

目錄

简介

YYF简介	1.1
快速上手	1.2
[对比(更新中)]	1.3

环境和安装

基本环境	2.1
开发环境配置	2.2
虚拟机环境	2.2.1
Windows 下虚拟机	2.2.1.1
Linux和Mac虚拟机	2.2.1.2
本机开发	2.2.2
Windows 下配置	2.2.2.1
Linux和Mac配置	2.2.2.2
Docker运行环境	2.3
服务器-生产环境	2.4

[架构]

目录结构	3.1
MVC	3.2
REST	3.3
[格式规范(更新中)]	3.4
[运行生命周期(更新中)]	3.5

[配置]

[配置类型和特点(更新中)]	4.1
[ini基础配置(更新中)]	4.2
[特殊配置(更新中)]	4.3
[配置内容(更新中)]	4.3.1
[开发环境(更新中)]	4.3.2
[生产环境(更新中)]	4.3.3
Config 配置读取	4.4

[控制器]

[REST控制器(更新中)]	5.1
[YAF基础控制器(更新中)]	5.2
[异常处理(更新中)]	5.3

数据库与Model

数据库	6.1
使用示例	6.2
Db 辅助类	6.3
Model 数据模型	6.4
Orm 数据对象映射	6.5
Database 数据库连接	6.6

调试

调试方法	7.1
header	7.2
assert 断言	7.3
Logger 日志管理	7.4
[异常(更新中)]	7.5
[数据库调试(更新中)]	7.6

[路由]

默认路由	8.1
[路由配置(更新中)]	8.2
[简单路由(更新中)]	8.3
[正则路由(更新中)]	8.4

[输入]

[input(更新中)]	9.1
[请求方式(更新中)]	9.1.1
[过滤和回调(更新中)]	9.1.2
Cookie加密	9.2
[参数自动绑定(更新中)]	9.3

存储

快速存储	10.1
Cache 缓存	10.2
Kv 键值对存储	10.3
Session 存取封装	10.4

【安全】

[常用加密(更新中)]	11.1
[格式保留加密(更新中)]	11.1.1
RSA非对称加密	11.2
[安全统计(更新中)]	11.3
CORS 跨站请求	11.4

【库】

[文件上传(更新中)]	12.1
[邮件(更新中)]	12.2
[短信(更新中)]	12.3
Wechat微信	12.4

前端插件

YYF-Debugger调试插件	13.1
YYFJS API请求接口库	13.2

【优化】

[php优化(更新中)]	14.1
[yaf配置(更新中)]	14.2
[apache(更新中)]	14.3

YYF-book

YYF 开发文档

Documentation for YUNYIN YAF Framework

- 文档主站(gitbook服务): <https://yyf.newfuture.cc>
- 文档加速站(CDN 加速): <https://yyf.newfuture.xyz>
- 文档源码: <https://github.com/NewFuture/yyf-book>
- PDF版下载: <https://yyf.newfuture.xyz/yyf.pdf>

YYF(Yunyin Yaf Framework)

基于PHP的YAF扩展构建的高效,安全,简单,优雅的 开源RESTful 框架。

项目主页 <https://github.com/YunYinORG/YYF>

设计宗旨: 以生产环境下安全高效运行为前提,尽量让开发优雅方便,尽力提高运行性能和开发便捷。

YYF最初是从第二版云印系统后端核心框架萃取和完善发展而来,在不同环境下提供简单一致的开发体验,并在服务器上快速部署和高效运行; 鉴于流行Laravel框架和使用较多的ThinkPHP框架的使用习惯,以 yaf扩展作为底层框架提高整体性能,开发的RESTful后端PHP框架。

如果使用过Laraval或者Thinkphp等任何PHP框架,或者熟悉Rails等类似的web框架,可轻松上手YYF。

主要特点:

- 安全:
 - 数据库完全采用PDO封装从底层防止SQL注入
 - 输入参数类型绑定, 提供输入过滤封装
 - 高效封装常用加密库, 包括云印系统的格式保留加密算法
 - 生产环境,对文件权限进行严格限制
 - CORS封装管理和限制跨域请求
- 高效:
 - 使用YAF扩展(c编译)作为框架底层驱动;
 - 核心库保证安全和高效运行为前提, 独立模块内部紧耦合, 按需加载;
 - 底层框架配置文件常驻内存,减少文件IO;
 - 针对PHP7特性优化,在PHP7下性能更优
- 简单:
 - 自带跨平台的一键初始化和命令脚本(不需要PHP环境)
 - 对REST路由和输出采用配置管理,并可根据浏览器请求方便的配置跨站请求(CORS)
 - 对常用操作高效封装,并对数据库,邮件,微信, 七牛云等常用服务进行高效定制封装
 - 开发环境自动进行性能统计, 方便后期优化
 - 提供Chrome调试插件YYF-Debugger, 在浏览器中查看调试信息
- 优雅:

- 静态封装,对于常用操作进行静态封装,让开发代码更简洁
- 开发环境调试注入,无需改动代码,自动根据系统配置切换环境
- 不同环境和服务尽量提高一致的接口,
- 开发环境自动header输出调试信息和日志
- 兼容:
 - 支持PHP5.3及以上所有稳定版本,可自动根据版本安装YAF
 - 在各种服务器环境包括云平台之间平滑迁移和部署
 - 提供Vagrant虚拟机开发环境,为不同系统和使用习惯的开发者提供稳定一致的开发体验
 - 集成单元测试,与travis-ci无缝对接,可在不同环境自动测试和持续集成

Copyright © NewFuture 2016 all right reserved, powered by Gitbook文档编译时间: 2017-02-21 11:03:42

快速开始

介绍几个简单controller，体验一下YYF的运行流程。以下示例不需要额外路由配置。

几个简单的例子

1. [hello world](#)
2. [REST请求\(GET POST\)](#)
3. [id映射](#)

1. hello world 示例

首先输出一个hello world!

首先新建一个 `app/controllers/Index.php` (实际上已经存在了，可以打开直接修改!)

```
<?php
class IndexController extends Rest
{
    /*首页*/
    public function indexAction()
    {
        echo 'hello world!';
    }
}
```

然后打开浏览器 输入你的调试地址 `192.168.23.33` (YYF虚拟机), `127.0.0.1:1122` (PHP测试服务器)或者 `localhost` (本机)

就能看到如下内容,就成功了

```
hello world!
```

这是经典MVC的控制器路由流程，请求 `/` 内部的过程 `IndexController` (默认) -> `indexAction` (默认)

2. REST请求

常用的请求如 `GET` , `POST` , `PUT` , `DELETE` 等,不同的请求使用不同的action来响应。

数据默认采用json来编码。

在建一个TestController `app/controllers/Test.php`

2.1. GET 请求

`app/controllers/Test.php` ,内容如下

```
<?php
class TestController extends Rest
{
    /*响应 GET /Test/demo*/
    public function GET_demoAction()
    {
        $this->response(1, 'Hello, it is a GET request!');//响应数据
    }

    /*可以继续添加其他action*/
}
```

打开浏览器 {测试主机}/Test/demo (其中 {测试主机} 为 192.168.23.33 , 127.0.0.1:1122 或 localhost)

会看到如下数据

```
{"status":1,"info":"Hello, it is a GET request!"}
```

浏览器默认打开URL是GET 请求, GET /Test/demo 被路由到 TestController->GET_demoAction() 。
response() 会直接把数据格式换成json输出。

2.2. POST 请求

接着在 app/controllers/Test.php 添加一个新的action POST_demoAction ,如下

```
/*响应 POST /Test/demo*/
public function POST_demoAction()
{
    $info['method']='POST';
    Input::post('msg',$info['msg']);//获取POST数据赋值到$info['msg']

    $this->response['status'] = 1;//响应状态
    $this->response['info']   = $info;//响应数据
}
```

然后用curl命令(windows可以使用浏览器插件测试)模拟一个POST请求(192.168.23.33 换成测试主机地址即可)

```
curl -X POST -d "msg=这是一条POST请求!" 192.168.23.33/Test/demo
```

返回数据如下(为了方便阅读json数据已经格式化)

```
{
  "status":1,
  "info":
  {
    "method":"POST",
    "msg":"这是一条POST请求!"
  }
}
```

POST /Test/demo => TestController->POST_demoAction() 进行响应

3. id参数映射

在 RESTful 的API设计中，URL 通常是这样的例如

1. `https://yyf.yunyun.org/products/1234` 获取id为1234的产品信息(非restful设计可能是这样的 `https://yyf.yunyun.org/products/info?id=1234`)
2. `https://yyf.yunyun.org/products/1234/comments` 获取id为1234的产品评论(非restful设计可能是这样的 `https://yyf.yunyun.org/comments/list?products_id=1234`)

建一个ProductsController `app/controllers/Products.php`

3.1. infoAction

在 `app/controllers/Products.php` 中添加一个 `GET_infoAction`

```
<?php
class ProductsController extends Rest
{
    /*响应 GET /Products/{id}*/
    public function GET_infoAction($id=0)
    {
        $product=['id'=>$id, 'more'=>'products 详情'];
        $this->response(1,$product);//响应数据

        /* //实际上可能要查询数据库
        if($product=Db::table('product')->find(intval($id)){
            $this->response(1,$product);//响应数据
        }else{
            $this->response(0,'no such product');//无查询结果
        }
        */

    }

    /*可以继续添加其他action*/
}
```

REST 默认会把数字1234绑定到参数 `$id` 上,并映射到默认的 `infoAction` (名字可以在配置中修改)操作上。

浏览器打开 `http://192.168.23.33/Products/123` (其中192.168.23.33换成你的测试地址)

```
{
  "status": 1,
  "info": {
    "id": 123,
    "more": "products 详情"
  }
}
```

3.2 id参数绑定

继续在 `app/controllers/Products.php` 中添加一个 `GET_commentsAction`

```
/*响应 GET /Products/{id}/comments*/
public function GET_commentsAction($id=0)
{
    $comments=[
        ['id'=>1, 'product_id'=>$id, 'content'=>'nice!'],
        ['id'=>3, 'product_id'=>$id, 'content'=>''],
    ];
    /* //实际可能需要查询数据库
        $comments=Db::table('comment')->where('product_id',$id)->select();
    */
    $this->response(1,$comments); //响应数据
}
```

数字123被绑定到参数 `$id` 上, 映射到 `commentsAction`。

浏览器打开 `http://192.168.23.33/Products/123/comments` (其中192.168.23.33换成你的测试地址)

```
{
  "status": 1,
  "info": [
    {"id": 1, "product_id": 123, "content": "nice!"},
    {"id": 3, "product_id": 123, "content": "" }
  ]
}
```

YYF 运行环境

YYF 会根据不同环境切换运行方式。

开发环境最大程度的方便调试和提高开发效率， 和提供简单统一环境。

生产环境，保证安全性和尽可能高的性能(执行效率)。

- [开发环境](#)(甚至你的电脑上不用需要PHP环境)
- [生产环境](#)(可一键部署)

基本环境

1. 必要基础环境

YYF 是基于 [YAF](#) 扩展的 [PHP](#) 框架，所以这两点是必须的

- 【必需】PHP (版本>=5.3)
- 【必需】[YAF扩展](#)
- 【可选】mcrypt和openssl扩展(使用加密相关库需要)
- 【可选】PDO(使用数据库连接需要)
- 【可选】CURL(使用第三方接口需要)

2. 数据库和服务支持

- 数据库支持:(PDO封装,支持大部分关系型数据库)
 - MaraiaDB: 完全封装
 - MySQL: 完全封装
 - SQLite: 常用支持
 - 其他数据库: 未知
- 服务器支持: (无限制)
 - Apache
 - Nginx
 - IIS
 - 其他

3. 生产环境服务器部署支持

YYF对库进行轻量级的封装，可以在不同平台和服务器之间平滑迁移，而不必修改代码。

- 云服务器(虚拟机) 完全支持
 - Linux 服务器
 - Windows 服务器
- Docker
 - [YAF-docker](#)

- 云引擎(PAAS)支持
 - SAE(新浪云): 完全支持
 - BAE(百度云): 未测试

Copyright © NewFuture 2016 all right reserved, powered by Gitbook文档编译时间: 2017-02-21 11:03:42

开发环境配置

开发环境应该尽可能高效和方便，你有两种方式可供选择

- [虚拟机环境](#)
- [本机开发环境](#)

I. 虚拟机环境(推荐)

YYF定制了一个集成了YYF所需环境的虚拟机镜像(大约350M),并提供几乎一键安装和自动配置的脚本(`init.cmd`). 可以让不同人在不同系统具有完全一致的开发环境，尤其适合团队使用和新手开发。

(注：虚拟理解只提供服务器运行环境，代码存和修改仍然在本机，会自动映射到虚拟机中。)

使用YYF虚拟机镜像你可能需要：

1. 安装 [VirtualBox](#)(安装即可，不需额外操作)；
2. 下载安装[vagrant](#)；
3. 双击或者运行(`init.cmd`)自动配置

不同系统请参考：

- [Windows下虚拟机环境安装](#)
- [Linux 和 Mac下虚拟机环境安装](#)

II. 本机开发环境

如果你希望在本机上进行开发,你需要安装PHP并配置好必要的扩展即可。(不同人配置相同的环境总能有意想不到的问题)

从头开始配置可以按照以下方式进行

1. 下载或者编译PHP
2. 下载或者编译YAF扩展
3. 双击或者运行(`init.cmd`)配置

具体针对不同操作系统可以参照

- [Windows下配置YYF开发环境](#)
- [Linux下配置YYF开发环境](#)
- [\[Mac下配置YYF环境\]](#)

在现有基础上配置(已有 **WAMP** 或者 **LAMP** 之类的环境)

如果你已经有有PHP的web服务运行环境。可以直接clone到 或者下载YYF的源码解压到 web目录下即可。

1. 如果打开网站出现下面错误，说明没有安装yaf扩展

```
PHP Fatal error: Class 'Yaf_Application' not found in XXX/public/index.php
```

可以安装如下方法配置，或者搜索引擎搜索 如何配置yaf扩展

- [windows 配置yaf方法](#)
- [linux 配置yaf方法](#)
- 如果出现目录不可写的提示

运行 `init.cmd` 即可,(windows直接双击即可)

配置完成

配置完成打开你的测试页面和 yyf.yunyin.org一样说明基本配置成功

如果遇到问题，可以[google](#)，[百度](#)，或者在[github](#)上留言，如果文档有疏漏之处，[欢迎修改](#)。

Copyright © NewFuture 2016 all right reserved, powered by Gitbook文档编译时间: 2017-02-21 11:03:42

Windows安装 YYF 虚拟机

1. 下载安装最新版 [virtualbox](#)
2. 下载安装最新版 [vagrant](#)
3. 双击 `init.cmd` 开始自动安装配置

Copyright © NewFuture 2016 all right reserved, powered by Gitbook文档编译时间: 2017-02-21 11:03:42

Linux 安装 YYF 虚拟机

1. 安装[virtualbox](#) 和 [vagrant](#)
2. [clone YYF源码](#) [init](#)

Ubuntu 一键配置脚本 `curl -#L https://yyf.newfuture.cc/assets/code/vm-ubuntu.sh |bash`

1. 安装virtualbox和vagrant

Ubuntu 上安装

一行命令即可

```
sudo apt install -y virtualbox vagrant git
```

Centos 上安装

```
# virtualbox
sudo curl http://download.virtualbox.org/virtualbox/rpm/rhel/virtualbox.repo -o /etc/yum.repos.d/virtualbox.repo
sudo yum -y update
sudo yum -y install VirtualBox-5.1 git
# vagrant
curl https://releases.hashicorp.com/vagrant/1.8.5/vagrant_1.8.5_x86_64.rpm -o vagrant.rpm
sudo yum -y localinstall vagrant.rpm
```

2. clone 初始化yyf

clone源码

```
git clone https://github.com/YunYinORG/YYF.git
```

初始化环境

```
./YYF/init.cmd
```

正常情况一路回车即可(首次会自动下载一个350M的镜像)

Copyright © NewFuture 2016 all right reserved, powered by Gitbook文档编译时间: 2017-02-21 11:03:42

Windows 上配置YYF环境

1. 下载PHP
2. 配置YAF
3. 下载配置YYF

1. 下载PHP

如果已有PHP或者安装了 `WAMP` ,可以跳过此步骤。

Windows版PHP下载地址: <http://windows.php.net/download>. 选择对应版本下载解压即可。

2. 配置YAF

YAF下载地址: <https://pecl.php.net/package/yaf>

对照PHP版本以及下载对应YAF, `dll`文件。将`dll`文件放入相应PHP目录的`ext`文件夹下

并在 `php.ini` 文件(位于PHP目录,如果没有将 `php.ini-development` 改成 `php.ini`)中加入

```
extension = php_yaf.dll

yaf.envIRON = dev;开发环境是dev,服务器生产环境使用production
```

3. 下载YYF

1.clone <https://github.com/YunYinORG/YYF.git> 或者下载zip解压

2.双击 `init.cmd`

正常情况,配置和清理完成后会出现如下选项:

```
select which development environment you want to use?

1) Use virtual Machine with vagrant;
2) Use local development (with PHP);
0) Exit;

Input your choice (default[ENTER] is 1):
```

输入 `2` 回车(选择本地开发环境).

如果`php`在系统目录下会自动创建启动脚本,否则需要 输入PHP的路径 (可以直接将`php.exe`拖拽到命令行:)

即可配置和启动PHP测试服务器。

3.快速启动脚本

初始化完成后会自动生成 `server.cmd` ,以后只需运行此脚本即可快速启用`php`测试服务器。

(当然如果使用 `apache` 或者 `IIS` 等作为服务器可以将,YYF 放于web根目录下即可。)

4.测试服务

完成后浏览器打开 localhost:1122 如果显示类似与<https://yyf.yunyin.org/>就配置OK了！

Copyright © NewFuture 2016 all right reserved, powered by Gitbook文档编译时间: 2017-02-21 11:03:42

Linux 上配置YYF环境

以ubuntu 16.04 为例； 包括三个步骤:

1. 安装PHP
2. 编译配置YAF
3. clone YYF源码运行

1. 安装PHP

安装PHP和必要扩展(其中php-dev和gcc是编译yaf所需).扩展名称在不同系统名称可能有所不同

```
sudo apt install -y php php-mcrypt php-curl php-pdo-sqlite php-pdo-mysql php-dev gcc
```

2. 编译和配置yaf

自动编译(支持不同系统和PHP版本,自动切换)

复制运行下面命令,自动安装脚本YAF的并配置PHP开发环境(dev).(需要已安装gcc和php-dev,否则会报错)。
如果权限不够会自动切换到 `sudo`

```
curl -#L https://yyf.newfuture.cc/assets/code/yaf.dev.sh |bash
```

手动编译

可以在<https://pecl.php.net/package/yaf>选择最新稳定版yaf编译. (php7使用yaf 3.x版本,php5使用 2.x版本)

- Ubuntu 16.04默认使用PHP7, 使用 **yaf-3.0.3** 为例,可以根据需要换成对应版本号
- Ubuntu 16.04默认PHP扩展配置路径 `/etc/php/7.0/cli/conf.d/` 其他系统不一样, 最后

```
#下载YAF,不同版本
curl https://pecl.php.net/get/yaf-3.0.3.tgz |tar zx -C ~/
#编译yaf
cd ~/yaf-3.0.3/; phpize;./configure && make
# 安装yaf
sudo make install
#添加yaf.ini到PHP配置中, 不同系统路径不同
sudo sh -c "echo 'extension=yaf.so\n[yaf]\nyaf.envirom=dev'>/etc/php/7.0/cli/conf.d/yaf.i
ni"
```

3. clone YYF源码和运行

1.clone最新代码到工作目录,当然也可以直接下载zip解压

```
git clone https://github.com/YunYinORG/YYF.git
```

2.执行 `init.cmd` ,配置环境

切换到项目目录执行

```
./init.cmd
```

正常情况,配置和清理完成后会出现如下选项:

```
select which development environment you want to use?
```

- 1) Use virtual Machine with vagrant; [自动配置虚拟机环境]
- 2) Use php server (local development); [安装配置本机PHP开发环境]
- 3) install yaf with DEV environ (local); [只安装YAF并设置为开发环境]
- 4) install yaf with PRODUCT environ (server); [安装YAF设置生产环境]
- 0) Exit (Manual); [退出(手动配置)]

```
Input your choice (default[ENTER] is 1):
```

输入 `2` 回车(选择本地开发环境)即可配置和启动PHP测试服务器。

3.快速启动脚本

初始化完成后会自动生成 `server.cmd` ,以后只需运行此脚本即可快速启用php测试服务器。

当然如果使用 `apache` 或者 `nginx` 作为服务器可以将web根目录设置为 `项目目录/public/` 即可

4.测试服务

完成后浏览器打开 127.0.0.1:1122 如果显示类似与<https://yyf.yunyin.org/>就配置OK了!

Copyright © NewFuture 2016 all right reserved, powered by Gitbook文档编译时间: 2017-02-21 11:03:42

使用Docker运行YYF

YYF 提供了基于集成运行环境的超小docker镜像，

如果使用docker或者系统支持docker可用采用此方法快速部署开发环境。

- [最小运行环境\(12M\)](#)
- [完整YYF环境\(60M\)](#)

1. 最小YAF的docker环境

最小YAF的docker环境仅集成了YAF和php的基本环境，如果不包含数据等，可用使用此镜像开发或者演示。

基本的YAFdocker仅包含PHP和必要的扩展:

- YAF
- redis
- memcached
- PDO-*
- mcrypt
- curl
- gd

体积: 仅12M  

使用方式: 在工程目录下运行

```
sudo docker run -it -p 1122:80 -v "$(pwd)":/newfuture/yaf newfuture/yaf
```

更多"食用方式"参看仓库地址: <https://github.com/NewFuture/YAF-docker>

2. 完整YYF开发环境

完整YYF docker除了上述的php环境，还集成下列服务了:

- redis
- memcached
- sqlite
- mariadb 和 mariadb-client

体积: 大约60M  

使用方式: 在项目目录下运行

```
sudo docker run -it --rm -p 1122:80 -v "$(pwd)":/yyf newfuture/yyf
```

更多"食用方式"参看仓库地址: <https://github.com/NewFuture/YYF-docker>

服务器上配置YYF生产环境

"一键部署"

- **Centos (rpm系列):** `curl -#L https://yyf.newfuture.cc/assets/code/server-centos.sh | bash`
- **Ubuntu (deb系列):** `curl -#L https://yyf.newfuture.cc/assets/code/server-ubuntu.sh | bash`

对于“裸机”可以直接选择对应的系统运行后面的命令自动安装和配置(包括 apache,php,mysql或mariadb,和yaf扩展)。

注意: 服务器上使用生产环境(product)配置,同时配置文件会一直缓存在内存中(更新配置需重启PHP进程)。

1. Centos 上默认配置

```
curl -#L https://yyf.newfuture.cc/assets/code/server-centos.sh | bash
```

使用系统默源进行安装httpd和php和数据库,不同版本系统安装的结果会不一样。

组件	centos 6.x	centos 7.x	
httpd(apache)	2.2	2.4	
PHP 版本	<=5.4	5.5 或5.6	
数据库	mysql	mariadb	

如果代码有误[可以在GITHUB上修改](#)

```
#!/usr/bin/env bash

PROJECT_PATH="/var/www/YYF"
TEMP_PATH="/tmp/"
CONF_PATH="/etc/httpd/conf/httpd.conf"

# sudo yum -y update

#####
###[LAMP]
### 安装 apache php mysql
#####

# install httpd
# 安装 apache php gcc和git
sudo yum install -y httpd git \
    php php-opcache php-pdo_mysql php-mcrypt php-mbstring php-curl \
    php-devel gcc

# 安装mysql或者mariadb 会二者选一
sudo yum install -y mysql-server mariadb-server
```

```
#####
###[YAF_EXTENTSION]
### 安装 yaf
#####
#判断yaf版本和php版本
#check the version of php and yaf
PHP_VERSION=$(php -v|grep --only-matching --perl-regexp "\d\.\d+\.\d+"|head -1);
if [[ ${PHP_VERSION} == "7.*" ]]; then
    #php 7
    YAF_VERSION=yaf-3.0.3
else
    #php 5
    YAF_VERSION=yaf-2.3.5
fi;
# 下载解压yaf
# download yaf
curl https://pecl.php.net/get/${YAF_VERSION}.tgz | tar zx -C $TEMP_PATH
# 编译安装 YAF
# compile and install YAF
cd ${TEMP_PATH}/${YAF_VERSION} && phpize;
./configure && make && sudo make install

# 配置yaf(product 环境)
# configure yaf with product environment
sudo tee /etc/php.d/yaf.ini> /dev/null <<EOF
extension=yaf.so
[yaf]
yaf.envron=product
yaf.cache_config = 1
EOF

# configure the apache(httpd)
# 配置apache开机启动
sudo systemctl start httpd.service
sudo systemctl enable httpd
#防火墙允许httpd 部分系统有效
sudo firewall-cmd --permanent --add-service=http

# httpd webroot
# 配置 apache 根目录
sudo cp $CONF_PATH ${CONF_PATH}.back
sudo sed -i.back -e "s|\"/var/www/html\"|\"${PROJECT_PATH}/public\"|g" $CONF_PATH

# clone YYF and initialize
# clone 代码 初始化
if [ ! -f $PROJECT_PATH ]; then
    sudo mkdir -p ${PROJECT_PATH}
fi;

sudo chown $UID ${PROJECT_PATH}

git clone https://github.com/YunYinORG/YYF.git ${PROJECT_PATH}
```

```
echo 0 | ${PROJECT_PATH}/init.cmd
```

```
#重启apache服务器  
#restart apache  
sudo service httpd restart
```

2. Ubuntu 上默认配置

```
curl -#L https://yyf.newfuture.cc/assets/code/server-ubuntu.sh |bash
```

使用系统默源进行安装apache,mysql和PHP (ubuntu 16.04及以上会自动安装php7)

如果代码有误可以在[GITHUB](#)上修改

```
#!/usr/bin/env bash

PROJECT_PATH="/var/www/YYF"
TEMP_PATH="/tmp/"

echo " UPDATE..."
sudo apt update
#&>${TEMP_PATH}/yyf_install.log

#####
###[LAMP]
### 安装 apache php
#####

#apache
echo "INSTALL apache"
sudo apt-get -y install apache2 gcc git &>>${TEMP_PATH}/yyf_install.log

echo "INSTALL php"
#php7
sudo apt-get -y install php php-mcrypt php-curl php-pdo-sqlite php-pdo-mysql php-dev liba
pache2-mod-php &>>${TEMP_PATH}/yyf_install.log
#php5
sudo apt-get -y install php5 php5-mcrypt php5-curl php5-sqlite php5-mysql php5-dev libpcr
e3-dev &>>${TEMP_PATH}/yyf_install.log

# httpd webroot
sudo tee /etc/apache2/sites-available/yyf.conf> /dev/null <<EOF
DocumentRoot "${PROJECT_PATH}/public"
<Directory "${PROJECT_PATH}/public">
Options FollowSymLinks
AllowOverride all
Require all granted
</Directory>
EOF
```

```

sudo a2ensite yyf.conf
sudo a2dissite 000-default.conf
sudo a2enmod php*
sudo a2enmod rewrite

#####
###[YAF_EXTENSION]
#####
# check PHP version
PHP_VERSION=$(php -v|grep --only-matching --perl-regexp "\d\.\d+\.\d+"|head -1);
if [[ ${PHP_VERSION} == "7.*" ]]; then
    #php 7
    YAF_VERSION=yaf-3.0.3
else
    #php 5
    YAF_VERSION=yaf-2.3.5
fi;

# download yaf
# 下载解压yaf
curl https://pecl.php.net/get/${YAF_VERSION}.tgz | tar zx -C $TEMP_PATH
# 编译安装 YAF
# compile and install YAF
cd $TEMP_PATH${YAF_VERSION} && phpize
./configure && make && sudo make install

## 创建yaf配置文件(product 环境)
## create temp yaf config with product environment
cat <<EOF>${TEMP_PATH}yaf.ini
extension=yaf.so
[yaf]
yaf.envIRON=product
yaf.cache_config = 1
EOF

# 获取 PHP ini 配置目录
# Scan for additional .ini path
PHP_INI_PATH=$(php --ini|grep --only-matching --perl-regexp "/.*\.d$")
PHP_INI_PATH=$(echo $PHP_INI_PATH | sed -r -e 's/cli/*/')
# 复制配置文件到各个目录
# cp the yaf configure to each file
echo $PHP_INI_PATH | xargs -n 1 sudo cp $TEMP_PATH/yaf.ini
# 删除临时文件
# remove temp ini
rm ${TEMP_PATH}yaf.ini

#####
###[YYF]
### 下载YYF
#####
# clone YYF and initialize
# clone 代码 初始化

```



```

if [ ! -f $PROJECT_PATH ]; then
    sudo mkdir -p ${PROJECT_PATH}
fi;

sudo chown $UID ${PROJECT_PATH}
git clone https://github.com/YunYinORG/YYF.git ${PROJECT_PATH}
echo 0 | ${PROJECT_PATH}/init.cmd

#重启apache服务器
#restart apache
sudo service apache2 restart

MYSQL_SERVER=$(dpkg -l | grep -c "mysql-server")
if [ ${MYSQL_SERVER} -gt 1 ] ;then
    echo "mysql-server was installed"
else
    echo "INSTALL mysql-server"
    # 静默安装mysql, 不显示密码框
    echo mysql-server mysql-server/root_password password | sudo debconf-set-selections
    echo mysql-server mysql-server/root_password_again password | sudo debconf-set-select
ions
    sudo apt install -y mysql-server
fi;

```

YYF 文件目录结构

主要内容

1. 整体目录文件结构
2. [public](#)(网站根目录)
3. [运行时目录](#)(数据存储位置)
4. [app](#)应用目录

整体目录文件结构

```
| .htaccess      Apache开发环境和SAE重定向url
| .travis.yml    travis-ci测试配置
| init.cmd       开发环境初始化通用脚本
| LICENSE        Apache 2.0 许可证
| README.MD
|
|─app
| |
| |   README.MD
| |
| |   └─controllers      控制器目录【添加代码的主战场】
| |       Error.php      默认错误
| |       Index.php      DEMO控制器
| |
| |   └─email            邮件模板目录
| |       verify.tpl     默认验证邮件模板示例
| |
| |   └─models           数据模型目录
| |       README.md
| |
| |   └─views            视图目录
| |       └─index
| |           index.phtml
| |
| └─conf              配置目录
|     app.ini          基础配置
|     secret.common.ini 示例私密配置
|     secret.product.ini 生产环境私密配置
|
|─library            库目录
| |   Cache.php        缓存管理类
| |   Config.php        配置读取类
| |   Cookie.php        安全Cookie接口
| |   Db.php            数据库操作封装
| |   Debug.php         调试类
| |   Encrypt.php       加密库
| |   Input.php         输入过滤接口
| |   Kv.php            key-value存取类
```

```
| | Logger.php    日志管理类
| | Mail.php     邮件发送
| | Model.php    基础model
| | Orm.php      ORM数据库对象映射
| | Random.php   随机字符生成类
| | README.md
| | Rest.php     基础REST类
| | Rsa.php      RSA加密类
| | Safe.php     安全统计类
| | Session.php  session管理接口
| | Validate.php 类型验证类
| | Wechat.php   微信登录接口库类
|
|
| | —Bootstrap 启动加载
| |     dev.php      开发环境启动加载
| |     product.php  生产环境启动加载
|
| | —Debug 调试相关库(开发环境)
| |     Assertion.php 断言处理类
| |     Header.php    header头输出类
| |     Listener.php   日志监听类
| |     Tracer.php     消耗统计类
|
| | —Parse 格式解析
| |     Filter.php
| |     Xml.php
|
| | —Service 系统基础服务
| |     Api.php
| |     Database.php
| |     Message.php
| |     Qiniu.php
| |     README.MD
| |     Smtplib.php
| |     Ucpaas.php
|
| | —Storage 存储驱动
| |     File.php      文件缓存类
|
| | —Test 单元测试库
| |     YafTest.php   Yaf框架测试基类
|
| —public 公共目录【前端资源目录，生产环境根目录】
|     .htaccess      url重写
|     favicon.ico
|     index.php      入口文件
|     robots.txt
|
| —runtime 默认缓存日志临时文件夹【保证程序具有可读写权限】
|
|
| —tests 单元测试目录
```

网站根目录

`public` 前端目录(用户唯一可以访问的目录)

- 前端资源目录：静态资源css,js等放置于此目录
- `web`根目录：生产环境时作为网站的根目录

运行时目录

`runtime` 文件缓存等数据会存于此目录，保证程序对目录可读写权限；

可以配置 `conf/app.ini` 中 `runtime` 指向系统的其他位置。

注意生产环境生成的存储文件会设置为**700**权限，保证安全性。

应用目录

`app` 应用

多模块是添加到 `app/modules/` 目录下

如添加一个admin目录 `app/modules/admin/controllers/Index.php`

Copyright © NewFuture 2016 all right reserved, powered by Gitbook文档编译时间： 2017-02-21 11:03:42

MVC架构

主要内容

1. MVC框架
2. 控制器Controller
3. 数据模型Model
4. 视图模板View

MVC框架

WEB开发框架中，通常分为MVC三层。

- Model(M)数据模型层，对数据库操作进行封装;
- View(V)视图层,对数据渲染输出页面;
- Controller(C)是核心逻辑控制层.

在YYF框架中：(app下三个字目录) **C** 控制层(controller/)是必须，处理不同请求,同时对REST做了相应映射; **M** 数据模型层(models/),数据库底层核心库中，已做了简单封装; **V** 视图模板层(views/),通常省略，因为API通常输出是JSON字符串。

控制器Controller

控制器是最核心和主要的部分，文件存放于 `controllers` 目录下，一个Controller又包含多个Action。每个请求会映射到一个对应的Action，通常一个Action对应controller中的一个方法(method)。

在YYF中，需要从基类集成而来，可以选择从 `REST` (YYF的REST控制器)，或从最底层 `Yaf_Controller_Abstract` (Yaf的控制器基类)继承而来。class名以controller为后缀结束,文件名不包含(controller)。

例如一个请求的URI `/Abc/xyz` 对应控制器 `AbcController` 的方法 `xyzAction()`

```
<?
/**
 * 文件名 app/controllers/Abc.php
 */
class AbcController extends Rest
{
    function xyzAction(){
        $this->success('response success');
    }
}
```

通过 `REST` 控制器不仅可以每个URI对应到不同的action方法上,还可以将不同的请求方式 (`GET` , `POST` 等)映射到action上 如 `GET_xyzAction()` 对应GET请求, `PUT_xyzAction()` 对应PUT请求。

数据模型Model

数据模型对数据库操作进行封装,yaf中没有提供对数据库的封装,YYF对常用的MySQL和SQLite进行了安全简单的封装(ORM),并提供了一致的操作接口。详细接口参见[Orm](#)和[Model](#)

YYF中提供两种方式处理数据模型

- 简单高效的, 动态创建数据模型(Orm)使用 `new Orm('name')` 或者 `Db::table('name')`
- 优雅自定义, 使用自定义Model继承自 `Model` 并保存于 `app/models/` 目录下。

视图模板View

YYF中默认是关闭了视图渲染, 因为作为API接口通常返回的数据是JSON等纯数据格式, 不需要视图渲染。

而YAF中提供了最简单的模板, 使用最原始的php语法解析(也是最高效的解析方式), 存储文件为phtml。文件名如下可自动对应 `app/views/控制器/Action.phtml`

Copyright © NewFuture 2016 all right reserved, powered by Gitbook文档编译时间: 2017-02-21 11:03:42

RESTful 风格设计

RESTful API 设计用不同的URI地址表示不同的资源(通常是名词)，用不同的请求方式表示不同的操作。

请求方式

REST风格设计中，通常使用不同的请求方式对应不同的操作。

常见的请求方式:

- **GET** 获取或者查询
- **POST** 创建或者新增
- **PUT** 修改内容
- **DELETE** 删除(delete通常无额外参数，因此不会对参数进行解析)
- 其他(如 **PATCH** 等)

YYF 中可以将不同的请求方式映射到不同的Action上进行处理

URI地址

RESTful URI表示资源

实际中常用的表示方式: 资源组/id标识[/属性] (如 `User/1/address`)或者 资源组/属性 (如 `Setting/banner`)

YYF框架中对id(数字)映射做了特殊处理 `Controller/:id/Action` 会映射到Controller控制器的Action方法上并绑定参数 `:id` 到 `$id`

RESTful 请求举例

以对项目资源(Project)的操作为例

```
#获取项目列表(ProjectController::GET_indexAction())
GET /Project/

#创建项目(ProjectController::POST_indexAction())
POST /Project/

#获取项目id为1的详细信息(ProjectController::GET_infoAction($id))
GET /Project/1
#修改id为1的项目内容(ProjectController::PUT_infoAction($id))
PUT /Project/1
#删除id为1的项目(ProjectController::DELETE_infoAction($id))
DELETE /Project/1
#获取项目id为1的贡献者(ProjectController::GET_contributorsAction($id))
GET /Project/1/contributors
```

Config 配置读取

提供一个高效的数据读取接口

接口列表

- `Config::get($key,$default)` 读取当前应用配置
- `Config::getSecret($name,$key)` 读取私密配置

get 获取配置

`get`方法用于快速读取 `conf/app.ini` 中的配置(当前如果)。

```
function get(string $key [, mixed $default=null]):mixed
```

- 参数:
 1. `string $key` : 获取的键值
 2. `mixed $default` (可选): 默认值, 如果读取的值不存在返回此值(默认为null)
- 返回: `mixed` , 位设置默认值时字符串或Object或者null
 - `objec(Yaf_Config_Ini)`:如果是多级配置,返回配置只读, 可以使用 `toArray` 转为array
 - `string` :如果是最后一项(完整的键) 数字等配置也是string;
- tips: 如果要转成可写的数组可以使用 `toArray` 创建一个数组副本;

```
//获取配置
Config::get('version');

//设置默认值
Config::get('log.type', 'file');

//多级参数
Config::get('application')->num_param;

//数组方式
Config::get('cors')['Access-Control-Allow-Origin'];

//转换数组
$cors=Config::get('cors')->toArray();
```

getSecret 获取配置

`getSecret`方法用于快速读取 `conf/secret.comment.ini` (生产环境为 `conf/secret.product.ini`)中的配置。

```
function getSecret(string $name [, mixed $key=null]):mixed
```


- 参数:
 1. `string $name` : 配置项如 `database` , `wechat`
 2. `string $key` (可选): 默认值读取的字段,默认返回整个配置对象
- 返回: `mixed` , 位设置默认值时字符串或Object或者null
 - `objec(Yaf_Config_Ini)`:如果是多级配置,返回配置只读, 可以使用 `toArray` 转为array
 - `string` :如果是最后一项(完整的键) 数字等配置也是string;
- tips: 如果要转成可写的数组可以使用 `toArray` 创建一个数组副本;

```
//获取配置项
Config::getSecret('wechat');

//获取值
Config::getSecret('database','prefix');
```

数据库（Database）

数据库是MVC 中 M层处理的核心业务。

整个框架中数据库这一层提供致力于提供一个安全,高效,简单的数据操作接口。

数据库设计命名基本规范

数据设计如果满足前三条，在使用模型和关联时大部分细节时候可以自动完成

1. 数据库表名小写下划线(或者全部小写)如： `user` , `user_info` , `admin_log` (推荐下划线)或者 `amdinlog`
2. 数据库表的主键: `id` (建议所有表均设置一个自增主键)
3. 数据库表的外键 `${table}_id` 如:信息表 `info` 有个用户表(`user`)的外键,则信息表中的外键为 `user_id` ,

其他无特殊要求根据团队习惯尽量保持一致即可。

如果不满可以通过配置和参数达到同样目的。

数据库配置

数据库配置在 `secret` 配置文件开发环境和生产环境使用不同的配置文件

```
[database]
;数据库配置
prefix      = '';数据库表前缀
exception = 0 ;sql执行出错是否抛出异常，可以try catch

;默认数据库(主库)
db._.dsn      = "mysql:host=localhost;port=3306;dbname=yyf;charset=utf8"
db._.username = 'root'
db._.password = ''
;读数据库(从库)
db._read.dsn      = "sqlite:/temp/databases/mydb.sq3"; 以sqlite配置为例
db._read.username = 'username'
db._read.password = ''
```

需要添加数据库是在 `db` 后继续追加：

- `db.{name}.dsn` (数据库DSN);
- `db.{name}.username` (数据库账号 可选)
- `db.{name}.password` (数据库密码可选)

其中 `{name}` 为数据库配置名称

数据库相关类库

快捷辅助类

- [Db](#) 数据库操作管理类
- [Model](#) 数据模型封装

核心类

- [Database](#) 数据库连接类
- [Orm](#) 数据库对象关系映射类

Copyright © NewFuture 2016 all right reserved, powered by Gitbook文档编译时间: 2017-02-21 11:03:42

数据库使用示例

SQL语句查询

```
//1
Db::query('SELECT * FROM user WHERE id=?',[1]);
//2
Db::execute('INSERT INTO user (name,org) VALUES (?, ?)', ['future', 'NKU']);
/*等效操作*/
//3
Db::execute('INSERT INTO user (name,org) VALUES (:name, :org)', ['name'=>'future', 'org'=>'NKU']);
```

数据库对象映射

```
/*2, 3, 4, 5, 6操作等效*/
//4
Db::table('user')->insert(['name'=>'future', 'org'=>'NKU']);
//5
Db::table('user')->set(['name'=>'future', 'org'=>'NKU'])->add();

/*插入时使用别名*/
//6
$data=['username'=>'future', 'organization'=>'NKU'];
$user->field('name', 'username')
    ->field('org', 'organization')
    ->insert($data);

/*字段过滤和批量插入*/
//7
$data=[
    ['name'=>'test'],
    ['name'=>'test2'],
    ['id'=>1000, 'name'=>'test3'],
    ['password'=>'xyz', 'name'=>'test4'],
    ['id'=>123]
];
$response = UserModel::field('name')->insertAll($data);

/*查询*/
//7
$response = Db::table('user')
    ->where('id', 'BETWEEN', 5, 8)
    ->select('id, name');

//8
$user = Db::table('user')->find(1);
```

```

//9
$username = Db::table('user')->where('id',1)->get('name');
//等效操作
$username = Db::table('user')->find(1)->name;

//10
$response= Db::table('user')
    ->where('name', 'NOT LIKE', '%future')
    ->where('id', 'in', [100,10,1])
    ->select();

//11
$response= Db::table('user')
    ->where(['name'=>'future'])
    ->where('id', '>', 10)
    ->limit(5);
    ->select();

//12
Db::table('user')
    ->where()
    ->page(3,5)
    ->select();

/*修改*/
//13
Db::table('user')->where('id',1)->update(['name'=>'first user']);

//14
if($user = Db::table('user')->find(2))
{
    $user->field('name')->set('name', 'second user')->save();
}

/*删除*/

Db::table('user')->delete(10);
//等效
Db::table('user')->where('id', '=', 10)->delete();

```

使用model

定义如下model `app/models/User.php`

```
<?php class UserModel extends Model{}
```

```
UserModel::set('name','yyf')->add();  
  
$user=UserModel::find(1);  
  
UserModel::page(2,10)->select('id,name');
```

Db 数据库辅助类

Db 类提供封装和简化了数据库相关操作的调用，提高简单的静态调用接口。

接口和方法列表

- 常用方法接口
 - `current()` 获取当前数据库连接
 - `connect()` 连接数据库
 - `set()` 设定数据库
 - `table()` 快速创建数据库表
 - `query()` 查询sql语句
 - `exec()` 执行sql命令
 - `execute()` `exec` 别名
- Database接口调用
 - `Database::errorInfo()` 获取出错信息
 - `Database::isOk()` 上次查询是否出错
 - `Database::transact($func)` 执行事务
- 继承自PDO(下面链接均为PHP文档)
 - `PDO::beginTransaction()` — 启动一个事务
 - `PDO::commit()` — 提交一个事务
 - `PDO::rollBack()` — 回滚一个事务
 - `PDO::lastInsertId()` — 返回最后插入行的ID或序列值
 - `PDO::prepare($sql)` — 查询预处理
 - `PDO::setAttribute($key,$value)` — 设置属性
 - 其他全部的PDO接口

快速创建ORM对象

可以通过 `table` 方法创建一个Orm对象，映射到数据库表,快速查询。`table`方法返回的是一个 `Orm` 对象

`table()` 方法快速查询

```
function table(string $name [,string $pk, [string $prefix]]):Orm
```

- 参数:
 1. `string $name` [必填]:数据库表名
 2. `string $pk` [选填]: 主键值，默认orm而定为 `id`
 3. `string $prefix` [选填]: 前缀，默认读取配置
- 返回: `Orm` 对象
- 如果需要对同一张表进行多次操作,可以创建赋值给一个变量，每次调用此变量提高性能

```
//快速查询
$name=Db::table('user')->where('id',2)->get('name');

//添加数据使用field进行字段过滤和别名设置
Db::table('feedback')
->field('user','name')
->field('content,call AS phone')
->insert($_POST);//快速插入数据,只是示例,对于写入用户数据进行检查是必要的
```

数据库管理

Db 类会根据需要自动创建数据库连接，并在生命周期类保存这些了连接资源。单数据库默认不需要使用数据库管理和切换操作。

Db 操作同时提供多数据库操作接口。

Db 默认会自动使用配置中的默认的数据库进行调用，不会自动的进行读写分离或者切换。

Db切换数据库不会影响**Model**或者**Orm**中数据库的调用，`table()` 方法使用的数据会按照[Orm数据库切换规则](#)切换。

current() 方法:获取当前数据库

`current` 返回当前正在使用的数据库对象

```
function current() :Database
```

- 返回当前数据连接对象
- 示例代码

```
$db=Db::current();
//切换数据库，等各种操作
//继续原来的$db
$db->query('some thing');
```

set() 方法：设定并切换数据库

`set` 方法手动切换设置数据库。

此方法不会影响直接查询对数据库的选择，但是会影响

注意：如果修改保留名称，此方法会影响 [Model](#)和[Orm](#)中默认读写数据库的调用。

```
function $name(string $name,mixed $config,[string $username,[string $password]]) :Database
```

- 参数 `$name` (string): 数据库配置名称如果无则创建，有则覆盖
- 参数 `$config` (必填): 可以是下列三项之一
 - string 数据库 配置名称 如 "_read","mydb",只要在[database] 下配置了即可;

- string dsn设置 如: "sqlite:/tmp/sql.db";
- array 数据库链接配置, 包括dsn, username, password;
- 参数 `$username` string: 数据库账号 (当\$cofnig为dsn时选填)
- 参数 `$password` string: 数据库密码 (当\$cofnig为dsn时选填)
- tips :
 - 当 `$name` 为 `_` 会修改 `Db` , `Model`和`Orm` 等数据库操作使用的默认数据库
 - 当 `$name` 为 `_read` 会修改 `Db` , `Model`和`Orm` 等数据库读取操作的数据库
 - 当 `$name` 为 `_write` 会修改 `Db` , `Model`和`Orm` 等数据库写入操作的数据库
- 返回: 数据库对象
- 示例代码

```
/*配置名称切换数据库, '_','_read','_write'三个是保留数据库名*/
Db::set('_', '_read')->query('query something');//切换到读数据库
Db::set('_write', 'mydb')->exec($sql);//执行

Db::exec($sql2);//此时仍然使用mydb写

/*dsn*/
Db::set('temp', 'sqlite:/tmp/sql.db')->query($sql);
/*多参数设置*/
Db::set('_write', 'mysql:host=localhost;port=3306;dbname=yyf;charset=utf8', 'root', 'root');
/*array*/
Db::set('test', [
    'dsn'=>'mysql:host=localhost;port=3306;dbname=yyf;charset=utf8',
    'username'=>'root'
]);
```

connect() 方法: 建立数据库

`connect` 方法建立数据库, 而不影响之后或者其他的数据调用

注意: 此方法为临时调用, 不会影响之后数据库切换

```
function connect(mixed $config):Database
```

- 参数 `$config` (必填): 可以是下列两项之一
 - string 数据库 配置名称 如 `"_read"`, `"mydb"`, 只要在`[database]` 下配置了即可;
 - array 数据库链接配置, 包括dsn, username, password;
- 返回: 数据库`Database`对象
- tips: 此方法不会影响之后数据库调用使用的数据库连接
- 示例代码

```

/*配置名称连接*/
Db::use('_read')->query('query something');//切换到读数据库

/*新建一个数据库连接*/
Db::connect('mysql:host=localhost;port=3306;dbname=yyf;charset=utf8','root','root');
/*array方式连接*/
Db::connect([
    'dsn'=>'mysql:host=localhost;port=3306;dbname=yyf;charset=utf8',
    'username'=>'root'
])->exec($sql);

//使用默认数据库
Db::query($sql);//此时任然是_read数据库

```

sql语句查询

Db 根据需要自动建立数据库连接,可以调用 `Database` 的所有方法接口

可以直接执行或者调用sql语句.但是对于新手或者对安全不太了解的,不推荐这么使用,因为这样容易造成潜在的SQL注入风险,如果确实要这么做,务必使用参数分离的方式进行查询。

同时,常用的数据方法使用静态方式加速,并自动读写分离。

query() 查询sql语句(读):

数据库读取(select)查询,是对 `Database::query()` 的快速调用。

示例代码

```

//原生sql查询
$list=Db::query('select id,name from user');
//键值对参数分离
$data=Db::query('select * from user where id=:id',['id'=>2]);

```

column() 查询sql语句(读):

数据库单条读取(select)查询,是对 `Database::column()` 的快速调用。

示例代码

```

//键值对参数分离
$name=Db::column('select name from user where id=?',['123']);

```

execute() 执行sql语句:

`execute` 是 `exec` 的别名,用来覆盖Db方法的私有方法`execute`。

建议尽量 `exec` 来调用,因为 `Database` 提供了 `exec` 接口而不是`execute`。

如 `Db::execute($sql)` 可以执行,等效 `Db::exec($sql)` ;

但是 `Db::current()->execute($sql)` 会出错,应该使用 ``Db::current()->exec($sql)` ;

`exec()` 执行sql语句(写):

数据库写操作执行, 是对 `Database::exec()` 的快速调用。

示例代码:

```
//执行sql语句, 参数绑定
Db::exec('UPDATE`user`SET(`time`=?)WHERE(`id`=?)',[date('Y-m-d h:i:s'),2]);
```

Database 静态方式调用

Db 类会根据需要自动创建数据库连接, 并在生命周期类保存这些了连接资源

支持所有 `Database` 接口

如预处理:

```
//预处理方式查询
Db::prepare('UPDATE`user`SET(`time`=?)WHERE(`id`=?)')
    ->execute([date('Y-m-d h:i:s'),2]);
```

如原生事务:

```
Db::beginTransaction();//开始事务
try{
    Db::exec('do something ...');
    Db::query('do something ...');
    Db::exec('do something ...');
    /*更多查询...*/
    Db::commit();//事务提交
} catch (Exception $e) {
    Db::rollBack();//出错回滚
}
```

注意:对于多数据库的情况,事务调用过程中不能切换数据库!

对ORM或者model的操作, 建议使用 `orm` 封装的事务操作 `transact()` ;

或者使用数据库 `transact` 方法。

Copyright © NewFuture 2016 all right reserved, powered by Gitbook文档编译时间: 2017-02-21 11:03:42

Model 数据库模型

数据库模型的核心,当需要在不同地方需要重复使用相同的数据设定时可以考虑创建 `Model` 。

`Model`在性能上会比直接创建 `Orm` 或者使用 `Db::table()` 心里要低,但是在代码易读性和重用上较好。

接口

属性接口

- `$name` 表面名
- `$pk` 主键
- `$fields` 字段
- `$prefix` 表前缀
- `$dbname` 限制数据库

方法接口

- `getOrm` 获取对应的ORM
- `ORM方法静态调用`

创建一个Model

一句话的Model

在目录 `app/models/` 下新建 `User.php` 文件中写入如下代码

```
<?php class UserModel extends Model{}
```

然后就可以在其他地方,如controller里面通过 `UserModel` 来调用了

```
/*快速调用*/
UserModel::Insert(['name'=>'future']);
$name=UserModel::where('id',2)->get('name');

/*实例化,可以接收一个数组参数预设数据,重复使用时可以clear*/
$user=new UserModel($data);
```

注意:上面的这种方式,依赖于数据库命名小写下划线链接和 `id` 作为主键的命名规范。

如果 数据库 中user表不是 `user` 而是 `User` (大写)甚至是 驼峰式命名 。这种方式需要制定数据表名称

设置Model的属性

name 表名

```
string $name
```

数据库表名，默认为当前class名称小写下划线链接，

如 `UserModel` 映射==> `user` 表， `UserInfoModel` 映射==> `user_info` 表

如果数据库不是按照此方式命名为，比如用户表命名为 `User` ，

```
<?php
class UserModel extends Model
{
    protected $name = 'User'; //数据库表
}
```

pk 主键

```
string $pk
```

默认主键为 `id` 。

如果主键为 `user_id` ，可以如下定义。

```
<?php
class UserModel extends Model
{
    protected $pk = 'user_id'; //主键
}
```

\$fields 字段设置

```
mixed $fields
```

字段，预知字段过滤或者别名设置，参见[Orm的field\(\)方法](#)。

写入时进行字段过滤或者读取时指定字段和别名。可以使用 `clear` 清除这个设置，或者使用 `field` 继续设置字段。

支持字符串(简洁)和数组(清晰)方式进行设置：

- 字符串(string): 使用 `,` 分割, `AS` 设置别名

```
protected $fields='id,name AS username';
```

- 数组: 可以使用键值对 `=>` 指定别名

```
protected $fields=[
    'id',
    'name'=>'username',
];
```

```

/* 定义在文件 app/models/User.php */
<?php
class UserModel extends Model
{
    protected $fields='id,name AS username';
}
?>

/*调用*/
$user=UserModel::find(1);//$user中包含字段`id`和`username`

$user_list=UserModel::limit(100)
    ->select();//选出100个用户的'id'和'username'(字段对应`name`)

UserModel::where('id',1)->update([
    'username'=>'future',//别名username会自动转成`name`字段
    'password'=>'hihi',//这个字段不在fields中会被过滤掉
]); //最终只有user表中'name'被改成future

UserModel::insert([
    'username'=>'future',//别名username会自动转成`name`字段
    'password'=>'hihi',//这个字段不在fields中会被过滤掉
]); //最终插入数据为 ['name'=>'future'];

```

prefix 前缀

```
string $prefix
```

默认主键使用配置中的prefix。

如果要覆盖掉可自行设置 prefix 设置为空串('')将不使用前缀。

```

<?php
class UserModel extends Model
{
    protected $prefix = 'yyf_'; //设置数据表前缀
}

```

dbname 数据库设定

```
string $dbname
```

使用[database]配置中db数据库的配置。

```
;若干设置
;conf/secret.common.ini
[database]
;数据库配置
;...若干设置
db.data.dsn = "sqlite:/temp/databases/data.db"; 以sqlite配置为例
db.data.username = ""
```

配置

```
<?php
class UserModel extends Model
{
    protected $dbname = 'data'; //使用配置名为`data`的数据库
}
```

Model的方法接口

这里的 `Model` 本质上是对 `Orm` 增强和再次封装的外观模式(Facade Design)

`toArray()` 获取Orm

获取Model中数据，以数组的方式返回

`toJson()` 获取Orm

```
function toJson($type=JSON_UNESCAPED_UNICODE):string
```

获取Model中数据，以json的方式返回,参数为编码方式

`getOrm()` 获取Orm

`getOrm` 返回当前model中的的ORM对象

所有 `Orm` 接口

`Model` 可以以静态和动态的方式调用所有Orm的接口。

Copyright © NewFuture 2016 all right reserved, powered by Gitbook文档编译时间: 2017-02-21 11:03:42

ORM 数据库操作对象

Object-Relational Mapping (对象关系映射)

数据库操作的对象核心封装

全部接口列表

- 查询操作
 - **select** 批量查询
 - **find** 单条查询
 - **get** 获取单条或单个属性
- 插入操作
 - **insert** 插入
 - **insertAll** 批量插入
 - **add** 添加
- 更新操作
 - **update** 更新数据
 - **save** 保持数据
 - **set** 修改属性
 - **put** 快速修改字段
 - **increment** 字段自增
 - **decrement** 字段自减
- **delete** 删除操作
- 条件限制
 - **where** 设置条件
 - **orWhere** 设置 或条件
 - **whereField** 字段关系
 - **orWhereField** 字段 或关系
 - **having** 计算条件
 - **orhaving** 计算OR条件
 - **exists** 子查询存在
 - **orExists** 不存在
- 结果筛选
 - **distinct** 去重
 - **group** 分组
 - **field** 字段过滤
 - **order** 排序
 - **limit** 限制数量
 - **page** 翻页
- 聚合函数
 - **count** 统计
 - **sum** 求和
 - **avg** 均值
 - **max** 最大值

- `min` 最小值
- 多表操作
 - `join` 接表
 - `has` 包含外键关联
 - `belongs` 从属外键关联
 - `union` 合并查询
 - `unionAll` 合并查询不去重
- 其他方法
 - `alias` 别名
 - `clear` 清空
 - `transact` 事务
 - `debug` 调试sql
 - `autoClear` 自动清空
 - `safe` 安全模式
 - `setDb` 切换数据库

创建Orm

有三种方式创建一个ORM对象，所有说明都是基于此对象说明

1. new创建

这是基本的创建方式如下

```
$orm=new Orm('user');//为user表创建一个orm对象,默认使用id作为主键
```

Orm 构造函数接受三个参数: `$table` , `$pk` , `$prefix`

- `$table` string : 数据库表名
- `[$pk]` string : 主键, 默认"id"
- `[$prefix]` string : 数据库表前缀,默认读取配置

```
$orm=new Orm('user','uid');//为user表创建一个orm对象,设定主键uid
```

2. Db类静态调用

Db类提供静态方法 `table` , 创建一个orm对象

```
$orm=new Db::table('user');//创建参数和Orm构造函数的一致
```

3. Model类创建

通过 `UserModel` 来直接调用, 参见 [Model一节](#)

基本操作

查询数据 (query)

读取数据提供 `select` , `find` , `get` 三种方法

`select()` 方法: 批量获取数据

```
array function select([string $fields=''])
```

- 参数 `$fields` [可选]: 指定查询的字段 逗号 , 分隔符, 别名用 `AS` 链接
- 返回 `array` 多维数组
- 示例代码

```
$list=$orm->select('*');//查询所有字段和所有数据  
$list=$orm->select('id AS uid,time');//查询id和time, 在返回的数据中id用uid表示
```

`find()` 方法: 单条数据读取

`find` 方法会自动限制数据的条数

```
object function find([mixed $id=null])
```

- 参数:
 - `int | string` : 数据的主键值
 - `array` : 查找的条件参看 [where](#) 数组参数
 - `NULL` : 无参数使用 `where` 等设置查找
- 返回 `Orm Object` 指向调用的 `Orm` 对象自身(查询成功)或 `null` (查询失败)
- 示例代码

```
/*主键find*/  
$user=$orm->find(2);//查询id为2的数据  
  
/*数组条件*/  
$orm->find([  
    'id'=>2,  
    'status'=>1  
]);//查询id为2, status为1的数据
```

`get()` 方法: 获取单条或者单个数据

```
mixed function get([string $key = '', boolean $auto_query = true])
```

- 参数 `$key` [可选]: 要查询的数据键值, 默认获取全部数据
- 参数 `$auto_query` [可选]: 数据不存在时是否自动查询数据库, 默认自动查询
- 返回 `mixed` (`array` | 基本类型): 查询的数据
- 示例代码

```
/*获取全部数据，返回数组或者null*/
$user=$orm->where('id',2)->get();//查询id为2的全部数据数据
/*获取指定键的值*/
$username=$orm->get('name');//查询用户的姓名，自动同步数据库
```

添加数据 (insert)

添加数据提供 `add` , `insert` , `insertAll` 三种方法。

`insert()` 方法: 插入单条数据

```
int function insert(array $data)
```

- 参数 `$data` [必须]: 要插入的数据(键值对)
- 返回 `int` : 插入成功的id(主键值), 适用于自增主键的数据表, 操作失败返回 `false`
- tips: 数据可以使用过滤 `field()` 对数据字段进行过滤
- tips: 之前set的数据对insert无影响
- 示例代码

```
$uid=$orm->insert(['name'=>'future', 'org'=>'nku']);//插入一条数据
/*对于无自增主键的数据表，不会返回id，可如下===判断插入结果*/
if($orm->insert(['uid'=>1, 'pid'=>2])===false){
    // 插入失败
}else{
    //插入成功
}
```

`insertAll()` 方法: 批量插入数据

```
int function insertAll(array $data)
```

- 参数 `$data` [必须]: 要插入的数据二维数组
- 返回 `int` : 插入成功的条数
- 数据可以使用过滤 `field()` 对数据字段进行过滤
- 示例代码

```
$uid=$orm->insert(['name'=>'future', 'org'=>'nku']);//插入一条数据
```

`add()` 方法: 插入已经设置的数据

```
object function add()
```

- 无参数
- 返回 `Orm` 对象或NULL: 操作成功返回自身, 可以继续其他操作
- 数据可以使用过滤 `field()` 对数据字段进行过滤

- tips: 与 insert的区别是会使用之前set的数据
- 示例代码

```
$uid=$orm
  ->set('name','future')
  ->set('org','nku')
  ->add();//插入之前set的数据
```

跟新数据 (update)

更新数据提供 `update` , `save` 两种方法

`update()` 方法: 更新数据

```
int function update(array $data)
```

- 参数 `$data` [必须]: 要跟新的数据二维数组
- 返回 `int` : 跟新成功的条数
- 数据可以使用过滤 `field()` 对数据字段进行过滤
- tips:
 - 之前set的数据对update无影响,如果要保留使用 (save)[#save]
 - 可以跟新多条 `limit` 限制最大条数
- 示例代码

```
/*跟新全部是时间*/
$orm->update(['time'=>date('Y-m-d h:i:s')]);
/*字段过滤*/
$data=['id'=>2,'name'=>'changed name','password'=>'secret'];
$orm->where('id',1)
  ->field('name')
  ->update($data);//只有name字段被更新,其他被过滤
```

`save()` 方法: 保存数据

```
object function save([string $id])
```

- 参数 `$id` (可选): 保存的主键值
- 返回 `Orm` 对象或者 `NULL` : 操作成功返回自身, 可以继续其他操作
- 数据可以使用 `field()` 对数据字段进行过滤
- tips: 与 insert的区别是会使用之前set的数据
- 示例代码

```
/*字段过滤*/
$data=['id'=>2,'name'=>'changed name'];
$orm->field('name')//只有name字段被更新,其他被过滤
  ->set($data)
  ->save(1);//跟新主键为1的name
```

put() 方法: 快速写入

PUT 快速修改单个字段,会立即写入数据库

```
int function put(string $key,mixed $value)
```

- 参数 string \$key : 字段名称
- 参数 mixed \$value : 对应的值
- 返回 int : 影响的条数
- 数据可以使用 field() 对数据字段进行过滤和设置别名
- tips:
- 示例代码

```
/*把id为1的状态修改为1*/  
$orm->where('id',1)->put('status',1);
```

delete() 方法: 删除数据

```
int function delete([string $id])
```

- 参数 \$id (可选): 删除的主键值
- 返回 int : 删除成功的条数
- tips: 可以跟新多条 limit 进行限制
- 示例代码

```
$orm->delete(1); //删除id为1的  
//where限制  
$orm->where('id',1)->delete();
```

条件限制 (condition)

条件查询 (where)

支持where操作如下表

类型	表达式操作(\$op)	值	例子
值比较	= , <> , != , > , >= , < , <=	基本类型	where(\$key,>,10)
空值比较	= , <> , IS	NULL	where(\$key,'<> ',null)
LIKE比较	[NOT]LIKE , [NOT]LIKE BINARY	string	where(\$key,'LIKE','head%')
IN 比较	IN , NOT IN	array	where(\$key,'in',[1,3,5])
BETWEEN	BETWEEN , NOT BETWEEN	array 或 跟两参数	where(\$key,'BETWEEN',1,10) , where(\$key,'BETWEEN', [1,10])

where() 方法: 添加选择条件

```
object function where(mixed $condition [...])
```

- 参数支持多种方式:

- 三元比较: (参见where表)

```
where($field,$operator,$value)
```

1. `string` 字段名(`$field`): 字段名如 `name` , `user.id` (多表查询存在同名字段时, 需要加上表名)
2. `string` 比较符(`$oprater`): 支持 `=` , `<>` , `!=` , `>` , `>=` , `<` , `<=` , `LIKE` , `NOT LIKE` ,等表中所有操作
3. `mixed` 比较的值(`$value`): 数值或者字符串或者NULL等, `in` 和 `between` 操作可以是数组

- 二元相当关系: (三元操作省略 `"="`)

```
where($field,$value)
```

1. `string` 字段名(`$field`): 字段名, 多表查询存在同名字段时, 需要加上表名
2. `scalar` (基本类型) 比较的值(`$value`): 字段的值, `NULL` 会被特殊处理变成IS NULL语句

- 一元数组: (数组批量条件)

```
where($array)
```

1. 关联数组 `array ($field=>$value)`: 每一组键值对相当于二元相等条件
2. 二维索引数组 `array ([$condition1,$condition2,...])`: 每组条件相当于一组where条件 (不递归)

- 四元区间比较: (BETWEEN条件)

```
where($field,$BETWEEN,$min,$max)
```

1. `string` 字段名(`$field`): 字段名, 多表查询存在同名字段时, 需要加上表名
2. `string` (基本类型) 条件: `BETWEEN` 或者 `NOT BETWEEN`
3. `scalar` (基本类型) 最小值(`$min`): 下界(或者上界)
4. `scalar` (基本类型) 最大值(`$max`): 上界(或者下界)

- 返回 `Object` (`Orm` 对象): 返回`$this`继续操作

- tips:

- `NULL` 值(`NULL` 类型,不是`string "NULL"` ,后者会作字符串处理)会被特殊处理
 - 字段值不能是计算表达式,表达式计算用[having](#)
 - 值不能是字段(会被字符串处理),多表联合可以用[join](#)
 - 关闭 `safe` 模式可以在where中使用原生sql条件不推荐使用 (不安全, 也可能造成编译的sql出错, 不利于sql缓存)

- 示例代码

```

/*where 基本操作*/
$form->where('status','>',0);//大于0: WHERE `status`>0
//null特殊处理
$form->where('a.status','!=',null);//非空: WHERE `a`.`status` IS NOT NULL

/*where相等简化*/
//缺省等于
$form->where('status',0);//status为0: WHERE `status`=0
$form->where('data',null);//查找NULL值: WHERE `data` IS NULL
$form->where('id',1)->where('status',1);//并列: WHERE `id`=1 AND `status`=1

/*数组参数型*/
//in array
$form->where('type','IN',[1,3,7]);// 为1, 3或者7 : WHERE `type` IN (1,3,7);
// between
//在不范围之内,status< 1或者status>3: WHERE `status` NOT BETWEEN 1 AND 3
$form->where('status','NOT BETWEEN',[1,3]);

/*四元between*/
$form->where('status','NOT BETWEEN',1,3);//同上

/*关联数组*/
$form->where(['id'=>1,'status'=>1]);//WHERE `id`=1 AND `status`=1
/*二维索引数组*/
$condition=[
    ['status','>',0],
    ['name','LIKE','%future%'],
];
$form->where($condition);//WHERE `status`>0 AND `name` LIKE "%future%"

```

whereField() 方法： 字段比较条件

由于 `where` 默认会将比较的值进行参数绑定，所以如果是字段会按照字符处理，`whereField` 就是用来比较字段之间的关系，值会按照字段处理。

```
object function whereField(mixed $condition [...])
```

- 与 `where` 用法一致
- 示例代码

```

/*whereField 比较*/
$form->whereField('up','>','dwon');//up字段值>down的值

```

orWhere() 方法： OR条件

同 `where` 连接条件变成 OR

```
object function orWhere(mixed $condition [...])
```

示例代码

```
/*where和orWhere限制*/
$form->where('id','<',10)
    ->orWhere('id','>',1000)
    ->select('name');//查询id< 10或者id>1000的用户名
```

orWhereField() 方法： 字段比较条件OR

同where 连接条件一样

```
object function orWhereField(mixed $condition [...])
```

示例代码

```
/*where和orWhere限制*/
$form->where('id','<',100)
    ->orWhereField('regtime','logtime')
    ->select('idname');
```

子查询是否存在exists

exists() 方法

判断子查询是否存在需要使用exist

```
object function exists(Orm $query[, boolean $not=false,[ string $type='AND']])
```

- 参数 (Orm) \$query : 包含查询条件的 Orm 对象
- 参数 (boolean) \$not : 为 true 时 查询 not exists,默认是 false
- 参数 (string) \$type : 连接条件 AND 或者 OR
- 返回 Orm Object : 返回\$this
- 示例代码

```
/*子查询*/
$subQuery=new $orm('user');
$subQuery->where('id',$id);
$form->exists($subQuery)
    ->where('id',$id)
    ->select('id,content');
/*另一种方式*/
InfoModel::exists(
    Db::table('user')->where('id',$id)
)->where('id',$id)
->select('id,content');
```

orExists() 方法

判断子查询是否存在需要使用exist, OR条件链接


```
object function orExists(Orm $query[, boolean $not=false])
```

用法同 [exists](#)

分组和去重

distinct() 方法: 去除相同的结果

数据库在查询的时候返回所有数据库,distinct 可以去除查询结果中重复的结果(同样的查询记录)

```
object function distinct([boolean $is_distinct = true])
```

- 参数 (boolean) \$is_distinct : 设置是否去重,默认参数是 true
- 返回 orm 对象: 可以继续其他操作
- 示例代码

```
/*查询所有的状态,每种状态显示一个*/  
$orm->distinct()->select('status');
```

group() 方法 : 查询结果分组

GROUP 可以按条件或者字段进行分组, 可以连续使用多个GROUP条件

```
object function group(string $field [, string $operator, mixed $value])
```

- 参数: 与where相似但是不接收数组参数.
 - 一个参数:
 1. string (\$field)[必须]: 分组的字段
 - 两个参数:
 1. string (\$field)[必须]: 字段
 2. string (\$value): 相等条件
 - 三个参数:
 1. string (\$field)[必须]: 分组的字段
 2. string (\$operator): 比较符 参照where
 3. mixed (\$value): 比较值
- 返回 orm : 可以继续后续操作
- 示例代码:

```
/*统计每种状态有多少*/  
$orm->group('status')  
->select('status,count(*) as count');
```

计算条件 (**having**)

当查询条件需要使用聚合函数时,需要having函数。WHERE 关键字无法与聚合函数一起使用(sql 中where 先执行)。

having() 方法： 添加选择条件

HAVING AND链接的条件

```
object function having(string $field,string $operator,string $value)
```

- 参数: 与where相似但是不接收数组参数。
 - 两个参数:
 1. string (\$field)[必须]: 字段
 2. string (\$value): 相等条件
 - 三个参数:
 1. string (\$field)[必须]: 分组的字段
 2. string (\$operator): 比较符 参照where
 3. mixed (\$value): 比较值
- 返回 orm : 可以继续后续操作
- 示例代码:

```
/*统计每种状态出现次数大于100的*/  
$orm->group('status')  
->having('count','>',100)//where 会报错  
->select('status,count(*) as count');
```

orHaving() 方法： having条件 or

HAVING 条件 OR 关系，类似于 orWhere

```
object function orHaving(string $field,string $operator,string $value)
```

用法同 having。

字段

修改数据或者读取数据时需要进行数据过滤,或者对字段名进行映射时，可以使用 field 方法

field() 方法： 字段过滤和别名设置

```
object function field(mixed $field [, string $alias])
```

- 参数支持多种方式:
 - 两个参数: (字段别名设置)

```
field($field,$alias)
```

1. `string` 字段(`$field`): 字段名如 `name` , `user.id` (多表查询存在同名字段时, 需要加上表名)
 2. `string` 别名(`$alias`): 别名如 `uid`
- 数组参数:

```
field($array) [$field=>$alias]
```

关联数组 `array ($field=>$value)`: 每一组键值对是一组字段别名隐身

- 字符串参数: (数组批量条件)

```
field($string)
```

- 多个字段用 `,` 隔开
- 别名用 `AS` 链接 如 `'user.id AS id'`

- 返回 `Object` (`Orm` 对象): 返回`$this`继续操作
- tips:
 - 字段值是聚合表达式时时 必须指定别名
 - 如果设置了`field`,`修改`,`插入`和`查询`操作会对其过滤
- 示例代码

```
/*field 二元参数设置别名*/
$orm->field('user_id','uid')
    ->field('name','user')
    ->select();

/*array*/
$orm->field([
    'user_id'=>'uid',
    'name'=>'user',
])->select();

/*字符串*/
$orm->field('user_id AS uid,name AS user')
    ->select();
//select 快捷方法
$orm->select('user_id AS uid,name AS user');

/*update过滤,只有name和info会被更新*/
$orm->field('name,info')->update($data);
```

排序

`order()` 方法: 设置位置和偏移

添加字段

```
object function order(string $fields [, boolean $desc = false])
```

- 最多两个参数:
 1. `string` (`$field`) [必须]: 要排序的字段

- 2. `bool ($desc)` [默认false]: 是否按照降序排列(默认升序排列)
- 返回 `Object (Orm 对象)`: 返回`$this`继续操作
- tips: 排序通常和`limit`结合使用
- 示例代码

```
/*order排序*/
$orm->order('name',true) // 按照name降序
->ordee('id') //再安装id升序(从小到大)
->select('name,id');
```

分页

limit() 方法: 限制读取条数和偏移

```
object function limit( int $maxsize [, int $offset = 0])
```

- 最多两个参数:
 1. `int ($maxsize)` [必须]: 最大条数
 2. `int ($offset)` [默认0]: 偏移量(起始位置)
- 返回 `Object (Orm 对象)`: 返回`$this`继续操作
- 示例代码

```
/*limit 限制读取条数*/
$orm->limit(10) //读取10条数据
->select('name,id');

/*limit 设置偏移量*/
$orm->limit(10,12) //从12条开始读取10条(到22)
->select('name,id');
```

page() 方法: 翻页

实际应用中`limit` 操作通常用来快速翻页,`page`方法是用来翻页的快速操作

```
object function page( int $number [, int $size = 10])
```

- 最多两个参数:
 1. `int ($number)` [必须]: 页码
 2. `int ($size)` [默认10]: 每页条数
- 返回 `Object (Orm 对象)`: 返回`$this`继续操作
- 示例代码

```
/*page 限制读取条数*/
$orm->page(1) //读取第一页(前10条数据)
->select('name,id');

/*page 设置偏移量*/
$orm->page(2,15) //每页15条, 读取第二页
->select('name,id');
```

函数 {function}

Orm 中内置一些常用sql函数和操作

聚合函数

count() 方法: 统计字段

```
int function count( [string $column_name='*', [, boolean $is_distinct = false]])
```

- 最多两个参数:
 1. string (\$column_name) [默认*]: 要统计字段默认全部条数
 2. bool (\$is_distinct) [默认false]: 是否去重
- 返回 int : 统计的数目
- 示例代码

```
/*统计总数*/
$orm->count();

/*统计不重复的字段*/
$orm->count('type',true);
```

sum() 方法: 求和

```
int function sum(string $column_name)
```

- 参数: string (\$column_name)要计算的字段
- 返回 int : 求和结果
- 示例代码

```
/*统计总数*/
$orm->sum('score');
```

avg() 方法: 求均值

```
int function avg(string $column_name)
```

用法同sum

max() 方法： 求最大值

```
int function max(string $column_name)
```

用法同sum

min() 方法： 求最小值

```
int function min(string $column_name)
```

用法同sum

自增自减(写操作)

increment() 方法： 字段值自增

```
int function increment(string $column_name [,int $step=1])
```

- 参数:
 1. `string ($column_name)`: 自增的字段
 2. `int ($step)` [可选]: 增加步长默认为1
- 返回: `int` 操作成功的条数
- 示例代码

```
/*score值+1*/
$form->where('id',1)->increment('score');

/*score值+5*/
$form->where('id',1)->increment('score',5);
```

decrement() 方法： 字段值自减

```
int function decrement(string $column_name [,int $step=1])
```

- 参数:
 1. `string ($column_name)`: 自减少的字段
 2. `int ($step)` [可选]: 减少步长默认为1
- 返回: `int` 操作成功的条数
- 示例代码

```
/*score值-1 相当于 increment('score',-1)*/
$form->where('id',1)->decrement('score');

/*score值-5*/
$form->where('id',1)->decrement('score',5);
```

多表操作

多表查询

join 可以链接多个数据库表，通常 `has 方法` 和 `belongs 方法` 的封装可以满足绝大多数应用场景，推使用这两个方法。

join() 方法

```
Object function join( string $type, string $table, mixed $on [, string related_key=
null])
```

- 参数:
 - 四个参数简单join: `join($type,$table,string $table_key, string $related_key)`
 1. `string ($type)`: JOIN 类型 `INNER,LEFT,RIGHT,OUTER,FULL OUTER` 等
 2. `string ($table)`: JOIN的表名, 支持 `AS` 别名
 3. `string ($on)`: JOIN 表中的关联字段,不用加上表名或别名
 4. `string ($related_key)`: 主表获其他表与之相等的关联字段, 通常要加上表名
 - 三个参数复杂逻辑(数组条件): `join($type,$table,array $on)`
 5. `string ($type)`: JOIN 类型
 6. `string ($table)`: JOIN的表名, 支持 `AS` 别名
 7. `array ($on)` 三维数组: 对于多个条件或者复杂逻辑可以使用这种方式, 每个数组包含一下内容
 - [必须] `on =>array($field,$op,$value)`,参考`where`表达式参数
 - [可选] `logic =>` 条件关系'`AND`'或者'`OR`', 默认采用`AND`连接
 - [可选] `value =>` `NULL` 或者 '`VALUE`',如果不设置或者为`NULL`, `on` 条件中的值会按字段处理, 否则按照值进行绑定
- 返回 `Orm` 对象: 可以继续后续操作
- tips:
 - `has`或者`belongs`等操作比复杂条件效率更高也更容易理解
 - 复杂join的 `on` 条件可以考虑放到`where`条件
- 示例代码

```

/*简单join关联user表的user.id和article表的user_id*/
$orm->join('INNER','article','user_id','user.id');

/*复杂关联*/
$response= Db::table('comment')
    ->field('from.id as from_id,from.name as from')
    ->field('to.id AS to_id,to.name as to')
    ->join('LEFT','article','id','comment.article_id')
    ->join('INNER','user as from',[// 评论发出的用户
        [
            'on'=>['from.id','=','comment.user_id'],
            'logic'=>'AND',//默认是AND可以省略
            'value'=>'NULL',//value为NULL安装字段处理，可以省略
        ],
        [
            'on'=>['from.status','>','0'],
            'logic'=>'AND',
            'value'=>'value',//值绑定,on的第三个参数“0” 会按照值处理，而不是字段
        ],
    ])
    ->join('LEFT','user AS to',[//评论文章的作者
        [
            'on'=>['article.user_id','=','to.id'],
        ],
        [
            'on'=>['to.status','>','0'],
            'value'=>'value',
        ],
    ])
    ->select();

```

has() 方法

has 是对 LEFT JOIN 方法的快捷封装，表示 一个表在逻辑上“拥有”另一个表,比如 用户(user表) 拥有 文章(article表). 可以表示 \$user->has('article'). 此时会使用article的外键关联user主键。

```

Object function has(string $table [, string $table_fk = null [, string $related_key = null]])

```

- 参数:
 1. string (\$table): 关联的表名，可以使用 AS 设置别名
 2. string (\$table_fk) [可选]: has 的 表中对应的外键，默认采用当前Orm对应的表名+'_id'
 3. string (\$related_key) [可选]: 默认是此表的主键，如果多表连接，不是当前表可以加上表名如'table.id'
- 返回 Orm 对象：可以继续后续操作
- 示例代码


```

/*简单用法*/
$form->has('article');

/*多级关联，用户有文章，文章有评论*/
$user->has('article')
    ->has('comment','article_id','article.id');

/*完整查询实例*/
$feed=Db::table('user')
    ->has('feedback AS fb')//用户有feedback 设置别名
    ->where('user.id',1)
    ->select('user.id,user.name,fb.title,fb.content as feedback');

```

belongs() 方法

belongs 是对 INNER JOIN 方法的快捷封装，表示 一个表在逻辑上“从属”另一个表,于has相反。比如 文章属于用户 (article 表的外键如 user_id 关联 user表的主键如id)。

```

Object belongs(string $table [, string $related_key = null [, string $primary_key = 'id']])

```

- 参数:
 1. string (\$table): 关联的表名，可以使用 AS 设置别名
 2. string (\$related_key) [可选]: 与之关联的外键默认 \$table_id ,如果是其他表可以加上表名 table.fk_id
 3. string (\$primary_key) [可选]: \$table表的主键，默认是 id
- 返回 orm 对象: 可以继续后续操作
- 示例代码

```

/*简单用法*/
$article->belongs('user');

/*多级关联，文章属于用户，文章有评论*/
//与has的例子逻辑关系一样，但是查询的主表由user表变成article表
$article->belongs('user')
    ->has('comment');

/*多级关联，实例*/
Db::table('comment')
    ->belongs('user')//评论属于用户
    ->field('user.name','user')//用户名
    ->belongs('article')//评论属于文章
    ->field(['article.title'=>'article'])//选取article 标题
    ->belongs('user AS reply','article.user_id')//文章属于另一个用户
    ->field('reply.id AS rid,reply.name as reciever')//另一个用户的id和姓名
    ->select('comment.*');//comment的所有内容

```

合并查询

union() 方法: 合并

UNION 将结果合并在一起

```
object function union(Orm $query [, boolean $is_all = false])
```

- 参数 (Orm) \$query : 包含查询条件的 Orm 对象, 相当于执行 select 的结果
- 参数 (boolean) \$is_all 默认false: 为 true 时 UNION ALL
- 返回 Orm Object : 返回\$this
- 示例代码

```
Db::table('student')
    ->field('id,name,number')
    ->union(
        Db::table('teacher')
            ->field('id,name,number')
    )->select();
```

unionAll() 方法:全部合并

UNION 默认会去除相同的结果, UNION ALL 不去重

```
object function union(Orm $query)
```

- 参数 (Orm) \$query : 包含查询条件的 Orm 对象, 相当于执行 select 的结果
- 返回 Orm Object : 返回\$this
- 用法与union同
- 示例代码

```
$orm1->where('...')
    //更多设置
    ->field('...');
$orm->unionAll($orm1)
    ->select();
```

其他

transact() 方法: 处理事务

几个操作必须都成功执行的时候, 需要使用事务.

更底层的事务参见[Database::tansaction](#)

```
function transact(callable $func) : mixed
```

- 参数callable \$func: 调用函数过程(可以是匿名函数)
 - \$func 参数是当前对象(\$this)
 - 返回值, 如果是 false (严格的false,null,0等控不是false), 同样执行回滚

- 返回:
 - `false` (执行失败)
 - `$func` 的回调值(执行成功)
- tips:
 - 执行过程中出错同样回滚
 - `$func` 返回 `false` 会强制回滚
- 代码

```
/*事务操作转积分*/
$Orm->transact(function ($user) {
    $user->where('id',1)
        ->increment('score',5);//id为1的用户积分+5
    $user->clear() //清空查询重用
        ->where('id',2)
        ->decrement('score',5);//id为2的积分-5
    return $user->get('score')>0;//判断加分是否为正,如果此时积分小于0依然回滚
});
```

autoClear() 方法: 开启调试输出

查询结束后自动清理掉查询条件和查询参数, 可直接再次使用

```
object function autoClear(boolean $clear[])
```

- 参数 (`boolean`) `$clear` : 是否开启, 设为`false`时关闭自动清除
- 返回 `Orm Object` : 返回`$this`, 可以进行后续操作
- 示例代码

```
$query=Db::table('user')->autoClear();
```

debug() 方法: 开启调试输出

程序调试过程中, 可能需要输出sql语句, `debug`开启之后。对数据库的操作不会执行, 而是直接返回sql语句和参数。影响的操作包括一下操作(返回数组包含 `sql` 和 `param`)

- 查询: `select` , `find` , `get` , `count` , `min` , `max` , `avg` , `sum`
- 添加: `insert` , `add` ,
- 修改: `update` , `save` , `put` , `increment` , `decrement`
- 删除: `delete` ,

```
object function debug([boolean $enable=true])
```

- 参数 (`boolean`) `$enable` : 是否开启, 默认参数是 `true` , 设为`false`时关闭调试
- 返回 `Orm Object` : 返回`$this`, 可以进行后续操作
- 示例代码

```
$query=Db::table('user')
    ->debug()
    ->count();
/*$query 结果如下
Array (
    [sql] => 'SELECT COUNT(*)FROM`user`',
    [param] => Array ( )
);
*/
```

safe() 方法:安全模式

orm 在生成sql语句时，会对所有操作进行严格的格式检查，where和field等操作不能使用原生的sql语句或者复杂的查询条件。必要时可以把safe模式关闭，从而关闭字段格式检查和包装。

警告：尽量不要使用此功能，它会降低安全性同时带来不确定因素！

```
object function safe([boolean $enable=true])
```

- 参数 (boolean) \$enable : 是否开启,默认参数是 true , 设为false时关闭安全模式
- 返回 Orm Object : 返回\$this, 可以进行后续操作
- 示例代码

```
/*各种统计性别人数*/
Db::table('user')->safe(false)//关闭安全模式
    ->field([
        'SUM(CASE WHEN sex = "m" THEN 1 ELSE 0 END)'=>'male',
        'SUM(CASE WHEN sex = "f" THEN 1 ELSE 0 END)'=>'female'
    ])->select();//默认情况会报错
```

clear() 方法: 清空设置和查询

清空之前此ORM所有的查询设置和数据。但是 别名 alias 和 数据库设置 不会清除。通常用来放弃之前的操作或者重用对象。

```
object function clear()
```

- 返回 Orm Object : 返回\$this, 可以进行后续操作
- 示例代码

```
$orm->clear()->select();
```

设定数据库

ORM 会根据配置自动链接数据库

- _ (必须设置): 主数据库(没有额外设置会使用此数据库)

- `_read` (可选): 从数据库(读操作数据库),设置此数据库后读操作默认使用此数据库
- `_write` (可选): 写数据库,设置此数据库写操作优先使用此数据库

`setDb()` 方法:设定数据库

设置数据之后,当前**Orm**对象,读写操作都会直接使用设定的数据库,覆盖默认行为。

```
object function setDb(mixed $db)
```

- 参数 `mixed $db` :
 - `string` : 配置名称,后会自动使用此配置链接
 - `array` 键值对数组: 数据库连接包含
 1. `$db['dsn']` 必须: 数据库的DSN
 2. `$db['username']` 可选: 数据库账号
 3. `$db['password']` 可选: 数据库密码
 - `Database` 对象: 直接使用已经建立连接的数据库对象
- 返回 `Orm Object` : 返回`$this`, 可以进行后续操作
- 示例代码

```
/*配置名称*/
$orm->setDb('_');//强制使用默认数据库

/*数组配置*/
$orm->setDb([
    'dsn'=>'sqlite:/temp/databases/test.db'
]); //切换到此sqlite数据库

/*数据库对象*/
$db=Db::current();//获取默认数据库
$orm->setDb($db);
```

`alias()` 方法: 设置别名

多表操作有时为了方便需要使用别名设置。**Alias** 可以设置当前主表的别名。

```
object function alias(string $alias)
```

- 参数 `string $alias`: 数据表的别名
- 返回 `Orm Object` : 返回`$this`, 可以进行后续操作
- 示例代码

```
$orm->alias('a')//数据库表别名设为a
->select('a.id');
```

数据操作

存取方法

set() 方法：设置数据

```
object function set(mixed $key,mixed $value)
```

- 参数可以是键值对或者数组：
 - 两个参数 键值：
 1. `string (key)`键: 设置的字段
 2. `mixed ($value)`: 设置值
 - 数组：
 - 一个参数 `array` : 批量设置键值对
- 返回 `Orm Object` : 返回`$this`，可以进行后续操作
- tips:
 - 后设置的值会覆盖之前的值
 - 只有进行写入操作(`add` , `save`)之后数据才会保存到数据库
- 示例代码

```
/*设置参数*/  
$orm->set('name','future')  
->set('status',1)  
->add();//添加数据
```

get() 方法：获取数据

快速获取数据 [get](#)

put() 方法：快速修改

快速修改数据参看 [put](#)

Object操作

`Orm` 实现了类的 `__set()` 和 `__get()` 方法，可以直接使用对象成员 `->` 操作符读取和设置数据 但是这种方式读取数据不会读取或写入数据库。

```
/*修改数据*/  
$orm->status=1;//与下面操作等效  
$orm->set('status',1);  
  
/*读取数据*/  
$status=$orm->status;//与下面操作等效  
$status=$orm->get('status',false);
```

Array接口

`Orm` 也实现了数组接口可以直接使用 `[]` 操作符读取和修改数据

```
/*修改数据*/
$orm['status']=1;//与下面操作等效
$orm->set('status',1);

/*读取数据*/
$status=$orm['status'];//与下面操作等效
$status=$orm->get('status',false);
```

JSON序列化

可以对 `orm` 对象直接进行 `json_encode()` 对其中的数据进行虚拟化。因此可以在YYF的**REST**控制器中可以赋值给 `response` ,会直接虚拟化其中的数据。

```
$orm->find(1);
echo json_encode($orm);
```

Database 底层数据库连接

Service/Database 类:

- 实现对底层PDO的继承和轻量封装，提供数据库访问接口。
- 执行出错会Log记录ERROR信息
- 开发环境默认会记录所有的SQL查询请求和结果以及耗时统计

通常你并不需要直接使用此类，Orm和Model在数据连接时会自动的处理此类。

不建议使用原生SQL语句除非有**200%**的把握(对自己100%和对其他修改代码的人100%把握)。此框架中的Orm对sql语句的生成非常安全高效，建议使用对象函数时的方式来查询，采用完全参数化封装防止SQL注入。

接口和方法列表

- 方法接口
 - Database::exec(\$sql,\$params) 执行一条SQL(写),并返回受影响的行数
 - Database::query(\$sql,\$params) 查询一条SQL(读),并返回执行结果
 - Database::column(\$sql,\$params)(查询一条SQL(读),并返回一个值
 - Database::isok() 是否出错
 - Database::errorInfo() 获取出错信息
 - Database::transact(\$func) 执行事务
- 全局接口
 - Database::\$before 数据库请求处理之前调用
 - Database::\$after 数据库操作完成后调用
 - Database::\$debug 调试输出,参数执行出错dump出结果
- 继承自PDO(下面链接均为PHP文档)
 - PDO::beginTransaction() — 启动一个事务
 - PDO::commit() — 提交一个事务
 - PDO::rollBack() — 回滚一个事务
 - PDO::lastInsertId() — 返回最后插入行的ID或序列值
 - PDO::prepare(\$sql) — 查询预处理
 - PDO::setAttribute(\$key,\$value) — 设置属性
 - 其他全部的PDO接口

创建数据库连接

1. 构造函数

如果希望使用数据原生对象可以直接使用下列方式

```
$db= new Service\Database($dsn [, $username=null, $password=null, array $options=null]);
```

1. 使用 Db 辅助类自动创建

Db类对数据库对象进行了封装，可以根据需要自动创建数据库对象和重用。


```
$db=Db::connect();
```

方法接口

sql查询和执行方法

对于SQL语句的处理，做了简化和优化：

- 有参数时，对sql进行预处理和参数绑定保证sql执行的安全性,同时会自动根据参数的类型绑定数据库类型(1 绑定数值, '1' 绑定字符串, true 绑定bool型)
- 无参数时(比如统计全表信息)直接查询或者执行,提高执行速度。
- 操作出错返回的数据统一 false (可以 \$result===false 判断是否成功)

Database::exec() 方法:执行sql命令，返回修改结果

```
function exec(string $sql [, array $params = null]): int
```

- 参数:
 1. string \$sql: SQL 写操作处理语句(UPDATE , INSERT , DELETE),(select 语句使用query或者column查询)
 2. array \$params: 查询参数数组，索引数组(对应 ? 参数)或者键值对数组(对应 :xx 型参数，如果参数无 : 会自动补全)
- 返回: 执行影响的条数
- tips: 插入操作可以使用 lastInsertId() 方法获取查入的ID
- 代码

```
/*参数绑定*/  
//?索引型  
$db->query('DELETE FROM user WHERE id =?',[1]);  
//:键值对型  
$db->query('DELETE FROM user WHERE id =:id',[:id=>1]);  
//:省略型  
$db->query('DELETE FROM user WHERE id =:id',[:id=>1]);
```

Database::query() 方法 查询一条SQL(读),并返回执行结果

```
function query(string $sql [, array $params = null [, boolean $fetchAll= true [, $fetchmode = \PDO::FETCH_ASSOC]]]): array
```

- 参数:
 1. string \$sql: SQL 写操作处理语句(UPDATE , INSERT , DELETE),(select 语句使用query或者column查询)
 2. array \$params: 查询参数数组，索引数组(对应 ? 参数)或者键值对数组(对应 :xx 型参数，如果参数无 : 会自动补全)
 3. boolean \$fetchAll: 结果读取方式 默认 fetchAll 全部二维数组， false 时使用fetch 一维数组
 4. \$fetchmode: 结果返回方式

- 返回: \$fetchmode 确定, 默认二维数组
- 代码

```
/*无参数*/
$db->query('SELECT * FROM user');

/*参数绑定*/
//返回二维数组或者null或者false
$db->query('SELECT * FROM user WHERE id=? AND status=?',[1,0]);
/*参数绑定*/
$db->query('SELECT * FROM user WHERE id=:id AND status>:status',
    [
        ':id'=>1,
        ':status'=>0,
    ]);
$db->query('SELECT * FROM user WHERE id=:id AND status>:status',['id'=>1,'status'=>0]);
```

Database::column() 方法 查询一条SQL(读),并返回一个值

```
function column(string $sql [, array $params = null]): scalar
```

- 参数:
 1. string \$sql: SQL 写操作处理语句(UPDATE , INSERT , DELETE),(select 语句使用query或者column查询)
 2. array \$params: 查询参数数组, 索引数组(对应 ? 参数)或者键值对数组(对应 :xx 型参数, 如果参数无 : 会自动补全)
- 返回: 基本类型
- 代码:

```
/*与query用法基本相同*/
$name=$db->column('SELECT name FROM user WHERE id=?',[1]);//返回的是字符串
```

获取错误信息

查询是否出错,返回bool

```
$db->isOk();
```

获取错误信息

可以在[database]的配置中配置 exception=1 来开启异常, (执行出错抛出,异常, 可以使用 try , catch 处理)

获取错误或者执行是否出错或者粗出结果可以使用

- Database::errorInfo() 获取出错信息
 - 返回数组, 一般三个值
 - 第一个值错误码
 - 第二个错误的代码

- 第三个错误原因

```
$db->errorInfo();
```

事务

几个操作必须都成功执行的时候，需要使用事务

可以使用PDO相关的接口执行事务也可以使用 `transact` 方法来处理：

- `PDO::beginTransaction()` — 启动一个事务
- `PDO::commit()` — 提交一个事务
- `PDO::rollBack()` — 回滚一个事务
- `Database::transact($func)` 快捷事务

`transact()` 方法：处理事务

```
function transact(callable $func [, boolean $err_exception=true]):boolean
```

- 参数 `callable $func`: 调用函数过程(可以是匿名函数)
 - `$func` 参数是当前对象(`$this`)
 - 返回值，如果是 `false` (严格的`false`,`null`,`0`等不是`false`),同样执行回滚
- `boolean $err_exception`: 将错误作为异常处理，这样可以坚持代码内部错误
- 返回: `false` (执行失败)或者`$func`的返回值(执行成功)
- tips: 如果`$func` 无返回值，执行出错同样回滚
- 代码

```

/*简单事务操作*/
$db->transact(function ($DB) {
    $DB->exec('DELETE FROM article WHERE user_id =?', [1]);
    //更多操作...
    $DB->exec('DELETE FROM user WHERE id =?', [1]);
});

/*等效实务操作*/
try{
    $db->beginTransaction();
    if($db->exec('DELETE FROM article WHERE user_id =?', [1]))
    {
        //更多操作...
        $db->exec('DELETE FROM user WHERE id =?', [1]);
        $db->commit();
    }else{
        $db->rollBack();
    }
}catch(Exception $e){
    $db->rollBack();
}

/*实例*/
$id=1;
if($db->transact(function ($DB) use ($id) {
    $DB->exec('DELETE FROM article WHERE user_id =?', [$id]);
    return $DB->exec('DELETE FROM user WHERE id =?', [$id]);
})!==false){
    echo "删除成功!";
}else{
    echo "删除出错失败";
}

```

全局接口

\$before 数据库请求处理之前调用

可以注册一个数据处理接口来拦截请求数据

```
before(string &$sql, array &$params, string name);
```

before 包含三个参数执行的:

- **\$sql** : sql语句,在注册的函数中可以对其修改
- **\$params** : 执行参数, 支持引用传参可以被修改
- **string** : 当前调用的入口名称(**query** , **exec** , **column**)

调试的SQL记录日志采用此接口实现.

\$after 数据库请求处理之前调用

```
after(Database &$this, mixed &$result, string name);
```

after 回调包含三个参数执行的:

- **\$this** : 当前查询对象,在注册的函数中可以对其修改和获取其状态码
- **\$result** : 返回的结果, 支持引用传参可以被修改
- **string** : 当前调用的入口名称(**query** , **exec** , **column**)

\$debug 调试输出

调试时, PDO参数出错,直接dump传入参数。请勿在生产环境使用。

开发环境可以使用debug的一项配置开启。

```
/*手动开启*/  
\Service\Database::$debug=true;  
/*手动关闭*/  
\Service\Database::$debug=false;
```

开发调试

YYF提供多种工具在开发和测试的过程中尽快定位问题。开发环境下自动注入的方式方便查看和调试。同时提供chrome 扩展 YYF-Debugger

- [assert断言错误](#)
- [logger日志记录](#)
- [header调试](#)
- [资源消耗统计]
- [SQL记录和统计]
- [Debug调试工具]

选择调试工具

保留在代码中(长期存在)

assert和logger可以在代码中长期保留,保证系统稳定性。

断言(assert)仅仅在开发环境下有效, 生产环境自动忽略;

日志(logger)是唯一可以在生产环境下高效安全的调试工具;

临时调试 (类似断点)

Debug调试工具包括header用于临时调试或者输出程序中间的变量值。

自动加载

统计工具会在开发环境自动加载.

Copyright © NewFuture 2016 all right reserved, powered by Gitbook文档编译时间: 2017-02-21 11:03:42

Header 浏览器调试输出

API调试过程中为了方便查看中间变量或者查看运行状态,将调试写入响应头中 (不会破坏后端输出是完整的JSON数据)

为了方便查看和显示, 提供chrome插件在浏览器中显示

扩展和接口

- [YYF-Debugger扩展](#)
- [dump](#)
- [SQL记录查询](#)
- [Tracer资源消耗统计](#)

YYF-Debugger (#YYF-Debugger)

[YYF-Debugger扩展](#)



安装此扩展可自动解析header调试信息显示在chrome console中

dump 输出数据

```
function Debug::header([mixed $data]):Header;
```

将数据dump到header中而不影响输出,可以通过插件显示在console中

```
Debug::header($data);//$data写如header或者浏览器console,变量名为`dump`

Debug::header()->something($data);//变量名为`something`

支持连续输出
Debug::header('quick dump')
    ->s1('with special name s1')
    ->a1([1,2,'3'])
    ->i1(2333);
```

SQL查询

开发环境默认会监听所有的sql查询,会详细记录查询过程和结果

```
debug.sql.* = 关闭或者设置相关输出
```

```
Yyf-Sql-[id]:
{
    "T": "查询耗时",
    "Q": "带参数的查询语句",
    "P": "查询参数",
    "E": "查询错误",
    "R": "result"
}
```

通过扩展插件可以格式化显示到浏览器控制台

Tracer资源统计

自动统计和记录当前请求的资源消耗:内存,时间, 文件

tips: 生产环境不会加载调试相关工具和启用配置缓存, 内存和时间消耗都会降低许多

Copyright © NewFuture 2016 all right reserved, powered by Gitbook文档编译时间: 2017-02-21 11:03:42

Assert 断言

断言在框架中多出使用，以确保库被正确规范使用，减少开发过程中不必要的调用错误和编码错误。

在生产环境中，通过禁用系统断言以提高安全性和效率(PHP7生产环境可以对assert断言跳过编译完全0消耗)。

开启断言

默认情况必须开启断言才能正常运行; php5.x YYF会在开发环境自动开启断言。

自己安装的PHP7 需要手动开启(大概 1529行,可搜索 `zend.assertions`)。修改 `zend.assertions` 为 `1` 或者 `0` ;

```
zend.assertions = 1;或者0
```

关闭断言

生产环境关闭断言提高系统稳定性和性能。

```
;php 7完全关闭断言
zend.assertions = -1

;关闭断言处理
assert.active = 0
assert.quiet_eval = 1
```

使用断言

可以使用如下方式进行断言[参PHP Manual](#)

```
assert('the assert code which should be TRUE','message on failed');
```

tips:

- 为了保证安全性和运行效率，assert断言务必使用单引号(')包裹起来
- 由于php5.3不支持第二个参数，YYF对此进行了hack以支持第二个参数，但是性能会下降。

Copyright © NewFuture 2016 all right reserved, powered by Gitbook文档编译时间: 2017-02-21 11:03:42

日志(Logger)

日志记录是重要的调试工具和排错依据。尤其在生产环境,日志记录几乎是最有效的debug信息。YYF对系统日志进行轻量封装,完全兼容[PSR-3日志接口](#)。

- 开发环境: (方便调试)
 - 开发模式默认使用文件(`file`)日志保存到 `runtime\log\yy-mm-dd-TYPE.log` (每种类型一天一份)方便查看
 - 默认记录 `EMERGENCY` , `ALERT` , `CRITICAL` , `ERROR` , `WARN` , `NOTICE` , `INFO` , `DEBUG` , `SQL` (数据库查询), `TRACER` (资源消耗统计)等信息
 - 为了方便调试, 开发环境会对日志记录进行自动监视。
- 生产环境: (安全高效)
 - 默认发送到系统日志(`system`)(写入效率更高, 尤其在高并发写入时)。
 - 默认记录 `EMERGENCY` , `ALERT` , `CRITICAL` , `ERROR` , `WARN` 等信息
 - 如果生产环境使用文件(`file`)日志,为保证数据安全文件权限会默认设为 `600`

日志接口

基础方法

- `Logger::log($level, $message [, $context])`
- `Logger::write($msg, $level = 'NOTICE')`
- `Logger::clear()`

PSR-3日志接口

- `Logger::emergency($message [, $context])`
- `Logger::alert($message [, $context])`
- `Logger::critical($message [, $context])`
- `Logger::error($message [, $context])`
- `Logger::warning($message [, $context])`
- `Logger::warn($message [, $context])`
- `Logger::notice($message [, $context])`
- `Logger::info($message [, $context])`
- `Logger::debug($message [, $context])`

基础方法最底层接口

`write()` 快速写入

快速写入会根据日志级别设置自动过滤日志level

```
function write(string $message [,string $level="NOTICE"]):boolean
```

- 参数:
 1. `string $message` [必填]: 记录消息
 2. `string $level` [选填]: 日志级别 默认是 NOTICE(自动转成大写)
- 返回: `boolean` 日志是否写入成功
- 示例代码

```
//快速查询
Logger::write('some message');
Logger::write('error message', 'ERROR');
```

log() 写入日志

对write的扩展,可以写入数组对象或者模板消息

```
function log(string $level, mixed $message, [array context]):boolean
```

- 参数:
 1. `string $level` [必填]: 日志级别
 2. `string|mixed $message` [必须]: 日志内容, 如果能转换字符串会进行json格式化
 3. `array $context` [可选]: 模板消息替换,三个参数是第二个参数必须是字符串。模板用 `{ }` 标记
- 返回: `boolean` 日志是否写入成功
- 注意: 消息如果是 `object` 且实现了 `__toString()` 方法, 可直接转字符串
- 示例代码

```
//字符串写入
Logger::log('ERROR', 'error message');

//模板消息
Logger::log('WARN', 'login from {ip}', ['ip'=>'12.34.56.78']);//实际消息"login from 12.34.56.78"
//数组对象
Logger::log('DEBUG', ['name'=>'tester', 'info'=>'test']);//实际消息{"name":"tester","info":"test"}

//模板数组
Logger::log('INFO', 'post message {msg} at {time}', [
    'msg'=$_POST;
    'time'=time();
]);//其中{msg}会被 json_encode($_POST)替换;
```

clear() 清空日志

```
function clear()
```

清空所有日志文件.仅对文件模式 `file` 有效,(如果系统日志配置了日志文件也可清除).

其他 **PSR-3**日志接口

实现8中日志接口类型,对 `log` 进行封装,方便快速高效写入日志

```
function emergency($message [, $context]):boolean;

function alert($message [, $context]):boolean;

function critical($message [, $context]):boolean;

function error($message [, $context):boolean;

function warning($message [, $context]):boolean;
function warn($message [, $context):boolean;

function notice($message [, $context):boolean;

function info($message [, $context):boolean;

function debug($message [, $context):boolean;
```

示例代码

```
//字符串写入
Logger::error('error message');

//模板消息
Logger::warn('login from {ip}', ['ip'=>'12.34.56.78']);
Logger::warning('login from {ip}', ['ip'=>'12.34.56.78']);
//数组对象
Logger::debug(['name'=>'tester', 'info'=>'test']); //实际消息{"name":"tester", "info":"test"}

//模板数组
Logger::info('post message {msg} at {time}', [
    'msg'=$_POST;
    'time'=time();
]);

、
```

默认路由

默认路由(无需任何配置)会自动将请求(`http://your.domain/index.php/REQUEST_PATH`)路由到不同action操作上:

缺省Action为 `index` ,缺省Controller为 `index` 。

- [基本路由 /Controller/Action](#)
- [REST路由 /Controller/Action](#)
- [ID参数路由 /Controller/:id/Action](#)
- [多模块路由 /Module/Controller/Action](#)

基本路由

```
/Controller/Action
```

最基本的根据控制器和action映射到对应的操作上.

如请求: `/C/a` ,对应到 `CController` 的 `aAction()` 方法上

REST路由

针对不同的请求方式 `$METHOD` (如 `GET` , `POST` , `PUT` , `DELETE` 等), 会映射到 `$METHOD_actionAction` 上

如GET请求: `/C/a` ,对应到 `CController` 的 `GET_aAction()` 方法上

注: YYF的REST控制器种 `rest`路由的优先级高于`basic`优先级, 当 `METHOD_NAMEAction` 不存在时, 尝试调用 `NAMEAction` 。

ID参数路由

```
/Controller/:id/Action
```

特殊的, 包含数字(id)的请求,会做特殊映射, 此数字会绑定到 `Action` 的参数 `$id` 上,

如GET请求: `/C/123/a` ,对应到 `CController` 的 `aAction($id)` 方法上并绑定参数 `$id=123`

注: REST路由, 同样适用。

多模块路由

```
/Module/Controller/Action
```

当启用多模块支持时, 模块名作为前导。

Cookie 加密存取

YYF中提供了对cookie加密存储和读取的方式(使用AES服务器端加密),可以通过cookie方式调用。

此方法设置可以防止客户端(浏览器)和中间人获取和修改cookie的真实内容。

接口方法

- `Cookie::set($name,$value)`设置cookie
- `Cookie::get($name)`读取
- `Cookie::del()`删除
- `Cookie::flush()`清空

set设置cookie

set快速保存cookie

```
function set(string $name, $value, $path = '', $expire = null, $domain = null)
```

- 参数
 - `$name string` : 存储的cookie键值名
 - `$value mixed` : 存储的cookie值, 可以是任意类型
 - `$path string` : 存储路径, 默认读取配置
 - `$expire int` : cookie过期时间, 默认读取配置
 - `$domain string` : Cookie保存域名, 默认读取配置
- 返回Cookie对象

```
Cookie::set('test_cookie','something');
```

get设置cookie

get快速获取cookie

```
function get(string $name, mixed $default = null):mixed
```

- 参数
 - `$name string` : 存储的cookie键值名
 - `$default mixed` : 可选默认值, 可以是任意类型
- 返回cookie的值或者默认值

```
Cookie::get('test_cookie');  
Cookie::get('test_cookie2','default string');
```

del 删除

del快速删除

```
function del(string $key):boolean;
```

- 参数: string \$key: 键值

```
Cookie::del('test_key');
```

flush 清空

清空全部数据

```
function flush();
```

```
Cookie::flush();
```


内存存储

- 缓存Cache
- 键值对存储Kv

Copyright © NewFuture 2016 all right reserved, powered by Gitbook文档编译时间: 2017-02-21 11:03:42

缓存(Cache)

缓存存储提供快速一致的缓存服务接口(缓存可能会被清理)。支持存储类型:(配置中指明类型即可)

- memcached 内存缓存
- redis 能键值对缓存服务
- file 文件存储磁盘存储
- memcache memcache内存缓存(包括sae)

存储接口

- `Cache::set()`存储值
- `Cache::get()`读取
- `Cache::del()`删除
- `Cache::flush()`清空
- `Cache::handler()`获取当前Kv底层存储对象

set 保存

set快速存储键值

```
function set(string $key, string $value, int $expire=0):boolean;
function set(array $data, int $expire=0):boolean
```

- 双参数:
 1. `string $key`: 获取的键值
 2. `string $value`: 值字符串
- 数组参数: 关联数组多组键值同时设置
- 返回: `boolean`
- tips: 使用redis会调用 `mset` 相当于数据库中的事务, 只有都写入成功才继续。

```
Cache::set('test_key', 'some value', 60);

Cache::set([
    'key1'=>'value1',
    'key2'=>'value2'
], 60);
```

get 获取

get快速获取存储

```
function get(string $key, string $default=false):boolean|string;
function get(string $array):array;
```

- 双参数:
 1. `string $key`: 获取的键值
 2. `string $default`: 默认值
- 数组参数: 返回array

```
Cache::get('test_key');
Cache::get('no_key');//false
Cache::get('no_key','default');//返回'default'

Cache::get(['key1','key2']);//返回数组['key1'=>'value1','key2'=>'value2']
Cache::get(['key1','key2','no_key']);//返回数组['key1'=>'value1','key2'=>'value2','no_key'=>false]
```

del 删除

del快速删除

```
function del(string $key):boolean;
```

- 参数 `string $key`: 键值

```
Cache::del('test_key');
```

flush 清空

清空全部数据

```
function flush();
```

```
Cache::flush();
```

handler 清空

获取处理方式

```
function handler();
```

```
$handler=Cache::handler();
```

键值对存储(Kv)

高效字符串键值对存储提供快速一致的 永久存储服务 (理论上是可靠不会被清理的))接口。支持存储类型:(配置中指明类型即可)

- redis 高性能键值对存储服务
- file 文件存储磁盘存储
- sae sae KVDB键值对存储

存储接口

- `Kv::set()`存储值
- `Kv::get()`读取
- `kv::del()`删除
- `Kv::flush()`清空
- `Kv::handler()`获取当前Kv底层存储对象

set 保存

set快速存储键值

```
function set(string $key, string $value):boolean;  
function set(array $data):boolean
```

- 双参数:
 1. `string $key`: 获取的键值
 2. `string $value`: 值字符串
- 数组参数: 关联数组多组键值同时设置
- 返回: `boolean`
- tips: 使用redis会调用 `mset` 相当于数据库中的事务, 只有都写入成功才继续。

```
Kv::set('test_key', 'some value');  
  
Kv::set([  
    'key1'=>'value1',  
    'key2'=>'value2'  
]);
```

get 获取

get快速获取存储

```
function get(string $key, string $default=false):boolean|string;  
function get(string $data):array;
```

- 双参数:
 1. `string $key`: 获取的键值
 2. `string $default`: 默认值
- 数组参数: 返回array

```
Kv::get('test_key');
Kv::get('no_key');//false
Kv::get('no_key','default');//返回'default'

Kv::get(['key1','key2']);//返回数组['key1'=>'value1','key2'=>'value2']
Kv::get(['key1','key2','no_key']);//返回数组['key1'=>'value1','key2'=>'value2','no_key'=>false]
```

del 删除

del快速删除

```
function del(string $key):boolean;
```

- 参数: `string $key`: 键值

```
Kv::del('test_key');
```

flush 清空

清空全部数据

```
function flush();
```

```
Kv::flush();
```

handler 清空

获取处理方式

```
function handler();
```

```
$handler=Kv::handler();
```

Session 加密存取

YYF中提供对session的便捷操作。可根据需要自动启动session。

接口方法

- `Session::set($name,$value)`保存session
- `Session::get($name,$default)`读取
- `Session::del()`删除
- `Session::flush()`清空
- `Session::start()`指定ID

set设置Session

set快速保存session

```
function set(string $name, $value)
```

- 参数
 - `$name string` : 存储的session键值名
 - `$value mixed` : 存储的session值，可以是任意类型
- 返回Session对象

```
Session::set('key', 'some value');
```

get设置Session

get快速保存session

```
function get(string $name, $default = null)
```

- 参数
 - `$name string` : 存储的session键值名
 - `$default mixed` : 存储的session值，可以是任意类型
- 返回mixed session值

```
Session::get('key');  
Session::get('key2', 'default value');
```

..

del 删除

del快速删除

```
function del(string $key):boolean;
```

- 参数: string \$key: 键值

```
Session::del('test_key');
```

flush 清空

清空全部数据

```
function flush();
```

```
Session::flush();
```

start 启动

启动session

```
function start( string $id=null);
```

```
Session::start();  
Session::start('someidstring');
```

RSA 加密存取

RSA是非对称加密， `Rsa` 对其简单封装(需要rsa扩展),可以放到加密和解密.

接口方法

- `Rsa::encode($string,$id=null)` Rsa加密
- `Rsa::decode($string,$id=null)` Rsa解密
- `Rsa::pubKey($id=null)` 获取对应的公钥

encode 加密

Rsa加密

```
function encode( string $id=null);
```

```
Rsa::encode('someidstring');  
Rsa::encode('someidstring',123456);
```

decode 解密

Rsa解密

```
function decode( string $id=null);
```

```
Rsa::decode('xxxxx');  
Rsa::decode('xxxx',123456);
```

pubKey 获取公钥

获取加密公钥 Rsa 公钥

```
function pubKey( string $id=null);
```

```
Rsa::pubKey();  
Rsa::pubKey(123456);
```


CORS 跨站请求

- [cors 跨域请求简介](#)
- [服务器端跨域配置](#)
- [跨域cookie支持](#)

cors 跨域请求简介

HTML5 标准中，明确了针对浏览器(javascript中)跨域请求(CORS)的标准。

通过服务器特定的响应头来设置跨域内容。其中必须包含的内容:

- `Access-Control-Allow-Origin` : 跨域请求来源, 即前端网站地址)和
- `Access-Control-Allow-Methods` : 允许的请求方式

示例显示

```
Access-Control-Allow-Origin : your.domain
Access-Control-Allow-Methods : GET,POST,PUT,DELETE
Access-Control-Allow-Headers : x-requested-with,accept,content-type,session-id,token
```

提示: 浏览器端可使用yyfjs前端库方便处理跨域请求(优化跨域请求预处理, 支持cookie)

服务器端跨域配置

YYF默认对跨域提供了支持, 同时提供了方便的配置方便自由调整。并允许生产环境和开发环境使用不同的配置

配置中即为CORS相关配置

```
cors.Access-Control-Allow-xxx
```

同时提供了原生**CORS**协议不支持的方式

- 多域名支持逗号分割: ``Access-Control-Allow-Origin = "site1.your.domain,site2.your.domain"``
- 泛域名cookie支持

提示: `*` 设置为允许的请求源,是不安全的方式, 生产环境指明请求源最佳。

跨域cookie支持

服务器端

CORS请求默认不允许使用cookie(上传到服务器端),需要在服务器端设置

```
Access-Control-Allow-Credentials : true
```

才能开启cookie(虽然cors协议做此设置不支持泛域名 `*` ,但是YYF中响应时做了些 `hack` 支持此配置,)

客户端

`XMLHttpRequest` 默认允许设置cookie。需要设置参数 `Credentials=true` 才可。

当使用yyfjs时，如下配置即可自动使用cookie:

```
YYF({  
  cookie:true,  
});
```

Copyright © NewFuture 2016 all right reserved, powered by Gitbook文档编译时间: 2017-02-21 11:03:42

Wechat 微信常用操作静态封装

Wechat 类: 对微信登录和JS签名等进行快捷封装,只需调用指定的方法, 会自动生成和验证。

使用前保证账号具有相应权限, 并在微信上正确配置相关域名。

(正确配置后此接口会自动生成验证state和缓存签名授权)

接口和方法列表

- 方法接口
 - **Wechat::getAuthUrl()** 获取微信内端认证跳转链接
 - **Wechat::checkCode()** 静默认证验证
 - **Wechat::getUserInfo()** 获取用户相信信息
 - **Wechat::signJs()** 对URL进行JS签名数据生成
 - **Wechat::loginConfig()** web端微信登录JS配置生成
 - **Wechat::state()** 设置或者读取state设置
- 相关配置(secret.*.ini中 **[wechat]** 相关配置)
 - **appid** : 微信开发应用ID
 - **secret** : 微信开发APPKEY
 - **state** : 回调自动验证state方式防止重复 'cookie' 或这 'session', 为空[""]不进行自动验证
 - **redirect_base** : 微信内静默授权回调 **snsapi_base** (不弹出授权页面, 直接跳转, 只能获取用户openid)
 - **redirect_userinfo** : 微信内授权回调URL ;**snsapi_userinfo** (弹出授权页面, 可通过openid拿到昵称、性别、所在地)
 - **redirect_login** : 微信网页登录回调URL 开发者接口 ;**snsapi_login** (网页端登录)
 - **js.*** : 生成签名配置时的附加参数如 **js.debug=1** 开启微信JS调试

微信类接口

getAuthUrl生成

```
function getAuthUrl($scope = 'userinfo', $redirect = null): string
```

- 参数 **\$scope string**: 授权类型
 1. **userinfo** (获取详细信息,默认)
 2. **base** (静默授权)
- 参数**\$redirect string**: 回调url, 默认读取配置
- 返回: **string** 重定向URL
- 代码

```
//根据配置生成获取详细信息url
$url=Wechat::getAuthUrl();

//微信内静默授权
$url=Wechat::getAuthUrl('base');

//在controller中可以直接重定向
$this->redirect($url);
```

checkCode验证微信返回的code

获取用户openid或者token通常配合静默授权使用

```
function checkCode($key = 'openid', $code = false): mixed
```

- 参数\$key, string: 指定获取的内容
 1. openid ,默认获取openid
 2. access_token ,获取授权token
 3. null 或者false返回整个数组
- 参数\$code, string: 验证的code,默认读取 GET 参数
- 返回:
 - 验证失败: 返回false
 - 验证成功返回 对应的值 (string或者array)
- 代码

```
//微信回调操作,配合 Wechat::getAuthUrl('base')使用
$openid=Wechat::checkCode();
```

getUserInfo微信详细信息回掉

获取用户的详细信息

```
function getUserInfo($code = false): array
```

- 参数\$code, string: 微信回调code, 默认自动读取get参数
- 返回:
 - 验证失败: 返回false或者null(code无效返回null)
 - 验证成功返回 array包括{openid,nickname,headimgurl,sex,language,city,province,country,privilege}
- 代码

```
//微信回调操作,配合 Wechat::getAuthUrl('userinfo')使用
$info=Wechat::getUserInfo();
```

signJs JS授权签名

微信分享等接口需要使用JS签名

```
function signJs($url = false): array
```

- 参数\$**url**, string: 签名的url,如果未设置默认自动读取请求的refer
- 返回:
 - 验证失败: 返回null
 - 验证成功返回 array包括{appId,timestamp,nonceStr,signature}
- tips: 授权返回数据可直接返回到客户端进行签名验证
- tips: 如果没有此即可任何跨域限制和授权验证, 请务必制定url防止接口被其他网站调用
- 代码

```
$data=Wechat::signJs();//自动获取访问url签名  
$data=Wechat::signJs('http://yyf.yunyin.org/h5.html');//对指定url签名  
echo json_encode($data);
```

loginConfig 生成网页端登录的配置

网页端调用微信登录API

```
function loginConfig($redirect = false): array
```

- 参数\$**redirect** string: 回调url, 默认读取配置
- 返回配置数组: {'appid','scope','redirect_uri'}
- 代码

```
$config=Wechat::loginConfig();
```

state 自定义状态设置或者读取

微信验证中, 有一个get参数 `state` 供回调验证,默认生成随机字符验证.

也可以通过state()方法动态设置state或获取参数

```
function state([$state]): mixed
```

- 无参数时, 返回当前设置的state状态
- 有参数时, 返回当前实例可继续操作

```
$state=wechat::state();//获取当前的state设置

//使用指定的state生成跳转url
$url=wechat::state('specail_state_string')->getAuthUrl();
//等效操作
wechat::state('specail_state_string')
$url=wechat::getAuthUrl()

//使用指定的state进行验证
$info=wechat::state('specail_state_string')->getUserInfo();
//等效操作
wechat::state('specail_state_string');
$info=wechat::getUserInfo();

//临时关闭,验证
$openid=wechat::state(FALSE)->checkCode();
```