Object} config object

Read or setup config, it could be the global config object.

```
// get the config
let value = think.config('name');
// get config in admin module
let value = think.config('name', undefined, 'admin');

// write into config
think.config('name', 'value');
```

JavaScript

## think.getModuleConfig(module)

- `module` {String} module name
- `return` {Object}

Get all config of module, which contains config of module, comon module and the framework default config.

```
// get all config of admin module
let configs = think.getModuleConfig('admin');
```

JavaScript

## think.hook()

Register, get and execute hook, what can be appended or modified if need.

## Get event's middleware list



JavaScript

```
think.hook('view_template');
//returns
['locate_template']
```

### Setup hook

JavaScript

```
// replace ex-hook
think.hook('view_template', ['locate_template1']);

// insert before old one
think.hook('view_template', ['locate_template1'], 'prepend');

// insert after old one
think.hook('view_template', ['locate_template1'], 'append');
```

### Delete hook

JavaScript

```
think.hook('view_template', null);
```

### Execute hook

JavaScript

```
let result = think.hook('view_template', http, data);
//result is a promise
```

## think.middleware()

Register, create, get and execute middleware.

### Create middleware

JavaScript

```
// analyzing XML example
var ParseXML = think.middleware({
 run: function(){
 var http = this.http;
 return http.getPayload().then(function(payload){
 var data = xmlParse.parse(payload); // use a xml parser, this xmlParse here is an example
 http._post = data; // assign parsed data to http._post, then can get data from http._post('xxx')
 });
 }
});
```

### Using ES6 to create middleware

JavaScript

```
let Cls1 = class extends think.middleware.base {
 run(){
 let http = this.http;
 }
}
```

### Register middleware

middleware can be sample function, or complex class.

JavaScript

```
// register a functional middleware
think.middleware('parse_xml', http => {

})
```

JavaScript

```
// redister a class middleware
// it will call run automatically
let Cls = think.middleware({
 run: function(){
 let http = this.http;
 }
});
think.middleware('parse_xml', Cls);
```

## Get middleware

```
let middleware = think.middleware('parse_xml');
```

JavaScript

## Execute middleware

```
let result = think.middleware('parse_xml', http);
//result is a promise
```

JavaScript

## think.adapter()

Create, register, get and execute adapter.

### Create adapter

```
// create an adapter
var Cls = think.adapter({

});

// create a session adapter, which instance of session base class
var Cls = think.adapter('session', 'base', {

});
```

JavaScript

```
// create a session adapter in ES6
let Cls = class extends think.adapter.session {

}
```

JavaScript

### Register adapter

```
// register some type of session adapter
think.adapter('session', 'xxx', Cls);
```

JavaScript

### Get adapter

```
// get file type of session adapter
let Cls = think.adapter('session', 'file');
```

JavaScript

### Execute adapter

```
let Adapter = think.adapter('session', 'file');
let instance = new Adapter(options);
```

JavaScript

## think.gc(instance)

- `instance` {Object} instance of object

Register a instance object to garbage collection queue, the instance object must have `gcType` method and `gc` function.

Something like cache or session which have expiration time, when after expire need to clean up. framework offered some handlers to clean expired file.

```
let Cls = class extends think.adapter.cache {
 init(options){
 super.init(options);
 this.gcType = 'xFileCache';
 think.gc(this);
 }
 gc(){
 // find expired content to clean.
 }
}
```

JavaScript

## think.http(req, res)

- `req` {Object} request object

- `res` {Object} response object
- `return` {Promise}

Base on request and response packed into http object, by the way, req and res could be other object by custom.

```
// based on an url object packed into a http object, which is useful to command mode calling.
think.http('/index/test').then(http => {

});
```

JavaScript

### think.uuid(length)

- `length` {Number} the length of generate string, 32 by default

Generate a random string.

### think.session(http)

- `http` {Object} http object

Generate a session, and write it to http object, if exist, return directly.

### think.controller()

Create and execute a controller

#### Create controller

```
// create controller, instance of think.controller.base
let C1s = think.controller({

})
// create controller, instance of think.controller.rest
let C1s = think.controller('rest', {

})
```

JavaScript

```
// create a controller by using ES6
let C1s1 = class extends think.controller.base {

}
```

JavaScript

#### Instance of controller

```
// instance of user controller belong to home module
let instance = think.controller('user', http, 'home');
```

JavaScript

### think.logic()

Create and execute logic

#### Create logic

```
// create logic, which instance of think.logic.base
let C1s = think.logic({

})
```

JavaScript

```
// create logic by using ES6
let C1s1 = class extends think.logic.base {

}
```

JavaScript

#### Instance of logic

```
// instance of user logic which is belong to home
let instance = think.logic('user', http, 'home');
```

JavaScript

## think.model()

Create or get model.

### Create model

```
// Create a model
let model = think.model({
 getList: function(){
 }
});

// in ES6 , instance of think.model.base class directly
let model = class extends think.model.base {
 getList(){
 }
}

// create a model which instance of mongo model
let model = think.model('mongo', {
 getList: function(){
 }
});
// in ES6, instance of think.model.mongo class directly
let model = class extends think.model.mongo {
 getList(){
 }
}
```

JavaScript

### get the instance of model

```
let configs = {
 host: '127.0.0.1',
 name: 'user'
}
// get user model which is belong to home module.
let instance = think.model('user', configs, 'home');
```

JavaScript

## think.service()

Create or get service.

### Create service #####

```
// Create a service class
let service = think.service({
})

// in ES6 , instance of think.service.base class directly
let service = class extends think.service.base {
}
```

JavaScript

service base class based on [think.base](#), so can use functions in think.base.

if don't want to write service to class, so it's not necessary to create by using this way.

### get service

```
// get post service which belong to home module, passed a `{}`
// if got service is a class, it will be instancing automatically
think.service('post', {}, 'home');
```

JavaScript

## think.cache(name, value, options)

- `name` {String} cache key

- `value` {Mixed} cache value
- `options` {Object} cache options
- `return` {Promise} return a Promise

Get, setup or delete cache, value is `undefined` means read cache, value is `null` means delete cache.

if value assigned to `Function` means read cache but when cannot got a result, this function will be calling, then return the function return value which has been setup to cache.

```

// get cache
think.cache('name').then(data => {});

// setup the type of cache, read cache from redis for example
think.cache('name', undefined, {type: 'redis'});

// if cache userList is not exist, then query the database, assign return value to cache
think.cache('userList', () => {
 return think.model('user').select();
});

// setup cache
think.cache('name', 'value');

// delete cache
think.cache('name', null);

```

JavaScript

### think.locale(key, ...data)

- `key` {String} the key which need to get
- `data` {Array} arguments

Get the corresponding value based on language, the current language can get from `think.lang`, which can setup when system start.

```

think.locale('CONTROLLER_NOT_FOUND', 'test', '/index/test');
//returns
'controller `test` not found. url is `/index/test`.'

```

JavaScript

### think.validate()

Register, get or execute validation.

#### register validate function

```

// register the validate type is not_number
think.validate('not_number', value => {
 return !(/^d+$/).test(value);
})

```

JavaScript

#### get validate function

```

let fn = think.validate('not_number');

```

JavaScript

#### validate data

```

let result = think.validate({
 name: {
 value: 'name',
 required: true,
 not_number: true
 },
 pwd: {
 value: 'xxx',
 required: true,
 minLength: 6
 }
});
// if result is isEmpty, it means result is expected.
if(think.isEmpty(result)){
}

```

JavaScript

### think.await(key, callback)

- `key` {String}
- `callback` {Function}

Execute `await`, to avoid a long-running operation has been called many times,

For example, one user request that get data from a remote interface can not be processed in time will result in a large number of similar requests, it's a wasting of resources. So these users can share a common request to the remote interface.

```
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);
 }
}
```

JavaScript

### think.npm(pkg)

- `pkg` {String} module name

Load module, if module not exist, module will be install automatically.

```
// if mysql module exist, project will install it with npm.
let mysql = think.npm('mysql');
```

JavaScript

```
// load a specify version of mysql
let mysql = think.npm('mysql@2.0.0')
```

JavaScript

### think.error(err, addon)

- `err` {Error | Promise | String} error information
- `addon` {Error | String} addon error message.

Formatting error message, make some system error message completely.

```
let error = think.error(new Error('xxx'));
```

JavaScript

### Catch promise error message

```
let promise = Project.reject(new Error('xxx'));
promise = think.error(promise)
```

JavaScript

Add catch for promise automatically, to catch error message.

### think.statusAction(status, http, log)

- `status` {Number} status number
- `http` {Object} contained http object
- `log` {Boolean} whether log error message or not

When system is abnormal like system error, page not found, permission denied, then render the right page.

while creating project, it will generate file `src/common/controller/error.js` in common module, which is specially use for handle error state.

Default support types of error are: `400` , `403` , `404` , `500` , `503` .

According to the project's need, it can be modified like error page or extension.

```
export default class extends think.controller.base {
 indexAction(){
 if(xxxx){
 let error = new Error('not found');
 // assign error information to http object, to render with template
 this.http.error = error;
 return think.statusAction(404, this.http);
 }
 }
}
```

## Class

### think.base

think.base: More information read [here](#)

### think.http.base

think.http.base: More information read [here](#)

## think.base

`think.base` is the base class, all classes will inherit it, it supports some basic functions.

Inherit Base Class with ES6:

```
export default class extends think.base {
 /**
 * init method
 * @return {} []
 */
 init(){
 }
}
```

**Notice** : while using ES6, don't write the `constructor`, instead putting some initial operations in the function `init`, this function will be called automatically when class instancing, the effect is equivalent to use `constructor`.

Inherit Base Class Use Normal Way:

```
module.exports = think.Class(think.base, {
 /**
 * init method
 * @return {} []
 */
 init: function(){
 }
})
```

### init(...args)

- `args` {Array}

Initialization function, which can do some assign and other operations.

```
class a extends think.base {
 init(name, value){
 this.name = name;
 this.value = value;
 }
}
```

**Notice** : Different from version `1.x`, the `init` function of `2.x` did not return a `Promise`, some common operations are in `__before` magic functions.

### \_\_before()

Pre-magic function, you can put some common behavior here, such as check whether user is login or not in controller.

```
export default class think.controller.base {
 /**
 * pre-magic function
 * @return {Promise} []
 */
 * __before(){
 let userInfo = yield this.session('userInfo');
 // if not login yet, it will jump to login page.
 if(think.isEmpty(userInfo)){
 return this.redirect('/login');
 }
 this.assign('userInfo', userInfo)
 }
}
```

JavaScript

## \_\_after()

Post-magic function, it will execute after function executed.

## filename()

- **return** {String} return the current class file's name.

Get the current class file's name, not contains detail of file path or file's extension.

```
// suppose current class file path is /home/xxx/project/app/controller/user.js
class a extends think.base {
 test(){
 var filename = this.filename();
 //returns 'user'
 }
}
```

JavaScript

## invoke(method, ...data)

- **method** {String} the function name to be invoked
- **data** {Array} arguments
- **return** {Promise}

To invoke a function, automatically invoke `__before` and `__after` no matter whether the function return `Promise` or not, this function will return `Promise`.

This function supports `*/yield` and `async/await`.

```
//use 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 => {

});
```

JavaScript

```
//use */yield
class Cls extends think.base {
 * getValue(){
 let value = yield this.getValue();
 return value;
 }
}
let instance = new Cls();
instance.invoke('getValue').then(data => {

});
```

JavaScript

## think.http.base



The `think.http.base` class inherit from `think.base`, it is the base class that contains all of the operations related to http. Middleware, controller and view class are all inherit from this class.

### Inheritance with ES6:

```
export default class extends think.http.base {
 /**
 * initial function, will automatically invoked while instacing, didn't need constructor.
 * @return {}
 */
 init(){
 }
}
```

JavaScript

### Inheritance With Normal Way

```
module.exports = think.Class(think.http.base, {
 init: function(){
 }
});
```

JavaScript

## Property

### http

Packaged http object, contained methods and function to be seen in [API -> http](#).

## Methods

### config(name, value)

- `name` {String} config file
- `value` {Mixed} config value

Read or setup config, it is read config when value assigned to `undefined`, otherwise it is setup config.

This function can not only read system default config, but also read project config.

**Notice**: Don't setup with request user's information, it will be covered by other user.

```
export default class extends think.controller.base {
 indexAction(){
 // get config value
 let value = this.config('name');
 }
}
```

JavaScript

### action(controller, action)

- `controller` {Object | String} controller instance
- `action` {String} action name
- `return` {Promise}

Invoke action in controller, return a Promise, invoke `__before` and `__after` automatically.

If controller is a string, it will automatically to find this controller.

```
// invoke action of current module's controller
export default class extends think.controller.base {
 * indexAction(){
 // invoke default function in user controller
 let value = yield this.action('user', 'detail');
 }
}
```

JavaScript

```
// invoke action in cross module's controller
export default class extends think.controller.base {
 * indexAction(){
 // invoke detail function of user controller in admin module
 let value = yield this.action('admin/user', 'detail');
 }
}
```

### cache(name, value, options)

- **name** {String} cache name
- **value** {Mixed | Function} cache value
- **options** {Object} cache options, more information in cache config.

Read or set cache, it is read cache when assign **value** to **undefined**, otherwise, it is setup cache. default type is **file**.

```
export default class extends think.controller.base {
 * indexAction(){
 // get cache
 let value = yield this.cache('name');
 }
}
```

When **value** is function, it means read cache, if cache's value didn't exist, it will invoke this function, and assign the returning value to cache and return the value. It is very useful to avoid a trouble which judge the cache is exist when developing project and then to read cache and set cache in other place.

```
export default class extends think.controller.base {
 * indexAction(){
 // setup cache, when cache didn't exist, it invoke function automatically, and set cache at the same time
 let value = yield this.cache('name', () => {
 return this.model('user').select();
 });
 }
}
```

Setup cache and modify the type:

```
export default class extends think.controller.base {
 * indexAction(){
 // setup cache, cache type is redis
 yield this.cache('name', 'value', {
 type: 'redis'
 });
 }
}
```

### hook(event, data)

- **event** {String} event name
- **data** {Mixed} argument
- **return** {Promise}

Execute hook event, a hook has some middleware, it will execute those middleware orderly.

Hook event can be assigned in `src/common/config/hook.js`, also it can be registered with think.hook.

```
export default class extends think.controller.base {
 * indexAction(){
 let result = yield this.hook('parse_data');
 }
}
```

### model(name, options)

- **name** {String} model name
- **options** {Object} options, more detail seen in database config
- **return** {Object} model instance

Get the instance of model, which is instance of current module by default, it also can get instance of other module.

```
export default class extends think.controller.base {
 indexAction(){
 // get instance of user model in current module
 let model = this.model('user');
 // get instance of article model in admin module
 let model1 = this.model('admin/article');
 // get instance of test model in current module, and it is sqlite database
 let model2 = this.model('test', {
 type: 'sqlite' // setup type of database to sqlite, more detail to see in database config
 })
 }
}
```

### controller(name)

- **name** {String} controller name
- **return** {Object} controller instance

Get the instance of Controller, if cannot find Controller, it will report errors.

```
export default class extends think.controller.base {
 indexAction(){
 // get instance of user controller in current module
 let controller = this.controller('user');
 // get instance of user controller in admin module
 let controller1 = this.controller('admin/user');
 }
}
```

### service(name)

- **name** {String} service name
- **return** {Class}

Get the service, it maybe return a class, or an object, so it will not instance automatically.

```
export default class extends think.controller.base {
 indexAction(){
 // get the service
 let service = this.service('user');
 // get instance of service
 let instance = new service(...args);
 // get user service in admin module
 let service1 = this.service('admin/user');
 }
}
```

## http

This http object is not the one in Node.js, it is a new object which packaged with request object and response object.

```
var http = require('http');

http.createServer(function (request, response) {
 response.writeHead(200, {'Content-Type': 'text/plain'});
 response.end('Hello World\n');
}).listen(8124);
```

As the above code shows, when Node.js create service, it will pass request and response to callback. For the convenience of invoke, ThinkJS packaged these two objects into its own http object, and offer some useful functions.

The `http` object will be passed to middleware, logic, controller and view.

**Note** : http object is an instance of EventEmitter, so you register event listeners to it.

### Properties

#### http.req

System native request object.

### **http.res**

System native response object.

### **http.startTime**

A start time of request, it is a `unix` timestamp.

### **http.url**

Url of current request.

### **http.version**

Http version of current request.

### **http.method**

Type of current request.

### **http.headers**

Header informations of current request.

### **http.pathname**

Pathname of current request, router depended on it's value and will change it in some operations. so the return value of action maybe different from the initial value.

### **http.query**

Query data of current request.

### **http.host**

Host of current request, contain host port.

### **http.hostname**

Host of current request, not contain host port.

### **http.payload**

Payload data of current request, it has data only if the request is submit type.

### **http.\_get**

Store GET arguments.

### **http.\_post**

Store POST arguments.

### **http.\_file**

Store upload file data.

### **http.\_cookie**

Store cookie data.

### **http.module**

The module name of current request parsed.

### **http.controller**

The controller name of current request parsed.

### **http.action**

The action name of current request parsed.

## Methods

### http.getPayload()

- `return` {Promise} payload data

Get payload data.

### http.config(name)

- `name` {String} config name
- `return` {Mixed} return config value

Get the argument of current request config.

### http.referrer()

- `return` {String} referrer of request

Return the referrer of current request.

### http.userAgent()

- `return` {String} userAgent of request

Return the userAgent of current request.

### http.isGet()

- `return` {Boolean}

Return current request is GET request or not.

### http.isPost()

- `return` {Boolean}

Return current request is POST request or not.

### http.isAjax(method)

- `method` {String} type of request
- `return` {Boolean}

Return current request is Ajax request or not.

```
http.isAjax(); // judge request is ajax request or not
http.isAjax('GET'); // judge request is ajax request and is GET type or not
```

JavaScript

### http.isJsonp(name)

- `name` {String} callback parameter name, default is callback
- `return` {Boolean}

Return current request is jsonp request or not.

```
//url is /index/test?callback=testxxx
http.isJsonp(); //true
http.isJsonp('cb'); //false
```

JavaScript

### http.get(name, value)

- `name` {String} parameter name
- `value` {Mixed} parameter value

Get or set GET parameter, it can be used to set GET argument for somewhere can get it.

```
// 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} parameter name
- **value** {Mixed} parameter value

Get or set POST parameter, it can be used to set POST argument for somewhere can get it.

```
http.post('email'); // get the submitted email
```

### http.param(name)

- **name** {String} parameter name
- **return** {Mixed}

Get parameter value, firstly to get from POST, if return null, it will get the value from URL parameter.

### http.file(name)

- **name** {String} field name
- **return** {Object}

Get the uploaded file.

```
http.file('image');
//returns
{
 fieldName: 'image', // the filed name in form
 originalFilename: filename, // origin file name
 path: filepath, // the temp path of store files
 size: size // file size
}
```

### http.header(name, value)

- **name** {String} header name
- **value** {String} header value

Get or set header information.

```
http.header('accept'); // get accept
http.header('X-NAME', 'thinkjs'); // set header
```

### http.expires(time)

- **time** {Number} expire time, unit is second.

Strange cache, set **Cache-Control** and **Expires** header information.

```
http.header(86400); // set expire time is one day.
```

### http.status(status)

set status code, if header has sent, it cannot set status code.

```
http.status(400); // set status code to 400
```

### http.ip()

Get user's ip, it will be incorrect if user used proxy.

### http.lang(lang, asViewPath)

- `lang` {String} the setup of language.
- `asViewPath` {Boolean} whether add a directory layer for language template.

Get or set global language, it support more directory layer for language template.

### Get language

```
let lang = http.lang();
```

JavaScript

The order to get language is `http._lang` -> `get from cookie` -> `get from header`, if need to parse language from url, you can set `http._lang` with `http.lang(lang)` after get url.

### set language

```
let lang = getFromUrl();
http.lang(lang, true); // set language, and set a directory layer for language template.
```

JavaScript

### http.theme(theme)

Get or set theme, after setting, it will generate a lay for theme.

### http.cookie(name, value)

- `name` {String} cookie name
- `value` {String} cookie value

Read or set cookie.

```
http.cookie('think_test'); // get cookie named think_test
http.cookie('name', 'value'); // get cookie, invalid if header has sent.
```

JavaScript

### http.session(name, value)

- `name` {String} session name
- `value` {Mixed} session value
- `return` {Promise}

Read, set and clean session.

### Read Session

```
let value = yield http.session('userInfo');
```

JavaScript

### set Session

```
yield http.session('userInfo', data);
```

JavaScript

### clean Session

```
yield http.session();
```

JavaScript

### http.redirect(url, status)

- `url` {String} the url will jump
- `status` {Number} status code, 301 or 302, default is 302.

Jump page.

```
http.redirect('/login'); // jump to login page.
```

JavaScript

### http.type(contentType, encoding)

- `contentType` {String} contentType which need to modify
- `encoding` {String} encode to set

Read or set Content-Type.

```
http.type(); // get Content-Type
http.type('text/html'); // get Content-Type, it will add charset automatically
http.type('audio/mpeg', false); // set Content-Type, not add charset
```

### http.write(content, encoding)

- `content` {Mixed} the content to write
- `encoding` {String} charset

Write content, end request only invoke `http.end`.

### http.end(content, encoding)

- `content` {Mixed} the content to write
- `encoding` {String} charset

Write content and stop current request.

### http.success(data, message)

- `data` {Mixed} the content to write
- `message` {String} added message

Response a format normal data , always after operate success.

```
http.success({name: 'thinkjs'});
//writes
{
 errno: 0,
 errmsg: '',
 data: {
 name: 'thinkjs'
 }
}
```

Client can based on `error` is `0` or not to judge current request is success.

### http.fail(errno, errmsg, data)

- `errno` {Number} error number
- `errmsg` {String} error message
- `data` {Mixed} extra data

Output an unusual formatted data, normally after operate failed.

**Notice** : field name `errno` and `errmsg` can be modified in config.

```
http.fail(100, 'fail')
//writes
{
 errno: 100,
 errmsg: 'fail',
 data: ''
}
```

In this way, client will get detail error number and error message, then show message according to the need.

**Notice** : field name `errno` and `errmsg` can be modified in config.

### http.json(data)

- `data` {Object}

Output data in json way, it will set Content-Type to `application/json` , its config is `json_content_type` .

## controller

The `think.controller.base` class inherit from `think.http.base` class, controllers in project need to inherit it.

### Inheritance with ES6:



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

## Inheritance With Normal Way

```
module.exports = think.controller({
 indexAction(){
 }
})
```

## Property

### controller.http

Passed [http](#) object.

## Methods

### controller.ip()

- `return` {String}

Get user ip of current request, it is equal to `http.ip`.

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

### controller.method()

- `return` {String}

Get type of current request, and convert to lowercase.

```
export default class extends think.controller.base {
 indexAction(){
 let method = this.method(); //get or post ...
 }
}
```

### controller.isMethod(method)

- `method` {String} method
- `return` {Boolean}

Judge type of current request is named types.

### controller.isGet()

- `return` {Boolean}

Judge is GET request or not.

### controller.isPost()

- `return` {Boolean}

Judge is POST request.

### controller.isAjax(method)

- `method` {String}
- `return` {Boolean}

Judge is Ajax request, if named method, then as same as the type of request.

```
export default class extends think.controller.base {
 indexAction(){
 // is ajax and request type is POST
 let isAjax = this.isAjax('post');
 }
}
```

JavaScript

### controller.isWebSocket()

- `return` {Boolean}

Whether is websocket request or not.

### controller.isCli()

- `return` {Boolean}

Whether is run in command mode or not.

### controller.isJsonp(callback)

- `callback` {String} callback name
- `return` {Boolean}

Whether is jsonp request.

### controller.get(name)

- `name` {String} parameter name

Get parameter of GET.

```
export default class extends think.controller.base {
 indexAction(){
 // get a parameter
 let value = this.get('xxx');
 // get all parameter
 let values = this.get();
 }
}
```

JavaScript

### controller.post(name)

- `name` {String} parameter name

Get parameter of POST data.

```
export default class extends think.controller.base {
 indexAction(){
 // get a value of parameter
 let value = this.post('xxx');
 // get all parameter of POST
 let values = this.post();
 }
}
```

JavaScript

### controller.param(name)

- `name` {String} parameter name

Get parameter value, first to read from POST, if return null, then get from GET.

### controller.file(name)

- `name` {String} field name of upload file

Get uploaded file, return value is a object, contains these method below:

```
{
 fieldName: 'file', // field name
 originalFilename: filename, // original file name
 path: filepath, // path of temp store file, need to move this path when using, or exists until request ends.
 size: 1000 // file size
}
```

If file not exist, this returning is an empty object `{}`.

### controller.header(name, value)

- `name` {String} header name
- `value` {String} header value

Get or set header.

```
export default class extends think.controller.base {
 indexAction(){
 let accept = this.header('accept'); // get header
 this.header('X-NAME', 'thinks'); // set header
 }
}
```

### controller.expires(time)

- `time` {Number} expires time, the unit is seconds

Strong cache, set `Cache-Control` and `Expires` header information.

```
export default class extends think.controller.base {
 indexAction(){
 this.expires(86400); // set expire time to one day.
 }
}
```

### controller.userAgent()

Get userAgent.

### controller.referrer(onlyHost)

- `referrer` {Boolean} whether only need host

Get referrer.

### controller.cookie(name, value, options)

- `name` {String} cookie name
- `value` {String} cookie value
- `options` {Object}

Get or set cookie.

```
export default class extends think.controller.base {
 indexAction(){
 // get value of cookie
 let value = this.cookie('think_name');
 }
}
```

```
export default class extends think.controller.base {
 indexAction(){
 // get value of cookie
 this.cookie('think_name', value, {
 timeout: 3600 * 24 * 7 // expires time is one week
 });
 }
}
```

### controller.session(name, value)

- `name` {String} session name
- `value` {Mixed} session value
- `return` {Promise}

Read, set and clean session.

#### Read Session

```
export default class extends think.controller.base {
 * indexAction(){
 // read session
 let value = yield this.session('userInfo');
 }
}
```

JavaScript

#### set Session

```
export default class extends think.controller.base {
 * indexAction(){
 //set session
 yield this.session('userInfo', data);
 }
}
```

JavaScript

#### Clean Session

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

JavaScript

### controller.lang(lang, asViewPath)

- `lang` {String} the setup of language
- `asViewPath` {Boolean} whether add a directory layer for language template.

Read or set language.

### controller.locale(key)

- `key` {String}

Based on language to get the language version.

### controller.redirect(url, statusCode)

- `url` {String} the url to jump
- `statusCode` {Number} status code, default is 302

Page jump.

### controller.assign(name, value)

- `name` {String | Object} variable name
- `value` {Mixed} variable value

Assign variable into template.

```
export default class extends think.controller.base {
 indexAction(){
 // single assign
 this.assign('title', 'thinkjs');
 // multi-assign
 this.assign({
 name: 'xxx',
 desc: 'yyy'
 })
 }
}
```

### controller.fetch(templateFile)

- `templateFile` {String} tempate file path
- `return` {Promise}

Get the parsed template content.

### Get directly #####

```
// suppose the file path is /foo/bar/app/home/controller/index.js
export default class extends think.controller.base {
 * indexAction(){
 // home/index_index.html
 let content = yield this.fetch();
 }
}
```

### Change action #####

```
// suppose file path is /foo/bar/app/home/controller/index.js
export default class extends think.controller.base {
 * indexAction(){
 // home/index_detail.html
 let content = yield this.fetch('detail');
 }
}
```

### Change controller and action #####

```
// suppose file path is /foo/bar/app/home/controller/index.js
export default class extends think.controller.base {
 * indexAction(){
 // home/user_detail.html
 let content = yield this.fetch('user/detail');
 }
}
```

### Change module, controller 和 action #####

```
// suppose file path is /foo/bar/app/home/controller/index.js
export default class extends think.controller.base {
 * indexAction(){
 // admin/user_detail.html
 let content = yield this.fetch('admin/user/detail');
 }
}
```

### Change file extension #####

```
// suppose file path is /foo/bar/app/home/controller/index.js
export default class extends think.controller.base {
 * indexAction(){
 // home/index_detail.xml
 let content = yield this.fetch('detail.xml');
 }
}
```

## Get absolute file path #####

```
// suppose file path is /foo/bar/app/home/controller/index.js
export default class extends think.controller.base {
 * indexAction(){
 // /home/xxx/aaa/bbb/c.html
 let content = yield this.fetch('/home/xxx/aaa/bbb/c.html');
 }
}
```

JavaScript

## controller.display(templateFile)

- `templateFile` {String} template file path

Output template content to browser side. strategy of finding template is the same as `controller.fetch`.

## controller.jsonp(data)

- `data` {Mixed} content to output

Using the way of jsonp to output content, after getting callback's name and security filter then output.

```
export default class extends think.controller.base {
 indexAction(){
 this.jsonp({name: 'thinkjs'});
 //writes
 'callback_fn_name({name: "thinkjs"})'
 }
}
```

JavaScript

## controller.json(data)

- `data` {Mixed} the output content

Json way to output.

## controller.status(status)

- `status` {Number} status code, default is 404

Set status code.

## controller.deny(status)

- `status` {String} status code, default is 403

Deny current request.

## controller.write(data, encoding)

- `data` {mixed} the output content
- `encoding` {String} charset

Output content.

## controller.end(data, encoding)

- `data` {mixed} the output content
- `encoding` {String} charset

After output content, end current request.

## controller.type(type, charset)

- `type` {String} Content-Type
- `charset` {Boolean} whether append charset or not

Set Content-Type.

## controller.download(filePath, contentType, fileName)

- `filePath` {String} specified path of download file

- `content-Type` {String} Content-Type
- `fileName` {String} error file name

Download file.

```
export default class extends think.controller.base {
 indexAction(){
 let filePath = think.RESOUC_PATH + '/a.txt';
 // auto identify Content-Type, save file to a.txt
 this.download(filePath);
 }
}
```

JavaScript

```
export default class extends think.controller.base {
 indexAction(){
 let filePath = think.RESOUC_PATH + '/a.log';
 // auto identify Content-Type, save file to b.txt
 this.download(filePath, 'b.txt');
 }
}
```

JavaScript

```
export default class extends think.controller.base {
 indexAction(){
 let filePath = think.RESOUC_PATH + '/a.log';
 // specify Content-Type to text/html, save file to b.txt
 this.download(filePath, 'text/html', 'b.txt');
 }
}
```

JavaScript

### controller.success(data, message)

- `data` {Mixed} the output data
- `message` {String} appended message

Output an normal formatted data, often after operate success.

```
http.success({name: 'thinkjs'});
//writes
{
 errno: 0,
 errmsg: '',
 data: {
 name: 'thinkjs'
 }
}
```

JavaScript

Client can based on `error` is `0` or not to judge current request is success.

### controller.fail(errno, errmsg, data)

- `errno` {Number} error number
- `errmsg` {String} error message
- `data` {Mixed} extra data

Output an unusual formatted data, normally after operate failed.

**Notice** : field name `errno` and `errmsg` can be modified in config.

```
http.fail(100, 'fail')
//writes
{
 errno: 100,
 errmsg: 'fail',
 data: ''
}
```

JavaScript

In this way, client will get detail error number and error message, then show message according to the need.

**Notice** : filed name `errno` and `errmsg` can be modified in config.

### controller.sendTime(name)

- `name` {String} header key

The execute time of send request, send with header.

## rest controller

The `think.controller.rest` class inherit from [think.controller.base](#), used for handle Rest API.

### Inheritance with ES6:

```
export default class extends think.controller.rest {
 }
}
```

JavaScript

### Inheritance With Normal Way

```
module.exports = think.controller('rest', {
 })
```

JavaScript

## Properties

### controller.isRest

Identify this controller is Rest api. if in `init` function, it assigned to `false`, and this controller is not a Rest interface no more.

### controller.\_method

The way to get method, by default read from http method, but some client don't support send some request type like DELETE, PUT, so it can set to get from GET parameter.

```
export default class extends think.controller.rest {
 init(http){
 super.init(http);
 // set _method, means get _method field value from GET parameters
 // if is null, it will get from http method
 this._method = '_method';
 }
}
```

JavaScript

### controller.resource

The Resource name of current Rest

### controller.id

Resource ID

### controller.modelInstance

The instance model of resource.

## Methods

### controller.\_\_before()

It can do some operate like filter field, pagination, access control in magic function `__before`.

```
export default class extends think.controller.rest{
 __before(){
 // filter password field
 this.modelInstance.field('password', true);
 }
}
```

JavaScript

### controller.getAction()

Get resource data, if id exist, then get one, or get the list.



```
// function implementation, it can be modified if need.
export default class extends think.controller.rest {
 * getAction(){
 let data;
 if (this.id) {
 let pk = yield this.modelInstance.getPk();
 data = yield this.modelInstance.where({[pk]: this.id}).find();
 return this.success(data);
 }
 data = yield this.modelInstance.select();
 return this.success(data);
 }
}
```

### controller.postAction()

Add data.

```
// function implementation, it can be modified if need.
export default class extends think.controller.rest {
 * postAction(){
 let pk = yield this.modelInstance.getPk();
 let data = this.post();
 delete data[pk];
 if(think.isEmpty(data)){
 return this.fail('data is empty');
 }
 let insertId = yield this.modelInstance.add(data);
 return this.success({id: insertId});
 }
}
```

### controller.deleteAction()

Delete data.

```
// function implementation, it can be modified if need.
export default class extends think.controller.rest {
 * deleteAction(){
 if (!this.id) {
 return this.fail('params error');
 }
 let pk = yield this.modelInstance.getPk();
 let rows = yield this.modelInstance.where({[pk]: this.id}).delete();
 return this.success({affectedRows: rows});
 }
}
```

### controller.putAction()

Update data.

```
// function implementation, it can be modified if need.
export default class extends think.controller.rest {
 * putAction(){
 if (!this.id) {
 return this.fail('params error');
 }
 let pk = yield this.modelInstance.getPk();
 let data = this.post();
 delete data[pk];
 if (think.isEmpty(data)) {
 return this.fail('data is empty');
 }
 let rows = yield this.modelInstance.where({[pk]: this.id}).update(data);
 return this.success({affectedRows: rows});
 }
}
```

### controller.\_\_call()

Invoked when cannot find function

```
export default class extends think.controller.rest {
 __call(){
 return this.fail(think.locale('ACTION_INVALID', this.http.action, this.http.url));
 }
}
```

JavaScript

## model

The `think.model.base` class inherit from `think.base` class.

### Inheritance with ES6:

```
export default class extends think.model.base {
 getList(){
 }
}
```

JavaScript

### Inheritance With Normal Way

```
module.exports = think.model({
 getList: function(){
 }
})
```

JavaScript

## Properties

### model.pk

The primary key of database, default is `id`.

### model.name

Model name, default is current file name.

Suppose current file path is `for/bar/app/home/model/user.js`, then the model name is `user`.

### model.tablePrefix

The Prefix of table in database, default is `think_`.

### model.tableName

The name of data table, not contains prefix name, default equals to model name.

### model.fields

The fields of data table, auto analyse the data table.

### model.indexes

The indexes of data table, auto analyse the data table.

### model.readonlyFields

The readonly fields list, when data updated, these fields will not been updated.

### model.config

Config, specify when instancing.

### model.\_db

Handler of connect database.

### model.\_data

Data of operation.

## model.\_options

Options of operation.

## Methods

### model.model(name, options, module)

- **name** {String} model name
- **options** {Object} confing options
- **module** {String} module name
- **return** {Object}

Get instance of model, it can read cross module.

```
export default class extends think.model.base {
 * getList(){
 // get instance of user model
 let instance = this.model('user');
 let list = yield instance.select();
 let ids = list.map(item => {
 return item.id;
 });
 let data = yield this.where({id: ['IN', ids]}).select();
 return data;
 }
}
```

JavaScript

### model.getTablePrefix()

- **return** {string}

Get the prefix of table.

### model.getConfigKey()

- **return** {String}

Get config key, use it when cache db handler.

### model.db()

- **return** {Object}

Based on current config to get instance of db, if exist, return directly.

### model.getModelName()

- **return** {String} model name

Return directly if configed, or parse current file name.

### model.getTableName()

- **return** {String} get table name, contains prefix

Get table name, contains prefix.

### model.cache(key, timeout)

- **key** {String} cache key
- **timeout** {Number} cache expire time, the unit is seconds.
- **return** {this}

Set cache config.

### Set key and time of cache

```
export default class extends think.model.base {
 getList(){
 return this.cache('getList', 1000).where({id: {'>': 100}}).select();
 }
}
```

### Only set cache time, cache key auto generate

```
export default class extends think.model.base {
 getList(){
 return this.cache(1000).where({id: {'>': 100}}).select();
 }
}
```

### Set more cache information

```
export default class extends think.model.base {
 getList(){
 return this.cache({
 key: 'getList',
 timeout: 1000,
 type: 'file' // use file cache
 }).where({id: {'>': 100}}).select();
 }
}
```

### model.limit(offset, length)

- **offset** {Number} set the start position of query
- **length** {Number} set the length of query
- **return** {this}

Set the limit of query result.

### Set length of data

```
export default class extends think.model.base {
 getList(){
 // query twenty data
 return this.limit(20).where({id: {'>': 100}}).select();
 }
}
```

### Limit data start position and length

```
export default class extends think.model.base {
 getList(){
 // start from position 100, query twenty data
 return this.limit(100, 20).where({id: {'>': 100}}).select();
 }
}
```

### model.page(page, listRows)

- **page** {Number} current page, start with one
- **listRows** {Number} number of per page
- **return** {this}

Set query pagination data, convert to **limit** data automatically.

```
export default class extends think.model.base {
 getList(){
 // query the second page data, ten data of per page.
 return this.page(2, 10).where({id: {'>': 100}}).select();
 }
}
```

### model.where(where)

- `where` {String | Object} where condition
- `return` {this}

Set where query condition, it can set logic with method `__logic`, default is `AND`. Mulpty query with method `__complex`.

**Noatice** : 1. example below don't suit for mengo model.in mongo, seting where condition to seen in model.mongo. 2.where condition need to been validated in Logic, or maybe cause some bug.

### Normal condition

```
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();
 }
}
```

JavaScript

### null condition

```
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();
 }
}
```

JavaScript

### EXP condition

ThinkJS will transfer field and value by default for security bugs. sometimes, if not want to transfer in some special case, you can use EXP way, like:

```
export default class extends think.model.base {
 where1(){
 //SELECT * FROM `think_user` WHERE ((`name` ='name'))
 return this.where({name: ['EXP', "=\`name\`"]}).select();
 }
}
```

JavaScript

### LIKE condition

```

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 mult-value
 where3(){
 //SELECT * FROM `think_user` WHERE ((`title` LIKE 'welefen' OR `title` LIKE 'suredy')
 return this.where({title: ['like', ['welefen', 'suredy']]}).select();
 }
 // muti-field or relation like one value
 where4(){
 //SELECT * FROM `think_user` WHERE ((`title` LIKE '%welefen%') OR (`content` LIKE '%welefen%'))
 return this.where({'title|content': ['like', '%welefen%']}).select();
 }
 // muti-filed and relation like one value
 where5(){
 //SELECT * FROM `think_user` WHERE ((`title` LIKE '%welefen%') AND (`content` LIKE '%welefen%'))
 return this.where({'title&content': ['like', '%welefen%']}).select();
 }
}

```

### IN condition

```

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 query

```

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();
 }
}

```

### multi-field query

```

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();
 }
 // modify logic to OR
 where2(){
 //SELECT * FROM `think_user` WHERE (`id` = 10) OR (`title` = 'www')
 return this.where({id: 10, title: "www", _logic: 'OR'}).select();
 }
 // modify logic to XOR
 where2(){
 //SELECT * FROM `think_user` WHERE (`id` = 10) XOR (`title` = 'www')
 return this.where({id: 10, title: "www", _logic: 'XOR'}).select();
 }
}

```

### multi-condition query

```

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();
 }
 // modify logic to OR
 where2(){
 //SELECT * FROM `think_user` WHERE (`id` < 10 OR `id` > 20)
 return this.where({id: {'<': 10, '>': 20, _logic: 'OR'}}).select()
 }
}

```

### complex query

```

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} set query field, can be string or array
- `return` {this}

Set query field.

### String way

```

export default class extends think.controller.base {
 async indexAction(){
 let model = this.model('user');
 // set string need to queyr, in string way, use comma to split
 let data = await model.field('name,title').select();
 }
}

```

### Invoke SQL function

```
export default class extends think.controller.base {
 // invoke sql function in field
 async listAction(){
 let model = this.model('user');
 let data = await model.field('id, INSTR(\`'30,35,31,\`,`id + \`,` as d').select();
 }
}
```

### array way

```
export default class extends think.controller.base {
 async indexAction(){
 let model = this.model('user');
 // set query string in array way
 let data = await model.field(['name', 'title']).select();
 }
}
```

### model.fieldReverse(field)

- **field** {String | Array} reverse field, means query except this field
- **return** {this}

Set reverse field, it will filter this field when querying, it support string way and array way.

### model.table(table, hasPrefix)

- **table** {String} table way
- **hasPrefix** {Boolean} whether table has prefix or not, if table value contains space, then don't add prefix.
- **return** {this}

Set table name, which can named a SQL statement.

### Set current table name

```
export default class extends think.model.base {
 getList(){
 return this.table('test', true).select();
 }
}
```

### SQL statement as table name

```
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} union query SQL or table name
- **all** {Boolean} Whether is UNION ALL way or not
- **return** {this}

Union query.

### SQL union query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();
 }
}
```

### union query table name



```
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} combine statement, default is `LEFT JOIN`
- `return` {this}

Combine query, support string, array, object and so on.

### String

```
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();
 }
}
```

### Array

```
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();
 }
}
```

### Object: single table

```
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 way, contains left, right, inner three ways
 as: 'c', // table alias name
 on: ['cate_id', 'id'] //ON condition
 }).select();
 }
}
```

### Object: multi-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()
 }
}
```

### Object: multi-table

JavaScript

```
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();
 }
}
```

JavaScript

```
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', // has left,right,inner three values
 as: 'c',
 on: ['id', 'id']
 },
 group_tag: {
 join: 'left',
 as: 'd',
 on: ['id', 'group_id']
 }
 }).select()
 }
}
```

**Object: ON condition has multi-field**

JavaScript

```
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` L
 return this.join({
 cate: {on: 'id, id'},
 group_tag: {on: ['id', 'group_id']},
 tag: {
 on: { // multi-field's ON
 id: 'id',
 title: 'name'
 }
 }
 }).select()
 }
}
```

**Object: table value is SQL statement**

JavaScript

```
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 think_user.`gid`=(SELECT * FROM `think_group`).`id`
 return this.join({
 table: sql,
 on: ['gid', 'id']
 }).select();
 }
}
```

**model.order(order)**

- `order` {String | Array | Object} sort order
- `return` {this}

Set sort order.

**String**

JavaScript

```
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();
 }
}
```

## Array

JavaScript

```
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();
 }
}
```

## Object

JavaScript

```
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} table alias name
- `return` {this}

Set tabel alias name.

JavaScript

```
export default class extends think.model.base {
 getList(){
 //SELECT * FROM think_user AS a;
 return this.alias('a').select();
 }
}
```

## model.having(having)

- `having` {String} query string with having
- `return` {this}

Set having query.

JavaScript

```
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} group query field
- `return` {this}

Set group query.

```
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} distinct field
- `return` {this}

Distinct field

```
export default class extends think.model.base {
 getList(){
 //SELECT DISTINCT `name` FROM `think_user`
 return this.distinct('name').select();
 }
}
```

### model.explain(explain)

- `explain` {Boolean} Whether add explain execution or not
- `return` {this}

Whether add explain execution before SQL for performance of SQL or not.

### model.optionsFilter(options)

Options for filter.

### model.dataFilter(data)

- `data` {Object | Array} data to operate

Filter data.

### model.beforeAdd(data)

- `data` {Object} data will add

Add before operate.

### model.afterAdd(data)

- `data` {Object} data will add

Add after data.

### model.afterDelete(data)

Delete after operation.

### model.beforeUpdate(data)

- `data` {Object} data will add

Update before operation.

### model.afterUpdate(data)

- `data` {Object} data will add

Update after operation.

### model.afterFind(data)

- `data` {Object} single data to query
- `return` {Object | Promise}

After `find` query operation.

### model.afterSelect(data)

- `data` [Array] data to query
- `return` {Array | Promise}

After `select` query operation.

### model.data(data)

- `data` {Object}

The data which is added and updated.

### model.options(options)

- `options` {Object}

Configure options, like:

```
export default class extends think.model.base {
 getList(){
 return this.options({
 where: 'id = 1',
 limit: [10, 1]
 }).select();
 }
}
```

JavaScript

### model.close()

About database connection, normally do not invoke directly.

### model.getTableFields(table)

- `table` {String} table name
- `return` {Promise}

Get table field information, read from database directly.

### model.getLastSql()

- `return` {String}

Get the last SQL statement.

### model.buildSql()

- `return` {Promise}

Make current query condition to generate a SQL statement.

### model.parseOptions(oriOpts, extraOptions)

- `oriOpts` {Object}
- `extraOptions` {Object}
- `return` {Promise}

Options which are based on some conditions to parse current operation.

### model.getPK()

- `return` {Promise}

Return value of `pk`, returning is a Promise.

### model.parseType(field, value)

- `field` {String} the field name of data table
- `value` {Mixed}
- `return` {Mixed}

Based on field type of data table to parse value.

### **model.parseData(data)**

- `data` {Object} data to parse
- `return` {Object}

Invoke `parseType` to parse data.

### **model.add(data, options, replace)**

- `data` {Object} data to add
- `options` {Object} operate options
- `replace` {Boolean} whether is replace or not
- `return` {Promise} return inserted ID

add one data.

### **model.thenAdd(data, where)**

- `data` {Object} data to add
- `where` {Object} where condition
- `return` {Promise}

When where condition didn't passed any data then to add data.

### **model.addMany(dataList, options, replace)**

- `dataList` {Array} data list to add
- `options` {Object} operate options
- `replace` {Boolean} is replace or not
- `return` {Promise} return the inserted ID

Add many data in one time.

### **model.delete(options)**

- `options` {Object} operate options
- `return` {Promise} return affected row

Delete data.

### **model.update(data, options)**

- `data` {Object} data to update
- `options` {Object} operate options
- `return` {Promise} return affected rows

Update data.

### **updateMany(dataList, options)**

- `dataList` {Array} data to update
- `options` {Object} operate options
- `return` {Promise}

Update multi-data, dataList must contains value of primary key, it will set to update condition automatically.

### **model.increment(field, step)**

- `field` {String} field name
- `step` {Number} add value, default is 1
- `return` {Promise}

Increase value of field.

### **model.decrement(field, step)**

- `field` {String} field name
- `step` {Number} decrease value, default is 1
- `return` {Promise}

Decrease value of field.

### model.find(options)

- `options` {Object} operate options
- `return` {Promise} return one data

Query one data, type of data is object, if there is not result, return `{}`.

### model.select(options)

- `options` {Object} operate options
- `return` {Promise} return multi-data

Query many data, type of data is array, if there is not result, return `[]`.

### model.countSelect(options, pageFlag)

- `options` {Object} operate options
- `pageFlag` {Boolean} if page number is illegal, true means changed to first page, false means changed to last page, default is no change.
- `return` {Promise}

Page query, normally need to use with `page`, like:

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

JavaScript

returned data structure like this below:

```
{
 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"
 }, ...]
}
```

JavaScript

### model.getField(field, one)

- `field` {String} field name, split with comma
- `one` {Boolean | Number} the number of result
- `return` {Promise}

Get value of specify field.

### model.count(field)

- `field` {String} field name
- `return` {Promise} return the number of fields

Get the number of fields.

### model.sum(field)

- `field` {String} field name
- `return` {Promise}

Get the sum of field value

### model.min(field)

- `field` {String} field name
- `return` {Promise}

Get the minimum of field

### model.max(field)

- `field` {String} field name
- `return` {Promise}

Get the maximum of field

### model.avg(field)

- `field` {String} field name
- `return` {Promise}

Get the avg of field

### model.query(...args)

- `return` {Promise}

Specify SQL statement to query.

### model.execute(...args)

- `return` {Promise}

Execute SQL statement.

### model.parseSql(sql, ...args)

- `sql` {String} to parsed SQL statement
- `return` {String}

Parse SQL statement, invoke `util.format` to parse SQL statement, and parse `__TABLENAME__` of SQL statement to table name.

```
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
 }
}
```

JavaScript

### model.startTrans()

- `return` {Promise}

Start transaction.

### model.commit()

- `return` {Promise}

Commit transaction.

### model.rollback()

- `return` {Promise}

rollback transaction.

### model.transaction(fn)

- `fn` {Function} to executed function
- `return` {Promise}

Use transaction to execute passed function, which must return Promise.



JavaScript

```
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

The `think.model.mongo` class inherit from [think.model.base](#).

### Inheritance with ES6:6

JavaScript

```
export default class extends think.model.mongo {
 getList(){
 }
}
```

### Inheritance With Normal Way

JavaScript

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

## Method

### model.indexes

Set indexes of field, before operate data it will set index automatically.

JavaScript

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

 }
 }
}
```

### Single field index

JavaScript

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

### Unique index

With `$unique` to set unique index, like:

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

### Multi-field index

Multi-field index, like:

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

### model.pk

Primary key name, default is `_id`, get it with `this.getPk`.

## Function

### model.where(where)

Where condition in mongo model is different from relational database.

#### equal condition

```
export default class extends think.model.mongo {
 where1(){
 return this.where({ type: "snacks" }).select();
 }
}
```

#### AND condition

```
export default class extends think.model.mongo {
 where1(){
 return this.where({ type: 'food', price: { $lt: 9.95 } }).select();
 }
}
```

#### OR condition

```
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();
 }
}
```

## Inserted document

```
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();
 }
}
```

JavaScript

## IN condition

```
export default class extends think.model.mongo {
 where1(){
 return this.where({ type: { $in: ['food', 'snacks'] } }).select();
 }
}
```

JavaScript

More details in <https://docs.mongodb.org/manual/reference/operator/query/#query-selectors>.

## model.collection()

- `return` {Promise}

Get handler which operate current table.

```
export default class extends think.model.mongo {
 async getIndexes(){
 let collection = await this.collection();
 return collection.indexes();
 }
}
```

JavaScript

## model.aggregate(options)

Aggregate query, more details in <https://docs.mongodb.org/manual/core/aggregation-introduction/>.

## model.mapReduce(map, reduce, out)

mapReduce operate, more details in <https://docs.mongodb.org/manual/core/map-reduce/>.

## model.createIndex(indexes, options)

- `indexes` {Object} index options
- `options` {Object}

Create indexes.

## model.getIndexes()

- `return` {Promise}

Get indexes.

## middleware

The `think.middleware.base` class inherit from [think.http.base](#).

## Inheritance with ES6:

JavaScript

```
export default class extends think.middleware.base {
 run(){
 }
}
```

## Dynamic Creating Class

JavaScript

```
module.exports = think.middleware({
 run: function(){
 }
})
```

## Methods

### middleare.run()

- `return` {Promise}

middleware exported entrance, invoke this function when calling middleware.