



CakePHP – the rapid development php framework

中文手册 version 1.0 by Ken & Luxel

目录

基本概念 & 安装 CakePHP

配置 CakePHP

Scaffolding 脚手架

Model 定义 & 关联关系定义

Controller 控制器

View 视图

Component 组件

Helper

Cake 的全局常量及方法

数据检验

Plugins 插件

ACL 控制

数据清理 (YY 友情参与)

Cake Session Component
(papa 友情参与)

The Request Handler
Component

The Security Component

视图(View)缓存

简单用户认证示例

Cake 的命名约定 (YY 友情参与)

什么是 CakePHP?

CakePHP 是一个开源的 PHP on rails 的 full-stack framework。最开始从 Ruby On Rails 框架里得到灵感。程序员可使用它来快速创建的 Web 应用程序。我们的首要目的是使你以一种预置的快速且不失灵活性的方式开展工作。

为什么是 CakePHP?

CakePHP 有多个特点，这些特点让 CakePHP 成为了快速开发框架中的佼佼者之一。

1. 活跃友好的社区
2. 灵活的许可协议 (Licensing)
3. 兼容 PHP4 和 PHP5
4. 数据库交互和简单查询的集成
5. 应用程序 Scaffolding
6. MVC 体系结构
7. 友好的表现形式，自定义的 URL 的请求分配器 (Request dispatcher)
8. 内置验证机制
9. 快速灵活的模版 (PHP 语法，利用 helper)
10. AJAX, JavaScript, HTML Form 以及更多的 View Helper..
11. 安全，会话 (Session)，请求处理组件 (Request Handling Components)
12. 灵活的 ACL 机制
13. 数据的清理 (Data Sanitization)
14. 灵活的视图缓存 (Flexible View Caching)
15. 可在任何 web 站点的子目录里工作，不需要改变 Apache 配置

Cntjdavid 编辑于 2008-02-03 cntjdavid@gmail.com

资料来源 <http://www.1x3x.net/cakephp/>

基本概念

Section1 简介

本章节篇幅很短，简单的介绍一下 MVC 概念以及在 **cake** 中是如何实现的。如果你对 MVC 模式很陌生，这个章节将会非常适合你。我们由常规的 MVC 概念开始我们的讨论，然后用我们的方式在 **CakePHP** 中使用 MVC 模式，并且展示几个简单的示例。

Section 2 MVC 模式

[TODO 考虑到 MVC 模式是必需过桥技能，所以建议还是先看一下相关文章，原文很简单，最后补上望海涵]

Section 3 Cake 目录结构一览

当你解压缩 **Cake** 后，会发现有 3 个主目录

- app
- cake
- vendors

cake 目录下是核心 lib，通常情况下你不要接触它们。

app 目录下是你的程序文件所存放的位置。将 **app** 目录和 **cake** 目录分开，从而可以使多个应用共享一个 **cake** 类库。同样的也使得 **CakePHP** 的升级变得很容易：你只需要下载最新的版本并且覆盖原来的类库就可以了。不用担心会覆盖掉你的应用程序文件。

你可以使用 **vendors** 目录来存放一些第三方类库。后面的章节将会讲到更多关于 **vendors** 的内容，一个最基本的概念就是你可以非常方便的通过 **vendor()** 方法来调用 **vendor** 类库。

让我们来看一下完整的目录结构：

/app

/config - 配置文件目录，包括 Database, ACL 等

/controllers - Controllers 文件

/components - Components 文件

/index.php - 允许你将 app 目录部署为 DocumentRoot (译注：参见 Apache 相关配置)

/models - Model 文件

/plugins - Plugins 文件

/tmp - Cache 和日志存放处

/vendors - 你的应用中使用到的第三方类库

/views - 视图文件

/elements - 视图元素文件

/errors	- 自定义错误页面
/helpers	- Helpers 文件
/layouts	- 页面布局文件
/pages	- 静态页面文件
/webroot	- web 根目录
/css	
/files	
/img	
/js	
/cake	- 核心类库，请不要随意修改任何文件，除非你确信你有这个能力
index.php	
/vendors	- 服务器端的第三方类库

安装 CakePHP

Section 1 简介

现在你已经了解了每一件事情，包括系统的结构和整个类库的目的，或者说你略过了上述章节，因为你对这些资料并不感兴趣只是希望快点开始工作。Anyway 你已经摩拳擦掌准备完毕了。

这个章节来介绍了哪些是必须被安装到服务器上的，不同的配置方案，如何下载并安装 **CakePHP**，访问默认页面，以及一些 **troubleshooting** 以备一些意外情况的发生。

Section 2 安装必需

[TODO 一些译者认为比较基础的内容暂时优先级设为低 完美版释出时会补上]

Section 3 安装 CakePHP

[TODO]

Section 4 设置 CakePHP

第一种设置方案仅推荐使用在开发环境下，因为它不是最安全的方案。第二种方案考虑了更多的安全性问题因而可以放心的使用在生产环境下。

注意：/app/tmp 目录必须具有写权限

开发环境设置

对于开发环境来说，我们可以把整个 **Cake** 目录放置在 **DocumentRoot** 下：

/wwwroot

```
/cake
  /app
  /cake
  /vendors
  .htaccess
  index.php
```

这样的设置方案下，你的 URL 会如下这般(假设你使用了 `mod_rewrite`):

`www.example.com/cake/controllerName/actionName/param1/param2`

生产环境设置

使用生产环境配置，你必须拥有修改 **Web Server DocumentRoot** 的权限。那样的话，整个域就如同一个 **CakePHP** 应用一般。

生产环境配置使用如下目录结构

```
../path_to_cake_install
  /app
  /config
  /controllers
  /models
  /plugins
  /tmp
  /vendors
  /views

  /webroot <-- 这将是你的新的 DocumentRoot

  .htaccess
  index.php

  /cake
  /vendors
  .htaccess
  index.php
```

建议修改 **Apache** 的配置文件如下:

`DocumentRoot /path_to_cake/app/webroot`

这样的配置下，你的 URL 将会如下这般:

`http://www.example.com/controllerName/actionName/param1/param2`

高级设置

在有些情况下，你希望能够将 **Cake** 应用的目录放在磁盘不同目录下。这可能是因为虚拟主机的限制，或者你希望多个 **app** 应用能够共享同一个 **Cake Lib**。

对于一个 **Cake** 应用，有 3 个主要组成部分：

1. CakePHP 核心 Lib - /cake 目录下
2. 你的应用的代码（例如：controllers, models, layouts and views） - /app 目录下
3. 你的应用 web 目录下的文件（例如 图片 js 脚本以及 css 样式文件等） - /app/webroot 目录下

这 3 个目录都可以放置在磁盘的任意位置，但是 **webroot** 目录必须是 **web server** 可以访问的。你甚至可以把 **webroot** 目录移出 **app** 目录，只要你告诉 **Cake** 你把它放到哪里去了。

你需要修改 `/app/webroot/index.php` 来完成配置（和 **Cake** 一起分发）。你需要修改 3 个常量：ROOT, APP_DIR, and CAKE_CORE_INCLUDE_PATH。

1. ROOT 为包含 app 目录的根路径
2. APP_DIR app 目录路径
3. CAKE_CORE_INCLUDE_PATH Cake 核心 Lib 目录

你的应用 web 目录下的文件（例如 图片 js 脚本以及 css 样式文件等） - /app/webroot 目录下

这是范例：

```
/app/webroot/index.php (partial, comments removed)

if (!defined('ROOT')){ define('ROOT', dirname(dirname(dirname(__FILE__))));}

if (!defined('APP_DIR')){ define ('APP_DIR', basename(dirname(dirname(__FILE__))));}

if (!defined('CAKE_CORE_INCLUDE_PATH')){ define('CAKE_CORE_INCLUDE_PATH', ROOT);}
```

下面通过一个具体的例子来解释我的配置

1. 我希望 Cake Lib 能够被共享，所以放在 user/lib/cake 下（译注：作者都是 linux 配置，windows 环境下随便放放吧）
2. Cake webroot 目录为 /var/www/mysite/。
3. 程序文件目录为 /home/me/mysite。

下面为具体的目录结构，不再赘述

```
/home
  /me
    /mysite                                <-- Used to be /cake_install/app
      /config
      /controllers
      /models
      /plugins
      /tmp
      /vendors
```

```

    /views
    index.php
/var
    /www
        /mysite                <-- Used to be /cake_install/app/webroot
        /css
        /files
        /img
        /js
        .htaccess
        css.php
        favicon.ico
        index.php
/usr
    /lib
        /cake                <-- Used to be /cake_install/cake
        /cake
            /config
            /docs
            /libs
            /scripts
            app_controller.php
            app_model.php
            basics.php
            bootstrap.php
            dispatcher.php
            /vendors

```

我按照上面的目录结构修改/var/www/mysite/index.php 如下:

我建议使用'DS'常量代替路径中的斜杠。这样会保证你不会写错导致找不到文件。(考虑跨平台)

```

1. if (!defined('ROOT'))
2. {
3.     define('ROOT', DS.'home'.DS.'me');
4. }

```

```
5.

6. if (!defined('APP_DIR'))

7. {

8.     define ('APP_DIR', 'mysite');

9. }

10.

11. if (!defined('CAKE_CORE_INCLUDE_PATH'))

12. {

13.     define('CAKE_CORE_INCLUDE_PATH', DS.'usr'.DS.'lib'.DS.'cake');

14. }

15.
```

Section 5 配置 Apache 的 mod_rewrite

CakePHP 和 mod_rewrite 一同工作时会会是最佳体验，我们发现有部分用户在他们的系统上不能让每件事情都很好的运作起来。所以我罗列了一些可能会有帮助的信息：

1. 首先确认.htaccess 覆写是被允许的：在 apache 配置文件中检查<Directory>配置下的 AllowOverride 是不是设置为 All。
2. 确认你修改的是 server 的配置文件而不是当前用户或者特定站点的配置文件。
3. 有的时候，可能你得到的 CakePHP 副本缺少了必须的.htaccess 文件。因为有的操作系统会将以.开头的文件视作隐藏文件，因而不会拷贝它们。确定你的 CakePHP 副本是我们网站上下载或者是我们的 SVN repository 上 checkout 的。
4. 确认你是否成功加载了 mod_rewrite！察看配置文件中是否有'LoadModule rewrite_module libexec/httpd/mod_rewrite.so' 和 'AddModule mod_rewrite.c'。
5. 如果你把 Cake 安装在一个用户目录下的话 (http://example.com/~myusername/)，你需要修改根目录中的.htaccess 文件，加上一行"RewriteBase /~myusername/"。
6. 如果你发现 URL 后面加上了冗长乏味的 sessionId 的话
(http://example.com/posts/?CAKEPHP=4kgj577sgabvnmhjgkdiuy1956if6ska)
你需要加上一行"php_flag session.trans_id off"就 ok 了。

Section 6 确认 Cake 可以出炉了

好吧，让我们来看看这个 baby 吧。鉴于你选择的配置方式，你可以通过 <http://www.example.com> or <http://www.example.com/> 来访问它。你会看到 CakePHP 的默认主页，和一个简短的通知消息，描述了你当前的数据库连接状态。祝贺你！你已经完成了开始第一个 CakeApp 的准备工作

配置 CakePHP

Section 1 数据库配置

`app/config/database.php` 这个文件包含了所有的数据库配置。新安装的 Cake 是没有 `database.php` 这个文件的，你需要从 `database.php.default` 复制一份过来并重新命名，内容如下：

```
1. app/config/database.php

2. var $default = array('driver'    => 'mysql',

3.                        'connect' => 'mysql_connect',

4.                        'host'     => 'localhost',

5.                        'login'    => 'user',

6.                        'password' => 'password',

7.                        'database' => 'project_name',

8.                        'prefix'   => '');

9.

10. 和大多数 php 程序一样，修改一下你的配置就 ok 了。

11.
```

关于前缀要提醒一下：这里定义的前缀将会用在任何一处由 Cake 发起的对数据表的 SQL 操作中。你在这里定义过后不能再在其它地方重新定义它。假设你的服务器只给了你一个单独的 database，它同样可以让你遵循 Cake 的表名约定。注意：对于 HABTM(has-and-belongs-to-many)，也就是多对多关联表，你只需要添加一次前缀 `prefix_apples_bananas`，而不是 `prefix_apples_prefix_bananas`。

CakePHP 支持下列数据库驱动：

- mysql
- postgres
- sqlite
- pear-drivername (so you might enter pear-mysql, for example)
- adodb-drivername

`$default` 连接中的 'connect' 可以指定这个连接是否是 **persistent** 连接。参考 `database.php.default` 中的注释来决定采用何种方式。

Rails 两大原则之一，就是约定大于配置，所以你定义的数据表名必须遵循下面的命名约定：

1. 表名为英语单词的复数形式，例如 "users"，"authors" 或者 "articles"。注意与之对应的 model 对象则使用单数形式做名字。
2. 表字段必须有一个主键字段叫 'id'。（相信用过 ORM 工具的都明白个中道理吧）
3. 如果你使用外键关联表，外键字段必须定义成这个样子：'article_id'。即关联表名的单数形式加上下划线然后 id。其实也很好理解，并不需要特别记忆吧。
4. 如果你定义了 'created' 'modified' 这样的 auditor 字段的话，Cake 会自动在操作的时候去填充。（译注：rails 果然周全，hibernate 里面的 interceptor 就那么容易的做到了）

你应该注意到，同样的还有一个`$test`的数据库连接配置项，这是考虑到一般开发数据库和生产数据库通常并不是同一个，所以你可以在程序通过 `var $useDbConfig = 'test'`来切换数据库连接设置。

在你的 `model` 对象中，你可以通过这样的方式来添加任意多个附加数据库连接。

Section 2 全局配置

CakePHP 的全局配置文件为 `/app/config/core.php`。尽管我们非常讨厌配置文件，但是还是有些配置需要去做。文件中的注释对每一个配置项都有很详尽的说明。

DEBUG: 这个选项定义了应用程序的 `Debug` 级别。设置为一个非零的数值 `Cake` 会输出所有的 `pr()`和 `debug()`函数调用，并且在 `Forward` 之前会有提示信息。设置为 `2` 或者是更高的数值，则会在页面底部输出 `sql` 语句。

在 `debug` 模式下 (`DEBUG` 被设为 `1` 或者更高)，`Cake` 会输出错误自定义错误页面，例如 `"Missing Controller", "Missing Action"` 等等。在生产环境下 (`DEBUG` 被设为 `0`)，`Cake` 输出 `"Not Found"` 页面，你可以自行修改该页面 (`app/views/errors/error404.thtml`)。

CAKE_SESSION_COOKIE: 这个设置可以设定你所希望使用的 `cookie` 名字。

CAKE_SECURITY: 改变这个值标示了你所希望采用 `Session` 校验级别。`Cake` 会根据你设定的值来执行 `Session` 超时，生成新的 `session id`，删除历史 `session file` 等操作。可以选择的值如下所示：

1. `high`: 最高安全级别，10 分钟没有活动的 `session` 被置为超时，同时每一次请求都将生成新的 `session id`
2. `medium`: 20 分钟没有活动的 `session` 会被置为超时
3. `low`: 30 分钟没有活动的 `session` 会被置为超时

CAKE_SESSION_SAVE: 设置你的 `session` 数据储存策略：

1. `cake`: `Session` 被保存在 `Cake` 安装目录下的 `tmp` 目录
2. `php`: `Session` 被保存在 `php.ini` 文件设置的储存路径下
3. `database`: `Session` 保存在 `$default` 数据库连接所指定的库中

Section 3 路由(Route)配置

"Routing" 是一个纯 `PHP`，类似 `mod_rewrite` 的机制，来实现 `URL` 和 `controller/action/params` 的自动映射。通过这个机制 `Cake` 可以拥有简洁且可配置的 `URL` 并且可以摆脱对 `mod_rewrite` 的依赖。当然使用 `mod_rewrite` 会让你的浏览器地址栏变得更加简洁。

`Routes`（路由）是独立的 `URL` 和指定的 `controller` 和 `action` 映射规则。路由规则在 `app/config/routes.php` 里面配置。示例如下：

```
1. Route Pattern
2. <?php
3. $Route->connect (
4.     'URL',
5.     array('controller'=>'controllername',
6.         'action'=>'actionname', 'firstparam')
7. );
8. ?>
9.
```

```
10.
11.     1.
12.         URL 是一个正则表达式，表示你希望采用的 URL 格式
13.     2.
14.         controllername 是希望调用的 controller 的名字
15.
16.     3.
17.         actionname 是希望调用的 controller 的 action 名字
18.
19.     4.
20.         firstparam 是 action 的第一个参数名
21.
```

第一个参数之后的所有参数作为 **parameters** 被传递到 **action**。

下面是一个简单的示例，所有的 **url** 都被定向到 **BlogController**。默认的 **action** 为 **BlogController::index()**。

```
1. Route Example
2. <?php
3. $Route->connect ( '/blog/:action/*' , array('controller'=>'Blog' , 'action'=>'index') );
4. ?>
5. A URL like /blog/history/05/june can then be handled like this:
6. Route Handling in a Controller
7. <?php
8.
9. class BlogController extends ApplicationController
10. {
11.     function history($year, $month=null)
12.     {
13.         // .. Display appropriate content
14.     }
15. }
16. ?>
17.
```

URL 中的 'history' 通过 Blog's route 中的 :action 匹配。URL 中通过 * 匹配的内容都被当作 **parameters** 传递到 :action 对应的方法中，就像这里的 **\$year** 和 **\$month**。如果 URL 是 **/blog/history/05**，则 **history()** 方法只会得到一个参数，**05**。

下面这个例子是 CakePHP 默认设置的一个 route 来为 PagesController::display('home')配置路由。Home 是 Cake 的默认首页视图，你可以在这个位置找到并修改它/app/views/pages/home.thtml。

```
1. Setting the Default Route

2. <?php

3. $Route->connect ( '/', array('controller'=>'Pages', 'action'=>'display', 'home'));

4. ?>

5.
```

Section 4 高级路由配置: Admin 路由与 Webservice

在/app/config/core.php 中还有那么一些高级选项可以供你使用，你可以利用它们使你的 URL 精巧地适应绝大多数情况。

第一个介绍的选项是管理员路由(admin routing)。假设你有两个 Controller，ProductsController 和 NewsController，你肯定希望能有一些特别的 URL 给具有管理权限的用户来访问 Controller 中一些特别的 action。为了保持 URL 的优雅和可读性，有些程序员相比 /products/adminAdd 和 /news/adminPost 更喜欢 /admin/products/add 和 /admin/news/post。

激活这个功能，首先你需要把/app/config/core.php 中 CAKE_ADMIN 的注释去掉。CAKE_ADMIN 的默认值是'admin'，不过你可以修改为任何你喜欢的名字。但是你要记住这个名字，因为你需要在每一个 admin action 名字前面加上这个前缀，格式为 admin_actionName()。

```
/admin/products/add          CAKE_ADMIN = 'admin'

                               name of action in ProductsController =
'admin_add()'

/superuser/news/post         CAKE_ADMIN = 'superuser'

                               name of action in NewsController =
'superuser_post()'

/admin/posts/delete          CAKE_ADMIN = 'admin'

                               name of action in PostsController =
'admin_delete()'
```

使用 admin route 可以让你的业务逻辑非常井井有条。

注意！！并不是说使用了 admin route 同时 Cake 也替你完成了安全和认证，这些还是需要你自己来完成的。

类似的，你也能很快的为 webservice 建立路由。你需要将某个 action 暴露为 webservice 接口吗？首先，将/app/config/core.php 中的 WEBSERVICES 选项设为'on'。和先前的 admin 一样，打开这个选项可以很方便的让带有如下前缀的 action 自动暴露为 webservice 接口。

- rss
- xml
- rest
- soap
- xmlrpc

这样做的结果就是同一个 action，拥有了两种不同的视图，一个是标准的 HTML 视图，一个是 webservice 接口。因此你可以非常简单的让你的程序拥有 webservice 的能力。

举个例子，我的程序中有这么一个逻辑，有这么一个 **action** 可以告诉用户当前办公室里有多少人正在通话。我已经有有了一个 **HTML** 视图，不过我希望再提供一个 **XML** 格式的接口供我的桌面 **Widget**(译注：就是类似 **yahoo widget** 一类的桌面小贴士)或者是掌上设备来使用。首先，我打开了 **webservice** 路由功能：

```
1. /app/config/core.php (partial)

2. /**

3. * The define below is used to turn cake built webservices

4. * on or off. Default setting is off.

5. */

6.     define('WEBSERVICES', 'on');

7.
```

然后我在 **controller** 中构建代码如下：

```
1. <?php

2. class PhonesController extends AppController

3. {

4.     function doWhosOnline()

5.     {

6.         // this action is where we do all the work of seeing who's on the phone...

7.

8.         // If I wanted this action to be available via Cake's xml webservices route,

9.         // I'd need to include a view at /app/views/posts/xml/do_whos_online.thtml.

10.        // Note: the default view used here is at /app/views/layouts/xml/default.thtml.

11.

12.        // If a user requests /phones/doWhosOnline, they will get an HTML version.

13.        // If a user requests /xml/phones/doWhosOnline, they will get the XML version.

14.    }

15. }

16. ?>

17.
```

Section 5 自定义反射配置（可选）

Cake 的命名约束真的是非常不错，你可以命名 **model** 为 **Box**，**controller** 为 **Boxes**，于是所有的问题都解决了。但是考虑到有的情况下（特别是非英语国家）这样的命名约束不如你意的话，你可以通过自定义反射配置来满足你的需求。

/app/config/inflections.php，这里定义了 **Cake** 的一些变量，你可以来调整名字单复数约定等。你需要详细阅读文件中的注释，并且最好具备正则表达式知识，否则请误随便修改该配置文件。

Scaffolding 脚手架

Section 1 Cool! Cake 的脚手架

Cake 的脚手架真的非常酷，以至于你会希望将他用到你的生产环境下。我们也认为它非常的酷，但是请意识到脚手架再好也还是脚手架。它只是一大堆素材供你在项目起始阶段能够迅速的搭起整个项目的架子。所以，当你发现你需要实现自己的业务逻辑的时候就是该拆掉脚手架的时候了。

对于 web 应用项目，由脚手架开始项目是非常不错的办法。早期的数据库设计通常是经常会变动的。有这么一种趋势，web 开发人员非常厌恶创建一下根本不会实际使用的页面。为了减少开发人员的这种压力，Cake 内置了脚手架功能。脚手架会根据你当前的表结构创建基础的 CRUD 页面。在你的 controller 中使用脚手架只需要一行代码，加入一个变量就 ok。

```
1. <?php

2. class CategoriesController extends AppController

3. {

4.     var $scaffold;

5. }

6. ?>

7.
```

有一个非常重要的事情：脚手架中，要求每一个外键字段都必须以 `name_id` 这样的形式命名。因此，当你多级目录结构的时候，你可能会有一个字段叫 `parent_id`，这样的时候你就不能如此命名了，因为根据约定，它会去映射 `parent` 这个 model，而实际上是不存在这个 model 的，所以最好命名为 `parentid`。但是如果你是有一个外键关联另一个 model 的话（例如：`titles` 表中有 `category_id` 作为外键），你首先要确保正确的处理关联对象（参见 正确理解关联关系，6.2），在 `show/edit/new` 这 3 个视图里面会自动生成一个下拉框来展示/选择 `category` 信息。可以通过设置 model 中的 `$displayField` 变量来取舍那些外键字段是需要显示的。具体示例如下：

```
1. <?php

2. class Title extends AppModel

3. {

4.     var $name = 'Title';

5.

6.     var $displayField = 'title';

7. }

8. ?>

9.
```

Section 2 自定义脚手架视图

如果你希望脚手架的页面有一些自定义的元素，你可以自己修改模版。我们仍然不建议你在生产环境下使用脚手架视图，不过一定的自定义将会对于原形构建有这非常大的帮助。

如果你不希望修改默认提供的脚手架视图，你可以自己提供这些模版：

自定义单个 Controller 默认脚手架视图

PostsController 的自定义脚手架视图放置位置如下：

```
/app/views/posts/scaffold/index.scaffold.thtml  
/app/views/posts/scaffold/show.scaffold.thtml  
/app/views/posts/scaffold/edit.scaffold.thtml  
/app/views/posts/scaffold/new.scaffold.thtml
```

自定义全局脚手架默认视图

全局默认脚手架视图放置位置如下：

```
/app/views/scaffold/index.scaffold.thtml  
/app/views/scaffold/show.scaffold.thtml  
/app/views/scaffold/edit.scaffold.thtml  
/app/views/scaffold/new.scaffold.thtml
```

当你发现你需要实现自己的业务逻辑的时候也就是该拆除脚手教的时候。

Cake 中 **code generator** 有一个非常有用的功能：**Bake**。**Bake** 可以为你的应用生成一个基础的架子供你填充业务逻辑。

Models

Section 1 What is a model?

什么是 model 呢? model 就是 MVC 模式中的 M。

Model 用来做什么呢? 它将领域逻辑从表现层和独立的业务逻辑中剥离出来。

Model 通常是数据库的访问介质, 而且更明确的指定于某一张表(译注: 在 rails 框架中 model 扮演着 active record 的角色, 因此同时承担着 DAO 对象的职责)。默认情况下, model 使用的表名为 model 名字的复数形式, 比如'User'这个 model 对应的表名为'users'。Model 可以包含数据校验规则, 关联关系以及针对这张表的业务逻辑。下面展示了 User model 在 cake 中的具体定义:

```
1. Example User Model, saved in /app/models/user.php
2. <?php
3.
4. //AppModel gives you all of Cake's Model functionality
5.
6. class User extends AppModel
7. {
8.     // Its always good practice to include this variable.
9.     var $name = 'User';
10.
11.     // This is used for validation, see Chapter 11.
12.     var $validate = array();
13.
14.     // You can also define associations.
15.     // See section 6.3 for more information.
16.
17.     var $hasMany = array('Image' =>
18.                           array('className' => 'Image')
19.                           );
20.
21.     // You can also include your own functions:
22.     function makeInactive($uid)
23.     {
24.         //Put your own logic here...
25.     }
26. }
```

```
27.  
28. ?>  
29.
```

（译注：关于 **Model** 应该是贫血型，失血型还是充血型的论战持续至今尚无明显得胜的一方，本文中的这句话：**Model** 将领域逻辑从表现层和独立的业务逻辑中剥离出来 说得还比较清晰的，起码在 **cake** 里面他就是这么遵循的）

Section 2 Model Functions

从 PHP 的角度来看 **Model**，只是一个继承了 **AppModel** 的类。**AppModel** 类在 **/cake** 目录中已经定义了，如果你希望创建一个自己的 **AppModel** 基类，你可以放在 **app/app_model.php** 位置。**AppModel** 中定义的方法将被所有 **model** 所继承并共享。而 **AppModel** 则继承于 **Cake Lib** 中的 **Model** 类，文件为 **cake/libs/model.php**。

本章将会讨论到大量已经在 **Cake Model** 类中定义的方法，所以这里提醒，<http://api.cakephp.org> 将是您非常重要的参考。

用户定义的 **Model** 函数

下面是一个用户定义的 **Model** 方法，是帖子的隐藏/显示方法：

```
1. Example Model Functions  
2. <?php  
3. class Post extends AppModel  
4. {  
5.     var $name = 'Post';  
6.  
7.     function hide ($id=null)  
8.     {  
9.         if ($id)  
10.         {  
11.             $this->id = $id;  
12.             $this->saveField('hidden', '1');  
13.         }  
14.     }  
15.  
16.     function unhide ($id=null)  
17.     {  
18.         if ($id)  
19.         {  
20.             $this->id = $id;  
21.             $this->saveField('hidden', '0');  
22.         }  
23.     }
```



```
24. }
```

```
25. ?>
```

```
26.
```

获取你需要的数据

下面是 **Model** 中一些获取你需要的数据的标准方法：

```
findAll
```

```
    string $conditions
```

```
    array $fields
```

```
    string $order
```

```
    int $limit
```

```
    int $page
```

```
    int $recursive
```

[\$conditions Type: string] 检索条件,就是 sql 中 where 子句, 就像这样 `$conditions = "race = 'wookie' AND thermal_detonators > 3"`。

[\$fields Type: array] 检索属性, 就是投影, 指定所有你希望返回的属性。 (译注: 这里我没有使用字段这个亲 SQL 的名字而是用了属性这个亲对象的名字, 我相信很多 PHPer 更熟悉基于 Sql 和 DTO 的开发模式, 但是引入 **Model** 的一个初衷就是将关系数据库转换为对象操作, 所以投影查询应该理解为检索对象的某些属性)

[\$order Type: string] 排序属性 指定了 oder by 的属性名 [TODO: check whether the multiple order field be supported]

[\$limit Type: int] 结果集数据条目限制

[\$page Type: int] 结果集分页 index, 默认为返回第一页

[\$recursive Type: int] 递归 当设为大于 1 的整数时, 将返回该 model 关联的其他对象。(译注: 如果你使用过类似于 **Hibernate** 这样的 ORM 工具的话, 不难理解这个属性就是 **LazyLoad** 属性, 但也不完全等同, 数字的值代表级联查询的层次数, 例如这样, `user.country.city.address.id`)

```
find
```

```
    string $conditions
```

```
    array $fields
```

```
    string $order
```

```
    int $recursive
```

find 方法和 **findAll** 方法的区别在于, **findAll** 方法返回所有符合的结果集, **find** 方法只返回 list 中的第一个结果。

```
findAllBy<fieldName>
```

```
    string $value
```

这是一个非常有魔力的方法, 你可以看成是一个快速按照某个属性检索数据的方法, 下面是在 **controller** 中的一段示例代码:

```

1. $this->Post->findByTitle('My First Blog Post');

2. $this->Author->findByLastName('Rogers');

3. $this->Property->findAllByState('AZ');

4. $this->Specimen->findAllByKingdom('Animalia');

```

返回类型和 `find()` `findAll()` 一样，是一个 `array`。

findNeighbours

```

string $conditions
array $field
string $value

```

这个方法首先是通过 `conditions` 过滤获取结果集，然后根据 `field=value` 查找符合的对象（仅包含 `$field` 中指定的属性），最终返回该对象以及该对象的上一个对象和下一个对象。这在诸如相册这样的应用场景中将会非常有作用，你可以方便的获取当前相片，上一张和下一张这样的结果集合。注意：该方法只能作用于数值型和日期时间型属性。下面是示例代码：

```

1. class ImagesController extends ApplicationController

2. {

3.     function view($id)

4.     {

5.         // Say we want to show the image...

6.         $this->set('image', $this->Image->find("id = $id");

7.         // But we also want the previous and next images...

8.         $this->set('neighbours', $this->Image->findNeighbours(null, 'id', $id);

9.     }

10. }

```

我们拿到了包含一个完整 `$image['Image']` 对象的 `array`，以及 `$neighbours['prev']['Image']['id']` 和 `$neighbours['next']['Image']['id']`。

field

```

string $name
string $conditions
string $order

```

返回 `conditions` 过滤后按 `order` 排序的第一个结果中的 `name` 属性值，返回类型为 `string`。

findCount

```

string $conditions

```

返回 `conditions` 过滤后结果集的数量。即 `select count`

generateList

```
string $conditions
string $order
int    $limit
string $keyPath
string $valuePath
```

generateList 方法可以非常快捷的获取一个 **key-value** 这样的 **list**，其实就是其他语言中的 **map** 类型，在 **web** 领域中，下拉框可能将是该方法最大的受益者。**\$conditions**，**\$order** 和 **\$limit** 这 3 个参数的用法和 **findAll()** 没有区别，**\$keyPath** 和 **\$valuePath** 是 **model** 中用来填充结果集中 **key** 和 **value** 的属性。举个例子，在权限操作中，角色的定义往往是 **id - name** 这样的组合，参见下面的示例代码：

```
1. $this->set(
2.     'Roles',
3.     $this->Role->generateList(null, 'role_name ASC', null, '{n}.Role.id', '{n}.Role.role_name')
4. );
5.
6. //This would return something like:
7. array(
8.     '1' => 'Account Manager',
9.     '2' => 'Account Viewer',
10.    '3' => 'System Manager',
11.    '4' => 'Site Visitor'
12.);
```

```
read
    string $fields
    string $id
```

根据 **\$fields** 中的属性名从当前对象中读出对应的值，或者是 **\$id** 指定的对象中读取。注意：**read()** 方法仅级联读取该对象的一级关联对象，不管 **model** 中的 **\$recursive** 的值为多少。如果你是希望获取所有级联属性的话，请使用 **find()** 或者 **findAll()** 方法。

```
query
    string $query

execute
    string $query
```

有的时候希望执行自定义的 **sql** 语句或者是出于性能上的考虑要优化 **sql** 语句，则可以使用 **query(string \$query)** 和 **execute(string \$query)** 方法，这两个方法不同的地方在于 **execute** 方法无返回值，适用于执行脚本而不是查询结果集。

```

1. Custom Sql Calls with query()

2. <?php

3. class Post extends AppModel

4. {

5.     var $name = 'Post';

6.

7.     function posterFirstName()

8.     {

9.         $ret = $this->query("SELECT first_name FROM posters_table

10.                                WHERE poster_id = 1");

11.         $firstName = $ret[0]['first_name'];

12.         return $firstName;

13.     }

14. }

15. ?>

```

复合查询条件（使用 array）

绝大多数的 **Model** 查询方法中都会通过各种方式来传递一组条件。最简单的方式莫过于使用一个 **SQL** 中的 **where** 子句，不过如果你希望获得更多的控制，你可以使用 **array**。使用 **rray** 的好处是代码会非常的清晰和易读，并且可以比较方便的构造查询。而且这样的语法可以保证你的查询组成元素（属性，值，运算符等）被拆分为精巧可控的几部分。这可以保证 **Cake** 能够生成绝大部分非常有效率的查询语句，并且避免了 **SQL** 语法上易犯的错误。

先看看非常基础的基于 **array** 的查询：

简单查询条件 **array** 示例：

```
$conditions = array("Post.title" => "This is a post");
```

这个结构恐怕是不言自明了：该查询返回所有 **title** 为 "This is a post" 的 **Post** 对象。注意，只使用 "title" 作为查询的属性名也是可以的，但是在构造查询的时候，使用 **[ModelName.FieldName]** 这样的写法，一者可以让你的代码更加清晰，二者可以防止命名冲突，所以请遵循这个 **Good Practice**。那其他的检索形式如何呢？同样很简单，比如我们想检索所有 **title** 不是 "This is a post" 的 **Post** 对象：

```
array("Post.title" => "<> This is a post")
```

唯一增加的就是 '<>' 这个表达式。**Cake** 能够解析所有的合法 **SQL** 比较运算符，包括 **LIKE**, **BETWEEN** 或者是正则表达式，注意，运算符和值或者表达式之间需要有一个空格。唯一不同的操作是 **IN** 操作。我们看下具体的例子：

```
array("Post.title" => array("First post", "Second post", "Third post"))
```

为 **conditions** 增加更多 **filter** 就像往 **array** 中添加一对 **key-value** 那样简单：

```

1. array

2. (

3.     "Post.title"    => array("First post", "Second post", "Third post"),

4.     "Post.created" => "> " . date('Y-m-d', strtotime("-2 weeks"))

```

```
5. )
```

```
6.
```

默认情况下, **Cake** 使用'AND'来连接多个条件;这意味着,上面的示例代码检索结果为过去 2 周内 title 为"First post", "Second post"或者"Third post"的记录。当然我们也同样可以使用其他的逻辑运算符来连接查询条件:

```
1. array
```

```
2. ("or" =>
```

```
3.     array
```

```
4.     (
```

```
5.         "Post.title" => array("First post", "Second post", "Third post"),
```

```
6.         "Post.created" => "> " . date('Y-m-d', strtotime("-2 weeks"))
```

```
7.     )
```

```
8. )
```

Cake 可以使用任何 SQL 语句中允许的逻辑运算符,如 AND, OR, NOT, XOR 等等,并且大小写不敏感(**Conditions** 可以无限嵌套,但我并不推荐使用这样的 **magic code**)。Posts 和 Authors 两个对象间分别是 hasMany/belongsTo 的关系(熟悉 Hibernate 的同学或许更喜欢一对多 多对一的叫法)。假设你希望检索所有过去两周内发布的 posts 或者是包含有指定关键字的 posts,并且你希望限定作者为 **Bob**,让我们看如下代码:

```
1. array
```

```
2. ("Author.name" => "Bob", "or" => array
```

```
3.     (
```

```
4.         "Post.title" => "LIKE %magic%",
```

```
5.         "Post.created" => "> " . date('Y-m-d', strtotime("-2 weeks"))
```

```
6.     )
```

```
7. )
```

保存数据

当需要保存 **model** 对象时(译注:或者使用持久化这个词更贴切),你需要提供如下形式的数据给 **save()**方法:

```
1. Array
```

```
2. (
```

```
3.     [modelName] => Array
```

```
4.     (
```

```
5.         [fieldname1] => 'value'
```

```
6.         [fieldname2] => 'value'
```

```
7.     )
```

```
8. )
```

为了能够让数据通过这样的形式提交给 **controller**,最方便的办法就是使用 **HTML Helper** 来完成这个任务,因为 **HTML Helper** 会为提交的 **Form** 封装成 **Cake** 希望的这种形式。当然你可以不使用,只要页面上的元素的 **name** 被设置成 **date[modelname][fieldname]**形式就 ok 了。不过我还是要说, **\$html->input("Model/fieldname")**是最方便的办法。

(译注: OGNL 的 php 版, 太棒了, 我的意见是如果不嫌麻烦的话尽量使用类似 OGNL 的做法, 因为 tag 产生的页面在设计期是无法预览的, 我相信 web 应用前台页面的设计的复杂性并不亚于后台业务逻辑)

从页面 **Form** 中提交的数据自动被格式化并且注入到 **controller** 中的 `$this->data` 变量, 所以保存从 web 页面提交的数据只是举手之劳。下面我们来看一段示例代码:

```
1. function edit($id)
2. {
3.     //Note: The property model is automatically loaded for us at $this->Property.
4.     // Check to see if we have form data...
5.     if (empty($this->data))
6.     {
7.         $this->Property->id = $id;
8.         $this->data = $this->Property->read(); //populate the form fields with the current row
9.     }
10.    else
11.    {
12.        // Here's where we try to save our data. Automagic validation checking
13.        if ($this->Property->save($this->data['Property']))
14.        {
15.            //Flash a message and redirect.
16.            $this->flash('Your information has been saved.',
17.                '/properties/view/'.$this->data['Property']['id'], 2);
18.        }
19.        //if some fields are invalid or save fails the form will render
20.    }
21. }
22.
```

注意保存数据前会触发 **model** 的校验机制, 更多关于数据校验的内容请参见 **Chapter 12**。如果你不希望进行数据校验, 可以使用 `save($date, false)`。

其他一些有用的数据保存函数

```
del
string $id
boolean $cascade
```

删除指定 `id` 的 `model` 对象，或者当前 `model` 对象。

如果 `$cascade` 设为 `true`，则会级联删除当前 `model` 所关联的所有 `model` 中，设为 `'dependent'` 的 `model` 对象。删除成功返回 `true`

```
saveField
    string $name
    string $value
```

保存单个属性值。

```
getLastInsertId
```

返回最后当前 `ID` 属性的 `nextValue`。

[ToDo 扩展支持多种主键生成策略 例如 `sequence` `UUID`]

Model 中的回调函数

我们在 `model` 中增加了一些回调函数以帮助你在 `model` 操作前后能够织入一些业务逻辑（原文为 `sneak in`，借用了 `AOP` 中的织入一词，因为从操作来看这些回调函数等同于 `AOP` 中的 `advice`）。为了获得这样的能力，需要使用 `model` 中的一些参数并且在你的 `model` 中覆写这些方法。

```
beforeFind
    string $conditions
```

`beforeFind()` 回调函数是在 `model` 的 `find` 方法执行前的前置操作。你可以加入任何检索前的业务逻辑。你覆写该方法只要保证在前置操作成功后返回 `true` 来执行真正的 `find` 方法，返回 `false` 中断 `find` 方法就可以了。（译注：在一些复杂场景中，需多次持久化的情况下请慎用）

```
afterFind
    array $results
```

使用 `afterFind` 回调函数能够更改 `find` 方法的返回结果集，或者在检索动作完成后加上一些业务逻辑。该函数的参数 `$results` 为经过回调函数处理以后的 `find` 检索结果集。

```
beforeValidate
```

`beforeValidate` 回调函数能够在 `model` 校验数据之前更改 `model` 中的一些数据。同样也可以用来在 `model` 校验之前加入更为复杂的额外校验规则。和 `beforeFind` 一样，必须保证返回 `true` 来调用真正的操作，返回 `false` 来中断校验乃至 `save` 操作。

```
beforeSave
```

和先前介绍的回调函数类似，在校验完成之后，保存动作之前加入额外的处理（如果校验失败是不会触发该回调函数的）。返回 `true` 或者 `false`，不再赘述。

一个比较常见的 `beforeSave` 的应用场景就是在保存动作之前格式化日期属性以适应不同的数据库：

```
1. // Date/time fields created by HTML Helper:
2. // This code would be seen in a view
3. $html->dayOptionTag( 'Event/start' );
4. $html->monthOptionTag( 'Event/start' );
5. $html->yearOptionTag( 'Event/start' );
```

```

6. $html->hourOptionTag('Event/start');

7. $html->minuteOptionTag('Event/start');

8. /*=====*/

9. // Model callback functions used to stitch date

10. // data together for storage

11. // This code would be seen in the Event model:

12. function beforeSave()

13. {

14.     $this->data['Event']['start'] = $this->_getDate('Event', 'start');

15.     return true;

16. }

17.

18. function _getDate($model, $field)

19. {

20.     return date('Y-m-d H:i:s', mktime(

21.         intval($this->data[$model][$field . '_hour']),

22.         intval($this->data[$model][$field . '_min']),

23.         null,

24.         intval($this->data[$model][$field . '_month']),

25.         intval($this->data[$model][$field . '_day']),

26.         intval($this->data[$model][$field . '_year'])));

27. }

```

afterSave

保存完成后执行的动作。[ToDo 如果保存出错是否会触发?]

beforeDelete

afterDelete

不需要多说了吧，删除操作前后的回调函数。

Section 3 Model 中的变量

当你创建一个新的 model 的时候，有很多特殊的变量可以设置，从而是 model 具备 Cake 赋予的相应操作。

\$primaryKey

如果主键字段不为'id'，COC 无法发挥的时候，你可以通过该变量来指定主键字段名字。

\$recursive

这个我们上面已经介绍过了，指定了 model 级关联对象的深度。

想象这样一个场景，Group 下的 User 下面还有各自的 Articles。

```
$recursive = 0 Cake 只会检索 Group 对象
$recursive = 1 Cake 会检索包含 User 对象的 Group 对象
$recursive = 2 Cake 会检索完成的包含 Article 的 Group 对象
```

\$transactional

决定 Model 是否允许事务处理，true 或者 false。注意，仅在支持事务的数据库上有效。

\$useTable

和\$primaryKey一样，如果你的表名不能和 model 匹配的话，而且你也不想或者不能修改表名，则可以通过该变量来指定数据表名。

\$validate

该变量为一个 array，表示一组校验规则，详细请参见 Chapter 12。

\$useDbConfig

还记得我们在配制一章介绍的如何配置数据库连接变量吗？可以通过该变量非常方便的切换数据库连接，默认的是什么？猜一下，当然就是'default'，rails 就是让你忘却那些难以记忆的配置。

Section 4 [重头戏]关联对象

简介

CakePHP 中提供的一个非常重要的功能就是关联数据表间的映射。在 CakePHP 中，关联表通过 association（关联）来处理。关联是这些逻辑上相关的单元间的胶水一般。

CakePHP 中一共有 4 种类型的关联：

```
hasOne
hasMany
belongsTo
hasAndBelongsToMany
```

一旦 model 间的关联关系被定义，Cake 能够自动检索你当前操作的 model 中包含的关联对象。也就是将基于关系数据库的数据映射为基于对象的领域模型。举个例子，Post 和 Author 之间是 hasMany(一对多)关系，当我们在 controller 中通过\$this->Post->findAll()来检索所有的 Post 对象时，也会同时把所有关联的 Author 对象检索出来。

遵循 CakePHP 的命名约定是正确定义关联关系的有利保证（参见附录 B）。如果你使用了 CakePHP 的命名约定，可以通过脚手架来可视化你的数据，因为脚手架会自动侦测并使用你定义的关联关系，这样也能某种程度上供你检查是否定义正确。当然，你也可以完全不使用命名约定来定义关联，不过我们稍后再介绍关于这种定义。现在我们仅关注使用命名约定的情况下的关联定义。命名约定中我们需要考虑的有这 3 个内容，外键，model 名字，表名。

这里先简单的介绍一下关于这 3 者的要求，更为详细的请查看附录：

外键：[单数形式的 model 名字]_id 假设在"authors"表中有 Post 的外键关联，则外键字段名字应该为 "post_id"。

表名：[复数形式的 model 名字] 表名必须为 model 名字的复数形式，例如："posts" "authors"。

Model 的名字：[驼峰法命名 单数形式]。

CakePHP 的脚手架希望你的关联定义的顺序和表中的外键字段的顺序是一致的。所以如果我有一个 Article 对象[belongsTo(属于)]另外 3 个对象 (Author Editor Publisher) 的话，我需要 3 个外键 author_id, editor_id, publisher_id。脚手架要求你 model 中对应的关联和数据库中的列顺序保持一致。

为了更好的描述关联对象是如何运作的，让我们继续以 **Blog** 为应用场景来介绍。假设我们现在需要为 **Blog** 系统创建一个简单的用户管理模块。我们假设我们不需要跟踪用户情况，但是我们希望每个用户都一个个人记录（用户 **[hasOne]** 个人记录）。用户可以创建多条评论记录（用户 **[hasMany]** 评论记录）。同样的，我们的文章会被分配到多个 **tag**，同时每个 **tag** 都包含多篇文章，也就是多对多关系（文章 **[hasAndBelongsToMany]** **Tag**）。

hasOne 关联的定义与查询

假设你已经准备好了 **User** 和 **Profile** 两个 **model**，让我们来定义他们之间的关联。**hasOne** 关联的定义是通过在 **model** 中增加一个 **array** 来实现的。下面是示例代码：

```
1. /app/models/user.php hasOne
2. <?php
3. class User extends AppModel
4. {
5.     var $name = 'User';
6.     var $hasOne = array('Profile' =>
7.         array('className' => 'Profile',
8.             'conditions' => '',
9.             'order' => '',
10.            'dependent' => true,
11.            'foreignKey' => 'user_id'
12.        )
13.    );
14. }
15. ?>
```

\$hasOne 变量是一个 **array**，**Cake** 通过该变量来构建 **User** 与 **Profile** 之间的关联。我们来看每一个元素代表的意义：

- **className (required)**: 关联对象的类名，上面代码中我们设为'**Profile**'表示关联的是 **Profile** 对象。
- **conditions**: 关联对象的选择条件，（译注：类似 **hibernate** 中的 **formula**）。具体到我们的例子来看，假设我们仅关联 **Profile** 的 **header color** 为绿色的文件记录，我们可以这样定义 **conditions**，"**Profile.header_color = 'green'**"。
- **order**: 关联对象的排序方式。假设你希望关联的对象是经过排序的，你可以为 **order** 赋值，就如同 **SQL** 中的 **order by** 子句：**"Profile.name ASC"**。
- **dependent**: 这是个布尔值，如果为 **true**，父对象删除时会级联删除关联子对象。在我们的 **Blog** 中，如果"**Bob**"这个用户被删除了，则关联的 **Profile** 都会被删除。类似一个外键约束。
- **foreignKey**: 指向关联 **Model** 的外键字段名。仅在你不遵循 **Cake** 的命名约定时需要设置。

现在，现在当我们使用 **find()** **findAll()**检索 **User** 对象时，你会发现关联的 **Profile** 对象也被检索回来，非常的方便：

```
1. $user = $this->User->read(null, '25');
2. print_r($user);
3. //output:
4. Array
```

```

5. (
6.     [User] => Array
7.     (
8.         [id] => 25
9.         [first_name] => John
10.        [last_name] => Anderson
11.        [username] => psychic
12.        [password] => c4k3roxx
13.    )
14.
15.    [Profile] => Array
16.    (
17.        [id] => 4
18.        [name] => Cool Blue
19.        [header_color] => aquamarine
20.        [user_id] = 25
21.    )
22.)

```

belongsTo 关联的定义与使用

现在 **User** 对象能够得到对应的 **Profile** 对象，当然我们也应该为 **Profile** 对象定义一个关联使之能够获取它的所有者，也就是对应的 **User** 对象。在 **Cake** 中，我们使用 **belongsTo** 关联来实现：

```

1. /app/models/profile.php belongsTo
2. <?php
3. class Profile extends AppModel
4. {
5.     var $name = 'Profile';
6.     var $belongsTo = array('User' =>
7.         array('className' => 'User',
8.             'conditions' => '',
9.             'order' => '',
10.            'foreignKey' => 'user_id'
11.        )
12.    );
13. }

```

和 `hasOne` 关联一样，`belongsTo` 也是一个 `array` 变量，你可以通过设置其中的值来具体定义 `belongsTo` 关联：

- **className (required):** 关联对象的类名，这里我们关联的是 `User` 对象，所以应该是 `'User'`。
- **conditions:** SQL 条件子句以限定关联的对象，假定我们只允许 `Profile` 关联状态为 `active` 的用户，我们可以这样写：`"User.active = '1'"`，当然也可以是类似的其它条件。
- **order:** 关联对象的排序子句，假如你希望关联的对象经过排序，你可以类似 `"User.last_name ASC"` 这样来定义。
- **foreignKey:** 关联对象所对应的外键字段名，仅在你不遵循 `Cake` 的命名约定时需要设置。

现在当我们使用 `find()` `findAll()` 来检索 `Profile` 对象时，会发现关联的 `User` 对象也一同被检索回来。

```
1. $profile = $this->Profile->read(null, '4');
2. print_r($profile);
3. //output:
4. Array
5. (
6.
7.     [Profile] => Array
8.     (
9.         [id] => 4
10.        [name] => Cool Blue
11.        [header_color] => aquamarine
12.        [user_id] = 25
13.    )
14.
15.    [User] => Array
16.    (
17.        [id] => 25
18.        [first_name] => John
19.        [last_name] => Anderson
20.        [username] => psychic
21.        [password] => c4k3roxx
22.    )
23. )
```

hasMany 关联的定义与查询

我们已经为 `User` 和 `Profile` 对象建立起了双向关联，那让我们开始为 `User` 和 `Comment` 对象之间建立关联吧，先看下面的示例代码：

```

1. /app/models/user.php hasMany
2. <?php
3. class User extends AppModel
4. {
5.     var $name = 'User';
6.     var $hasMany = array('Comment' =>
7.         array('className' => 'Comment',
8.             'conditions' => 'Comment.moderated = 1',
9.             'order' => 'Comment.created DESC',
10.            'limit' => '5',
11.            'foreignKey' => 'user_id',
12.            'dependent' => true,
13.            'exclusive' => false,
14.            'finderQuery' => ''
15.        )
16.    );
17.
18.    // Here's the hasOne relationship we defined earlier...
19.    var $hasOne = array('Profile' =>
20.        array('className' => 'Profile',
21.            'conditions' => '',
22.            'order' => '',
23.            'dependent' => true,
24.            'foreignKey' => 'user_id'
25.        )
26.    );
27. }
28. ?>

```

`$hasMany` array 用来定义 `User` 包含多条 `Comment` 这样的关联关系。还是老样子，介绍一下包含的 `key`，但是一些和之前同样含义的 `key` 我将不再赘述详细。

- `className (required)`: 关联对象类名。
- `conditions`: 关联对象限定条件。
- `order`: 关联对象排序子句。
- `limit`: 因为是一对多关系，所以可以通过 `limit` 来限定检索的关联对象数量。比如我们可以只关联 5 条评论记录。

- **foreignKey**: 外键字段名。仅当不遵循命名约定时起用。
- **dependent**: 是否级联删除。（该动作可能会造成数据的误删除，请谨慎设定）
- **exclusive**: 如果设为 **true**，所有的关联对象将在一句 **sql** 中删除，**model** 的 **beforeDelete** 回调函数不会被执行。但是如果没有复杂的逻辑在级联删除中，这样的设定会带来性能上的优势。（译注：**Cake** 的确方便，但是使用时一定要记住控制 **sql** 语句发送数量）
- **finderQuery**: 定义一句完整的 **sql** 语句来检索关联对象，能够对关联规则进行最大程度上的控制。当关联关系特别复杂的时候，比如 **one table – many model one model – many table** 的情况下，**Cake** 无法准确的替你完成映射动作，需要你自己来完成这个艰巨的任务。

现在看一下如何在检索 **user** 对象的时候一并读回 **comment** 对象集合

```

1. $user = $this->User->read(null, '25');

2. print_r($user);

3. //output:

4. Array

5. (

6.     [User] => Array

7.     (

8.         [id] => 25

9.         [first_name] => John

10.        [last_name] => Anderson

11.        [username] => psychic

12.        [password] => c4k3roxx

13.    )

14.

15.    [Profile] => Array

16.    (

17.        [id] => 4

18.        [name] => Cool Blue

19.        [header_color] => aquamarine

20.        [user_id] => 25

21.    )

22.

23.    [Comment] => Array

24.    (

25.        [0] => Array

26.        (

```

```
27.         [id] => 247
28.         [user_id] => 25
29.         [body] => The hasMany association is nice to have.
30.     )
31.
32.     [1] => Array
33.     (
34.         [id] => 256
35.         [user_id] => 25
36.         [body] => The hasMany association is really nice to have.
37.     )
38.
39.     [2] => Array
40.     (
41.         [id] => 269
42.         [user_id] => 25
43.         [body] => The hasMany association is really, really nice to have.
44.     )
45.
46.     [3] => Array
47.     (
48.         [id] => 285
49.         [user_id] => 25
50.         [body] => The hasMany association is extremely nice to have.
51.     )
52.
53.     [4] => Array
54.     (
55.         [id] => 286
56.         [user_id] => 25
57.         [body] => The hasMany association is super nice to have.
58.     )
59.
60. )
```

你同样可以为 **Comment** 加上关联 **User** 对象的 **belongsTo** 关联，但是在文档中就不再详细描述了。

hasAndBelongsToMany 关联的定义与查询

我相信你已经掌握了简单的关联定义，让我们来看最后一个，也是最为复杂的关联关系：**hasAndBelongsToMany(HABTM)**。这个关联会让你头大的，不过也是最有用的。（译注：我倒认为应该数据库设计上尽量的避免出现大量的多对多关联，有的时候多对多关联可以比较简单拆分为两个一对多关联。）**HABTM** 关联也就是 **3** 张表的关联关系，关系数据库中应该说只存在多对一外键关联，所以如果要做多对多关联必然需要一张关联表来保存关联关系。

hasMany 和 **hasAndBelongsToMany** 的不同处在于，**hasMany** 关联所关联的对象只会属于本对象，不会同时属于其他对象。但是 **HABTM** 不同，所关联的对象同时会被其他对象所关联持有。比如 **Post** 和 **Tag** 之间的关联就是这种关系，一篇日志可以属于多个不同的 **Tag**，一个 **Tag** 也会包含多篇不同的日志。

为了实现多对多关联，首先要建立那张关联关系表（参照表）。除了 **"tags"** **"posts"** 表以外，根据 **Cake** 的命名约定，关联表的名字应该是[复数形式的 **model1** 名字]_[复数形式的 **model2** 名字]，至于两个 **model** 谁先谁后则根据字典排序法。

下面是一些示例：

Posts and Tags: **posts_tags**

Monkeys and IceCubes: **ice_cubes_monkeys**

Categories and Articles: **articles_categories**

关联表至少需要两个关联对象的外键字段，例如 **"post_id"** 和 **"tag_id"**。当然你也可以加入一些其他的属性。

下面是生成的数据库脚本：

Here's what the SQL dumps will look like for our Posts HABTM Tags example:

--

-- Table structure for table `posts`

--

CREATE TABLE `posts` (

`id` int(10) unsigned NOT NULL auto_increment,

`user_id` int(10) default NULL,

`title` varchar(50) default NULL,

`body` text,

`created` datetime default NULL,

`modified` datetime default NULL,

`status` tinyint(1) NOT NULL default '0',

PRIMARY KEY (`id`)

) TYPE=MyISAM;

-- -----

--

-- Table structure for table `posts_tags`

--


```

CREATE TABLE `posts_tags` (
  `post_id` int(10) unsigned NOT NULL default '0',
  `tag_id` int(10) unsigned NOT NULL default '0',
  PRIMARY KEY (`post_id`,`tag_id`)
) TYPE=MyISAM;

-- -----
--
-- Table structure for table `tags`
--
CREATE TABLE `tags` (
  `id` int(10) unsigned NOT NULL auto_increment,
  `tag` varchar(100) default NULL,
  PRIMARY KEY (`id`)
) TYPE=MyISAM;

With our tables set up, let's define the association in
the Post model:

/app/models/post.php hasAndBelongsToMany

<?php
class Post extends AppModel
{
  var $name = 'Post';
  var $hasAndBelongsToMany = array('Tag' =>
    array('className' => 'Tag',
          'joinTable' => 'posts_tags',
          'foreignKey' => 'post_id',
          'associationForeignKey'=> 'tag_id',
          'conditions' => '',
          'order' => '',
          'limit' => '',
          'uniq' => true,
          'finderQuery' => '',
          'deleteQuery' => ''
        )
    );
}
?>

```

`$hasAndBelongsToMany` array 是定义 HABTM 关联的变量，简单介绍一下需要定义的 key:

- **className (required):** 关联对象类名。
- **joinTable:** 如果你没有遵循 **Cake** 的命名约定建立关联表的话，则需要设置该 **key** 来指定关联表。
- **foreignKey:** 注意和 **associationForeignKey** 的区别，这个是定义本 **model** 在关联表中的外键字段。当然也是仅在你没有遵循 **Cake** 命名约定的时候才需要。
- **associationForeignKey:** 关联表中指向关联对象的外键字段名。
- **conditions:** 关联对象限定条件。
- **order:** 关联对象排序子句。
- **limit:** 关联对象检索数量限制。
- **uniq:** 设为 **true** 的话，重复的关联对象将被过滤掉。
- **finderQuery:** 完整的关联对象检索语句。
- **deleteQuery:** 完整的删除关联关系的 **sql** 语句。当你需要自己实现删除操作的时候可以使用该值。

1. 最后我们来看一下代码:

2.

```
3. $post = $this->Post->read(null, '2');
```

```
4. print_r($post);
```

```
5. //output:
```

```
6. Array
```

```
7. (
```

```
8.     [Post] => Array
```

```
9.     (
```

```
10.         [id] => 2
```

```
11.         [user_id] => 25
```

```
12.         [title] => Cake Model Associations
```

```
13.         [body] => Time saving, easy, and powerful.
```

```
14.         [created] => 2006-04-15 09:33:24
```

```
15.         [modified] => 2006-04-15 09:33:24
```

```
16.         [status] => 1
```

```
17.     )
```

```
18.
```

```
19.     [Tag] => Array
```

```
20.     (
```

```
21.         [0] => Array
```

```
22.         (
```

```

23.             [id] => 247
24.             [tag] => CakePHP
25.         )
26.
27.         [1] => Array
28.         (
29.             [id] => 256
30.             [tag] => Powerful Software
31.         )
32.     )
33.)

```

保存关联对象

请记住一件非常重要的事情，当保存对象时，很多时候需要同时保存关联对象，比如当我们保存 **Post** 对象和它关联的 **Comment** 对象时，我们会同时用到 **Post** 和 **Comment** 两个 **model** 的操作。

抽象来说，当关联的两个对象都没有持久化（即未保存在数据库中），你需要首先持久化主对象，或者是父对象。我们通过保存 **Post** 和关联的一条 **Comment** 这个场景来具体看看是如何操作的：

```

1. //-----Post Comment 都没有持久化-----
2. /app/controllers/posts_controller.php (partial)
3. function add()
4. {
5.     if (!empty($this->data))
6.     {
7.         //We can save the Post data:
8.         //it should be in $this->data['Post']
9.
10.        $this->Post->save($this->data);
11.
12.        //Now, we'll need to save the Comment data
13.        //But first, we need to know the ID for the
14.        //Post we just saved...
15.        $post_id = $this->Post->getLastInsertId();
16.        //Now we add this information to the save data
17.        //and save the comment.
18.        $this->data['Comment']['post_id'] = $post_id;

```

```

19.         //Because our Post hasMany Comments, we can access
20.         //the Comment model through the Post model:
21.         $this->Post->Comment->save($this->data);
22.     }
23. }

```

换一种情形，假设为现有的一篇 **Post** 添加一个新的 **Comment** 记录，你需要知道父对象的 ID。你可以通过 **URL** 来传递这个参数或者使用一个 **Hidden** 字段来提交。

```

1. /app/controllers/posts_controller.php (partial)
2. //Here's how it would look if the URL param is used...
3. function addComment($post_id)
4. {
5.     if (!empty($this->data))
6.     {
7.         //You might want to make the $post_id data more safe,
8.         //but this will suffice for a working example..
9.         $this->data['Comment']['post_id'] = $post_id;
10.        //Because our Post hasMany Comments, we can access
11.        //the Comment model through the Post model:
12.        $this->Post->Comment->save($this->data);
13.    }
14. }

```

如果你使用 **hidden** 字段来提交 ID 这个参数，你需要对这个隐藏元素命名（如果你使用 **HtmlHelper**）来正确提交：

假设日志的 ID 我们这样来命名 `$post['Post']['id']`

```
<?php echo $html->hidden('Comment/post_id', array('value' => $post['Post']['id'])); ?>
```

这样来命名的话，**Post** 对象的 ID 可以通过 `$this->data['Comment']['post_id']` 来访问，同样的通过 `$this->Post->Comment->save($this->data)` 也能非常简单的调用。

当保存多个子对象时，采用一样的方法，只需要在一个循环中调用 **save()** 方法就可以了（但是要记住使用 **Model::create()** 方法来初始化对象）。

小结一下，无论是 **belongsTo**, **hasOne** 还是 **hasMany** 关联，在保存关联子对象时候都要记住把父对象的 ID 保存在子对象中。

保存 **hasAndBelongsToMany** 关联对象

如果定义关联一样，最复杂的莫过于 **hasAndBelongsToMany** 关联，**hasOne**, **belongsTo**, **hasMany** 这 3 种关联只需要很简单的保存一下关联对象外键 ID 就可以了。但是 **hasAndBelongsToMany** 却没有那么容易了，不过我们也做了些努力，使之尽可能变得简单些。继续我们的 **Blog** 的例子，我们需要保存一个 **Post**，并且关联一些 **Tag**。

实际项目中你需要有一个单独的 **form** 来创建新的 **tag** 然后来关联它们，不过为了叙述简单，我们假定已经创建完毕了，只介绍如何关联它们的动作。

当我们在 **Cake** 中保存一个 **model**，页面上 **tag** 的名字（假设你使用了 **HtmlHelper**）应该是这样的格式 **'Model/field_name'**。
好了，让我们开始看页面代码：

```
/app/views/posts/add.thtml Form for creating posts

<h1>Write a New Post</h1>

<table>

<tr>

<td>Title:</td>

<td><?php echo $html->input('Post/title')?></td>

</tr>

<tr>

<td>Body:<td>

<td><?php echo $html->textarea('Post/title')?></td>

</tr>

<tr>

<td colspan="2">

<?php echo $html->hidden('Post/user_id',
array('value'=>$this->controller->Session->read('User.id')))?>

<?php echo $html->hidden('Post/status' ,
array('value'=>'0'))?>

<?php echo $html->submit('Save Post')?>

</td>

</tr>

</table>
```

上述页面仅仅创建了一个 **Post** 记录，我们还需要加入些代码来关联 **tag**：

```
/app/views/posts/add.thtml (Tag association code added)

<h1>Write a New Post</h1>

<table>

<tr>

<td>Title:</td>

<td><?php echo $html->input('Post/title')?></td>

</tr>

<tr>

<td>Body:</td>

<td><?php echo $html->textarea('Post/title')?></td>
```

```

        </tr>

        <tr>

            <td>Related Tags:</td>

            <td><?php echo $html->selectTag('Tag/Tag', $tags, null,
array('multiple' => 'multiple')) ?>

        </td>

        </tr>

        <tr>

            <td colspan="2">

                <?php echo $html->hidden('Post/user_id',
array('value'=>$this->controller->Session->read('User.id')))?>

                <?php echo $html->hidden('Post/status' ,
array('value'=>'0'))?>

                <?php echo $html->submit('Save Post')?>

            </td>

        </tr>

    </table>

```

我们在 **controller** 中通过调用 `$this->Post->save()` 来保存当前 **Post** 以及关联的 **tag** 信息，页面元素的命名必须是这样的格式 **"Tag/Tag"**（Cake Tag Render 之后实际的 Html Tag 名字为 **'data[ModelName][ModelName]'** 这样的格式）。提交的数据必须是单个 ID，或者是一个 ID 的 **array**。因为我们使用了一个复选框，所以这里提交的是一个 ID 的 **array**。

\$tags 变量是一个 **array**，包含了复选框所需要的 **tag** 的 ID 以及 **Name** 信息。

使用 `bindModel()` 和 `unbindModel()` 实时地改变关联关系

有的时候可能你会需要实时地，动态的改变 **model** 的关联关系，比如在一个异常情况下。特别是 **LazyLoad** 的问题，有的时候我们并需要一个完整的 **model**，所以我们可以使用 `bindModel()` 和 `unbindModel()` 来绑定或者解除绑定关联对象。

代码说话，**Start**:

```

1. leader.php and follower.php

2. <?php

3. class Leader extends AppModel

4. {

5.     var $name = 'Leader';

6.     var $hasMany = array(

7.         'Follower' => array(

8.             'className' => 'Follower',

9.             'order' => 'Follower.rank'

10.        )

```

```

11.     );

12. }

13. ?>

14. <?php

15. class Follower extends AppModel

16. {

17.     var $name = 'Follower';

18. }

19. ?>

```

上述两个 Model，在 Leader Model 中，有一个 hasMany 关联，定义了 "Leader hasMany Followers" 这样的关系。下面我们演示如何在 Controller 中动态地解除这种关联绑定。

```

1. leaders_controller.php (partial)

2. function someAction()

3. {

4.     //This fetches Leaders, and their associated Followers

5.     $this->Leader->findAll();

6.     //Let's remove the hasMany...

7.     $this->Leader->unbindModel(array('hasMany' => array('Follower')));

8.     //Now a using a find function will return Leaders, with no Followers

9.     $this->Leader->findAll();

10.    //NOTE: unbindModel only affects the very next find function.

11.    //注意: unbindModel 方法只作用一次，第二次 find 方法调用时则仍然是关联关系有效的

12.    //An additional find call will use the configured association information.

13.    //We've already used findAll() after unbindModel(), so this will fetch

14.    //Leaders with associated Followers once again...

15.    $this->Leader->findAll();

16. }

```

对于其他各种关联的 unbindModel() 的用法是类似的，你只需要更改名字和类型就可以了，下面介绍一些基础的 Usage:

通用的 unbindModel()

```
$this->Model->unbindModel(array('associationType' => array('associatedModelClassName')));
```

掌握了如何动态的解除绑定之后，让我们看看如何动态的绑定关联关系。

```

1. leaders_controller.php (partial)

2. function anotherAction()

3. {

```

```

4.    //There is no Leader hasMany Principles in the leader.php model file, so
5.    //a find here, only fetches Leaders.

6.    $this->Leader->findAll();

7.    //Let's use bindModel() to add a new association to the Principle model:

8.    $this->Leader->bindModel(

9.        array('hasMany' => array(

10.            'Principle' => array(

11.                'className' => 'Principle '

12.            )

13.        )

14.    )

15. );

16.    //Now that we're associated correctly, we can use a single find function

17.    //to fetch Leaders with their associated principles:

18.    $this->Leader->findAll();

19. }

```

`bindModel()` 方法不单能创建一个关联，同样可以用来动态的修改一个关联。

```

1. 下面是通常的用法:

2.

3. Generic bindModel() example

4. $this->Model->bindModel(

5.     array('associationName' => array(

6.         'associatedModelClassName' => array(

7.             // normal association keys go here...

8.         )

9.     )

10. )

11. );

```

注意：这些的前提是你的数据库表中的外键关联等已经正确设置。

Controller

一个 **controller** 用于管理应用程序里某一方面的逻辑。大多数来说，**controller** 被用于管理独立 **model** 的逻辑。比如，当你想构建一个管理录像带的站点，你也许会有一个 **VideoController** 和一个 **RentalController** 分别管理你的录像带和租用记录。在 **Cake** 里边，**controller** 的名字总是复数的形式。（译注：例如刚才提到过的 **VideoController**，我们应该使用 **VideosController**，而不是 **VideoController**）应用程序里的所有 **controller** 都是继承自 **Cake** 的 **AppController** 的类，而 **AppController** 类又继承自核心的 **Controller** 类。每个 **controller** 可以包含任意数量的 **action**——一些在 **web** 应用程序中用于显示视图的方法。

AppController 类可以在 `/app/appcontroller.php` 里定义并且它应该包含那些两个或者多个 **controller** 所共享的方法。它本身继承了 **Cake** 标准类库里的 **Controller** 类。一个 **action**，是 **controller** 里的一个独立的方法。在进行了适当的 **route** 配置后，当页面请求指定某个 **action** 时，这个 **action** 对应的方法将被 **Dispatcher** 自动分发执行。（It is run automatically by the Dispatcher if an incoming page request specifies it in route configuration）回到我们的录像带的例子，我们的 **VideoController** 也许会包含 `view()`、`rent()` 和 `search()` 的 **action**。这个 **controller** 可以在 `/app/controllers/videos_controller.php` 里找到并且包含以下内容：

```
1. class VideosController extends AppController
2. {
3.     function view($id)
4.     {
5.         //action logic goes here..
6.     }
7.     function rent($customer_id, $video_id)
8.     {
9.         //action logic goes here..
10.    }
11.    function search($query)
12.    {
13.        //action logic goes here..
14.    }
15. }
```

你可以通过以下的示例 **URL** 来访问这些 **action**：

```
http://www.example.com/videos/view/253
http://www.example.com/videos/rent/5124/0-235253
http://www.example.com/videos/search/hudsucker+proxy
```

但是这些页面会有怎样的外观呢？你需要为每一个 **action** 定义一个 **view**——你可以在下一章里找到具体的内容，但在本章继续我们的 **controller**——接下来的小节将会告诉你怎样利用 **Cake** 的 **controller** 的威力并且怎样发挥其优势。具体的说，你会学到怎样用你的 **controller** 传递数据给 **view**，怎样重定向用户，以及更多的功能。

Section 1 Controller 的方法

尽管本小节会介绍 **Cake** 模型中大多数频繁使用的方法，但也请记住 <http://api.cake.org> 可以获得完整的 **API** 参考。

与你的 view 进行交互

```
set
    string $var
    mixed $value
```

这个方法是你的 **view** 从 **controller** 得到数据的主要方法。你可以用它传递任何数据：单一变量值，整个数组，等等。一旦调用了 **set()**，相应的变量就可以在 **view** 里访问到了：在 **controller** 里调用 **set('color', 'blue')** 使得变量 **\$color** 在 **view** 里变得可用了。

```
validateErrors
```

返回在一次不成功的保存中生成的错误个数。

```
validate
```

根据一个 **model** 预定义的有效性规则验证该 **model** 的数据。关于数据验证的更多内容，请参见第 11 章。

```
render
    string $action
    string $layout
    string $file
```

你也许不会经常使用这个方法，因为 **render** 方法是在 **controller action** 结束时自动被调用的，输出按 **action** 名字命令的 **view**。同时，你也可以在 **controller** 逻辑里的任意位置调用来这个方法输出视图。

用户重定向

```
redirect
    string $url
```

通过此方法告诉你的用户应该继续访问什么地方。这里传入的 **URL** 参数可以是一个 **Cake** 内部 **URL**，也可以是一个完整的 **URL** (**http://...**)。

```
flash
    string $message
    string $url
    int $pause
```

该方法将在你的 **flash** 页面（该页面位于 **app/views/layouts/flash.html**）上显示提示信息 [**\$message**]，停顿时间若干 [**\$pause**] 秒，然后重定向用户到指定的 **Url** [**\$url**]。**Cake** 的 **redirect()** 和 **flash()** 方法并不包含 **exit()** 的调用。如果你希望自己的应用程序在 **redirect()** 或 **flash()** 之后停止继续执行，你需要自己在这些方法之后立即调用 **exit()**。你也可能想要 **return()** 而不是 **exit()**，依你的具体情况而定（例如，你需要执行某些回调）。

controller 中的回调函数

Cake 的 **controller** 特别提供了许多回调方法，你可以用它们在一些重要的 **controller** 方法之前或者之后插入一些逻辑。如果要利用这些功能，请在你的 **controller** 中用这里描述的参数和返回值定义这些方法。

beforeFilter

在每个 **controller action** 调用前执行。它是一个检查当前 **session** 状态和检查用户角色的好地方。

afterFilter

在每个 **controller action** 调用后执行。

beforeRender

在 **controller** 逻辑之后，并且在输出视图之前被调用。

其他有用的方法

尽管这些方法是 **Cake** 的 **Object** 类的一部分，他们在 **controller** 里仍然可用。

requestAction

```
string $url  
array $options
```

这个方法可以在任意位置调用某个 **controller** 的 **action** 并且返回 **render** 后的视图。**\$url** 是一个 **Cake URL** (**/controllername/actionname/params**)。如果 **\$extra** (译注：该方法里并没有出现 **\$extra** 参数，这里是不是应该为 **\$options**?) 数组包含 'return', 这个 **action** 的 **AutoRender** 将会自动被设置为 **true**。你可以用 **requestAction** 从另一个 **controller action** 获取数据，也可以从另一个 **controller** 获取整个输出后的视图。

首先，从一个 **controller** 获取数据是很简单的。你只需在需要数据的地方使用 **requestAction** 方法。

```
1. // Here is our simple controller:  
2. class UsersController extends AppController  
3. {  
4.     function getUserList()  
5.     {  
6.         $this->User->findAll();  
7.     }  
8. }
```

设想我们需要创建一个简单的表格来显示系统里的用户。我们不用在另一个 **controller** 里重复编码，而只需要用 **requestAction()** 来调用 **UserController::getUserList()** 就能得到用户数据。

```
1. class ProductsController extends AppController  
2. {  
3.     function showUserProducts()  
4.     {  
5.         $this->set('users', $this->requestAction('/users/getUserList'));  
6.         // Now the $users variable in the view will have the data from  
7.         // UsersController::getUserList().  
8.     }
```

```
9. }
```

如果在你的程序里，有一个频繁使用而又非静态的 **html** 元素（**element**），你也许会想使用 **requestAction()** 来将它插入到任意 **view** 里边。实际上我们不只是想要从 **UsersController::getUserList** 获取数据，还想要在另一个 **controller** 里边 **render** 这个 **action** 的 **view**（这个 **view** 也许已包含了表格）。这样的方式可以避免重复编写 **view** 的代码。

```
1. class ProgramsController extends AppController
2. {
3.     function viewAll()
4.     {
5.         $this->set('userTable', $this->requestAction('/users/getUserList', array('return')));
6.         // Now, we can echo out $userTable in this action's view to
7.         // see the rendered view that is also available at /users/getUserList.
8.     }
9. }
```

请注意这里用 **requestAction()** 调用的 **action** 用的是一个空的布局（**layout**）—这样的话你就不必担心布局与布局之间嵌套带来的麻烦了。

在使用 **AJAX** 的情况下，当在一个 **AJAX** 调用之前或调用中间想从一个 **view** 生成一个 **html** 元素，**requestAction()** 方法也是很有用的。

```
log
    string $message
    int $type = LOG_ERROR
```

你可以使用这个方法记录 **web** 应用程序里发生的不同事件。所有生成的 **log** 可以在 **Cake** 的 **/tmp** 目录下找到。

如果 **\$type** 的值为 **PHP** 常量 **LOG_DEBUG**，消息将作为 **debug** 信息被写入 **log**。对于其他任何值，消息作为 **error** 信息被写入 **log**。

```
1. // Inside a controller, you can use log() to write entries:
2. $this->log('Mayday! Mayday!');
3. //Log entry: 06-03-28 08:06:22 Error: Mayday! Mayday!
4. $this->log("Look like {$_SESSION['user']} just logged in.", LOG_DEBUG);
5. //Log entry: 06-03-28 08:06:22 Debug: Looks like Bobby just logged in.
```

```
postConditions
    array $data
```

一个你可以用其将传入的 **\$this->data** 格式化成 **model** 条件数组（**conditions array**）的方法。

例如，你有一个人员搜索表单：

```
// app/views/people/search.html:
```

提交包含这个元素的表单将会产生以下 **\$this->data** 数组：

```
Array
```

```
(
    [Person] => Array
        ([last_name] => Anderson )
)
```

在这里，我们就可以用 `postConditions()` 来格式化这些数据以便于在 `model` 里使用：

```
1. // app/controllers/people_controller.php:
2. $conditions = $this->postConditions($this->data);
3. // Yields an array looking like this:
4. Array
5. (
6.     [Person.last_name] => Anderson
7. )
8. // Which can be used in model find operations:
9. $this->Person->findAll($conditions);
```

Section 2 controller 变量

在你的 `controller` 里利用一些特殊的变量可以发挥一些额外的 `Cake` 功能：

`$name`

PHP 4 返回的类名并不遵循 **CamelCase**（驼峰命名法）格式。如果你因此遇到了问题，使用这个变量来为你的类设置正确的遵循 **CamelCase** 格式的名字。（译注：对于 `UsersController` 类，默认情况下 `Cake` 将通过 **CamelCase** 格式将“`UsersController`”拆分成“`Users`”和“`Controller`”，并以此来定位到与之对应的 `UserModel`，而 php 4 下面返回很可能是 `userscontroller`（不符合 **CamelCase**），所以导致了问题）

`$uses`

你的 `controller` 会使用多个 `model` 吗？你的 `FragglesControllers` 会自动载入 `$this->Fraggle`，但如果你还想访问 `$this->Smurf`，尝试在你的 `controller` 里添加类似以下的代码：

```
1. var $uses = array('Fraggle', 'Smurf');
```

注意：你仍然需要在 `$uses` 数组里包含你的 `Fraggle model`，即使在这之前它（`Fraggle model`）就已经自动可用了。

`$helpers`

使用这个变量来让你的 `controller` 在 `view` 里边装载 `helper`。`HTML helper` 是自动被读取的，但是你可以用这个变量来指定其他的 `helper`：

```
1. var $helpers = array('Html', 'Ajax', 'Javascript');
```

记住你仍然需要在 `$helpers` 数组里包含 `Html helper` 如果你想要使用它。它（`Html helper`）在默认情况下是可用的，但如果你定义了 `$helpers` 而没有指定它，系统将在 `view` 里边显示错误信息。

`$layout`

设置该变量的值为你想为这个 `controller` 使用的 `layout`（布局）的名字。

`$autoRender`

将这个变量设为 `false` 能让 `action` 在自动 `render` 之前自动停止。

\$beforeFilter

如果你想要一些代码在每次 **action** 被调用的时候执行（并且在该 **action** 任何代码运行之前），使用 **\$beforeFilter**。这个功能用来访问控制是非常完美的一你可以在任何 **action** 执行之前检查当前用户的权限。只要将这个变量设置成一个数组，该数组包含了你想要运行的 **controller action**（在其他 **action** 运行之前执行的 **action**）。

```
1. class ProductsController extends AppController
2. {
3.     var $beforeFilter = array('checkAccess');
4.     function checkAccess()
5.     {
6.         //Logic to check user identity and access would go here....
7.     }
8.
9.     function index()
10.    {
11.        //When this action is called, checkAccess() is called first.
12.    }
13. }
```

\$components

就像 **\$helpers** 和 **\$uses** 一样，这个变量用来装载你需要的组件：

```
var $components = array('acl');
```

Section 3 controller 参数

在你的 **Cake controller** 里，你可以通过 **\$this->params** 来访问 **controller** 的参数。这个变量用来获取传递到 **controller** 的数据，以及提供对当前请求信息的访问。**\$this->params** 最常见的用法是用于访问客户端通过 **POST** 或者 **GET** 操作递交给 **controller** 的信息。

\$this->data

用来处理来自 **HTML helper** 的 **POST** 表单数据。

```
1. // A HTML Helper is used to create a form element
2.
3. $html->input('User/first_name');
4. // When rendered in the HTML would look something like:
5.
6. <input name="data[User][first_name]" value="" type="text" />
7.
8. // And when submitted to the controller via POST,
9. // shows up in $this->data['User']['first_name']
10.
```

```

11. Array
12. (
13.     [data] => Array
14.     (
15.         [User] => Array
16.         (
17.             [username] => mrrogers
18.             [password] => myn3ighb0r
19.             [first_name] => Mister
20.             [last_name] => Rogers
21.         )
22.     )
23. )
24. )

```

`$this->params['form']`

来自任何表单的 **POST** 数据都储存在这里，包括 `$_FILES` 里的信息。

`$this->params['bare']`

如果当前布局是 **bare** 返回 `'1'`，否则返回 `'0'`。

`$this->params['ajax']`

如果当前布局是 **ajax** 返回 `'1'`，否则返回 `'0'`。

`$this->params['controller']`

返回处理该请求的当前 **controller** 的名字。例如，如果 URL `/posts/view/1` 被调用，`$this->params['controller']` 的值应该是 `'posts'`。

`$this->params['action']`

返回处理该请求的当前 **action** 的名字。例如，如果 URL `/posts/view/1` 被调用，`$this->params['action']` 的值应该是 `view`。

`$this->params['pass']`

返回当前请求传入的 **GET** 查询字符串。例如，如果 URL `/posts/view/?var1=3&var2=4` 被调用，`$this->params['pass']` 应该等于 `"?var1=3&var2=4"`。

`$this->params['url']`

返回当前被请求的 **URL**，连同 **get** 参数的键值对一起。例如如果 `/posts/view/?var1=3&var2=4` 被调用，`$this->params['url']` 应该是以下内容：

```

1. [url] => Array
2. (
3.     [url] => posts/view
4.     [var1] => 3
5.     [var2] => 4
6. )

```

Views

Section 1 Views 视图

一个 **view** 是一个页面模版，通常按 **action** 的名字命名。例如，`PostsController::add` 的 **view** 位于 `/app/views/posts/add.thtml`。**Cake view** 是非常简单的 **PHP** 文件，因此你可以在其中使用任何 **PHP** 代码。尽管你的大多数 **view** 文件包含 **HTML** 代码，一个 **view** 可以是对一个特定数据集的任意表示，**XML**，图像，等等。

在 **view** 模版文件里，你可以使用相关 **model** 提供的数据。这些数据通过 `$data` 数组传递过来。你在 **controller** 里边使用 `set()` 方法递交给 **view** 的所有数据也是可用的。

默认情况下，**HTML helper** 对于每一个 **view** 都是可用的，并且到目前为止它是 **view** 里边最常用的 **helper**。它对于创建表单，插入客户端脚本和多媒体，协助数据验证都是非常有用的。关于 **HTML helper** 的详细讨论，请参见“**Helpers**”章的 **Section 1**。

在 **view** 里使用的大多数功能都是由 **helper** 提供的。**Cake** 提供了许许多多的 **helper**（在“**Helpers**”章里讨论），并且你也可以定义自己的 **helper**。由于 **view** 不应该包含太多业务逻辑，**view** 类并没有提供太多广泛使用的公共方法。其中一个很有用的是 `renderElement()`，我们将在 **section 1.2** 看到它。

Layouts 布局

一个 **layout**（布局）包含了所有包裹在 **view** 之外的表现代码。**Layout** 文件都放在 `/app/views/layouts` 目录里。你可以重写一个默认的 **layout** 来取代位于 `/app/views/layouts/default.thtml` 的 **Cake** 默认 **layout**。一旦创建了新的默认 **layout**，当页面 **render** 的时候 **controller view** 的内容会被替换到默认 **layout** 中。

当你创建一个 **layout** 的时候，你需要告诉 **Cake** 在哪里放置 **controller view** 的内容：确保你的 **layout** 包含 `$content_for_layout`（还有 `$title_for_layout`，不过它是可选的）。以下是一个默认 **layout** 的示例：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title><?php echo $title_for_layout?></title>
<link rel="shortcut icon" href="favicon.ico" type="image/x-icon">
</head>
<body>
<!-- If you'd like some sort of menu to show up on all of your views,
include it here -->
<div id="header">
    <div id="menu">...</div>
</div>
<!-- Here's where I want my views to be displayed -->
<?php echo $content_for_layout ?>
<!-- Add a footer to each displayed page -->
<div id="footer">...</div>
</body>
</html>
```


要为 **layout** 设置标题 (**title**)，最简单的办法是在 **controller** 里使用 **\$pageTitle** 变量来设置。

```
1. class UsersController extends AppController
2. {
3.     function viewActive()
4.     {
5.         $this->pageTitle = 'View Active Users';
6.     }
7. }
```

只要需要,你可以随意为你的 **Cake** 站点创建任意数量的 **layout**,只要把他们放在 **app/views/layouts** 目录,并且在你的 **controller action** 里边使用 **controller** 的 **\$layout** 变量或者 **setLayout()** 方法来切换 **layout**。

比如,我的站点中有一个包含小型广告区域的部分,我也许可以创建一个具有更小的广告区域的 **layout** 并且使用以下语句来指定它为所有 **controller action** 的 **layout**:

```
1. var $layout = 'default_small_ad';
```

Elements 元素

许多应用程序都会有一些在各个页面见不停重复的表现层代码,有时他们只是在 **layout** 里的不同地方。**Cake** 可以帮助你复制站点中的某些区块,如果需要的话。我们将这些可重用的页面区块称为 **Element** (元素)。广告,帮助框,导航栏,菜单和插图都被 **Cake** 实现成为 **element**。一个 **element** 其实可以看作是可以包含在其他 **view** 里边的 **mini-view**。

所有 **element** 都生活在 **/app/views/elements** 目录下,并且文件扩展名为 **.thtml**。

默认情况下,**element** 访问不到任何数据。想让它具有对数据的访问权,你需要将数据放在数组里传递给它,并不要忘了为这些数据带上名字(键-值对数组)。

无参数地调用一个 **element**

```
1. <?php echo $this->renderElement('helpbox'); ?>
```

调用一个 **element**, 并传入包含数据地数组

```
1. <?php echo
2. $this->renderElement('helpbox', array("helptext" => "Oh, this text is very helpful."));
3. ?>
```

在 **element** 文件里,所有传入的变量都可以通过他们在参数数组里的键名来使用(有点类似于在 **view** 里边使用 **controller** 用 **set()** 设置的变量)。在上边这个例子中, **/app/views/elements/helpbox.thtml** 文件可以使用 **\$helptext** 变量。当然,如果传递一个数组给 **element** 会更佳有用。

element 使 **view** 具有更佳的可读性,而把 **render** 重复的 **element** 的部分放在对应的 **element** 文件中。它们也可以帮助你重用站点中的内容区块。

Component(组件)

Section 1 简介

Component 是用来在特定的场景下帮助 **controller** 更好的完成业务。与其扩展一些 **Cake** 的核心类库，反而不如写一个 **Component**，因为一些特殊的方法可以被加入到 **Component** 中。

IRC 频道中有一位 olle 同学有次这么描述 **Component**：**Component** 就是一个可以被共享的"controllerette"，也就是说是一个精简版的可以重用的 **controller**。我们觉得是一个非常不错的定义。**Component** 最主要的目标就是：重用性。**Component** 是面向 **Controller** 的，就像 **Helper** 是面向 **View** 的一样。两者最明显的差别在于 **Component** 缩减了业务逻辑，而 **Helper** 则缩减了表现层逻辑。这是非常重要的一点，我相信很多入门者都有着同样的困扰，当他想做一件事情，为了使之能够被重用，是该写成一个 **Component** 还是一个 **Helper** 呢？答案非常的简单，你是用来做什么事情呢？是完成某个业务逻辑还是某个表现层逻辑，或者是两者皆有？如果是业务逻辑，那答案就是 **Component**，如果是表现层逻辑，显然就该是一个 **Helper**。如果两者皆有，那只好写两个了，记住解耦是我们写程序最重要的地方之一。后面章节中，验证模块会就此做出示范。你需要登陆，注销，限制访问，资源访问权限（比如对 **aciton** 或者 **url** 的访问权限），这些都是业务逻辑，所以应该是一个 **Component**。但是你肯定也需要一些页面上主菜单的入口共用户登陆用，这就是表现层逻辑。不要混淆二者。

Section 2 自己动手来创建一个 Component

在 **app/controllers/components/** 路径下创建一个文件是第一步。

我们假设已经创建了 **foo.php**。然后我们在文件中定义我们的 **Component**，该类名应该为文件名加上 '**Component**' 的后缀，如下所示：

```
1. A simple component
2.
3. class FooComponent extends Object
4. {
5.     var $someVar = null;
6.     var $controller = true;
7.
8.     function startup(&$controller)
9.     {
10.         // This method takes a reference to the controller which is loading it.
11.         // Perform controller initialization here.
12.     }
13.
14.     function doFoo()
15.     {
16.         $this->someVar = 'foo';
17.     }
18. }
19.
```

当使用该 **Component** 时，你需要添加下面的代码到 **controller** 定义中去：

```
var $components = array('Foo');
```

然后如下这般调用：

```
$this->Foo->doFoo();
```

一个 **Component** 访问调用它的 **Controller** 一般是通过 **Component** 自己的 **startup()**方法来实现的。这个方法会在 **Controller::beforeFilter()**之后立即被执行。所以如果希望设置 **Component** 的一些属性，**beforeFilter** 是最佳的场所。

如果希望在 **Component** 中使用 **model**，你可以创建一个新的 **Component** 实例：

```
$foo =& new Foo();
```

你也可以在一个 **Component** 中使用其它的 **Component**。如下所示，你可以非常简单的声明其它的 **Component**。

```
var $components = array('Session');
```

Section 3 开源你的 **Component**

如果你觉得你的 **Component** 对大家都会有些帮助，请把加到 **CakeForge** 上去。如果对社区有帮助，我们会考虑把它集成到 **Core** 类库里面去。

你可以查看 [snippet archive](#) 获取其它用户提交的 **Component**。

Helper(帮助类)

Section 1 Helpers

Helper 类就是为视图提供了常见且通用的方法，更好更快的格式化或者展现数据。

HTML Helper

介绍

HTML Helper 是 Cake 用来使开发变得迅速且不会乏味的一个好方法。HTML Helper 有两个主要的目的：帮助插入那些经常用到的 HTML 代码，帮助更快更方便的创建一个 Form。下面的文章中，我们将介绍绝大多数有用的方法，不过记住 <http://api.cakephp.org> 永远是最好的参考。

HTML Helper 中的许多方法使用了 tags.ini.php 里面定义的 HTML tag 定义。Cake 的核心配置中包含了这个定义文件，不过如果你希望做些改变，复制一份 /cake/config/tags.ini.php，并且放置在 /app/config/ folder 目录下就可以了。HTML Helper 会根据这个文件中的定义来为你创建 HTML Tag。使用 HTML Helper 来创建部分视图代码会让工作变得轻松不少（我倒希望尽可能少使用 HTML Tag，这种对页面的侵入会导致页面在设计期纯粹成了代码），如果你修改了 tag 定义文件，这将是一个全局性的改变，请谨慎对待。

另外，如果 AUTO_OUTPUT 被设为 true，Helper 会自动输出 HTML Tag，而不是返回值。这是为了照顾那些不喜欢使用 short tag 或者大量的 echo() 调用的开发人员。当然你可以在方法中包含 \$return 参数来强制使用返回值，无论配置文件 AUTO_OUTPUT 被设置成 true 或者 false。

HTML helper 方法也包含了 \$htmlAttributes 参数，允许你为 tag 添加额外的属性。比如如果你希望为你的 tag 增加 class 属性，你可以通过 \$htmlAttribute 来实现：

```
array('class'=>'someClass')
```

插入良好格式的元素

如果你希望使用 Cake 来插入一些良好格式（通常是大量重复）的 HTML 元素，HTML Helper 在这方面会令你非常满意。Helper 中有许多方法可以插入各种媒体资源，帮助创建 table，甚至可以帮你通过一个 array 来创建一个树型列表。

```
charset
    string $charset
    boolean $return
```

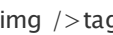
生成一个 charset 的 META-Tag

```
css
    string $path
    string $rel = 'stylesheet'
    array $htmlAttributes
    boolean $return = false
```

创建 CSS 样式文件的链接。\$rel 参数可以为该 tag 提供 rel=value，通常为 rel='stylesheet'

```
image
    string $path
    array $htmlAttributes
```

```
boolean $return = false
```

生成, 该方法的返回值可以作为参数传入 `link()` 方法生成一个图片链接。

```
link
    string $title
    string $url
    array $htmlAttributes
    string $confirmMessage = false
    boolean $escapeTitle = true
    boolean $return = false
```

该方法可以在视图中生成一个超链接。`$confirmMessage` 是一个非常有用的参数, 很多时候我们需要超链接点击后弹出一个提醒或者是确认信息, 以前你需要写一行 `js`, 现在只需要定义该参数就 `ok` 了。举个例子, 通常删除一条记录的 '删除' 链接会弹出一个“你确认要删除吗?”的确认框, 确认后才触发删除动作。`$escapeTitle` 参数设为 `true`, 可以忽略 `$title` 属性。

```
tableHeaders
    array $names
    array $tr_options
    array $th_options
```

创建指定格式的 `table` 表头

```
tableCells
    array $data
    array $odd_tr_options
    array $even_tr_options
```

创建指定格式的单元格

```
guiListTree
    array $data
    array $htmlAttributes
    string $bodyKey = 'body'
    string $childrenKey = 'children'
    boolean $return = false
```

创建一个内嵌的 `` 列表

表单与校验

通过 `HTLM helper` 在视图中快速的创建 `Form` 代码真的会让你感觉就像投入代码中的一缕阳光。它会替你生成所有的 `form tag`, 出错情况下还会自动帮你回填数据, 并显示出错信息。为了让你更好的体会到这种感觉, 让我们来看一个例子吧。假定我们现在有了一个 `Note model`, 需要创建一个 `controller`, 以及建立一个视图来新增和修改 `Note` 信息。下面是 `controller` 代码:

```
1. Edit Action inside of the NotesController

2.     function edit($id)

3.     {

4.         //First, let's check to see if any form data has been

5.         //submitted to the action.

6.         if (!empty($this->data['Note']))

7.         {

8.             //Here's where we try to validate the form data (see Chap. 12)

9.             //and save it

10.            if ($this->Note->save($this->data['Note']))

11.            {

12.                //If we've successfully saved, take the user

13.                //to the appropriate place

14.                $this->flash('Your information has been saved.', '/notes/edit/' . $id);

15.                exit();

16.            }

17.            else

18.            {

19.

20.                //Generate the error messages for the appropriate fields

21.                //this is not really necessary as save already does this, but it is an example

22.                //call $this->Note->validates($this->data['Note']); if you are not doing a save

23.                //then use the method below to populate the tagErrorMsg() helper method

24.                $this->validateErrors($this->Note);

25.

26.                //And render the edit view code

27.                $this->render();

28.            }

29.        }

30.

31.        // If we haven't received any form data, get the note we want to edit, and hand

32.        // its information to the view
```

```

33.     $this->set( 'note', $this->Note->find( "id = $id" ));
34.     $this->render();
35. }
36.

```

controller 创建好了，让我们来看一下视图代码（文件在 `app/views/notes/edit.html`）。我们的 **Note model** 非常的简单，只有 **Note ID**，提交者 **ID** 和 **Note 内容** 3 个属性。所以视图页面只需要展示这些数据并供用户来修改。

HTML helper 可以在任何一个页面上通过 `$html` 来调用。

我们只看页面上最核心的那块 **Form** 代码：

```

1. Edit View code (edit.html) sample
2. <!-- This tag creates our form tag -->
3.
4. <?php echo $html->formTag( '/notes/edit/' . $html->tagValue( 'Note/id' )?>
5.
6. <table cellpadding= "10" cellspacing= "0">
7. <tr>
8.     <td align= "right">Body: </td>
9.     <td>
10.
11.         <!-- Here's where we use the HTML helper to render the text
12.             area tag and its possible error message the $note
13.             variable was created by the controller, and contains
14.             the data for the note we're editing. -->
15.         <?php echo
16.             $html->textarea( 'Note/body', array( 'cols' => '60', 'rows' => '10' ));
17.         ?>
18.         <?php echo $html->tagErrorMsg( 'Note/body',
19.             'Please enter in a body for this note.' ) ?>
20.     </td>
21. </tr>
22. <tr>
23.     <td></td>
24.     <td>
25.
26.         <!-- We can also use the HTML helper to include

```

```

27.         hidden tags inside our table -->

28.

29.         <?php echo $html->hiddenTag('Note/id')?>

30.         <?php echo $html->hiddenTag('note/submitter_id', $this->controller->Session-
            >read('User.id'))?>

31.     </td>

32.</tr>

33.</table>

34.

35.<!-- And finally, the submit button-->

36.<?php echo $html->submit()?>

37.

38.</form>

39.

```

大部分的 **form tag** 生成函数（包括 **tagErrorMsg**）需要你提供 **\$fieldName**。这个参数能够让 **Cake** 知道你提交的是什么数据，然后可以对应的去保存和校验数据。**\$fieldName** 参数的格式为 **"modelname/fieldname"**。如果你要为 **Note model** 增加一个 **title** 属性，你可以加上下面的代码：

```

1. <?php echo $html->input('Note/title') ?>

2. <?php echo $html->tagErrorMsg('Note/title', 'Please supply a title for this note.')?>

3.

```

tagErrorMsg() 用来输出错误信息，并且会被嵌入 **<div class="error_message"></div>** 中，所以你可以很容易的控制 **CSS** 样式。

下面是所有 **HTML helper** 可以生成的 **form tag**（大部分是一目了然的）：

```

submit
    string  $buttonCaption
    array   $htmlAttributes
    boolean $return = false

password
    string  $fieldName
    array   $htmlAttributes
    boolean $return = false

textarea
    string  $fieldName

```



```
array    $htmlAttributes
```

```
boolean $return = false
```

checkbox

```
string  $fieldName
```

```
array    $htmlAttributes
```

```
boolean $return = false
```

file

```
string  $fieldName
```

```
array    $htmlAttributes
```

```
boolean $return = false
```

hidden

```
string  $fieldName
```

```
array    $htmlAttributes
```

```
boolean $return = false
```

input

```
string  $fieldName
```

```
array    $htmlAttributes
```

```
boolean $return = false
```

radio

```
string  $fieldName
```

```
array    $options
```

```
array    $inbetween
```

```
array    $htmlAttributes
```

```
boolean $return = false
```

tagErrorMsg

```
string  $fieldName
```

```
string  $message
```

HTML helper 中还包含了一组函数来生成日期相关的 tag。\$tagName 参数和\$fieldName 处理方式类似，提供该日期 tag 所对应的 model 属性。当处理数据时，你会发现 controller 中获得的日期数据被分成了几个部分，你需要做一个连接处理。

```
1. Concatenating time data before saving a model (excerpt from NotesController)

2. function edit($id)

3.     {

4.         //First, let's check to see if any form data has been submitted to the action.

5.         if (!empty($this->data['Note']))

6.         {

7.

8.             //Concatenate time data for storage...

9.             $this->data['Note']['deadline'] =

10.                 $this->data['Note']['deadline_year'] . "-" .

11.                 $this->data['Note']['deadline_month'] . "-" .

12.                 $this->data['Note']['deadline_day'];

13.

14.             //Here's where we try to validate the form data (see Chap. 10) and save it

15.             if ($this->Note->save($this->data['Note']))

16.             {

17.

18.                 ...

19.
```

```
dayOptionTag ($tagName, $value=null, $selected=null, $optionAttr=null)

yearOptionTag ($tagName, $value=null, $minYear=null, $maxYear=null,
$selected=null, $optionAttr=null)

monthOptionTag ($tagName, $value=null, $selected=null, $optionAttr=null)

hourOptionTag ($tagName, $value=null, $format24Hours=false,
$selected=null, $optionAttr=null)

minuteOptionTag ($tagName, $value=null, $selected=null, $optionAttr=null)

meridianOptionTag ($tagName, $value=null, $selected=null,
$optionAttr=null)
```

```
dateTimeOptionTag ($tagName, $dateFormat= 'DMY', $timeFormat= '12',  
$selected=null, $optionAttr=null)
```

AJAX Helper

Cake Ajax helper 使用了最为流行的 **Prototype** 和 **Scriptaculous** 两个 JS 库来完成 Ajax 操作和客户端特效（我相信 **Prototype** 改变了很多人对 JS 的看法，我就是其中一个）。为了使用该 **Helper**，你需要下载最新的 **Scriptaculous** 库（<http://script.aculo.us>），并放置在 `/app/webroot/js/` 下面。任何需要使用 **Ajax helper** 的页面都能使用该类库。

大部分 **Ajax helper** 中的函数都需要一个特别的参数 **\$options**，熟悉 **Prototype** 的人应该不会觉得陌生。**\$options** 是一个 array，可以定义 **Ajax** 操作各个选项。下面是选项的简单介绍：

AjaxHelper \$options Keys

```
/* General Options */
```

```
$options['url']           //Ajax 调用的 action 的 URL。
```

```
$options['frequency']     //remoteTimer() 或 observeField() 调用间隔
```

```
$options['update']        //根据 Ajax 调用结果去更新内容的元素 DOM ID
```

```
$options['with']          //根据指定的 ID 将元素 serialize 后随 Ajax Form 一起提交
```

```
$options['type']          //决定调用的方式是异步的还是同步的
```

```
/* Callbacks : JS code to be executed at certain
```

```
times during the XMLHttpRequest process */
```

```
$options['loading']       // 当远程文档被浏览器载入时的回调函数
```

```
$options['loaded']        // 当远程文档被浏览器载入完毕后的回调函数
```

```
$options['interactive']   // 当用户可以和远程文档交互时的回调函数，忽略是否被载入完毕
```

```
$options['complete']      // XMLHttpRequest 调用完成时的回调函数
```

```
$options['confirm']       // 在发起一个 XMLHttpRequest 前弹出的确认窗口
```

```
$options['condition']     // XMLHttpRequest 初始化前的判断条件
```

```
$options['before']        // XMLHttpRequest 初始化前调用的 JS 代码
```

```
$options['after']         // 在 XMLHttpRequest 初始化完成但 'loading' 之前立即被执行的 JS 代码
```

下面是一些让你更快更简单使用 **Ajax** 的 **helper** 函数：

link

```
string $title
```

```
string $href
```

```
array $options
```

```
boolean $confirm
```

```
boolean $escapeTitle
```

显示一个超链接，**\$title** 文本，通过 **\$options['url']** 获取远程内容并更新 **\$options['update']** 元素内容，回调函数可以使用。

```
remoteFunction
    array $options
```

该函数创建远程调用所必需的 **javascript**。主要用来作为 **linkToRemote** 的基础函数，并不太由用户直接使用，因为需要你写操作所需的 **js**。

```
remoteTimer
    array $options
```

以 **\$options['frequency']** 为频率（单位为秒，默认为 10 秒），周期性的调用 **\$options['url']** 定义的远程 **action**。通常用来根据远程调用返回结果来更新一个特定的 **div**（由 **\$options['update']** 来指定），比如 **web** 聊天窗口，**NBA** 文字直播，回调函数可以使用。

```
form
    string $action
    string $type
    array $options
```

返回一个 **form tag** 来提交到 **\$action**，使用 **XMLHttpRequest** 在后台提交代替原来页面刷新式的提交方式。**form tag** 中的 **data** 和通常的 **form** 提交一样（你可以在 **controller** 中通过 **\$this->params['form']** 使用）。提交完成后根据回发的内容来更新 **\$options['update']** 指定的元素。回调函数可以使用。

```
observeField
    string $field_id
    array $options
```

根据 **\$options['frequency']** 定义的频率侦测 **\$field_id** 所指定的元素的内容是否发生变化，如果发生变化调用 **\$options['url']** 所指定的 **action**。可以根据 **\$options['update']** 更新指定的元素或者使用 **\$options['with']** 来提交相应的元素内容。回调函数可以使用。

```
observeForm
    string $form_id
    array $options
```

和 **observeField()** 一样的功能，唯一不同的是侦测的是一个 **form** 内容的变化。

```
autoComplete
    string $field
    string $url
    array $options
```

Scriptaculous 中的那个 **autoComplete** 恐怕用过的人都会留下极其深刻的印象，而我们所做就是让他变得更好用，虽然他已经非常的好用了。**\$field** 指定绑定 **autoComplete** 的文本框。**\$url** 所指定的 **action** 需要返回一个自动完成项的列表：最基本的需要返回一个 **** 列表。如果你希望一个具有自动完成功能的文本框能够提示你 **blog** 上的日志标题，你的 **controller** 需要完成下面几件事情：

```
1. function autocomplete ()
```

```
2. {  
3.     $this->set('posts',  
4.         $this->Post->findAll(  
5.             "subject LIKE '{$this->data['Post']['subject']}'")  
6.         );  
7.     $this->layout = "ajax";  
8. }  
9.
```

10. (译注: 在生产环境下使用自动完成功能, 必须使用缓存, 否则会带来速度上的瓶颈)

11.

12. 在视图中需要定义个来展示数据

13.

```
14. <ul>  
15. <?php foreach ($posts as $post): ?>  
16. <li><?php echo $post['Post']['subject']; ?></li>  
17. <?php endforeach; ?>  
18. </ul>
```

19.

20. 真正的用户页面上只需要这样定义就能使用该自动完成功能:

21.

```
22. <form action="/users/index" method="POST">  
23.     <?php echo $ajax->autocomplete('Post/subject', '/posts/autocomplete') ?>  
24.     <?php echo $html->submit('View Post') ?>  
25. </form>
```

26.

`autocomplete()`函数会根据参数创建一个文本框, 并且当你键入内容时会显示一个 **div** 包含有自动完成内容供你选择。你可以为该 **div** 加上 **CSS** 样式, 这些 **style** 的命名都是 **Scriptaculous** 中定义好的, 所以你不能随便更改。(译注: 有兴趣的话可以看看 **controls.js**, 我相信很多情况下默认的功能并不能满足我们的需要)

```
<style type="text/css">  
div.auto_complete {  
    position      :absolute;  
    width         :250px;  
    background-color :white;  
    border        :1px solid #888;
```

```

margin          :0px;

padding         :0px;
}
li.selected { background-color: #ffb; }
</style>

```

drag

```

string $id
array  $options

```

使**\$id** 指定的元素具有拖曳的能力。你可以通过 **\$options** 来指定一些附加的选项：

```

// (版本号是 scriptaculous 的版本号)

$options['handle']      // (v1.0) 可拖曳元素句柄

$options['handle']      // (v1.5) 在 1.5 中该参数可以为一个 CSS class 的名称，即可以使用一个字符串。

    // 通过该 CSS class 名称匹配成功的第一个节点作为句柄参数传入，就和 1.0 版本中的处理是一致了。

$options['revert']      // (v1.0) 当设为 true 时，拖曳动作停止后会元素自动回到初始的位置。

$options['revert']      // (v1.5) revert 参数也可以是一个函数引用，当拖曳动作停止时触发。

$options['constraint']  // 可设置为 horizontal（水平）或 vertical（垂直），拖拽动作会限制在水平或者垂直方向上。

```

drop

```

string $id
array  $options

```

使**\$id** 所指定的元素具备拖放的能力。下面照例是一些设置项：

```

$options['accept']      // 该参数可以为 string 或者是 JavaScript string 数组，包含了 CSS class 名称。

                        // 具备拖放能力的元素只接受应用这些 CSS class 的拖曳元素。

$options['containment'] // 该参数可以为单个元素句柄或者是元素数组，具备拖放能力的元素只接受被包含在这些元素中的拖曳元素。

$options['overlap']     // 该参数可以设为 horizontal（水平）或者 vertical（垂直），可拖放元素只会在拖曳元素和自己在指定方向上重叠面积超过 50% 的情况响应该动作。

```

dropRemote

```

string $id
array  $options
array  $ajaxOptions

```

创建一个拖放元素并为之初始化一个 XMLHttpRequest，当一个拖曳元素被拖放至此，触发一个 Ajax 动作。**\$options** 参数和 **drop()** 一样，**\$ajaxOptions** 参数和 **link()** 一样。

```
sortable

    string $id

    array $options
```

sortable 函数可以让一个 **list** 对象或者是一组 **float**ed 对象（通过 **\$id** 参数指定 **DOM ID**）可以被排序。**\$options** 数组包含了排序相关的设置：

\$options['tag'] // 设定可排序的 **tag** 类型（即容器中的子元素）。对于 **UL** 和 **OL** 来说，子元素类型是 **LI**，对于其他类型的 **tag** 则需要你指定子元素类型。默认为 **li**。

\$options['only'] // 进一步限定排序子元素，只有应用给定的 **CSS class** 的子元素才能被排序。

\$options['overlap'] // 可以为 **vertical** (默认) 或者 **horizontal**。对于 **float**ed 元素或者是水平列表，请选择 **horizontal**，而对于垂直列表请选择 **vertical**。

\$options['constraint'] // 限定元素的拖动方向，**vertical** 或者 **horizontal**。

\$options['containment'] // 允许排序元素间能够拖放。该参数为一个数组，可以是元素句柄，也可以是元素 **ID**，这里说的元素指的都是作为拖放容器的元素。

\$options['handle'] // 元素句柄

```
editor

    string $id

    string $url

    array $options
```

通过 **editor** 函数第一个参数 **\$id** 所提供的 **DOM id**，为指定的元素创建一个 **in-place**(现场) **Ajax** 编辑框。函数执行后，当鼠标悬停该元素时，该元素会高亮显示，当点击时会变为一个单行文本编辑框。**editor** 函数的第二个参数 **\$url**，指定了编辑完成后提交的 **Action**。**Action** 会返回更新后的元素内容。更多关于 **in-place** 编辑器的选项请参阅 [Script.aculo.us wiki](http://Script.aculo.us/wiki)。

Javascript Helper

Javascript helper 旨在帮助开发人员输出良好格式的 **JS** 相关 **tag** 和数据。

```
codeBlock

    string $string
```

将 **\$string** 传入的代码包含在 **<script>** 标记内返回。

```
link

    string $url
```

返回一个 **JavaScript** 标价指向 **\$url** 所指定的脚本引用地址。

```
linkOut

    string $url
```

和 **link()** 函数功能一致，但是假定 **\$url** 和本应用程序不在同一个域下面。

```
escapeScript
```

```
string $script
```

去除 JavaScript 代码中的回车，单引号和双引号。

```
event
```

```
string $object
```

```
string $event
```

```
string $observer
```

```
boolean $useCapture
```

使用 **Prototype** 库中的方法来为元素添加一个事件。（正好还有在写关于 **Prototype** 的手册，可以参考以了解更多关于 **Prototype**，自卖一下）。

```
cacheEvents
```

缓存由 `event()` 创建的 JavaScript 事件。

```
writeEvents
```

输出由 `cacheEvents()` 缓存起来的事件代码。

```
includeScript
```

```
string $script
```

```
[TODO what's meaning]
```

Number Helper

Number helper 中包含了一些非常出色的函数，可以帮助你格式化数值型数据。

```
precision
```

```
mixed $number
```

```
int $precision = 3
```

指定传入参数 `$number` 的精度为 `$precision`，并返回。

```
toReadableSize
```

```
int $sizeInBytes
```

将传入的 **byte** 数值转换为具备更高可读性的数值单位。根据传入的数值，函数会返回适合的结果，并选择使用 **KB**, **MB**, **GB** 或者 **TB** 等单位。

```
toPercentage
```

```
mixed $number
```

```
int $precision = 2
```

将传入的数值格式化为相应的百分比形式返回。

Text Helper

Text helper 提供了一些格式化文本数据的函数。

highlight

```
string $text  
string $highlighter = '<span class="highlight">\1</span>'
```

[TODO check the src]

stripLinks

```
string $text
```

返回去除所有超链接的(<a href= ...)的\$text 文本。

autoLinkUrls

```
string $text  
array $htmlOptions
```

将\$text 文本中的 URL 包装在<a>标记中后返回。

autoLinkEmails

```
string $text  
array $htmlOptions
```

将\$text 文本中的 Email 地址包装在<a>标记中后返回。

autoLink

```
string $text  
array $htmlOptions
```

将\$text 文本中的 URL 和 Email 地址包装在<a>标记中后返回。

truncate

```
string $text  
int $length  
string $ending = '...'
```

根据\$length 参数的长度截取字符串并加上'...', 后缀可通过\$ending 参数来定义。(译注: Prototype1.5 新加的方法, 默认截取长度为 30, 刚写完 Prototype 相关内容。)

excerpt

```
string $text  
string $phrase  
int $radius = 100  
string $ending = '...'
```

从\$text 文本中截取摘录, 摘录\$phrase 为中心, 前后各\$radius 个字符的文本片段后返回。

```
autoLinkEmails
    string $text
    boolean $allowHtml = false
```

将文本转换为 **HTML**，和 **Textile** 或者 **Ruby** 中的 **RedCloth** 类似，只是语法上有点小小的区别。（译注：**textile** 是个很有意思的项目 <http://hobix.com/textile/quick.html>）

Time Helper

Time helper 提供了一些函数供程序员输出诸如 **Unix** 时间戳格式或者是更加易读的日期字符串。

所有的函数都能够接受合法的 **PHP** 日期字符串或者是 **Unix** 时间戳格式的参数。

```
fromString
    string $dateString
```

返回传入参数的 **Unix** 时间戳格式。

```
nice
    string $dateString
    boolean $return = false
```

返回完整格式化后的日期字符串，格式为 "D, M jS Y, H:i" 或者 'Mon, Jan 1st 2005, 12:00'。

```
niceShort
    string $dateString
    boolean $return = false
```

和 **nice()** 功能一致，不过对于昨天，今天会返回更加易读的 "Today, 12:00" "Yesterday, 12:00" 这样的格式。

```
isToday
    string $dateString
```

判断传入参数是否为今天。

```
daysAsSql
    string $begin
    string $end
    string $fieldName
    boolean $return = false
```

返回一个查询某时间段内所有数据的 **SQL** 语句片段。

```
dayAsSql
    string $dateString
    string $fieldName
    boolean $return = false
```

返回一个查询同一天内所有数据的 **SQL** 语句片段。

```
wasYesterday  
  
    string  $dateString  
  
    boolean $return = false
```

判断传入日期是否是昨天。

```
isTomorrow  
  
    string  $dateString  
  
    boolean $return = false
```

判断传入日期是否是明天。

```
toUnix  
  
    string  $dateString  
  
    boolean $return = false
```

将日期字符串转换为 **Unix** 时间戳格式，即 **PHP** 中 **strtotime()** 函数的包装函数。

```
toAtom  
  
    string  $dateString  
  
    boolean $return = false
```

将日期字符串转换为 **Atom RSS Feed** 的时间格式。

```
toRSS  
  
    string  $dateString  
  
    boolean $return = false
```

将日期字符串转换为 **RSS Feed** 的时间格式。

```
timeAgoInWords  
  
    string  $dateString  
  
    boolean $return = false
```

返回一个相对日期或者一个格式化的日期，此日期取决于当前时间和给定时间的差。**\$datetime** 的格式应该是 **strtotime()** 可解析的，像 **MySQL** 的格式。

```
relativeTime  
  
    string  $dateString  
  
    boolean $return = false
```

和 **timeAgoInWords()** 非常类似，但是多了输出未来时间戳的能力（例如 "Yesterday, 10:33", "Today, 9:42", 和 "Tomorrow, 4:34"）。

```
relativeTime
    string $timeInterval
    string $dateString
    boolean $return = false
```

如果指定的时间在指定的间隔内，返回 **true**，否则返回 **false**。时间间隔应该以数字的格式指定，单位也是：'6 hours', '2 days' 等。

Section 2 创建自己的 Helper

你在编写视图的时候是不是需要一些帮助？如果你发现有一些特别的视图逻辑，而且重复使用，你可以为此创建自己的 **Helper**。

继承扩展 Cake Helper Class

假设我们想创建一个 **helper**，它可以用来输出一个使用特定 **CSS** 样式的超链接。为了让你的逻辑适应 **Cake** 现有的 **Helper** 结构，你需要在 `/app/views/helpers` 目录下创建一个新类。我们将这个 **helper** 命名为 **LinkHelper** 吧。PHP 类文件如下：

```
1. /app/views/helpers/link.php
2. class LinkHelper extends Helper
3. {
4.     function makeEdit($title, $url)
5.     {
6.         // Logic to create specially formatted link goes here...
7.     }
8. }
9.
```

你可能需要利用一些在 **Cake Helper** 类里定义的函数：

```
output
    string $string
    boolean $return = false
```

决定是输出还是返回字符串，选择是基于 **AUTO_OUTPUT** 选项的设置（在 `/app/config/core.php`）和 `$return` 参数的值来决定的。你应该使用该函数来返回所有的数据给视图。

```
loadConfig
```

返回应用程序当前核心配置以及标签定义。

让我们使用 **output()** 来格式化我们的超链接和 **URL**，并将数据返回给视图。

```
1. /app/views/helpers/link.php (logic added)
2. class LinkHelper extends Helper
3. {
4.     function makeEdit($title, $url)
5.     {
```

```

6.         // Use the helper's output function to hand formatted
7.         // data back to the view:
8.
9.         return $this->output("<div class=\"editOuter\"><a href=\"\$url\" class=\"edit\">{$title}
    e</a></div>");
10.     }
11. }
12.

```

包含其它的 Helper

可能你会在你的 **Helper** 中用到其它现有的 **Helper**。你可以在 `$helpers` array 里指定你想使用的外部 **helper**，和你在 **controller** 包含其它 **controller** 的格式一样。

```

1. /app/views/helpers/link.php (using other helpers)
2. class LinkHelper extends Helper
3. {
4.
5.     var $helpers = array('Html');
6.
7.     function makeEdit($title, $url)
8.     {
9.         // Use the HTML helper to output
10.        // formatted data:
11.
12.        $link = $this->Html->link($title, $url, array('class' => 'edit'));
13.
14.        return $this->output("<div class=\"editOuter\">{$link}</div>");
15.    }
16. }
17.

```

使用自定义的 Helper

当你创建了自己的 **Helper** 并将其放置在 `/app/views/helpers/` 目录下，你就可以通过在 **controller** 中 `$helpers` array 变量中包含该 **Helper** 来使用它。

```

class ThingsController
{
    var $helpers = array('Html', 'Link');
}

```

```
}
```

如果你打算再别处使用 **Html Helper**，记住也要包含 **Html Helper**。命名约定和 **Model** 一样。

LinkHelper = 类名

link = **helpers array** 中的 **key**

link.php = **php** 文件的名字

捐献您的代码，我们提倡和鼓励您贡献您自定义的帮助类。

Cake's Global Constants and Functions Cake 的全局常量及方法

Section 1 全局方法

这些都是 **Cake** 领域内全局可用的方法。它们中有很多是对 **PHP** 长名方法更便利的包装，但是也有一些方法（比如 **vendor()**和 **uses()**）可以用来包含代码，或者执行其他有用的功能。如果你想有一个短小精悍的方法来做一些烦人的事情，这里就可以找到答案。

config

读取 **Cake** 的核心配置文件。如果成功，返回 **true**。

uses

```
string $lib1  
string $lib2
```

用来载入 **Cake** 的核心类库（位于 **cake/libs/**）。你需要提供不包含扩展名 **'.php'** 的类库文件名。

```
1. uses('sanitize', 'security');
```

vendor

```
string $lib1  
string $lib2...
```

用来载入位于 **/vendors** 目录下的外部类库。你需要提供不包含扩展名 **'.php'** 的类库文件名。

```
1. vendor('myWebService', 'nusoap');
```

debug

```
mixed $var  
  
Boolean $showHtml = false
```

如果应用程序的 **DEBUG** 等级被设置为非零的数字，**\$var** 的值将被输出。如果 **\$showHtml** 的值为 **true**，将在浏览器上显示友好的数据信息。

a

将传入此包装方法（译注：**wrapping function**，可以将多个参数打包成单一的数组）的所有参数打包成一个单一数组，并返回这个数组。

```
1. function someFunction()  
2. {  
3.     echo print_r(a('foo', 'bar'));  
4. }  
5.  
6. someFunction();  
7.
```

```

8. // output:
9.
10. array(
11.     [0] => 'foo',
12.     [1] => 'bar'
13. )

```

aa

将传入此包装方法的所有参数打包成一个关联数组，并返回这个数组。

```

1. echo aa('a','b');
2.
3. // output:
4.
5. array(
6.     'a' => 'b'
7. )

```

e

string \$text

对方法 `echo()` 的包装，使用起来更加方便。

low

对方法 `strtolower()` 的包装，使用起来更加方便。

up

对方法 `strtoupper()` 得包装，使用起来更加方便。

r

string \$search
string \$replace
string \$subject

对方法 `str_replace()` 得包装，使用起来更加方便。

pr

mixed \$data

一个方便的方法，它等同于：

```

1. echo "<pre>" . print_r($data) . "</pre>";

```


只有当 **DEBUG** 被设置为非零时它才会输出信息。

```
am
    array $array1
    array $array2...
```

将所有参数数组合并，并返回得到的新数组。

```
env
    string $key
```

可以从任何可用的资源获取到一个环境变量的值。你可以把它看成 **\$_SERVER** 或 **\$_ENV** 的后备（某些情况下，这两个变量也许会被禁用）。

```
cache
    string $path
    string $expires
    string $target = 'cache'
```

将 **\$data** 里的数据（译注：原文中这里并没有出现 **\$data** 参数，是否是原作者漏写了？）以缓存形式写入 **/app/tmp** 下由 **\$path** 指定的目录中。由 **\$expires** 指定的超时时间必须是一个有效的 **strtotime()** 字符串。缓存数据的 **\$target**（目标）可以为 **'cache'**（缓存）或者 **'public'**（公共数据）。

```
clearCache
    string $search
    string $path = 'views'
    string $ext
```

用来删除缓存目录下的文件，或者说清空缓存目录下的内容。

如果 **\$search** 是一个字符串，名字与之相匹配的缓存目录或文件会从缓存中被移除。**\$search** 也可以是一个数组，这个数组由需要清除的文件或目录的名字组成。如果为空，**/app/tmp/cache/views** 下的所有文件都将被清除。**\$path** 参数用来指定 **/tmp/cache** 的哪个目录将被清除。默认为 **'views'**。**\$ext** 参数用来指定你想清除的文件的扩展名。

```
stripslashes_deep
    array $array
```

递归地将数组所包含的数据里所有斜杠（**'/'**）去掉（译注：如果该数组里还包含数组元素，将自动递归地对这些数组调用这个方法）。

```
countdim
    array $array
```

以数字的形式返回数组 **\$array** 的维度。

```
fileExistsInPath
    string $file
```

在当前路径下搜索指定的文件。如果找到该文件，返回其路径；否则返回 `false`。

```
convertSlash  
    string $string
```

将指定字符串里的反斜杠（`'\'`）转换成下划线（`'_'`），并且去掉的第一个和最后一个下划线。

Section 2 CakePHP 内核定义的常量

- **ACL_CLASSNAME**：当前正在运行并管理 **CakePHP ACL** 的类的类名。这个常量是为了在适当的时候可以被用来整合第三方的类。
- **ACL_FILENAME**：包含 **ACL_CLASSNAME** 类的文件的文件名。
- **AUTO_SESSION**：如果设为 `false`，在处理对应用程序的请求时不会自动调用 `session_start()`。
- **CACHE_CHECK**：如果设为 `false`，在整个应用程序范围内对 **view** 不使用缓存。
- **CAKE_SECURITY**：按照 **CAKE_SESSION_TIMEOUT** 来决定应用程序的 **session**（会话状态）安全等级（译注：原文中这里本是“**accorance**”一词，但这个单词在字典里并不存在，笔者认为这里应该是“**accordance**”）。它可以被设为 `'low'`，`'medium'` 或者 `'high'`。依赖于这个设置，**CAKE_SESSION_TIMEOUT** 将按照下面的规则变化：
 - **low**：300
 - **medium**：100
 - **high**：10
- **CAKE_SESSION_COOKIE**：应用程序的客户端 **cookie** 的名称。
- **CAKE_SESSION_SAVE**：可以设为 `'php'`，`'file'` 或者 `'database'`：
 - **php**：Cake 使用 PHP 的默认 **session** 处理方式（通常被定义在 `php.ini` 里）。
 - **file**：**session** 数据将被储存并管理在 `/tmp` 目录下。
 - **database**：使用 **Cake** 的数据库 **session** 处理（详见“The **Cake Session Component**”（“**Cake** 的 **session** 组件”）一章）
- **CAKE_SESSION_STRING**：一个用于 **session** 管理的随机字符串。
- **CAKE_SESSION_TABLE**：储存 **session** 数据的表名（如果 **CAKE_SESSION_SAVE** == `'database'`）。不要在这里包含表名的前缀，如果你为默认数据库连接已经指定过一个前缀了。
- **CAKE_SESSION_TIMEOUT**：一个 **session** 超时所需的持续非活动时间（秒）。这个数字由 **CAKE_SECURITY** 决定。
- **COMPRESS_CSS**：如果设为 `true`，**CSS** 样式表将在输出的时候被压缩。想这样做的话，必须让 **web** 服务器具有对 `/var/cache` 目录的写入权限。在使用的时候，用 `/ccss`（而不是 `/css`）来引用你的样式表，或者使用 `Controller::cssTag()`。
- **DEBUG**：定义由 **CakePHP** 程序 **render** 的错误报告和调试（**debug**）输出的等级。可以设置为 0 到 3 之间的整数：
 - **0**：正式运行模式（**Production mode**）。没有错误输出，也不会显示调试信息。
 - **1**：开发模式（**Development mode**）。警告和错误都将被显式，并且包括调试信息。
 - **2**：和 1 一样，但是具有 **SQL** 输出。
 - **3**：和 2 一样，但是会输出当前对象的完整内存（堆内存）情况（通常是 **Controller**）。
- **LOG_ERROR**：错误常量。用来区分错误日志和错误调试活动。目前 **PHP** 支持 **LOG_DEBUG**。
- **MAX_MD5SIZE**：用来控制执行 `md5()` 哈希加密的最大输出长度（字节）。
- **WEBSERVICES**：如果设为 `true`，**Cake** 的内置 **webservices** 功能将被打开。

Section 3 CakePHP 的路径常量

- APP: 应用程序所处目录的路径。
- APP_DIR: 当前应用程序的 **app** 目录的目录名。
- APP_PATH: 应用程序 **app** 目录的绝对路径。
- CACHE: 缓存文件所在目录的路径。
- CAKE: 应用程序的 **Cake** 目录的路径。
- COMPONENTS: 应用程序的 **components** 目录的路径。
- CONFIGS: 配置文件目录的路径。
- CONTROLLER_TESTS: **controller tests** 目录的路径。
- CONTROLLERS: 应用程序的所有 **controller** 所在目录的路径。
- CSS: CSS 文件所在目录的路径。
- ELEMENTS: **elements** 目录的路径。
- HELPER_TESTS: **helper tests** 目录的路径。
- HELPERS: **helpers** 目录的路径。
- INFLECTIONS: **inflections** 目录的路径（通常在 **configuration** 目录之下）。
- JS: JavaScript 文件所在目录的路径。
- LAYOUTS: **layouts** 目录的路径。
- LIB_TESTS: Cake 类库 **tests** 目录的路径。
- LIBS: Cake **libs** 目录的路径。
- LOGS: **logs** 目录的路径。
- MODEL_TESTS: **model tests** 目录的路径。
- MODELS: **models** 目录的路径。
- SCRIPTS: Cake **scripts** 目录的路径。
- TESTS: **tests** 目录的路径（该目录为 **models**, **controller** 等等 **tests** 目录的父目录）。
- TMP: **tmp** 目录的路径。
- VENDORS: **vendors** 目录的路径。
- VIEWS: **views** 目录的路径。

Section 4 CakePHP 网站根目录（Webroot）配置的路径

- CORE_PATH: Cake 核心类库所在目录的路径。
- WWW_ROOT: 应用程序的 **webroot** 目录的路径（通常在 **/cake/** 下）。
- CAKE_CORE_INCLUDE_PATH: Cake 核心类库所在目录的路径。
- ROOT: CakePHP 的 **index.php** 文件所在目录的路径。
- WEBROOT_DIR: 应用程序 **webroot** 目录的目录名。

Data Validation 数据检验

Section 1 数据检验

创建自定义的检验规则可以确保 **model** 里的数据符合应用程序的业务规则 (**business rule**)，例如密码长度只能为 8 个字符，用户名只能包含字母，等等。数据检验的第一步是在 **model** 里创建检验规则。我们在 **model** 的定义里通过 **Model::validate** 数组来实现这一规则，例如：

/app/models/user.php

```
1. <?php
2.
3. class User extends AppModel
4. {
5.     var $name = 'User';
6.
7.     var $validate = array(
8.         'login' => '/[a-z0-9\_\-]{3,}$/i' ,
9.         'password' => VALID_NOT_EMPTY,
10.        'email' => VALID_EMAIL,
11.        'born' => VALID_NUMBER
12.    );
13. }
14.
15. ?>
```

检验规则使用和 Perl 兼容的正则表达式来定义，其中一些常用的已被预定义了，位于 `/libs/validators.php`。它们是：

- VALID_NOT_EMPTY
- VALID_NUMBER
- VALID_MAIL
- VALID_YEAR

如果在 **model** 定义里有任何检验规则出现(例如，在 `$validate` 数组里)，它们将在执行保存操作的时候(例如，在执行 **Model::save()** 方法的时候)将被解析并检验数据。要验证数据也可以直接调用 **Model::validates()** (返回 **false** 如果验证失败) 和 **Model::invalidFields()** (返回一个包含错误信息的数组)。

但是通常，数据都隐含在 **controller** 代码之中。下面的例子展示了怎样创建一个处理表单的 **action**：

Form-handling Action in /app/controllers/blog_controller.php

```
1. <?php
2.
3. class BlogController extends AppController {
```

```

4.
5.     var $uses = array('Post');
6.
7.     function add ()
8.     {
9.         if (empty($this->data))
10.        {
11.            $this->render();
12.        }
13.        else
14.        {
15.            if($this->Post->save($this->data))
16.            {
17.                //ok cool, the stuff is valid
18.            }
19.            else
20.            {
21.                //Danger, Will Robinson. Validation errors.
22.                $this->set('errorMessage', 'Please correct errors below.');
```

这个 action 使用的 view 如下所示:

The add form view in /app/views/blog/add.html

```

1. <h2>Add post to blog</h2>
2. <form action="<?php echo $html->url('/blog/add')?>" method="post">
3.     <div class="blog_add">
4.         <p>Title:
5.             <?php echo $html->input('Post/title', array('size'=>'40'))?>
```

```

6.         <?php echo $html->tagErrorMsg('Post/title', 'Title is required.')}?>
7.     </p>
8.     <p>Body
9.         <?php echo $html->textarea('Post/body') ?>
10.        <?php echo $html->tagErrorMsg('Post/body', 'Body is required.')}?>
11.    </p>
12.    <p><?=$html->submit('Save')?></p>
13. </div>
14.</form>

```

Controller::validates (*\$model* [, *\$model...*]) 用来检查任何加到该 *model* 里的自定义检验。**Controller::validationErrors()** 方法返回 *model* 抛出的任何错误信息，使得我们可以用 **tagErrorMsg()** 在 *view* 里显示它们。

如果你想执行一些不同于基于正则表达式的 **Cake** 验证的自定义验证规则，你可以用 *model* 的 **invalidate()** 方法来将一个字段 (*field*) 标价为不正确的。想象一下，当一个用户尝试创建一个已存在的用户名的时候，你需要在表单里显示一个错误信息。因为你无法让 **Cake** 用正则表达式来找出这个无效数据，你需要执行自己的检验，并且把该字段标记为无效来触发 **Cake** 对无效数据的通常处理流程。

这个 **controller** 也许和以下的代码类似：

```

1. <?php
2.
3. class UsersController extends AppController
4. {
5.     function create()
6.     {
7.         // Check to see if form data has been submitted
8.         if (!empty($this->data['User']))
9.         {
10.            //See if a user with that username exists
11.            $user = $this->User->findByUsername($this->data['User']['username']);
12.
13.            // Invalidate the field to trigger the HTML Helper's error messages
14.            if (!empty($user['User']['username']))
15.            {
16.                $this->User->invalidate('username');//populates tagErrorMsg('User/username')
17.            }
18.
19.            //Try to save as normal, shouldn't work if the field was invalidated.

```

```
20.         if($this->User->save($this->data))
21.         {
22.             $this->redirect('/users/index/saved');
23.         }
24.         else
25.         {
26.             $this->render();
27.         }
28.     }
29. }
30. }
31.
32. ?>
```

Plugins 插件

CakePHP 允许你将一系列的 controller, model 和 view 打包成一个 package (包) 并发布成一个打包好的程序 plugin, 使得其他 CakePHP 应用程序可以直接使用它们。在你的某些应用程序里有一个漂亮的用户管理模块, 或者一个简单的 bog, 或者 web service (web 服务) 模块? 把它们打包成 CakePHP 插件吧, 这样你就可以方便地把它们使用到其他程序之中。

一个 plugin 和安装了这个 plugin 的应用程序之间最主要的联系是程序的配置 (数据库连接等等)。否则, 它只能在自己有限的空间里执行操作, 看起来就好像是一个单独的应用程序一样了 (而和安装 plugin 的应用程序没有关联了)。

Section 1 创建一个 plugin

让我们重新创建一个披萨订购的 plugin 来作为示例。对于任何 CakePHP 程序来说, 什么才是更有用的呢? 做为第一步, 我们需要把所有的 plugin 放在 /app/plugins 目录里。所有 plugin 文件的父目录的目录名是非常重要的, 并且在许多地方都会使用到它, 所以谨慎地挑选一个名字吧。就这个 plugin 来说, 我们使用的名字为 'pizze'。文件目录结构的设置最终会是如下所示:

Pizza Ordering Filesystem Layout 披萨订购插件的文件系统布局:

```
/app
  /plugins
    /pizza
      /controllers      <- plugin controllers go here
      /models           <- plugin models go here
      /views            <- plugin views go here
      /pizza_app_controller.php <- plugin's AppController, named
after the plugin
      /pizza_app_model.php   <- plugin's AppModel, named after
the plugin
```

虽然对于普通程序来说, 定义 AppController 和 AppModel 并不是必要的, 但对于 plugin 来说却是非常重要的。要是你的 plugin 能工作, 你必须事先创建它们 (AppController 和 AppModel)。这两个特别的类按照 plugin 的名字来命名, 并且继承上层 application 的 AppController 和 AppModel。它们看起来就像这样:

Pizza Plugin AppController: /app/plugins/pizza_app_controller.php

```
1. <?php
2.
3. class PizzaAppController extends AppController
4. {
5.     //...
6. }
7. ?>
```

Pizza Plugin AppModel: /app/plugins/pizza_app_model.php

```
1. <?php
```



```

2.
3. class PizzaAppModel extends AppModel
4. {
5.     //...
6. }
7. ?>

```

如果你忘记定义这些类，CakePHP 会传给你“Missing Controller”的错误信息直到你纠正了这个错误。

Section 2 Plugin 的 controller （插件的控制器）

我们的 Pizza Plugin 的 controller 储存在 `/app/plugin/pizza/controllers`。由于我们需要处理的最主要的对象是披萨的订单，我们需要为 plugin 建立一个 OrdersController。尽管不是绝对必要，但强烈建议为你的 plugin controller 选取相对唯一的名字来避免与上层应用程序的命名空间（namespace）发生冲突。不难想象，也许我们的上层应用程序会有 UsersController，OrdersController 或 ProductsController: 所以你也可能需要为 controller 想一些建设性的名字，或者将 controller 的类名以 plugin 的名字开头（在这个例子里，PizzaOrdersController）。因此，我们把这个新的类 PizzaOrdersController 放在 `/app/plugins/pizza/controllers` 目录下，并且它的内容会像这样：

`/app/plugins/pizza/controllers/pizza_orders_controller.php`

```

1. <?php
2.
3. class PizzaOrdersController extends PizzaAppController
4. {
5.     var $name = 'PizzaOrders';
6.
7.     function index()
8.     {
9.         //...
10.    }
11.
12.     function placeOrder()
13.     {
14.         //...
15.    }
16. }
17. ?>

```

请注意，这个 controller 继承了 plugin 的 AppController（名字叫 Pizza AppController）而不是上层应用程序的 AppController。

Section 3 Plugin 的 model

Plugin 的 model 都存放在 `/app/plugins/pizza/models`。我们已经为这个 plugin 定义了 `PizzaOrdersController`，解西来，让我们来为这个 controller 创建 model，叫做 `PizzaOrders`（`PizzaOrders` 这个类名符合我们的命名规则，并且已经具有足够的唯一性了，所以我们就用这个名字）。

`/app/plugins/pizza/models/pizza_order.php`

```
1. <?php
2. class PizzaOrder extends PizzaAppModel
3. {
4.     var $name = 'PizzaOrder';
5. }
6. ?>
```

再次强调，注意这个类继承的是 `PizzaAppModel` 而不是 `AppModel`。

Section 4 Plugin 的 view

对于插件来说，view 的用法和普通程序完全一样。只是把它们放在指定的目录下：`/app/plugins/[plugin]/views`（对披萨示例来说，`/app/plugins/pizza/views`）。对于我们的披萨订购插件，我们至少为我们的 `PizzaOrdersController::index()` action 创建一个 view，所以我们把以下代码也包含进来：

`/app/plugins/pizza/views/pizza_orders/index.shtml`

```
<h1>Order A Pizza</h1>
<p>Nothing goes better with Cake than a good pizza!</p>
<!-- An order form of some sort might go here....-->
```

Section 5 使用 plugin

好，现在我们已经建立好所有东西，是时候发布我们的 plugin 了（尽管，我们建议你一同发布一些额外的东西，比如 `readme`，`sql` 脚本文件，等等）。

一旦 plugin 安装在了 `/app/plugins`，你就可以用 `/pluginname/controllername/action` 这样的 URL 来访问它了。在我们的披萨订购插件示例中，我们应该通过 `/pizza/pizzaOrders` 来访问 `PizzaOrdersController`。

最后，一些在 `CakePHP` 应用程序里使用 plugin 的小技巧：

- 如果你没有 `[Plugin]AppController` 和 `[Plugin]AppModel`，当你试图访问 plugin 的 controller 时会收到找不到 controller 的错误信息。
- 你可以创建一个以你的 plugin 命名的默认 controller。如果你创建了它，你可以通过 `/[plugin]/action` 来访问它。例如，一个名叫 `'users'` 的插件有一个 controller 名为 `UsersController`，那么可以用 `/users/add` 来访问这个 controller，只要这个 plugin 的 `[plugin]/controllers` 目录里并没有一个真正叫 `AddController` 的类。
- Plugin 使用 `/app/views/layouts` 目录下的 layout（布局）文件来做为其默认的 layout。
- 你也可以在你的 controller 里用 `$this->requestAction('/plugin/controller/action');` 来访问内部 plugin。
- o 如果你希望使用 `requestActoin` 方法，确保所有的 controller 和 model 的名字都尽可能地唯一。否则，你也许会收到 `"redefined class ... (类重定义)"` 的 PHP 错误信息 3002。

感谢 `Felix Geisendorfer`（`the_undefined`）为本章提供的材料。

ACL 访问控制

Section 1 理解 ACL 是如何工作的

大部分重要的资源都需要访问控制。**ACL(Access control lists)**是一种细粒度，易维护和易管理的管理程序权限的方式。**ACL** 主要处理两方面内容：一个是发起资源请求的对象，一个是希望获取的资源。用 **ACL** 的行话来说，请求使用资源的对象（通常是用户）称作**请求访问对象**，简称为 **ARO(access request object)**。而希望获取的资源（通常为 **action** 和数据）被称作**访问控制对象**，简称为 **ACO(access control object)**。我们将这些实体称之为'**object**'是因为有的时候发起请求的并不是用户本身，这种情况下你可以希望限制某些特定的 **controller** 的访问权限，而这些 **controller** 又不得不在应用程序的其他位置初始化逻辑。**ACO** 可以是任何你希望进行访问控制的对象，从 **controller** 中的 **action** 到一个 **web** 服务接口，甚至是你祖母在线日记上的某一行。

一言以蔽之：**ACL** 是用来定义一个 **ARO** 什么时候可以访问一个 **ACO**。

为了帮助你更好的理解 **ACL**，我们来看一个实际的例子。想象一下，我们的魔戒小队共同使用一台电脑。队长希望在最终决战之前保持任务的秘密和安全，具体的 **ARO** 如下：

Gandalf
Aragorn
Bilbo
Frodo
Gollum
Legolas
Gimli
Pippin
Merry

这些是系统中会发起访问请求的对象(**ACO**)。有一点需要说明：**ACL 不是** 一个用来验证用户合法性的系统。你应该已经有了保存用户信息和校验用户登录合法性的方法。当你知道了用户是谁后，**ACL** 才开始发挥作用。好了，回到我们的双塔奇谋中。

我们的大法师甘道夫需要制作一张 **ACO** 的列表，确定有哪些需要控制的资源：

Weapons	武器
The One Ring	魔戒
Salted Pork	肉干
Diplomacy	外交协议
Ale	啤酒

传统情况下，我们会使用一种矩阵来管理这些，此矩阵展示了用户和与资源间相关对象的权限情况。如果此信息存储在一个表里，它可能会像下面这样，其中 **X** 表示拒绝访问，**O** 代表允许访问：

	Weapons	The One Ring	Salted Pork	Diplomacy	Ale
Gandalf	X	X	O	O	O
Aragorn	O	X	O	O	O
Bilbo	X	X	X	X	O

Frodo	X	O	X	X	O
Gollum	X	X	O	X	X
Legolas	O	X	O	O	O
Gimli	O	X	O	X	X
Pippin	X	X	X	O	O
Merry	X	X	X	X	O

初看来，这个系统工作的非常好。这样的分配可以保证安全（只有 **Frodo** 可以碰魔戒），也可以防止一些意外（不让 **hobbits** 吃光所有的肉干）。看上去这个列表已经足够细致，并且简明易懂，但真的是这样吗？

对于一个小型系统，这样的矩阵列表可以完成任务。但是对于一个规模不断增长或者是拥有大量资源(**ACO** 和 **ARO**)的系统来说，这样一个表格将会变得非常笨重。比如我们试图控制上百个军营并且希望按组管理他们。矩阵的另一个缺点是你不能真正地按逻辑来将用户分组，并且按照逻辑分组对一组用户做出一连串的权限变更。比如，当战争一结束就能自动的允许 **Hobbit** 大快朵颐的享用肉干和啤酒，我们能很容易的改变一组用户的访问权限，为每一个用户创建独立的规则即乏味也不正确。

ACL 是采用树状结构来实现的。通常 **ARO** 一棵树，**ACO** 一棵树。使用树状结构来组织你的对象，权限仍然可以是细粒度的，而且同时可以对大局有很好的把握。最终成为白袍的甘道夫，当然明智的选择了使用 **ACL** 作为新的管理方式，结构如下：

```
Fellowship of the Ring:
Warriors
    Aragorn
    Legolas
    Gimli
Wizards
    Gandalf
Hobbits
    Frodo
    Bilbo
    Merry
    Pippin
Vistors
    Gollum
```

按照这样的方式来组织我们的团队，我们可以定义这棵树的访问控制，并且应用这些权限到任意子节点上。默认的权限是禁止访问任何资源。

```
Fellowship of the Ring: [Deny: ALL]
    Warriors [Allow: Weapons, Ale, Elven Rations, Salted Pork]
        Aragorn
    Legolas
```

Gimli	
Wizards	[Allow: Salted Pork, Diplomacy, Ale]
Gandalf	
Hobbits	[Allow: Ale]
Frodo	
Bilbo	
Merry	
Pippin	
Vistors	[Allow: Salted Pork]
Gollum	

如果我们希望知道 **Pippin** 是不是可以获得啤酒，我们首先要找出他在树种的路径，**Fellowship->Hobbits->Pippin**。我们会发现在 3 个节点拥有不同的属性，我们使用最明确和 **Pippin**，啤酒相关的权限。

Fellowship = DENY Ale, 禁止访问 (因为默认情况时禁止访问任何资源)

Hobbits = ALLOW Ale, 允许访问

Pippin = ?; 没有明确定义，所以我们依然选择允许访问

最终结果：允许获得啤酒。

树结构也同样可以让我们非常方便的进行一些细粒度的调整，而且仍然具备了对 **ARO** 做出交替的变更：

Fellowship of the Ring: [Deny: ALL]	
Warriors	[Allow: Weapons, Ale, Elven Rations, Salted Pork]
Aragorn	[Allow: Diplomacy]
Legolas	
Gimli	
Wizards	[Allow: Salted Pork, Diplomacy, Ale]
Gandalf	
Hobbits	[Allow: Ale]
Frodo	[Allow: Ring]
Bilbo	
Merry	[Deny: Ale]
Pippin	[Allow: Diplomacy]
Vistors	[Allow: Salted Pork]
Gollum	

你可以看到，**Aragorn** 比其他的战士多了使用外交协议的权限，这就是特别的微调。再说一下，默认的权限是禁止，并且只会在向下遍历树的时候被变更为允许。看一下 **Merry** 是不是可以获得啤酒，我们先查找她的路径：**Fellowship->Hobbits->Merry** 并查看对应的啤酒相关的权限：

```
Fellowship = DENY (默认禁止所有)
Hobbits = ALLOW: ale, 允许获得啤酒
Merry = DENY ale, 禁止获得啤酒
最终结果，禁止获得啤酒。
```

Section 2 定义权限：基于 ini 文件的 ACL

Cake 中第一种 **ACL** 实现是基于 **ini** 文件的。尽管 **ini** 文件非常有用并且稳定，我们仍然建议你使用基于数据库的 **ACL** 方案，因为这样就可以实时地创建新的 **ACO** 和 **ARO**。即只在非常简单的小项目或者因为某些原因不能使用数据库的情况下使用这种 **ACL** 方案。

ARO/ACO 权限设定保存在 **/app/config/acl.ini.php** 文件中。在文件的最前面你可以找到使用方法：

```
; acl.ini.php - Cake ACL Configuration
; -----
; Use this file to specify user permissions.
; aco = access control object (something in your application)
; aro = access request object (something requesting access)
;
; User records are added as follows:
;
; [uid]
; groups = group1, group2, group3
; allow = aco1, aco2, aco3
; deny = aco4, aco5, aco6
;
; Group records are added in a similar manner:
;
; [gid]
; allow = aco1, aco2, aco3
; deny = aco4, aco5, aco6
;
; The allow, deny, and groups sections are all optional.
; NOTE: groups names *cannot* ever be the same as usernames!
```

使用 `ini` 文件，你可以指定用户(ARO)，用户所属的组和他们各自的权限。同样也可以设定组的权限。如何使用 **Cake ACL component** 来检查用户的权限，请参看 11.4。

Section 3 定义权限：基于数据库的 ACL

开始

默认的 **ACL** 方案是保存在数据库中的。**dbACL** 由一组核心 **model** 和一个命令行脚本组成。**Model** 用来和数据库交互以保存和获取 **ACL** 树。而命令行脚本可以帮助你创建数据库。

首先需要正确的配置你的数据库连接，具体请参考先前的章节。

然后使用 **Cake** 提供的命令行脚本来创建储存 **ACL** 的数据表，`/cake/scripts/acl.php` 可以帮你完成这一系列操作。在命令行里 `/cake/scripts/` 目录下执行该命令：

```
$ php acl.php initdb
[TODO check in win env]
Initializing Database...
Creating access control objects table (acos)...
Creating access request objects table (aros)...
Creating relationships table (aros_acos)...
Done.
```

好了，你可以检查数据库看有没有新的表。如果你对 **Cake** 如何存储树状信息感到好奇，你可以研读一下修改过的数据库中树的遍历。基本上来说，它存储了节点以及它们在树中的位置。**ACO** 和 **ARO** 分别在各自的树中保存这些节点，并且和 **model** 相对应。

现在你可以创建你的 **ARO ACO** 树了。

创建 ARO 和 ACO

有两种方式来引用 **ARO** 或者 **ACO**。一种是给它们一个 **ID**，通常是它们对应的表的主键 **ID**。另一种是给它们一个字符串别名。这两种方式并不是互斥的。

在 **Aro model** 中定义了创建新的 **ARO** 对象的方法：`create()`。该方法有 3 个参数：`$link_id` `$parent_id` `$alias`。此方法创建一个由 `$parent_id` 指定的父对象之下的 **ACL** 对象，如果传入的 `$parent_id` 为 `null`，则作为一个根对象。`$link_id` 允许你将当前用户对对象链接到 **Cake** 的 **ACL** 结构。`$alias` 参数允许你放入一个非整型 **ID** 的对象。

在你能够创建 **ACO** 和 **ARO** 之前，我们需要加载这些类。实现加载的最简单方法是使用 `$components` 数组，并将 **Cake ACL** 组件包含在 **Controller** 里：

```
var $components = array('Acl');
```

准备完毕，让我们看一个创建对象的例子，下列代码可以放置在一个 **action** 中：

```
1. $aro = new Aro();
2.
3. // First, set up a few AROs.
4. // These objects will have no parent initially.
5.
6. $aro->create( 1, null, 'Bob Marley' );
7. $aro->create( 2, null, 'Jimi Hendrix');
```

```

8. $aro->create( 3, null, 'George Washington');
9. $aro->create( 4, null, 'Abraham Lincoln');
10.
11. // Now, we can make groups to organize these users:
12. // Notice that the IDs for these objects are 0, because
13. //     they will never tie to users in our system
14.
15. $aro->create(0, null, 'Presidents');
16. $aro->create(0, null, 'Artists');
17.
18. //Now, hook AROs to their respective groups:
19.
20. $aro->setParent('Presidents', 'George Washington');
21. $aro->setParent('Presidents', 'Abraham Lincoln');
22. $aro->setParent('Artists', 'Jimi Hendrix');
23. $aro->setParent('Artists', 'Bob Marley');
24.
25. //In short, here is how to create an ARO:
26. $aro = new Aro();
27. $aro->create($user_id, $parent_id, $alias);
28.

```

你也可以使用命令行方式创建 ARO，`$acl.php create aro <link_id> <parent_id> <alias>`

创建 ACO 的方式和 ARO 类似：

```

1. $aco = new Aco();
2.
3. //Create some access control objects:
4. $aco->create(1, null, 'Electric Guitar');
5. $aco->create(2, null, 'United States Army');
6. $aco->create(3, null, 'Fans');
7.
8. // I suppose we could create groups for these
9. // objects using setParent(), but we'll skip that
10. // for this particular example
11.

```



```
12. //So, to create an ACO:

13. $aco = new Aco();

14. $aco->create($id, $parent, $alias);

15.
```

对应的脚本命令为: `$acl.php create aco <link_id> <parent_id> <alias>`

分配权限

创建好了 ARO ACO, 我们最后来为两者分配一下权限。这需要使用 **ACL component**。让我们继续先前的例子:

```
1. // First, in a controller, we'll need access
2. // to Cake's ACL component:
3.
4. class SomethingsController extends AppController
5. {
6.     // You might want to place this in the AppController
7.     // instead, but here works great too.
8.
9.     var $components = array('Acl');
10.
11.     // Remember: ACL will always deny something
12.     // it doesn't have information on. If any
13.     // checks were made on anything, it would
14.     // be denied. Let's allow an ARO access to an ACO.
15.
16.     function someAction()
17.     {
18.         //ALLOW
19.
20.         // Here is how you grant an ARO full access to an ACO
21.         $this->Acl->allow('Jimi Hendrix', 'Electric Guitar');
22.         $this->Acl->allow('Bob Marley', 'Electric Guitar');
23.
24.         // We can also assign permissions to groups, remember?
25.         $this->Acl->Allow('Presidents', 'United States Army');
26.
27.         // The allow() method has a third parameter, $action.
```

```

28.         // You can specify partial access using this parameter.
29.         // $action can be set to create, read, update or delete.
30.         // If no action is specified, full access is assumed.
31.
32.         // Look, don't touch, gentlemen:
33.         $this->Acl->allow('George Washington', 'Electric Guitar', 'read');
34.         $this->Acl->allow('Abraham Lincoln', 'Electric Guitar', 'read');
35.
36.         //DENY
37.
38.         //Denies work in the same manner:
39.
40.         //When his term is up...
41.         $this->Acl->deny('Abraham Lincoln', 'United States Army');
42.
43.
44.     }
45. }
46.

```

这个特别的 **controller** 并不是特别有用，主要是用来展示这个流程是怎么样的。在用户管理 **controller** 中使用 **ACL component** 会使最佳的场景。当创建了一个用户之后，他的 **ARO** 同时被创建并且放置在树中合适的位置，并且根据他的标识分配特定的 **ACO** 或者是 **ACO** 组。

可以使用与 **Cake** 一起打包的命令行脚本分配权限。此语法和 **model** 中的函数有些类似，而且可以通过执行 `$php acl.php` 帮助来查看它。

Section 4 校验权限：ACL Component

校验权限是 **ACL** 中最简单的部分了：只需要使用 **ACL Component** 中的 **check()** 函数就可以了。一个很好的实践就是在 **AppController** 中放置一个 **action** 来执行 **ACL** 权限校验。这样的好处是，可以在任何 **action** 中调用该方法来校验权限，我们看一下示例：

```

1. class AppController extends Controller
2. {
3.     // Get our component
4.     var $components = array('Acl');
5.
6.     function checkAccess($aco)
7.     {

```

```
8.         // Check access using the component:
9.         $access = $this->Acl->check($this->Session->read('user_alias'), $aco, $action = "*");
10.
11.         //access denied
12.         if ($access === false)
13.         {
14.             echo "access denied";
15.             exit;
16.         }
17.         //access allowed
18.         else
19.         {
20.             echo "access allowed";
21.             exit;
22.         }
23.     }
24. }
25.
```

基本上，通过在 **AppController** 基类里包含 **ACL component**，从而使应用程序中的任何 **Controller** 中都可使用该方法。

```
$this->Acl->Check($aro, $aco, $action = '*');
```

数据清理

Section 1 在应用中使用 Sanitize

Cake 为我们提供了 **Sanitize**，这个类可以用来去除用户提交的数据中带有的恶意攻击性数据和其它一些不必要的数据。**Sanitize** 是一个核心类库，所以它可以被我们用在代码中的任何一部分，但是最好用在 **controllers** 或者 **models** 中

```
1. //首先，包含这个核心库
2. uses('sanitize');
3.
4. //接下来，创建一个新的 Sanitize 对象
5. $mrClean = new Sanitize();
6.
7. //在这里，你可以使用 Sanitize 来清除你的数据
8. //这些方法将在下个章节进行说明
9.
```

Section 2 保证在 SQL 和 HTML 中数据的安全性

这一小节将说明怎么样使用 **Sanitize** 提供的一些函数

- paranoid
 - string \$string
 - array \$allowedChars

当 **\$string** 这个字符串不是纯粹由文字数字式字符组成时，函数能去除对象中任何不需要的字符。当然你仍然可以忽略那些在 **\$allowed** 数组中的字符，以便让它们通过。

```
1. $badString = ";<script><html><    // >@@#";
2.
3. echo $mrClean->paranoid($badString);
4.
5. // output: scriphtml
6.
7. echo $mrClean->paranoid($badString, array(' ', '@'));
8.
9. // output: scriphtml    @@
10.
```

- html
 - string \$string
 - boolean \$remove = false

这个方法能帮助你获得准备在一个存在的 **html** 布局中显示的数据。如果你不想用户能破坏你的布局或插入图片或 **blog** 日志中的编写脚本、在论坛中发帖等等这些行为，那么这个函数对你特别有用。如果 **\$remove** 选项被设置为 **true**，那么任何 **html** 元素将被去除，这将比作为一个 **html** 实体提交要好得多。

```
1. $badString = '<font size="99" color="#FF0000">HEY</font><script>...</script>';
2.
3. echo $mrClean->html($badString);
4.
5. // output: <font size="99" color="#FF0000">HEY</font><script>...</script>
6.
7. echo $mrClean->html($badString, true);
8.
9. // output: font size=99 color=#FF0000 HEY fontscript...script
10.
```

- **sql**
 - **string \$string**

使用该方法可以省去你在 **sql** 语句中添加 **'\'** 来输入特殊字符，这取决于系统当前对 **magic_quotes_gpc** 的设定

- **cleanArray**
 - **array @\$dirtyArray**

这个函数是一个有着高强度和多功能的清洁器，意思是说它能被用在整个数组上(比如说 **\$this->params['form']**)。这个函数取得一个数组并清理它：没有返回值是因为 **\$dirtyArray** 是传递的对象引用。接下来的清理操作将在这个数组的每一个元素上进行(以递归的形式)：

- 不对称的空格 (包括 **0xCA**) 会被替换为符合规格的空格
- **HTML** 将被替换为与其对应的 **HTML** 实体 (包括 **'\n'** 替换为 **
**)
- 为了增加 **sql** 的安全性对特殊字符进行双重检查并且去除了回车
- 为 **sql** 增加 **'\'** (仅在上述的 **sql** 函数调用的时候进行)
- 替换用户输入的反斜线符号为正确的反斜线符号

Cake Session Component

Section 1 Cake Session 存储选项

Cake 预设 3 种 Session 数据保存方式：存储为 Cake 安装目录下的临时文件，采用 PHP 的默认机制，或者序列化到数据库中。默认情况下，Cake 采用 PHP 的默认设置。如果想要更改为采用临时文件或数据库，编辑你的核心配置文件 `/app/config/core.php`，根据需要把 `CAKE_SESSION_SAVE` 设置为“cake”，“php”，或者“database”。

core.php Session 配置

```
/*
 * CakePHP 有 3 种 Session 存储方式
 *
 * CakePHP 包含 3 种类型的 session 来保存数据库或者文件，选择您中意的方法
 *
 * 如果你想用自己的存储方式将其保存到 app/config/name.php
 *
 * 不要附值为“file”或者“database”
 *
 * 请使用以下配置：
 *
 * 设置为“cake”，保存文件到/cakedistro/tmp 目录
 *
 * 设置为“php”，采用 php 的缺省路径
 *
 * 设置为“database”，保存到数据库
 */
define('CAKE_SESSION_SAVE', 'php');
```

为了在数据库中存储 Session 数据，你需要在数据库中建立一张表。 `/app/config/sql/sessions.sql` 为创建数据库的脚本。

Section 2 使用 Cake Session Component

Cake Session Component 用来与 Session 进行交互。包含基本的 Session 读写，也包括通过 Session 来提示错误、发出提示信息（例“您的数据已经保存”）。 Session Component 在所有 Cake controller 中默认为可用。

- check
 - string \$name

检查 Session 中是否已有 \$name 为键值的数据项。

- del
 - string \$name
- delete
 - string \$name

删除\$name 指定的 Session 变量。

- error

返回最近由 Cake Session Component 产生的错误，常用于调试。

- flash
 - string \$key = 'flash'

返回最后一条 **Session** 中用 **setFlash()** 设置的消息。如果 **\$key** 已设置，将返回最近存储于其中的消息。

- **read**
 - **string \$name**

返回 **\$name** 变量值

- **renew**

通过创建新的 **session ID**，删除原有的 **ID**，将原有 **Session** 中信息传递到新的 **Session** 中来更新当前 **Session**。

- **setFlash**
 - **string \$flashMessage**
 - **string \$layout = 'default'**
 - **array \$params**
 - **string \$key = 'flash'**

将变量 **\$flashMessage** 中的信息写入 **Session**（提供给之后的 **flash()** 方法来获取）。

如果 **\$layout** 设置为“**default**”，该消息被存储为 '**<div class="message"> '.\$flashMessage.'</div> '**。如果 **\$default** 设置为"，该消息就按原样保存。如果参数为其他任何值，该消息以 **\$layout** 所指定的格式保存在 **Cake view** 里。

该方法中的 **\$params** 参数会在未来版本中赋予功能，请记得查看更新信息以获取相关内容。

参数 **\$key** 允许你在键下存储提示消息，**flash()** 方法是基于键来读取提示信息的。

- **valid**

当 **Session** 有效时返回 **true**。最好在 **read()** 操作前用它来确定你要访问的会话是否确实有效。

- **write**
 - **string \$name**
 - **mixed \$value**

将变量 **\$name**、**\$value** 写入会话。

The Request Handler Component

Section1 简介

Request Handler component 在 **Cake** 中是用来判断接收到的 **HTTP** 请求信息的。你可以使用它更好的通知 **controller** 来处理那些 **Ajax** 请求, 获取发起请求的客户端 **IP** 地址和请求类型, 或者去除无用的数据。你需要在 **controller** 中的 **\$components array** 中包含 **request handler** 来使用它。

```
class ThingsController extends AppController
{
    var $components = array('RequestHandler');

    // ...
}
```

Section 2 获取客户端请求信息

让我们看一下提供的函数:

```
accepts
string $type
```

该函数返回客户端是否能接受指定的 **content-type**, 传入的参数 **\$type** 为 **content-type** 类型。如果参数为 **null** 或者为空, 则返回客户端所能接受的全部 **content-type array**。如果传入指定的类型, 客户端能接受的话返回 **true**, 并且必须包含在 **content-type map** 中 (参见 **setContent()**)。如果 **\$type** 传入一个 **array** 的话, **array** 中的每一个字符串都将被单独测试, 如果其中有一个能被接受就返回 **true**。

```
1. class PostsController extends AppController
2. {
3.     var $components = array('RequestHandler');
4.
5.     function beforeFilter ()
6.     {
7.         if ($this->RequestHandler->accepts('html'))
8.         {
9.             // Execute code only if client accepts an HTML (text/html) response
10.        }
11.        elseif ($this->RequestHandler->accepts('rss'))
12.        {
13.            // Execute RSS-only code
14.        }
15.        elseif ($this->RequestHandler->accepts('atom'))
16.        {
```



```

17.         // Execute Atom-only code

18.     }

19.     elseif ($this->RequestHandler->accepts('xml'))

20.     {

21.         // Execute XML-only code

22.     }

23.

24.     if ($this->RequestHandler->accepts(array('xml', 'rss', 'atom')))

25.     {

26.         // Executes if the client accepts any of the above: XML, RSS or Atom

27.     }

28. }

29. }

30.

```

getAjaxVersion

如果你使用 Prototype JS 库，它会在发起的 Ajax 请求头部加上特殊的标记，该函数能够返回 Prototype 的版本。

getClientIP

返回客户端 IP。

getReferrer

返回最初发起请求的 server name。

isAjax

返回当前请求是不是由 XMLHttpRequest 发起的。

isAtom

返回客户端是否能接受 Atom feed content(application/atom+xml)。

isDelete

返回当前请求方式是否为 DELETE。

isGet

返回当前请求方式是否为 GET。

isMobile

返回当前请求客户端是否为移动设备浏览器。

isPost

返回当前请求方式是否为 POST。

isPut

返回当前请求方式是否为 PUT。

isRss

返回当前客户端是否能够接受 RSS feed content (application/rss+xml)。

isXml

返回当前客户端是否能够接受 XML content (application/xml or text/xml)。

setContent

```
string $name
```

```
string $type
```

添加 content-type 别名, 方便 accepts() 和 prefers() 使用。\$name 为 map 的 key, \$type 为 map 的 value, 可以为单个字符串或者是字符串数组, 为 MIME 类型名。内置的 content-type map 如下:

```
// Name      => Type
'js'         => 'text/javascript',
'css'        => 'text/css',
'html'       => 'text/html',
'form'       => 'application/x-www-form-urlencoded',
'file'       => 'multipart/form-data',
'xhtml'      => array('application/xhtml+xml', 'application/xhtml',
'text/xhtml'),
'xml'        => array('application/xml', 'text/xml'),
'rss'        => 'application/rss+xml',
'atom'       => 'application/atom+xml'
```

Section 3 精简数据

有的时候你可能会想从 **request** 或者 **output** 中删除一些数据。使用下列函数来完成这些操作。

stripAll

```
string $str
```

从 \$str 中去除所有的空格, 图片, 脚本 (使用 stripWhitespace(), stripImages(), and stripScripts())。

stripImages

```
string $str
```

从 \$str 中去除所有内嵌的 HTML img 标记。

stripScripts

```
string $str
```

从 \$str 中去除<script>和<style>标记内容。

stripTags

```
string $str
```

```
string $tag1
```

```
string $tag2...
```

从 `$str` 中移除参数指定的 tag，参数个数不限。

```
$someString = '<font color="#FF0000"><bold>Foo</bold></font><br><em>Bar</em>';
```

```
echo $this->RequestHandler->stripTags($someString, 'font', 'bold');
```

```
// output: Foo <em>Bar</em>
```

```
stripWhiteSpace
```

```
string $str
```

Strips whitespace from `$str`.

从 `$str` 中去除空格。

Section 4 其他有用的函数

当你的应用中包含 Ajax 请求时，Request Handler component 特别有用。`setAjax()`函数可以自动侦测 Ajax 请求，并且设置 controller 的 layout 为 Ajax layout。这样的好处在于你可以使小模块视图兼职成为 Ajax 视图。

```
1. // list.thtml
2. <ul>
3. <? foreach ($things as $thing):?>
4. <li><?php echo $thing;?></li>
5. <?endforeach;?>
6. </ul>
7.
8. //The list action of my ThingsController:
9. function list()
10. {
11.     $this->RequestHandler->setAjax($this);
12.     $this->set('things', $this->Thing->findAll());
13. }
14.
```

当请求是由平常的浏览器发起的，``会在默认的 layout 中输出。如果请求是 Ajax 请求，则会输出干净的 Ajax layout。

The Security Component

Section1 简介

Security component 可以用来保护你的 **controller action** 不受恶意的，错误的请求侵入。你可以定制 **action** 请求条件，并且可以定义如何处理那些不满足条件的请求。再说一次，使用前，请先把 **Security component** 加到 **controller** 的 **\$components array** 中。

Section 2 保护 Controller Action

该组件包含两个主要的函数来限制对 **action** 的访问：

```
requirePost
    string $action1
    string $action2
    string $action3...
```

指定某些 **action** 必须通过 **POST** 方式访问。

```
requireAuth
    string $action1
    string $action2
    string $action3...
```

确保每一个请求都经过认证校验，校验内容为 **POST** 提交过来的数据中的认证 **key** 和存储在 **User Session** 中的认证 **key**。如果两者匹配，则允许执行 **action** 动作。注意，为了保证程序的扩展性，只有在 **POST** 提交方式才会触发校验动作。如果 **action** 是由常规的 **GET** 方式请求触发，校验函数不会执行任何校验动作。为了确保最完整的安全机制，你最好组合使用 **requirePost()** 和 **requireAuth()** 两个函数，保证那些希望完全保护的 **action** 得到保护。在 **Section 4** 中会介绍如何生成认证 **key** 以及 **key** 的生命期。

```
1. 让我们先看一个简单的示例：
2.
3.
4. class ThingsController extends ApplicationController
5. {
6.     var $components = array('Security');
7.
8.     function beforeFilter()
9.     {
10.         $this->Security->requirePost('delete');
11.     }
12.
13.     function delete($id)
```

```
14.     {  
15.         // This will only happen if the action is called via an HTTP POST request  
16.         $this->Thing->del ($id);  
17.     }  
18. }  
19.
```

这里，我们指定 'delete' action 必须由 POST 方式提交请求。beforeFilter() 函数通常用来通知 Security（和大多数其他 Component）Component 来执行各自的操作。在 action 被真正执行之前，这些 Component 能完成自身被调用的动作。

你可以在地址栏输入该 action 的 URL 来测试，你会发现刚才定义的安全校验已经生效了。

Section 3 处理不合法的请求

如果有的请求不能满足我们刚才定义的安全校验条件，那会发生什么情况呢？默认情况下，该请求会石沉大海，换句话说，你将看到一个 404 错误，立即退出程序。然而，Security Component 中有一个 \$blackHoleCallback 属性，你可以设置一个回调函数来处理这些不合法的请求。

与直接返回一个 404 错误相比，通过回调函数可以执行一些额外的校验，重定向请求，或者记录客户端 IP。如果你使用了回调函数来处理这些，那退出应用程序的责任也将由你承担。如果回调函数返回 true，Security Component 会继续执行其他的安全校验规则。反之则停止校验但程序并未得到终止。

Section 4 高级请求验证

requireAuth() 函数允许你非常细致的校验是否允许访问一个 action，不过这是基于正确的使用约定上的。首先，requireAuth() 是通过比较 POST 信息中的认证 key 和用户 Session 中的认证 key 来实现校验的。所以，Security Component 必须被包含在接受请求的 Controller 中和发起请求的 Controller 中。

举个例子，如果在 PostsController 中有一个 action 要提交数据到 CommentsController 中的 action，首先，Security Component 必须被包含在两个 controller 中，分别为接受请求的 CommentsController 和发出请求的 PostsController。

每次 Security Component 被加载时，甚至于并没有真正被使用，也同样会执行下列操作：首先，它使用核心 Security 类生成一个认证 key。然后，将生成的 key、失效时间和一些附加信息置入 session 中（失效时间是在 /app/config/core.php 配置的）。再将 key 置入你的 controller 中已备后用。

在视图(view)文件中，任何使用 \$html->formTag() 生成的 form tag 都会包含一个 hidden 字段用来保存认证 key。当 form 被提交的时候，Security Component 可以将之和 Session 中的认证 key 进行比对。一旦比对完成，就会重新生成新的认证 key 供下次使用。

View Caching

Section 1 视图(View)缓存

什么是 View Cache

从 0.10.9.2378_final 版本开始, **Cake** 开始支持视图缓存(译注:为了更好的表达,我改用“页面缓存”)。我们没有开玩笑,现在你可以缓存你的页面了,而且可以标记部分视图不被缓存。可以预见当大范围使用该功能时,你的应用程序速度将获得可观的提升。

当对指定 URL 发起请求时, **Cake** 首先检查该 URL 是否已经被缓存。如果已经被缓存, **Cake** 绕开分发器(dispatcher)直接返回已经输出并缓存起来的页面。反之,则走正常的流程来输出页面。

如果你已经激活了 **Cake** 的缓存机制, **Cake** 会缓存输出的页面以备下次调用。当下一次请求该页面, **Cake** 会从缓存中提取该页面并返回。你是不是在想就这么简单? 好吧,让我们来看它是怎么来操作的。

Section 2 Cache 是如何工作的

激活缓存

默认情况下,页面缓存机制是被禁用的。为了激活此功能,你首先需要在 `/app/config/core.php` 中将 `CACHE_CHECK` 的值设为 `true`。

```
/app/config/core.php (partial)
define ('CACHE_CHECK', true);
```

这个变量决定了是否启用页面缓存机制

在你希望缓存的视图(View)对应的 controller 中包含 Cache Helper:

```
var $helpers = array('Cache');
```

然后你需要指定哪些是需要被缓存的。

Controller 中的 \$cacheAction 变量

本小节中,我们介绍如何告诉 **Cake** 哪些是需要被缓存的。这由为 controller 中的 `$cacheAction` 变量的值决定。该变量是一个 array,包含了所有希望被缓存的 action 名字和对应缓存的生命周期。这个时间的值可以是一个友好的日期字符串(例如: '1 day' 或者 '60 seconds')。

假设我们有一个 **ProductsController**,其中有几个 action 希望被缓存。下面的示例代码告诉你应该如何告知 **Cake** 哪些该被缓存,注意, **Cake** 的缓存是基于 URL 的:

```
1. $cacheAction Examples
2. //Cache a few of the most oft visited product pages for six hours:
3. var $cacheAction = array(
4.     'view/23/' => 21600,
5.     'view/48/' => 21600
6. );
7.
8. //Cache an entire action. In this case the recalled product list, for one day:
9. var $cacheAction = array('recalled/' => 86400);
10.
```

```
11. //If we wanted to, we could cache every action by setting it to a string:

12. //that is strtotime() friendly to indicate the caching time.

13. var $cacheAction = "1 hour";

14.

15. //You can also define caching in the actions using $this->cacheAction = array()...

16.
```

视图中缓存标记

有的时候我们希望页面上的部分内容不被缓存。比如你希望新上架的货品能够得到高亮显示，或类似这种具有时间特性的内容，你可以告诉 **Cake** 不要缓存这部分内容。

而如何通知 **Cake** 呢，很简单，使用 将不希望缓存的内容包起来就可以了。

```
1. <h5><cake:nocache> example</h5>

2.

3.

4. <h1> New Products! </h1>

5. <cake:nocache>

6. <ul>

7. <?php foreach ($newProducts as $product): ?>

8. <li>$product['name']</li>

9. <?endforeach;?>

10.</ul>

11.</cake:nocache>

12.
```

Clearing the cache 清空缓存

首先我们先要告诉你的是，当数据库发生变动时 **Cake** 会自动清空缓存。比如有一个视图的信息是从 **Post model** 获取的，假如 **Post model** 有了 **INSERT**, **UPDATE**, **DELETE** 操作后，**Cake** 会自动清空该缓存。

但是有的时候可能会需要手动的去清空缓存。**Cake** 为此提供了 **clearCache** 函数来执行该动作。该函数是一个全局的变量：

```
1. //Remove all cached pages that have the controller name.

2. clearCache('controller');

3.

4. //Remove all cached pages that have the controller_action name.

5. clearCache('controller_action/');

6.

7. //Remove all cached pages that have the controller_action_params name.

8. //Note: you can have multiple params
```

```
9. clearCache('controller_action_params');  
  
10.  
  
11. //You can also use an array to clear muliple caches at once.  
  
12. clearCache(array('controller_action_params','controller2_action_params));  
  
13.
```

Section 3 需要记住的事情

下面是一些你需要记住的事情：

- 在/app/config/core.php 中设置 CACHE_CHECK 为 true。
- 对于希望缓存的 Controller 必须包含 Cache Helper。
- 配置 \$cacheAction 来缓存指定的 URL。
- 使用<cake:nocache> </cake:nocache>来忽略页面上的部分内容不被缓存。
- 在数据发生变更时，Cake 会自动清空相关联的缓存。
- 可以使用 clearCache() 函数来手动清空缓存。

示例：简单的用户认证

Section 1 Big Picture

如果你刚刚接触 **CakePHP**，你会被这个例子所吸引从而将这些代码复制到你的关键业务场景或者敏感数据处理应用中去。注意，本章讨论的是 **Cake** 的内部工作机制，并不是应用的安全模块。我不确定我是不是提供了一些显而易见的安全漏洞，这个例子的目的是展示 **Cake** 内部是如何处理安全问题的，你可以如何为自己的应用创建独立的“防弹衣”

Cake 的访问控制是基于内置的 **ACL** 引擎，但是用户的认证以及认证信息的持久化是怎么样的呢？

我们发现对于时下的各种应用系统，用户认证模块是各不相同。有些喜欢使用哈希编码后的密文密码，有的喜欢使用 **LDAP** 认证，并且对于每个系统，**User model** 总是有着些许差别的。我们将这部分工作留给了你。这会不会在未来的版本中改变呢？现在我们还不是很确定，因为就目前的情况来看，将用户认证模块纳入框架还不是很值得的一件事情，创建你自己的用户认证系统是非常简单的。

你需要做 3 件事情：

- 认证用户的方式（通常为校验用户的标识，比如用户名/密码组合）
- 跟踪用户访问情况的方式（通常使用 **Session**）
- 校验用户是否已经认证通过的方式（通常是和 **Session** 进行交互）

这个示例中，我们会为一个客户管理系统创建简单的用户认证系统。这样一个虚拟项目是用来管理客户联系信息和相关的客户记录等。除了少数的公共的视图用来展示客户姓名和职务以外的所有功能都必须通过用户认证后才能访问。

我们从如何验证那些试图访问系统的用户开始。通过认证的用户信息会被 **Cake Session Component** 存储在 **PHP session** 中。我们从 **session** 中取到用户信息后就可以判断哪些操作是该用户可以执行的。

有件事情需要注意——认证并不是访问控制的同义词。我们现在所做的事情是检查用户是不是真的是他所宣称的那个用户，并且允许他访问相应部分的功能。如果你希望更加具体地调节这种访问，请参考前面的 **ACL** 章节。我们觉得 **ACL** 比较适合那样的场景，不过现在我们还是只关注最简单的用户认证。

我也要再表达一次，这并不代表这个例子只能服务于最基本应用安全。我们只是希望你提供足够的蔬菜让你建立自己的“防弹衣”。

Section 2 认证与持久化

首先，我们需要一种将用户信息持久化的方式。这个客户管理系统中我们使用数据库方式来持久化用户信息：

```
1. Table 'users', Fictional Client Management System Database
2. CREATE TABLE `users` (
3.   `id` int(11) NOT NULL auto_increment,
4.   `username` varchar(255) NOT NULL,
5.   `password` varchar(32) NOT NULL,
6.   `first_name` varchar(255) NOT NULL,
7.   `last_name` varchar(255) NOT NULL,
8.   PRIMARY KEY (`id`)
9. )
10.
```

非常简单是吧？我们的 **User model** 也同样的简单：

```
1. <?php
```

```

2. class User extends AppModel
3. {
4.     var $name = 'User';
5. }
6. ?>
7.

```

第一个要完成的是登陆的 **view** 和 **action**。这能给用户提供一个登陆的入口，同时也为系统提供了处理用户信息判断是否可以访问系统的机会。使用 **HTML helper** 可以很简单的创建该 **Form**：

```

1. /app/views/users/login.thtml
2. <?if ($error): ?>
3. <p>The login credentials you supplied could not be recognized. Please try again.</p>
4. <? endif; ?>
5.
6. <form action="<?php echo $html->url('/users/login'); ?>" method="post">
7. <div>
8.     <label for="username">Username:</label>
9.     <?php echo $html->input('User/username', array('size' => 20)); ?>
10.</div>
11.<div>
12.     <label for="password">Password:</label>
13.     <?php echo $html->password('User/password', array('size' => 20)); ?>
14.</div>
15.<div>
16.     <?php echo $html->submit('Login'); ?>
17.</div>
18.</form>
19.

```

对应这个简单的视图(view)，还需要一个 **action**（/users/login），代码如下所示：

```

1. /app/controllers/users_controller.php (partial)
2. <?php
3. class UsersController extends AppController
4. {
5.     function login()
6.     {

```

```
7.         //Don't show the error message if no data has been submitted.
8.         $this->set('error', false);
9.
10.        // If a user has submitted form data:
11.        if (!empty($this->data))
12.        {
13.            // First, let's see if there are any users in the database
14.            // with the username supplied by the user using the form:
15.
16.            $someone = $this->User->findByUsername($this->data['User']['username']);
17.
18.            // At this point, $someone is full of user data, or its empty.
19.            // Let's compare the form-submitted password with the one in
20.            // the database.
21.
22.            if(!empty($someone['User']['password']) && $someone['User']['password'] ==
                $this->data['User']['password'])
23.            {
24.                // Note: hopefully your password in the DB is hashed,
25.                // so your comparison might look more like:
26.                // md5($this->data['User']['password']) == ...
27.
28.                // This means they were the same. We can now build some basic
29.                // session information to remember this user as 'logged-in'.
30.
31.                $this->Session->write( 'User', $someone['User']);
32.
33.                // Now that we have them stored in a session, forward them on
34.                // to a landing page for the application.
35.                $this->redirect('/clients');
36.            }
37.            // Else, they supplied incorrect data:
38.            else
39.            {
```

```

40.         // Remember the $error var in the view? Let's set that to true:
41.         $this->set('error', true);
42.     }
43. }
44. }
45.
46. function logout()
47. {
48.     // Redirect users to this action if they click on a Logout button.
49.     // All we need to do here is trash the session information:
50.
51.     $this->Session->delete('User');
52.
53.     // And we should probably forward them somewhere, too...
54.
55.     $this->redirect('/');
56. }
57. }
58. ?>
59.

```

还不是很坏：如果你写的简炼点，代码应该不会超过 20 行。这个 **action** 的结果有这样两种：

- 1 用户通过认证，将信息存入 **Session**，并转向到系统首页
- 2 未通过认证，返回到登陆页面，并显示相关错误信息。

Section 3 访问校验

现在我们可以认证用户了，让我们使系统可以踢除那些不登陆就希望自己访问非公共内容的用户。

一种办法就是在 **controller** 中添加一个函数来检查 **session** 状态。

```

1. /app/app_controller.php
2. <?php
3. class AppController extends Controller
4. {
5.     function checkSession()
6.     {
7.         // If the session info hasn't been set...
8.         if (!$this->Session->check('User'))

```

```

9.         {
10.             // Force the user to login
11.             $this->redirect('/users/login');
12.             exit();
13.         }
14.     }
15. }
16. ?>
17.

```

现在你拥有了一个可以确保未经登陆的用户不能访问系统受限内容的函数了，你可以在任意级别来控制，下面是一些例子：

1. 强制所有 action 都必须经过认证

```

2. <?php
3. class NotesController extends AppController
4. {
5.     // Don't want non-authenticated users looking at any of the actions
6.     // in this controller? Use a beforeFilter to have Cake run checkSession
7.     // before any action logic.
8.
9.     function beforeFilter()
10.    {
11.        $this->checkSession();
12.    }
13. }
14. ?>
15.

```

16. 在单独的 action 中要求认证

```

17. <?php
18. class NotesController extends AppController
19. {
20.     function publicNotes($clientID)
21.     {
22.         // Public access to this action is okay...
23.     }
24.

```

```
25.     function edit($noteId)
26.     {
27.         // But you only want authenticated users to access this action.
28.         $this->checkSession();
29.     }
30. }
31. ?>
32.
```

你已经掌握了一些基础知识，可以开始实现自定义或者高级功能。我们建议整合 **Cake** 的 **ACL** 控制是个不错的开始。

Cake 的命名约定

Section 1 嗯？命名约定 ？

是的，命名约定。按照字典上的解释，**Convention** 有如下意思：

- 被普遍同意或接受的明确的惯例或看法，比如在大多数地图中都是上北下南
- 被一个群体普遍遵循的惯例或程序，尤其在社交方面。比如挥手含义的约定。
- 一个被广泛使用和接受的策略或技巧，就像在戏剧、文学或绘画中旁白的戏剧特性约定。

Cake 中的命名约定产生的魔法，我们称之为“自动魔法（**automagic**）”。我们对 CoC 的喜好自不必多言，**Cake** 在没有丢失任何灵活性的情况下将使你的开发效率达到惊人的水平。**Cake** 的命名约定真的非常简单和直观。它们萃取自有经验的 **web** 开发人员在 **web** 领域使用多年的最佳实践。

Section 2 文件的命名

文件名是带有**下划线**的。遵照一般规则，如果你有一个类 **MyNiftyClass**，那么在 **Cake** 中包含这个类的文件必须命名为 **my_nifty_class.php**。

因此如果你看到这些代码片断，你会很自然的发现：

1. 如果一个控制器 (Controller) 被命名为 **KissesAndHugsController**，
那么它的文件名必须为 **kisses_and_hugs_controller.php**（注意文件名中的 **_controller**）
2. 如果一个模型 (Model) 被命名为 **OptionValue**，那么他的文件名必须为 **option_value.php**
3. 如果一个组件 (Component) 被命名为 **MyHandyComponent**，那么他的文件名必须为 **my_handy.php**（文件名中不需要 **_component**）
4. 如果一个 Helper 被命名为 **BestHelperEver**，那么他的文件名必须为 **best_helper_ever.php**

Section 3 Models

- Model 类名是**单数**的
- Model 类名的首字母大写，而且如果是多个单词，每个单词的首字母都要大写 例如：Person, Monkey, GlassDoor, LineItem, ReallyNiftyThing
- 多重联接的表必须这样命名：按照字母排序，第一个表名 (s) _ 第二个表名 (s)
- Model 文件名使用小写加下划线的结构。 例如：person.php, monkey.php, glass_door.php, line_item.php, really_nifty_thing.php
- 和 Model 相关的数据库表名也使用小写加下划线的结构，但是它们是**复数**的。 例如：people, monkeys, glass_doors, line_items, really_nifty_things

CakePHP 命名约定是为了使代码的创建过程更加精简合理，且使代码具有更高的可读性。如果你发现前面 2 条已经掌握，你可以忽略。

- Model 命名：在 Model 定义的时候使用 **var \$name**
- 和 Model 相关的表：在 Model 定义的时候使用 **var \$useTable**

Section 4 Controllers

在 **/app/config/core.php** 中还有那么一些高级选项可以供你使用，你可以利用它们使你的 URL 精巧地适应绝大多数情况。

- Controller 类名是复数的
- Controller 类名的首字母大写，而且如果是多个单词，每个单词的首字母都要大写。类名要以 Controller 结尾。 例
如: PeopleController, MonkeysController, GlassDoorsController, LineItemsController, ReallyNiftyThingsController
- Controller 类的文件名由小写字母和下划线组成，必须以“_controller”结尾。因此如果你有一个 Controller 名为 PostsController，那么文件名必须为 posts_controller.php
- 为了 protected 成员的可见性，controller 的 action 名必须有“-”
- 为了 private 成员的可见性，controller 的 action 名必须有“-”

Section 5 Views

- Views 依照它们显示的 action 来命名
- view 的文件名依照 action 的名字，并使用小写 例如: PeopleController::worldPeace() 指向一个 view /app/views/people/world_peace.html;
MonkeysController::banana()指向一个 view /app/views/monkeys/banana.html

你可以强制一个 action 提交到一个确定的 view 上，通过在你的 action 的结尾调用
`$this->render('name_of_view_file_without_dot_html');`
来实现

Section 6 Helpers

Helper 类是以首字母大写以及 Helper 来结尾的方式命名的，文件名中要有下划线

例如: `class MyComponentComponent extends Object` 在如下文件中
/app/controllers/components/my_component.php.

如果包含的 controller 中有如下语句:

```
var $helpers = array('Html','MyHelper');
```

那么在 view 中就可以进行这样的访问:

```
$myHelper->method()。
```

Section 7 Components

Component 类名首字母要大写并以"Component"结尾，文件名则要下划线。

例如: `class MyHelperHelper extends Helper` 在如下文件中 /app/views/helpers/my_helper.php.

如果包含的 controller 中有如下语句:

```
var $helpers = array('Html','MyHelper');
```

那么在 view 中就可以进行这样的访问:

```
$this->MyComponent->method()。
```

Section 8 Vendors

Vendors 并不遵循任何的约定，原因很显然：它们是一些第三方的代码，Cake 对它们没有任何影响力