

前言

[下载](#)

[文档](#)

[2.x 文档](#)



[贡献](#)

[Admin](#)

[体验](#)

PhalApi官网请见：www.phalapi.net.

最新文档

- [官方在线版 - 推荐](#)
- [PDF离线版](#)
- [HTML离线版](#)
- [Markdown源代码版](#)

什么是PhalApi 2.x?

PhalApi，简称π框架，是一个PHP轻量级开源接口框架，专注于接口开发，致力让接口开发更简单。它：

- 致力于快速、稳定、持续交付有价值的接口服务
- 关注于测试驱动开发、领域驱动设计、极限编程、敏捷开发
- 有众多的扩展类库，与更多开源项目一起提供高效便捷的解决方案
- 支持HTTP、SOAP和RPC协议，可用于快速搭建微服务、RESTful接口或Web Services

PhalApi现存有两大系列版本。分别是经典的第一代版本，即1.x系列版本，主要是使用了较为古老传统的做法；以及全新的第二代版本，即2.x系列版本，主要区别是：

- 使用了[composer](#)统一管理依赖包
- 引入了命名空间
- 遵循[PSR-4](#)规范

温馨提示：在本开发文档中，若未特别标明，PhalApi指PhalApi 1.x 版本和PhalApi 2.x 版本。

PhalApi有哪些特点？

PhalApi是一个很酷的开源框架，对它越了解，越能发现它的炫酷所在。以下是部分关键的特性。

特点1：学习成本低

PhalApi始终坚持KISS原则，并遵循Unix哲学中的最小立异原则。除了遵循国际惯例，采用约定俗成的做法，PhalApi还在设计时优先考虑大家所熟悉的方案。例如，接口返回结果格式便是路人皆知的JSON格式。对于刚接触PHP编程语言的初级开发同学，甚至是之前未曾接触过PHP的客户端开发同学，根据以往的学习经验，大部分情况下，可以在一周内完成PhalApi框架的基础学习，并投入到实际项目开发中。

特点2：自动生成的在线接口文档

按框架指定的格式完成接口代码编写后，PhalApi会自动生成在线接口列表文档和在线接口详情文档，以方便客户端实时查看最新的接口签名和返回字段。

自动生成的在线文档主要有两类：

- 在线接口列表文档



#	接口服务	接口名称	更多说明
1	User.GetBaseInfo	获取用户基本信息	用于获取单个用户
2	User.GetMultiBaseInfo	批量获取用户基本信息	用于获取多个用户

© Powered By PhalApi 1.4.1

- 在线接口详情文档

接口 : Default.Index

[默认接口服务](#)

[接口说明](#)

```
//请使用@desc 注释
```

接口参数

参数名字	类型	是否必须	默认值	其他	说明
username	字符串	可选	PHPer		

特点3：众多可重用的扩展类库

PhalApi框架扩展类库，是各自独立，可重用的组件或类库，可以直接集成到PhalApi开发项目，从而让项目开发人员感受搭建积木般的编程乐趣，降低开发成本。

目前，已经提供的扩展类库有40+个，包括：微信公众号开发扩展、微信小程序开发扩展、支付扩展、上传扩展、Excel表格和Word文档扩展等。

温馨提示：部分扩展类库需要调整移植到PhalApi 2.x风格方能使用。

特点4：活跃的开源社区

PhalApi不是“我们”的框架，而是我们大家每个人的开源框架。PhalApi开源社区非常活跃，除了有1000+人的实时交流群，还有自主搭建的[问答社区](#)，以及近百名参与贡献的同学。

PhalApi 2.x的学习资料目前还在陆续补充中，但依然可以参考PhalApi 1.x 版本系列丰富的学习资料，有：[开发文档](#)、[视频教程](#)、[《初识PhalApi》免费电子书](#)、[博客教程](#)等。

适用场景与范围

PhalApi代码开源、产品开源、思想开源，请放心使用。

PhalApi适用的场景，包括但不限于：

- 为移动App（包括iOS、iPad、Android、Windows Phone等终端）提供接口服务
- 用于搭建接口平台系统，提供聚合类接口服务，供其他后端系统接入使用
- 为前后端分离的H5混合页面应用，提供Ajax异步接口

对于架构无关、专注架构及提升架构这三种情况，PhalApi都能胜任之。

正如其他负责任的开源框架一样，PhalApi也有其不适宜使用的时机。包括但不限于：

- 开发CLI项目（但已提供支持命令行项目开发的[CLI扩展类库](#)）

- 开发网站项目，即有界面展示和视图渲染（但已提供支持视图渲染的[View扩展类库](#)）
- 对数据严谨性要求高，如金融行业的相关项目，毕竟PHP是弱类型语言

文档目标读者

本开发文档的目标读者是：

- 初次接触PhalApi框架的开发同学
- 正在使用PhalApi进行项目开发的同学
- 任何想了解或学习PhalApi框架的同学

联系我们

关于本开发文档，任何问题，都可反馈到[这里](#)，谢谢！

下载与安装

PhalApi 2.x 与 PhalApi 1.x 系列一样，要求 PHP >= 5.3.3。

快速安装

PhalApi 2.x 版本的安装很简单，有两种方式。

composer一键安装

安装Composer

如果还没有安装 Composer，你可以按 [getcomposer.org](#) 中的方法安装。在 Linux 和 Mac OS X 中可以运行如下命令：

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

温馨提示：关于composer的使用，请参考[Composer 中文网](#) / [Packagist 中国全量镜像](#)。

安装PhalApi 2.x

使用composer创建项目的命令，可实现一键安装。

```
$ composer create-project phalapi/phalapi
```

手动下载安装

或者，也可以进行手动安装。首先下载[phalapi](#)项目**master-2x分支**源代码。下载解压后，进行可选的composer更新，即：

```
$ composer update
```

温馨提示：为提高友好度，phalapi中已带有缺省vendor安装包，从而减轻未曾接触过composer开发同学的学习成本。即便composer安装失败，也可正常运行PhalApi 2.x。

配置

Nginx配置

如果使用的是Nginx，可参考以下配置。

```
server {  
    listen 80;  
    server_name dev.phalapi.net;  
    root /path/to/phalapi/public;  
    charset utf-8;  
  
    location / {  
        index index.php;  
    }  
  
    location ~ \.php$ {  
        fastcgi_split_path_info ^(.+\.php)(/.+)$;  
        fastcgi_pass 127.0.0.1:9000;  
        fastcgi_index index.php;  
        include fastcgi_params;  
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;  
    }  
  
    access_log logs/dev.phalapi.net.access.log;  
    error_log logs/dev.phalapi.net.error.log;  
}
```

重启Nginx并配置本地HOSTS后，可通过以下链接，访问默认接口服务。

<http://dev.phalapi.net>

温馨提示：推荐将访问根路径指向/path/to/phalapi/public。后续开发文档中，如无特殊说明，均约定采用此配置方式。

Apache配置

如果使用的是Apache，可参考以下配置。目录结构：

```
htdocs
└── phalapi
    └── .htaccess
```

.htaccess内容：

```
<IfModule mod_rewrite.c>
    RewriteEngine on
    RewriteBase /
    RewriteCond %{HTTP_HOST} ^dev.phalapi.net$ 
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteCond %{REQUEST_URI} !^/phalapi/public/
    RewriteRule ^(.*)$ /phalapi/public/$1
    RewriteRule ^(/)?$ index.php [L]
</IfModule>
```

XAMPP配置

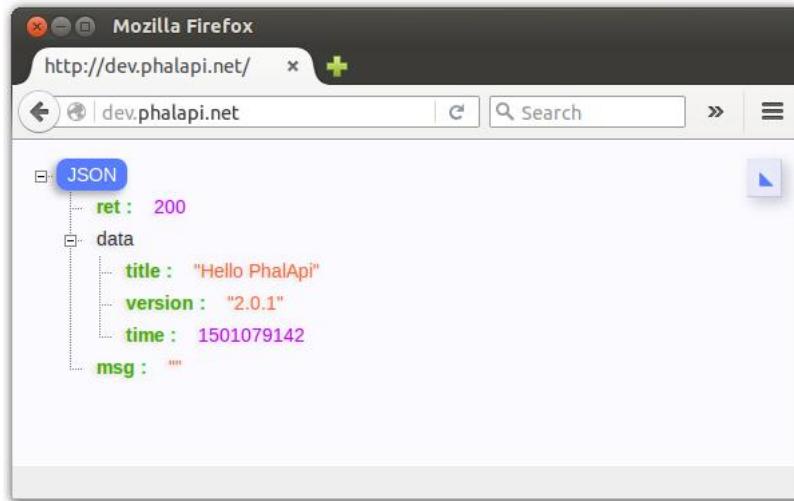
如果使用的是XAMPP集成环境，只需要将项目源代码phalapi整个目录复制到xampp的htdocs目录下即可。打开XAMPP控制面板并启动Apache后，便可通过以下链接，访问默认接口服务。

<http://localhost/phalapi/public/>

以上不管何种配置，正常情况下，访问默认接口服务可以看到类似这样的输出：

```
{
    "ret": 200,
    "data": {
        "title": "Hello PhalApi",
        "version": "2.0.1",
        "time": 1501079142
    },
    "msg": ""
}
```

运行效果，截图如下：



至此，安装完毕！

如何升级PhalApi 2.x框架？

在composer的管理下，升级PhalApi 2.x 版本系列非常简单。只需要修改composer.json文件，指定相应的版本即可。PhalApi的框架内核项目在[phalapi/kernel](#)，你可以指定版本，也可以跟随最新版本。

例如，当需要指定PhalApi 2.0.1版本时，可以这样配置：

```
{
    "require": {
        "phalapi/kernel": "2.0.1"
    }
}
```

当需要保持最新版本时，则可以改成：

```
{
    "require": {
        "phalapi/kernel": "2.*.*"
    }
}
```

```
}
```

这样，当PhalApi 2.x 有版本更新时，只需执行composer更新操作即可。对应命令操作为：

```
$ composer update
```

至此，升级完毕！

温馨提示：关于composer版本的说明，可参考[Composer中文文档 - 包版本](#)。

运行Hello World

此文章假设你已成功安装PhalApi2项目，如果尚未安装，可阅读[下载与安装](#)。

编写一个接口

在PhalApi 2.x 版本中，项目源代码放置在/path/to/PhalApi2/src目录中。里面各个命名空间对应一个子目录，默认命名空间是app，里面主要有Api、Domain、Model这三个目录以及存放函数的functions.php文件。例如像是这样的目录结构：

```
./src/
└── app
    ├── Api
    ├── Domain
    ├── functions.php
    └── Model
```

当需要新增一个接口时，先要在Api层添加一个新的接口文件。例如对于Hello World示例，可以使用你喜欢的编辑器创建一个./src/app/Api/Hello.php文件，并在里面放置以下代码。

```
// 文件 ./src/app/Api/Hello.php
<?php
namespace App\Api;

use PhalApi\Api;

class Hello extends Api {

    public function world() {
        return array('title' => 'Hello World!');
    }
}
```

编写接口时，需要特别注意：

- 1、默认所在命名空间必须为App\Api。
- 2、具体实现的接口类必须是PhalApi\Api的子类。

访问一个接口

通常情况下，建议可访问的根路径设为/path/to/PhalApi2/public。若未设置，此时接口访问的URL格式为：接口域名/public/?s=Namespace.Class.Action。其中，s参数用于指定待请求的接口服务，由三部分组成。分别是：

组成部分	是否必须	默认值	说明
Namespace	可选	App	Api命名空间前缀，多级命名空间时用下划线分割
Class	必须	无	待请求的接口类名，通常首字母大写
Action	必须	无	待请求的接口类方法名，通常首字母大写。若Class和Action均未指定时，默认为Site.Index

温馨提示：s参数为service参数的缩写，即使使用?s=Class.Action等效于?service=Class.Action，两者都存在时优先使用service参数。

例如，上面新增的Hello World接口的访问链接为：

```
http://dev.phalapi.net/?s>Hello.World
```

或者可以使用完整的写法，带上命名空间App：

```
http://dev.phalapi.net/?s=App>Hello.World
```

接口返回

默认情况下，接口的结果以JSON格式返回，并且返回的顶级字段有状态码ret、业务数据data，和错误提示信息msg。其中data字段对应接口类方法返回的结果。如Hello World示例中，返回的结果是：

```
{"ret":200,"data":{"title":"Hello World!"}, "msg":""}
```

JSON可视化后是：

```
{
    "ret": 200,
    "data": {
        "title": "Hello World!"
    },
}
```

```
        "msg": ""  
    }  
}
```

恭喜！你已顺便完成PhalApi 2.x 简单的接口开发了！# 如何请求接口服务

HTTP协议下的请求方式

对于PhalApi，默认是通过HTTP协议进行通信的。根据接口服务的具体实现，可以使用GET或POST方式请求。

访问入口

如前所言，PhalApi推荐将系统对外可访问的根目录设置为/path/to/phalapi/public。PhalApi的统一访问入口文件是/path/to/phalapi/public/index.php文件。

当配置的域名为：dev.phalapi.net，并且已将根目录设置到public，此时访问的URL是：

http://dev.phalapi.net

当未配置域名，亦未配置根目录时，此时访问的URL是（显然更长更不优雅）：

http://localhost/phalapi/public/index.php

如果尚未安装，请先阅读[下载与安装](#)。

如何指定待请求的接口服务？

默认情况下，可以通过s参数指定待请求的接口服务，当s未传时，缺省使用默认接口服务，即：App.Site.Index。以下三种方式是等效的，都是请求默认接口服务。

- 未传s参数
- ?s=Site.Index，省略命名空间，默认使用App
- ?s=App.Site.Index，带有命名空间前缀

也就是说，当请求除默认接口服务以外的接口服务时，其格式可以二选一：

- ?s=Class.Action
- 或者：?s=Namespace.Class.Action

其中，Namespace表示命名空间前缀，Class为接口服务类名，Action为接口服务方法名，这三者通常首字母大写，并使用英文点号分割。最终执行的类方法是：Namespace\Api\Class::Action()。需要注意的是：

温馨提示：s参数为service参数的缩写，即使用?s=Class.Action等效于?service=Class.Action，两者都存在时优先使用service参数。

关于Namespace命名空间

Namespace是指命名空间中\Api/的前半部分。并且需要在根目录下的composer.json文件中进行autoload的注册，以便能正常自动加载类文件。如默认已经注册的App命名空间：

```
{  
    "autoload": {  
        "psr-4": {  
            "App\\": "src/app"  
        }  
    }  
}
```

当命名空间存在子命名空间时，在请求时使用下划线分割。反过来，当不存在多级命名空间时，命名空间不应该含有下划线。

关于Class接口服务类名

Class接口服务类名是指命名空间中\Api/的后半部分，并且必须是[PhalApi\Api](#)的子类。当命名空间存在子命名空间时，在请求时同样改用下划线分割。类似的，当不存在多级命名空间时，命名空间不应该含有下划线。

关于Action接口服务方法名

待请求的Action，应该是public访问级别的类方法，并且不能是[PhalApi\Api](#)已经存在的方法。

一些示例

以下是一些综合的示例。

PhalApi 2.x 请求的s参数	对应的文件	执行的类方法
无	./src/app/Api/Site.php	App\Api\Site::Index()
?s=Site.Index	./src/app/Api/Site.php	App\Api\Site::index()
?s=Weibo.Login	./src/app/Api/Weibo.php	App\Api\Weibo::login()
?s=User.Weibo.Login	./src/user/Api/Weibo.php	User\Api\Weibo::login()
?s=Company_User.Third_Weibo.Login	./src/company_user/Api/Third/Weibo.php	Company\User\Api\Third\Weibo::login()

上面示例中假设，已经在composer.json中配置有：

```
{  
    "autoload": {  
        "psr-4": {  
            "App\\": "src/app",  
            "User\\": "src/user",  
            "Company\\User\\": "src/company_user"  
        }  
    }  
}
```

扩展：如何定制接口服务的传递方式？

虽然我们约定统一使用?s=Namespace.Class.Action的格式来传递接口服务名称，但如果项目有需要，也可以采用其他方式来传递。例如类似于Yii框架的请求格式：?r=Namespace/Class/Action。

如果需要定制传递接口服务名称的方式，可以重写PhalApi\Request::getService()方法。以下是针对改用斜杠分割，并换用r参数名字的实现代码片段。

```
// 文件 ./src/app/Common/Request.php  
  
<?php  
namespace App\Common;  
  
class Request extends \PhalApi\Request {  
  
    public function getService() {  
        // 优先返回自定义格式的接口服务名称  
        $service = $this->get('r');  
        if (!empty($service)) {  
            $namespace = count(explode('/', $service)) == 2 ? 'App.' : '';  
            return $namespace . str_replace('/', '.', $service);  
        }  
  
        return parent::getService();  
    }  
}
```

实现好自定义的请求类后，需要在项目的DI配置文件[/config/di.php](#)进行注册。在最后的加上一行：

```
$di->request = new App\Common\Request();
```

这时，便可以通过新的方式进行接口服务的请求的了。即：

原来的方式	现在的方式
?s=Site.Index	?r=Site/Index
?s=App.Site.Index	?r=App/Site/Index
?s=Hello.World	?r=Hello/World
?s=App.Hello.World	?r=App>Hello/World

这里有几个注意事项：

- 1、重写后的方法需要转换为原始的接口服务格式，即：Namespace.Class.Action，注意别遗漏命名空间。
- 2、为保持兼容性，在取不到自定义的接口服务名称参数时，应该返回parent::getService()。

是不是觉得很好玩？可以立马亲自尝试一下哦。定制你最喜欢的请求方式。

接口响应与在线调试

对于接口响应，PhalApi默认使用了HTTP + JSON。通过HTTP/HTTPS协议进行通讯，返回的结果则使用JSON格式进行传递。正常情况下，当接口服务正常响应时，如前面的Hello World接口，可能看到以下这样的响应头部信息和返回内容。

```
HTTP/1.1 200 OK  
Content-Type: application/json; charset=utf-8
```

... ...

```
{"ret":200,"data":{"title":"Hello World!","msg":""}
```

而当接口项目抛出了未捕捉的异常，或者因PHP语法问题而出现Error时，则没有内容返回，并且得到一个500的响应状态码。类似如下：

```
HTTP/1.1 500 Internal Server Error
```

响应结构 data-ret-msg

回顾一下默认接口服务返回的内容。类似如下：

```
{  
    "ret": 200,  
    "data": {  
        "title": "Hello World!",  
        "content": "PHPer您好，欢迎使用PhalApi！",  
        "version": "2.0.0",  
        "time": 1499477583  
    },  
    "msg": ""  
}
```

ret字段是返回状态码，200表示成功；data字段是项目提供的业务数据，由接口开发人员定义；msg是异常情况下的错误提示信息。下面分别说之。

业务数据 data

业务数据data为接口和客户端主要沟通对接的数据部分，可以为任何类型，由接口开发人员定义。但为了更好地扩展、向后兼容，建议都使用可扩展的集合形式，而非原生类型。也就是说，应该返回一个数组，而不应返回整型、布尔值、字符串这些基本类型。

业务数据主要是在Api层返回，即对应接口类的方法的返回结果。如下面的默认接口服务?`s=Site.Index`的实现代码。

```
<?php
namespace App\Api;

use PhalApi\Api;

class Site extends Api {

    public function index() {
        return array(
            'title' => 'Hello World!',
            'content' => \PhalApi\T('Hi {name}, welcome to use PhalApi!', array('name' => $this->username)),
            'version' => PHALAPI_VERSION,
            'time' => $_SERVER['REQUEST_TIME'],
        );
    }
}
```

实际上，具体的业务数据需要一段复杂的处理，以满足特定业务场景下的需要。Api层需要与Domain层和Model层共同协作，完成指定的功能。这里暂且知道接口结果是在Api层返回，对应接口类成员方法返回的结果即可。

返回状态码 ret

返回状态码ret，用于表示接口响应的情况。参照自HTTP的状态码，ret主要分为四大类：正常响应、重定向、非法请求、服务器错误。

分类	ret范围	基数	说明
正常响应	200 ~ 299	200	表示接口服务正常响应
重定向	300 ~ 399	300	表示重定向，对应异常类 RedirectException 的异常码
非法请求	400 ~ 499	400	表示客户端请求非法，对应异常类 BadRequestException 的异常码
服务器错误	500 ~ 599	500	表示服务器内容错误，对应异常类 InternalServerException 的异常码

正常响应时，通常返回`ret = 200`，并且同时返回`data`部分的业务数据，以便客户端能实现所需要的业务功能。

值得注意的是，抛出的异常应该继承于[PhalApi\Exception](#)类，并且构造函数的第一个参数，是返回给客户端的错误提示信息，对应下面将讲到的`msg`字段。第二个参数是返回状态码的叠加值，也就是说最终的`ret`状态码都会在400的基数上加上这个叠加值，即：`401 = 400 + 1`。

例如，常见地，当签名失败时可以返回一个401错误，并提示“签名失败”。

```
<?php
namespace App\Api;

use PhalApi\Api;
use PhalApi\Exception\BadRequestException;

class Hello extends Api {

    public function fail() {
        throw new BadRequestException('签名失败', 1);
    }
}
```

会得到以下结果输出：

```
{
    "ret": 401,
    "data": [],
    "msg": "Bad Request: 签名失败"
}
```

错误提示信息 msg

当接口不是正常响应，即`ret`不在2XX系列内时，`msg`字段会返回相应的错误提示信息。即当有异常触发时，会自动将异常的错误信息作为错误信息`msg`返回。

如何设置JSON中文输出？

默认情况下，输出的中文会被转换成Unicode，形如`\\uXXXX`，如：

```
"content": "PHPer\\u60a8\\u597d\\uff0c\\u6b22\\u8fce\\u4f7f\\u7528PhalApi\\uff01"
```

虽然不影响使用，但不便于查看。如果需要不被转码，可以使用[JSON_UNESCAPED_UNICODE](#)选项进行配置。重新注册`DI()->response`并指定配置选项。例如可以：

```
$di->response = new \PhalApi\Response\JsonResponse(JSON_UNESCAPED_UNICODE); // 中文显示
```

设置后，重新请求，将会看到：

```
"content": "PHPer您好，欢迎使用PhalApi！"
```

类似地，还可以设置更多其他的选项，如追加强制使用对象格式：

```
$di->response = new \PhalApi\Response\JsonResponse(JSON_UNESCAPED_UNICODE | JSON_FORCE_OBJECT); // 中文显示且强制对象格式
```

扩展：如何使用其他返回格式？

除了使用JSON格式返回外，还可以使用其他格式返回结果。

例如在部分H5混合应用页面进行异步请求的情况下，客户端需要服务端返回JSONP格式的结果，则可以这样在DI配置文件./config/di.php中去掉以下注释。

```
// 支持JsonP的返回
if (!empty($_GET['callback'])) {
    $di->response = new \PhalApi\Response\JsonpResponse($_GET['callback']);
}
```

目前，PhalApi 2.x 已经支持的响应格式有：

响应格式	实现类
JSON格式	PhalApi\Response\JsonResponse
JSONP格式	PhalApi\Response\JsonpResponse
XML格式	PhalApi\Response\XmlResponse
控制台格式	PhalApi\Response\ExplorerResponse

当需要返回一种当前PhalApi没提供的格式，需要返回其他格式时，可以：

- 1、实现抽象方法[PhalApi\Response::formatResult\(\\$result\)](#)并返回格式化后结果
- 2、在./config/di.php文件中重新注册\PhalApi\DI()->response服务

在线调试

开启调试模式

开启调试模式很简单，主要有两种方式：

- **单次请求开启调试：**默认添加请求参数`&__debug__=1`
- **全部请求开启调试：**把配置文件`./Config/sys.php`文件中的配置改成`'debug' => true,`

调试信息有哪些？

正常响应的情况下，当开启调试模式后，会返回多一个`debug`字段，里面有相关的调试信息。如下所示：

```
{
    "ret": 200,
    "data": [
    ],
    "msg": "",
    "debug": {
        "stack": [ // 自定义埋点信息
        ],
        "sqls": [ // 全部执行的SQL语句
        ]
    }
}
```

温馨提示：调试信息仅当在开启调试模式后，才会返回并显示。

在发生未能捕捉的异常时，并且开启调试模式后，会将发生的异常转换为对应的结果按结果格式返回，即其结构会变成以下这样：

```
{
    "ret": 0, // 异常时的错误码
    "data": [],
    "msg": "", // 异常时的错误信息
    "debug": {
        "exception": [ // 异常时的详细堆栈信息
        ],
        "stack": [ // 自定义埋点信息
        ],
        "sqls": [ // 全部执行的SQL语句
        ]
    }
}
```

• 查看全部执行的SQL语句

`debug.sqls`中会显示所执行的全部SQL语句，由框架自动搜集并统计。最后显示的信息格式是：

[序号 - 当前SQL的执行时间ms] 所执行的SQL语句及参数列表

示例：

```
[1 - 0.32ms]SELECT * FROM tbl_user WHERE (id = ?); -- 1
```

表示是第一条执行的SQL语句，消耗了0.32毫秒，SQL语句是`SELECT * FROM tbl_user WHERE (id = ?);`，其中参数是1。

• 查看自定义埋点信息

`debug.stack`中埋点信息的格式如下：

[#序号 - 距离最初节点的执行时间ms - 节点标识] 代码文件路径(文件行号)

示例:

```
[#0 - 0ms]/path/to/phalapi/public/index.php(6)
```

表示，这是第一个埋点（由框架自行添加），执行时间为0毫秒，所在位置是文件/path/to/phalapi/public/index.php的第6行。即第一条的埋点发生在框架初始化时。

与SQL语句的调试信息不同的是，自定义埋点则需要开发人员根据需要自行纪录，可以使用全局追踪器PhalApi\DI()>tracer进行纪录，其使用如下：

```
// 添加纪录埋点  
PhalApi\DI()>tracer->mark();  
  
// 添加纪录埋点，并指定节点标识  
PhalApi\DI()>tracer->mark('DO_SOMETHING');
```

通过上面方法，可以对执行经过的路径作标记。你可以指定节点标识，也可以不指定。对一些复杂的接口，可以在业务代码中添加这样的埋点，追踪接口的响应时间，以便进一步优化性能。当然，更专业的性能分析工具推荐使用XHprof。

参考：用于性能分析的[XHprof扩展类库](#)。

• 查看异常堆栈信息

当有未能捕捉的接口异常时，开启调试模式后，框架会把对应的异常转换成对应的返回结果，并在debug.exception中体现。而不是像正常情况直接500，页面空白。这些都是由框架自动处理的。

例如，让我们故意制造一些麻烦，手动抛出一个异常。

```
class Hello extends Api {  
  
    public function fail() {  
        throw new Exception('这是一个演示异常调试的示例', 501);  
    }  
}
```

再次请求后，除了SQL语句和自定义埋点信息外，还会看到这样的异常堆栈信息。然后便可根据返回的异常信息进行排查定位问题。

• 添加自定义调试信息

当需要添加其他调试信息时，可以使用PhalApi\DI()>response->setDebug()进行添加。

如：

```
class Hello extends Api {  
  
    public function fail() {  
        $x = 'this is x';  
        $y = array('this is y');  
        \PhalApi\DI()>response->setDebug('x', $x);  
        \PhalApi\DI()>response->setDebug('y', $y);  
    }  
}
```

请求后，可以看到：

```
"debug": {  
    "x": "this is x",  
    "y": [  
        "this is y"  
    ]  
}
```

Api接口服务层

Api接口层称为接口服务层，负责对客户端的请求进行响应，处理接收客户端传递的参数，进行高层决策并对领域业务层进行调度，最后将处理结果返回给客户端。

接口参数规则配置

接口参数，对于接口服务本身来说，是非常重要的。对于外部调用的客户端来说，同等重要。对于接口参数，我们希望能够既减轻后台开发对接口参数获取、判断、验证、文档编写的痛苦；又能方便客户端快速调用，明确参数的意义。由此，我们引入了**参数规则**这一概念，即：通过配置参数的规则，自动实现对参数的获取和验证，同时自动生成在线接口文档。

一个简单的示例

假设我们现在需要提供一个用户登录的接口，接口参数有用户名和密码，那么新增的接口类和规则如下：

```
// 文件 ./src/app/Api/User.php  
<?php  
namespace App\Api;  
  
use PhalApi\Api;  
  
class User extends Api {  
  
    public function getRules() {  
        return array(  
    }
```

```

        'login' => array(
            'username' => array('name' => 'username'),
            'password' => array('name' => 'password'),
        ),
    ),
}

public function login() {
    return array('username' => $this->username, 'password' => $this->password);
}
}

```

当请求此接口服务，并类似这样带上username和password参数时：

```
?s=User.Login&username=dogstar&password=123456
```

就可以得到这样的返回结果。

```
{"ret":0,"data":{"username":"dogstar","password":"123456"},"msg":""}
```

接口返回

回顾一下，在PhalApi中，接口返回的结果的结构为：

```
{
    "ret": 200, // 状态码
    "data": { // 业务数据
    },
    "msg": "" // 错误提示信息
}
```

正常情况下的返回

正常情况下，在Api层返回的数据结果，会在返回结果的data字段中体现。例如：

```
class Hello extends Api {

    public function world() {
        return array('title' => 'Hello World!');
    }
}
```

对应：

```
{
    "ret": 200,
    "data": {
        "title": "Hello World!"
    },
    "msg": ""
}
```

成功返回时，状态码ret为200，并且错误信息msg为空。

失败情况下的返回

对于异常情况，包括系统错误或者应用层的错误，可以通过抛出[PhalApi\Exception](#)系列的异常，中断请求并返回相关的错误信息。例如：

```
class Hello extends Api {

    public function fail() {
        throw new BadRequestException('签名失败', 1);
    }
}
```

会得到以下结果输出：

```
{
    "ret": 401,
    "data": [],
    "msg": "Bad Request: 签名失败"
}
```

扩展：返回JSONP、XML等其他格式

JSONP返回格式

如果需要支持JSONP返回格式，可以将 ./config/di.php 中的以下代码注释去掉：

```
// 支持JsonP的返回
if (!empty($_GET['callback'])) {
    $di->response = new \PhalApi\Response\JsonpResponse($_GET['callback']);
}
```

然后在请求时，传入回调函数的名称callback，即可返回JSONP格式。例如请求：

```
http://dev.phalapi.net/?s>Hello.World&callback=test
```

返回：

```
test({"ret":200,"data":{"title":"Hello World!"},"msg":""})
```

XML返回格式

如果需要返回XML格式，需要将`\PhalApi\DI()->response`切换到XML响应类，如：

```
$di->response = new \PhalApi\Response\XmlResponse();
```

然后，可看到请求的接口返回类似如下：

```
<?xml version="1.0" encoding="utf-8"?><xml><ret><![CDATA[200]]></ret><data><title><![CDATA[Hello World!]]></title></data><msg><![CDATA[]]></msg></xml>
```

其他返回格式

常用的返回格式有如上的JSON、JSONP、XML返回格式。如果需要返回其他的格式，你可以：

- 1、实现`\PhalApi\Response`抽象中的`formatResult($result)`格式化返回结果
- 2、重新注册`\PhalApi\DI()->response`服务

如果希望能由客户端指定返回格式，可通过参数来动态切换。

扩展：修改默认返回的ret/data/msg结构

对于默认返回的字段结构，源代码实现在`\PhalApi\Response::getResult()`方法。相关代码片段如下：

```
abstract class Response {  
    public function getResult() {  
        $rs = array(  
            'ret' => $this->ret,  
            'data' => is_array($this->data) && empty($this->data) ? (object)$this->data : $this->data, // # 67 优化  
            'msg' => $this->msg,  
        );  
        if (!empty($this->debug)) {  
            $rs['debug'] = $this->debug;  
        }  
        return $rs;  
    }  
}
```

如果需要修改默认的ret/data/msg，可以重载此方法，然后进行修改。值得注意的是，你可以基于现有的具体响应类进行继承重载。例如针对JSON的返回格式，先添加一个自己的扩展子类。例如只返回data部分：

```
<?php  
// 新建 ./src/app/Common/MyResponse.php 文件  
  
namespace App\Common;  
  
use PhalApi\Response\JsonResponse;  
  
class MyResponse extends JsonResponse {  
    public function getResult() {  
        // 只返回data部分  
        $rs = parent::getResult();  
        return $rs['data'];  
    }  
}
```

接着在`/config/di.php`文件中重新注册`$di->response`服务为此新的响应类实例。例如：

```
$di->response = new \App\Common\MyResponse();
```

完成子类重载，以及`response`服务重新注册后，再次访问接口，就会看到返回的结果去掉了ret和msg部分。

例如，对于Hello World接口服务，原来返回是：

```
{"ret":200,"data":{"title":"Hello World!"}, "msg":""}
```

现在，返回是（只返回data部分）：

```
{"title":"Hello World!"}
```

Domain领域业务层与ADM模式解说

PhalApi使用的是ADM分层模式，Domain是连接Api层与Model层的桥梁。

何为Api-Domain-Model模式？

在传统Web框架中，惯用MVC模式。可以说，MVC模式是使用最为广泛的模式，但同时也可能是误解最多的模式。然而，接口服务这一领域，与传统的Web应用所面向的领域和需要解决的问题不同，最为明显的是接口服务领域中没有View视图。如果把MVC模式生搬硬套到接口服务领域，不但会产生更多对MVC模式的误解，还不利于实际接口服务项目的开发和交付。

仔细深入地再思考一番，接口服务除了需要处理输入和输出，以及从持久化的存储媒介中提取、保存、删除、更新数据外，还有一个相当重要且不容忽视的任务——处理特定领域的业务规则。而这些规则的处理几乎都是逻辑层面上对数据信息的加工、转换、处理等操作，以满足特定场景的业务需求。对于这些看不见，摸不着，听不到的领域规则处理，却具备着交付有价值的业务功能的使命，与此同时也是最容易出现问题，产生线上故障，引发损失的危险区。所以，在接口服务过程中，我们应该把这些领域业务规则的处理，把这些受市场变化而频繁变动的热区，单独封装成一层，并配套完备的自动化测试体系，保证核心业务的稳定性。

基于以上考虑，在MVC模式的基础上，我们去掉了View视图层，添加了Domain领域业务层。从而涌现了Api-Domain-Model模式，简称ADM模式。

简单来说，

- **Api层** 称为接口服务层，负责对客户端的请求进行响应，处理接收客户端传递的参数，进行高层决策并对领域业务层进行调度，最后将处理结果返回给客户端。
- **Domain层** 称为领域业务层，负责对领域业务的规则处理，重点关注对数据的逻辑处理、转换和加工，封装并体现特定领域业务的规则。
- **Model层** 称为数据模型层，负责技术层面上对数据信息的提取、存储、更新和删除等操作，数据可来自内存，也可以来自持久化存储媒介，甚至可以是来自外部第三方系统。

专注领域的Domain业务层

Domain领域业务层，主要关注的是领域业务规则的处理。在这一层，不应过多关注外界客户端接口调用的签名验证、参数获取、安全性等问题，也不应过多考虑数据从何而来、存放于何处，而是着重关注对领域业务数据的处理。

ADM职责划分与调用关系

传统的接口开发，由于没有很好的分层结构，而且热衷于在一个文件里面完成绝大部分事情，最终导致了臃肿漫长的代码，也就是通常所说的意大利面条式的代码。

在PhalApi中，我们针对接口领域开发，提供了新的分层思想：Api-Domain-Model模式。即便这样，如果项目在实际开发中，仍然使用原来的做法，纵使再好的接口开发框架，也还是会退化到原来的局面。

为了能让大家更为明确Api接口层的职责所在，我们建议：

Api接口服务层应该做：

- 应该：对用户登录态进行必要的检测
- 应该：控制业务场景的主流程，创建领域业务实例，并进行调用
- 应该：进行必要的日记纪录
- 应该：返回接口结果
- 应该：调度领域业务层

Api接口服务层不应该做：

- 不应该：进行业务规则的处理或者计算
- 不应该：关心数据是否使用缓存，或进行缓存相关的直接操作
- 不应该：直接操作数据库
- 不应该：将多个接口合并在一起

Domain领域业务层应该做：

- 应该：体现特定领域的业务规则
- 应该：对数据进行逻辑上的处理
- 应该：调度数据模型层或其他领域业务层

Domain领域业务层不应该做：

- 不应该：直接实现数据的操作，如添加并实现缓存机制

Model数据模型层应该：

- 应该：进行数据库的操作
- 应该：实现缓存机制

在明确了上面应该做的和不应该做的，并且也完成了接口的定义，还有验收测序驱动开发的场景准备后，相信这时，即使是新手也可以编写出高质量的接口代码。因为他会受到约束，他知道他需要做什么，主要他按照限定的开发流程和约定稍加努力即可。

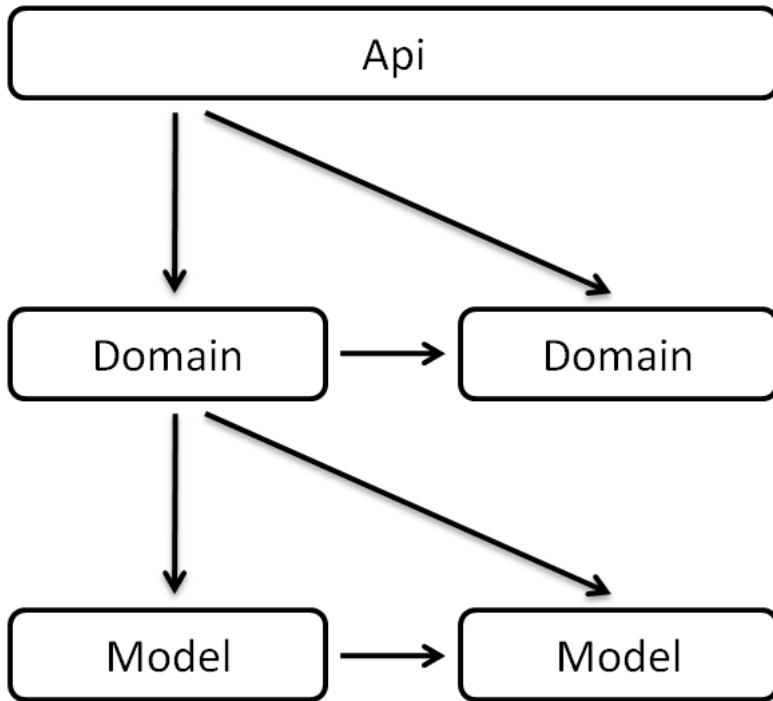
如果真的这样，相信我们也就慢慢能体会到精益开发的乐趣。

至于调用关系，整体上讲，应根据从Api接口层、Domain领域层再到Model数据源层的顺序进行开发。

在开发过程中，需要注意不能**越层调用**也不能**逆向调用**，即不能Api调用Model。而应该是**上层调用下层，或者同层级调用**，也就是说，我们应该：

- Api层调用Domain层
- Domain层调用Domain层
- Domain层调用Model层
- Model层调用Model层

如果用一张图来表示，则是：



为了更明确调用的关系，以下调用是错误的：

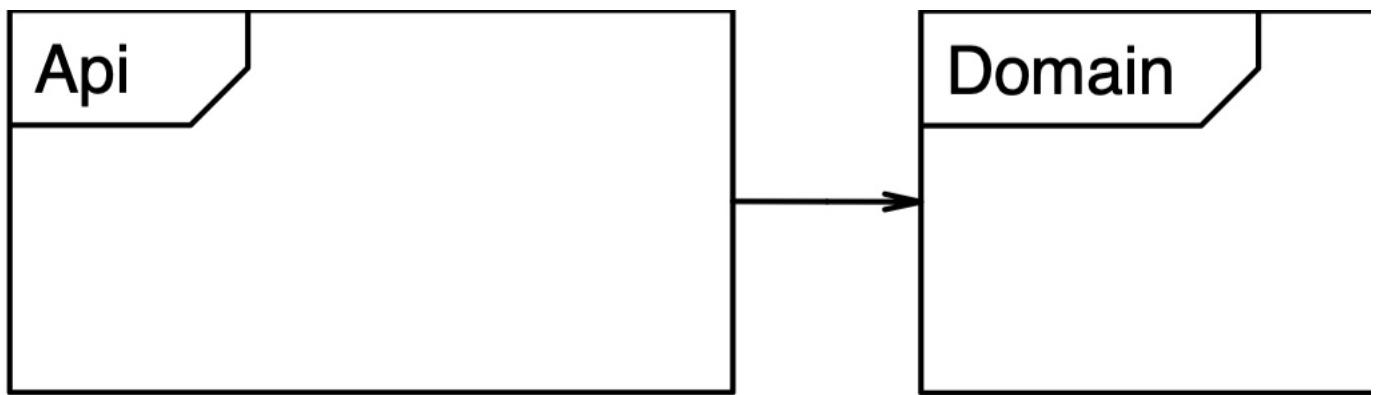
- 错误的做法1：Api层直接调用Model层
- 错误的做法2：Domain层调用Api层，也不应用将Api层对象传递给Domain层
- 错误的做法3：Model层调用Domain层

这样的约定，便于我们形成统一的开发规范，降低学习维护成本。比如需要添加缓存，我们知道应该定位到Model层数据源进行扩展；若发现业务规则处理不当，则应该进入Domain层探其究竟；如果需要对接口的参数进行调整，即使是新手也知道应该找到对应的Api文件进行改动。# Model数据模型层与数据库操作

Model层称为数据模型层，负责技术层面上对数据信息的提取、存储、更新和删除等操作，数据可来自内存，也可以来自持久化存储媒介，甚至可以是来自外部第三方系统。

可以说，PhalApi的Model层是广义上的数据层，而非狭义的数据层。但考虑到大部分数据都是来自于数据库的操作，所以后面会重点讲解如何进行数据库操作。

先一个抽象概括的图来了解Model层所处的位置和重要性。



在Model包的左侧，是它的上游，也就是它的调用方或者客户端。从Api层开始，再调用到Domain领域层，再调用Model层。

而在右侧，另一方面，Model的实现依赖于其需要处理的数据来源。当数据是传统的数据库时，则可以使用NotORM（后面会详细和重点介绍）；当数据是存在高效缓存时，如Redis、Memcached时可以使用PhalApi\Cache接口的具体实现类；如果数据是来自第三方远程系统，则可以通过CURL的方式进行通信。

一个简单的Model例子

简单地，我们可以通过一个简单的例子来入门。

假设，我们需要查找user表中，id = 1的用户信息。可以这样编写Model类。

新增App\Model\User.php文件，编写：

```
<?php  
namespace App\Model;  
  
use PhalApi\Model\NotORMModel as NotORM;  
  
class User extends NotORM {  
  
    public function getUserInfo($id) {
```

```
        return $this->getORM()->where('id', 1)->fetchOne();
    }
}
```

然后，就可以在Domain层使用了。

```
namespace App\Domain;

use App\Model\User as UserModel;

class User {
    public function getUserInfo() {
        $userId = 1;
        $model = new UserModel();
        return $model->getUserInfo($userId);
    }
}
```

最后，就可以在Api层调用封装好的Domain层。

传统的Model数据库层

基于接口的处理重点在于数据，而数据的来源就目前而言，又主要来自数据库，而数据库又集中以MySQL开源数据库为主。因此，我们将重点讲解在PhalApi中如何在Model层使用MySQL数据库。

涉及的内容和知识点，整理成数据库大章节。主要分为：

- [数据库连接](#)
- [数据库与NotORM](#)
- [数据库使用和查询](#)
- [数据库分库分表策略](#)
- [连接多个数据库](#)
- [定制你的Model基类](#)

详细可分别查看上面的文档。# PhalApi 2.x 单元测试

测试驱动开发与PHPUnit

PhalApi推荐使用测试驱动开发最佳实践，并主要使用的是PHPUnit进行单元测试。

PHPUnit官网：<https://phpunit.de>，如需进行单元测试，请先安装PHPUnit。

以下是在PhalApi下简化后TDD步骤。

定义接口服务的函数签名

当新增一个接口服务时，可先定义好接口服务的函数签名，通俗来说，即确定类名和方法名，以及输入、输出参数、接口服务的名称与描述等。

例如，对于获取评论的接口服务，可以这样定义。

```
<?php
namespace App\Api;

use PhalApi\Api;

/**
 * 评论服务
 */
class Comment extends Api {

    public function getRules() {
        return array(
            'get' => array(
                'id' => array('name' => 'id', 'type' => 'int', 'require' => true, 'min' => 1, 'desc' => '评论ID'),
            ),
        );
    }

    /**
     * 获取评论
     * @desc 根据评论ID获取对应的评论信息
     * @return int id 评论ID, 不存在时不返回
     * @return string content 评论内容, 不存在时不返回
     */
    public function get() {
    }
}
```

通过在线接口详情文档，可以看到对应生成的接口文档内容。

接口 : App.Comment.Get

• 获取评论

接口说明

根据评论ID获取对应的评论信息

接口参数

参数名字	类型	是否必须	默认值	其他	说明
id	整型	必须		最小：1	评论ID

返回结果

返回字段	类型	说明
id	整型	评论ID，不存在时不返回
content	字符串	评论内容，不存在时不返回

请求模拟

参数	是否必填	值
service	必须	App.Comment.Get
id	必须	评论ID

这样就完成了我们伟大的第一步，是不是很简单，很有趣？

phalapi-buildtest自动生成测试代码

接下来是为新增的接口类编写对应的单元测试。单元测试的代码，可以手动编写，也可以使用phalapi-buildtest脚本命令自动生成。

生成的命令是：

```
phalapi$ ./bin/phalapi-buildtest ./src/app/Api/Comment.php App\\Api\\Comment > ./tests/app/Api/Comment_Test.php
```

保存的测试文件，统一放在tests目录下，保持与产品代码结构平行，并以“_Test.php”为后缀。

查看生成的单元测试代码文件./tests/app/Api/Comment_Test.php，可以看到类似以下代码：

```
class PhpUnderControl_AppApiComment_Test extends \PHPUnit_Framework_TestCase
{
    public $appApiComment;

    protected function setUp()
    {
        parent::setUp();
        $this->appApiComment = new App\Api\Comment();
    }

    protected function tearDown()
    {
        // 输出本次单元测试所执行的SQL语句
        // var_dump(DI()->tracer->getSqls());
        // 输出本次单元测试所涉及的追踪埋点
        // var_dump(DI()->tracer->getSqls());
    }

    /**
     * @group testGet
     */
    public function testGet()
    {
        $rs = $this->appApiComment->get();
        $this->assertTrue(is_int($rs));
    }
}
```

```
}
```

生成的骨架只是初步的代码，还需要手动调整一下才能最终正常运行。例如需要调整bootstrap.php的文件引入路径。

```
require_once dirname(__FILE__) . '/../../../bootstrap.php';
```

完善单元测试用例

最为重要的是，应该根据**构造-操作-检验（BUILD-OPERATE-CHECK）模式**编写测试用例。对于Api接口层，还需要依赖`Mockery`进行模拟请求。例如这里的：

```
class PhpUnderControl_AppApiComment_Test extends \PHPUnit_Framework_TestCase
{
    public function testGet()
    {
        // Step 1. 构造
        $url = 's=Comment.Get';
        $params = array('id' => 1);

        // Step 2. 操作
        $rs = PhalApi\Helper\TestRunner::go($url, $params);

        // Step 3. 检验
        $this->assertEquals(1, $rs['id']);
        $this->assertArrayHasKey('content', $rs);
    }
}
```

执行单元测试

使用`phpunit`，可以执行刚生成的测试文件。执行：

```
phalapi$ phpunit ./tests/app/Api/Comment_Test.php
```

会看到类似这样的输出：

```
PHPUnit 4.3.4 by Sebastian Bergmann.

.F

Time: 39 ms, Memory: 8.00Mb

There was 1 failure:

1) PhpUnderControl_AppApiComment_Test::testGet
Failed asserting that false is true.

/path/to/phalapi/tests/app/Api/Comment_Test.php:53

FAILURES!
Tests: 2, Assertions: 1, Failures: 1.
```

实现接口服务

在单元测试驱动的引导下，完成接口服务的具体功能，例如这里简单地返回：

```
<?php
namespace App\Api;

use PhalApi\Api;

class Comment extends Api {

    public function get() {
        return array('id' => 1, 'content' => '这是一条模拟的评论');
    }
}
```

再次执行单元测试，便可通过了。

温馨提示：以上示例代码可从[这里查看](#)。# 自动加载和PSR-4

PhalApi 2.x 的自动加载很简单，完全遵循于[PSR-4规范](#)，并且兼容 PhalApi 1.x 版本的加载方式。

在PhalApi 2.x这里，我们主要介绍PSR-4的使用，如果你已经熟悉此约定成俗的命名规范，可跳过这一节。

PSR-4规范一瞥

简单来说，类的全称格式如下：

```
\<NamespaceName>(\<SubNamespaceNames>)*\<ClassName>
```

其中，`<NamespaceName>`为顶级命名空间；`<SubNamespaceNames>`为子命名空间，可以有多层；`<ClassName>`为类名。

PhalApi 2.x 的命名规范

PhalApi 2.x 的项目的顶级命名空间，默认是 app。

Api 层命名规范

默认情况下，假设有一个为 `?s=User.Login` 的用户登录接口服务，则它对应的 Api 层接口类文件为：

/path/to/phalapi/src/app/Api/User.php

类名则为 `app\Api\User`，即顶级命名为 `app`，子命名空间为 `Api`，类名为 `User`。由于存在命名空间，所以其代码实现片段如下：

```
<?php
namespace App\Api;

use PhalApi\Api;

class User extends Api {

    public function Login() {
        // TODO
    }
}
```

多层子命名空间

当存在多层次命名空间时，则需要多层次目录；反之亦然，即如果存在多个子目录，则需要多层次命名空间。例如，对于接口服务 `?s=Weixin_User.Login`，其文件路径为：

/path/to/phalapi/src/app/Api/Weixin/User.php

实现代码片段为：

```
<?php
namespace App\Api\Weixin;

use PhalApi\Api;

class User extends Api {

    public function Login() {
        // TODO
    }
}
```

需要注意的是，此时当前的命名空间为 `App\Api\Weixin`，而不再是 `App\Api`。

Domain 层和 Model 层的命名规范

Domain 层，和 Model 层的命名规范，和 Api 层的一样，或者说其他层级或者目录的规范也是如此，可依次类推。

例如，对于类 `app\Domain\User`，其文件路径为：

/path/to/phalapi/src/app/Domain/User.php

实现代码片段为：

```
<?php
namespace App\Domain;

class User { }
```

而对于类 `app\Domain\Weixin\User`，其文件路径为：

/path/to/phalapi/src/app/Domain/Weixin/User.php

实现代码片段为：

```
<?php
namespace App\Domain\Weixin;

class User { }
```

如何实例化

实例化的方式有两种，对应命名空间两种不同的使用方式。

先 use，再实例

通常情况下，都是先 use，然后再实例化。例如，在 Api 层需要用到 Domain 层的类时，可以这样：

```
<?php
namespace App\Api;

use PhalApi\Api;
use App\Domain\User as DomainUser; // 在这里先 use

class User extends Api {

    public function Login() {
        $domainUser = new DomainUser();
    }
}
```

```
}
```

因为存在两个User类，所以在use领域类时需要改用别名`DomainUser`。如果当前命名空间和待使用的类是同一命名空间，则可以省略use。例如在`App\Domain\User`类中使用`App\Domain\Friends`。

```
<?php
namespace App\Domain;

class User {

    public function Login() {
        $friend = new Friend(); // 可直接使用Friend类
    }
}
```

使用完整类名实例化

另一种情况下，可不用先use，直接使用带命名空间前缀的完整类名来实例化。例如，上面的可改成：

```
<?php
namespace App\Api;

use PhalApi\Api;

class User extends Api {

    public function Login() {
        $domainUser = new \App\Domain\User();
    }
}
```

值得注意的是，在当前命名空间下，如果需要引用其他类，应该在最前面加上反斜杠，表示绝对路径，否则会导致类名错误，从而加载失败。即：

当前命名空间	类名	最终解析类名	区别
App\Api	\App\Domain\User	\App\Domain\User	最前面有反斜杠，正确
App\Api	App\Domain\User	App\Api\App\Domain\User	最前面缺少反斜杠，错误
App\Api	\Exception	Exception	最前面有反斜杠，使用PHP官方的异常类，正确
App\Api	Exception	App\Api\Exception	最前面缺少反斜杠，使用当前命名空间的异常类

如果当前没有命名空间，则最前面可不用加上反斜杠。

如何增加一个顶级命名空间？

在composer下，增加一个顶级命名空间很简单。首先，需要在根目录下的`composer.json`文件中追加`psr-4`配置，如在原有基础上添加一个`Foo`命名空间，则：

```
{
    "autoload": {
        "psr-4": {
            "App\\": "src/app",
            "Foo\\": "src/foo"
        }
    }
}
```

配置好后，执行`composer update`操作：

```
$ composer update
```

此时，对于顶级命名空间`Foo`，其源代码保存在`/path/to/phalapi/src/foo`下。其他类似，这里不再赘述。

需要注意的是，源代码目录需要自己手动添加，即分别添加以下几个常见目录：`Api`、`Domain`、`Model`、`Common`。以这里的`Foo`命名空间为例，需要去创建以下目录：

- `src/foo/Api`
- `src/foo/Domain`
- `src/foo/Model`
- `src/foo/Common`

接下来就可以正常开始开发了。在`src/foo/Api`目录下新增的接口服务，会同步实时显示在线接口文档上。如这里添加`src/foo/Api>Hello.php`文件，并放置以下代码：

```
// 文件 ./src/foo/Api>Hello.php
<?php
namespace Foo\Api;

use PhalApi\Api;

/**
 * Foo下的示例
 */
class Hello extends Api {

    public function world() {
        return array('title' => 'Hello World in Foo!');
    }
}
```

就可以看到：

The screenshot shows the PhalApi documentation interface. On the left, there's a sidebar with a dropdown menu. The first item in the dropdown is 'Foo\ (1)', which is highlighted with a red box. Underneath it, the option 'Foo下的示例' (Examples under Foo) is visible. Other items in the sidebar include 'App\ (9)', '数据库CURD基本操作示例', '生成二维码', '文件上传示例', and '默认接口服务类'. On the right, there's a table titled 'Interface Service' with one row: '# 1 接口服务 Foo.Hello.World'.

注意事项

对于初次使用composer和初次接触PSR-4的同学，以下事项需要特别注意，否则容易导致误解、误用、误导。

- 1、在当前命名空间使用其他命名空间的类时，应先use再使用，或者使用完整的、最前面带反斜杠的类名。
- 2、在定义类时，当前命名空间应置于第一行，且当存在多级命名空间时，应填写完整。
- 3、命名空间和类，应该与文件路径保持一致，并区别大小写。

例如：

```
<?php
namespace App\Api;
use PhalApi\Api;

class Site extends Api {

    public function test() {
        // 错误！会提示 App\Api\DI() 函数不存在！
        DI()->logger->debug('测试函数调用');

        // 正确！调用PhalApi官方函数要用绝对命名空间路径
        \PhalApi\DI()->logger->debug('测试函数调用');
    }

    public function testMyFun() {
        // 错误！会提示 App\Api\my_fun() 函数不存在！
        // (假设在./src/app/functions.php有此函数)
        my_fun();

        // 正确！调用前要加上用绝对命名空间路径
        \App\my_fun();
    }
}
```

接口文档

在线接口文档

PhalApi提供一些非常实用而又贴心的功能特性，其中最具特色的就是自动生成的在线可视化文档。在线接口文档主要分为两大类，分别是：

- 在线接口列表文档
- 在线接口详情文档

当客户端不知道有哪些接口服务，或者需要查看某个接口服务的说明时，可借助此在线接口文档。访问在线接口列表文档的URL是：

<http://dev.phalapi.net/docs.php>

打开后，便可看到类似下面这样的在线接口文档。

接口服务列表 (6)		#	接口服务	接口名称	更多说明
默认接口服务类		1	User.GetBaseInfo	获取用户基本信息	用于获取单个用户基本信息
用户信息类		2	User.GetMultiBaseInfo	批量获取用户基本信息	用于获取多个用户基本信息
微信小程序登录					
Hello World					

温馨提示：此接口服务列表根据后台代码自动生成，可在接口类的文件注释的第一行修改左侧菜单标题。

© Powered By PhalApi 1.4.1

此在线文档是实时生成的，可根据接口源代码以及注释自动生成。当有新增接口服务时，刷新后便可立即看到效果。通过在接口列表文档，可点击进入相应的接口详情文档页面。

温馨提示：如果打开在线文档，未显示任何接口服务，请确保服务环境是否已关闭PHP的opcache缓存。

注释与接口文档

PhalApi提供了自动生成的在线接口文档，对于每一个接口服务，都有对应的在线接口详情文档。如默认接口服务Site.Index的在线接口详情文档为：

接口 : App.Site.Index

• 默认接口服务

接口说明

默认接口服务，当未指定接口服务时执行此接口服务

接口参数

参数名字	类型	是否必须	默认值	其他	说明
username	字符串	可选	PHPer		

返回结果

返回字段	类型	说明
title	字符串	标题
content	字符串	内容
version	字符串	版本，格式：X.X.X
time	整型	当前时间戳

此在线接口详情文档，从上到下，依次说明如下。

接口服务名称

接口服务名称是指用于请求时的名称，对应s参数（或service参数）。接口服务的中文名称，为不带任何注解的注释，通常为接口类成员函数的第一行注释。

```
class Site extends Api {
```

```
/**  
 * 默认接口服务  
 */  
public function index()  
{}
```

接口说明

接口说明对应接口类成员函数的@desc注释。

```
class Site extends Api {  
  
    /**  
     * 默认接口服务  
     * @desc 默认接口服务，当未指定接口服务时执行此接口服务  
     */  
    public function index()  
    {  
}
```

接口参数

接口参数是根据接口类配置的参数规则自动生成，即对应该当前接口类getRules()方法中的返回。其中最后的“说明”字段对应参数规则中的desc选项。可以配置多个参数规则。此外，配置文件./config/app.php中的公共参数规则也会显示在此接口参数里。

```
class Site extends Api {  
  
    public function getRules()  
    {  
        return array(  
            'index' => array(  
                'username' => array('name' => 'username', 'default' => 'PHPer', ),  
            ),  
        );  
    }  
}
```

返回结果

返回结果对应接口类成员函数的@return注释，可以有多组，格式为：@return 返回类型 返回字段 说明。

```
class Site extends Api {  
  
    /**  
     * 默认接口服务  
     * @desc 默认接口服务，当未指定接口服务时执行此接口服务  
     * @return string title 标题  
     * @return string content 内容  
     * @return string version 版本，格式：X.X.X  
     * @return int time 当前时间戳  
     */  
    public function index()  
    {  
}
```

异常情况

异常情况对应@exception注释，可以有多组，格式为：@exception 错误码 错误描述信息。例如：

```
/**  
 * @exception 406 签名失败  
 */  
public function index()
```

刷新后，可以看到新增的异常情况说明。

公共注释

对于当前类的全部函数成员的公共@exception异常情况注释和@return返回结果注释，可在类注释中统一放置。而对于多个类公共的@exception和@return注释，则可以在父类的类注释中统一放置。

也就是说，通过把@exception注解和@return注解移到类注释，可以添加全部函数成员都适用的注解。例如，Api\User类的全部接口都返回code字段，且都返回400和500异常，则可以：

```
<?php  
namespace App\Api;  
  
use PhalApi\Api;  
  
/**  
 * @return int code 操作码，0表示成功  
 * @exception 400 参数传递错误  
 * @exception 500 服务器内部错误  
 */  
class User extends Api {
```

这样就不需要在每个函数成员的注释中重复添加注解。此外，也可以在父类的注释中进行添加。对于相同异常码的@exception注解，子类的注释会覆盖父类的注释，方法的注释会覆盖类的注释；而对于相同的返回结果@return注释，也一样。

需要注意的是，注释必须是紧挨在类的前面，而不能是在namespace前面，否则会导致注释解析失败。

通过在线接口文档进行测试

在线接口文档，不仅可以用来查看接口文档，包括接口参数、返回字段和功能说明外，还可以在上面进行接口测试。这将会直接请求当前的接口。效果如下：

The screenshot shows the PhalApi online documentation interface. At the top, there is a navigation bar with links for 'PhalApi开源接口框架', 'PhalApi', '文档', and '社区'. On the right side of the navigation bar is a search bar labeled '搜索接口'. Below the navigation bar, there is a section titled '请求模拟' (Request Simulation) with a table for entering parameters. The table has three columns: '参数' (Parameter), '是否必填' (Required), and '值' (Value). Two rows are filled in:

参数	是否必填	值
service	必须	App.Site.Index
username	可选	PhalApi

Below the table is a text input field containing the URL 'http://demo.phalapi.net/'. To the right of the URL is a green button labeled '请求当前接口' (Request Current Interface). A red arrow points from the '请求当前接口' button towards the response area. The response area contains the following text:

```
200 OK
connection: keep-alive
content-type: application/json; charset=utf-8
date: Sat, 20 Apr 2019 07:34:43 GMT
server: nginx/1.10.2
transfer-encoding: chunked
x-powered-by: PHP/7.1.15

{
    "ret": 200,
    "data": {
        "title": "Hello PhalApi",
        "version": "2.5.0",
        "time": 1555745683
    },
    "msg": ""
}
```

如何生成离线接口文档？

上面在线的接口文档，也可以一键生成离线版的HTML文档，方便传阅，离线查看。

当需要生成离线文档时，可以在终端，执行以下命令：

```
phalapi$ php ./public/docs.php
Usage:
生成展开版:  php ./public/docs.php expand
生成折叠版:  php ./public/docs.php fold
脚本执行完毕! 离线文档保存路径为: /path/to/phalapi/public/docs
```

执行后，可以看到类似上面的提示和结果输出。再查看生成的离线文档，可以看到类似有：

```
phalapi$ tree ./public/docs
./public/docs
├── App.Examples_CURD.Delete.html
├── App.Examples_CURD.Get.html
├── App.Examples_CURD.GetList.html
├── App.Examples_CURD.Insert.html
├── App.Examples_CURD.Update.html
├── App.Examples_Upload.Go.html
└── App.Site.Index.html
index.html
```

最后，可以在页面访问此离线版文档，如访问链接：

```
http://dev.phalapi.net/docs/index.html
```

也可以将此docs目录打包，在本地打开访问查看。

数据库连接

截至PhalApi 2.6.0 版本，内置支持的数据库连接有：

- MySQL (PDO)
- MS SQL Server (PDO)
- PostgreSQL (PDO)

数据库基本配置

数据库的配置文件为./config/dbs.php， 默认使用的是MySQL数据库。主要有两部分配置：servers和tables。分别是：

- servers， 针对数据库的配置，可以配置多个数据库
- tables， 针对表的配置，支持配置分表（不需要分表可不配置分表）

servers数据库配置

servers选项用于配置数据库服务器相关信息，可以配置多组数据库实例，每组包括数据库的账号、密码、数据库名字等信息。不同的数据库实例，使用不同标识作为下标。格式如下：

servers数据库配置项	配置说明	是否必须	默认值	示例
type	数据库类型	否	mysql	可以是mysql, sqlsrv或pgsql
host	数据库域名	否	localhost	127.0.0.1、或数据库域名
name	数据库名字	是	test	
user	数据库用户名	是	root	
password	数据库密码	是	123456	
port	数据库端口	否	3306或1433 3306	
charset	数据库字符集	否	UTF8	UTF8、utf8mb4

例如下面这份默认配置。

```
return array(
    /**
     * DB数据库服务器集群
     */
    'servers' => array(
        'db_master' => array(
            'host'      => '127.0.0.1',           //服务器标记
            'name'      => 'phalapi',             //数据库域名
            'user'      => 'root',                //数据库用户名
            'password'  => '',                  //数据库密码
            'port'      => 3306,                 //数据库端口
            'charset'   => 'UTF8',               //数据库字符集
        ),
    ),
);
```

默认的数据库标记是db_master，也就是servers的下标，注意db_master不是数据库名称，也是一个称号，通常是指主数据库，一般不需要修改。

tables表配置

tables选项用于配置数据库表的表前缀、主键字段和路由映射关系，可以配置多个表，下标为不带表前缀的表名，其中__default__下标选项为缺省的数据库路由，即未配置的数据库表将使用这一份默认配置。

简单来说，如果是简单的数据库使用，不需要分表，简单这样配置即可：

```
/**
 * 自定义路由表
 */
'tables' => array(
    //通用路由
    '__default__' => array(
        'prefix' => 'tbl_',
        'key' => 'id',
        'map' => array(
            array('db' => 'db_master'),
        ),
    ),
),
```

其中，tables.__default__是通配表，此下标不能更改或删除。每一组表的配置格式如下：

tables表配置项	配置说明	示例
prefix	表前缀	tbl_，如果没有统一表前缀可以为空
key	表主键	id
map	数据库实例映射关系，可配置多组。array(array('db' => 'db_master'))	

其中，map的每组格式为：array('db' => 服务器标识, 'start' => 开始分表标识, 'end' => 结束分表标识)，start和end要么都不提供，要么都提供。此外，db_master不能更改或者需要在前面的servers已经配置。后面会在分表再详细解释，暂时不用理会。

将servers配置和tables的配置，放在一起就是：

```
return array(
    /**
     * DB数据库服务器集群
     */
```

```

'servers' => array(
    'db_master' => array(
        'host'      => '127.0.0.1',           //服务器标记
        'name'      => 'phalapi',            //数据库域名
        'user'       => 'root',              //数据库用户名
        'password'  => '',                 //数据库密码
        'port'       => 3306,                //数据库端口
        'charset'   => 'UTF8',              //数据库字符集
    ),
),
/***
 * 自定义路由表
 */
'tables' => array(
    //通用路由
    '__default__' => array(
        'prefix' => 'tbl_',
        'key'   => 'id',
        'map'   => array('db' => 'db_master'),
    ),
),
),
);

```

其中，在servers中配置了名称为db_master数据库实例，意为数据库主库，其host为localhost，名称为phalapi，用户名为root等。在tables中，只配置了通用路由，并且表前缀为tbl_，主键均为id，并且全部使用db_master数据库实例。

温馨提示：当tables中配置的db数据库实例不存在servers中时，将会提示数据库配置错误。

如何排查数据库连接错误？

普通情况下，数据库连接失败时会这样提示：

```
{
    "ret": 500,
    "data": [],
    "msg": "服务器运行错误：数据库db_demo连接失败"
}
```

考虑到生产环境不方便暴露服务器的相关信息，故这样简化提示。当在开发过程中，需要定位数据库连接失败的原因时，可使用debug调试模式。开启调试后，当再次失败时，会看到类似这样的提示：

```
{
    "ret": 500,
    "data": [],
    "msg": "服务器运行错误：数据库db_demo连接失败，异常码：1045，错误原因：SQLSTATE[28000] [1045] ... ..."
}
```

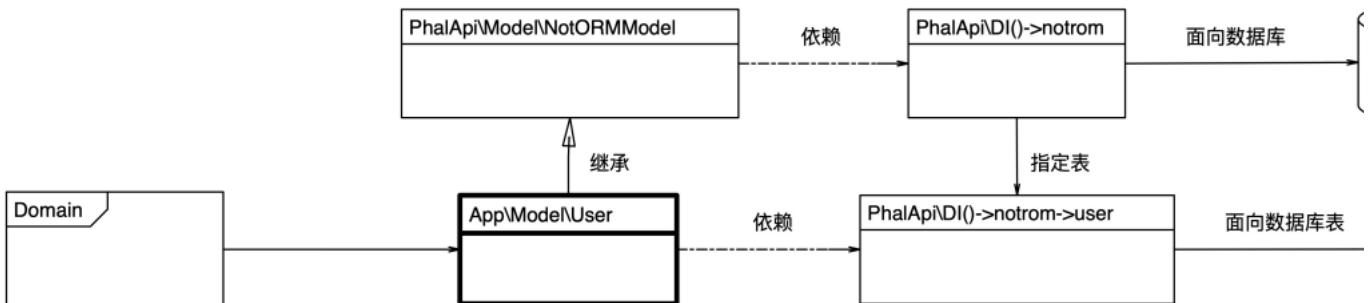
然后，便可根据具体的错误提示进行排查解决。

数据库与NotORM

PhalApi的Model层，如果主要是针对数据库，那么就有必要先来了解NotORM。因为PhalApi框架主要是使用了NotORM来操作数据库。

PhalApi的Model和NotORM整体架构

首先，为避免混淆概念，我们先来看下PhalApi 2.x中的Model和NotORM整体架构。



当我们需要操作数据库时，主要分为三个步骤：连接数据库、实现数据库表操作、调用。

• 第一步、连接数据库

如前面章节介绍，在./config/dbs.php文件中配置好数据库后，在./config/di.php注册PhalApi\DI()->notrom服务，就可以实现数据库连接。

对应上图的右上角部分，这时PhalApi\DI()->notrom是针对数据库的，一个notorm对应一个数据库。反之，如果有多个数据库，则需要注册多个不知名的notorm，后面会再介绍。

• 第二步、实现数据库表操作

原则上，推荐一张表一个Model子类。Model子类需要继承`PhalApi\Model\NotORMModel`。PhalApi框架会根据类名自动映射表名，你也可以通过`PhalApi\Model\NotORMModel::getTableName($id)`手动指定表名。

对应上图的App\Model\User示例，之所以加粗是表示我们这章会重点关注这一Model层的实现。这个示例对应数据库表的user用户表。

• 第三步，使用

遵循实现和使用分离，当我们在Model层封装好数据库表的操作后，就可以提供给客户端使用了。通常Model层的调用方是Domain层，也就是PhalApi框架的ADM分层模式。

下面，将通过user表示例详细介绍。

实现一个Model子类

根据“一张表一个Model类”的原则，我们先来针对user表创建一个Model子类。假设，用户user表结构如下：

```
CREATE TABLE `tbl_user` (
    `id` int(11) NOT NULL,
    `name` varchar(45) DEFAULT NULL,
    `age` int(3) DEFAULT NULL,
    `note` varchar(45) DEFAULT NULL,
    `create_date` datetime DEFAULT NULL,
    PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

当需要新增一个Model子类时，可以继承于`PhalApi\Model\NotORMModel`类，并放置在App\Model命名空间下。例如新增App\Model\User.php文件，并在里面放置以下代码。

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
```

如何指定表名？

上面的App\Model\User类，自动匹配的表名为：user，加上配置前缀“tbl_”，完整的表名是：tbl_user。

默认表名的自动匹配规则是：取“\Model\”后面部分的字符全部转小写，并且在转化后会加上配置的表前缀。

又如：

```
<?php
namespace App\Model\User;
use PhalApi\Model\NotORMModel as NotORM;

class Friends extends NotORM {
```

则类App\Model\User\Friends自动匹配的表名为user_friends。以下是2.x版本的一些示例：

2.x 的Model类名	对应的文件	自动匹配的表名	自动添加表前缀的完整表名
App\Model\User	./src/app/Model/User.php	user	tbl_user
App\Model\User\Friends	./src/app/Model/User/Friends.php	user_friends	tbl_user_friends
App\User\Model\Friends	./src/app/user/Model/Friends.php	friends	tbl_friends
App\User\Model\User\Friends	./src/app/user/Model/User/Friends.php	user_friends	tbl_user_friends

但在以下场景或者其他需要手动指定表名的情况下，可以重写`PhalApi\Model\NotORMModel::getTableName($id)`方法并手动指定表名。

- 存在分表
- Model类名不含有“Model_”
- 自动匹配的表名与实际表名不符
- 数据库表使用蛇形命名法而类名使用大写字母分割的方式

如，当Model_User类对应的表名为：my_user表时，可这样重新指定表名：

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    protected function getTableName($id) {
        return 'my_user'; // 手动设置表名为 my_user
    }
}
```

其中，\$id参数用于进行分表的参考主键，只有当存在分表时才需要用到。通常传入的\$id是整数，然后对分表的总数进行求余从而得出分表标识。

即存在分表时，需要返回的格式为：表名称 + 下划线 + 分表标识。分表标识通常从0开始，为连续的自然数。

简单：4个CURD基本操作

对于基本的Model子类，可以得到基本的数据库操作。以下示例演示了Model的基本CURD操作。

```
$model = new App\Model\User();

// 查询
$row = $model->get(1);
$row = $model->get(1, 'id, name'); //取指定的字段
$row = $model->get(1, array('id', 'name')); //可以数组取指定要获取的字段

// 更新
$data = array('name' => 'test', 'update_time' => time());
$model->update(1, $data); //基于主键的快速更新

// 插入
$data = array('name' => 'phalapi');
$id = $model->insert($data);
// $id = $model->insert($data, 5); //如果是分表，可以通过第二个参数指定分表的参考ID

// 删除
$model->delete(1);
```

特别提醒！需要特别注意的是，继承于PhalApi\Model\NotORMModel的Model子类，只拥有基本的四个数据库操作：
get/update/insert/delete。更多其他数据库操作须经过NotORM实例来进行。

显然，只有CURD这四个基本操作是满足不了项目对于数据库操作的需求，下面继续介绍更全面的数据库操作方式。但在继续深入前，我们需要先来了解如何获取NotORM实例。

如何获取NotORM实例？

NotORM是一个优秀的开源PHP类库，可用于操作数据库。PhalApi的数据库操作，主要是依赖此NotORM来完成，但PhalApi 2.x已经基于最初的NotORM升级成了[phalapi/notorm](#)。

参考：NotORM官网：www.notorm.com。

重要事情讲三遍，默认情况下，

- 在PhalApi中，全部数据库操作都要经过NotORM实例来进行
- 在PhalApi中，全部数据库操作都要经过NotORM实例来进行
- 在PhalApi中，全部数据库操作都要经过NotORM实例来进行

这意味着，不能直接通过Model子类来进行（除下面将说到的4个基本操作外，即get/insert/update/delete），这是一种委托组合而非继承关系。以下示例可以加深理解：

```
// 错误！不能直接通过Model实例来操作，并且不能在Model类外面实现
$model = new \App\Model\User();
$users = $model->select('*')->fetchAll();
```

正确，并推荐写法是：

```
// 正确&推荐！通过NotORM实例看你咯，并且在Model内部实现
namespace App\Model;

class User {
    public function fetchAllUsers() {
        return $this->getORM()->select('*')->fetchAll();
    }
}

$model = new \App\Model\User();
$users = $model->fetchAllUsers();
```

Model层只是针对NotORM的一层代理，而非直接继承的关系。这样的好处是能方便我们灵活、快速切换不同的数据库操作类库。

那么，如何获取NotORM实例呢？

在PhalApi中获取NotORM实例，有两种方式：

- 全局获取方式，能在任何地方使用
- 局部获取方式，只能在Model子类中使用（推荐此用法）

全局获取方式

第一种全局获取的方式，可以用于任何地方，使用DI容器中的全局notorm服务：`\PhalApi\DI()->notorm->表名`。

这是因为我们已经在初始化文件中注册了`\PhalApi\DI()->notorm`这一服务。继续在后面追加表名，就可以获取到NotORM实例了。如这里的：`\PhalApi\DI()->notorm->user`。

全局获取的方式，是为了方便编写脚本，并且可以指定任意表名。例如查user表的总数：

```
$num = \PhalApi\DI()->notorm->user->count();
```

局部获取方式

第二种局部获取的方式，在继承PhalApi\Model\NotORMModel的子类中使用：`$this->getORM()`。

这只限于继承PhalApi\Model\NotORMModel的子类中，并且只能获取当前Model类指定表名的NotORM实例。例如前面的App\Model\User类只能获取\PhalApi\DI()->notorm->user。如取总数：

```
class User extends NotORM {  
    public function count() {  
        // 局部获取  
        $orm = $this->getORM()->count();  
    }  
}
```

如果你不想写Model类，可以直接使用第一种全局获取方式。但是，我们PhalApi推荐使用封装的第二种方式，并且下面所介绍的使用都是基于第二种快速方式。

特别注意NotORM的状态！

特别注意！不管是全局获取，还是局部获取，NotORM实例是带状态的，如果需要再次查询、更新或者删除等，需要获取新的实例！

下面演示了一个不清除状态、错误的使用示例。

```
// 获取一个NotORM实例  
$orm = \PhalApi\DI()->notorm->user;  
  
// 先带条件查一次  
// SELECT * FROM tbl_user WHERE id = 1  
$user1 = $orm->where('id', 1)->fetchOne();  
  
// 再查一次  
// 注意！此时where条件会叠加！！  
// SELECT * FROM tbl_user WHERE id = 1 AND id = 2  
$user2 = $orm->where('id', 2)->fetchOne();
```

正确写法是每次获取一个新的notorm实例：

```
$user1 = \PhalApi\DI()->notorm->user->where('id', 1)->fetchOne();  
$user2 = \PhalApi\DI()->notorm->user->where('id', 2)->fetchOne();
```

附录：PhalApi对NotORM的优化

如果了解NotORM的使用，自然而然对PhalApi中的数据库操作也就一目了然了。但为了更符合接口类项目的开发，PhalApi对NotORM的底层进行优化和调整。以下改动点包括但不限于：

- 将原来返回的结果全部从对象类型改成数组类型，便于数据流通
- 添加查询多条纪录的接口：`NotORM_Result::fetchAll()`和`NotORM_Result::fetchRows()`
- 添加支持原生SQL语句查询的接口：`NotORM_Result::queryAll()`和`NotORM_Result::queryRows()`
- `limit`操作的调整，取消原来`OFFSET`关键字的使用
- 当数据库操作失败时，抛出`PDOException`异常
- 将结果集中以主键作为下标改为以顺序索引作为下标
- 禁止全表删除，防止误删
- 调整调试模式
- 更多优化请见版本更新说明和文档介绍.....

这些优化点可以作为课外的兴趣了解。# 数据库使用和查询

这一章，主要讲解PhalApi主流的数据库使用方式。

常用：数据库操作大全

基本上，全部的常用数据库操作，都可以在下面找到对应的使用说明，以及演示示例。

假设对于前面的tbl_user表，有以下数据。

```
INSERT INTO `tbl_user` VALUES ('1', 'dogstar', '18', 'oschina', '2015-12-01 09:42:31');  
INSERT INTO `tbl_user` VALUES ('2', 'Tom', '21', 'USA', '2015-12-08 09:42:38');  
INSERT INTO `tbl_user` VALUES ('3', 'King', '100', 'game', '2015-12-23 09:42:42');
```

下面将结合示例，分别介绍NotORM更为丰富的数据库操作。在开始之前，假定已有：

```
<?php  
namespace App\Model;  
use PhalApi\Model\NotORMModel as NotORM;  
  
class User extends NotORM {  
    public function test() {  
        $user = $this->getORM(); // 在Model子类内，进行数据库操作前，先获取NotORM实例  
        // $user = $this->getORM(1000); // getORM()的第一个参数是指进行分表的依据，没有时可不传  
    }  
}
```

SQL基本语句介绍

- **SELECT字段选择**

选择单个字段：

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // SELECT id FROM `tbl_user`
        return $this->getORM()->select('id')->fetchAll();
    }
}
```

选择多个字段:

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // SELECT id, name, age FROM `tbl_user`
        return $this->getORM()->select('id, name, age')->fetchAll();
    }
}
```

使用字段别名:

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // SELECT id, name, MAX(age) AS max_age FROM `tbl_user`
        return $this->getORM()->select('id, name, MAX(age) AS max_age')->fetchAll();
    }
}
```

选择全部表字段:

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // SELECT * FROM `tbl_user`
        return $this->getORM()->select('*')->fetchAll();

        // 或不指定select字段，默认取全部
        return $this->getORM()->fetchAll();
    }
}
```

选择去重后的字段:

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // SELECT DISTINCT name FROM `tbl_user`
        return $this->getORM()->select('DISTINCT name')->fetchAll();
    }
}
```

• WHERE条件

单个条件:

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // WHERE id = 1
        return $this->getORM()->where('id', 1)->fetchOne();

        // 或 使用占位符传参
        return $this->getORM()->where('id = ?', 1)->fetchOne();

        // 或 数组形式传参
        return $this->getORM()->where(array('id', 1))->fetchOne();
    }
}
```

多个AND条件:

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // WHERE id > 1 AND age > 18
        return $this->getORM()->where('id > ?', 1)->where('age > ?', 18)->fetchAll();

        // 或 用多个 and 连贯操作
        return $this->getORM()->and('id > ?', 1)->and('age > ?', 18)->fetchAll();
    }
}
```

```

    // 或用含占位符的字符串组合多个条件
    return $this->getORM()->where('id > ? AND age > ?', 1, 18)->fetchAll();

    // 或用多个元素的数组传参
    return $this->getORM()->where(array('id > ?' => 1, 'age > ?' => 10))->fetchAll();
}

public function test2() {
    // 如果只是判断相等，可以直接不用比较符号
    // WHERE name = 'dogstar' AND age = 18
    return $this->getORM()->where(array('name' => 'dogstar', 'age' => 18))->fetchAll();
}
}

```

多个OR条件:

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // WHERE name = 'dogstar' OR age = 18
        return $this->getORM()->or('name', 'dogstar')->or('age', 18)->fetchAll();
    }
}

```

嵌套条件:

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // WHERE ((name = ? OR id = ?) AND (note = ?) -- 'dogstar', '1', 'xxx'

        // 实现方式1：使用AND拼接
        return $this->getORM()->where(' (name = ? OR id = ?)', 'dogstar', '1')->and(' note = ?', 'xxx')->fetchAll();

        // 实现方式2：使用WHERE，并顺序传递多个参数
        return $this->getORM()->where(' (name = ? OR id = ?) AND note = ?', 'dogstar', '1', 'xxx')->fetchAll();

        // 实现方式3：使用WHERE，并使用一个索引数组顺序传递参数
        return $this->getORM()->where(' (name = ? OR id = ?) AND note = ?', array('dogstar', '1', 'xxx'))->fetchAll();

        // 实现方式4：使用WHERE，并使用一个关联数组传递参数
        return $this->getORM()->where(' (name = :name OR id = :id) AND note = :note',
            array('name' => 'dogstar', ':id' => '1', ':note' => 'xxx'))->fetchAll();
    }
}

```

IN查询:

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // 简单的IN查询
        // WHERE id IN (1, 2, 3)
        return $this->getORM()->where('id', array(1, 2, 3))->fetchAll();
    }

    public function test2() {
        // 排除IN
        // WHERE id NOT IN (1, 2, 3)
        return $this->getORM()->where('NOT id', array(1, 2, 3))->fetchAll();
    }

    public function test3() {
        // 多个IN查询
        // WHERE (id, age) IN ((1, 18), (2, 20))
        return $this->getORM()->where(' (id, age)', array(array(1, 18), array(2, 20)))->fetchAll();
    }
}

```

模糊匹配查询:

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // 像
        // WHERE name LIKE '%dog%'
        return $this->getORM()->where('name LIKE ?', '%dog%')->fetchAll();
    }

    public function test2() {
        // 不像
        // WHERE name NOT LIKE '%dog%'
        return $this->getORM()->where('name NOT LIKE ?', '%dog%')->fetchAll();
    }
}

```

温馨提示：需要模糊匹配时，不可写成：`where('name LIKE %?%', 'dog')`。

NULL判断查询:

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // WHERE (name IS NULL)
        return $this->getORM()->where('name', null)->fetchAll();
    }
}
```

非NULL判断查询:

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // WHERE (name IS NOT NULL)
        return $this->getORM()->where('name IS NOT ?', null)->fetchAll();
    }
}
```

• ORDER BY排序

单个字段升序排序:

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // ORDER BY age
        return $this->getORM()->order('age')->fetchAll();

        // 或指定排序方式, 默认是升序
        return $this->getORM()->order('age ASC')->fetchAll();
    }
}
```

单个字段降序排序:

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // ORDER BY age DESC
        return $this->getORM()->order('age DESC')->fetchAll();
    }
}
```

多个字段排序:

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // ORDER BY id, age DESC
        return $this->getORM()->order('id')->order('age DESC')->fetchAll();

        // 或 连起来写
        return $this->getORM()->order('id, age DESC')->fetchAll();
    }
}
```

• LIMIT数量限制

限制数量, 如查询前10个:

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // LIMIT 10
        return $this->getORM()->limit(10)->fetchAll();
    }
}
```

分页限制, 如从第5个位置开始, 查询前10个:

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // LIMIT 5, 10
        return $this->getORM()->limit(5, 10)->fetchAll();
    }
}
```

```
}
```

• GROUP BY和HAVING

只有GROUP BY, 没有HAVING:

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // GROUP BY note
        return $this->getORM()->group('note')->fetchAll();
    }
}
```

既有GROUP BY, 又有HAVING:

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // GROUP BY note HAVING age > 10
        return $this->getORM()->group('note', 'age > 10')->fetchAll();
    }
}
```

CURD之插入操作

插入操作可分为插入单条纪录、多条纪录，或根据条件插入。

操作	说明	示例	备注	是否 PhalApi新增
insert()	插入数据	\$user->insert(\$data);	全局方式需要再调用 insert_id()获取插入的ID	否
insert_multi()	批量插入 TRUE时进行INSERT IGNORE INTO操作	\$user->insert_multi(\$rows, \$isIgnore = FALSE); 可批量插入 否，但有优化，\$isIgnore为 TRUE时进行INSERT IGNORE INTO操作		
insert_update()	插入/ 更新	接口签名: insert_update(array \$unique, array \$insert, array \$update = array())	不存时插入，存在时更新	否

插入单条纪录数据，注意，必须是保持状态的同一个NotORM表实例，方能获取到新插入的行ID，且表必须设置了自增主键ID。

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        $data = array('name' => 'PhalApi', 'age' => 1, 'note' => 'framework');

        // INSERT INTO tbl_user (name, age, note) VALUES ('PhalApi', 1, 'framework')
        $orm = $this->getORM();
        $orm->insert($data);

        // 返回新增的ID（注意，这里不能使用连贯操作，因为要保持同一个ORM实例）
        return $orm->insert_id();
    }
}
```

或者使用Model封装的insert()基本方法

```
// App\Model\User类，不需要额外的实现
$model = new App\Model\User();
$id = $model->insert($data);
var_dump($id); // 返回新增的ID
```

批量插入多条纪录数据:

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        $rows = array(
            array('name' => 'A君', 'age' => 12, 'note' => 'AA'),
            array('name' => 'B君', 'age' => 14, 'note' => 'BB'),
            array('name' => 'C君', 'age' => 16, 'note' => 'CC'),
        );

        // INSERT INTO tbl_user (name, age, note) VALUES ('A君', 12, 'AA'), ('B君', 14, 'BB'), ('C君', 16, 'CC')
        // 返回成功插入的条数
        return $this->getORM()->insert_multi($rows);

        // PhalApi 5.2.0 及以上版本才支持
        // 如果希望使用 IGNORE，可加传第二个参数
        // INSERT IGNORE INTO tbl_user (name, age, note) VALUES ('A君', 12, 'AA'), ('B君', 14, 'BB'), ('C君', 16, 'CC')
        return $this->getORM()->insert_multi($rows, true);
    }
}
```

```

    }
}

```

插入/更新（组合操作：有则更新，没有则插入）：

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        $unique = array('id' => 8);
        $insert = array('id' => 8, 'name' => 'PhalApi', 'age' => 1, 'note' => 'framework');
        $update = array('age' => 2);

        // INSERT INTO tb1_user (id, name, age, note) VALUES (8, 'PhalApi', 1, 'framework')
        // ON DUPLICATE KEY UPDATE age = 2
        // 返回影响的行数
        return $this->getORM()->insert_update($unique, $insert, $update);
    }
}

```

CURD之更新操作

操作	说明	示例	备注	是否PhalApi新增
update()	更新数据	\$user->where('id', 1)->update(\$data);	更新异常时返回false，数据无变化时返回0，成功更新返回影响的行数	否
updateCounter()	更新单个计数器	接口签名：updateCounter(\$column, \$number = 1)，示例： \$user->where('id', 1)->updateCounter('age', 1)	返回影响的行数	是，PhalApi 2.6.0 版本及以上支持
updateMultiCounters()	更新多个计数器	接口签名：updateMultiCounters(array \$data)，示例：\$user->where('id', 1)->updateMultiCounters(array('age' => 1))	返回影响的行数	是，PhalApi 2.6.0 版本及以上支持

根据条件更新数据：

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        $data = array('age' => 2);

        // UPDATE tbl_user SET age = 2 WHERE (name = 'PhalApi');

        // 返回更新后的结果（注意区分微妙细节）
        // int(1)，表示 正常影响的行数
        // int(0)，表示 无更新，或者数据没变化
        // boolean(false)，表示 更新异常、失败

        return $this->getORM()->where('name', 'PhalApi')->update($data);
    }
}

```

再重复一下，对于更新后返回的结果。

- int(1)，表示 正常影响的行数
- int(0)，表示 无更新，或者数据没变化
- boolean(false)，表示 更新异常、失败

在使用update()进行更新操作时，如果更新的数据和原来的一样，则会返回0（表示影响0行）。这时，会和更新失败（同样影响0行）混淆。但NotORM是一个优秀的类库，它已经提供了优秀的解决文案。我们在使用update()时，只须了解这两者返回结果的微妙区别即可。因为失败异常时，返回false；而相同数据更新会返回0。即：

- 1、更新相同的数据时，返回0，严格来说是：int(0)
- 2、更新失败时，如更新一个不存在的字段，返回false，即：bool(false)

用代码表示，就是：

```

$model = new \App\Model\User();
$rs = $model->test();

if ($rs >= 1) {
    // 成功
} else if ($rs === 0) {
    // 相同数据，无更新
} else if ($rs === false) {
    // 更新失败
}

```

更新数据，进行加1操作：

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // UPDATE tbl_user SET age = age + 1 WHERE (name = 'PhalApi')
        // 返回影响的行数
        return $this->getORM()->where('name', 'PhalApi')->update(array('age' => new \NotORM_Literal("age + 1")));
    }
}

```

```
}
```

温馨提示：在2.x版本中，当需要使用NotORM_Literal类进行加1操作时，须注意两点：需要先获取NotORM实例再创建NotORM_Literal对象；注意命名空间，即要在最前面加反斜杠。

上面的数据更新操作更灵活，因为可以在更新其它字段的同时进行数值的更新。但考虑到还需要创建NotORM_Literal对象，增加了认知成本。所以对于简单的计数器更新操作，可以使用updateCounter()接口。

注意：updateCounter()接口和updateMultiCounters()接口，需要PhalApi 2.6.0 及以上版本，方可支持。

对比改用updateCounter()接口后的简化版本：

```
class User extends NotORM {  
    public function test() {  
        // UPDATE tbl_user SET age = age + 1 WHERE (name = 'PhalApi')  
        // 返回影响的行数  
        return $this->getORM()->where('name', 'PhalApi')->updateCounter('age');  
    }  
}
```

须留意到，updateCounter()的第一个参数是字段名称，第二个参数是待更新数值，可以是正数或负数，默认是1，表示加1。返回的结果是影响的行数，而非最新的字段值。下以是更多示例：

```
// 加1  
$this->getORM()->where('name', 'PhalApi')->updateCounter('age', 1);  
  
// 减1  
$this->getORM()->where('name', 'PhalApi')->updateCounter('age', -1);
```

与此相似，updateMultiCounters()接口也可用于更新计数器，不同的是此接口可以同时更新多个计数器，且第一个参数是数组。数组下标为字段名，数组元素值为待更新数值。例如：

```
// age加1，同时points加10  
$this->getORM()->where('name', 'PhalApi')->updateMultiCounters(array('age' => 1, 'points' => 10));  
  
// age减1，同时points减10  
$this->getORM()->where('name', 'PhalApi')->updateMultiCounters(array('age' => -1, 'points' => -10));
```

CURD之查询操作

查询操作主要有获取一条纪录、获取多条纪录以及聚合查询等。

操作	说明	示例	备注	是否 PhalApi新 增
fetch()	循环获取每一行	while(\$row = \$user->fetch()) { ... }	否	
fetchOne()	只获取第一行	\$row = \$user->where('id', 1)->fetchOne();	等效于fetchRow()	是
fetchRow()	只获取第一行	\$row = \$user->where('id', 1)->fetchRow();	等效于fetchOne()	是
fetchPairs()	获取键值对	\$row = \$user->fetchPairs('id', 'name');	第二个参数为空时，可取多个值，并且多条纪录；也可以指定单个字段，还可以指定多个字段。	否
fetchAll()	获取全部的行	\$rows = \$user->where('id', array(1, 2, 3))->fetchAll();	等效于fetchRows()	是
fetchRows()	获取全部的行	\$rows = \$user->where('id', array(1, 2, 3))->fetchRows();	等效于fetchAll()	是
queryAll()	复杂查询下获取全部的行，默 认下以主键为下标	\$rows = \$user->queryAll(\$sql, \$parmas);	等效于queryRows()	是
queryRows()	复杂查询下获取全部的行，默 认下以主键为下标	\$rows = \$user->queryRows(\$sql, \$parmas);	等效于queryAll()	是
count()	查询总数	\$total = \$user->count('id');	第一参数可省略	否
min()	取最小值	\$minId = \$user->min('id');	否	
max()	取最大值	\$maxId = \$user->max('id');	否	
sum()	计算总和	\$sum = \$user->sum('age');	否	

循环获取每一行，并且同时获取多个字段：

```
<?php  
namespace App\Model;  
use PhalApi\Model\NotORMModel as NotORM;  
  
class User extends NotORM {  
    public function test() {  
        // SELECT id, name FROM tbl_user WHERE (age > 18);  
        $orm = $this->getORM()->select('id, name')->where('age > 18');  
  
        while ($row = $orm->fetch()) {  
            var_dump($row);  
        }  
    }  
}
```

```
// 输出
array(2) {
    ["id"]=>
        string(1) "2"
    ["name"]=>
        string(3) "Tom"
}
array(2) {
    ["id"]=>
        string(1) "3"
    ["name"]=>
        string(4) "King"
}
...
...
```

循环获取每一行，并且只获取单个字段。需要注意的是，指定获取的字段，必须出现在select里，并且返回的不是数组，而是字符串。

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // SELECT id, name FROM tbl_user WHERE (age > 18);
        $orm = $this->getORM()->select('id, name')->where('age > 18');

        while ($row = $orm->fetch('name')) { // 指定获取的单个字段
            var_dump($row); // 此时，输出的是一个字段值，而非一条数组纪录
        }
    }
}

// 输出
string(3) "Tom"
string(4) "King"
...
```

注意！以下是错误的用法。还记得前面所学的NotORM状态的保持吗？因为这里每次循环都会新建一个NotORM表实例，所以没有保持前面的查询状态，从而死循环。

```
while ($row = $this->getORM()->select('id, name')->where('age > 18')->fetch('name')) {
    var_dump($row);
}
```

只获取第一行，并且获取多个字段，等同于fetchRow()操作。

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // SELECT id, name FROM tbl_user WHERE (age > 18) LIMIT 1;
        return $this->getORM()->select('id, name')->where('age > 18')->fetchOne();
    }
}

// 返回结果示例
array(2) {
    ["id"]=>
        string(1) "2"
    ["name"]=>
        string(3) "Tom"
}
```

只获取第一行，并且只获取单个字段，等同于fetchRow()操作。

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // SELECT id, name FROM tbl_user WHERE (age > 18) LIMIT 1;
        return $this->getORM()->where('age > 18')->fetchOne('name'); // 只获取单个字段
    }
}

// 返回结果示例
string(3) "Tom"
```

获取键值对，并且获取多个字段：

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // SELECT id, name, age FROM tbl_user LIMIT 2;
        return $this->getORM()->select('name, age')->limit(2)->fetchPairs('id'); // 指定以ID为KEY
    }
}

// 返回结果示例
array(2) {
    [1]=> // 下标对应id字段
    array(3) {
        ["id"]=>
            string(1) "1"
        ["name"]=>
```

```

        string(7) "dogstar"
        ["age"]=>
        string(2) "18"
    }
    [2]=>
    array(3) {
        ["id"]=>
        string(1) "2"
        ["name"]=>
        string(3) "Tom"
        ["age"]=>
        string(2) "21"
    }
}

```

获取键值对，并且只获取单个字段。注意，这时的值不是数组，而是字符串。

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // SELECT id, name FROM tbl_user LIMIT 2
        return $this->getORM()->limit(2)->fetchPairs('id', 'name'); //通过第二个参数，指定VALUE的列
    }
}

// 返回结果示例
array(2) {
    [1]=>
    string(7) "dogstar"
    [2]=>
    string(3) "Tom"
}

```

获取全部的行，相当于fetchRows()操作。

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // SELECT * FROM tbl_user
        return $this->getORM()->fetchAll(); // 全部表数据
    }
}

```

高级：使用原生SQL语句进行查询

使用原生SQL语句进行查询，并获取全部的行：

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // SELECT name FROM tbl_user WHERE age > :age LIMIT 1
        $sql = 'SELECT name FROM tbl_user WHERE age > :age LIMIT 1';
        $params = array(':age' => 18);

        return $this->getORM()->queryAll($sql, $params);
    }

    // 除了使用上面的关联数组传递参数，也可以使用索引数组传递参数
    public function test2() {
        // SELECT name FROM tbl_user WHERE age > :age LIMIT 1
        $sql = 'SELECT name FROM tbl_user WHERE age > ? LIMIT 1';
        $params = array(18);

        // 也使用queryRows()别名
        return $this->getORM()->queryRows($sql, $params);
    }

    // 输出
    array(1) {
        [0]=>
        array(1) {
            ["name"]=>
            string(3) "Tom"
        }
    }
}

```

在使用queryAll()或者queryRows()进行原生SQL操作时，需要特别注意：

- 1、需要手动填写完整的表名字，包括分表标识，并且需要通过任意表实例来运行
- 2、尽量使用参数绑定，而不应直接使用参数来拼接SQL语句，慎防SQL注入攻击

下面是不好的写法，很有可能会导致SQL注入攻击

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    // 存在SQL注入的写法
}

```

```

public function test() {
    // 存在SQL注入的写法
    $id = 1;
    $sql = "SELECT * FROM tbl_demo WHERE id = $id";

    // 存在SQL注入的写法
    return $this->getORM()->queryAll($sql);
}
}

```

对于外部不可信的输入数据，应改用参数传递的方式。

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    // 使用参数绑定方式，避免SQL注入
    public function test() {
        $id = 1;
        $sql = "SELECT * FROM tbl_demo WHERE id = ?";

        return $this->getORM()->queryAll($sql, array($id));
    }
}

```

查询总数：

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // SELECT COUNT(id) FROM tbl_user
        return $this->getORM()->count('id');
    }
}

// 输出
string(3) "3"

```

查询最小值：

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // SELECT MIN(age) FROM tbl_user
        return $this->getORM()->min('age');
    }
}

// 输出
string(2) "18"

```

查询最大值：

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // SELECT MAX(age) FROM tbl_user
        return $this->getORM()->max('age');
    }
}

// 输出
string(3) "100"

```

计算总和：

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // SELECT SUM(age) FROM tbl_user
        return $this->getORM()->sum('age');
    }
}

// 输出
string(3) "139"

```

CURD之删除操作

操作	说明	示例	备注	是否PhalApi新增
delete()	删除	\$user->where('id', 1)->delete();	禁止无where条件的删除操作	否

:

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // DELETE FROM tbl_user WHERE (id = 404);
        // 按条件进行删除，并返回影响的行数
        return $this->getORM()->where('id', 404)->delete();
    }
}

```

请特别注意，PhalApi禁止全表删除操作。即如果是全表删除，将会被禁止，并抛出异常。如：

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // DELETE FROM tbl_user WHERE (id = 404);
        // Exception: sorry, you can not delete the whole table
        // 禁止全表删除！！！
        return $this->getORM()->delete();
    }
}

```

高级：执行原生sql操作并返回结果

简单总结一下，对于执行原生sql操作的支持，主要有以下三个接口：

- queryAll/queryRows，主要用于进行SELECT查询，并可以返回查询的数据结果集
- executeSql，主要用于进行带返回结果的UPDATE、INSERT、DELETE以及数据库变更等操作，但只会返回影响的行数
- query，最底层的原生操作，不返回任何结果

接下来，简单通过示例说明executeSql()接口的使用。

请注意，executeSql()接口需要PhalApi 2.6.0 及以上版本，方可支持。

如果是在Model子类内，可以这样实现：

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class NotORMTest extends NotORM {
    public function getTableName($id) {
        return 'notormtest';
    }

    public function executeSqlInsert() {
        // 原生插入
        $sql = "INSERT INTO tbl_notormtest (`content`, `ext_data`) VALUES ('phpunit_e_sql_1', '\" . '(\\"year\\":2019)' . \"');";
        return $this->executeSql($sql);
    }

    public function executeSqlUpdate() {
        // 原生更新
        $sql = "UPDATE tbl_notormtest SET `content` = 'phpunit_e_sql_3' WHERE (content = ? OR content = ?);";
        $params = array('phpunit_e_sql_1', 'phpunit_e_sql_2');
        return $this->getORM()->executeSql($sql, $params);
    }

    public function executeSqlDelete() {
        // 原生删除
        $sql = "DELETE FROM tbl_notormtest WHERE (content IN ('phpunit_e_sql_3'));";
        return $this->getORM()->executeSql($sql);
    }
}

```

另一方面，也可以通过全局NotORM实例来调用，并通过单元测试来验证执行的效果是否符合预期。

```

public function testExcuteSql()
{
    // 原生插入
    $sql = "INSERT INTO tbl_notormtest (`content`, `ext_data`) VALUES ('phpunit_e_sql_1', '\" . '(\\"year\\":2019)' . \"');";
    $rs = \PhalApi\DI()->notorm->notormtest->executeSql($sql);

    $this->assertEquals(1, $rs);

    // 原生绑定参数插入
    $sql = "INSERT INTO tbl_notormtest (`content`, `ext_data`) VALUES (:content, :ext_data);";
    $params = array(':content' => 'phpunit_e_sql_2', ':ext_data' => '(\\"year\\":2020)');
    $rs = \PhalApi\DI()->notorm->notormtest->executeSql($sql, $params);

    $this->assertEquals(1, $rs);

    // 原生更新
    $sql = "UPDATE tbl_notormtest SET `content` = 'phpunit_e_sql_3' WHERE (content = ? OR content = ?);";
    $params = array('phpunit_e_sql_1', 'phpunit_e_sql_2');
    $rs = \PhalApi\DI()->notorm->notormtest->executeSql($sql, $params);

    $this->assertEquals(2, $rs);

    // 如果是查询呢？只会返回影响的行数，而非结果
    $sql = "SELECT * FROM tbl_notormtest WHERE content IN ('phpunit_e_sql_3')";
    $rs = \PhalApi\DI()->notorm->notormtest->executeSql($sql, $params);
}

```

```

    $this->assertEquals(2, $rs);

    // 原生删除
    $sql = "DELETE FROM tbl_notormtest WHERE (content IN ('phpunit_e_sql_3'))";
    $rs = \PhalApi\DI()->notorm->notormtest->executeSql($sql);

    $this->assertEquals(2, $rs);
}

```

复杂：事务操作、关联查询和其他操作

事务操作

以下是事务操作的一个示例。

```

// Step 1: 开启事务
\PhalApi\DI()->notorm->beginTransaction('db_demo');

// Step 2: 数据库操作
\PhalApi\DI()->notorm->user->insert(array('name' => 'test1'));
\PhalApi\DI()->notorm->user->insert(array('name' => 'test2'));

// Step 3: 提交事务/回滚
\PhalApi\DI()->notorm->commit('db_demo');
//\PhalApi\DI()->notorm->rollback('db_demo');

```

关联查询

对于关联查询，简单的关联可使用NotORM封装的方式，而复杂的关联，如多个表的关联查询，则可以使用PhalApi封装的接口。

如果是简单的关联查询，可以使用NotORM支持的写法，这样好处在于我们使用了一致的开发，并且能让PhalApi框架保持分布式的操作方式。需要注意的是，关联的表仍然需要在同一个数据库。

以下是一个简单的示例。假设我们有这样的数据：

```

INSERT INTO `phalapi_user` VALUES ('1', 'wx_edebc', 'dogstar', '***', '4CHqOhe1', '1431790647', '');
INSERT INTO `phalapi_user_session_0` VALUES ('1', '1', 'ABC', '', '0', '0', '0', null);

```

那么对应关联查询的代码如下面：

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class Session extends NotORM { // 注意，是Session类
    public function test() {
        // SELECT expires_time, user.username, user.nickname FROM tbl_session
        // LEFT JOIN tbl_user AS user
        // ON tbl_session.user_id = user.id
        // WHERE (token = 'ABC') LIMIT 1
        return $this->getORM()
            ->select('expires_time', 'user.username', 'user.nickname')
            ->where('token', 'ABC')
            ->fetchRow();
    }
}

```

会得到类似这样的输出：

```

array(3) {
    ["expires_time"]=>
        string(1) "0"
    ["username"]=>
        string(35) "wx_edebc"
    ["nickname"]=>
        string(10) "dogstar"
}

```

这样，我们就可以实现关联查询的操作。按照NotORM官网的说法，则是：

If the dot notation is used for a column anywhere in the query ("\$table.\$column") then NotORM automatically creates left join to the referenced table. Even references across several tables are possible ("\$table1.\$table2.\$column"). Referencing tables can be accessed by colon: \$applications->select("COUNT(application_tag:tag_id)").

所以->select('expires_time, user.username, user.nickname')这一行调用将会NotORM自动产生关联操作，而ON的字段，则是这个字段关联你配置的表结构，外键默认为：表名_id。

如果是复杂的关联查询，则是建议使用原生的SQL语句，但仍然可以保持很好的写法，如这样一个示例：

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class Vote extends NotORM {
    public function test() {
        $sql = 'SELECT t.id, t.team_name, v.vote_num ,
        . 'FROM phalapi_team AS t LEFT JOIN phalapi_vote AS v '
        . 'ON t.id = v.team_id ,
        . 'ORDER BY v.vote_num DESC';
        return $this->getORM()->queryAll($sql, array());
    }
}

```

如前面所述，这里需要手动填写完整的表名，以及慎防SQL注入攻击。

其他数据库操作

有时，我们还需要进行一些其他的数据库操作，如创建表、删除表、添加表字段等。对于需要进行的数据库操作，而上面所介绍的方法未能满足时，可以使用更底层更通用的接口，即：`\NotORM_Result::query($query, $parameters)`。

例如，删除一张表。

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function test() {
        // DROP TABLE tbl_user
        return $this->getORM()->query('DROP TABLE tbl_user', array());
    }
}# 数据库分库分表策略
```

也许，大家会觉得PhalApi对于NotORM的封装略过于复杂，但这样设计的初衷以及好处是能快速实现分库分表策略。这一策略能在海量数据和高并发访问下是非常行之有效的。

所以，这一章，我们将进入学习如何在PhalApi中配置数据库分库分表策略，以及如何自动生成表的SQL变更语句。

我们先来看分表（一个数据库内，分多张表），再来看分库（多个数据库）。

分表策略配置

假设设有以下多个数据库表，它们的表结构一样。

数据库表	数据库实例
tbl_demo | db_master
tbl_demo_0 | db_master
tbl_demo_1 | db_master
tbl_demo_2 | db_master

为了使用分表存储，可以修改数据库表的配置，让它支持分表的情况。

```
php
return array(
    'tables' => array(
        'demo' => array(
            'prefix' => 'tbl_',
            'key' => 'id',
            'map' => array(
                array('db' => 'db_master'),
                array('start' => 0, 'end' => 2, 'db' => 'db_master'),
            ),
        ),
    ),
);
```

上面配置map选项中`array('db' => 'master')`用于指定缺省主表使用master数据库实例，而下一组映射关系则是用于配置连续在同一台数据库实例的分表区间，即tbl_demo_0、tbl_demo_1、tbl_demo_2都使用了master数据库实例。

map配置详解

在配置分表时，map配置是关键的配置。可以配置多组，通常配置的顺序是：

- 默认的非分表配置
- 从0开始的前N个分表配置
- 从N+1到.....的分表配置

例如，默认的非分表配置，主要是配置使用哪个数据库，通过数据库标识（如默认的：db_master）指定。

```
'map' => array(
    array('db' => 'db_master'),
),
```

接下来是分表的配置，分表的下标通常从0开始，这取决于你在Model子类中的分表策略。你也可以从1001开始，可以从任意数字开始。在配置过程中，主要能保证的分表连续性即可。

例如，对于上面的分表配置，我们还可以这样配置，效果是一样的。

一个极端的方式，分别配置分表tbl_demo_0、tbl_demo_1、tbl_demo_2，即各张表配置一个策略：

```
'map' => array(
    array('start' => 0, 'end' => 0, 'db' => 'db_master'),
    array('start' => 1, 'end' => 1, 'db' => 'db_master'),
    array('start' => 2, 'end' => 2, 'db' => 'db_master'),
),
```

此外，也可以配置两组。分别配置分表tbl_demo_0和tbl_demo_1，以及tbl_demo_2（前2后1）：

```
'map' => array(
    array('start' => 0, 'end' => 1, 'db' => 'db_master'),
    array('start' => 2, 'end' => 2, 'db' => 'db_master'),
),
```

当然，也可以前1后2，即第一个库1张分表，第二个库2张分表：

```
'map' => array(
    array('start' => 0, 'end' => 0, 'db' => 'db_master'),
    array('start' => 1, 'end' => 2, 'db' => 'db_master'),
),
```

关键点再重复说明一下，要保证map中分表后缀的连续性。

温馨提示：当分表找不到时，PhalApi会自动退化使用缺省主表，即去掉分表后缀的表名。例如tbl_demo_0找不到则退化为tbl_demo。由此，推论出另一外需要特别注意的点。

推论：如果不需要分表，禁止在表名添加下划线+数字后缀。

例如，不要这么设计表名：

- `tbl_user_1`
- `user_1`
- `user_20190101`

可以改为去掉下划线或者再加个字母作为后缀，例如改为：

- `tbl_user`
- `user_1_bak`
- `user_20190101_tag`

Model子类实现分表逻辑

假设分别的规则是根据ID对3进行求余。当需要使用分表时，在使用Model基类的情况下，可以通过重写`PhalApi\Model\NotORMModel::getTableName($id)`实现相应的分表规则。

```
<?php
namespace App\Model;

use PhalApi\Model\NotORMModel as NotORM;

class Demo extends NotORM {

    protected function getTableName($id) {
        $tableName = 'demo';
        if ($id !== null) {
            $tableName .= '_' . ($id % 3);
        }
        return $tableName;
    }
}
```

然后，便可使用之前一样的CURD基本操作，但框架会自动匹配分表的映射。例如：

```
$model = new App\Model\Demo();

$row = $model->get('3', 'id'); // 使用分表tbl_demo_0
$row = $model->get('10', 'id'); // 使用分表tbl_demo_1
$row = $model->get('2', 'id'); // 使用分表tbl_demo_2
```

回到使用Model基类的上下文，更进一步，我们可以通过`$this->getORM($id)`来获取分表的实例从而进行分表的操作。如：

```
<?php
namespace App\Model;

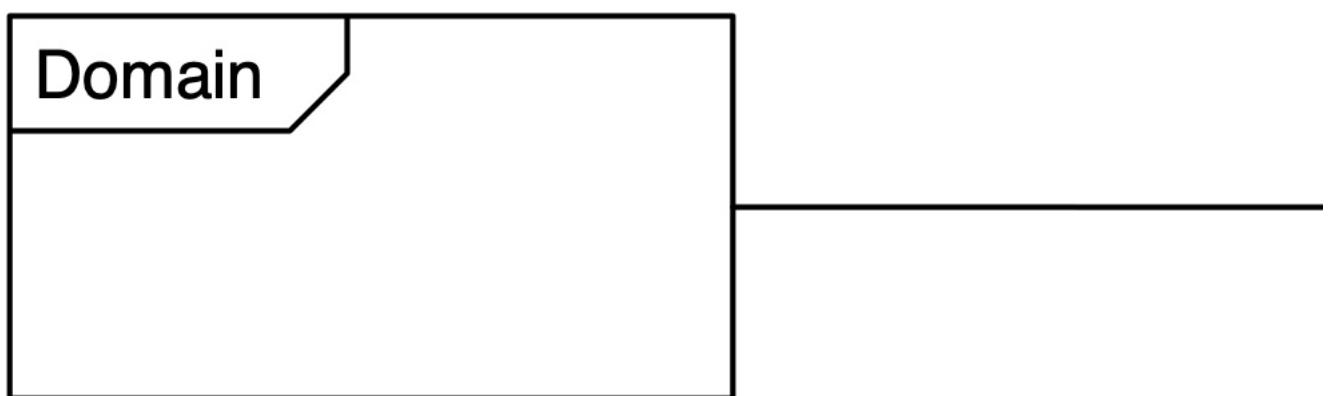
use PhalApi\Model\NotORMModel as NotORM;

class Demo extends NotORM {

    public function getNameById($id) {
        $row = $this->getORM($id)->select('name')->fetchRow();
        return !empty($row) ? $row['name'] : '';
    }
}
```

通过传入不同的\$id，即可获取相应的分表实例。

至此，整体的架构总结如下：



回顾前面学的知识点，获取NotORM实例有两种方式：

- 全局获取方式
- 局部获取方式

之所以强烈推荐使用局部获取方式，不仅是封装所带来的好处，更在于当配置了分表策略时，能更好地统一管控，避免过高的、人为的偶然复杂性。而这些技术债务，可以通过约定统一使用局部获取方式，在一开始就避免。

自动生成SQL建表语句

把数据库表的基本建表语句保存到./data目录下，文件名与数据库表名相同，后缀统一为 ".sql" 。如这里的./data/demo.sql文件。

```
`name` varchar(11) DEFAULT NULL,
```

需要注意的是，这里说的基本建表语句是指：仅是这个表所特有的字段，排除已固定公共有的自增主键id、扩展字段ext_data和CREATE TABLE关键字等。

然后可以使用phalapi-buildsqls脚本命令，快速自动生成demo缺省主表和全部分表的建表SQL语句。如下：

```
$ ./bin/phalapi-buildsqls ./config/dbs.php demo
```

正常情况下，会生成类似以下的SQL语句：

```

CREATE TABLE `demo` (
    `id` int(11) unsigned NOT NULL AUTO_INCREMENT,
    `name` varchar(11) DEFAULT NULL,
    `ext_data` text COMMENT 'json data here',
    PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `tbl_demo_0` ... ...;
CREATE TABLE `tbl_demo_1` ... ...;
CREATE TABLE `tbl_demo_2` ... ...;

```

在将上面的SQL语句导入数据库后，或者手动创建数据库表后，便可以像之前那样操作数据库了。

分库配置策略

在了解了分表策略后，再来了解分库配置策略就非常简单了。

分库是指，同一张表，不仅在逻辑上有分表（前面是配置在同一个数据库内），还可以在物理存储上存放在多个数据库服务器中。

例如对于日记表，我们可以配置10张分表，并且存放在两个数据库服务器上。也就是：

- 前面50张log日志表，tbl_log_0 ~ tbl_log_49，存在第一个数据库db_log_first
- 后面50张log日志表，tbl_log_50 ~ tbl_log_99，存在第二个数据库db_log_second

首先，通过./config/dbs.php的简单配置，就能实现连接多个数据库。假设我们有两个数据库：

- 第一个数据库：db_log_first
- 第二个数据库：db_log_second

假设都是MySQL数据库，按前面介绍的格式，则可以在./config/dbs.php文件中配置：

```

return array(
    /**
     * DB数据库服务器集群
     */
    'servers' => array(
        // 第一个数据库
        'db_master' => array(
            'host'      => '127.0.0.1',           //服务器标记
            'name'      => 'db_log_first',         //数据库域名
            'user'      => 'root',                 //数据库用户名
            'password'  => '',                   //数据库密码
            'port'      => 3306,                  //数据库端口
            'charset'   => 'UTF8',                //数据库字符集
        ),
        // 第二个数据库
        'db_ext' => array(
            'host'      => '192.168.1.100',        //服务器标记
            'name'      => 'db_log_second',        //数据库域名
            'user'      => 'root',                 //数据库用户名
            'password'  => '',                   //数据库密码
            'port'      => 3306,                  //数据库端口
            'charset'   => 'UTF8',                //数据库字符集
        ),
    ),
    // 略.....
)

```

第二步，再继续配置，指定不同的数据库分表使用哪个数据库。可以这样配置：

```

'tables' => array(
    //通用路由（默认的配置要保留，其他数据库表要用）
    '__default' => array(
        'prefix' => 'tbl_',
        'key' => 'id',
        'map' => array(
            array('db' => 'db_master'),
        ),
    ),
    'log' => array(
        'prefix' => 'tbl_',
        'key' => 'id',
        'map' => array(
            // 注意，是配置数据库标记，不是真实数据库名
            array('db' => 'db_master'),
            // 前面50张log日志表：db_log_first.tbl_log_0 ~ db_log_first.tbl_log_49
            array('start' => 0, 'end' => 49, 'db' => 'db_master'),
            // 后面50张log日志表：db_log_second.tbl_log_50 ~ db_log_second.tbl_log_99
            array('start' => 50, 'end' => 99, 'db' => 'db_ext'),
        ),
    ),
)

```

上面配置，分别配置两组分表的策略。前面50张log日志表：db_log_first.tbl_log_0 ~ db_log_first.tbl_log_49，存在db_master这份数据库配置的数据库服务器中；后面50张log日志表：db_log_second.tbl_log_50 ~ db_log_second.tbl_log_99则存在db_ext这个数据库标记的数据库服务器中。

最后，在Model层编写的代码和平时一样即可。不同的是，需要在获取NotORM实例时，指定哪张分表。

例如，可以为不同的用户存储在不同的日志分表。根据user_id对10进行求余，可得到日志分表位置。

实现代码如下：

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class Log extends NotORM {
    protected function getTableName($id) {
        // 此时的id是user_id
        $tableName = 'log';
        if ($id !== null) {
            $tableName .= '_' . ($id % 100);
        }
        return $tableName;
    }

    public function countWhick($userId) {
        // 获取NotORM时指定userId
        return $this->getORM($userId)->count();
    }
}
```

当需要查user_id = 1的日志有多少条时，就可以这样写：

```
$log = new \App\Model\Log();
$userLogCount = $log->count(1);

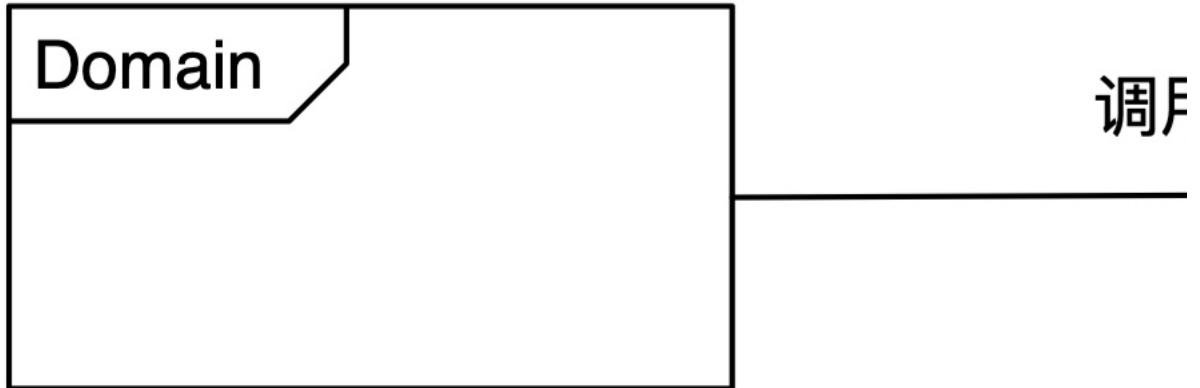
// 等效于：SELECT COUNT(*) FROM db_log_first.tbl_log_1
```

如果查user_id = 88的日志有多少条时，则可以这样写：

```
$log = new \App\Model\Log();
$userLogCount = $log->count(88);

// 等效于：SELECT COUNT(*) FROM db_log_second.tbl_log_88
```

此时的整体架构图如下：



恭喜你！又学习了分库分表的新技能！

连接多个数据库

在其他情况下，项目需要连接多个数据库也是常见的需求。解决方案可以有多种，简单的方案，可以通过配置直接实现，但有一定局限性。复杂的方案，能解决更多应用场景遇到的问题并能更好满足约束限制。

这一章，将带你开启一段组合爆炸的神奇旅程。但本质就看实际有多少个数据库，以及最终有多少个NotORM实例。请记住这个经验法则：

一个数据库，对应一个NotORM实例；但一个NotORM实例可以对应多个数据库。

简单方案：通过配置连接多个数据库

首先，通过./config/dbs.php的简单配置，就能实现连接多个数据库。

假设我们有两个数据库：

- 第一个数据库：db_1
- 第二个数据库：db_2

假设都是MySQL数据库，按前面介绍的格式，则可以在./config/dbs.php文件中配置：

```
return array(
    /**
     * DB数据库服务器集群
     */
    'servers' => array(
        // 第一个数据库
        'db_master' => array(
            'host'          => '127.0.0.1',           //数据库域名
            'name'          => 'db_1',                 //数据库名字
            'user'          => 'root',                //数据库用户名
            'password'      => '',                   //数据库密码
            'port'          => 3306,                  //数据库端口
            'charset'       => 'UTF8',                //数据库字符集
        ),
        // 第二个数据库
        'db_other' => array(
            'host'          => '192.168.1.100',        //数据库域名
            'name'          => 'db_2',                 //数据库名字
            'user'          => 'root',                //数据库用户名
            'password'      => '',                   //数据库密码
            'port'          => 3306,                  //数据库端口
            'charset'       => 'UTF8',                //数据库字符集
        ),
    ),
    // 略.....
)
```

第二步，再继续配置，不同的数据库表使用哪个数据库。参考分表配置的格式，只是这里是一个极端，即全部的分表只都有一张表，可以这样配置：

```
'tables' => array(
    // 库表: db_1.user
    'user' => array(
        'prefix' => 'tbl_',
        'key' => 'id',
        'map' => array('db' => 'db_master'),
    ),
),
),
// 库表: db_2.log
'log' => array(
    'prefix' => 'tbl_',
    'key' => 'id',
    'map' => array(
        'array('db' => 'db_other'),
    ),
),
),
```

上面配置，分别配置了user用户表用db_1， log日志表用db_2。其他依此类推。

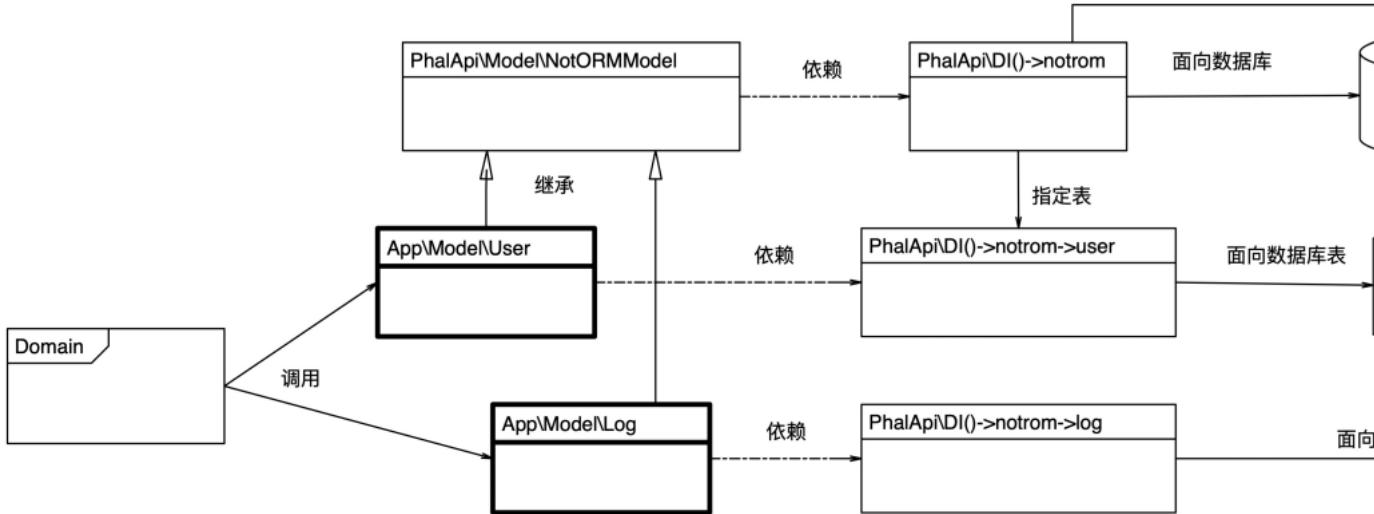
最后，在Model层编写的代码和平时一样即可。

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
    public function count() {
        // user表查db_1
        return $this->getORM()->count();
    }
}

// 另外的log表
class Log extends NotORM {
    public function count() {
        // log日记表查db_2
        return $this->getORM()->count();
    }
}
```

至此，我们就可以通过配置来实现连接多个数据库。当前，整体架构如下：



但局限是：

- 局限1：不同数据库不能有同名数据库表，否则会表名冲突。可以通过加前缀区分
- 局限2：只支持PhalApi默认的数据库类型，例如：MySQL

复杂方案：支持任意多个不同数据库

PhalApi 2.x 使用的是[NotORM](#)来进行数据库操作，而NotORM底层则是采用了PDO。目前，NotORM支持： MySQL, SQLite, PostgreSQL, MS SQL, Oracle (Dibi support is obsolete)。

当需要支持多个数据库时，可以按以下步骤来实现，共分为两大部分。第一部分，实现其他数据库的连接；第二部分，实现多个数据库共存。

第一部分如下：

- 第一步、每一个数据库，单独一份./config/dbs.php配置文件（可复制此文件，如：./config/dbs_2.php）
- 第二步、继承[PhalApi\Database\NotORMDatabase::createPDOBy\(\\$dbCfg\)](#)接口，并实现指定数据库PDO的创建和连接
- 第三步、在./config/di.php文件中，为新的数据库连接注册新的notorm服务

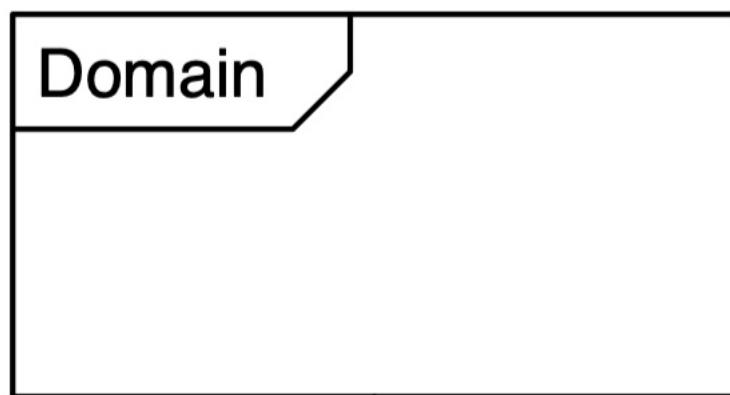
接着，是第二部分：

- 第四步、为新的数据库连接实现新的Model基类，继承并重载[PhalApi\Model\NotORMModel::getORM\(\\$id = NULL\)](#)方法，返回第三步的notorm服务
- 第五步、在Model层，在具体的Model子类中，继承第四步的基类
- 第六步，完成，正常的数据库操作

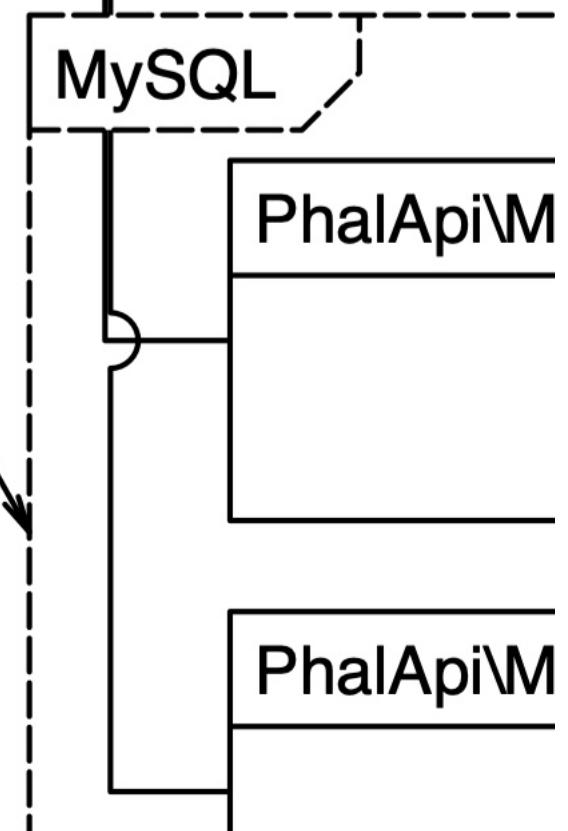
如果只有一个数据库，但不是MySQL数据库，则只需要完成第一部分；如果有多个数据库，则需要完成第一部分和第二部分。下面通过一个示例来概括介绍。

先来提前预览整体的架构，方便全局把控和了解。

继承



调用





假设，我们现在需要连接三个数据库，分别是：

数据库类型	数据库名称	数据库域名	数据库端口	数据库账号	数据库密码
MySQL	phalapi	192.168.1.1	3306	root	123456
Ms Server	phalapi_ms	192.168.1.2	1433	root	abcdef
postgreSQL	phalapi_pg	192.168.1.3	3306	root	abc123

为了能同时使得这三个数据库，第一步，为这三个数据库，分别准备以下配置三个dbs.php文件。

MySQL默认数据库的配置文件./config/dbs.php:

```
<?php
return array(
    'servers' => array(
        'db_master' => array(
            'host'      => '192.168.1.1',           //服务器标记
            'name'      => 'phalapi',                //数据库域名
            'user'      => 'root',                  //数据库用户名
            'password'  => '123456',                //数据库密码
            'port'      => 3306,                   //数据库端口
            'charset'   => 'UTF8',                 //数据库字符集
        ),
    ),
    /**
     * 自定义路由表
     */
    'tables' => array(
        //通用路由
        '__default__' => array(
            'prefix'  => 'tbl_',
            'key'     => 'id',
            'map'     => array(
                array('db' => 'db_master'),
            ),
        ),
    ),
);
);
```

MS Server的数据库配置文件，由于PhalApi 2.x内置已支持MS Server的连接，因此创建配置文件./config/dbs_ms.php，并放置：

```
<?php
return array(
    'servers' => array(
        'db_master' => array(
            'type'      => 'sqlsrv',             //服务器标记
            'host'      => '192.168.1.2',        //指定使用sqlsrv
            'name'      => 'phalapi_ms',         //数据库域名
            'user'      => 'root',                //数据库用户名
            'password'  => 'abcdef',              //数据库密码
            'port'      => 1433,                 //数据库端口
            'charset'   => 'UTF8',               //数据库字符集
        ),
    ),
    /**
     * 自定义路由表
     */
    'tables' => array(
        //通用路由
        '__default__' => array(
            'prefix'  => 'tbl_',
            'key'     => 'id',
            'map'     => array(
                array('db' => 'db_master'),
            ),
        ),
    ),
);
);
```

最后，是postgreSQL数据库的配置，在PhalApi 2.6.0 版本前，框架不支持此类型的数据库连接，需要创建配置文件./config/dbs_pg.php：并放置：

```
<?php
return array(
    'servers' => array(
        'db_master' => array(
            'type'      => 'pgsql',              //服务器标记
            // 指定使用pgsql
        ),
    ),
);
);
```

```

        'host'      => '192.168.1.3',           //数据库域名
        'name'      => 'phalapi_pg',            //数据库名字
        'user'      => 'root',                  //数据库用户名
        'password'  => 'abc123',                //数据库密码
        'port'      => 3306,                   //数据库端口
        'charset'   => 'UTF8',                 //数据库字符集
    ),
),
),

/***
 * 自定义路由表
 */
'tables' => array(
    //通用路由
    '__default__' => array(
        'prefix' => 'tbl_',
        'key' => 'id',
        'map' => array(
            array('db' => 'db_master'),
        ),
    ),
),
),
);

```

到这里，第一步完成。

第二步是，进行不同数据库的连接。参考PHP官方手册[PHP: PDO - Manua](#)， PDO可支持以下数据库的连接：

- MySQL (PDO)
- MS SQL Server (PDO)
- PostgreSQL (PDO)
- SQLite (PDO)
- Oracle (PDO)
- Firebird (PDO)
- CUBRID (PDO)
- IBM (PDO)
- Informix (PDO)
- ODBC and DB2 (PDO)
- 4D (PDO)

在PhalApi 2.5.0 版本后，可内置支持MySQL (PDO)、 MS SQL Server (PDO)、 PostgreSQL (PDO)，如果需要其他类型数据库的连接，则需要继承 PhalApi\Database\NotORMDatabase::createPDOBy(\$dbCfg) 接口，并实现指定数据库PDO的创建和连接。以PostgreSQL (PDO)为例，可以这样实现代码。创建./src/app/Common/MyPostgreDB.php文件，并放置以下代码。

```

<?php
namespace App\Common;

use PhalApi\Database;
use PhalApi\Database\NotORMDatabase;

class MyPostgreDB extends NotORMDatabase {
    protected function createPDOBy($dbCfg)
    {
        $dsn = sprintf('%s:dbname=%s;host=%s;port=%d',
            $dsn = sprintf('pgsql:dbname=%s;host=%s;port=%d',
                $dbCfg['name'],
                isset($dbCfg['host']) ? $dbCfg['host'] : 'localhost',
                isset($dbCfg['port']) ? $dbCfg['port'] : 3306
            );
        );
        $charset = isset($dbCfg['charset']) ? $dbCfg['charset'] : 'UTF8';
        $pdo = new \PDO(
            $dsn,
            $dbCfg['user'],
            $dbCfg['password']
        );
        $pdo->exec("SET NAMES '{$charset}'");
        return $pdo;
    }
}

```

在完成这些准备工作后，就可以在./config/di.php文件中，注册这些不同的数据库实例。在./config/di.php文件中添加以下代码。

```

// 数据操作 - 基于NotORM
$di->notorm = new NotORMDatabase($di->config->get('dbs'), $di->debug);

// 追加

// MS Server数据库
$di->notorm_ms = new NotORMDatabase($di->config->get('dbs_ms'), $di->debug);

// PostgreSQL数据库（切换成自己的新类）
$di->notorm_pg = new App\Common\MyPostgreDB($di->config->get('dbs_pg'), $di->debug);

```

下面进行第二部分，到了第四步，需要分别实现两个Model基类，分别用于MS Server数据库和PostgreSQL数据库。

首先，是MS Server数据库的Model基类，创建./src/app/Model/MSModelBase.php文件，代码如下：

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel;

class MSModelBase extends NotORMModel {
    protected function getORM($id = NULL) {
        $table = $this->getTableName($id);
        return \PhalApi\DI()->notorm_ms->$table; // 注意这一行，改为：notorm_ms
    }
}

```

```
}
```

然后，对于PostgreSQL数据库也这类似这样，即添加./src/app/Model/PostgreModelBase.php文件，代码如下：

```
<?php
namespace App\Model;
use PhalApi\Model\NotORMModel;

class PostgreModelBase extends NotORMModel {
    protected function getORM($id = NULL) {
        $table = $this->getTableName($id);
        return \PhalApi\DI()->notorm_pg->$table; // 注意这一行，改为：notorm_pg
    }
}
```

当你完成到这里，恭喜你，离成功不远啦！剩下的就是使用，基本上和平常的Model使用是一样的。

第五步，在需要的Model子类中，继承对应的数据表基类。为方便区分，可以为不同的数据库划分不同的目录。例如，对于MS Server，创建目录./src/app/Model/MSServer。假设有一张user的表，则可以创建./src/app/Model/MSServer/User.php文件，放置代码：

```
<?php
namespace App\Model\MSServer;
use App\Model\MSModelBase;

class User extends MSModelBase { // 注意，这里换成新的基类
    protected function getTableName($id) {
        return 'user';
    }
}
```

PostgreSQL和这类似，不再赘述。

最后一步，就可以正常使用啦。例如：

```
<?php
class User extends MSModelBase { // 注意，这里换成新的基类
    protected function getTableName($id) {
        return 'user';
    }

    public function count() {
        return $this->getORM()->count();
    }
}
```

搞定，收工！

小结

在PhalApi中，数据库操作主要是基于NotORM来实现。而对于数据库的连接，以及对于分库分表，则可以通过配置或者自定义开发来扩展。这种组合是非常灵活、优雅且设计巧妙的。

与传统的框架不同的是，PhalApi天生就支持多个数据库、分表分库的配置。更多复杂的组合功能，可以在熟悉前面这些配置和策略后自由发挥。期待你的大作品！

定制你的Model基类

如上面所介绍，在Model基类中，你可以完成很多事情，可以设置表名，可以指定使用哪个NotORM实例（在多数据库中特别有用），下面继续介绍更多高级的功能：如LOB序列化等。如果PhalApi现有的解决方案不能满足项目的需求，可作进行定制化处理。

默认的Model基类与接口

PhalApi基于NotORM的Model基类是PhalApi\Model\NotORMModel，它主要有以下接口：

- PhalApi\Model\NotORMModel::getTableName(\$id)，用于指定表名，或指定分表名
- PhalApi\Model\NotORMModel::getTableKey(\$table)，根据表名获取主键名
- PhalApi\Model\NotORMModel::getORM(\$id = NULL)，快速获取ORM实例，可用于切换数据库
- PhalApi\Model\NotORMModel::formatExtData(&\$data)，对LOB的ext_data字段进行格式化(序列化)
- PhalApi\Model\NotORMModel::parseExtData(&\$data)，对LOB的ext_data字段进行解析(反序列化)

下同分别介绍。

指定表名，指定分表名

这个特性，在前面数据库相关章节中已有介绍，这里再简单重温一下。

通常，框架会根据Model类名自动映射到表名。当需要手动指定表名时，可以这样写：

```
<?php
namespace App\Model;

use PhalApi\Model\NotORMModel as NotORM;

class User extends NotORM {
```

```

protected function getTableName($id) {
    return 'my_user'; // 手动设置表名为 my_user
}
}

```

如果存在分表，那么可以自定义分表策略，即根据什么参考依据，分多少张表。例如前面根据user_id对10进行求余，得到的日志分表。实现代码如下：

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel as NotORM;

class Log extends NotORM {
    protected function getTableName($id) {
        // 此时的id是user_id

        $tableName = 'log';
        if ($id !== null) {
            $tableName .= '_' . ($id % 100);
        }

        return $tableName;
    }

    public function countWhick($userId) {
        // 获取NotORM时指定userId
        return $this->getORM($userId)->count();
    }
}

```

根据表名获取主键名

并不是全部的表的主键名称都叫id，但我们希望能遵循这一国际惯例。

其次是分表处理，同样考虑到分表的情况，以及不同的表可能配置不同的主键表，而基于主键的CURD又必须要先知道表的主键名才能进行SQL查询。所以，问题就演变成了如何找到表的主键名。这里可以自动匹配，也可以手工指定。自动匹配是智能的，因为当我们更改表的主键时，可以自动同步更新而不需要担心遗漏（虽然这种情况很少发生）。手工指定可以大大减少系统不必要的匹配操作，因为我们开发人员也知道数据库的主键名是什么，但需要手工编写一些代码。在这里，提供了可选的手工指定，即可重写getTableKey(\$table)来指定你的主键名。

如，当user表的主键都为new_id时（希望不要真的发生）：

```

class User extends NotORM {
    protected function getTableKey($table) {
        return 'new_id';
    }
}

```

快速获取ORM实例，可用于切换数据库

这里所说的获取ORM实例，是指局部获取NotORM的方式，与之对应的是全局获取方式。

如果需要切换不同的数据库，那么可以在这里统一控制。例如前面可以写一个基类，统一切换到MS Server数据库。

```

<?php
namespace App\Model;
use PhalApi\Model\NotORMModel;

class MSModelBase extends NotORMModel {
    protected function getORM($id = NULL) {
        $table = $this->getTableName($id);
        return \PhalApi\DI()->notorm_ms->$table; // 注意这一行，改为: notorm_ms
    }
}

```

如果只有一个数据库，通常不用理会。当然，如果项目需要根据不同的场景切换数据库配置，除了可以在入口注册DI服务时使用不同配置外，也可以在这里切换。在这切换可以方便同时使用多个数据库。

序列化和反序列化

(1) LOB序列化

先是LOB序列化，考虑到有分表的存在，当发生数据库变更时（特别在线上环境）会有一定的难度和风险，因此引入了扩展字段ext_data。当然，此字段也应对数据库变更的同时，也可以作为简单明了的值对象的大对象。序列化LOB首先要考虑的问题是使用二进制（BLOB）还是文本（CLOB），出于通用性、易读性和测试性，我们目前使用了json格式的文本序列化。所以，如果考虑到空间或性能问题（在少量数据下我认为问题不大，如果数据量大，应该及时重新调整数据库表结构），可以重写formatExtData() & parseExtData()。

如改成serialize序列化：

```

<?php
namespace App\Common;

abstract class MyNotORM extends \PhalApi\Model\NotORMModel {

    /**
     * 对LOB的ext_data字段进行格式化(序列化)
     */
    protected function formatExtData(&$data) {
        if (isset($data['ext_data'])) {
            $data['ext_data'] = serialize($data['ext_data']);
        }
    }
}

```

```

}

/**
 * 对LOB的ext_data字段进行解析(反序列化)
 */
protected function parseExtData(&$data) {
    if (isset($data['ext_data'])) {
        $data['ext_data'] = unserialize($data['ext_data'], true);
    }
}

// ...
}

```

将Model类继承于App\Common\MyNotORM后,

```

// $ vim ./app/Model/User.php

<?php
namespace App\Model;

class User extends \App\Common\MyNotORMModel {
    //...
}

就可以轻松切换到序列化, 如:

$model = new \App\Model\User();

//带有ext_data的更新
$extData = array('level' => 3, 'coins' => 256);
$data = array('name' => 'test', 'update_time' => time(), 'ext_data' => $extData);
$model->update(1, $data); //基于主键的快速更新

```

接口参数

参数规则格式

参数规则是针对各个接口服务而配置的多维规则数组, 由接口类的getRules()方法返回。其中,

- 一维下标是接口类的方法名, 对应接口服务的Action;
- 二维下标是类属性名称, 对应在服务端获取通过验证和转换化的最终客户端参数;
- 三维下标name是接口参数名称, 对应外部客户端请求时需要提供的参数名称。

小结如下:

```

public function getRules() {
    return array(
        '接口类方法名' => array(
            '接口类属性' => array('name' => '接口参数名称', ...),
        ),
    );
}

```

在接口实现类里面getRules()成员方法配置参数规则后, 便可以通过类属性的方式, 根据配置指定的名称获取对应的接口参数, 如上面的: \$this->username 和\$this->password。

三级参数规则配置

参数规则主要有三种, 分别是: 系统参数规则、应用参数规则、接口参数规则。

系统参数

系统参数是指被框架保留使用的参数。目前已被PhalApi占用的系统参数只有一个, 即: service参数 (缩写为s参数), 前面已有介绍。

应用参数

应用参数是指在一个接口系统中, 全部项目的全部接口都需要的参数, 或者通用的参数。假如我们的商城接口系统中全部的接口服务都需要必须的签名sign参数, 以及非必须的版本号, 则可以在./config/app.php中的apiCommonRules进行应用参数规则的配置:

```

<?php
return array(
    /**
     * 应用接口层的统一参数
     */
    'apiCommonRules' => array(
        //签名
        'sign' => array(
            'name' => 'sign', 'require' => true,
        ),
        //客户端App版本号, 默认为: 1.4.0
        'version' => array(
            'name' => 'version', 'default' => '1.4.0',
        ),
    ),
)

```

接口参数

接口参数是指各个具体的接口服务所需要的参数，为特定的接口服务所持有，独立配置。并且进一步在内部又细分为两种：

- **通用接口参数规则**：使用*作为下标，对当前接口类全部的方法有效。
- **指定接口参数规则**：使用方法名作为下标，只对接口类的特定某个方法有效。

例如为了加强安全性，需要为全部的用户接口服务都加上长度为4位的验证码参数：

```
public function getRules() {
    return array(
        '*' => array(
            'code' => array('name' => 'code', 'require' => true, 'min' => 4, 'max' => 4),
            'login' => array(
                'username' => array('name' => 'username', 'require' => true),
                'password' => array('name' => 'password', 'require' => true, 'min' => 6),
            ),
        );
}
```

现在，当再次请求用户登录接口，除了要提供用户名和密码外，我们还要提供验证码code参数。并且，对于Api\User类的其他方法也一样。

多个参数规则时的优先级

当同一个参数规则分别在应用参数、通用接口参数及指定接口参数出现时，后面的规则会覆盖前面的，即具体化的规则会替换通用的规则，以保证接口参数满足特定场合的定制要求。

简而言之，多个参数规则的优先级从高到下，分别是（正如你想到的那样）：

- 1、指定接口参数规则
- 2、通用接口参数规则
- 3、应用参数规则
- 4、系统参数规则

温馨提示：如果对过滤器配置了白名单，必选参数最终会自动切换为可选参数，即require = false，详细请参考[白名单配置](#)。

参数规则配置详细说明

具体的参数规则，根据不同的类型有不同的配置选项，以及一些公共的配置选项。目前，主要的类型有：字符串、整数、浮点数、布尔值、时间戳/日期、数组、枚举类型、文件上传和回调函数。

类型 type	参数名称 name	是否必须 require	默认值 default	最小值 min, 最大值 max	更多配置选项 (无特殊说明, 均为可选)
字符串 string		TRUE/FALSE, 默认FALSE	应为字符串	可选	regex选项用于配置正则匹配的规则；format选项用于定义字符编码的类型，如utf8、gbk、gb2312等
整数 int		TRUE/FALSE, 默认FALSE	应为整数	可选	---
浮点数 float		TRUE/FALSE, 默认FALSE	应为浮点数	可选	---
布尔值 boolean		TRUE/FALSE, 默认FALSE	true/false	---	以下值会转换为TRUE: ok, true, success, on, yes, 1，以及其他PHP作为TRUE的值
时间戳/日期 date		TRUE/FALSE, 默认FALSE	日期字符串	可选，仅当为format配置为timestamp时才判断，且最大值应为时间戳	format选项用于配置格式，为timestamp时会将字符串的日期转换为时间戳
数组 array		TRUE/FALSE, 默认FALSE	字符串或者数组，为非数组会自动转换/解析成数组	可选，判断数组元素个数	format选项用于配置数组和格式，为explode时根据separator选项将字符串分割成数组，为json时进行JSON解析
枚举 enum		TRUE/FALSE, 默认FALSE	应为range选项中的某个元素	---	必须的range选项，为一数组，用于指定枚举的集合
文件 file		TRUE/FALSE, 默认FALSE	数组类型	可选，用于表示文件大小范围，单位为B	range选项用于指定可允许上传的文件类型；ext选项用于表示需要过滤的文件扩展名
回调 callable/callback		TRUE/FALSE, 默认FALSE	---	---	callable/callback选项用于设置回调函数，params选项为回调函数的第三个参数（另外第一个为参数值，第二个为所配置的规则）

公共参数配置选项

公共的配置选项，除了上面的类型、参数名称、是否必须、默认值，还有说明描述、数据来源。下面分别简单说明。

• 类型 type

用于指定参数的类型，可以是string、int、float、boolean、date、array、enum、file、callable，或者自定义的类型。未指定时，默认为字符串。

• 参数名称 name

接口参数名称，即客户端需要传递的参数名称。与PHP变量规则一样，以下划线或字母开头。此选项必须提供，否则会提示错误。

• 是否必须require

为TRUE时，表示此参数为必须值；为FALSE时，表示此参数为可选。未指定时，默认为FALSE。

- **默认值 default**

未提供接口参数时的默认值。未指定时，默认为NULL。

- **最小值 min, 最大值 max**

部分类型适用。用于指定接口参数的范围，比较时采用的是闭区间，即范围应该为：[min, max]。也可以只使用min或max，若只配置了min，则表示：[min, +∞)；若只配置了max，则表示：(-∞, max]。

- **说明描述 desc**

用于自动生成在线接口详情文档，对参数的含义和要求进行扼要说明。未指定时，默认为空字符串。

- **数据来源 source**

指定当前单个参数的数据来源，可以是post、get、cookie、server、request、header、或其他自定义来源。未指定时，默认为统一数据源。目前支持的source与对应的数据源映射关系如下：

- **错误提示 message**

如果配置此项，当接口参数错误时优先显示此错误提示信息，让开发人员可以自定义友好的错误提示信息，并支持i18n国际翻译。

温馨提示：message配置，需要PhalApi V2.5.0 及以上版本才支持。

source 对应的数据源

```
post    $_POST
get     $_GET
cookie  $_COOKIE
server  $_SERVER
request $_REQUEST
header  $_SERVER['HTTP_X']
```

通过source参数可以轻松、更自由获取不同来源的参数。以下是一些常用的配置示例。

```
// 获取HTTP请求方法，判断是POST还是GET
'method' => array('name' => 'REQUEST_METHOD', 'source' => 'server'),

// 获取COOKIE中的标识
'is_new_user' => array('name' => 'is_new_user', 'source' => 'cookie'),

// 获取HTTP头部中的编码，判断是否为utf-8
'charset' => array('name' => 'Accept-Charset', 'source' => 'header'),
```

若配置的source为无效或非法时，则会抛出异常。如配置了'source' => 'NOT_FOUND'，会得到：

```
"msg": "服务器运行错误：参数规则中未知的数据源：NOT_FOUND"
```

9种基本接口参数类型

对于各种参数类型，结合示例说明如下。

- **字符串 string**

当一个参数规则未指定类型时，默认为string。如最简单的：

```
array('name' => 'username')
```

温馨提示：这一小节的参数规则配置示例，都省略了类属性，以关注配置本身的内容。

这样就配置了一个参数规则，接口参数名字叫username，类型为字符串。

一个完整的写法可以为：

```
array('name' => 'username', 'type' => 'string', 'require' => true, 'default' => 'nobody', 'min' => 1, 'max' => 10)
```

这里指定了为必选参数，默认值为nobody，且最小长度为1个字符，最大长度为10个字符，若传递的参数长度过长，如&username=alonglonglonglongname，则会异常失败返回：

```
"msg": "非法请求：username.len应该小于等于10，但现在username.len = 21"
```

当需要验证的是中文的话，由于一个中文字符会占用3个字节。所以在min和max验证的时候会出现一些问题。为此，PhalApi提供了format配置选项，用于指定字符集。如：

```
array('name' => 'username', 'type' => 'string', 'format' => 'utf8', 'min' => 1, 'max' => 10)
```

我们还可以使用regex下标来进行正则表达式的验证，一个邮箱的例子是：

```
array('name' => 'email', 'regex' => "/^([0-9A-Za-z\\-_\\.]+@[0-9a-z]+\\.[a-z]{2,3}(\\.[a-z]{2})?)$/i")
```

- **整型 int**

整型即自然数，包括正数、0和负数。如通常数据库中的id，即可配置成：

```
array('name' => 'id', 'type' => 'int', 'require' => true, 'min' => 1)
```

当传递的参数，不在其配置的范围内时，如&id=0，则会异常失败返回：

```
"msg": "非法请求：id应该大于或等于1，但现在id = 0"
```

另外，对于常见的分页参数，可以这样配置：

```
array('name' => 'page_num', 'type' => 'int', 'min' => 1, 'max' => 20, 'default' => 20)
```

即每页数量最小1个，最大20个，默认20个。

- **浮点 float**

浮点型，类似整型的配置，此处略。

- **布尔值 boolean**

布尔值，主要是可以对一些字符串转换成布尔值，如ok, true, success, on, yes，以及会被PHP解析成true的字符串，都会转换成TRUE。如通常的“是否记住我”参数，可配置成：

```
array('name' => 'is_remember_me', 'type' => 'boolean', 'default' => TRUE)
```

则以下参数，最终服务端会作为TRUE接收。

```
?is_remember_me=ok  
?is_remember_me=true  
?is_remember_me=success  
?is_remember_me=on  
?is_remember_me=yes  
?is_remember_me=1
```

- **日期 date**

日期可以按自己约定的格式传递，默认是作为字符串，此时不支持范围检测。例如配置注册时间：

```
array('name' => 'register_date', 'type' => 'date')
```

对应地，register_date=2015-01-31 10:00:00则会被获取到为：“2015-01-31 10:00:00”。

当需要将字符串的日期转换成时间戳时，可追加配置选项'format' => 'timestamp'，则配置成：

```
array('name' => 'register_date', 'type' => 'date', 'format' => 'timestamp')
```

则上面的参数再请求时，则会被转换成：1422669600。

此时作为时间戳，还可以添加范围检测，如限制时间范围在31号当天：

```
array('name' => 'register_date', 'type' => 'date', 'format' => 'timestamp', 'min' => 1422633600, 'max' => 1422719999)
```

当配置的最小值或最大值为字符串的日期时，会自动先转换成时间戳再进行检测比较。如可以配置成：

```
array('name' => 'register_date', ... ... 'min' => '2015-01-31 00:00:00', 'max' => '2015-01-31 23:59:59')
```

- **数组 array**

很多时候在接口进行批量获取时，都需要提供一组参数，如多个ID，多个选项。这时可以使用数组来进行配置。如：

```
array('name' => 'uids', 'type' => 'array', 'format' => 'explode', 'separator' => ',')
```

这时接口参数&uids=1,2,3则会被转换成：

```
array ( 0 => '1', 1 => '2', 2 => '3', )
```

如果设置了默认值，那么默认值会从字符串，根据相应的format格式进行自动转换。如：

```
array( ... ... 'default' => '4,5,6')
```

那么在未传参数的情况下，自动会得到：

```
array ( 0 => '4', 1 => '5', 2 => '6', )
```

又如接口需要使用JSON来传递整块参数时，可以这样配置：

```
array('name' => 'params', 'type' => 'array', 'format' => 'json')
```

对应地，接口参数¶ms={"username":"test","password":"123456"}则会被转换成：

```
array ( 'username' => 'test', 'password' => '123456', )
```

温馨提示： 使用JSON传递参数时，建议使用POST方式传递。若使用GET方式，须注意参数长度不应超过浏览器最大限制长度，以及URL编码问题。

若使用JSON格式时，设置了默认值为：

```
array( ... ... 'default' => '{"username":"dogstar","password":"xxxxxx"})
```

那么在未传参数的情况下，会得到转换后的：

```
array ( 'username' => 'dogstar', 'password' => 'xxxxxx', )
```

特别地，当配置成了数组却未指定格式format时，接口参数会转换成只有一个元素的数组，如接口参数：&name=test，会转换成：

```
array ( 0 => 'test' )
```

- **枚举 enum**

在需要对接口参数进行范围限制时，可以使用此枚举型。如对于性别的参数，可以这样配置：

```
array('name' => 'sex', 'type' => 'enum', 'range' => array('female', 'male'))
```

当传递的参数不合法时，如`&sex=unknow`，则会被拦截，返回失败：

```
"msg": "非法请求：参数sex应该为：female/male，但现在sex = unknow"
```

关于枚举类型的配置，这里需要特别注意配置时，应尽量使用字符串的值。因为通常而言，接口通过GET/POST方式获取到的参数都是字符串的，而如果配置规则时指定范围用了整型，会导致底层规则验证时误判。例如接口参数为`&type=N`，而接口参数规则为：

```
array('name' => 'type', 'type' => 'enum', 'range' => array(0, 1, 2))
```

则会出现以下这样的误判：

```
var_dump(in_array('N', array(0, 1, 2))); // 结果为true，因为 'N' == 0
```

为了避免这类情况发生，应该使用使用字符串配置范围值，即可这样配置：

```
array('name' => 'type', 'type' => 'enum', 'range' => array('0', '1', '2'))
```

- **文件 file**

在需要对上传的文件进行过滤、接收和处理时，可以使用文件类型，如：

```
array(  
    'name' => 'upfile',  
    'type' => 'file',  
    'min' => 0,  
    'max' => 1024 * 1024,  
    'range' => array('image/jpeg', 'image/png'),  
    'ext' => array('jpeg', 'png')  
)
```

其中，`min`和`max`分别对应文件大小的范围，单位为字节；`range`为允许的文件类型，使用数组配置，且不区分大小写。

如果成功，返回的值对应的是`$_FILES["upfile"]`，即会返回：

```
array(  
    'name' => ..., // 被上传文件的名称  
    'type' => ..., // 被上传文件的类型  
    'size' => ..., // 被上传文件的大小，以字节计  
    'tmp_name' => ..., // 存储在服务器的文件的临时副本的名称  
)
```

对应的是：

- `$_FILES["upfile"]["name"]` - 被上传文件的名称
- `$_FILES["upfile"]["type"]` - 被上传文件的类型
- `$_FILES["upfile"]["size"]` - 被上传文件的大小，以字节计
- `$_FILES["upfile"]["tmp_name"]` - 存储在服务器的文件的临时副本的名称
- `$_FILES["upfile"]["error"]` - 由文件上传导致的错误代码

参考：以上内容来自W3School，文件上传时请使用表单上传，并`enctype`属性使用"multipart/form-data"。更多请参考[PHP 文件上传](#)。

若需要配置默认值`default`选项，则也应为一数组，且其格式应类似如上。

其中，`ext`是对文件后缀名进行验证，当如果上传文件后缀名不匹配时将抛出异常。文件扩展名的过滤可以类似这样进行配置：

- 单个后缀名 - 数组形式

```
'ext' => array('jpg')
```

- 单个后缀名 - 字符串形式

```
'ext' => 'jpg'
```

- 多个后缀名 - 数组形式

```
'ext' => array('jpg', 'jpeg', 'png', 'bmp')
```

- 多个后缀名 - 字符串形式 (以英文逗号分割)

```
'ext' => 'jpg, jpeg, png, bmp'
```

- **回调 callable/callback**

当需要利用已有函数进行自定义验证时，可采用回调参数规则，如配置规则：

```
array('name' => 'version', 'type' => 'callable', 'callback' => 'App\\Common\\Request\\Version::formatVersion')
```

然后，回调时将调用下面这个新增的类函数：

```
<?php  
namespace App\Common\Request;  
  
use PhalApi\Exception\BadRequestException;  
  
class Version {  
  
    public static function formatVersion($value, $rule) {  
        if (count(explode('.', $value)) < 3) {
```

```

        throw new BadRequestException('版本号格式错误');
    }
    return $value;
}

```

回调函数的签名为：function format(\$value, \$rule, \$params)，第一个为参数原始值，第二个为所配置的规则，第三个可选参数为配置规则中的params选项。最后应返回转换后的参数值。

扩展：定制接口参数来源、解密和预处理

把我们的API接口服务想象成一个函数，那么请求的参数就是我们的参数列表；而接口响应的数据则对应函数返回的结果。

对于请求，正如前面所看到的，我们可以使用\$_GET，也可以使用\$_POST，也可以两者都使用，还可以在测试时自己指定模拟的请求数据包。

或者，在实际项目开发中，我们还需要根据自身的需求，跟我们自己的客户端作一些约定。如通常地，我们会要求客户端 **service参数使用GET方式**，以便服务器返回500时定位接口服务位置。对此，简单的我们把\$_POST['service']去掉即可，如在入口文件前面添加：

```
unset($_POST['service']); //只接收GET方式的service参数
```

更高级的功能将介绍如下。

(1) 在index.php入口处指定数据源

很多时候，不同的项目对数据接收有不同的需求。如简单地，强制统一使用\$_POST参数，我们可以把在./config/di.php进行注册的代码调整：

```
// 注册新的请求服务
$di->request = new \PhalApi\Request($_POST); // 只允许POST参数

// JSON中文输出
// $di->response = new \PhalApi\Response\JsonResponse(JSON_UNESCAPED_UNICODE);
```

对于复杂的情况，如需要使用post_raw数据，则可以继承[PhalApi\Request](#)实现相应的数据源解析。如创建./src/app/Common/MyRequest.php文件。

```
<?php
namespace App\Common;
use PhalApi\Request;

class MyRequest extends Request {
    public function __construct($data = NULL) {
        parent::__construct($data);

        // json处理
        $this->post = json_decode(file_get_contents('php://input'), TRUE);

        // 普通xml处理
        $this->post = simplexml_load_string(
            file_get_contents('php://input'),
            'SimpleXMLElement',
            LIBXML_NOCDATA
        );
        $this->post = json_encode($this->post), TRUE);
    }
}
```

然后在子类实现对各类参数的数据源的准备。可以说，[PhalApi\Request::__construct\(\)](#)构造函数用于初始化各类辅助候选的数据源，而[PhalApi\Request::getData\(\)](#)则用于生成主要默认的数据源。

(2) 单元测试时指定数据源

在进行单元测试时，我们需要模拟接口的请求动作，也需要提供接口参数。这时的参数的指定更为灵活。可通过以下代码来实现，即：

```
//数据源
$data = array(...);

\PhalApi\DI()->request = new \PhalApi\Request($data);
```

或者使用PhalApi封装的测试类来快速模拟调用接口：

```
public function testIndex()
{
    //Step 1. 构建请求URL
    $url = 'service=App.Site.Index&username=dogstar';

    //Step 2. 执行请求
    $rs = TestRunner::go($url);

    //Step 3. 验证
    $this->assertNotEmpty($rs);
    $this->assertArrayHasKey('title', $rs);
}
```

(3) 接口数据的加密传送

有时，出于安全性的考虑，项目需要对请求的接口参数进行对称加密传送。这时可以通过重载[PhalApi\Request::genData\(\)](#)来轻松实现。

假设，我们现在需要把全部的参数base64编码序列化后通过\$_POST['data']来传递，则相应的解析代码如下。

第一步，先定义自己的扩展请求类，在里面完成对称解析的动作：

```

<?php
namespace App\Common;
use PhalApi\Request;

class MyRequest extends Request {
    public function genData($data) {
        if (!isset($data) || !is_array($data)) {
            $data = $_POST; //改成只接收POST
        }

        return isset($data['data']) ? base64_decode($data['data']) : array();
    }
}

```

第二步，在index.php入口文件中重新注册请求类（即添加以下代码）：

```

// 注册新的请求服务
$di->request = new \PhalApi\Request(); // 内含参数解密的实现

// JSON中文输出
// $di->response = new \PhalApi\Response\JsonResponse(JSON_UNESCAPED_UNICODE);

```

然后，就可以轻松实现了接口参数的对称加密传送。

至此，你也许已经发现：指定数据源和对称加密是可以结合起来使用的。

(4) 接口参数级别的数据源

除了可以指定全局的接口数据源外，还可以进行更细致的配置，即为某个接口参数指定使用\$_GET、\$_POST、\$_COOKIE、\$_SERVER、\$_REQUEST或头部等其他数据源。

其使用方式是在配置接口参数规则时，使用source配置来指定当前参数的数据源，如指定用户在登录时，用户名使用\$_GET、密码使用\$_POST。

```

public function getRules() {
    return array(
        'login' => array(
            'username' => array('name' => 'username', 'source' => 'get'),
            'password' => array('name' => 'password', 'source' => 'post'),
        ),
    );
}

```

此部分前面已有说明，不再赘述。

扩展：定制你的参数规则

当PhalApi提供的参数规则不能满足接口参数的规则验证时，除了使用callable类型进行扩展外，还可以扩展[PhalApi\Request\Formatter](#)接口来定制项目需要的类型。

一如既往，分两步：

- 1、扩展实现PhalApi\Request\Formatter接口
- 2、在DI注册你的参数规则新类型

下面以大家所熟悉的邮件类型为例，说明扩展的步骤。

首先，我们需要一个实现了邮件类型验证的功能类，创建./src/app/Common/EmailFormatter，放置代码：

```

<?php
namespace App\Common;

use PhalApi\Request\Formatter;
use PhalApi\Exception\BadRequestException;

class EmailFormatter implements Formatter {

    public function parse($value, $rule) {
        if (!preg_match('/^(\w+)(\.\w+)*@(\w+)((\.\w+)+)$/', $value)) {
            throw new BadRequestException('邮箱地址格式错误');
        }

        return $value;
    }
}

```

然后，在./config/di.php文件中追加注册：

```
$di->_formatterEmail = new App\Common\EmailFormatter();
```

温馨提示：在DI中手动注册服务时，名称的格式为：_formatter + 参数类型（首字母大写）。

系统已自动注册的格式化服务有：

- _formatterArray 数组格式化服务
- _formatterBoolean 布尔值格式化服务
- _formatterCallable 回调格式化服务
- _formatterDate 日期格式化服务
- _formatterEnum 枚举格式化服务
- _formatterFile 上传文件格式化服务
- _formatterFloat 浮点数格式化服务

- `_formatterInt` 整数格式化服务
- `_formatterString` 字符串格式化服务

至此，便可使用自己定制的类型规则了，

```
array('name' => 'user_email', 'type' => 'email')
```

配置

配置文件说明

默认情况下，项目里会有以下几个配置文件：

```
$ tree ./config/  
./Config/  
|   app.php  
|   dbs.php  
|   di.php  
|   sys.php
```

其中`app.php`为项目应用配置；`dbs.php`为分布式存储的数据库配置；`sys.php`为不同环境下的系统配置。这三个文件都是用于存放配置信息，可通过`\PhalApi\DI()>config`进行读取。

值得注意的是，`./config/di.php`文件则属于依赖注入配置文件，用于配置需在`\PhalApi\DI()`容器中注册的服务资源。

配置的简单读取

默认已注册配置组件服务，名称是`\PhalApi\DI()>config`。

```
// 配置  
$di->config = new FileConfig(API_ROOT . '/config');
```

假设`app.php`配置文件里有：

```
return array(  
    'version' => '1.1.1',  
    'email' => array(  
        'address' => 'chanzonghuang@gmail.com',  
    ),  
) ;
```

可以分别这样根据需要获取配置：

```
// app.php里面的全部配置  
\PhalApi\DI()>config->get('app'); //返回: array( ... )  
  
// app.php里面的单个配置  
\PhalApi\DI()>config->get('app.version'); //返回: 1.1.1  
  
// app.php里面的多级配置  
\PhalApi\DI()>config->get('app.email.address'); //返回: 'chanzonghuang@gmail.com'
```

其他配置文件的读取类似，你也可以根据需要添加新的配置文件。

读取失败与默认值

当一个配置不存在时，返回NULL。例如：

```
\PhalApi\DI()>config->get('app.not_found'); //返回: NULL
```

当需要指定默认值时，可通过第二个参数指定。例如：

```
\PhalApi\DI()>config->get('app.not_found', 404); //返回: 404
```

使用Yaconf扩展快速读取配置

Yaconf扩展需要PHP 7及以上版本，并且需要先安装Yaconf扩展。

温馨提示： Yaconf扩展的安装请参考[laruence/yaconf](#)。

安装部署完成后，先重新注册`\PhalApi\DI()>config`：

```
// 配置  
$di->config = new PhalApi\Config\YaconfConfig();
```

然后，便和正常的配置一样使用。

```
// 相当于Yaconf::get("foo")  
\PhalApi\DI()>config->get('foo')  
  
//相当于Yaconf::has("foo")  
\PhalApi\DI()>config->has('foo');
```

需要注意的是，使用Yaconf扩展与默认的文件配置的区别的是，配置文件的目录路径以及配置文件的格式。当然也可以把Yaconf扩展的配置目录路径设置到`\PhalApi`的配置目录`./config`。

扩展：其他配置读取方式

如果需要使用其他方式读取配置，可根据实情需要，实现[PhalApi\Config](#)接口，然后在[./config/di.php](#)文件重新注册[\PhalApi\DI\(\)->config](#)即可。

例如数据库配置，这样可以支持不发布代码的情况下进行配置更改。# 日志

关于日志接口，PSR规范中给出了相当好的说明和定义，并且有多种细分的日志级别。

```
/***
 * Describes log levels
 */
class LogLevel
{
    const EMERGENCY = 'emergency';
    const ALERT = 'alert';
    const CRITICAL = 'critical';
    const ERROR = 'error';
    const WARNING = 'warning';
    const NOTICE = 'notice';
    const INFO = 'info';
    const DEBUG = 'debug';
}
```

简化版的日志接口

虽然PSR规范中详尽定义了日志接口，然而在使用开源框架或内部框架进行项目开发过程中，实际上日志的分类并没有使用得那么丰富，通常只是频繁集中在某几类。为了减少不必要的复杂性，PhalApi特地将此规范的日志接口精简为三种，只有：

- **error**: 系统异常类日记
- **info**: 业务纪录类日记
- **debug**: 开发调试类日记

error 系统异常类日记

系统异常类日志用于纪录**在后端不应该发生却发生的事情**，即通常所说的系统异常。例如：调用第三方、的接口失败了，此时需要纪录一下当时的场景，以便复查和定位出错的原因。又如：写入一条纪录到数据纪录却失败了，此时需要纪录一下，以便进一步排查。

纪录系统异常日志，用法很简单。可以使用[PhalApi\Logger::error\(\\$msg, \\$data\)](#)接口，第一个参数\$msg用于描述日志信息，第二个可选参数为上下文场景的信息。下面是一些使用示例。

```
// 只有描述
\PhalApi\DI()->logger->error('fail to insert DB');

// 描述 + 简单的信息
\PhalApi\DI()->logger->error('fail to insert DB', 'try to register user dogstar');

// 描述 + 当时的上下文数据
$data = array('name' => 'dogstar', 'password' => '123456');
\PhalApi\DI()->logger->error('fail to insert DB', $data);
```

上面三条纪录，会在日记文件中生成类似以下的日志内容。

```
$ tailf ./runtime/log/201502/20150207.log
2015-02-07 20:37:55|ERROR|fail to insert DB
2015-02-07 20:37:55|ERROR|fail to insert DB|try to register user dogstar
2015-02-07 20:37:55|ERROR|fail to insert DB|{"name":"dogstar","password":"123456"}
```

info 业务纪录类日记

业务纪录日志，是指纪录业务上关键流程环节的操作，以便发生系统问题后进行回滚处理、问题排查以及数据统计。如在有缓存的情况下，可能数据没及时写入数据库而导致数据丢失或者回档，这里可以通过日志简单查看是否可以恢复。以及说明一下操作发生的背景或原由，如通常游戏中用户的经验值添加：

```
// 假设：10 + 2 = 12
\PhalApi\DI()->logger->info('add user exp', array('name' => 'dogstar', 'before' => 10, 'addExp' => 2, 'after' => 12, 'reason' => 'help one more phper'));
```

对应的日记为：

```
2015-02-07 20:48:51|INFO|add user exp|{"name":"dogstar","before":10,"addExp":2,"after":12,"reason":"help one more phper"}
```

debug 开发调试类日记

开发调试类日记，主要用于开发过程中的调试。用法如上，这里不再赘述。以下是一些简单的示例。

```
// 只有描述
\PhalApi\DI()->logger->debug('just for test');

// 描述 + 简单的信息
\PhalApi\DI()->logger->debug('just for test', '一些其他的描述 ...');

// 描述 + 当时的上下文数据
\PhalApi\DI()->logger->debug('just for test', array('name' => 'dogstar', 'password' => '*****'));
```

更灵活的日志分类

若上面的error、info、debug都不能满足项目的需求时，可以使用[PhalApi\Logger::log\(\\$type, \\$msg, \\$data\)](#)接口进行更灵活的日记纪录。

```
\PhalApi\DI()->logger->log('demo', 'add user exp', array('name' => 'dogstar', 'after' => 12));
\PhalApi\DI()->logger->log('test', 'add user exp', array('name' => 'dogstar', 'after' => 12));
```

对应的日记为：

```
2015-02-07 21:13:27|DEMO|add user exp|{"name":"dogstar","after":12}
2015-02-07 21:15:39|TEST|add user exp|{"name":"dogstar","after":12}
```

注意到，第一个参数为日记分类的名称，在写入日记时会自动转换为大写。其接口函数签名为：

```
/**
 * 日记纪录
 *
 * 可根据不同需要，将日记写入不同的媒介
 *
 * @param string $type 日记类型，如：info/debug/error, etc
 * @param string $msg 日记关键描述
 * @param string/array $data 场景上下文信息
 * @return NULL
 */
abstract public function log($type, $msg, $data);
```

指定日志级别

在使用日志纪录前，在注册日志[\PhalApi\DI\(\)->logger](#)服务时须指定开启的日志级别，以便允许指定级别的日志得以纪录，从而达到选择性保存所需要的目的。

通过[PhalApi\Logger](#)的构造函数的参数，可以指定日志级别。多个日记级别使用或运算进行组合。

```
// 日记纪录
$di->logger = new FileLogger(API_ROOT . '/runtime', Logger::LOG_LEVEL_DEBUG | Logger::LOG_LEVEL_INFO | Logger::LOG_LEVEL_ERROR);
```

上面的三类日记分别对应的标识如下。

日志类型	日志级别标识
error 系统异常类	PhalApi\Logger::LOG_LEVEL_ERROR
info 业务纪录类	PhalApi\Logger::LOG_LEVEL_INFO
debug 开发调试类	PhalApi\Logger::LOG_LEVEL_DEBUG

扩展：定制你的日志

普遍情况下，我们认为将日记存放在文件是比较合理的，因为便于查看、管理和统计。当然，如果你的项目需要将日记纪录保存在其他存储媒介中，也可以快速扩展实现的。例如实现数据库的存储思路。

```
<?php
namespace App\Common\Logger;

use PhalApi\Logger;

class DBLogger extends Logger {

    public function log($type, $msg, $data) {
        // TODO 数据库的日记写入 ...
    }
}
```

随后，重新注册[\PhalApi\DI\(\)->logger](#)服务即可。

```
$di->logger = new App\Common\Logger\DBLogger(API_ROOT . '/runtime', Logger::LOG_LEVEL_DEBUG | Logger::LOG_LEVEL_INFO | Logger::LOG_LEVEL_ERROR);
```

缓存

这一章，将从简单的缓存、再到高速缓存、最后延伸到多级缓存，逐步进行说明。

简单本地缓存

这里所指的简单缓存，主要是存储在单台服务器上的缓存，例如使用系统文件的文件缓存，PHP语言提供的APCU缓存。因为实现简单，且部署方便。但其缺点也是明显的，如文件I/O读写导致性能低，不能支持分布式。所以在没有集群服务器下是适用的。

文件缓存

例如，当需要使用文件缓存时，先在DI容器中注册对文件缓存到[\PhalApi\DI\(\)->cache](#)。

```
$di->cache = new PhalApi\Cache\FileCache(array('path' => API_ROOT . '/runtime', 'prefix' => 'demo'));
```

初始化文件缓存时，需要传入配置数组，其中path为缓存数据的目录，可选的前缀prefix，用于区别不同的项目。

然后便可在适当的场景使用缓存。

```
// 设置  
PhalApi\DI()->cache->set('thisYear', 2015, 600);  
  
// 获取, 输出: 2015  
echo PhalApi\DI()->cache->get('thisYear');  
  
// 删除  
PhalApi\DI()->cache->delete('thisYear');
```

可以看到, 在指定的缓存目录下会有类似以下这样的缓存文件。

```
$ tree ./runtime/cache/  
./runtime/cache/  
└── 483  
    └── 11303fe8f96da746aa296d1b0c11d243.dat
```

APCU缓存

安装好APCU扩展和设置相关配置并重启PHP后, 便可开始使用APCU缓存。APCU缓存的初始化比较简单, 只需要简单创建实例即可, 不需要任何配置。

```
$di)->cache = new PhalApi\Cache\APCUCache();
```

其他使用参考缓存接口, 这里不再赘述。

高速集群缓存

这里的高速集群缓存, 是指具备分布式存储能力, 并且进驻内存的缓存机制。高速集群缓存性能优于简单缓存, 并且能够存储的缓存容量更大, 通常配置在其他服务器, 即与应用服务器分开部署。其缺点是需要安装相应的PHP扩展, 另外部署缓存服务, 例如常见的Memcached、Redis。若需要考虑缓存落地, 还要进一步配置。

Memcache/Memcached缓存

若需要使用Memcache/Memcached缓存, 则需要安装相应的PHP扩展。PHP 7中已经逐渐不支持Memcache, 因此建议尽量使用Memcached扩展。

如使用Memcached:

```
$di->cache = new PhalApi\Cache\MemcachedCache(array('host' => '127.0.0.1', 'port' => 11211, 'prefix' => 'demo_'));
```

初始化Memcached时, 需要传递一个配置数组, 其中host为缓存服务器, port为缓存端口, prefix为可选的前缀, 用于区别不同的项目。配置前缀, 可以防止同一台MC服务器同一端口下key名冲突。对于缓存的配置, 更好的建议是使用配置文件来统一管理配置。例如调整成:

```
$di->cache = new PhalApi\Cache\MemcachedCache(DI()->config->get('sys.mc'));
```

相应的配置, 则在./config/sys.php中的mc选项中统一维护。

完成了Memcached的初始化和注册后, 便可考缓存接口进行使用, 这里不再赘述。Memcache的初始化和配置和Memcached一样。

如何配置多个Memcache/Memcached实例?

实际项目开发中, 当需要连接多个Memcache/Memcached实例, 可以在单个实例配置基础上采用以下配置:

```
$config = array(  
    'host'    => '192.168.1.1, 192.168.1.2', //多个用英文逗号分割  
    'port'    => '11211, 11212', //多个用英文逗号分割  
    'weight'  => '20, 80', // (可选) 多个用英文逗号分割  
)  
  
$di->cache = new PhalApi\Cache\MemcachedCache($config);
```

上面配置了两个MC实例, 分别是:

- 192.168.1.1, 端口为11211, 权重为20
- 192.168.1.2, 端口为11212, 权重为80

其中, 权重是可选的。并且以host域名的数量为基准, 即最终MC实例数量以host的个数为准。端口数量不足时取默认值11211, 多出的端口会被忽略; 同样, 权重数量不足时取默认值0, 多出的权重会被忽略。

如下, 是一份稀疏配置:

```
$config = array(  
    'host'    => '192.168.1.1, 192.168.1.2, 192.168.1.3',  
    'port'    => '11210',  
)
```

相当于:

- 192.168.1.1, 端口为11210, 权重为0(默认值)
- 192.168.1.2, 端口为11211(默认值), 权重为0(默认值)
- 192.168.1.3, 端口为11211(默认值), 权重为0(默认值)

请注意, 通常不建议在权重weight使用稀疏配置, 即要么全部不配置权重, 要么全部配置权重, 以免部分使用默认权重为0的MC实例不生效。

Redis缓存

当需要使用Redis缓存时，需要先安装对应的Redis扩展。

简单的Redis缓存的初始化如下：

```
$config = array('host' => '127.0.0.1', 'port' => 6379);
$di->cache = new PhalApi\Cache\RedisCache($config);
```

关于Redis的配置，更多选项如下。

Redis配置项	是否必须	默认值	说明
type	否	unix	当为unix时使用socket连接，否则使用http连接
socket	type为unix时必须	无	unix连接方式
host	type不为unix时必须	无	Redis域名
port	type不为unix时必须	6379	Redis端口
timeout	否	300	连接超时时间，单位秒
prefix	否	phalapi:	key前缀
auth	否	空	Redis身份验证
db	否	0	Redis库

扩展：添加新的缓存实现

当需要实现其他缓存机制时，例如使用COOKIE、SESSION、数据库等其他方式的缓存，可以先实现具体的缓存类，再重新注册\PhalApi\DI()->cache即可。

首先，简单了解下PhalApi中的缓存接口[PhalApi\Cache](#)。

```
<?php
namespace PhalApi;

/**
 * PhalApi\Cache 缓存接口
 *
 * @package    PhalApi\Cache
 * @license    http://www.phalapi.net/license GPL 协议
 * @link      http://www.phalapi.net/
 * @author    dogstar <chanzonghuang@gmail.com> 2015-02-04
 */

interface Cache {
    /**
     * 设置缓存
     *
     * @param string $key 缓存key
     * @param mixed $value 缓存的内容
     * @param int $expire 缓存有效时间，单位秒，非时间戳
     */
    public function set($key, $value, $expire = 600);

    /**
     * 读取缓存
     *
     * @param string $key 缓存key
     * @return mixed 失败情况下返回NULL
     */
    public function get($key);

    /**
     * 删除缓存
     *
     * @param string $key
     */
    public function delete($key);
}
```

此PhalApi\Cache缓存接口，主要有三个操作：设置缓存、获取缓存、删除缓存。设置时，缓存不存在时添加，缓存存在时则更新，过期时间单位为秒。当获取失败时，约定返回NULL。

所以，新的缓存实现类应按规约层的接口签名完成此缓存接口的实现。#过滤器

默认可用的MD5签名

基于很多同学对接口签名验证比较陌生，PhalApi提供了一个基本版的接口验证服务。主要是基于md5进行的签名生成，这个只能作为一般性的参考。大家可以在此基础上进行调整延伸。

默认情况下，在./config/di.php文件中去掉注释便可开启此接口验证，即：

```
// 签名验证服务
$di->filter = new \PhalApi\Filter\SimpleMD5Filter();
```

其验签的算法如下（如注释所示）：

- 1、排除签名参数（默认是sign）
- 2、将剩下的全部参数，按参数名字进行字典排序
- 3、将排序好的参数，全部用字符串拼接起来
- 4、进行md5运算

以下面的示例参数为例，即：

1、排除签名参数（默认是sign）
?service=Examples_CURD.Get&id=1

2、将剩下的全部参数，按参数名字进行字典排序
id=1
service=Examples_CURD.Get

3、将排序好的参数，全部用字符串拼接起来
"1Examples_CURD.Get" = "1" + "Examples_CURD.Get"

4、进行md5运算
sign = 3ba5f5f03a90b2a648f5dd1df7387e26 = md5("1Examples_CURD.Get")

5、请求时，加上签名参数
?service=Examples_CURD.Get&id=1&sign=3ba5f5f03a90b2a648f5dd1df7387e26

下面是两个调用示例，错误请求下（即签名失败）：

http://dev.phalapi.net/?service=Examples_CURD.Get&id=1&sign=xxx

返回：
{
 "ret": 406,
 "data": [],
 "msg": "非法请求：签名错误"
}

温馨提示：签名错误情况下，可以查看日记获得正确的sign，如：

2017-07-22 12:02:18|DEBUG|Wrong Sign| {"needSign": "3ba5f5f03a90b2a648f5dd1df7387e26"}

正常请求下（带sign签名）：

http://dev.phalapi.net/?service=Examples_CURD.Get&id=1&sign=3ba5f5f03a90b2a648f5dd1df7387e26

如果不使用sign作为关键的签名参数，可以在注册时指定，如使用缩写s：

\$di->filter = new \PhalApi\Filter\SimpleMD5Filter('s');

白名单配置

对于不需要进行签名验证的接口服务，可以使用白名单配置，通过框架自身实现对指定配置的接口服务排除。即调用的接口服务，如果配置了白名单，则不调用过滤器。

接口服务白名单配置是：app.service_whitelist，即配置文件./config/app.php里面的service_whitelist配置，其默认值是：

```
'service_whitelist' => array(  
    'Site.Index',  
)
```

如源代码里的注释所示，配置的格式有以下四种。

类型	配置格式	匹配规则	示例及说明
全部	*.*	匹配全部接口服务（慎用！）	如果配置了此规则，即全部的接口服务都不触发过滤器。
方法通配	Site.*	匹配某个类的任何方法	即App\Api\Site接口类的全部方法
类通配	*.Index	匹配全部接口类的某个方法	即全部接口类的Index方法
具体匹配	Site.Index	匹配指定某个接口服务	即App\Api\Site::Index()

如果有多个生效的规则，按短路判断原则，即有任何一个白名单规则匹配后就跳过验证，不触发过滤器。

以下是更多的示例：

```
'service_whitelist' => array(  
    '*.Index', // 全部的Index方法  
    'Test.*', // Api_Test的全部方法  
    'User.GetBaseInfo', // Api_User::GetBaseInfo()方法  
)
```

配置好上面的白名单后，以下这些接口服务全部不会触发过滤器：

```
// 全部的Index方法  
?service=Site.Index  
?service=User.Index  
  
// Api_Test的全部方法  
?service=Test.DoSth  
?service=Test.Hello  
?service=Test.GOGOGO  
  
// Api_User::GetBaseInfo()方法  
?service=User.GetBaseInfo
```

命名空间白名单独立配置

如果需要为不同的命名空间独立配置白名单，只需要简单加多一层配置即可，即单独配置的路径是：

app.service_whitelist.{命名空间}

对应的配置示例是：

```
'service_whitelist' => array(  
    'Site.Index',
```

```
// 以命名空间名称为key
'App' => array(
    // 在这里, 单独配置……
),
),
```

更好地建议

通常关于接口签名这块, 我们还需要:

- 1、为不同的接入方定义不同的密钥和私钥;
- 2、如果业务需要, 为各个接口、各个接入方分配调用权限;
- 3、统一签名参数的规则, 可以配置在./config/app.php中的, 如上面的签名需要的参数, 我们可以追加统一的参数规则:

```
/*
 * 应用接口层的统一参数
 */
'apiCommonRules' => array(
    'signature' => array('name' => 'signature', 'require' => true),
    'timestamp' => array('name' => 'timestamp', 'require' => true),
    'nonce' => array('name' => 'nonce', 'require' => true),
),
```

扩展: 实现你的签名方式

如果我们需要实现签名验证, 只需要简单的两步即可:

- 1、实现过滤器接口 [PhalApi\Filter::check\(\)](#);
- 2、注册过滤器服务 [PhalApi\DI\(\)->filter](#);

下面以大家熟悉的 [微信验签](#) 为例, 进行示例说明。

实现过滤器接口

通常我们约定返回ret = 402表示验证失败, 所以当签名失败时, 我们可以返回ret = 402以告知客户端签名不对。根据微信的检验signature的PHP示例代码, 我们可以快速实现自定义签名规则, 如:

```
// 文件 ./src/app/Common/SignFilter.php
<?php
namespace App\Common;

use PhalApi\Filter;
use PhalApi\Exception\BadRequestException;

class SignFilter implements Filter
{
    public function check()
    {
        $signature = \PhalApi\DI()->request->get('signature');
        $timestamp = \PhalApi\DI()->request->get('timestamp');
        $nonce = \PhalApi\DI()->request->get('nonce');

        $token = 'Your Token Here ...';
        $tmpArr = array($token, $timestamp, $nonce);
        sort($tmpArr, SORT_STRING);
        $tmpStr = implode( $tmpArr );
        $tmpStr = sha1( $tmpStr );

        if ($tmpStr != $signature) {
            throw new BadRequestException('wrong sign', 1);
        }
    }
}
```

注册过滤器服务

随后, 我们只需要再简单地注册一下过滤器服务即可, 在./config/di.php文件最后追加:

```
// 签名验证服务
$di->filter = new App\Common\SignFilter();
```

COOKIE

当使用HTTP/HTTPS协议并需要使用COOKIE时, 可参考此部分的使用说明。

COOKIE的基本使用

如同其他的服务一样, 我们在使用前需要对COOKIE进行注册。COOKIE服务注册在\PhalApi\DI()->cookie中, 可以使用[PhalApi\Cookie](#)实例进行初始化, 如:

```
$config = array('domain' => '.phalapi.net');
\PhalApi\DI()->cookie = new PhalApi\Cookie($config);
```

其中, [PhalApi\Cookie](#)的构造函数是一个配置数组, 上面指定了Cookie的有效域名/子域名。其他的选项还有:

配置选项	说明	默认值
------	----	-----

配置选项	说明	默认值
path	Cookie有效的服务器路径	NULL
domain	Cookie的有效域名/子域名	NULL
secure	是否仅仅通过安全的HTTPS连接传给客户端	FALSE
httponly	是否仅可通过HTTP协议访问	FALSE

注册COOKIE服务后，便可以开始在项目中使用了。COOKIE的使用主要有三种操作，分别是：设置COOKIE、获取COOKIE、删除COOKIE。下面是一些简单的使用示例。

```
// 设置COOKIE
// Set-Cookie:"name=phalapi; expires=Sun, 07-May-2017 03:26:45 GMT; domain=.phalapi.net"
\PhalApi\DI()->cookie->set('name', 'phalapi', $_SERVER['REQUEST_TIME'] + 600);

// 获取COOKIE，输出: phalapi
echo \PhalApi\DI()->cookie->get('name');

// 删除COOKIE
\PhalApi\DI()->cookie->delete('name');
```

记忆加密升级版

实际情况，项目对于COOKIE的使用情况更为复杂。比如，需要对数据进行加解密，或者需要突破COOKIE设置后下一次请求才能生效的限制。为此，PhalApi提供一个升级版的COOKIE服务。其特点主要有：

- 1、对COOKIE进行加密输出、解密获取
- 2、自带记忆功能，即本次请求设置COOKIE后便可直接获取

当需要使用这个升级版COOKIE替代简单版COOKIE服务时，可使用[PhalApi\Cookie\MultiCookie](#)实例进行重新注册。在初始化时，PhalApi\Cookie\MultiCookie构建函数的第一个参数配置数组，除了上面简单版的配置项外，还有：

配置选项	说明	默认值
crypt	加解密服务，须实现PhalApi\Crypt接口	\PhalApi\DI()->crypt
key	crypt使用的密钥	debcf37743b7c835ba367548f07aad3

假设项目中简单地使用base64对COOKIE进行加解密，则可先添加加解密服务的实现类。

```
<?php
namespace App\Common\Crypt\Base64Crypt;

use PhalApi\Crypt;

class Base64Crypt implements Crypt {
    public function encrypt($data, $key) {
        return base64_encode($data);
    }

    public function decrypt($data, $key) {
        return base64_decode($data);
    }
}
```

随后，在文件./config/di.php使用该加解密实现类重新注册\PhalApi\DI()->cookie服务，由于加解密中未使用到密钥\$key，所以可以不用配置。

```
$config = array('domain' => '.phalapi.net', 'crypt' => new App\Common\Crypt\Base64Crypt());
$di->cookie = new PhalApi\Cookie\Multi($config);
```

最后，便可在项目中像简单版原来那样使用升级版的COOKIE服务了，但设置的COOKIE值则是经过加密后的。

```
// 设置COOKIE
// Set-Cookie:"name=cGhhbGFwaQ%3D%3D; expires=Sun, 07-May-2017 03:27:57 GMT; domain=.phalapi.net"
\PhalApi\DI()->cookie->set('name', 'phalapi', $_SERVER['REQUEST_TIME'] + 600);
```

此外，在同一次请求中，设置了某个COOKIE后，也可以“即时”获取了。

在使用COOKIE时，需要注意：

- 1、敏感数据不要存到COOKIE，以保证数据安全性
- 2、尽量不要在COOKIE存放过多数据，避免产生不必要的流量消耗

扩展：定制专属的COOKIE

当项目中需要定制专属的COOKIE服务时，可以继承[PhalApi\Cookie](#)基类，并按需要重写对应的接口。主要的接口有三个：

- **设置COOKIE**: PhalApi\Cookie::set(\$name, \$value, \$expire = NULL)
- **获取COOKIE**: PhalApi\Cookie::get(\$name = NULL)
- **删除COOKIE**: PhalApi\Cookie::delete(\$name)

值得注意的是，在实现子类的构造函数中，需要调用PhalApi_Cookie基类的构造方法，以便初始化配置选项。实现子类后，重新注册便可使用，这里不再赘述。# 加密

PHP的mcrypt加密扩展

在PhalApi中，同样也是使用了mcrypt作为底层的数据加密技术方案。请查看：[PHP手册 函数参考 加密扩展](#)。

不过需要注意的是，在PHP7中，将废弃此扩展。

加解密的使用

在单元测试中，我们可以快速找到加密和解密的使用，这里再简单举一例：

```
$mcrypt = new PhalApi\Crypt\McryptCrypt('12345678');

$data = 'The Best Day of My Life';
$key = 'phalapi';

$encryptData = $mcrypt->encrypt($data, $key);
var_dump($encryptData);

$decryptData = $mcrypt->decrypt($encryptData, $key);
var_dump($decryptData);
```

上面将会输出(有乱码)：



更富弹性和便于存储的加密方案

上面看到，mcrypt下的加密在两点不足：

- 1、有乱码，不能很好地永久化存储；
- 2、只针对文本字符串的加密，不支持数组等，且无法还原类型；

为此，我们提供了更富弹性和便于存储的加密方案，即：序列化 + base64 + mcrypt的多重加密方案。

以下是上面的示例-多重加密版：

```
$mcrypt = new PhalApi\Crypt\MultiMcryptCrypt('12345678');

$data = 'The Best Day of My Life';
$key = 'phalapi';

$encryptData = $mcrypt->encrypt($data, $key);
var_dump($encryptData);

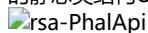
$decryptData = $mcrypt->decrypt($encryptData, $key);
var_dump($decryptData);
```

对应的输出（这里使用了文字结果输出，是因为没了乱码）：

```
string(44) "rmFdhvszAkHh0dzwt/APBACK/Mn/SqhV1Ahp1xTOGk="
string(23) "The Best Day of My Life"
```

RSA的支持与超长字符串的应对方案

基于项目有使用RSA进行加密和解密的需求，这里特扩展对RSA的支持。同时针对到RSA对字符串长度的限制，提供了分段处理的方案。RSA加密模块的静态类结构UML如下：



原生态的通信加密和解密

此部分只是简单地封装了openssl相关函数的操作，可以实现与其他语言和客户端下RSA的加密通信。

唯一需要注意的是，对于“私钥加密，公钥解密” 和 “公钥加密，私钥解密” 这两种情况下key的互换和对应问题。不要混淆。

超长字符串的分段处理

这里重点说明一下超长字符串通信加密的问题。

解决方案主要涉及两点：一是分段的处理，二是中间层转换。分段是指将待加密的字符串分割成允许最大长度117（有用户反馈说是127）内的数组，再各自处理；中间层转换是为了稳定性、通用性和方便落地存储，使用了json和base64的结合编码。

虽然此方案解决了超长字符串的问题，但需要特别指出的是，**不能与其他语言、或者PHP其他框架和客户端进行原生态的RSA通信**。

我们突破了长度的限制，但失去了通用性。这里罗列一下各个场景和对应的处理方式：

- 支持：PhalApi项目A <--> PhalApi项目A
- 支持：PhalApi项目A <--> PhalApi项目B, PhalApi项目C, PhalApi项目D, ...
- 不支持：PhalApi项目 <--> 非PhalApi项目的PHP项目
- 不支持：PhalApi项目 <--> 非PHP语言的项目。
解决方案：参考PhalApi对RSA超长字符串的处理，同步实现。
- 不支持：PhalApi项目 <--> 客户端（iOS/Android/Windows Phone, etc）。
解决方案：参考PhalApi对RSA超长字符串的处理，同步实现。

使用示例

以下是单元测试中的使用示例。

```

public function testDecryptAfterEncrypt()
{
    $keyG = new PhalApi\Crypt\RSA\KeyGenerator();
    $privkey = $keyG->getPriKey();
    $pubkey = $keyG->getPubKey();

    \PhalApi\DI()->crypt = new PhalApi\Crypt\RSA\MultiPri2PubCrypt();

    $data = 'AHA! I have $2.22 dollars!';

    $encryptData = \PhalApi\DI()->crypt->encrypt($data, $privkey);

    $decryptData = \PhalApi\DI()->crypt->decrypt($encryptData, $pubkey);

    $this->assertEquals($data, $decryptData);
}

```

建议

在上面的加密中，接口项目在开发时，需要自定义两个值：加密向量和私钥。

为了提高数据加密的安全度，建议：

- 加密向量项目统一在./Config/app.php中配置；
- 各模块业务数据加密所用的Key则由各业务点自定义；

这样，可以对不同的数据使用不同的加密私钥，即使破解了某一个用户的数据，也难以破解其他用户的。

扩展：实现你的加密方式

尤其对于加密方案和算法，我们在项目开发决策时，更应该优先考虑使用现在行业内成熟公认的加密方案和算法，而不是自己去从头研发。

但如果你项目确实有此需要，或者需要在mcrypt的基础上再作一些变通，也是可以很快地实现和注册使用。

首先，请先实现下面的加密接口：

```

<?php
namespace PhalApi;

interface Crypt {
    public function encrypt($data, $key);
    public function decrypt($data, $key);
}

```

然后，重新注册加密服务即可。

i18n国际化

一直以来，在项目开发中，都是以硬编码方式返回中文文案或者提示信息的，如：

```
$rs['msg'] = '用户不存在';
```

这种写法在根本不需要考虑国际化翻译的项目中是没问题的，但当开发的项目面向的是国际化用户人群时，使用i18n则是很有必要的。

语言设定

在初始化文件./public/init.php中，通过快速函数\PhalApi\SL(\$language)可以设定当前所使用的语言。例如设置语言为简体中文，可以：

```
// 翻译语言包设定
\PhalApi\SL('zh_cn');
```

设定的语言即为语言目录下对应语言的目录名称，例如可以是：de、en、zh_cn、zh_tw等。

```
$ tree ./language/
./Language/
├── de
└── en
...
└── zh_cn
    └── zh_tw
```

此处，也可以通过客户端传递参数动态选择语言。简单地：

```
\PhalApi\SL(isset($_GET['lan']) ? $_GET['lan'] : 'zh_cn');
```

翻译包

翻译包的文件路径为：./language/语言/common.php，例如简体中文zh_cn对应的翻译包文件为：./Language/zh_cn/common.php。此翻译包文件返回的是一个数组，其中键为待翻译的内容，值为翻译后的内容。例如：

```
return array(
    'Hi {name}, welcome to use PhalApi!' => '{name}您好，欢迎使用PhalApi!',
    'user not exists' => '用户不存在',
);
```

对于需要动态替换的参数，可以使用大括号括起来，如名字参数name对应为{name}。除了这种关联数组的方式，还可以使用索引数组的方式来传递动态参数。例如：

```
return array(
    ... ...
    'I love {0} because {1}' => '我爱{0}，因为{1}' ,
);
```

通用的翻译写法

当需要进行翻译时，可以使用快速函数[\PhalApi\T\(\\$msg, \\$params = array\(\)\)](#)，第一个参数为待翻译的内容，第二个参数为可选的动态参数。例如前面的文案调整成：

```
$rs['msg'] = \PhalApi\T('user not exists');
```

最后显示的内容将是对应翻译包里的翻译内容，如这里对应的是：

```
// 文件 ./language/zh_cn/common.php
return array(
    ... ...
    'user not exists' => '用户不存在',
);
```

当翻译中存在动态参数时，根据待翻译中参数的传递方式，可以相应提供对应的动态参数。例如对于关联数组方式，可以：

```
// 输出: dogstar您好，欢迎使用PhalApi!
echo \PhalApi\T('Hi {name}', array('name' => 'dogstar'));
```

关联数组方式中参数的对应关系由键名对应，而索引数组方式则要严格按参数出现的顺序对应传值，例如：

```
// 输出: 我爱PhalApi，因为它专注于接口开发
echo \PhalApi\T('I love {0} because {1}', array('PhalApi', '它专注于接口开发'));
```

若是翻译不存在时怎么办？翻译不存在，有两种情况：一种是指定的语言包不存在；另一种是语言包存在但翻译不存在。无论何种情况，当找不到翻译时，都会返回待翻译时的内容。

扩展：添加翻译包

默认的翻译包存放在项目根目录的language目录下。当需要添加其他路径的翻译包时，例如在进行扩展类库开发时。

对于也拥有翻译包的扩展类库，其翻译包文件可以放在扩展类库本身目录的language子目录中，其结构一样。但由于不在项目根目录下，这时需要手动引入翻译包目录，以便框架可以加载识别。当需要加载其他路径的翻译包时，可以使用[\PhalApi\Translator::addMessage\(\\$path\)](#)进行添加，后面添加的翻译包会覆盖前面的翻译包。例如User扩展类库中的：

```
PhalApi\Translator::addMessage('/path/to/user/language');
```

这样，就可以添加/path/to/user/language目录下的翻译包了。# CURL请求

当需要进行curl请求时，可使用PhalApi封装的CURL请求类[\PhalApi\CUrL](#)，从而实现快捷方便的请求。

发起GET请求

例如，需要请求的链接为：<http://demo2.phalapi.net/>，则可以：

```
// 先实例
$curl = new \PhalApi\CUrL();

// 第二个参数，表示超时时间，单位为毫秒
$rs = $curl->get('http://demo2.phalapi.net/?username=dogstar', 3000);

echo $rs;
// 输出类似如下:
// {"ret":200,"data":{"title":"Hello dogstar","version":"2.1.2","time":1513506356},"msg":""}
```

发起POST请求

当需要发起POST请求时，和GET方式类似，但需要把待POST的参数单独传递，而不是拼接在URL后面。如：

```
try {
    // 实例化时也可指定失败重试次数，这里是2次，即最多会进行3次请求
    $curl = new \PhalApi\CUrL(2);

    // 第二个参数为待POST的数据；第三个参数表示超时时间，单位为毫秒
    $rs = $curl->post('http://demo2.phalapi.net/?', array('username' => 'dogstar'), 3000);

    // 一样的输出
    echo $rs;
} catch (\PhalApi\Exception\InternalServerErrorException $ex) {
    // 错误处理.....
}
``` # DI服务汇总

DI服务初始化
全部依赖注入的资源服务，都位于`./config/di.php`文件内。
```

```
基本注册
```

默认情况下，会进行基本注册如下：

```

$di = \PhalApi\DI();

// 配置 $di->config = new FileConfig(API_ROOT . '/config');

// 调试模式, $_GET['debug']可自行改名 $di->debug = !empty($_GET['debug']) ? true : $di->config->get('sys.debug');

// 日记纪录 $di->logger = new FileLogger(API_ROOT . '/runtime', Logger::LOG_LEVEL_DEBUG | Logger::LOG_LEVEL_INFO | Logger::LOG_LEVEL_ERROR);

// 数据操作 - 基于NotORM $di->notorm = new NotORMDatabase($di->config->get('dbs'), $di->debug);

定制注册

可以根据项目的需要, 进行定制化的注册, 只需要把下面的注释去掉即可。

// 签名验证服务 // $di->filter = new \PhalApi\Filter\SimpleMD5Filter();

// 缓存 - Memcache/Memcached // $di->cache = function () { // return new \PhalApi\Cache\MemcacheCache(DI()->config->get('sys.mc')); // };

// 支持JsonP的返回 // if (!empty($_GET['callback'])) { // $di->response = new \PhalApi\Response\JsonpResponse($_GET['callback']); // }

如果需要更多的DI服务, 也可以参考并使用下面的DI服务资源一览表。

```

## DI服务资源一览表

假设, 我们已有:

```
$di = \PhalApi\DI();
```

则:

服务名称|是否启动时自动注册|是否必须|接口/类|作用说明

服务名称	是否启动时自动注册	是否必须	接口/类	作用说明
\$di->config	否	是	[PhalApi\Config](https://github.com/phalapi/kernal/blob/master/src/Config.php)	配置: 负责项目配置的读取, 需要手动注册, 指定存储媒介, 默认是[PhalApi\Config]
\$di->logger	否	是	[PhalApi\Logger](https://github.com/phalapi/kernal/blob/master/src/Logger.php)	日记纪录: 负责日记的写入, 需要手动注册, 指定日记级别和存储媒介, 默认是[PhalApi\Logger]
\$di->request	是	是	[PhalApi\Request](https://github.com/phalapi/kernal/blob/master/src/Request.t.php)	接口参数请求: 用于收集接口请求的参数
\$di->response	是	是	[PhalApi\Response](https://github.com/phalapi/kernal/blob/master/src/Response.php)	结果响应: 用于输出返回给客户端的结果, 默认为[PhalApi\Response]
\$di->notorm	是	是	[Database\NotORMDatabase](https://github.com/phalapi/kernal/blob/master/src/Database/NotORMDatabase.php)	数据操作: 基于NotORM的DB操作, 是[Database\NotORMDatabase]
\$di->cache	否	推荐	[PhalApi\Cache](https://github.com/phalapi/kernal/blob/master/src/Cache.php)	缓存: 实现缓存读写, 需要手动注册, 指定缓存
\$di->filter	否	推荐	[PhalApi\Filter](https://github.com/phalapi/kernal/blob/master/src/Filter.php)	拦截器: 实现签名验证、权限控制等操作
\$di->crypt	否	否	[PhalApi\Crypt](https://github.com/phalapi/kernal/blob/master/src/Crypt.php)	对称加密: 实现对称加密和解密, 需要手动注册
\$di->curl	否	否	[PhalApi\Curl](https://github.com/phalapi/kernal/blob/master/src/Curl.php)	CURL请求类: 通过curl实现的快捷方便的接口请求类, 需要手动注册
\$di->cookie	否	否	[PhalApi\Cookie](https://github.com/phalapi/kernal/blob/master/src/Cookie.php)	COOKIE的操作
\$di->tracer	是	是	[PhalApi\Helper\Tracer](https://github.com/phalapi/kernal/blob/master/src/Helper/Tracer.php)	内置的全局追踪器, 支持自定义节点标识
\$di->debug	否	否	boolean	应用级的调试开关, 通常可从配置读取, 为true时开启调试模式

## DI服务是否已注册的判断误区

### (1) 错误的判断方法

当需要判断一个DI服务是否已被注册, 出于常识会这样判断:

```
php
if (isset(\PhalApi\DI()->cache)) {
```

但这样的判断永远为false, 不管注册与否。

追其原因在于, DI类使用了魔法方法的方式来提供类成员属性, 并存放于PhalApi\DependencyInjection::\$data中。

这就导致了如果直接使用isset(\PhalApi\DI()->cache)的话, 首先不会触发魔法方法 PhalApi\DependencyInjection::\_\_get(\$name) 的调用, 其次也确实没有 PhalApi\DependencyInjection::\$cache 这个成员属性, 最终判断是否存在时都为false。

简单来说, 以下两种判断, 永远都为false:

```
$di = \PhalApi\DI();

// 永远为false
var_dump(isset($di->XXX));
var_dump(!empty($di->XXX));
```

## (2)正确判断的写法: 先获取, 再判断

正确的用法应该是:

```
// 先获取, 再判断
$XXX = $di->XXX;
var_dump(isset($XXX));
var_dump(!empty($XXX));
```

# PhalApi框架扩展类库

## 扩展类库简介

### 致力于与开源项目一起提供企业级的解决方案!

此部分类库为PhalApi框架下可重用的扩展类库, 各个扩展相自独立, 可以根据需要自动安装使用。

此扩展类库可以是基于已有的第三方开源类库的二次开发和集成，也可以是自主研发的组件、工具、模块。通过使用可重用的扩展类库，可大大减少开发成本，并且慢慢地会发现，原来编程本来就是一件如此简单的事情，就像搭积木一样。

正如我们一直提倡的：**接口，从简单开始！**

## 扩展类库列表

扩展类库	composer 名称	扩展类库名称	简要说明
<a href="#">phalapi/apk</a>		APK文件解包处理	对APK进行解包，支持绝大部分APK文件处理。
<a href="#">phalapi/auth</a>		Auth权限扩展	实现了基于用户与组的权限认证功能，与RBAC权限认证类似，主要用于对服务级别的功能进行权限控制。 by twodayw
<a href="#">phalapi/cli</a>		CLI扩展类库	可用于开发命令行应用，基于GetOpt，主要作用是将命令参数进行解析和处理。
Cluster		基于PhalApi的DB集群拓展	为了解决大量数据写入分析的问题，支持大量select、和大量insert。
CryptTraffic		移动设备通信加密	用于移动设备通信加密。
Excel		PhalApi-Excel	读取Excel。
Facepp		face++接口	face++接口。
<a href="#">phalapi/fast-route</a>		FastRoute快速路由	基于FastRoute实现，通过配置实现自定义路由配置，从而轻松映射service接口服务。
<a href="#">phalapi-image</a>		PhalApi-Image图像处理	按照尺寸压缩上传图片，参考自ThinkPHP图形处理。 by 春春小猴
KafKa		简单舒适的PHP-Kafka拓展	基于rdKafka封装的一个简单舒适Kafka拓展。
Log4php		基于log4php的日志扩展	兼容PhalApi日志的接口操作，同时基于log4php完成更多出色的日志工作。
Medoo		Medoo数据库驱动	Medoo数据库驱动。
<a href="#">phalapi-aliyun-oss</a>		PhalApi-OSS阿里云OSS包	对阿里云的OSS文件服务器的封装。 by vivlong
PHPExcel		PhalApi-PHPExcel扩展	提供了更为强大的Excel处理功能。
<a href="#">phalapi/PHPMailer</a>		基于PHPMailer的邮件发送	用于发送邮件。
<a href="#">phalapi-phprpc</a>		代理模式下phprpc协议的轻松支持	可用于phprpc协议的调用，服务端只需要简单添加入口即可完美切换。
<a href="#">phalapi-pay</a>		基于PhalApi的第三方支付扩展	支持微信支付和支付宝支付。
Payment		微信支付及支付宝支付扩展	支持微信支付和支付宝支付。
<a href="#">phalapi/qiniu</a>		七牛云存储接口调用	可用于将图片上传到七牛云存储，或者七牛SDK包提供的其他功能。
RabbitMQ		PhalApi-RabbitMQ队列拓展	基于队列标杆中的RabbitMQ的队列扩展。
<a href="#">phalapi/redis</a>		基于PhalApi的Redis拓展	提供更丰富的Redis操作，并且进行了分库处理可以自由搭配。 by 喵了个咪
<a href="#">phalapi-sms</a>		PhalApi-SMS容联云短信服务器扩展	基于容联云通讯，发送短信。
<a href="#">ctbsea/phalapi-smarty</a>		基于PhalApi的Smarty扩展	基于老牌的PHP模版引擎Smarty，提供视图渲染功能。
<a href="#">chenall/phalapi-soap</a>		SOAP扩展	使用PHP官方提供的SOAP协议，用于搭建Web Services。 by chenall
Swoole		Swoole扩展	基于swoole，支持的长链接和异步任务实现。
<a href="#">phalapi/task</a>		计划任务扩展	用于后台计划任务的调度。
ThirdLogin		第三方登录扩展	第三方登录。
Translate		PhalApi-Translate百度翻译扩展	基于百度翻译的翻译。
UCloud		图片上传扩展	用于图片文件上传。
User		User用户扩展	提供用户、会话和集成第三方登录。
<a href="#">steveak/view</a>		View视图扩展	提供视图渲染功能。 by steve
Wechat		微信开发扩展	可用于微信的服务号、订阅号、设备号等功能开发。
Xhprof		性能分析工具PhalApi-Xhprof	对Facebook开源的轻量级PHP性能分析工具进行了封装拓展。
YoukuClient		优酷开放平台接口扩展	用于调用优酷开放平台的接口。
Zip		PhalApi-Zip压缩文件处理	用于处理文件压缩。
<a href="#">phalapi/qrcode</a>		PhalApi 二维码扩展	二维码扩展，基于PHP QRCode实现。可用于生成二维码图片。
<a href="#">phalapi/pinyin</a>		PhalApi 2.x 拼音扩展	PhalApi 2.x 拼音扩展，基于overtrue/pinyin实现。
<a href="#">phalapi-gtcode</a>		极验验证码扩展	极验验证码扩展， by 春春小猴
<a href="#">phalapi/jwt</a>		基于PhalApi2的JWT拓展	JSON Web Token (JWT) 是一个非常轻巧的规范。这个规范允许我们使用JWT在用户和服务器之间传递安全可靠的信息。 by twodayw
<a href="#">chenall/phalapi-weixin</a>		微信扩展	微信公众号、企业号等开发扩展，使用Eastwechat。 by chenall
<a href="#">phalapi/wechatmini</a>	微信小程序扩展	PhalApi 2.x 微信小程序扩展	by JamesLiuquan
<a href="#">chenall/phalapi</a>	请求参数规则扩展	by chenall	
<a href="#">phalapi/cors</a>	CORS跨域扩展	by gongshunkai (春春小猴)	
<a href="#">phalapi/session</a>	Session 操作工具	phalapi session 会话封装	by Zhangzijing

扩展类库composer 名称	扩展类库名称	简要说明
<a href="#">Phalapi- Workerman</a>	Phalapi-Workerman扩展 为PhalApi封装的workerman库，暂不支持composer。 by logmein	

温馨提示：未有composer链接的，表示尚未从1.x迁移到2.x版本，可在原来的[Phalapi-Library扩展类库](#)项目中查阅。

## 扩展类库的使用

对于某个扩展类库，当需要使用时，可以按“安装、配置、使用”三步曲进行。

### 安装

扩展类库的安装很简单，在PhalApi 2.x版本下，直接通过在composer.json文件中配置需要依赖的扩展类库即可。

例如，项目本身自带的Task扩展类库：

```
{
 "require": {
 "phalapi/task": "2.0.*"
 }
}
```

配置好后，执行`composer update`更新操作即可。

至此，便完成了扩展类库的安装，相当简单。

### 配置注册

根据不同的扩展类库，其配置和注册的情况不同，有些不需要配置也不需要注册，有些需要配置、注册中的一种，有些可能配置、注册都需要。

#### 何为配置？

这里说的配置是指在项目配置文件./config/app.php内添加对应扩展类库的配置，配置选项的路径通常为：`app.扩展类库名称`。此外，有的扩展类库可能还需要配置数据库配置文件./config/dbs.php。

#### 何为注册？

而注册则是指将对应的扩展类库注册到DI容器\PhalApi\DI()中，需要在./config/di.php文件中配置。注册的服务名称通常为扩展类库的小写名称。

注册好后，便可以在项目需要的位置进行调用了。

### 使用

不同的扩展类库，其提供的功能不同，所以具体的使用也不尽相同。当使用到某个扩展类库时，可以参考对应的文档说明。有的扩展可能需要调用其内部接口才能实现对应的功能，有些扩展可能提供了直接可用的接口服务。

## 扩展类库开发指南

为了统一扩展类库的风格、便于用户更容易使用，这里建议：

- 代码：遵循composer和psr-4的风格，并尽量Lite.php为入口类，一个扩展，一个Git项目，源代码可放置在自己的Git仓库；
- composer：建议统一注册在[phalapi](#)下，可联系dogstar；
- 配置：统一放置在\PhalApi\DI()->config->get('app.扩展包名')中，避免配置冲突；
- 文档：统一提供README.md文件，对扩展类库的功能、安装和配置、使用示例以及运行效果进行说明；

### 在composer下开发扩展类库的建议

- 约定上依赖phalapi/kernal，但扩展类库内不要配置依赖，以免框架升级导致扩展不兼容
- 为简化起见，扩展类库默认统一使用master分支作为安装版本，免去多版本管理

### 从微架构到扩展类库的演进

在应用项目的实际开发，我们也可以有意识地将一些通用的工具和操作与业务分离，以便可以在项目内更好地重用。当抽离成工具或者通用类后，则可以进一步推广到公司内其他项目，即组件复用。如果觉得可以，则发扬开源精神，分享给社区。这也是符合从微架构到应用构架、系统架构、乃至企业架构的演进之路。

## . PhalApi 2.x 的SDK包

### SDK包列表

已经支持的SDK有：

- [Java版SDK](#)
- [Object-C版SDK](#)

- [Object-C版SDK, 遵循AFNetworking](#)
- [Javascript版SDK](#)
- [Golang版SDK](#)
- [PHP版SDK](#)
- [Python版SDK](#)
- [React-Native版SDK](#)
- [Ruby版SDK](#)

## 一句话描述

为了给客户端统一接口请求调用的规范性、流畅性和简单易懂，我们特别为此使用了内部领域特定语言：**接口查询语言**（Api Structured Query Language）。

## 外部DSL

从外部DSL的角度来看待接口查询的操作

```
create
withHost host
withFilter filter
withParser parser

reset #特别注意：重复查询时须重置请求状态

withService service
withParams paramName1 paramValue1
withParams paramName2 paramValue2
withParams ...
withTimeout timeout

request
```

根据此设计理念，各客户端语言都可以实现此接口请求的操作。

## 接口查询语言设计理念与示例

### 文法: create -> with -> request

所用到的查询文法如下（通常从上往下依次操作，顺序不强制）：

操作	参数	是否必须	是否可重复调用	作用说明
create	无	必须	可以，重复调用时新建一个实例，非单例模式	需要先调用此操作创建一个接口实例
withHost	接口域名	必须	可以，重复时会覆盖	设置接口域名，如： <a href="http://demo.phalapi.net/">http://demo.phalapi.net/</a>
withFilter	过滤器	可选	可以，重复时会覆盖	设置过滤器，与服务器的PhalApi\DI()>filter对应，需要实现PhalApiClientFilter接口
withParser	解析器	可选	可以，重复时会覆盖	设置结果解析器，仅当不是JSON返回格式时才需要设置，需要实现PhalApiClientParser接口
reset	无	通常必须	可以	重复查询时须重置请求状态，包括接口服务名称、接口参数和超时时间
withService	接口服务名称	通常必选	可以，重复时会覆盖	设置将在调用的接口服务名称，如：Site.Index
withParams	接口参数名、值	可选	可以，累加参数	设置接口参数，此方法是唯一一个可以多次调用并累加参数的操作
withTimeout	超时时间	可选	可以，重复时会覆盖	设置超时时间，单位毫秒，默认3秒
request	无	必选	可以，重复发起接口请求	最后执行此操作，发起接口请求

## JAVA示例

以JAVA版本为例，演示如何调用：

最简单的调用，也就是默认接口的调用：

```
PhalApiClientResponse response = PhalApiClient.create()
 .withHost("http://demo.phalapi.net/") //接口域名
 .request(); //发起请求
```

通常的调用，即有设置接口服务名称、接口参数和超时：

```
PhalApiClientResponse response = PhalApiClient.create()
 .withHost("http://demo.phalapi.net/")
 .withService("Site.Index") //接口服务
 .withParams("username", "dogstar") //接口参数
 .withTimeout(3000) //接口超时
 .request();
```

高级复杂调用，即设置了过滤器、解析器的操作：

```
PhalApiClientResponse response = PhalApiClient.create()
 .withHost("http://demo.phalapi.net/")
```

```
.withService("Site.Index")
.withParser(new PhalApiClientParserJson()) //设置JSON解析， 默认已经是此解析，这里仅作演示
.withParams("username", "dogstar")
.withTimeout(3000)
.request();
```

## 更好的建议

### 不支持面向对象的实现方式

此接口查询的用法是属于基础的用法，其实现与宿主语言有强依赖关系，在不支持面向对象语言中，如javascript，可以使用函数序列的方式，如：

```
create();
withHost('http://demo.phalapi.net/');
withService('Site.Index');
withParams('username', 'dogstar');
withTimeout(3000);
var rs = request();
```

### 封装自己的接口实例

通常，在一个项目里面我们只需要一个接口实例即可，但此语言没默认使用单例模式，是为了大家更好的自由度。基于此，大家在项目开发时，可以再进行封装：提供一个全局的接口查询单例，并组装基本的接口公共查询属性。

即分两步：初始化接口实例，以及接口具体的查询操作。

如第一步先初始化：

```
PhalApiClient client = PhalApiClient.create()
 .withHost("http://demo.phalapi.net/")
 .withParser(new PhalApiClientParserJson());
```

第二步进行具体的接口请求：

```
PhalApiClientResponse response = client.reset() #重复查询时须重置
 .withService("Site.Index")
 .withParams("username", "dogstar")
 .withTimeout(3000)
 .request();
```

这样，在其他业务场景下就不需要再重复设置这些共同的属性（如过滤器、解析器）或者共同的接口参数。

### 超时重试

当接口请求超时时，统一返回 ret = 408，表示接口请求超时。此时可进行接口重试。

如：

```
PhalApiClient client = PhalApiClient.create()
 .withHost("http://demo.phalapi.net/")

PhalApiClientResponse response = client.request();

if (response.getRet() == 408) {
 response = client.request(); //请求重试
}
```

---

## 脚本命令的使用

自动化是提升开发效率的一个有效途径。PhalApi致力于简单的接口服务开发，同时也致力于通过自动化提升项目的开发速度。为此，生成单元测试骨架代码、生成数据库建表SQL这些脚本命令。应用这些脚本命令，能快速完成重复但消耗时间的工作。下面将分别进行说明。

### phalapi-buildtest命令

当需要对某个类进行单元测试时，可使用phalapi-buildtest命令生成对应的单元测试骨架代码，其使用说明如下：

```

dogstar@ubuntu:PhalApi$./bin/phalapi-buildtest

Usage:
 php ./bin/phalapi-buildtest <file_path> <class_name> [bootstrap] [author]

Options:
 file_path Require. Path to the PHP source code file
 class_name Require. The class name need to be tested
 bootstrap NOT require. Path to the bootstrap file, usually is test_env.php
 author NOT require. Your great name here, default is dogstar

Demo:
 ./bin/phalapi-buildtest ./Demo.php Demo > Demo_Test.php
 ./bin/phalapi-buildtest ./Demo.php Demo > Demo_Test.php
 ./bin/phalapi-buildtest ./src/Request.php PhalApi\\Request > Request_Test.php

Tips:
 This will output the code directly, you can save them to test file like with _Test.php suffix.

```

其中,

- 第一个参数`file_path` 是待测试的源文件相对/绝对路径。
- 第二个参数`class_name` 是待测试的类名。
- 第三个参数`bootstrap` 是测试启动文件, 通常是`/path/to/phalapi/tests/bootstrap.php`文件。
- 第四个参数`author` 你的名字, 默认是`dogstar`。

通常, 可以先写好类名以及相应的接口, 然后再使用此脚本生成单元测试骨架代码。以默认接口服务`Site_Index`为例, 当需要为其生成单元测试骨架代码时, 可以执行以下命令。

```
$./bin/phalapi-buildtest ./src/app/Api/Site.php App\\Api\\Site > ./tests/app/Api/Site_Test.php
```

最后, 需要将生成好的骨架代码, 重定向保存到你要保存的位置。通常与产品代码对齐, 并以 “{类名} + `_Test.php`” 方式命名, 如这里的`app/Api/Site_Test.php`。

生成的骨架代码类似如下:

```

<?php

//require_once dirname(__FILE__) . '/bootstrap.php';

if (!class_exists('App\\Api\\Site')) {
 require dirname(__FILE__) . './src/app/Api/Site.php';
}

/**
 * PhpUnderControl_App\Ap\Site_Test
 *
 * 针对 ./src/app/Api/Site.php App\Ap\Site 类的PHPUnit单元测试
 *
 * @author: dogstar 20170725
 */

class PhpUnderControl_AppApiSite_Test extends \PHPUnit_Framework_TestCase
{
 public $appApiSite;

 protected function setUp()
 {
 parent::setUp();
 $this->appApiSite = new App\Ap\Site();
 }
}

```

简单修改后, 便可运行。

## phalapi-buildsqls命令

当需要创建数据库表时, 可以使用`phalapi-buildsqls`脚本命令, 再结合数据库配置文件`./config/dbs.php`即可生成建表SQL语句。此命令在创建分表时尤其有用, 其使用如下:

```

dogstar@ubuntu:PhalApi$./bin/phalapi-buildsqls

Usage:
 ./bin/phalapi-buildsqls <dbs_config> <table> [engine] [sqls_folder]

Options:
 dbs_config Require. Path to ./Config/dbs.php
 table Require. Table name
 engine NOT require. Database engine, default is Innodb
 sqls_folder NOT require. Data folder, default is API_ROOT/data

Demo:
 ./bin/phalapi-buildsqls ../Config/dbs.php User

Tips:
 This will output the sql directly, enjoy yourself!

```

其中，

- 第一个参数**dbs\_config** 是指向数据库配置文件的路径，如./Config/dbs.php，可以使用相对路径。
- 第二个参数**table** 是需要创建SQL的表名，每次生成只支持一个。
- 第三个参数**engine** 可选参数，是指数据库表的引擎，MySQL可以是：Innodb或者MyISAM。
- 第四个参数**sqli\_folder** 可选参数，SQL文件的目录路径。

在执行此命令先，需要提前先将建表的SQL语句，排除除主键id和ext\_data字段，放置到./data目录下，文件名为：{表名}.sql。

例如，我们需要生成10张user\_session用户会话表的建表语句，那么需要先添加数据文件./data/user\_session.sql，并将除主键id和ext\_data字段外的其他建表语句保存到该文件。

```
'user_id` bigint(20) DEFAULT '0' COMMENT '用户id',
'token` varchar(64) DEFAULT '' COMMENT '登录token',
'client` varchar(32) DEFAULT '' COMMENT '客户端来源',
'times` int(6) DEFAULT '0' COMMENT '登录次数',
'login_time` int(11) DEFAULT '0' COMMENT '登录时间',
'expires_time` int(11) DEFAULT '0' COMMENT '过期时间',
```

然后，进入到项目根目录，执行命令：

```
$ php ./bin/phalapi-buildsqls ./config/dbs.php user_session
```

正常情况下，会看到生成好的SQL语句，类似下面这样的输出。

```
CREATE TABLE `phalapi_user_session` (
 `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
 `user_id` bigint(20) DEFAULT '0' COMMENT '用户id',
 `token` varchar(64) DEFAULT '' COMMENT '登录token',
 `client` varchar(32) DEFAULT '' COMMENT '客户端来源',
 `times` int(6) DEFAULT '0' COMMENT '登录次数',
 `login_time` int(11) DEFAULT '0' COMMENT '登录时间',
 `expires_time` int(11) DEFAULT '0' COMMENT '过期时间',
 `ext_data` text COMMENT 'json data here',
 PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `phalapi_user_session_1` (
 `id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
 ...
 `ext_data` text COMMENT 'json data here',
 PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

CREATE TABLE `phalapi_user_session_2`
CREATE TABLE `phalapi_user_session_3`
CREATE TABLE `phalapi_user_session_4`
CREATE TABLE `phalapi_user_session_5`
CREATE TABLE `phalapi_user_session_6`
CREATE TABLE `phalapi_user_session_7`
CREATE TABLE `phalapi_user_session_8`
CREATE TABLE `phalapi_user_session_9`
```

最后，便可把生成好的SQL语句，导入到数据库，完成建表的操作。

值得注意的是，生成的SQL建表语句默认会带有自增ID主键id和扩展字段ext\_data这两个字段。所以保存在./data目录下的建表语句可省略主键字段，以免重复。

```
'id` bigint(20) unsigned NOT NULL AUTO_INCREMENT,
...
`ext_data` text COMMENT 'json data here',
```

## phalapi-cli命令

此脚本可用于在命令行终端，直接运行接口服务，也可用于作为命令行终端应用的执行入口。

需要注意的是，要先确保在composer.json文件内有以下配置：

```
{
 "require": {
 "phalapi/cli": "dev-master"
 }
}
```

并确保已经成功安装phalapi/cli。

phalapi/cli扩展地址：<https://github.com/phalapi/cli>

以默认接口服务App.Site.Index为例，执行方式如下：

```
$./bin/phalapi-cli -s App.Site.Index --username dogstar
{
 "ret": 200,
 "data": {
 "title": "Hello dogstar",
 "version": "2.2.3",
 "time": 1535207991,
 "msg": ""
 }
}
```

如果想查看帮助提示信息，可以在指定了接口服务后，使用--help参数。例如：

```
$./bin/phalapi-cli -s App.Site.Index -h
Usage: ./bin/phalapi-cli [options] [operands]
Options:
 -s, --service <arg> 接口服务
 -h, --help 查看帮助
```

## 注意事项

在使用这些脚本命令前，需要注意以下几点。

## 执行权限

第一点是执行权限，当未设置执行权限时，脚本命令会提示无执行权限，类似这样。

```
$./phalapi/bin/phalapi-buildtest
-bash: ./phalapi/bin/phalapi-buildtest: Permission denied
```

那么需要这样设置脚本命令的执行权限。

```
$ chmod +x ./phalapi/bin/phalapi-build*
```

## 编码问题

其次，对于Linux平台，可能会存在编码问题，例如提示：

```
$./phalapi/bin/phalapi-buildtest
bash: ./phalapi/bin/phalapi-buildtest: /bin/bash^M: bad interpreter: No such file or directory
```

这时，可使用dos2unix命令转换一下编码。

```
$ dos2unix ./phalapi/bin/phalapi-buildtest*
dos2unix: converting file ./phalapi/bin/phalapi-buildsqls to Unix format ...
dos2unix: converting file ./phalapi/bin/phalapi-buildtest to Unix format ...
```

## 软链

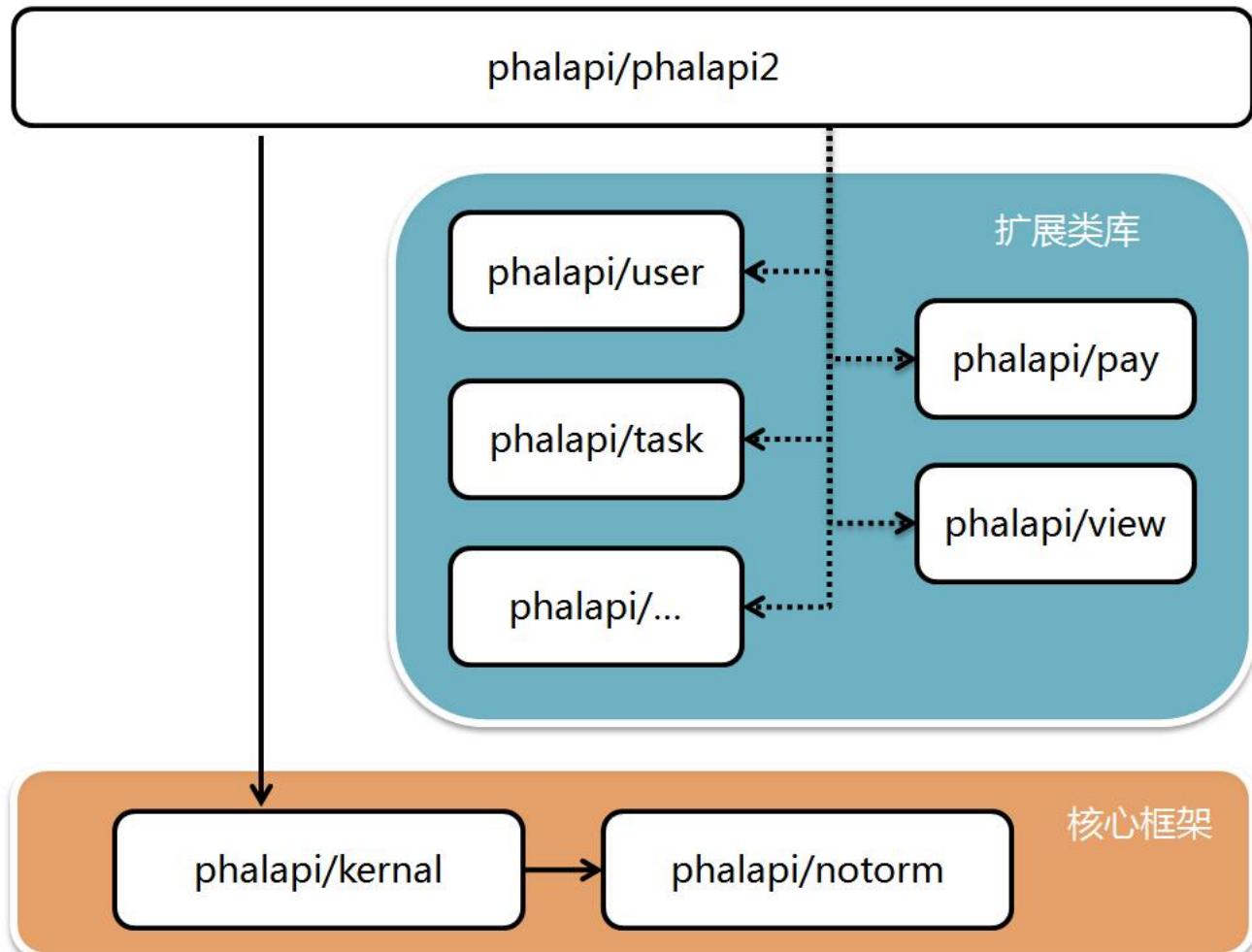
最后一点是，在任意目录位置都是可以使用这些命令的，但会与所在的项目目录绑定。通常，为了更方便使用这些命令，可以将这些命令软链到系统命令下。例如：

```
$ sudo ln -s /path/to/phalapi/bin/phalapi-buildsqls /usr/bin/phalapi-buildsqls
$ sudo ln -s /path/to/phalapi/bin/phalapi-buildtest /usr/bin/phalapi-buildtest
```

# PhalApi 2.x 版本完美诠释

## 2.x 版本系统架构

PhalApi 2.x 版本的系统架构如下：



主要分为三层：

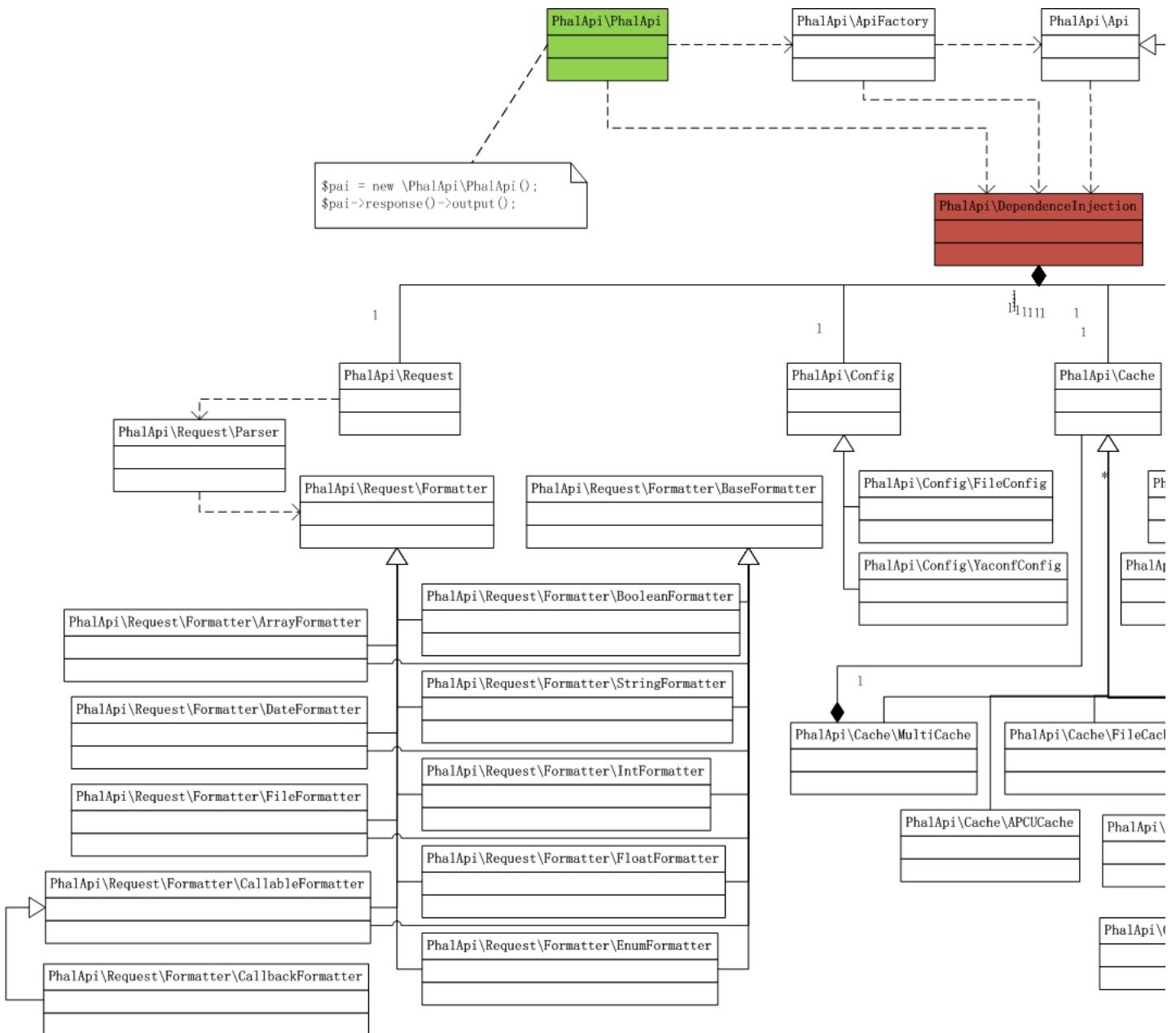
- **phalapi/phalapi** 项目应用层，可使用phalapi/phalapi搭建微服务、接口系统、RESTful、WebServices等。
- **扩展类库** 扩展类库是指可选的、可重用的组件或类库，可以直接集成使用，由广大开发人员维护分享，对应原来的PhalApi-Library项目。
- **核心框架** 分别两大部分，PhalApi核心部分kernal，以及优化后的notorm。

其中，各自的composer和github项目分别是：

项目	composer	github
phalapi/phalapi	<a href="#">phalapi/phalapi</a>	<a href="#">phalapi/phalapi</a>
扩展类库	由广大开发人员共同维护、分享，composer建议统一注册到 <a href="#">phalapi</a> 。	由广大开发人员共同维护、分享，源代码可维护在开发者各自的 Github仓库。
核心框架	<a href="#">phalapi/kernal</a>	<a href="#">phalapi/kernal</a>

## 框架核心部分UML静态结构图

PhalApi 2.x 版本的核心框架部分的UML静态结构图，高清版如下所示：



首先，绿色部分的PhalApi\PhalApi类是整个接口系统的访问入口，也就是项目应用系统、客户端使用的关键所在。相关的调用代码，可以参考统一入口文件的实现代码片段。

```
$pai = new \PhalApi\PhalApi();
$pai->response()->output();
```

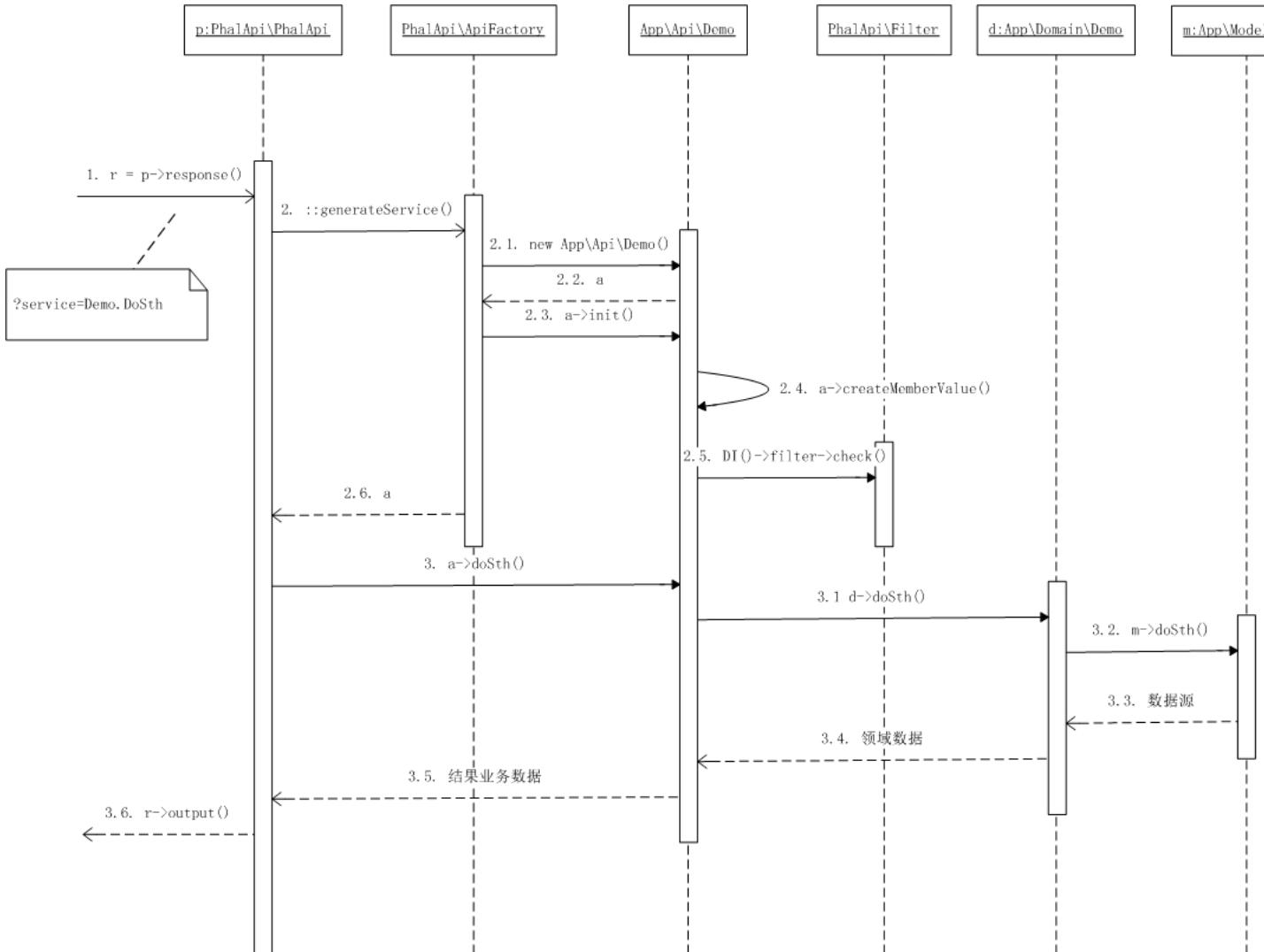
只需要两行代码，便可完成对接口服务的请求响应。

其次，是黄色部分的Api、Domain和Model这三层，也就是我们常说的ADM分层架构。这部分，需要开发人员关注，因为这也是具体项目开发需要自己实现的部分。

最后，是红色部分的DI依赖注入，也是整个框架的核心所在。不仅在核心框架中使用频率最高，乃至在项目应用中也会经常被用到。

## 核心执行流程时序图

PhalApi 2.x 版本的核心执行流程时序图，与1.x 版本基本一致，可以看出，不管技术如何升级，PhalApi的最初的核心时序流程仍保持着活力。唯一变化的是各个类名。



## PhalApi 2.x 升级指南

PhalApi自2015年初正式开源以来，版本主要经历了 v1.0.0 ~ v1.4.1，其间，我们一直在致力于“接口，从简单开始！”的同时，也致力框架的兼容性升级。但考虑到2.x版本是全新的实现机制和方式，经过综合和慎重考虑，2.x版本将不再支持向前兼容，即1.x版本不能完美升级到2.x版本。对此带来的不便，我深表歉意。

为方便1.x版本的开发者更容易从概念上切换到2.x版本，以下是快速升级指南。

### 从石器农耕到现代自动化

自2017年x月x日发布 v2.0.0 版本开始，PhalApi社区主要存在两大系列版本，分别是：

- **PhalApi v1.x 版本 slogan：接口，从简单开始！**
- **PhalApi v2.x 版本 slogan：助你创造价值！**

这两系列版本，最大的差异在于，PhalApi v2.x 版本是：

- 使用了composer
- 引入了命名空间
- 遵循psr-4规范

相比之下，PhalApi v1.x 版本则是：

- 未使用composer，自主构建自动加载器PhalApi\_Loader
- 完全不使用命名空间
- 遵循psr-0规范以及PEAR命名规范

使用composer的方式，更有利于可重用组件的管理，并且向国际化、向主流靠拢，同时也要求我们需要对PhalApi进行重新划分与调整。而引入命名空间，则更是要求核心框架代码、扩展类库乃至应用项目开发，都需要重新调整。

这是历史发展的必然趋势，因为composer和命名空间将会被越来越多的开发人员所熟悉。如果在1.x版本，我们是石器农耕时代，那么2.x版本，我们迎来的将是现代自动化时代。

温馨提示：关于composer，更多可访问：[Composer 中文网 / Packagist 中国全量镜像](#)。

## composer的使用方式

在1.x版本中，PhalApi更多是以自主研发的自动加载方式为主，这种方式原始、简单，并且粗糙。在2.x版本中，PhalApi使用的是composer管理方式，这要求对已有的代码仓库进行重新划分，并分为：项目、扩展类库和核心框架这三部分。

- 项目：可用于实际项目开发，提供给开发人员使用。
- 扩展类库：由广大开发人员共同维护、分享。
- 核心框架：由PhalApi团队核心人员长期维护，并接受贡献者的merge request。

### 如何创建新项目？

如果需要使用PhalApi 2.x 版本开发新项目，可以直接使用[phalapi/phalapi](#)。

在composer方式下，当需要自动加载类和函数时，可使用autoload配置，参考：

```
{
 "autoload": {
 "files": [
 "src/app/functions.php"
],
 "psr-4": {
 "App\\": "src/app"
 }
 }
}
```

### 如何使用扩展类库？

当需要使用扩展类库时，你只需要简单的告诉 Composer 需要依赖哪些包。在composer.json中添加相应的包名称和版本信息，然后更新即可。以添加Task计划任务扩展为例，首先修改composer.json并添加：

```
{
 "require": {
 "phalapi/task": "1.0.*"
 }
}
```

然后，进行composer更新：

```
$ composer update
```

如果扩展类库还提供了直接可用的接口服务，这里则还需要配置psr-4，以便让composer能正常自动加载对应的类文件，并且显示在在线接口文档。

### 如何升级框架？

借助composer，可轻松更新框架，同时也可指定需要依赖的具体版本。例如，若希望从2.0.0升级到2.0.1版本，可以：

```
{
 "require": {
 "phalapi/kernel": "2.0.1"
 }
}
```

然后，进行composer更新即可。

## 命名空间下的差异

由于PhalApi 1.x 版本不使用命名空间，而PhalApi 2.x 版本使用了命名空间，因此会有所差异。主要有：

### 接口服务请求时的差异

1.x 版本的默认接口服务是Default\_Index，对应文件./Demo/Api/Default.php；2.x 版本的默认接口服务是App\_Site\_Index，对应文件./src/app/Api/Site.php。

1.x 版本的接口服务的请求格式是：Class.Action；2.x 版本的接口服务的请求格式是：Namespace.Class.Action，比旧版本多了命名空间前缀，并且为了兼容原来的格式，缺省的命名空间是App。当命名空间和类名空间存在多组时，将会使用下划线分割。以下是2.x版本的一些示例：

2.x 请求的service参数	对应的文件	执行的类方法
无	/src/app/Api/Site.php	App/Api/Site::Index()
?s=Site.Index	/src/app/Api/Site.php	App/Api/Site::index()
?s=Weibo.Login	/src/Api/Weibo.php	App/Api/Weibo::login()

2.x 请求的service参数	对应的文件	执行的类方法
?s=User.Weibo.Login	./src/user/Api/Weibo.php	User/Api/Weibo::login()
?s=App_User.Third_Weibo.Login	./src/app/user/Api/Third/Weibo.php	App/User/Api/Third/Weibo::login()

## Model层自动匹配的表名差异

在1.x 版本中， 默认表名的自动匹配规则是：取Model\_后面部分的字符全部转小写。例如：

```
<?php
class Model_User extends PhalApi_Model_NotORM {
```

类Model\_User自动匹配的表名为user。

在2.x 版本中， 规则类似， 但由于多了命名空间， 默认表名的自动匹配规则是：取\Model\后面部分的字符全部转小写， 将用下划线分割。例如：

```
<?php
namespace App\Model\User;
use PhalApi\Model\NotORMModel as Model;

class Friends extends Model {
```

则类App\Model\User\Friends自动匹配的表名为user\_friends。以下是2.x版本的一些示例：

2.x 的Model类名	对应的文件	自动匹配的表名
App\Model\User	./src/app/Model/User.php	user
App\ModelUser\Friends	./src/app/Model/User/Friends.php	user_friends
App\User\Model\Friends	./src/app/user/Model/Friends.php	friends
App\User\Model\User\Friends	./src/app/user/Model/User/Friends.php	user_friends

和1.x 版本相同的是， 自动匹配的表名后面会自动加上表前缀。

当自动匹配的表名不能满足实际开发需求时， 1.x 和 2.x 版本均支持自定义表名。

## 可访问的入口差异

在PhalApi 1.x 版本中， 对外可访问的入口主要有：

- 不同项目的访问入口， 如： /Public/demo/index.php
- 在线接口列表文档， 如： /Public/demo/listAllApis.php
- 在线接口详情文档， 如： /Public/demo/checkApiParams.php

如何访问在线接口文档？

鉴于PhalApi 1.x版本中的访问入口过于分散， 且在线接口文档链接过于冗长，在PhalApi 2.x 版本中， 我们同时对此进行了优化。即精简为：

- 访问入口默认只有一个， 即： /public/index.php
- 在线接口列表与接口详情文档， 二合为一， 链接改为： /public/docs.php

## 项目目录结构的差异

在PhalApi 1.x中， 项目目录结构为：

```
.
├── PhalApi //PhalApi框架， 后期可以整包升级
├── Library //PhalApi扩展类库， 可根据需要自由添加扩展
└── SDK //PhalApi提供的SDK包， 客户可根据需要选用

├── Public //对外访问目录， 建议隐藏PHP实现
│ └── demo //Demo服务访问入口

├── Config //项目接口公共配置， 主要有： app.php, sys.php, dbs.php
├── Data //项目接口公共数据
├── Language //项目接口公共翻译
└── Runtime //项目接口运行文件目录， 用于存放日记， 可软链接到别的区

└── Demo //应用接口服务， 名称自取， 可多组
 ├── Api //接口响应层
 ├── Domain //接口领域层
 ├── Model //接口持久层
 └── Tests //接口单元测试
```

在PhalApi 2.x中， 项目目录结构为：

```
.
└── config // 项目接口公共配置
 ├── app.php // 项目配置
 ├── dbs.php // 数据库配置
 ├── di.php // DI依赖注入配置
 └── sys.php // 系统环境配置
```

```

 └── public // 对外访问目录，推荐将web根路径设定在此目录
 ├── docs.php // 自动生成的在线接口文档
 ├── examples // 示例
 ├── index.php // 接口服务统一访问入口
 └── init.php // 统一初始化文件

 └── runtime // 项目运行时产生的文件目录

 └── src // 项目PHP源代码
 └── app // 默认使用此App命名空间，可创建多上命名空间
 ├── Api // 接口响应控制层
 ├── Domain // 接口领域业务层
 ├── functions.php // 面向过程式的函数
 └── Model // 接口数据模型层

 └── tests // 单元测试

 └── bin // 脚本命令
 └── data // 用于存放SQL建表基本语句
 └── language // 语言翻译包
 └── sdk // 客户端SDK开发包，支持9+种语言

 └── composer.json // composer.json文件
 └── vendor // 依赖安装包

```

主要区别在于，对于不放置PHP源代码的目录，全部改用小写。

## 附录：对照表

### 附录1：类对照表

以下是PhalApi 2.x 与PhalApi 1.x 的类对照关系。

PhalApi v2.x	PhalApi v1.x	备注
PhalApi\PhalApi	PhalApi	
PhalApi\Api	PhalApi_Api	
PhalApi\ApiFactory	PhalApi_ApiFactory	
PhalApi\Cache	PhalApi_Cache	
PhalApi\Cache\APCUCache	PhalApi_Cache_APCU	
PhalApi\Cache\FileCache	PhalApi_Cache_File	
PhalApi\Cache\MemcacheCache	PhalApi_Cache_Memcache	
PhalApi\Cache\MemcachedCache	PhalApi_Cache_Memcached	
PhalApi\Cache\MultiCache	PhalApi_Cache_Multi	
PhalApi\Cache\NoneCache	PhalApi_Cache_None	
PhalApi\Cache\RedisCache	PhalApi_Cache_Redis	
PhalApi\Config	PhalApi_Config	
PhalApi\Config\FileConfig	PhalApi_Config_File	
PhalApi\Config\YaconfConfig	PhalApi_Config_Yaconf	
PhalApi\Cookie	PhalApi_Cookie	
PhalApi\Cookie\MultiCookie	PhalApi_Cookie_Multi	
PhalApi\Crypt	PhalApi_Crypt	
PhalApi\Crypt\McryptCrypt	PhalApi_Crypt_Mcrypt	
PhalApi\Crypt\MultiMcryptCrypt	PhalApi_Crypt_MultiMcrypt	
PhalApi\Crypt\RSA\KeyGenerator	PhalApi_Crypt_RSA_KeyGenerator	
PhalApi\Crypt\RSA\MultiBase	PhalApi_Crypt_RSA_MultiBase	
PhalApi\Crypt\RSA\MultiPri2PubCrypt	PhalApi_Crypt_RSA_MultiPri2Pub	
PhalApi\Crypt\RSA\MultiPub2PriCrypt	PhalApi_Crypt_RSA_MultiPub2Pri	
PhalApi\Crypt\RSA\Pri2PubCrypt	PhalApi_Crypt_RSA_Pri2Pub	
PhalApi\Crypt\RSA\Pub2PriCrypt	PhalApi_Crypt_RSA_Pub2Pri	
PhalApi\CUrл	PhalApi_CUrl	
PhalApi\Database	PhalApi_DB	
PhalApi\Database\NotORMDatabase	PhalApi_DB_NotORM	改用Database全称
PhalApi\DependencyInjection	PhalApi_DI	改用DependenceInjection全称
PhalApi\Exception	PhalApi_Exception	
PhalApi\Exception\BadRequestException	PhalApi_Exception_BadRequest	
PhalApi\Exception\InternalServerErrorException	PhalApi_Exception_InternalServerError	
PhalApi\Exception\RedirectException	PhalApi_Exception_Redirect	
PhalApi\Filter	PhalApi_Filter	
PhalApi\Filter\NoneFilter	PhalApi_Filter_None	
PhalApi\Filter\SimpleMD5Filter	PhalApi_Filter_SimpleMD5	
PhalApi\Helper\ApiDesc	PhalApi_Helper_ApiDesc	应用层不用关注
PhalApi\Helper\ApiList	PhalApi_Helper_ApiList	应用层不用关注
PhalApi\Helper\ApiOnline	PhalApi_Helper_ApiOnline	应用层不用关注
PhalApi\Helper\TestRunner	PhalApi_Helper_TestRunner	

PhalApi v2.x	PhalApi v1.x	备注
PhalApi\Helper\Tracer	PhalApi_Helper_Tracer	
PhalApi\Loader	PhalApi_Loader	
PhalApi\Logger	PhalApi_LOGGER	
PhalApi\Logger\ExplorerLogger	PhalApi_LOGGER_Explorer	
PhalApi\Logger\FileLogger	PhalApi_LOGGER_File	
PhalApi\Model	PhalApi_MODEL	
PhalApi\Model\NotORMModel	PhalApi_MODEL_NotORM	
PhalApi\Model\Proxy	PhalApi_MODELProxy	
PhalApi\Model\Query	PhalApi_MODELQuery	
PhalApi\Request	PhalApi_REQUEST	
PhalApi\Request\Formatter	PhalApi_REQUEST_Formatter	
PhalApi\Request\Formatter\ArrayFormatter	PhalApi_REQUEST_Formatter_Array	应用层不用关注
PhalApi\Request\Formatter\BaseFormatter	PhalApi_REQUEST_Formatter_Base	应用层不用关注
PhalApi\Request\Formatter\BooleanFormatter	PhalApi_REQUEST_Formatter_Boolean	应用层不用关注
PhalApi\Request\Formatter\CallableFormatter	PhalApi_REQUEST_Formatter_Callable	应用层不用关注
PhalApi\Request\Formatter\CallbackFormatter	PhalApi_REQUEST_Formatter_Callback	应用层不用关注
PhalApi\Request\Formatter\DateFormatter	PhalApi_REQUEST_Formatter_Date	应用层不用关注
PhalApi\Request\Formatter\EnumFormatter	PhalApi_REQUEST_Formatter_Enum	应用层不用关注
PhalApi\Request\Formatter\FileFormatter	PhalApi_REQUEST_Formatter_File	应用层不用关注
PhalApi\Request\Formatter\FloatFormatter	PhalApi_REQUEST_Formatter_Float	应用层不用关注
PhalApi\Request\Formatter\IntFormatter	PhalApi_REQUEST_Formatter_Int	应用层不用关注
PhalApi\Request\Formatter\StringFormatter	PhalApi_REQUEST_Formatter_String	应用层不用关注
PhalApi\Request\Parser	PhalApi_REQUEST_Var	重命名为Parser，避免与关键字var冲突
PhalApi\Response	PhalApi_RESPONSE	
PhalApi\Response\ExplorerResponse	PhalApi_RESPONSE_Explorer	
PhalApi\Response\JsonResponse	PhalApi_RESPONSE_Json	
PhalApi\Response\JsonpResponse	PhalApi_RESPONSE_JsonP	注意p字母为小写
PhalApi\Response\XmlResponse	PhalApi_RESPONSE_Xml	
PhalApi\Tool	PhalApi_TOOL	
PhalApi\Translator	PhalApi_TRANSLATOR	

## 类名重命名规则

原来的类名遵循PEAR规范，现需要调整遵循PSR-4规范。如：

原来的：PhalApi\_Filter

调整后：\PhalApi\Filter

对于有继承的情况，为了避免最后的关键字有冲突，统一在子类后面添加父类的名称作为后续。如：

原来的：

PhalApi\_COnfig\_File  
PhalApi\_COnfig\_Yaconf

调整后：

PhalApi\Config\FileConfig  
PhalApi\Config\YaconfConfig

## 附录2：函数对照表

以下是PhalApi 2.x 与PhalApi 1.x 的函数对照关系。

### PhalApi v2.x PhalApi v1.x 备注

PhalApi\DI() DI()  
PhalApi\SL() SL()  
PhalApi\T() T()

函数名保持一致，但需要注意前面添加PhalApi命名空间前缀。

## 附录3：脚本命令对照表

以下是PhalApi 2.x 与PhalApi 1.x 的脚本命令对照关系。

PhalApi v2.x	PhalApi v1.x	备注
./bin/phalapi-buildsqls	./PhalApi/phalapi-buildsqls	生成SQL语句
./bin/phalapi-buildtest	./PhalApi/phalapi-buildtest	生成测试骨架代码
暂未迁移	./PhalApi/phalapi-buildcode	创建项目代码
暂未迁移	./PhalApi/phalapi-buildapp	创建新项目，暂不需要迁移

## 附录4：可访问入口对照表

以下是PhalApi 2.x 与 PhalApi 1.x 的可访问入口对照关系。

PhalApi v2.x	PhalApi v1.x	备注
./public/index.php	./Public/demo/index.php	可省略index.php文件
./public/docs.php	./Public/demo/listAllApis.php	在线接口列表文档
./public/docs.php?detail=1	./Public/demo/checkApiParams.php	在线接口详情文档，通过detail参数区分
./public/docs/	./Public/demo/docs/	离线文档生成目录

## PhalApi 2.x VS PhalApi 1.x

不同的使用方式和组织方式，不仅决定了框架内部特质上的差异，还影响了外部使用上的区别。本文章主要介绍**PhalApi v2.0.0 版本与PhalApi v1.4.1 版本**之间的内部差异。并约定，下文中，**新版本**是指PhalApi v2.0.0 版本，**旧版本**是指PhalApi v1.4.1 版本。

新、旧版本主要的对比结论，汇总如下：

- 新、旧版基准测试结果基本一致
- 新、旧版本执行时间相差约为1毫秒
- 新版的单元测试，覆盖率达90%以上，通过率为100%
- 新版的技术债务仅1天，质量更优！

### 基准测试对比

#### 对比结论：新、旧版基准测试结果基本一致

压测环境配置为：

- 阿里云服务器ECS (CPU: 1核 内存: 1 GB 宽带: 1Mbps)
- 操作系统: CentOS release 6.7 (Final)
- nginx/1.8.0
- PHP 5.3.5

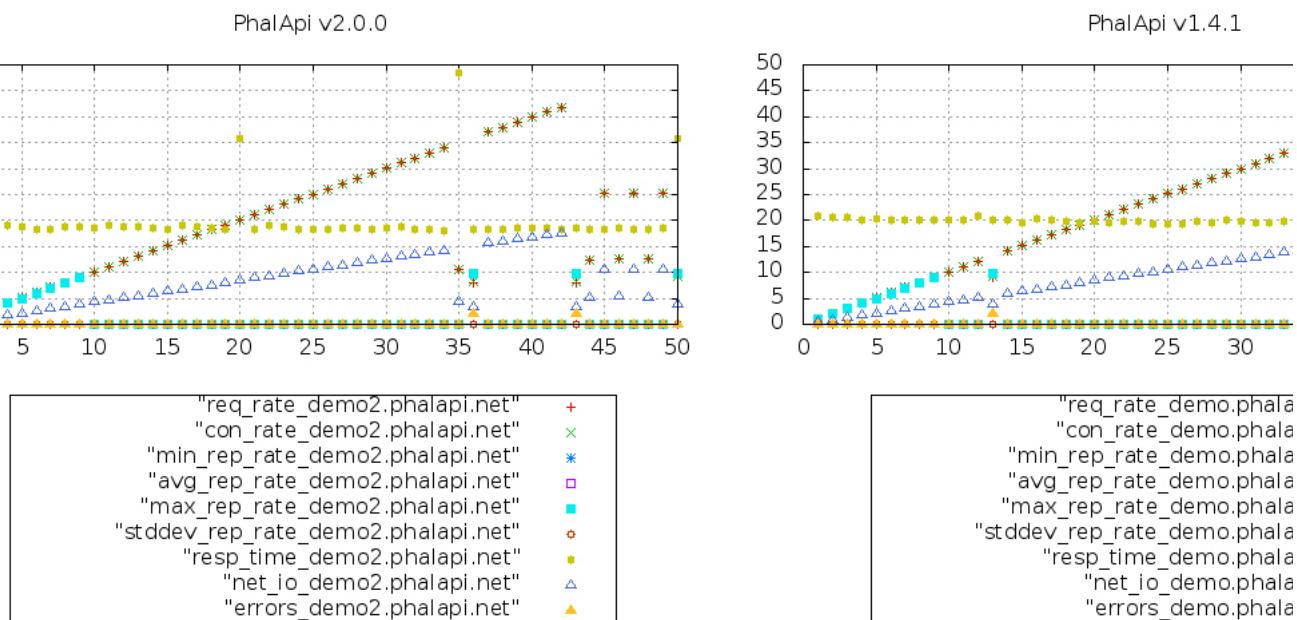
待压测的接口服务链接：

- PhalApi v2.0.0 默认接口服务：<http://demo2.phalapi.net/>
- PhalApi v1.4.1 默认接口服务：<http://demo.phalapi.net/>

这里，使用Autobench进行基准测试，压测脚本为：

```
autobench \
--single_host \
--host1=$DM \
--port1=80 \
--uri1=$URL \
--low_rate=1 \
--high_rate=50 \
--rate_step=1 \
--num_call=1 \
--num_conn=50 \
--timeout=5 \
--file ./${DM}.tsv
```

对于新、旧版本，其压测结果对比如下（左图为新版，右图为旧版）：



在并发量为50以内时，新、旧版本的响应时间基本一致，约为20 ms。对于新版，详细的压测报告数据如下：

dem_req_rate	req_rate_demo2.phalapi.net	con_rate_demo2.phalapi.net	min_rep_rate_demo2.phalapi.net	avg_rep_rate_demo2.phalapi.net	max_rep_rate_demo2.phalapi.net
1	1.0 1.0 1.0 1.0 0.0 20.2	0.4 0			
2	2.0 2.0 2.0 2.0 0.0 18.7	0.9 0			
3	3.1 3.1 3.0 3.0 0.0 18.9	1.3 0			
4	4.1 4.1 4.0 4.0 0.0 19.2	1.7 0			
5	5.1 5.1 5.0 5.0 0.0 18.7	2.1 0			
6	6.1 6.1 6.0 6.0 0.0 18.4	2.6 0			
7	7.1 7.1 7.0 7.0 0.0 18.4	3.0 0			
8	8.1 8.1 8.0 8.0 0.0 18.7	3.4 0			
9	9.1 9.1 9.0 9.0 0.0 18.9	3.8 0			
10	10.1 10.1 0.0 0.0 0.0 18.5	4.3 0			
11	11.1 11.1 0.0 0.0 0.0 19.0	4.7 0			
12	12.2 12.2 0.0 0.0 0.0 18.9	5.1 0			
13	13.2 13.2 0.0 0.0 0.0 18.9	5.5 0			
14	14.2 14.2 0.0 0.0 0.0 18.5	6.0 0			
15	15.2 15.2 0.0 0.0 0.0 18.4	6.4 0			
16	16.2 16.2 0.0 0.0 0.0 19.0	6.8 0			
17	17.2 17.2 0.0 0.0 0.0 18.8	7.2 0			
18	18.2 18.2 0.0 0.0 0.0 18.5	7.6 0			
19	19.2 19.2 0.0 0.0 0.0 18.4	8.1 0			
20	20.2 20.2 0.0 0.0 0.0 35.9	8.5 0			
21	21.2 21.2 0.0 0.0 0.0 18.3	8.9 0			
22	22.2 22.2 0.0 0.0 0.0 19.0	9.3 0			
23	23.2 23.2 0.0 0.0 0.0 18.8	9.7 0			
24	24.1 24.1 0.0 0.0 0.0 18.4	10.2 0			
25	25.1 25.1 0.0 0.0 0.0 18.3	10.6 0			
26	26.1 26.1 0.0 0.0 0.0 18.4	11.0 0			
27	27.1 27.1 0.0 0.0 0.0 18.5	11.4 0			
28	28.1 28.1 0.0 0.0 0.0 18.5	11.8 0			
29	29.1 29.1 0.0 0.0 0.0 18.4	12.3 0			
30	30.1 30.1 0.0 0.0 0.0 18.5	12.7 0			
31	31.1 31.1 0.0 0.0 0.0 18.7	13.1 0			
32	32.0 32.0 0.0 0.0 0.0 18.3	13.5 0			
33	33.0 33.0 0.0 0.0 0.0 18.3	13.9 0			
34	34.0 34.0 0.0 0.0 0.0 18.1	14.3 0			
35	10.5 0.0 0.0 0.0 0.0 48.5	4.4 0			
36	8.1 8.3 9.8 9.8 9.8 0.0 18.3	3.4 2.04081632653061			
37	37.0 0.0 0.0 0.0 0.0 18.2	15.6 0			
38	37.9 0.0 0.0 0.0 0.0 18.4	15.9 0			
39	38.9 0.0 0.0 0.0 0.0 18.6	16.4 0			
40	39.9 0.0 0.0 0.0 0.0 18.6	16.8 0			
41	40.9 0.0 0.0 0.0 0.0 18.6	17.2 0			
42	41.8 0.0 0.0 0.0 0.0 18.4	17.6 0			
43	8.0 8.2 9.8 9.8 9.8 0.0 18.6	3.4 2.04081632653061			
44	12.4 0.0 0.0 0.0 0.0 18.4	5.2 0			
45	25.2 0.0 0.0 0.0 0.0 18.2	10.6 0			
46	12.5 0.0 0.0 0.0 0.0 18.5	5.3 0			
47	25.2 0.0 0.0 0.0 0.0 18.3	10.6 0			
48	12.5 0.0 0.0 0.0 0.0 18.4	5.2 0			
49	25.2 0.0 0.0 0.0 0.0 18.6	10.6 0			
50	9.3 9.3 9.8 9.8 9.8 0.0 35.7	3.9 0			

## XHprof性能剖析对比

### 对比结论：新、旧版本执行时间相差约为1毫秒

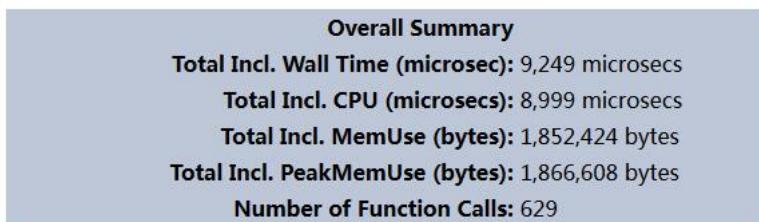
使用XHprof对新、旧版本进行性能剖析，经过多次分析并取各自最优值，对关键性能指标的对比如下：

性能指标	PhalApi v2.0.0 新版本	PhalApi v1.4.1 旧版本	趋势
Total Incl. Wall Time (microsec)	9,249 microsecs	8,393 microsecs	+ 10.20%
Total Incl. CPU (microsecs)	8,999 microsecs	6,999 microsecs	+ 28.58%
Total Incl. MemUse (bytes)	1,852,424 bytes	1,608,600 bytes	+ 15.16%
Total Incl. PeakMemUse (bytes)	1,866,608 bytes	1,619,544 bytes	- 10.27%
Number of Function Calls	629	701	+ 13.69%

就上面报告的数据可以看出，新版本的各项性能指标比旧版本有所增加。这是因为引入了composer机制所产生的影响。虽然有所涨幅，但由于基数低，新版本的性能还是非常优异的。例如对于Wall Time，新版本为9,249毫秒，旧版本的Wall Time为8,393 microsecs，仅相差了0.856毫秒，即不到1毫秒。执行时间会随系统环境配置不同，执行时的系统状态不同，会相对变化，而函数调用的次数则是固定的。如果仅从函数调用次数来对比，新版本则比旧版本少了72次调用。

## 新版本的XHprof报告

新版本的XHprof性能报告概览如下：

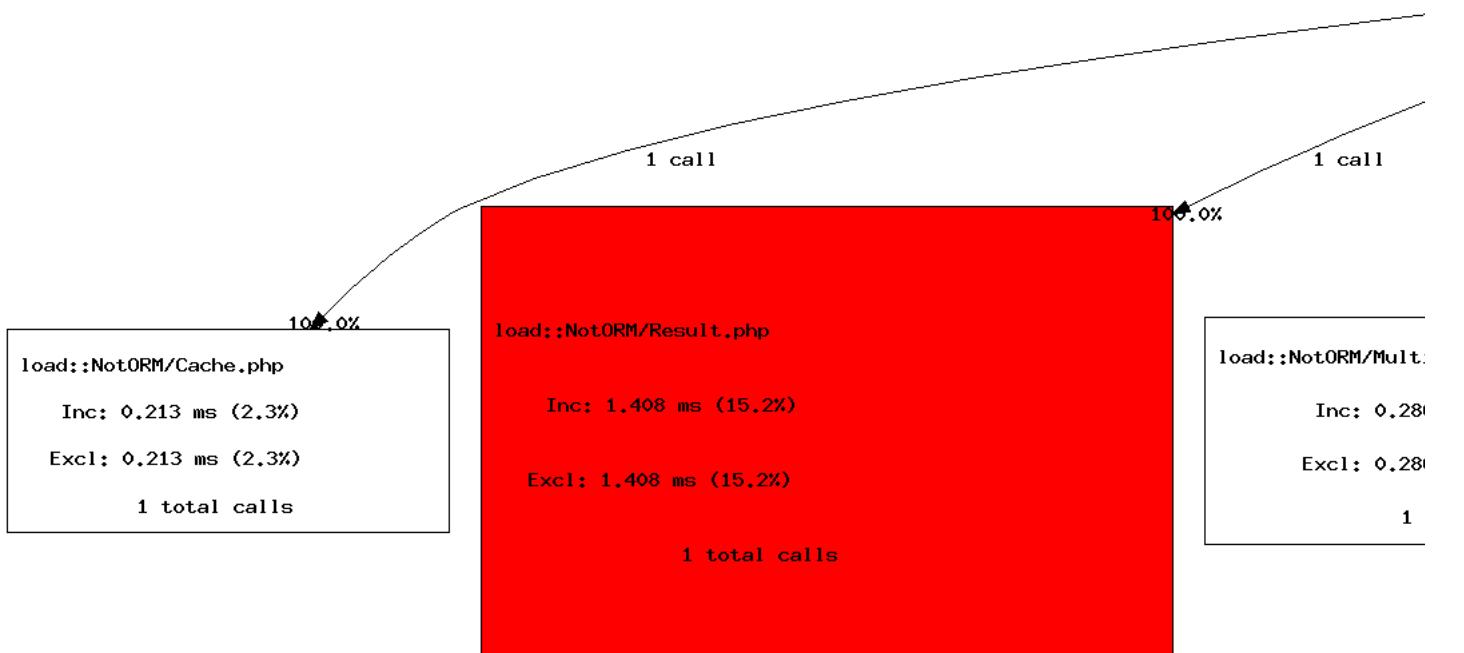


新版本的Top 10耗时操作是：

Function Name	Calls	Calls%	Excl. Wall Time (microsec)	EWall%
---------------	-------	--------	----------------------------	--------

Function Name	Calls	Calls%	Excl. Wall Time (microsec)	EWall%
load::NotORM/Result.php	1	0.20%	1408	15.20%
run_init::src/bootstrap.php	1	0.20%	2813	30.40%
load::composer/ClassLoader.php	1	0.20%	449	4.90%
load::Database/NotORMDatabase.php	1	0.20%	314	3.40%
load::NotORM/MultiResult.php	1	0.20%	286	3.10%
load::NotORM/Structure.php	1	0.20%	255	2.80%
load::NotORM/Row.php	1	0.20%	252	2.70%
load::src/Request.php	1	0.20%	243	2.60%
load::NotORM/Cache.php	1	0.20%	213	2.30%
PhalApi\Request::getAllHeaders	1	0.20%	229	2.50%

对应的高清版可视化图表如下：



可以看到，最耗时的操作是对NotORM文件的引入（上图红色部分），这与旧版本最耗时的操作是一样的。

更多点击查看：[更多](#)

- [新版本的XHprof报告 - 20170716](#)
- [旧版本的XHprof报告 - 20170709](#)

## 单元测试覆盖率对比

**对比结论：新版依然保持着90%以上的单元测试覆盖率**

PhalApi一直推荐使用测试驱动的开发方式，通过意图导向编程，提高开发效率、提升代码质量。

对于PhalApi自身框架的开发，我们同样也是遵循TDD的最佳实践，争取为开源社区产出优质的框架。对于新版本，其核心框架代码部分的单元测试覆盖率达94%以上，如下图所示：



### Legend

Low: 0% to 50%   Medium: 50% to 90%   High: 90% to 100%

Generated by PHP\_CodeCoverage 2.2.4 using PHP 5.3.10-1ubuntu3.26 and PHPUnit 4.8.36 at Sat Jul 8 21:31:57 CST 2017.

此外，核心框架的单元测试通过率是100%。执行单元测试套件的输出效果，类似如下：

```
/path/to/phalapi/kernal/tests$ phpunit -c ./phpunit_silence.xml
PHPUnit 4.3.4 by Sebastian Bergmann.

Configuration read from /path/to/phalapi/kernal/tests/phpunit_silence.xml
..... 63 / 327 (19%)
..... 126 / 327 (38%)
..... 189 / 327 (57%)
..... 252 / 327 (77%)
..... 315 / 327 (96%)
.....
```

Time: 14.05 seconds, Memory: 26.25Mb

OK (327 tests, 480 assertions)

## 静态代码质量分析对比

**对比结论：新版的技术债务仅1天，质量更优！**

借助于开源中国码云上的代码分析服务，可以得到以下Sonar分析报告，从中可以看到新版框架的核心部分技术债务**仅有1天**。

### sonarQube Sonar常规代码分析

master 的当前版本已经分析成功 [分析其他分支](#)

分析结果 (完成时间:2017-07-09 15:15:37)

 kernal kernel-2152409

git地址: <http://git.oschina.net/dogstar/kernal.git>

Profiles: Sonar way (PHP)

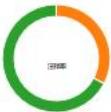
代码行数 2,663 文件 72

PHP 目录 15 行数 6,007

方法 260

类 语句 69 1,396

文档和注释 1295.0 (注释行数) 0 (公共API数)

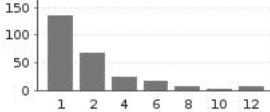


- 已注释
- 未注释

技术债务 1天 问题 97

阻断	0
严重	0
主要	13
次要	79
信息	5

复杂度 2.7 /方法 10.2 /类 10.6 /文件  
总数: 764



方法 文件

问题 打开 重开 确认

97	97	0	0
----	----	---	---

技术债务金字塔 ■ 技术债务 总数

快速对比新、旧版的静态代码质量，可以得出：新版本在遵循composer和psr-4规范下，代码质量更优。例如，技术债务从原来1天5小时降为1天，问题总数从158个降为97个。

质量指标|PhalApi v2.0.0 新版本|PhalApi v1.4.1 旧版本|趋势

代码行数|2663|2267| + 17.46% 技术债务|1天|1天 5小时| - 17.24% 问题总数|97|158| - 38.61% 复杂度（方法）|2.7|2.7| 0%

更多点击查看：

- [PhalApi v2.0.0 新版本Sonar分析报告](#)
- [PhalApi v1.4.1 旧版本Sonar分析报告](#)

[还有疑问？欢迎到社区提问！](#) 切换到[PhalApi 2.x 开发文档](#)。