MyResty Framework

MyResty Framework

基于 OpenResty 的 Web 框架，设计参考 PHP CodeIgniter 框架。

OpenResty based Web framework, designed with reference to PHP CodeIgniter framework.

> 本项目由 AI 开发 - 使用 OpenCode 平台，基于 MiniMax 模型。所有代码（包括本 README.md）均由 AI 生成。
>
> AI-Developed Project - Built using OpenCode platform, powered by MiniMax model. All code (including this README.md) was generated by AI.

致敬 OpenResty 项目 - 感谢 OpenResty 提供了如此优秀的 Web 平台。

Tribute to OpenResty - Thanks to OpenResty for providing such an excellent Web platform.

LuaJIT Best Practices / LuaJIT 最佳实践

本项目包含一份详细的 LuaJIT 最佳实践指南，基于 OpenResty 源码分析编写。

This project includes a detailed LuaJIT best practices guide, based on OpenResty source code analysis.

查看文档: LUAJIT_BEST_PRACTICES.md

主要内容包括:

- 模块加载与代码缓存
- 连接池使用（MySQL、Redis）
- Lua FFI 使用最佳实践
- Lua Cosocket 使用指南
- Table 性能优化（table.new、table.clear）
- Shared Dictionary 使用
- 代码组织模式
- 性能优化建议
- 错误处理模式
- 安全实践

Topics Covered:

- Module loading & code caching
- Connection pool usage (MySQL, Redis)
- Lua FFI best practices
- Lua Cosocket usage guide
- Table performance optimization (table.new, table.clear)
- Shared Dictionary usage
- Code organization patterns
- Performance optimization tips
- Error handling patterns

- Security practices

Requirements / 系统要求

Operating System / 操作系统

- **OS**: Ubuntu 24.04.3 LTS / Ubuntu 24.04.3 LTS

Nginx Configuration / Nginx 配置

参考 nginx/conf/ 目录下的配置文件：

Reference nginx configuration files in the nginx/conf/ directory:

```
# 主 nginx 配置模板 / Main nginx configuration template
```

```
nginx/conf/nginx.conf
```

```

```

```
# MyResty 服务器配置 / MyResty server configuration
```

```
nginx/conf/myresty.conf
```

```

```

```
# FastCGI 配置 / FastCGI configuration
```

```
nginx/conf/fcgi.conf
```

注意：生产环境的实际配置位于服务器上的 /usr/local/web/nginx/conf/nginx.conf，该文件会 include 本项目中的 myresty.conf。

Note: The actual production configuration is located at /usr/local/web/nginx/conf/nginx.conf on the server, which includes the myresty.conf file from this project.

System Dependencies / 系统依赖包

运行前需要安装 Lua FFI 调用所需的系统包（验证码、图像处理、加密）：

Before running, install the required system packages for Lua FFI calls (captcha, image processing, encryption):

```
apt-get update && apt-get install -y build-essential libc6-dev libgd-dev libpng
```

依赖说明 / Dependencies included:

Quick Start / 快速开始

```
# 安装依赖 / Install dependencies (if not already installed)
```

```
apt-get update && apt-get install -y build-essential libc6-dev libgd-dev libpng
```

```
# 启动 nginx / Start nginx
```

```
/usr/local/web/nginx/sbin/nginx
```

```
# 测试 API / Test API
```

```
curl http://localhost:8080/
```

Architecture / 项目架构

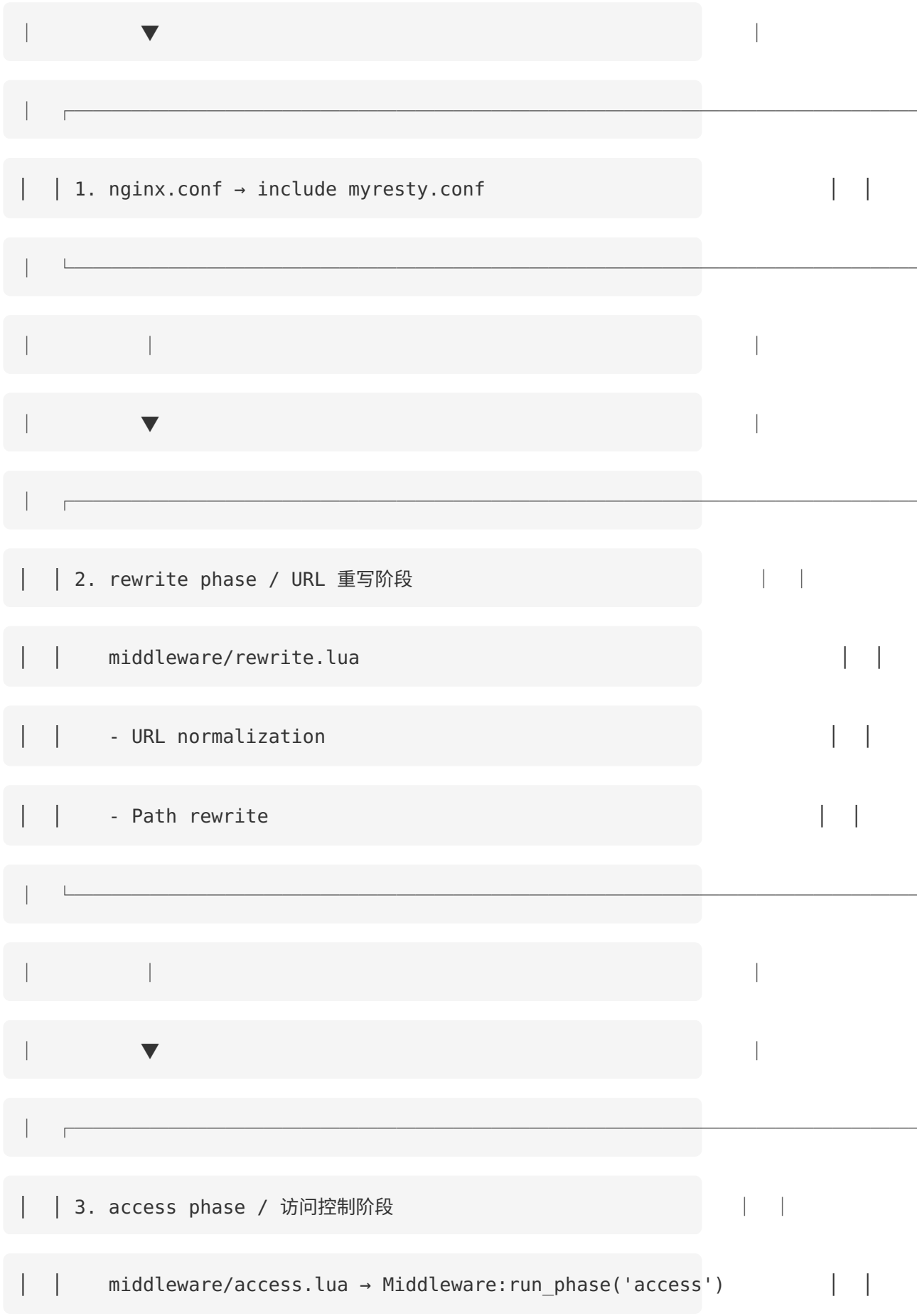
```
┌──────────────────────────────────────────────────────────────────────────────
│                          Nginx Server                                       │
│                        (OpenResty 1.27.1)                                   │
├──────────────────────────────────────────────────────────────────────────────
│  Request Flow / 请求流程:                                                   │
│                                                                             │
│   ┌──────────────────┐     ┌──────────────────┐     ┌──────────────────┐     ┌──
```


rewrite -> access -> content -> log
phase phase phase phase
rewrite auth bootstrap logger
.lua cors .lua .lua
rate_lim
Application Layer / 应用层
Bootstrap / 启动入口
bootstrap.lua

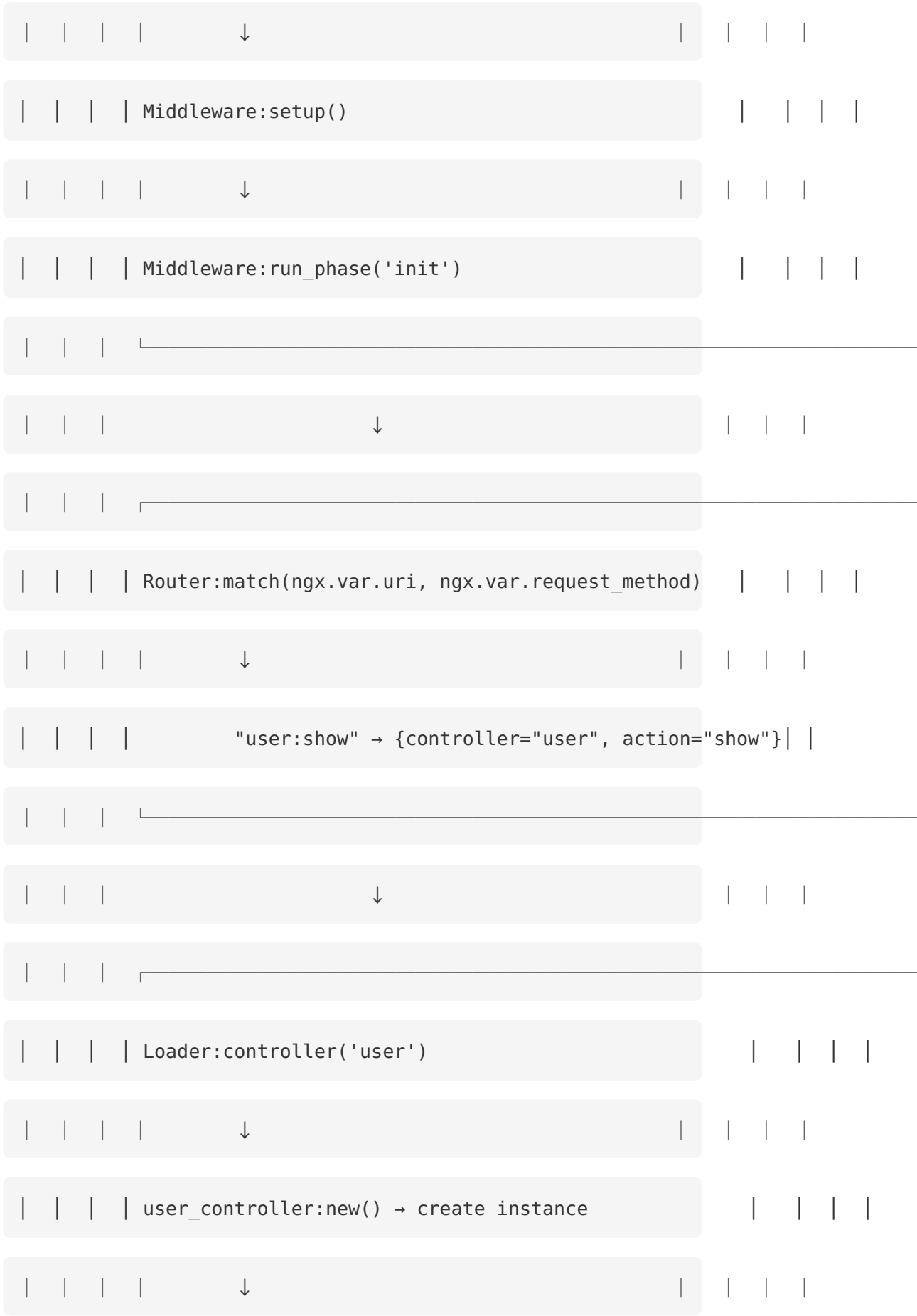

Router
Loader
Config
路由分发
自动加载
配置加载
Controller / 控制器
app/controllers/*.lua
(user.lua, session.lua, cache.lua, etc.)

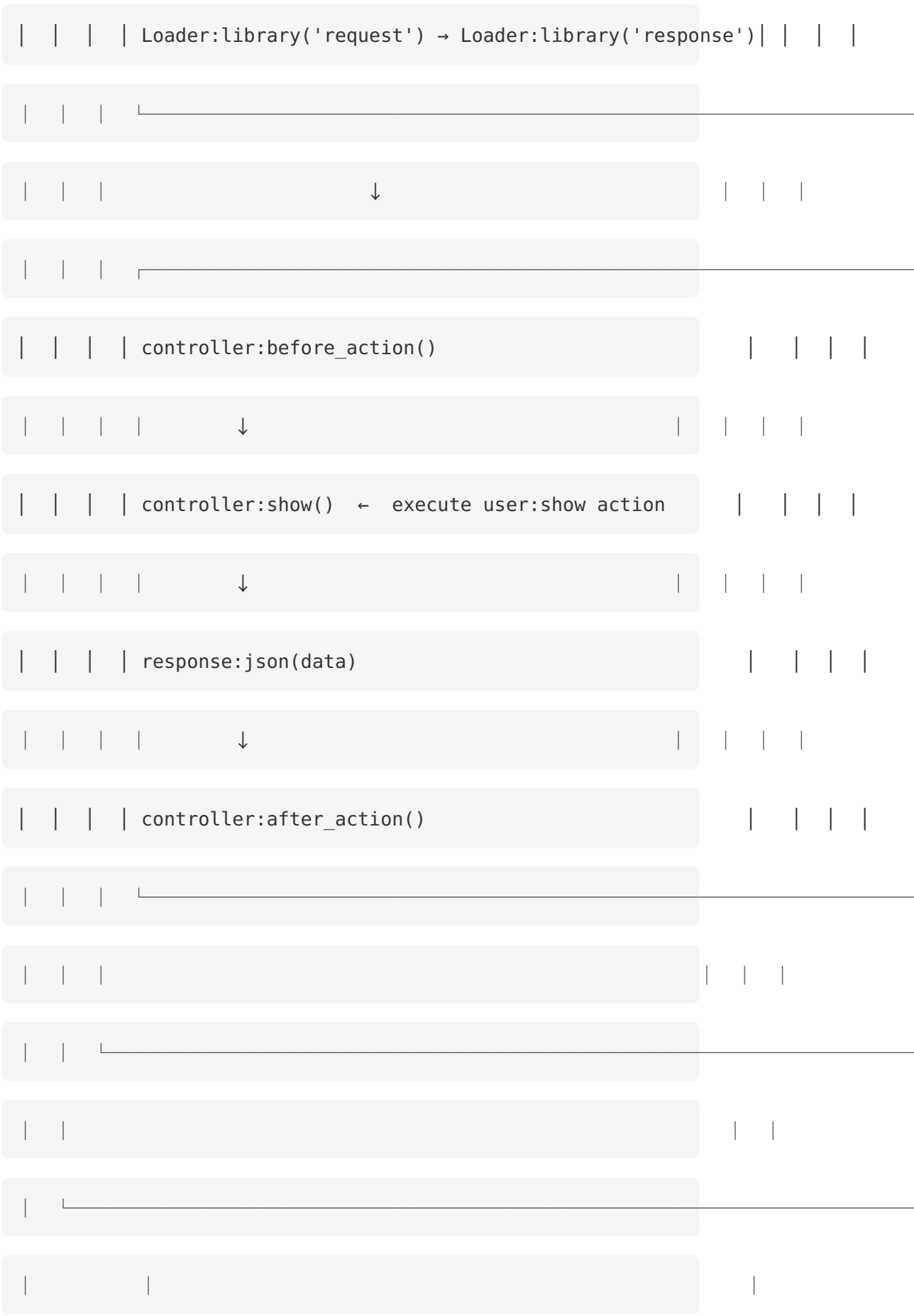

Models
Library
Helpers
数据模型
库函数
辅助函数
user_model.lua
mysql, redis,
session, cache
url, file,
string, etc.
External Services / 外部服务
MySQL
:3306
Redis
:6379
Files
System
Cache
Shared

Request Lifecycle / 请求生命周期


▼
1. nginx.conf → include myresty.conf
▼
2. rewrite phase / URL 重写阶段
middleware/rewrite.lua
- URL normalization
- Path rewrite
▼
3. access phase / 访问控制阶段
middleware/access.lua → Middleware:run_phase('access')


auth.lua
认证检查
cors.lua
跨域处理
rate_limit
限流
4. content phase / 内容处理阶段
content_by_lua_file → bootstrap.lua
bootstrap.lua
Config:load()


Middleware:setup()
Middleware:run_phase('init')
Router:match(ngx.var.uri, ngx.var.request_method)
"user:show" → {controller="user", action="show"}
Loader:controller('user')
user_controller:new() → create instance


Loader:library('request') → Loader:library('response')
controller:before_action()
controller:show() ← execute user:show action
response:json(data)
controller:after_action()


5. header_filter / 响应头过滤
middleware/header_filter.lua
- Add CORS headers
- Add rate limit headers
- Set Content-Type
6. body_filter / 响应体过滤
middleware/body_filter.lua
- Response body processing

Module Introduction / 模块功能介绍

Core Modules / 核心模块

Library Modules / 库模块

Utility Modules / 工具模块

Middleware Modules / 中间件模块

Nginx Middleware / Nginx 中间件

Controllers / 控制器

API Reference / API 参考

Controller / 控制器

控制器是所有业务控制器的基类，提供常用方法。

Controller is the base class for all business controllers, providing common methods.

```
local BaseController = require('app.core.Controller')

local UserController = {}

function UserController:new()
local instance = BaseController:new()
instance.user_model = self:load_model('user')
return setmetatable(instance, { __index = UserController })
end
```

控制器方法 / Controller Methods

Request / 请求处理

Request 模块用于解析 HTTP 请求。

Request module is used to parse HTTP requests.

```
local Request = require('app.core.Request')
```

```
local req = Request:new()
```

```
req:fetch()
```

Request 方法 / Request Methods

请求数据示例 / Request Data Example

```
-- GET 请求: /users?name=john&age=25
```

```
req:get        -- { name = "john", age = "25" }
```

```
req:segment(1) -- "users"
```

```

```

```
-- POST 请求 (JSON): {"name":"john","email":"john@example.com"}
```

```
req:post       -- { name = "john", email = "john@example.com" }
```

```
req:json       -- { name = "john", email = "john@example.com" }
```

```
req:all_input  -- 合并后的所有输入
```

Response / 响应处理

Response 模块用于构建 HTTP 响应。

Response module is used to build HTTP responses.

```
local Response = require('app.core.Response')
```

```
local res = Response:new()
```

Response 方法 / Response Methods

响应示例 / Response Examples

```
-- JSON 响应
```

```
res:json({ success = true, data = { id = 1, name = "John" } })
```

```
res:send()
```

```

```

```
-- 成功响应
```

```
res:success({ id = 1 }, "操作成功")
```

```
res:send()
```

```

```

```
-- 失败响应

res:fail("参数错误", { field = "email" }, 400)

res:send()

-- 分页响应

res:paginate(users, 100, 1, 10)

res:send()

-- 重定向

res:redirect("/home")
```

Session / 会话管理

Session 模块提供基于加密 Cookie 的会话管理。

Session module provides encrypted cookie-based session management.

```
local Session = require('app.lib.session')
```

```
local session = Session:new()
```

```
session:start()
```

Session 方法 / Session Methods

Session 配置选项 / Session Options

Session 使用示例 / Session Example

```
local session = Session:new()
```

```
session:start()
```

```

```

```
-- 存储用户信息
```

```
session:set('user_id', 123)
```

```
session:set('username', 'john')
```

```

```

```
-- 获取用户信息
```

```
local user_id = session:get('user_id')
```

```
local username = session:get('username')
```

```

```

```
-- 检查是否存在

if session:has('user_id') then

-- 用户已登录

end


-- 销毁会话

session:destroy()
```

Cache / 缓存管理

Cache 模块基于 Nginx shared_dict 提供高性能缓存。

Cache module provides high-performance caching based on Nginx shared_dict.

```
local Cache = require('app.lib.cache')

local cache = Cache:new({

dict_name = 'my_resty_cache',

default_ttl = 3600,
```

```
prefix = 'cache:'
```

```
})
```

Cache 方法 / Cache Methods

Cache 使用示例 / Cache Example

```
local cache = Cache:new()
```

```

```

```
-- 设置缓存
```

```
cache:set('user:123', { name = 'John', age = 25 }, 3600)
```

```

```

```
-- 获取缓存
```

```
local user = cache:get('user:123')
```

```

```

```
-- 检查是否存在
```

```
if cache:exists('user:123') then
```

```
-- 缓存存在
```

```
end
```

```

```

```
-- 删除缓存
cache:delete('user:123')

-- 批量操作
cache:set('key1', 'value1')
cache:set('key2', 'value2')
local values = cache:get_multi('key1', 'key2')
```

Validation / 数据验证

Validation 模块提供 35+ 数据验证规则。

Validation module provides 35+ data validation rules.

```
local Validation = require('app.lib.validation')
local validator = Validation:new()
```

验证规则 / Validation Rules

Validation 方法 / Validation Methods

Validation 使用示例 / Validation Example

```
local validator = Validation:new()

local data = {

email = "john@example.com",

password = "password123",

confirm_password = "password123",

age = 25

}

local rules = {

email = { "required", "email" },

password = { "required", "length_min:6" },

confirm_password = { "required", "match:password" },

age = { "required", "number", "min:18", "max:120" }

}
```

```
validator:make(data, rules)

if validator:validate() then
-- 验证通过
else
local errors = validator:errors()
-- 处理错误
end
```

Router / 路由

Router 模块负责将 URL 映射到控制器方法。

Router module maps URLs to controller methods.

```
local Router = require('app.core.Router')
local route = Router:new()
```

路由方法 / Router Methods

路由参数 / Route Parameters

Router 使用示例 / Router Example

```
-- 静态路由

route:get('/users', 'user:list')

route:post('/users', 'user:create')

route:get('/users/{id}', 'user:show')

route:put('/users/{id}', 'user:update')

route:delete('/users/{id}', 'user:destroy')

-- 参数路由

route:get('/posts/{slug}', 'post:view')

route:get('/users/{id}/posts/{post_id}', 'user_post:show')

-- RESTful 资源路由

route:resource('users', 'user')
```

```
-- 自动生成:
-- GET    /users           -> user:index
-- GET    /users/new       -> user:new
-- GET    /users/{id}      -> user:show
-- GET    /users/{id}/edit -> user:edit
-- POST   /users           -> user:create
-- PUT    /users/{id}      -> user:update
-- DELETE /users/{id}      -> user:destroy

-- 路由分组
route:group({ prefix = '/api/v1', middleware = { 'auth' } }, function()
route:get('/users', 'api_user:list')
route:get('/orders', 'api_order:list')
end)
```

Loader / 自动加载

Loader 模块提供模块自动加载功能。

Loader module provides automatic module loading.

```
local Loader = require('app.core.Loader')
```

Loader 方法 / Loader Methods

Config / 配置管理

Config 模块加载和管理应用配置。

Config module loads and manages application configuration.

```
local Config = require('app.core.Config')
```

```
Config.load()
```

Config 方法 / Config Methods

Config 配置结构 / Config Structure

```
-- app/config/config.lua

return {

app = {

host = '0.0.0.0',

port = 8080,

base_url = '',

log_threshold = 4,

},

mysql = {

host = '127.0.0.1',

port = 3306,

user = 'root',

password = '',

database = '',

pool_size = 100,

},

redis = {
```

```
host = '127.0.0.1',
port = 6379,
password = '',
pool_size = 100,
},
session = {
cookie_name = 'session',
cookie_max_age = 86400,
},
autoload = {},  -- 自动加载的模型
middleware = {
{ name = 'cors', phase = 'header_filter' },
{ name = 'rate_limit', phase = 'access' },
},
}
```

Middleware / 中间件

Middleware 模块管理系统中间件。

Middleware module manages application middleware.

```
local Middleware = require('app.middleware')
```

Middleware 方法 / Middleware Methods

内置中间件 / Built-in Middleware

Configuration / 配置说明

配置文件位于 app/config/config.lua，包含应用的所有可配置参数。

Configuration file is located at app/config/config.lua, containing all configurable parameters.

配置文件结构 / Config File Structure

```lua
-- app/config/config.lua

return {

-- 应用基础配置

app = { ... },

-- MySQL 数据库配置

mysql = { ... },

-- Redis 配置

redis = { ... },

-- 文件上传配置

upload = { ... },

-- 图像处理配置

image = { ... },

-- 验证码配置

captcha = { ... },

-- 会话配置

session = { ... },

-- 日志配置
```

```
logger = { ... },

-- 限流配置

limit = { ... },

-- 数据验证配置

validation = { ... },

-- 中间件配置

middleware = { ... }

}
```

App / 应用基础配置

MySQL / 数据库配置

Redis / 缓存配置

Upload / 文件上传配置

```
-- 默认允许的文件类型
allowed_types = {
'jpg', 'jpeg', 'png', 'gif', 'webp',  -- 图片
'pdf',                                 -- 文档
'doc', 'docx',                         -- Word
'xls', 'xlsx',                         -- Excel
'zip'                                  -- 压缩包
}
```

Image / 图像处理配置

Captcha / 验证码配置

安全说明: 验证码加密密钥与会话使用相同的 session.secret_key，无需单独配置。

Session / 会话配置

安全警告:

- `secret_key` 必须保持机密，泄露后攻击者可伪造任意 session
- 生成新密钥命令: `openssl rand -hex 16`
- 修改密钥会导致所有现有 session 失效

配置优先级:

1. 环境变量 SESSION_SECRET (最高)
2. 环境变量 MYRESTY_SESSION_SECRET
3. config.session.secret_key

Logger / 日志配置

日志级别:

Limit / 限流配置

限流策略 / Limit Strategies

限流区域 / Limit Zones

```
limit = {
zones = {
api = { limit = 60, window = 60, burst = 10 },
login = { limit = 5, window = 300, burst = 0 },
upload = { limit = 10, window = 60, burst = 2 },
default = { limit = 100, window = 60, burst = 20 }
}
}
```

Validation / 数据验证配置

默认错误消息 / Default Messages

默认字段标签 / Default Labels

Middleware / 中间件配置

中间件配置是一个数组，每个元素包含：

内置中间件 / Built-in Middleware

CORS 中间件选项 / CORS Options

Rate Limit 中间件选项 / Rate Limit Options

Auth 中间件选项 / Auth Options

中间件配置示例 / Middleware Config Example

```
middleware = {
-- 日志中间件
{
name = 'logger',
phase = 'log',
options = {
level = 'info',
format = 'combined',
request_id = true,
timing = true,
exclude_paths = {'/health', '/favicon.ico'}
}
},
-- CORS 中间件
{
name = 'cors',
```

```
phase = 'header_filter',
options = {
origin = '*',
methods = {'GET', 'POST', 'PUT', 'DELETE', 'PATCH', 'OPTIONS'},
credentials = true,
max_age = 86400
}
},
-- 限流中间件
{
name = 'rate_limit',
phase = 'access',
options = {
zone = 'default',
headers = true,
log_blocked = true
},
```

```
routes = {'/api/*', '/upload/*'}
},
-- 认证中间件
{
name = 'auth',
phase = 'access',
options = {
mode = 'session',
allow_guest = true
},
exclude = {'/health', '/captcha', '/static/*', '/middleware/*'}
}
}
```

Project Structure / 项目结构

```
├── app/ # 应用核心代码
│   ├── core/ # 核心类模块
```

```
│   ├── Config.lua # 配置加载器
│   ├── Controller.lua # 控制器基类
│   ├── Loader.lua # 自动加载器
│   ├── Model.lua # 数据模型基类
│   ├── QueryBuilder.lua # 查询构建器
│   ├── Request.lua # 请求处理
│   ├── Response.lua # 响应处理
│   └── Router.lua # 路由分发
│
├── controllers/ # 控制器 (18个)
│   ├── welcome.lua # 默认首页
│   ├── user.lua # 用户管理
│   ├── session.lua # 会话管理
│   ├── cache.lua # 缓存控制
│   ├── captcha.lua # 验证码
│   ├── upload.lua # 文件上传
│   ├── image.lua # 图像处理
│   ├── validate.lua # 数据验证
│   ├── rate_limit.lua # 限流控制
│   ├── http_client.lua # HTTP 客户端
│   ├── demo.lua # 功能演示
│   ├── request_demo.lua # 请求演示
│   ├── middleware_demo.lua # 中间件演示
│   ├── request_test.lua # 请求测试
│   └── example.lua # 示例控制器
│
├── lib/ # 库文件 (11个)
```

```
│   │   ├── mysql.lua # MySQL 客户端
│   │   ├── redis.lua # Redis 客户端
│   │   ├── session.lua # 会话管理
│   │   ├── cache.lua # 缓存管理
│   │   ├── limit.lua # 限流控制
│   │   ├── validation.lua # 数据验证
│   │   ├── logger.lua # 日志管理
│   │   ├── http.lua # HTTP 客户端
│   │   ├── crypto.lua # 加密工具
│   │   ├── test.lua # 测试工具
│   │   └── validator.lua # 验证辅助
│   │
│   ├── middleware/ # 应用中间件
│   │   ├── auth.lua # 认证中间件
│   │   ├── cors.lua # 跨域中间件
│   │   ├── logger.lua # 日志中间件
│   │   └── rate_limit.lua # 限流中间件
│   │
│   ├── helpers/ # 辅助函数
│   │   ├── url_helper.lua # URL 辅助
│   │   ├── file_helper.lua # 文件辅助
│   │   ├── string_helper.lua # 字符串辅助
│   │   ├── captcha_helper.lua # 验证码辅助
│   │   ├── request_helper.lua # 请求辅助
│   │   └── query_helper.lua # 查询辅助
│   │
│   ├── utils/ # 工具函数
```

```
│   │   ├── captcha.lua # 验证码生成 (FFI)
│   │   ├── image.lua # 图像处理 (FFI)
│   │   ├── file.lua # 文件操作 (FFI)
│   │   ├── helper.lua # 通用辅助
│   │   └── test.lua # 测试工具
│   │
│   ├── models/ # 数据模型
│   │   └── user_model.lua # 用户模型
│   │
│   ├── routes/ # 路由配置
│   │   └── routes.lua # 路由定义
│   │
│   ├── config/ # 应用配置
│   │   └── config.lua # 主配置
│   │
│   └── middleware.lua # 中间件管理
│
├── middleware/ # Nginx 中间件
│   ├── rewrite.lua # URL 重写
│   ├── access.lua # 访问控制
│   ├── header_filter.lua # 响应头过滤
│   ├── body_filter.lua # 响应体过滤
│   ├── log.lua # 日志记录
│   └── set.lua # 变量设置
│
├── nginx/conf/ # Nginx 配置模板
│   ├── myresty.conf # MyResty 服务器配置
```

```
│   ├── nginx.conf # 主配置模板
│   └── fcgi.conf # FastCGI 配置
│
├── config/ # 配置目录
│   └── validation/ # 验证规则配置
│   ├── users.lua # 用户验证规则
│   ├── products.lua # 产品验证规则
│   ├── orders.lua # 订单验证规则
│   └── common.lua # 通用验证规则
│
├── tests/ # 测试套件
│   ├── integration/ # 集成测试
│   │   ├── test.sh # 测试脚本
│   │   └── README.md # 测试说明
│   │
│   ├── unit/ # 单元测试
│   │   ├── all.lua # 测试入口
│   │   ├── run.lua # 测试运行器
│   │   ├── config_spec.lua # 配置测试
│   │   ├── request_spec.lua # 请求测试
│   │   ├── response_spec.lua # 响应测试
│   │   ├── router_spec.lua # 路由测试
│   │   ├── validation_spec.lua # 验证测试
│   │   └── ...
│   │
│   └── api.lua # API 测试
│
```

```
├── fonts/ # 字体文件
├── logs/ # 日志目录
├── static/ # 静态文件
│
├── bootstrap.lua # 应用启动入口
├── init.lua # 初始化脚本
├── init_worker.lua # Worker 初始化
├── generate_readme_pdf.py # PDF 生成脚本
├── generate_pdf.py # PDF 生成工具
└── README.md # 项目说明
```

```
---
```

```
## Testing / 测试
```

```
MyResty 框架提供完整的测试体系，包括单元测试和集成测试。
```

```
MyResty framework provides a complete testing system, including unit tests and
```

```
---
```

```
### Unit Tests / 单元测试
```

单元测试位于 `tests/unit/` 目录，用于测试框架核心模块。

Unit tests are located in `tests/unit/`, used to test framework core modules.

单元测试目录结构 / Unit Test Structure

tests/unit/
├─── all.lua # 测试入口，运行所有测试套件
├─── run.lua # 测试运行器
├─── config_spec.lua # 配置模块测试
├─── router_spec.lua # 路由模块测试
├─── helper_spec.lua # 辅助函数测试
├─── request_spec.lua # 请求模块测试
├─── response_spec.lua # 响应模块测试
├─── query_builder_spec.lua # 查询构建器测试
├─── cache_spec.lua # 缓存模块测试
├─── session_spec.lua # 会话模块测试
├─── http_spec.lua # HTTP 客户端测试
└─── ...

测试工具 / Test Framework

测试框架位于 `app/utils/test.lua`，提供 BDD 风格的测试 API。

```
Test framework is located at `app/utils/test.lua`, providing BDD-style testing
```

local Test = require('app.utils.test')

```
#### 测试工具 API / Test Framework API

| 函数 | 参数 | 说明 |
|------|------|------|
| **describe(name, fn)** | `name`: 测试套件名, `fn`: 测试函数 | 定义测试套件 |
| **it(name, fn)** | `name`: 测试名, `fn`: 测试函数 | 定义测试用例 |
| **before_each(fn)** | `fn`: 前置函数 | 每个测试前执行 |
| **after_each(fn)** | `fn`: 后置函数 | 每个测试后执行 |
| **pending(name, reason)** | `name`: 测试名, `reason`: 原因 | 标记待完成测试 |
| **run(options)** | `options`: 选项 | 运行所有测试 |
| **reset()** | 无 | 重置测试状态 |

#### 断言函数 / Assertions

| 断言 | 说明 |
|------|------|
```

```
| `assert.equals(expected, actual, msg)` | 期望值等于实际值 |
| `assert.not_equals(expected, actual)` | 期望值不等于实际值 |
| `assert.is_true(value, msg)` | 值为 true |
| `assert.is_false(value, msg)` | 值为 false |
| `assert.is_nil(value, msg)` | 值为 nil |
| `assert.not_nil(value, msg)` | 值不为 nil |
| `assert.is_function(value, msg)` | 值是函数 |
| `assert.is_table(value, msg)` | 值是表 |
| `assert.is_string(value, msg)` | 值是字符串 |
| `assert.has_key(key, table, msg)` | 表包含键 |
| `assert.error(fn, msg, expected_err)` | 函数抛出错误 |
| `assert.no_error(fn, msg)` | 函数不抛出错误 |
| `assert.matches(pattern, value, msg)` | 值匹配模式 |
| `assert.approx(actual, expected, tolerance)` | 近似值比较 |

#### 运行单元测试 / Run Unit Tests

# 运行所有测试
lua tests/unit/all.lua
```

安静模式（最小输出）

lua tests/unit/all.lua --quiet

JSON 格式输出

lua tests/unit/all.lua --json

运行单个测试套件

lua tests/unit/run.lua

```
#### 编写单元测试 / Write Unit Tests
```

-- tests/unit/my_feature_spec.lua

package.path = '/var/www/web/my-openresty/?.lua;/var/www/web/my-openresty/?/init.lua;/usr/local/web/?.lua;;'

package.cpath = '/var/www/web/my-openresty/?.so;/usr/local/web/lualib/?.so;;'

local Test = require('app.utils.test')

describe = Test.describe

it = Test.it

pending = Test.pending

before_each = Test.before_each

after_each = Test.after_each

assert = Test.assert

```
describe('MyFeature', function()
local feature

before_each(function()
-- 每个测试前创建新实例
feature = { value = 0 }
end)

after_each(function()
-- 每个测试后清理
feature = nil
end)

it('should initialize with zero value', function()
assert.equals(0, feature.value)
end)

it('should increment value', function()
feature.value = feature.value + 1
assert.equals(1, feature.value)
end)

it('should handle string operations', function()
local result = string.upper('hello')
assert.equals('HELLO', result)
assert.is_string(result)
```

end)

it('should match pattern', function()
local email = 'user@example.com'
assert.matches('@', email)
assert.matches('example', email)
end)

pending('should handle error cases', 'Not implemented yet')
end)

```
---

### Integration Tests / 集成测试

集成测试位于 `tests/integration/`, 通过 curl 命令测试 API 端点。

Integration tests are located in `tests/integration/`, testing API endpoints vi

#### 集成测试目录 / Integration Test Directory
```

tests/integration/
├── test.sh # 集成测试脚本
├── nginx_test.conf # 测试用 Nginx 配置
└── README.md # 测试说明文档

```
#### 运行集成测试 / Run Integration Tests
```

进入测试目录

cd /var/www/web/my-openresty

确保 nginx 已启动

/usr/local/web/nginx/sbin/nginx

运行所有集成测试

./tests/integration/test.sh

自定义测试服务器地址

BASE_URL=http://localhost:8080 ./tests/integration/test.sh

详细输出模式

VERBOSE=true ./tests/integration/test.sh

```
#### 测试脚本命令 / Test Script Commands
```

```
| 命令 | 说明 |
```

```
|------|------|
```

```
| `./tests/integration/test.sh` | 运行所有测试 |
```

```
| `BASE_URL=http://localhost:8080` | 自定义测试服务器 |
```

```
| `VERBOSE=true` | 显示详细输出 |

| `COOKIE_JAR=/tmp/cookies.txt` | 自定义 Cookie 文件 |

#### 集成测试脚本函数 / Test Script Functions

| 函数 | 参数 | 说明 |

|------|------|------|

| **curl_get(endpoint, description)** | `endpoint`: 端点, `description`: 描述 | G

| **curl_post(endpoint, data, description, content_type)** | `endpoint`: 端点, `

| **curl_put(endpoint, data, description)** | `endpoint`: 端点, `data`: 数据, `de

| **curl_delete(endpoint, description)** | `endpoint`: 端点, `description`: 描述

#### 集成测试输出示例 / Integration Test Output

==========================================

MyResty API Integration Test Suite

==========================================

Base URL: http://localhost:8080

---

[PASS] GET / - Root endpoint (HTTP 200)
```

[PASS] GET /test - Test endpoint (HTTP 200)

[PASS] GET /users - List users (HTTP 200)

[PASS] GET /users/123 - Get user by ID (HTTP 200)

[PASS] POST /users - Create user (HTTP 201)

[PASS] PUT /users/123 - Update user (HTTP 200)

[PASS] DELETE /users/123 - Delete user (HTTP 200)

[PASS] GET /session - Session index (HTTP 200)

[PASS] POST /session/set - Set session value (HTTP 200)

[PASS] POST /session/get - Get session value (HTTP 200)

[PASS] POST /session/clear - Clear session (HTTP 200)

==========

Test Results Summary

==========

Total tests: 50

Passed: 50

Failed: 0

All tests passed!

```
---
```

```
### Helper Functions / 辅助函数

Helper 函数提供便捷的常用操作封装。

Helper functions provide convenient wrappers for common operations.

---

#### Request Helper / 请求辅助函数

请求辅助函数位于 `app/helpers/request_helper.lua`, 提供强大的请求数据处理能力。

Request helper is located at `app/helpers/request_helper.lua`, providing powerf
```

local RequestHelper = require('app.helpers.request_helper')

```
##### Request Helper 方法 / Request Helper Methods

| 方法 | 参数 | 返回值 | 说明 |

|------|------|--------|------|

| **get(fields, rules)** | `fields`: 字段列表, `rules`: 验证规则 | `data, errors` |
```

```
| **get_get(fields, rules)** | 同上 | 同上 | 仅获取 GET 参数 |

| **get_post(fields, rules)** | 同上 | 同上 | 仅获取 POST 参数 |

| **get_json(fields, rules)** | 同上 | 同上 | 仅获取 JSON 参数 |

| **get_only(fields, rules, source)** | `source`: 数据源 | `data, errors` | 带验证

| **get_except(fields, blacklist)** | `blacklist`: 排除列表 | table | 获取除指定外的

| **only(...)** | `...`: 字段名 | table | 仅获取指定字段 |

| **except(...)** | `...`: 排除字段 | table | 排除指定字段 |

| **merge(defaults)** | `defaults`: 默认值 | table | 合并默认值 |

| **validate(fields, rules)** | `fields`: 字段, `rules`: 规则 | `valid, data, erro

| **get_pagination_params(default_per_page)** | `default_per_page`: 默认每页数 |

| **get_search_params(search_fields)** | `search_fields`: 搜索字段 | `params, key

| **get_order_params(default_field, default_order)** | `default_field`: 默认排序

| **get_date_range_params(prefix)** | `prefix`: 前缀 | table | 获取日期范围参数 |

##### 请求数据源 / Request Data Sources

| 来源 | 说明 | 示例 |

|------|------|------|
```

| `get` | URL 查询参数 | `?page=1&limit=10` |

| `post` | POST 表单数据 | `application/x-www-form-urlencoded` |

| `json` | JSON Body | `{"name":"john","age":25}` |

| `all` | 合并所有来源 | 自动合并 |

验证规则 / Validation Rules

| 规则 | 参数 | 说明 |

|------|------|------|

| `required` | 无 | 必填 |

| `type` | `"string"`/`"number"`/`"integer"`/`"boolean"`/`"array"` | 类型转换 |

| `default` | 默认值 | 默认值 |

| `trim` | `true`/`false` | 是否 trim |

| `strip_tags` | `true`/`false` | 是否去除 HTML 标签 |

| `lowercase` | `true`/`false` | 转为小写 |

| `uppercase` | `true`/`false` | 转为大写 |

Request Helper 使用示例 / Request Helper Example

```
local RequestHelper = require('app.helpers.request_helper')

-- 基本字段获取
local data = RequestHelper:get(self, {'name', 'email', 'age'})

-- 带类型转换
local rules = {
name = { type = "string", default = "" },
age = { type = "integer", default = 0 },
is_active = { type = "boolean", default = false }
}
local data = RequestHelper:get(self, {'name', 'age', 'is_active'}, rules)

-- 带验证
local rules = {
username = { required = true, message = '用户名必填' },
email = { required = true, type = "email", message = '邮箱格式错误' }
}
local valid, data, errors = RequestHelper:validate(self, {'username', 'email'}, rules)

-- 分页参数
local pagination = RequestHelper:get_pagination_params(self, 20)
-- 返回: { page = 1, per_page = 20, sort_by = 'id', sort_order = 'DESC', offset = 0, limit = 20 }
```

-- 搜索参数

local params, keyword = RequestHelper:get_search_params(self, {'title', 'content'})

-- keyword = "hello", params = { title = "hello", content = "hello" }

```
---

#### File Helper / 文件辅助函数

文件辅助函数位于 `app/helpers/file_helper.lua`，提供文件操作和类型检查功能。

File helper is located at `app/helpers/file_helper.lua`, providing file operati
```

local FileHelper = require('app.helpers.file_helper')

```
##### File Helper 方法 / File Helper Methods

| 方法 | 参数 | 返回值 | 说明 |
|------|------|--------|------|
| **format_size(bytes)** | `bytes`: 字节数 | 格式化大小 | 格式化文件大小 |
| **safe_path(base_path, filename)** | `base_path`: 基础路径, `filename`: 文件名 |
| **sanitize_filename(filename)** | `filename`: 文件名 | 净化后文件名 | 净化文件名 |
```

| **get_extension(filename)** | `filename`: 文件名 | 扩展名 | 获取文件扩展名 |

| **mime_to_ext(mime)** | `mime`: MIME 类型 | 扩展名 | MIME 转扩展名 |

| **is_image(mime)** | `mime`: MIME 类型 | boolean | 是否为图片 |

| **is_document(mime)** | `mime`: MIME 类型 | boolean | 是否为文档 |

| **is_archive(mime)** | `mime`: MIME 类型 | boolean | 是否为压缩包 |

| **is_audio(mime)** | `mime`: MIME 类型 | boolean | 是否为音频 |

| **is_video(mime)** | `mime`: MIME 类型 | boolean | 是否为视频 |

支持的 MIME 类型 / Supported MIME Types

图片 (image): `image/jpeg`, `image/png`, `image/gif`, `image/webp`, `image/sv

文档 (document): `application/pdf`, `application/msword`, `application/vnd.op

压缩包 (archive): `application/zip`, `application/x-zip-compressed`, `applicat

音频 (audio): `audio/mpeg`, `audio/wav`, `audio/ogg`, `audio/mp3`

视频 (video): `video/mp4`, `video/x-msvideo`, `video/webm`, `video/quicktime`

```
##### File Helper 使用示例 / File Helper Example
```

```
local FileHelper = require('app.helpers.file_helper')

-- 格式化文件大小
FileHelper.format_size(1024) -- "1.00 KB"
FileHelper.format_size(1048576) -- "1.00 MB"
FileHelper.format_size(1073741824) -- "1.00 GB"

-- 安全路径（防止目录遍历攻击）
local path, err = FileHelper.safe_path('/var/www/uploads', '../../../etc/passwd')
-- path = nil, err = "Path outside allowed directory"

-- 净化文件名
local safe_name = FileHelper.sanitize_filename('../../hack.php')
-- 返回 "hack.php"

-- 检查文件类型
if FileHelper.is_image('image/jpeg') then
-- 是图片
end

if FileHelper.is_document('application/pdf') then
-- 是文档
end
```

```
---

#### String Helper / 字符串辅助函数

字符串辅助函数位于 `app/helpers/string_helper.lua`, 提供字符串处理功能。

String helper is located at `app/helpers/string_helper.lua`, providing string m
```

local StringHelper = require('app.helpers.string_helper')

```
##### String Helper 方法 / String Helper Methods

| 方法 | 参数 | 返回值 | 说明 |
|------|------|--------|------|
| **trim(s)** | `s`: 字符串 | 去除首尾空格 | 去除首尾空格 |
| **ltrim(s)** | `s`: 字符串 | 去除左侧空格 | 去除左侧空格 |
| **rtrim(s)** | `s`: 字符串 | 去除右侧空格 | 去除右侧空格 |
| **random_string(length)** | `length`: 长度 | 随机字符串 | 生成随机字符串 |
| **ucfirst(s)** | `s`: 字符串 | 首字母大写 | 首字母大写 |
```

```
##### String Helper 使用示例 / String Helper Example
```

local StringHelper = require('app.helpers.string_helper')

StringHelper.trim(' hello ') -- "hello"

StringHelper.ltrim(' hello') -- "hello"

StringHelper.rtrim('hello ') -- "hello"

StringHelper.random_string(16) -- "aBc1XyZ9PqR2mNkL"

StringHelper.ucfirst('hello') -- "Hello"

```
---
```

```
#### URL Helper / URL 辅助函数
```

```
URL 辅助函数位于 `app/helpers/url_helper.lua`, 提供 URL 生成功能。
```

```
URL helper is located at `app/helpers/url_helper.lua`, providing URL generation
```

local UrlHelper = require('app.helpers.url_helper')

```
##### URL Helper 方法 / URL Helper Methods
```

```
| 方法 | 参数 | 返回值 | 说明 |
```

```
|------|------|--------|------|

| **base_url()** | 无 | 基础 URL | 获取应用基础 URL |

| **site_url(uri)** | `uri`: 相对路径 | 完整 URL | 生成站点 URL |

##### URL Helper 使用示例 / URL Helper Example
```

local UrlHelper = require('app.helpers.url_helper')

UrlHelper.base_url() -- "http://localhost:8080"

UrlHelper.site_url('users') -- "http://localhost:8080/users"

UrlHelper.site_url('api/v1') -- "http://localhost:8080/api/v1"

```
---

#### Captcha Helper / 验证码辅助函数

验证码辅助函数位于 `app/helpers/captcha_helper.lua`，提供验证码生成和验证功能。

Captcha helper is located at `app/helpers/captcha_helper.lua`, providing captch
```

local CaptchaHelper = require('app.helpers.captcha_helper')

```
```

```
##### Captcha Helper 方法 / Captcha Helper Methods

| 方法 | 参数 | 返回值 | 说明 |
|------|------|--------|------|
| **generate(length, key)** | `length`: 长度, `key`: 密钥 | `code, encrypted` | 生
| **validate(input_code, ngx)** | `input_code`: 用户输入, `ngx`: 请求对象 | `valid,
| **refresh(ngx, key)** | `ngx`: 请求对象, `key`: 密钥 | 新验证码 | 刷新验证码 |
| **get_captcha_image(code, width, height)** | `code`: 验证码, `width`: 宽度, `hei
| **get_captcha_png_base64(code, width, height)** | 同上 | Base64 字符串 | 生成 Bas

##### Captcha Helper 使用示例 / Captcha Helper Example
```

local CaptchaHelper = require('app.helpers.captcha_helper')

-- 生成验证码

local captcha = CaptchaHelper:generate(5)

-- 返回: { code = "ABC12", encrypted = "...", cookie_name = "captcha_token", expires = 300 }

-- 验证验证码

local valid, message = CaptchaHelper:validate(user_input, ngx)

-- valid = true, message = "Captcha validated"

-- 生成验证码图片

local png_data = CaptchaHelper:get_captcha_image("ABC12", 120, 40)

-- 生成 Base64 图片

local base64 = CaptchaHelper:get_captcha_png_base64("ABC12", 120, 40)

-- 返回: "..."

```
---

#### Query Helper / 查询辅助函数

查询辅助函数位于 `app/helpers/query_helper.lua`，提供便捷的数据库查询方法。

Query helper is located at `app/helpers/query_helper.lua`, providing convenient
```

local QueryHelper = require('app.helpers.query_helper')

```
##### Query Helper 方法 / Query Helper Methods

| 方法 | 参数 | 返回值 | 说明 |

|------|------|--------|------|
```

```
| **table(table_name)** | `table_name`: 表名 | QueryBuilder | 获取查询构建器 |
| **qb(table_name)** | `table_name`: 表名 | QueryBuilder | 同上 |
| **db()** | 无 | Model 实例 | 获取数据库实例 |
| **query(sql)** | `sql`: SQL 语句 | 结果 | 执行原生 SQL |
| **select(sql)** | `sql`: SQL 语句 | 结果 | 执行查询 |
| **insert(table, data)** | `table`: 表, `data`: 数据 | 插入结果 | 插入数据 |
| **update(table, data, where)** | `table`: 表, `data`: 数据, `where`: 条件 | 更新结
| **delete(table, where)** | `table`: 表, `where`: 条件 | 删除结果 | 删除数据 |
| **count(table, where)** | `table`: 表, `where`: 条件 | 数量 | 统计数量 |
| **transaction(callback)** | `callback`: 回调函数 | `ok, err` | 事务操作 |

##### Query Helper 使用示例 / Query Helper Example
```

local QueryHelper = require('app.helpers.query_helper')

-- 查询构建器

local users = QueryHelper.table('users'):where('status', 'active'):get()

-- 便捷方法

QueryHelper.insert('users', { name = 'John', email = 'john@example.com' })

QueryHelper.update('users', { status = 'active' }, { id = 1 })

QueryHelper.delete('users', { status = 'deleted' })

local count = QueryHelper.count('users', { status = 'active' })

-- 事务操作
local ok, err = QueryHelper.transaction(function()
QueryHelper.insert('users', { name = 'John' })
QueryHelper.insert('orders', { user_id = result_id, total = 100 })
end)

HTTP Client / HTTP 客户端

HTTP 客户端位于 `app/lib/http.lua`，基于 OpenResty cosocket 的异步 HTTP 客户端。

HTTP client is located at `app/lib/http.lua`, an async HTTP client based on Ope

local HttpClient = require('app.lib.http')
local client = HttpClient:new({ timeout = 30000 })

HTTP Client 方法 / HTTP Client Methods

方法	参数	返回值	说明
new(options)	`options`: 配置	实例	创建客户端
get(url, options)	`url`: 地址, `options`: 选项	`response, err`	GET 请求
post(url, options)	`url`: 地址, `options`: 选项	`response, err`	POST 请
put(url, options)	`url`: 地址, `options`: 选项	`response, err`	PUT 请求
patch(url, options)	`url`: 地址, `options`: 选项	`response, err`	PATCH
delete(url, options)	`url`: 地址, `options`: 选项	`response, err`	DELE
head(url, options)	`url`: 地址, `options`: 选项	`response, err`	HEAD 请
options(url, options)	`url`: 地址, `options`: 选项	`response, err`	OPT
json(url, data, method)	`url`: 地址, `data`: 数据, `method`: 方法	`respon
form(url, data, method)	`url`: 地址, `data`: 数据, `method`: 方法	`respon
download(url, filepath)	`url`: 地址, `filepath`: 保存路径	`success, err`
set_timeout(timeout)	`timeout`: 超时时间	self	设置超时

响应结构 / Response Structure

{

status = 200, -- HTTP 状态码

body = '{"data":"..."}', -- 响应体

headers = { -- 响应头

['content-type'] = 'application/json',

['content-length'] = '100'

},

success = true -- 是否成功 (200-299)

}

```
#### HTTP Client 使用示例 / HTTP Client Example
```

local HttpClient = require('app.lib.http')

local client = HttpClient:new({

timeout = 30000,

pool_size = 10

})

-- GET 请求

local res, err = client:get('https://api.example.com/users')

if res.success then

local data = ngx.decode_json(res.body)

end

-- POST JSON 请求

local res, err = client:json('https://api.example.com/users', {

```
name = 'John',
email = 'john@example.com'
})

-- POST 表单请求
local res, err = client:form('https://api.example.com/login', {
username = 'john',
password = 'secret'
})

-- 带查询参数
local res, err = client:get('https://api.example.com/users', {
query = {
page = 1,
limit = 10,
status = 'active'
},
headers = {
['Authorization'] = 'Bearer token123'
}
})

-- 下载文件
local ok, err = client:download('https://example.com/file.zip', '/tmp/file.zip')

-- 设置超时
```

client:set_timeout(60000)

```
---

### Crypto Library / 加密库

加密库位于 `app/lib/crypto.lua`, 基于 OpenSSL FFI 的统一加密库。

Crypto library is located at `app/lib/crypto.lua`, unified encryption library b
```

local Crypto = require('app.lib.crypto')

```
#### Crypto 方法 / Crypto Methods

| 方法 | 参数 | 返回值 | 说明 |
|------|------|--------|------|
| **encrypt(data, key)** | `data`: 数据, `key`: 密钥 | `encrypted, err` | AES-256-
| **decrypt(data, key)** | `data`: 数据, `key`: 密钥 | `decrypted, err` | AES-256-
| **base64_encode(data)** | `data`: 数据 | Base64 字符串 | Base64 编码 |
| **base64_decode(data)** | `data`: 数据 | 原始数据 | Base64 解码 |
| **random_bytes(length)** | `length`: 长度 | 随机字节 | 生成随机字节 |
```

```
| **encrypt_captcha(plaintext)** | `plaintext`: 明文 | 加密字符串 | 加密验证码 |
| **decrypt_captcha(encrypted)** | `encrypted`: 密文 | 明文 | 解密验证码 |
| **encrypt_session(plaintext)** | `plaintext`: 明文 | 加密字符串 | 加密会话 |
| **decrypt_session(encrypted)** | `encrypted`: 密文 | 明文 | 解密会话 |

#### 密钥获取顺序 / Key Priority

1. 环境变量 `SESSION_SECRET`
2. 环境变量 `MYRESTY_SESSION_SECRET`
3. 配置文件 `session.secret_key`
4. 默认密钥（不推荐用于生产）

#### Crypto 使用示例 / Crypto Example
```

local Crypto = require('app.lib.crypto')

-- 基本加密/解密

local encrypted = Crypto.encrypt('Hello World', '32-byte-secret-key-here!')

local decrypted = Crypto.decrypt(encrypted, '32-byte-secret-key-here!')

-- Base64 编码/解码

local encoded = Crypto.base64_encode('Hello World')

local decoded = Crypto.base64_decode(encoded)

-- 生成随机字节

local random = Crypto.random_bytes(32)

-- 返回 32 字节的随机数据

-- 验证码加密

local encrypted_captcha = Crypto.encrypt_captcha('ABC12')

local decrypted_captcha = Crypto.decrypt_captcha(encrypted_captcha)

-- decrypted_captcha = 'ABC12'

-- 会话加密

local encrypted_session = Crypto.encrypt_session('{"user_id":1}')

local decrypted_session = Crypto.decrypt_session(encrypted_session)

-- decrypted_session = '{"user_id":1}'

```

---

```

```
### Middleware Details / 中间件详情
```

```
#### Auth Middleware / 认证中间件
```

```
认证中间件位于 `app/middleware/auth.lua`, 支持 Session 和 Token 两种认证模式。
```

```
Auth middleware is located at `app/middleware/auth.lua`, supporting both Sessio
```

local Auth = require('app.middleware.auth')

```
##### Auth 方法 / Auth Methods
```

```
| 方法 | 参数 | 返回值 | 说明 |
```

```
|------|------|--------|------|
```

```
| **setup(options)** | `options`: 配置 | self | 配置中间件 |
```

```
| **handle(options)** | `options`: 配置 | boolean | 执行认证 |
```

```
| **login(options)** | `options`: 登录数据 | session | 用户登录 |
```

```
| **logout()** | 无 | boolean | 用户登出 |
```

```
| **get_user()** | 无 | `user_id, user_data, type` | 获取当前用户 |
```

```
| **is_guest()** | 无 | boolean | 是否为游客 |
```

```
| **has_role(role)** | `role`: 角色名 | boolean | 是否有角色 |
```

```
##### Auth 配置选项 / Auth Options
```

```
| 选项 | 默认值 | 说明 |
|------|--------|------|
| `mode` | `'session'` | 认证模式: `session`, `token`, `both` |
| `token_header` | `'Authorization'` | Token 请求头 |
| `token_prefix` | `'Bearer'` | Token 前缀 |
| `session_name` | `'myresty_session'` | Session 名 |
| `login_url` | `'/auth/login'` | 登录页 URL |
| `allow_guest` | `false` | 允许游客访问 |
| `api_key_enabled` | `false` | 启用 API Key |
| `roles` | `nil` | 角色限制 |

##### Auth 使用示例 / Auth Example
```

local Auth = require('app.middleware.auth')

-- 基本认证

Auth:setup():handle()

-- 允许游客

Auth:setup():handle({ allow_guest = true })

```
-- 角色限制
Auth:setup():handle({ roles = { 'admin', 'moderator' } })

-- Token 认证
Auth:setup({
mode = 'token',
token_header = 'Authorization',
token_prefix = 'Bearer'
}):handle()

-- 登录
local session = Auth:login({
user_id = 123,
user_data = {
username = 'john',
email = 'john@example.com',
role = 'admin'
}
})

-- 登出
Auth:logout()

-- 获取当前用户
local user_id, user_data, auth_type = Auth:get_user()

-- 检查角色
```

if Auth:has_role('admin') then

-- 是管理员

end

```
---

#### Rate Limit Middleware / 限流中间件

限流中间件位于 `app/middleware/rate_limit.lua`, 提供滑动窗口限流策略。

Rate limit middleware is located at `app/middleware/rate_limit.lua`, providing
```

local RateLimit = require('app.middleware.rate_limit')

```
##### RateLimit 方法 / RateLimit Methods

| 方法 | 参数 | 返回值 | 说明 |

|------|------|--------|------|

| **setup(options)** | `options`: 配置 | self | 配置中间件 |

| **handle(options)** | `options`: 配置 | boolean | 执行限流 |

| **create_zone(name, limit, window, burst)** | `name`: 区域名, `limit`: 限制, `wi
```

```
| **zone(name)** | `name`: 区域名 | builder | 区域构建器 |

##### RateLimit 配置选项 / RateLimit Options

| 选项 | 默认值 | 说明 |
|------|--------|------|
| `zone` | `'default'` | 限流区域 |
| `default_limit` | `60` | 默认限制 |
| `default_window` | `60` | 默认窗口(秒) |
| `headers` | `true` | 输出限流头 |
| `key_by_ip` | `true` | 按 IP 限流 |
| `key_by_user` | `false` | 按用户限流 |
| `response_status` | `429` | HTTP 状态码 |
| `response_message` | `'Too Many Requests'` | 响应消息 |
| `log_blocked` | `true` | 记录被拦截 |

##### 预定义区域 / Predefined Zones

| 区域 | 限制 | 窗口 | 说明 |
```

```
|------|------|------|------|
| `api` | 60 | 60 | API 接口限流 |
| `login` | 5 | 300 | 登录接口限流 |
| `upload` | 10 | 60 | 文件上传限流 |
| `default` | 100 | 60 | 默认限流 |

##### RateLimit 使用示例 / RateLimit Example

local RateLimit = require('app.middleware.rate_limit')

-- 基本限流
RateLimit:setup():handle()

-- 自定义区域
RateLimit:setup({
zone = 'api',
headers = true,
log_blocked = true
}):handle()

-- 创建新区域
RateLimit:create_zone('special', 10, 60, 5)
RateLimit:setup({ zone = 'special' }):handle()
```

-- 响应头

-- X-RateLimit-Limit: 60

-- X-RateLimit-Remaining: 59

-- X-RateLimit-Reset: 1699999999

-- X-RateLimit-Window: 60

```
---

### Example Controllers / 示例控制器

框架提供多个示例控制器，演示各种功能的使用方法。

The framework provides several example controllers demonstrating various featur

---

#### Example Controller / 查询构建器示例

`app/controllers/example.lua` 演示 QueryBuilder 的各种用法。

`app/controllers/example.lua` demonstrates various QueryBuilder usages.
```

```
| 端点 | 方法 | 说明 |
```

```
|------|------|------|
```

```
| `/query/basic` | GET | 基本查询 |
```

```
| `/query/joins` | GET | JOIN 查询 |
```

```
| `/query/where` | GET | 条件查询 |
```

```
| `/query/aggregates` | GET | 聚合函数 |
```

```
| `/query/insert` | GET | 插入示例 |
```

```
| `/query/update` | GET | 更新示例 |
```

```
| `/query/delete` | GET | 删除示例 |
```

```
| `/query/complex` | GET | 复杂查询 |
```

```
| `/query/raw` | GET | 原生表达式 |
```

```
```

-- 示例代码

local QueryBuilder = require('app.core.QueryBuilder')

-- 基本查询

local sql = QueryBuilder.new('users')

:select('id', 'name', 'email')

:where('status', 'active')

:order_by('created_at', 'DESC')

:limit(10)

:get_sql()

-- JOIN 查询
local sql = QueryBuilder.new('users')
:select('users.id', 'users.name', 'orders.total')
:left_join('orders')
:on('users.id', '=', 'orders.user_id')
:get_sql()

```
---

#### Request Demo Controller / 请求处理演示

`app/controllers/request_demo.lua` 演示 RequestHelper 的完整用法。

`app/controllers/request_demo.lua` demonstrates complete RequestHelper usage.

| 端点 | 方法 | 说明 |
|------|------|------|
| `/request-demo` | GET | 索引 |
| `/request-demo/basic` | POST | 基本字段提取 |
| `/request-demo/typed` | POST | 类型转换 |
```

```
| `/request-demo/validate` | POST | 验证示例 |
```

```
| `/request-demo/pagination` | POST | 分页参数 |
```

```
| `/request-demo/search` | POST | 搜索参数 |
```

```
| `/request-demo/filter` | POST | 过滤参数 |
```

```
| `/request-demo/only` | POST | 仅获取指定字段 |
```

```
| `/request-demo/except` | POST | 排除指定字段 |
```

```
| `/request-demo/complete` | POST | 完整示例 |
```

```
| `/request-demo/get-post` | POST | GET vs POST vs JSON |
```

```

```

```
---
```

```

```

```
#### Middleware Demo Controller / 中间件演示
```

```

```

```
`app/controllers/middleware_demo.lua` 演示中间件系统的使用方法。
```

```

```

```
`app/controllers/middleware_demo.lua` demonstrates middleware system usage.
```

```

```

```
| 端点 | 方法 | 说明 |
```

```
|------|------|------|
```

| `/middleware` | GET | 中间件索引 |

| `/middleware/list` | GET | 列出中间件 |

| `/middleware/info` | GET | 中间件信息 |

| `/middleware/headers` | GET | 响应头 |

| `/middleware/auth-test` | POST | 认证测试 |

| `/middleware/login` | POST | 登录 |

| `/middleware/logout` | POST | 登出 |

| `/middleware/cors-test` | GET | CORS 测试 |

| `/middleware/rate-limit-test` | GET | 限流测试 |

Request Test Controller / 请求测试

`app/controllers/request_test.lua` 测试各种请求数据的获取方式。

`app/controllers/request_test.lua` tests various request data retrieval methods

| 端点 | 方法 | 说明 |

```
|------|------|------|
| `/request` | GET | 索引 |
| `/request/get` | GET | GET 参数 |
| `/request/post/form` | POST | POST 表单 |
| `/request/post/json` | POST | POST JSON |
| `/request/mixed` | ANY | 混合数据 |
| `/request/all` | GET | 所有输入 |

---

#### Validate Config Controller / 配置验证

`app/controllers/validate_config.lua` 演示基于配置文件的验证方式。

`app/controllers/validate_config.lua` demonstrates config-based validation.

| 端点 | 方法 | 说明 |
|------|------|------|
| `/validate-config` | GET | 索引 |
```

```
| `/validate-config/tables` | GET | 列出验证表 |
```

```
| `/validate-config/table/{name}` | GET | 获取表规则 |
```

```
| `/validate-config/users/{scenario}` | POST | 用户验证 |
```

```
| `/validate-config/products/{scenario}` | POST | 产品验证 |
```

```
| `/validate-config/orders/{scenario}` | POST | 订单验证 |
```

```

```

```
**支持的场景 / Supported Scenarios**:
```

```

```

```
| 表名 | 场景 |
```

```
|------|------|
```

```
| `users` | `create`, `login`, `profile`, `update` |
```

```
| `products` | `create`, `update` |
```

```
| `orders` | `create`, `ship` |
```

```

```

```
---
```

```

```

```
### Available Routes / 可用路由
```

```

```

```
框架定义了 150+ 路由，涵盖所有功能模块。
```

```

```

```
The framework defines 150+ routes covering all functional modules.

#### 核心路由 / Core Routes

| 路由 | 控制器 | 说明 |
|------|--------|------|
| `GET /` | welcome | 首页 |
| `GET /hello/{name}` | welcome | 问候 |
| `GET /users` | user | 用户列表 |
| `GET /users/{id}` | user | 用户详情 |
| `POST /users` | user | 创建用户 |
| `PUT /users/{id}` | user | 更新用户 |
| `DELETE /users/{id}` | user | 删除用户 |

#### Query Builder 路由 / Query Builder Routes

| 路由 | 说明 |
|------|------|
| `GET /query/basic` | 基本查询 |
```

| `GET /query/joins` | JOIN 查询 |

| `GET /query/where` | 条件查询 |

| `GET /query/aggregates` | 聚合函数 |

| `GET /query/insert` | 插入示例 |

| `GET /query/update` | 更新示例 |

| `GET /query/delete` | 删除示例 |

| `GET /query/complex` | 复杂查询 |

| `GET /query/raw` | 原生表达式 |

会话路由 / Session Routes

| 路由 | 方法 | 说明 |

|---|---|---|

| `GET /session` | GET | 索引 |

| `POST /session/set` | POST | 设置会话 |

| `POST /session/get` | POST | 获取会话 |

| `POST /session/remove` | POST | 删除键 |

| `POST /session/clear` | POST | 清空会话 |

| `POST /session/destroy` | POST | 销毁会话 |

| `POST /session/flash/set` | POST | 设置 Flash |

| `POST /session/flash/get` | POST | 获取 Flash |

| `POST /session/user/login` | POST | 用户登录 |

| `GET /session/user/info` | GET | 用户信息 |

| `POST /session/user/logout` | POST | 用户登出 |

缓存路由 / Cache Routes

| 路由 | 方法 | 说明 |

|------|------|------|

| `GET /cache` | GET | 索引 |

| `POST /cache/set` | POST | 设置缓存 |

| `POST /cache/get` | POST | 获取缓存 |

| `POST /cache/delete` | POST | 删除缓存 |

| `POST /cache/clear` | POST | 清空缓存 |

| `POST /cache/incr` | POST | 增值 |

| `POST /cache/decr` | POST | 减值 |

| `GET /cache/keys` | GET | 列出键 |

| `POST /cache/remember` | POST | 记住缓存 |

| `GET /cache/stats` | GET | 统计信息 |

验证码路由 / Captcha Routes

| 路由 | 方法 | 说明 |

|------|------|------|

| `GET /captcha` | GET | 生成验证码 |

| `GET /captcha/code` | GET | 获取验证码 |

| `POST /captcha/verify` | POST | 验证验证码 |

| `POST /captcha/refresh` | POST | 刷新验证码 |

文件上传路由 / Upload Routes

| 路由 | 方法 | 说明 |

|------|------|------|

| `GET /upload` | GET | 索引 |

| `POST /upload` | POST | 上传文件 |

```
| `POST /upload/multiple` | POST | 多文件上传 |
```

```
| `POST /upload/validate` | POST | 验证上传 |
```

```
| `GET /upload/form` | GET | 上传表单 |
```

```

```

```
#### 图像处理路由 / Image Routes
```

```

```

```
| 路由 | 方法 | 说明 |
```

```
|------|------|------|
```

```
| `GET /image` | GET | 索引 |
```

```
| `POST /image/upload` | POST | 上传图片 |
```

```
| `POST /image/upload/multiple` | POST | 多图上传 |
```

```
| `POST /image/upload/avatar` | POST | 上传头像 |
```

```
| `POST /image/upload/variants` | POST | 上传变体 |
```

```
| `GET /image/info/{path}` | GET | 图片信息 |
```

```
| `GET /image/thumbnail/{path}` | GET | 生成缩略图 |
```

```
| `GET /image/optimize/{path}` | GET | 优化图片 |
```

```

```

```
#### HTTP 客户端路由 / HTTP Client Routes
```

```

```

```
| 路由 | 方法 | 说明 |
|------|------|------|
| `GET /httpclient` | GET | 索引 |
| `GET /httpclient/get` | GET | GET 请求 |
| `POST /httpclient/post` | POST | POST 请求 |
| `POST /httpclient/json` | POST | JSON 请求 |
| `POST /httpclient/api` | POST | API 调用 |
| `GET /httpclient/benchmark` | GET | 性能测试 |

#### 限流路由 / Rate Limit Routes

| 路由 | 方法 | 说明 |
|------|------|------|
| `GET /rate-limit` | GET | 索引 |
| `GET /rate-limit/test` | GET | 限流测试 |
| `GET /rate-limit/status` | GET | 状态查询 |
| `POST /rate-limit/check` | POST | 检查限流 |
| `GET /rate-limit/zone` | GET | 区域信息 |
```

| `GET /rate-limit/keys` | GET | 列出键 |

| `POST /rate-limit/reset` | POST | 重置限流 |

| `GET /rate-limit/login` | GET | 登录限流 |

| `GET /rate-limit/api` | GET | API 限流 |

| `GET /rate-limit/strict` | GET | 严格限流 |

| `GET /rate-limit/combined` | GET | 组合限流 |

| `GET /rate-limit/user` | GET | 用户限流 |

验证路由 / Validation Routes

| 路由 | 方法 | 说明 |

|------|------|------|

| `GET /validate` | GET | 索引 |

| `POST /validate/basic` | POST | 基本验证 |

| `POST /validate/login` | POST | 登录验证 |

| `POST /validate/register` | POST | 注册验证 |

| `POST /validate/update` | POST | 更新验证 |

| `POST /validate/custom` | POST | 自定义验证 |

```
| `POST /validate/messages` | POST | 自定义消息 |
| `POST /validate/array` | POST | 数组验证 |
| `POST /validate/all` | POST | 完整验证 |
| `POST /validate/login-captcha` | POST | 登录验证码 |
| `POST /validate/api-key` | POST | API Key 验证 |

---

## Features / 功能特性

- MVC 架构（CodeIgniter 风格）/ MVC architecture (CodeIgniter style)
- RESTful 路由 / RESTful routing
- MySQL 和 Redis 连接池 / MySQL & Redis connection pools
- 带 AES 加密的会话管理 / Session management with AES encryption
- 共享字典缓存 / Shared dictionary caching
- 限流 / Rate limiting
- 数据验证（35+ 规则）/ Data validation (35+ rules)
- 验证码生成 / Captcha generation
- 文件上传和图像处理 / File upload & image processing
```

```
- HTTP 客户端 / HTTP client

- 全面的中间件系统 / Comprehensive middleware system

- Helper 辅助函数系统 / Helper functions system

- 基于 FFI 的高性能加密 / FFI-based high-performance encryption

- 配置驱动的数据验证 / Config-driven data validation


---
```