

开源 PHP 开发框架 Yii 全方位教程

开源 PHP 开发框架 Yii 全方位教程 (1) 应用 (Yii::app).....	2
开源 PHP 开发框架 Yii 全方位教程 (2) 控制器 CController.....	5
开源 PHP 开发框架 Yii 全方位教程 (3) 模型 CModel.....	9
开源 PHP 开发框架 Yii 全方位教程 (4) 视图 View.....	10
开源 PHP 开发框架 Yii 全方位教程 (5) 组件 CComponent.....	13
开源 PHP 开发框架 Yii 全方位教程 (6) 模块.....	16
开源 PHP 开发框架 Yii 全方位教程 (7) 路径别名和命名空间.....	19
开源 PHP 开发框架 Yii 全方位教程 (8) 惯例.....	21
开源 PHP 开发框架 Yii 全方位教程 (9) 开发流程.....	23
开源 PHP 开发框架 Yii 全方位教程 (11) Active Record (AR).....	29
开源 PHP 开发框架 Yii 全方位教程 (12) 片段缓存.....	41
开源 PHP 开发框架 Yii 全方位教程 (13) 页面缓存.....	44
开源 PHP 开发框架 Yii 全方位教程 (14) 动态内容.....	45
开源 PHP 开发框架 Yii 全方位教程 (15) 使用扩展.....	46
开源 PHP 开发框架 Yii 全方位教程 (16) 创建扩展.....	52
开源 PHP 开发框架 Yii 全方位教程 (17) 使用第三方库.....	57
开源 PHP 开发框架 Yii 全方位教程 (18) 定义 fixture.....	58
开源 PHP 开发框架 Yii 全方位教程 (19) 单元测试.....	60
开源 PHP 开发框架 Yii 全方位教程 (20) 功能测试.....	62
开源 PHP 开发框架 Yii 全方位教程 (21) 自动生成代码.....	64
开源 PHP 开发框架 Yii 全方位教程 (22) URL 管理.....	71

开源 PHP 开发框架 Yii 全方位教程 (1) 应用 (Yii::app)

应用代表了整个请求的运行过程。其主要任务是解析用户请求，并将其分配给相应的控制器以进行进一步的处理。它同时也是保存应用级配置的核心。因此，应用一般被称为“前端控制器”。

在入口脚本中，应用被创建为一个单例。它可以在任何位置通过 `Yii::app()` 来被访问。

应用配置

默认情况下，应用是 `CWebApplication` 类的一个实例。要对其进行定制，通常是在应用实例被创建的时候提供一个配置文件（或数组）来初始化其属性值。另一个定制应用的方法就是扩展 `CWebApplication` 类。

配置是一个键值对的数组。每个键名都对应应用实例的一个属性，相应的值为属性的初始值。举例来说，下面的代码设定了应用的 `name` 和 `defaultController` 属性。

```
1 array(  
2     'name'=>'Yii Framework',  
3     'defaultController'=>'site',  
4 )
```

复制代码

我们一般将配置保存在一个单独的 PHP 脚本中(如 `protected/config/main.php`)。在这个脚本中，我们按如下方式返回配置数组，

```
5 return array(...);
```

复制代码

为应用这些配置，我们一般将这个文件的文件名作为一个参数，传递给应用的构造器。或者像下述例子这样传递给 `Yii::createWebApplication()`，就像我们经常在入口脚本里做的那样：

```
6 $app=Yii::createWebApplication($configFile);
```

复制代码

如果应用配置非常复杂，我们可以将这分成几个文件，每个文件返回一部分配置参数。接下来，我们在主配置文件里用 PHP 的 `include()` 把其它配置文件加载进来并合并成一个配置数组。

应用的主目录

应用的主目录是指包含所有安全系数比较高的 PHP 代码和数据的根目录。在默认情况下，这个目录一般和入口脚本所在目录同级的一个子目录：`protected`。这个路径可以通过在应用配置里设置 `basePath` 属性来改变。

不应该让 WEB 用户访问应用文件夹里的内容。在 Apache HTTP 服务器里，我们可以在这

个文件夹里放一个 `.htaccess` 文件来实现。`.htaccess` 的文件内容是这样的：

```
7 deny from all
```

复制代码

应用组件

我们可以很容易的通过组件(`component`)设置和丰富一个应用(`Application`)的功能。一个应用可以有应用组件，每个组件都执行一些特定的功能。比如说，一个应用可能通过 `CUrlManager` 和 `CHttpRequest` 组件来解析用户的访问请求。

通过配置应用的 `components` 属性，我们可以为应用中的每个应用组件，配置类名及其参数。例如，我们可以配置 `CMemCache` 组件以使用服务器的内存当缓存：

```
8 array(  
9     .....  
10    'components'=>array(  
11        .....  
12        'cache'=>array(  
13            'class'=>'CMemCache',  
14            'servers'=>array(  
15                array('host'=>'server1', 'port'=>11211, 'weight'=>60),  
16                array('host'=>'server2', 'port'=>11211, 'weight'=>40),  
17            ),  
18        ),  
19    ),  
20 )
```

复制代码

- 在上述例子中，我们将 `cache` 元素加在 `components` 数组里。这个 `cache` 元素告诉我们这个组件的类是 `CMemCache`，以及其 `servers` 属性应该如何初始化。

要调用组件，可以使用：`Yii::app()->ComponentID`，其中 `ComponentID` 是指这个组件的 ID。（比如 `Yii::app()->cache`）。

我们可以在应用配置里，将 `enabled` 设置为 `false` 来关闭一个组件。当我们访问一个被禁止的组件时，系统会返回一个 `NULL` 值。

默认情况下，应用组件是根据需要而创建的。这意味着一个组件只有在被访问的情况下才会创建。因此，系统的整体性能不会因为配置了很多组件而下降。有些组件，（比如 `CLogRouter`）是不管用不用都要创建的。在这种情况下，我们在应用的配置文件里将这些组件的 ID 加入到应用的 `preload` 属性中。

应用的核心组件

Yii 预定义了一套核心应用组件提供 Web 应用程序的常见功能。例如，`request` 组件用于解析用户请求和提供网址、`cookie` 等信息。几乎在每一个方面，我们都可以通过配置这

些核心组件的属性，来更改 Yii 的默认行为。

下面我们列出 CWebApplication 预先声明的核心组件。

assetManager: CAssetManager -管理发布私有 asset 文件。

- authManager: CAuthManager - 管理基于角色控制 (RBAC)。
- cache: CCache - 提供数据缓存功能。请注意，您必须指定实际的类（例如 CMemCache, CDbCache）。否则，将返回空当访问此元件。
- clientScript: CClientScript -管理客户端脚本(javascripts and CSS)。
- coreMessages: CPhpMessageSource -提供翻译 Yii 框架使用的核心消息。
- db: CDbConnection - 提供数据库连接。请注意，你必须配置它的 connectionString 属性才能使用此元件。
- errorHandler: CErrorHandler - 处理没有捕获的 PHP 错误和例外。
- format: CFormatter - 为显示目的格式化数据值。已自版本 1.1.0 可用。
- messages: CPhpMessageSource - 提供翻译 Yii 应用程序使用的消息。
- request: CHttpRequest - 提供和用户请求相关的信息。
- securityManager: CSecurityManager -提供安全相关的服务,例如散列 (hashing), 加密 (encryption)。
- session: CHttpSession - 提供会话 (session) 相关功能。
- statePersister: CStatePersister - 提供全局持久方法 (global state persistence method)。
- urlManager: CUrlManager - 提供网址解析和某些函数。
- user: CWebUser - 代表当前用户的身份信息。
- themeManager: CThemeManager - 管理主题 (themes)。
-

应用的生命周期

当处理一个用户请求时，一个应用程序将经历如下生命周期：

使用 CApplication::preinit() 预初始化应用。

- 建立类自动加载器和错误处理；
- 注册核心应用组件；
- 载入应用配置；
 - 用 CApplication::init()初始化应用程序。
 - 注册应用行为；
 - 载入静态应用组件；
- 触发 onBeginRequest 事件；
 - 处理用户请求；
 - 解析用户请求；
 - 创建控制器；
 - 执行控制器；
- 触发 onEndRequest 事件；

开源 PHP 开发框架 Yii 全方位教程 (2) 控制器 CController

1 控制器是 `CController` 或者其子类的实例。控制器在用户请求应用时创建。控制器执行所请求的 `action`，`action` 通常加载必要的模型并渲染恰当的视图。最简单的 `action` 仅仅是一个控制器类方法，此方法的名字以 `action` 开始。

控制器有默认的 `action`。用户请求不能指定哪一个 `action` 执行时，将执行默认的 `action`。缺省情况下，默认的 `action` 名为 `index`。可以通过设置 `CController::defaultAction` 改变默认的 `action`。

下边是最小的控制器类。因此控制器未定义任何 `action`，请求时会抛出异常。`class SiteController extends CController`

```
2 {  
3 }
```

复制代码

路由

控制器和 `actions` 通过 ID 进行标识的。控制器 ID 的格式：`path/to/xyz` 对应的类文件 `protected/controllers/path/to/xyzController.php`，相应的 `xyz` 应该用实际的控制器名替换（例如 `post` 对应 `protected/controllers/PostController.php`）。Action ID 与 是没有 `action` 前缀的 `action` 方法名字。例如，控制器类包含一个 `actionEdit` 方法，对应的 `action ID` 就是 `edit`。

注意：在 1.0.3 版本之前，控制器 ID 的格式是 `path.to.xyz` 而不是 `path/to/xyz`。

用户请求一个特定的 `controller` 和 `action` 用术语即为 路由。路由一个 `controller ID` 和一个 `action ID` 连结而成，二者中间以斜线分隔。例如，`route post/edit` 引用的是 `PostController` 和它的 `edit action`。默认情况下，URL `http://hostname/index.php?r=post/edit` 将请求此 `controller` 和 `action`。

注意：默认地情况下，路由是大小写敏感的。从版本 1.0.1 开始，可以让其大小写不敏感，通过应用配置中设置 `CUrlManager::caseSensitive` 为 `false`。当在大小写不敏感模式下，确保你遵照约定：包含 `controller` 类文件的目录是小写的，`controller map` 和 `action map` 都使用小写的 `keys`。

自版本 1.0.3，一个应用可以包含 模块 (`module`)。一个 `module` 中的 `controller` 的 `route` 格式是 `moduleID/controllerID/actionID`。更多细节，查阅 `section about modules`。

控制器实例化

`CWebApplication` 在处理一个新请求时，实例化一个控制器。程序通过控制器的 ID，并按如下规则确定控制器类及控制器类所在位置

- 若设置了 `CWebApplication::catchAllRequest`，一个基于此属性的 `controller` 将被创建，同时用户指定的 `controller ID` 将被忽略。这主要用来将 `application` 置于维护模式，并显示一个静态的提醒页面。

- 若此 ID 出现在 `CWebApplication::controllerMap`, 对应的 `controller` 配置将被用来创建此 `controller` 实例.
- 若此 ID 的格式是 `'path/to/xyz'`, `controller` 类名字被假定为 `XyzController` 而相应的类文件是 `protected/controllers/path/to/XyzController.php`. 例如, 一个 `controller` ID `admin/user` 将被解析为 `controller` 类 `UserController`, `class` 文件是 `protected/controllers/admin/UserController.php`. 若此 `class` 文件不存在, 会触发一个 `404 CHttpException`

4

一旦使用了 `modules` (自版本 1.0.3 可用), 上面的过程有少许不同. 特别的, `application` 将检查此 ID 是否引用的是一个 `module` 中的 `controller`, 如果是, 此 `module` 实例首先被创建, 然后创建 `controller` 实例.

Action

如之前所述, 一个 `action` 可以被定义为一个方法, 其名字以单词 `action` 开头. 一个更高级的方式是定义一个 `action` 类, 当它被请求的时候让 `controller` 实例化它. 这将允许 `action` 可被重用, 因此更加具有可重用性.

要定义一个新 `action` 类, 这样做:

```
class UpdateAction extends CAction
```

```
5 {
6     public function run()
7     {
8         // place the action logic here
9     }
10 }
```

复制代码

11 要让 `controller` 知道此 `action` 的存在, 我们重写 `controller` 类的 `actions()` 方法:

```
class PostController extends CController
```

```
12 {
13     public function actions()
14     {
15         return array(
16             'edit' => 'application.controllers.post.UpdateAction',
17         );
18     }
19 }
```

复制代码

20 如上所示, 使用路径别名 `application.controllers.post.UpdateAction` 确定 `action` 类文件为 `protected/controllers/post/UpdateAction.php`.

编写基于类的(class-based) action, 我们可以以模块化的方式组织程序。例如, 可以使用下边的目录结构组织控制器代码:protected/

```
21     controllers/  
22         PostController.php  
23         UserController.php  
24     post/  
25         CreateAction.php  
26         ReadAction.php  
27         UpdateAction.php  
28     user/  
29         CreateAction.php  
30         ListAction.php  
31         ProfileAction.php  
32         UpdateAction.php
```

复制代码

33 过滤器(Filter)

Filter 是一个代码片段,被配置用来在一个控制器的动作执行之前/后执行。例如, **an access control filter** 可被执行以确保在执行请求的 **action** 之前已经过验证; 一个 **performance filter** 可被用来衡量此 **action** 执行花费的时间。

一个 **action** 可有多于一个 **filter**. **filter** 以出现在 **filter** 列表中的顺序来执行. 一个 **filter** 可以阻止当前 **action** 及剩余未执行的 **filter** 的执行。

一个 **filter** 可被定义为一个 **controller** 类的方法. 此方法的名字必须以 **filter** 开始. 例如,方法 **filterAccessControl** 的存在定义了一个名为 **accessControl** 的 **filter**. 此 **filter** 方法必须如下:public function filterAccessControl(\$filterChain)

```
34 {  
35     // call $filterChain->run() to continue filtering and action execution  
36 }
```

复制代码

37 **\$filterChain** 是 **CFilterChain** 的一个实例, **CFilterChain** 代表了与被请求的 **action** 相关的 **filter** 列表. 在此 **filter** 方法内部, 我们可以调用 **\$filterChain->run()** 以继续 执行其他过滤器以及 **action** 的执行。

一个 **filter** 也可以是 **CFilter** 或其子类的一个实例. 下面的代码定义了一个新的 **filter** 类: `class PerformanceFilter extends CFilter`

```

38 {
39     protected function preFilter($filterChain)
40     {
41         // logic being applied before the action is executed
42         return true; // false if the action should not be executed
43     }
44     protected function postFilter($filterChain)
45     {
46         // logic being applied after the action is executed
47     }
48 }

```

复制代码

49 要应用 **filter** 到 **action**, 我们需要重写 **CController::filters()** 方法. 此方法应当返回一个 **filter** 配置数组. 例如, `class PostController extends CController`

```

50 {
51     .....
52     public function filters()
53     {
54         return array(
55             'postOnly + edit, create',
56             array(
57                 'application.filters.PerformanceFilter - edit, create',
58                 'unit'=>'second',
59             ),
60         );
61     }
62 }

```

复制代码

上面的代码指定了两个 **filter**: **postOnly** 和 **PerformanceFilter**. **postOnly filter** 是基于方法的 (对应的 **filter** 方法已被定义在 **CController** 中); 而 **PerformanceFilter filter** 是基于对象的 (object-based). 路径别名 `application.filters.PerformanceFilter` 指定 **filter** 类文件是 `protected/filters/PerformanceFilter`. 我们使用一个数组来配置 **PerformanceFilter** 以便它可被用来初始化此 **filter** 对象的属性值. 在这里 **PerformanceFilter** 的 **unit** 属性被将初始化为 `'second'`.

使用 **+** 和 **-** 操作符, 我可以指定哪个 **action** 此 **filter** 应当和不应被应用. 在上面的例子中, **postOnly** 被应用到 **edit** 和 **create** **action**, 而 **PerformanceFilter** 被应用到所有的 **actions** 除了 **edit** 和 **create**. 若 **+** 或 **-** 均未出现在 **filter** 配置中, 此 **filter** 将被用到所有 **action**.

开源 PHP 开发框架 Yii 全方位教程 (3) 模型 CModel

模型是 CModel 或其子类的实例。模型用于保持数据以及和数据相关的业务规则。

模型描述了一个单独的数据对象。它可以是数据表中的一行数据或者用户输入的一个表单。数据中的各个字段都描述了模型的一个属性。这些属性都有一个标签，都可以被一套可靠的规则验证。

Yii 实现了表单模型和 active record 两种模型，它们都继承自基类 CModel。

表单模型是 CFormModel 的实例。表单模型用于保存通过收集用户输入得来的数据。这样的数据通常被收集，使用，然后被抛弃。例如，在一个登录页面上，我们可以使用一个表单模型来描述诸如用户名，密码这样的由最终用户提供的信息。若想了解更多，请参阅 [Working with Form](#)。

Active Record (AR) 是一种面向对象风格的，用于抽象数据库访问的设计模式。任何一个 AR 对象都是 CActiveRecord 或其子类的实例，它描述的数据表中的单独一行数据。这行数据中的字段被描述成 AR 对象的一个属性。关于 AR 的更多信息可以在 [Active Record](#) 中找到。

开源 PHP 开发框架 Yii 全方位教程 (4) 视图 View

1 视图是一个包含了主要的用户交互元素的 PHP 脚本。他可以包含 PHP 语句，但是我们建议这些语句不要去改变数据模型，且最好能够保持其单纯性(单纯作为视图)!为了实现逻辑和界面分离，大部分的逻辑应该被放置于控制器或模型里，而不是视图里。

一个 view 有一个当渲染(render)时用来识别 view 脚本的名字。view 名字和它的 view 脚本文件的名字相同。例如:视图 edit 的名称出自一个名为 edit.php 的脚本文件。通过 CController::render() 调用视图的名称可以渲染一个视图。这个方法将在 protected/views/ControllerID 目录下寻找对应的视图文件。

在视图脚本内部，我们可以通过 \$this 来访问控制器实例。我们可以在视图里以 \$this->propertyName 的方式获取 (pull) 控制器的任何属性。

我们也可以使用以下 push 的方式传递数据到视图里:\$this->render('edit', array(

```
2     'var1'=>$value1,  
3     'var2'=>$value2,  
4 ));
```

复制代码

5 在以上的方式中，render() 方法将提取数组的第二个参数到变量里。其结果是，在视图脚本里，我们可以直接访问变量 \$var1 和 \$var2。

布局

布局是一种特殊的视图文件，用来修饰视图。它通常包含了用户交互过程中常用到的一部分视图。例如:视图可以包含 header 和 footer 的部分，然后把内容嵌入其间。...

```
6 <?php echo $content; ?>  
7 ...
```

复制代码

8 而 \$content 则储存了内容视图的渲染结果。

当使用 render() 时，布局被隐含的应用。视图脚本 protected/views/layouts/main.php 是默认的布局文件。它可以通过改变 CWebApplication::layout 或者 CController::layout 来实现定制。要渲染(render)一个 view 而不应用任何布局，换用 renderPartial()。

部件

部件是 `CWidget` 或其子类的实例。它是一个主要用于呈现目的的组件。部件通常内嵌于一个视图来产生一些复杂却独立的用户界面。例如，一个日历部件可以用于渲染一个复杂的日历界面。部件可以在用户界面上更好的实现重用。

要使用一个部件在一个 `view` 脚本中这样做：`<?php $this->beginWidget('path.to.WidgetClass'); ?>`

```
9 ...body content that may be captured by the widget...
```

```
10 <?php $this->endWidget(); ?>
```

复制代码

```
11 或者<?php $this->widget('path.to.WidgetClass'); ?>
```

复制代码

12 后者用于不需要任何 `body` 内容的 `widget`。

`widget` 可以通过配置来定制它的行为。这些是通过调用 `CBaseController::beginWidget` 或者 `CBaseController::widget` 设置它们的初始化属性值来完成的。例如，当使用 `CMaskedTextField` 组件时，我们想指定被使用的 `mask`。我们通过传递一个携带这些属性初始化值的数组来实现。这里的数组的键是属性的名称，而数组的值则是组件属性所对应的值。正如以下所示：`<?php`

```
13 $this->widget('CMaskedTextField',array(
```

```
14     'mask'=>'99/99/9999'
```

```
15 ));
```

```
16 ?>
```

复制代码

17 继承部件以及重载它的 `init()` 和 `run()` 方法，可以定义一个新的组件：`class MyWidget extends CWidget`

```
18 {
```

```
19     public function init()
```

```
20     {
```

```
21         // this method is called by CController::beginWidget()
```

```
22     }
```

```
23     public function run()
```

```
24     {
```

```
25         // this method is called by CController::endWidget()
```

```
26     }
```

```
27 }
```

复制代码

部件可以像一个控制器一样拥有它自己的视图。默认的， `widget` 的视图文件位于包含了 `widget` 文件的 `views` 子目录之下。这些视图可以通过调用 `CWidget::render()` 渲染，这一点和控制器很相似。唯一不同的是， `widget` 的视图没有布局文件支持。同时， `view` 文件中的 `$this` 指的是 `widget` 实例而不是 `controller` 实例。

系统视图(System View)

系统视图的渲染通常用于展示 `Yii` 的错误和日志信息。例如，当用户请求来一个不存在的控制器或动作时， `Yii` 会抛出一个异常来解释这个错误。这时， `Yii` 就会使用一个特殊的系统视图来展示这个错误。

系统视图的命名遵从了一些规则。比如像 `errorXXX` 这样的名称就是用于渲染展示错误号 `XXX` 的 `CHttpException` 的视图。例如，如果 `CHttpException` 抛出来一个 `404` 错误，那么 `error404` 就会被展示出来。

在 `framework/views` 下， `Yii` 提供了一系列默认的系统视图。他们可以通过在 `protected/views/system` 下创建同名视图文件来实现定制。

开源 PHP 开发框架 Yii 全方位教程 (5) 组件 CComponent

1 Yii 应用构建于组件之上。组件是 CComponent 或其子类的实例。使用组件主要就是涉及访问其属性和处理/处理它的事件。基类 CComponent 指定了如何定义属性和事件。

组件属性

组件的属性就像对象的公开成员变量。我们可以读取或设置组件属性的值。例如:
`$width=$component->textWidth; // 获取 textWidth 属性`

2 `$component->enableCaching=true; // 设置 enableCaching 属性`

复制代码

3 要定义组件属性，我们可以简单的在组件类里声明一个公共成员变量。更灵活的方法就是，如下所示的，定义 `getter` 和 `setter` 方法:
`public function getTextWidth()`

```
4 {
5     return $this->_textWidth;
6 }
7 public function setTextWidth($value)
8 {
9     $this->_textWidth=$value;
10 }
```

复制代码

11 以上的代码定义了一个名称为 `textWidth`(大小写不敏感) 的可写属性。当读取此属性时，`getTextWidth()`被调用，然后它的返回值成为属性的值；同样的，当写入属性时，`setTextWidth()`被调用。如果 `setter` 方法没有定义，属性就是只读的，如果向其写入将抛出一个异常。使用 `getter` 和 `setter` 方法来定义属性有这样的好处:当属性被读取或者写入的时候，附加的逻辑(例如执行校验,唤起事件)可以被执行。

通过 `getter/setter` 方法来定义一个属性和通过定义类的一个成员变量定义一个属性，有一个微小的差异.前者大小写不敏感而后者大小写敏感。

组件事件

组件事件是一种特殊的属性，它可以将方法(称之为 事件句柄(event handlers))作为它的值。附加(分配)一个方法到一个事件将会引起方法在事件被触发时自动被调用。因此，一个组件的行为可能会被一种在组件开发过程中不可预见的方式修改。

组件事件以 `on` 开头的命名方式定义。和属性通过 `getter/setter` 方法来定义的命名方式一样，事件的名称是大小写不敏感的。以下代码定义了一个 `onClicked` 事件:

```
public function  
onClicked($event)
```

```
12 {  
13     $this->raiseEvent('onClicked', $event);  
14 }
```

复制代码

15 这里作为事件参数的 `$event` 是 `CEvent` 或其子类的实例。

我们可以附加一个方法到此 `event`，如下所示：`$component->onClicked=$callback;`

复制代码

16 这里的 `$callback` 指向了一个有效的 PHP 回调。它可以是一个全局函数也可以是类中的一个方法。如果是后者，它的提供方式必须是一个数组 `array($object,'methodName')`。

事件句柄(event handler)必须按照如下来签署：`function methodName($event)`

```
17 {  
18     .....  
19 }
```

复制代码

20 这里的 `$event` 是描述事件(源于 `raiseEvent()` 调用的)的参数。`$event` 参数是 `CEvent` 或其子类的实例。它至少包含了"是谁唤起这个事件"的信息。

从版本 1.0.10 开始，一个 `event handler` 也可以是一个 PHP 5.3以后支持的匿名函数。例如，`$component->onClicked=function($event) { }`

复制代码

21 如果我们现在调用了 `onClicked()`，`onClicked` 事件将被触发(inside `onClicked()`)，然后被绑定的事件句柄(event handler)将被自动调用。

一个事件可以绑定多个句柄。当事件被唤起时，句柄将会以他们被绑定到事件的先后顺序调用。如果句柄决定在调用期间防止其他句柄的调用，它可以设置 `$event->handled` 为 `true`。

组件行为

自 1.0.2 版起，部件开始支持 `mixin` 从而可以绑定一个或者多个行为。一个行为(behavior) 就是一个对象，其方法可以被它绑定的部件通过收集功能的方式来实现 '继承(inherited)'，而不是专有

化继承(即普通的类继承).简单的来说,就是一个部件可以以'多重继承'的方式实现多个行为的绑定.

行为类必须实现 **IBehavior** 接口.大多数行为可以从 **CBehavior** 基类扩展而来.如果一个行为需要绑定到一个模型,它也可以从专为模型实现绑定特性的 **CModelBehavior** 或者 **CActiveRecordBehavior** 继承.

使用一个行为,必须首先通过调用行为的 **attach()** 方法绑定到一个组件.然后我们就可以通过组件调用行为了:// **\$name** 是行为在部件中唯一的身份标识.

```
22 $behavior->attach($name,$component);
23 // test() 是一个方法或者行为
24 $component->test();
```

复制代码

25 一个已绑定的行为是可以被当作组件的一个属性一样来访问的.例如,如果一个名为 **tree** 的行为被绑定到组件,我们可以获得行为对象的引用:**\$behavior=\$component->tree;**

```
26 // 相当于以下:
27 // $behavior=$component->asa('tree');
```

复制代码

28 行为是可以被临时禁止的,此时它的方法开就在组件中失效.例如:**\$component->disableBehavior(\$name);**

```
29 // 以下语句将抛出一个异常
30 $component->test();
31 $component->enableBehavior($name);
32 // 当前可用
33 $component->test();
```

复制代码

两个同名行为绑定到同一个组件下是很有可能.在这种情况下,先绑定的行为则拥有优先权.

当和 **events** 一起使用时,行为会更加强大.当行为被绑定到组件时,行为里的一些方法就可以绑定到组件的一些事件上了.这样一来,行为就有机观察或者改变组件的常规执行流程.

自版本 **1.1.0** 开始,一个行为的属性也可以通过绑定到的组件来访问.这些属性包含公共成员变量以及通过 **getters** 和/或 **setters** 方式设置的属性.例如,若一个行为有一个 **xyz** 的属性,此行为被绑定到组件 **\$a**,然后我们可以使用表达式**\$a->xyz** 访问此行为的属性.

开源 PHP 开发框架 Yii 全方位教程 (6) 模块

1 一个模块是一个自我包含的软件单元，它由模型，视图，控制器和另外组件组成。在很多方面，一个模块类似于一个应用。主要的不同是一个模块不能单独部署，它必须位于一个应用的内部。用户可以访问一个模块中的控制器，就像访问一个普通的应用的控制器。

模块在一些情况下是有用的。对于一个大型应用，我们可以将它分离为几个模块。每个被单独的开发和维护。一些常用的特征，例如用户管理，评论管理，可以以模块的方式开发以便它们在未来的项目容易的重用。

创建模块

一个模块被组织为一个和它 ID 名字相同的目录。模块目录的结构类似于应用的目录。下面展示一个名为 `forum` 的典型目录结构：`forum/`

2	<code>ForumModule.php</code>	the module class file
3	<code>components/</code>	containing reusable user components
4	<code>views/</code>	containing view files for widgets
5	<code>controllers/</code>	containing controller class files
6	<code>DefaultController.php</code>	the default controller class file
7	<code>extensions/</code>	containing third-party extensions
8	<code>models/</code>	containing model class files
9	<code>views/</code>	containing controller view and layout files
10	<code>layouts/</code>	containing layout view files
11	<code>default/</code>	containing view files for DefaultController
12	<code>index.php</code>	the index view file

[复制代码](#)

13 一个模块必须有一个扩展自 `CWebModule` 的模块类。类的名字是表达式 `ucfirst($id).'Module'`，`$id` 是模块 ID(或模块目录名)。模块类是在模块代码中存储信息以及分享的核心。例如，我们可以使用 `CWebModule::params` 来存储模块参数，使用 `CWebModule::components` 在模块级分享应用组件。

我们可以使用 `yiic` 工具来创建一个新模块的框架。例如，要创建上面的 `forum` 模块，我们可以在命令行窗口中执行下面的命令：

```
% cd WebRoot/testdrive
% protected/yiic shell
Yii Interactive Tool v1.0
Please type 'help' for help. Type 'exit' to quit.
```



```
>> module forum
```

使用模块

要使用一个模块，首先放置模块目录到应用基本目录下。然后在应用的 `modules` 属性中声明模块 ID。例如，为了使用上面的 `forum` 模块，我们可使用下面的应用配置：`return array(`

```
14     .....
15     'modules'=>array('forum',...),
16     .....
17 );
```

复制代码

18 一个模块也可以使用初始值来配置。其用法非常类似于配置应用组件。例如，模块 `forum` 在它的模块类中可以有一个属性 `postPerPage`，在应用配置中可以如下配置：`return array(`

```
19     .....
20     'modules'=>array(
21         'forum'=>array(
22             'postPerPage'=>20,
23         ),
24     ),
25     .....
26 );
```

复制代码

27 模块实例可以通过当前活动控制器的 `module` 属性来访问。通过模块实例，我们可以访问在模块级分享的信息。例如，为了访问上面的 `postPerPage` 信息，我们可以使用下面的表达式：

```
$postPerPage=Yii::app()->controller->module->postPerPage;
28 // or the following if $this refers to the controller instance
29 // $postPerPage=$this->module->postPerPage;
```

复制代码

一个模块中的控制器动作可以使用路由 `moduleID/controllerID/actionID` 来访问。例如，假设上面的 `forum` 模块有一个名为 `PostController` 的控制器，我们可以使用路由 `forum/post/create` 来指向此控制器的 `create` 动作。相应的路由的 URL 是 <http://www.example.com/index.php?r=forum/post/create>。

若一个控制器是一个控制器的子目录，我们仍然可以使用上面的路由格式。例如，假设 `PostController` 位于 `forum/controllers/admin` 目录中，我们可以指向 `create` 动作使用 `forum/admin/post/create`。

嵌套模块

模块可以嵌套，一个模块可以包含另外的模块。我们称前者为 **parent module**（父模块）后者为 **child module**（子模块）。子模块必须放置在父模块的 **modules** 目录下。要访问一个子模块中的控制器动作，我们应当使用路由 **parentModuleID/childModuleID /controllerID/actionID**。

开源 PHP 开发框架 Yii 全方位教程 (7) 路径别名和命名空间

- Yii 广泛的使用了路径别名.路径别名是和目录或者文件相关联的.它是通过使用点号(".")语法指定的,类似于以下这种被广泛使用的命名空间的格式:

RootAlias.path.to.target

RootAlias 则是一些已经存在目录的别名.通过调用 `YiiBase::setPathOfAlias()` 我们可以定义新的路径别名.

为了方便起见,Yii 预定义了以下根目录别名:

system: 指向 Yii 框架目录;

- **zii:** 指向 zii library 目录;
- **application:** 指向应用程序 基本目录(base directory);
- **webroot:** 指向包含里 入口脚本 文件的目录. 此别名自 1.0.3 版起生效.
- **ext:** 指向包含所有第三方扩展的目录, 从版本 1.0.8 可用;

另外,若应用使用了 `module`,一个根别名也被定义为每个 `module ID` 并指向相应 `module` 的 `base path`. 此特征从版本 1.0.3 可用.

通过使用 `YiiBase::getPathOfAlias()`, 一个别名可以被转换成它的对应的路径. 例如, `system.web.CController` 可以被转换成 `yii/framework/web/CController`.

使用别名,来导入已定义的类是非常方便的.例如,如果我们想要包含 `CController` 类的定义,我们可以通过以下方式调用:

```
1  Yii::import('system.web.CController');
```

复制代码

`import` 方法不同于 `include` 和 `require`,它是更加高效的.实际上被导入(`import`) 的类定义直到它第一次被调用之前都是不会被包含的. 同样的,多次导入同一个命名空间要比 `include_once` 和 `require_once` 快很多.

当调用一个通过 Yii 框架定义的类时,我们不必导入或者加载它.所有的 Yii 核心类都是被预加载的.

我们也可以按照以下的语法导入整个目录,以便目录下所有的类文件都可以在需要时被包含.

```
2  Yii::import('system.web.*');
```

复制代码

除了 `import` 外, 别名同样被用在其他很多地方来调用类.例如, 别名可以被传递到 `Yii::createComponent()` 以创建对应类的一个实例, 即使这个类文件没有被预先包含.

不要把别名和命名空间混淆了.命名空间调用了一些类名的逻辑分组以便他们可以同其他类名区分开,即使他们的名称是一样的,而别名则是用来引用类文件或者目录的.所以路径别名和命名空间并不冲突.

因为 PHP 5.3.0 以前的版本并不内置支持命名空间,所以你并不能创建两个有着同样名称但是不同定义的类的实例.为此,所有 Yii 框架类都以字母 'C'(代表 'class') 为前缀,以便避免与用户自定义类产生冲突.在这里我们推荐为 Yii 框架保留 'C' 字母前缀的唯一使用权,用户自定义类则可以使用其他字母作为前缀.

开源 PHP 开发框架 Yii 全方位教程 (8) 惯例

Yii favors conventions over configurations。遵循约定您可以不需编写和管理复杂的配置，就可以创建复杂的 Yii 应用。当然，当需要时 Yii 可对几乎所有方面进行定制配置。

下面我们描述 Yii 开发推荐的约定。为了方便起见，我们假设 WebRoot 是 Yii 应用安装目录。

网址 (URL)

URL 默认地，Yii 识别以下格式 URL：

```
http://hostname/index.php?r=ControllerID/ActionID
```

Get 变量 r 被 Yii 路由解释为控制器与动作。如果省略 ActionId，控制器会使用默认动作。（通过 CController::defaultAction 定义）；如果 ControllerId 也省略（或 r 变量没有值）应用程序会使用默认控制器（通过 CWebApplication::defaultController 定义）。

在 CUrlManager 的帮助下，有可能生成和识别许多对搜索引擎优化友好的 URL，如 http://hostname/ControllerID/ActionID.html。此功能详细情况在 URL Management。

代码 (Code)

Yii 建议变量，函数和类类型使用骆驼方式命名，就是大写名字中的每个单词并不用空格连接起来。变量和函数名首字母小写，为了区分于类名称（如：\$basePath，runController（），LinkPager）。对于私有的类成员变量，建议将他们的名字前缀加下划线字符（例如：\$_actionList）。

因为在 PHP5.3.0 之前不支持命名空间，建议以一些独特的方式命名这些类，以避免和第三方类名称冲突。出于这个原因，所有 Yii 框架类以字母"C"开头。

控制器类名的特别规则是，他们必须附上 Controller 后缀。类名的首字母小写，然后切掉结尾的 Controller 便是控制器的 ID。例如，PageController 类将有 ID page。这条规则使得应用更加安全。这也使得 controller 相关的 URL 更加简洁（例如 /index.php?r=page/index 替代 /index.php?r=PageController/index）。

配置 (Configuration)

配置是 键-值 对组成的数组。每个键代表要被配置的对象的一个属性，每个值是相应属性的初始值。例如，array('name'=>'My application', 'basePath'=>'./protected')初始 name 和 basePath 属性为其相应的数组值。

一个对象任何可写的属性均可以配置。如果未被配置，属性将使用它们的默认值。当设定属性，应该阅读相应的文档，以便使初始值设定正确。

文件 (File)

文件命名和使用的约定取决于其类型。

类文件应命名应使用包含的公共类名字。例如，`CController` 类是在 `CController.php` 文件中。公共类是一个可用于任何其他类的类。每个类文件应包含最多一个公共类。私有类（只被单独一个公共类使用）可以和该公共类存放在同一个文件里。

视图文件应使用视图名称命名。例如，`index` 视图在 `index.php` 文件里。视图文件是一个 PHP 脚本文件包含 HTML 和 PHP 代码，主要用来显示的。

配置文件可任意命名。配置文件是一个 PHP 脚本,其唯一目的就是要返回一个代表配置的关联数组。

目录 (Directory)

Yii 默认设定了被用于各种目的的一个目录集合.需要时，它们每个均可被自定义。

- `WebRoot/protected`: 这是 `application base directory` 包括所有安全敏感的 PHP 脚本和数据文件。Yii 有一个默认的别名为 `application` 代表此路径。这个目录和下面的一切文件目录，将得到保护不被网络用户访问。它可通过 `CWebApplication::basePath` 自定义。
- `WebRoot/protected/runtime`: 此目录拥有应用程序在运行时生成的私有临时文件。这个目录必须可被 Web 服务器进程写。它可通过 `CApplication::runtimePath` 定制。
- `WebRoot/protected/extensions`: 此目录拥有所有第三方扩展。它可通过 `CApplication::extensionPath` 定制。
- `WebRoot/protected/modules`: 此目录拥有所有应用 `modules`，每个代表作为一个子目录。
- `WebRoot/protected/controllers`: 此目录拥有所有控制器类文件。它可通过 `CWebApplication::controllerPath` 定制。
- `WebRoot/protected/views`:此目录包括所有的视图文件，包括控制视图，布局视图和系统视图。可通过 `CWebApplication::viewPath` 定制。
- `WebRoot/protected/views/ControllerID`: 此目录包括某个控制类的视图文件。这里 `ControllerID` 代表控制类的 ID。可通过 `CController::viewPath` 定制。
- `WebRoot/protected/views/layouts`: 此目录包括所有的布局视图文件。可通过 `CWebApplication::layoutPath` 来定制。
- `WebRoot/protected/views/system`: 此目录包括所有的系统视图文件。系统视图文件是显示错误和异常的模板。可通过 `CWebApplication::systemViewPath` 定制。
- `WebRoot/assets`: 此目录包括发布的 `asset` 文件。一个 `asset` 文件是一个私有文件，可被发布来被 Web 用户访问。此目录必须 Web 服务进程可写。可通过 `CAssetManager::basePath` 定制。
- `WebRoot/themes`: 此目录包括各种适用于应用程序的各种主题。每个子目录代表一个主题，名字为子目录名字。可通过 `CThemeManager::basePath` 定制。

数据库 (Database)

大多数 Web application 由数据库支撑的. For best practice, 我们建议数据库的表和列遵循如下命名约定. 注意它们不是 Yii 必需的.

- 数据库的表和列均以小写格式命名.
- 名字中的单词应以下划线分隔(例如 `product_order`).
- 对于表的名字，可以使用单数或复数名字，但不要两者均采用。简化起见，我们推荐使用单数名字.
- 表名字可以使用前缀,如 `tbl_`. 当一个 `application` 的表和其他 `application` 的表共存在同一个数据库时非常有用. 使用不同的表前缀可以容易地将它们分开.

开源 PHP 开发框架 Yii 全方位教程 (9) 开发流程

- 已经描述了 Yii 的基本概念，现在我们看看用 Yii 开发一个 web 程序的基本流程。前提是我这个程序我们已经做了需求分析和必要的设计分析。
创建目录结构。参看 [建立第一个 Yii 应用](#) 写的 yiic 工具可以帮助我们快速完成这一步。
- 配置应用。就是修改应用的配置文件。这步有可能会写一些 application 组件(例如：用户组件)
- 每种类型的数据都创建一个 model 类来管理。同样，yiic 可以为我们需要数据库表自动生成 active record 类。
- 每种类型的用户请求都创建一个 controller 类。依据实际的需求对用户请求进行分类。一般来说，如果一个 model 类需要用户访问，就应该对应一个 controller 类。yiic 工具也能自动完成这一步。
- 实现 actions 和相应的 views。这是真正需要我们编写的工作。
- 在 controller 类里配置需要的 action filters 。
- 如果需要主题功能，编写 themes。
- 如果需要 internationalization 国际化功能，编写翻译语句。
- 使用 caching 技术缓存数据和页面。
- 最后 tune up 调整程序和发布。

以上每个步骤，有可能需要编写测试案例来测试。

Data Access Objects (DAO) 提供了一个通用的 API 以访问存储在不同 DBMS 中的数据。这样，数据库改变时可以无需修改访问数据库的代码。

Yii DAO 建立于 PHP Data Objects (PDO) ，它是一个为很多 DBMS 提供统一数据访问的扩展，支持 MySQL, PostgreSQL 等。因此，要使用 Yii DAO, PDO 扩展和指定的 PDO 数据库驱动 (例如 PDO_MYSQL) 需要被安装。

Yii DAO 主要由下面四个类组成：

- CDbConnection: 代表一个数据库连接。
- CDbCommand: 代表一个执行到数据库的 SQL 语句。
- CDbDataReader: represents a forward-only stream of rows from a query result set.
- CDbTransaction: represents a DB 事务处理。

1

下面我们介绍在不同场景中 Yii DAO 的用法。

建立数据库连接

要建立一个数据库连接，需要创建一个 CDbConnection 实例并激活它。一个数据源名字(DSN) 被用来指定数据库连接信息。可能也会需要用户名和密码来建立连接。若在连接数据库时出现错误，将会触发一个异常(例如，错误的 DSN 或无效的用户名/密码)。\$connection=new CDbConnection(\$dsn,\$username,\$password);

```
2 // establish connection. You may try...catch possible exceptions
3 $connection->active=true;
4 .....
5 $connection->active=false; // close connection
```

复制代码

DSN 的格式取决于使用的 PDO 数据库驱动。通常一个 DSN 由 PDO 驱动名字，跟上一个冒号，以及驱动专有的连接句法组成。PDO documentation 可以查看到完整信息。下面是一个常用的 DSN 格式列表：

- SQLite: sqlite:/path/to/dbfile
- MySQL: mysql:host=localhost;dbname=testdb
- PostgreSQL: pgsql:host=localhost;port=5432;dbname=testdb
- SQL Server: mssql:host=localhost;dbname=testdb
- Oracle: oci:dbname=//localhost:1521/testdb

6

因为 CDbConnection 扩展自 CApplicationComponent，我们也可以使用它作为一个应用组件。我们可以在应用配置中如下配置 db (或其他名字) 应用组件，来实现此目的 array(

```
7 .....
8 'components'=>array(
9 .....
10 'db'=>array(
11     'class'=>'CDbConnection',
12     'connectionString'=>'mysql:host=localhost;dbname=testdb',
```



```
13         'username'=>'root',
14         'password'=>'password',
15         'emulatePrepare'=>true, // needed by some MySQL installations
16     ),
17 ),
18 )
```

复制代码

19 除非我们明确配置 `CDbConnection::autoConnect` 为 `false`, 否则我们就可以通过已自动被激活的 `Yii::app()->db` 来访问此 DB 连接, 通过这个方法, 这个单一的 DB 连接可以在代码中的多处位置共享.

执行 SQL 语句

一旦一个数据库连接建立, 就可以使用 `CDbCommand` 来执行 SQL 语句. 可以通过调用 `CDbConnection::createCommand()` 来创建一个 `CDbCommand` 实例, 参数是一个 SQL 语句: `$command=$connection->createCommand($sql);`

```
20 // 若需要, SQL 语句可以被如下更新:
```

```
21 // $command->text=$newSQL;
```

复制代码

一个 SQL 语句被执行通过 `CDbCommand` 以下面两种方式:

- `execute()`: 执行一个非查询的 SQL 语句, 例如 `INSERT`, `UPDATE` 和 `DELETE`. 若成功执行, 返回影响的记录数目.
- `query()`: 执行一条返回数据记录的 SQL 语句, 例如 `SELECT`. 若成功, 返回一个 `CDbDataReader` 实例. 方便起见, 一些 `queryXXX()` 方法也可以执行直接以返回查询结果.

```
22
```

若在 SQL 语句查询过程中出现错误, 会触发一个异常. `$rowCount=$command->execute();` // execute the non-query SQL

```
23 $dataReader=$command->query(); // execute a query SQL
```

```
24 $rows=$command->queryAll(); // query and return all rows of result
```

```
25 $row=$command->queryRow(); // query and return the first row of result
```

```
26 $column=$command->queryColumn(); // query and return the first column of result
```

```
27 $value=$command->queryScalar(); // query and return the first field in the first row
```

复制代码

28 读取查询结果

在 `CDbCommand::query()` 产生 `CDbDataReader` 实例后, 可以通过反复调用 `CDbDataReader::read()` 来取得结果集的记录. 也可以在 PHP 的 `foreach` 语言结构中使用

```

CDbDataReader 以逐行检索记录.$dataReader=$command->query();
29 // calling read() repeatedly until it returns false
30 while(($row=$dataReader->read())!==false) { ... }
31 // using foreach to traverse through every row of data
32 foreach($dataReader as $row) { ... }
33 // retrieving all rows at once in a single array
34 $rows=$dataReader->readAll();

```

复制代码

不同于 `query()`, 所有 `queryXXX()` 方法直接返回数据. 例如, `queryRow()` 返回一个数组, 它代表着查询结果中的第一行记录.

使用事务处理

当在一个应用执行一些查询时, 每次读取 和/或 写入数据库中的信息, 确保数据库不是只执行了一部分查询, 这一点非常重要. 一个事务处理, 在 Yii 的代表是一个 `CDbTransaction` 实例, **may be initiated in this case:**

- 开始事务处理.
- 逐个执行查询. 任何对数据库的更新对于外部都是不可见的.
- 提交(**Commit**)事务. 若事务成功执行, 对数据库的更改变得可见.
- 若其中一个查询失败, 整个事务被回滚(**rolle back**).

35

上面的流程可以使用下面的代码来执行: `$transaction=$connection->beginTransaction();`

```

36 try
37 {
38     $connection->createCommand($sql1)->execute();
39     $connection->createCommand($sql2)->execute();
40     //... other SQL executions
41     $transaction->commit();
42 }
43 catch(Exception $e) // an exception is raised if a query fails
44 {
45     $transaction->rollBack();
46 }

```

复制代码

47 绑定参数

为了避免 SQL 注入攻击 和改善执行反复的 SQL 语句的性能, 你可以"prepare" 一个 SQL 语句, 其中的可选参数占位符在参数绑定过程中被实际的数据代替.

参数占位符可以是 命名的 (represented as unique tokens) 或 未命名的 (represented as

question marks). 调用 `CDbCommand::bindParam()` 或 `CDbCommand::bindValue()` 以替换这些占位符为实际的参数. 参数无需以引号环绕: 底层数据库驱动为你完成. 参数绑定必须在 SQL 语句被执行前完成.// an SQL with two placeholders ":username" and ":email"

```
48 $sql="INSERT INTO tbl_user (username, email) VALUES(:username,:email)";
49 $command=$connection->createCommand($sql);
50 // replace the placeholder ":username" with the actual username value
51 $command->bindParam(":username",$username,PDO::PARAM_STR);
52 // replace the placeholder ":email" with the actual email value
53 $command->bindParam(":email",$email,PDO::PARAM_STR);
54 $command->execute();
55 // insert another row with a new set of parameters
56 $command->bindParam(":username",$username2,PDO::PARAM_STR);
57 $command->bindParam(":email",$email2,PDO::PARAM_STR);
58 $command->execute();
```

复制代码

59 方法 `bindParam()` 和 `bindValue()` 非常类似. 唯一不同点是前者以一个 PHP 变量引用 (reference)绑定一个参数, 而后者以一个值绑定一个参数. 对于大量参数(For parameters that represent large block of data memory),为了性能考虑应当使用前者.

关于绑定参数的更详细信息, 查看 [相关 PHP 文档](#).

绑定字段(Binding Columns)

当取出查询结果时, 你也可以绑定字段为 PHP 变量以便它们被每次取出的相应值自动填充.\$sql="SELECT username, email FROM tbl_user";

```
60 $dataReader=$connection->createCommand($sql)->query();
61 // bind the 1st column (username) with the $username variable
62 $dataReader->bindColumn(1,$username);
63 // bind the 2nd column (email) with the $email variable
64 $dataReader->bindColumn(2,$email);
65 while($dataReader->read()!==false)
66 {
67     // $username and $email contain the username and email in the current row
68 }
```

复制代码

69 使用表前缀

从版本 1.1.0 开始, Yii 为使用数据表前缀提供了完整的支持. 表前缀是一个字符串,放置在数据

表名字的前面.主要用于共享主机环境,多个应用分享一个数据库,使用不同的表前缀以相互区分.例如,一个可以使用 `tbl_` 作为表前缀而另一个使用 `yii_`.

要使用表前缀,配置 `CDbConnection::tablePrefix` 属性为你的表前缀.然后,在 SQL 语句中使用 `{{TableName}}` 指向表的名字, `TableName` 指的是不加前缀的表名字.例如,若数据库中有一个名为 `tbl_user` 的表,同时 `tbl_` 被配置为表前缀,然后我们可以使用下面的代码查询用户:
`$sql='SELECT * FROM {{user}}';`

```
70 $users=$connection->createCommand($sql)->queryAll();
```

[复制代码](#)

开源 PHP 开发框架 Yii 全方位教程 (11) Active Record (AR)

1 虽然 Yii DAO 可以处理事实上任何数据库相关的任务，但编写一些通用的 SQL 语句来执行 CRUD 操作（创建，读取，更新和删除）往往会让我们花费掉90%的时间。同时我们也很难维护这些 PHP 和 SQL 语句混合的代码。要解决这些问题，我们可以使用 Active Record。

Active Record (AR) 是一种流行的对象关系映射 (ORM) 技术。每个 AR 类代表一个数据表（或视图），数据表或者视图的字段作为 AR 类的属性，一个 AR 实例代表在表中的一行。常见的 CRUD 操作被作为 AR 类的方法执行。于是，我们可以使用更面向对象的方法处理我们的数据。例如，我们可以使用下面的代码在 `tbl_post` 表中插入一个新行：`$post=new Post;`

```
2 $post->title='sample post';
3 $post->content='post body content';
4 $post->save();
```

复制代码

5 在下面我们将介绍如何设置 AR 和用它来执行 CRUD 操作。在下一小节我们将展示如何使用 AR 处理数据库中的关联表。为了简单起见，在本节中，我们使用下面的数据库表作为例子。请注意，如果你使用 MySQL 数据库，您应该把下面 SQL 语句中的 AUTOINCREMENT 替换为 AUTO_INCREMENT。CREATE TABLE `tbl_post` (

```
6     id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
7     title VARCHAR(128) NOT NULL,
8     content TEXT NOT NULL,
9     create_time INTEGER NOT NULL
10 );
```

复制代码

11 AR 不是要解决所有与数据库相关的任务。它最好用于在 PHP 结构中模型化数据表和执行不复杂的 SQL 语句。而 Yii DAO 应该用于复杂的情况下。

建立数据库连接

AR 需要一个数据库连接以执行数据库相关的操作。默认情况下，应用中的 `db` 组件提供了 `CDbConnection` 实例作为我们需要的数据库连接。可参看如下配置：`return array(`

```
12     'components'=>array(
13         'db'=>array(
14             'class'=>'system.db.CDbConnection',
15             'connectionString'=>'sqlite:path/to/dbfile',
16             // turn on schema caching to improve performance
17             // 'schemaCachingDuration'=>3600,
```

```
18         ),
19     ),
20 );
```

复制代码

由于 **Active Record** 需要表的元数据来确定数据表的字段信息，这需要时间来读取和分析元数据。如果您的数据库结构是比较固定的，你应该打开缓存。打开方法是配置 `CDbConnection::schemaCachingDuration` 属性为一个大于 0 的值。

AR 的支持受限于数据库管理系统。目前，只有以下数据库管理系统支持：

- MySQL 4.1 或以后版本
- PostgreSQL 7.3 或以后版本
- SQLite 2 和 3
- Microsoft SQL Server 2000 或以后版本
- Oracle

21

Microsoft SQL Server 自 1.0.4 版本提供支持;而对 Oracle 自 1.0.5 版本即提供支持。

如果你想使用其他组件而不是 **db**，或者你使用 **AR** 访问多个数据库,你应该重写 `CActiveRecord::getDbConnection()`。 `CActiveRecord` 类是所有 **AR** 类的基类。

有两种方法可以在 **AR** 模式下使用多种数据库系统。如果数据库的模式不同，您可以对 `getDbConnection()` 进行不同的实现，来创建不同的 **AR** 基类。否则，动态改变静态变量 `CActiveRecord::db` 是一个更好的主意。

定义 **AR** 类

为了使用一个数据表，我们首先需要扩展 `CActiveRecord` 来定义一个 **AR** 类。每个 **AR** 类代表一个数据库表，每个 **AR** 实例代表数据表中的一行。下面的代码介绍了要创建一个对应 `tbl_post` 表的 **AR** 类所需要的最少的代码。`class Post extends CActiveRecord`

```
22 {
23     public static function model($className=<strong>CLASS</strong>)
24     {
25         return parent::model($className);
26     }
27     public function tableName()
28     {
29         return 'tbl_post';
30     }
31 }
```

复制代码

32 因为 AR 类在很多地方被引用,我们可以导入包含 AR 类的整个目录,而不是逐个引入它们.例如,若我们所有的 AR 类文件位于 `protected/models`,我们可以如下配置:`return array(`

```
33     'import'=>array(
34         'application.models.*',
35     ),
36 );
```

复制代码

37 默认的, AR 类的名字和数据表的名字相同. 若它们不同需要重写 `tableName()` 方法. 每个 AR 类的 `model()`方法都如此声明(会在稍后介绍).

要使用版本 1.1.0 引入的表前缀特征, AR 的方法 `tableName()` 可以被如下重写,`public function tableName()`

```
38 {
39     return '{{post}}';
40 }
```

复制代码

41 在这里,不再返回一个完整的表名,而是返回去掉了前缀的表名,并把它环绕在双弯曲括号中.

数据库表中一条数据的字段可以作为相应 AR 实例的属性被访问. 例如,下面的代码设置了 `title` 字段(属性):`$post=new Post;`

```
42 $post->title='a sample post';
```

复制代码

43 虽然我们没有在 `Post` 类中明确声明 `title` 属性,我们仍然可以在上面的代码中访问它. 这是因为 `title` 是表 `tbl_post` 中的字段,在 `PHP __get()` 魔术方法的帮助下,`CActiveRecord` 可以将其作为一个属性来访问. 若以同样方式尝试访问不存在的字段,一个异常将被抛出.

在此指南中,我们为所有的数据表和字段采取小写格式.这是因为在不同的 DBMS 中,对于大小写的敏感是不同的.例如,`PostgreSQL` 默认对字段名字是大小写不敏感的,如果一个列名是大小写混合的,在查询条件中我们就必须把它引用起来.使用小写格式可以避免此问题.

AR 依赖于数据表良好定义的主键. 若一个表没有一个主键,需要相应的 AR 类通过重写 `primaryKey()` 方法来指定哪些字段应当为主键,`public function primaryKey()`

```
44 {
45     return 'id';
46     // For composite primary key, return an array like the following
47     // return array('pk1', 'pk2');
```

复制代码

49 创建记录

要插入新的一行记录到数据表中，我们创建一个新的对应的 **AR** 类实例，设置和字段对应的属性的值，并调用 **save()** 方法来完成插入。`$post=new Post;`

```
50 $post->title='sample post';
51 $post->content='content for the sample post';
52 $post->create time=time();
53 $post->save();
```

复制代码

54 若表的主键是自增的，在插入后 **AR** 实例将包含一个更新后的主键。在上面的例子中，属性 **id** 将映射为新插入的主键值，即使我们没有明确更改它。

若在表模式中，一个字段被定义为一些静态默认值 (**static default value**) (例如一个字符串，一个数字)，在这个 **AR** 实例被创建后，实例中相应的属性将自动有相应的默认值。改变此默认值的一个方式是在 **AR** 类中明确声明此属性：`class Post extends CActiveRecord`

```
55 {
56     public $title='please enter a title';
57     .....
58 }
59 $post=new Post;
60 echo $post->title; // this would display: please enter a title
```

复制代码

61 从版本 **1.0.2** 开始，在记录被保存前(插入或更新)一个属性可以赋值为 **CDbExpression** 类型的值。例如，为了保存由 **MySQL NOW()** 函数返回的时间戳，我们可以使用下面的代码：`$post=new Post;`

```
62 $post->create time=new CDbExpression('NOW()');
63 // $post->create time='NOW()'; will not work because
64 // 'NOW()' will be treated as a string
65 $post->save();
```

复制代码

66 **AR** 允许我们执行数据库操作而无需编写麻烦的 **SQL** 语句，我们常常想要知道什么 **SQL** 语句被 **AR** 在下面执行了。这可以通过打开 **Yii** 的记录(logging)特征来实现。例如，我们可以在应用配置中打开 **CWebLogRoute**，我们将看到被执行的 **SQL** 语句被显示在每个页面的底

部. 自版本 1.0.5 开始, 我们可以在应用配置设置 `CDbConnection::enableParamLogging` 为 `true` 以便绑定到 SQL 语句的参数值也被记录.

读取记录

要读取数据表中的数据, 我们可以调用下面其中一个 `find` 方法: `// find the first row satisfying the specified condition`

```
67 $post=Post::model()->find($condition,$params);
68 // find the row with the specified primary key
69 $post=Post::model()->findByPk($postID,$condition,$params);
70 // find the row with the specified attribute values
71 $post=Post::model()->findByAttributes($attributes,$condition,$params);
72 // find the first row using the specified SQL statement
73 $post=Post::model()->findBySql($sql,$params);
```

复制代码

74 在上面, 我们使用 `Post::model()` 调用 `find` 方法. 记得静态方法 `model()` 是每个 AR 类所必需的. 此方法返回一个 AR 实例, 此实例被用来访问类级的方法(类似于静态的类方法).

若 `find` 方法找到一行记录满足查询条件, 它将返回一个 `Post` 实例, 此实例的属性包含表记录对应的字段值. 然后我们可以读取被载入的值如同我们访问普通对象的属性, 例如, `echo $post->title;`

`find` 方法将返回 `null` 若在数据库中没有找到满足条件的记录.

当调用 `find`, 我们使用 `$condition` 和 `$params` 来指定查询条件. 这里 `$condition` 可以是字符串代表一个 SQL 语句中的 `WHERE` 子语句, `$params` 是一个参数数组, 其中的值应被绑定到 `$condition` 的占位符. 例如, `// find the row with postID=10`

```
75 $post=Post::model()->find('postID=:postID', array(':postID'=>10));
```

复制代码

76 注意: 在上面的例子中, 对于某些 DBMS 我们可能需要转义对 `postID` 字段的引用. 例如, 若我们使用 PostgreSQL, 我们需要写 `condition` 为 `"postID"=:postID`, 因为 PostgreSQL 默认情况下对待字段名字为大小写不敏感的.

我们也可以使用 `$condition` 来指定更复杂的查询条件. 不使用字符串, 我们让 `$condition` 为一个 `CDbCriteria` 实例, 可以让我们指定条件而限于 `WHERE` 子语句. 例如, `$criteria=new CDbCriteria;`

```
77 $criteria->select='title'; // only select the 'title' column
78 $criteria->condition='postID=:postID';
```

```
79 $criteria->params=array(':postID'=>10);
80 $post=Post::model()->find($criteria); // $params is not needed
```

复制代码

81 注意, 当使用 **CDbCriteria** 作为查询条件, 不再需要参数 **\$params** 因为它可以在 **CDbCriteria** 中被指定, 如上所示.

一个可选的 **CDbCriteria** 方式是传递一个数组到 **find** 方法. 数组的键和值分别对应于 **criteria** 的属性名字和值. 上面的例子可以被如下重写, `$post=Post::model()->find(array(`

```
82     'select'=>'title',
83     'condition'=>'postID=:postID',
84     'params'=>array(':postID'=>10),
85 ));
```

复制代码

86 当一个查询条件是关于匹配一些字段用指定的值, 我们可以使用 **findByAttributes()**. 我们让参数 **\$attributes** 为一个数组, 数组的值由字段名字索引. 在一些框架中, 此任务可以通过调用类似于 **findByNameAndTitle** 的方法来实现. 虽然这个方法看起来很有吸引力, 但它常常引起混淆和冲突, 例如字段名字的大小写敏感性问题.

当多行记录满足指定的查询条件, 我们可以使用下面的 **findAll** 方法将它们聚合在一起, 每个都有它们自己的副本 **find** 方法. `// find all rows satisfying the specified condition`

```
87 $posts=Post::model()->findAll($condition,$params);
88 // find all rows with the specified primary keys
89 $posts=Post::model()->findAllByPk($postIDs,$condition,$params);
90 // find all rows with the specified attribute values
91 $posts=Post::model()->findAllByAttributes($attributes,$condition,$params);
92 // find all rows using the specified SQL statement
93 $posts=Post::model()->findAllBySql($sql,$params);
```

复制代码

94 若没有符合条件的记录, **findAll** 返回一个空数组. 不同于 **find** 方法, **find** 方法会返回 **null**.

除了上面所说的 **find** 和 **findAll** 方法, 为了方便, 下面的方法也可以使用: `// get the number of rows satisfying the specified condition`

```
95 $n=Post::model()->count($condition,$params);
96 // get the number of rows using the specified SQL statement
97 $n=Post::model()->countBySql($sql,$params);
98 // check if there is at least a row satisfying the specified condition
99 $exists=Post::model()->exists($condition,$params);
```

复制代码

100 更新记录

一个 AR 实例被字段值填充后，我们可以改变它们并保存回它们到数据表中。`$post=Post::model()->findByPk(10);`

```
101 $post->title='new post title';
```

```
102 $post->save(); // save the change to database
```

复制代码

103 如我们所见，我们使用相同的 `save()` 方法来执行插入和更新操作。若一个 AR 实例被使用 `new` 操作符创建，调用 `save()` 将插入一行新记录到数据表中；如果 AR 实例是一些 `find` 或 `findAll` 方法调用的结果，调用 `save()` 将更新表中已存在的记录。事实上，我们可以使用 `CActiveRecord::isNewRecord` 来检查一个 AR 实例是否是新建的。

更新一行或多行表中的记录而不预先载入它们也是可能的。AR 提供如下方便的类级的方法来实现它：`// update the rows matching the specified condition`

```
104 Post::model()->updateAll($attributes,$condition,$params);
```

```
105 // update the rows matching the specified condition and primary key(s)
```

```
106 Post::model()->updateByPk($pk,$attributes,$condition,$params);
```

```
107 // update counter columns in the rows satisfying the specified conditions
```

```
108 Post::model()->updateCounters($counters,$condition,$params);
```

复制代码

109 在上面，`$attributes` 是一个值由字段名索引的数组；`$counters` 是一个增加值由字段名索引的数组；`$condition` 和 `$params` 已在之前被描述。

删除记录

我们也可以删除一行记录若一个 AR 实例已被此行记录填充。`$post=Post::model()->findByPk(10);`
`// assuming there is a post whose ID is 10`

```
110 $post->delete(); // delete the row from the database table
```

复制代码

111 注意，在删除后，此 AR 实例仍然未改变，但相应的表记录已经不存在了。

下面类级别的(class-level)方法被用来删除记录，而无需预先载入它们：`// delete the rows matching the specified condition`

```
112 Post::model()->deleteAll($condition,$params);
```

```
113 // delete the rows matching the specified condition and primary key(s)
```

```
114 Post::model()->deleteByPk($pk,$condition,$params);
```

复制代码

115 数据验证

当插入或更新一行记录，我们常常需要检查字段的值是否符合指定的规则。若字段值来自用户时这一点特别重要。通常我们永远不要信任用户提交的数据。

AR 自动执行数据验证在 `save()` 被调用时。验证基于在 AR 类中的 `rules()`方法中指定的规则。如何指定验证规则的更多信息，参考 [声明验证规则](#) 部分。下面是保存一条记录典型的工作流程：`if($post->save())`

```
116 {  
117     // data is valid and is successfully inserted/updated  
118 }  
119 else  
120 {  
121     // data is invalid. call getErrors() to retrieve error messages  
122 }
```

复制代码

123 当插入或更新的数据被用户在 HTML 表单中提交，我们需要赋值它们到对象的 AR 属性。我们可以这样做：`$post->title=$_POST['title'];`

```
124 $post->content=$_POST['content'];
```

```
125 $post->save();
```

复制代码

126 若有很多字段，我们可以看到一个很长的赋值列表。可以使用下面的 `attributes` 属性来缓解。更多细节可以在安全属性设置部分和创建动作部分找到。`// assume $_POST['Post'] is an array of column values indexed by column names`

```
127 $post->attributes=$_POST['Post'];
```

```
128 $post->save();
```

复制代码

对比记录

类似于表记录，AR 实例由它们的主键值来被识别。因此，要对比两个 AR 实例，我们只需要对比它们的主键值，假设它们属于相同的 AR 类。然而，一个更简单的方式是调用 `CActiveRecord::equals()`。

提示: 不同于 AR 在其他框架的执行, Yii 在其 AR 中支持多个主键. 一个复合主键由两个或更多字段构成. 对应的, 主键值在 Yii 中表示为一个数组. The primaryKey 属性给出一个 AR 实例的主键值。

定制

CActiveRecord 提供了一些占位符(placeholder)方法可被用来在子类中重写以自定义它的工作流程.

- beforeValidate 和 afterValidate: 它们在验证执行 之前/后 被调用.
- beforeSave 和 afterSave: 它们在保存一个 AR 实例之前/后 被调用.
- beforeDelete 和 afterDelete: 它们在一个 AR 实例被删除 之前/后 被调用.
- afterConstruct: 这将在每个 AR 实例被使用 new 操作符创建之后被调用.
- beforeFind: 它在一个 AR finder 被用来执行一个查询之前被调用 (例如 find(), findAll()). 从版本 1.0.9 可用.
- afterFind: 它在每个 AR 实例被创建作为一个查询结果后被调用.

129

在 AR 中使用事务处理

每个 AR 实例包含一个名为 dbConnection 的属性,它是一个 CDbConnection 实例. 这样我们在使用 AR 时就可以使用 Yii DAO 提供的事务处理特征:\$model=Post::model();

```
130 $transaction=$model->dbConnection->beginTransaction();
131 try
132 {
133     // find and save are two steps which may be intervened by another request
134     // we therefore use a transaction to ensure consistency and integrity
135     $post=$model->findByPk(10);
136     $post->title='new post title';
137     $post->save();
138     $transaction->commit();
139 }
140 catch(Exception $e)
141 {
142     $transaction->rollBack();
143 }
```

复制代码

144 命名空间

命名空间是一个命名的查询约束,可以和其他的命名空间组合,并可以用于数据库查询 (a named scope represents a named query criteria that can be combined with other named scopes and applied to an active record query.)

命名空间被主要在 CActiveRecord::scopes() 方法中声明,格式是 name-criteria 对。下面的代

码在 `Post` 模型类中声明了两个命名空间，`published` 和 `recently`:`class Post extends ActiveRecord`

```
145 {
146     .....
147     public function scopes()
148     {
149         return array(
150             'published'=>array(
151                 'condition'=>'status=1',
152             ),
153             'recently'=>array(
154                 'order'=>'create time DESC',
155                 'limit'=>5,
156             ),
157         );
158     }
159 }
```

复制代码

160 每个命名空间被声明为一个被用来初始化一个 `CDbCriteria` 实例的数组。例如，命名空间 `recently` 指定了 `order` 属性为 `create_time DESC`，`limit` 属性为 `5`，转换成的查询条件就是应当返回最近发表的5篇帖子。

命名空间大多数作为 `find` 方法的 修改限制 (`modifier`) 来使用。几个命名空间可以连接在一起，则可以得到一个更加有限性的查询结果集。例如，要找到最近发表的帖子，我们可以使用下面的代码：`$posts=Post::model()->published()->recently()->findAll();`

复制代码

161 通常命名空间必须出现在一个 `find` 方法的左边。 其中的每一个提供了一个查询约束，和其他约束相结合, `including the one passed to the find method call`. 这种网络结构很像在查询上加了一些过滤.

从版本 `1.0.6` 开始，命名空间也可以使用 `update` 和 `delete` 方法。例如，下面的代码将删除所有最近发表的帖子：`Post::model()->published()->recently()->delete();`

复制代码

162 注意：命名空间只可以被作为类级别的方法使用。也就是说，此方法必须使用 `ClassName::model()` 来调用它。

参数化命名空间(Parameterized Named Scopes)

命名空间可以被参数化。例如，我们想要定制命名空间 **recently** 指定的帖子数目。要这样做，不是在 `CActiveRecord::scopes` 方法中声明命名空间，我们需要定义一个新的方法，它的名字和空间的名字相同：`public function recently($limit=5)`

```
163 {
164     $this->getDbCriteria()->mergeWith(array(
165         'order'=>'create time DESC',
166         'limit'=>$limit,
167     ));
168     return $this;
169 }
```

复制代码

170 然后，我们可以使用下面的语句来检索 3 个最近发表的帖子：<http://wenku.baidu.com/view/9748b29951e79b896802263b.html>

复制代码

171 若我们不使用上面的参数 3，默认情况下我们将检索 5 个最近发表的内容。

默认命名空间

一个模型类可以有一个默认命名空间，它被应用于此模型所有的查询（包括 **relational ones**）。例如，一个支持多种语言的网站只是以当前用户指定的语言来显示内容。因为有很多关于站点内容的查询，我们可以定义一个默认命名空间来解决这个问题。要这样做，我们重写 `CActiveRecord::defaultScope` 方法如下，`class Content extends CActiveRecord`

```
172 {
173     public function defaultScope()
174     {
175         return array(
176             'condition'=>"language='".Yii::app()->language."'";
177         );
178     }
179 }
```

复制代码

180 现在，若调用下面的方法将自动使用上面定义的查询条件：`$contents=Content::model()->findAll();`

复制代码

注意默认命名空间只应用于 `SELECT` 查询。它忽视 `INSERT`, `UPDATE` 和 `DELETE` 查询。

开源 PHP 开发框架 Yii 全方位教程 (12) 片段缓存

1 片段缓存指缓存网页某片段。例如，如果一个页面在表中显示每年的销售摘要，我们可以存储此表在缓存中，减少每次请求需要重新产生的时间。

要使用片段缓存，在控制器视图脚本中调用 `CController::beginCache()` 和 `CController::endCache()`。这两个方法标识了被缓存的页面内容的开始和结束。类似 `data caching`，我们需要一个编号，识别被缓存的片段。...别的 HTML 内容...

```
2 <?php if($this->beginCache($id)) { ?>
3 ...被缓存的内容...
4 <?php $this->endCache(); } ?>
5 ...别的 HTML 内容...
```

复制代码

6 在上面的，如果 `beginCache()` 返回 `false`，缓存的内容将此地方自动插入；否则，在 `if` 语句内的内容将被执行并在 `endCache()` 触发时缓存。

缓存选项(Caching Options)

当调用 `beginCache()`，可以提供一个由缓存选项组成的数组作为第二个参数，以自定义片段缓存。事实上为了方便，`beginCache()` 和 `endCache()` 方法是 `COutputCache widget` 的包装。因此 `COutputCache` 的所有属性都可以在缓存选项中初始化。

有效期 (Duration)

也许是最常见的选项是 `duration`，指定了内容在缓存中多久有效。和 `CCache::set()` 过期参数有点类似。下面的代码缓存内容片段最多一小时：...其他 HTML 内容...

```
7 <?php if($this->beginCache($id, array('duration'=>3600))) { ?>
8 ...被缓存的内容...
9 <?php $this->endCache(); } ?>
10 ...其他 HTML 内容...
```

复制代码

11 如果我们不设定期限，它将默认为60，这意味着60秒后缓存内容将无效。

依赖(Dependency)

像 `data caching`，被缓存的内容片段也可以有依赖。例如，文章的内容被显示取决于文章是否

被修改。

要指定一个依赖，我们设置了 `dependency` 选项，可以是一个实现 `ICacheDependency` 的对象或可用于生成依赖对象的配置数组。下面的代码指定片段内容取决于 `lastModified` 列的值是否变化：...其他 HTML 内容...

```
12 <?php if($this->beginCache($id, array('dependency'=>array(
13     'class'=>'system.caching.dependencies.CDbCacheDependency',
14     'sql'=>'SELECT MAX(lastModified) FROM Post')))) { ?>
15 ...被缓存的内容...
16 <?php $this->endCache(); } ?>
17 ...其他 HTML 内容...
```

复制代码

变化(Variation)

缓存的内容可根据一些参数变化。例如，每个人的档案都不一样。缓存的档案内容将根据每个人 ID 变化。这意味着，当调用 `beginCache()`时将用不同的 ID。

`COutputCache` 内置了这一特征，程序员不需要编写根据 ID 变动内容的模式。以下是摘要。

- `varyByRoute`: 设置此选项为 `true`，缓存的内容将根据 `route` 变化。因此，每个控制器和行动的组将有一个单独的缓存内容。
- `varyBySession`: 设置此选项为 `true`，缓存的内容将根据 `session ID` 变化。因此，每个用户会话可能会看到由缓存提供的不同内容。
- `varyByParam`: 设置此选项的数组里的名字，缓存的内容将根据 `GET` 参数的值变动。例如，如果一个页面显示文章的内容根据 `id` 的 `GET` 参数，我们可以指定 `varyByParam` 为 `array('id')`，以使我们能够缓存每篇文章内容。如果没有这样的变化，我们只能能够缓存某一文章。
- `varyByExpression`: 设置此选项为一个 `PHP` 表达式，我们可以让缓存的内容根据此 `PHP` 表达式的结果而变化。此选项自版本 `1.0.4` 可用。

18

请求类型(Request Types)

有时候，我们希望片段缓存只对某些类型的请求启用。例如，对于某张网页上显示表单，我们只想要缓存表单当其被初次访问时(通过 `GET` 请求)。任何随后显示(通过 `POST` 请求)的表单将不被缓存，因为表单可能包含用户输入。要做到这一点，我们可以指定 `requestTypes` 选项：...其他 HTML 内容...

```
19 <?php if($this->beginCache($id, array('requestTypes'=>array('GET')))) { ?>
20 ...被缓存的内容...
21 <?php $this->endCache(); } ?>
22 ...其他 HTML 内容...
```

复制代码

23 嵌套缓存(Nested Caching)

片段缓存可以嵌套。就是说一个缓存片段附在一个更大的片段缓存里。例如，评论缓存在内部片段缓存，而且它们一起在外部缓存中在文章内容里缓存。...其他 HTML 内容...

```
24 <?php if($this->beginCache($id1)) { ?>
25 ...外部被缓存内容...
26     <?php if($this->beginCache($id2)) { ?>
27     ...内部被缓存内容...
28     <?php $this->endCache(); } ?>
29 ...外部被缓存内容...
30 <?php $this->endCache(); } ?>
31 ...其他 HTML 内容...
```

[复制代码](#)

嵌套缓存可以设定不同的缓存选项。例如， 在上面的例子中内部缓存和外部缓存可以设置时间长短不同的持续值。当数据存储在外部缓存无效，内部缓存仍然可以提供有效的内部片段。 然而，反之就不行了。如果外部缓存包含有效的数据， 它会永远保持缓存副本，即使内部缓存中的内容已经过期。

开源 PHP 开发框架 Yii 全方位教程 (13) 页面缓存

页面缓存指的是缓存整个页面的内容。页面缓存可以发生在不同的地方。例如，通过选择适当的页面头，客户端的浏览器可能会缓存网页浏览有限时间。Web 应用程序本身也可以在缓存中存储网页内容。在本节中，我们侧重于后一种办法。

页面缓存可以被看作是片段缓存的一个特例。由于网页内容是往往通过应用一个布局到一个视图来生成，如果我们只是简单的在布局中调用 `beginCache()`和 `endCache()`，将无法正常工作。这是因为布局在 `CController::render()`方法里的加载是在页面内容产生之后。

缓存整个页面，我们应该跳过产生网页内容的动作执行。我们可以使用 `COutputCache` 作为动作过滤器来完成这一任务。下面的代码演示如何配置缓存过滤器：

```
1 public function filters()
2 {
3     return array(
4         array(
5             'COutputCache',
6             'duration'=>100,
7             'varyByParam'=>array('id'),
8         ),
9     );
10 }
```

[复制代码](#)

上述过滤器配置会使过滤器应用于控制器中的所有行动。通过使用 `+` 操作符，我们可能会限制它应用于一个或几个 `action`。更多的细节中可以看过滤器。

开源 PHP 开发框架 Yii 全方位教程 (14) 动态内容

1 当使用片段缓存或页面缓存时，我们常常遇到的这样的情况：整个部分的输出除了个别地方都是相对静态的。例如，帮助页可能会显示静态的帮助信息，而将当前登录的用户的名称显示在页面顶部。

解决这个问题，我们可以根据用户名变化缓存内容，但是这将是我们的宝贵缓存空间一个巨大的浪费，因为缓存除了用户名其他大部分内容是相同的。我们还可以把网页切成几个片段并分别缓存，但这种情况会使页面和代码变得非常复杂。更好的方法是使用由 `CController` 提供的动态内容(dynamic content)特征。

动态内容是指片段输出即使是在片段缓存包括的内容中也不会被缓存。即使是包括的内容是从缓存中取出，为了使动态内容在所有时间是动态的，每次都得重新生成。出于这个原因，我们要求动态内容通过一些方法或函数生成。

调用 `CController::renderDynamic()`在你想的地方插入动态内容。...别的 HTML 内容...

```
2 <?php if($this->beginCache($id)) { ?>
3 ...被缓存的片段内容...
4     <?php $this->renderDynamic($callback); ?>
5 ...被缓存的片段内容...
6 <?php $this->endCache(); } ?>
7 ...别的 HTML 内容...
```

复制代码

在上面的，`$callback` 指的是有效的 PHP 回调。它可以是指向当前控制器类的方法或者全局函数的字符串名。它也可以是一个数组名指向一个类的方法。任何传递到 `renderDynamic()`方法中的额外参数将被传递给回调(callback)。回调将返回动态内容而不是显示它。

开源 PHP 开发框架 Yii 全方位教程 (15) 使用扩展

1 适用扩展通常涉及以下三个步骤:

1. 从 Yii 的 扩展库 下载扩展.
2. 解压到 应用程序的基目录 的子目录 `extensions/xyz` 下,这里的 `xyz` 是扩展的名称.
3. 导入, 配置和使用扩展.

每个扩展都有一个唯一的标识的名字.把一个扩展命名为 `xyz`,我们可以使用路径别名 `ext.xyz` 定位到包含了 `xyz` 所有文件的基目录.

不同的扩展有着不同的导入,配置,使用要求.以下是我们通常会用到扩展的场景,按照他们在 `概述` 中的描述分类.

Zii 扩展

在开始介绍第三方扩展的用法之前,我们首先介绍 Zii 扩展库,它是由 Yii 开发团队开发的扩展集合,自 Yii 版本 1.1.0 开始包含在发布中。Zii 库在 Google 代码上的名字是 `zii`。

当使用一个 Zii 扩展时,必须以格式 `zii.path.to.ClassName` 指向对应的类。这里的根别名 `zii` 被预定义在 Yii 中。它指向 Zii 库的根目录。例如,要使用 `CGridView`,当引用此扩展时,我们应当在一个视图中使用如下代码:`$this->widget('zii.widgets.grid.CGridView', array(`

```
2     'dataProvider'=>$dataProvider,  
3 ));
```

复制代码

4 应用组件

要使用 应用组件,首先我们需要添加一个新条目到 应用配置 的 `components` 属性,如下所示:`return array(`

```
5     // 'preload'=>array('xyz',...),  
6     'components'=>array(  
7         'xyz'=>array(  
8             'class'=>'ext.xyz.XyzClass',  
9             'property1'=>'value1',  
10            'property2'=>'value2',  
11        ),  
12        // 其他部件配置  
13    ),
```

14);

复制代码

15 然后,我们可以在任何地方通过使用 `Yii::app()->xyz` 来访问组件.部件将会被 惰性创建(就是,仅当它第一次被访问时创建.),除非我们把它配置到 `preload` 属性里.

行为(Behavior)

Behavior 可以使用在各种组件中。它的用法涉及两个步骤。第一部,一个行为被附加到一个目标组件。第二步,通过目标组件调用行为方法。例如: `// $name uniquely identifies the behavior in the component`

```
16 $component->attachBehavior($name,$behavior);
17 // test() is a method of $behavior
18 $component->test();
```

复制代码

19 经常,一个行为被附加到一个组件,使用一个配置化的方式而不是调用 `attachBehavior` 方法。例如,要附加一个行为到一个应用组件,我们可以使用下面的应用配置: `return array(`

```
20     'components'=>array(
21         'db'=>array(
22             'class'=>'CDbConnection',
23             'behaviors'=>array(
24                 'xyz'=>array(
25                     'class'=>'ext.xyz.XyzBehavior',
26                     'property1'=>'value1',
27                     'property2'=>'value2',
28                 ),
29             ),
30         ),
31         //....
32     ),
33 );
```

复制代码

34 上面的代码附加 `xyz` 行为到 `db` 应用组件。我们可以这样做,因为 `CApplicationComponent` 定义了一个名为 `behaviors` 的属性。通过使用一个行为配置列表设置此属性,组件当它被初始化时将附加对应的行为。

对于 `CController`, `CFormModel` 和 `CActiveRecord` 这些经常被扩展的类,可以重写它们的 `behaviors()` 方法来附加行为。当这些类初始化时,自动将在此方法中声明的行为附加到类中。

例如, public function behaviors()

```
35 {
36     return array(
37         'xyz' => array(
38             'class' => 'ext.xyz.XyzBehavior',
39             'property1' => 'value1',
40             'property2' => 'value2',
41         ),
42     );
43 }
```

复制代码

44 部件

部件主要用在视图里.假设部件类 **XyzClass** 属于 **xyz** 扩展,我们可以如下在视图中使用它://

组件不需要主体内容

```
45 <?php $this->widget('ext.xyz.XyzClass', array(
46     'property1' => 'value1',
47     'property2' => 'value2')); ?>
48 // 组件可以包含主体内容
49 <?php $this->beginWidget('ext.xyz.XyzClass', array(
50     'property1' => 'value1',
51     'property2' => 'value2')); ?>
52 ...组件的主体内容...
53 <?php $this->endWidget(); ?>
```

复制代码

54 动作

动作 被 控制器 用于响应指定的用户请求.假设动作的类 **XyzClass** 属于 **xyz** 扩展,我们可以
在我们的控制器类里重写 **CController::actions** 方法来使用它 :class TestController extends

CController

```
55 {
56     public function actions()
57     {
58         return array(
59             'xyz' => array(
60                 'class' => 'ext.xyz.XyzClass',
61                 'property1' => 'value1',
62                 'property2' => 'value2',
```



```
63         ),
64         // 其他动作
65     );
66 }
67 }
```

复制代码

68 然后,我们可以通过 路由 `test/xyz` 来访问.

过滤器

过滤器 也被 控制器 使用。过滤器主要用于当其被 动作 操纵时预处理, 后期处理(pre- and post-process)用户的请求。假设过滤器的类 `XYZClass` 属于 `xyz` 扩展,我们可以在我们的控制器类里重写 `CController::filters` 方法来使用它:

```
class TestController extends CController
```

```
69 {
70     public function filters()
71     {
72         return array(
73             array(
74                 'ext.xyz.XyzClass',
75                 'property1'=>'value1',
76                 'property2'=>'value2',
77             ),
78             // 其他过滤器
79         );
80     }
81 }
```

复制代码

82 在上述代码中,我们可以在数组的第一个元素离使用 `+` 或者 `-` 操作符来限定过滤器只在那些动作中生效.更多信息,请参照文档的 `CController`.

控制器

控制器 提供了一套可以被用户请求的动作。为了使用一个控制器扩展,我们需要在 应用配置 里设置 `CWebApplication::controllerMap` 属性:

```
return array(
```

```
83     'controllerMap'=>array(
84         'xyz'=>array(
85             'class'=>'ext.xyz.XyzClass',
86             'property1'=>'value1',
```

```
87         'property2'=>'value2',
88     ),
89     // 其他控制器
90 ),
91 );
```

复制代码

92 然后, 一个在控制里的 **a** 行为就可以通过 路由 **xyz/a** 来访问了.

校验器

校验器主要用在 模型类 (继承自 `CFormModel` 或者 `CActiveRecord`) 中. 假设校验器类 `XYZClass` 属于 `xyz` 扩展, 我们可以在我们的模型类中通过 `CModel::rules` 重写 `CModel::rules` 来使用它: `class MyModel extends CActiveRecord // or CFormModel`

```
93 {
94     public function rules()
95     {
96         return array(
97             array(
98                 'attr1, attr2',
99                 'ext.xyz.XyzClass',
100                'property1'=>'value1',
101                'property2'=>'value2',
102            ),
103            // 其他校验规则
104        );
105    }
106 }
```

复制代码

107 控制台命令

控制台命令扩展通常使用一个额外的命令来增强 `yiic` 的功能. 假设一个控制台命令 `XYZClass` 属于 `xyz` 扩展, 我们可以通过设定控制台应用的配置来使用它: `return array(`

```
108     'commandMap'=>array(
109         'xyz'=>array(
110             'class'=>'ext.xyz.XyzClass',
111             'property1'=>'value1',
112             'property2'=>'value2',
113         ),
```

```
114         // 其他命令
115     ),
116 );
```

复制代码

然后,我们就能使用配备了额外命令 `xyz` 的 `yiic` 工具了.

注意: 控制台应用通常使用了一个不同于 `Web` 应用的配置文件.如果使用了 `yiic webapp` 命令 创建了一个应用,这样的话,控制台应用的 `protected/yiic` 的配置文件就是 `protected/config/console.php` 了,而 `Web` 应用的配置文件 则是 `protected/config/main.php`.

一般组件

使用一个一般 组件, 我们首先需要通过使用

```
Yii::import('ext.xyz.XyzClass');
```

来包含它的类文件.然后,我们既可以创建一个类的实例,配置它的属性,也可以调用它的方法.我们还可以创建一个新的子类来扩展它。

开源 PHP 开发框架 Yii 全方位教程 (16) 创建扩展

由于扩展意味着是第三方开发者使用，需要一些额外的努力去创建它。以下是一些一般性的指导原则：

- 扩展最好是自给自足。也就是说，其外部的依赖应是最少的。如果用户的扩展需要安装额外的软件包，类或资源档案，这将是一个头疼的问题。
- 文件属于同一个扩展的，应组织在同一目录下，目录名用扩展名称。
- 扩展里面的类应使用一些单词字母前缀，以避免与其他扩展命名冲突。
- 扩展应该提供详细的安装和 API 文档。这将减少其他开发人员使用扩展时花费的时间和精力。
- 扩展应该用适当的许可。如果您想您的扩展能在开源和闭源项目中使用，你可以考虑使用许可证，如 BSD 的，麻省理工学院等，但不是 GPL 的，因为它要求其衍生的代码是开源的。

1

在下面，我们根据 `overview` 中所描述的分类，描述如何创建一个新的扩展。当您创建一个主要用于在您自己项目的组件，这些描述也适用。

Application Component（应用组件）

一个 `application component` 应实现接口 `IApplicationComponent` 或继承自 `CApplicationComponent`。主要需要实现的方法是 `IApplicationComponent::init`，部件在此执行一些初始化工作。此方法在部件创建和属性值（在 `application configuration` 里指定的）被赋值后调用。

默认情况下，一个应用程序部件创建和初始化，只有当它首次访问期间要求处理。如果一个应用程序部件需要在应用程序实例被创建后创建，它应要求用户在 `CApplication::preload` 的属性中列出他的编号。

行为(Behavior)

要创建一个 `behavior`，必须实现 `IBehavior` 接口。方便起见，Yii 提供了一个基础类 `CBehavior`，它已经实现了这个接口并提供了一些额外的方便方法。子类主要需要实现为要附加的组件可用的额外方法。

当为 `CModel` 和 `CActiveRecord` 开发行为时，可以分别扩展 `CModelBehavior` 和 `CActiveRecordBehavior`。这些基础类专为 `CModel` 和 `CActiveRecord` 提供了额外的特征。例如，`CActiveRecordBehavior` 类实现一些方法集合来响应 `ActiveRecord` 对象里唤起的生命周期事件。子类可以重写这些方法，让自定义的代码参与到 AR 的生命周期里。

下面的代码展示了一个 `ActiveRecord` 行为的例子。当这个行为被附加到一个 AR 对象，当 AR 对象通过调用 `save()` 被保存时，它将自动以当前时间戳设置 `create_time` 和 `update_time` 属性。`class TimestampBehavior extends CActiveRecordBehavior`

2 {

```

3     public function beforeSave($event)
4     {
5         if($this->owner->isNewRecord)
6             $this->owner->create time=time();
7         else
8             $this->owner->update time=time();
9     }
10 }

```

复制代码

11 部件

`widget` 应继承 `CWidget` 或其子类。

最简单的方式建立一个新的 `widget` 是扩展一个现成的 `widget` 和重载它的方法或改变其默认的属性值。例如，如果您想为 `CTabView` 使用更好的 CSS 样式，当使用 `widget` 时，您可以配置其 `CTabView::cssFile` 属性。您还可以扩展 `CTabView` 如下，让您在使用 `widget` 时，不再需要配置属性。class `MyTabView` extends `CTabView`

```

12 {
13     public function init()
14     {
15         if($this->cssFile===null)
16         {
17             $file=dirname(<strong>FILE</strong>).DIRECTORY_SEPARATOR.'tabview.css';
18             $this->cssFile=Yii::app()->getAssetManager()->publish($file);
19         }
20         parent::init();
21     }
22 }

```

复制代码

23 在上面，我们重载 `CWidget::init` 方法和指定 `CTabView::cssFile` 的 URL 到我们的新的默认 CSS 样式如果此属性未设置时。我们把新的 CSS 样式文件和 `MyTabView` 类文件放在相同的目录下，以便它们能够封装成扩展。由于 CSS 样式文件不是通过 Web 访问，我们需要发布作为一项 `asset` 资源。

要从零开始创建一个新的 `widget`，我们主要是需要实现两个方法：`CWidget::init` 和 `CWidget::run`。第一种方法是当我们在视图中使用 `$this->beginWidget` 插入一个 `widget` 时被调用，第二种方法在 `$this->endWidget` 被调用时调用。如果我们想在这两个方法调用之间捕捉和处理显示的内容，我们可以在 `CWidget::init` 开始 `output buffering` 并在 `CWidget::run` 中检索

缓冲后的输出以作进一步处理。

在网页中使用的 `widget` 需要引入 `CSS`，`Javascript` 或其他资源文件。我们称这些文件为 `assets`，因为它们和 `widget` 类文件在一起，而且通常 `Web` 用户无法访问。为了使这些档案通过 `Web` 访问，我们需要用 `CWebApplication::assetManager` 发布它们，例如上述代码段所示。

此外，如果我们想包括 `CSS` 或 `JavaScript` 文件在当前的网页，我们需要使用 `CClientScript` 注册它：

```
class MyWidget extends CWidget
24 {
25     protected function registerClientScript()
26     {
27         // ...publish CSS or JavaScript file here...
28         $cs=Yii::app()->clientScript;
29         $cs->registerCssFile($cssFile);
30         $cs->registerScriptFile($jsFile);
31     }
32 }
```

复制代码

33 一个 `widget` 也可能有自己的视图文件。如果是这样，在包含 `widget` 类文件的目录下创建一个目录 `views`，并把所有的视图文件放里面。在 `widget` 类中使用 `$this->render('ViewName')` 来渲染 `widget` 视图，类似于我们在控制器里做的。

Action (动作)

`action` 应扩展自 `CAction` 或者其子类。 `action` 要实现的主要方法是 `IAction::run` 。

Filter (过滤器)

`filter` 应扩展自 `CFilter` 或者其子类。 `filter` 要实现的主要方法是 `CFilter::preFilter` 和 `CFilter::postFilter`。前者是在 `action` 之前被执行，而后者是在之后。

```
class MyFilter extends CFilter
34 {
35     protected function preFilter($filterChain)
36     {
37         // logic being applied before the action is executed
38         return true; // false if the action should not be executed
39     }
40     protected function postFilter($filterChain)
41     {
42         // logic being applied after the action is executed
43     }
44 }
```

复制代码

45 参数\$filterChain 的类型是 CFilterChain，它包含了当前被过滤的 action 的相关信息。

Controller（控制器）

controller 要作为扩展需扩展自 CExtController，而不是 CController。主要的原因是因为 CController 假设控制器视图文件位于 application.views.ControllerID 下，而 CExtController 认定视图文件在 views 目录下，也是包含控制器类目录的一个子目录。因此，很容易重新分配控制器，因为它的视图文件和控制类是在一起的。

Validator（验证）

Validator 需继承 CValidator 和实现 CValidator::validateAttribute 方法。class MyValidator extends CValidator

```
46 {
47     protected function validateAttribute($model,$attribute)
48     {
49         $value=$model->$attribute;
50         if($value has error)
51             $model->addError($attribute,$errorMessage);
52     }
53 }
```

复制代码

54 Console Command（控制台命令）

console command 应继承 CConsoleCommand 和实现 CConsoleCommand::run 方法。或者，我们可以重载 CConsoleCommand::getHelp 来提供一些更好的有关帮助命令。class MyCommand extends CConsoleCommand

```
55 {
56     public function run($args)
57     {
58         // $args gives an array of the command-line arguments for this command
59     }
60     public function getHelp()
61     {
62         return 'Usage: how to use this command';
63     }
64 }
```

复制代码

Module（模块）

请参阅 [modules 一节](#) 了解如何创建一个模块。

一般准则制订一个模块，它应该是独立的。模块所使用的资源文件（如 CSS ， JavaScript ， 图片），应该和模块一起分发。还有模块应发布它们，以便可以 Web 访问它们 。

Generic Component（通用组件）

开发一个通用组件扩展类似写一个类。再次强调，该组件还应该自足，以便它可以很容易地被其他开发者使用。

开源 PHP 开发框架 Yii 全方位教程 (17) 使用第三方库

Yii 是精心设计，使第三方库可易于集成，进一步增强 Yii 的功能。 当在一个项目中使用第三方库，程序员往往遇到关于类命名和文件包含的问题。 因为所有 Yii 类以 C 字母开头，这就减少可能会出现的类命名问题;而且因为 Yii 依赖 SPL autoload 执行类文件包含，如果他们使用相同的自动加载功能或 PHP 包含路径包含类文件，它可以很好地结合。

下面我们用一个例子来说明如何在一个 Yii 应用中使用 Zend framework 的 Zend_Search_Lucene 组件。

首先，假设 `protected` 是 application base directory，我们提取 Zend Framework 的发布文件到 `protected/vendors` 目录。 假设 `protected` 是应用的基础目录。 确认 `protected/vendors/Zend/Search/Lucene.php` 文件存在。

第二，在一个 controller 类文件的开始，加入以下行：

```
1  Yii::import('application.vendors.*');
2  require_once('Zend/Search/Lucene.php');
```

复制代码

上述代码包含类文件 `Lucene.php`。 因为我们使用的是相对路径，我们需要改变 PHP 的包含路径，以使文件可以正确定位。 这是通过在 `require_once` 之前调用 `Yii::import` 完成的。

一旦上述设立准备就绪后，我们可以在 controller action 里使用 `Lucene` 类，类似如下：

```
3  $lucene=new Zend_Search_Lucene($pathOfIndex);
4  $hits=$lucene->find(strtolower($keyword));
```

复制代码

开源 PHP 开发框架 Yii 全方位教程 (18) 定义 fixture

自动化测试需要被执行很多次。要确保测试过程是重复的，我们想要运行测试以一些已知的状态，称为fixture。例如，要测试 blog 应用的文章创建功能，每次我们运行这个测试，存储文章相关数据的表（例如 Post 表， Comment 表）应被还原为一些固定的状态。PHPUnit 文档 已经对通用 fixture 建立做了很好的描述。在这一小节中，我们主要描述如何建立数据库fixtures，如我们在例子中讲到的。

在测试数据库驱动的 Web 应用时，建立数据库 fixtures 可能是最耗时的部分之一。Yii 引入了 CDbFixtureManager 应用组件来缓解这个问题。当运行一个测试集合时，它基本上执行如下工作：

- 在所有测试运行之前，它重设所有与测试相关的表为一些已知的状态。
- 在一个单独的测试方法运行之前，它重设指定的表为一些已知的状态。
- 在一个测试方法执行过程中，it provides access to the rows of the data that contribute to the fixture.

```
1
为了使用 CDbFixtureManager，我们在应用配置中如下设置它，return array(
2     'components'=>array(
3         'fixture'=>array(
4             'class'=>'system.test.CDbFixtureManager',
5         ),
6     ),
7 );
```

复制代码

8 然后我们提供fixture 数据，位于目录 protected/tests/fixtures 中。通过在应用配置中设置 CDbFixtureManager::basePath，可以将这个目录设置成别的目录。 fixture 数据被组织为一个 PHP 文件集合，称为 fixture 文件。Each fixture file returns an array representing the initial rows of data for a particular table. 文件的名称和表的名称相同。下面是一个例子， Post 表的 fixture 数据存储在一个名为 Post.php 的文件中： <?php

```
9 return array(
10     'sample1'=>array(
11         'title'=>'test post 1',
12         'content'=>'test post content 1',
13         'createTime'=>1230952187,
14         'authorId'=>1,
15     ),
16     'sample2'=>array(
17         'title'=>'test post 2',
18         'content'=>'test post content 2',
19         'createTime'=>1230952287,
20         'authorId'=>1,
```

21),

22);

复制代码

如我们看到的，在上面两行数据被返回。每行记录表示为一个关联数组，它的键是字段名，它的值是对应的字段值。此外，每行记录由一个字符串(例如 `sample1, sample2`) 索引，这个字符串称为 `row alias`。稍后当我们编写测试脚本时，我们可以方便的根据它的别名指向一行记录。在下一个小节，我们将详细描述。

你可能注意到在上面的 `fixture` 中我们不指定 `id` 字段值。这是因为 `id` 字段被定义为一个自增的主键，当我们插入新记录时，它的值被填充。

当 `CDbFixtureManager` 被首次引用时，它将检查每个 `fixture` 文件并使用它来重设对应的表。它通过截取表，重设表的自增主键的顺序值来重设表，然后插入 `fixture` 文件中的记录到表中。

有时， `we may not want to reset every table which has a fixture file before we run a set of tests`，因为重设太多的 `fixture` 文件将花费很长时间。这时，我们可以编写一个 `PHP` 脚本以一个定制的方式来做初始化的工作。脚本应当被保存为 `init.php`，放置在含有其他 `fixture` 文件的相同目录。当 `CDbFixtureManager` 探测到这个文件存在，它将执行这个脚本而不是重设每个表。

若你不喜欢重设一个表的默认方式，例如截取它并插入 `fixture` 文件中的数据，改变它也是可能的。这时，我们可以为指定的 `fixture` 文件编写一个初始化脚本。这个脚本必须命名为表名字跟上 `.init.php`。例如， `Post` 表的初始化脚本将是 `Post.init.php`。当 `CDbFixtureManager` 看到这个脚本，它将执行这个脚本而不是使用默认的方式来重设数据表。

太多 `fixture` 文件将极大增加测试时间。由于这个原因，你应当只为那些内容在测试中改变的表提供 `fixture` 文件。那些用于查询的表不发生改变，因此无需 `fixture` 文件。

在下面两个小节，我们将描述如何在[单元测试](#)和[功能测试](#)中使用由 `CDbFixtureManager` 管理的 `fixtures`。

开源 PHP 开发框架 Yii 全方位教程 (19) 单元测试

- 因为 Yii 测试框架基于 PHPUnit, 推荐你首先阅览 PHPUnit 文档, 对如何编写一个单元测试有一个基本理解。我们总结在 Yii 中编写一个单元测试的基本原则如下:
一个单元测试编写为一个类 `XYZTest`, 它扩展自 `CTestCase` 或 `CDbTestCase`, `XYZ` 代表被测试的类。例如, 要测试 `Post` 类, 根据约定我们命名对应的测试单元为 `PostTest`。基础类 `CTestCase` 是为通用单元测试准备的, 而 `CDbTestCase` 适合测试 `active record` 模型类。因为 `PHPUnit_Framework_TestCase` 是这两个类的根类, 我们可以使用从这个类继承而来的所有方法。
- 单元测试类被保存为一个名为 `XYZTest.php` 的 PHP 文件。根据约定, 单元测试文件必须被保存到目录 `protected/tests/unit` 中。
- 测试类主要包含名为 `testAbc` 的测试方法集合, `Abc` 通常是被测试的类方法的名字。
- 一个测试方法经常包含 a sequence of assertion statements (e.g. `assertTrue`, `assertEquals`) which serve as checkpoints on validating the behavior of the target class.

下面我们主要描述如何为 `active record` 模型类编写单元测试。我们将扩展 `CDbTestCase` 类编写测试类, 因为它提供了之前我们介绍的数据库 `fixture` 支持。

假设我们想要测试 `blog` 演示 中的 `Comment` 模型类, 我们首先创建 `CommentTest` 类并保存为 `protected/tests/unit/CommentTest.php`:

```
1 class CommentTest extends CDbTestCase
2 {
3     public $fixtures=array(
4         'posts'=>'Post',
5         'comments'=>'Comment',
6     );
7     .....
8 }
```

[复制代码](#)

在这个类中, 我们指定成员变量 `fixtures` 是一个数组, 指示在测试中使用那些 `fixtures`。数组代表了一个从 `fixture` 名字到模型类名字或 `fixture` 表名字(例如 从 `fixture` 名字 `posts` 到模型类 `Post`)之间的映射。注意当映射到 `fixture` 表名字时, 我们应当在表名字前加前缀冒号(例如 `:Post`)来和模型类名字相区别。当使用模型类名字时, 对应的表被看作 `fixture` 表。我们之前讲过, 当每次一个测试方法被执行时, `fixture` 表将被重设到一些已知的状态。

`fixture` 名字允许我们以一个方便的方式来访问测试方法中的 `fixture` 数据。如下代码展示了它的典型用法:

```
9 // return all rows in the 'Comment' fixture table
10 $comments = $this->comments;
11 // return the row whose alias is 'sample1' in the 'Post' fixture table
12 $post = $this->posts['sample1'];
13 // return the AR instance representing the 'sample1' fixture data row
```

```
14 $post = $this->posts('sample1');
```

复制代码

若一个 `fixture` 被声明为使用它的表名字 (例如 `'posts'=>':Post'`), 然后上面第三个用法是无效的, 因为我们没有这个表关联的模型类的信息。

接下来, 我们编写 `testApprove` 方法来测试 `Comment` 模型类中的 `approve` 方法。代码非常明了: 我们首先插入一条评论, 状态是等待审核; 然后我们通过在数据库中检索它来验证这条评论处于等待审核状态; 组后我们调用 `approve` 方法并验证状态如预期的那样改变。

```
15 public function testApprove()
16 {
17     // insert a comment in pending status
18     $comment=new Comment;
19     $comment->setAttributes(array(
20         'content'=>'comment 1',
21         'status'=>Comment::STATUS PENDING,
22         'createTime'=>time(),
23         'author'=>'me',
24         'email'=>'me@example.com',
25         'postId'=>$this->posts['sample1']['id'],
26     ),false);
27     $this->assertTrue($comment->save(false));
28     // verify the comment is in pending status
29     $comment=Comment::model()->findByPk($comment->id);
30     $this->assertTrue($comment instanceof Comment);
31     $this->assertEquals(Comment::STATUS PENDING,$comment->status);
32     // call approve() and verify the comment is in approved status
33     $comment->approve();
34     $this->assertEquals(Comment::STATUS APPROVED,$comment->status);
35     $comment=Comment::model()->findByPk($comment->id);
36     $this->assertEquals(Comment::STATUS APPROVED,$comment->status);
37 }
```

复制代码

开源 PHP 开发框架 Yii 全方位教程 (20) 功能测试

在阅读这一小节之前，推荐你首先阅读 Selenium 文档 和 PHPUnit 文档。我们总结在 Yii 中编写一个功能测试的基本原则如下：

- 类似于单元测试，一个功能测试编写为一个类 `XYZTest`，它扩展自 `CWebTestCase`，`XYZ` 代表被测试的类。因为 `PHPUnit_Extensions_SeleniumTestCase` 是 `CWebTestCase` 的根类，我们调用从根类继承而来的所有方法。
- 功能测试类被保存到名为 `XYZTest.php` 的 PHP 文件中。根据约定，功能测试文件必须保存到目录 `protected/tests/functional` 中。
- 测试类主要包含名为 `testAbc` 的测试方法集合，`Abc` 同时是被测试的功能的名字。例如，要测试用户登录特征，我们可以有一个测试方法名为 `testLogin`。
- A test method usually contains a sequence of statements that would issue commands to Selenium RC to interact with the Web application being tested. It also contains assertion statements to verify that the Web application responds as expected.

1

在描述如何使用功能测试之前，让我们看一下由 `yiic webapp` 命令产生的 `WebTestCase.php` 文件。这个文件定义了 `WebTestCase`，它充当所有功能测试类的基础类。`define('TEST_BASE_URL','http://localhost/yii/demos/blog/index-test.php/');`

```
2 class WebTestCase extends CWebTestCase
3 {
4     /**
5      * Sets up before each test method runs.
6      * This mainly sets the base URL for the test application.
7      */
8     protected function setUp()
9     {
10         parent::setUp();
11         $this->setBrowserUrl(TEST_BASE_URL);
12     }
13     .....
14 }
```

复制代码

15 类 `WebTestCase` 主要设置被测试页面的基础 URL。稍后在测试方法中，我们可以使用相对 URL 来指定哪些页面被测试。

我们也应当注意在基础测试 URL 中，我们使用 `index-test.php` 而不是 `index.php` 作为入口文件。`index-test.php` 和 `index.php` 唯一的不同是前者使用 `test.php` 作为应用配置文件，而后者使用 `main.php` 作为配置文件。

现在我们描述在 `blog` 演示 中如何测试展示一篇文章的功能。我们首先编写测试类如下。

noting that the test class extends from the base class we just described:`class PostTest extends WebTestCase`

```
16 {
17     public $fixtures=array(
18         'posts'=>'Post',
19     );
20     public function testShow()
21     {
22         $this->open('post/1');
23         // verify the sample post title exists
24         $this->assertTextPresent($this->posts['sample1']['title']);
25         // verify comment form exists
26         $this->assertTextPresent('Leave a Comment');
27     }
28     .....
29 }
```

[复制代码](#)

类似于编写一个单元测试，我们声明这个测试使用的 `fixture`，这里我们指示使用 `Post fixture`。在 `testShow` 测试方法中，我们首先指示 `Selenium RC` 打开 URL `post/1`，注意这是一个相对 URL，完整的 URL 应当是我们在基础中设置的基础 URL 再加上相对 URL(例如 `http://localhost/yii/demos/blog/index-test.php/post/1`)。然后我们验证我们可以找到 `sample1` 文章的标题出现在当前网页。我们也可以验证网页包含了文本 `Leave a Comment`。

在运行功能测试之前，`Selenium-RC` 服务器必须开启。可以通过在 `Selenium` 服务器安装目录中执行命令 `java -jar selenium-server.jar` 实现。

开源 PHP 开发框架 Yii 全方位教程 (21) 自动生成代码

1 自动生成代码从版本1.1.2开始，Yii 装备了一个基于 web 的代码生成工具，叫做 Gii。它替代之前的 yiic shell 生成工具(它运行在命令行)。在这一小节中，我们将描述如何使用 Gii 以及如何扩展 Gii 来增加我们的开发生产力。使用 Gii

Gii 以一个模块的方式运行，必须在一个已存在的 Yii 应用内部使用。要使用 Gii，我们首先改变应用配置如下：return array(

```
2     .....
3     'modules'=>array(
4         'gii'=>array(
5             'class'=>'system.gii.GiiModule',
6             'password'=>'pick up a password here',
7             // 'ipFilters'=>array(...a list of IPs...),
8             // 'newFileMode'=>0666,
9             // 'newDirMode'=>0777,
10        ),
11    ),
12 );
```

复制代码

13 在上面，我们声明了一个模块名为 gii，它的类是 GiiModule。我们也为这个模块指定了一个密码，当访问 Gii 时需要输入。

默认的，出于安全考虑，Gii 被配置为只允许在本地访问。若我们想要在另外信任的机器上访问，可以在如上代码中配置 GiiModule::ipFilters 属性。

因为 Gii 可以产生并保存新代码文件到已存在的应用中，我们需要确保 web 服务器进程有权限这样做。在上面的 GiiModule::newFileMode 和 GiiModule::newDirMode 属性控制这些新文件和目录应当如何被产生。

现在我们可以通过 URL `http://hostname/path/to/index.php?r=gii` 访问 Gii，这里我们假设 `http://hostname/path/to/index.php` 是访问已存在 Yii 应用的 URL。

若已存在的 Yii 应用使用 path 格式的 URL，我们可以通过 URL `http://hostname/path/to/index.php/gii` 访问 Gii。我们也需要增加如下 URL 规则到已存在 URL 规则的前面:'components'=>array(

```
14     .....
15     'urlManager'=>array(
```



```

16     'urlFormat'=>'path',
17     'rules'=>array(
18         'gii'=>'gii',
19         'gii/<controller:\w+>'=>'gii/<controller>',
20         'gii/<controller:\w+>/<action:\w+>'=>'gii/<controller>/<action>',
21         ...existing rules...
22     ),
23 ),
24 )

```

复制代码

25 **Gii** 有一个新的默认代码生成器。每个代码生成器负责生成一个特定类型的代码。例如，**controller** 生成器生成一个控制器类以及一些动作视图脚本；**model** 生成器为指定的数据表生成一个 **ActiveRecord** 类。

使用一个生成器的基本工作流程如下：

1. 进入生成器页面；
2. 填写字段以指定代码生成器的参数。例如，要使用 **module** 生成器创建一个新模块，你需要指定模块 ID；
3. 点击 **Preview** 按钮预览生成的代码。你将看到一个表展示了将被生成的一个代码文件列表。你可以点击其中的任何文件以预览代码。
4. 点击 **Generate** 按钮以生成代码文件；
5. 查看代码生成日志。

扩展 GII

尽管默认的代码生成器已经可以产生很强大的代码，我也经常希望能定制他们或者创建个新的来满足我们的需求，例如，我们希望生成的代码能符合我们的编码风格，或者我们希望代码能支持国际化，这些都可以轻松的通过 **Gii** 实现 **Gii** 可以通过两种模式进行扩展：定制代码模板来扩展已有的代码生成器，和编写新的代码生成器

创建代码生成器

代码生成器所在的目录名被视为代码生成器的名字。该目录往往包含以下内容“model/
the model generator root folder

26	ModelCode.php	the code model used to generate code
27	ModelGenerator.php	the code generation controller
28	views/	containing view scripts for the generator
29	index.php	the default view script
30	templates/	containing code template sets

31	default/	the 'default' code template set
32	model.php	the code template for generating model class code

复制代码

33 代码生成器的检索路径

Gii 通过 `GiiModule::generatorPaths` 属性设置的路径搜索可用的代码生成器。如果需要定制，我们可以在应用的配置中如下配置这个属性 `return array(`

```
34     'modules'=>array(  
35         'gii'=>array(  
36             'class'=>'system.gii.GiiModule',  
37             'generatorPaths'=>array(  
38                 'application.gii', // a path alias  
39             ),  
40         ),  
41     ),  
42 );
```

复制代码

43 上面的配置表明 Gii 除了在默认的 `system.gii.generators` 之外，还在路径别名为 `application.gii` 的目录下检索可用的代码生成器，

有可能会在不同的目录底下有相同名字的代码生成器，这种情况下在 `GiiModule::generatorPaths` 中最先指定的，具有优先权。

自定义代码模板

这是最简单的也是最常用的扩展 Gii 的方法。我们使用一个例子来解释如何抵制代码模板。假设我们想要自定义 `model` 代码生成器生成的代码。

我们首先创建一个名为 `protected/gii/model/templates/compact` 的路径。这里的 `model` 意味着我们要重写摩尔恩的 `model` 代码生成器。`templates/compact` 表示我们要添加一个新的代码模板，名字为 `compact`

然后我们修改应用的配置，把 `application.gii` 加入到 `GiiModule::generatorPaths` 中，如同上一节中所叙

现在打开 `model` 代码生成器页面。点击代码模板字段，在出现的下拉框里我们能看到我们新加的模板路径 `compact`，如果我们选择这个模板来生成代码，我们会看到一个错误，那是因为我们还没有在 `compact` 底下放入任何的实际代码模板。

下面开始我们真正的自定义的工作。打开文件 `protected/gii/model/templates/compact/model.php` 进行编辑。记住这个文件会像被作为视图文件使用，这意味着它可以包含 PHP 语句和声明。让我们修改模板，以便生成器产生的 `attributeLabels()` 方法，使用 `Yii::t()` 来对标签支持国际化 `public function attributeLabels()`

```
44 {
45     return array(
46         <?php foreach($labels as $name=>$label): ?>
47             <?php echo "'$name' => Yii::t('application', '$label'),\n"; ?>
48         <?php endforeach; ?>
49     );
50 }
```

复制代码

在每一个代码模板，我们可以访问一些预定义的变量，例如上面例子中的 `$labels`。这些变量由相应的代码生成器产生。不同的代码生成器在他们各自的模板中可能产生不同的变量。请仔细阅读默认代码模板中的描述

创建新的代码生成器

在本节中，我们讲解了如何创建一个代码生成器，该生成器的作用是生成了一个生成一个部件类。

我们首先创建路径 `protected/gii/widget`。在这个路径下，我们创建以下文件

- `WidgetGenerator.php`: contains the `WidgetGenerator` controller class. 这是部件生成器的入口.
- `WidgetCode.php`: contains the `WidgetCode` model class. 代码生成的主要逻辑.
- `views/index.php`: 代码生成器输入表单的视图脚本
- `templates/default/widget.php`: 生成部件类的默认代码模板

51

Creating `WidgetGenerator.php`

`WidgetGenerator.php` 文件非常简单，仅包含以下代码 `class WidgetGenerator extends CCodeGenerator`

```
52 {
53     public $codeModel='application.gii.widget.WidgetCode';
54 }
```

复制代码

55 在上面的代码中，我们指定生成器使用的 `model` 类的路径别名是 `application.gii.widget.WidgetCode`。`WidgetGenerator` 扩展自 `CCodeGenerator`，`CCodeGenerator` 实现了许多功能，including the controller actions needed to coordinate the code generation process.

Creating `WidgetCode.php`

`WidgetCode.php` 文件了 `WidgetCode` 模型类, 它包含了根据用户输入生成部件的类的主要逻辑。在本例中, 我们假设用户仅仅输入了部件的类名, 参考如下: `class WidgetCode extends CCodeModel`

```
56 {
57     public $className;
58     public function rules()
59     {
60         return array_merge(parent::rules(), array(
61             array('className', 'required'),
62             array('className', 'match', 'pattern'=> '/^\w+$/'),
63         ));
64     }
65     public function attributeLabels()
66     {
67         return array_merge(parent::attributeLabels(), array(
68             'className'=> 'Widget Class Name',
69         ));
70     }
71     public function prepare()
72     {
73         $path=Yii::getPathOfAlias('application.components.' . $this->className) . '.php';
74         $code=$this->render($this->templatepath.'/widget.php');
75         $this->files[]=new CCodeFile($path, $code);
76     }
77 }
```

复制代码

78 `WidgetCode` 类继承于 `CCodeModel`。想普通的模型类一样, 在本例中我们生成了 `rules()`和 `attributeLabels()` 分别来验证用户输入和显示属性标签。注意因为 `CCodeModel` 类已经定义了一些 `rule` 和属性标签, 我们需要把已经定义的和我们的 `ruler` 和标签进行合并

`prepare()`方法准备要被产生的代码。它的主要任务是准备 `CCodeFile` 对象列表, 其中的每一个都被渲染了一个代码文件。在我们的例子中, 我们只需要产生一个 `CCodeFile` 对象来渲染要生成的部件类文件。生成的部件类被保存在 `protected/components` 目录底下, 我们调用 `CCodeFile::render` 来生成实际的代码, 该方法把代码模板当做 `php` 脚本加载, 并返回输出内容作为生成的代码。

Creating views/index.php

完成了控制器 (`WidgetGenerator`) 和模型 (`WidgetCode`) 之后, 现在让我们来创建视图文件

```

views/index.php<h1>Widget Generator</h1>
79 <?php $form=$this->beginWidget('CCodeForm', array('model'=>$model)); ?>
80     <div class="row">
81         <?php echo $form->labelEx($model,'className'); ?>
82         <?php echo $form->textField($model,'className',array('size'=>65)); ?>
83         <div class="tooltip">
84             Widget class name must only contain word characters.
85         </div>
86         <?php echo $form->error($model,'className'); ?>
87     </div>
88 <?php $this->endWidget(); ?>

```

复制代码

89 上面的代码中，我们主要通过 **CCodeForm** 部件显示了一个表单。在这个表单中，我们显示了一个字段用于输入类的名字。（ In this form, we display the field to collect the input for the **className** attribute in **WidgetCode**）

创建这个表单的时候，我们可以用 **CCodeForm** 部件的两个不错的功能。一个是 **input tooltips**，另一个是 **sticky inputs**。

如果你使用过任何的默认代码生成器，你将会注意到，当焦点放到输入框上的时候，一个友好的 **tooltip** 会显示在一侧，这可以通过编写挨着的 **CSS class** 名为 **tooltip** 的 **div** 来很容易的实现。

（ This can easily achieved here by writing next to the input field a div whose **CSS class** is **tooltip**。）

对一些输入框来说，我们需要记住他们最后一次的有效输入，这样用户使用生成器生成代码的时候不用重新输入。例如，默认的控制生成器重控制器基类名字的输入框，粘性字段最初用高亮的静态文本显示，当我们点击的时候，会变成输入框让用户进行输入。

声明一个字段有粘性，我们需要做两件事

首先我们需要为相应的模型属性声明一个粘性验证规则。例如默认的控制生成器有如下的规则来声明 **baseClass** 和 **attributes** 具有粘性 `public function rules()`

```

90 {
91     return array_merge(parent::rules(), array(
92         .....
93         array('baseClass, actions', 'sticky'),
94     ));
95 }

```

复制代码

```
96 然后，我们需要为视图中的输入框所在的 div 添加名为 sticky 的 css class，如下： <div
class="row sticky">
97     ...input field here...
98 </div>
```

复制代码

99 创建 `templates/default/widget.php`

最后我们创建代码模板 `templates/default/widget.php`，我们已经提到过，这个最为视图脚本，可以包含 PHP 声明和表达式。在模板中，我们经常使用的 `$this` 是模型示例的引用。在本例中 `$this` 是 `WidgetModel` 对象的一个引用。我们可以这样读取用户输入的类型名：`$this->className``<?php echo '<?php'; ?>`

```
100 class <?php echo $this->className; ?> extends CWidget
101 {
102     public function run()
103     {
104     }
105 }
```

复制代码

这便是代码生成器的创建，我们可以通过地址 <http://hostname/path/to/index.php?r=gii/widget> 来访问这个代码生成器。

开源 PHP 开发框架 Yii 全方位教程 (22) URL 管理

1 Web 应用程序完整的 URL 管理包括两个方面。首先，当用户请求约定的 URL，应用程序需要解析它变成可以理解的参数。第二，应用程序需要提供一种创造 URL 的方法，可以让创建的 URL 应用程序正确理解。对于 Yii 应用程序，这些通过 `CUrlManager` 辅助完成。

Creating URLs (创建网址)

虽然 URL 可被硬编码在控制器的视图 (view) 文件，但动态的创建他们会更加灵活：

```
$url=$this->createUrl($route,$params);
```

复制代码

2 `$this` 指的是控制器实例; `$route` 指定请求的 route 的要求; `$params` 列出了附加在网址中的 GET 参数。

默认情况下，URL 以 `get` 格式使用 `createUrl` 创建。例如，提供 `$route='post/read'` 和 `$params=array('id'=>100)`，我们将获得以下网址：`/index.php?r=post/read&id=100`

复制代码

3 参数以一系列 `Name=Value` 通过符号串联起来出现在请求字符串，`r` 参数指的是请求的 route。这种 URL 格式用户友好性不是很好，因为它需要一些非单词字符。

我们可以使上述网址看起来更简洁，更不言自明，通过采用所谓的 `path` 格式，省去查询字符串并把 GET 参数加到路径信息，作为网址的一部分：`/index.php/post/read/id/100`

复制代码

4 要更改 URL 格式，我们应该配置 `urlManager` 应用组件，以便 `createUrl` 可以自动切换到新格式，并可以让应用程序可以正确理解新的网址：`array(`

```
5     .....
6     'components'=>array(
7         .....
8         'urlManager'=>array(
9             'urlFormat'=>'path',
10        ),
11    ),
12 );
```

复制代码

13 请注意，我们不需要指定的 `urlManager` 元件的类，因为它在 `CWebApplication` 预声明为 `CUrlManager`。

此网址通过 `createUrl` 方法所产生的的是一个相对地址。为了得到一个绝对的 `url`，我们可以用前缀 `yii::app()->hostInfo`，或调用 `createAbsoluteUrl`。

User-friendly URLs（用户友好的 URL）

当用 `path` 作为 URL 格式，我们可以指定某些 URL 规则使我们的网址对用户更加友好性。例如，我们可以产生一个短短的 `URL/post/100`，而不是冗长 `/index.php/post/read/id/100`。网址创建和解析都是通过 `CUrlManager` 指定网址规则。

要指定的 URL 规则，我们必须设定 `urlManager` 应用组件的属性 `rules`：array(

```
14     .....
15     'components'=>array(
16         .....
17         'urlManager'=>array(
18             'urlFormat'=>'path',
19             'rules'=>array(
20                 'pattern1'=>'route1',
21                 'pattern2'=>'route2',
22                 'pattern3'=>'route3',
23             ),
24         ),
25     ),
26 );
```

复制代码

27 规则被指定为一个由 `pattern-route pairs` 组成的数组，每一个对应一个规则。一个规则的 `pattern` 是一个字符串，用来匹配 URL 的路径信息部分(the path info part of URLs)。一个规则的 `route` 应当指向一个有效的控制器路由。

除了上面的 `pattern-route` 格式，一个规则也可以以自定义选项来指定，如下：`'pattern1'=>array('route1', 'urlSuffix'=>'.xml', 'caseSensitive'=>false)`

复制代码

在上面，数组包含一个自定义选项列表。从版本 1.1.0 起，下面的选项可用：

- `urlSuffix`: 此规则使用的 URL 后缀。默认是 `null`，意味着使用 `CUrlManager::urlSuffix` 的值。
- `caseSensitive`: 是否规则区分大小写。默认是 `null`，意味着使用 `CUrlManager::caseSensitive` 的值。

- `defaultParams`: 此规则提供的默认 GET 参数(`name=>value`)。When this rule is used to parse the incoming request, the values declared in this property will be injected into GET
- `matchValue` : whether the GET parameter values should match the corresponding sub patterns in the rule when creating a URL. Defaults to null, meaning using the value of `CUrlManager::matchValue`. If this property is false, it means a rule will be used for creating a URL if its route and param patterns. Note that setting this property to true will degrade performance.

28

使用命名参数(Using Named Parameters)

规则可以绑定少量的 GET 参数。这些出现在规则格式的 GET 参数，按照如下的特殊格式：

`ParamName` 表示 GET 参数名字，可选项 `ParamPattern` 表示将用于匹配 GET 参数值的正则表达式。如果省略了 `ParamPattern`，表示该参数匹配除了 `/` 之外的任何字符。当生成一个网址 (URL) 时，这些参数将被相应的参数值替换；当解析一个网址时，相应的 GET 参数将通过解析结果来生成。

我们使用一些例子来解释 URL 规则如何工作。我们假设我们的规则包括如下三个：array(

```
29     'posts'=>'post/list',
30     'post/<id:\d+>'=>'post/read',
31     'post/<year:\d{4}>/<title>'=>'post/read',
32 )
```

复制代码

- 调用 `$this->createUrl('post/list')` 生成 `/index.php/posts`。第一个规则适用。
- 调用 `$this->createUrl('post/read',array('id'=>100))` 生成 `/index.php/post/100`。第二个规则适用。
- 调用 `$this->createUrl('post/read',array('year'=>2008,'title'=>'a sample post'))` 生成 `/index.php/post/2008/a%20sample%20post`。第三个规则适用。
- 调用 `$this->createUrl('post/read')` 产生 `/index.php/post/read`。请注意，没有规则适用。

33

也就是说，当使用 `createUrl` 生成 URL，路由 (route) 和传递给该方法的 GET 参数被用来决定哪些 URL 规则适用。如果规则中的每个相关参数可以在被传递给 `createUrl` 的 GET 参数中找到的，并且路由的规则也匹配路由参数，规则将用来生成网址。

如果传递到 `createUrl` 的 GET 参数比规则这种给出的参数要多，俺么其他参数将出现在查询字符串中。例如，如果我们调用 `$this->createUrl('post/read',array('id'=>100,'year'=>2008))`，我们将获得 `/index.php/post/100?year=2008`。为了使这些额外参数出现在路径信息的一部分，我们应该给规则附加 `/*`。因此，使用规则 `post/<id:\d+>/*`，我们可以获取网址 `/index.php/post/100/year/2008`。

正如我们提到的，URL 规则的另一个用途是解析请求网址。当然，这是 URL 生成的一个逆过程。例如，当用户请求 `/index.php/post/100`，上面例子的第二个规则将适用来解析路由 `post/read`

和 GET 参数 `array('id'=>100)`（可通过 `$_GET` 获得）。

使用 URL rule 会降低系统性能，这是因为解析一个 URL 请求的时候，`CUrlManager` 会尝试用每一个 rule 进行匹配，直到有一个符合，ruler 的数量越多，性能的开销越大，因此高流量的网站应该尽量减少 URL ruler 的应用。

参数化路由

从 1.0.5 版本开始，我们可以在 ruler 中对路由路径参数化。这允许一条 ruler 可以应用到多条相匹配路由。这也有利于减少应用 ruler 的数量，提高整体性能。

我们使用下面的例子来解释如何使用命名参数对路由进行参数化 `array(`

```
34     '<_c:(post|comment)>/<id:\d+>/<_a:(create|update|delete)>' => '<_c>/<_a>',
35     '<_c:(post|comment)>/<id:\d+>' => '<_c>/read',
36     '<_c:(post|comment)>>s' => '<_c>/list',
37 )
```

复制代码

38 在上面，我们在 ruler 的路由部分使用了两个命名参数 `_c` 和 `_a`，前者匹配一个控制器 id 是 `post` 或者 `comment`，后者匹配一个动作 id 是 `create`、`update` 或者 `delete`。你可以把参数的名字起的尽可能长，以避免和出现在 GET 中的参数冲突。

使用上面的规则 URL `/index.php/post/123/create`，会被解释为 `post/create`，并具有 GET 参数 `id=123`。给出路由 `comment /list` 和 GET 参数 `page=2`，我们可以创建 URL `/index.php/comments?page=2`

参数化主机名

从 1.0.11 版本开始，分析和创建 URLs 的 rule 中可以支持主机名。一方面，可以读取路径中的主机名作为 GET 变量，例如：地址 <http://admin.example.com/en/profile>，可以被解析为 GET 变量 `user=admin` 和 `lang=en`。另一方面，开主机名的 ruler 可以创建参数化主机名的 URLs

为了使用参数化的主机名，我们可以用主机信息声明 URL rules，例如：`array(`

```
39     'http://<user:\w+>.example.com/<lang:\w+>/profile' => 'user/profile',
40 )
```

复制代码

41 上面的例子表示，主机名的第一部分被当做 `user` 参数，而第二部分被当做 `lang` 参数，符合的路由是 `user/profile` 路由。

请注意 当 URL 是由参数化的主机来创建时，`CUrlManager::showScriptName` 不会起作用。

同样需要注意如果应用是在 WEB 根目录的子文件夹底下，参数化主机名的 ruler 不能包含子文件夹，例如，如果应用的路径是在 <http://www.example.com/sandbox/blog> 下面，我们可以仍然使用和上面描述相同的 URL ruler: `sandbox/blog` 而不需要子文件夹

隐藏 `index.php`

还有一点，我们可以进一步清理我们的网址，即在 URL 中藏匿 `index.php` 入口脚本。这就要求我们配置 Web 服务器，以及 `urlManager` 应用程序元件。

我们首先需要配置 Web 服务器，这样一个 URL 没有入口脚本仍然可以由入口脚本来处理。如果是 Apache HTTP server，可以通过打开网址重写引擎和指定一些重写规则。这两个操作可以在包含入口脚本的目录下的 `.htaccess` 文件里实现。下面是一个示例: `Options +FollowSymLinks`

```
42 IndexIgnore <em>/</em>
43 RewriteEngine on
44 # if a directory or a file exists, use it directly
45 RewriteCond %{REQUEST_FILENAME} !-f
46 RewriteCond %{REQUEST_FILENAME} !-d
47 # otherwise forward it to index.php
48 RewriteRule . index.php
```

复制代码

然后，我们设定 `urlManager` 元件的 `showScriptName` 属性为 `false`。

现在，如果我们调用 `$this->createUrl('post/read',array('id'=>100))`，我们将获取网址 `/post/100`。更重要的是，这个 URL 可以被我们的 Web 应用程序正确解析。

Faking URL Suffix(伪造 URL 后缀)

我们还可以添加一些网址的后缀。例如，我们可以用 `/post/100.html` 来替代 `/post/100`。这使得它看起来更像一个静态网页 URL。为了做到这一点，只需配置 `urlManager` 组件的 `urlSuffix` 属性为你所喜欢的后缀。