第 15 章 单元测试和功能测试

自从有了面向对象的程序设计方法，自动化测试也迅速发展。特别是在开发 web 应用程序时，即使应用程序非常复杂，测试自动化也能保证程序的质量。symfony 提供了多种工具来实现测试自动化，本章将介绍这些方法。

测试自动化

任何有经验的 web 应用程序开发者都很在意要花多少时间来进行完备的测试。准备并运行测试案例然后分析测试结果是非常费时的工作。而且，web 应用的需求经常变动，因而一个应用会有一连串的版本，代码重构成为常事，错误因此不断地产生。

因而，虽然一个开发环境并不一定要有测试自动化，但是成功的开发环境还是应该包括测试自动化的。一组测试用例可以确保应用程序按照预计的去执行。尽管经常会重新设计程序内部结构，但是测试自动化可以防止重复犯错。而且，测试自动化会要求开发者用测试框架能理解的方式，去书写标准化了的、固定格式的测试用例。

因为测试自动化能清晰地描述一个应用程序能做些什么，因而有时侯它可以替代开发文档。一个好的测试包可以揭示出在一组测试输入下应该得到哪些输出结果，因而可以很好地解释一个方法的运行目标。

symfony 框架本身就采用了测试自动化方法。框架内部由测试自动化进行验证，这些单元测试和功能测试并不随 symfony 一起发布，但你可以从 SVN 库导出或从网站 http://www.symfony-project.com/trac/browser/trunk/test 得到。

单元测试和功能测试

单元测试检测一个单元的代码，以确定对于一个给定的输入，它能否得到正确的输出结果。对每个具体的测试用例，它可以验证函数或方法是否能正确执行。一次单元测试处理一个用例，所以如果一个方法在不同的情况下运行，就需要进行多次单元测试。

功能测试不是简单地测试输入输出间的转化是否正确，而是针对一个完整的功能特性。例如，一个缓存系统只能用一个功能测试去验证，因为它包括多个运行步骤。第一次请求一个页面时，产生缓存数据，第二次再请求页面时，页面就直接从缓存中取出。所以，功能测试可以验证一个运行过程，同时也需要一个运行环境，你可以用 symfony 为你的每一个动作写出功能测试代码。

对于最复杂的交互，只有这两种测试是不够的。比如说 ajax 交互，就需要一个 web 浏览器来执行 javascript，所以要自动测试还需要一个特殊的第三方工具。

更进一步的例子是视觉效果测试，它只能由人亲自去完成。

如果你有多种测试自动化的方法，你可能组合使用这些方法。有一个原则，你应该尽量让测试简单而且易于理解。

NOTE 测试自动化将测试结果和预期输出进行比较，也就是说，它会计算一个象$a==2 这样的断言（assertion）的值。断言的结果要么是真要么是假，对应着测试通过或失败。在测试自动化技术中，经常会用到断言这个词。

测试驱动的开发方法

在测试驱动的开发(TDD)方法中，编码之前就写好了测试用例。先写测试用例可以帮助你在真正开发一个功能之前明确一个功能会完成什么任务。象极限编程(XP)之类的开发方法也建议这样做。而且一个不可否认的事实是：如果你不事先写好单元测试用例，你将再也写不出来。

例如，你如果想写一个文本删节的功能，目的是将字符串开头和末尾的空格清除，用下划线替代非字母字符，并将所有大写字母转换成小写字母。在测试驱动的开发方法中，你要考虑到所有可能出现的情况，并对每种情况提供一个输入和预期输出的实例，如表 15-1 所示。

表 15-1 一个文本删节功能的测试用例表

输入	预期输出
" foo "	"foo"
"foo bar"	"foo_bar"
"-)foo:..=bar?"	"__foo___bar_"
"FooBar"	"foobar"
"Don't foo-bar me!"	"don_t_foo_bar_me_"

你要写出单元测试用例，运行这些测试用例，然后看结果是否正确。你要在代码中添加一些内容来处理第一个测试用例，运行看是否通过，再处理第二个测试用例，一直到所有测试用例都通过了，表明功能正确了。

采用驱动测试的开发方法设计的应用程序，测试代码几乎和实际代码一样多。如果你不想在调试测试用例方面花费太多时间，你应该尽力保持测试用例简单。

NOTE 重构一个方法会产生之前没有的错误。所以在将一个新版应用程序部署到生产环境去之前，应该运行所有的自动化测试代码——这个过程叫做回归测试。

lime 测试框架

PHP 开发领域有许多单元测试框架，最著名的应该是 PhpUnit 和 SimpleTest。~~Symfony~~symfony 采用自己的测试框架，叫做 lime，它基于 Test::More Perl 库，并且遵守 TAP 规则，也就是说，测试结果按照 Test Anything Protocol（译者

注：一种测试输出的格式标准，见本章的示例）规定的格式显示，这样有助于理解测试的输出结果。

Lime 用于单元测试，它是比较轻量级的 php 测试框架，并有以下一些特点：

- 它在一个受限定的环境中运行测试文件，这样可以避免每次测试之间产生奇怪的副作用。不是所有的测试框架都能保证为每次测试提供一个干净的测试环境的。
- lime 测试用例和测试结果都非常容易理解，在某些系统中，lime 还可以用彩色输出这种直观的方法来区分重要信息。
- symfony 就是利用 lime 进行回归测试的，所以在 symfony 源代码中可以看到许多单元测试和功能测试的例子。
- lime 核心就被单元测试验证过。
- lime 是用 PHP 写的，速度快且经过良好地编码。它只需一个 lime.php 文件，而不依赖于其它任何文件。

以下将使用 lime 语法描述各种不同的测试，在 symfony 安装版本中不包括这些功能。

NOTE 在生产环境中不应该运行单元测试和功能测试。这些是开发工具，应该运行在用于开发的机器上，而不应该运行在实际的服务器上。

单元测试

symfony 单元测试是以 Test.php 结尾的一些 PHP 文件，存放在你的应用程序的 test/unit/目录下。它们采用一种简单易读的语法。

单元测试概述

例 15-1 给出了一组典型的对 strtolower()函数进行单元测试的代码。从实例化 lime_test 对象开始（现在你无需考虑如何设置参数），接下来，每条单元测试就是对 lime_test 实例的方法的一次调用。这些方法的最后一个参数都是可选的用作输出的字符串。

例 15-1 单元测试文件示例，位于 test/unit/strtolowerTest.php

```
<?php

include(dirname(__FILE__).'/../bootstrap/unit.php');
require_once(dirname(__FILE__).'/../../lib/strtolower.php');

$t = new lime_test(7, new lime_output_color());

// strtolower()
```

```
$t->diag('strtolower()');
$t->isa_ok(strtolower('Foo'), 'string',
    'strtolower() returns a string');
$t->is(strtolower('FOO'), 'foo',
    'strtolower() transforms the input to lowercase');
$t->is(strtolower('foo'), 'foo',
    'strtolower() leaves lowercase characters unchanged');
$t->is(strtolower('12#?@~'), '12#?@~',
    'strtolower() leaves non alphabetical characters unchanged');
$t->is(strtolower('FOO BAR'), 'foo bar',
    'strtolower() leaves blanks alone');
$t->is(strtolower('FoO bAr'), 'foo bar',
    'strtolower() deals with mixed case input');
$t->is(strtolower(''), 'foo',
    'strtolower() transforms empty strings into foo');
```

在命令行执行 test-unit 即可启动测试。命令行输出非常简明，可以帮助你确定哪些测试失败，哪些通过。例 15-2 列出了测试的结果。

例 15-2 从命令行启动单个单元测试

```
> symfony test-unit strtolower

1..7
# strtolower()
ok 1 - strtolower() returns a string
ok 2 - strtolower() transforms the input to lowercase
ok 3 - strtolower() leaves lowercase characters unchanged
ok 4 - strtolower() leaves non alphabetical characters unchanged
ok 5 - strtolower() leaves blanks alone
ok 6 - strtolower() deals with mixed case input
not ok 7 - strtolower() transforms empty strings into foo
#     Failed test (.\batch\test.php at line 21)
#            got: ''
#       expected: 'foo'
# Looks like you failed 1 tests of 7.
```

TIP 例 15-1 中开头的 include 命令是可选的，但是有了这条 include 命令，这个测试文件就变成了一个独立的 PHP 脚本，无需 symfony 命令行就可以执行。

单元测试方法

Lime_test 对象包含了大量测试方法，如表 15-2 所示。

表 15-2 lime_test 对象中用于单元测试的方法

方法	描述
diag($msg)	仅输出备注信息而不进行测试
ok($test, $msg)	测试一个条件，如果条件为真，则通过
is($value1, $value2, $msg)	比较两个数值，如果全等(==)则通过
isnt($value1, $value2, $msg)	比较两个数值，如果不等则通过
like($string, $regexp, $msg)	测试字符串是否匹配正则表达式
unlike($string, $regexp, $msg)	测试字符串是否不匹配正则表达式
cmp_ok($value1, $operator, $value2, $msg)	比较两个参数的值是否与某个运算符匹配
isa_ok($variable, $type, $msg)	测试变量的类型
isa_ok($object, $class, $msg)	测试对象所属的类
can_ok($object, $method, $msg)	测试一个方法是否适用于某个对象或某个类
is_deeply($array1, $array2, $msg)	测试两个数组是否有相同的值
include_ok($file, $msg)	验证某个文件是否存在并且已经被包含
fail()	永远失败——用于测试异常
pass()	永远通过——用于测试异常
skip($msg, $nb_tests)	跳过$nb_tests 条后续的测试——用于条件测试
todo()	作为一条测试参加测试计数——为将要写但还未写的测试预留位置

语法很直观，并且你会看出多数方法用一个 message(信息)作为最后一个参数，当测试通过时，就会在屏幕上输出这个参数值。实际上，掌握这些方法的最好办法就是去测试它们，例 15-3 列出了调用这些方法的代码。

例 15-3 lime_test 对象的测试方法，位于 test/unit/exampleTest.php

```
<?php

include(dirname(__FILE__).'/../bootstrap/unit.php');

// 用于测试的桩对象和函数
class myObject
{
  public function myMethod()
  {
  }
}

function throw_an_exception()
```

```
{
  throw new Exception('exception thrown');
}

// 初始化测试对象
$t = new lime_test(16, new lime_output_color());

$t->diag('hello world');
$t->ok(1 == '1', 'the equal operator ignores type');
$t->is(1, '1', 'a string is converted to a number for comparison');
$t->isnt(0, 1, 'zero and one are not equal');
$t->like('test01', '/test\d+/', 'test01 follows the test numbering
pattern');
$t->unlike('tests01', '/test\d+/', 'tests01 does not follow the
pattern');
$t->cmp_ok(1, '<', 2, 'one is inferior to two');
$t->cmp_ok(1, '!==', true, 'one and true are not identical');
$t->isa_ok('foobar', 'string', '\'foobar\' is a string');
$t->isa_ok(new myObject(), 'myObject', 'new creates object of the
right class');
$t->can_ok(new myObject(), 'myMethod', 'objects of class myObject do
have amyMethod method');
$array1 = array(1, 2, array(1 => 'foo', 'a' => '4'));
$t->is_deeply($array1, array(1, 2, array(1 => 'foo', 'a' => '4')),
    'the first and the second array are the same');
$t->include_ok('./fooBar.php', 'the fooBar.php file was properly
included');

try
{
  throw_an_exception();
  $t->fail('no code should be executed after throwing an exception');
}
catch (Exception $e)
{
  $t->pass('exception catched successfully');
}

if (!isset($foobar))
{
  $t->skip('skipping one test to keep the test count exact in the
condition', 1);
}
else
```

```
{
  $t->ok($foobar, 'foobar');
}

$t->todo('one test left to do');
```

在 symfony 单元测试中你会看到更多使用这些方法的例子。

TIP 你会奇怪这里为什么用 is()而不是 ok()。原因是 is()的输出信息比 ok()更精确，ok()仅指出条件失败，而 is()还同时输出测试的成员。

测试参数

初始化 lime_test 对象时，它的第一个参数代表将要执行测试的项数。如果最终执行测试的项数与该参数值不同，lime 会输出相关的警告信息。 比如，例 15-3 的测试会输出例 15-4 的内容。对象初始化时要求进行 16 项测试，而最终只测试了 15 项，输出结果中给出了相关信息。

例 15-4 测试计数有助于进行测试规划

```
> symfony test-unit example

1..16
# hello world
ok 1 - the equal operator ignores type
ok 2 - a string is converted to a number for comparison
ok 3 - zero and one are not equal
ok 4 - test01 follows the test numbering pattern
ok 5 - tests01 does not follow the pattern
ok 6 - one is inferior to two
ok 7 - one and true are not identical
ok 8 - 'foobar' is a string
ok 9 - new creates object of the right class
ok 10 - objects of class myObject do have a myMethod method
ok 11 - the first and the second array are the same
not ok 12 - the fooBar.php file was properly included
#     Failed test (.\test\unit\testTest.php at line 27)
#       Tried to include './fooBar.php'
ok 13 - exception catched successfully
ok 14 # SKIP skipping one test to keep the test count exact in the
condition
ok 15 # TODO one test left to do
# Looks like you planned 16 tests but only ran 15.
# Looks like you failed 1 tests of 16.
```

diag()方法只用于显示注释信息，而不作为一条测试被计数，利用它，你的测试输出可以保持条理分明。另一方面，todo() 和 skip()方法则按照正常的测试被计数。在 try/catch 块中的 pass()/fail()组合会被当作一项测试被计数。

一个经过仔细规划的测试策略应该包括这个预计的测试条数。你会发现用它来验证你自己的测试文件非常有用，有些测试运行在判断条件或处理异常的复杂情况下，这时候测试条数就更为有用。

构造函数的第二个参数是一个继承了 lime_output 类的输出对象。因为大多数时候，测试是指在命令行方式下进行，输出一个 lime_output_color 对象，就可以利用 bash 的彩色去显示。

测试单元任务

从命令行启动的单元测试称为测试单元任务，它以一组测试名或一个包含通配符的文件模式作为参数。如例 15-5 所示。

例 15-5 启动单元测试

```
// 测试目录结构
test/
  unit/
    myFunctionTest.php
    mySecondFunctionTest.php
    foo/
      barTest.php

> symfony test-unit myFunction                    ## 运行
myFunctionTest.php
> symfony test-unit myFunction mySecondFunction  ##运行两种测试
> symfony test-unit 'foo/*'                      ##运行 barTest.php
> symfony test-unit '*'                          ##递归运行所有的测试
```

测试桩（Stubs），测试资源和自动加载

默认情况下，单元测试不支持自动加载特性。所以， 测试中的每个类要么在一个测试文件中定义，要么提供外部支持文件。所以许多测试文件都象例 15－6 所示的那样以一组 include 行开头。

例 15-6 在单元测试中包含类

```
<?php

include(dirname(__FILE__).'/../bootstrap/unit.php');
```

```
include(dirname(__FILE__).'/../../config/config.php');
require_once($sf_symfony_lib_dir.'/util/sfToolkit.class.php');

$t = new lime_test(7, new lime_output_color());

// isPathAbsolute()
$t->diag('isPathAbsolute()');
$t->is(sfToolkit::isPathAbsolute('/test'), true,
    'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('\\test'), true,
    'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('C:\\test'), true,
    'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('d:/test'), true,
    'isPathAbsolute() returns true if path is absolute');
$t->is(sfToolkit::isPathAbsolute('test'), false,
    'isPathAbsolute() returns false if path is relative');
$t->is(sfToolkit::isPathAbsolute('../test'), false,
    'isPathAbsolute() returns false if path is relative');
$t->is(sfToolkit::isPathAbsolute('..\\test'), false,
    'isPathAbsolute() returns false if path is relative');
```

在单元测试中，你不仅要实例化要测试的对象，而且要实例化它们所依赖的对象。因为单元测试必须保持单一性，所以依赖于其它类，将可能导致因为一个类错误而引起多个测试失败。另外，从代码行数和执行时间的角度来看，设置真正的对象也是非常费时的。因为开发人员很快就不能忍受缓慢的测试，所以要牢记单元测试的速度非常重要。

无论何时你开始进行一个包含多个脚本的单元测试，你都需要一个简单的自动加载系统。为此，sfCore 类(需要手工包含进去)提供了一个 initSimpleAutoload()方法，它用一条绝对路径作为参数，在这条路径下的所有类都会被自动加载。例如，如果想自动加载路径$ sf_symfony_lib_dir/util/下的所有类，就在你的测试脚本中以下列代码开始：

```
require_once($sf_symfony_lib_dir.'/util/sfCore.class.php');
sfCore::initSimpleAutoload($sf_symfony_lib_dir.'/util');
```

TIP 因为生成的 Propel 对象依赖于一连串的类，所以只要你测试 Propel 对象，就应该自动加载。注意，如果你要让 Propel 工作，你还需要包含 vendor/propel 目录下的文件，这样，调用 sfCore 就变成了 sfCore::initSimpleAutoload(array (SF_ROOT_ DIR.'/lib/model', $sf_symfony_lib_dir.'/vendor/propel')); 同时还要用 set_include_path($ sf_symfony_lib_dir.'/vendor'.PATH_SEPARATOR.SF_ROOT_DIR.PATH_SEPARATOR.get_include_path ()将 Propel 核心加入到包含路径中去。

替代自动加载的另一种方法是使用测试桩。测试桩用另一种方法实现一个类，也就是用一些简单的封闭好的数据替代真实的类方法，以模拟真实类的行为，同时又不需要象真实的类那样复杂。测试桩的一个典型的例子就是一个数据库连接或一个 web service 接口。在例 15-7 中，要对一个映射 API 进行单元测试需要 WebService 类，但在测试时却不调用真正的 web service 类的 fetch()方法，而是用一个测试桩去返回测试数据。

例 15-7 在单元测试中运用测试桩

```
require_once(dirname(__FILE__).'/../../lib/WebService.class.php');
require_once(dirname(__FILE__).'/../../lib/MapAPI.class.php');

class testWebService extends WebService
{
  public static function fetch()
  {
    return
file_get_contents(dirname(__FILE__).'/fixtures/data/fake_web_service.
xml');
  }
}

$myMap = new MapAPI();

$t = new lime_test(1, new lime_output_color());

$t->is($myMap->getMapSize(testWebService::fetch(), 100));
```

测试数据有时比一个字符串或一个方法调用更复杂，复杂的测试数据常被称为测试资源。为了清晰地编写代码，最好是将测试资源放在独立的文件中，特别是这些测试数据会被多个单元测试文件用到的时候。另外，不要忘记 symfony 可以用 sfYAML::load()方便地将 YAML 文件转换为数组。这样就可以将测试数据放在一个 YAML 文件中，而不用写很长的 PHP 数组，如例 15-8 所示。

例 15-8 在单元测试中运用测试资源文件

```
// 在 fixtures.yml：
-
  input:   '/test'
  output:  true
  comment: isPathAbsolute() returns true if path is absolute
-
  input:   '\\test'
```

```
  output:  true
  comment: isPathAbsolute() returns true if path is absolute
-
  input:   'C:\\test'
  output:  true
  comment: isPathAbsolute() returns true if path is absolute
-
  input:   'd:/test'
  output:  true
  comment: isPathAbsolute() returns true if path is absolute
-
  input:   'test'
  output:  false
  comment: isPathAbsolute() returns false if path is relative
-
  input:   '../test'
  output:  false
  comment: isPathAbsolute() returns false if path is relative
-
  input:   '..\\test'
  output:  false
  comment: isPathAbsolute() returns false if path is relative

// 在 testTest.php
<?php

include(dirname(__FILE__).'/../bootstrap/unit.php');
include(dirname(__FILE__).'/../../config/config.php');
require_once($sf_symfony_lib_dir.'/util/sfToolkit.class.php');
require_once($sf_symfony_lib_dir.'/util/sfYaml.class.php');

$testCases = sfYaml::load(dirname(__FILE__).'/fixtures.yml');

$t = new lime_test(count($testCases), new lime_output_color());

// isPathAbsolute()
$t->diag('isPathAbsolute()');
foreach ($testCases as $case)
{
  $t->is(sfToolkit::isPathAbsolute($case['input']), $case['output'],
$case['comment']);
}
```

功能测试

功能测试的目的是验证应用系统的各个部分。它们会模拟浏览器会话、发出请求、再检测响应中的元素值，就象你手工验证一个动作是否按预计的去执行一样。在功能测试中，你将运行一个与用例一致的场景。

功能测试概要

你可以用一个文本浏览器和一大堆正则表达式断言来进行功能测试，但是这样做是很费时间的。symfony 提供了一个称为 sfBrowser 的特别的对象，它连接到 symfony 应用程序，象一个浏览器一样工作，而不需要一个真实的服务器，同时不会降低 HTTP 传输速度。有了它，你可以存取每个请求的核心对象，包括 request、session、context 和 response 对象。symfony 还专门为功能测试提供了该类的一个扩展类，称为 sfTestBrowser，它提供了 sfBrowser 的所有功能并添加了一些灵活的断言方法。

一般来说，功能测试从初始化一个测试浏览器对象开始，这个对象发出一个动作请求然后检验响应中的元素值。

例如，每当你用 init-module 或 propel-init-crud 生成一个模块框架时，symfony 就为这个模块创建一个简单的功能测试。这个测试可以对默认的动作发出请求，然后检测响应的状态代码、 路由得到的模块和动作、以及响应内容中的某个句子。例 15-9 是为 foobar 模块生成的 foobarActionsTest.php 文件。

例 15-9 一个新模块的默认功能测试文件，位于 tests/functional/frontend/foobarActionsTest.php

```
<?php

include(dirname(__FILE__).'/../../bootstrap/functional.php');

// 创建一个新的测试浏览器
$browser = new sfTestBrowser();
$browser->initialize();

$browser->
  get('/foobar/index')->
  isStatusCode(200)->
  isRequestParameter('module', 'foobar')->
  isRequestParameter('action', 'index')->
  checkResponseElement('body', '!/This is a temporary page/')
;
```

TIP 浏览器方法返回一个 sfTestBrowser 对象，为了让你的测试文件更具可读性，你可以将方法调用串接起来。因为这一串方法调用可以不停顿地执行，所以称之为对象的流动接口。

功能测试可以包含多个请求和更复杂的断言，在接下来的章节中你会看到所有的可能情况。

运用 symfony 的命令行语句 test-funcitonal，可以执行一个功能测试，参见例 15-10。这个命令以一个应用程序名和一个测试名为参数，测试名省略了 Test.php 后缀。

例 15-10 从命令行启动一个功能测试

```
> symfony test-functional frontend foobarActions

# get /comment/index
ok 1 - status code is 200
ok 2 - request parameter module is foobar
ok 3 - request parameter action is index
not ok 4 - response selector body does not match regex /This is a
temporary page/
# Looks like you failed 1 tests of 4.
1..4
```

默认情况下，为一个新模块而生成的功能测试是不会通过的。这是因为在一个新创建的模块里，index 动作会指向 symfony 的 default 模块 s 的祝贺页面，其中包含”This is a temporary page”的句子。只要你没有修改 index 动作，测试该模块就总是失败，这样可以保证未完成的模块不能通过所有测试。

NOTE 功能测试启动了自动加载特性，所以你不需要用 include 包括所需文件。

用 sfTestBrowser 对象浏览

测试浏览器可以发出 GET 和 POST 请求，并且都是用一个真实的 URI 作为参数。例 15-11 显示了如何调用 sfTestBrowser 对象的方法来模拟请求。

例 15-11 模拟 sfTestBrowser 对象的请求

```
include(dirname(__FILE__).'/../../bootstrap/functional.php');

// 创建一个新的浏览器
$b = new sfTestBrowser();
$b->initialize();

$b->get('/foobar/show/id/1');                    // GET 请求
$b->post('/foobar/show', array('id' => 1));      // POST 请求

// get() 和 post()方法是 call()方法的简写形式
```

```
$b->call('/foobar/show/id/1', 'get');
$b->call('/foobar/show', 'post', array('id' => 1));

// call() 可以模拟任何请求方式
$b->call('/foobar/show/id/1', 'head');
$b->call('/foobar/add/id/1', 'put');
$b->call('/foobar/delete/id/1', 'delete');
```

一个典型的浏览器会话不仅包括对具体动作的请求，还包括在超链接和浏览器按钮上点击。sfTestBrowser 对象也能模拟这些请求，如例 15-12 所示。

例 15-12 模拟 sfTestBrowser 对象的浏览

```
$b->get('/');                    // 浏览主页
$b->get('/foobar/show/id/1');
$b->back();                      // 倒退一页
$b->forward();                   // 前进一页
$b->reload();                    // 刷新当前页
$b->click('go');                 // 找到名为`go`的链接或按钮并单击
```

测试浏览器能够处理调用栈，所以 back()和 forward()方法就象在一个真正的浏览器里一样工作。

TIP 测试浏览器有自己的管理 session(sfTestStorage)和 cookie 的机制。

在最需要测试的交互中，与表单有关的交互可能要排在首位。为了模拟表单的输入和提交，你有三种选择：一种是用你希望提交的参数发出一个 POST 请 求，第二种是调用 click()并将表单参数作为一个数组来传递，最后一种是在每个表单字段中一个又一个地填入值并点击提交按钮。三种方法最后都得到同样的 POST 请求。例 15-13 是一个示例。

例 15-13 用 sfTestBrowser 对象模拟表单输入

```
// 示例模板，位于 modules/foobar/templates/editSuccess.php
<?php echo form_tag('foobar/update') ?>
  <?php echo input_hidden_tag('id', $sf_params->get('id')) ?>
  <?php echo input_tag('name', 'foo') ?>
  <?php echo submit_tag('go') ?>
  <?php echo textarea('text1', 'foo') ?>
  <?php echo textarea('text2', 'bar') ?>
</form>

// 为该表单设计的功能测试示例 $b = new sfTestBrowser(); $b-
```

```
>initialize(); $b->get('/foobar/edit/id/1');

// 方法 1：POST 请求
$b->post('/foobar/update', array('id' => 1, 'name' => 'dummy',
'commit' => 'go'));

// 方法 2：用数组作参数并点击提交按钮
$b->click('go', array('name' => 'dummy'));

// 方法 3：根据表单字段名输入值后点击提交按钮.
$b->setField('name', 'dummy')-> click('go');
```

NOTE 注意。后两种方法在提交表单时，会自动包括默认的表单值，并且无需指明表单目标(处理表单的方法)。

当一个方法以 redirect()结束时，测试浏览器并不自动重定向，你必须用 followRedirect()手工重定向，参见例 15-14。

例 15-14 测试浏览器不能自动重定向

```
// 动作示例文件位于 modules/foobar/actions/actions.class.php
public function executeUpdate()
{
  ...
  $this->redirect('foobar/show?id='.$this-
>getRequestParameter('id'));
}

// 该动作的功能测试示例
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit?id=1')->
    click('go', array('name' => 'dummy'))->
    isRedirected()->     // 检测请求是否被重定向
    followRedirect();    // 手工重定向
```

对浏览有用的最后一个方法是 restart()方法，它可以重新初始化浏览器的 history、session 和 cookie——就象你重新启动了浏览器一样。

一旦有了第一个请求，sfTestBrowser 对象就可以存取 request、context 和 response 对象。也就是说，从文本内容到响应头以及请求参数和配置等，你都可以进行检测。

```
$request  = $b->getRequest();
$context  = $b->getContext();
$response = $b->getResponse();
```

SIDEBAR sfBrowser 对象

除了用于测试，在例 15-10 到例 15-13 中的所有浏览方法，还可以用于别的领域，只要通过 sfBrowser 对象来调用就可以，你可以用如下所示的方法去调用：

```
// 创建一个新的浏览器
$b = new sfBrowser();
$b->initialize();
$b->get('/foobar/show/id/1')->
    setField('name', 'dummy')->
    click('go');
$content = $b->getResponse()->getContent();
...
```

sfBrowser 对象对于批处理脚本是非常有用的，比如说，当你想浏览一组页面，以便为每个页面生成缓存版本的时候（请参考第 18 章的具体示例）。

运用断言

因为 sfTestBrowser 对象可以存取响应和请求的组成内容，所以你可以对这些组成内容进行测试。虽然为了测试你可以创建一个 lime_test 对象，但是 sfTestBrowser 对象的 test()方法就能返回一个 lime_test 对象，有了这个对象，你就可以调用前面介绍的单元断言方法。例 15-15 显示了如何通过 sfTestBrowser 来操作断言。

例 15-15 测试浏览器利用 test()方法来提供测试手段

```
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit/id/1');
$request  = $b->getRequest();
$context  = $b->getContext();
$response = $b->getResponse();

// 通过 test()方调用 lime_test 的方法
$b->test()->is($request->getParameter('id'), 1);
$b->test()->is($response->getStatuscode(), 200);
$b->test()->is($response->getHttpHeader('content-type'),
'text/html;charset=utf-8');
```

```
$b->test()->like($response->getContent(), '/edit/');
```

NOTE getResponse()、getContext()、getRequest()和 test() 等方法并不返回一个 sfTestBrowser 对象，所以你不能在这些方法之后串接其他 sfTestBrowser 的方法。

如例 15-16 所示，你可以通过请求和响应对象方便地检测输入输出 cookie 值。

例 15-16 用 sfTestBrowser 检测 cookie 值

```
$b->test()->is($request->getCookie('foo'), 'bar');      // 输入 cookie
$cookies = $response->getCookies();
$b->test()->is($cookies['foo'], 'foo=bar');             // 输出 cookie
```

利用 test 方法测试请求元素会导致代码行很长。为此，sfTestBrowser 提供了一组代理方法，以便让你的功能测试变得简明易读，并且返回一个 sfTestBrowser 对象。比如，在例 15-17 中，你可以用更快的方法写出与例 15-15 等价的代码。

例 15-17 直接用 sfTestBrowser 测试

```
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit/id/1')->
    isRequestParameter('id', 1)->
    isStatutsCode()->
    isResponseHeader('content-type', 'text/html; charset=utf-8')->
    responseContains('edit');
```

isStatusCode()的状态参数默认值为 200，所以你可以用不加参数的调用来测试响应是否成功。

代理方法的另一个优点是，你不需要象调用 lime_test 方法那样去指明输出文本，代理方法会自动生成输出信息，所以测试结果简明易懂。

```
# get /foobar/edit/id/1
ok 1 - request parameter "id" is "1"
ok 2 - status code is "200"
ok 3 - response header "content-type" is "text/html"
ok 4 - response contains "edit"
1..4
```

例 15-17 的代理方法涵盖了大多数常规测试 ，所以你很少会再用

sfTestBrowser 对象的 test()方法去测试。

例 15-14 中指出了 sfTestBrowser 不会自动重定向。这有一个好处，就是你可以测试一个重定向。例 15-18 显示了如何测试例 15-14 中的响应。

例 15-18 用 sfTestBrowser 测试重定向

```
$b = new sfTestBrowser();
$b->initialize();
$b->
    get('/foobar/edit/id/1')->
    click('go', array('name' => 'dummy'))->
    isStatusCode(200)->
    isRequestParameter('module', 'foobar')->
    isRequestParameter('action', 'update')->

    isRedirected()->      // 检测响应是否是一个重定向
    followRedirect()->    //手工重定向。

    isStatusCode(200)->
    isRequestParameter('module', 'foobar')->
    isRequestParameter('action', 'show');
```

运用 CSS 选择器

许多功能测试通过检测文本是否出现在页面中来验证页面是否正确。有了 responseContains()方法中的正则表达式的帮助，你可以测试显示出来的文本、标记属性和值是否正确。但是，一旦你想测试深藏在响应的 DOM 中的内容的话，正则表达式就不太好用了。

为此，sfTestBrowser 对象提供了一个返回 libXML2DOM 对象的 getResponseDom()方法，编译和测试该 libXML2DOM 对象比编译和测试一个普通文本要容易。例 15-19 给出一个使用这个方法的例子。

例 15-19 用测试浏览器存取以 DOM 对象表示的响应内容

```
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit/id/1');
$dom = $b->getResponseDom();
$b->test()->is($dom->getElementsByTagName('input')->item(1)->getAttribute('type'),'text');
```

但是，用 PHP DOM 方法去编译一个 HTML 文档仍旧不够快捷。如果你熟悉 CSS 选择器，你知道要从 HTML 文档中检索元素值，CSS 选择器才是更有效的工具。symfony 提供一个 sfDomCssSelector 工具类，它以 DOM 文档作为构造函数的参数。其中的 getTexts()方法根据一个 CSS 选择器返回一个字符串数组，而 getElements()方法则返回一个 DOM 元素数组。例 15-20 给出一个示例。

例 15-20 测试浏览器存取以一个 sfDomCssSelector 对象表示的响应内容

```
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit/id/1');
$c = new sfDomCssSelector($b->getResponseDom())
$b->test()->is($c->getTexts('form input[type="hidden"][value="1"]'),
array('');
$b->test()->is($c->getTexts('form textarea[name="text1"]'),
array('foo'));
$b->test()->is($c->getTexts('form input[type="submit"]'), array(''));
```

为了更加简明，symfony 提供了一种缩写方法，即代理方法 checkResponseElement()。例 15-20 利用这个方法可以得到和例 15-20 同样的结果。

例 15-21 测试浏览器通过 CSS 选择器存取响应元素

```
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit/id/1')->
    checkResponseElement('form input[type="hidden"][value="1"]',
true)->
    checkResponseElement('form textarea[name="text1"]', 'foo')->
    checkResponseElement('form input[type="submit"]', 1);
```

如何执行 checkResponseElement()方法，取决于它接收到的第二个参数的类型：

- 如果是逻辑型，则测试与 CSS 选择器匹配的元素是否存在。
- 如果是整型，则测试 CSS 选择器是否返回该整数个数的结果。
- 如果是正则表达式，则测试由 CSS 选择器发现的第一个元素是否与之匹配。
- 如果是由!开头的正则表达式，则测试第一个元素是否不匹配该正则表达式所表示的模式。
- 其他情形，则将由 CSS 选择器发现的第一个元素与第二个字符串参数进行比较。

该方法还接受以关联数组形式出现的第三个参数。它将测试某个指定位置的另一个元素，而不是测试由CSS选择器返回的第一个元素，参见例15-22。

例15-22 用位置选项匹配某个指定位置的元素

```
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit?id=1')->
    checkResponseElement('form textarea', 'foo')->
    checkResponseElement('form textarea', 'bar', array('position' =>
1));
```

这个数组选项还可以用来同时进行两个测试。你可以测试是否有一个元素匹配CSS选择器以及有几个匹配元素，参见例15-23。

例15-23 运用count选项计算匹配的个数

```
$b = new sfTestBrowser();
$b->initialize();
$b->get('/foobar/edit?id=1')->
    checkResponseElement('form input', true, array('count' => 3));
```

选择器工具功能很强，它可以接受CSS2.1的大多数选择器，你也可以用它测试类似例15-24中那样的复杂查询。

例15-24 checkResponseElement()接受的复杂CSS选择器

```
$b->checkResponseElement('ul#list li a[href]', 'click me');
$b->checkResponseElement('ul > li', 'click me');
$b->checkResponseElement('ul + li', 'click me');
$b->checkResponseElement('h1, h2', 'click me');
$b->checkResponseElement('a[class$="foo"][href*="bar.html"]', 'my
link');
```

在测试环境中工作。

sfTestBrowser对象可以使用一个特别的前端控制器来设置一个测试环境,。例15-25中是这种环境的默认配置。

例15-25 myap/config/settings.php中的默认测试环境配置

```
test:
```

```
.settings:
  # E_ALL | E_STRICT & ~E_NOTICE = 2047
  error_reporting:        2047
  cache:                  off
  web_debug:              off
  no_script_name:         off
  etag:                   off
```

在这个环境中，cache 和 web debug toolbar 设为 off，但是，代码执行仍旧会在日志文件中留下执行痕迹，与 dev 和 prod 日志文件不同，test 日志存放在 myproject/log/myapp_test.log 中，这样，你可以分开查询。在这个环境里，异常不会阻止脚本的执行---所以即使某个测试出错，也可以完成整个测试。你还可以指明数据库连接设置，比如，可以使用含有测试数据的另一个数据库。

在运用 sfTestBrowser 对象之前，你要先对它进行初始化。如果有必要，你可以为一个应用程序指定一个主机名，为客户端指定一个 IP 地址---如果你的应用程序通过主机名和 IP 地址进行控制的话，例 15-26 是一个示例。

例 15-26 用主机名和 IP 设置测试浏览器

```
$b = new sfTestBrowser();
$b->initialize('myapp.example.com', '123.456.789.123');
```

功能测试任务

功能测试任务根据接收到的参数的个数可以执行一个或多个功能测试，其规则看上去与单元测试任务很接近，只是功能测试任务总是要以一个应用程序作为第一个参数。参见例 15-27。

例 15-27 功能测试任务语法

```
// 测试目录结构
test/
  functional/
    frontend/
      myModuleActionsTest.php
      myScenarioTest.php
    backend/
      myOtherScenarioTest.php

## 对一个应用程序的所有功能进行递归测试
> symfony test-functional frontend

## 运行一个给定的功能测试
```

```
> symfony test-functional frontend myScenario

## 按照一个模式运行多个测试
> symfony test-functional frontend my*
```

为测试命名

本节给出一些实用方法，以便于组织和维护你的测试。

对于文件结构，你应该用需要测试的类来命名单元测试文件，用需要测试的模块或场景来命名功能测试文件。例 15-28 是一个例子。你的 test 目录很快就会包含许多文件，如果你不遵守这些原则的话，在长期的运行中将很难找到你要的测试文件。

例 15-28 文件命名实例

```
test/
  unit/
    myFunctionTest.php
    mySecondFunctionTest.php
    foo/
      barTest.php
  functional/
    frontend/
      myModuleActionsTest.php
      myScenarioTest.php
    backend/
      myOtherScenarioTest.php
```

对单元测试来说，根据函数或方法分组，并且用一个 diag()调用来启动每个测试组是一个比较好的方法。每个单元测试的信息应该包括要测试的功能或方法的名字，后接动作动词或属性名，这样测试的输出结果就象一个描述对象属性的句子。例 15-29 是个例子。

例 15-29 单元测试命名实例

```
// srttolower()
$t->diag('strtolower()');
$t->isa_ok(strtolower('Foo'), 'string', 'strtolower() returns a
string');
$t->is(strtolower('FOO'), 'foo', 'strtolower() transforms the input
to lowercase');

# strtolower()
```

```
ok 1 - strtolower() returns a string
ok 2 - strtolower() transforms the input to lowercase
```

功能测试应该根据页面分组并且用一个请求来启动。例 15-30 是个例子。

例 15-30 功能测试命名实例

```
$browser->
  get('/foobar/index')->
  isStatusCode(200)->
  isRequestParameter('module', 'foobar')->
  isRequestParameter('action', 'index')->
  checkResponseElement('body', '/foobar/')
;

# get /comment/index
ok 1 - status code is 200
ok 2 - request parameter module is foobar
ok 3 - request parameter action is index
ok 4 - response selector body matches regex /foobar/
```

如果你遵循这个习惯，你的测试输出结果将会非常清晰，它可以用作你的项目的开发文档，甚至有时候，这个文档可以完全取代实际的开发文档。

特别的测试需求

symfony 提供的单元和功能测试工具可以满足大多数需要。下面列出的一些技术用于解决测试自动化中会遇到的一些共同问题，包括：在孤立环境中启动测试，在测试中访问数据库，测试缓存及测试客户端的交互等。

在测试框架（Test Harness）中进行测试

单元测试任务和功能测试任务可以启动一个或一组测试。但是如果你不指明任何参数就调用这些任务，它们会启动 test 目录下的所有单元和功能测试。为了避免各个测试间互相影响，一种专门的机制用于将每个测试文件孤立在一个独立的环境中。进一步说，因为象保留单个测试的输出结果那样保留所有测试的输出结果 (那种情况下,输出可能达到几千行)没有意义，因此测试结果将被压缩进一个综合的视图中。这就是为什么要用一个测试框架（也就是一个具备特殊能力的自动测试框架）来运行大量的测试文件的原因。测试框架来自于 lime 框架的 lime_harness 组件，它可以一个文件一个文件地显示测试状态，并且在测试完大量文件候给出一个统计结果。如例 15-31 所示。

例 15-31 在测试框架中启动所有的测试

```
> symfony test-unit

unit/myFunctionTest.php.............ok
unit/mySecondFunctionTest.php.......ok
unit/foo/barTest.php................not ok

Failed Test                    Stat  Total   Fail  List of Failed
---------------------------------------------------------------
unit/foo/barTest.php              0      2      2  62 63
Failed 1/3 test scripts, 66.66% okay. 2/53 subtests failed, 96.22%
okay.
```

测试的运行和一个一个分别运行一样，只是输出结果变得简短，特别是最后有关失败测试的数据可以帮助你找到原因，非常有用。

你也可以用一条 test-all 任务来启动所有的测试，该任务也使用测试框架，参见例 15-32。这是在每次传送到生产环境之前你必须要做的工作，目的是确保最后的版本不存在回归错误。

例 15-32 启动一个项目的所有测试

```
> symfony test-all
```

存取数据库

单元测试经常需要访问数据库。要初始化数据库连接，可以调用 Propel 类的 getConnection()方法，如例 15-33 所示。

例 15-33 在测试中初始化数据库

```
$databaseManager = new sfDatabaseManager();

$databaseManager->initialize();



// 可以取得当前数据库链接，这不是必须的

$con = Propel::getConnection();
```

在测试之前，你应该用 sfPropelData 对象将测试资源放入你的数据库中。该对象可以象 propel-load-data 任务一样从一个文件导入数据，或者从一个数组导入数据，参见例 15-34。

例 15-34 从一个测试文件导入数据库

```
$data = new sfPropelData();

// 从文件导入数据
$data-
>loadData(sfConfig::get('sf_data_dir').'/fixtures/test_data.yml');

// 从数组导入数据
$fixtures = array(
  'Article' => array(
    'article_1' => array(
      'title'      => 'foo title',
      'body'       => 'bar body',
      'created_at' => time(),
    ),
    'article_2'    => array(
      'title'      => 'foo foo title',
      'body'       => 'bar bar body',
      'created_at' => time(),
    ),
  ),
);
$data->loadDataFromArray($fixtures);
```

然后你就可以根据需要，象在通常的应用中那样使用Propel对象了。记住要在单元测试中包括它们的文件（在测试桩、测试资源和自动加载的TIPS中已经介绍过用 sfCore::sfSimpleAutoloading()方法自动加载）。在功能测试中，Propel对象是自动加载的。

测试缓存

当你在一个应用程序中使用缓存时，功能测试需要检测被缓存的动作是否按照期望的动作执行。

首先要做的是在测试环境中(setting.yml)允许缓存功能运行。然后，你就能用sfTestBrowser对象提供的isCached()测试方法来测试一个页面是生成的还是来自于缓存的。例15-35示例了这个方法的使用。

例15-35 用isCached()方法测试缓存

```
<?php

include(dirname(__FILE__).'/../../bootstrap/functional.php');
```

```
// 创建一个新的测试浏览器
$b = new sfTestBrowser();
$b->initialize();

$b->get('/mymodule');
$b->isCached(true);        // 测试响应是否来自缓存
$b->isCached(true, true); // 测试被缓存的响应是否带有布局
$b->isCached(false);       // 测试响应是否不是来自缓存
```

NOTE 在功能测试开始时不需要清除缓存，启动脚本会替你执行。

测试客户端交互

前面介绍的技术的主要缺陷是它们不能模拟 Javascript。对于象 Ajax 交互那样的非常复杂的交互，你需要准确地复制由用户操作的鼠标点击和键盘输入并在客户端运行脚本。通常这些测试是手工完成的，但这样非常费时而且非常容易出错。

有一个完全用 Javascript 写成的测试框架 Selenium 可以作为一种解决方法加以考虑(http://www.openqa.org/selenium/)，它可以利用当前的浏览器窗口，象用户操作的那样在一个页面上执行一组操作。与 sfBrowser 对象相比 Selenium 的好处是它能够在页面内执行 Javascript，因此你可以用它来测试 Ajax 交互。

Selenium 没有被自动封装在 symfony 中。如果要安装它，你可以在 web 目录下建一个新的 selenium 目录，在这个目录里解压 Selenium 文件(http://www.openqa.org/selenium-core/download.action)。这是因为 Selenium 依赖于 Javascript，而大多数浏览器在标准的安全设置中都禁止运行 Javascript，除非在和你的应用程序相同的主机和端口上可以运行。

CAUTION 不要将 selenium 目录传送到你的生产服务器中，因为任何能通过浏览器访问你的 web 文档根目录的人都可以操作它。

Selenium 测试用 HTML 编写，并存放在 web/selenium/tests 目录中。例 15-36 给出了一个功能测试，它导入主页，点击 click me 链接，然后在响应中能查到”Hello, World”。记住，为了能在测试环境中访问应用程序，你必须指定 myapp_test.php 前端控制器。

例 15-36 一个 Selenium 测试实例，位于 web/selenium/test/testIndex.html

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
  <meta content="text/html; charset=UTF-8" http-equiv="content-type">
```

```
  <title>Index tests</title>
</head>
<body>
<table cellspacing="0">
<tbody>
  <tr><td colspan="3">First step</td></tr>
  <tr><td>open</td>              <td>/myapp_test.php/</td>
<td> </td></tr>
  <tr><td>clickAndWait</td>      <td>link=click me</td>
<td> </td></tr>
  <tr><td>assertTextPresent</td> <td>Hello, World!</td>
<td> </td></tr>
</tbody>
</table>
</body>
</html>
```

这个测试用例是用HTML写的，中间的一个表包含了三列，分别是命令、目标和值，不是所有命令都有对应的值。在本例中，要么让列空着，要么用 来让表更好看一些。要了解完整的Selenium命令，请参考Selenium网站。

在同一个目录中，你可以通过在TestSuite.html中的表格里插入一个新行，将这个测试添加到总的测试包中。例15-37是相关的示例。

例15-37 在测试包中添加一个测试文件，位于
web/selenium/test/TestSuite.html

```
...
<tr><td><a href='./testIndex.html'>My First Test</a></td></tr>
...
```

要运行这个测试，只需在浏览器地址栏中输入：

http://myapp.example.com/selenium/index.html

选择Main Test Suite，单击按钮运行所有的测试，观察你的浏览器是否模拟你想要执行的步骤。

NOTE 因为Selenium测试在真实的浏览器中运行，所以它可以测试浏览器的不一致性。你可以在某种浏览器中编制一个测试，然后在所有将支持你的应用程序的浏览器上检验。

由于Selenium测试是用HTML编写的，所以要编写一个Selenium测试是非常困难的。但是现在有了Firefox的Selenium扩展
([http://seleniumrecorder.mozdev.org/]

(http://seleniumrecorder.mozdev.org/))，要创建一个测试，只需在一个被记录的会话中执行一次测试即可。当你浏览记录的会话时，只需右击浏览器窗口并在弹出菜单中的Append Selenium Command项下选中适当的选项，就可以增加assert-type测试。

你可以将测试保存在一个HTML文件中，以便为你的应用程序建立一个测试包。Firefox扩展还允许你运行你已经记录过的Selenium测试。

NOTE 在启动Selenium测试之前，不要忘了重新初始化测试数据。

总结

测试自动化包括单元测试和功能测试两部分：前者用于验证方法或函数，后者验证功能特性。symfony的单元测试基于lime测试框架，并且为功能测试专门提供了sfTestBrowser类。两者都提供许多断言方法，从最基本的到最高级的，例如CSS选择器。使用symfony命令行启动测试，可以用test-unit和test-functional任务一个接一个地测试，也可以用test-all任务在一个测试框架中测试。当处理数据时，自动化测试要利用测试资源和测试桩模块，而这些在symfony单元测试中是很容易取得的。

如果，你确实已经为应用程序的大部分内容写了足够多的单元测试(也许你用的就是测试驱动的开发方法)，那么，当你重构内部结构或增加新的功能特性时，你就会感到更加放心，同时这也缩短了编写文档的时间。