MIX PHP 开发指南

目　录

欢迎使用 MixPHP

高性能 • 轻量级 • 命令行

『 基于 Swoole 的FPM、常驻内存、协程三模 PHP 高性能框架 』

当前文档对应版本号 >= [MixPHP v1.1.0-beta]

核心特征

- 高性能：极简架构 + Swoole引擎，超过 Phalcon 这类 C 扩展框架的性能；
- 服务器：框架自带 mix-httpd 替代 Apache/PHP-FPM 作为高性能 HTTP 服务器；
- 协程：采用 Swoole 原生协程与最新的 PHP Stream 一键协程化技术。
- 连接池：通用的连接池组件，PDO/Redis 等组件默认接入连接池。
- WebSocket：具备长连接开发能力，扩展了 PHP 开发领域；
- 多进程：简易的多进程命令行开发，充分利用多核性能，可处理大量数据；
- 长连接：按进程保持的长连接，支持 Mysql/Redis；
- 命令行：封装了命令行开发基础设施，可快速开发定时任务、守护进程；
- 组件：基于组件的框架结构，并集成了大量开箱即用的组件；
- 中间件：AOP (面向切面编程)，注册方便，能更好的对请求进行过滤和处理；
- 门面：核心组件全部内置门面类，助力快速开发；
- 路由：底层全正则实现，性能高，配置简单；
- 验证器：集成了使用简单但功能强大的验证器，支持多场景控制；
- 视图：使用 PHP 做模板引擎，支持布局、属性；
- 自动加载：基于 PSR-4 ，完全使用 Composer 构建；
- 模块化：支持 Composer ，可以很方便的使用第三方库；
- 日志：基于 PSR-3 的日志组件。

GitHub

> 支持的用户请加个Star吧，让更多人发现MixPHP。

https://github.com/mixstart/mixphp

官网

http://mixphp.cn

技术交流

作者微博：http://weibo.com/onanying ，关注最新进展

官方QQ群：284806582(满) 825122875，敲门暗号：phper

License

Apache License Version 2.0, http://www.apache.org/licenses/

安装说明

常规安装

常规安装

常驻内存模式、协程模式的 HTTP 开发都使用该方法安装。

环境要求

> 必须的

- PHP 版本 >= 7.0
- Swoole >= 1.9.5 （常驻同步模式）
- Swoole >= 4.2.2 （常驻协程模式）
- mbstring 扩展

> 可选的

- Composer (修改一级目录，安装第三方库需要)
- gd 扩展 (Image组件需要)
- pdo 扩展 (Pdo组件需要)
- redis 扩展 (Redis组件需要)

环境搭建

1. 安装 Swoole 扩展

pecl 在 php/bin 目录，国内 pecl 安装 swoole 有时很慢，如果无法忍受，可选择 编译安装。

```
$> pecl install swoole
```

2. 安装 MixPHP

使用 composer 安装，但是一般情况下， `composer` 安装的是最新的稳定版本，不一定是最新版本。

```
composer create-project mixstart/mixphp --prefer-dist
```

如果你需要安装实时更新的版本：

```
composer create-project mixstart/mixphp=v1.1.1 --prefer-dist
```

入口文件安装至 `/usr/local/bin` ，（可选，不安装可直接执行入口文件）。

```
$> cd /data/mixphp-master
$> chmod 777 install.sh
$> ./install.sh
```

3. 确认安装成功

启动 mix-httpd 服务器。

> - 请使用 root 账号启动 mix-httpd。
> - 初次部署建议不使用 `-d` 参数，这样能方便发现目录权限不足，路径不对等系统级错误问题。

```
$> mix-httpd service start -d
```

访问测试：

```
$> curl http://127.0.0.1:9501/
Hello World
```

如果显示 "Hello World" 的欢迎语那就表示 MixPHP 已经正常运行。

4. 增加 Nginx 反向代理

```
server {
    server_name www.test.com;
    listen 80;
    root /data/mixphp/apps/httpd/public;

    location = / {
        rewrite ^(.*)$ /index last;
    }

    location / {
        proxy_http_version 1.1;
        proxy_set_header Connection "keep-alive";
        proxy_set_header Host $http_host;
        proxy_set_header Scheme $scheme;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        if (!-e $request_filename) {
             proxy_pass http://127.0.0.1:9501;
        }
    }
}
```

> 在 MixPHP 中通过读取 Request::header('x-real-ip') 或者 Request::header('x-forwarded-for') 来获取客户端的真实IP。

Swoole IDE 自动补全

这个不是必须安装的，只是能方便在需要写一些原生 Swoole 时，能让 IDE 自动补全，很方便的一个工具，推荐安装。

>> 到 GitHub 下载 swoole-ide-helper-phar <<

只安装命令行

只安装命令行

适合只使用 MixPHP 的多进程命令行开发、协程命令行开发，WebSocket 开发等这些传统框架所不具备的高级功能，而不使用其他功能的用户。

环境要求

> 必须的

- PHP 版本 >= 7.0
- Swoole >= 1.9.5 （常驻同步模式）
- Swoole >= 4.2.2 （常驻协程模式）
- mbstring 扩展

> 可选的

- Composer (修改一级目录，安装第三方库需要)
- gd 扩展 (Image组件需要)
- pdo 扩展 (Pdo组件需要)
- redis 扩展 (Redis组件需要)

环境搭建

1. 安装 Swoole 扩展

pecl 在 php/bin 目录，国内 pecl 安装 swoole 有时很慢，如果无法忍受，可选择 编译安装。

```
$> pecl install swoole
```

2. 安装 MixPHP

使用 composer 安装，但是一般情况下， `composer` 安装的是最新的稳定版本，不一定是最新版本。

```
composer create-project mixstart/mixphp --prefer-dist
```

如果你需要安装实时更新的版本：

```
composer create-project mixstart/mixphp=v1.1.1 --prefer-dist
```

入口文件安装至 `/usr/local/bin` ，（可选，不安装可直接执行入口文件）。

```
$> cd /data/mixphp-master
$> chmod 777 install.sh
$> ./install.sh
```

3. 确认安装成功

查看命令行程序的帮助。

```
$> mix-console -h
```

执行成功将显示默认注册的全部命令。

```
$> mix-console -h
Usage: ./mix-console [command] [options]

Commands:
- assemblyline
    assemblyline exec
- push
    push exec
- clear
    clear exec
- coroutine
    coroutine exec
```

4. 删除多余模块

默认代码里有以下模块，如果只使用其中某一模块的功能，可以根据需要删除多余的模块。

```
apps/
├── common
├── console
├── daemon
├── httpd
└── websocketd
```

Apache/PHP-FPM安装

Apache/PHP-FPM 安装

传统模式的 HTTP 开发使用该方法安装，可以部署在 Apache/PHP-FPM 下运行，该种方式不需要 Swoole 扩展，可以部署在 Windows 系统，特别适合 HTTP 开发阶段使用，有些个人用户使用的服务器无法定制环境，也可以使用该方式部署。

1. 修改应用配置

在 Apache/PHP-FPM 中部署需要修改应用配置，按下表修改相关组件的Class路径。

> 在初始代码的 http_compatible.php 文件中，我们已经帮你做了这些，你可以直接跳过这一步。

组件名	mix-httpd 部署	Apache/PHP-FPM 部署
request	mix\http\Request	mix\http\compatible\Request
response	mix\http\Response	mix\http\compatible\Response

2. 配置Apache/Nginx的Root目录

将你的Apache/Nginx的配置文件中的root目录指向：

```
// Apache
DocumentRoot /mixphp/apps/模块目录/public

// Nginx
root /mixphp/apps/模块目录/public;
```

3. URL重写

框架的路由是 `PATHINFO` 实现的，需要通过URL重写去掉URL中的 `index.php` 。

[Apache]

1. httpd.conf 配置文件中加载了mod_rewrite.so 模块
2. AllowOverride None 将 None 改为 All

[Nginx + PHP-FPM]

在 nginx.conf 中配置转发规则

```
server {
    server_name www.test.com;
    listen 80;
    root /data/mixphp/apps/httpd/public/;
    index index.php index.html index.htm;

    location / {
        if (!-e $request_filename) {
            rewrite ^/(.*)$ /index.php/$1 last;
        }
    }

    location ~ ^(.+\.php)(.*)$ {
        fastcgi_pass 127.0.0.1:9000;
        fastcgi_split_path_info ^(.+\.php)(.*)$;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_param PATH_INFO $fastcgi_path_info;
        include fastcgi_params;
    }
}
```

开发与调试

开发注意事项

开发注意事项

MixPHP 的 Web 应用是运行在 HTTP 服务器 mix-httpd 内，而 mix-httpd 基于 swoole_server 开发，其本质上就是一个持续运行的命令行应用程序，所以会有下面这些注意事项。

1. 热更新

在开发阶段我们希望编写的代码能够实时生效, 马上看到效果, 这个时候我们需要热更新功能，实现热更新我们只需要在 mix-httpd 的启动命令加上 `-u` 参数，命令如下：

```
mix-httpd service start -d -u
```

> - 该机制只能热加载应用层代码, 如果涉及 composer.json、路由、框架源码修改则需要重启服务器。
> - 需关闭 PHP 的 OPcache。
> - -u 会使 worker 只处理一次请求就销毁，所以不要在生产环境中使用。

开发阶段你也可以在 Apache/PHP-FPM 中部署，这样就能在 Windows 系统中做开发，也没有热更新问题，完成开发后再部署至 MixHttpd 即可。

2. 全局变量

有3类全局变量：

- 使用 `global` 关键词声明的变量
- 使用 `static` 关键词声明的类静态变量、函数静态变量
- PHP的超全局变量，包括 `$_GET` 、 `$_POST` 、 `$GLOBALS` 等

swoole_server 中全局变量，类静态变量当次请求结束后不会被释放，下次请求时还在，需要程序员自行处理这些变量的销毁工作，所以：

> 1. 不要有全局变量递增操作，如： Im::$msg[] = 'msg'; Im::$msg .= 'msg'; 。
> 2. 不要使用PHP提供的GET/POST，请使用框架提供的 request 组件。

3. exit/die

代码中任何位置都不能使用 `exit` 、 `die` ，使用它们会导致当前进程终止运行，所以：

> 采用 app()->end(); 来替代 exit、die。

4. Session

基于 MixHttpd 的 Web 本质上是运行在一个CLI程序中，所以PHP原生的Session是无法使用的，MixPHP为该环境下单独实现了一套Session，所以：

> 不要使用PHP提供的Session，请使用MixPHP提供的 session 组件。

调试与错误

调试

由于 MixPHP 是在命令行中启动的，所以使用 `echo` 、 `var_dump` 、 `print_r` 调试时，输出结果并不在浏览器中，而是在执行命令的终端里，无法方便的看到，所以 MixPHP 提供了一个调试方法来替代他们。

app()->dump

等同于 `var_dump` ，不同的是发送的内容不会与业务内容混合在一起，所以一定要在最后一次打印时指定第二个参数为 `true` ，才可看到响应内容。

```
// 打印变量的相关信息
app()->dump($var);

// 打印变量的相关信息并发送至客户端
app()->dump($var, true);
```

HTTP 404 / 500

HTTP `404` / `500` 两个错误是最常见的，通常我们需要定制他们，来提升用户体验。
当前 web 应用使用哪种错误类型，可在配置文件中 `error` 组件的 `format` 配置中指定。
可以有三种类型：

- mix\http\Error::FORMAT_HTML
- mix\http\Error::FORMAT_JSON
- mix\http\Error::FORMAT_XML

如何修改响应内容

`views/errors` 目录内有三个目录：

- html
- json
- xml

要修改错误响应内容，只需修改对应目录内的文件内容即可。

基础架构

目录结构

目录结构

下载最新版框架后，初始的目录结构如下：

```
├── apps
│   ├── common
│   │   ├── facades
│   │   ├── libraries
│   │   └── models
│   ├── console
│   │   ├── commands
│   │   ├── config
│   │   ├── libraries
│   │   └── runtime
│   ├── daemon
│   │   ├── commands
│   │   ├── config
│   │   ├── libraries
│   │   └── runtime
│   ├── httpd
│   │   ├── commands
│   │   ├── components
│   │   ├── config
│   │   ├── controllers
│   │   ├── libraries
│   │   ├── messages
│   │   ├── middleware
│   │   ├── models
│   │   ├── public
│   │   ├── runtime
│   │   └── views
│   └── websocketd
│       ├── commands
│       ├── config
│       ├── controllers
│       ├── libraries
│       ├── models
│       └── runtime
├── bin
│   ├── mix-console
│   ├── mix-daemon
│   ├── mix-httpd
│   └── mix-websocketd
├── composer.json
├── download.php
```

```
├── install.sh
├── LICENSE
├── README.md
└── vendor
```

> 请确保 runtime public 目录有可写权限

目录设计

MixPHP 支持多个应用，但只支持单一模块， `apps` 目录内你可以建立多个应用，应用即可以是 Web 应用，也可以是命令行应用。

> 通常一个 HTTP 应用对应一个子域名，如：api.test.com 对应 apps/api 。

增加或修改应用

`apps` 目录内的应用都可随意修改或增加，以 `HTTP` 应用为例：

1. 建立应用目录

在 `apps` 目录内建一个子目录，比如： `api` ，然后将默认应用 `httpd` 的全部文件复制过来。

2. 修改 mix-httpd 服务器的入口文件

默认代码中每个 HTTP 应用中都集成了一个 mix-httpd 服务器，入口文件在：

```
├── bin
│   ├── mix-httpd
```

重命名 `mix-httpd` 为 `mix-httpd-api` ，并修改文件中引用的配置路径为：

```
$config   = require __DIR__ . '/../apps/api/config/httpd.php';
```

3. 修改 App 配置文件

在 `api/config` 目录下找到你正在使用的 App 配置文件。

> 在 mix-httpd 服务器的配置文件 `api/config/httpd.php` 中找到 `configFile` 配置项，可以找到正在使用的配置文件。

修改 `controllerNamespace` 字段为：

```
// 控制器命名空间
'controllerNamespace' => 'apps\api\controllers',
```

4. 修改类文件的命名空间

找到所有类文件，全部修改为新的命名空间 `apps\api` 。

URL访问

URL路由原理

MixPHP 是通过 `$_SERVER['PATH_INFO']` 实现的URL路由，所以你可以这样访问。

```
http://localhost/index.php/news/article
```

开启URL重写隐藏入口文件后，可以这样访问。

```
http://localhost/news/article
```

默认的URL路由规则

MixPHP 默认只支持以下这种，URL分段无法带参数，如有需求，需要配置路由规则。

> http://localhost/控制器/操作

默认不支持以下传参方式

ThinkPHP

> http://localhost/模块/控制器/操作/[参数名1/参数值1]/[参数名2/参数值2]

CodeIgniter

> http://localhost/控制器/操作/[参数1]/[参数2]

> 更多内容请查看“HTTP 服务 -> 路由”章节。

命名空间

> 如果不清楚命名空间的基本概念，请参考 PHP命名空间。

根命名空间

框架根目录内的 `composer.json` 文件中定义了 app 的根命名空间，初始代码如下：

> key为命名空间名称，value为目录地址。

```
{
  "autoload": {
    "psr-4": {
      "apps\\": "apps/"
    }
  }
}
```

文件与命名空间对应规则

下面是一个 `Index` 控制器类，类文件路径为：

```
apps/index/controller/IndexController.php
```

代码为：

```
namespace apps\index\controller;

use mix\http\Controller;

class IndexController extends Controller
{

    public function actionIndex()
    {
        echo 'Hello World';
    }

}
```

从代码中可看出两条规则：

- namespace 等于文件所在目录的地址。

- 类名等于文件名。

> 只要符合命名空间与文件路径对应的规则，你可以建立任意名称的目录与文件。

自动加载

> 如果不清楚Composer自动加载，请参考 Composer 自动加载。
> 如果不清楚PSR-4，请参考 PSR-4 规范。

Composer

MixPHP 自动加载全部使用 `Composer` 内 `PSR-4` 规范来实现自动加载类库文件，实现了更加高效的类库自动加载机制，通常只有需修改一级目录结构，安装其他 Composer 库才需要更新自动加载，也就是说正常情况下你是不需要使用到 Composer 的。

安装 Composer

Linux

```
$> curl -sS https://getcomposer.org/installer | php
$> mv composer.phar /usr/local/bin/composer
```

Windows

```
// 下载安装
https://getcomposer.org/Composer-Setup.exe
```

composer.json 文件

框架根目录内的 `composer.json` 是 composer 的配置文件，初始代码如下：

```
{
  "name": "mixstart/mixphp",
  "description": "基于 Swoole 的常驻内存型 PHP 高性能框架 http://www.mixphp.cn",
  "type": "project",
  "keywords": [
    "framework",
    "mixphp",
    "swoole"
  ],
  "homepage": "http://www.mixphp.cn/",
  "license": "GPL-2.0",
  "authors": [
    {
      "name": "Jian Liu",
      "email": "coder.liu@qq.com"
    }
```

```
    ],
    "require": {
      "php": ">=5.4.0",
      "mixstart/framework": "1.0.*"
    },
    "autoload": {
      "psr-4": {
        "apps\\": "apps/"
      }
    }
}
```

> 如果你需要安装其他库，可以修改这个配置文件。

入口文件

入口文件

MixPHP 有两种类型的入口文件，如果需要修改目录结构，那么你需要同步修改入口文件中对应的路径。

命令行应用入口文件

MixPHP 在 CLI 模式下的命令行应用开发使用该入口文件，框架大部分入口文件为该种类型，文件路径为：

```
工程目录/bin/mix-***
```

> 命令行应用入口文件不加 .php 扩展名。

内容如下：

```
#!/usr/bin/env php
<?php

// console入口文件

require __DIR__ . '/../vendor/autoload.php';

mix\base\Env::load(__DIR__ . '/../.env');

$config   = require __DIR__ . '/../apps/httpd/config/httpd.php';
$exitCode = (new mix\console\Application($config))->run();
exit($exitCode);
```

Apache/PHP-FPM 安装入口文件

MixPHP 只有在 Apache/PHP-FPM 中部署才需使用该入口文件，该文件路径为：

```
工程目录/apps/模块目录/public/index.php
```

内容如下：

```
<?php

// web入口文件

require __DIR__ . '/../../../vendor/autoload.php';
```

入口文件

```
mix\base\Env::load(__DIR__ . '/../../../.env');

$config = require __DIR__ . '/../config/http_compatible.php';
(new mix\http\Application($config))->run();
```

框架核心

Application

Application 类

MixPHP 里 Application 类是每个应用的核心，也是组件的容器，下面简称为 App 类。

类	调用
mix\console\Application	app()
mix\http\Application	app()

开发中 App 对象有什么用

框架中所有系统类库都注册在 App 对象里，开发中到处都要使用到它。

如何使用

下面的语句就能获取到 App 对象，在框架内任何地方都可以使用。

```
// 原始调用
\Mix::app()

// 助手调用
app()
```

获取 GET 参数

```
$get = app()->request->get();
```

建立一个 session 变量

```
app()->session->set('userName', '小明');
```

获取 App 路径信息

获取应用目录路径

```
app()->basePath
```

获取运行目录路径

```
app()->getRuntimePath()
```

> 以下两个方法在 mix\console\Application 中没有。

获取公开目录路径

```
app()->getPublicPath()
```

获取视图目录路径

```
app()->getViewPath()
```

配置

环境配置

框架根目录的 `.env` 文件为环境配置文件，环境配置通常配置一些在不同环境中参数值不同的配置，如：数据库账号密码。

> 正确的打开方式：环境配置只使用在应用配置中，程序中只使用应用配置。

可配置多个环境配置文件，如： `.env.dev` 、 `.env.test` 、 `.env.pro` ，然后在入口文件中切换使用，以适应不同的环境。

在 `bin` 目录的入口文件中切换环境配置文件：

```
// 切换这里载入的文件名称即可
mix\base\Env::load(__DIR__ . '/../.env');
```

> .env 文件，需要 `ls -a` 才能在服务器上显示出来。

应用配置

MixPHP 的 App 类实例化时需要传入一些配置信息，配置信息是一个数组，这些信息就是应用配置。

通常应用配置会单独存放在一个文件中，这个文件就叫应用配置文件，一个 App 可以有多个配置文件来适应不同环境，但只能使用其中一个。

下面是一个入口文件的源码，能看到配置信息是如何导入 App 类的：

```
$config = require __DIR__ . '/../config/main.php';
(new mix\web\Application($config))->run();
```

配置详情

一个典型的应用配置文件内容如下：

```
return [

    // 基础路径
    'basePath'            => dirname(__DIR__),

    // 控制器命名空间
    'controllerNamespace' => 'apps\httpd\controllers',

    // 中间件命名空间
```

```
    'middlewareNamespace' => 'apps\httpd\middleware',

    // 全局中间件
    'middleware'          => [],

    // 组件配置
    'components'            => [
    ],

    // 类库配置
    'libraries'             => [
    ],

];
```

- 控制器命名空间需要根据实际情况而作修改

```
// HTTP应用
'controllerNamespace' => 'apps\模块名称\controllers',

// Console应用
'controllerNamespace' => 'apps\模块名称\commands',
```

- `components` 字段内是组件配置信息，详情请查看 "组件" 章节。
- `libraries` 字段内是对象配置信息，详情请查看 "对象" 章节。

自定义配置

> MixPHP 并不建议这样直接获取配置，因为有更好的方式，情请阅读 "组件"、"对象" 两个章节。

配置文件内的全部 `key` ，都将变为 App 对象的属性名称， `value` 会成为该属性的值，该方式是 MixPHP 的核心思想，可查看 "对象" 章节了解更多。

因此，我们可以这样获取到配置信息：

```
// 方法1
echo app()->basePath;
// 方法2
echo app()->config('libraries.[coroutine.pdo].dsn');
```

也就是说，你如果想增加自己的配置信息，在配置文件内增加一个新的字段即可，使用上面的方法即可获取。

但是，MixPHP 并不建议这样直接获取配置，因为：

- 现今的 PHP 开发，几乎所有需求都是采用面向对象的方式封装，所以配置信息最终都会赋值为类的属性，供

该类调用。

- 而 MixPHP 提供了 `components` 、 `libraries` 两种方式可直接面向对象传递配置，所以你不需要再增加新的字段来处理这些问题了，以上两种方式详情请阅读 "组件"、"对象" 两个章节。

对象

对象基类

MixPHP 的核心类大部分都是继承对象基类，组件全部继承对象基类，了解对象基类有助于我们更加了解框架的运行机制与设计方式。

类
mix\base\BaseObject

该类的作用

使配置的使用更加“面向对象”，下面对比一下类库封装。

ThinkPHP：

```
class Http
{

    public $baseurl

    public function __construct()
    {
        Config::load('config');
        $this->url = Config::get('config.baseurl');
        $this->init();
    }

    public function init()
    {
        // 初始化处理
    }

}

$http = new Http();
```

MixPHP：

```
class Http extends BaseObject
{

    public $baseurl

    // 当属性导入完成后，会自动执行该方法
```

```
    public function onInitialize()
    {
        // 初始化处理
    }

}

$attributes = [
    'baseurl' => '',
];
$http = new Http($attributes);
```

对比上面两种方式，在配置处理方面显然使用对象基类的方式更好一些。

更多的好处

对象基类使我们可以通过一个配置数组就可动态控制类的全部属性，使得我们可以控制整个框架的运行参数。

通过配置实例化对象

通过阅读“组件”章节，我们了解到：使用频繁的类需定义为组件，那使用不频繁的类该如何处理呢？

MixPHP 提供了统一的实例化方法：

在应用配置内的 `libraries` 字段内注册，下面是一个自定义类的注册配置：

```
// 类库配置
'libraries'             => [

    // 普通命名
    'myObject' => [
        // 类路径
        'class' => 'apps\httpd\libraries\Http',
        // 属性
        'baseurl'  => 'http://www.baidu.com',
    ],

    // 带前缀命名
    'prefix.myObject' => [
        // 类路径
        'class' => 'apps\httpd\libraries\Http',
        // 属性
        'baseurl'  => 'http://www.baidu.com',
    ],

    // 带多级前缀命名
    'prefix.prefix.myObject' => [
        // 类路径
        'class' => 'apps\httpd\libraries\Http',
```

```
        // 属性
        'baseurl'  => 'http://www.baidu.com',
    ],

],
```

- prefix.prefix.myObject 是配置名称，实例化时使用。
- class 需要实例化类的命名空间。
- 其他字段：都会在该类实例化后，导入为对象属性，key为属性名称，value为属性的值。

在框架内任何位置都可使用以下方法实例化：

```
 $http = \apps\httpd\libraries\Http::newInstanceByConfig('libraries.[prefix.prefix.myObject]');
```

>> 到 GitHub 查看默认类库配置 <<

对象基类的事件

对象基类包含了三个事件：

- onConstruct：构造事件，相当于 __construct 方法。
- onInitialize：当组件完成构造事件并导入配置信息为属性后触发该事件，用于做一些初始化处理。
- onDestruct：析构事件，相当于 __destruct 方法。

使用时只需重写这几个事件方法即可。

```
// 构造事件
public function onConstruct()
{
    parent::onConstruct();
    // ...
}

// 初始化事件
public function onInitialize()
{
    parent::onInitialize();
    // ...
}

// 析构事件
public function onDestruct()
{
    parent::onDestruct();
```

```
    // ...
}
```

第三方类库通过配置实例化 `Trait`

如果你想通过配置实例化第三方类库，由于第三方类库肯定是不会继承 `BaseObject` 的，构造、析构、处理配置信息的方式也是不会与 `BaseObject` 一样的，所以只能交由用户自行处理。

用户只需新增一个类，继承第三方类库，实现 `mix\base\StaticInstanceInterface` 接口，并引用 `mix\base\StaticInstanceTrait` 即可。

代码如下：

```
class MyLibrary extends ThirdClass implements StaticInstanceInterface
{

    use StaticInstanceTrait;

    // 如果需要接收配置，可以在这里手动处理，$config 就是传入的配置信息
    public function __construct($config = [])
    {
        parent::__construct();
    }

}
```

然后将这个类注册到类库配置 `libraries` 下即可，注册后就可通过配置实例化：

```
$object = ThirdClass::newInstanceByConfig('libraries.[***]');
```

组件

组件

组件是 MixPHP 的核心设计思想，整个框架都是由众多核心组件构成。

类
mix\base\Component

> 通过用户可自定义组件、组件常驻内存这两个特性 MixPHP 能让用户将频繁调用的业务代码也可常驻于内存，达到更高的性能。

组件的核心特征：

- 常驻内存
- 事件
- 协程隔离

> 请谨慎注册太多组件，组件应该是全局使用的，使用频繁的。

创建一个组件

你只需继承 `\mix\base\Component` 类，就可以了，下面是一个范例：

```
<?php

namespace apps\index\components;

use \mix\base\Component;

class MyComponent extends Component
{

    public $name;

    public function hello()
    {
        echo 'hello, ', $this->name;
    }

}
```

组件注册

所有组件都是在应用配置内的 `components` 字段内注册，下面是一个自定义组件的注册配置：

```
// 组件配置
'components'            => [

    // 普通命名
    'myComponent' => [
        // 类路径
        'class' => 'apps\index\components\MyComponent',
        // 属性
        'name'  => '小花',
    ],

    // 带前缀的名称
    'prefix.myComponent' => [
        // 类路径
        'class' => 'apps\index\components\MyComponent',
        // 属性
        'name'  => '小花',
    ],

     // 带多级前缀的名称
    'prefix.prefix.myComponent' => [
        // 类路径
        'class' => 'apps\index\components\MyComponent',
        // 属性
        'name'  => '小花',
    ],

],
```

- myComponent 是组件名称，调用时使用。
- class 需要实例化类的命名空间。
- 其他字段：都会在该类实例化后，导入为对象属性，key为属性名称，value为属性的值。

>> 到 GitHub 查看默认组件配置 <<

如何调用组件

在框架内任何位置都可使用，包括：控制器、模型、自定义类、第三方类。

```
// 普通命名
app()->myComponent->hello();
// 带前缀的名称
app('prefix')->myComponent->hello();
// 带多级前缀的名称
app('prefix.prefix')->myComponent->hello();
```

MixPHP 支持两种运行模式，不同模式下组件初始化的方式不同：

- mix-httpd：全部组件在服务器启动时已经加载完成。
- Apache/PHP-FPM：懒加载，只有调用 `app()->[ComponentName]` 时组件才会加载。

组件的事件

由于组件是常驻内存的，请求结束后不会销毁，而有些特殊情况下需要对请求周期内做一些初始化、数据清理方面的处理，所以 MixPHP 设计了事件机制，提供下面两个请求级别的事件函数：

- onRequestBefore：每次请求开始时触发，用于请求级别的初始化处理（代码内没有调用的组件不会触发）。
- onRequestAfter：每次请求结束时触发，用于请求结束后组件数据清理（代码内没有调用的组件不会触发）。

使用时只需在组件内重写这几个事件方法即可。

```
// 请求前置事件
public function onRequestBefore()
{
    parent::onRequestBefore();
    // ...
}

// 请求后置事件
public function onRequestAfter()
{
    parent::onRequestAfter();
    // ...
}
```

由于 `\mix\base\Component` 类继承了 `mix\base\BaseObject` 类，所以还包含三个事件：

- onConstruct：构造事件，相当于 __construct 方法。
- onInitialize：当组件完成构造事件并导入配置信息为属性后触发该事件，用于做一些初始化处理。
- onDestruct：析构事件，相当于 __destruct 方法。

使用时只需在组件内重写这几个事件方法即可。

```
// 构造事件
public function onConstruct()
{
    parent::onConstruct();
```

```
    // ...
}

// 初始化事件
public function onInitialize()
{
    parent::onInitialize();
    // ...
}

// 析构事件
public function onDestruct()
{
    parent::onDestruct();
    // ...
}
```

直接将第三方类库设置为组件 `Trait`

> 由于 Swoole 为常驻程序，所有使用全局变量的库，都有可能存在变量污染问题而无法设置为组件，特别是在协程模式。

用户只需新增一个类，继承第三方类库，实现 `mix\base\ComponentInterface` ，`mix\base\StaticInstanceInterface` 接口，并引用 `mix\base\StaticInstanceTrait` 、`mix\base\ComponentTrait` 即可。

代码如下：

```
class MyComponent extends ThirdClass implements StaticInstanceInterface, Compon
entInterface
{

    use StaticInstanceTrait, ComponentTrait;

    // 如果需要接收配置，可以在这里手动处理，$config 就是传入的配置信息
    public function __construct($config = [])
    {
        parent::__construct();
    }

}
```

然后将这个类注册到组件配置 `components` 下即可。

门面

门面 (Facade)

MixPHP 的门面具有如下功能：

- 为 组件 提供了一个静态调用接口，带来了更好的可读性与快速性，
- 同类型组件多源切换，如：多个 PDO、Redis 连接切换。

你可以为任何组件定义一个 `facade` 类。

核心门面类库

系统给内置的常用类库定义了 Facade 类库，可直接使用，包括：

组件	门面类
app()->request	mix\facades\Request
app()->response	mix\facades\Response
app()->input	mix\facades\Input
app()->output	mix\facades\Output
app()->log	mix\facades\Log
app()->error	mix\facades\Error
app()->token	mix\facades\Token
app()->session	mix\facades\Session
app()->cookie	mix\facades\Cookie
app()->pdo	mix\facades\PDO
app()->redis	mix\facades\Redis

自定义门面

为下面的组件定义一个门面：

```
// 组件配置
'components'          => [

    // 普通命名
    'myComponent' => [
        // 类路径
        'class' => 'apps\index\components\MyComponent',
        // 属性
        'name'  => '小花',
```

```
    ],

],
```

然后在 `apps\httpd\facades` 目录新增一个 `myComponent` 类文件。

```php
<?php

namespace apps\httpd\facades;

use mix\base\Facade;

class myComponent extends Facade
{

    // 获取实例
    public static function getInstance()
    {
        return app()->myComponent;
    }

}
```

门面使用

原本组件的调用：

```php
app()->myComponent->foo();
```

门面类的调用，是不是简单很多。

```php
myComponent::foo();
```

代码补全

自己创建的门面类是没有代码补全的，需要用户自己在注释中添加。

例如：

```
 * @method emergency($message, array $context = []) static
 * @method alert($message, array $context = []) static
 * @method critical($message, array $context = []) static
 * @method error($message, array $context = []) static
```

>> 到 GitHub 查看门面注释 DEMO <<

同类型组件多源切换

系统提供的如下 Facade 类库具有多源切换功能：

门面类
mix\facades\PDO
mix\facades\Redis

我们先看一下源码，看看于普通的有何不同：

>> 到 GitHub 查看多源切换门面 <<

- `getInstances` 方法返回一个具有多个组件的数组。
- `name` 方法可以通过名称切换当前使用的组件，这样就达到了切换数据库的效果。

当然我们实际使用中一定不可能去修改这个核心门面，只需写一个新的门面继承 `mix\facades\PDO` 类，并重写 `getInstances` 方法即可。

框架默认代码已经帮你继承好了，如下：

>> 到 GitHub 查看门面 DEMO <<

当需要频繁使用多个连接时，只需增加一个 pdo 组件：

```
// 数据库
'pdo'      => [
    // 类路径
    'class'     => 'mix\client\PDO',
    // 数据源格式
    'dsn'       => env('DB_DSN'),
    // 数据库用户名
    'username'  => env('DB_USERNAME'),
    // 数据库密码
    'password'  => env('DB_PASSWORD'),
    // 设置PDO属性: http://php.net/manual/zh/pdo.setattribute.php
    'attribute' => [
        // 设置默认的提取模式: \PDO::FETCH_OBJ | \PDO::FETCH_ASSOC
        \PDO::ATTR_DEFAULT_FETCH_MODE => \PDO::FETCH_ASSOC,
    ],
],

// 数据库
'db1.pdo'      => [
    // 类路径
    'class'     => 'mix\client\PDO',
    // 数据源格式
    'dsn'       => env('DB_DSN'),
```

```
    // 数据库用户名
    'username'  => env('DB_USERNAME'),
    // 数据库密码
    'password'  => env('DB_PASSWORD'),
    // 设置PDO属性: http://php.net/manual/zh/pdo.setattribute.php
    'attribute' => [
        // 设置默认的提取模式: \PDO::FETCH_OBJ | \PDO::FETCH_ASSOC
        \PDO::ATTR_DEFAULT_FETCH_MODE => \PDO::FETCH_ASSOC,
    ],
],
```

然后修改 `apps/common/facades/PDO` 类代码如下：

```
/**
 * 获取实例集合
 * @return array
 */
public static function getInstances()
{
    return [
        'default' => app()->pdo,
        'db1'     => app('db1')->pdo,
    ];
}
```

使用：

```
// 使用 default
/apps/common/facades/PDO::createCommand($sql)->queryAll();
// 使用 db1
/apps/common/facades/PDO::name('db1')->createCommand($sql)->queryAll();
```

系统服务

中间件

中间件

中间件主要用于拦截或过滤应用的 HTTP 请求，并进行必要的业务处理，通常使用在登录验证的场景。

定义中间件

源码中默认自带了两个中间件，一个前置，一个后置。

> - 中间件类名必须带 Middleware 后缀。
> - 配置文件 main.php 中可定义中间件目录的命名空间。

前置中间件

如果想拦截请求不往下执行，只需在 $next() 前 return 响应内容即可。

```php
<?php

namespace apps\index\middleware;

/**
 * 前置中间件
 * @author 刘健
 */
class BeforeMiddleware
{

    public function handle($callable, \Closure $next)
    {
        // 获取控制器与方法名称，可做细粒度的权限控制
        list($controller, $actionName) = $callable;
        $controllerName = get_class($controller);

        // 添加中间件执行代码
        // ...

        // 执行下一个中间件
        return $next();
    }

}
```

后置中间件

```php
<?php

namespace apps\index\middleware;

/**
 * 后置中间件
 * @author 刘健
 */
class AfterMiddleware
{

    public function handle($callable, \Closure $next)
    {
        // 获取后面全部中间件执行后的响应结果
        $response = $next();

        // 添加中间件执行代码
        // ...

        // 返回响应内容
        return $response;
    }

}
```

注册中间件

全局中间件

配置文件 `main.php` 中的 `middleware` 配置项目可配置全局中间件，全局中间件是全局有效的，对任何路由都有效。

```
// 全局中间件
'middleware'          => ['After'],
```

> 配置时不需要加 Middleware 后缀。

路由中间件

我们也可以在路由中为某一个路由规则配置要执行的中件间，如下：

```
// 路由规则
'rules'          => [

    // 一级路由
    ':controller/:action' => [':controller', ':action', 'middleware' => ['Befor
```

```
e']],

],
```

如果我们需要排除某些路由规则不使用中间件

只需要在 带通配符规则的前面 增加一条不带 middleware 配置的路由。

```
// 路由规则
'rules'           => [

    // 首页不使用中间件
    '' => ['Index', 'Index'],

    // 一级路由中URL /profile/userinfo 不使用中间件
    'profile/userinfo' => ['Profile', 'Userinfo'],

    // 一级路由
    ':controller/:action' => [':controller', ':action', 'middleware' => ['Befor
e']],

],
```

验证器

验证器定义

验证器

MixPHP 的验证器结合了多个框架的优点，如下：

- 支持场景控制。
- 验证成功后字段将赋值为验证类的属性，文件则直接实例化为文件对象。
- 更细粒度的错误消息设置。
- 支持在 WebSocket 开发中使用。

验证器定义

我们定义一个 `\apps\index\models\UserForm` 验证器类用于 `User` 控制器的验证。

```
<?php

namespace apps\index\models;

use mix\validators\Validator;

class UserForm extends Validator
{

    public $name;
    public $age;
    public $email;

    // 规则
    public function rules()
    {
        return [
            'name'  => ['string', 'maxLength' => 25, 'filter' => ['trim']],
            'age'   => ['integer', 'unsigned' => true, 'min' => 1, 'max' => 120]
,
            'email' => ['email'],
        ];
    }

    // 场景
    public function scenarios()
    {
        return [
            'create' => ['required' => ['name'], 'optional' => ['email', 'age']]
,
        ];
```

```
    }

    // 消息
    public function messages()
    {
        return [
            'name.required'  => '名称不能为空.',
            'name.maxLength' => '名称最多不能超过25个字符.',
            'age.integer'    => '年龄必须是数字.',
            'age.unsigned'   => '年龄不能为负数.',
            'age.min'        => '年龄不能小于1.',
            'age.max'        => '年龄不能大于120.',
            'email'          => '邮箱格式错误.',
        ];
    }

}
```

> 如果没有定义错误提示信息，则使用系统默认的提示信息

数据验证

在需要进行 `User` 验证的控制器方法中，添加如下代码即可：

```
<?php

namespace apps\index\controllers;

use apps\index\models\UserForm;
use mix\facades\Request;
use mix\http\Controller;

class UserController extends Controller
{

    public function actionCreate()
    {
        app()->response->format = \mix\http\Response::FORMAT_JSON;

        // 使用模型
        $model             = new UserForm();
        $model->attributes = Request::get() + Request::post();
        $model->setScenario('create');
        if (!$model->validate()) {
            return ['code' => 1, 'message' => 'FAILED', 'data' => $model->getEr
rors()];
        }
```

```
        // 执行保存数据库
        // ...

        // 响应
        return ['code' => 0, 'message' => 'OK'];
    }

}
```

验证失败

验证失败可以通过以下方法获取错误消息：

- $model->getErrors() ：获取全部错误信息，返回数组。
- $model->getError() ：获取单条错误信息，返回字符串。

验证成功

验证成功后，验证规则中通过验证的字段，将会赋值到同名的验证类的属性中，未通过的字段则为 `null` 。

这个功能有什么用？

> 有了这个功能，我们就只需要把真个验证类的对象传入模型，就可以在模型里安全的使用这些属性操作数据库。

验证规则

验证规则

全部的验证类型与对应的验证选项如下，大部分能根据语义理解，特殊的几个下文会单独说明。

```
// 规则
public function rules()
{
    return [
        'a' => ['integer', 'unsigned' => true, 'min' => 1, 'max' => 1000000, 'l
ength' => 10, 'minLength' => 3, 'maxLength' => 5],
        'b' => ['double', 'unsigned' => true, 'min' => 1, 'max' => 1000000, 'le
ngth' => 10, 'minLength' => 3, 'maxLength' => 5],
        'c' => ['alpha', 'length' => 10, 'minLength' => 3, 'maxLength' => 5],
        'd' => ['alphaNumeric', 'length' => 10, 'minLength' => 3, 'maxLength' =>
 5],
        'e' => ['string', 'length' => 10, 'minLength' => 3, 'maxLength' => 5, '
filter' => ['trim', 'strip_tags', 'htmlspecialchars']],
        'f' => ['email', 'length' => 10, 'minLength' => 3, 'maxLength' => 5],
        'g' => ['phone'],
        'h' => ['url', 'length' => 10, 'minLength' => 3, 'maxLength' => 5],
        'i' => ['in', 'range' => ['A', 'B'], 'strict' => true],
        'j' => ['date', 'format' => 'Y-m-d'],
        'k' => ['compare', 'compareAttribute' => 'a'],
        'l' => ['match', 'pattern' => '/^[\w]{1,30}$/'],
        'm' => ['call', 'callback' => [$this, 'check']],
        'n' => ['file', 'mimes' => ['audio/mp3', 'video/mp4'], 'maxSize' => 1024
 * 1],
        'r' => ['image', 'mimes' => ['image/gif', 'image/jpeg', 'image/png'], '
maxSize' => 1024 * 1],
    ];
}
```

`call` 验证类型

该类型为用户自定义验证规则， `callback` 内指定一个用户自定义的方法来验证。

自定义的方法如下：

```
// 自定义验证
public function check($fieldValue)
{
    // 验证代码
    // ...
    // 返回结果
```

```
    return true;
}
```

> 返回 true 通过验证，false 返回错误。

`file` / `image` 验证类型

该类型用来验证文件，包含的两个验证选项如下：

- mimes：输入你想要限制的文件mime类型，MIME参考手册 。
- maxSize：允许的文件最大尺寸，单位 KB。

验证成功后模型类会增加一个同名属性，该属性为 `mix\http\UploadFile` 类的实例化对象，在模型内可直接调用 `$this->[attributeName]->saveAs($filename)` 另存为你需要存放的位置，更多方法请查看 "文件上传" 章节。

静态调用

静态调用

虽然我非常反对该种使用方式，但最后我还是提供静态调用的支持。

直接使用 `mix\validators\Validate` 类静态调用：

```
// 验证是否为字母与数字
Validate::isAlphaNumeric($value); // true

// 验证是否为字母
Validate::isAlpha($value); // true

// 验证是否为日期
Validate::isDate($value, $format); // true

// 验证是否为浮点数
Validate::isDouble($value); // true

// 验证是否为邮箱
Validate::isEmail($value); // true

// 验证是否为整数
Validate::isInteger($value); // true

// 验证是否在某个范围
Validate::in($value, $range, $strict = false); // true

// 正则验证
Validate::match($value, $pattern); // true

// 验证是否为手机
Validate::isPhone($value); // true

// 验证是否为网址
Validate::isUrl($value); // true
```

模型

模型

MixPHP 并没有对模型做封装，也就是说用户创建模型类时，不需要继承 `model` 类，只需通过 `PDO` 组件在模型类直接操作数据库即可。

模型分层

为避免出现 "胖模型"，我们建议将模型分层处理，比如这样分层：

- 表单模型：一个控制器配套一个表单模型，表单模型继承验证器类，操作数据时调用数据模型。
- 数据模型：一个数据库表配套一个数据模型，通过数据库组件直接操作数据库。

数据库操作

数据库操作请阅读 “同步客户端” 章节。

日志

> 该组件为系统组件，在组件树中只可命名为 log ，不可修改为其他名称。

日志

日志组件通常使用在开发环境中用来debug，生产环境中监控程序异常与运行状态。

类	调用
mix\base\Log	app()->log

门面类	调用
mix\facades\Log	Log::

数据库配置

App配置文件中，该组件的默认配置如下：

```
// 日志
'log'      => [
    // 类路径
    'class'       => 'mix\base\Log',
    // 日志记录级别
    'level'       => ['error', 'info', 'debug'],
    // 日志目录
    'logDir'      => 'logs',
    // 日志轮转类型
    'logRotate'   => mix\base\Log::ROTATE_DAY,
    // 最大文件尺寸
    'maxFileSize' => 0,
    // 换行符
    'newline'     => PHP_EOL,
    // 在写入时加独占锁
    'writeLock'   => false,
],
```

`logRotate` 全部常量明细：

- mix\base\Log::ROTATE_HOUR
- mix\base\Log::ROTATE_DAY
- mix\base\Log::ROTATE_WEEKLY

日志类型

全部日志类型如下：

- debug：调试日志
- info：信息日志
- error：错误日志

日志文件

生成的日志文件默认在 `runtime/logs` 目录，也可以使用绝对路径定义为其他目录，日志文件格式如下：

```
文件前缀_轮转时间_[自增编号].log
```

自定义日志

可以自定义输出日志到某个文件， `$message` 只能为字符型，且没有时间信息，需要用户自己封装将 array/object 转换为 json 字符型，然后增加时间信息。

```
Log::write($filePrefix, $message)
```

调试日志

```
Log::debug($message);
```

信息日志

```
Log::info($message);
```

写入错误日志

错误日志是被动的，当应用运行出现异常时框架自动记录。

```
Log::error($message);
```

日志记录级别

配置中的 `level` 字段设定了写入日志的级别，没有定义在里面的日志类型，不管是被动调用还是主动掉用，都不会写入到日志文件。

> 日志记录级别对 write 方法无效。

```
'level'       => ['error', 'info', 'debug'],
```

例如：只记录错误日志，忽略其他类型的日志，配置如下：

```
'level'       => ['error'],
```

命令行

简介

命令行

MixPHP 的全部程序都是命令行应用，基于 Swoole 的特性，MixPHP 可以完成更后端的开发需求，本章内容主要描述如何开发命令行应用。

类	调用	运行环境
mix\console\Application	app()	CLI

应用场景

任务处理类开发：

- 定时任务，如：清理数据、统计数据等。
- 守护进程，如：消息队列(MQ)消费处理，消息推送，数据采集等。

服务类开发：

- HTTP服务，如：mix-httpd
- WebSocket服务，如： 消息推送、在线聊天、直播弹幕、棋牌游戏等。
- 多进程任务服务，如：消息队列(MQ)消费处理，消息推送，数据采集等。

命令行开发常识

命令行开发常识

由于 CLI 开发，并不像 HTTP 服务开发一样在 PHPer 中如此普及，所以该章节针对命令行开发的一些常识，整理说明一下：

> - 命令行应用全部是常驻内存的。
> - 命令行应用中开启的 Mysql / Redis 等连接只要不主动去关闭该连接，那都是长连接。

命令行开发的分类

命令行开发的分类并不像 HTTP 服务一样清晰，但 MixPHP 依然根据需求特征进行了分类：

- 控制台程序，如：清理数据、统计数据、数据转换等。
- 守护程序，如：消息队列消费处理，数据入库，消息推送，数据爬取等。

各分类在 MixPHP 中对应的开发目录

各类型在框架中对应的目录如下：

```
apps/
├── console
├── daemon
```

命令行开发注意事项

数据库连接处理

控制台程序：
控制台程序是一次性执行，执行完就结束，所以不需要考虑数据库连接超时问题，数据库连接采用短连接即可。
守护程序：
守护程序实际上就是个一直在持续执行的控制台程序，所以需要考虑数据库连接超时问题，传统 MVC 框架，需要手动使用 try/catch 来重启循环流程，并手动重建数据库连接，而 MixPHP 不需要，你只需要使用长连接版本的数据库客户端即可，MixPHP 会帮你重建连接，你什么都不需要处理。

蜕变为守护进程

PHP 在 CLI 模式下执行，正常是无法蜕变为一个守护进程的，通常传统 PHP 开发是使用以下方法脱离终端在后台运行。

```
php command.php >/dev/null 2>&1 &
```

MixPHP 提供了更方便的方法，只需要控制器代码第一行，增加：

```
public function actionIndex()
{
    // 蜕变为守护进程
    ProcessHelper::daemon();
    // ...
    return ExitCode::OK;
}
```

> 该方法依赖 Swoole 扩展，Windows下无法执行。

命令行应用如何操作数据库

命令行程序由于数据逻辑的复杂性，实际上是不适合 `MVC` 分层开发的，所以我们 不推荐以组件的形式在命令行使用数据库客户端，我们推荐在命令行开发中直接在 `Command` 类中使用 `Class::newInstanceByConfig('libraries.[***]')` 通过配置实例化数据库客户端操作数据库。

创建命令

创建命令

命令就是一个命令行程序， `Command` 类似于 HTTP 应用的 `Controller` 控制器，负责业务逻辑，不同的是命令行程序通常是处理复杂的数据处理逻辑，相比简单的 `CRUD` 操作要复杂很多。

类
mix\console\Command

> 初始代码中命令行应用的命令在 commands 目录。

一个简单的命令

首先在配置文件中增加命令。

```
// 命令
'commands'          => [

    'clear exec' => ['Clear', 'Exec'],

],
```

详解命令规则：

- 命令： `clear exec`
- 命令类的类名，不包括 Command 后缀： `Clear`
- 命令类的方法名，不包括 action 前缀： `Exec`

创建命令类，代码如下：

```
<?php

namespace apps\console\commands;

use mix\console\Command;
use mix\console\ExitCode;
use mix\facades\Output;

/**
 * Clear 命令
 * @author 刘健
 */
```

```
class ClearCommand extends Command
{

    // 执行任务
    public function actionExec()
    {
        // 响应
        Output::writeln('SUCCESS');
        // 返回退出码
        return ExitCode::OK;
    }

}
```

命名空间与文件位置的关系

控制器定义的命名空间为：

```
namespace apps\console\commands;
```

因为根命名空间 `apps` 在 `composer.json` 内定义的路径为 ：

```
"apps\\": "apps/"
```

所以控制器的完整路径为：

```
apps/console/commands/ClearCommand.php
```

命令行执行

执行上面写的命令。

```
mix-console clear exec
```

如何使用命令选项

修改上面的命令类，代码如下：

```
<?php

namespace apps\console\commands;

use mix\console\Command;
```

```
use mix\console\ExitCode;
use mix\facades\Input;
use mix\facades\Output;
use mix\helpers\ProcessHelper;

/**
 * Clear 命令
 * @author 刘健
 */
class ClearCommand extends Command
{

    // 是否后台运行
    public $daemon = false;

    // 选项配置
    public function options()
    {
        return ['daemon'];
    }

    // 选项别名配置
    public function optionAliases()
    {
        return ['d' => 'daemon'];
    }

    // 执行任务
    public function actionExec()
    {
        // 蜕变为守护进程
        if ($this->daemon) {
            ProcessHelper::daemon();
        }
        // 修改进程名称
        Process::setName('mix-crontab: ' . Input::getCommandName());

        // 响应
        Output::writeln('SUCCESS');
        // 返回退出码
        return ExitCode::OK;
    }

}
```

- options 方法定义了一个 daemon 选项，所以命令只会接收 `--daemon` 选项。
- optionAliases 方法给 daemon 选项定义了一个别名，所以 `-d` 等效于 `--daemon` 。

执行上面写的命令，并传入选项：

```
mix-console clear exec -d
```

执行与选项

应用的执行

命令行应用是在 shell 中执行，命令格式如下：

> php [入口文件] [命令] [选项]

```
php mix-httpd service start -d
```

以上命令的各部分拆解如下：

- 入口文件： `mix-httpd`
- 命令： `service start`
- 选项： `-d`

入口文件

当你的 `php` 加入环境变量时，可以这样执行你的入口文件：

```
./mix-httpd service start -d
```

当执行完 `install.sh` 后，入口文件可在任意位置执行，如下：

```
mix-httpd service start -d
```

选项参数规则

- 参数必须使用 "一个或两个中杠" 开头，否则会被丢弃，
- 参数支持一个中杠、二个中杠，如： `-option1` 、 `--option2` 。
- 参数可以有值、也可以没有值，如： `--option3=value` 、 `--option2` 。

一个完整的执行范例

下面演示一个带参数的 Console 应用的执行。

```
mix-crontab order timeout --start --time-range=30
```

命令行选项

命令行选项会根据命令的 `options` 方法定义的值传递至控制器内，成为控制器的属性，属性的传递规则如下：

- 参数名称成为控制器的属性名称，如： `--option3=value` ，变为 `$this->option3` 。
- 没有值的参数，如： `-option1` 、 `--option2` ，框架默认赋值为 `true` 。
- 有值的参数，如： `--option3=value` ，赋值为等号后的值。

控制台程序

控制台程序

控制台程序是一次性执行，执行完就结束的任务处理类 CLI 程序。

使用场景

如：清理数据、统计数据、数据转换等。

> 可使用 linux 的 crontab 等工具定时触发命令。

开发目录

```
apps/
├── console
```

命令执行

```
mix-console [入口文件] [命令] [选项]
```

范例代码

>> 到 GitHub 查看 DEMO <<

一次性执行的命令行程序是最简单的，当代码中 `actionExec` 内的代码执行结束后，进程就会退出。

命令管理

在命令行使用以下命令管理：

```
// 查看帮助
mix-console -h

// 执行命令
mix-console clear exec
```

也可使用如下 Linux 命令手动管理进程。

```
// 查找进程
ps -ef | grep mix-console

// 结束进程
```

```
kill <PID>
```

守护程序

守护程序

守护程序实际上就是个一直在持续执行的控制台程序，正常情况下，代码执行完该进程就会结束，而守护进程就是使用一个循环使该进程一直处于工作状态，PHP 在 CLI 下启动的程序，默认是单进程单线程的，MixPHP 可以实现多进程编程，请查看“多进程”章节。

使用场景

单进程：

如：WebSocket 的广播推送服务等。

多进程：

如：消息队列(MQ)消费处理，消息推送，数据采集等。

开发目录

```
apps/
├── daemon
```

命令执行

```
mix-daemon [入口文件] [命令] [选项]
```

范例代码

>> 到 GitHub 查看 DEMO <<

同步单进程守护程序，其实都是通过一个循环让代码一直保持执行状态，从而达到守护执行的目的，通常由于一些未知异常，会导致程序报错而引起守护程序退出，DEMO 中 `startWork` 方法实现了捕获错误、重建执行流程，能让该程序在任何错误下仍可以继续执行，达到守护的目的。

命令管理

在命令行使用以下命令管理：

```
// 查看帮助
mix-daemon -h

// 启动
mix-daemon single start
```

```
// 启动（守护）
mix-daemon single start -d

// 停止
mix-daemon single stop

// 重启
mix-daemon single restart

// 状态
mix-daemon single status
```

也可使用如下 Linux 命令手动管理进程。

```
// 查找进程
ps -ef | grep mix-daemon

// 结束进程
kill <PID>
```

HTTP 服务

简介

HTTP 服务

本章内容是 MixPHP 的 HTTP 服务开发，MixPHP 底层对 Swoole 做了大量兼容性处理，让用户可以像使用传统 MVC 框架一样使用 Swoole 开发高性能 HTTP 服务，降低了使用门槛。

类	调用
mix\http\Application	app()

运行模式

为了让用户能低学习成本的使用 MixPHP，我们为 `HTTP` 开发提供了技术难度递进的三种执行模式：

- 传统模式：与传统 Apache/PHP-FPM 环境执行的框架一样，但更加轻量化，通常只用来开发。
- 常驻同步模式 (默认)：常驻内存带来传统框架无法比拟的高性能，同时对团队技术要求不会太高。
- 常驻协程模式：除了具有常驻内存的优势，协程带来的并行优势让总体并发性能提升N倍，适合技术能力较强的团队使用。

> 三种模式的切换，几乎全部代码可以无缝迁移，只需修改配置与少量类名即可。

应用场景

HTTP API

- `HTTP` 接口开发。

WebSite

- 网站开发。
- 后台管理开发。

服务器

> 该服务依赖 Swoole 扩展，Windows下无法执行。

`mix-httpd` 服务器

`mix-httpd` 是官方开发的 HTTP 服务器，用于执行 "HTTP 服务"，基于 Swoole 扩展的 `swoole_http_server` ，拥有比 Apache/PHP-FPM 更高的性能。

> mix-httpd 其实就是一个使用 MixPHP 开发的一个命令行应用程序。

位置

默认代码 `httpd` 模块中有集成 `mix-httpd` 服务器，代码路径为：

```
apps/httpd/commands/ServiceCommand.php
```

入口文件为：

```
bin/mix-httpd
```

配置文件

mix-httpd 的配置文件路径为：

```
apps/httpd/config/httpd.php
```

只需要配置文件内的 `httpServer` 字段内的参数即可，如下：

`host` 参数

设置服务器绑定的主机。

`port` 参数

设置服务器绑定的端口。

`configFile` 参数

执行在 `mix-httpd` 服务器内的 HTTP 应用的配置文件。

`settings` 参数

正式环境需设置合适的运行参数，与性能相关，非常重要。

全部参数列表：https://wiki.swoole.com/wiki/page/274.html

命令管理

mix-httpd 集成了一些命令，让用户能很方便的操作服务器。

> 请使用 root 账号启动 mix-httpd。

全部命令如下：

- service start : 启动服务器。
- service stop : 停止服务器。
- service restart : 重启服务器。
- service reload : 重启所有工作进程，用于刷新代码。

`service start` 命令有两个很好用的参数：

- -d : 后台运行 (为了方便错误调试，不建议在开发阶段使用该参数)。
- -u : 代码热更新 (开发阶段使用)，需关闭 PHP 的 OPcache，该参数会使 worker 进程只处理一次请求就销毁，所以不要在生产环境中使用。

在命令行使用以下命令管理：

```
// 查看帮助
mix-httpd -h

// 启动
mix-httpd service start

// 启动（守护）
mix-httpd service start -d

// 启动（守护 + 热更新）
mix-httpd service start -d -u

// 停止
mix-httpd service stop

// 重启
mix-httpd service restart

// 重启工作进程
mix-httpd service reload
```

```
// 状态
mix-httpd service status
```

路由

> 该组件为系统组件，在组件树中只可命名为 route ，不可修改为其他名称。

路由

路由是 MixPHP 的核心组件之一，秉承极简理念，底层使用正则构建，好用又简单，默认配置了控制器一级目录的访问规则，控制器多级目录需增加路由规则。

类	调用
mix\http\Route	app()->route

路由配置

在App配置文件中，关于路由组件的默认配置如下：

> 通常你不需要修改配置就能完成大部分的开发任务。

>> 到 GitHub 查看默认配置 <<

路由规则

路由规则在 rules 字段内定义，例如：

```
// 路由规则
'rules'           => [
    // 二级路由
    'api/:controller/:action' => ['api/:controller', ':action'],
],
```

上面定义了一个 API 接口的规则，匹配的URL与指向的功能如下：

URL	控制器::方法
http://site.com/api/user	controller\api\UserController::actionIndex
http://site.com/api/user/setting	controller\api\UserController::actionSetting
http://site.com/api/user_info/setting_profile	controller\api\UserInfoController::actionSettingProfile
http://site.com/api/user-info/setting-profile	controller\api\UserInfoController::actionSettingProfile

路由规则还支持HTTP请求方法匹配：

```
// 路由规则
'rules'          => [
    // 只有GET或POST才可访问
    'GET|POST api/:controller/:action' => ['api/:controller', ':action'],
],
```

框架支持的全部请求方法如下：

```
CLI|GET|POST|PUT|PATCH|DELETE|OPTIONS|HEAD|TRACE
```

通过HTTP请求方法匹配，能够很简单的构建出 `RESTful` 风格。

```
// 路由规则
'rules'          => [
    'GET api/:controller'               => ['api/:controller', 'Index'],
    'POST api/:controller'              => ['api/:controller', 'Save'],
    'GET api/:controller/:id'           => ['api/:controller', 'Read'],
    'PUT api/:controller/:id'           => ['api/:controller', 'Update'],
    'DELETE api/:controller/:id'        => ['api/:controller', 'Delete'],
],
```

默认路由规则

如果你没有定义任何路由规则，框架会默认定义下面的通用路由规则：

```
// 一级路由
':controller/:action' => [':controller', ':action'],
```

> 所以你什么都不定义就可以访问 首页 与 一级目录的控制器。

通用多级路由配置

从 `v1.1.1` 开始可定义任意级通用路由，以下代码定义了三级通用路由：

```
// 一级路由
':controller/:action' => [':controller', ':action', 'middleware' => ['Before']],

// 二级路由
':second/:controller/:action' => [':second/:controller', ':action', 'middleware'
 => ['Before']],
// 三级路由
':three/:second/:controller/:action' => [':three/:second/:controller', ':action'
```

```
, 'middleware' => ['Before']],
```

路由变量

上一节中 `:controller` `:action` 就是路由变量，但是这两个变量是特殊变量，是专用于指向控制器与方法的，其他名称的变量为普通变量。

下面演示一下普通变量的使用：

```
// 路由规则
'rules'          => [
    'news/article/:id' => ['News', 'Article'],
],
```

匹配的URL与指向的功能如下：

URL	控制器::方法
http://site.com/news/article/548762154	controller\NewsController::actionArticle

上面定义的普通变量并没有在规则中使用，而是需要在控制器代码中使用，代码中可以这样获取变量值：

```
// 获取全部路由变量
app()->request->route();
// 获取单个路由变量
app()->request->route('id');
```

路由变量规则

路由变量也是可以定义规则的，规则是正则表达式，在 `patterns` 字段内定义。

> 定义了变量规则后，当变量所在URL段不符合规则时，框架会抛出404错误。

默认变量规则

如果你没有为变量定义规则，默认为 `defaultPattern` 字段内定义的规则，如果你连 `defaultPattern` 也没有定义，则默认为 `[\w-]+` 。

请求

> 该组件为系统组件，在组件树中只可命名为 request ，不可修改为其他名称。

请求

请求组件用来获取所有HTTP请求参数。

类	调用	运行环境
mix\http\Request	app()->request	mix-httpd
mix\http\compatible\Request	app()->request	Apache/PHP-FPM

门面类	调用
mix\facades\Request	Request::

组件配置

App配置文件中，该组件的默认配置如下：

> 由于该类没有使用到其他参数，所以只有一个class字段。

```
// 请求
'request'  => [
    // 类路径
    'class' => 'mix\http\Request',
],
```

获取参数

方法	描述
route	获取路由参数
get	获取 $_GET 参数
post	获取 $_POST 参数
files	获取 $_FILES 参数
server	获取 $_SERVER 参数 (全部小写)
header	获取 HEADER 参数 (全部小写)
getRawBody	返回原始的 HTTP 包体

> 以上所有方法变量名不存在时返回 null。

请求类型

方法	描述
method	返回请求类型
isGet	是否为 GET 请求
isPost	是否为 POST 请求
isPut	是否为 PUT 请求
isPatch	是否为 PATCH 请求
isDelete	是否为 DELETE 请求
isHead	是否为 HEAD 请求
isOptions	是否为 OPTIONS 请求

请求路径

方法	描述
root	返回请求的域名
path	返回请求的路径
url	返回请求的URL
fullUrl	返回请求的完整URL

获取路由参数

```
// 获取单个参数
Request::route('name');

// 获取所有参数，返回数组
Request::route();
```

获取 `GET` 参数

```
// 获取单个参数
Request::get('name');

// 获取所有参数，返回数组
Request::get();
```

获取 `POST` 参数

```
// 获取单个参数
Request::post('name');

// 获取所有参数，返回数组
Request::post();
```

获取 `FILES` 参数

```
// 获取单个参数
Request::files('name');

// 获取所有参数，返回数组
Request::files();
```

获取 `SERVER` 参数

```
// 获取单个参数
Request::server('name');

// 获取所有参数，返回数组
Request::server();
```

获取 `HEADER` 参数

```
// 获取单个参数
Request::header('name');

// 获取所有参数，返回数组
Request::header();
```

返回原始的 `HTTP` 包体

```
Request::getRawBody();
```

返回请求路径

```
Request::root(); // http//www.domain.com
Request::path(); // index/index.html
Request::url(); // http//www.domain.com/index/index.html
Request::fullUrl(); // http//www.domain.com/index/index.html?s=hello
```

响应

> 该组件为系统组件，在组件树中只可命名为 response ，不可修改为其他名称。

响应

响应组件用来将控制器返回的数据、设置的HTTP报头发送至客户端。

类	调用	运行环境
mix\http\Response	app()->response	mix-httpd
mix\http\compatible\Response	app()->response	Apache/PHP-FPM

门面类	调用
mix\facades\Response	Response::

组件配置

App配置文件中，该组件的默认配置如下：

>> 到 GitHub 查看默认配置 <<

参数 `defaultFormat` 全部常量明细：

- mix\http\Response::FORMAT_HTML
- mix\http\Response::FORMAT_JSON
- mix\http\Response::FORMAT_JSONP
- mix\http\Response::FORMAT_XML

设置响应格式

当开发API接口时，通常需要响应 `JSON` 、 `JSONP` 、 `XML` 格式，这时可在控制中指定响应格式，代码如下：

```
public function actionIndex()
{
    app()->response->format = \mix\http\Response::FORMAT_JSON;
    return ['errcode' => 0, 'errmsg' => 'ok'];
}
```

也可以在App配置文件中的 `defaultFormat` 字段中定义默认的响应格式：

```
// 默认输出格式
'defaultFormat' => mix\http\Response::FORMAT_JSON,
```

重定向

重定向到首页。

```
Response::redirect('/');
```

设置 `HTTP` 状态码

设置响应的HTTP状态码为404。

```
app()->response->statusCode = 404;
```

设置 `HTTP` 报头

设置报头Content-Type为json格式utf8编码。

```
Response::setHeader('Content-Type', 'application/json;charset=utf-8');
```

控制器

控制器

控制器是应用程序中处理用户交互的部分，通常控制器负责读取请求数据，与模型交换数据，渲染视图并发送数据。

类
mix\http\Controller

一个简单的控制器

新建一个文件 IndexController.php ， 然后放入以下代码:

```
<?php

namespace apps\index\controllers;

use mix\http\Controller;

class IndexController extends Controller
{

    public function actionIndex()
    {
        return 'Hello World!';
    }

}
```

命名空间与文件位置的关系

控制器定义的命名空间为：

```
namespace apps\index\controller;
```

因为根命名空间 `apps` 在 `composer.json` 内定义的路径为：

```
"apps\\": "apps/"
```

所以控制器的完整路径为：

```
apps/index/controller/IndexController.php
```

URL访问控制器

MixPHP 默认定义了首页与一级目录的默认路由规则，所以上面的控制器可以这样访问：

```
http://site.com/index/index
```

第一段 `index` 指向 `IndexController` 类
第二段 `index` 指向 `actionIndex` 方法

首页控制器

首页控制器就是当URL中没有指定控制器名称时默认访问的控制器， `IndexController` 为MixPHP的首页控制器。
当访问下面的URL时：

```
http://site.com
```

默认访问:

```
apps/index/controller/IndexController.php
```

默认方法

默认方法就是当URL中没有指定方法名称时默认访问的方法， `actionIndex` 为MixPHP的默认方法。
当访问下面的URL时：

```
http://site.com/index
```

默认访问：

```
apps/index/controller/IndexController::actionIndex
```

视图

视图

简单来说，一个视图其实就是一个 Web 页面，或者页面的一部分，像页头、页脚、侧边栏等，MixPHP的视图支持布局。

类
mix\http\View

创建一个视图

下面演示为控制器 `ProfileController` 创建一个视图，控制器代码如下：

```
<?php

namespace apps\index\controllers;

use mix\http\Controller;

class ProfileController extends Controller
{

    public $layout = 'main';

    public function actionIndex()
    {
        $data = [
            'name'    => '小明',
            'age'     => 18,
            'friends' => ['小红', '小花', '小飞'],
        ];
        return $this->render('index', $data);
    }

}
```

先在 `view/layout` 目录建立一个布局文件 `main.php` ，代码如下：

```
<html>
<head>
    <title><?= $this->title ?></title>
</head>
<body>
    <?= $content ?>
```

```
</body>
</html>
```

然后在 `view` 目录创建一个 `profile` 目录，在目录中创建一个 `index.php` 文件，代码如下:

- MixPHP 的视图直接使用 PHP 做为引擎。
- 视图文件名全部使用小写，多个单词时，使用下划线分隔，例如： `setting_profile.php` 。
- 通过 `$this->name` 可以传递数据到布局文件中使用。

```
<?php
$this->title = 'Profile';
?>

<p>name: <?= $name ?>, age: <?= $age ?></p>
<p>friends:</p>
<ul>
    <?php foreach($friends as $name): ?>
        <li><?= $name ?></li>
    <?php endforeach; ?>
</ul>
```

渲染视图

从上面的例子中可看出，视图的渲染是在控制器中，代码如下：

> return $this->render(视图名, 数组);

视图名

不需要加上目录，框架会自动获取，只需输入视图文件名称，不需要带 `.php` 后缀。

```
// 当前控制器目录
return $this->render('index', $data);
// 其他目录
return $this->render('dirname.index', $data);
```

数组

需要传递给视图使用的数据，是一个数组类型，数组 `key` 会变为视图内的变量名称，数组 `value` 会变为变量的值。

视图布局

当使用 `$this->render` 渲染视图时，MixPHP会获取控制器属性 `layout` 的值，用来读取对应的布局文

件。

> 如果控制器未定义该属性，则该属性默认为 main。

```
public $layout = 'main';
```

不使用布局

当有需求不需要使用到布局时，使用下面的代码渲染视图：

> return $this->renderPartial(视图文件名, 数组);

视图嵌套

当你在布局中使用公共的侧边栏等类似的需求时，需要在视图中加载另一个视图，如下：

```
<?= $this->render('子视图名', $__data__); ?>
```

> $__data__ 为当前视图传入所有变量的数组，可以让子视图使用父视图的全部变量。

Token

`Token` 组件

Token 组件可以理解为 API 中使用的 Session，因为 API 是脱离 Cookie 的，Session 无法运行，所以 API 通常是在 GET/POST/HEADER 中带上一个 access_token 参数来保持会话状态，MixPHP 提供了 Token 来帮助用户在 API 中操作会话。

类	调用
mix\http\Token	app()->token

门面类	调用
mix\facades\Token	Token::

> Token 组件暂时只支持 Redis，使用前需先安装 Redis 数据库。

组件配置

App配置文件中，该组件的默认配置如下：

>> 到 GitHub 查看默认配置 <<

使用场景

Token 通常有三种使用场景：

- 通过 username、password 获取 access_token，用于授权给客户端，使用 access_token 可调用用户相关的接口。
- 通过 appid、appsecret、grant_type 获取 access_token，用于授权给第三方，使用 access_token 可调用平台内 grant_type 参数定义的相关权限的接口。
- OAuth 2.0：将自己平台内获取用户相关信息的权限授权给第三方。

> Token 其实是使用 Redis 组件开发的组件，参考源代码可改造出 OAuth 2.0 Token。

使用范例

第一种使用场景的范例代码。

获取 Token，控制器：

```
// 获取 token 方法
public function actionToken()
```

```
{
    /* 验证账号密码成功后 */

    // 创建 tokenId
    Token::createTokenId();
    // 保存会话信息
    $userinfo = [
        'uid'      => 1088,
        'openid'   => 'yZmFiZDc5MjIzZDMz',
        'username' => '小明',
    ];
    Token::set('userinfo', $userinfo);
    // 设置唯一索引
    Token::setUniqueIndex($userinfo['openid']);

    // 响应
    return [
        'errcode'      => 0,
        'access_token' => Token::getTokenId(),
        'expires_in'   => app()->token->expiresIn,
        'openid'       => $userinfo['openid'],
    ];
}
```

效验Token：

在前置中间件中校验。

```
// 前置中间件的 handle 方法
public function handle($callable, \Closure $next)
{
    // 添加中间件执行代码
    $userinfo = Token::get('userinfo');
    if (empty($userinfo)) {
        // 返回错误码
        return ['errcode' => 300000, 'errmsg' => 'Permission denied'];
    }
    // 执行下一个中间件
    return $next();
}
```

`createTokenId` 方法

> 需要在 set 前使用。

创建一个TokenId。

`setUniqueIndex` 方法

> 需要在 set 后使用。

设置唯一索引，设置后会从上次设置的索引找出上次的tokenId，并删除上次的token数据，使上次的token失效，然后再将本次的tokenId存入索引。

```
Token::setUniqueIndex($openid);
```

`set` 方法

变量赋值。

```
Token::set('name', '小华');
```

`get` 方法

获取变量的值。

```
Token::get('name');
```

> name不存在时返回null。

`has` 方法

判断变量是否存在。

```
Token::has('name');
```

`delete` 方法

删除变量。

```
Token::delete('name');
```

`clear` 方法

清空全部变量。

```
Token::clear();
```

`getTokenId` 方法

获取TokenId。

```
Token::getTokenId();
```

`refresh` 方法

刷新 token，在旧 token 有效期内，生成一个新的 token，刷新成功后可通过 getTokenId 获取新的 tokenId。

```
Token::refresh();
```

Session

`Session` 组件

Session（会话）组件可以让你保持一个用户的 "状态" ，并跟踪他在浏览你的网站时的活动。

类	调用
mix\http\Session	

门面类	调用
mix\facades\Session	Session::

> Session 组件暂时只支持 Redis，使用前需先安装 Redis 数据库。

组件配置

App配置文件中，该组件的默认配置如下：

>> 到 GitHub 查看默认配置 <<

使用范例

用户登陆控制器：

```
// 登陆方法
public function actionLogin()
{
    /* 验证账号密码成功后 */

    // 创建 sessionId
    Session::createSessionId();
    // 保存会话信息
    $userinfo = [
        'uid'      => 1088,
        'openid'   => 'yZmFiZDc5MjIzZDMz',
        'username' => '小明',
    ];
    Session::set('userinfo', $userinfo);
    // 响应
    return $this->render('login', ['message' => '新增成功']);
}
```

效验Session：

在前置中间件中校验。

```
// 前置中间件的 handle 方法
public function handle($callable, \Closure $next)
{
    // 添加中间件执行代码
    $userinfo = Session::get('userinfo');
    if (empty($userinfo)) {
        // 跳转到首页
        return Response::redirect('/');
    }
    // 执行下一个中间件
    return $next();
}
```

set 方法

变量赋值。

```
Session::set('name', '小华');
```

get 方法

获取变量的值。

```
Session::get('name');
```

> name不存在时返回null。

has 方法

判断变量是否存在。

```
Session::has('name');
```

delete 方法

删除变量。

```
Session::delete('name');
```

clear 方法

清空全部变量。

```
Session::clear();
```

`getSessionId` 方法

获取SessionId。

```
Session::getSessionId();
```

Cookie

`Cookie` 组件

Cookie是储存在用户本地终端上的数据，该类用来操作这些数据。

类	调用
mix\http\Cookie	app()->cookie

门面类	调用
mix\facades\Cookie	Cookie::

组件配置

App配置文件中，该组件的默认配置如下：

>> 到 GitHub 查看默认配置 <<

`get` 方法

获取变量的值。

> name不存在时返回null。

```
Cookie::get('name');
```

`set` 方法

变量赋值。

```
Cookie::set('name', '小华');
```

`has` 方法

判断变量是否存在。

```
Cookie::has('name');
```

`delete` 方法

删除变量。

```
Cookie::delete('name');
```

`clear` 方法

清空 当前域 所有变量。

```
Cookie::clear();
```

更细粒度操作

当需要更细粒度操作时，使用 `Request` 组件的 `cookie` 方法获取值：

> app()->request->cookie($name)

使用 `Response` 组件的 `setCookie` 方法设置值：

> app()->response->setCookie($name, $value = '', $expires = 0, $path = '', $domain = '', $secure = false, $httpOnly = false)

文件上传

文件上传

MixPHP 的文件上传类只用于处理已经上传的文件，而对上传文件的各种限制是在“模型”里完成的。

> 通常情况下，你无需自己实例化，模型验证完成后会自动实例化为模型的属性。

类	调用
mix\http\UploadFile	UploadFile::newInstanceByName($name)

全部属性

- name : 文件名
- type : MIME类型
- tmpName : 临时文件名
- error : 错误码
- size : 文件尺寸

获取实例

通过 `$_FILES` 数组的 `name` 获取实例。

```
$file = UploadFile::newInstanceByName($name);
```

获取基础名称

获取上传的文件名名称部分。

```
$file->getBaseName();
```

获取扩展名

获取上传的文件名扩展名部分。

```
$file->getExtension();
```

获取随机文件名

调用后可获取一个随机文件名称（含扩展名）。

```
$file->getRandomFileName();
```

文件另存为

```
// 另存为自定义名称
$file->saveAs($filename)

// 另存为随机名称
$path = app()->getRuntimePath() . 'tmp/' . $file->getRandomFileName();
$file->saveAs($path);
```

图片处理

图片处理

MixPHP 的图片处理类可以使你完成以下的操作：

- 等比缩放
- 居中剪裁
- 顶部剪裁

类	调用
mix\http\Image	Image::open($filename)

全部属性

- filename : 图片的路径 (含路径)
- width : 图片宽度
- height : 图片高度
- mime : 图片的 MIME 信息

打开图片

通过图片的路径生成图片对象。

```
$image = Image::open($filename);
```

获取图片文件大小

```
$image->getSize();
```

等比缩放

> resize($width, $height)

```
// 普通
$image->resize(200, 200);

// 链式操作
Image::open($filename)->resize(200, 200);
```

图片剪裁

> crop($width, $height, $mode)

`$mode` 的常量明细如下：

- Image::CROP_CENTER
- Image::CROP_TOP

```
// 普通
$image->crop(200, 200, Image::CROP_CENTER);

// 链式操作
Image::open($filename)->resize(200, 200, Image::CROP_CENTER);
```

保存

将操作后的图片保存到原来的路径。

```
// 链式操作
Image::open($filename)->resize(200, 200)->save();
```

另存为

将操作后的图片另存为其他文件。

```
// 链式操作
$filename = app()->getPublicPath() . 'uploadfile/img001.jpg';
$thumb = str_replace('.', '.thumb.', $filename);
Image::open($filename)->resize(200, 200)->saveAs($thumb);
```

分页

分页

MixPHP 的分页类采用的一种非常灵活的设计方式，分页的构建都由视图层完成，可以构建任意结构的分页式样。

类	调用
mix\http\Pagination	new Pagination([配置]);

> 配置参数放在配置文件 objects 字段，再使用 app()->createObject($name) 实例化对象是更好的方式。

模型范例

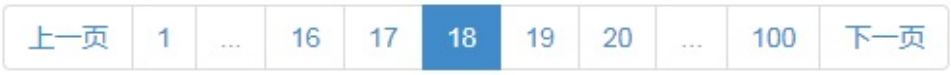
MixPHP 建议用户在模型中使用分页类，因为在模型内更加方便获取分页数据，分页数据可随分页对象由控制器传递至视图。

```
// 模型内的方法
public function getPagination($page)
{
    return new Pagination([
        // 数据结果集
        'items'       => $data,
        // 数据总行数
        'totalItems'  => 987,
        // 当前页, 值 >= 1
        'currentPage' => $page,
        // 每页显示数量
        'perPage'     => 10,
        // 固定最小最大页码
        'fixedMinMax' => true,
        // 数字页码展示数量
        'numberLinks' => 5,
    ]);
}
```

视图范例

我们设计了三个常用的分页式样，使用 bootstrap 的用户可直接复制代码使用，使用其他前端框架的用户只需修改相关的 HTML 即可。

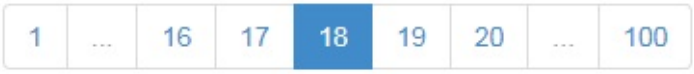
1. 带上下页且固定最小最大页



当前第 18 页，共 100 页

```
<?php if ($pagination->display()): ?>
    <nav aria-label="Page navigation">
        <ul class="pagination">
            <?php if ($pagination->hasPrev()): ?>
                <li><a href="/?page=prev(); ?>">上一页</a></li>
            <?php else: ?>
                <li class="disabled"><span>上一页</span></a></li>
            <?php endif; ?>
            <?php foreach ($pagination->numbers() as $number): ?>
                <?php if ($number->text == 'ellipsis'): ?>
                    <li class="disabled"><span>...</span></li>
                <?php else: ?>
                    <li <?= $number->selected ? 'class="active"' : ''; ?>><a hr
ef="/?page=text; ?>"><?= $number->text; ?></a></li>
                <?php endif; ?>
            <?php endforeach; ?>
            <?php if ($pagination->hasNext()): ?>
                <li><a href="/?page=next(); ?>">下一页</a></li>
            <?php else: ?>
                <li class="disabled"><span>下一页</span></a></li>
            <?php endif; ?>
        </ul>
    </nav>
    <p><?php echo "当前第 ", $pagination->currentPage, " 页，共 ", $pagination->t
otalPages, " 页"; ?></p>
<?php endif; ?>
```

2. 纯数字固定最小最大页



当前第 18 页，共 100 页

```
<?php if ($pagination->display()): ?>
    <nav aria-label="Page navigation">
        <ul class="pagination">
            <?php foreach ($pagination->numbers() as $number): ?>
                <?php if ($number->text == 'ellipsis'): ?>
                    <li class="disabled"><span>...</span></li>
                <?php else: ?>
                    <li <?= $number->selected ? 'class="active"' : ''; ?>><a hr
ef="/?page=text; ?>"><?= $number->text; ?></a></li>
                <?php endif; ?>
```

```
            <?php endforeach; ?>
        </ul>
    </nav>
    <p><?php echo "当前第 ", $pagination->currentPage, " 页, 共 ", $pagination->t
otalPages, " 页"; ?></p>
<?php endif; ?>
```

3. 带首尾页上下页

当前第 18 页，共 100 页

```
<?php if ($pagination->display()): ?>
    <nav aria-label="Page navigation">
        <ul class="pagination">
            <?php if ($pagination->hasFirst()): ?>
                <li><a href="/">首页</a></li>
            <?php else: ?>
                <li class="disabled"><span>首页</span></a></li>
            <?php endif; ?>
            <?php if ($pagination->hasPrev()): ?>
                <li><a href="/?page=prev(); ?>">上一页</a></li>
            <?php else: ?>
                <li class="disabled"><span>上一页</span></a></li>
            <?php endif; ?>
            <?php foreach ($pagination->numbers() as $number): ?>
                <li <?= $number->selected ? 'class="active"' : ''; ?>><a href="
/?page=text; ?>"><?= $number->text; ?></a></li>
            <?php endforeach; ?>
            <?php if ($pagination->hasNext()): ?>
                <li><a href="/?page=next(); ?>">下一页</a></li>
            <?php else: ?>
                <li class="disabled"><span>下一页</span></a></li>
            <?php endif; ?>
            <?php if ($pagination->hasLast()): ?>
                <li><a href="/?page=totalPages; ?>">尾页</a></li>
            <?php else: ?>
                <li class="disabled"><span>尾页</span></a></li>
            <?php endif; ?>
        </ul>
    </nav>
    <p><?php echo "当前第 ", $pagination->currentPage, " 页, 共 ", $pagination->t
otalPages, " 页"; ?></p>
<?php endif; ?>
```

验证码

验证码

MixPHP 的验证码类并没有像其他框架一样设计成非常耦合的方式，使用户无法了解验证码实现原理，而是让验证码回归原始，只做验证码生成， 其他交互功能由框架组件来完成。

类	调用
mix\http\Captcha	new Captcha([配置]);

控制器 (输出)

在控制器的方法中输出验证码图片，微调 `angleRand` 、 `xSpacing` 、 `yRand` 三个参数可调整文字的相对位置，验证码中的文字越是粘连越难以破解。

英文验证码

> `TimesNewRomanBold.TTF` 英文字体文件框架内不包含，需用户自行下载。

```
public function actionIndex()
{
    Response::setHeader('Content-Type', 'image/png');
    $captcha = new Captcha([
        'width'      => 100,
        'height'     => 40,
        'fontFile'   => app()->basePath . '/fonts/TimesNewRomanBold.TTF',
        'fontSize'   => 20,
        'wordNumber' => 4,
        'angleRand'  => [-20, 20],
        'xSpacing'   => 0.82,
        'yRand'      => [5, 15],
    ]);
    $captcha->generate();
    Session::set('captchaText', $captcha->getText());
    return $captcha->getContent();
}
```

中文验证码

> `SIMLI.TTF` 中文字体文件框架内不包含，需用户自行下载。

```
public function actionIndex()
{
    Response::setHeader('Content-Type', 'image/png');
```

```
    $captcha = new Captcha([
        'width'      => 100,
        'height'     => 40,
        'fontFile'   => app()->basePath . '/fonts/SIMLI.TTF',
        'fontSize'   => 22,
        'wordSet'    => '酸旧却充秋遍锻玉夏疗尖殖井费州访吹荣铜沿替滚客召旱悟刺脑措贯藏敢令隙炉壳硫煤迎铸粘探临薄旬善福纵择礼愿伏残雷延烟句纯渐耕跑泽慢栽鲁赤繁境潮横掉锥希池败船假亮谓托伙哲怀割摆贡呈劲财仪沉炼麻罪祖息车穿货销齐鼠抽画饲龙库守筑房歌寒喜哥洗蚀废纳腹乎录镜妇恶脂庄擦险赞钟摇典柄辩竹谷卖乱虚桥奥伯赶垂途额壁网截野遗静谋弄挂课镇妄盛耐援扎虑键归符庆聚绕摩忙舞遇索顾胶羊湖钉仁音迹碎伸灯避泛亡答勇频皇柳哈揭甘诺概宪浓岛袭谁洪谢炮浇斑讯懂灵蛋闭孩释乳巨徒私银伊景坦累匀霉杜乐勒隔弯绩招绍胡呼痛峰零柴簧午跳居尚丁秦稍追梁折耗碱殊岗挖氏刃凸役剪川雪链渔啦脸户洛孢勃盟买杨宗焦赛旗滤硅缸夹念兰映沟乙吗儒杀汽磷艰晶插埃燃欢铁补咱芽永瓦倾阵碳演威附牙芽永瓦斜灌欧献顺猪洋腐请透司危括脉宜笑若尾束壮暴企菜穗楚汉愈绿拖牛份染稳夺硬价努翻奇甲预职评读泥辟告卵箱掌氧恩爱停曾溶营终纲孟钱待尽俄缩沙退陈讨炭股坐蒸凝竟陷枪黎救冒暗洞犯筒您宋淡允叛畜俘摸锈扫毕璃宝芯爷鉴秘净蒋钙肩腾枯抛轨堂拌爸循诱祝励肯酒绳穷塘燥泡袋朗喂铝软渠颗惯贸粪综墙趋彼届墨碍启逆卸航衣孙龄岭骗休借们以我到他会作时要动国产的一是工就年阶义发成部民可出能方进在了不和有饿',
        'wordNumber' => 3,
        'angleRand'  => [-20, 20],
        'xSpacing'   => 0.85,
        'yRand'      => [5, 10],
    ]);
    $captcha->generate();
    Session::set('captchaText', $captcha->getText());
    return $captcha->getContent();
}
```

视图 (引用)

在视图中使用 `img` 标签，在 `src` 属性中指向验证码控制器方法的 URL 地址。

```
<img src="/captcha" height=38>
```

验证器 (验证)

用户提交验证码后需要在验证器中验证，使用验证规则 `call` 来自定义验证。

1. 首先定义规则、场景、消息。

```
// 规则
public function rules()
{
    return [
        'captcha' => ['call', 'callback' => [$this, 'captchaCheck']],
    ];
}

// 场景
```

```
public function scenarios()
{
    return [
        'index' => ['required' => ['captcha']],
    ];
}

// 消息
public function messages()
{
    return [
        'captcha' => '验证码不正确.',
    ];
}
```

2. 增加 `captchaCheck` 方法

```
public function captchaCheck($attributeValue)
{
    $captchaText = Session::get('captchaText');
    if (strcasecmp($attributeValue, $captchaText) == 0) {
        return true;
    }
    return false;
}
```

WebSocket 服务

简介

> - 该服务依赖 Swoole 扩展，Windows下无法执行。
> - 异步 redis 需按 Swoole 官方要求安装 hiredis 与 重新编译 swoole，详情：https://wiki.swoole.com/wiki/page/p-redis.html 。

WebSocket 服务

MixPHP 封装了非常完整的 WebSocket 解决方案，开箱即用，还在源代码中附带了完整范例代码。

类	调用
mix\websocket\WebSocketServer	app()->createObject('name')

使用场景

如： 消息推送、在线聊天、直播弹幕、棋牌游戏等。

优点

- 能与 Session / Token 无缝对接，实现会话机制；
- 可异步对接 Redis 的订阅，实现通过消息队列主动发消息至客户端，这样做出来的 WebSocket 服务可以做负载均衡，实现高性能；
- 可通过消息处理器进行命令路由，实现传统MC分离的开发方式；
- 模型验证器可在 WebSocket 的模型中使用，验证数据的有效性与合法性，远离脏数据、攻击的风险；

开发目录

```
apps/
└── websocketd
```

命令执行

```
mix-websocketd [入口文件] [命令] [选项]
```

范例

配置文件

>> 到 GitHub 查看默认配置 <<

控制器代码

源代码的范例里几乎把全部流程都写了，就差业务代码了。

>> 到 GitHub 查看 DEMO <<

进程管理

在命令行使用以下命令管理。

```
// 查看帮助
mix-websocketd -h

// 启动
mix-websocketd service start

// 启动（守护）
mix-websocketd service start -d

// 停止
mix-websocketd service stop

// 重启
mix-websocketd service restart

// 状态
mix-websocketd service status
```

也可使用如下 Linux 命令管理进程。

```
// 查找进程
ps -ef | grep mix-websocketd

// 结束主进程
kill <PID>
```

回调函数

回调函数

WebSocketServer 有三个回调函数：

- Open：连接握手成功后执行
- Message：接收到消息时执行
- Close：连接关闭时执行，非 WebSocket 连接不会执行

下面是 DEMO 中创建服务的部分代码 ，这里绑定了三个回调函数。

```
// 创建服务
$server = app()->createObject('webSocketServer');
$server->on('Open', [$this, 'onOpen']);
$server->on('Message', [$this, 'onMessage']);
$server->on('Close', [$this, 'onClose']);
// 启动服务
$server->start();
```

> 源码 DEMO 中关于回调函数中的代码是在生产环境中验证过的最佳范例，全部代码用户都可根据自己的业务需求去修改。

消息处理器

消息处理器

消息处理器是负责将用户发送的 WebSocket 消息，根据路由转发到控制器去处理的类，使用户可以用 MC 的方式开发。

类	调用
mix\websocket\MessageHandler	app('websocket')->messageHandler

路由配置

在App配置文件中，关于该组件的默认配置如下：

```
// 消息处理器
'websocket.messageHandler' => [
    // 类路径
    'class'               => 'mix\websocket\MessageHandler',
    // 控制器命名空间
    'controllerNamespace' => 'apps\websocketd\controllers',
    // 路由规则
    'rules'               => [

        'JOIN'    => ['Join', 'Room'],
        'MESSAGE' => ['Message', 'Emit'],

    ],
],
```

路由规则

详解路由规则：

- 动作： `JOIN`
- 控制器的类名，不包括 Command 后缀： `Join`
- 控制器的方法名，不包括 action 前缀： `Room`

客户端测试

客户端测试

1. 在 Web 应用中生成一个 sessionid ，用于测试代码中的会话。

修改 `apps/httpd/controllers/IndexController.php` 文件的默认动作，代码如下：

```
// 默认动作
public function actionIndex()
{
    \Mix::app()->session->set('userinfo', ['uid' => 1008, 'name' => '小明']);
    return \Mix::app()->session->getSessionId();
}
```

在浏览器中访问将得到一个 `sessionid` 值。

2. 启动 mix-websocketd 服务。

```
mix-websocketd service start
```

3. 修改下面代码的 ip / port / sessionid 这些值，另存为一个 HTML 文件，在 Chrome 调试模式的 Console 窗口中调试。

```
<html>
<head>
    <title>WebSocket</title>
</head>
<body>
<script>
    var webSocket = function () {
        ws = new WebSocket("ws://192.168.181.131:9502?session_id=M1WqwAlYmp1bVU
kuB6cVUlD2Rq");
        ws.onopen = function() {
            console.log("连接成功");
        };
        ws.onmessage = function(e) {
            console.log("收到服务端的消息：" + e.data);
        };
        ws.onclose = function() {
            console.log("连接关闭");
        };
    };
    webSocket();
</script>
</body>
```

```
</html>
```

4. 开始测试

> 加入房间范例代码测试。

在 Console 窗口中输入：

```
ws.send('{"event":"joinRoom","params":{"room_id":88888}}');
```

会收到以下响应：

```
收到服务端的消息：{"callback":"joinRoom","data":{"message":"小明 加入房间"}}
收到服务端的消息：{"callback":"joinRoom","data":{"message":"我 加入房间"}}
```

> 发送消息给用户范例代码测试。

在 Console 窗口中输入：

```
ws.send('{"event":"messageEmit","params":{"to_uid":1008,"message":"Hello World"
}}');
```

会收到以下响应：

```
收到服务端的消息：Hello World
```

\n\n\n~~~\n\n**4. 开始测试**\n\n>[success] 加入房间范例代码测试。\n\n在 Console 窗口中输入：\n\n~~~\nws.send('{\"event\":\"joinRoom\",\"params\":{\"room_id\":88888}}');\n~~~\n\n会收到以下响应：\n\n~~~\n收到服务端的消息：{\"callback\":\"joinRoom\",\"data\":{\"message\":\"小明 加入房间\"}}\n收到服务端的消息：{\"callback\":\"joinRoom\",\"data\":{\"message\":\"我 加入房间\"}}\n~~~\n\n>[success] 发送消息给用户范例代码测试。\n\n在 Console 窗口中输入：\n\n~~~\nws.send('{\"event\":\"messageEmit\",\"params\":{\"to_uid\":1008,\"message\":\"Hello World\"}}');\n~~~\n\n会收到以下响应：\n\n~~~\n收到服务端的消息：Hello World\n~~~\n"},"options":{"base":"./","environment":"reader"},"style":""}

nginx代理

nginx代理

在 `http` 节点内， `server` 节点外，增加如下：

```
upstream websocket {
    server 127.0.0.1:9502;
}
```

在 `server` 节点内，增加如下：

```
location /websocket {
    proxy_pass http://websocket;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "Upgrade";
}
```

配置完后，如果 `server_name` 为 `www.test.com` ，就可以在 JavaScript 中这样访问：

```
ws://www.test.com/websocket
```

WSS

只需在 `server` 节点内正常配置 ssl 即可，如下：

```
ssl on;
ssl_certificate ***.crt;
ssl_certificate_key ***.key;
```

60s无消息断线

60s无消息断线

WebSocket 在建立连接后，如果没有消息接收与发送，60s内连接被服务器断开，这是因为 Nginx 或者 负载均衡层做了设置。

解决方案：

- 如果你的服务器 Nginx 是在最外层，那么修改 proxy_read_timeout 为更大的值即可解决。
- 如果最外层是阿里的负载均衡 SLB，那么恭喜你，修改 Nginx 也无效，只能通过 Javascript 做 ping/pong，但是 H5 并没有设计 ping/pong 的接口，所以只能使用 json 来当 ping 使用，就是在 Javascript 发送 `{"event":"PING"}` ，服务器回复 `{"callback":"PONG"}` 即可。

多进程

ProcessPoolTaskExecutor

> 该服务依赖 Swoole 扩展，Windows下无法执行。

ProcessPoolTaskExecutor

进程池任务执行器是一个多进程开发工具，能充分利用多核性能，并行处理大量数据，PHP 原生是不支持多进程的，多进程是基于 Swoole 开发，MixPHP 已经建立好了进程模型，参考范例代码即可非常容易的编写出多进程任务处理。

> 多进程中只能使用同步代码，不可使用协程。

类	调用
mix\task\ProcessPoolTaskExecutor	new ProcessPoolTaskExecutor([配置]);

模式

- 流水线模式：适合消息队列(MQ)消费处理、数据采集、定时大量计算等场景。
- 推送模式：适合消息推送等场景。
- 守护模式：当需要常驻执行时，增加此模式，该模式只可与上两种模式一起使用。

优点

- 平滑重启：当 kill 主进程时，子进程处理完工作会自动退出，不丢失数据。
- 高可用：子进程异常奔溃时，主进程将重建子进程。
- 高性能：多进程运行，充分利用多个CPU并行计算，性能强劲。
- 多功能：左、中、右进程分工合作，可以处理各种后端计算需求。

开发目录

> 通常在控制台程序模块、守护程序模块中开发。

```
├── apps
│   ├── console
│   │   ├── commands
│   │   ├── config
│   │   ├── libraries
│   │   └── runtime
│   ├── daemon
│   │   ├── commands
```

```
│   │   ├── config
│   │   ├── libraries
│   │   └── runtime
```

流水线模式

流水线模式

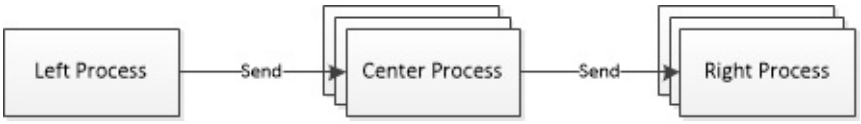
流水线模式：就是像流水线生产一样，前面投放，中间生产，后面打包，这样的明确分工合作的执行方式。

适合消息队列(MQ)消费处理、数据采集、定时大量计算等场景。

单次执行

适合单次计算大量数据，也可通过使用 linux 的 crontab 等工具定时触发命令。

进程模型



范例代码

>> 到 GitHub 查看 DEMO <<

守护执行

适合消息队列(MQ)消费处理，长期执行任务处理。

进程模型

单个左进程：通常一个左进程性能就够了，因为左进程只是取数据，计算量非常小。

多个左进程：只有在处理非常大量的数据时，才会用到。

范例代码

>> 到 GitHub 查看 DEMO <<

推送模式

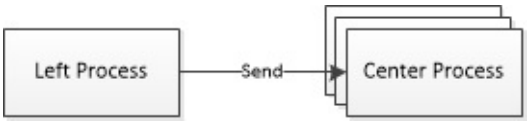
推送模式

适合消息推送等场景。

单次执行

适合单次推送大量数据，也可通过使用 linux 的 crontab 等工具定时触发命令。

进程模型



范例代码

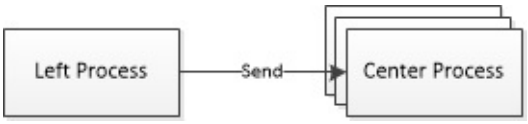
>> 到 GitHub 查看 DEMO <<

守护执行

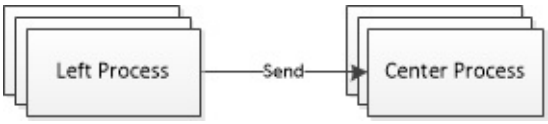
适合消息队列(MQ)消费处理，长期执行推送处理。

进程模型

单个左进程：通常一个左进程性能就够了，因为左进程只是取数据，计算量非常小。



多个左进程：只有在处理非常大量的数据时，才会用到。



范例代码

>> 到 GitHub 查看 DEMO <<

在 Supervisor 中使用

在 `Supervisor` 中使用

在 `supervisor` 中使用 `ProcessPoolTaskExecutor` 时的一些疑问。

修改代码后如何快速重启

错误的方法：

直接使用 `supervisor` 的重启命令

```
supervisorctl restart program_name
```

如果 `ProcessPoolTaskExecutor` 的进程队列中有很多未处理完的任务，主进程会要等待全部任务处理完成才会退出，所以这个命令会 卡住很久，在修改代码需要立即重启进程的时候这个等待显然难以接收的。

正确的方法：

```
// 使用 -f 参数强制退出守护进程
mix-daemon single stop -f
```

以上命令会找到守护程序对应的 PID，并发送 `kill -s SIGUSR1` 命令，让进程立即退出 (这个退出是优雅的，只是进程队列的数据没有处理完成，这些数据在内存中，在新进程启动时可继续使用，不会丢失)，这时，由于 `supervisor` 监听到这个进程退出了，就会立即重启一个新的进程，这时启动的新进程就是新的代码了，如此就完成了一次修改代码后的快速重启。

>> 到 GitHub 查看强制退出实现源码 <<

协程

简介

协程

协程，又称微线程，纤程。英文名 `Coroutine` ，MixPHP 从 `v1.1.0` 开始支持协程开发。

优势

协程模式与常驻模式/传统模式相比：

常驻模式/传统模式都属于同步阻塞编程，由于同一个进程不能并行处理请求，所以为了提高并发，只能开启更多的进程，通常超过 `100` 甚至更多，每个进程都有基础的内存消耗，加起来就很多了，而且受限于 Linux 总进程数限制，并发总数无法突破，加上进程非常多之后，CPU 需要更多的线程切换，浪费了很多性能，当然相比 FPM 的传统模式每次都需从头开始，常驻模式还是要好非常多的，但是协程显然更加优秀。

协程模式的执行方式：

协程模式中一个进程可以同时执行 N 个请求，但同一时刻只执行其中的某一个请求，也就是说，当执行到 MySQL/Redis 这些客户端时，由于需要等待客户端响应，常驻模式/传统模式通常是在傻傻的等待响应，而协程这个时候会挂起当前协程，切换到其他协程中去处理其他请求，所以协程能同时处理 N 个请求，每增加一个请求只需增加一些内存消耗，相比增加一个进程的内存消耗，显然是少太多的，由于协程能并行处理，所以通常只需配置于 CPU 数量 1～2 倍左右的进程数即可，更少的进程带来更少的 CPU 线程切换，又减少很多性能损耗。

如何开启协程

如何开启协程

> 需 Swoole >= 4.2.2 才可开启。

MixPHP 默认是关闭协程的，可按下面操作开启协程。

打开 `mix-httpd` 服务器的配置文件：

```
apps/httpd/config/httpd.php
```

>> 到 GitHub 查看 mix-httpd 配置 <<

- 将 `enable_coroutine` 设置为 `true` 。
- 将 `configFile` 修改为协程专用配置文件的路径： `__DIR__ . '/http_coroutine.php'` 。

>> 到 GitHub 查看协程配置文件有什么不一样 <<

重新启动 `mix-httpd` 即可。

HTTP 协程开发

HTTP 协程开发

MixPHP 的 `HTTP` 协程开发，简单到不可思议，协程变量隔离什么的都处理好了，用户只需直接使用协程客户端即可，而且协程客户端使用方法于常驻模式完全一致，也就是说协程模式与常驻模式只有配置文件上的差别，还有多了一个连接池的配置而已。

协程客户端请阅读 “客户端” 章节的内容。

>> 到 GitHub 查看协程配置文件有什么不一样 <<

命令行协程开发

命令行协程开发

客户端

MySQL

PDO

PDO 组件

PDO 组件用于 MySQL 等关系型数据库的操作，语法简单明了，且具有独特的查询构造方式，可构造任何复杂的 SQL。

> 该组件基于 pdo 扩展，语句预处理 将帮助你免于SQL注入攻击。

类	调用	连接方式
mix\client\PDO	app()->pdo	短连接

门面类 (通常在 HTTP 开发中使用)	调用
mix\facades\PDO	PDO::

通过配置实例化调用 (通常在命令行开发使用)
$pdo = mix\client\PDO::newInstanceByConfig('libraries.[***]');

组件配置

App配置文件中，该组件的默认配置如下：

>> 到 GitHub 查看默认配置 <<

插入

```
$data = [
    'name'    => 'xiaoliu',
    'content' => 'hahahaha',
];
$success = PDO::insert('post', $data)->execute();
// 获得刚插入数据的id
$insertId = PDO::getLastInsertId();
```

批量插入

```
$data = [
    ['name' => 'xiaoliu', 'content' => 'hahahaha'],
    ['name' => 'xiaoliu', 'content' => 'hahahaha'],
    ['name' => 'xiaoliu', 'content' => 'hahahaha'],
    ['name' => 'xiaoliu', 'content' => 'hahahaha'],
];
$success = PDO::batchInsert('post', $data)->execute();
```

```
// 获得受影响的行数
$affectedRows = PDO::getRowCount();
```

更新

常规更新：

```
$set = [
    'num'  => 2,
    'name' => 'xiaoliu2',
];
$where = [
    ['id', '=', 23],
];
$success = PDO::update('post', $set, $where)->execute();
// 获得受影响的行数
$affectedRows = PDO::getRowCount();
```

自增、自减：

```
$set = [
    'num'   => ['+', 2],
    'num1'  => ['-', 1],
];
$where = [
    ['id', '=', 23],
];
$success = PDO::update('post', $set, $where)->execute();
// 获得受影响的行数
$affectedRows = PDO::getRowCount();
```

删除

```
$success = PDO::delete('post', [['id', '=', 15]])->execute();
// 获得受影响的行数
$affectedRows = PDO::getRowCount();
```

查询

执行原生 SQL

> 请不要直接把参数拼接在 SQL 内执行，带参数的 SQL 请使用参数绑定。

```
$rows = PDO::createCommand("SELECT * FROM `post`")->queryAll();
```

参数绑定

普通参数绑定：

```
$sql = "SELECT * FROM `post` WHERE id = :id AND name = :name";
$rows = PDO::createCommand($sql)->bindParams([
    'id'   => 28,
    'name' => 'xiaoliu',
])->queryOne();
```

`IN` 、 `NOT IN` 参数绑定：

> `PDO` 扩展是不支持绑定数组参数的，所以 WHERE IN 都只能直接 `implode(', ',$array)` 拼接到 SQL 里面，MixPHP 帮你做了这一步，所以只需像下面这样使用。

```
$sql  = "SELECT * FROM `post` WHERE id IN (:id)";
$rows = PDO::createCommand($sql)->bindParams([
    'id' => [28, 29, 30],
])->queryAll();
```

查询组合

MixPHP 推崇原生 SQL 查询数据库，但由于原生 SQL 在动态 Where 时，做参数绑定会导致逻辑有些复杂，所以 MixPHP 封装了一个查询组合的功能，方便动态控制 Where 与 自动参数绑定。

用户可将整个 SQL 拆分为多个部分，每个部分可选择使用下例两个参数：

- `params` 字段内的值会绑定到对应的sql中。
- `if` 字段的值为 false 时，该段 SQL 会丢弃。

常用查询组合：

```
$rows = PDO::createCommand([
    ['SELECT * FROM `post`'],
    ['WHERE id = :id AND name = :name ORDER BY id ASC',
        'params' => [
            'id'   => $this->id,
            'name' => $this->name,
        ],
    ],
])->queryAll();
```

动态 Where 查询组合：

```
$rows = PDO::createCommand([
    ['SELECT * FROM `post` WHERE 1 = 1'],
    ['AND id = :id', 'params' => ['id' => $this->id], 'if' => isset($this->id)],

    ['AND name = :name', 'params' => ['name' => $this->name], 'if' => isset($th
is->name)],
    ['ORDER BY `post`.id ASC'],
])->queryAll();
```

> WHERE 1 = 1 是一个小技巧，能避免没有 Where 时 SQL 错误的情况出现。

更复杂的查询组合，包含了：

- 动态 Join
- 动态 Where
- 分页

```
$rows = PDO::createCommand([
    ['SELECT *'],
    ['FROM `post`'],
    [
        'INNER JOIN `user` ON `user`.id = `post`.id',
        'if' => isset($this->name),
    ],
    ['WHERE 1 = 1'],
    [
        'AND `post`.id = :id',
        'params' => ['id' => $this->id],
        'if'     => isset($this->id),
    ],
    [
        'AND `user`.name = :name',
        'params' => ['name' => $this->name],
        'if'     => isset($this->name),
    ],
    ['ORDER BY `post`.id ASC'],
    ['LIMIT :offset, :rows', 'params' => ['offset' => ($this->currentPage - 1) *
 $this->perPage, 'rows' => $this->perPage]],
])->queryAll();
```

查询返回结果集

返回多行，每行都是列名和值的关联数组。

> 没有结果返回空数组。

```
$rows = PDO::createCommand("SELECT * FROM `post`")->queryAll();
```

返回一行 (第一行)。

> 没有结果返回 false。

```
$row = PDO::createCommand("SELECT * FROM `post` WHERE id = 28")->queryOne();
```

返回一列。

> 没有结果返回空数组。

```
// 第一列
$titles = PDO::createCommand("SELECT title FROM `post`")->queryColumn();

// 第二列
$titles = PDO::createCommand("SELECT * FROM `post`")->queryColumn(1);
```

返回一个标量值。

> 如果该查询没有结果则返回 false。

```
$count = PDO::createCommand("SELECT COUNT(*) FROM `post`")->queryScalar();
```

返回一个原生结果集 `PDOStatement` 对象。

> 通常在查询大量结果时，为了避免内存溢出时使用。

```
$result = PDO::createCommand("SELECT * FROM `post`")->query();
while ($item = $result->fetch()) {
    var_dump($item);
}
```

返回原生 SQL 语句

`PDO` 扩展是无法获取最近执行的 SQL 的，所以这个功能是 MixPHP 通过参数构建出来的，这个在调试时是非常好用的功能。

```
$sql = PDO::getRawSql();
```

事务

手动事务：

```
PDO::beginTransaction();
try {
    PDO::insert('test', [
        'text' => '测试测试',
    ])->execute();
    PDO::commit();
} catch (\Exception $e) {
    PDO::rollback();
    throw $e;
}
```

自动事务：等同于上面的手动事务。

```
PDO::transaction(function () {
    PDO::insert('test', [
        'text' => '测试测试',
    ])->execute();
});
```

PDOPersistent

`PDOPersistent` 组件

PDOPersistent 是 PDO 的长连接版本，使用方法与 PDO 完全一至，仅配置不同。

> 长连接比短连接可提升两倍左右的并发性能。

类	调用	连接方式
mix\client\PDOPersistent	app()->pdo	长连接

门面类 (通常在 HTTP 开发中使用)	调用
mix\facades\PDO	PDO::

通过配置实例化调用 (通常在命令行开发使用)
$pdo = mix\client\PDOPersistent::newInstanceByConfig('libraries.[***]');

长连接超时问题

MySQL 配置文件内的 `interactive_timeout` 与 `wait_timeout` 参数，决定了 sleep 多长时间的连接会被主动 kill，正常情况下是需要用户自己来处理连接超时的问题，但使用 `PDOPersistent` 组件，用户不需要处理，组件底层已经帮你处理了。

组件配置

与 `PDO` 组件一至，仅 `class` 不同。

PDOMasterSlave

`PDOMasterSlave` 组件

PDOMasterSlave 是 PDO 的主从版本，当数据库需要主从配置时使用，使用方法与 PDO 完全一至，仅配置不同。

类	调用	连接方式
mix\client\PDOMasterSlave	app()->pdo	短连接

门面类 (通常在 HTTP 开发中使用)	调用
mix\facades\PDO	PDO::

组件配置

App配置文件中，该组件配置如下：

```
// 数据库
'pdo'      => [
    // 类路径
    'class'                         => 'mix\client\PDOMasterSlave',
    // 主服务器组
    'masters'                       => [
        'mysql:host=192.168.1.11;port=3306;charset=utf8;dbname=test',
        'mysql:host=192.168.1.12;port=3306;charset=utf8;dbname=test',
    ],
    // 配置主服务器
    'masterConfig'                  => [
        // 数据库用户名
        'username' => 'root',
        // 数据库密码
        'password' => '',
    ],
    // 从服务器组
    'slaves'                        => [
        'mysql:host=192.168.1.75;port=3306;charset=utf8;dbname=test',
        'mysql:host=192.168.1.76;port=3306;charset=utf8;dbname=test',
        'mysql:host=192.168.1.77;port=3306;charset=utf8;dbname=test',
        'mysql:host=192.168.1.78;port=3306;charset=utf8;dbname=test',
    ],
    // 配置从服务器
    'slaveConfig'                   => [
        // 数据库用户名
        'username' => 'root',
```

```
        // 数据库密码
        'password' => '',
    ],
    // 设置PDO属性: http://php.net/manual/zh/pdo.setattribute.php
    'attribute'                         => [
        // 设置默认的提取模式: \PDO::FETCH_OBJ | \PDO::FETCH_ASSOC
        \PDO::ATTR_DEFAULT_FETCH_MODE => \PDO::FETCH_ASSOC,
    ],
],
```

PDOCoroutine

> 需 Swoole >= 4.2.1 才可使用。

`PDOCoroutine` 组件

PDOCoroutine 是 PDO 的协程版本，使用方法与 PDO 完全一至，仅配置不同。

> 该组件使用 Swoole 的一键协程转换技术。

类	调用	连接方式
mix\client\PDOCoroutine	app()->pdo	长连接

门面类 (通常在 HTTP 开发中使用)	调用
mix\facades\PDO	PDO::

通过配置实例化调用 (通常在命令行开发使用)
$pdo = mix\client\PDOCoroutine::newInstanceByConfig('libraries.[***]');

长连接超时问题

MySQL 配置文件内的 `interactive_timeout` 与 `wait_timeout` 参数，决定了 sleep 多长时间的连接会被主动 kill，正常情况下是需要用户自己来处理连接超时的问题，但使用 `PDOCoroutine` 组件，用户不需要处理，组件底层已经帮你处理了。

组件配置

>> 到 GitHub 查看默认配置 <<

连接池

PDOCoroutine 支持连接池，是否使用连接池是可选的，移除 `connectionPool` 配置即可不使用连接池。

Redis

Redis

`Redis` 组件

Redis 组件是使用魔术方法对 redis 扩展提供的方法做映射处理，可调用扩展内提供的所有方法。

类	调用	连接方式
mix\client\Redis	app()->redis	短连接

门面类 (通常在 HTTP 开发中使用)	调用
mix\facades\Redis	Redis::

通过配置实例化调用 (通常在命令行开发使用)
$pdo = mix\client\Redis::newInstanceByConfig('libraries.[***]');

组件配置

App配置文件中，该组件的默认配置如下：

>> 到 GitHub 查看默认配置 <<

如何使用

这里只举例几个常用方法，更多方法请自行百度。

```
// 写入一个string值
Redis::set($key, $value);

// 写入一个带生存时间的string值
Redis::setex($key, 3600, $value);

// 在名称为key的list左边（头）添加一个值为value的 元素
Redis::lpush($key, $value);
```

RedisPersistent

`RedisPersistent` 组件

RedisPersistent 是 Redis 的长连接版本，使用方法与 Redis 完全一至，仅配置不同。

> 长连接比短连接可提升两倍左右的并发性能。

类	调用	连接方式
mix\client\RedisPersistent	app()->redis	长连接

门面类 (通常在 HTTP 开发中使用)	调用
mix\facades\Redis	Redis::

通过配置实例化调用 (通常在命令行开发使用)
$pdo = mix\client\RedisPersistent::newInstanceByConfig('libraries.[***]');

长连接超时问题

Redis 配置文件内的 `timeout` 参数，决定了 sleep 多长时间的连接会被主动 kill，正常情况下是需要用户自己来处理连接超时的问题，但使用 `RedisPersistent` 组件，用户不需要处理，组件底层已经帮你处理了。

组件配置

与 `Redis` 组件一至，仅 `class` 不同。

RedisCoroutine

> 需 Swoole >= 4.2.1 才可使用。

`RedisCoroutine` 组件

RedisCoroutine 是 Redis 的协程版本，使用方法与 Redis 完全一至，仅配置不同。

> 该组件使用 Swoole 的一键协程转换技术。

类	调用	连接方式
mix\client\RedisCoroutine	app()->redis	长连接

门面类 (通常在 HTTP 开发中使用)	调用
mix\facades\Redis	Redis::

通过配置实例化调用 (通常在命令行开发使用)
$pdo = mix\client\RedisCoroutine::newInstanceByConfig('libraries.[***]');

长连接超时问题

Redis 配置文件内的 `timeout` 参数，决定了 sleep 多长时间的连接会被主动 kill，正常情况下是需要用户自己来处理连接超时的问题，但使用 `RedisCoroutine` 组件，用户不需要处理，组件底层已经帮你处理了。

组件配置

>> 到 GitHub 查看默认配置 <<

连接池

RedisCoroutine 支持连接池，是否使用连接池是可选的，移除 `connectionPool` 配置即可不使用连接池。

外部工具库

简介

外部工具库

`Composer` 上有一些非常优秀的库，因为这些库都是专为 FPM 传统模式开发的，在 常驻模式、协程模式 多少有一些兼容问题。

我们会改造并二次封装一些常用的库为组件， 以 `Composer` 库的形式发布在这里。

> 通常 Composer 中的网络库是无法支持协程的，但是基于 Swoole 最新的 PHP Stream 一键协程化技术，才让有些库在经过改造后能支持协程。

think-orm

接入 `think-orm`

> 只能在 常驻同步模式 接入，无法在 常驻协程模式 使用，因为 think-orm 的代码里使用了大量的全局变量，在协程中存在全局变量污染的问题。

1. 使用 composer 安装 `think-orm` ：

```
composer require topthink/think-orm
```

2. 新增 `apps\common\libraries\ThinkOrmInitialize` 类：

```
<?php

namespace apps\common\libraries;

/**
 * Class ThinkOrmInitialize
 * @package apps\common\libraries
 */
class ThinkOrmInitialize
{

    public static function handle()
    {
        $dsn = self::parseDSN(env('DB.DSN'));
        // 数据库配置信息设置
        $config = [
            // 数据库类型
            'type'     => $dsn['type'],
            // 服务器地址
            'hostname' => $dsn['host'],
            // 数据库名
            'database' => $dsn['dbname'],
            // 数据库用户名
            'username' => env('DB.USERNAME'),
            // 数据库密码
            'password' => env('DB.PASSWORD'),
            // 数据库连接端口
            'hostport' => $dsn['port'],
            // 数据库连接参数
            'params'   => [],
            // 数据库编码默认采用utf8
            'charset'  => $dsn['charset'],
```

```
            // 数据库表前缀
            'prefix'   => 'think_',
        ];
        \think\Db::setConfig($config);
    }

    public static function parseDSN($dsn)
    {
        $type = strstr($dsn, ':', true);
        parse_str(str_replace(';', '&', substr(strstr($dsn, ':'), 1)), $data);
        return ['type' => $type] + $data;
    }

}
```

3. 在当前应用配置文件中增加 `initialize` 配置项：

```
// 初始化回调
'initialize' => ['apps\common\libraries\ThinkOrmInitialize::handle'],
```

接下来就可以正常在代码中使用 `think-orm` 了。

官方文档

- https://www.kancloud.cn/manual/thinkphp5_1/353998
- https://github.com/top-think/think-orm

guzzlehttp

基于 GuzzleHttp 的 HttpClient 组件 (带协程，连接池)

点击下面的网址，根据说明安装使用。

https://github.com/mixstart/mixphp-guzzlehttp

psr-log

改造为基于 PSR-3 的日志组件

`MixPHP` 的 `mix\base\Log` 组件本来就是支持 `PSR-3` 的，但是为了保持框架的轻量级特性，所以框架没有直接依赖 `psr/log` 包，因此用户只需增加 `LoggerInterface` 的 implements 即可。

改造步骤

首先安装 `psr/log` 库。

```
composer require psr/log
```

然后创建一个继承 `mix\base\Log` 类，实现 `Psr\Log\LoggerInterface` 接口的日志组件。

```
<?php
namespace apps\httpd\components;

use Psr\Log\LoggerInterface;
use mix\base\Log;

class Logger extends Log implements LoggerInterface
{
}
```

- 然后修改配置文件的注册：

class `mix\base\Log` 修改为 `apps\httpd\components\Logger` 即可。

```
// 日志
'log'      => [
    // 类路径
    'class'       => 'apps\httpd\components\Logger',
    // 日志记录级别
    'level'       => ['emergency', 'alert', 'critical', 'error', 'warning', 'no
tice', 'info', 'debug'],
    // 日志目录
    'dir'         => 'logs',
    // 日志轮转类型
    'rotate'      => apps\httpd\components\Logger::ROTATE_DAY,
    // 最大文件尺寸
    'maxFileSize' => 0,
],
```

安全建议

安全建议

框架的安全性通常是非常重要的，而大部分的安全问题都来源于编码人员对用户输入参数的信任而未作数据验证，MixPHP 提供了验证器来避免安全问题。
Web 方面通常的安全风险为：SQL注入、跨站脚本攻击，下面分别介绍下 MixPHP 的安全机制。

SQL注入

MixPHP 通过两层机制来防止SQL注入：

- 验证器：使用验证器，能对用户的输入做全面的验证，使不安全的注入数据无法进入到数据库去执行。
- PDO组件：该组件基于 pdo 扩展，组件内部封装了 语句预处理 功能，使SQL与参数分离，确保不会发生 SQL 注入。

跨站脚本攻击 (XSS)

跨站脚本攻击通常是在 string 类型的字段进入数据库，MixPHP 的模型验证器有专门针对 XSS 的处理。

- 验证器：string 验证器的 filter 参数提供了 'strip_tags', 'htmlspecialchars' 两个方法专门用于过滤或转义跨站脚本攻击。

常见问题

启动多个 HTTP 服务器

启动多个 HTTP 服务器

框架中的 `HTTP` 模块中都有集成一个单独的 `mix-httpd` 服务器，由于要同时启动多个，所以需要配置不同的端口，然后给每个服务都设置对应的 `Nginx` 代理。

第一步：

修改每一个 `mix-httpd` 的端口不重复，配置文件路径：

```
apps/[HTTP模块]/config/httpd.php
```

然后启动全部的 `mix-httpd` 。

第二步：

为每个 `mix-httpd` 配置 Nginx 代理。

```
server {
    server_name my.test.com;
    listen 80;
    root /data/mixphp/apps/my/public/;
    index index.html;

    location = / {
        rewrite ^(.*)$ /index last;
    }

    location / {
        proxy_http_version 1.1;
        proxy_set_header Connection "keep-alive";
        proxy_set_header Host $http_host;
        proxy_set_header Scheme $scheme;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        if (!-e $request_filename) {
             proxy_pass http://127.0.0.1:9501;
        }
    }
}

server {
    server_name shop.test.com;
    listen 80;
    root /data/mixphp/apps/shop/public/;
```

```
    index index.html;

    location = / {
        rewrite ^(.*)$ /index last;
    }

    location / {
        proxy_http_version 1.1;
        proxy_set_header Connection "keep-alive";
        proxy_set_header Host $http_host;
        proxy_set_header Scheme $scheme;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        if (!-e $request_filename) {
            proxy_pass http://127.0.0.1:9511;
        }
    }
}
```

第三步：

重新启动 nginx 即可。

连接多个数据库

连接多个数据库

通常需求中连接两个数据库有以下两种情况：

- 两个库都频繁使用。
- 只有一个库频繁使用，另一个只有少数业务使用。

两个库都频繁使用

请阅读 “框架核心” - “门面” 章节的 “同类型组件多源切换” 的内容。

只有一个库频繁使用，另一个只有少数业务使用

请阅读 “框架核心” - “对象” 章节的 “通过配置实例化对象” 的内容。

如何设置跨域

如何跨域

跨域只需在前置中间件增加跨域的代码，如下：

```
<?php

namespace apps\httpd\middleware;

use mix\facades\Request;
use mix\facades\Response;

/**
 * 前置中间件
 * @author 刘健
 */
class BeforeMiddleware
{

    public function handle($callable, \Closure $next)
    {
        // 添加中间件执行代码
        list($controller, $action) = $callable;

        // 跨域设置
        $origin      = Request::header('origin');
        $allowOrigin = [
            'http://www.test.com',
            'http://www.test1.com',
        ];
        if (in_array($origin, $allowOrigin)) {
            Response::setHeader('Access-Control-Allow-Origin', $origin);
            Response::setHeader('Access-Control-Allow-Headers', 'Origin, X-Requ
ested-With, Content-Type, Accept, Authorization');
            Response::setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT
, DELETE, OPTIONS, PATCH');
        }
        if (Request::isOptions()) {
            return '';
        }

        // 执行下一个中间件
        return $next();
    }

}
```

- Access-Control-Allow-Origin：允许跨域的域名
- Access-Control-Allow-Headers：允许跨域的请求头
- Access-Control-Allow-Methods：允许跨域的请求方法

mix-httpd service stop 无效

mix-httpd service stop 无效

当出现以下情况：刚启动 mix-httpd，service stop 时就提示没有在运行。

```
[www@localhost bin]# mix-httpd service start -d
                                _____
_______ ___ _____ ___ _____  / /_  ____
__/ __ `__ \/ /\ \/ / / __ \/ __ \/ __ \
_/ / / / / / /\ \/ / /_/ / / / / /_/ /
/_/ /_/ /_/_/ /_/\_\/ .___/_/ /_/ .___/
                   /_/          /_/

[2018-03-09 11:28:24] Server    Name: mix-httpd
[2018-03-09 11:28:24] PHP    Version: 5.6.34
[2018-03-09 11:28:24] Swoole Version: 1.9.21
[2018-03-09 11:28:24] Listen    Addr: 127.0.0.1
[2018-03-09 11:28:24] Listen    Port: 9501

[www@localhost bin]# mix-httpd service stop
mix-httpd is not running.
```

原因

mix-httpd 的 stop 原理是通过 pid 文件，pid 文件保存在：

```
/var/run/mix-httpd.pid
```

因为非 root 账号无该文件夹的写权限，导致 pid 文件无法生成，使 mix-httpd 无法得知自己的执行状态。

解决方法

- 请使用 sudo mix-httpd service start -d 启动。
- 或者切换到 root 账号后 mix-httpd service start -d 启动。

No such file or directory

No such file or directory

执行 `bin` 目录的入口文件，或者 `install.sh` 时，抛出以下错误：

```
[root@localhost bin]# ./mix-httpd
: No such file or directory
```

原因：这是因为代码下载到 windows 后，文件回车换行默认为： `CRLF` ，而 Liunx 的 Shell 执行文件只能是 `LF` 。

解决：使用编辑器将文件修改为 `LF` 再上传即可。

推进计划

推进计划

MixPHP 的理念 "普及 PHP 常驻内存型解决方案，促进 PHP 往更后端发展" 的前半句已经开发完成，现在向后半句出发，接下来的开发路线将步入 TP5/CI/Yii2 等传统 Web 框架不太擅长或没有涉及的领域。

第一阶段：

- 框架架构设计 -- 完成
- 实现Web/Console执行流程 -- 完成
- 实现核心组件 -- 完成

第二阶段：

- 构建模型 -- 完成
- 增加数据库相关组件 -- 完成

第三阶段：

- 构建WebSite常用组件 -- 完成

第四阶段：

全面优化与完善。

- 内部流程优化 -- 完成
- Pdo主从 -- 完成

第五阶段：

构建WebAPI常用组件。

- token -- 完成

第六阶段：

构建Console常用类库。

- 守护进程 -- 完成
- 多进程 -- 完成
- 消息队列消费 -- 完成

第七阶段：

- 增加 Http 同步客户端 -- 完成
- 增加 WebSocket 支持 -- 完成

V1.1

- HTTP 与 CLI 协程化开发 -- 完成

> 已经发布

V1.2

> 正在开发中

- 连接池改造，与 golang 的实现方式看齐。
- flags 改造，与 golang 的实现方式看齐。
- 命令修改为命令+子命令的方式，支持子命令可为空。
- 多进程支持查看进程队列状态，提供命令查看与方法调用两种方式。
- 命名空间全部修改为大写。

V1.3

- WebSocket 协程化，修改为更 MVC 的开发方式。

以下是待评估增加的功能

- 验证规则生成器。
- UDP日志服务器。

文档历史

开发文档历史记录

本文档每次描述的都是最新版本，所以旧版本的用户如果需要查看文档，请移步 GitHub 中下载：

>> 去 GitHub 查看旧版本开发文档 <<

> 当然最好的方式还是升级到最新版本。