Capítulo 20. Zend_Layout

Traduzido por Flávio Gomes da Silva Lisboa (versão 1.5.0 do Zend Framework)

Sumário

20.1. Introdução

`Zend_Layout` implementa um clássico padrão de projeto Two Step View[1], permitindo aos desenvolvedores empacotar o conteúdo da aplicação dentro de outra view, geramente representando o template do site. Tais templates são freqüentemente denominados de layouts por outros projetos, e Zend Framework adotou esse termo por consistência.

Os objetivos globais de `Zend_Layout` são os seguintes:

- Automatizar seleção e renderização de layouts quando usado com os componentes MVC do Zend Framework.
- Fornecer escopo separado para variáveis relacionadas ao layout e ao conteúdo.
- Permitir configuração, incluindo o nome do layout, a resolução do script de layout (inflection), e o caminho do script de layout.
- Permitir a desabilitação de layouts, mudança de scripts de layout, e outros estados; permitir essas ações de dentro de action controllers e view scripts.
- Seguir as mesmas regras de resolução de script (inflection) que ViewRenderer, mas permitir também o uso de regras diferentes.
- Permitir o uso sem os componentes MVC do Zend Framework.

20.2. Guia Rápido Zend_Layout

Há dois casos de uso primários para `Zend_Layout`: com o MVC do Zend Framework MVC, e sem.

1 Esse padrão foi criado por Martin Fowler

20.2.1. Scripts de Layout

Em ambos os casos, entretanto, você precisará criar um script de layout. Scripts de layout simplesmente utilizam Zend_View (ou qualquer outra implementação de view que você esteja usando). Variáveis de layout são registradas com um marcador `Zend_Layout`, e podem ser acessadas via helper de marcador ou buscando-as como propriedades de objeto do objeto layout via helper de layout.

Como um exemplo:

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>My Site</title>
</head>
<body>
<?php
    // busca a chave 'content' usando um helper de layout:
    echo $this->layout()->content;
    // busca a chave 'foo' usando um helper marcador:
    echo $this->placeholder('Zend_Layout')->foo;
    // busca o objeto de layout e recupera várias chaves dele:
    $layout = $this->layout();
    echo $layout->bar;
    echo $layout->baz;
?>
</body>
</html>
```

Como `Zend_Layout` utiliza `Zend_View` para renderização, você pode também usar quaisquer helpers de view registradas, e também ter acesso a quaisquer variáveis de view associadas. Particularmente útil são os vários helpers marcadores, que permitem a você recuperar conteúdo para áreas tais como a seção <head>, navegação, etc.:

```
<!DOCTYPE html
    PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <?= $this->headTitle() ?>
    <?= $this->headScript() ?>
    <?= $this->headStyle() ?>
</head>
<body>
    <?= $this->render('header.phtml') ?>
    <div id="nav"><?= $this->placeholder('nav') ?></div>
    <div id="content"><?= $this->layout()->content ?></div>
    <?= $this->render('footer.phtml') ?>
</body>
</html>
```

20.2.2. Usando Zend_Layout com o MVC do Zend Framework

`Zend_Controller` oferece um rico conjunto de funcionalidades para extensão por meio de seus front controller plugins e action controller helpers. `Zend_View` também tem helpers. `Zend_Layout` toma vantagem desses diversos pontos de extensão quando usado com os componentes MVC.

`Zend_Layout::startMvc()` cria uma instância de `Zend_Layout` com qualquer configuração opcional que você fornecer. Ele registra então um front controller plugin que renderiza o layout com qualquer conteúdo de aplicação uma vez que o laço de despacho foi feito, e registra um action helper para permitir o acesso ao objeto layout a partir de seus action controllers. Adicionalmente, você pode a qualquer tempo pegar a instância de dentro de um view script usando o view helper `layout`.

Primeiro, dê uma olhada em como inicializar Zend_Layout para uso com o MVC:

```
<?php
// Em seu arquivo de inicialização (bootstrap):
Zend_Layout::startMvc();
?>
```

`startMvc()` pode tomar uma matriz opcional de opções ou um objeto `Zend_Config` para customizar a instância; essas opções são detalhadas na Seção 20.3, "Opções de Configuração Zend_Layout".

Em um action controller, você pode então acessar a instância de layout como um action helper:

```
<?php
class FooController extends Zend_Controller_Action
{
    public function barAction()
    {
        // desabilita layouts para este action:
        $this->_helper->layout->disableLayout();
    }
    public function bazAction()
    {
        // usa um script de layout diferente com este action:
        $this->_helper->layout->setLayout('foobaz');
    };
}
?>
```

Em seus view scripts, você pode então acessar o objeto de layout via view helper `layout`. Esse view helper é levemente diferente dos outros no que toca a não tomar argumentos, e retornar um objeto ao invés de um valor string. Isso permite que você imediatamente chame métodos no objeto de layout:

```
<?php $this->layout()->setLayout('foo'); // configura layout alternativo ?>
```

A qualquer tempo, você pode buscar a instância de `Zend_Layout` registrada com MVC via método estático `getMvcInstance()`:

```php
<?php
// Retorna null se startMvc() não foi o primeiro a ser chamado
$layout = Zend_Layout::getMvcInstance();
?>
```

Finalmente, o front controller plugin de `Zend_Layout` tem uma importante característica em adição a renderização do layout: ele recupera todos os segmentos nomeados do objeto de resposta e associa variáveis, associando o segmento 'default' a variável 'content'. Isso permite a você acessar o conteúdo de sua aplicação e renderizá-lo em seus view scripts.

Como um exemplo, diremos que seu código primeiro ativa `FooController::indexAction()`, que renderiza algum conteúdo para o segmento de resposta padrão, e então prossegue para `NavController::menuAction()`, que renderiza conteúdo para o segmento de resposta 'nav'. Finalmente, você prossegue para `CommentController::fetchAction()` e busca alguns comentários, mas renderizar isso para o segmento de resposta é bom (o que adiciona conteúdo aquele segmento). Seu view script poderia então renderizar da um separadamente:

```
<body>
    <!-- renders /nav/menu -->
    <div id="nav"><?= $this->layout()->nav ?></div>
    <!-- renders /foo/index + /comment/fetch -->
    <div id="content"><?= $this->layout()->content ?></div>
</body>
```

Essa característica é particularmente útil quando usada em conjunção com o action helper e plugin ActionStack o qual você pode usar para configurar uma pilha de ações através da qual itertamos, e então criamos páginas widgetizadas.

20.2.3. Usando Zend_Layout como um Componente

Como um componente autônomo, Zend_Layout não oferece tantas características ou tanta conveniência como quanto é usado com o MVC. Entretanto, ele ainda tem dois benefícios a destacar:

- Escopo de variáveis de layout.
- Isolamento do view script de layout de outros scripts view.

Quando usado como um componente autônomo, simplesmente inicie o objeto layout, use os vários acessores para configurar estado, configure variáveis como propriedades de objeto, e renderize o layout:

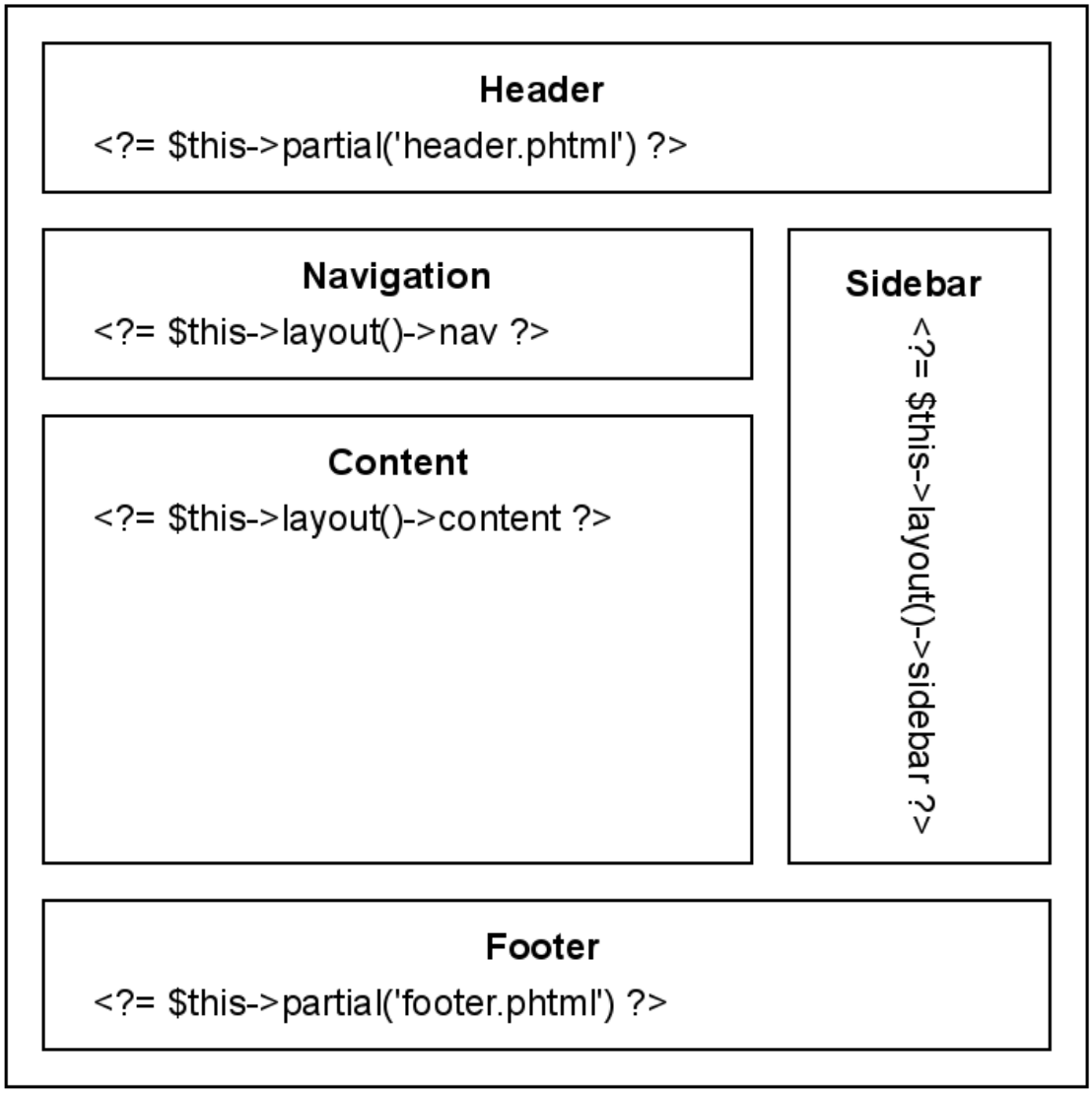
```php
<?php
$layout = new Zend_Layout();
// Configura um caminho de script de layout:
$layout->setLayoutPath('/path/to/layouts');
// Configura algumas variáveis:
$layout->content = $content;
$layout->nav     = $nav;
// Escolhe um script de layout diferente:
$layout->setLayout('foo');
// Renderiza o layout final
echo $layout->render();
```

```
?>
```

20.2.4. Exemplo de Layout

Algumas vezes uma imagem vale mais que mil palavras. A seguir temos um exemplo de um script de layout.

```
<?= $this->docType('XHTML1_STRICT') ?>
<html>
   <head>
      <?= $this->headTitle() ?>
      <?= $this->headScript() ?>
      <?= $this->headStylesheet() ?>
   </head>
   <body>
```




```
   </body>
</html>
```

A ordem real dos elementos pode variar, dependendo do CSS que você tenha configurado; por exemplo, se você está usando posicionamento absoluto, você pode ser capaz de ter a navegação exibida mais tarde no documento, mas ainda mostrar-se no topo; o mesmo poderia ser dito para a barra lateral ou o cabeçalho. A mecância real de puxar o conteúdo permanece a mesma, entretanto.

20.3. Opções de Configuração Zend_Layout

`Zend_Layout` tem uma variedade de opções de configuração. Essas podem ser configuradas chamando os acessores apropriados, passando uma matriz ou objeto `Zend_Config` para o construtor ou `startMvc()`, passando uma matriz de opções para `setOptions()`, ou passando um objeto `Zend_Config` para `setConfig()`.

- *layout*: o layout a ser usado. Utiliza o inflector atual para resolver o nome fornecido para o view script de layout apropriado. Por padrão, esse valor é 'layout' e resolve para 'layout.phtml'. Os acessores são `setLayout()` e `getLayout()`.
- *layoutPath*: o caminho base para os view scripts de layout. Os acessores são `setLayoutPath()` e `getLayoutPath()`.
- *contentKey*: a variável de layout usada para o conteúdo padrão (quando usado com o MVC). O valor padrão é 'content'. Os acessores são `setContentKey()` e `getContentKey()`.
- *mvcSuccessfulActionOnly*: quando usar o MVC, se uma ação lança uma exceção e esse marco é verdadeiro, o layout não será renderizado (isso é para prevenir dupla renderização do layout quando o ErrorHandler plugin estiver em uso). Por padrão, o marco é verdadeiro. Os acessores são `setMvcSuccessfulActionOnly()` e `getMvcSuccessfulActionOnly()`.
- *view*: O objeto view para ser usado na renderização. Quando usado com o MVC, `Zend_Layout` tentará usar o objeto view registrado com the ViewRenderer se nenhum objeto view tiver sido passado explicitamente. Os acessores são `setView()` e `getView()`.
- *helperClass*: a classe action helper para usar quando `Zend_Layout` estiver utilizando os componentes MVC. Por padrão, ela é `Zend_Layout_Controller_Action_Helper_Layout`. Os acessores são `setHelperClass()` e `getHelperClass()`.
- *pluginClass*: a classe front controller plugin para usar quando `Zend_Layout` estiver utilizando os componentes MVC. Por padrão, ela é `Zend_Layout_Controller_Plugin_Layout`. Os acessores são `setPluginClass()` e `getPluginClass()`.
- *inflector*: o inflector a ser usado quando resolver nomes para caminhos de view scripts de layout; veja a documentação `Zend_Layout` inflector para mais detalhes. Os acessores são `setInflector()` e `getInflector()`.

helperClass e pluginClass devem ser passados para startMvc()
Para que as configurações `helperClass` e `pluginClass` tenham efeito, elas devem ser passadas como opções para `startMvc()`; se forem configuradas mais tarde, elas não tem efeito.

20.3.1. Exemplos

Os seguintes exemplos assumem a seguinte matriz `$options` e objeto `$config`:

```
<?php
$options = array(
```

```
    'layout'     => 'foo',
    'layoutPath' => '/path/to/layouts',
    'contentKey' => 'CONTENT',           // ignorado quando o MVC não é usado
);
?>
```

```
<?php
/**
[layout]
layout = "foo"
layoutPath = "/path/to/layouts"
contentKey = "CONTENT"
*/
$config = new Zend_Config_Ini('/path/to/layout.ini', 'layout');
?>
```

Exemplo 20.1. Passando opções para o construtor or startMvc()

Tanto o construtor quanto o método estático `startMvc()` podem aceitar tanto uma matriz de opções quanto um objeto `Zend_Config` com opções de modo a configurar a instância de `Zend_Layout`.

Primeiro, dê uma olhada na passagem de uma matriz:

```
<?php
// Usando um construtor:
$layout = new Zend_Layout($options);
// Usando startMvc():
$layout = Zend_Layout::startMvc($options);
?>
```

E agora usando um objeto config:

```
<?php
$config = new Zend_Config_Ini('/path/to/layout.ini', 'layout');
// Usando construtor:
$layout = new Zend_Layout($config);
// Usando startMvc():
$layout = Zend_Layout::startMvc($config);
?>
```

Basicamente, esse é o modo mais fácil de customizar sua instância `Zend_Layout`.

Exemplo 20.2. Usando setOption() e setConfig()

Algumas vezes você precisa configurar o objeto `Zend_Layout` depois que ele já foi instanciado; `setOptions()` e `setConfig()` dão a você um modo fácil e rápido de fazer isso:

```
<?php
// Usando uma matriz de opções:
$layout->setOptions($options);
```

```
// Usando um objeto Zend_Config:
$layout->setConfig($options);
?>
```

Note, entretanto, que certas opções, tais como `pluginClass` e `helperClass`, não serão afetadas quando a passagem for feita por esse método; elas precisam ser passadas ao construtor ou método `startMvc()`.

Exemplo 20.3. Usando Acessores

Finalmente, você pode também configurar sua instância `Zend_Layout` via acessores. Todos os acessores implementam uma interface fluente, significando que suas chamadas podem ser encadeadas:

```
<?php
$layout->setLayout('foo')
       ->setLayoutPath('/path/to/layouts')
       ->setContentKey('CONTENT');
?>
```

20.4. Uso Avançado de Zend_Layout

`Zend_Layout` tem um número de casos de uso para o desenvolvedor que deseja adaptá-lo para diferentes implementações de view, layouts de sistema de arquivos, e mais.

Os principais pontos de extensão são:

- *Custom view objects.* `Zend_Layout` permite a você utilizar qualquer classe que implemente `Zend_View_Interface`.
- *Custom front controller plugins.* `Zend_Layout` vem embarcado com um front controller plugin padrão que automatiza a renderização de layouts antes de retornar a resposta. Você pode substituir por seu próprio plugin.
- *Custom action helpers.* `Zend_Layout` vem embarcado com um action helper padrão que deve servir para a maioria das necessidades já que é um proxy mudo para o próprio objeto de layout.
- *Custom layout script path resolution*. `Zend_Layout` permite a você usar seu próprio inflector para resolução do caminho do script de layout, ou simplesmente modificar o inflector anexado para especificar suas próprias regras de inflexão.

20.4.1. Objetos View Customizados

`Zend_Layout` permite a você usar qualquer classe que implemente `Zend_View_Interface` ou estenda `Zend_View_Abstract` para renderizar seu script de layout. Simplesmente passe seu objeto view customizado como um parâmetro para o construtor/`startMvc()`, ou configure o usando o acessor `setView()`:

```
<?php
$view = new My_Custom_View();
$layout->setView($view);
```

```
?>
```

Nem todas as implementações de Zend_View são iguais

Enquanto `Zend_Layout` permite que você use qualquer classe que implemente `Zend_View_Interface`, você pode entrar bem se elas não puderem utilizar os vários helpers `Zend_View`, particularmente os helpers layout e placeholder. Isso ocorre porque `Zend_Layout` torna o conjunto de variáveis no objeto disponível via ele mesmo e placeholders.

Se você precisa usar uma implementação customizada de `Zend_View` que não suporta esses helpers, você precisará descobrir um modo de obter as variáveis de layout para o view. Isso pode ser feito ou pela extensão do objeto `Zend_Layout` com alteração do método `render()` para passar variáveis para o view, ou criando sua própria classe plugin que as passa antes de renderizar o layout.

Alternativamente, se sua implementação de view suporta qualquer espécie de capacidade do plugin, você pode acessar as variáveis por meio do placeholder 'Zend_Layout' usando o helper placeholder:

```
<?php
$placeholders = new Zend_View_Helper_Placeholder();
$layoutVars   = $placeholders->placeholder('Zend_Layout')->getArrayCopy();
?>
```

20.4.2. Plugins Front Controller Customizados

Quando o usamos com os componentes MVC, `Zend_Layout` registra um plugin front controller que renderiza o layout como a última ação antes de abandonar o laço de despacho. Na maioria dos casos, o plugin padrão servirá, mas você se você desejar escrever o seu próprio, você pode especificar o nome da classe plugin a ser carregada carregar pela passagem da opção `pluginClass` ao método `startMvc()`.

Qualquer classe plugin que você escrever para esse propósito precisará estender `Zend_Controller_Plugin_Abstract`, e deverá aceitar uma instância de objeto layout como um argumento para o construtor. Caso contrário, os detalhes de sua implementação ficarão acima de você.

A classe plugin padrão usada é `Zend_Layout_Controller_Plugin_Layout`.

20.4.3. Action Helpers Customizados

Quando o usamos com componentes MVC, `Zend_Layout` registra um helper action controller com o helper broker. O helper padrão, `Zend_Layout_Controller_Action_Helper_Layout`, age como um proxy mudo para a própria instância do objeto de layout, e deve servir para a maioria dos casos de uso.

Se você sentir necessidade de escrever funcionalidades customizadsa, simplesmente escreva uma classe action helper estendendo `Zend_Controller_Action_Helper_Abstract` e passe o

nome da classe como uma opção `helperClass` para o método `startMvc()`. Detalhes da implementação ficarão acima de você.

20.4.4. Resolução de Caminho de Script de Layout Customizada : Usando o Inflector

`Zend_Layout` usa `Zend_Filter_Inflector` para estabelecer uma cadeia de filtro para traduzir um nome de layout para caminho de script de layout. Por padrão, ela usa as regras 'CamelCaseToDash' seguida por 'StringToLower', e o sufixo 'phtml' para transformar o nome em um caminho. Alguns exemplos:

- 'foo' será transformado em 'foo.phtml'.
- 'FooBarBaz' será transformado em 'foo-bar-baz.phtml'.

Você tem três opções para moficar inflexão: modificar o alvo de inflexão e/ou sufixo da view via acessores de `Zend_Layout`, modificar as regras do inflector e alvo do inflector associado com a instância `Zend_Layout`, ou criar sua própria instância de inflector instance e passá-la para `Zend_Layout::setInflector()`.

Exemplo 20.4. Usando acessores Zend_Layout para modificar o inflector

O inflector `Zend_Layout` padrão usa referências estáticas para o alvo e sufixo de view script, e te m acessores para configurar esses valores.

```
<?php
// Configure o alco do inflector:
$layout->setInflectorTarget('layouts/:script.:suffix');
// Configura o sufixo do view script de layout:
$layout->setViewSuffix('php');
?>
```

Exemplo 20.5. Modificação direta do inflector Zend_Layout

Inflectores tem um alvo e uma ou mais regras. O alvo padrão usado com `Zend_Layout` é ':script.:suffix'; ':script' passa o nome do layout registrado, enquanto ':suffix' é uma regra estática do inflector.

Digamos que você queira que o script de layout termine no sufixo 'html', e que você queira separar palavras CamelCased com underscores ao invés de hífens, e não deixe o nome em caixa baixa. Adicionalmente, você quer procurar em um subdiretório 'layouts' pelo script.

```
<?php
$layout->getInflector()->setTarget('layouts/:script.:suffix')
                       ->setStaticRule('suffix', 'html')
                       ->setFilterRule(array('CamelCaseToUnderscore'));
?>
```

Exemplo 20.6. Inflectores Customizados

Na maioria dos casos, modificar o inflector existente será suficiente. Entretanto, você pode ter um

inflector que você deseja usar em diversos lugares, com diferentes objetos de diferentes tipos. `Zend_Layout` suporta isso.

```
<?php
$inflector = new Zend_Filter_Inflector('layouts/:script.:suffix');
$inflector->addRules(array(
    ':script' => array('CamelCaseToUnderscore'),
    'suffix'  => 'html'
));
$layout->setInflector($inflector);
?>
```

Inflexão pode ser desabilitada

Inflexão pode ser desabilitada e habilitada usando acessores no objeto `Zend_Layout`. Isso pode ser útil se você quiser especificar um caminho absoluto para um view script de layout, ou saber que o mecanismo que você usará para especificar o script de layout não precisa de inflexão. Simplesmente use os métodos `enableInflection()` e `disableInflection()`.