yaf框架学习笔记

liustone

目　录

yaf简介

简介

PHP框架层出不穷，比如laravel,symfony,Zend Framework,codeigniter,国内的thinkphp等。

在做项目的时候使用框架的好处是提高开发效率，缺点就是损失了性能。

以下是鸟哥的自白：

也就是说, 有没有那么一个框架, 既不会有损性能, 又能提高开发效率呢.

Yaf, 就是为了这个目标而生的.

Yaf有着和Zend Framework相似的API, 相似的理念, 而同时又保持着对Bingo的兼容, 以此来提高开发效率, 规范开发习惯. 本着对性能的追求, Yaf把框架中不易变的部分抽象出来,采用PHP扩展实现(c语言),以此来保证性能.在作者自己做的简单测试中, Yaf和原生的PHP在同样功能下, 性能损失小于10%, 而和Zend Framework的对比中, Yaf的性能是Zend Framework的50-60倍.

yaf的优势

Yaf的优点

天下武功无坚不破，唯快不破

1. 用C语言开发的PHP框架, 相比原生的PHP, 几乎不会带来额外的性能开销.
2. 所有的框架类, 不需要编译, 在PHP启动的时候加载, 并常驻内存.
3. 更短的内存周转周期, 提高内存利用率, 降低内存占用率.
4. 灵巧的自动加载. 支持全局和局部两种加载规则, 方便类库共享.
5. 高性能的视图引擎.
6. 高度灵活可扩展的框架, 支持自定义视图引擎, 支持插件, 支持自定义路由等等.
7. 内建多种路由, 可以兼容目前常见的各种路由协议.
8. 强大而又高度灵活的配置文件支持. 并支持缓存配置文件, 避免复杂的配置结构带来的性能损失.
9. 在框架本身,对危险的操作习惯做了禁止.
10. 更快的执行速度, 更少的内存占用.

Yaf安装/配置

2.1. Yaf的安装

Yaf只支持PHP5.2及以上的版本

Yaf需要SPL的支持. SPL在PHP5中是默认启用的扩展模块

Yaf需要PCRE的支持. PCRE在PHP5中是默认启用的扩展模块

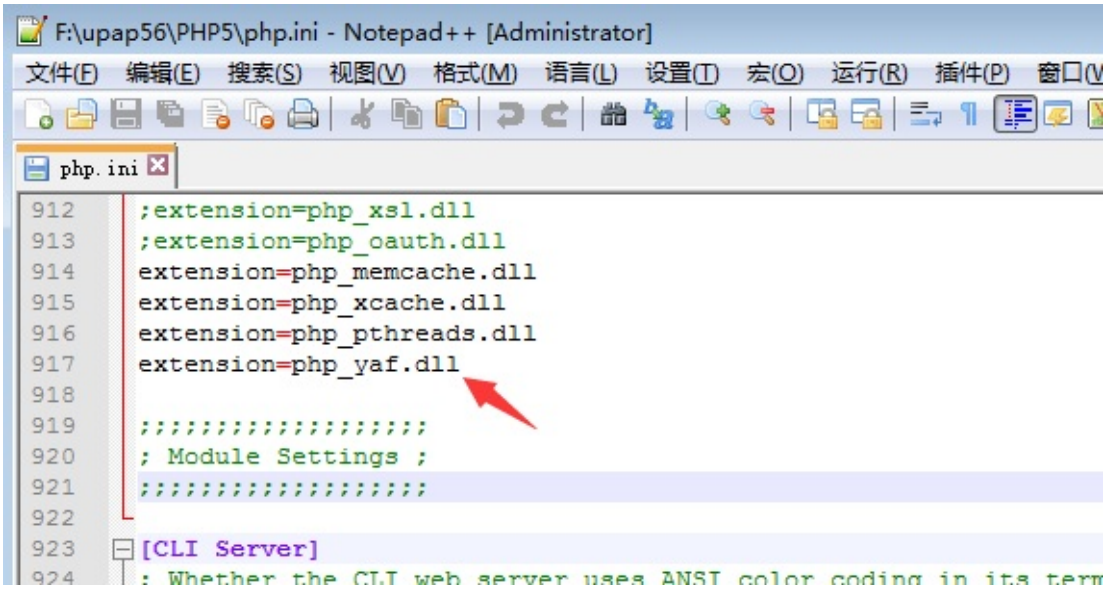
在 Windows 系统下安装

PHP 5.2+

1. 打开yaf在php官网上的目录：http://pecl.php.net/package/yaf
2. 目前yaf的最新版为3.0.0,仅支付php7,建议选择2.3.5版本
3. 我这里选择2.3.5后面的win图标+DLL字样的链接，进入页面下载php_yaf.dll
4. 在打开的页面根据自己的环境来选择对应的版本，我这里选择的是php5.6 Thread Safe (TS) x86(php5.6版本 安全线程 32位操作系统)

	DLL List
PHP 5.6	5.6 Non Thread Safe (NTS) x86 5.6 Thread Safe (TS) x86 5.6 Non Thread Safe (NTS) x64 5.6 Thread Safe (TS) x64
PHP 5.5	5.5 Non Thread Safe (NTS) x86 5.5 Thread Safe (TS) x86 5.5 Non Thread Safe (NTS) x64 5.5 Thread Safe (TS) x64
PHP 5.4	5.4 Non Thread Safe (NTS) x86 5.4 Thread Safe (TS) x86
PHP 5.3	5.3 Non Thread Safe (NTS) x86 5.3 Thread Safe (TS) x86

5. 点击后自动下载了一个压缩包：php_yaf-2.3.5-5.6-ts-vc11-x86.zip
6. 把压缩包中的php_yaf.dll复制出来，打到你的php目录，打开目录下的ext文件夹，粘贴进去
7. 再打开您的PHP配置文件php.ini，加入 'extension=php_yaf.dll',重启web服务器,就OK了



在 Linux 系统下安装

下载Yaf的最新版本, 解压缩以后, 进入Yaf的源码目录, 依次执行(其中PHP_BIN是PHP的bin目录):

> $PHP_BIN/phpize
>
> ```
> ./configure --with-php-config=$PHP_BIN/php-config
> make
> make install
> ```

然后在php.ini中载入yaf.so, 重启PHP.

Yaf_Request_Abstract的getPost, getQuery等方法, 并没有对应的setter方法. 并且这些方法是直接从PHP内部的$_POST, $_GET等大变量的原身变量只读的查询值, 所以就有一个问题:通过在PHP脚本中对这些变量的修改, 并不能反映到getPost/getQuery等方法上.

Yaf定义的常量

Yaf定义的常量

常量(启用命名空间后的常量名)	说明
YAF_VERSION(Yaf\VERSION)	Yaf框架的三位版本信息
YAF_ENVIRON(Yaf\ENVIRON	Yaf的环境常量, 指明了要读取的配置的节, 默认的是product
YAF_ERR_STARTUP_FAILED(Yaf\ERR\STARTUP_FAILED)	Yaf的错误代码常量, 表示启动失败, 值为512
YAF_ERR_ROUTE_FAILED(Yaf\ERR\ROUTE_FAILED)	Yaf的错误代码常量, 表示路由失败, 值为513
YAF_ERR_DISPATCH_FAILED(Yaf\ERR\DISPATCH_FAILED)	Yaf的错误代码常量, 表示分发失败, 值为514
YAF_ERR_NOTFOUND_MODULE(Yaf\ERR\NOTFOUD\MODULE)	Yaf的错误代码常量, 表示找不到指定的模块, 值为515
YAF_ERR_NOTFOUND_CONTROLLER(Yaf\ERR\NOTFOUD\CONTROLLER)	Yaf的错误代码常量, 表示找不到指定的Controller, 值为516
YAF_ERR_NOTFOUND_ACTION(Yaf\ERR\NOTFOUD\ACTION)	Yaf的错误代码常量, 表示找不到指定的Action, 值为517
YAF_ERR_NOTFOUND_VIEW(Yaf\ERR\NOTFOUD\VIEW)	Yaf的错误代码常量, 表示找不到指定的视图文件, 值为518
YAF_ERR_CALL_FAILED(Yaf\ERR\CALL_FAILED)	Yaf的错误代码常量, 表示调用失败, 值为519
YAF_ERR_AUTOLOAD_FAILED(Yaf\ERR\AUTOLOAD_FAILED)	Yaf的错误代码常量, 表示自动加载类失败, 值为520
	Yaf的错误代码常量,

	数错误, 值为521

快速开始

假设Yaf已经正确编译, 安装.

Hello Yaf

目录结构

一个典型的目录结构

> - public
> |- index.php //入口文件
> |- .htaccess //重写规则
> |+ css
> |+ img
> |+ js
> >+ conf
> |- application.ini //配置文件
> >+ application
> |+ controllers
> |- Index.php //默认控制器
> |+ views
> |+ index //控制器
>
> ```
> |- index.phtml //默认视图
> ```
>
> |+ modules //其他模块
> |+ library //本地类库
> |+ models //model目录
> |+ plugins //插件目录

入口文件

几乎所有php框架都采用了单一入口方式，入口文件是所有请求的入口, 一般都借助于rewrite规则, 把所有的请求都重定向到这个入口文件.

一个经典的入口文件public/index.php

```
<?php
define("APP_PATH",  realpath(dirname(__FILE__) . '/../')); /* 指向public的上一级
$app  = new Yaf_Application(APP_PATH . "/conf/application.ini");
$app->run();
```

配置文件

在Yaf中, 配置文件支持继承, 支持分节. 并对PHP的常量进行支持. 你不用担心配置文件太大造成解析性能问题, 因为Yaf会在第一个运行的时候载入配置文件, 把格式化后的内容保持在内存中. 直到配置文件有了修改, 才会再次载入.

一个简单的配置文件application/conf/application.ini

```
[product]
;支持直接写PHP中的已定义常量
application.directory=APP_PATH "/application/"
```

控制器

在Yaf中, 默认的模块/控制器/动作, 都是以Index命名的, 当然,这是可通过配置文件修改的.
对于默认模块, 控制器的目录是在application目录下的controllers目录下, Action的命名规则是"名字+Action"

默认控制器application/controllers/Index.php
`<?php
class IndexController extends Yaf_Controller_Abstract {
public function indexAction() {//默认Action

```
    $this->getView()->assign("content", "Hello Yaf");
```

}
}
?>`

视图文件

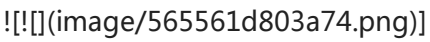
Yaf支持简单的视图引擎, 并且支持用户自定义自己的视图引擎, 比如Smarty.
对于默认模块, 视图文件的路径是在application目录下的views目录中以小写的action名的目录中.

`

```
<?php echo $content;?>
```

`

html在这个文档里不显示，我这里截了一张图



运行

在浏览器输入 http://yourhostname/public/index.php

是不是看到了页面输出的Hello Yaf

额外说明一下：

必要的配置项

Yaf和用户共用一个配置空间, 也就是在Yaf_Application初始化时刻给出的配置文件中的配置. 作为区别, Yaf的配置项都以ap开头. Yaf的核心必不可少的配置项只有一个(其实, 这个也可以有默认参数, 但是作者觉得完全没有配置, 显得太寒酸了).

application.directory String 应用的绝对目录路径

就是我们刚在application/conf/application.ini里配置的内容

Yaf的项目可选配置信息

详情可以稳步到yaf文档查看http://www.laruence.com/manual/yaf.config.optional.html

关于加载

自动加载器

Yaf在自启动的时候, 会通过SPL注册一个自己的Autoloader, 出于性能的考虑, 对于框架相关的MVC类, Yaf Autoloader只以目录映射的方式尝试一次.

Yaf目录映射规则

类型	后缀	映射路径
控制器	Controller	默认模块下为{项目路径}/controllers/, 否则为{项目路径}/modules/{模块名}/controllers/
数据模型	Model	{项目路径}/models/
插件	Plugin	{项目路径}/plugins/

一个简单的自我理解

`<?php

class IndexController extends Yaf_Controller_Abstract {

```
public function indexAction() {//默认Action
    $mod = new TserModel();  //自动加载model下面的test.php文件
    $mod->query();  //调用TestModel里的query方法
    $user = new UserPlugin(); //自动加载plugins下面的user.php文件
    $this->getView()->assign("title", "Hello Yaf");
    $this->getView()->assign("content", "Hello Yaf Content");
}
```

`

类的加载规则

而类的加载规则, 都是一样的: Yaf规定类名中必须包含路径信息, 也就是以下划线"_"分割的目录信息. Yaf将依照类名中的目录信息, 完成自动加载. 如下的例子, 在没有申明本地类的情况下:

`

```
public function indexAction() {
    $upload = new upload_aliyun(); //这个就会按下划线分割目录来寻找文件，所以他会寻找 \library\upload\aliyun.ph
p

}`
```

先这么简单理解，还有一个registerLocalNamespace的内容，后续再来说一说，怕混了。

手动载入

Yaf_Loader::import

导入一个PHP文件, 因为Yaf_Loader::import只是专注于一次包含, 所以要比传统的require_once性能好一些

示例：

```
<?php
//绝对路径
Yaf_Loader::import("/usr/local/foo.php);
//相对路径, 会在APPLICATION_PATH."/library"下加载
Yaf_loader::import("plugins/User.php");
?>
```

使用Bootstrap

Bootstrap, 也叫做引导程序. 它是Yaf提供的一个全局配置的入口, 在Bootstrap中, 你可以做很多全局自定义的工作.

使用Bootstrap

在一个Yaf_Application被实例化之后, 运行(Yaf_Application::run)之前, 可选的我们可以运行Yaf_Application::bootstrap

改写index.php文件如下：

```
<?php
define("APP_PATH",  realpath(dirname(__FILE__)));
$app  = new Yaf_Application(APP_PATH . "/conf/application.ini");
$app->bootstrap()->run();
```

当bootstrap被调用的时刻, Yaf_Application就会默认的在APPLICATION_PATH下, 寻找Bootstrap.php, 而这个文件中, 必须定义一个Bootstrap类, 而这个类也必须继承自Yaf_Bootstrap_Abstract.

实例化成功之后, 所有在Bootstrap类中定义的, 以_init开头的方法, 都会被依次调用, 而这些方法都可以接受一个Yaf_Dispatcher实例作为参数.

也可以通过在配置文件中修改application.bootstrap来变更Bootstrap类的位置.

简单的示例Bootstrap.php

`<?php
class Bootstrap extends Yaf_Bootstrap_Abstract{

```
    public function _initConfig() {
            $config = Yaf_Application::app()->getConfig();
            Yaf_Registry::set("config", $config);
    }
    public function _initDefaultName(Yaf_Dispatcher $dispatcher) {
            $dispatcher->setDefaultModule("Index")->setDefaultController("Index")->setDefaultAction("in
dex");
    }
```

}`

使用插件

Yaf支持用户定义插件来扩展Yaf的功能, 这些插件都是一些类. 它们都必须继承自Yaf_Plugin_Abstract. 插件要发挥功效, 也必须现实的在Yaf中进行注册, 然后在适当的实际, Yaf就会调用它.

Yaf支持的Hook

名称	触发时机	说明
routerStartup	在路由之前触发	这个是7个事件中, 最早的一个. 但是一些全局自定的工作, 还是应该放在Bootstrap中去完成
routerShutdown	路由结束之后触发	此时路由一定正确完成, 否则这个事件不会触发
dispatchLoopStartup	分发循环开始之前被触发	
preDispatch	分发之前触发	如果在一个请求处理过程中, 发生了forward, 则这个事件会被触发多次
postDispatch	分发结束之后触发	此时动作已经执行结束, 视图也已经渲染完成. 和preDispatch类似, 此事件也可能触发多次
dispatchLoopShutdown	分发循环结束之后触发	此时表示所有的业务逻辑都已经运行完成, 但是响应还没有发送

定义插件

插件类是用户编写的, 但是它需要继承自Yaf_Plugin_Abstract. 对于插件来说, 上一节提到的7个Hook, 它不需要全部关心, 它只需要在插件类中定义和上面事件同名的方法, 那么这个方法就会在该事件触发的时候被调用.

而插件方法, 可以接受俩个参数, Yaf_Request_Abstract实例和Yaf_Response_Abstract实例. 一个插件类例子如下:
plugins/User.php

`<?php

```
class UserPlugin extends Yaf_Plugin_Abstract {
public function routerStartup(Yaf_Request_Abstract $request, Yaf_Response_Abstract $response) {
}
public function routerShutdown(Yaf_Request_Abstract $request, Yaf_Response_Abstract $response) {
}
}`
```

注册插件

插件要生效, 还需要向Yaf_Dispatcher注册, 那么一般的插件的注册都会放在Bootstra中进行. 一个注册插件的例子如下:

`<?php
class Bootstrap extends Yaf_Bootstrap_Abstract{

```
    public function _initPlugin(Yaf_Dispatcher $dispatcher) {
        $user = new UserPlugin();
        $dispatcher->registerPlugin($user);
    }
```

}`

目录

一般的, 插件应该放置在APPLICATION_PATH下的plugins目录, 这样在自动加载的时候, 加载器通过类名, 发现这是个插件类, 就会在这个目录下查找.

当然, 插件也可以放在任何你想防止的地方, 只要你能把这个类加载进来就可以

获取参数

Yaf_Request_Http

代表了一个实际的Http请求, 一般的不用自己实例化它, Yaf_Application在run以后会自动根据当前请求实例它，在控制器内可以使用$this->getRequest()来获取请求信息。
更多Yaf_Request_Http类的内容可参见文档：
http://www.laruence.com/manual/yaf.class.request.html#yaf.class.request.http

使用示例
`<?php
class IndexController extends Yaf_Controller_Abstract {

```
public function indexAction($name='', $value='') {
    print_r($this->getRequest()->getQuery());
}`
```

扩展Yaf_Request_Http，比如加上过滤，数据处理等。先在library定义一个request的类,再在Bootstrap.php里设置Request

文件示例：library/Request.php

`<?php
class Request extends Yaf_Request_Http
{

```
private $_posts;
private $_params;
private $_query;
public function getPost()
{
    if ($this->_posts) {
        return $this->_posts;
    }
    $this->_posts = $this->filter_params(parent::getPost());
    return $this->_posts;
}
public function getParams()
{
    if ($this->_params) {
        return $this->_params;
    }
    $this->_params = $this->filter_params(parent::getParams());
    return $this->_params;
}
public function getQuery()
{
    if ($this->_query) {
        return $this->_query;
    }
```

```
        $this->_query = $this->filter_params(parent::getQuery());
        return $this->_query;
    }
    private function filter_params($params)
    {
        if (!empty($params)) {
            array_walk_recursive($params, function(&$value, $key){
                $value=htmlspecialchars($value, ENT_QUOTES, 'UTF-8');
            });
        }
        return $params;
    }
```

}`

文件示例：Bootstrap.php

`<?php

class Bootstrap extends Yaf_Bootstrap_Abstract{

```
public function _initRequest(Yaf_Dispatcher $dispatcher)
{
    $dispatcher->setRequest(new Request());
}`
```

然后在控制器中可以使用$this->getRequest()->getQuery()来获取参数

`<?php

class IndexController extends Yaf_Controller_Abstract {

```
public function indexAction() {
    print_r($this->getRequest()->getQuery());
}`
```

关于更多的该类的使用方法，可以参考：

http://www.laruence.com/manual/yaf.class.request.html

session和cookie

session

简介

Yaf_Session是Yaf对Session的包装, 实现了Iterator, ArrayAccess, Countable接口, 方便使用.

关于Yaf_Session的文档介绍：http://www.laruence.com/manual/yaf.class.session.html

使用示例：

`<?php

class IndexController extends Yaf_Controller_Abstract {

```
public function indexAction() {
    $y_session = Yaf_Session::getInstance();
    $username = $y_session->set('username','liu');
    var_dump($y_session->get('username'));`
```

Yaf_Session

```
<?php
final Yaf_Session implements Iterator , ArrayAccess , Countable {
public static Yaf_Session getInstance ( void );
public Yaf_Session start ( void );
public mixed get ( string $name = NULL );
public boolean set ( string $name ,
mixed $value );
public mixed __get ( string $name );
public boolean __set ( string $name ,
mixed $value );
public boolean has ( string $name );
public boolean del ( string $name );
public boolean __isset ( string $name );
public boolean __unset ( string $name );
}
```

cookie

目前还没有看到Yaf里面有cookie的介绍

模板

模板

The Yaf_View_Simple class

官方文档：http://www.laruence.com/manual/yaf.class.view.html

Yaf_View_Simple是Yaf自带的视图引擎, 它追求性能, 所以并没有提供类似Smarty那样的多样功能, 和复杂的语法.

对于Yaf_View_Simple的视图模板, 就是普通的PHP脚本, 对于通过Yaf_View_Interface::assgin的模板变量, 可在视图模板中直接通过变量名使用.

使用$this->getView()->assign（ ）在控制器中定义变量

`<?php

class IndexController extends Yaf_Controller_Abstract {

```
public function indexAction() {
    $mod = new UserModel();
    $list = $mod->where('id',1)->get();
    $this->getView()->assign("list", $list);
    $this->getView()->assign("title", "Smarty Hello World");
    $this->getView()->assign("content", "Hello World");
}`
```

在模板文件中使用php脚本来输出

`

```
<meta charset="utf-8">
<title><?php echo $title;?></title>
```

```
<?php echo $content;?>
<?php foreach($list as $val){?>
<p><?php echo $val['username']?></p>
<?}?>
```

`

关闭Yaf框架里的自动加载模板

Yaf框架默认是开启自动加载模板的，如要关闭自动加载，可在Bootstrap.php里设置全局关闭，如：

`<?php
class Bootstrap extends Yaf_Bootstrap_Abstract
{

```
public function _initConfig()
{
    Yaf_Registry::set('config', Yaf_Application::app()->getConfig());
    Yaf_Dispatcher::getInstance()->autoRender(FALSE);  // 关闭自动加载模板
}
```

}`

单独关闭模板加载，可以需要关闭的控制器内利用Yaf_Dispatcher::getInstance()->disableView（ ）操作：
`<?php

```
class IndexController extends Yaf_Controller_Abstract {
    /**
     * Controller的init方法会被自动首先调用
     */
     public function init() {
        /**
         * 如果是Ajax请求，则关闭HTML输出
         */
        if ($this->getRequest()->isXmlHttpRequest()) {
            Yaf_Dispatcher::getInstance()->disableView();
        }
     }
}
?>`
```

手动调用指定模板

在控制器里手动调用的方式有2种：
一，调用当前$this->_module目录下的模版，下面是手动调用view/index/目录下hello.phtml模板
`<?php
class IndexController extends Yaf_Controller_Abstract
{

```
public function indexAction()
{
    $this->getView()->assign("content", "Hello World");
    $this->display('hello');
}
```

} 二，随意调用view目录下的模板，下面是调用view/test/world.phtml模板 <?php
class IndexController extends Yaf_Controller_Abstract

{

```
public function indexAction()
{
    $this->getView()->assign("content", "Hello World");
    $this->getView()->display('test/world.phtml');
}
```

}`

模型model

模型model

在看模型之前，我们先看一段yaf作者鸟哥在他自己博客里的一段文字吧：

> 还有不少同学问, 为什么Yaf没有ORM, 这里有俩方面的考虑:
> 首先, Yaf并不是万能的, 它只是解决了应用中, 最基本的一个问题, 就是框架带来的额外的性能开销, 然而这本部分的开销和你的应用实际的开销相比, 往往是很小的.
> 但是, Yaf却代表着一种精神, 就是追求简单, 追求高效, 追求:" 简单可依赖" , 所以Yaf专注于实现最核心的功能, 提供最稳定的实现.
> 相比ORM, 如果要实现的很方便, 那必然会很复杂, 在当时的情况下, 实现ORM有可能会引入不稳定性
> 第二, 也是最重要的一点是PHP已经提供了对DB的一个轻度封装的PDO, 我认为直接使用PDO, 会更加简单, 更加高效, 我不希望提供一个复杂的ORM包装, 鼓励大家去抛弃简单的PDO而使用ORM. 所以, 最初的时候, Yaf并不包含ORM.
> 诚然, ORM可以提高开发效率, 尤其对于一些简单应用, 所以我想在后续的Yaf的版本中, 会考虑加入ORM, 但是那也绝对会是一个简单的ORM, 类似于Yaf的内建视图引擎: Yaf_View_Simple, 简单可依赖.

显然，目前yaf是没有内置的操作数据库类了，那只能自己diy了，yaf的model规则是，类名以Model为后缀，放在放置在models文件夹下面
下面我们来Diy:

先在application.ini配置文件里添加数据库配置信息:

```
db.type=mysql
db.host=localhost
db.database=test
db.username=root
db.password=123
db.charset = utf8
db.log = false
db.collation=utf8_unicode_ci
db.prefix =
```

在models文件夹下面新建一个base.php文件:
`<?php
class baseModel
{

```
  protected $link;
  //构造函数
  public function __construct($db_config = array())
  {
```

```
    if(empty($db_config)){
        $db_config = Yaf_Application::app()->getConfig()->db;
    }
    $this->link = $this->connect($db_config["host"], $db_config["username"], $db_config["password"], $d
b_config["database"]);
}
//数据库连接
public function connect($dbhost, $dbuser, $dbpw, $dbname, $charset = 'utf8')
{
    $this->link = @mysqli_connect($dbhost, $dbuser, $dbpw, $dbname);
    if (!$this->link)
    {
        return "database error:" . mysqli_connect_errno();
    }
    if (!@mysqli_select_db($this->link, $dbname))
    {
        return 'database error';
    }
    mysqli_query($this->link,"set names " . $charset);
    return $this->link;
}
//查询
public function query($sql)
{
    $query = mysqli_query($this->link, $sql);
    return $query;
}
//获取全部记录
public function get_all($sql,$result_type = MYSQL_ASSOC) {
    $query = $this->query($sql);
    $i = 0;
    $rt = array();
    while($row =& mysqli_fetch_array($query,$result_type)) {
        $rt[$i]=$row;
        $i++;
    }
    return $rt;
}
public function add($data = array()){
    return true;
}
```

}
?>`

当然，上面只是一个简单的数据库操作演示，因为懒，只做了连接数据，然后一个执行sql的query()方法。到这里我们就可以直接在控制器里调用这个model去操作数据库了。

在控制器中调用baseModel操作

`<?php
class IndexController extends Yaf_Controller_Abstract {

```
public function indexAction() {
    Yaf_Dispatcher::getInstance()->disableView();
    $mod = new baseModel();
    $sql = 'select * from user';
    $data = $mod->get_all($sql);
```

```
    print_r($data);
}`
```

在控制器中使用$mod = new baseModel()实例化模型的时候，系统会自动加载model下面的base.php文件，无需手动加载。
如果我们在模型层还有其它的一些操作需求，那可以再扩展一下，比如说：

在刚才的models文件夹下我们可以再新建一个user.php
`<?php
class UserModel extends baseModel{

```
//再这里就可以再定义这个模型下面的操作方法
```

}
?>`

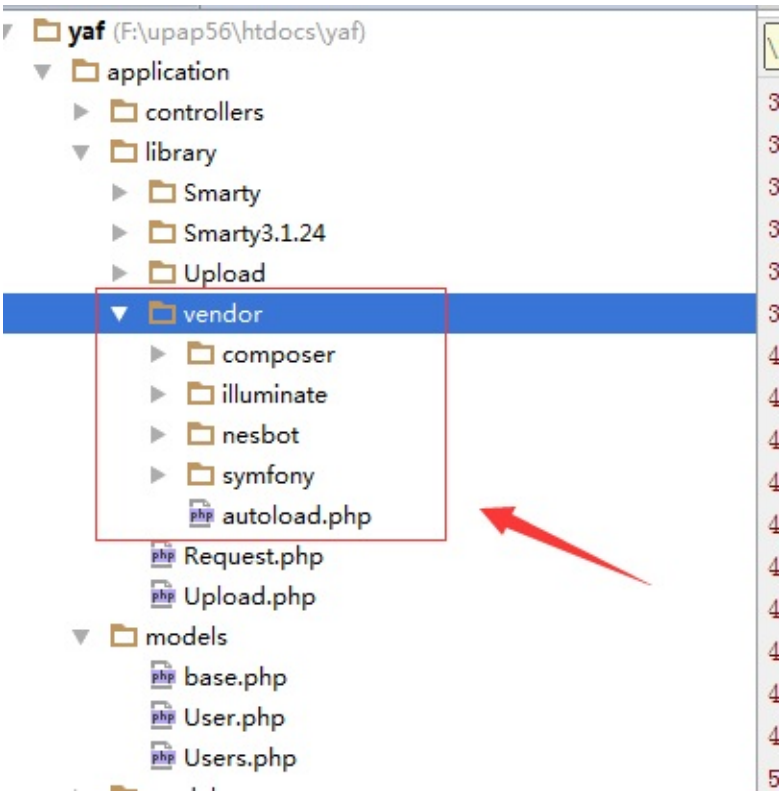
这个模型只要指定继承刚才定义的baseModel，它就拥有了baseModel里面的所有方法，在控制器中使用UserModel:
`<?php
class IndexController extends Yaf_Controller_Abstract {

```
public function indexAction() {
    $mod = new UserModel();
    $sql = 'select * from user';
    $data = $mod->get_all($sql);
    print_r($data);
}`
```

载入第三方的ORM

上面只是一个简单的model实现方法，大家可以再自行完善。
在一些项目中，ORM可以提高开发效率,我这里也尝试着载入lavarel框架中所使用的Eloquent ORM。
Eloquent ORM操作介绍：http://www.golaravel.com/laravel/docs/4.2/eloquent/
因为下载有点慢，我就直接从laravel5.1的包里面直接复制出的eloquent。

1、将文件夹放置到library下面,如下所示



2、在Bootstarp.php初始化eloquent

`<?php

Yaf_loader::import("/vendor/autoload.php");

use Illuminate\Container\Container;

use Illuminate\Database\Capsule\Manager as Capsule;

class Bootstrap extends Yaf_Bootstrap_Abstract{

```
private $config;
public function _initConfig() {
    $this->config = Yaf_Application::app()->getConfig();
    Yaf_Registry::set("config", $this->config);
}
//载入数据库ORM
public function _initDatabase()
{
    $database = array(
        'driver'    => $this->config->db->type,
        'host'      => $this->config->db->host,
        'database'  => $this->config->db->database,
        'username'  => $this->config->db->username,
        'password'  => $this->config->db->password,
        'charset'   => $this->config->db->charset,
        'collation' => $this->config->db->collation,
        'prefix'    => $this->config->db->prefix,
    );
    $capsule = new Capsule;
    // 创建链接
    $capsule->addConnection($database);
    // 设置全局静态可访问
```

```
    $capsule->setAsGlobal();
    // 启动Eloquent
    $capsule->bootEloquent();
}`
```

在models文件夹下新建UsersModel的Users.php:
`<?php
use Illuminate\Database\Eloquent\Model as Mymodel;
class UsersModel extends Mymodel{

```
protected $table = 'user';
```

}
?>`

在控制器中调用：
`<?php
class IndexController extends Yaf_Controller_Abstract {

```
public function indexAction() {
    Yaf_Dispatcher::getInstance()->disableView();
    $mod = new UsersModel();
    $data = $mod->find(1)->toArray();
    print_r($data);
}`
```

更多关于Eloquent ORM的操作介绍可移步：http://www.golaravel.com/laravel/docs/4.2/eloquent/

命令行cli模式

官方文档地址：http://yaf.laruence.com/manual/yaf.incli.times.html
感觉文档写得有点简单，不好理解，这里聊下我是怎么用的yaf命令行。

简述

在yaf中用到命令行大多是为了跑Crontab，首先，为了更好的与web区分(配置文件,argc、argv判断等等).重新创建一个入口文件是比较好的做法。

方法一

在项目根目录下面新建一个cli.php文件：

```
<?php
define("APP_PATH",  realpath(dirname(__FILE__)));
$app = new Yaf_Application(APP_PATH . "/conf/application.ini");
$app->getDispatcher()->dispatch(new Yaf_Request_Simple());
```

这样入口文件就完成了。

然后再新建一个接收命令和操作的控制器Crontab.php:
`<?php
class CrontabController extends Yaf_Controller_Abstract {

```
public function init()
{
    Yaf_Dispatcher::getInstance()->disableView();
}
public function indexAction($username = '')
{
    //to do a crontab
        echo 'we get the name is : '.$username;
}
```

}
?>`
接下来，我们在命令行中调用。在命令行中切换到你的项目目录，就是cli.php所在目录，然后输入如下命令：

> php cli.php request_uri="/crontab/index"

OK,是不是在命令行看到了输出的字符串。

request_uri=“ /crontab/index” 中的路径便是Controller的路由路径.
在例子里指向/controllers/Crontab.php 中的 indexAction().

方法二

还有一种方法，通过Yaf_Application::execute(..)去实现。

先看一下这个函数的定义：

> public void Yaf_Application::execute (callable $entry , string $...)
> This method is typically used to run Yaf_Application in a crontab work. Make the crontab work can also use the autoloader and Bootstrap mechanism.

第一参数需要定义一个回调函数,也可以是一个类中的某个函数。

示例：$application->execute(“main”, $argc, $argv);

或

$application->execute(array(“Class”,” Method”), $argc, $argv);

后面的参数为一个可变列表，值为你希望传入的参数。

如些，我们将刚才新建的cli.php文件改写成：

```
<?php
define("APP_PATH",  realpath(dirname(__FILE__)));
$app = new Yaf_Application(APP_PATH . "/conf/application.ini");
$app->bootstrap()->execute(array('CrontabController','indexAction'),'wulei');
```

其中如果你需要用bootstrap的初始化的，可以保留，如果不需要的话，也可以把bootstrap去掉。

异常和错误

Yaf实现了一套错误和异常捕获机制, 主要是对常见的错误处理和异常捕获方法做了一个简单抽象, 方便应用组织自己的错误统一处理逻辑。
前题是需要配置过或是在程序中启用
1、配置

> application.dispatcher.throwException=1
> application.dispatcher.catchException=1

2、在程序中启用

> Yaf_Dispatcher::throwException(true)

在application.dispatcher.catchException(配置文件, 或者可通过Yaf_Dispatcher::catchException(true))开启的情况下, 当Yaf遇到未捕获异常的时候, 就会把运行权限, 交给当前模块的Error Controller的Error Action动作, 而异常或作为请求的一个参数, 传递给Error Action.

新建一个Error Controller
`<?php
class ErrorController extends Yaf_Controller_Abstract
{

```
public function errorAction($exception)
{
    assert($exception);
    $this->getView()->assign("code", $exception->getCode());
    $this->getView()->assign("message", $exception->getMessage());
    $this->getView()->assign("line", $exception->getLine());
}
```

}
?>`

新建一个Error显示模板文件
`

```
<meta charset="utf-8">
<title>Error Page <{$code}></title>
<style>
    body{background-color:#f0c040}
```

```
    h2{color:#fafafa}
</style>
```

```
<h2>Error Page</h2>
<p>Error Code:<{$code}></p>
<p>Error Message:<{$message}></p>
<p>Error Line:<{$line}></p>
```

`

在Bootstrap.php中新建一个error_handler方法

`public static function error_handler($errno, $errstr, $errfile, $errline)
{

```
if (error_reporting() === 0) return;
throw new ErrorException($errstr, 0, $errno, $errfile, $errline);
```

}`

在Bootstrap.php中初始化ErrorHandler

`public function _initErrorHandler(Yaf_Dispatcher $dispatcher)
{

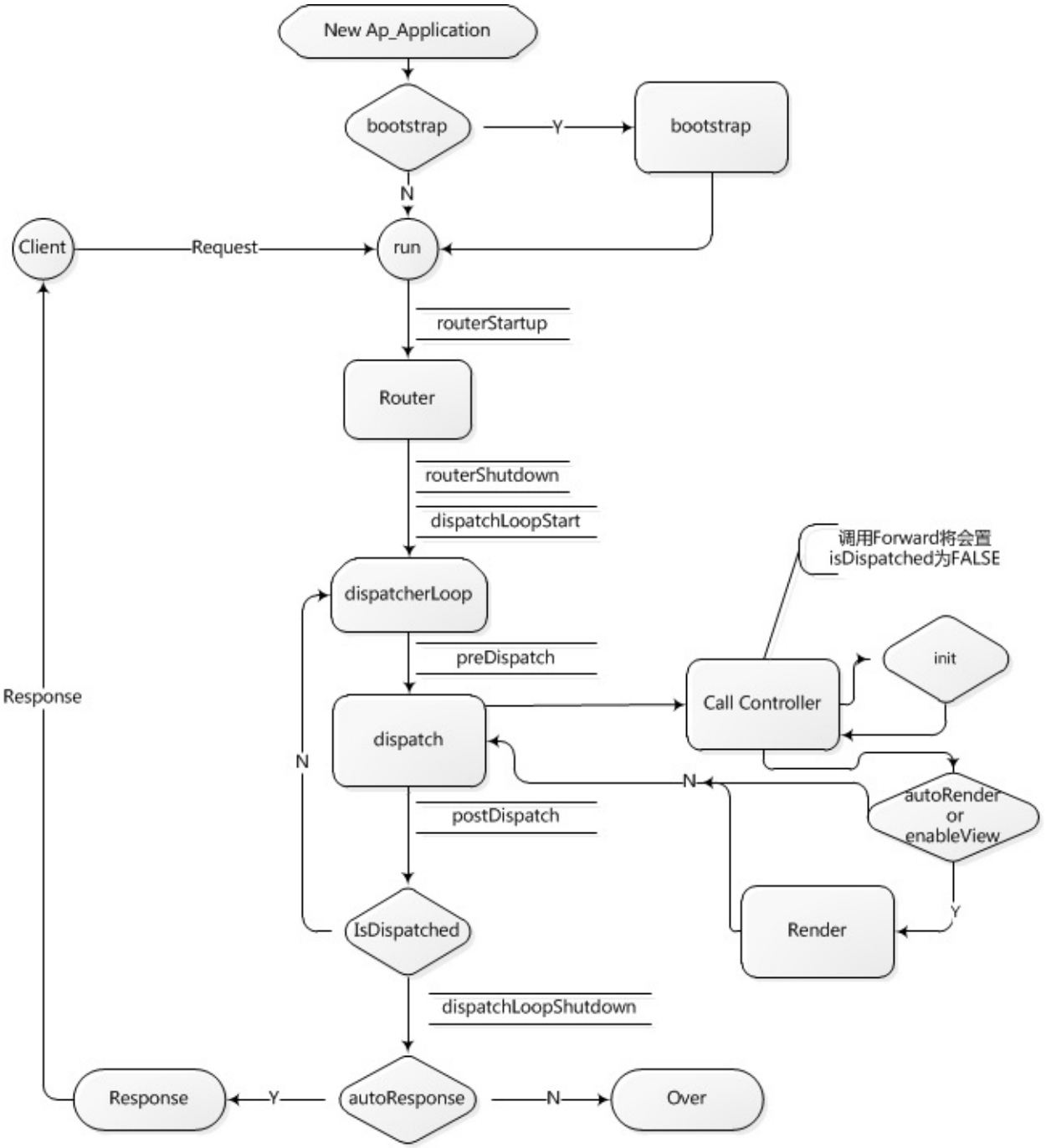
```
$dispatcher->setErrorHandler(array(get_class($this),'error_handler'));
```

}`

这样当有有程序异常时会转到ErrorController

路由和分发

在说路由之前我们先看一张yaf的整体流程图如下:



简单的理解

就我的理解来说，路由分发过程的执行动作是，获取用户请求的URl，根据路由规则解析这个URL，得到

module、controller、action、param、query，根据获得的module和controller去载入控制器，执行对应的action方法。

插件钩子
路由器也有插件钩子,就是routerStartup和routerShutdown,他们在路由解析前后分别被调用.

设置路由的方法

1、先添加配置

```
routes.regex4.type="regex"
routes.regex4.match="#^/news/([^/])/([^/])#"
routes.regex4.route.controller=news
routes.regex4.route.action=detail
routes.regex4.map.1=id
routes.regex4.map.2=sort
```

2、在Bootstap.php中添加路由配置
`<?php
class Bootstrap extends Yaf_Bootstrap_Abstract{

```
public function _initRoute(Yaf_Dispatcher $dispatcher) {
        $router = Yaf_Dispatcher::getInstance()->getRouter();
        $router->addConfig(Yaf_Registry::get("config")->routes);
}
```

}`

3、添加接收的控制器
`<?php
class NewsController extends Yaf_Controller_Abstract {

```
public function init()
{
    Yaf_Dispatcher::getInstance()->disableView();
}
public function detailAction($id = 0,$sort = '')
{
    print_r($this->getRequest()->getParams());
    echo 'News Detail:'.$id.',sort:'.$sort;
}
```

}
?>`

4、访问url: yourhost/news/78/create_time

当访问这个url，yaf先根据我们的路由规则解析出默认的module,news控制器,detailAction,第一个参数id,第二个参数，sort。

我们来分析一下解析流程：
Yaf_Application::app()->bootstrap()->getDispatcher->dispatch();

1.在yaf_dispatcher_route中，完成路由解析，得到module=''，controller=news，action=detail

2.在yaf_dispatcher_fix_default中，通过其处理得到module=index，controller=news，action=detail

3.在2中完成之后，通过如果有hook机制，就会执行插件钩子：routerShutdown

4.在yaf_internal_autoload中完成自动加载类文件，application/controllers/News.php

5.执行detailAction

在Bootstrap.php中配置路由规则

上面就是一个简单的通过正则的方式来设置路由的示例，我们还可以直接在Bootstrap.php添加我们的路由规则：

` public function _initRoute(Yaf_Dispatcher $dispatcher) {

```
$router = Yaf_Dispatcher::getInstance()->getRouter();
$router->addConfig(Yaf_Registry::get("config")->routes);
//在刚才的示例里添加上下面两行
$route = new Yaf_Route_Simple("m", "c", "a");
$router->addRoute("simple", $route);
```

} `

测试一下
我们就可以尝试用 yourhost?c=news&a=detail 访问你的newsController,detailAction了。

Yaf_Route_Simple

上面是Yaf_Route_Simple的一个示例
Yaf_Route_Simple是基于请求中的query string来做路由的, 在初始化一个Yaf_Route_Simple路由协议的时候, 我们需要给出3个参数, 这3个参数分别代表在query string中Module, Controller, Action的变量名，它也可以直接在配置信息里设置

```
routes.simple.type="simple"
routes.simple.controller=c
routes.simple.module=m
routes.simple.action=a
```

更多关于路由的信息可以参见官方文档：http://www.laruence.com/manual/yaf.routes.static.html