ThinkJS 3.0 Documentation

快速入门

介绍

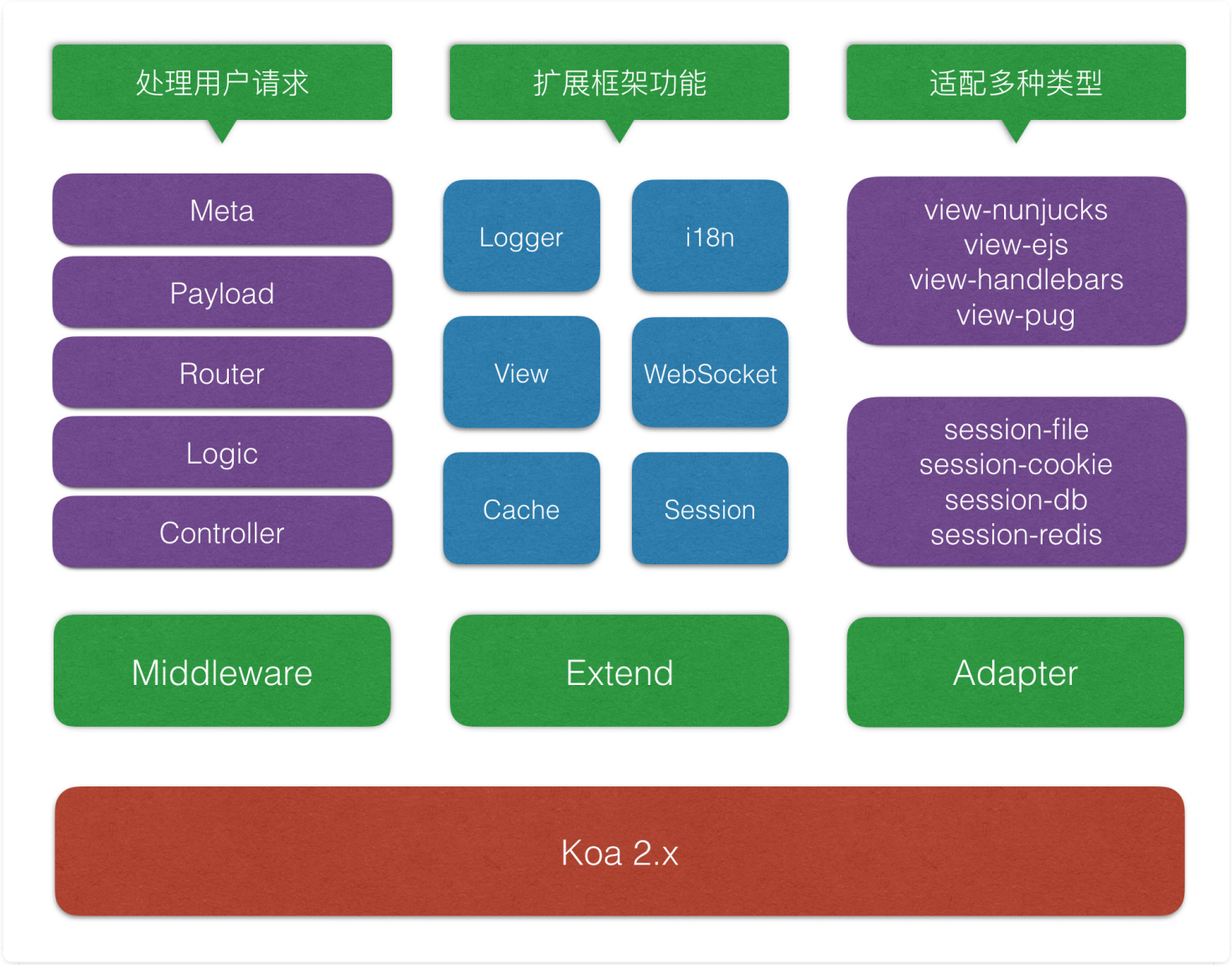
ThinkJS 是一款面向未来的 Node.js 开发框架，整合了很多最佳实践，让企业级开发变得更加简单、高效。从 3.0 开始，ThinkJS 基于 koa 2.x，完全兼容 koa 里的 middleware 等插件。

同时，ThinkJS 支持 Extend 和 Adapter 等方式，方便扩展框架里的各种功能。

特性

- 支持 Middleware、Extend、Adapter 等扩展方式
- 基于 Koa 2.x 开发
- 性能优异，单元测试覆盖程度高
- 内置自动编译、自动更新机制，方便开发环境快速开发
- 直接使用 `async/await` 解决异步问题，不支持 `*/yield`

架构



快速入门

借助 ThinkJS 提供的脚手架，可以快速的创建一个项目。为了可以使用更多的 ES6 特性，框架要求 Node.js 的版本至少是 `6.x`，建议使用 LTS 版本。

安装 ThinkJS 命令

```
$ npm install -g think-cli
```

安装完成后，系统中会有 `thinkjs` 命令（可以通过 thinkjs -v 查看版本号）。如果找不到这个命令，请确认环境变量是否正确。

如果是从 `2.x` 升级，需要将之前的命令删除，然后重新安装。

创建项目

执行 `thinkjs new [project_name]` 来创建项目，如：

```
$ thinkjs new demo;
$ cd demo;
$ npm install;
$ npm start;
```

执行完成后，控制台下会看到类似下面的日志：

```
[2017-06-25 15:21:35.408] [INFO] - Server running at http://127.0.0.1:8360
[2017-06-25 15:21:35.412] [INFO] - ThinkJS version: 3.0.0-beta1
[2017-06-25 15:21:35.413] [INFO] - Enviroment: development
[2017-06-25 15:21:35.413] [INFO] - Workers: 8
```

打开浏览器访问 `http://127.0.0.1:8360/`，如果是在远程机器上创建的项目，需要把 IP 换成对应的地址。

项目结构

默认创建的项目结构如下：

```
|--- development.js   //开发环境下的入口文件
|--- nginx.conf  //nginx 配置文件
|--- package.json
|--- pm2.json //pm2 配置文件
|--- production.js //生成环境入口文件
|--- README.md
|--- src
| |--- bootstrap  //启动字执行目录
| | |--- master.js //Master 进程下自动执行
| | |--- worker.js //Wokre 进程下自动执行
| |--- config  //配置文件目录
| | |--- adapter.js  // adapter 配置
| | |--- config.js  // 默认配置文件
| | |--- config.production.js  //生产环境下的默认配置文件，和 config.js 合并
| | |--- extend.js  //项目扩展配置文件
| | |--- middleware.js //中间件配置文件
| | |--- router.js //自定义路由配置文件
| |--- controller  //控制器目录
| | |--- base.js
| | |--- index.js
| |--- logic //logic 目录
```

```
| | |--- index.js
| |--- model //模型目录
| | |--- index.js
|--- view  //模板目录
| |--- index_index.html
|--- www
| |--- static  //存放静态资源目录
| | |--- css
| | |--- img
| | |--- js
```

升级指南

本文档为 2.x 升级到 3.x 的文档，由于本次升级接口改动较大，所以无法平滑升级。本文档更多的是介绍接口变化指南。

核心变化

3.0 抛弃了已有的核心架构，基于 Koa 2.x 版本构建，兼容 Koa 里的所有功能。主要变化为：

- 之前的 `http` 对象改为 `ctx` 对象
- 执行完全改为调用 `middleware` 来完成
- 框架内置的很多功能不再默认内置，可以通过扩展来支持

项目启动

2.x 中项目启动时，会自动加载 `src/bootstrap/` 目录下的所有文件。3.0 中不再自动加载所有的文件，而是改为：

- 在 Master 进程中加载 `src/boostrap/master.js` 文件
- 在 Woker 进程中加载 `src/boostrap/worker.js` 文件

如果还要加载其他的文件，那么可以在对应的文件中 require 进去。

配置

2.x 中会自动加载 `src/config/` 目录下的所有文件，3.0 中改为根据功能加载对应的文件。

hook 和 middleware

移除 3.x 里的 hook 和 middleware，改为 Koa 里的 middleware，middleware 的管理放在 `src/config/middleware.js` 配置文件中。

2.x 下的 middleware 类无法在 3.0 下使用，3.0 下可以直接使用 Koa 的 middleware。

Controller

将基类 `think.controller.base` 改为 `think.Controller`，并移除 `think.controller.rest` 类。

Model

将基类 `think.model.base` 改为 `think.Model`。

基础功能

启动流程

本文档带领大家一起看看 ThinkJS 是如何启动服务和处理用户请求的。

系统服务启动

- 执行 `npm start` 或者 `node development.js`
- 实例化 ThinkJS 里的 Application 类，执行 `run` 方法。
- 根据不同的环境（Master 进程、Worker 进程、命令行调用）处理不同的逻辑
- 如果是 Master 进程
    - 加载配置文件，生成 `think.config` 和 `think.logger` 对象。
    - 加载文件 `src/bootstrap/matser.js` 文件
    - 如果配置文件监听服务，那么开始监听文件的变化，目录为 `src/`。
    - 文件修改后，如果配置文件编译服务，那么会对文件进行编译，编译到 `app/` 目录下。
    - 根据配置 `workers` 来 fork 对应数目的 Worker。Worker 进程启动完成后，触发 `appReady` 事件。（可以通过 `think.app.on("appReady")` 来捕获）
    - 如果文件发生了新的修改，那么会触发编译，然后杀掉所有的 Worker 进程并重新 fork。
- 如果是 Worker 进程
    - 加载配置文件，生成 `think.config` 和 `think.logger` 对象。
    - 加载 Extend，为框架提供更多的功能，配置文件为 `src/config/extend.js`。
    - 获取当前项目的模块列表，放在 `think.app.modules` 上，如果为单模块，那么值为空数组。
    - 加载项目里的 controller 文件（`src/controller/*.js`），放在 `think.app.controllers` 对

象上。
  - 加载项目里的 logic 文件（`src/logic/*.js`），放在 `think.app.logics` 对象上。
  - 加载项目里的 model 文件（`src/model/*.js`），放在 `think.app.models` 对象上。
  - 加载项目里的 service 文件（`src/service/*.js`），放在 `think.app.services` 对象上。
  - 加载路由配置文件 `src/config/router.js`，放在 `think.app.routers` 对象上。
  - 加载校验配置文件 `src/config/validator.js`，放在 `think.app.validators` 对象上。
  - 加载 middleware 配置文件 `src/config/middleware.js`，并通过 `think.app.use` 方法注册。
  - 加载定时任务配置文件 `src/config/crontab.js`，并注册定时任务服务。
  - 加载 `src/bootstrap/worker.js` 启动文件。
  - 监听 process 里的 `onUncaughtException` 和 `onUnhandledRejection` 错误事件，并进行处理。可以在配置 `src/config.js` 自定义这二个错误的处理函数。
  - 等待 `think.beforeStartServer` 注册的启动前处理函数执行，这里可以注册一些服务启动前的事务处理。
  - 如果自定义了创建服务配置 `createServer`，那么执行这个函数 `createServer(port, host, callback)` 来创建服务。
  - 如果没有自定义，则通过 `think.app.listen` 来启动服务。
  - 服务启动完成时，触发 `appReady` 事件，其他地方可以通过 `think.app.on("appReady")` 监听。
  - 创建的服务赋值给 `think.app.server` 对象。

服务启动后，会打印下面的日志：

```
[2017-07-02 13:36:40.646] [INFO] - Server running at http://127.0.0.1:8360
[2017-07-02 13:36:40.649] [INFO] - ThinkJS version: 3.0.0-beta1
[2017-07-02 13:36:40.649] [INFO] - Enviroment: development
[2017-07-02 13:36:40.649] [INFO] - Workers: 8
```

用户请求处理

当用户请求服务时，会经过下面的步骤进行处理。

- 请求到达 webserver（如：nginx），通过反向代理将请求转发给 node 服务。如果直接通过端口访问 node 服务，那么就没有这一步了。
- node 服务接收用户请求，Master 进程将请求转发给对应的 Worker 进程。
- Worker 进程通过注册的 middleware 来处理用户的请求：
  - meta 来处理一些通用的信息，如：设置请求的超时时间、是否发送 ThinkJS 版本号、是否发送处理的时间等。
  - resource 处理静态资源请求，静态资源都放在 `www/static/` 下，如果命中当前请求是个静态资源，那么这个 middleware 处理完后提前结束，不再执行后面的 middleware。

    - trace 处理一些错误信息，开发环境下打印详细的错误信息，生产环境只是报一个通用的错误。
    - payload 处理用户上传的数据，包含：表单数据、文件等。解析完成后将数据放在 `request.body` 对象上，方便后续读取。
    - router 解析路由，解析出请求处理对应的 Controller 和 Action，放在 `ctx.controller` 和 `ctx.action` 上，方便后续处理。如果项目是多模块结构，那么还有 `ctx.module`。
    - logic 根据解析出来的 controller 和 action，调用 logic 里对应的方法。
        - 实例化 logic 类，并将 `ctx` 传递进去。如果不存在则直接跳过
        - 执行 `__before` 方法，如果返回 `false` 则不再执行后续所有的逻辑（提前结束处理）
        - 如果 `xxxAction` 方法存在则执行，结果返回 `false` 则不再执行后续所有的逻辑
        - 如果 `xxxAction` 方法不存在，则试图执行 `__call` 方法
        - 执行 `__after` 方法，如果返回 `false` 则不再执行后续所有的逻辑
        - 通过方法返回 `false` 来阻断后续逻辑的执行
    - controller 根据解析出来的 controller 和 action，调用 controller 里的对应的方法。
        - 具体的调用策略和 logic 完全一致
        - 如果不存在，那么当前请求返回 404
        - action 执行完成时，可以将结果放在 `this.body` 属性上用户返回给用户。
- 当 Worker 报错，触发 `onUncaughtException` 或者 `onUnhandledRejection` 事件，或者 Worker 异常退出时，Master 会捕获到错误，重新 fork 一个新的 Worker 进程，并杀掉当前的进程。

可以看到，所有的用户请求处理都是通过 middleware 来完成的。具体的项目中，可以根据需求，组装更多的 middleware 来处理用户的请求。

阻止后续行为

上面说到，处理用户请求的所有逻辑都是通过执行 middleware 来完成的。middleware 执行是一个洋葱模型，中可以通过 next 来控制是否执行后续的行为。

```
function middlewareFunction(options){
  return (ctx, next) => {
    if(userLogin){ //这里判断如果用户登录了才执行后续的行为
      return next();
    }
  }
}
```

在 Logic 和 Controller 中，提供了 `__before` 和 `__after` 这些魔术方法，如果想在这些魔术方法里阻止后续的行为执行怎么办呢？

可以通过返回 `false` 来处理，如：

```
__before(){
  if(!userLogin) return false; //这里用户未登录时返回了 false，那么后面的 xxxAction 不再执行
}
```

配置

实际项目中，肯定需要各种配置，包括：框架需要的配置以及项目自定义的配置。ThinkJS 中所有的配置文件都是放在 `src/config/` 目录下，并根据不同的功能划分为不同的配置文件。

- `config.js` 通用的一些配置
- `adapter.js` adapter 配置
- `router.js` 自定义路由配置
- `middleware.js` middlware 配置
- `validator.js` 数据校验配置
- `extend.js` 项目扩展功能配置

配置格式

```
// src/config.js

module.exports = {
  port: 1234,
  redis: {
    host: '192.168.1.2',
    port: 2456,
    password: ''
  }
}
```

配置值即可以是一个简单的字符串，也可以是一个复杂的对象。具体是什么类型根据具体的需求来决定。

多环境配置

有些配置需要在不同的环境下配置不同的值，如：数据库的配置在开发环境和生产环境是不一样的。此时可以通过环境下对应不同的配置文件来完成。

多环境配置文件格式为：`[name].[env].js`，如：`config.development.js`，`config.production.js`

在以上的配置文件中，`config.js` 和 `adapter.js` 是支持不同环境配置文件的。

配置合并方式

系统启动时，会对配置合并，最终提供给开发者使用。具体流程为：

- 加载 `[ThinkJS]/lib/config.js`
- 加载 `[ThinkJS]/lib/config.[env].js`
- 加载 `[ThinkJS]/lib/adapter.js`
- 加载 `[ThinkJS]/lib/adapter.[env].js`
- 加载 `src/config/config.js`
- 加载 `src/config/config.[env].js`
- 加载 `src/config/adapter.js`
- 加载 `src/config/adapter.[env].js`

`[env]` 为当前环境名称。最终会将这些配置按顺序合并在一起，同名的 key 后面会覆盖前面的。

配置加载是通过 think-loader 模块实现的，具体代码见： https://github.com/thinkjs/think-loader/blob/master/loader/config.js。

使用配置

框架提供了在不同的环境下不同的方式快速获取配置：

- 在 ctx 中，可以通过 `ctx.config(key)` 来获取配置
- 在 controller 中，可以通过 `controller.config(key)` 来获取配置
- 其他情况下，可以通过 `think.config(key)` 来获取配置

实际上，`ctx.config` 和 `controller.config` 是基于 `think.config` 包装的一种更方便的获取配置的方式。

```
const redis = ctx.config('redis'); //获取 redis 配置
```

动态设置配置

除了获取配置，有时候需要动态设置配置，如：将有些配置保存在数据库中，项目启动时将配置从数据库中读取出来，然后设置上去。

框架也提供了动态设置配置的方式，如：`think.config(key, value)`

```
// src/bootstrap/worker.js
```

```
//HTTP 服务启动前执行
think.beforeStartServer(async () => {
  const config = await think.model('config').select();
  think.config('userConfig', config); //从数据库中将配置读取出来，然后设置
})
```

常见问题

能否将请求中跟用户相关的值设置到配置中？

`不能`。配置设置是全局的，会在所有请求中生效。如果将请求中跟用户相关的值设置到配置中，那么多个用户同时请求时会相互干扰。

config.js 和 adapter.js 中的 key 能否重名？

`不能`。由于 config.js 和 adapter.js 是合并在一起的，所以要注意这二个配置不能有相同的 key，否则会被覆盖。

Context

Context 是 Koa 中处理用户请求中的一个对象，包含了 `request` 和 `response`，在 middleware 和 controller 中使用，一般简称为 `ctx`。

ThinkJS 里继承了该对象，但扩展了更多的方法以便使用。这些方法是通过 Extend 来实现的，具体代码见 https://github.com/thinkjs/thinkjs/blob/3.0/lib/extend/context.js。

API

userAgent

可以通过 `ctx.userAgent` 属性获取用户的 userAgent。

```
const userAgent = ctx.userAgent;
if(userAgent.indexOf('spider')){
  ...
}
```

isGet

可以通过 `ctx.isGet` 判断当前请求类型是否是 `GET`。

```
const isGet = ctx.isGet;
if(isGet){
  ...
}
```

isPost

可以通过 `ctx.isPost` 判断当前请求类型是否是 `POST`。

```
const isPost = ctx.isPost;
if(isPost){
  ...
}
```

isCli

可以通过 `ctx.isCli` 判断当前请求类型是否是 `CLI`（命令行调用）。

```
const isCli = ctx.isCli;
if(isCli){
  ...
}
```

referer(onlyHost)

- `onlyHost` {Boolean} 是否只返回 host
- `return` {String}

获取请求的 referer。

```
const referer1 = ctx.referer(); // http://www.thinkjs.org/doc.html
const referer2 = ctx.referer(true); // www.thinkjs.org
```

referrer(onlyHost)

等同于 `referer` 方法。

isMethod(method)

- `method` {String} 请求类型
- `return` {Boolean}

判断当前请求类型与 method 是否相同。

```
const isPut = ctx.isMethod('PUT');
```

isAjax(method)

- `method` {String} 请求类型
- `return` {Boolean}

判断是否是 ajax 请求（通过 header 中 `x-requested-with` 值是否为 `XMLHttpRequest` 判断），如果执行了 method，那么也会判断请求类型是否一致。

```
const isAjax = ctx.isAjax();
const isPostAjax = ctx.isAjax('POST');
```

isJsonp(callbackField)

- `callbackField` {String} callback 字段名，默认值为 `this.config('jsonpCallbackField')`
- `return` {Boolean}

判断是否是 jsonp 请求。

```
const isJsonp = ctx.isJson('callback');
if(isJsonp){
  ctx.jsonp(data);
}
```

jsonp(data, callbackField)

- `data` {Mixed} 要输出的数据
- `callbackField` {String} callback 字段名，默认值为 `this.config('jsonpCallbackField')`
- `return` {Boolean} false

输出 jsonp 格式的数据，返回值为 false。可以通过配置 `jsonContentType` 指定返回的 `Content-Type`。

```
ctx.jsonp({name: 'test'});

//output
jsonp111({
  name: 'test'
})
```

json(data)

- `data` {Mixed} 要输出的数据
- `return` {Boolean} false

输出 json 格式的数据，返回值为 false。可以通过配置 `jsonContentType` 指定返回的 `Content-Type`。

```
ctx.json({name: 'test'});

//output
{
  name: 'test'
}
```

success(data, message)

- `data` {Mixed} 要输出的数据
- `message` {String} errmsg 字段的数据
- `return` {Boolean} false

输出带有 `errno` 和 `errmsg` 格式的数据。其中 `errno` 值为 0，`errmsg` 值为 message。

```
{
  errno: 0,
  errmsg: '',
  data: ...
}
```

字段名 `errno` 和 `errmsg` 可以通过配置 `errnoField` 和 `errmsgField` 来修改。

fail(errno, errmsg, data)

- `errno` {Number} 错误号
- `errmsg` {String} 错误信息
- `data` {Mixed} 额外的错误数据
- `return` {Boolean} false

```
{
  errno: 1000,
  errmsg: 'no permission',
  data: ''
}
```

字段名 `errno` 和 `errmsg` 可以通过配置 `errnoField` 和 `errmsgField` 来修改。

expires(time)

- `time` {Number} 缓存的时间，单位是毫秒。可以 `1s`，`1m` 这样的时间
- `return` {undefined}

设置 `Cache-Control` 和 `Expires` 缓存头。

```
ctx.expires('1h'); //缓存一小时
```

config(name, value, m)

- `name` {Mixed} 配置名
- `value` {Mixed} 配置值
- `m` {String} 模块名，多模块项目下生效
- `return` {Mixed}

获取、设置配置项。内部调用 `think.config` 方法。

```
ctx.config('name'); //获取配置
ctx.config('name', value); //设置配置值
ctx.config('name', undefined, 'admin'); //获取 admin 模块下配置值，多模块项目下生效
```

param(name, value)

- `name` {String} 参数名
- `value` {Mixed} 参数值
- `return` {Mixed}

获取、设置 URL 上的参数值。由于 `get`、`query` 等名称已经被 Koa 使用，所以这里只能使用 param。

```
ctx.param('name'); //获取参数值，如果不存在则返回 undefined
ctx.param(); //获取所有的参数值，包含动态添加的参数
ctx.param('name1,name2'); //获取指定的多个参数值，中间用逗号隔开
ctx.param('name', value); //重新设置参数值
ctx.param({name: 'value', name2: 'value2'}); //重新设置多个参数值
```

post(name, value)

- `name` {String} 参数名
- `value` {Mixed} 参数值
- `return` {Mixed}

获取、设置 POST 数据。

```
ctx.post('name'); //获取 POST 值，如果不存在则返回 undefined
ctx.post(); //获取所有的 POST 值，包含动态添加的数据
ctx.post('name1,name2'); //获取指定的多个 POST 值，中间用逗号隔开
ctx.post('name', value); //重新设置 POST 值
ctx.post({name: 'value', name2: 'value2'}); //重新设置多个 POST 值
```

file(name, value)

- `name` {String} 参数名
- `value` {Mixed} 参数值
- `return` {Mixed}

获取、设置文件数据。

```
ctx.file('name'); //获取 FILE 值，如果不存在则返回 undefined
ctx.file(); //获取所有的 FILE 值，包含动态添加的数据
ctx.file('name', value); //重新设置 FILE 值
ctx.file({name: 'value', name2: 'value2'}); //重新设置多个 FILE 值
```

文件的数据格式为：

```
{
  "size": 287313, //文件大小
  "path": "/var/folders/4j/g57qvmmd1lb_9h605w_d38_r0000gn/T/upload_fa6bf8c44179851
f1cfec99544b4ef22", //临时存放的位置
  "name": "An Introduction to libuv.pdf", //文件名
  "type": "application/pdf", //类型
  "mtime": "2017-07-02T07:55:23.763Z" //最后修改时间
}
```

cookie(name, value, options)

- `name` {String} Cookie 名
- `value` {mixed} Cookie 值
- `options` {Object} Cookie 配置项
- `return` {Mixed}

获取、设置 Cookie 值。

```
ctx.cookie('name'); //获取 Cookie
ctx.cookie('name', value); //设置 Cookie
ctx.cookie(name, null); //删除 Cookie
```

设置 Cookie 时，如果 value 的长度大于 4094，则触发 `cookieLimit` 事件，该事件可以通过 `think.app.on("cookieLimit")` 来捕获。

controller(name, m)

- `name` {String} 要调用的 controller 名称
- `m` {String} 模块名，多模块项目下生效
- `return` {Object} Class Instance

获取另一个 Controller 的实例，底层调用 `think.controller` 方法。

```
//获取 src/controller/user.js 的实例
const controller = ctx.controller('user');
```

service(name, m)

- `name` {String} 要调用的 service 名称
- `m` {String} 模块名，多模块项目下生效
- `return` {Mixed}

获取 service，这里获取到 service 后并不会实例化。

```
// 获取 src/service/github.js 模块
const github = ctx.service('github');
```

Middleware

由于 3.0 是在 Koa@2 版本之上构建的，所以完全兼容 Koa@2 里的 middleware。

在 Koa 中，一般是通过 `app.use` 的方式来使用中间件的，如：

```
const app = new Koa();
const bodyparser = require('koa-bodyparser');
app.use(bodyparser({}));
```

为了方便管理和使用 middleware，ThinkJS 提供了一个统一的配置来管理 middleware，配置文件为 `src/config/middleware.js`。

配置格式

```
const path = require('path')
const isDev = think.env === 'development'

module.exports = [
  {
    handle: 'meta',
    options: {
      logRequest: isDev,
      sendResponseTime: isDev,
    },
  },
  {
    handle: 'resource',
    enable: isDev,
    options: {
      root: path.join(think.ROOT_PATH, 'www'),
```

```
      publicPath: /^\/(static|favicon\.ico)/,
    },
  }
]
```

配置项为项目中要使用的 middleware 列表，每一项支持 `handle`，`enable`，`options`，`match` 等属性。

handle

middleware 的处理函数，可以用系统内置的，也可以是引入外部的，也可以是项目里的 middleware。

middleware 的函数格式为：

```
module.exports = (options, app) => {
  return (ctx, next) => {

  }
}
```

这里 middleware 接收的参数除了 options 外，还多了个 `app` 对象，该对象为 Koa Application 的实例。

enable

是否开启当前的 middleware，比如：某个 middleware 只在开发环境下才生效。

```
{
  handle: 'resouce',
  enable: think.env === 'development' //这个 middleware 只在开发环境下生效
}
```

options

传递给 middleware 的配置项，格式为一个对象。

```
module.exports = [
  {
    handle:
  }
```

```
]
```

match

匹配特定的规则后才执行该 middleware，支持二种方式，一种是路径匹配，一种是自定义函数匹配。如：

```
module.exports = [
  {
    handle: 'xxx-middleware',
    match: '/resource' //请求的 URL 是 /resource 打头时才生效这个 middleware
  }
]
```

```
module.exports = [
  {
    handle: 'xxx-middleware',
    match: ctx => { // match 为一个函数，将 ctx 传递给这个函数，如果返回结果为 true，则启用该 middleware
      return true;
    }
  }
]
```

框架内置的 middleware

框架内置了几个 middleware，可以通过字符串的方式直接引用。

```
module.exports = [
  {
    handle: 'meta',
    options: {}
  }
]
```

- meta 显示一些 meta 信息，如：发送 ThinkJS 的版本号，接口的处理时间等等
- resource 处理静态资源，生产环境建议关闭，直接用 webserver 处理即可。
- trace 处理报错，开发环境将详细的报错信息显示处理，也可以自定义显示错误页面。
- payload 处理表单提交和文件上传，类似于 koa-bodyparser 等 middleware
- router 路由解析，包含自定义路由解析

- logic logic 调用，数据校验
- controller controller 和 action 调用

项目中自定义的 middleware

有时候项目中根据一些特定需要添加 middleware，那么可以放在 `src/middleware` 目录下，然后就可以直接通过字符串的方式引用了。

如：添加了 `src/middleware/csrf.js`，那么就可以直接通过 `csrf` 字符串引用这个 middleware。

```
module.exports = [
  {
    handle: 'csrf',
    options: {}
  }
]
```

引入外部的 middleware

引入外部的 middleware 非常简单，只需要 require 进来即可。注意：引入的 middlware 必须是 Koa@2 的。

```
const csrf = require('csrf');
module.exports = [
  ...,
  {
    handle: csrf,
    options: {}
  },
  ...
]
```

常见问题

middleware 配置是否需要考虑顺序?

middleaware 执行是安排配置的顺序执行的，所以需要开发者考虑配置单顺序。

怎么看当前环境下哪些 middleware 生效?

可以通过 `DEBUG=koa:application node development.js` 来启动项目，这样控制台下会看到 `koa:application use ...` 相关的信息。

注意：如果启动了多个 worker，那么会打印多遍。

Logic

当在 Action 里处理用户的请求时，经常要先获取用户提交过来的数据，然后对其校验，如果校验没问题后才能进行后续的操作；当参数校验完成后，有时候还要进行权限判断等，这些都判断无误后才能进行真正的逻辑处理。如果将这些代码都放在一个 Action 里，势必让 Action 的代码非常复杂且冗长。

为了解决这个问题， ThinkJS 在控制器前面增加了一层 `Logic`，Logic 里的 Action 和控制器里的 Action 一一对应，系统在调用控制器里的 Action 之前会自动调用 Logic 里的 Action。

Logic 层

Logic 目录在 `src/[module]/logic`，在项目根目录通过命令 `thinkjs controller test` 会创建名为 test 的 Controller 同时会自动创建对应的 Logic。

Logic 代码类似如下：

```
module.exports = class extends think.Logic {
  __before() {
    // todo
  }
  indexAction() {
    // todo
  }
  __after() {
    // todo
  }
}
```

注：若自己手工创建时，Logic 文件名和 Controller 文件名要相同

其中，Logic 里的 Action 和 Controller 里的 Action 一一对应。Logic 里也支持 `__before` 和 `__after` 魔术方法。

请求类型校验

对应一个特定的 Action，有时候需要限定为某些请求类型，其他类型的请求给拒绝掉。可以通过配置特定的请求类型来完成对请求的过滤。

```
module.exports = class extends think.Logic {
 indexAction() {
    this.allowMethods = 'post'; //  只允许 POST 请求类型
 }
 detailAction() {
    this.allowMethods = 'get,post'; // 允许 GET、POST 请求类型
 }
}
```

校验规则格式

数据校验的配置格式为 `字段名`：`JSON 配置对象`，如下：

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      username: {
        string: true,        // 字段类型为 String 类型
        required: true,      // 字段必填
        default: 'thinkjs', // 字段默认值为 'thinkjs'
        trim: true,          // 字段需要trim处理
        method: 'GET'        // 指定获取数据的方式
      },
      age: {
        int: {min: 20, max: 60} // 20到60之间的整数
      }
    }
    let flag = this.validate(rules);
  }
}
```

基本数据类型

支持的数据类型有：`boolean`、`string`、`int`、`float`、`array`、`object`，对于一个字段只允许指定为一种基本数据类型，默认为 `string` 类型。

手动设置数据值

如果有时候不能自动获取值的话（如：从 header 里取值），那么可以手动获取值后配置进去。如：

```
module.exports = class extends think.Logic {
  saveAction(){
```

```
    let rules = {
      username: {
        value: this.header('x-name') // 从 header 中获取值
      }
    }
  }
}
```

指定获取数据来源

如果校验 `version` 参数， 默认情况下会根据当前请求的类型来获取字段对应的值，如果当前请求类型是 `GET`，那么会通过 `this.param('version')` 来获取 `version` 字段的值；如果请求类型是 `POST`，那么会通过 `this.post('version')` 来获取字段的值， 如果当前请求类型是 `FILE`，那么会通过 `this.file('version')` 来获取 `verison` 字段的值。

有时候在 `POST` 类型下，可能会获取上传的文件或者获取 URL 上的参数，这时候就需要指定获取数据的方式了。支持的获取数据方式为 `GET`， `POST` 和 `FILE`。

字段默认值

使用 `default:value` 来指定字段的默认值，如果当前字段值为空，会把默认值赋值给该字段，然后执行后续的规则校验。

消除前后空格

使用 `trim:true` 如果当前字段支持 `trim` 操作，会对该字段首先执行 `trim` 操作，然后再执行后续的规则校验。

数据校验方法

配置好校验规则后，可以通过 `this.validate` 方法进行校验。如:

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      username: {
        required: true
      }
    }
    let flag = this.validate(rules);
    if(!flag){
      return this.fail('validate error', this.validateErrors);
      // 如果出错，返回
      // {"errno":1000,"errmsg":"validate error","data":{"username":"username can
```

```
not be blank"}}
    }
  }
}
```

如果你在controller的action中使用了`this.isGet`或者`this.isPost`来判断请求的话，在上面的代码中也需要加入对应的`this.isGet`或者`this.isPost`，如：

```
module.exports = class extends think.Logic {
  indexAction(){
    if(this.isPost) {
      let rules = {
        username: {
          required: true
        }
      }
      let flag = this.validate(rules);
      if(!flag){
        return this.fail('validate error', this.errors());
      }
    }

  }
}
```

如果返回值为`false`，那么可以通过访问`this.validateErrors`属性获取详细的错误信息。拿到错误信息后，可以通过`this.fail`方法把错误信息以 JSON 格式输出，也可以通过`this.display`方法输出一个页面，Logic 继承了 Controller 可以调用 Controller 的 方法。

自动调用校验方法

多数情况下都是校验失败后，输出一个 JSON 错误信息。如果不想每次都手动调用`this.validate`进行校验，可以通过将校验规则赋值给`this.rules`属性进行自动校验，如：

```
module.exports = class extends think.Logic {
  indexAction(){
    this.rules = {
      username: {
        required: true
      }
    }
  }
}
```

相当于

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      username: {
        required: true
      }
    }
    let flag = this.validate(rules);
    if(!flag){
      return this.fail(this.config('validateDefaultErrno') , this.validateErrors);
    }
  }
}
```

将校验规则赋值给 `this.rules` 属性后，会在这个 Action 执行完成后自动校验，如果有错误则直接输出 JSON 格式的错误信息。

数组校验

数据校验支持数组校验，但是数组校验只支持一级数组，不支持多层级嵌套的数组。`children` 为所有数组元素指定一个相同的校验规则。

```
module.exports = class extends think.Logic {
  let rules = {
    username: {
      array: true,
      children: {
        string: true,
        trim: true,
        default: 'thinkjs'
      },
      method: 'GET'
    }
  }
  this.validate(rules);
}
```

对象校验

数据校验支持对象校验， 但是对象校验只支持一级对象，不支持多层级嵌套的对象。`children` 为所有对象属性指定一个相同的校验规则。

```
module.exports = class extends think.Logic {
```

```
    let rules = {
      username: {
        object: true,
        children: {
          string: true,
          trim: true,
          default: 'thinkjs'
        },
        method: 'GET'
      }
    }
    this.validate(rules);
  }
```

校验前数据的自动转换

对于指定为 `boolean` 类型的字段，`'yes'`，`'on'`，`'1'`，`'true'`，`true` 会被转换成 `true`，其他情况转换成 `false`，然后再执行后续的规则校验；

对于指定为 `array` 类型的字段，如果字段本身是数组，不做处理；如果该字段为字符串会进行 `split(',')` 处理，其他情况会直接转化为 `[字段值]`，然后再执行后续的规则校验。

校验后数据的自动转换

对于指定为 `int`、`float`、`numeric` 数据类型的字段在校验之后，会自动对数据进行 `parseFloat` 转换。

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      age: {
        int: true,
        method: 'GET'
      }
    }
    let flag = this.validate(rules);
  }
}
```

如果 url 中存在参数 `age=26`，在经过 Logic 层校验之后，typeof this.param('age') 为 `number` 类型。

全局定义校验规则

在单模块下项目下的 `config` 目录下建立 `valadator.js` 文件；在多模块项目下的 `common/config` 目录下建立 `valadator.js`。在 `valadator.js` 中添加自定义的校验方法：

例如, 我们想要验证 `GET` 请求中的 `name1` 参数是否等于字符串 `lucy` 可以如下添加校验规则; 访问你的服务器地址/index/?name1=jack

```
// logic index.js
module.exports = class extends think.Logic {
  indexAction() {
    let rules = {
      name1: {
        eqValid: 'lucy',
        method: 'GET'
      }
    }
    let flag = this.validate(rules);
    if(!flag) {
      console.log(this.validateErrors); // name1 shoud eq lucy
    }
  }
  }
}

// src/config/validator.js
module.exports = {
  rules: {
    /**
     * @param  {Mixed} value         [相应字段的请求值]
     * @param  {Mixed} parsedValue   [校验规则的参数值]
     * @param  {String} validName    [校验规则的参数名称]
     * @return {Boolean}             [校验结果]
     */
    eqValid(value, parsedValue, validName) {
      console.log(value, parsedValue, validName); // 'jack', 'lucy', 'eqValid'
      return value === parsedValue;
    }
  },
  messages: {
    eqValid: '{name} should eq {args}'
  }
}
```

解析校验规则参数

有时我们想对校验规则的参数进行解析，只需要建立一个下划线开头的同名方法在其中执行相应的解析，并将解析后的结果返回即可。

例如我们要验证 `GET` 请求中的 `name1` 参数是否等于 `name2` 参数， 可以如下添加校验方法：访问 你的服务器地址/index/?name1=tom&name2=lily

```
// logic index.js
module.exports = class extends think.Logic {
  indexAction() {
    let rules = {
      name1: {
        eqValid: 'name2',
        method: 'GET'
      }
    }
    let flag = this.validate(rules);
    if(!flag) {
      console.log(validateErrors); // name1 shoud eq name2
    }
  }
}

// src/config/validator.js
module.exports = {
  rules: {
    /**
     * 需要下划线开头的同名方法
     * @param  {Mixed}  validValue [校验规则的参数值]
     * @param  {Mixed}  query      [校验规则method指定的参数来源下的参数]
     * @param  {String} validName  [校验规则的参数名称]
     * @return {Mixed}             [解析之后的校验规则的参数值]
     */
    _eqValid(validValue, query, validName){
      console.log(validValue, query, validName); // 'name2', {name1: 'tom', name2:
 'lily'}, 'eqValid'
      let parsedValue = query[validValue];
      return parsedValue;
    },

    /**
     * @param  {Mixed} value        [相应字段的请求值]
     * @param  {Mixed} parsedValue  [_eqValid方法返回的解析之后的校验规则的参数值]
     * @param  {String} validName   [校验规则的参数名称]
     * @return {Boolean}            [校验结果]
     */
    eqValid(value, parsedValue, validName) {
      console.log(value, parsedValue, validName); // 'tom', 'lily', 'eqValid'
      return value === parsedValue;
    }
  },
  messages: {
    eqValid: '{name} should eq {args}'
  }
}
```

自定义错误信息

错误信息中可以存在三个插值变量 `{name}`、`{args}`、`{pargs}`。`{name}` 会被替换为校验的字段名称，`{args}`会被替换为校验规则的值，`{pargs}` 会被替换为解析方法返回的值。如果 `{args}`、`{pargs}` 不是字符串，将做 `JSON.stringify` 处理。

对于非 `Object: true` 类型的字段，支持三种自定义错误的格式：规则1：规则：错误信息；规则2：字段名：错误信息；规则3：字段名：{ 规则： 错误信息} 。

对于同时指定了多个错误信息的情况，优先级 规则3 > 规则2 > 规则1。

```
module.exports = class extends think.Logic {
  let rules = {
    username: {
      required: true,
      method: 'GET'
    }
  }
  let msgs = {
    required: '{name} can not blank',          // 规则 1
    username: '{name} can not blank',          // 规则 2
    username: {
      required: '{name} can not blank'         // 规则 3
    }
  }
  this.validate(rules, msgs);
}
```

对于 `Object: true` 类型的字段，支持以下方式的自定义错误。优先级为 规则 5 > (4 = 3) > 2 > 1 。

```
module.exports = class extends think.Logic {
  let rules = {
    address: {
      object: true,
      children: {
        int: true
      }
    }
  }
  let msgs = {
    int: 'this is int error message for all field',              // 规则1
    address: {
      int: 'this is int error message for address',              // 规则2
      a: 'this is int error message for a of address',           // 规则3
      'b,c': 'this is int error message for b and c of address' // 规则4
      d: {
```

```
        int: 'this is int error message for d of address'       // 规则5
      }
    }
  }
  let flag = this.validate(rules, msgs);
}
```

支持的校验类型

required

`required: true` 字段必填，默认 `required: false`。`undefined`、`空字符串` 、`null`、`NaN` 在 `required: true` 时校验不通过。

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      name: {
        required: true
      }
    }
    this.validate(rules);
    // todo
  }
}
```

`name` 为必填项。

requiredIf

当另一个项的值为某些值其中一项时，该项必填。如：

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      name: {
        requiredIf: ['username', 'lucy', 'tom'],
        method: 'GET'
      }
    }
    this.validate(rules);
    // todo
  }
}
```

对于上述例子， 当 `GET` 请求中的 `username` 的值为 `lucy`、`tom` 任何一项时， `name` 的值必填。

requiredNotIf

当另一个项的值不在某些值中时，该项必填。如:

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      name: {
        requiredNotIf: ['username', 'lucy', 'tom'],
        method: 'POST'
      }
    }
    this.validate(rules);
    // todo
  }
}
```

对于上述例子，当 `POST` 请求中的 `username` 的值不为 `lucy` 或者 `tom` 任何一项时， `name` 的值必填。

requiredWith

当其他几项有一项值存在时，该项必填。

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      name: {
        requiredWith: ['id', 'email'],
        method: 'GET'
      }
    }
    this.validate(rules);
    // todo
  }
}
```

对于上述例子，当 `GET` 请求中 `id`， `email` 有一项值存在时， `name` 的值必填。

requiredWithAll

当其他几项值都存在时，该项必填。

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      name: {
        requiredWithAll: ['id', 'email'],
        method: 'GET'
      }
    }
    this.validate(rules);
    // todo
  }
}
```

对于上述例子，当 `GET` 请求中 `id`， `email` 所有项值存在时， `name` 的值必填。

requiredWithOut

当其他几项有一项值不存在时，该项必填。

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      name: {
        requiredWithOut: ['id', 'email'],
        method: 'GET'
      }
    }
    this.validate(rules);
    // todo
  }
}
```

对于上述例子，当 `GET` 请求中 `id`， `email` 有任何一项值不存在时， `name` 的值必填。

requiredWithOutAll

当其他几项值都不存在时，该项必填。

```
module.exports = class extends think.Logic {
```

```
  indexAction(){
    let rules = {
      name: {
        requiredWithOutAll: ['id', 'email'],
        method: 'GET'
      }
    }
    this.validate(rules);
    // todo
  }
}
```

对于上述例子，当 `GET` 请求中 `id`， `email` 所有项值不存在时，`name` 的值必填。

contains

值需要包含某个特定的值。

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      name: {
        contains: 'ID-',
        method: 'GET'
      }
    }
    this.validate(rules);
    // todo
  }
}
```

对于上述例子，当 `GET` 请求中 `name` 得值需要包含字符串 `ID-`。

equals

和另一项的值相等。

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      name: {
        equals: 'username',
        method: 'GET'
      }
    }
    this.validate(rules);
```

```
    // todo
  }
}
```

对于上述例子，当 `GET` 请求中的 `name` 与 `username` 的字段要相等。

different

和另一项的值不等。

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      name: {
        equals: 'username',
        method: 'GET'
      }
    }
    this.validate(rules);
    // todo
  }
}
```

对于上述例子，当 `GET` 请求中的 `name` 与 `username` 的字段要不相等。

before

值需要在一个日期之前，默认为需要在当前日期之前。

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      time: {
        before: '2099-12-12 12:00:00', // before: true 早于当前时间
        method: 'GET'
      }
    }
    this.validate(rules);
    // todo
  }
}
```

对于上述例子，当 `GET` 请求中的 `time` 字段对应的时间值要早于 `2099-12-12 12:00:00`。

after

值需要在一个日期之后，默认为需要在当前日期之后，`after: true | time string`。

alpha

值只能是 [a-zA-Z] 组成，`alpha: true`。

alphaDash

值只能是 [a-zA-Z_] 组成，`alphaDash: true`。

alphaNumeric

值只能是 [a-zA-Z0-9] 组成，`alphaNumeric: true`。

alphaNumericDash

值只能是 [a-zA-Z0-9_] 组成，`alphaNumericDash: true`。

ascii

值只能是 ascii 字符组成，`ascii: true`。

base64

值必须是 base64 编码，`base64: true`。

byteLength

字节长度需要在一个区间内，`byteLength: options`。

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      field_name: {
        byteLength: {min: 2, max: 4} // 字节长度需要在 2 - 4 之间
        // byteLength: {min: 2} // 字节最小长度需要为 2
        // byteLength: {max: 4} // 字节最大长度需要为 4
      }
    }
  }
}
```

creditCard

需要为信用卡数字，`creditCard: true`。

currency

需要为货币，`currency: true | options`， `options` 参见 `https://github.com/chriso/validator.js`。

date

需要为日期，`date: true`。

decimal

需要为小数，例如：0.1， .3， 1.1， 1.00003， 4.0，`decimal: true`。

divisibleBy

需要被一个数整除，`divisibleBy: number`。

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      field_name: {
        divisibleBy: 2 //可以被 2 整除
      }
    }
  }
}
```

email

需要为 email 格式，`email: true | options`， `options` 参见 `https://github.com/chriso/validator.js`。

fqdn

需要为合格的域名，`fqdn: true | options`，`options` 参见 `https://github.com/chriso/validator.js`。

float

需要为浮点数，`float: true | options`，`options` 参见 `https://github.com/chriso/validator.js`。

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      money: {
        float: true, //需要是个浮点数
        // float: {min: 1.0, max: 9.55} // 需要是个浮点数，且最小值为 1.0，最大值为 9.55
      }
    }
    this.validate();
    // todo
  }
}
```

fullWidth

需要包含宽字节字符，`fullWidth: true`。

halfWidth

需要包含半字节字符，`halfWidth: true`。

hexColor

需要为个十六进制颜色值，`hexColor: true`。

hex

需要为十六进制，`hex: true`。

ip

需要为 ip 格式，`ip: true`。

ip4

需要为 ip4 格式，`ip4: true`。

ip6

需要为 ip6 格式，`ip6: true`。

isbn

需要为国际标准书号，`isbn: true`。

isin

需要为证券识别编码，`isin: true`。

iso8601

需要为 iso8601 日期格式，`iso8601: true`。

issn

国际标准连续出版物编号，`issn: true`。

uuid

需要为 UUID（3，4，5 版本)，`uuid: true`。

dataURI

需要为 dataURI 格式，`dataURI: true`。

md5

需要为 md5，`md5: true`。

macAddress

需要为 mac 地址，`macAddress: true`。

variableWidth

需要同时包含半字节和全字节字符，`variableWidth: true`。

in

在某些值中，`in: [...]`。

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      version: {
        in: ['2.0', '3.0'] //需要是 2.0, 3.0 其中一个
      }
    }
    this.validate();
    // todo
  }
}
```

notIn

不能在某些值中，`notIn: [...]`。

int

需要为 int 型，`int: true | options`，`options` 参见 `https://github.com/chriso/validator.js`。

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      field_name: {
        int: true, //需要是 int 型
        //int: {min: 10, max: 100} //需要在 10 - 100 之间
      }
    }
    this.validate();
    // todo
  }
}
```

length

长度需要在某个范围，`length: options`。

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      field_name: {
        length: {min: 10}, //长度不能小于10
        // length: {max: 20}, //长度不能大于10
        // length: {min: 10, max: 20} //长度需要在 10 - 20 之间
      }
    }
    this.validate();
    // todo
  }
}
```

lowercase

需要都是小写字母，`lowercase: true`。

uppercase

需要都是大写字母，`uppercase: true`。

mobile

需要为手机号，`mobile: true | options`，`options` 参见 `https://github.com/chriso/validator.js`。

```
module.exports = class extends think.Logic {
  indexAction(){
    let rules = {
      mobile: {
        mobile: 'zh-CN' //必须为中国的手机号
      }
    }
    this.validate();
    // todo
  }
}
```

mongoId

需要为 MongoDB 的 ObjectID，`mongoId: true`。

multibyte

需要包含多字节字符，`multibyte: true`。

url

需要为 url，`url: true|options`，`options` 参见 `https://github.com/chriso/validator.js`。

order

需要为数据库查询 order，如：name DESC，`order: true`。

field

需要为数据库查询的字段，如：name,title，`field: true`。

image

上传的文件需要为图片，`image: true`。

startWith

需要以某些字符打头，`startWith: string`。

endWith

需要以某些字符结束，`endWith: string`。

string

需要为字符串，`string: true`。

array

需要为数组，`array: true`，对于指定为 `array` 类型的字段，如果字段对应的值是字符串不做处理；如果字段对应的值是字符串，进行 `split(,)` 处理；其他情况转化为 `[字段值]`。

boolean

需要为布尔类型。`'yes'`， `'on'`， `'1'`， `'true'`， `true` 会自动转为布尔 `true` 。

object

需要为对象，`object: true`。

regexp

字段值要匹配给出的正则。

```
module.exports = class extends think.Logic {
  indexAction(){
    this.rules = {
      name: {
        regexp: /thinkjs/g
      }
    }
    this.validate();
    // todo
  }
}
```

Controller

控制器是一类操作的集合，用来响应用户同一类的请求。比如：将用户相关的操作都放在 `user.js` 里，每一个操作就是里面一个 Action。

创建 Controller

创建的 controller 都需要继承自 `think.Controller` 类，这样就能使用一些内置的方法。当然项目一般会创建一些通用的基类，然后实际的 controller 都继承自这个基类。

项目创建时会自动创建了一个名为 `base.js` 的基类，其他 controller 继承该类即可。

```
//src/controller/user.js

const Base = require('./base.js');
module.exports = class extends Base {
  indexAction(){
    this.body = 'hello word!';
```

```
  }
}
```

创建完成后，框架会监听变化然后重启服务。这时访问 `http://127.0.0.1:8360/user/index` 就可以看到输出的 `hello word!`

前置操作 __before

项目中，有时候需要在一个统一的地方做一些操作，如：判断是否已经登录，如果没有登录就不能继续后面行为。此种情况下，可以通过内置的 `__before` 来实现。

`__before` 是在调用具体的 Action 之前调用的，这样就可以在其中做一些处理。

```
module.exports = class extends think.Controller {
  async __before(){
    const userInfo = await this.session('userInfo');
    //获取用户的 session 信息，如果为空，返回 false 阻止后续的行为继续执行
    if(think.isEmpty(userInfo)){
      return false;
    }
  }
  indexAction(){
    // __before 调用完成后才会调用 indexAction
  }
}
```

后置操作 __after

后置操作 `__after` 与 `__before` 对应，只是在具体的 Action 执行之后执行，如果具体的 Action 执行返回了 `false`，那么 `__after` 不再执行。

```
module.exports = class extends think.Controller {
  indexAction(){

  }
  __after(){
    //indexAction 执行完成后执行，如果 indexAction 返回了 false 则不再执行
  }
}
```

ctx 对象

controller 实例化时会传入 `ctx` 对象，在 controller 里可以通过 `this.ctx` 来获取 ctx 对象。并且 controller 上很多方法也是通过调用 ctx 里的方法来实现的。

多级控制器

有时候项目比较复杂，文件较多，所以希望根据功能进行一些划分。如：用户端的功能放在一块、管理端的功能放在一块。

这时可以借助多级控制器来完成这个功能，在 `src/controller/` 目录下创建 `user/` 和 `admin/` 目录，然后用户端的功能文件都放在 `user/` 目录下，管理端的功能文件都放在 `admin/` 目录下。

访问时带上对应的目录名，路由解析时会优先匹配目录下的控制器。

API

属性

controller.ctx

传递进来的 ctx 对象。

controller.body

- `return` {String}

设置或者获取返回内容

```
module.exports = class extends think.Controller {
  indexAction(){
    this.body = 'hello world';
  }
}
```

controller.ip

- `return` {String}

获取当前请求用户的 ip，等同于 ctx.ip 方法。

controller.ips

- `return` {Array}

获取当前请求链路的所有 ip，等同于 ctx.ips 方法。

controller.method

- `return` {String}

获取当前请求的类型，转化为小写。

controller.isGet

- `return` {Boolean}

判断是否是 GET 请求。

controller.isPost

- `return` {Boolean}

判断是否是 POST 请求。

controller.isCli

- `return` {Boolean}

是否是命令行下调用。

controller.userAgent

获取 userAgent。

方法

controller.isMethod(method)

- `method` {String} 类型
- `return` {Boolean}

判断当前的请求类型是否是指定的类型。

controller.isAjax(method)

- `method` {String}
- `return` {Boolean}

判断是否是 Ajax 请求。如果指定了 method，那么请求类型也要相同。

```
module.exports = class extends think.Controller {
  indexAction(){
    //是ajax 且请求类型是 POST
    let isAjax = this.isAjax('post');
  }
}
```

controller.isJsonp(callback)

- `callback` {String} callback 名称
- `return` {Boolean}

是否是 jsonp 请求。

controller.get(name)

- `name` {String} 参数名

获取 GET 参数值。

```
module.exports = class extends think.Controller {
  indexAction(){
    //获取一个参数值
    let value = this.get('xxx');
    //获取所有的参数值
    let values = this.get();
  }
}
```

controller.post(name)

- `name` {String} 参数名

获取 POST 提交的参数。

```
module.exports = class extends think.Controller {
  indexAction(){
    //获取一个参数值
```

```
    let value = this.post('xxx');
    //获取所有的 POST 参数值
    let values = this.post();
  }
}
```

controller.param(name)

- `name` {String} 参数名

获取参数值，优先从 POST 里获取，如果取不到再从 GET 里获取。

controller.file(name)

- `name` {String} 上传文件对应的字段名

获取上传的文件，返回值是个对象，包含下面的属性：

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

如果文件不存在，那么值为一个空对象 `{}`。该方法等同于 ctx.file 方法。

controller.header(name, value)

- `name` {String} header 名
- `value` {String} header 值

获取或者设置 header。

```
module.exports = class extends think.Controller {
  indexAction(){
    let accept = this.header('accept'); //获取 header
    this.header('X-NAME', 'thinks'); //设置 header
  }
}
```

controller.expires(time)

- `time` {Number} 过期时间，单位为秒

强缓存，设置 `Cache-Control` 和 `Expires` 头信息。

```
module.exports = class extends think.Controller {
  indexAction(){
    this.expires(86400); //设置过期时间为 1 天。
  }
}
```

该方法等同于 ctx.expires 方法。

controller.referer(onlyHost)

- `referrer` {Boolean} 是否只需要 host

获取 referrer。

controller.referrer(onlyHost)

该方法等同于 controller.referer 方法。

controller.cookie(name, value, options)

- `name` {String} cookie 名
- `value` {String} cookie 值
- `options` {Object}

获取、设置或者删除 cookie。

```
module.exports = class extends think.Controller {
  indexAction(){
    //获取 cookie 值
    let value = this.cookie('think_name');
  }
}
```

```
module.exports = class extends think.Controller {
  indexAction(){
    //设置 cookie 值
```

```
    this.cookie('think_name', value, {
      timeout: 3600 * 24 * 7 //有效期为一周
    });
  }
}
```

```
module.exports = class extends think.Controller {
  indexAction(){
    //删除 cookie
    this.cookie('think_name', null);
  }
}
```

controller.redirect(url)

- `url` {String} 要跳转的 url

页面跳转。

controller.jsonp(data, callback)

- `data` {Mixed} 要输出的内容
- `callback` {String} callback方法名

jsonp 的方法输出内容，获取 callback 名称安全过滤后输出。

```
module.exports = class extends think.Controller {
  indexAction(){
    this.jsonp({name: 'thinkjs'}, 'callback_fn_name');
    //writes
    'callback_fn_name({name: "thinkjs"})'
  }
}
```

controller.json(data)

- `data` {Mixed} 要输出的内容

json 的方式输出内容。

controller.status(status)

- `status` {Number} 状态码，默认为 404

设置状态码。

controller.success(data, message)

格式化输出一个正常的数据，一般是操作成功后输出，等同于 ctx.success。

controller.fail(errno, errmsg, data)

格式化输出一个异常的数据，一般是操作失败后输出，等同于 ctx.fail。

View

由于某些项目下并不需要 View 的功能，所以 3.0 里并没有直接内置 View 的功能，而是通过 Extend 和 Adapter 来实现的。

Extend 来支持 View

配置 `src/config/extend.js`，添加如下的配置，如果已经存在则不需要再添加：

```
const view = require('think-view');
module.exports = [
  view
]
```

通过添加 view 的扩展，让框架有渲染模板文件的能力。

配置 View Adapter

在 `src/config/adapter.js` 中添加如下的配置，如果已经存在则不需要再添加：

```
const nunjucks = require('think-view-nunjucks');
const path = require('path');

exports.view = {
  type: 'nunjucks',
  common: {
    viewPath: path.join(think.ROOT_PATH, 'view'), //模板文件的根目录
    sep: '_', //Controller 与 Action 之间的连接符
    extname: '.html' //文件扩展名
  },
```

```
  nunjucks: {
    handle: nunjucks
  }
}
```

这里用的模板引擎是 `nunjucks`，项目中可以根据需要修改。

具体使用

配置了 Extend 和 Adapter 后，就可以在 Controller 里使用了。如：

```
module.exports = class extends think.Controller {
  indexAction(){
    this.assign('title', 'thinkjs'); //给模板赋值
    return this.display(); //渲染模板
  }
}
```

assign

给模板赋值。

```
this.assign('title', 'thinkjs'); //单条赋值
this.assign({title: 'thinkjs', name: 'test'}); //单条赋值
this.assign('title'); //获取之前赋过的值，如果不存在则为 undefined
this.assign(); //获取所有赋的值
```

render

获取渲染后的内容。

```
const content1 = await this.render(); //根据当前请求解析的 controller 和 action 自动匹配模板文件
const content2 = await this.render('doc'); //指定文件名

const content3 = await this.render('doc', 'ejs'); //切换模板类型
const content4 = await this.render('doc', {type: 'ejs', xxx: 'yyy'});//切换模板类型，并配置额外的参数
```

display

渲染并输出内容。

```
return this.display(); //根据当前请求解析的 controller 和 action 自动匹配模板文件

return this.display('doc'); //指定文件名

return this.display('doc', 'ejs'); //切换模板类型
return this.display('doc', {type: 'ejs', xxx: 'yyy'});//切换模板类型，并配置额外的参数
```

支持的 Adapter

View 支持的 Adapter 见 https://github.com/thinkjs/think-awesome#view。

路由

当用户访问一个 URL 时，最终执行哪个模块（module）下哪个控制器（controller）的哪个操作（action），这是路由模块解析后决定的。除了默认的解析外，ThinkJS 提供了一套灵活的路由机制，让 URL 更加简单友好。路由在 ThinkJS 中是以中间件（middleware）的形式存在的。

路由参数配置

在路由中间件配置文件中可以针对路由进行基础的参数配置。单模块下，配置文件 `src/config/middleware.js`：

```
const router = require('think-router');
module.exports = [
  {
    handle: router,
    options: {
      prefix: 'home',                // 多模块下，默认的模块名
      defaultController: 'index',    // 默认的控制器名
      defaultAction: 'index',        // 默认的操作名
      prefix: [],                    // 默认去除的 url 前缀
      suffix: ['.html'],             // 默认去除的 url 后缀
      enableDefaultRouter: true,     // 在不匹配情况下是否使用默认路由
      subdomainOffset: 2,            // 子域名偏移
      subdomain: {},                 // 子域名映射
      denyModules: []                // 多模块下，禁止访问的模块
    }
  }
];
```

pathname 预处理

当用户访问服务时，服务端首先拿到的是一个完整的 URL，如：访问本页面，得到的 URL 为 `http://www.thinkjs.org/zh-cn/doc/3.0/router.html`，我们得到的初始 `pathname` 为 `/zh-cn/doc/3.0/router.html`。

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

ThinkJS 里提供了下面的配置可以去除 `pathname` 的某些前缀和后缀。在路由中间件配置文件中：

```
{
  prefix: [],
  suffix: ['.html'],
  // 其他配置...
}
```

`prefix` 与 `subffix` 为数组，数组的每一项可以为字符串或者正则表达式， 在匹配到第一个之后停止后续匹配。对于上述 `pathname` 在默认配置下进行过滤后，拿到纯净的 pathname 为 `/zh-cn/doc/3.0/router`。

如果访问的 URL 是 `http://www.thinkjs.org/`，那么最后拿到纯净的 `pathname` 则为字符串 `/`。

路由规则

单模块路由规则配置文件 `src/config/router.js`，路由规则为二维数组：

```
module.exports = [
  [/libs\/(.*)/i, '/libs/:1', 'get'],
  [/fonts\/(.*)/i, '/fonts/:1', 'get,post'],
];
```

路由的匹配规则为：从前向后逐一匹配，如果命中到了该项规则，则不再向后匹配。 对于每一条匹配规则，参数为：

```
[
  match,      // url 匹配规则，预先使用 path-to-regexp 转换
  path,       // 对应的操作（action）的路径
```

```
  method,     // 允许匹配的请求类型，多个请求类型之间逗号分隔，get|post|redirect|rest|cli
  [options]   // 额外参数，如 method=redirect时，指定跳转码 {statusCode: 301}
]
```

路由解析

对于匹配规则中的 `match` 会使用 path-to-regexp 预先转换：

```
module.exports = [
  ['/api_libs/(.*)/:id', '/libs/:1/', 'get'],
]
```

对于 `match` 中的 `:c`(c 为字符串)，在 `match` 匹配 `pathname` 之后获取到 `c` 的值， `c` 的值会被附加到 `this.ctx`； `path` 可以引用 `match` 匹配 `pathname` 之后的结果，对于 `path` 中的 `:n`(n 为数字)， 会被 `match` 匹配 `pathname` 的第 n 个引用结果替换。

路由识别默认根据 `[模块]/控制器/操作/...` 来识别过滤后的 `pathname`，从而对应到最终要执行的操作(action)。

例如，对于上述规则，在访问 URL `http:/www.thinkjs.org/api_libs/inbox/123` 时，规则的 `path` 将变为 `/libs/inbox/`,同时`{id: '123'}` 会附加到 `this.ctx` 上。之后对 `pathname` 进行解析，就对应到了 `libs` 控制器（controller）下的 `inbox` 操作（action）。在 `inboxAction` 下可以通过 `this.ctx.param('id')` 获取到 `id` 的值 `123`。

子域名部署

当项目比较复杂时，可能希望将不同的功能部署在不同的域名下，但代码还是在一个项目下。ThinkJS 提供子域名来处理这个需求，在路由中间件配置文件中：

```
{
  subdomainOffset: 2,
  prefix: [],
  suffix: ['.html'],
  subdomain: {
    'news,zm': 'news'
  },
  // 其他配置...
}
```

域名偏移(subdomainOffset) 默认为 2， 例如对于域名 zm.news.so.com， `this.ctx.subdomains` 为 `['news', 'zm']`,当域名偏移为 3 时， `this.ctx.subdomains` 为 `['zm']`。

如果路由中间件配置的 `subdomain` 项中存在 `this.ctx.subdomains.join(,)` 对应的 key，此 key 对应的值将会被附加到 `pathname` 上，然后再进行路由的解析。

对于上述配置，当我们访问 `http://zm.news.so.com:8360/api_lib/inbox/123`，我们得到的 `pathname` 将为 `/news/api_lib/inbox/123`。

另外 `subdomain` 的配置也可以为一个数组，我们会将数组转化为对象。例如 `subdomain: ['admin', 'user']` 将会被转化为 `subdomain: {admin: 'admin', user: 'user'}`。

Adapter

Adapter 是用来解决一类功能的多种实现，这些实现提供一套相同的接口。如：支持多种数据库，支持多种模版引擎等。通过这种方式，可以很方便的再不同的类型中进行切换。

框架提供了多种 Adapter，如： View，Model，Cache，Session，Websocket 等。

Adapter 配置

Adapter 的配置文件为 `src/config/adapter.js`，格式如下：

```
const nunjucks = require('think-view-nunjucks');
const ejs = require('think-view-ejs');
const path = require('path');

exports.view = {
  type: 'nunjucks', // 默认的模板引擎为 nunjucks
  common: { //通用配置
    viewPath: path.join(think.ROOT_PATH, 'view'),
    sep: '_',
    extname: '.html'
  },
  nunjucks: { // nunjucks 的具体配置
    handle: nunjucks
  },
  ejs: { // ejs 的具体配置
    handle: ejs,
    viewPath: path.join(think.ROOT_PATH, 'view/ejs/'),
  }
}

exports.cache = {
  ...
}
```

- `type` 默认使用 Adapter 的类型，具体调用时可以传递参数改写

- `common` 配置通用的一些参数，会跟具体的 adapter 参数作合并
- `nunjucks,ejs` 配置特定类型的 Adapter 参数，最终获取到的参数是 common 参数与该参数进行合并
- `handle` 对应类型的处理函数，一般为一个类

项目中创建 Adapter

除了引入外部的 Adapter 外，项目内也可以创建 Adapter 来使用。

Adapter 文件放在 `src/adapter/` 目录下，如：`src/adapter/cache/xcache.js`，表示加了一个名为 `xcache` 的 cache Adapter 类型，然后该文件实现 cache 类型一样的接口即可。

实现完成后，就可以直接通过字符串引用这个 Adapter 了，如：

```
exports.cache = {
  type: 'file',
  xcache: {
    handle: 'xcache', //这里配置字符串，项目启动时会自动查找 src/adapter/cache/xcache.js
 文件
    ...
  }
}
```

支持的 Adapter

框架支持的 Adapter 为 https://github.com/thinkjs/think-awesome#adapters。

Extend

虽然框架内置了很多功能，但在实际项目开发中，提供的功能还是远远不够的。3.0 里引入了 Extend 机制，方便对框架进行扩展。支持的扩展类型为：`think`，`application`，`context`，`request`，`response`，`controller` 和 `logic`。

框架内置的很多功能也是 Extend 来实现的，如：`Session`，`Cache`。

Extend 配置

Extend 配置文件为 `src/config/extend.js`，格式为数组：

```
const view = require('think-view');
const fetch = require('think-fetch');
```

```
const model = require('think-model');

module.exports = [
  view, //make application support view
  fetch, // HTTP request client
  model(think.app)
];
```

如上，通过 view Extend 框架就支持渲染模板的功能，Controller 类上就有了 `assign`、`display` 等方法。

项目里的 Extend

除了引入外部的 Extend 来丰富框架的功能，也可以在项目中对对象进行扩展。扩展文件放在 `src/extend/` 目录下。

- `src/extend/think.js` 扩展 think 对象，如： think.xxx
- `src/extend/application.js` 扩展 think.app 对象，Koa 里的 application 实例
- `src/extend/request.js` 扩展 Koa 里的 request 对象
- `src/extend/response.js` 扩展 Koa 里的 response 对象
- `src/extend/context.js` 扩展 ctx 对象
- `src/extend/controller.js` 扩展 controller 类
- `src/extend/logic.js` 扩展 logic 类，logic 继承 controller 类，所以 logic 包含 controller 类所有方法

比如：我们想给 `ctx` 添加个 `isMobile` 方法来判断当前请求是不是手机访问，可以通过下面的方式：

```
//src/extend/context.js
module.exports = {
  isMobile: function(){
    const userAgent = this.userAgent.toLowerCase();
    const mList = ['iphone', 'android'];
    return mList.some(item => userAgent.indexOf(item) > -1);
  }
}
```

这样后续就可以通过 `ctx.isMobile()` 来判断是否是手机访问了。当然这个方法没有任何的参数，我们也可以变成一个 `getter`。

```
//src/extend/context.js
module.exports = {
  get isMobile: function(){
```

```
    const userAgent = this.userAgent.toLowerCase();
    const mList = ['iphone', 'android'];
    return mList.some(item => userAgent.indexOf(item) > -1);
  }
}
```

这样在 ctx 中就可以直接用 `this.isMobile` 来使用，其他地方通过 `ctx.isMobile` 使用，如： 在 controller 中用 `this.ctx.isMobile`。

如果在 controller 中也想通过 `this.isMobile` 使用，怎么办呢？ 可以给 controller 也扩展一个 `isMobile` 属性来完成。

```
//src/extend/controller.js
module.exports = {
  get isMobile: function(){
    return this.ctx.isMobile;
  }
}
```

通过也给 controller 扩展 `isMobile` 属性后，后续在 controller 里可以直接使用 `this.isMobile` 了。

当然这样扩展后，只能在当前项目里使用这些功能，如果要在其他项目中使用，可以将这些扩展发布为一个 npm 模块。

发布的模块中在入口文件里需要定义对应的类型的扩展，如：

```
const controllerExtend = require('./controller.js');
const contextExtend = require('./context.js');

// 模块入口文件
module.exports = {
  controller: controllerExtend,
  context: contextExtend
}
```

Extend 里使用 app 对象

有些 Extend 需要使用一些 app 对象上的数据，那么可以导出为一个函数，配置时把 app 对象传递进去即可。

```
// src/config/extend.js
const model = require('think-model');
```

```
module.exports = [
  model(think.app) //将 think.app 传递给 model 扩展
];
```

当然传了 app 对象，也可以根据需要传递其他对象。

推荐 Extend 列表

推荐的 Extend 列表见 https://github.com/thinkjs/think-awesome#extends，如果你开发了比较好的 Extend，也欢迎发 Pull Request。

进阶应用

think 对象

框架中内置 `think` 全局对象，方便在项目中随时随地使用。

think.app

`think.app` 为 Koa Application 对象的实例，系统启动时生成。

此外为 app 扩展了更多的属性。

- `think.app.think` 等同于 think 对象，方便有些地方传入了 app 对象，但又要使用 think 对象上的其他方法
- `think.app.modules` 模块列表，单模块项目下为空数组
- `think.app.controllers` 存放项目下的 controller 文件，便于后续快速调用
- `think.app.logics` 存放项目下的 logic 文件
- `think.app.models` 存放项目下的模型文件
- `think.app.services` 存放 service 文件
- `think.app.routers` 存放自定义路由配置
- `think.app.validators` 存放校验配置

如果想要查下这些属性具体的值，可以在 `appReady` 事件中进行。

```
think.app.on('appReady', () => {
  console.log(think.app.controllers)
})
```

API

think 对象上集成了 think-helper 上的所有方法，所以可以通过 `think.xxx` 来使用这些方法。

think.env

当前运行环境，等同于 `think.app.env`，值在 `development.js` 之类的入口文件中定义。

think.isCli

是否为命令行调用。

think.version

当前 ThinkJS 的版本号。

think.Controller

控制器基类。

think.Logic

logic 基类。

think.controller(name, ctx, m)

- `name` {String} 控制器名称
- `ctx` {Object} Koa ctx 对象
- `m` {String} 模块名，多模块项目下使用

获取控制器的实例，不存在则报错。

think.beforeStartServer(fn)

- `fn` {Function} 要注册的函数名

服务启动之前要注册执行的函数，如果有异步操作，fn 需要返回 Promise。

启动自定义

当通过 `npm start` 或者 `node production.js` 来启动项目时，虽然可以在这些入口文件里添加其他的逻辑代码，但并不推荐这么做。系统给出了其他启动自定义的入口。

bootstrap

系统启动时会加载 `src/boostrap/` 目录下的文件，具体为：

- Master 进程下时加载 `src/bootstrap/master.js`
- Worker 进程下时加载 `src/bootstrap/worker.js`

所以可以将一些需要在系统启动时就需要执行的逻辑放在对应的文件里执行。

如果有一些代码需要在 Master 和 Worker 进程下都调用，那么可以放在一个单独的文件里，然后 master.js 和 worker.js 去 required。

```
// src/bootstrap/common.js
global.commonFn = function(){

}

// src/bootstrap/master.js
require('./common.js')


// src/boostrap/worker.js
require('./common.js')
```

启动服务前执行

有时候需要在 node 启动 http 服务之前做一些特殊的逻辑处理，如：从数据库中读取配置并设置，从远程还在一些数据设置到缓存中。

这时候可以借助 `think.beforeStartServer` 方法来处理，如：

```
think.beforeStartServer(async () => {
  const data = await getDataFromApi();
  think.config(key, data);
})
```

可以通过 `think.beforeStartServer` 注册多个需要执行的逻辑，如果逻辑函数中有异步的操作，需要返回 Promise。

系统会等待注册的多个逻辑执行完成后才启动服务，当然也不会无限制的等待，会有个超时时间。超时时间可以通过配置 `startServerTimeout` 修改，默认为 3 秒。

```
//src/config/config.js
module.exports = {
  startServerTimeout: 5000 // 将超时时间改为 5s
}
```

自定义创建 http 服务

系统默认是通过 Koa 里的 listen 方法来创建 http 服务的，如果想要创建 https 服务，此时需要自定义创建服务，可以通过 `createServer` 配置来完成。

```
//src/config/config.js
const https = require('https');
const fs = require('fs');

const options = {
  key: fs.readFileSync('test/fixtures/keys/agent2-key.pem'),
  cert: fs.readFileSync('test/fixtures/keys/agent2-cert.pem')
};

module.exports = {
  // app 为 Koa app 对象
  createServer: function(app, ...args){
    https.createServer(options, app.callback()).listen(...args);
  }
}
```

think.app.server 对象

创建完 http 服务后，会将 server 对象赋值给 `think.app.server`，以便于在其他地方使用。

appReady 事件

http 服务创建完成后，会触发 `appReady` 事件，其他地方可以通过 `think.app.on("appReady")` 来捕获该事件。

```
think.app.on("appReady", () => {
  const server = think.app.server;
})
```

Service

Cookie

配置

没有默认配置。需要在`src/config/config.js`中添加 cookie 配置，比如：

```
module.exports = {
  cookie: {
    domain: '',
    path: '/',
    httponly: false, // 是否 http only
    secure: false,
    timeout: 0  // 有效时间，0 表示载浏览器进程内有效，单位为秒。
  }
}
```

获取 cookie

在 controller 或者 logic 中，可以通过`this.cookie`方法来获取 cookie。如：

```
module.exports = class extends think.Controller {
  indexAction(){
    let cookie = this.cookie('theme'); // 获取名为 theme 的 cookie
  }
}
```

`this.ctx`也提供了`cookie`方法来设置 cookie。如：

```
this.ctx.cookie('theme');
```

设置 cookie

在 controller 或者 logic 中，可以通过 `this.cookie` 方法来设置 cookie。如：

```
module.exports = class extends think.Controller {
  indexAction(){
    let cookie = this.cookie('theme', 'default'); // 将 cookie theme 值设置为 defau
lt
  }
}
```

`this.ctx` 也提供了 `cookie` 方法来设置 cookie。如：

```
this.ctx.cookie('theme', 'default');
```

如果希望在设置 cookie 时改变配置参数，可以通过第三个参数来控制。比如：

```
module.exports = class extends think.Controller {
  indexAction(){
    let cookie = this.cookie('theme', 'default', {
      timeout: 7 * 24 * 3600 // 设置 cookie 有效期为 7 天
    }); // 将 cookie theme 值设置为 default
  }
}
```

删除 cookie

在 controller 或者 logic 中，可以通过 `this.cookie` 方法来删除。比如：

```
module.exports = class extends think.Controller {
  indexAction(){
    let cookie = this.cookie('theme', null); //  删除名为 theme 的 cookie
  }
}
```

`this.ctx` 也提供了 `cookie` 方法来删除 cookie。如：

```
this.ctx.cookie('theme', null);
```

Session

Thinkjs 内置了 Session 功能。框架已经为 controller 和 context 添加了`session`方法。我们支持用多种方式存储 session，包括：cookie，mysql，file，redis 等。

支持的Session存储方式

- `cookie` Cookie方式
- `mysql` Mysql数据库方式
- `file` 文件方式
- `redis` Redis方式

Mysql session

使用`mysql`类型的 Session 需要创建对应的数据表，可以用下面的 SQL 语句创建

```
DROP TABLE IF EXISTS `think_session`;
CREATE TABLE `think_session` (
  `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
  `cookie` varchar(255) NOT NULL DEFAULT '',
  `data` text,
  `expire` bigint(11) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `cookie` (`cookie`),
  KEY `expire` (`expire`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

redis Session

使用`redis`类型的Session需要依赖`think-redis`模块。

如何配置Session

配置文件`src/config/adapter.js`，添加如下选项（假设你默认使用 Cookie 方式的 Session）：

```
const cookieSession = require('think-session-cookie');
```

```
const fileSession = require('think-session-file');

exports.session = {
  type: 'cookie',
  common: {
    cookie: {
      name: 'test',
      keys: ['werwer', 'werwer'],
      signed: true
    }
  },
  cookie: {
    handle: cookieSession,
    cookie: {
      maxAge: 1009990 * 1000,
      keys: ['welefen', 'suredy'],
      encrypt: true
    }
  },
  file: {
    handle: fileSession,
    sessionPath: path.join(think.ROOT_PATH, 'runtime/session')
  }
}
```

接下来解释一下这个配置文件的各种参数：
* `type`：默认使用的 Session 类型，具体调用时可以传递参数改写（见“使用`session`方法”一节）
* `common`：配置通用的一些参数，会跟具体的 Adapter 参数合并
* `cookie,file,mysql`：配置特定类型的 Adapter 参数，最终获取到的参数是 common 参数与该参数进行合并后的结果。我们注意到，在这几个配置里面都有一个`handle`参数。
* `handle`：对应类型的处理函数，一般为一个类。

具体看一下 Cookie 方式的参数。`handle`上面已经介绍过，略过不表。
* `maxAge`：该 session 要在 cookie 中保留多长时间
* `keys`：当`encrypt`参数为`true`时，需要提供`keys`数组。该数组充当了加解密的密钥。
* `encrypt`：为`true`表示需要加密存储 session。

接着看一下file方式的参数：
* `sessionPath`：存储 session 的文件的路径。在本例中，如果使用文件方式，会将 session 存储*'/path_of_your_project/runtime/session'*下。

使用session方法：

读取Session

- `this.session()` 获取所有的 session 数据
- `this.session(name)` 获取 `name` 对应的 session 数据

设置Session

- `this.session(name, value)` 设置 session 数据。

清除Session

- `this.session(null)` 删除所有的 session 数据。

在读取／设置／清除时，改写默认配置

例如：

- `this.session(name, undefined, options)` 以 `options` 配置项来获取 session 数据。

`options` 配置项会与 adapter 中的默认配置合并。如果有相同的配置属性，对每个 ctx，首次调用 `this.session` 时，将以 `options` 的配置属性为准。

注意：对于每个 ctx，session 只会初始化一次。

Logger

ThinkJS 通过 think-logger3 模块实现了强大的日志功能，并提供适配器扩展，可以很方便的扩展内置的日志模块。系统默认使用 log4js 模块作为底层日志记录模块，具有日志分级、日志分割、多进程等丰富的特性。

基本使用

系统已经全局注入了 logger 对象 `think.logger`，其提供了 `debug`, `info`, `warn`, `error` 四种方法来输出日志，日志默认是输出至控制台中。

```
think.logger.debug('debug message');
think.logger.info('info message');
think.logger.warn('warning');
think.logger.error(new Error('error'));
```

基本配置

系统默认自带了 `Console`, `File`, `DateFile` 三种适配器。默认是使用 `Console` 将日志输出到控制台中。

文件

如果想要将日志输出到文件，你可以这么设置：

- 将以下内容追加到 `src/config/adapter/logger.js` 文件中：
  ```javascript
  const path = require('path');
  const {File} = require('think-logger3');

  module.exports = {
  type: 'file',
  file: {
  handle: File,
  backups: 10,
  absolute: true,
  maxLogSize: 50 * 1024, //50M
  filename: path.join(think.ROOT_PATH, 'logs/xx.log')
  }
  }
  ```

- 编辑 `src/config/adater.js` 文件，增加 `exports.logger = require('./adapter/logger.js')`。

该配置表示系统会将日志写入到 `logs/xx.log` 文件中。当该文件超过 `maxLogSize` 值时，会创建一个新的文件 `logs/xx.log.1`。当日志文件数超过 `backups` 值时，旧的日志分块文件会被删除。文件类型目前支持如下参数：

- `filename`：日志文件地址
- `maxLogSize`：单日志文件最大大小，单位为 KB，默认日志没有大小限制。
- `backups`：最大分块地址文件数，默认为 5。
- `absolute`：`filename` 是否为绝对路径地址，如果 `filename` 是绝对路径，`absolute` 的值需要设置为 `true`。
- `layouts`：定义日志输出的格式。

日期文件

如果想要将日志按照日期文件划分的话，可以如下配置：

- 将以下内容追加到 `src/config/adapter/logger.js` 文件中：
  ```javascript
  const path = require('path');
  const {DateFile} = require('think-logger3');

  module.exports = {
  type: 'dateFile',
  dateFile: {
  handle: DateFile,
  level: 'ALL',
  absolute: true,
  pattern: '-yyyy-MM-dd',
  alwaysIncludePattern: false,
  filename: path.join(think.ROOT_PATH, 'logs/xx.log')
  }
  }
  ```

- 编辑 `src/config/adater.js` 文件，增加 `exports.logger = require('./adapter/logger.js')`。

该配置会将日志写入到 `logs/xx.log` 文件中。隔天，该文件会被重命名为 `xx.log-2017-07-01`（时间以当前时间为准），然后会创建新的 `logs/xx.log` 文件。时间文件类型支持如下参数：

- `level`：日志等级
- `filename`：日志文件地址
- `absolute`：`filename` 是否为绝对路径地址，如果 `filename` 是绝对路径，`absolute` 的值需要设置为 `true`。
- `pattern`：该参数定义时间格式字串，新的时间文件会按照该格式格式化后追加到原有的文件名后面。目前支持如下格式化参数：
  - `yyyy` - 四位数年份，也可以使用 `yy` 获取末位两位数年份
  - `MM` - 数字表示的月份，有前导零
  - `dd` - 月份中的第几天，有前导零的2位数字
  - `hh` - 小时，24小时格式，有前导零
  - `mm` - 分钟数，有前导零
  - `ss` - 秒数，有前导零
  - `SSS` - 毫秒数（不建议配置该格式以毫秒级来归类日志）
  - `O` - 当前时区
- `alwaysIncludePattern`：如果 `alwaysIncludePattern` 设置为 `true`，则初始文件直接会被命名为 `xx.log-2017-07-01`，然后隔天会生成 `xx.log-2017-07-02` 的新文件。

- `layouts`：定义日志输出的格式。

Level

日志等级用来表示该日志的级别，目前支持如下级别：

- ALL
- ERROR
- WARN
- INFO
- DEBUG

Layout

`Console`，`File` 和 `DateFile` 类型都支持 `layout` 参数，表示输出日志的格式，值为对象，下面是一个简单的示例：

```
const path = require('path');
const {File} = require('think-logger3');

module.exports = {
  type: 'file',
  file: {
    handle: File,
    backups: 10,
    absolute: true,
    maxLogSize: 50 * 1024,  //50M
    filename: path.join(think.ROOT_PATH, 'logs/xx.log'),
    layouts: {
      type: 'coloured',
      pattern: '%[[%d] [%p]%] - %m',
    }
  }
}
```

layouts 支持如下参数：

- `type`：目前支持如下类型
  - basic
  - coloured
  - messagePassThrough
  - dummy
  - pattern

    - 自定义输出类型可参考 Adding your own layouts
- `pattern`：输出格式字串，目前支持如下格式化参数
    - `%r` - `.toLocaleTimeString()` 输出的时间格式，例如 `下午5:13:04`。
    - `%p` - 日志等级
    - `%h` - 机器名称
    - `%m` - 日志内容
    - `%d` - 时间，默认以 ISO8601 规范格式化，可选规范有 `ISO8601`, `ISO8601_WITH_TZ_OFFSET`, `ABSOUTE`, `DATE` 或者任何可支持格式化的字串，例如 `%d{DATE}` 或者 `%d{yyyy/MM/dd-hh.mm.ss}`。
    - `%%` - 输出 `%` 字符串
    - `%n` - 换行
    - `%z` - 从 `process.pid` 中获取的进程 ID
    - `%[` - 颜色块开始区域
    - `%]` - 颜色块结束区域

自定义 handle

如果觉得提供的日志输出类型不满足大家的需求，可以自定义日志处理的 `handle`。自定义 handle 需要实现一下几个方法：

```
module.exports = class {
  /**
   * @param {Object}  config  {}  配置传入的参数
   * @param {Boolean} clusterMode true  是否是多进程模式
   */
  constructor(config, clusterMode) {

  }

  debug() {

  }

  info() {

  }

  warn() {

  }

  error() {

  }
}
```

多进程

node 中提供了 `cluster` 模块来创建多进程应用，这样可以避免单一进程挂了导致服务异常的情况。框架是通过 think-cluster 模块来运行多进程模型的。

多进程配置

可以配置 `workers` 指定子进程的数量，默认为 `0`（当前 cpu 的个数）

```
//src/config/config.js
module.exports = {
  workers: 0 // 可以根据实际情况修改，0 为 cpu 的个数
}
```

多进程模型

多进程模型下，Master 进程会根据 `workers` 的大小 `fork` 对应数量的 Worker 进程，由 Woker 进程来处理用户的请求。当 Worker 进程异常时会通知 Master 进程 Fork 一个新的 Worker 进程，并让当前 Worker 不再接收用户的请求。

进程间通信

多个 Worker 进程之间有时候需要进行通信，交换一些数据。但 Worker 进程之间并不能直接通信，而是需要借助 Master 进程来中转。

框架提供 `think.messenger` 来处理进程之间的通信，目前有下面几种方法：

- `think.messenger.broadcast` 将消息广播到所有 Worker 进程

```
//监听事件
think.messenger.on('test', data => {
//所有进程都会捕获到该事件，包含当前的进程
});

if(xxx){
//发送广播事件
think.messenger.broadcast('test', data)
}
```

自定义进程通信

有时候内置的一些通信方式还不能满足所有的需求，这时候可以自定义进程通信。由于 Master 进程执行时调用 `src/bootstrap/master.js`，Worker 进程执行时调用 `src/bootstrap/worker.js`，那么处理进程通信就比较简单。

```
// src/bootstrap/master.js
process.on('message', (worker, message) => {
  // 接收到特定的消息进程处理
  if(message && message.act === 'xxx'){

  }
})

// src/bootstrap/worker.js
process.send({act: 'xxxx', ...args}); //发送数据到 Master 进程
```

Babel 转译

由于框架依赖的 Node 最低版本为 `6.0.0`，但这个版本还不支持 `async/await`，所以在项目里使用 `async/await` 时，需要借助 babel 转译。

Babel 会将 `src/` 目录转译到 `app/` 目录下，并添加 `sourceMap` 文件。

关闭 Babel 转译

如果想关闭 Babel 转译，那么需要 Node 的版本大于 `7.6.0`（推荐使用 8.x.x LTS 版本），创建项目时可以指定 `-w` 参数来关闭 Babel 转译。

```
thinkjs new demo -w;
```

其实使不使用 Babel 转译，其实只是入口文件里引用有一些区别。

- 有 Babel 转译的入口文件（development.js）

```
const Application = require('thinkjs');
const babel = require('think-babel');
const watcher = require('think-watcher');
const notifier = require('node-notifier');
```

```
const instance = new Application({
  ROOT_PATH: __dirname,
  watcher: watcher,
  transpiler: [babel, {  //转译器，这里使用的是 babel，并指定转译参数
    presets: ['think-node']
  }],
  notifier: notifier.notify.bind(notifier), //通知器，当转译报错时如何通知
  env: 'development'
});

instance.run();
```

- 去除 Babel 转译的入口文件（development.js）

```
const Application = require('thinkjs');
const watcher = require('think-watcher');
const instance = new Application({
  ROOT_PATH: __dirname,
  watcher: watcher,
  env: 'development'
});

instance.run();
```

对比可以看到，去除 Babel 转译，只是移除了 `transpiler` 和 `notifier` 2 个配置，一个是指定转译器，一个是当转译报错时的通知处理方式。

当然除了指定 `-w` 参数关闭 Babel 转译，手工删除 `development.js` 里的相关代码也是可以的。

RESTful API

项目中，经常要提供一个 API 供第三方使用，一个通用的 API 设计规范就是使用 REST API。REST API 是使用 HTTP 中的请求类型来标识对资源的操作。如：

- `GET` `/ticket` 获取 ticket 列表
- `GET` `/ticket/:id` 查看某个具体的 ticket
- `POST` `/ticket` 新建一个 ticket
- `PUT` `/ticket/:id` 更新 ticket 12
- `DELETE` `/ticket/:id` 删除 ticekt 12

创建 REST Controller

可以通过 `-r` 参数来创建 REST Controller。如：

```
thinkjs controller user -r
```

会创建下面几个文件：

```
create : src/controller/rest.js
create : src/controller/user.js
create : src/logic/user.js
```

其中 `src/controller/user.js` 会继承 `src/controller/rest.js` 类， `rest.js` 会 REST Controller 的基类，具体的逻辑可以根据项目如果进行修改。

添加自定义路由

添加 REST Controller 后并不能立即对其访问，需要添加对应的自定义路由。修改路由配置文件 `src/config/router.js`，添加如下的配置：

```
module.exports = [
  [/\/user(?:\/(\d+))?/, 'user?id=:1', 'rest']
]
```

上面自定义路由的含义为：

- `/\/user(?:\/(\d+))?/` URL 的正则
- `user?id=:1` 映射后要解析的路由，:1 表示取正则里的 (\d+) 的值
- `rest` 表示为 REST API

通过自定义路由，将 `/user/:id` 相关的请求指定为 REST Controller，然后就可以对其访问了。

- `GET /user` 获取用户列表，执行 `getAction`
- `GET /user/:id` 获取某个用户的详细信息，执行 `getAction`
- `POST /user` 添加一个用户，执行 `postAction`
- `PUT /user/:id` 更新一个用户，执行 `putAction`
- `DELETE /user/:id` 删除一个用户，执行 `deleteAction`

数据校验

Controller 里的方法执行时并不会对传递过来的数据进行校验，数据校验可以放在 Logic 里处理，文件为 `src/logic/user.js`，具体的 Action 与 Controller 里一一对应。具体的使用方式请见 Logic。

子级 REST API

有时候有子级 REST API，如：某篇文章的评论接口，这时候可以通过下面的自定义路由完成：

```
module.exports = [
  [/\/post\/(\d+)\/comments(?:\/(\d+))?/, 'comment?postId=:1&id=:2', 'rest']
]
```

这样在对应的 Action 里，可以通过 `this.get("postId")` 来获取文章的 id，然后放在过滤条件里处理即可。

多版本 REST API

有些 REST API 有时候前后不能完全兼容，需要有多个版本，这时候也可以通过自定义路由管理，如：

```
module.exports = [
  [/\/v1\/user(?:\/(\d+))?/, 'v1/user?id=:1', 'rest'], //v1 版本
  [/\/v2\/user(?:\/(\d+))?/, 'v2/user?id=:1', 'rest']  //v2 版本
]
```

这时候只要在 `src/controller/` 下建立子目录 `v1/` 和 `v2/` 即可，执行时会自动查找，具体见多级控制器。

Mongo 的 REST API

由于 Mongo 的 id 并不是纯数字的，所以处理 Mongo 的 REST API 时只需要修改下对应的正则即可（将 \d 改为 \w）：

```
module.exports = [
  [/\/user(?:\/(\w+))?/, 'user?id=:1', 'rest']
]
```

模型

模型介绍

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

目前支持的数据库有：`MySQL`。

创建模型

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

```
thinkjs model user;
```

执行完成后，会创建文件 `src/model/user.js`。

模型属性

model.pk

主键 key，默认为 `id`。

model.schema

数据表字段定义，默认会从数据库中读取，读到的信息类似如下：

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

可以在模型添加额外的属性，如：默认值和是否只读，如：

```
const moment = require('moment');
```

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

`default` 默认只在添加数据时有效。如果希望在更新数据时也有效，需要添加属性 `update: true`。

`readonly` 只在更新时有效。

`注`：如果设置了 `readonly`，那么会忽略 `update` 属性。

更多属性请见 Model -> API。

模型实例化

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

```
module.exports = class extends think.Controller {
  indexAction(){
    let model = this.model('user');
  }
}
```

否则可以通过调用 `think.model` 方法获取实例化，如：

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

使用 `think.model` 获取模型的实例化时，需要带上模型的配置。

链式调用

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

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

模型中支持链式调用的方法有：

- `where`，用于查询或者更新条件的定义
- `table`，用于定义要操作的数据表名称
- `alias`，用于给当前数据表定义别名
- `data`，用于新增或者更新数据之前的数据对象赋值
- `field`，用于定义要查询的字段，也支持字段排除
- `order`，用于对结果进行排序
- `limit`，用于限制查询结果数据
- `page`，用于查询分页，生成 sql 语句时会自动转换为 limit
- `group`，用于对查询的 group 支持
- `having`，用于对查询的 having 支持
- `join`，用于对查询的 join 支持
- `union`，用于对查询的 union 支持
- `distinct`，用于对查询的 distinct 支持
- `cache` 用于查询缓存

链式调用方法具体使用方式请见 Model -> API。

数据库配置

数据库配置

- 修改以下内容，并将其写入到 `src/config/adapter/model.js` 文件中：
  `js module.exports = { type: 'mysql', common: { log_sql: true, log_connect: true, }, adapter: { mysql: { host: '127.0.0.1', port: '', database: '', //数据库名称 user: '', //数据库帐号 password: '', //数据库密码 prefix: 'think_', encoding: 'utf8' } } };`
- 编辑 `src/config/adater.js` 文件，增加 `exports.model = require('./adapter/model.js')` 。

也可以在其他模块下配置，这样请求该模块时就会使用对应的配置。

数据表定义

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

如果需要修改，可以通过下面 2 个属性进行：

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

```
module.exports = class extends think.Model {
  constructor(...args) {
    super(...args);
    this.tablePrefix = ''; //将数据表前缀设置为空
    this.tableName = 'user2'; //将对应的数据表名设置为 user2
  }
}
```

修改主键

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

```
module.exports = class extends think.Mode {
  constructor(...args) {
    super(...args);
    this.pk = 'user_id'; //将主键字段设置为 user_id
  }
}
```

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

配置多个数据库

如果项目中有连接多个数据库的需求，可以通过下面的方式连接多个数据库。

```
//src/config/adapter/model.js
module.exports = {
  type: 'mysql',
  mysql: {
    host: '127.0.0.1',
    port: '',
    database: 'test1',
    user: 'root1',
    password: 'root1',
    prefix: '',
    encoding: 'utf8'
  },
  mysql2: {
    type: 'mysql', //这里需要将 type 重新设置为 mysql
    host: '127.0.0.1',
    port: '',
    database: 'test2',
    user: 'root2',
    password: 'root2',
    prefix: '',
    encoding: 'utf8'
  }
}
```

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

配置完成后，调用的地方可以通过下面的方式调用。

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

分布式数据库

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

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

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

CRUD 操作

添加数据

添加一条数据

使用 `add` 方法可以添加一条数据，返回值为插入数据的 id。如：

```
module.exports = class extends think.Controller {
  async addAction(){
    let model = this.model('user');
    let insertId = await model.add({name: 'xxx', pwd: 'yyy'});
  }
}
```

有时候需要借助数据库的一些函数来添加数据，如：时间戳使用 mysql 的 `CURRENT_TIMESTAMP` 函数，这时可以借助 `exp` 表达式来完成。

```
export default class extends think.controller.base {
  async addAction(){
    let model = this.model('user');
    let insertId = await model.add({
      name: 'test',
      time: ['exp', 'CURRENT_TIMESTAMP()']
    });
  }
}
```

添加多条数据

使用 `addMany` 方法可以添加多条数据，如：

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

thenAdd

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

模型中提供了 `thenAdd` 方法简化这一操作。

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

更新数据

update

更新数据使用 `update` 方法，返回值为影响的行数。如：

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

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

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

有时候更新值需要借助数据库的函数或者其他字段，这时候可以借助 `exp` 来完成。

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

increment

可以通过 `increment` 方法给符合条件的字段增加特定的值，如：

```
module.exports = class extends think.Model {
  updateViewNums(id){
```

```
    return this.where({id: id}).increment('view_nums', 1); //将阅读数加 1
  }
}
```

decrement

可以通过 `decrement` 方法给符合条件的字段减少特定的值，如：

```
module.exports = class extends think.Model {
  updateViewNums(id){
    return this.where({id: id}).decrement('coins', 10); //将金币减 10
  }
}
```

查询数据

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

查询单条数据

可以使用 `find` 方法查询单条数据，返回值为对象。如：

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

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

查询多条数据

可以使用 `select` 方法查询多条数据，返回值为数据。如：

```
module.exports = class extends think.Controller {
  async listAction(){
```

```
    let model = this.model('user');
    let data = await model.limit(2).select();
    //data returns [{name: 'thinkjs', email: 'admin@thinkjs.org'}, ...]
  }
}
```

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

分页查询数据

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

```
module.exports = class extends think.Controller {
  async listAction(){
    let model = this.model('user');
    let data = await model.page(this.get('page'), 10).countSelect();
  }
}
```

返回值格式如下：

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

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

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

count

可以通过 `count` 方法查询符合条件的总条数，如：

```
module.exports = class extends think.Model {
  getMin(){
    //查询 status 为 publish 的总条数
    return this.where({status: 'publish'}).count();
  }
}
```

sum

可以通过 `sum` 方法查询符合条件的字段总和，如：

```
module.exports = class extends think.Model {
  getMin(){
    //查询 status 为 publish 字段 view_nums 的总和
    return this.where({status: 'publish'}).sum('view_nums');
  }
}
```

max

可以通过 `max` 方法查询符合条件的最大值，如：

```
module.exports = class extends think.Model {
  getMin(){
    //查询 status 为 publish，字段 comments 的最大值
    return this.where({status: 'publish'}).max('comments');
  }
}
```

min

可以通过 `min` 方法查询符合条件的最小值，如：

```
module.exports = class extends think.Model {
  getMin(){
    //查询 status 为 publish，字段 comments 的最小值
    return this.where({status: 'publish'}).min('comments');
  }
}
```

查询缓存

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

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

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

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

```
module.exports = class extends think.Model {
  getList(){
    //只设定缓存时间
    return this.cache(3600).where({id: {'>': 100}}).select();
  }
}
```

缓存配置

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

```
module.exports = {
  cache: {
    on: true,
    type: '',
    timeout: 3600
  }
}
```

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

删除数据

可以使用 `delete` 方法来删除数据，返回值为影响的行数。如：

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

模型中更多的操作方式请见相关的 API -> model。

事务

模型中提供了对事务操作的支持，但前提需要数据库支持事务。

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

`SQLite` 直接支持事务。

使用事务

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

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

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

```
}
```

transaction 方法

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

```
module.exports = class extends think.Controller {
  async indexAction(self){
    let model = this.model('user');
    let insertId = await model.transaction(async () => {
      let userId = await model.add({name: 'xxx'});
      return await self.model('user_group').add({user_id: userId, group_id: 1000})
;
    });
  }
}
```

---

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

操作多个模型

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

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

关联模型

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

ThinkJS 中支持关联模型，让处理这类操作非常简单。

支持的类型

关联模型中支持常见的 4 类关联关系。如：

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

创建关联模型

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

```
thinkjs model post --relation
```

会创建模型文件 `src/model/post.js`。

指定关联关系

可以通过 `relation` 属性来指定关联关系。如：

```
module.exports = class extends think.Model.Relation {
  constructor(...args) {
    super(...args);
    //通过 relation 属性指定关联关系，可以指定多个关联关系
    this.relation = {
      cate: {},
      comment: {}
    };
  }
}
```

也可以直接使用 ES7 里的语法直接定义 `relation` 属性。如：

```
module.exports = class extends think.Model.Relation {

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

单个关系模型的数据格式

```
module.exports = class extends think.Model.Relation {
  constructor(...args){
    super(...args);
    this.relation = {
      cate: {
        type: think.Model.Relation.MANY_TO_MANY, //relation type
        model: '', //model name
        name: 'profile', //data name
        key: 'id',
        fKey: 'user_id', //forign key
        field: 'id,name',
        where: 'name=xx',
        order: '',
        limit: '',
        rModel: '',
        rfKey: ''
      },
    };
  }
}
```

各个字段含义如下：

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

如果只用设置关联类型，不用设置其他字段信息，可以通过下面简单的方式：

```
module.exports = class extends think.Model.Relation {
  constructor(...args){
    super(...args);
```

```
    this.relation = {
      cate: think.Model.Relation.MANY_TO_MANY
    };
  }
}
```

HAS_ONE

一对一关联，表示当前表含有一个附属表。

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

```
module.exports = class extends think.Model.Relation {
  init(..args){
    super(...args);
    this.relation = {
      info: think.Model.Relation.HAS_ONE
    };
  }
}
```

执行查询操作时，可以得到类似如下的数据：

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

BELONG_TO

一对一关联，属于某个关联表，和 HAS_ONE 是相反的关系。

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

```
module.exports = class extends think.Model.Relation {
  init(..args){
```

```
    super(...args);
    this.relation = {
      user: think.Model.Relation.BELONG_TO
    };
  }
}
```

执行查询操作时，可以得到类似下面的数据：

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

HAS_MANY

一对多的关系。

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

```
module.exports = class extends think.Model.Relation {
  constructor(...args){
    super(...args);

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

执行查询数据时，可以得到类似下面的数据：

```
[{
  id: 1,
  title: 'first post',
```

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

如果关联表的数据需要分页查询，可以通过 `page` 参数进行，如：

```
module.exports = class extends think.Model.Relation {
  constructor(...args){
    super(...args);

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

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

MANY_TO_MANY

多对多关系。

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

```
module.exports = class extends think.Model.Relation {
  constructor(...args){
    super(...args);

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

```
}
```

查询出来的数据结构为：

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

关联死循环

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

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

```
module.exports = class extends think.Model.Relation {
  constructor(...args){
    super(...args);
    this.relation = {
      user: {
        type: think.Model.Relation.BELONG_TO,
        relation: false //关联表 user 查询时关闭关联查询
      }
    };
  }
}
```

也可以设置只关闭当前模型的关联关系，如：

```
module.exports = class extends think.Model.Relation {
  constructor(...args){
    super(...args);
    this.relation = {
      user: {
        type: think.Model.Relation.BELONG_TO,
        relation: 'info' //关联表 user 查询时关闭对 info 模型的关联关系
      }
    };
```

```
  }
}
```

临时关闭关联关系

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

全部关闭

通过 `setRelation(false)` 关闭所有的关联关系查询。

```
module.exports = class extends think.Model.Relation {
  constructor(...args){
    super(...args);
    this.relation = {
      comment: think.Model.Relation.HAS_MANY,
      cate: think.Model.Relation.MANY_TO_MANY
    };
  }
  getList(){
    return this.setRelation(false).select();
  }
}
```

部分启用

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

```
module.exports = class extends think.Model.Relation {
  constructor(...args){
    super(...args);
    this.relation = {
      comment: think.Model.Relation.HAS_MANY,
      cate: think.Model.Relation.MANY_TO_MANY
    };
  }
  getList2(){
    return this.setRelation('comment').select();
  }
}
```

部分关闭

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

```
module.exports = class extends think.Model.Relation {
  constructor(...args){
    super(...args);
    this.relation = {
      comment: think.Model.Relation.HAS_MANY,
      cate: think.Model.Relation.MANY_TO_MANY
    };
  }
  getList2(){
    return this.setRelation('comment', false).select();
  }
}
```

重新全部启用

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

```
module.exports = class extends think.Model.Relation {
  constructor(...args){
    super(...args);
    this.relation = {
      comment: think.Model.Relation.HAS_MANY,
      cate: think.Model.Relation.MANY_TO_MANY
    };
  }
  getList2(){
    return this.setRelation(true).select();
  }
}
```

设置查询条件

field

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

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

```
module.exports = class extends think.Model.Relation {
  constructor(...args){
    super(...args);
    this.relation = {
      user: {
        type: think.Model.Relation.BELONG_TO,
        field: 'id,name,email' //必须要包含关联字段 id
      }
    };
  }
}
```

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

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

形参 `model` 指向当前模型类。

where

设置 where 可以控制查询关联表时的查询条件，如：

```
module.exports = class extends think.Model.Relation {
  constructor(...args){
    super(...args);
    this.relation = {
      user: {
        type: think.Model.Relation.BELONG_TO,
        where: {
          grade: 1 //只查询关联表里 grade = 1 的数据
```

```
        }
      }
    };
  }
}
```

也可以动态的设置 where 条件，如：

```
module.exports = class extends think.Model.Relation {
  constructor(...args){
    super(...args);
    this.relation = {
      user: {
        type: think.Model.Relation.BELONG_TO,
        where: model => model._relationWhere
      }
    };
  }
  selectData(relationWhere){
    this._relationWhere = relationWhere;
    return this.select();
  }
}
```

形参 `model` 指向当前模型类。

page

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

```
module.exports = class extends think.Model.Relation {
  constructor(...args){
    super(...args);
    this.relation = {
      user: {
        type: think.Model.Relation.BELONG_TO,
        page: [1, 15] //第一页，每页 15 条
      }
    };
  }
}
```

也可以动态设置分页，如：

```
module.exports = class extends think.Model.Relation {
```

```
  constructor(...args){
    super(...args);
    this.relation = {
      user: {
        type: think.Model.Relation.BELONG_TO,
        page: model => model._relationPage
      }
    };
  }
  selectData(page){
    this._relationPage = [page, 15];
    return this.select();
  }
}
```

形参 `model` 指向当前模型类。

limit

可以通过 limit 设置查询的条数，如：

```
module.exports = class extends think.Model.Relation {
  constructor(...args){
    super(...args);
    this.relation = {
      user: {
        type: think.Model.Relation.BELONG_TO,
        limit: [10] //限制 10 条
      }
    };
  }
}
```

也可以动态设置 limit，如：

```
module.exports = class extends think.Model.Relation {
  constructor(...args){
    super(...args);
    this.relation = {
      user: {
        type: think.Model.Relation.BELONG_TO,
        limit: model => model._relationLimit
      }
    };
  }
  selectData(){
    this._relationLimit = [1, 15];
```

```
    return this.select();
  }
}
```

形参 `model` 指向当前模型类。

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

order

通过 order 可以设置关联表的查询排序方式，如：

```
module.exports = class extends think.Model.Relation {
  constructor(...args){
    super(...args);
    this.relation = {
      user: {
        type: think.Model.Relation.BELONG_TO,
        order: 'level DESC'
      }
    };
  }
}
```

也可以动态的设置 order，如：

```
module.exports = class extends think.Model.Relation {
  constructor(...args){
    super(...args);
    this.relation = {
      user: {
        type: think.Model.Relation.BELONG_TO,
        order: model => model._relationOrder
      }
    };
  }
  selectData(){
    this._relationOrder= 'level DESC';
    return this.select();
  }
}
```

形参 `model` 指向当前模型类。

注意事项

关联字段的数据类型要一致

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

Mysql

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

连接池

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

```
module.exports = {
  common: {
    connectionLimit: 10 //建立 10 个连接
  }
}
```

socketPath

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

```
module.exports = {
  common: {
    socketPath: '/tmp/mysql.socket'
  }
}
```

SSL options

可以通过下面的配置来指定通过 SSL 来连接：

```
const fs = require('fs');
module.exports = {
  common: {
    ssl: {
      ca: fs.readFileSync(__dirname + '/mysql-ca.crt')
    }
  }
}
```

数据库支持 emoji 表情

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

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

```
module.exports = {
  common: {
    encoding: 'utf8mb4'
  }
}
```

model

自定义模型需要先集成 `think.Model` 类

```
module.exports = class extends think.Model {
  getList(){

  }
}
```

属性

model.pk

数据表主键，默认为 `id`。

model.name

模型名，默认从当前文件名中解析。

当前文件路径为 `for/bar/app/home/model/user.js`，那么解析的模型名为 `user`。

model.tablePrefix

数据表名称前缀，默认为 `think_`。

model.tableName

数据表名称，不包含前缀。默认等于模型名。

model.schema

数据表字段，关系型数据库默认自动从数据表分析。

model.indexes

数据表索引，关系数据库会自动从数据表分析。

model.config

配置，实例化的时候指定。

model._db

连接数据库句柄。

model._data

操作的数据。

model._options

操作选项。

方法

model.model(name, options, module)

- `name` {String} 模型名称
- `options` {Object} 配置项
- `module` {String} 模块名
- `return` {Object}

获取模型实例，可以跨模块获取。

```
module.exports = class extends think.Model {
  async getList(){
    //获取 user 模型实例
    let instance = this.model('user');
    let list = await instance.select();
    let ids = list.map(item => {
      return item.id;
    });
    let data = await this.where({id: ['IN', ids]}).select();
    return data;
  }
}
```

model.getTablePrefix()

- `return` {string}

获取表名前缀。

model.getConfigKey()

- `return` {String}

获取配置对应的 key，缓存 db 句柄时使用。

model.db()

- `return` {Object}

根据当前的配置获取 db 实例，如果已经存在则直接返回。

model.getModelName()

- `return` {String} 模型名称

如果已经配置则直接返回，否则解析当前的文件名。

model.getTableName()

- `return` {String} 获取表名，包含表前缀

获取表名，包含表前缀。

model.cache(key, timeout)

- `key` {String} 缓存 key
- `timeout` {Number} 缓存有效时间，单位为秒
- `return` {this}

设置缓存选项。

设置缓存 key 和时间

```
module.exports = class extends think.Model {
  getList(){
    return this.cache('getList', 1000).where({id: {'>': 100}}).select();
  }
}
```

只设置缓存时间，缓存 key 自动生成

```
module.exports = class extends think.Model {
  getList(){
    return this.cache(1000).where({id: {'>': 100}}).select();
  }
}
```

设置更多的缓存信息

```
module.exports = class extends think.Model {
  getList(){
    return this.cache({
      key: 'getList',
      timeout: 1000,
      type: 'file' //使用文件方式缓存
    }).where({id: {'>': 100}}).select();
  }
}
```

model.limit(offset, length)

- `offset` {Number} 设置查询的起始位置
- `length` {Number} 设置查询的数据长度
- `return` {this}

设置查询结果的限制条件。

限制数据长度

```
module.exports = class extends think.Model {
  getList(){
    //查询20条数据
    return this.limit(20).where({id: {'>': 100}}).select();
  }
}
```

限制数据起始位置和长度

```
module.exports = class extends think.Model {
  getList(){
    //从起始位置100开始查询20调数据
    return this.limit(100, 20).where({id: {'>': 100}}).select();
  }
}
```

也可以直接传入一个数组，如：

```
module.exports = class extends think.Model {
  getList(){
    //从起始位置100开始查询20调数据
    return this.limit([100, 20]).where({id: {'>': 100}}).select();
  }
}
```

model.page(page, listRows)

- `page` {Number} 当前页，从 1 开始
- `listRows` {Number} 每页的条数
- `return` {this}

设置查询分页数据，自动转化为 `limit` 数据。

```
module.exports = class extends think.Model {
  getList(){
    //查询第 2 页数据，每页 10 条数据
    return this.page(2, 10).where({id: {'>': 100}}).select();
  }
}
```

也可以直接设置一个参数为数组，关联模型等情况下可能会有用。如：

```
module.exports = class extends think.Model {
  getList(){
    //查询第 2 页数据，每页 10 条数据
    return this.page([2, 10]).where({id: {'>': 100}}).select();
  }
}
```

model.where(where)

- `where` {String | Object} where 条件
- `return` {this}

设置 where 查询条件。可以通过属性 `_logic` 设置逻辑，默认为 `AND`。可以通过属性 `_complex` 设置复合查询。

`注`:
1. 以下示例不适合 mongo model，mongo 中设置 where 条件请见 model.mongo 里的 where 条件设定。
2. where 条件中的值需要在 Logic 里做数据校验，否则可能会有漏洞。

普通条件

```
module.exports = class extends think.Model {
  where1(){
    //SELECT * FROM `think_user`
    return this.where().select();
  }
  where2(){
    //SELECT * FROM `think_user` WHERE ( `id` = 10 )
    return this.where({id: 10}).select();
  }
  where3(){
    //SELECT * FROM `think_user` WHERE ( id = 10 OR id < 2 )
```

```
    return this.where('id = 10 OR id < 2').select();
  }
  where4(){
    //SELECT * FROM `think_user` WHERE ( `id` != 10 )
    return this.where({id: ['!=', 10]}).select();
  }
}
```

null 条件

```
module.exports = class extends think.Model {
  where1(){
    //SELECT * FROM `think_user` where ( title IS NULL );
    return this.where({title: null}).select();
  }
  where2(){
    //SELECT * FROM `think_user` where ( title IS NOT NULL );
    return this.where({title: ['!=', null]}).select();
  }
}
```

EXP 条件

ThinkJS 默认会对字段和值进行转义，防止安全漏洞。有时候一些特殊的情况不希望被转义，可以使用 EXP 的方式，如：

```
module.exports = class extends think.Model {
  where1(){
    //SELECT * FROM `think_user` WHERE ( (`name` ='name') )
    return this.where({name: ['EXP', "=\"name\""]}).select();
  }
}
```

LIKE 条件

```
module.exports = class extends think.Model {
  where1(){
    //SELECT * FROM `think_user` WHERE ( `title` NOT LIKE 'welefen' )
    return this.where({title: ['NOTLIKE', 'welefen']}).select();
  }
  where2(){
    //SELECT * FROM `think_user` WHERE ( `title` LIKE '%welefen%' )
    return this.where({title: ['like', '%welefen%']}).select();
  }
  //like 多个值
```

```
  where3(){
    //SELECT * FROM `think_user` WHERE ( (`title` LIKE 'welefen' OR `title` LIKE '
suredy') )
    return this.where({title: ['like', ['welefen', 'suredy']]}).select();
  }
  //多个字段或的关系 like 一个值
  where4(){
    //SELECT * FROM `think_user` WHERE ( (`title` LIKE '%welefen%') OR (`content`
LIKE '%welefen%') )
    return this.where({'title|content': ['like', '%welefen%']}).select();
  }
  //多个字段与的关系 Like 一个值
  where5(){
    //SELECT * FROM `think_user` WHERE ( (`title` LIKE '%welefen%') AND (`content`
 LIKE '%welefen%') )
    return this.where({'title&content': ['like', '%welefen%']}).select();
  }
}
```

IN 条件

```
module.exports = class extens think.Model {
  where1(){
    //SELECT * FROM `think_user` WHERE ( `id` IN ('10','20') )
    return this.where({id: ['IN', '10,20']}).select();
  }
  where2(){
    //SELECT * FROM `think_user` WHERE ( `id` IN (10,20) )
    return this.where({id: ['IN', [10, 20]]}).select();
  }
  where3(){
    //SELECT * FROM `think_user` WHERE ( `id` NOT IN (10,20) )
    return this.where({id: ['NOTIN', [10, 20]]}).select();
  }
}
```

BETWEEN 查询

```
module.exports = class extens think.Model {
  where1(){
    //SELECT * FROM `think_user` WHERE (  (`id` BETWEEN 1 AND 2) )
    return this.where({id: ['BETWEEN', 1, 2]}).select();
  }
  where2(){
    //SELECT * FROM `think_user` WHERE (  (`id` BETWEEN '1' AND '2') )
    return this.where({id: ['between', '1,2']}).select();
  }
}
```

多字段查询

```
module.exports = class extends think.Model {
  where1(){
    //SELECT * FROM `think_user` WHERE ( `id` = 10 ) AND ( `title` = 'www' )
    return this.where({id: 10, title: "www"}).select();
  }
  //修改逻辑为 OR
  where2(){
    //SELECT * FROM `think_user` WHERE ( `id` = 10 ) OR ( `title` = 'www' )
    return this.where({id: 10, title: "www", _logic: 'OR'}).select();
  }
  //修改逻辑为 XOR
  where2(){
    //SELECT * FROM `think_user` WHERE ( `id` = 10 ) XOR ( `title` = 'www' )
    return this.where({id: 10, title: "www", _logic: 'XOR'}).select();
  }
}
```

多条件查询

```
module.exports = class extends think.Model {
  where1(){
    //SELECT * FROM `think_user` WHERE ( `id` > 10 AND `id` < 20 )
    return this.where({id: {'>': 10, '<': 20}}).select();
  }
  //修改逻辑为 OR
  where2(){
    //SELECT * FROM `think_user` WHERE ( `id` < 10 OR `id` > 20 )
    return this.where({id: {'<': 10, '>': 20, _logic: 'OR'}}).select()
  }
}
```

复合查询

```
module.exports = class extends think.Model {
  where1(){
    //SELECT * FROM `think_user` WHERE ( `title` = 'test' ) AND (  ( `id` IN (1,2,
3) ) OR ( `content` = 'www' ) )
    return this.where({
      title: 'test',
      _complex: {id: ['IN', [1, 2, 3]],
        content: 'www',
        _logic: 'or'
      }
    }).select()
```

```
  }
}
```

model.field(field)

- `field` {String | Array} 设置要查询的字段，可以是字符串，也可以是数组
- `return` {this}

设置要查询的字段。

字符串方式

```
module.exports = class extends think.controller.base {
  async indexAction(){
    let model = this.model('user');
    //设置要查询的字符串，字符串方式，多个用逗号隔开
    let data = await model.field('name,title').select();
  }
}
```

调用 SQL 函数

```
module.exports = class extends think.controller.base {
  //字段里调用 SQL 函数
  async listAction(){
    let model = this.model('user');
    let data = await model.field('id, INSTR(\'30,35,31,\',id + \',\') as d').selec
t();
  }
}
```

数组方式

```
module.exports = class extends think.controller.base {
  async indexAction(){
    let model = this.model('user');
    //设置要查询的字符串，数组方式
    let data = await model.field(['name','title']).select();
  }
}
```

model.fieldReverse(field)

- `field` {String | Array} 反选字段，即查询的时候不包含这些字段
- `return` {this}

设置反选字段，查询的时候会过滤这些字段，支持字符串和数组 2 种方式。

model.table(table, hasPrefix)

- `table` {String} 表名
- `hasPrefix` {Boolean} 是否已经有了表前缀，如果 table 值含有空格，则不在添加表前缀
- `return` {this}

设置表名，可以将一个 SQL 语句设置为表名。

设置当前表名

```
module.exports = class extends think.Model {
  getList(){
    return this.table('test', true).select();
  }
}
```

SQL 语句作为表名

```
module.exports = class extends think.Model {
  async getList(){
    let sql = await this.model('group').group('name').buildSql();
    let data = await this.table(sql).select();
    return data;
  }
}
```

model.union(union, all)

- `union` {String | Object} 联合查询 SQL 或者表名
- `all` {Boolean} 是否是 UNION ALL 方式
- `return` {this}

联合查询。

SQL 联合查询

```
module.exports = class extends think.Model {
  getList(){
    //SELECT * FROM `think_user` UNION (SELECT * FROM think_pic2)
    return this.union('SELECT * FROM think_pic2').select();
  }
}
```

表名联合查询

```
module.exports = class extends think.Model {
  getList(){
    //SELECT * FROM `think_user` UNION ALL (SELECT * FROM `think_pic2`)
    return this.union({table: 'think_pic2'}, true).select();
  }
}
```

model.join(join)

- `join` {String | Object | Array} 要组合的查询语句，默认为 `LEFT JOIN`
- `return` {this}

组合查询，支持字符串、数组和对象等多种方式。

字符串

```
module.exports = class extends think.Model {
  getList(){
    //SELECT * FROM `think_user` LEFT JOIN think_cate ON think_group.cate_id=think
_cate.id
    return this.join('think_cate ON think_group.cate_id=think_cate.id').select();
  }
}
```

数组

```
module.exports = class extends think.Model {
  getList(){
    //SELECT * FROM `think_user` LEFT JOIN think_cate ON think_group.cate_id=think
_cate.id RIGHT JOIN think_tag ON think_group.tag_id=think_tag.id
    return this.join([
      'think_cate ON think_group.cate_id=think_cate.id',
      'RIGHT JOIN think_tag ON think_group.tag_id=think_tag.id'
    ]).select();
```

```
  }
}
```

对象：单个表

```
module.exports = class extends think.Model {
  getList(){
    //SELECT * FROM `think_user` INNER JOIN `think_cate` AS c ON think_user.`cate_
id`=c.`id`
    return this.join({
      table: 'cate',
      join: 'inner', //join 方式，有 left, right, inner 3 种方式
      as: 'c', // 表别名
      on: ['cate_id', 'id'] //ON 条件
    }).select();
  }
}
```

对象：多次 JOIN

```
module.exports = class extends think.Model {
  getList(){
    //SELECT * FROM think_user AS a LEFT JOIN `think_cate` AS c ON a.`cate_id`=c.`
id` LEFT JOIN `think_group_tag` AS d ON a.`id`=d.`group_id`
    return this.alias('a').join({
      table: 'cate',
      join: 'left',
      as: 'c',
      on: ['cate_id', 'id']
    }).join({
      table: 'group_tag',
      join: 'left',
      as: 'd',
      on: ['id', 'group_id']
    }).select()
  }
}
```

对象：多个表

```
module.exports = class extends think.Model {
  getList(){
    //SELECT * FROM `think_user` LEFT JOIN `think_cate` ON think_user.`id`=think_c
ate.`id` LEFT JOIN `think_group_tag` ON think_user.`id`=think_group_tag.`group_id`
    return this.join({
      cate: {
```

```
        on: ['id', 'id']
      },
      group_tag: {
        on: ['id', 'group_id']
      }
    }).select();
  }
}
```

```
module.exports = class extends think.Model {
  getList(){
    //SELECT * FROM think_user AS a LEFT JOIN `think_cate` AS c ON a.`id`=c.`id` L
EFT JOIN `think_group_tag` AS d ON a.`id`=d.`group_id`
    return this.alias('a').join({
      cate: {
        join: 'left', // 有 left,right,inner 3 个值
        as: 'c',
        on: ['id', 'id']
      },
      group_tag: {
        join: 'left',
        as: 'd',
        on: ['id', 'group_id']
      }
    }).select()
  }
}
```

对象：ON 条件含有多个字段

```
module.exports = class extends think.Model {
  getList(){
    //SELECT * FROM `think_user` LEFT JOIN `think_cate` ON think_user.`id`=think_c
ate.`id` LEFT JOIN `think_group_tag` ON think_user.`id`=think_group_tag.`group_id`
 LEFT JOIN `think_tag` ON (think_user.`id`=think_tag.`id` AND think_user.`title`=t
hink_tag.`name`)
    return this.join({
      cate: {on: 'id, id'},
      group_tag: {on: ['id', 'group_id']},
      tag: {
        on: { // 多个字段的 ON
          id: 'id',
          title: 'name'
        }
      }
    }).select()
  }
}
```

对象：table 值为 SQL 语句

```
module.exports = class extends think.Model {
  async getList(){
    let sql = await this.model('group').buildSql();
    //SELECT * FROM `think_user` LEFT JOIN ( SELECT * FROM `think_group` ) ON thin
k_user.`gid`=( SELECT * FROM `think_group` ).`id`
    return this.join({
      table: sql,
      on: ['gid', 'id']
    }).select();
  }
}
```

model.order(order)

- `order` {String | Array | Object} 排序方式
- `return` {this}

设置排序方式。

字符串

```
module.exports = class extends think.Model {
  getList(){
    //SELECT * FROM `think_user` ORDER BY id DESC, name ASC
    return this.order('id DESC, name ASC').select();
  }
  getList1(){
    //SELECT * FROM `think_user` ORDER BY count(num) DESC
    return this.order('count(num) DESC').select();
  }
}
```

数组

```
module.exports = class extends think.Model {
  getList(){
    //SELECT * FROM `think_user` ORDER BY id DESC,name ASC
    return this.order(['id DESC', 'name ASC']).select();
  }
}
```

对象

```
module.exports = class extends think.Model {
  getList(){
    //SELECT * FROM `think_user` ORDER BY `id` DESC,`name` ASC
    return this.order({
      id: 'DESC',
      name: 'ASC'
    }).select();
  }
}
```

model.alias(tableAlias)

- `tableAlias` {String} 表别名
- `return` {this}

设置表别名。

```
module.exports = class extends think.Model {
  getList(){
    //SELECT * FROM think_user AS a;
    return this.alias('a').select();
  }
}
```

model.having(having)

- `having` {String} having 查询的字符串
- `return` {this}

设置 having 查询。

```
module.exports = class extends think.Model {
  getList(){
    //SELECT * FROM `think_user` HAVING view_nums > 1000 AND view_nums < 2000
    return this.having('view_nums > 1000 AND view_nums < 2000').select();
  }
}
```

model.group(group)

- `group` {String} 分组查询的字段

- `return` {this}

设定分组查询。

```
module.exports = class extends think.Model {
  getList(){
    //SELECT * FROM `think_user` GROUP BY `name`
    return this.group('name').select();
  }
}
```

model.distinct(distinct)

- `distinct` {String} 去重的字段
- `return` {this}

去重查询。

```
module.exports = class extends think.Model {
  getList(){
    //SELECT DISTINCT `name` FROM `think_user`
    return this.distinct('name').select();
  }
}
```

model.explain(explain)

- `explain` {Boolean} 是否添加 explain 执行
- `return` {this}

是否在 SQL 之前添加 explain 执行，用来查看 SQL 的性能。

model.optionsFilter(options)

操作选项过滤。

model.dataFilter(data)

- `data` {Object | Array} 要操作的数据

数据过滤。

model.beforeAdd(data)

- `data` {Object} 要添加的数据

添加前置操作。

model.afterAdd(data)

- `data` {Object} 要添加的数据

添加后置操作。

model.afterDelete(data)

删除后置操作。

model.beforeUpdate(data)

- `data` {Object} 要更新的数据

更新前置操作。

model.afterUpdate(data)

- `data` {Object} 要更新的数据

更新后置操作。

model.afterFind(data)

- `data` {Object} 查询的单条数据
- `return` {Object | Promise}

`find` 查询后置操作。

model.afterSelect(data)

- `data` [Array] 查询的数据数据
- `return` {Array | Promise}

`select` 查询后置操作。

model.data(data)

- `data` {Object}

添加和更新操作时设置操作的数据。

model.options(options)

- `options` {Object}

设置操作选项。如：

```
module.exports = class extends think.Model {
  getList(){
    return this.options({
      where: 'id = 1',
      limit: [10, 1]
    }).select();
  }
}
```

model.close()

关闭数据库连接，一般情况下不要直接调用。

model.getSchema(table)

- `table` {String} 表名
- `return` {Promise}

获取表的字段信息，自动从数据库中读取。

model.getTableFields(table)

`已废弃`，使用 model.getSchema 方法替代。

model.getLastSql()

- `return` {String}

获取最后执行的 SQL 语句。

model.buildSql()

- `return` {Promise}

将当前的查询条件生成一个 SQL 语句。

model.parseOptions(oriOpts, extraOptions)

- `oriOpts` {Object}
- `extraOptions` {Object}
- `return` {Promise}

根据已经设定的一些条件解析当前的操作选项。

model.getPk()

- `return` {Promise}

返回 `pk` 的值，返回一个 Promise。

model.parseType(field, value)

- `field` {String} 数据表中的字段名称
- `value` {Mixed}
- `return` {Mixed}

根据数据表中的字段类型解析 value。

model.parseData(data)

- `data` {Object} 要解析的数据
- `return` {Object}

调用 `parseType` 方法解析数据。

model.add(data, options, replace)

- `data` {Object} 要添加的数据
- `options` {Object} 操作选项
- `replace` {Boolean} 是否是替换操作
- `return` {Promise} 返回插入的 ID

添加一条数据。

model.thenAdd(data, where)

- `data` {Object} 要添加的数据
- `where` {Object} where 条件
- `return` {Promise}

当 where 条件未命中到任何数据时才添加数据。

model.addMany(dataList, options, replace)

- `dataList` {Array} 要添加的数据列表
- `options` {Object} 操作选项
- `replace` {Boolean} 是否是替换操作
- `return` {Promise} 返回插入的 ID

一次添加多条数据。

model.delete(options)

- `options` {Object} 操作选项
- `return` {Promise} 返回影响的行数

删除数据。

删除 id 为 7 的数据。

```
model.delete({
  where: {id: 7}
});
```

model.update(data, options)

- `data` {Object} 要更新的数据
- `options` {Object} 操作选项
- `return` {Promise} 返回影响的行数

更新数据。

model.thenUpdate(data, where)

- `data` {Object} 要更新的数据

- `where` {Object} where 条件
- `return` {Promise}

当 where 条件未命中到任何数据时添加数据，命中数据则更新该数据。

updateMany(dataList, options)

- `dataList` {Array} 要更新的数据列表
- `options` {Object} 操作选项
- `return` {Promise}

更新多条数据，dataList 里必须包含主键的值，会自动设置为更新条件。

model.increment(field, step)

- `field` {String} 字段名
- `step` {Number} 增加的值，默认为 1
- `return` {Promise}

字段值增加。

model.decrement(field, step)

- `field` {String} 字段名
- `step` {Number} 增加的值，默认为 1
- `return` {Promise}

字段值减少。

model.find(options)

- `options` {Object} 操作选项
- `return` {Promise} 返回单条数据

查询单条数据，返回的数据类型为对象。如果未查询到相关数据，返回值为 `{}`。

model.select(options)

- `options` {Object} 操作选项
- `return` {Promise} 返回多条数据

查询多条数据，返回的数据类型为数组。如果未查询到相关数据，返回值为 `[]`。

model.countSelect(options, pageFlag)

- `options` {Object} 操作选项
- `pageFlag` {Boolean} 当页数不合法时处理，true 为修正到第一页，false 为修正到最后一页，默认不修正
- `return` {Promise}

分页查询，一般需要结合 `page` 方法一起使用。如：

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

返回值数据结构如下：

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

model.getField(field, one)

- `field` {String} 字段名，多个字段用逗号隔开
- `one` {Boolean | Number} 获取的条数
- `return` {Promise}

获取特定字段的值。

model.count(field)

- `field` {String} 字段名
- `return` {Promise} 返回总条数

获取总条数。

model.sum(field)

- `field` {String} 字段名
- `return` {Promise}

对字段值进行求和。

model.min(field)

- `field` {String} 字段名
- `return` {Promise}

求字段的最小值。

model.max(field)

- `field` {String} 字段名
- `return` {Promise}

求字段的最大值。

model.avg(field)

- `field` {String} 字段名
- `return` {Promise}

求字段的平均值。

model.query(...args)

- `return` {Promise}

指定 SQL 语句执行查询。

model.execute(...args)

- `return` {Promise}

执行 SQL 语句。

model.parseSql(sql, ...args)

- `sql` {String} 要解析的 SQL 语句
- `return` {String}

解析 SQL 语句，调用 `util.format` 方法解析 SQL 语句，并将 SQL 语句中的 `__TABLENAME__` 解析为对应的表名。

```
module.exports = class extends think.Model {
  getSql(){
    let sql = 'SELECT * FROM __GROUP__ WHERE id=%d';
    sql = this.parseSql(sql, 10);
    //sql is SELECT * FROM think_group WHERE id=10
  }
}
```

model.startTrans()

- `return` {Promise}

开启事务。

model.commit()

- `return` {Promise}

提交事务。

model.rollback()

- `return` {Promise}

回滚事务。

model.transaction(fn)

- `fn` {Function} 要执行的函数
- `return` {Promise}

使用事务来执行传递的函数，函数要返回 Promise。

```
module.exports = class extends think.Model {
  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;
    })
  }
}
```

扩展功能

多模块项目

一般的项目我们推荐使用单模块项目，如果项目较为复杂的话，可以使用多级控制器来按功能划分。如果这些还不能满足项目复杂度的需求，那么可以创建多模块项目。

创建项目时可以指定 `--mode=multi` 参数创建多模块项目。

```
thinkjs new demo --mode=multi
```

项目结构

项目结构跟单模块项目结构上有一些差别：

- `src/common` 存放一些公共的代码
- `src/home` 默认的模块
- `src/xxx` 按照功能添加模块

线上部署

定时任务