**Laravel 中间件使用及源码分析**

姓名：李嘉琪

学号：2017104142

时间：2018年10月10号

1. 中间件的研究背景及介绍

一般说来，中间件有两层含义。从狭义的角度，中间件意指 Middleware，它是表示网络环 境下处于操作系统等系统软件和应用软件之间的一种起连接作用的分布式软件，通过 API 的形式 提供一组软件服务，可使得网络环境下的若干进程、程序或应用可以方便的交流信息和有效的进 行交互与协同。简言之，中间件主要解决异构网络环境下分布式应用软件的通信、互操作和协同 问题，它可屏蔽并发控制、事务管理和网络通信等各种实现细节，提高应用系统的易移植性、适 应性和可靠性。从广义的角度，中间件在某种意义上可以理解为中间层软件，通常是指处于系统 软件和应用软件之间的中间层次的软件，其主要目的是对应用软件的开发提供更为直接和有效的支撑。

对于一个Web应用来说，在一个请求真正处理前，我们可能会对请求做各种各样的判断，然后才可以让它继续传递到更深层次中。  
而如果我们用if else这样子来，一旦需要判断的条件越来越来多，会使得代码更加难以维护，系统间的耦合会增加，而中间件就可以解决这个问题。  
我们可以把这些判断独立出来做成中间件，可以很方便的过滤请求。

**1.1 Laravel中的中间件的特点及使用**

所有的中间件都放在 app/Http/Middleware 目录内。

要创建一个新的中间件，则可以使用 make:middleware 这个 Artisan 命令。此命令将会在 app/Http/Middleware 目录内设定一个名称为 OldMiddleware 的类。

分为前置中间件/后置中间件

全局中间件每个 HTTP 请求都经过，设置app/Http/Kernel.php 的 $middleware 属性

路由中间件指派中间件给特定路由。设置app/Http/Kernel.php 的$routeMiddleware属性。

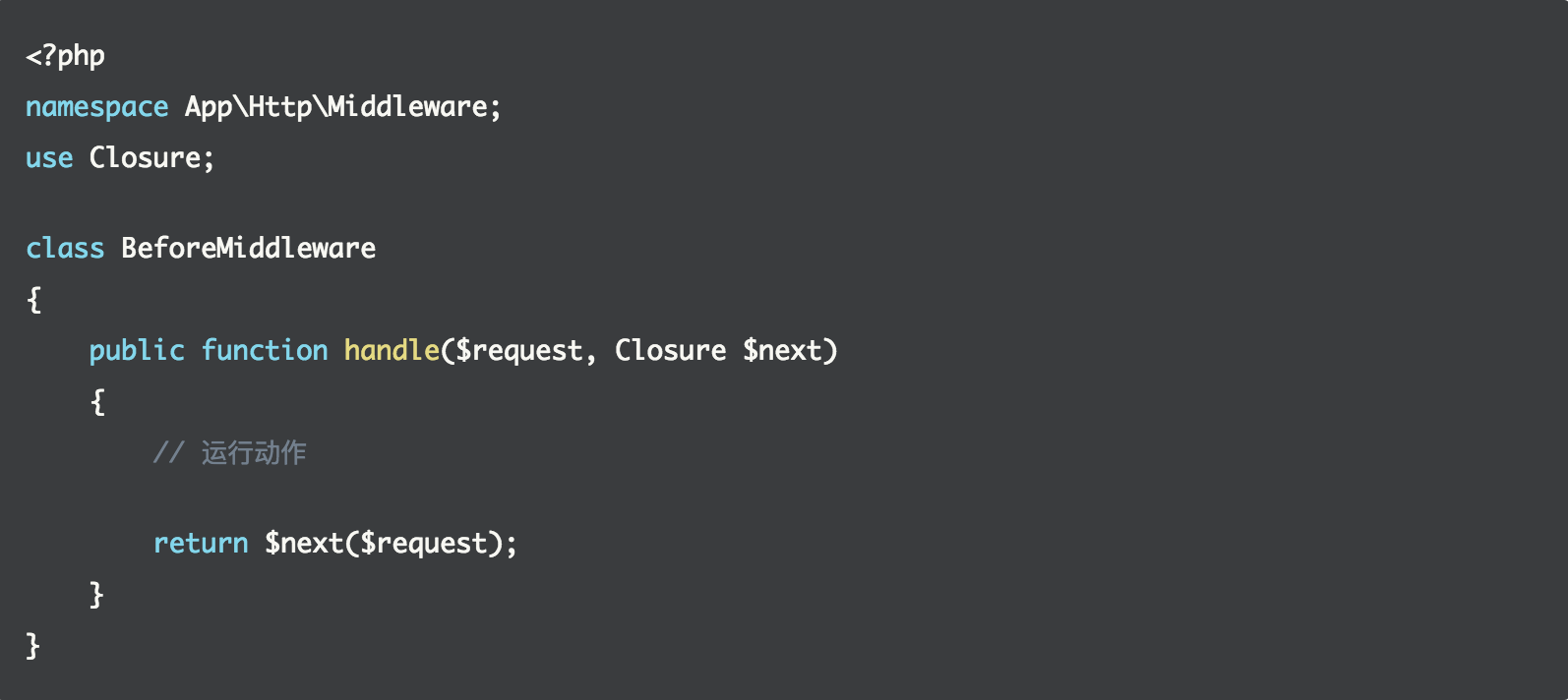
中间件可以接收自定义的参数

Terminable中间件，可以在 HTTP 响应被发送到浏览器之后才运行。

### **1.2 Laravel中间件的分类**

前置中间件 / 后置中间件

前置中间件：前置中间件运行的时间点是在每一个请求处理之前



后置中间件：后置中间件运行的时间点是在请求处理之后



### **使用中间件**

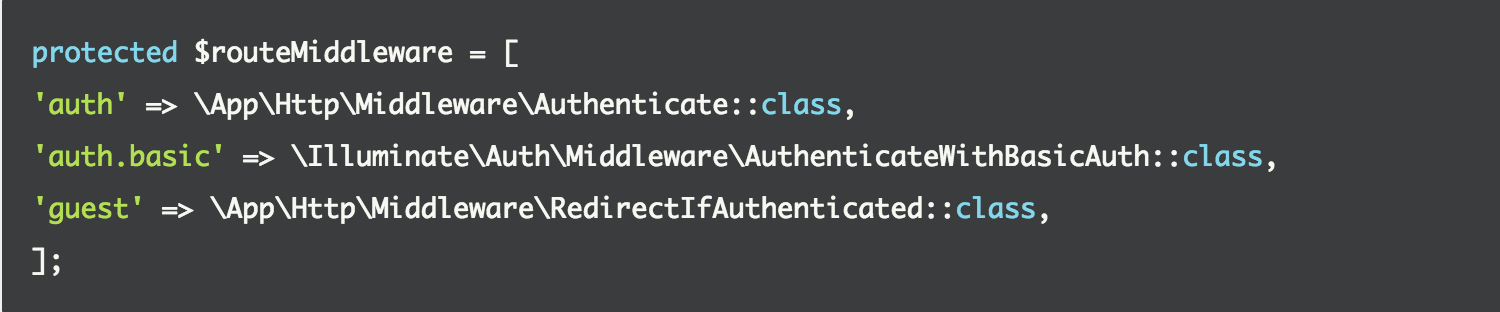
#### **全局中间件**

若是希望每个 HTTP 请求都经过一个中间件，只要将中间件的类加入到 app/Http/Kernel.php 的 $middleware 属性清单列表中。

#### **路由指派中间件**

如果你要指派中间件给特定路由，你得使用路由指派中间件，两个步骤完成。

1. 先在 app/Http/Kernel.php 给中间件设置一个好记的键。默认情况下，这个文件内的 $routeMiddleware 属性已包含了Laravel目前设置的中间件，你只需要在清单列表中加上一组自定义的键即可。



1. 中间件一旦在 HTTP kernel 文件内被定义，即可在路由选项内使用 middleware 键值指定。



**二．Laravel中间件代码实现**

2.1 触发中间件的代码

在Laravel中，中间件的实现其实是依赖于Illuminate\Pipeline\Pipeline这个类实现的，我们先来看看触发中间件的代码。

// kernell类handle方法调用。

$response = $this->sendRequestThroughRouter($request);

/\*\* \* Send the given request through the middleware / router.

\* \* @param \Illuminate\Http\Request $request

\* @return \Illuminate\Http\Response

\*/

protected function sendRequestThroughRouter($request)

{

// 在app容器中绑定一个request实例。

$this->app->instance('request', $request);

// 删除门面（静态调用）产生的实例。

Facade::clearResolvedInstance('request');

// 使用容器的bootstrapWith方法调用$bootstrappers数组中类。

$this->bootstrap(); // 中间件的核心代码

return (new Pipeline($this->app))

->send($request) ->through($this->

app->shouldSkipMiddleware() ? [] : $this->middleware) ->

then($this->dispatchToRouter()); }

/\*\* \* Bootstrap the application for HTTP requests.

\* \* @return void \*/

public function bootstrap()

{

if (! $this->app->hasBeenBootstrapped())

{ $this->app->bootstrapWith($this->bootstrappers()); } }

/\*\* \* The bootstrap classes for the application.

\* \* @var array \*/

protected $bootstrappers = [

'Illuminate\Foundation\Bootstrap\DetectEnvironment', 'Illuminate\Foundation\Bootstrap\LoadConfiguration', 'Illuminate\Foundation\Bootstrap\ConfigureLogging', 'Illuminate\Foundation\Bootstrap\HandleExceptions', 'Illuminate\Foundation\Bootstrap\RegisterFacades', 'Illuminate\Foundation\Bootstrap\RegisterProviders', 'Illuminate\Foundation\Bootstrap\BootProviders',

];

**2.2 核心代码**

**函数array\_reduce**

了解核心代码以前，需要先了解一下[array\_reduce](http://php.net/manual/zh/function.array-reduce.php)函数的用法。

重点，回调函数sum，接收两个参数。

****carry****  
携带上次迭代里的值（如果本次迭代是第一次，那么这个值是6 initial）。

****item****  
携带了本次迭代的值。

<?php

function sum($carry, $item)

{

return $carry + $item;

}

$a = array(1, 2, 3, 4, 5);

// 21 6+1+2+3+4+5

echo array\_reduce($a, "sum",6);

**模拟laravel中间件的代码**

* array\_reduce最终返回一个可执行的闭包。
* call\_user\_func 执行了这个闭包，而这个闭包中两个变量$carry、$item。
* $carry是上一个循环元素返回的可执行闭包，$item是当前数组元素。
* 上一个循环元素返回的可执行闭包,又使用了$carry、$item。依次循环。
* 类中的$next变量代指$carry，所以在$next()之前的执行逻辑先于$carry执行，之后的逻辑就晚于$carry执行。

<?Php

interface Milldeware

{

public static function handle(Closure $next);

}

class test1 implements Milldeware

{

public static function handle(Closure $next)

{

echo 'test11' . PHP\_EOL; $next();

}

}

class test2 implements Milldeware  
{  
public static function handle(Closure $next)  
{  
echo 'test21' . PHP\_EOL;  
$next();  
}  
}  
class test3 implements Milldeware  
{

public static function handle(Closure $next)

{

echo 'test31' . PHP\_EOL;

$next();

echo 'test32' . PHP\_EOL;

}

}  
class test4 implements Milldeware  
{  
public static function handle(Closure $next)  
{  
$next();  
echo 'test42' . PHP\_EOL;  
}  
}

class test5 implements Milldeware  
{  
public static function handle(Closure $next)  
{  
echo 'test51' . PHP\_EOL;  
$next();  
echo 'test52' . PHP\_EOL;  
}  
}  
class test6 implements Milldeware  
{  
public static function handle(Closure $next)  
{  
echo 'test61' . PHP\_EOL;  
$next();  
}  
}  
function then()  
{  
$pipe = [  
'test1',  
'test2',  
'test3',  
'test4',  
'test5',  
'test6'  
];

$firstSlice = function () {

echo 'firstSlice' . PHP\_EOL;

};

$pipe = array\_reverse($pipe);

$callback = array\_reduce($pipe, function ($carry, $item) {

return function () use ($carry, $item) {

return $item::handle($carry);

};

}, $firstSlice);

//var\_dump($callback);die;

call\_user\_func($callback);

}  
then();

/\*  
test11  
test21  
test31  
test51  
test61  
firstSlice  
test52  
test42  
test32  
/

### 真实laravel的代码

class Pipeline implements PipelineContract  
{  
/\*\*

* The container implementation.

@var \Illuminate\Contracts\Container\Container  
\*/  
protected $container;

/\*\*

* The object being passed through the pipeline.

@var mixed  
\*/  
protected $passable;

/\*\*

* The array of class pipes.

@var array  
\*/  
protected $pipes = [];

/\*\*

* The method to call on each pipe.

@var string  
\*/  
protected $method = 'handle';

/\*\*

* Create a new class instance.
* @param \Illuminate\Contracts\Container\Container $container

@return void  
\*/  
public function \_\_construct(Container $container)  
{  
$this->container = $container;  
}

/\*\*

* Set the object being sent through the pipeline.
* 设置一个通过管道传输的对象。
* @param mixed $passable

@return $this  
\*/  
public function send($passable)  
{  
$this->passable = $passable;

$pipes = array\_reverse($this->pipes);

return call\_user\_func(

array\_reduce($pipes, $this->getSlice(), $firstSlice), $this->passable

);

/\*\*

* Get a Closure that represents a slice of the application onion.

@return \Closure  
\*/  
protected function getSlice()  
{  
return function ($stack, $pipe) {  
return function ($passable) use ($stack, $pipe) {  
// If the pipe is an instance of a Closure, we will just call it directly but  
// otherwise we will resolve the pipes out of the container and call it with  
// the appropriate method and arguments, returning the results back out.  
// 如果是Closure直接调用它，否则处理一下变成可调用的Closure。  
if ($pipe instanceof Closure) {  
return call\_user\_func($pipe, $passable, $stack);  
} else {  
// parsePipeString的方式决定了我们在使用中间件时的方式，比如参数的传入方式。  
// 文档中“在路由中可使用冒号 : 来区隔中间件名称与指派参数，多个参数可使用逗号作为分隔”  
list($name, $parameters) = $this->parsePipeString($pipe);  
// $pipe通过parsePipeString分解，然后使用$this->container->make($name)获取实例，我们看出只要是可以被make获取的实例都可以当做管道的处理栈。  
return call\_user\_func\_array([$this->container->make($name), $this->method],  
array\_merge([$passable, $stack], $parameters));  
}  
};  
};  
}

/\*\*

* Get the initial slice to begin the stack call.
* @param \Closure $destination

@return \Closure  
\*/  
protected function getInitialSlice(Closure $destination)  
{  
return function ($passable) use ($destination) {  
return call\_user\_func($destination, $passable);  
};  
}

/\*\*

* Parse full pipe string to get name and parameters.
* @param string $pipe

@return array  
\*/  
protected function parsePipeString($pipe)  
{  
// 通过这行代码我们看出，$pipe是字符串，可以是用‘:’来分隔中间件名称与指派参数。  
list($name, $parameters) = array\_pad(explode(':', $pipe, 2), 2, []);

// 通过这行代码我们看出：多个参数可使用逗号作为分隔。if (is\_string($parameters)) {

$parameters = explode(',', $parameters);}

return [$name, $parameters];

}  
}

#### 生成最终匿名函数的过程

//array\_reduce执行//第一次时得到如下简化的匿名函数返回，将会继续作为第一个参数进行迭代：

object(Closure)#id (1) {

["static"]=>

array(2) {

["stack"]=>

object(Closure)#1 (0) { // $this->prepareDestination($destination)

}

["pipe"]=>

string(15) "Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull"

}

}//第二次：

object(Closure)#id (1) {

["static"]=>

array(2) {

["stack"]=>

object(Closure)#id (1) {

["static"]=>

array(2) {

["stack"]=>

object(Closure)#1 (0) { // $this->prepareDestination($destination)

}

["pipe"]=>

string(15) "Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull"

}

}

["pipe"]=>

string(15) "App\Http\Middleware\TrimStrings"

}

}//第三次：

object(Closure)#id (1) {

["static"]=>

array(2) {

["stack"]=>

object(Closure)#id (1) {

["static"]=>

array(2) {

["stack"]=>

object(Closure)#id (1) {

["static"]=>

array(2) {

["stack"]=>

object(Closure)#1 (0) { // $this->prepareDestination($destination)

}

["pipe"]=>

string(15) "Illuminate\Foundation\Http\Middleware\ConvertEmptyStringsToNull"

}

}

["pipe"]=>

string(15) "App\Http\Middleware\TrimStrings"

}

}

["pipe"]=>

string(15) "Illuminate\Foundation\Http\Middleware\ValidatePostSize"

}

}

**terminate方法的调用时机**

在Http的Kernel类中有个terminate方法，此方法代码如下：

/\*\*

\* Call the terminate method on any terminable middleware.

\*

\* @param \Illuminate\Http\Request $request

\* @param \Illuminate\Http\Response $response

\* @return void

\*/

public function terminate($request, $response)

{

$middlewares = $this->app->shouldSkipMiddleware() ? [] : array\_merge(

$this->gatherRouteMiddlewares($request),

$this->middleware

);

foreach ($middlewares as $middleware) {

list($name, $parameters) = $this->parseMiddleware($middleware);

$instance = $this->app->make($name);

if (method\_exists($instance, 'terminate')) {

$instance->terminate($request, $response);

}

}

$this->app->terminate();

}

通过$instance->terminate($request, $response);代码可以看出，将$response传入了中间件的terminate方法。