ThinkJS 1.2 Documentation

快速入门

介绍

ThinkJS 是一款高效、简单易用的 Node.js MVC 框架。基于 ES6 Promise 开发，让异步编程更加简单、方便。

特性

- 自动启动服务
- 支持 Http、命令行、WebSocket、Restful等多种方式调用
- CBD 架构，MVC 模式
- 基于 Promise，异步编程更加简单
- 封装了 Db, Cache, Session 等常用功能
- 开发模式下文件修改后立即生效，无需重启

安装、创建项目

ThinkJS 需要 Node.js 的版本 `>=0.10.x`，可以通过 `node -v` 命令查看当前 node 的版本。如果未安装 node 或者版本过低，请到 Node.js 官网进行安装或升级。

使用 ThinkJS 时，假设你已经有了 Node.js 开发相关的经验。

安装 ThinkJS

安装 ThinkJS 非常简单，通过如下的命令即可安装：

```Bash
npm install -g thinkjs-cmd
```

如果安装失败，可能是 npm 服务异常或者是被墙了，可以使用国内的 cnpm 服务进行安装。如：

```Bash
npm install -g thinkjs-cmd --registry=http://r.cnpmjs.org
```

安装完成后，可以通过下面的命令查看 ThinkJS 的版本号：

```JavaScript
thinkjs -v
```

如果能看到下面的字符，说明已经安装成功了。

```
 v1.1.0


 _______ _     _       _        _  _____
|__   __| |   (_)     | |      | |/ ____|
   | |  | |__  _ _ __ | | __   | | (___
   | |  | '_ \| |'_ \| |/ /   | |\___ \
   | |  | | | | | | | |   <|__| |____) |
   |_|  |_| |_|_|_| |_|_|\_\____/|_____/
```

更新 ThinkJS

更新 ThinkJS，分成 2 种，一种是更新系统的 ThinkJS 版本，更新后后续创建项目时使用新版的 ThinkJS。另一种是更新已有项目下的 ThinkJS 版本。

更新系统的 ThinkJS

```
npm update -g thinkjs-cmd; // 在 *nix 下，需要加上 sudo 执行
```

更新项目里的 ThinkJS

```
cd 项目目录;
npm update thinkjs;
```

新建项目

```Bash
# 在合适的位置创建一个新目录，new_dir_name 为你想创建的文件夹名字
mkdir new_dir_name;
# 进入这个目录
cd new_dir_name;
# 通过 thinkjs 命令创建项目
thinkjs .
```

执行后，如果当前环境有浏览器，会自动用浏览器打开 http://127.0.0.1:8360，并且会看到如下的内容：

```
hello, ThinkJS!
```

看到这个内容后，说明项目已经成功创建。

手动启动项目

创建项目时，会自动通过子进程来启动 node 服务。为了后续开发方便，最好还是手动来启动。

通过键盘操作 `ctrl + c` 结束当前的进程，cd 到 `www` 目录下，执行 `node index.js` 来启动服务。

`注：` 有些系统下用 apt-get 来安装 Node.js 的话，命令名可能为 nodejs。

项目结构说明

创建项目后，会生成如下的目录结构：

```
├── App
│   ├── Common
│   │   └── common.js    ---- 通用函数文件，一般将项目里的一些全局函数放在这里
│   ├── Conf
│   │   └── config.js    ---- 项目配置文件
│   ├── Lib
│   │   ├── Behavior     ---- 行为类存放位置
│   │   ├── Controller
│   │   │   └── Home
│   │   │       └── IndexController.js   ---- 逻辑控制类
│   │   └── Model        ---- 模型类
│   ├── Runtime          ---- 运行时的一些文件
│   │   ├── Cache        ---- 缓存目录
│   │   ├── Data         ---- 数据目录
│   │   ├── Log
│   │   └── Temp
│   └── View
│       └── Home
│           └── index_index.html      ---- 模版文件，默认使用 ejs 模版引擎
├── ctrl.sh              ---- 项目启动、停止脚本
└── www
    ├── index.js         ---- 入口文件
    └── resource         ---- 静态资源目录
        ├── css          ---- css 文件
        ├── img          ---- 图片文件
        ├── js           ---- js 文件
        ├── module       ---- 第三方的一些组件
        └── swf          ---- flash 文件
```

文件说明

下面对几个重要的文件进行简单的说明。

入口文件

`www/index.js`

```JavaScript
// 定义 APP 的根目录
global.APP_PATH = __dirname + "/../App";
// 静态资源根目录
global.RESOURCE_PATH = __dirname;
global.ROOT_PATH = __dirname;
global.APP_DEBUG = true; // 是否开启 DEBUG 模式
require('thinkjs');
```

默认开启 debug 模式，该模式下文件修改后立即生效，不必重启 node 服务。

线上环境切记要将 debug 模式关闭，即：APP_DEBUG=false

debug 模式详细说明请见 调试 里相关内容。

配置文件

`App/Conf/config.js`

```JavaScript
module.exports = {
  // 配置项：配置值
  port: 8360, // 监听的端口
  db_type: 'mysql', // 数据库类型
  db_host: 'localhost', // 服务器地址
  db_port: '', // 端口
  db_name: '', // 数据库名
  db_user: 'root', // 用户名
  db_pwd: '', // 密码
  db_prefix: 'think_', // 数据库表前缀
};
```

可以在配置文件中修改框架默认的配置值，如：将 http 监听的端口号由默认的 8360 改为 1234，那么这里加上 `"port": 1234`，重启服务后就生效了 (ps: 要把 url 中的端口号改为 1234 才能正常访问哦)。

框架默认的配置值请见 附录 - 默认配置

函数文件

`App/Common/common.js`

该文件下定义一些当前应用常用的函数，可以直接放在 global 下，该文件在系统启动时自动加载。

```JavaScript
global.getDate = function(){return 'xxx';};
global.getSliceUrl = function(url, length){return '';}
```

这些函数在其他地方可以直接使用，无需在 require。

控制器文件

`App/Lib/Controller/Home/IndexController.js`

```JavaScript
/**
 * controller
 * @return
 */
module.exports = Controller({
  indexAction: function(){
    //render View/Home/index_index.html file
    this.display();
  };
});
```

该文件为一个基础的控制器文件，只有一个 indexAction，这个 action 直接渲染 `View/Home/index_index.html` 模版文件。

除了渲染文件，你可以直接输出字符串。可以将这里改为 `this.end('hello word')`，刷新浏览器后，显示为 hello word。

控制器详细内容请见 控制器 相关内容。

进阶应用

配置

ThinkJS 提供了灵活的全局配置功能，这些配置值随着服务启动而生效，并且在后续所有的 http 请求中都有效。

系统支持默认配置、公共配置、调试配置、模式配置等多种配置方式。

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

配置格式

```JavaScript
// 配置格式
module.exports = {
    'port': 1234,
    'db_host': '127.0.0.1', // 服务器地址
    'db_name': 'think_web', // 数据库名
    'db_user': 'root', // 用户名
    'db_pwd': '', // 密码
    'use_websocket': true, // 使用 websocket
}
```

注意： 配置参数的 key 不区分大小写（key 会强制转为小写）。建议使用小写，便于阅读。

配置值除了是简单的数据外，也可以是数组、对象、函数等。

```JavaScript
// 配置
module.exports = {
    'list': ['1', '2'],
    'fn': function(){
        //do something
    }
}
```

配置加载

配置加载遵循下面的加载顺序，且后面加载的配置值覆盖前面加载的值。

`系统默认配置 -> 应用配置 -> 调试配置 -> 模式配置`

系统默认配置

系统默认配置包含了所有的 ThinkJS 中用到的配置，并给出了默认值。该文件在 ThinkJS 的 `lib/Conf/config.js`，你可以在 附录 -> 默认配置 中查看详细的配置。

系统配置文件会在服务启动时自动调用。

应用配置

应用配置文件在 `App/Conf/config.js` 里，服务启动时会自动调用。

调试配置

如果在入口文件将 APP_DEBUG 设置为 true, 那么会自动读取调试配置，配置文件为 `App/Conf/debug.js`。

如果该文件不存，则不加载。

模式配置

ThinkJS 除了默认的启 http 服务运行，也可以命令行下运行，命令行下对应的模式为 cli。

模式配置文件为 `App/Conf/mode.js`，该文件内容格式如下：

```JavaScript
// 配置
module.exports = {
    'cli': {  // 命令行模式下的配置
       'use_cluster': false
    },
    'cli_debug': {// 命令行模式开启了 debug 的配置

    }
}
```

配置读取

无论何种配置文件，定义了配置后，都统一使用函数 `C` 来获取配置值。如：

```JavaScript
var dbHost = C('db_host');
```

配置写入

配置写入除了默认加载预订的配置文件外，也可以通过函数 `C` 写入配置，如：

```JavaScript
// 设置配置
C('name', 'welefen');
// 设置二级配置
C('cache.db_time', 60);
// 也可以批量设置配置
C({
    'name': 'welefen',
    'use_cluster': false
})
```

`注意：` 如果是一级配置，则配置名中不能含有字符 `.`，含有 `.` 会自动当作二级配置。

扩展配置

如果配置项比较多的话，这时候可能就需要考虑将配置分类了，并且存放在不同的配置文件里。可以通过如下的方式进行：

```JavaScript
// 配置文件写入这个配置，表示额外加载 cache.js 和 db.js 2 个配置文件，配置文件也在 Conf 目录下
'load_ext_config': ['cache', 'db']
```

有这个配置后，系统启动时会自动加载 `App/Conf/cache.js` 和 `App/Conf/db.js` 2 个配置文件。

架构

模块化设计

ThinkJS 基于 `分组/控制器/操作` 的设计原则，一个典型的 URL 如下：

`http://hostname:port/分组/控制器/操作/参数名/参数值/参数名2/参数值 2?arg1=argv1&arg2=argv2`

- `分组` 一个应用下有多个分组，一个分组都是很独立的模块。比如：前台模块、用户模块、管理员模块
- `控制器` 一个分组下有多个控制器，一个控制器是多个操作的集合。如：商品的增删改查
- `操作` 一个控制器有多个操作，每个操作都是最小的执行单元。如：添加一个商品

项目中有哪些分组，需要在如下的配置中指定：

```JavaScript
// 支持的分组列表
'app_group_list': ['Home', 'Admin'], // 表示有 Home 和 Admin 2 个分组
```

默认是哪个分组，可以在下面的配置中指定：

```
// 默认分组
'default_group': 'Home', // 你可以将默认分组改成合适的，如：Blog
```

CBD 模式

ThinkJS 使用 CBD(核心 Core+ 行为 Behavior+ 驱动 Driver) 的架构模式，核心保留了最关键的部分，并在重要位置添加了 `切面`，其他功能都是以驱动的方式来完成。

核心 (Core)

ThinkJS 的核心部分包含通用函数库、系统默认配置、核心类库等组成，这些都是 ThinkJS 必不可少的部分。

```
lib/Common/common.js 通用函数库
lib/Common/extend.js js 原生对象的扩展
lib/Common/function.js 框架相关的函数库
lib/Conf/alias.js  系统类库别名，加载时使用
lib/Conf/config.js 系统默认配置
lib/Conf/debug.js debug 模式下的配置
lib/Conf/mode.js  不同模式下的配置
lib/Conf/tag.js  每个切面下的行为
lib/Lib/Core/App.js  应用核心库
lib/Lib/Core/Controller.js 控制器基类
lib/Lib/Core/Db.js  数据库基类
lib/Lib/Core/Dispatcher.js 路由分发类
lib/Lib/Core/Http.js 封装的 http 对象类
lib/Lib/Core/Model.js 模型基类
lib/Lib/Core/Think.js 框架类
lib/Lib/Core/View.js 视图类
lib/Lib/Util/Behavior.js 行为基类
lib/Lib/Util/Cache.js 缓存基类
lib/Lib/Util/Cookie.js cookie 类
lib/Lib/Util/Filter.js 数据过滤类
lib/Lib/Util/Session.js session 基类
lib/Lib/Util/Valid.js 验证类
```

行为 (Behavior)

行为是 ThinkJS 扩展机制中一项比较关键的扩展，行为可以独立调用，也可以整合到标签 (tag) 里一起调用，行为是执行过程中一个动作或事件。如：路由检测是个行为、静态缓存检测也是个行为。

标签 (tag) 是一组行为的集合，是在系统执行过程中切面处调用的。与 `EventEmitter` 不同，标签里的行为是按顺序执行的，当前的行为通过 Promise 机制控制后面的行为是否被执行。

系统标签位

当执行一个 http 请求时，会在对应的时机执行如下的标签位：

- `app_init` 应用初始化
- `path_info` 解析 path 路径
- `resource_check` 静态资源请求检测
- `route_check` 路由检测
- `app_begin` 应用开始
- `action_init` action 初始化
- `view_init` 视图初始化
- `view_template` 模版定位
- `view_parse` 模版解析
- `view_filter` 模版内容过滤
- `view_end` 视图结束
- `action_end` action 结束
- `app_end` 应用结束

在每一个标签位置都可以配置多个行为，系统的标签位行为如下：

```JavaScript
/**
 * 系统标签配置
 * 可以在 App/Conf/tag.js 里进行修改
 * @type {Object}
 */
module.exports = {
    // 应用初始化
    app_init: [],
    //pathinfo 解析
    path_info: [],
    // 静态资源请求检测
    resource_check: ['CheckResource'],
    // 路由检测
    route_check: ['CheckRoute'],
    // 应用开始
    app_begin: ['ReadHtmlCache'],
    //action 执行初始化
    action_init: [],
    // 模版解析初始化
    view_init: [],
    // 定位模版文件
    view_template: ["LocationTemplate"],
    // 模版解析
    view_parse: ["ParseTemplate"],
    // 模版内容过滤
    view_filter: [],
    // 模版解析结束
    view_end: ['WriteHtmlCache'],
    //action 结束
    action_end: [],
    // 应用结束
    app_end: []};
```

除了系统的标签位行为，开发人员也可以根据项目的需要自定义标签位行为。

自定义标签位行为文件在 `App/Conf/tag.js` 。

行为定义

行为定义有 2 种方式，一种是一个简单的 function，一种是较为复杂些的行为类。

可以通过下面直接 function 的方式创建一个简单的行为，文件为 `App/Conf/tag.js` ：

```JavaScript
module.exports = {app_begin: [
    function(http){ // 会传递一个包装的 http 对象
        if (http.group !== 'Home') {
            return;
        };
        var userAgent = http.getHeader('user-agent').toLowerCase();
        var flag = ["iphone", "android"].some(function(item){
            return userAgent.indexOf(item) > -1;
        })
        if (flag) {
            http.group = "Mobile";
        }
    }]
}
```

该行为的作用是：如果当前的分组是 Home 并且是手机访问，那么将分组改为 Mobile。这样就可以对同一个 url，PC 和 Mobile 访问执行不同的逻辑，输出不同的内容。

也可以继承行为基类来实现：

```JavaScript
module.exports = Behavior(function(){
    return {
        run: function(){
            var http = this.http; // 基类中的 init 方法会自动把传递进来的 http 对象放在当前对象上。
            if (http.group !== 'Home') {
                return;
            };
            var userAgent = http.getHeader('user-agent').toLowerCase();
            var flag = ["iphone", "android"].some(function(item){
                return userAgent.indexOf(item) > -1;
            })
            if (flag) {
                http.group = "Mobile";
            }
        }
    }
})
```

将内容保存在 `App/Lib/Behavior/AgentBehavior.js` 文件中，并在 `App/Conf/tag.js` 中配置如下的内容：
`js module.exports = { app_begin: ["Agent"] }`

使用哪种方式来创建行为可以根据行为里的逻辑复杂度来选择。

行为执行顺序

默认情况下，自定义的行为会和系统的行为一起执行，并且自定义行为是追加到系统行为之后的。

如果想更改行为执行的顺序，可以通过下面的方式：

- `app_begin: [true, 'Agent']` 将数组的第一个值设置为 true, 表示自定义行为替换系统默认的行为，那么系统的默认行为则不在执行。
- `app_begin: [false, 'Agent']` 将数组的第一个值设置为 false, 表示自定义行为放在系统的默认行为之前执行。

自定义标签位执行和行为执行

上面提到的标签位和行为会在 http 执行过程中自动被调用，同时标签和行为也可以手工调用：

```
// 执行 check_auto 标签位，
//http 为包装的 http 对象，在整个 http 执行过程中都可以获取到
//data 为传过去的数据， 如果行为里需要的是多个数据，那么这里应该传递个数组
tag("check_auth", http, data);
// 执行 Agent 这个行为
B("Agent", http, data);
```

驱动 (Driver)

除了核心和行为外，ThinkJS 里的很多功能都是通过驱动来实现的，如：Cache, Session, Db 等。

驱动包括：

- `lib/Lib/Driver/Cache` 缓存驱动
- `lib/Lib/Driver/Db` 数据库驱动
- `lib/Lib/Driver/Session` Session 驱动
- `lib/Lib/Driver/Socket` Socket 驱动
- `lib/Lib/Driver/Template` 模版引擎驱动

如果有些功能框架里还没实现，如：mssql 数据库，那么开发人员可以在项目里 `App/Lib/Driver/Db/` 里实现。

自动加载

Node.js 里虽然提供了 require 来加载模块，但对于应用内的文件加载并没有给出快捷的加载方式。为此 ThinkJS 实现了一套快速加载机制，这些快速加载的文件包含：

- 系统核心文件
- 行为文件
- 各种驱动文件

通过全局函数 `thinkRequire` 来调用。例如：

```JavaScript
// 调用这些文件时会自动到对应的一些目录下查找
var db = thinkRequire("mssqlDb");
var model = thinkRequire("userModel");
var behavior = thinkRequire("AgentBehavior");
```

这些文件具体包含：`xxxBehavior`，`xxxModel`，`xxxController`，`xxxCache`，`xxxDb`，`xxxTemplate`，`xxxSocket`，`xxxSession`

如果是这些之外的文件通过 `thinkRequire` 加载，那么会调用系统 require 函数。

系统流程

系统的执行流程分为启动服务和响应用户请求 2 块：

启动服务

- 通过 `node index.js` 启动
- 调用系统入口文件 `think.js`
- 常量定义，获取 ThinkJS 的版本号
- 加载 `lib/Lib/Core/Think.js` 文件，调用 start 方法
- 加载系统的函数库、系统默认配置
- 捕获异常
- 加载项目函数库、配置文件、自定义路由配置、行为配置、额外的配置文件
- 合并 autoload 的查找路径列表，注册 autoload 机制
- 记录当前 Node.js 的进程 id
- 加载 `lib/Lib/Core/App.js` 文件，调用 run 方法
- 识别是否使用 cluster，开启 http 服务

响应用户请求

- 用户发生了 url 访问
- 执行标签位 `form_parse`
- 发送 `X-Powered-By` 响应头信息，值为 ThinkJS 的版本号
- 执行标签位 `app_init`
- 执行标签位 `resource_check`，判断当前请求是否是静态资源类请求。静态资源类请求执行标签位 `resource_output`
- 执行标签位 `path_info`，获取修改后的 pathname
- 执行标签位 `route_check`，进行路由检测，识别对应的 Group, Controller, Action
- 执行标签位 `app_begin`，检测当前请求是否有静态化缓存
- 执行标签位 `action_init`，实例化 Controller
- 调用 `__before` 方法，如果存在的话
- 调用对应的 action 方法
- 调用 `__after` 方法，如果存在的话
- 执行标签位 `view_init`，初始化模版引擎
- 执行标签位 `view_template`，查到模版文件的具体路径
- 执行标签位 `view_parse`，解析模版内容
- 执行标签位 `view_filter`，对解析后的内容进行过滤
- 执行标签位 `view_end`，模版渲染结束
- 执行标签位 `app_end`，应用调用结束

路由

路由是 ThinkJS 中一个非常重要的特性，通过自定义路由，可以让 url 更加友好。

当访问 `http://hostname:port/分组/控制器/操作/参数名/参数值/参数名2/参数值 2?arg1=argv1&arg2=argv2` 时，通过 `url.parse` 解析到的 `pathname` 为 `/分组/控制器 /操作/参数名/参数值/参数名2/参数值2`。

url 过滤

有时候为了搜索引擎友好或者其他原因时，pathname 值并不是直接 `/ 分组 / 控制器 / 操作 /` 的方式，而是含有一些前缀后者后缀。

比如：url 后加 `.html` 让其更加友好，这种情况下可以通过下面的配置使其在识别的时候去除。

```JavaScript
//url 过滤配置
'url_pathname_prefix': '', // 前缀过滤配置'url_pathname_suffix':'.html' // 后缀过滤配置
```

经过前缀和后缀去除后，pathname 的值就比较干净了。

`注：` 如果前缀和后缀去除还满足不了需求，可以通过标签位 `path_info` 来修正 pathname 值。

路由识别

拿到干净的 pathname 值后，默认通过 `/ 分组 / 控制器 / 操作` 的规则来切割识别。

如：pathname 为 `/admin/group/list` 识别后的分组为 `admin`，控制器 `group`，操作为 `list`。

这里的前提是分组 `admin` 必须在配置 `app_group_list` 列表中。 `app_group_list` 默认值为：

```JavaScript
// 分组列表
'app_group_list': ["Home", "Admin"],
```

如果不在配置列表中，那么识别到的分组为默认分组名称，控制器为 admin，操作为 group。

分组、控制器、操作默认值在如下的配置中：

```JavaScript
'default_group': 'Home', // 默认分组
'default_controller': 'Index', // 默认控制器
'default_action': 'index',  // 默认操作
// 操作默认后缀
'action_suffix': 'Action',
```

识别到对应的分组、控制器、操作后，会调用对应的 controller，执行对应的 action 方法。如：

pathname 为： `/admin/group/list` ，会加载 `App/Lib/Controller/Admin/GroupController.js` 文件，实例化该类，并调用 `listAction` 方法。

自定义路由

在实际的项目中，有些重要的接口我们想让 url 尽量简单。如：文章的详细页面，默认路由可能是： `article/detail/id/10` ，但我们想要的 url 是 `article/10` 这种更简洁的方式。这种 url 如果用默认的路由规则解析，解析出来的控制器和操作并不是我们想要的。

为此 ThinkJS 提供一套自定义路由的策略，需要开启如下的配置：

```JavaScript
// 默认情况下开启自定义路由
'url_route_on': true
```

开启了自定义路由功能后，就可以在路由配置文件中定义想要的路由解析了。路由配置文件为： `App/Conf/route.js` ，格式如下：

```JavaScript
//
// 自定义路由规则
module.exports = [
    ["规则 1", "需要识别成的 path"], // 规则到具体 pathname 的映射
    ["规则 2", {  // 同一个规则根据不同的请求方式对应到不同的 pathname 上
        "get": "get 请求时识别成的 path",
        "delete": "delete 请求时识别成的 path",
        "put,post": "put,post 请求时识别成的 path" // 请求方式为 put 或者 post 时对应的 pathname
    }]
]
```

ThinkJS 里提供了 3 种自定义路由的方式，下面逐一介绍：

正则路由

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

```JavaScript
// 正则路由
module.exports = [
    [/^doc\/?(.*)$/, "index/doc?doc=:1"]
]
```

这个规则表示将 `http://hostname:port/doc/xxx/yyy` 这种请求识别到 `index/doc` 下。

对于正则表达式中的每个变量 (即子模式) 部分，如果需要在后面的路由地址中引用，那么可以采用 :1, :2 这样的方式，序号就是子模式的序号。

这里将 doc 后面的 `xxx/yyy` 值设置到参数 `doc` 中，那么在 `IndexController.js` 文件的 `docAction` 方法就可以通过 `this.get("doc")` 来获取该值。

规则路由

规则路由包含静态地址和动态地址，或者是两种地址的结合，如：

```JavaScript
module.exports = [
    ["my", "user/info"],
    ["blog/:id", "Blog/detail"],
    ["group/:year/:month", "group/list"]
]
```

规则路由以 "/" 进行参数分割，每个参数中以 ":" 开头的参数表示动态参数，并且会自动生成一个对应的 GET 参数，如：上面的 blog/:id，可以在 Controller 里的 Action 里通过 `this.get('id')` 来获取 id 的值。

静态路由

静态路由是一种纯静态字符串的路由规则，如：

```JavaScript
module.exports = [
    ["top", "index/top"],
    ["info", "index/user/info"]
]
```

静态路由有时候做一些地址重定向的时候可能会用到。

控制器

控制器是分组下一类功能的集合，每个控制器是一个独立的类文件，每个控制器下有多个操作。

定义控制器

创建文件 `App/Lib/Controller/Home/ArticleController.js`，表示 Home 分组下有名为 Article 的控制器。文件内容如下：

```JavaScript
// 控制器文件定义
module.exports = Controller(function(){
    return {
        indexAction: function(){
            return this.diplay();
        },
        listAction: function(){
            return this.display();
        },
        detailAction: function(){
            var doc = this.get("doc");
        }
    }
});
```

控制器的名称采用驼峰法命名，且首字母大写。

控制器类必须继承于 Controller 或者 Controller 的子类，建议每个分组下都有个 `BaseController`，其他的 Controller 继承该分组下的 BaseController，如：

```JavaScript
// 继承 Home 分组下的 BaseController
module.exports = Controller("Home/BaseController", function(){
    return {
        indexAction: function(){

        }
    }
})
```

注意： 这里的 `Home/BaseController` 不能只写成 `BaseController`，那样会导致 BaseController 文件找不到，必须要带上分组名称。

初始化方法

js 本身并没有在一个类实例化时自动调用某个方法，但 ThinkJS 实现一套自动调用的机制。自动调用的方法名为 `init`，如果类有 init 方法，那么这个类在实例化时会自动调用 init 方法。

为实现 init 方法调用的机制，ThinkJS 里所有的类都是通过 Class 函数创建。

如果控制器里要重写 init 方法，那么必须调用父类的 init 方法，如：
`js module.exports = Controller(function(){ return { init: function(http){ // 这里会传递一个包装后的 http 对象进来 this.super("init", http); /`

这里说个技巧：一般系统后台都需要用户登录才能访问，但判断用户是否登录一般都是异步的，不可能在每个 Action 里都去判断是否已经登录。

这时候就可以在 init 方法里判断是否已经登录，并且把这个 promise 返回，后续的 action 执行则是在这个 then 之后执行。如：

```JavaScript
module.exports = Controller(function(){
    return {
        // 这里会传递一个包装后的 http 对象进来
        init: function(http){
            // 调用父级的 init 方法，并将 http 作为参数传递过去
            this.super("init", http);
            //login 请求不判断当前是否已经登录
            if(http.action === 'login'){return;}
            // 通过获取 session 判断是否已经登录
            var self = this;
            return this.session("userInfo").then(function(data){
                if(isEmpty(data)){
                    // 如果未登录跳转到登录页。由于 redirect 方法返回的是个 pendding promise，那么后面的 action 方法并不会被执行
                    return self.redirect("/login");
                }else{
                    // 设置后，后面的 action 里直接通过 this.userInfo 就可以拿到登录用户信息了
                    self.userInfo = data;
                }
            })
        }
    }
})
```

前置和后置操作

ThinkJS 支持前置和后置操作，默认的方法名为 `__before` 和 `__after`，可以通过如下的配置修改：

```JavaScript
// 前置、后置配置名称
"before_action_name": "__before",
"after_action_name": "__after"
```

调用 `__before` 和 `__after` 方法时，会将当前请求的 action 传递过去。

```JavaScript
// 前置和后置操作
module.exports = Controller(function(){
    return {
        __before: function(action){
            console.log(action)
        },
        __after: function(action){
            console.log(action);
        }
    }
})
```

`__before`，`action`，`__after` 是通过 promise 的链式调用的，如果当前操作返回一个 reject promise 或者 pedding promise，那么则会阻止后面的执行。

Action 参数自动绑定

请求参数的值，一般是在 action 里通过 get(name) 或者 post(name) 来获取，如：

```JavaScript
var name = this.get("name");
var value = this.post("value");
```

为了获取方便，ThinkJS 里提供了一种方便的获取参数的方式，将参数绑定到 Action 上，需要开启如下的配置：

```JavaScript
'url_params_bind': true // 该功能默认开启
```

开启后，就可以通过下面的方式来获取参数

```JavaScript
module.exports = Controller(function(){
    return {
        detailAction: function(id){
            // 参数绑定后，这里的参数 id 值即为传递过来的 id 值
            // 如：访问 /article/10 的话，这里的 id 值为 10
            // 这里可以是多个参数
        }
    }
})
```

` 注意 `：参数绑定是通过将函数 toString 后解析形参字符串得到的，如果代码上线时将 js 压缩的话那么就不能使用该功能了。

空操作

空操作是指系统在找不到请求的操作方法的时候，会定位到空操作（`__call`）方法执行，利用这个机制，可以实现错误页面和一些 url 的优化。

默认空操作对应的方法名为 `__call`，可以通过下面的配置修改：

```JavaScript
// 空操作对应的方法
'call_method': '__call'
```

```JavaScript
// 空操作方法
module.exports = Controller(function(){
    return {
        __call: function(action){
            console.log(action);
        }
    }
})
```

如果控制器下没有空操作对应的方法，那么访问一个不存在的 url 时则会报错。

空控制器

空控制器是指系统找不到请求的控制器名称的时候，系统会尝试定位到一个默认配置的控制器上。配置如下：

```JavaScript
// 空控制器
// 表示当访问一个不存在的控制器时，会执行 Home 分组下 IndexController 下的_404Action 方法
// 如果指定的控制器或者方法不存在，则会报错
call_controller: "Home:Index:_404"
```

在 `Home/IndexController.js` 里定义 `_404Action` 方法：

```JavaScript
module.exports = Controller({
    _404Action: function(){
        this.status(404); // 发送 404 状态码
        this.end('not found');
    }
})
```

当然你也可以不输出 404 信息，而是输出一些推荐的内容，这些根据项目需要来进行。

常用方法

这里列举一些常用的方法，详细的可以去 Api 文档里查看。

- `get(key)` 获取 get 参数值
- `post(key)` 获取 post 参数值
- `file(key)` 获取 file 参数值
- `isGet()` 当前是否是 get 请求
- `isPost()` 当前是否是 post 请求
- `isAjax()` 是否是 ajax 请求
- `ip()` 获取请求用户的 ip
- `redirect(url)` 跳转到一个 url，返回一个 pedding promise 阻止后面的逻辑继续执行
- `echo(data)` 输出数据，会自动调用 JSON.stringify
- `end(data)` 结束当前的 http 请求
- `json(data)` 输出 json 数据，自动发送 json 相关的 Content-Type
- `jsonp(data)` 输出 jsonp 数据，请求参数名默认为 `callback`
- `success(data)` 输出一个正常的 json 数据，数据格式为 `{errno: 0, errmsg: "", data: data}`，返回一个 pedding promise 阻止后续继续执行
- `error(errno, errmsg, data)` 输出一个错误的 json 数据，数据格式为 `{errno: errno_value, errmsg: string, data: data}`，返回一个 pedding promise 阻止后续继续执行
- `download(file)` 下载文件
- `assign(name, value)` 设置模版变量
- `display()` 输出一个模版，返回一个 promise
- `fetch()` 渲染模版并获取内容，返回一个 prmose，内容需要在 promise then 里获取
- `cookie(name, value)` 获取或者设置 cookie
- `session(name, value)` 获取或者设置 session
- `header(name, value)` 获取或者设置 header

- `action(name, data)` 调用其他 Controller 的方法，可以跨分组

一些技巧

将 promise 返回

Action 里一般会从多个地方拉取数据，如：从数据库中查询数据，这些接口一般都是异步的，并且包装成了 Promise。

我们知道，Promise 会通过 try{}catch(e){} 来捕获异常，然后传递到 catch 里。如：

```JavaScript
indexAction: function(){
    D('User').page(1).then(function(data){

    }).catch(function(error){
        //error 为异常信息
        console.log(error)
    })
}
```

如果不加 catch，那么出错后，我们将无法看到异常信息，这对调试是非常不方便的。并且有时候会从多个地方拉取数据，每个都加 catch 也极为不便。

为此，ThinkJS 会自动捕获异常，并打印到控制台。但需要在 action 将 Promise 返回，如：

```JavaScript
indexAction: function(){
    // 这里将 Promise return 后，如果有异常会打印到控制台里
    return D('User').page(1).then(function(){

    })
}
```

var self = this

JS 中，函数是作为一等公民存在的，所以经常有函数嵌套。这时候需要将 this 赋值给变量供内部使用。

在使用 ThinkJS 开发项目中，推荐使用 `self` 作为这个变量名。如：

```JavaScript
listAciton: function(){
    var self = this;
    return D('Group').select().then(function(data){
        self.success(data);
    })
}
```

模型（Mysql）

在 ThinkJS 中基础的模型类就是 `Model` 类，该类完成了基本的增删改查、连贯操作和统计查询，同时还有 thenadd, countSelect 等功能，更高级的功能封装在另外的模型扩展中。

模型定义

模型并非必须定义，只有当存在独立的业务逻辑或者属性的时候才需要定义

模型类通常都需要继承系统的 `Model` 类，如：
`js // 模型定义 module.exports = Model(function(){ return { // 获取用户列表 getList: function(){} } })`

该文件保存在： `App/Lib/Model/UserModel.js` 。

模型类大多数情况下都是操作对应的数据表的，如果按照系统的规范来命名模型类，大多数情况下都可以自动关联到对应的数据表。

模型类的命名表现在对应的模型文件上，如上面的 `UserModel.js` ，那么该模型名称为 `UserModel` 。

模型类的命令采用驼峰时的规则，并且首字母大写，去除数据表的前缀。如：

- `UserModel` 对应的数据表为 `think_user` ，这里的数据表前缀为 `think_`
- `UserGroupModel` 对应的数据表为 `think_user_group` ，数据表前缀也为 `think_`

数据表定义

在模型中，有几个跟数据表名称相关的数据可以定义：

- `tablePrefix` 表前缀，默认对应配置： `db_prefix`

- `tableName` 表名称，默认情况下等于模型名称，即上面的 `User`
- `trueTableName` 包含前缀的数据表名称，也就是数据库中的实际表明，默认情况下不用设置。
- `dbName` 数据库名称

默认属性

假设数据库 test 中有数据表 `think_user`，那么对应的模型类名称为 `UserModel`，对应的模型文件为：`App/Lib/Model/UserModel.js`。这时候计算出来的几个属性值为：

- `tablePrefix` = `think_`
- `tableName` = `user`
- `trueTableName` = `think_user`
- `dbName` = `test`

这几个值会自动计算，不用在 Model 类里设置。

自定义属性

假设另一个数据库 test2 中有数据表 `think2_users`，但模型类文件为 `App/Lib/Model/UserModel.js`，那么就要在模型类中作如下的定义：
`js // 自定义属性的模型 module.exports = Model(function(){ return { dbName: 'test2', trueTableName: 'think2_users' } })`

模型实例化

直接实例化

```JavaScript
// 通过 thinkRequire 加载 UserModel 对应的 js 文件
var userModel = thinkRequire("UserModel");
var instance = userModel();
// 传递参数实例化
var newInstance = userModel("users", {db_prefix: "think2_"});
```

模型类实例化的时候可以传 3 个参数，分别是：模型名、详细配置，也可以在详细配置里指定数据表前缀。

D 函数实例化

为了简化实例化模型的代码，提供一个全局的 `D` 函数:

```JavaScript
// 实例化模型类
var model = D('User');
```

效果和上面手工实例化是一样的，可以传入的参数也一致。

数据库

数据库支持

ThinkJS 目前还只支持 `mysql` 数据库，后续会支持更多的数据库。

数据库连接

模型实例化默认会读取如下的配置，需要在 `App/Conf/config.js` 中设置对应的值：：
`js // 数据库连接信息 db_type: "mysql", // 数据库类型 db_host: "localhost", // 服务器地址 db_port: "", // 端口 db_name: "", // 数据库名 db_user:`

如果使用其他的配置信息连接数据库，可以在模型实例化的时候传入对应的配置：

```JavaScript
//App/Conf/config.js 里额外的数据库配置信息
'ext_db_config': {
    db_type: "mysql", // 数据库类型
    db_host: "localhost", // 服务器地址
    db_port: "", // 端口
    db_name: "", // 数据库名
    db_user: "root", // 用户名
    db_pwd: "", // 密码
    db_prefix: "think_", // 数据库表前缀
};
// 实例化时传入对应的配置
var model = D('User', C('ext_db_config'));
```

链式调用

ThinkJS 的基础模型类提供了很多链式调用的方法（类似于 jQuery里的链式调用方式），可以有效的提高数据存取的代码清晰度和开发效率，并且支持所有的 CURD 操作。

链式调用是通过方法里返回 `this` 来实现的。

使用非常简单，如：要查询用户表中 name 为 welefen 的部分用户信息，可以用下面的方式：

```JavaScript
var promise = D('User').where({name: 'welefen'}).field('sex,date').find().then(function(data){
    //data 为当前查询到用户信息，如果没有匹配到相关的数据，那么 data 为一个空对象
})
```

`注意：` 由于数据库操作都是异步的，执行 find() 方法后，并不能返回结果，而是返回一个 promise，在 promise.then 方法里可以拿到对应的数据。

系统支持的链式调用有：

- `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

对于每个方法的具体使用请见 API 里的详细文档。

CURD 操作

添加数据

模型里添加数据使用 `add` 方法，示例如下：

```JavaScript
D("User").add({
    name: "welefen",
    pwd: "xxx",
    email: "welefen@gmail.com"
}).then(function(insertId){
    //insertId 为插入到数据库中的 id
}).catch(function(err){
    // 插入异常，比如：email 已经存在
})
```

也可以一次添加多条数据，使用 `addAll` 方法，示例如下： “`js D(‘User’).addAll([{ name: "welefen", email: "welefen@gmail.com" }, { name: "suredy", email: "suredy@gmail.com" }]).then(function(insertId){

}) ”`

`注意：` 这里的一次插入多条数据最终拼成的 sql 语句只有一条，不会智能切割。如果数据量非常大（如：>1W 条）的话，一次插入可能会导致报错，请自行切割分块插入。

更新数据

更新数据使用 `update` 方法，如：
`js // 更新 email 为 welefen@gmail.com 用户的密码 D('User').where({email: "welefen@gmail.com"}).update({pwd: "xxx"}).then(function(affectedR`

查询数据

在 ThinkJS 中读取数据的方式有多种，通常分为：读取单条数据、读取多条数据和读取字段值。

数据查询方法支持很多的连贯操作方法，如：`where`，`table`，`field`，`order`，`group`，`having`，`join`，`cache` 等。

```JavaScript
// 查询 id=1 的用户数据，只需要 name,email 字段值
D('User').where({id: 1}).field('name,email').find().then(function(data){
    // 如果数据存在 data 为 {name: "welefen", "email": "welefen@gmail.com"}
    // 数据不存在 data 为 {}
});
// 查询 score>1000 的 10 条用户数据，按 score 降序
D('User').where({score: ['>', 1000]}).order('score DESC').limit(10).select().then(function(data){
    //data 为一个数组，数组里每一项为一个用户的详细信息
    // 如果没有数据，那么 data 为空数组
})
// 查询所有的用户数据
D('User').count().then(function(count){
    //count 为具体的用户数
})
```

删除数据

删除数据的方法为 `delete`，如：

```JavaScript
// 删除 id 为 1 的数据
D('User').where({id: 1}).delete().then(function(affectedRows){
    //affectedRows 为影响的行数
});
```

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。那我们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。

为了让使用者更加方便，这里提供了 `thenAdd` 的方法来简化这个问题的处理逻辑。

```JavaScript
// 第二个参数为插入条件，当使用这个插入条件查询时，如果没有查到相关的数据，则进行插入操作。
// 表示数据库中没有 email 为 `welefen@gmail.com` 数据时才进行插入操作
D('User').thenAdd({
    name: "welefen",
    email: "welefen@gmail.com"
}, {email: "welefen@gmail.com"}).then(function(id){
    //id 为 exist id 或者是 insert id
    // 如果需要返回详细的数据，可以传入第三个参数 true，那么返回的数据格式为 {type: "exist", id: "123"} type 值为 exist 或者 add
});
```

countSelect

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。

为了方便开发者使用，ThinkJS 提供了 `countSelect` 的方法。

```JavaScript
// 按每页 20 条数据来展现文件
D('Article').page(this.get("page"), 20).countSelect().then(function(data){
    //data 的数据格式为
    {
        count: 123, // 总条数
        total: 10, // 总页数
        page: 1, // 当前页
        num: 20, // 每页显示多少条
        data: [{}, {}] // 详细的数据
    }
});
// 如果传入一个不存在的页数，比如：总页数只有 10 页，如果传入的页数为 20，默认情况下返回的数据为空
// 可以传入参数来修正这个值
//countSelect(true)  修正到第一页
//countSelect(false) 修正到最后一页
```

关于模型接口的详细使用说明请见 API - Model 。

自动校验

在把数据往数据库里添加或者更新的时候，我们需要对数据进行校验，防止非法的数据提交到数据库中。ThinkJS 支持在数据入库之前对数据进行校验。

`App/Lib/Model/ArticleModel.js`

```JavaScript
module.exports = Model({
    // 定义数据校验的字段
    fields: {
        title: {
            valid: ['required', 'length'],
            length_args: [10],
            msg: {
                required: '标题不能为空',
                length: '标题长度不能小于 10'
            }
        },
        url: {
            valid: ['url'],
            msg: 'url 格式不正确'
        }
    }
})
```

添加数据，如：

```JavaScript
indexAction: function(){
    var self = this;
    D('Article').add({
        title: 'xxx',
        url: 'xxx'
    }).catch(function(err){
        var data = JSON.parse(err.json_message);
        self.error(100, 'data error', data);
    })
}
```

这样在添加数据的时候，自定校验 title 和 url 的值是否合法。

具体是数据格式请见 数据校验。

视图

模版引擎

ThinkJS 里默认使用的模版引擎是 `ejs`，关于 `ejs` 的使用文档你可以看 这里。

ThinkJS 除了支持 `ejs` 模版，还支持如下的模版：

- `jade` 需要在项目里手动安装 jade 模块
- `swig` 需要在项目里手动安装 swig 模块

模版引擎相关的配置如下：

```JavaScript
// 模版引擎相关的配置
tpl_content_type: "text/html", // 模版输出时发送的 Content-Type 头信息
tpl_file_suffix: ".html", // 模版文件名后缀
tpl_file_depr: "_", //controller 和 action 之间的分隔符
tpl_engine_type: "ejs", // 模版引擎名称
tpl_engine_config: {}, // 这里定义模版引擎需要的一些额外配置，如：修改左右定界符
```

如果使用其他的模版引擎，那么需要扩展 Template 对应的 Driver，具体方式请见 ThinkJS 里的 Template Driver 写法。

模版文件

默认的模版文件路径使用的规则为：

** 视图目录 / 分组名 / 控制器名 + 控制器与操作之间的分隔符 (默认为 `_`)+ 操作名 + 模版文件后缀 (默认为 `.html`) **

如：访问 `/group/detail?id=10`，识别到的分组名为 `home`，控制器名为 `group`，操作名为 `detail`，那么对应的模版文件为：`App/View/Home/group_detail.html`

变量赋值

如果要在模版中使用变量，必须在控制器中把变量传递到模版，系统提供了 `assign` 方法对模版变量进行赋值。

```JavaScript
// 单个变量赋值
this.assign("name", "welefen");
//assign 除了可以变量赋值，可以进行读取之前赋值过的变量值
var name = this.assign("name");
// 多个变量通过一个对象一起赋值
this.assign({
    name: "welefen",
    email: "welefen@gmail.com",
    extra: {score: 100}
})
```

注： `ejs` 模版中使用到的变量必须要在控制器里进行赋值，否则会报错。

除了手工进行赋值外，系统默认将所有配置和 http 对象也赋值到了模版中。模版中可以通过下面的方式获取配置值和 http 对象值：

```JavaScript
config.server_domain // 获取配置中 server_domain 的值
http.get.name // 从 http 对象中获取 get 参数 name 的值
```

模版渲染

模版定义后就可以渲染模版输出，系统也支持直接渲染内容输出，模版赋值必须在模版渲染之前进行。

模版渲染使用 `display` 方法，如：

```JavaScript
this.display(); // 自动识别模版路径
this.display("home:group:list"); // 渲染 Home/group_list.html 模版文件
this.display("/home/welefen/xxx/a.html"); //
```

模版渲染是一个异步的过程， `this.display` 返回的是一个 promise，为了方便错误捕获，需要将这个 promise 返回。如：

```JavaScript
// 分组详细页代码示例
detailAction: function(){var id = this.get("id");
    var self = this;
    // 需要将这个 promise 返回，系统会自动捕获异常
    // 如果进行不 return，那么需要额外添加 catch 来获取异常
    return D('Group').where({id: id}).find().then(function(data){
        self.assign("data", data);
        // 将模版渲染返回
        return self.display();})
}
```

获取内容

系统提供了 `fetch` 方法来获取渲染后的模版内容，如：

```JavaScript
// 获取模版内容
this.fetch().then(function(content){
    //content 为渲染后的模版内容
    // 对 content 进行额外的过滤和替换操作，然后可以通过 this.end(content) 进行输出
})
```

ejs 的基本使用方法

ejs 通过 `<%` 和 `%>` 来构造基本的语法。

使用 `<%= %>` 与 `<%- %>` 来显示模板变量数据。 `=` 表示进行 html 转义， `-` 表示不转义。

使用 `<%if(condition){%> code <%}%>` 来进行条件判断

使用 `<%data.forEach(function(item){%> <%=item%> <%})%>` 来进行数据遍历

高级功能

缓存

在项目中，合理使用缓存对性能有很大的帮助。ThinkJS 提供了方便的缓存方式，包括：内存缓存、文件缓存、memcache 缓存等。同时提供了快捷的方法进行缓存的读取操作。

数据缓存

在 ThinkJS 中进行缓存的读取操作，一般情况下不要直接操作缓存类，系统提供了快捷函数 `S` 来进行操作。

缓存配置

```JavaScript
// 缓存配置
cache_type: "File", // 数据缓存类型
cache_timeout: 6 * 3600, // 数据缓存有效期，单位：秒
cache_path: CACHE_PATH,  // 缓存路径设置 (File 缓存方式有效)
cache_file_suffix: ".json", //File 缓存方式下文件后缀名
cache_gc_hour: [4], // 缓存清除的时间点，数据为小时
```

缓存类型

内存缓存

使用 Node.js 的内存来缓存，如果使用 cluster 最好不要使用该方式。

```
cache_type: "", //cache_type 为空
```

文件缓存

基于文件的缓存

```
cache_type: "File", //cache_type 为 File
```

Memcache 缓存

使用 Memache 来缓存

```
cache_type: 'Memcache', //cache_type 为 Memcache
```

使用 Memache 来缓存需要追加如下的配置：

```JavaScript
//memcache 缓存
memcache_host: "127.0.0.1", //memcache host
memcache_port: 11211, //memecache 端口
```

Redis 缓存

使用 Redis 来缓存

```
cache_type: 'Redis', //cache_type 为 Redis
```

使用 Redis 来缓存需要追加如下的配置：

```JavaScript
//redis 缓存
redis_host: "127.0.0.1", //redis host
redis_port: 6379, //redis 端口
```

缓存读写

```JavaScript
// 写入缓存
S("name", "welefen"); // 返回一个 promise
// 读取缓存
S("name").then(function(value){
    //value 为获取到的缓存值
});
// 删除缓存
S("name", null);
// 缓存写入时改变缓存类型和缓存时间
S("name", "value", {
    type: "memcache",
    timeout: 100 * 10000
});
// 修改缓存类型来获取缓存值
S("name", undefined, {type: "memcache"}).then(function(value){

})
```

缓存的读、写、删都是异步操作，都是返回一个 promise。后续的操作如果有依赖，必须在 promise then 里执行。

文件快速缓存

如果有些数据不会过期，只会被新的数据覆盖。那么使用 `S` 函数就不太合适了，系统提供一种不会过期的数据快速存取方法。

可以通过函数 `F` 来对文件内容进行快速的读取和写入操作。文件快速写入和读取的根目录为 `App/Runtime/Data/`

```JavaScript
// 数据快速写入
F("name", {value: "xxx"}); // 写入的文件为 App/Runtime/Data/name.json
// 数据读取
var value = F("name");
// 重新指定数据写入的根目录
F("name", {value: "xxx"}, "/tmp/data"); // 写入的文件为 /tmp/data/name.json
// 数据读取
var value = F("name", undefined, "/tmp/data");
```

文件快速缓存的读、写操作都是同步的。

数据库查询缓存

对于及时性要求不高的数据查询，可以使用查询缓存来提高性能。系统里提供了 `cache` 方法来对查询数据缓存，无需自己使用缓存方法进行缓存和读取。

数据库查询缓存有如下的配置：

```JavaScript
{
    // 数据库查询缓存配置
    db_cache_on: true, // 是否启用查询缓存，如果关闭那么 cache 方法则无效
    db_cache_type: "", // 缓存类型，默认为内存缓存
    db_cache_path: CACHE_PATH + "/db", // 缓存路径，File 类型下有效
    db_cache_timeout: 3600, // 缓存时间，默认为 1 个小时
}
```

```JavaScript
// 查询 score>100 的数据
D('User').cache(true).where({score: ['>', 100]}).select().then(function(data){  // 开启查询缓存后，如果有缓存则直接读取缓存，不再从数据库查询
})
```

如果使用了 `cache(true)`，则在查询的时候会根据当前拼装的 sql 生成 md5 值作为缓存 key。也可以手工指定缓存 key，如：

```JavaScript
D('User').cache('cache_name').select().then(function(data){
    //data
})
```

也可以指定单条查询的的缓存时间，如：

```JavaScript
// 手工将缓存时间修改为 3 个小时
D('User').cache(3 * 3600).select().then(function(){

});
// 手工指定缓存 key 并修改缓存时间
D('User').cache('cache_name', 3 * 3600).select().then(function(){

});
```

如果指定了缓存 key，那么在外部可以通过 `S` 函数来读取缓存，如：
`js // 如果缓存类型和通用的缓存类型不同，那么这里需要带上缓存类型进行读取 S('cache_name').then(function(data){})`

页面静态化

对于项目中的有些数据，如：文章详细页，这些数据一旦生成后就不怎么修改了。为了更快的提高文章详细页的访问速度，可以将文章详细页面整个缓存起来，下次访问的时候如果还在缓存时间内，那么直接读取静态 html 返回即可。

这种方式下不用执行控制器、数据库查询、模版渲染等操作了，可以大大提高访问的性能。

如果页面中显示用登录信息，这种方式就不太适合了，除非用户登录信息通过异步接口来获取。

要使用页面静态化的功能，需要开启配置 `html_cache_on`，并且配置 `html_cache_rules` 规则来控制哪些请求才启动页面静态化的功能。

详细配置

页面静态化缓存只能使用文件缓存的方式，详细的配置如下：

```JavaScript
// 页面静态化配置
html_cache_on: false, //HTML 静态缓存
html_cache_timeout: 3600, // 缓存时间，单位为秒
html_cache_rules: {}, // 缓存详细的规则
html_cache_path: CACHE_PATH + "/html", // 缓存目录
html_cache_file_callback: undefined, // 由缓存 key 生成具体的缓存文件的回调函数
html_cache_file_suffix: ".html", // 缓存文件后缀名
```

缓存的规则如下：

```JavaScript
// 缓存规则
"html_cache_rules": {"静态地址": ["静态规则", "缓存时间", "生成缓存文件的回调函数"],
    "静态地址": "静态规则"
}
```

静态地址

静态地址是用来匹配当前的路由规则，匹配的静态地址列表为：

```JavaScript
// 静态地址匹配列表
var list = [
    group + ":" + controller + ":" + action,
    controller + ":" + action,
    action,
    "*"
];
```

如果这时候访问的 url 为 `group/detail?id=10`，识别到的分组为 `home`，控制器为 `group`，操作为 `detail`，那么静态地址可以配置为 `home:group:detail`，这样就命中了当前的请求。

当前也可以配置更泛的规则来匹配。

静态规则

静态地址是命中一组类似的请求，如果这组类似的请求生成的缓存文件是一样的，那肯定不行。

可以通过静态规则来区别不同的请求，静态规则里可以获取当前请求的动态数据。如：

```JavaScript
// 静态规则
// 动态获取参数 id 的值
"home:group:detail": "home_group_detail_{id}"
```

可以获取的动态数据有：

- `{key}` 获取参数 key 的值，如：`{id}`，`{page}`
- `{:group}` 获取分组
- `{:controller}` 获取控制器
- `{:action}` 获取操作
- `{cookie.key}` 获取 cookie 下 key 的值，如：`{cookie.skin}`

通过这些动态数据就可以将每个不同的请求都区分开来。

缓存文件回调函数

静态规则一般都是一个较长的字符串，如果直接拿这个字符串当文件名去生成缓存文件，可以导致缓存目录下的文件数过多。这时候我们可能希望建立一些子目录来放这些缓存文件，通过回调函数可以确立详细的缓存文件存在路径。

默认的缓存文件存放路径对应的函数为：

```JavaScript
// 生成缓存文件的函数
var getCacheFilename = function(key){var value = md5(key);
    // 这里生成一级子目录
    return value[0] + "/" + value;
}
```

如果想重新定义生成缓存文件的函数，可以修改配置 `html_cache_file_callback`， 如：

```JavaScript
{
    "html_cache_file_callback": function(key, http){
        var value = md5(key);
        // 生成二级子目录
        return value[0] + "/" + value[1] + "/" + value;
    }
}
```

也可以修改某个特定的静态规则下的回调函数，直接配置规则里第二个参数即可。

Session

Node.js 本身并没有提供 Session 的功能，但一般网站都有用户登录的功能，为了方便开发者使用，ThinkJS 提供一套 session 的机制。

Session 都需要依赖浏览器端的一个 Cookie 来实现，然后把这个 Cookie 值作为 key 到对应的地方去查询，如果有相应的数值表示已经登录，否则表示没有登录。

配置

Session 有如下的配置：

```JavaScript
//Session 配置
session_name: "ThinkJS", //session 对应的 cookie 名称
session_type: "File", //session 存储类型，空为内存，还可以为 Db
session_path: "", //File 类型下文件存储位置，默认为系统的 tmp 目录
session_options: {}, //session 对应的 cookie 选项
session_sign: "", //session 对应的 cookie 使用签名，如果使用签名，这里填对应的签名字符串
session_timeout: 24 * 3600, //session 失效时间，单位：秒
```

默认的 Session 的存储方式是 File 类型，可以修改为内存或者 Db 的方式，对应的值为 `""` 和 `Db`。

如果使用 `cluster` 模式，则不能使用内存的方式。

使用 Db（Mysql）来存储 Session 需要建如下的数据表：

```
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;
```

将数据表名 `think_session` 改为项目里对应的前缀 +session。

使用 Session

在 Controller 里，可以通过下面的方式使用 Session：

```JavaScript
// 使用 Session
//
// 获取 session 里 userInfo 的值
this.session("userInfo").then(function(data){})
// 设置 Session
this.session("userInfo", {name: "welefen", "email": "welefen@gmail.com"}).then(function(){

})
// 删除 Session
this.session().then(function(){});
```

Session 的读、写、删除操作都是异步的。

过期 Session 清除策略

一个用户登录后如果长期不再登录，那么这个 Session 对应的缓存数据就可以删除了。如果不删除那么会导致数据量越来越大，影响性能。

ThinkJS 提供了一套定时删除过期 Session 的策略，由于 Session 类继承自 Cache，所以 Session 和 Cache 的清除策略是一样的。 可以通过如下的参数来清除：

```JavaScript
// 过期数据清除策略
cache_gc_hour: [4], // 缓存清除的时间点，数据为小时。
```

这里表示在凌晨 4 点的时候进行一次清除，你可以修改多个时间点进行清除。如：早上 3 点、早上 9 点、下午 5 点、晚上 10 点，那么配置值为 `[3, 9, 17, 22]`。

示例

判断用户是否登录：

```JavaScript
// 判断用户是否登录
var self = this;
return this.session("userInfo").then(function(data){
    if(isEmpty(data)){
        // 未登录情况跳转到登录页
        return self.redirect("/login")
    }else{
        self.userInfo = data;
        // 将用户信息赋值到模版变量里，供模版里使用
        self.assign("userInfo", data);
    }
})
```

用户登录成功写入 Session：

```
js // 用户登录成功写入 Session var name = this.post("name"); // 获取 post 过来的用户名 var pwd = this.post("pwd"); // 获取 post 过来的密码 var s
```

WebSocket

ThinkJS 可以无缝的支持 `websocket`，写 WebSocket 的逻辑和普通的 Http 请求的方式一致，底层使用了 websocket-driver 模块。

websocket 的功能默认是关闭的，使用 WebSocket 需要开启下面的配置：

```JavaScript
//WebSocket 配置
"use_websocket": false, // 是否使用 websocket
```

还有下面的配置可以设置：

```JavaScript
//WebSocket 配置
websocket_allow_origin: "", // 允许从那里发送过来的 websocket，可以是字符串、数组、回调函数，为空表示不检测。如："www.welefen.com"websocket_sub_protocal:"", //w
websocket_message_handle: undefined, //websocket 消息处理函数
```

数据格式

为了更好的规范浏览器端和服务端之间传送的数据格式，ThinkJS 默认使用 jsonrpc 2.0 的规范，也可以根据项目数据更改传输的数据格式。

浏览器传送到服务端的数据格式为：

```JavaScript
// 浏览器发送给服务端的数据格式
var data = {
    jsonrpc: "2.0",
    method: "/test/websocket/message",
    params: {userAgent: navigator.userAgent},
    id: 1
}
```

服务端发送给浏览器端的数据格式为：

```JavaScript
var data = {
    jsonrpc: "2.0",
    id: 1,
    result: {name: "welefen"}
}
```

处理逻辑

建立连接

浏览器端可以通过下面的方式创建一个 websocket 连接。

```JavaScript
//websocket 连接
var socket = new WebSocket("ws://" + location.hostname + ":1234");
socket.onopen = function(){}
```

这种方式只是创建了个 websocket 连接，不会调用控制器上任何操作。如果要调用控制器上对应的操作，可以用下面的方式：

```JavaScript
//websocket 连接
var socket = new WebSocket("ws://" + location.hostname + ":1234/websocket/open");
socket.onopen = function(){}
```

这里会根据路由解析 url `/websocket/open`，如：解析到的 group 为 home，controller 为 websocket, action 为 open。那么则会执行 `App/Lib/Controller/Home/WebsocketController.js` 下的 `openAction` 方法，并且传递进去的 http 对象上多了如下的属性：

```JavaScript
openAction: function(){
    var websocket = this.http.websocket; // 通过这个属性可以取到 WebSocket 对象，如：可以将这个对象存在一个对象池里，方面后续使用（比如：广播事件）
}
```

WebSocket 建立连接时还是 HTTP 协议，所以在 Action 里可以取到 cookie，也可以设置 cookie。

消息处理

建立连接后，浏览器和服务端就可以双向传输数据了。如：

```JavaScript
function getWebSocket(){var socket = new WebSocket("ws://" + location.hostname + ":1234/websocket/open");
    var deferred = $.Deferred();
    socket.onopen = function(event) {
        deferred.resolve(socket);
        socket.onmessage = function(event) {
            console.log(JSON.parse(event.data));
        };
        socket.onclose = function(event) {
            socket = null;
        };
    };
    return deferred;
}
getWebSocket().then(function(ws){
    // 建立连接后，给服务端发送 jsonrpc 2.0 格式的数据
    ws.send(JSON.stringify({
        jsonrpc: "2.0",
        method: "/websocket/message",
        params: {name: "welefen"},
        id: 1
    }))
});
```

这里传输的 method 为 `/websocket/message`，表示对应的 url 为 `/websocket/message`，假如根据路由解析后的分组为 home，控制器为 websocket，操作为 message，

那么则会执行 `App/Lib/Controller/WebscoketController.js` 下的 `messageAction` 方法。

`params` 参数值会作为请求参数传递进去，控制器里可以通过 `this.get("name")` 来获取对应的值。

处理传递请求参数，如果还想传递 `headers` 信息，那么 params 格式为：

```JavaScript
// params 值
params: {
    // 请求的 headers
    headers: {
        userAgent: "xxx",
        xxx: "yyy"
    },
    // 请求参数
    data: {name: "welefen"}
}
```

和 `openAction` 方法一样， `messageAction` 里也可以从 `http` 对象上获取 websocket 对象。

服务端可以通过 `echo` 方法向浏览器发送数据，如：

```JavaScript
//action 里发送数据到浏览器
messageAction: function(){
    var data = this.get(); // 获取所有传递过来的参数
    this.echo(data); // 输出数据到浏览器，框架会自动 JSON.stringify
}
```

可以通过 `end` 方法来关闭 websocket 连接。

websocket 关闭

如果浏览器端将 websocket 关闭了，服务端是可以捕获到这个关闭事件的。捕获需要在 `openAction` 里进行。

```JavaScript
openAction: function(){
    // 监听 websocket 关闭事件
    this.http.on("websocket.close", function(){
        //websocket 关闭后逻辑处理
    })
}
```

选择子协议

如果项目非常复杂，需要支持不同的数据格式，这时候可以使用子协议的功能。 如：

```JavaScript
//websocket 连接
var socket = new WebSocket("ws://" + location.hostname + ":1234/websocket/open", ["json", "soap"]);
socket.onopen = function(){}
```

这里表示浏览器端支持 json 和 soap 协议（如果是一个值可以是个字符串），服务端返回时需要选择一个协议告知浏览器。

服务端可以通过下面的配置来指定用哪个子协议，如：

```JavaScript
// 服务端子协议配置
"websocket_sub_protocal": "soap" // 这里表示服务端使用 soap 协议
```

也可以配置一个回调函数，会将浏览器支持的子协议列表作为参数传递进去。如：

```JavaScript
// 服务端子协议配置
"websocket_sub_protocal": function(protocals){
    return protocals[0]; // 选择第一个子协议
}
```

`注意：` 如果服务端返回的子协议不在浏览器传递过去的列表里，则会报错。

自定义数据格式

ThinkJS 默认使用 `jsonrpc 2.0` 的数据格式来传输，如果这种数据格式不能满足项目的需要，那么可以根据项目特点自定义数据格式。

自定义数据格式后，需要在项目里实现数据解析和发送逻辑。

数据解析与发送

自定义数据格式后，需要定义下面的配置来实现数据的解析和发送。如：

```JavaScript
"websocket_message_handle": function(data, connection, app, type){
    //data 为浏览器传递过来的数据
    //connection 为 websocket 的连接句柄
    //app 为系统的 App 对象
    //type 为数据格式，一种是字符串，一种是二进制数据
}
```

逻辑中必须实现如下的逻辑：

```JavaScript
var pars = {
    host: "", // 请求 host
    url: url, // 请求的 url，从 data 某个属性读取
    headers: headers, //headers，从 data 某个属性读取
    // 发送数据
    write: function(data, encoding, errMsg){
        connection.send(JSON.stringify(data));
    },
    // 关闭连接
    end: function(data){
        if (data) {this.write(data)}
        connection.close();
    }
}
// 下面几行代码可以直接拷贝，不用修改
var defaultHttp = thinkHttp.getDefaultHttp(pars);
httpInstance = thinkHttp(defaultHttp.req, defaultHttp.res);
// 将 websocket 实例添加到 http 对象上
httpInstance.http.websocket = connection.socket;
httpInstance.run(app.listener);
```

其中 pars 里的 `url`，`write`，`end` 必须要实现，否则会报错。这里使用 `write` 和 `end`，而不是 `send` 和 `close` 是为了和 HttpResponse 对象的方法名相同。

广播数据发送逻辑

广播数据发送是指在一个 websocket 请求里向其他所有或者部分的 websocket 发送数据，需要在 `openAction` 里定义 `websocket.send` 方法。如：

```JavaScript
openAction: function(){
    var websocket = this.http.websocket;
    websocket.send = function(data){
        // 调用 websocket.connection.send 方法直接发送
        websocket.connection.send(JSON.stringify(data));
    }
}
```

websocket id

为了后续处理方便，系统会在 websocket 对象上加上 id 属性，属性值是单调增的，保证每个 websocket 对象的 id 值都不一样。

```JavaScript
// 获取 websocket 的 id
openAction: function(){
    var id = this.http.websocket.id;
},
//message 里也能获取 websocket 的 id
messageAction: function(){
    var id = this.http.websocket.id;
}
```

超时处理

有时候有些 websocket 会一直连接，但没有任何数据交互（比如：一些用来攻击连接的 websocket）。如果不把这些 websocket 清理掉，那么占用的内存一直无法释放，同时对广播事件的性能也有影响。

ThinkJS 会每个 websocket 都添加了 `activeTime` 属性，这个属性值在每次有数据传输时都会更新。

有了这个时间点，那么就可以在控制器里里加上超时处理的逻辑了。比如：三十分钟清理一次

```JavaScript
var websocketList = {};
//30 分钟执行一次清理操作
setInterval(function(){var now = Date.now();
    for(var id in websocketList){
        var websocket = websocketList[id];
        if((now - websocket.activeTime) >= 30 * 60 * 1000){
            // 超时后关闭 websocket
            websocket.close();
            // 从列表里清除
            delete websocketList[id];
        }
    }
}, 30 * 60 * 1000);

module.exports = Controller(function(){
    openAction: function(){
        var websocket = this.http.websocket;
        // 将当前的 websocket 加到列表里
        websocketList[websocket.id] = websocket;
    }
})
```

注意： 如果 Controller 中用类似于 `websocketList` 记录了所有的 websocket，然后对 websocket 广播事件，那么开发的时候需要将 APP_DEBUG 设置为 false，不然每次清除缓存的时候都将 `websocketList` 里的 websocket 清除了。

变量过滤器

在项目中，访问有些页面时需要带上各种各样的参数。如：显示某个数据列表需要带上分页的参数，这时候需要在控制器里获取这个分页参数值，如果不为数值或者不存在的话，就设置个默认值。

```JavaScript
listAction: function(){
    // 获取当前分页数，并转化为数字
    var page = this.get("page") | 0;
    // 默认在第一页
    page = page || 1;
}
```

项目中有很多类似分页这样的参数，如果每个都在控制器里写这样的逻辑就非常麻烦。

ThinkJS 里提供了变量过滤器的功能，通过下面的方式加载 Filter。

```JavaScript
// 加载变量过滤器
var filter = thinkRequire("Filter").filter;
// 通过过滤器来转化分页的值
var page = filter(this.get("page"), "page");
```

支持格式

- `page` 过滤分页值，默认为 1。ThinkJS 建议分页值从 1 开始
- `order` 数据库排序方式。如： `"id DESC"` , `"id DESC,name ASC"`
- `id` 转化为数字，默认为 0
- `ids` 将逗号分割的多个 id 值转化为数组。如： `"1,2,3" => [1, 2, 3]`
- `in` 是否在一个范围中，如果不在，返回空字符串
- `strs` 将逗号格式的多个字符串转化为数组，如： `"xx,yy" => ["xx", "yy"]`

自动转化

如果在某个操作里调用变量过滤器来过滤也很麻烦，这时候就可以使用 ThinkJS 里的行为功能。

修改 `App/Conf/tag.js` 文件，添加如下的代码：

```javascript
// 自定义个行为
var filter = thinkRequire("Filter").filter;
module.exports = {action_init: [function(http){
    var types = ["id", "ids", "page", "order"];
        // 对 get 请求参数进行变量过滤器
        for(key in http.get){
            // 使用对应的类型转化
            if(types.indexOf(key) > -1){
                http.get[key] = filter(http.get[key], key);
            }else if(/_id$/.test(key)){ // 如果参数名是_id 结尾的，使用 `id` 转化
                http.get[key] = filter(http.get([key], "id"));
            }
        }
    }]
}
```

当然，你可以根据项目里的定义来修改这里的过滤逻辑。 这里统一过滤后，在 Action 里拿到的参数值就是统一过来后的了。

```javascript
listAction: function(){
    // 这里拿到的 page 值是个数值，且默认值为 1
    var page = this.get("page");
}
```

数据校验

数据校验是指对表单提交类的数据进行校验。如：用户注册、下订单等。 需要对这类提交的数据进行校验，如果不合法则不能通过。

数据校验和变量过滤器的区别为：变量过滤器一般给 `get` 请求使用的，数据校验一般给 `post` 请求使用的。

ThinkJS 里提供了数据校验的功能，在 Action 里通过 `this.valid` 方法使用。如：

```javascript
var email = this.post("email");
// 检测 email 是否合法
// 检测单个值时返回是否合法
var isValid = this.valid(email, "email");
```

除了检测单个值，也可以同时检测多个值，返回所有的错误信息。如：

```javascript
// 检测多个值，返回全部的错误信息
var errMsg = this.valid([{
    name: "email",
    value: email,
    valid: "email",
    msg: {email: "email 不合法"}
}, {
    name: "pwd",
    value: password,
    valid: ["length"],
    length_args: [6, 20], // 密码长度限制 6-20 位
    msg: {length: "密码长度不合法"}
}])
```

返回的错误信息为：

```javascript
// 检测的错误信息
var errMsg = {
    email: "email 不合法",
    pwd: "密码长度不合法"
}
```

检测类型

支持的检测类型有：

- `length` 限制长度，需要传入限制长度的数值。如： `length_args: [6]` 长度不能小于 6, `length_args: [6, 20]` 长度为 6-20。
- `required` 长度必须大于 0
- `regexp` 自定义正则检测。如：regexp_args: [/\w{5}/]
- `email` 邮箱
- `time` 时间戳

- `cnname` 中文
- `idnumber` 身份证号码
- `mobile` 手机号
- `zipcode` 邮编
- `confirm` 2 次值是否一致
- `url` url
- `int` 整数
- `float` 浮点数
- `range` 整数范围。如：`range_args: [100, 200]` 100-200 之间
- `ip4` ip4
- `ip6` ip6
- `ip` ip
- `date` 日期

命令行模式

ThinkJS 无缝支持命令行模式的调用，控制器的逻辑可以和普通 HTTP 请求的逻辑完全一致，可以做到同一个接口即可以 HTTP 访问，又可以命令行调用。

比如要执行 IndexController 里的 indexAction，可以使用如下的命令：

```Bash
node index.js /index/index
```

如果需要带上参数，可以直接在后面加上对应的参数即可，如：

```Bash
node index.js /index/index?name=welefen
```

也可以是：

```Bash
node index.js /index/index/name/welefen
```

修改请求方法

命令行执行默认的请求类型是 GET，如果想改为其他的类型，可以用下面的方法：

```Bash
node index.js url=/index/index&method=post
```

这样就把请求类型改为了 post。但这种方式下，参数 url 的值里就不能包含 & 字符了（可以通过上面 / 的方式指定参数）。

除了修改请求类型，还可以修改下面的参数：

- `host` 修改请求的 host 默认为 127.0.0.1
- `ip` 修改请求的 ip 默认为 127.0.0.1

修改更多的 headers

有时候如果想修改更多的 headers，可以传一个完整的 json 数据，如：

```Bash
node index.js {"url":"/index/index","ip":"127.0.0.1","method":"POST","headers":{"xxx":"yyyy"}}
```

注：参数比如是一个合法的 json 数据结构。

超时机制

命令行模式下执行的操作一般都是比较耗时的，如果代码不够严谨，可能会导致进程一直没有结束的情况。ThinkJS 从 `1.1.7` 版本开始增加了超时的机制。可以通过下面的配置来指定超时的时间：

```JavaScript
cli_timeout: 60 // 超时时间，单位为秒
```

Restful

ThinkJS 可以非常方便的支持 Restful 类的接口，假如现在有个名为 `article` 的 Resource 想提供 Restful 接口，可以通过下面的方式进行。

关于 Restful 的介绍可以见这里 http://www.ruanyifeng.com/blog/2011/09/restful.html

配置路由

Restful 类的接口需要借助自定义路由来使用，可以在路由配置文件中 `Conf/route.js` 配置如下的规则：

```
module.exports = [
  [/(article)(?:\/(\d*))?/, 'restful']
]
```

这个规则表示：

- 将 `/article` 和 `/article/10` 类的接口标记为 Restful 接口
- 需要对资源 `article` 和 `id` 使用正则分组功能，方便后续能够取到对应的值

建立对应的资源数据表

在数据库中添加对应的资源数据表 `think_article`，其中 `think_` 为数据表前缀，需要改为实际项目中的数据表前缀。

添加相关的字段和添加数据。

访问 Restful 接口

有对应的资源数据表后，无需添加任何的 Controller 文件，即可通过下面的地址访问：

- `GET /article` 获取所有的 article 列表
- `GET /article/10` 获取 id 为 10 的 article 详细信息
- `POST /article` 添加一个 article
- `PUT /article/10` 更新 id 为 10 的 article 数据
- `DELETE /article/10` 删除 id 为 10 的 article 数据

定制 Restful 接口

上面的方式虽然无需写任何 Controller 文件即可访问 `article` 资源，但有时候我们希望加一些限制，如：有些字段不输出，权限控制等功能，那么可以通过自定义 Controller 来进行。

Controller 文件

创建 `App/Lib/Controller/Restful/ArticleController.js` 文件，并使用如下的内容：

```
module.exports = Controller('RestController', {})
```

- Restful 默认的分组是 `Restful`，可以通过配置 `C('restful_group')` 来修改
- Restful 类的 Controller 需要继承自 `RestController`

限制部分字段

如果有些字段不想输出，那么可以通过下面的方式来隐藏：

```
module.exports = Controller('RestController', {
  __before: function(){
    this.model.field('content', true);
  }
})
```

这里表示不输出 `content` 字段。

实际上这里的 `this.model` 即为 `D('article')`，这样不仅可以限制字段，也可以限制条数，还可以分页。

权限控制

如果需要进行权限校验，那么也可以在 `__before` 里进行，如：

```
module.exports = Controller('RestController', {
  __before: function(){
     if(!this.hasPermission()){
       return this.error('no permission');
     }
   }
})
```

接口对应

请求类型 `GET`，`POST`，`PUT`，`DELETE` 对应的 Action 为 `getAction`，`postAction`，`putAction`，`deleteAction`。如果还需要进行更强的定制，可以重写这几个方法。

如 `getAction` 的实现为：

```JavaScript
getAction: function(){
  var self = this;
  if (this.id) {
    return getPromise(this.model.getPk()).then(function(pk){
      return self.model.where(getObject(pk, self.id)).find();
    }).then(function(data){
      return self.success(data);
    }).catch(function(err){
      return self.error(err.message);
    })
  }
  return this.model.select().then(function(data){
    return self.success(data);
  }).catch(function(err){
    return self.error(err.message);
  });
},
```

http 对象

这里讲的 http 对象并不是 Node.js 里的 http 模块，而是 ThinkJS 里将 http 请求的 Request 和 Response 2 部分包装在一起的一个对象。

由于 Node.js 是启服务的方式运行，所以处理用户请求时必须将当前请求的 Request 和 Response 对象向后续的处理逻辑里传递，比如：express 里有 req, res 对象。ThinkJS 里为了方便处理，将 Request 和 Response 包装成了一个 http 对象。

传递路径

http 对象会在下面的功能模块会中传递：

- `Behavior` 行为类
- `Controller` 控制器类

控制器类会在 init 方法里将传递过来的 http 对象赋值给 this.http。

```JavaScript
module.exports = Class({
    init: function(http){
        this.http = http;
    }
})
```

项目中的控制器类会继承控制器基类，如果要重写 init 方法，那么必须调用控制器基类的 init 方法，并将 http 对象传递过去。如：

```JavaScript
module.exports = Controller({
    init: function(http){
        this.super("init", http);
        // 其他逻辑
    }
})
```

处理的数据

http 对象里包含了很多处理用户请求的数据，如：cookie 数据，get 参数，post 内容，上传的文件等等。

cookie 数据

解析的 cookie 数据存放在 `http.cookie` 对象里，在 `Controller` 里直接通过 `cookie` 方法获取即可。

get 参数

解析的 get 参数存放在 `http.get` 对象里，在 `Controller` 里直接通过 `get` 方法获取即可。

post 内容

post 内容在不同的场景下可能有不同的数据格式，ThinkJS 提供多种的解析方式。

querystring 解析

ThinkJS 默认使用 querystring 的方式来解析 post 的内容，如：

```
name=welefen&value=111
```

解析后的 post 数据为：

```
{
    "name": "welefen",
    "value": "111"
}
```

json 解析

对于复杂的数据，querystring 解析就不合适了。这时候浏览器端可以传递一个 json 的数据格式，服务端也用 json 的方式来解析。

ThinkJS 支持在发送数据的时候指定特定的 `Content-Type` 来使用 json 解析，默认的 `Content-Type` 为 `application/json` 。

可以通过下面的参数来修改：

```
post_json_content_type: ['application/json'], //post 数据为 json 时的 content-type
```

配置值为数组，这样就可以指定多个 Content-Type 来使用 json 解析。

自定义方式解析

ThinkJS 除了支持使用 querystring 和 json 的方式来解析 post 数据外，还可以使用自定义的方式解析。ThinkJS 是通过行为切面来完成这一功能的，具体的行为切面名称为 `form_parse` 。

可以在项目的 App/Conf/tag.js 里指定行为切面 `form_parse` 对应的行为进行解析的工作。如：

```
// 解析 xml 格式的数据
var xmlParse = function(http){
    var postData = http.payload; //post 数据在 http.payload 里
    // 解析 xml 格式数据的逻辑，并将解析的结果返回
    // 返回的可以是个 promise
}
module.exports = {
    form_parse: [xmlParse]
}
```

上传的文件

ThinkJS 支持表单文件上传和 ajax 文件上传 2 种方式，解析后的数据放在 `http.file` 对象里， `Controller` 里直接使用 file 方法获取即可。

表单文件上传

表单文件上传可以指定如下的配置参数：
`js post_max_file_size: 1024 * 1024 * 1024, // 上传文件大小限制，默认 1G post_max_fields: 100, // 最大表单数，默认为 100 post_max_fields_size: 2`

ajax 文件上传

高级浏览器下支持使用 ajax 来上传单个文件，如：
`js var xhr = new XMLHttpRequest(); xhr.onreadystatechange = function(e){ } xhr.open("POST", '/admin/project/upload', true); xhr.setReque`
对于 ajax 上传的内容，ThinkJS 是通过下面的配置来判断上传的是否是文件：

```
post_ajax_filename_header: 'x-filename', // 通过 ajax 上传文件时文件名对应的 header，如果有这个 header 表示是文件上传
```

该头信息的值作为具体的文件名来获取。

通过 ajax 上传的文件 filedName 固定为 file，在 Controller 中可以通过 `this.file("file")` 来获取。

获取到单个文件的信息如下：

```JavaScript
{
    fieldName: 'file',
    originalFilename: filename, // 原始文件文件名
    path: filepath, // 文件存放的临时目录
    size: fs.statSync(filepath).size // 文件大小
}
```

其他

原始的 Request 和 Response 对象

http 是一个包装的对象，但保留了原始的 Request 和 Response 对象，可以通过 `http.req` 和 `http.res` 来获取。

比如：页面的 url 可以通过 `http.req.url` 来获取，对应在 Action 里就是 `this.http.req.url`

反向代理

Node.js 是通过监听端口启动服务来运行的，访问的时候需要带上端口，除非端口是 80。但一般情况下，是不会让 Node.js 占用 80 端口的。并且 Node.js 在处理静态资源情况时并没有太大的优势。

所以一般的处理方式是通过 Nginx 之类的 web server 来处理，静态资源请求直接让 Nginx 来处理，动态类的请求通过反向代理让 Node.js 来处理。这样也方便做负载均衡。

配置

nginx 下的配置可以参考下面的方式：

```
server {
    listen       80;
    server_name meinv.ueapp.com;
    index index.js index.html index.htm;
    root  /Users/welefen/Develop/git/meinv.ueapp.com/www;

    if (-f $request_filename/index.html){rewrite (.*) $1/index.html break;
    }
    if (!-f $request_filename){rewrite (.*) /index.js;
    }
    location = /index.js {
        #proxy_http_version 1.1;
        proxy_set_header Connection "";
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_set_header X-NginX-Proxy true;
        proxy_pass http://127.0.0.1:6666$request_uri;
        proxy_redirect off;
    }
    location ~ .*\.(js|css|gif|jpg|jpeg|png|bmp|swf|ico|svg|cur|ttf|woff)$ {expires      1000d;}
}
```

需要改动 3 个地方：

- `server_name meinv.ueapp.com` 将 server_name 改为项目对应的域名
- `root /Users/welefen/Develop/git/meinv.ueapp.com/www` 配置项目的根目录，一定要到 www 目录下
- `proxy_pass http://127.0.0.1:6666$request_uri;` 将端口 `6666` 改为项目里配置的端口

禁止端口访问

配置反向代理后，我们可能就不希望通过端口能直接访问到 Node.js 服务了。那么可以在 `App/Conf/config.js` 设置如下的配置：

```JavaScript
use_proxy: true, // 是否使用代理访问，如：nginx。开启后不能通过 ip+ 端口直接访问
```

其他

调试

开启调试，需要将 `www/index.js` 里的 `global.APP_DEBUG` 设置为 true。

创建项目时，该配置项默认开启。

特性

开启调试后，会有如下的特性：

- 修改项目下的任意文件，保存后立即生效，不需要重启 node 服务
- 每执行一条 sql 操作，控制台下都会将 sql 语句打印处理
- 如有报错，控制台下可以看到详细的错误信息

关闭配置

开启调试后，会关闭如下的配置：

```JavaScript
// 调试下关闭的配置
"db_fields_cache": false, //debug 模式下关闭数据表字段的缓存
"db_cache_on": false,
"use_cluster": false,
"html_cache_on": false,
"log_process_pid": false,
"clear_require_cache": true, // 清除 require 的缓存文件
```

日志

ThinkJS 支持 console 和 memory 相关的日志，具体配置如下：

```JavaScript
log_console: false, // 是否记录日志，开启后会重写 console.error 等系列方法
log_console_path: LOG_PATH + '/console', // 日志文件存放路径
log_console_type: ['error'], // 默认只接管 console.error 日志
log_memory: false, // 记录内存使用和负载
log_memory_path: LOG_PATH + '/memory', // 日志文件存放路径
log_memory_interval: 60 * 1000, // 一分钟记录一次
```

开启了日志功能就可以在 App/Runtime/Log 下查看对应的日志了，如：`App/Runtime/Log/memory/2014-09-23.log`

```
[2014-09-23 15:01:35] rss:23.0MB heapTotal:9.8MB heapUsed:5.3MB freeMemory:677.6MB loadAvg:1.3,1.7,1.81
[2014-09-23 15:02:35] rss:23.0MB heapTotal:9.8MB heapUsed:5.3MB freeMemory:677.6MB loadAvg:1.3,1.7,1.81
[2014-09-23 15:03:35] rss:22.6MB heapTotal:9.8MB heapUsed:4.9MB freeMemory:677.6MB loadAvg:1.3,1.7,1.81
[2014-09-23 15:04:35] rss:22.9MB heapTotal:9.8MB heapUsed:5.2MB freeMemory:677.6MB loadAvg:1.3,1.7,1.81
```

- 日志按天分割文件
- `APP_DEBUG=true` 下不会记录日志

线上部署

你可以使用自己的工具将代码发布到线上服务器。

关闭 APP_DEBUG

ThinkJS 通过设置 `APP_DEBUG=true` 的方式达到修改文件立即生效的目的，这种方式是通过定时清除文件缓存的方式来达到的，所以长期开启后会有一定的内存泄漏。

服务在线上运行时，切记要将 `www/index.js` 里的 `APP_DEBUG` 设置为 `false`。

启动服务

线上服务建议使用 `pm2` 模块来管理。如：

```JavaScript
// 通过 pm2 来启动服务
pm2 start /home/welefen/www/www.welefen.com/www/index.js -n www.welefen.com
```

然后通过 `pm2 ls` 命令可以看到启动的服务：

```
┌──────────────────┬────┬──────┬──────┬────────┬─────────┬────────┬─────────────┬──────────┐
│ App name         │ id │ mode │ PID  │ status │ restarted │ uptime │      memory │    watching │
├──────────────────┼────┼──────┼──────┼────────┼─────────┼────────┼─────────────┼──────────┤
│ www.welefen.com  │ 6  │ fork │ 3319 │ online │         0 │ 2D     │  8.715 MB   │ unactivated │
│ www.ThinkJS.org  │ 7  │ fork │ 3332 │ online │         0 │ 2D     │ 43.773 MB   │ unactivated │
└──────────────────┴────┴──────┴──────┴────────┴─────────┴────────┴─────────────┴──────────┘
 Use `pm2 desc[ribe] <id>` to get more details
```

指导规范

全局函数

全局函数都定义在 `App/Common/common.js` 文件中，函数名为驼峰式命名。如：

```JavaScript
// 全局函数定义
global.getPicModel = function(groupId){
    var model = D('Model');
    //extra code
}
```

这样函数 `getPicModel` 在控制器里就可以直接使用了。

类文件

所有的类文件都通过函数 `Class` 来创建，没有特殊情况，直接赋值给 module.exports。如：

```JavaScript
//require 模块放在 module.exports 前面
var marked = require("markded");
var toc = require("marked-toc");

module.exports = Class(function(){
    // 类里面用到的一些变量放在这里，最好不要放在 Class 之外
    var keyList = { }
    return {
        init: function(){

        }
    }
});
```

如果创建的类还有一些属性或者方法，那么可以重新定义一个变量，如：

```JavaScript
var App = module.exports = Class(function(){...})
//listener 方法
App.listener = function(){}
```

ThinkJS 里包装了很多功能的基类，如：`Model`，`Db`，`Controller`，需要开发这些功能时，可以直接继承这些基类。

异步

ThinkJS 是基于 es6-promise 来实现的，大大简化了异步回调的代码逻辑。如果你的项目比较复杂，需要开发一些独立的模块，建议也使用 promise 的方式。

ThinkJS 提供了 `getDefer()` 获取 deferred 对象，`getPromise()` 获取 promise 对象。参见如下示例：

```JavaScript
// 获取页面内容
function getPageContent(){
    var deferred = getDefer();
    request.get('http://www.ThinkJS.org', function(err, response, body){
        if(err || response.statusCode !== 200){
            deferred.reject(err || new Error('statusCode:' + response.statusCode))
        }else{
            deferred.resolve(body);
        }
    })
    return deferred.promise
}
```

最佳实践

控制器基类

同一个分组下的 `Controller` 一般会有一些共同的特性，那么就可以把这些共同的特性放在一个控制器基类里，其他的控制器继承该控制器。

建议控制器基类名为 `BaseController`，如：

```JavaScript
// App/Lib/Controller/Home/BaseController.js
module.exports = Controller(function(){
    return {
        init: function(http){
            this.super("init", http);
            // 给模版里设置 title 等一些字段
            this.assign({
                title: "ThinkJS 官网",
                navType: "home"
            })
        },
        // 获取页面顶部的分类，几乎每个页面都会使用
        // 那么可以放在基类里，供子类调用
        getCate: function(){
            var self = this;
            return D('Cate').select().then(function(data){
                self.assign("cateList", data);
            })
        }
    }
})
```

基类可以通过 `BaseController` 来继承，如：

```JavaScript
// App/Lib/Controller/Home/IndexController.js
module.exports = Controller("Home/BaseController", function(){
    return {
        indexAction: function(){
            var self = this;
            // 获取分类列表
            var catePromise = this.getCate();
            // 获取文章列表
            var articlePromise = D('Article').page(this.page("page")).select().then(function(data){
                self.assign("articleList", data);
            })
            // 分类和文章列表数据都 OK 后渲染模版
            return Promise.all([catePromise, articlePromise]).then(function(){
                return self.display();
            })
        }
    }
})
```

用户登录后才能访问

在项目中，经常要判断当前访问的用户是否有权限，如果没有权限那么后续的代码就不能在执行。最基本的案例就是判断用户是否已经登录，比如：后台管理

```JavaScript
// App/Lib/Controller/Admin/BaseController.js
module.exprots = Controller(function(){
    return {
        //__before 会在执行具体的 action 之前执行
        __before: function(){
            // 登录页面不检测用户是否已经登录
            if(this.http.action === 'login'){return;}
            return this.session("userInfo").then(function(userInfo){
                // 用户信息为空
                if(isEmpty(userInfo)){
                    //ajax 访问返回一个 json 的错误信息
                    if(self.isAjax()){
                        return self.error(403, "用户未登录，不能访问")
                    }else{
                        // 跳转到登录页
                        return self.redirect("/index/login");
                    }
                }else{
                    // 用户已经登录
                    self.userInfo = userInfo;
                    self.assign("userInfo", userInfo);
                }
            })
        }
    }
})
```

由于 `error` 和 `redirect` 方法返回的都是 `pendding promise`，如果未登录的话，可以阻止后续的代码继续执行。

```JavaScript
indexAction: function(){
    //init 方法里将用户信息赋值到 this.userInfo 上，那么这里就可以直接获取了
    var userInfo = this.userInfo;
    var id = userInfo.id;
}
```

更多功能

自定义 http 服务

ThinkJS 默认会自动创建 `http` 服务，如果默认创建的服务无法满足需求时，可以通过下面的方式自定义创建 `http` 服务。

1) ** 开启自定义服务配置 **

在 `App/Conf/config.js` 文件中添加下面的配置：

```JavaScript
create_server_fn: 'create_server_fn_name',
```

其中字符串 `create_server_fn_name` 是函数名，可以根据需求修改。

2) ** 自定义创建服务实现 **

在 `App/Common/common.js` 文件中添加如下的代码：

```JavaScript
// 函数名 create_server_fn_name 需要跟配置 create_server_fn 值一致
global.create_server_fn_name = function(App){
    var server = require('http').createServer(function (req, res) {
        thinkRequire('Http')(req, res).run().then(App.listener);
      });
      server.listen(C('port'));
      // 这里可以加入项目的扩展代码
}
```

`注意：` 修改完成后，重启 Node.js 服务才能生效。

使用 cluster

如果项目想使用 cluster 的话很简单，只要在 `App/Conf/config.js` 文件中添加如下配置即可：

```
// 值为 true 表示实现当前 cpu 的个数，可以根据需要更改为实际的数字
use_cluster: true
```

禁止通过端口直接访问

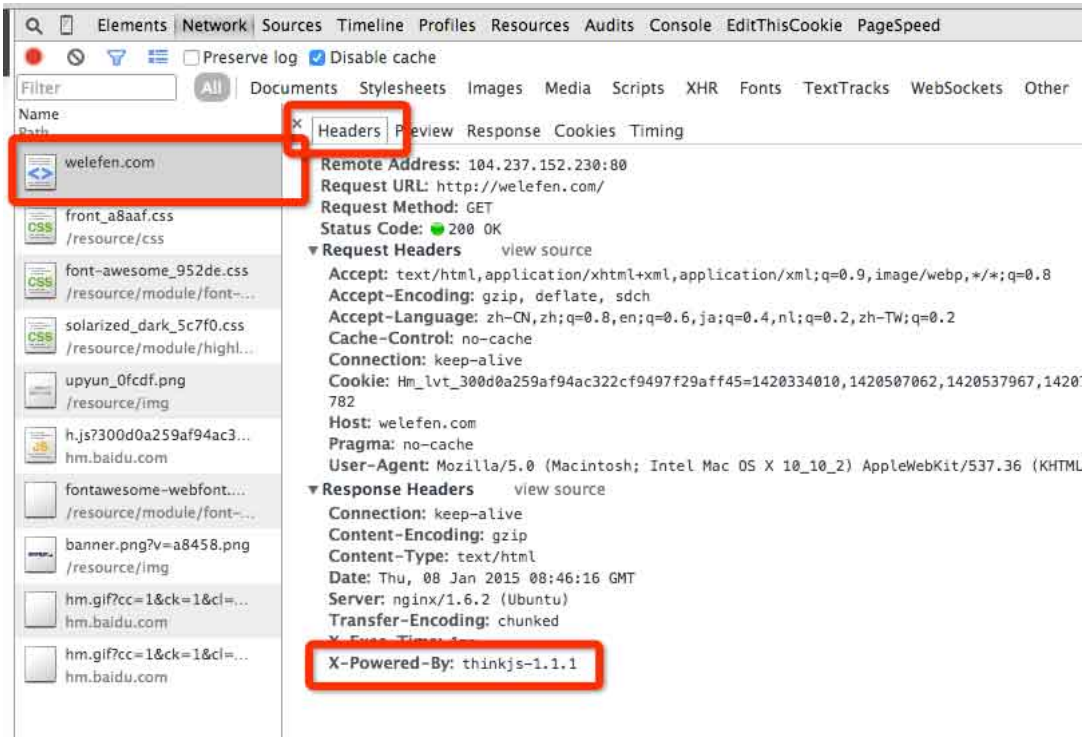
Node.js 启动的服务默认可以通过 `ip+ 端口` 直接访问。但项目上线后，一般是通过 nginx 做一层反向代理，这样可以很好的做负载均衡，这个时候我们就不希望可以通过 `ip+ 端口` 能够直接访问到 Node.js 的服务了。

ThinkJS 下可以在 `App/Conf/config.js` 里添加如下的配置禁止通过端口直接访问：

```
use_proxy: true
```

查看当前项目运行的 ThinkJS 版本

可以通过下面的方式查看当前项目运行的 ThinkJS 版本。



附录

默认配置

JavaScript

```
/**
* 框架默认配置
* 可以在 App/Conf/config.js 里修改下面的配置值
* @type {Object}
*/
module.exports = {
  port: 8360, // 监听端口
  host: '', // 监听的 host
  use_proxy: false, // 是否使用代理访问，如：nginx。开启后不能通过 ip+ 端口直接访问
  encoding: 'utf-8', // 输出数据的编码
  url_pathname_prefix: '',  // 不解析的 pathname 前缀
  url_pathname_suffix: '.html', // 不解析的 pathname 后缀，这样利于 seo
  app_tag_on: true, // 是否支持标签功能
  url_resource_on: true,  // 是否监听静态资源类请求
  url_resource_reg: /^(resource\/|static\/|favicon\.ico|robots\.txt)/, // 判断是否是静态资源的正则
  url_route_on: true, // 是否开启自定义路由功能
  filter_data: true, // 主要是安全过滤，强烈建议开启

  post_json_content_type: ['application/json'], //post 数据为 json 时的 content-type
  post_max_file_size: 1024 * 1024 * 1024, // 上传文件大小限制，默认 1G
  post_max_fields: 100, // 最大表单数，默认为 100
  post_max_fields_size: 2 * 1024 * 1024, // 单个表单长度最大值，默认为 2MB
  post_ajax_filename_header: 'x-filename', // 通过 ajax 上传文件时文件名对应的 header，如果有这个 header 表示是文件上传
  post_file_upload_path: APP_PATH + '/Runtime/Temp', // 文件上传的临时目录

  app_group_list: ['Home', 'Admin', 'Restful'], // 分组列表
  deny_group_list: [],
  default_group: 'Home', // 默认分组
  default_controller: 'Index', // 默认模块
  default_action: 'index',  // 默认 Action
  call_controller: 'Home:Index:_404', //controller 不存在时执行方法，此配置表示调用 Home 分组下 IndexController 的_404Action 方法
  call_method: '__call', // 当找不到方法时调用什么方法，这个方法存在时才有效
  before_action: '__before', // 调用一个 action 前调用的方法，会将 action 名传递进去
  after_action: '__after', // 调用一个 action 之后调用的方法，会将 action 名传递进去
  url_params_bind: true, // 方法参数绑定，将 URL 参数值绑定到 action 的参数上
  action_suffix: 'Action', //action 后缀
  url_callback_name: 'callback', //jsonp 格式的 callback 名字
  json_content_type: 'application/json', // 发送 json 时的 content-type
  auto_send_content_type: true, // 是否自动发送 Content-Type，默认值为 `tpl_content_type` 配置值
  log_process_pid: true, // 记录进程的 id，方便其他脚本处理。
  use_cluster: false, // 是否使用 cluster，默认不使用，0：为 cpu 的数量，可以自定义值
```

```
autoload_path: {}, //autoload 查找的 path，用于 thinkRequire 加载自定义库的时候查找
create_server_fn: '', // 自定义 create server 全局函数名，可以在 Common/common.js 里实现

restful_group: 'Restful', //RESTFUL API 默认分组

load_ext_config: [], // 加载额外的配置文件 CONF_PATH
load_ext_file: [], // 加载额外的文件 COMMON_PATH

use_websocket: false, // 是否使用 websocket
websocket_allow_origin: '', // 允许从那里发送过来的 websocket，可以是字符串、数组、回调函数，为空表示不检测
websocket_sub_protocal: '', //websocket 子协议，可以是个字符串也可以是回调函数
websocket_message_handle: undefined, //websocket 消息处理函数

error_tpl_path: THINK_PATH + '/View/error.html', // 错误页模版
error_code: 500, // 报错时的状态码
error_no_key: 'errno', // 错误号的 key
error_no_default_value: 1000, // 错误号默认值
error_msg_key: 'errmsg', // 错误消息的 key

cookie_domain: '', //cookie 有效域名
cookie_path: '/',   //cookie 路径
cookie_timeout: 0, //cookie 失效时间，0 为浏览器关闭，单位：秒

session_name: 'ThinkJS', //session 对应的 cookie 名称
session_type: 'File', //session 存储类型，空为内存，还可以为 File
session_path: '', //File 类型下文件存储位置，默认为系统的 tmp 目录
session_options: {}, //session 对应的 cookie 选项
session_sign: '', //session 对应的 cookie 使用签名
session_timeout: 24 * 3600, // 服务器上 session 失效时间，单位：秒

db_type: 'mysql', // 数据库类型
db_host: '127.0.0.1', // 服务器地址
db_port: '', // 端口
db_name: '', // 数据库名
db_user: 'root', // 用户名
db_pwd: '', // 密码
db_prefix: 'think_', // 数据库表前缀
db_charset: 'utf8', // 数据库编码默认采用 utf8
db_ext_config: {}, // 数据库连接时候额外的参数
db_fieldtype_check: false, // 是否进行字段类型检查
db_fields_cache: true, // 启用字段缓存
db_nums_per_page: 20, // 默认每页显示的条数
db_like_fields: [], // 自动进行模糊查询,| 连接，如：['title', 'content']
db_cache_on: true, // 是否启用查询缓存，如果关闭那么 cache 方法则无效
db_cache_type: '', // 缓存类型，默认为内存缓存
db_cache_path: CACHE_PATH + '/db', // 缓存路径，File 类型下有效
db_cache_timeout: 3600, // 缓存时间，默认为 1 个小时
db_log_sql: false, // 是否打印 sql 语句
db_buffer_tostring: true, // 是否将 buffer 转为字符串

tpl_content_type: 'text/html', // 模版输出类型
tpl_file_suffix: '.html', // 模版文件名后缀
tpl_file_depr: '_', //controller 和 action 之间的分隔符
tpl_engine_type: 'ejs', // 模版引擎名称
tpl_engine_config: {},

token_on: false, // 是否开启 token 功能
token_name: 'token', //token name
token_key: '{__TOKEN__}', // 记录 token 在模版中的位置替换用。默认自动查找 <form 和 </head> 标签替换

log_console: false, // 是否记录日志，开启后会重写 console.error 等系列方法
log_console_path: LOG_PATH + '/console', // 日志文件存放路径
log_console_type: ['error'], // 默认只接管 console.error 日志
log_memory: false, // 记录内存使用和负载
log_memory_path: LOG_PATH + '/memory', // 日志文件存放路径
log_memory_interval: 60 * 1000, // 一分钟记录一次

cache_type: 'File', // 数据缓存类型
cache_key_prefix: '__ThinkJS__', // 缓存 key 前置 (memcache 和 redis 下有效)
cache_timeout: 6 * 3600, // 数据缓存有效期，单位：秒
cache_path: CACHE_PATH,  // 缓存路径设置 (File 缓存方式有效)
cache_file_suffix: '.json', //File 缓存方式下文件后缀名
cache_gc_hour: [4], // 缓存清除的时间点，数据为小时

html_cache_on: false, //HTML 静态缓存
html_cache_timeout: 3600, // 缓存时间，单位为秒
html_cache_rules: {}, // 缓存规则
```

```
  html_cache_path: CACHE_PATH + '/html',
  html_cache_file_callback: undefined, // 生成缓存文件的回调函数
  html_cache_file_suffix: '.html', // 缓存文件后缀名

  memcache_host: '127.0.0.1', //memcache host
  memcache_port: 11211, //memecache 端口

  redis_host: '127.0.0.1', //redis host
  redis_port: 6379, // redis port
};
```

这些配置的值都可以在 `App/Conf/config.js` 文件里重新设置。

系统常量

系统里定义很多系统常量，方便在项目中使用。

- `APP_DEBUG` 开启调试，开发中使用，上线后需要关闭
- `APP_MODE` 运行模式
- `THINK_PATH` ThinkJS 库的目录
- `THINK_VERSION` 当前 ThinkJS 的版本
- `THINK_LIB_PATH` ThinkJS 的 lib 路径
- `THINK_EXTEND_PATH` ThinkJS 的扩展路径
- `APP_PATH` 项目 App 路径，如: `/home/welefen/www.test.com/App`
- `COMMON_PATH` Common 路径，对应为 `App/Common`
- `LIB_PATH` Lib 路径，对应为 `App/Lib`
- `CONF_PATH` 配置路径，对应为 `App/Conf`
- `VIEW_PATH` 模版路径，对应为 `App/View`
- `RUNTIME_PATH` runtime 路径，对应为 `App/Runtime`
- `DATA_PATH` 临时数据路径，对应为 `App/Runtime/Data`
- `CACHE_PATH` 缓存路径，对应为 `App/Runtime/Cache`
- `RESOURCE_PATH` 资源路径，对应为 `/www`

API

原生对象的扩展

Object.values(obj)

- obj `Object`
- return `Array`

返回一个对象的 value 列表。

```
js var obj = {name: "welefen", age: "29", sex: "male"}; var values = Object.values(obj); /* values is ["welefen", "29", "male"] */
```

全局函数

该页面列举的函数都是全局函数，不用 require 可以直接使用。

Promise

ThinkJS 中的 Promise 使用了 es6-promise 库，是个全局对象， 含有如下的方法：

- `all(array)`
- `resolve(promise | thenable | obj)`
- `reject(obj)`
- `race(array)`

Class(superCls, prop)

- superCls `function` 父类
- prop `function | object` 如果是 function，则执行这个 function，并获取结果
- return `function`

通过该函数动态创建一个类，可以实现类继承和自动调用 init 方法的功能，同时实例化类的时候可以省去 `new` 。如果只传了一个参数，则认为是 prop。

```JavaScript
//A 为通过 Class 动态创建的一个类
var A = Class(function(){
    return {
        init: function(name){
            this.name = name;
        },
        getName: function(){
            return "A" + this.name;
        }
    }
});
// 实例化类 A，可以不写 new
var instance = A("welefen");
var name = instance.getName(); /*name is `A welefen`*/
```

通过 Class 函数创建的类支持继承，含有以下 2 个静态方法：

- `extend(obj)` 扩展方法到类的原型上
- `inherits(superCls)` 指定该类的父类

子类可以继承父类的方法，同时可以对方法进行重写。

```JavaScript
var B = Class(A, {}); //B 类从 A 类继承而来
//B 类的实例化
var instance = B("welefen");
var name = instance.getName(); /*name is `A welefen`*/
B.extend({
    getName: function(){ //B 类对 getName 方法进行了重写
        return "B" + this.name;
    }
});
var name = instance.getName(); /*name is `B welefen`*/
```

也可以在重写的方法里调用父类的方法，如：

```JavaScript
var C = Class(A, {
    getName: function(){
        var name = this.super("getName");
        return "C" + name;
    }
}); // 从 A 类继承
var instance = C("welefen");
var name = instance.getName(); /*name is `C A welefen`*/
```

如果有多级继承，想跨级调用父类的方法时，只能通过 apply 的方式调用原形链上的方法，如：

```JavaScript
var D = Class(C, {
    getName: function(){
        var name = A.prototype.getName.apply(this, arguments);
        return 'D' + name;
    }
}); // 从 C 类继承
var instance = D('welefen');
var name = instnace.getName(); /*name is `D A welefen`*/;
```

`注意：` 不可用下面的方式来继承

```JavaScript
var A = Class();
var B = Class({
    getName: function(){}
}).inherits(A); // 此时 B 不含有 getName 方法
```

extend(target, source1, source2, ...)

- target `object`
- source1 `object`
- return `object`

将 source1, source2 等对象上的属性或方法复制到 target 对象上，类似于 jQuery 里的 $.extend 方法。

默认为深度复制，可以将第一个参数传 `false` 进行浅度复制。

`注意`： 赋值时，忽略值为 undefined 的属性。

isBoolean(obj)

- `obj` 要检测的对象
- `return` true OR false

检测一个对象是否是布尔值。

```JavaScript
// 判断是否是布尔值
isBoolean(true); //true
isBoolean(false); //true
```

isNumber(obj)

检测一个对象是否是数字。

```JavaScript
isNumber(1); //true
isNumber(1.21); //true
```

isObject(obj)

检测是否是对象

```JavaScript
isObject({}); //true
isObject({name: "welefen"}); //true
```

isString(obj)

检测是否是字符串

```JavaScript
isString("xxx"); // true
isString(new String("xxx")); //true
```

isFunction(obj)

检测是否是函数

```JavaScript
isFunction(function(){}); //true
isFunction(new Function("")); //true
```

isDate(obj)

检测是否是日期对象

```JavaScript
isDate(new Date()); //true
```

isRegexp(obj)

检测是否是正则

```JavaScript
isRegexp(/\w+/); //true
isRegexp(new RegExp("/\\w+/")); //true
```

isError(obj)

检测是否是个错误

```JavaScript
isError(new Error("xxx")); //true
```

isEmpty(obj)

检测是否为空

```JavaScript
// 检测是否为空
isEmpty({}); //true
isEmpty([]); //true
isEmpty(""); //true
isEmpty(0); //true
isEmpty(null); //true
isEmpty(undefined); //true
isEmpty(false); //true
```

isArray(obj)

检测是否是数组

```JavaScript
isArray([]); //true
isArray([1, 2]); //true
isArray(new Array(10)); //true
```

isIP4(obj)

检测是否是 IP4

```JavaScript
isIP4("10.0.0.1"); //true
isIP4("192.168.1.1"); //true
```

isIP6(obj)

检测是否是 IP6

```JavaScript
isIP6("2031:0000:130f:0000:0000:09c0:876a:130b"); //true
isIP6("2031:0000:130f::09c0:876a:130b"); //true
```

isIP(obj)

检测是否是 IP

```JavaScript
isIP("10.0.0.1"); //true
isIP("192.168.1.1"); //true
isIP("2031:0000:130f:0000:0000:09c0:876a:130b"); //true ip6
```

isFile(file)

检测是否是文件，如果在不存在则返回 false

```JavaScript
isFile("/home/welefen/a.txt"); //true
isFile("/home/welefen/dirname"); //false
```

isDir(dir)

检测是否是目录，如果不存在则返回 false

```JavaScript
isDir("/home/welefen/dirname"); //true
```

isBuffer(buffer)

检测是否是 Buffer

```JavaScript
isBuffer(new Buffer(20)); //true
```

isNumberString(obj)

是否是字符串类型的数字

```JavaScript
isNumberString(1); //true
isNumberString("1"); //true
isNumberString("1.23"); //true
```

isPromise(promise)

检测是否是个 promise

```JavaScript
isPromise(new Promise(function(){})); //true
isPromise(getPromise()); //true
```

isWritable(p)

判断文件或者目录是否可写，如果不存在则返回 false

mkdir(p, mode)

递归的创建目录

- `p` 要创建的目录
- `mode` 权限，默认为 `0777`

```JavaScript
// 假设 /home/welefen/a/b/ 不存在
mkdir("/home/welefen/a/b");
mkdir("home/welefne/a/b/c/d/e"); // 递归创建子目录
```

chmod(p, mode)

修改目录权限，如果目录不存在则直接返回

```JavaScript
chmod("/home/welefen/a", 0777);
```

ucfirst(name)

将首字符变成大写，其他变成小写

```JavaScript
ucfirst("welefen"); // Welefen
ucfirst("WELEFEN"); // Welefen
```

md5(str)

获取字符串的 md5 值，如果传入的参数不是字符串，则自动转为字符串

```JavaScript
md5("welefen"); //59dff65d54a8fa28fe372b75d459e13b
```

getPromise(obj, reject)

获取一个 promise 对象。默认为 `resolve promise`，如果 reject 参数为 true，那么返回 `reject promise`。

如果 obj 是 promise，那么直接返回。

```JavaScript
getPromise([]); //resolve promise
getPromise(new Error(""), true); //reject promise
var promise = getPromise("");
getPromise(promise); //
```

getDefer()

获取一个 `Deferred` 对象，对象含有如下的属性或者方法：

- `resolve` 方法：将 promise resolve
- `reject` 方法：将 promise reject
- `promise` 属性：Deferred 对应的 Promise

```JavaScript
// 把读取文件内容变成 promise
var fs = require("fs");
function getFileContent(file){var deferred = getDefer();
    fs.readFile(file, "utf8", function(err, content){
        // 如果有错误，那么 reject
        if(err){deferred.reject(err);
        }else{
            // 成功读取到内容
            deferred.resolve(content);
        }
    })
    return deferred.promise;
};

getFileContent("/home/welefen/a.txt").then(function(content){//}).catch(function(err){console.log(err.stack);
})
```

`deferred.promise` 默认为 `pedding` 状态， `pedding` 状态的 promise 不会执行后续的 then，也不会执行 catch。如果想阻止后面的代码继续执行，那么可以返回一个 `pedding promise` 。

```JavaScript
// 返回一个 pedding promise
var getPeddingPromise = function(){var deferred = getDefer();
    return deferred.promise;
}
getPeddingPromise().then(function(){// 这里的代码不会执行}).catch(function(){// 这里的代码也不会执行})
```

getObject(name, value)

在项目中，经常会遇到要动态创建一个对象。如：

```JavaScript
var data = {};
//name 和 value 从其他地方动态读取出来的
data[name] = value;
// 有时候还要设置多个
data[name1] = value1;
```

为了方便创建对象，可以通过 `getObject` 来完成。

```JavaScript
// 单个属性
var data = getObject(name, value);
// 多个属性
var data = getObject([name, name1], [value, valu1]);
// 更多的属性
var data = getObject([name, name1, name2, name3], [value, value1, value2, value3]);
```

arrToObj(arr, key, valueKey)

在项目中，经常会从数据库中查询多条数据，然后对数据进行一些操作。如：根据特定的 key 进行去除等。我们一般借助对象来完成此类操作，这时候需要把数组转化为对象。

可以借助 arrToObj 来完成。

```JavaScript

// 从数据库中查询出来的数据对象
var arr = [{id: 10, name: "name1", value: "value1"}, {id: 11, name: "name2", value: "value2"}];
// 把 id 值作为 key 生成一个对象
/* data = {10: {id: 10, name: "name1", value: "value1"}, 11: {id: 11, name: "name2", value: "value2"}} */
var data = arrToObj(arr, "id");
// 把 id 值作为 key，只需要 name 的值
//data = {10: "name1", 11: "name2"}
var data = arrToObj(arr, "id", "name");
// 只获取 id 的值
// ids = [10, 11];
var ids = arrToObj(arr, "id", null);
```

Controller

此文档表示控制器下接口，在子控制器下可以直接使用。

如果想在 Controller 里的 Action 查看 http 相关功能，可以查看 这里。

ip()

获取当前请求的用户 ip。

```JavaScript
testAction: function(){
    // 用户 ip
    var ip = this.ip();
}
```

如果项目部署在本地的话，返回的 ip 为 `127.0.0.1` 。

isGet()

判断当前请求是否是 get 请求。

```JavaScript
testAction: function(){
    // 如果是 get 请求直接渲染模版
    if(this.isGet()){
        return this.display();
    }
}
```

isPost()

判断当前请求是否是 post 请求。

```JavaScript
testAction: function(){
    // 如果是 post 请求获取对应的数据
    if(this.isPost()){
        var data = this.post();
    }
}
```

isMethod(method)

判断是否是一个特定类型的请求。

```JavaScript
testAction: function(){
    // 判断是否是 put 请求
    var isPut = this.isMethod("put");
    // 判断是否是 delete 请求
    var isDel = this.isMethod("delete");
}
```

http 支持的请求类型为： `GET` , `POST` , `PUT` , `DELETE` , `HEAD` , `OPTIONS` , `PATCH` 。

isAjax(method)

判断是否是 ajax 类型的请求。

```JavaScript
testAction: function(){
    // 只判断是否是 ajax 请求
    var isAjax = this.isAjax();
    // 判断是否是 get 类型的 ajax 请求
    var isGetAjax = this.isAjax("get");
    // 判断是否是 post 类型的 ajax 请求
    var isPostAjax = this.isAjax("post");
}
```

isWebSocket()

判断是否是 websocket 请求。

```JavaScript
testAction: function(){
    // 是否是 WebSocket 请求
    var isWS = this.isWebSocket();
}
```

get(name)

获取 get 参数值，默认值为空字符串。

```JavaScript
testAction: function(){
    // 获取 name 值
    var name = this.get("name");
}
```

如果不传 `name`，则为获取所有的参数值。

```JavaScript
testAction: function(){
    // 获取所有的参数值
    // 如: {name: "welefen", "email": "welefen@gmail.com"}
    var data = this.get();
}
```

post(name)

获取 post 过来的值，默认值为空字符串。

```JavaScript
testAction: function(){
    // 获取 post 值
    var name = this.post("name");
}
```

如果不传 `name`，则为获取所有的 post 值。

```JavaScript
testAction: function(){
    // 获取所有的 post 值
    // 如: {name: "welefen", "email": "welefen@gmail.com"}
    var data = this.post();
}
```

param(name)

获取 get 或者 post 参数，优先从 post 里获取。等同于下面的方式：

```JavaScript
testAction: function(){
    // 这 2 种方式的结果是一样的
    var name = this.param("name");
    var name = this.post("name") || this.get("name");
}
```

file(name)

获取上传的文件。

```JavaScript
testAction: function(){
    // 获取表单名为 image 上传的文件
    var file = this.file("image");
}
```

如果不传 `name`，那么获取所有上传的文件。

具体的文件格式见：

header(name, value)

获取或者发送 header 信息。

获取头信息

获取单个头信息

```JavaScript
// 获取单个头信息
testAction: function(){
    var value = this.header("accept");
}
```

获取所有的头信息

```JavaScript
// 获取所有的头信息
testAction: function(){
    var headers = this.header();
}
```

设置头信息

设置单个头信息

```JavaScript
// 设置单个头信息
testAction: function(){
    this.header("Content-Type", "text/xml");
}
```

批量设置头信息

```JavaScript
testAction: function(){
    this.header({
        "Content-Type": "text/html",
        "X-Power": "test"
    })
}
```

userAgent()

获取浏览器端传递过来的 userAgent。

```JavaScript
// 获取 userAgent
testAction: function(){
    var userAgent = this.userAgent();
}
```

referer()

获取 referer。

```JavaScript
// 获取 referrer
testAction: function(){
    var referrer = this.referer();
}
```

cookie(name, value, options)

获取或者设置 cookie

获取 cookie

获取单个 cookie

```JavaScript
// 获取单个 cookie
testAction: function(){
    var name = this.cookie("name");
}
```

获取所有 cookie

```JavaScript
// 获取所有 cookie
testAction: function(){
    var cookies = this.cookie();
}
```

设置 cookie

设置 cookie 默认会用如下的配置，可以在 `App/Conf/config.js` 里修改。

```JavaScript
// 设置 cookie 默认配置
cookie_domain: "", //cookie 有效域名
cookie_path: "/",   //cookie 路径
cookie_timeout: 0, //cookie 失效时间，0 为浏览器关闭，单位：秒
```

```JavaScript
// 设置 cookie
testAction: fucntion(){
    this.cookie("name", "welefen");
    // 修改发送 cookie 的选项
    this.cookie("value", "xxx", {
        domain: "",
        path: "/",
        httponly: true, //httponly
        secure: true, //https 下才发送 cookie 到服务端
        timeout: 1000 // 超时时间，单位秒
    })
}
```

session(name, value)

获取或者设置 session。

获取 session

```JavaScript
// 获取 session
testAction: function(){
    // 获取 session 是个异步的过程，返回一个 promise
    this.session("userInfo").then(function(data){
        if(isEmpty(data)){
            // 无用户数据
        }
    })
}
```

设置 session

```JavaScript
testAction: function(){
    // 设置 session 也是异步操作
    this.session("userInfo", {name: "welefen"}).then(function(){

    })
}
```

删除 session

```JavaScript
testAction: function(){
    // 不传任何参数表示删除 session，比如：用户退出的时候执行删除的操作
    this.session().then(function(){

    })
}
```

redirect(url, code)

url 跳转。

- `code` 默认值为 302
- `return` 返回一个 pendding promise，阻止后面代码继续执行

```JavaScript
testAction: function(){
    var self = this;
    return this.session("userInfo").then(function(data){
        if(isEmpty(data)){
            // 如果用户未登录，则跳转到登录页面
            return self.redirect("/login");
        }
    })
}
```

assign(name, value)

给模版变量赋值，或者读取已经赋值的模版变量。

变量赋值

单个变量赋值

```JavaScript
testAction: function(){
    // 单个变量赋值
    this.assign("name", "value");
}
```

批量赋值

```JavaScript
testAction: function(){
    this.assign({
        name: "welefen",
        url: "http://www.ThinkJS.org/"
    })
}
```

读取赋值变量

```JavaScript
testAction: function(){
    // 读取已经赋值的变量
    var value = this.assign("url");
}
```

fetch(templateFile)

获取渲染后的模版文件内容。

- `templateFile` 需要渲染的模版文件路径
- `return` 返回一个 promise

模版文件路径寻找规则如下：

- 如果不传 templateFile，那么自动根据当前的 Group、Controller、Action 拼接模版文件路径
- 如果 templateFile 是个相对路径，那么自动追加 `VIEW_PATH`
- 如果 templateFile 是 `group:controller:action`，那么会进行解析再拼接成对应的模版文件路径
- 如果 templateFile 是个绝对路径，那么直接调用

```JavaScript
testAction: function(){
    this.assign('name', 'xxx');
    // 自动分析模版文件路径
    this.fetch().then(function(content){
        //content 为渲染后的内容
    });
    // 路径前面追加 VIEW_PATH
    this.fetch('home/test_a.html');
    // 绝对路径，直接调用
    this.fetch('/home/xxx/www/www.domain.com/ttt.html');
    // 调用其他分组下的模版文件
    this.fetch('home:group:detail');
    // 调用当前分组下其他的模版文件
    this.fetch('group:detail');
}
```

display(templateFile)

输出渲染后的模版文件内容到浏览器。

- `templateFile` 需要渲染的模版文件路径
- `return` 返回一个 promise

`templateFile` 查找规则与 fetch 方法的 `templateFile` 查找规则相同。

action(action, data)

可以跨分组、跨控制器的 action 调用。

- `return` 返回一个 promise

```JavaScript
testAction: function(){
    // 调用相同分组下的控制器为 group，操作为 detail 方法
    var promise = this.action("group:detail", [10]);
    // 调用 admin 分组下的控制器为 group，操作为 list 方法
    var promise = this.action("admin:group:list", [10])
}
```

jsonp(data)

jsonp 数据输出。

会自动发送 `Content-Type`，默认值为 `application/json`，可以在配置 `json_content_type` 中修改。

jsonp 的 callback 名称默认从参数名为 `callback` 中获取，可以在配置 `url_callback_name` 中修改。

callback 名称会自动做安全过滤，只保留 `\w\.` 字符。

```JavaScript
testAction: function(){
    this.jsonp({name: "xxx"});
}
```

假如当前请求为 `/test?callback=functionname`，那么输出为 `functionname({name: "xxx"})`。

`注`： 如果没有传递 callback 参数，那么以 json 格式输出。

json(data)

json 数据输出。

会自动发送 `Content-Type`，默认值为 `application/json`，可以在配置 `json_content_type` 中修改。

status(status)

发送状态码，默认为 404。

```JavaScript
// 发送 http 状态码
testAction: function(){
    this.status(403);
}
```

echo(data, encoding)

输出数据，可以指定编码。

默认自动发送 `Content-Type`，值为 `text/html`。可以在配置 `tpl_content_type` 中修改，也可以设置 `auto_send_content_type = false` 来关闭发送 `Content-Type`。

- `data` 如果是数组或者对象，自动调用 JSON.stringify。如果不是字符串或者 Buffer，那么自动转化为字符串。

```JavaScript
testAction: function(){
    this.echo({name: "welefen"});
}
```

end(data, encoding)

结束当前的 http 请求数据输出。

如果是通过 `this.echo` 输出数据，那么在最后必须要调用 `this.end` 来结束输出。

- `data` 如果 data 不为空，那么自动调用 `this.echo` 来输出数据

```JavaScript
testAction: function(){
    this.end({name: "welefen"});
}
```

type(ext)

发送 `Content-Type` 。

- `ext` 如果是文件扩展名，那么自动查找该扩展名对应的 `Mime-Type` 。

```JavaScript
testAction: function(){
    this.type("text/html");
    this.type("js"); // 自动查找 js 对应的 Mime-Type
    this.type("css"); // 自动查找 css 对应的 Mime-Type
}
```

download(file, contentType, filename)

下载文件。

- `file` 要下载的文件路径
- `contentType` 要发送的 `Content-Type` , 如果没传，自动从文件扩展名里获取
- `filename` 下载的文件名
- `return` 返回一个 promise

```JavaScript
testAction: function(){
    var file = "/home/welefen/a.txt";
    this.download(file, 'text/html', '1.txt').then(function(){
        // 下载完成后可以在这里进行一些操作，如：将下载次数 +1
    });
}
```

success(data)

输出一个错误号为 0 的数据。

- `return` 返回一个 pendding promise，阻止后面继续执行。

```JavaScript
testAction: function(){
    return this.success({email: "xxx@gmail.com"})
}
```

浏览器拿到的数据为：

```JavaScript
{
    errno: 0,
    errmsg: "",
    data: {email: "xxx@gmail.com"}
}
```

其中 `errno` 表示错误号（此时为 0），`errmsg` 表示错误信息（此时为空）。`data` 里存放具体数据。

会自动发送 `Content-Type` ，默认值为 `application/json` ，可以在配置 `json_content_type` 中修改。

其中 `errno` 和 `errmsg` 可以通过下面的配置修改：

```JavaScript
error_no_key: "errno", // 错误号的 key
error_msg_key: "errmsg", // 错误信息的 key
```

error(errno, errmsg, data)

输出一个 errno > 0 的信息。

- `errno` 错误号，默认为 1000，可以通过配置 `error_no_default_value` 修改
- `errmsg` 错误信息，字符串
- `data` 额外的数据
- `return` 返回一个 pedding promise，阻止后续继续执行

```JavaScript
// 输出一个错误信息
testAction: function(){
    return this.error(1001, "参数不合法");
}
```

浏览器拿到的数据为：

```JavaScript
{
    errno: 1001,
    errmsg: "参数不合法"
}
```

也可以值输出错误信息，那么错误号为配置 `error_no_default_value` 的值。

```JavaScript
testAction: function(){
    return this.error("参数不合法");
}
```

浏览器拿到的数据为：

```JavaScript
{
    errno: 1000,
    errmsg: "参数不合法"
}
```

也可以传个对象进去，如：

```JavaScript
testAction: function(){
    return this.error({
        errno: 10001,
        errmsg: "参数不合法"
    })
}
```

filter(value, type)

变量过滤器，具体请见 这里。

valid(data, type)

数据校验，具体请见 这里。

Model

原型方法

field(field, reverse)

设置要查询的字段。

- `field` string | array 要查询的字段，可以是字符串，也可以是数组
- `reverse` boolean 是否反选字段
- `return` this

```JavaScript
// 生成的 sql 语句为，以下同：
// SELECT * FROM `meinv_group`
D('Group').field().select();
// SELECT `id`,`title` FROM `meinv_group`
D('Group').field('id, title').select();
// SELECT `id`,`title` FROM `meinv_group`
D('Group').field(['id', 'title']).select();
// SELECT `cate_id`,`md5`,`width`,`height`,`pic_nums`,`view_nums`,`date` FROM `meinv_group`
D('Group').field(['id', 'title'], true).select();
```

table(table, hasPrefix)

设置表名

- `table` String 表名
- `hasPrefix` Boolean 表名里是否已经含有了表前缀

```JavaScript
// 设置表名为 xxx
D('Group').table('xxx').select();

// 也可以将一条 sql 语句设置为表名
D('Group').group('name').buildSql().then(function(sql){
  return D('Article').table(sql, true).select();
}).then(function(data){

})
```

limit(offset, length)

设置查询的数量。

- `offset` 起始位置
- `length` 查询的数目
- `return` this

```JavaScript
// SELECT * FROM `meinv_group` LIMIT 10
D('Group').limit(10).select();
// SELECT * FROM `meinv_group` LIMIT 10,20
D('Group').limit(10, 20).select();
```

page(page, listRows)

设置当前查询的页数，页数从 1 开始

- `page` 当前的页数
- `listRows` 一页多少条记录，默认值为 `C('db_nums_per_page')`
- `return` this

```JavaScript
// SELECT * FROM `meinv_group`
D('Group').page().select();
// SELECT * FROM `meinv_group` LIMIT 0,20
D('Group').page(1).select();
// SELECT * FROM `meinv_group` LIMIT 10,10
D('Group').page(2, 10).select();
```

union(union, all)

联合查询

- `union` 联合查询的字符串
- `all` 是否为 UNION ALL 模式
- `return` this

```JavaScript
// SELECT * FROM `meinv_pic1` UNION (SELECT * FROM meinv_pic2)
D('Pic1').union('SELECT * FROM meinv_pic2').select();
// SELECT * FROM `meinv_pic1` UNION ALL (SELECT * FROM meinv_pic2)
D('Pic1').union('SELECT * FROM meinv_pic2', true).select();
// SELECT * FROM `meinv_pic1` UNION ALL (SELECT * FROM `meinv_pic2`)
D('Pic1').union({table: 'meinv_pic2'}, true).select();
// SELECT * FROM `meinv_pic1` UNION ALL (SELECT * FROM `meinv_pic2`) UNION (SELECT * FROM meinv_pic3)
D('Pic1').union({table: 'meinv_pic2'}, true).union({table: 'meinv_pic3'}).select();
```

join(join)

组合查询

- `join` 可以是字符串、数组、对象
- `return` this

```JavaScript
// SELECT * FROM `meinv_group` LEFT JOIN meinv_cate ON meinv_group.cate_id=meinv_cate.id
D('Group').join('meinv_cate ON meinv_group.cate_id=meinv_cate.id').select();

// SELECT * FROM `meinv_group` LEFT JOIN meinv_cate ON meinv_group.cate_id=meinv_cate.id RIGHT JION meinv_tag ON meinv_group.tag_id=meinv_tag.id
D('Group').join([
  'meinv_cate ON meinv_group.cate_id=meinv_cate.id',
  'RIGHT JOIN meinv_tag ON meinv_group.tag_id=meinv_tag.id'
]).select();

// SELECT * FROM meinv_group INNER JOIN `meinv_cate` AS c ON meinv_group.`cate_id`=c.`id`
D('Group').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();

// SELECT * FROM meinv_group AS a LEFT JOIN `meinv_cate` AS c ON a.`cate_id`=c.`id` LEFT JOIN `meinv_group_tag` AS d ON a.`id`=d.`group_id`
D('Group').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()

// SELECT * FROM meinv_group AS a LEFT JOIN `meinv_cate` AS c ON a.`id`=c.`id` LEFT JOIN `meinv_group_tag` AS d ON a.`id`=d.`group_id`
D('Group').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()

//SELECT * FROM `meinv_group` LEFT JOIN `meinv_cate` ON meinv_group.`id`=meinv_cate.`id` LEFT JOIN `meinv_group_tag` ON meinv_group.`id`=meinv_group_tag.`
D('Group').join({
  cate: {
    on: ['id', 'id']
  },
  group_tag: {
    on: ['id', 'group_id']
  }
}).select()

//SELECT * FROM `meinv_group` LEFT JOIN `meinv_cate` ON meinv_group.`id`=meinv_cate.`id` LEFT JOIN `meinv_group_tag` ON meinv_group.`id`=meinv_group_tag.`
D('Group').join({
  cate: {
    on: 'id, id'
  },
```

```
  group_tag: {
    on: ['id', 'group_id']
  },
  tag: {
    on: { // 多个字段的 ON
      id: 'id',
      title: 'name'
    }
  }
}).select()

// join 的 table 为 sql 语句
// 这个方式推荐和 buildSql 结合使用，也可以单独使用
// SELECT id as team_id,ta.team_name,ifnull(sum,0) as sum FROM ThinkJS_team AS ta LEFT JOIN (SELECT tt.id as team_id,`team_name`,`team_partin_year`,sum(IF
return D('team').alias('tt')
  .field('tt.id as team_id, team_name, team_partin_year, sum(IFNULL(invest_value, 0)) as sum')
  .join({
    table: 'invest',
    join: 'left',
    as: 'ti',
    on: ['tt.id', 'invest_team_id']
  })
  .where('invest_is_cancel = 0 or invest_is_cancel is null')
  .group('tt.id')
  .order({sum: 'desc'})
  .having('team_partin_year='+year).buildSql();}).then(function(sql){
    return D('team')
    .field('id as team_id, ta.team_name, ifnull(sum, 0) as sum')
    .alias('ta').join({
      table: sql,
      join: 'left',
      as: 'temp',
      on: ['id', 'temp.team_id']
    })
    .order('sum desc').select();});
```

order(order)

设置排序方式

- `order` 排序方式，字符串
- `return` this

```
JavaScript
// SELECT * FROM `meinv_group` ORDER BY id
D('Group').order('id').select();
// SELECT * FROM `meinv_group` ORDER BY id DESC
D('Group').order('id DESC').select();
// SELECT * FROM `meinv_group` ORDER BY id DESC,title ASC
D('Group').order('id DESC,title ASC').select();
// SELECT * FROM `meinv_group` ORDER BY id ASC
D('Group').order(['id ASC']).select();
// SELECT * FROM `meinv_group` ORDER BY id ASC,title DESC
D('Group').order(['id ASC', 'title DESC']).select()
// SELECT * FROM `meinv_group` ORDER BY `id` ASC,`title` DESC
D('Group').order({id: 'ASC', title: 'DESC'}).select()
```

alias(alias)

设置表别名

- `alaias` 表别名，字符串
- `return` this

```
JavaScript
//SELECT * FROM meinv_group AS a
D('Group').alias('a').select();
```

having(str)

having 查询

- `str` having 查询的字符串
- `return` this

```JavaScript
// SELECT * FROM `meinv_group` HAVING view_nums > 1000 AND view_nums < 2000
D('Group').having('view_nums> 1000 AND view_nums <2000').select();
```

group(field)

分组查询

- `field` 设定分组查询的字段
- `return` this

```JavaScript
// SELECT * FROM `meinv_group` GROUP BY `view_nums`
D('Group').group('view_nums').select();
```

distinct(field)

去重查询

- `field` 去重的字段
- `return` this

```JavaScript
// SELECT Distinct  `view_nums` FROM `meinv_group`
D('Group').distinct('view_nums').select();
```

where(where)

设置 where 条件

- `where` 查询条件，可以是字符串、对象
- `return` this

** 普通条件 **

```JavaScript
// SELECT * FROM `meinv_group`
D('Group').where().select();
// SELECT * FROM `meinv_group` WHERE (`id` = 10)
D('Group').where({id: 10}).select();

// 查询字符串
// SELECT * FROM `meinv_group` WHERE id = 10 OR id < 2
D('GROUP').where('id = 10 OR id < 2').select();

// 操作符
// SELECT * FROM `meinv_group` WHERE (`id` != 10)
D('Group').where({id: ['!=', 10]}).select(); // 这里的操作符有> >= < <= !=
```

**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

```JavaScript
//SELECT * FROM `meinv_group` WHERE name='name'
D('GROUP').where({name: ['EXP', "='name'"]});
```

```JavaScript
// 将 view_nums 字段值加 1
D('GROUP').update({view_nums: ['EXP', 'view_nums+1']});
```

**LIKE 条件 **

```JavaScript
// LIKE 和 NOT LIKE
// SELECT * FROM `meinv_group` WHERE (`title` NOT LIKE 'welefen')
D('Group').where({title: ['NOTLIKE', 'welefen']}).select();
// SELECT * FROM `meinv_group` WHERE (`title` LIKE '%welefen%')
D('Group').where({title: ['like', '%welefen%']}).select();

//LIKE 多个值
// SELECT * FROM `meinv_group` WHERE ((`title` LIKE 'welefen' OR `title` LIKE 'suredy') )
D('Group').where({title: ['like', ['welefen', 'suredy']]}).select()

// 多个字段 LIKE 同一个值，或的关系
//SELECT * FROM `meinv_group` WHERE ((`title` LIKE '%welefen%') OR (`content` LIKE '%welefen%') )
D('Group').where({'title|content': ['like', '%welefen%']}).select();
// 多个字段 LIKE 同一个只，与的关系
//SELECT * FROM `meinv_group` WHERE ((`title` LIKE '%welefen%') AND (`content` LIKE '%welefen%') )
D('Group').where({'title&content': ['like', '%welefen%']}).select();
```

**IN 条件 **

```JavaScript
//SELECT * FROM `meinv_group` WHERE (`id` IN ('10','20') )
D('Group').where({id: ['IN', '10,20']}).select();
//SELECT * FROM `meinv_group` WHERE (`id` IN (10,20) )
D('Group').where({id: ['IN', [10, 20]]}).select();
//SELECT * FROM `meinv_group` WHERE (`id` NOT IN (10,20) )
D('Group').where({id: ['NOTIN', [10, 20]]}).select()
```

** 多字段查询 **

```JavaScript
//SELECT * FROM `meinv_group` WHERE (`id` = 10) AND (`title` = 'www')
D('Group').where({id: 10, title: "www"}).select();

// 修改为或的关系
//SELECT * FROM `meinv_group` WHERE (`id` = 10) OR (`title` = 'www')
D('Group').where({id: 10, title: "www", _logic: 'OR'}).select();

// 修改为异或的关系
//SELECT * FROM `meinv_group` WHERE (`id` = 10) XOR (`title` = 'www')
D('Group').where({id: 10, title: "www", _logic: 'XOR'})
```

**BETWEEN 查询 **

```JavaScript
//SELECT * FROM `meinv_group` WHERE ((`id` BETWEEN 1 AND 2) )
D('Group').where({id: ['BETWEEN', 1, 2]}).select();
//SELECT * FROM `meinv_group` WHERE ((`id` BETWEEN '1' AND '2') )
D('Group').where({id: ['between', '1,2']}).select()
```

** 复合查询 ** “ `js // SELECT * FROM` meinv_group `WHERE (` id `> 10 AND` id` <20) D('Group').where({ id: { '>': 10, '<': 20 } }).select()

// SELECT * FROM `meinv_group` WHERE (`id` < 10 OR `id` > 20) D('Group').where({ id: { '<': 10, '>': 20, _logic: 'OR' } }).select()

//SELECT * FROM `meinv_group` WHERE (`id` >= 10 AND `id` <= 20) OR (`title` LIKE '%welefen%') OR (`date` > '2014-08-12') D('Group').where({ id: { '>=': 10, '<=': 20 }, 'title': ['like', '%welefen%'], date: ['>', '2014-08-12'], _logic: 'OR' }).select();

//SELECT * FROM `think_group` WHERE (`title` = 'test') AND ((`id` IN (1,2,3)) OR (`content` = 'www')) D('Group').where({ title: 'test', _complex: {id: ['IN', [1, 2, 3]], content: 'www', _logic: 'or' } }).select() "`

count(field)

查询符合条件的数目，可以有 where 条件

- `field` count 的字段，默认会从数据表里查找主键的字段
- `return` promise

```JavaScript
//SELECT COUNT(id) AS ThinkJS_count FROM `meinv_group` LIMIT 1
D('Group').count('id').then(function(count){
  //count 为符合条件的数目
});
```

sum(field)

对符合条件的字段值求和，可以有 where 条件

- `field` 求和的字段
- `return` promise

```JavaScript
//SELECT SUM(view_nums) AS ThinkJS_sum FROM `meinv_group` LIMIT 1
D('Group').sum('view_nums').then(function(sum){
  //sum 为求和的值
})
```

min(field)

求字段的最小值

- `field` 要求最小值的字段
- `return` promise

```JavaScript
//SELECT MIN(view_nums) AS ThinkJS_min FROM `meinv_group` LIMIT 1
D('Group').min('view_nums').then(function(min){
  //min 为最小的 view_nums 值
})
```

max(field)

求字段的最大值

- `field` 要求最大值的字段
- `return` promise

```JavaScript
//SELECT MAX(view_nums) AS ThinkJS_max FROM `meinv_group` LIMIT 1
D('Group').max('view_nums').then(function(max){
  //max 为最小的 view_nums 值
})
```

avg(field)

求字段的平均值

- `field` 要求平均值的字段
- `return` promise

```JavaScript
//SELECT AVG(view_nums) AS ThinkJS_avg FROM `meinv_group` LIMIT 1
D('Group').avg('view_nums').then(function(avg){

})
```

add(data)

插入数据

- `data` 要插入的数据，对象
- `return` promise

```JavaScript
var data = {
    title: 'xxx',
    content: 'yyy'
};
D('Group').add(data).then(function(insertId){
  // 如果插入成功，insertId 为插入的 id
}).catch(function(err){
  // 插入失败，err 为具体的错误信息
})
```

如果数据表中有字段设置为 `unique`，插入一个已经存在的值时就会报错。这种情况一般需要先按这个字段去查询下看有没有对应的记录，如果没有在进行插入。

为了简化开发者的使用，ThinkJS 提供了 thenAdd 方法。

thenAdd(data, where, returnDetail)

当数据表中不存在 where 条件对应的数据时，才进行插入。

- `data` 要插入的数据
- `where` 检测的条件
- `returnDetail` 是否返回详细的信息

```JavaScript
// 假设数据表中字段 title 类型为 UNIQUE
var data = {
    title: 'xxx',
    content: 'yyy'
};
var where = {title: 'xxx'}
D('Group').thenAdd(data, where).then(function(id){
  //id 为已经存在的 id 或者刚插入的 id
})
// 返回详细的信息
D('Group').thenAdd(data, where, true).then(function(data){
    //data 数据结构为
    data = {
        type: 'exist' || 'add', //exist 表示之前已经存在，add 表示新添加
        id: 111
    }
})
```

使用场景：用户注册时就可以通过该方法来检查用户名或者邮箱已经存在。

addAll(data)

一次添加多条数据

- `data` 要添加的数据，数组
- `return` promise

```JavaScript
var data = [{title: 'xxx'}, {title: 'yyy'}];
D('Group').addAll(data).then(function(insertId){
  // 插入成功
}).catch(function(err){
  // 插入失败
})
```

delete()

- `return` promise

删除符合条件的数据

```JavaScript
// 删除所有数据
D('Group').delete().then(function(affectedRows){
  //affectedRows 为影响的行数
})
// 删除 id 小于 100 的数据
D('Group').where({id: ['<', 100]}).delete().then(function(affectedRows){
  //affectedRows 为影响的行数
})
```

update(data)

更新符合条件的数据

- `data` 要更新的数据
- `return` promise

```JavaScript
// 将 id<10 的 title 设置为空
D('Group').where({id: ['<', 10]}).update({title: ''})
```

select()

查询符合条件的数据

- `return` promise

```JavaScript
D('Group').where({id: ['IN', [1, 2, 3]]}).select().then(function(data){
  //data 为数组
  // 如果查询数据为空，那么 data 为 []
  data = [{
    id: 1,
    title: '',
    ...
  },{
    id: 2,
    title: ''
    ...
  }
  ]
})
```

find()

查找某一条符合条件的数据。

- `return` promise

```JavaScript
// 查询 id=1000 的一条数据
D('Group').where({id: 1000}).find().then(function(data){
  //data 为一个数据对象
  // 如果数据为空，那么 data 为 {}
  data = {
    id: 1000,
    title: 'xxx',
    ...
  }
})
```

updateInc(field, step)

将字段值增加

- `field` 要增加的字段
- `step` 增加的数值，默认为 1
- `return` promise

```JavaScript
// 将 id=10 的浏览数加 1
D('Group').where({id: 10}).updateInc('view_nums').then(function(){

})
// 将 id=100 的浏览数加 10
D('Group').where({id: 100}).updateInc('view_nums', 10).then(function(){

})
```

updateDec(field, step)

将字段值减少

- `field` 要减少的字段
- `step` 减少的值，默认为 1
- `return` promise

```JavaScript
// 将 id=10 的浏览数减 1
D('Group').where({id: 10}).updateDec('view_nums').then(function(){

})
// 将 id=100 的浏览数减 10
D('Group').where({id: 100}).updateDec('view_nums', 10).then(function(){

})
```

getField(field, onlyOne)

获取某个字段的值。

- `field` 要获取的字段，可以是一个字段，也可以是多个字段，多个字段用, 隔开

- `onlyOne` 是否只需要一个值, true 或者数字
- `return` promise

```JavaScript
// 取 id>5000 的集合，只需要 id 值，不需要其他字段值
D('Group').where({id: ['>',5000]}).getField('id').then(function(data){
  //data 为一个数组
  data = [7565,7564,7563,7562,7561,7560,7559,7558,7557]
})
// 只需要 id>5000 的一个值
D('Group').where({id: ['>', 5000]}).getField('id', true).then(function(data){
  //data 为数字
  data = 7557;
})
// 只需要 id>5000 的 3 值
D('Group').where({id: ['>', 5000]}).getField('id', 3).then(function(data){
  //data 为数字
  data = [7559, 7558, 7557];
})

// 获取 id 和 view_nums 2 个字段的值
D('Group').getField('id,view_nums').then(function(data){data = {"id":[7565,7564,7563,7562,7561,7560,7559,7558,7557],
    "view_nums":[1965,1558,2335,2013,1425,1433,1994,2035,1118]
  }
})
```

countSelect(options, flag)

- `options` 查询参数
- `flag` 当分页值不合法的时候，处理情况。true 为修正到第一页，false 为修正到最后一页，默认不进行修正。
- `return` promise

```JavaScript
// 按每页 20 条数据来展现文件
D('Article').page(this.get("page"), 20).countSelect().then(function(data){
    //data 的数据格式为
    {
        count: 123, // 总条数
        total: 10, // 总页数
        page: 1, // 当前页
        num: 20, // 每页显示多少条
        data: [{}, {}] // 详细的数据
    }
});

//countSelect 里使用 group
D('Article').page(10).group('name').countSelect().then(function(data){})

//countSelect 里有子查询
D('Group').group('name').buildSql().then(function(sql){
  return D('Artigle').table(sql, true).countSelect();
}).then(function(data){})
```

buildSql(options)

将当前查询条件生成一个 SELECT 语句，可以用作子查询的 sql 语句。

- `options` 操作选项
- `return` promise

```JavaScript
D('Cate').where({id: ['>', 10]}).buildSql().then(function(sql){//sql = SELECT `id` FROM `meinv_cate` WHERE id> 10
  return D('GROUP').where({cate_id: ['IN', sql, 'exp']});
})
```

query(sql, parse)

自定义 sql 语句进行查询，用于非常复杂的 sql 语句时使用。

- `sql` 要执行的 sql 语句
- `parse` 格式参数的数据
- `return` promise

```JavaScript
var data = [
  value.field || '*',
  mapOptions.mapfKey,
  value.rTable || self.getRelationTableName(mapOptions.model),
  mapOptions.model.getTableName(),
  whereStr || 'WHERE',
  value.rfKey || (mapOptions.model.getModelName().toLowerCase() + '_id'),
  mapOptions.model.getPk(),
  value.where ? ('AND' + value.where) : ''
]
D('Group').query('SELECT b.%s, a.%s FROM %s as a, %s as b %s AND a.%s=b.%s %s', data).then(function(data){
  // 查询的数据
})
```

sql 语句中支持如下字符串的自动替换：

- `__TABLE__` 替换为当前模型里的表名
- `__USER__` 替换为 `C('db_prefix') + 'user'` 表，其他表类似

```JavaScript
// 解析后的 sql 为 SELECT * FROM meinv_group as a LEFT JOIN meinv_user as u ON a.id=u.id WHERE u.id > 10
D('Group').query('SELECT * FROM __TABLE__ as a LEFT JOIN __USER__ as u ON a.id=u.id WHERE u.id> %d', 10);
```

execute(sql, parse)

自定义 sql 语句执行，用户复杂 sql 语句的情况。

使用方式与 `query` 相同，只是 then 里拿到的结果不同。exectue 为影响的行数。

close()

关闭当前数据库连接，非特殊条件下不要使用该方法。

startTrans()

开启事务

- `return` Promise

commit()

提交事务

- `return` Promise

rollback()

回滚事务

- `return` Promise

事务操作 DEMO:

```JavaScript
var model = D('Group');
// 开启事务
model.startTrans().then(function(){
  return model.add(data);
}).then(function(){
  return model.commit(); // 提交事务
}).catch(function(){
  return model.rollback(); // 回滚事务
})
```

注意：只有支持事务的存储引擎使用这 3 个方法才有效

静态方法

close()

关闭所有数据库连接。

```JavaScript
thinkRequire('Model').close();
```

数据库默认使用长连接的方式，不建议关闭数据库连接。该接口非特殊条件下，不要使用。