使用Symfony2的组件创建自己的PHP框架（第九部分：创建事件与处理事件）

发表回复

英文原文地址：http://fabien.potencier.org/article/58/create-your-own-framework-on-top-of-the-symfony2-components-part-9

我们的框架依然缺少作为好框架必备的一个特点：扩展性。拥有扩展性意味着，开发者可以很方便的通过拦截（hook）的方式，修改请求被处理的过程。

我们说的拦截是什么东西呢？验证或者缓存就是两个例子（译者注：其实请求的处理就像通过一层层的筛子。比如访问一个需要做登陆验证的url，那么我们可以设计这么一个筛子，它将判断请求来源是否已登陆，如果没有登录请求将被拦截住，转向登录页面。如果一个url设计为可以被缓存，一个请求过来以后，缓存筛子判断这个url是否已经被缓存，如果是，这个筛子直接拦截此次请求不继续往下处理，而是直接把缓存的内容发送出去）。为了灵活性，拦截程序必须是即插即用型（plug and play）的。根据不同的需求，你所“注册”的拦截程序肯定有别于其他拦截程序。许多软件都有类似的概念，比如说Wordpress或者Drupal。在一些语言里，甚至会有相关的标准。比如Ruby的Rack和Python的WSGI。

因为在PHP里面没有相关的标准，我们将使用著名的设计模式“观察者模式”，来将各种拦截模块连接到我们的框架中。sf2的事件调度(EventDispatcher)组件为此模式做了一个轻量级的实现。

```
{
    "require": {
        "symfony/class-loader": "2.1.*",
        "symfony/http-foundation": "2.1.*",
        "symfony/routing": "2.1.*",
        "symfony/http-kernel": "2.1.*",
        "symfony/event-dispatcher": "2.1.*"
    },
    "autoload": {
        "psr-0": { "Simplex": "src/", "Calendar": "src/" }
    }
}

```

此模块如何工作呢？作为此组件的核心的调度器，将对每一个连接过它的监听器（listener）做出事件提醒。或

者这么说：你的代码将在某个事件发生的时候调用调度器，而调度器将提醒每一个监听器刚刚发生了什么事件，每个监听器收到消息后，将对此事件做出自己的不同处理行为。

举一个例子，让我们透明的为每一个响应都添加google的网站访问分析代码GA。

要达成此目的，我们得让框架在返回相应对象之前做一次事件分发：

```php
<?php

// example.com/src/Simplex/Framework.php

namespace Simplex;

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\Routing\Matcher\UrlMatcherInterface;
use Symfony\Component\Routing\Exception\ResourceNotFoundException;
use Symfony\Component\HttpKernel\Controller\ControllerResolverInte
use Symfony\Component\EventDispatcher\EventDispatcher;

class Framework
{
    protected $matcher;
    protected $resolver;
    protected $dispatcher;

    public function __construct(
        EventDispatcher $dispatcher,
        UrlMatcherInterface $matcher,
        ControllerResolverInterface $resolver
    ) {
        $this->matcher = $matcher;
        $this->resolver = $resolver;
        $this->dispatcher = $dispatcher;
    }

    public function handle(Request $request)
    {
        try {
            $request->attributes->add($this->matcher->match($reque
```

```
34
35              $controller = $this->resolver->getController($request)
36              $arguments = $this->resolver->getArguments($request, $
37
38              $response = call_user_func_array($controller, $argumer
39          } catch (ResourceNotFoundException $e) {
40              $response = new Response('Not Found', 404);
41          } catch (\Exception $e) {
42              $response = new Response('An error occurred', 500);
43          }
44
45          // dispatch a response event
46          $this->dispatcher->dispatch('response', new ResponseEvent(
47
48          return $response;
49      }
50  }
51
```

框架每次处理请求的时候，都将分发一个类型为ResponseEvent的事件：

```
1   <?php
2
3   // example.com/src/Simplex/ResponseEvent.php
4
5   namespace Simplex;
6
7   use Symfony\Component\HttpFoundation\Request;
8   use Symfony\Component\HttpFoundation\Response;
9   use Symfony\Component\EventDispatcher\Event;
10
11  class ResponseEvent extends Event
12  {
13      private $request;
14      private $response;
15
16      public function __construct(Response $response, Request $reque
17      {
18          $this->response = $response;
19          $this->request = $request;
```

```
20      }
21
22      public function getResponse()
23      {
24          return $this->response;
25      }
26
27      public function getRequest()
28      {
29          return $this->request;
30      }
31  }
32
```

最后一步我们将创建分发器，并为此添加一个监听器：

```
1   <?php
2
3   // example.com/web/front.php
4
5   require_once __DIR__.'/../vendor/.composer/autoload.php';
6
7   // ...
8
9   use Symfony\Component\EventDispatcher\EventDispatcher;
10
11  $dispatcher = new EventDispatcher();
12  $dispatcher->addListener('response', function (Simplex\ResponseEve
13      $response = $event->getResponse();
14
15      if ($response->isRedirection()
16          || ($response->headers->has('Content-Type')
17          && false === strpos($response->headers->get('Content-Type'
18          || 'html' !== $event->getRequest()->getRequestFormat()
19      ) {
20          return;
21      }
22
23      $response->setContent($response->getContent().'GA CODE');
24  });
25
```

```
$framework = new Simplex\Framework($dispatcher, $matcher, $resolve
$response = $framework->handle($request);

$response->send();

```

上面代码只是为了说明怎么加代码，若要真的添加*GA*代码你得自己把真实的代码写上去（译者注：而且用这种方式添加GA在真正实践中也不是一个好方法，像GA这种HTML代码，直接放在布局文件中更好）

如你所见，`addListener()`方法把将response事件和一个php回调函数联系在了一起。事件的名字必须跟`dispatch()`方法里提到的事件名字一样。

在监听代码中，我们只为不跳转的响应添加GA代码，而且响应的类型必须是html类型（此代码演示了操作请求或者响应对象是多么的手到擒来的事情）（译者注：你肯定不想为类似生成图片的响应添加GA代码）

让我们在为同样的事件添加另外一个监听者，此监听者检查响应是否有Content-length头，没有便添加一个：

```
$dispatcher->addListener('response', function (Simplex\ResponseEver
    $response = $event->getResponse();
    $headers = $response->headers;

    if (!$headers->has('Content-Length') && !$headers->has('Transfe
        $headers->set('Content-Length', strlen($response->getConter
    }
});

```

添加监听者时需要注意添加的顺序，否则你有可能会得到错误的Content-length值。监听者添加顺序很重要，但从默认设置来说，所有的监听者都是一样的优先级，值为0。要想让某个监听者优先执行，可以将优先级设置为一个正数，也可以为低优先级的监听控件设置优先级为负数。比如说设置content-length的监听控件，我们可以设置它为-255，表示最后执行。

```
$dispatcher->addListener('response', function (Simplex\ResponseEver
    $response = $event->getResponse();
    $headers = $response->headers;

    if (!$headers->has('Content-Length') && !$headers->has('Transfe
        $headers->set('Content-Length', strlen($response->getConter
```

```
7       }
8   }, -255);
9
```

当你创建框架的时候，请仔细思考如何设计优先级（比如给内部监听控件预留一些优先级数），以及为此做好文档

让我们将代码重构一下，把GA监听控件放在属于他自己的类里：

```
1   <?php
2
3   // example.com/src/Simplex/GoogleListener.php
4
5   namespace Simplex;
6
7   class GoogleListener
8   {
9       public function onResponse(ResponseEvent $event)
10      {
11          $response = $event->getResponse();
12
13          if ($response->isRedirection()
14              || ($response->headers->has('Content-Type')
15              && false === strpos($response->headers->get('Content-T
16              || 'html' !== $event->getRequest()->getRequestFormat()
17          ) {
18              return;
19          }
20
21          $response->setContent($response->getContent().'GA CODE');
22      }
23  }
24
```

其他的监听也做同样处理：

```
1   <?php
2
3   // example.com/src/Simplex/ContentLengthListener.php
```

```
4
5  namespace Simplex;
6
7  class ContentLengthListener
8  {
9      public function onResponse(ResponseEvent $event)
10     {
11         $response = $event->getResponse();
12         $headers = $response->headers;
13
14         if (!$headers->has('Content-Length')
15             && !$headers->has('Transfer-Encoding')
16         ) {
17             $headers->set('Content-Length', strlen($response->getC
18         }
19     }
20 }
21
```

如此以来我们的前端控制器代码可改成如下样子：

```
1  $dispatcher = new EventDispatcher();
2  $dispatcher->addListener('response', array(
3      new Simplex\ContentLengthListener(),
4      'onResponse'
5  ), -255);
6  $dispatcher->addListener('response', array(new Simplex\GoogleLister
7
```

虽然相关代码已被漂亮的封装起来了，但依然还残留一个小问题：优先级相关的代码被硬编码在了前段控制器里，而不是监听器自己来控制。每一个应用程序里你都得自己记着设置合适的优先级。除此之外，监听器所绑定的事件名字也暴露在前段控制器里，这意味着如果我们要重构监听器时，所有相关的应用程序都得所相应的修改。当然，有一个办法可以解决这个问题，使用订阅的方式来代替监听的方式：

```
1  $dispatcher = new EventDispatcher();
2  $dispatcher->addSubscriber(new Simplex\ContentLengthListener());
3  $dispatcher->addSubscriber(new Simplex\GoogleListener());
4
```

“订阅者”知道所有的事件，并且可通过`getSubscribedEvents()`方法给分发器传递信息。让我们看看新版本的GA监听器：

```php
<?php

// example.com/src/Simplex/GoogleListener.php

namespace Simplex;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class GoogleListener implements EventSubscriberInterface
{
    // ...

    public static function getSubscribedEvents()
    {
        return array('response' => 'onResponse');
    }
}

```

以及新版Content-length监听器：

```php
<?php

// example.com/src/Simplex/ContentLengthListener.php

namespace Simplex;

use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class ContentLengthListener implements EventSubscriberInterface
{
    // ...

    public static function getSubscribedEvents()
    {
        return array('response' => array('onResponse', -255));
    }
```

```
17 }
18
```

一个订阅者可以拥有多个监听者，以及处理多种事件

为了让框架更加的灵活，不要犹豫，给框架多添加点事件吧，当然为了让框架功能更给力，添加更多的监听器那也是必须的，直到你感觉合适了位置，然后我们可以更进一步改进我们的代码。