Yii 快速入门教程

（整理：编程之恋）

Ⅰ、基本概念

一、入口文件

入口文件内容：一般格式如下：

```
<?php

$yii=dirname(__FILE__).'/../../framework/yii.php';//Yii 框架位置
$config=dirname(__FILE__).'/protected/config/main.php';//当前应用程序的主配置文件位置

// 部署正式环境时，去掉下面这行
// defined('YII_DEBUG') or define('YII_DEBUG',true);//是否运行在调试模式下

require_once($yii);//包含 Yii 框架
Yii::createWebApplication($config)->run();//根据主配置文件建立应用实例，并运行。你可以在当前应用的任何位置通过 Yii::app()来访问这个实例。
```

二、主配置文件

保存位置：你的应用/protected/config/main.php

文件内容：一般格式如下：

```
<?php
return array(
    'basePath'=>dirname(__FILE__).DIRECTORY_SEPARATOR.'..', //当前应用根目录的绝对物理路径
    'name'=>'Yii Blog Demo', //当前应用的名称

    // 预载入 log（记录）应用组件，这表示该应用组件无论它们是否被访问都要被创建。该应用的参数配置在下面以“components”为关键字的数组中设置。
    'preload'=>array('log'), //log 为组件 ID

    // 自动载入的模型和组件类
    'import'=>array(
        'application.models.*', //载入“application/models/”文件夹下的所有模型类
        'application.components.*', //载入“application/components/”文件夹下的所有应用组件类
    ),

    'defaultController'=>'post', //设置默认控制器类

    // 当前应用的组件配置。更多可供配置的组件详见下面的“核心应用组件”
    'components'=>array(
        'user'=>array( //user（用户）组件配置，“user”为组件 ID
            // 可以使用基于 cookie 的认证
            'allowAutoLogin'=>true, //允许自动登录
        ),
        'cache'=>array( //缓存组件
```

```
                'class'=>'CMemCache', //缓存组件类
                'servers'=>array( //MemCache 缓存服务器配置
                    array('host'=>'server1', 'port'=>11211, 'weight'=>60), //缓存服务器 1
                    array('host'=>'server2', 'port'=>11211, 'weight'=>40), //缓存服务器 2
                ),
        ),
        'db'=>array( //db（数据库）组件配置，“db”为组件 ID
            'connectionString' => 'sqlite:protected/data/blog.db', //连接数据库的 DSN 字符串
            'tablePrefix' => 'tbl_', //数据表前缀
        ),
        // 如果要使用一个 MySQL 数据库，请取消下面的注释
        /*
        'db'=>array(
            'connectionString' => 'mysql:host=localhost;dbname=blog', //连接 mysql 数据库
            'emulatePrepare' => true,
            'username' => 'root', //MySQL 数据库用户名
            'password' => '', //MySQL 数据库用户密码
            'charset' => 'utf8', //MySQL 数据库编码
            'tablePrefix' => 'tbl_', //MySQL 数据库表前缀
        ),
        */
        'errorHandler'=>array(
            // 使用 SiteController 控制器类中的 actionError 方法显示错误
            'errorAction'=>'site/error', //遇到错误时，运行的操作。控制器名和方法名均小写，并用斜线“/”隔开
        ),
    //URL 路由管理器
        'urlManager'=>array(
            'urlFormat'=>'path', //URL 格式。共支持两种格式：'path'格式（如：/path/to/EntryScript.php/name1/value1/name2/value2...）和'get'格式（如：/path/to/EntryScript.php?name1=value1&name2=value2...）。当使用'path'格式时，需要设置如下的规则：
            'rules'=>array( //URL 规则。语法：<参数名:正则表达式>
                'post/<id:\d+>/<title:.*?>'=>'post/view', //将 post/12/helloword 指向 post/view?id=12&title=helloword
                'posts/<tag:.*?>'=>'post/index', //将 posts/hahahaha 指向 post/index?tag=hahahaha
                '<controller:\w+>/<action:\w+>'=>'<controller>/<action>',
            ),
        ),
        'log'=>array(  //记录
            'class'=>'CLogRouter', //处理记录信息的类
            'routes'=>array(
                array(
                    'class'=>'CFileLogRoute', //处理错误信息的类
                    'levels'=>'error, warning', //错误等级
                ),
                // 如要将错误记录消息在网页上显示，取消下面的注释即可
                /*
                array(
                    'class'=>'CWebLogRoute',
                ),
                */
            ),
```

```
        ),
    ), //应用组件配置结束


    // 使用 Yii::app()->params['参数名']可以访问应用层的参数
    'params'=>require(dirname(__FILE__).'/params.php'),
);
```

核心应用组件：
Yii 预定义了一系列核心应用组件，提供常见 Web 应用中所用的功能。例如， request 组件用于解析用户请求并提供例如 URL，cookie 等信息。通过配置这些核心组件的属性，我们可以几乎任意的修改 Yii 的默认行为。

下面我们列出了由 CWebApplication 预定义的核心组件。
assetManager: CAssetManager - 管理私有资源文件的发布。
authManager: CAuthManager - 管理基于角色的访问控制 (RBAC).
cache: CCache - 提供数据缓存功能。注意，你必须指定实际的类（例如 CMemCache, CDbCache）。否则，当你访问此组件时将返回 NULL。
clientScript: CClientScript - 管理客户端脚本 (javascripts 和 CSS).
coreMessages: CPhpMessageSource - 提供 Yii 框架用到的核心信息的翻译。
db: CDbConnection - 提供数据库连接。注意，使用此组件你必须配置其 connectionString 属性。
errorHandler: CErrorHandler - 处理未捕获的 PHP 错误和异常。
format: CFormatter - 格式化数值显示。此功能从版本 1.1.0 起开始提供。
messages: CPhpMessageSource - 提供 Yii 应用中使用的信息翻译。
request: CHttpRequest - 提供关于用户请求的信息。
securityManager: CSecurityManager - 提供安全相关的服务，例如散列，加密。
session: CHttpSession - 提供 session 相关的功能。
statePersister: CStatePersister - 提供全局状态持久方法。
urlManager: CUrlManager - 提供 URL 解析和创建相关功能
user: CWebUser - 提供当前用户的识别信息。
themeManager: CThemeManager - 管理主题。

要访问一个应用组件，使用 Yii::app()->组件的 ID

三、控制器（Controller）

控制器 是 CController 类的子类的实例。它在当用户请求时由应用创建。当一个控制器运行时，它执行所请求的动作（控制器类方法），动作通常会引入所必要的模型并渲染相应的视图。 动作，就是一个名字以 action 开头的控制器类方法（action+大写首字母的动作名）。
控制器类文件保存位置 protected/controllers/
控制器和动作以 ID 识别。
控制器 ID 是一种 '父目录/子目录/控制器名' 的格式，对应相应的控制器类文件 protected/controllers/父目录/子目录/大写首字母的控制器名 Controller.php；
动作 ID 是除去 action 前缀的动作方法名。

1、路由

用户以路由的形式请求特定的控制器和动作。路由是由控制器 ID 和动作 ID 连接起来的，两者以斜线分割。
例如，路由 post/edit 代表 PostController 及其 edit 动作。默认情况下，URL http://hostname/index.php?r=post/edit 即

请求此控制器和动作。
注意: 默认情况下，路由是大小写敏感的。可以通过设置应用配置中的 CUrlManager::caseSensitive 为 false 使路由对大小写不敏感。当在大小写不敏感模式中时， 要确保你遵循了相应的规则约定，即：包含控制器类文件的目录名小写，且 控制器映射 和 动作映射 中使用的键为小写。
路由的格式：控制器 ID/动作 ID 或 模块 ID/控制器 ID/动作 ID（如果是嵌套模块，模块 ID 就是 父模块 ID/子模块 ID）

2、控制器实例化

应用将使用如下规则确定控制器的类以及类文件的位置：
1、如果指定了 CWebApplication::catchAllRequest，控制器将基于此属性创建，而用户指定的控制器 ID 将被忽略。这通常用于将应用设置为维护状态并显示一个静态提示页面。
2、如果在 CWebApplication::controllerMap 中找到了 ID，相应的控制器配置将被用于创建控制器实例。
3、如果 ID 为 'path/to/xyz' 的格式，控制器类的名字将判断为 XyzController，相应的类文件则为 protected/controllers/path/to/XyzController.php。如果类文件不存在，将触发一个 404 CHttpException 异常。
在使用了模块的情况下，应用将检查此 ID 是否代表一个模块中的控制器。如果是的话，模块实例将被首先创建，然后创建模块中的控制器实例。

3、动作（action）

动作 就是被定义为一个以 action 单词作为前缀命名的方法。而更高级的方式是定义一个动作类并让控制器在收到请求时将其实例化。这使得动作可以被复用，提高了可复用度。
1、定义一个动作类，基本格式如下：

```
class UpdateAction extends CAction
{
    public function run()
    {
        // place the action logic here
    }
}
```

2、使用动作类：为了让控制器注意到这个动作，我们要用如下方式覆盖控制器类的 actions() 方法：

```
class PostController extends CController
{
    public function actions()
    {
        return array(
            'edit'=>'application.controllers.post.UpdateAction', //使用“应用程序文件夹/controllers/post/UpdateAction.php”文件中的类来处理 edit 动作
        );
    }
}
```

如上所示，我们使用了路径别名“application.controllers.post.UpdateAction”指定动作类文件为“protected/controllers/post/UpdateAction.php”。
通过编写基于类的动作，我们可以将应用组织为模块的风格。例如，如下目录结构可用于组织控制器相关代码：

```
protected/
    controllers/
        PostController.php
        UserController.php
        post/
            CreateAction.php
```

```
        ReadAction.php
        UpdateAction.php
    user/
        CreateAction.php
        ListAction.php
        ProfileAction.php
        UpdateAction.php
```

四、过滤器（filter）

过滤器是一段代码，可被配置在控制器动作执行之前或之后执行。
一个动作可以有多个过滤器。如有多个过滤器，则按照它们出现在过滤器列表中的顺序依次执行。过滤器可以阻止动作及后面其他过滤器的执行。
过滤器可以定义为一个控制器类的方法。过滤器方法名必须以 filter 开头。例如，现有的 filterAccessControl 方法定义了一个名为 accessControl 的过滤器。过滤器方法必须为如下结构：

```
public function filterAccessControl($filterChain)
{
    // 调用 $filterChain->run() 以继续后续过滤器与动作的执行。
}
```

$filterChain (过滤器链)是一个 CFilterChain 的实例，代表与所请求动作相关的过滤器列表。在过滤器方法中，我们可以调用 $filterChain->run() 以继续执行后续过滤器和动作。
如 动作 一样，过滤器也可以是一个对象，它是 CFilter 或其子类的实例。如下代码定义了一个新的过滤器类：

```
class PerformanceFilter extends CFilter
{
    protected function preFilter($filterChain)
    {
        // 动作被执行之前应用的逻辑
        return true; // 如果动作不应被执行，此处返回 false
    }

    protected function postFilter($filterChain)
    {
        // 动作执行之后应用的逻辑
    }
}
```

要对动作应用过滤器，我们需要覆盖 CController::filters() 方法。此方法应返回一个过滤器配置数组。例如：

```
class PostController extends CController
{
    ......
    public function filters()
    {
        return array(
            'postOnly + edit, create', //将 postOnly 过滤器应用于 edit 和 create 动作（这是基于方法的过滤器）
            array( //使用了数组来配置过滤器
                'application.filters.PerformanceFilter - edit, create', //将 application.filters.PerformanceFilter 过滤器应用于除了 edit 和 create 之外的所有动作（这是基于对象的过滤器）
                'unit'=>'second', //初始化过滤器对象中的 unit 属性值为 second
            ),
```

```
            );
        }
}
```

上述代码指定了两个过滤器： postOnly 和 PerformanceFilter。 postOnly 过滤器是基于方法的（相应的过滤器方法已在 CController 中定义）；而 performanceFilter 过滤器是基于对象的。路径别名 application.filters.PerformanceFilter 指定过滤器类文件是 protected/filters/PerformanceFilter。我们使用一个数组配置 PerformanceFilter ，这样它就可被用于初始化过滤器对象的属性值。此处 PerformanceFilter 的 unit 属性值将被初始为 second。

使用加减号，我们可指定哪些动作应该或不应该应用过滤器。上述代码中， postOnly 应只被应用于 edit 和 create 动作，而 PerformanceFilter 应被应用于 除了 edit 和 create 之外的动作。如果过滤器配置中没有使用加减号，则此过滤器将被应用于所有动作。

五、模型（Model）

模型是 CModel 或其子类的实例。模型用于保持数据以及与其相关的业务逻辑。
模型是单独的数据对象。它可以是数据表中的一行，或者一个用户输入的表单。
数据对象的每个字段对应模型中的一个属性。每个属性有一个标签（label），并且可以通过一系列规则进行验证。
Yii 实现了两种类型的模型：表单模型和 Active Record。二者均继承于相同的基类 CModel。
表单模型是 CFormModel 的实例。表单模型用于保持从用户的输入获取的数据。这些数据经常被获取，使用，然后丢弃。例如，在一个登录页面中，我们可以使用表单模型用于表示由最终用户提供的用户名和密码信息。
Active Record (AR) 是一种用于通过面向对象的风格抽象化数据库访问的设计模式。每个 AR 对象是一个 CActiveRecord 或其子类的实例。代表数据表中的一行。行中的字段对应 AR 对象中的属性。

六、视图

视图是一个包含了主要的用户交互元素的 PHP 脚本.
视图有一个名字，当渲染(render)时，名字会被用于识别视图脚本文件。视图的名称与其视图脚本名称是一样的。例如:视图 edit 的名称出自一个名为 edit.php 的脚本文件。要渲染时，需通过传递视图的名称调用 CController::render()。这个方法将在“protected/views/控制器 ID”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->属性名”的方式获取控制器的任何属性。
我们也可以用以下 推送 的方式传递数据到视图里:

```
$this->render('edit', array(
    'var1'=>$value1,
    'var2'=>$value2,
));
```

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。其产生的结果是,在视图脚本里,我们可以直接访问变量 $var1 和 $var2。

1、布局

布局是一种用来修饰视图的特殊的视图文件。它通常包含了用户界面中通用的一部分视图。例如:布局可以包含 header 和 footer 的部分,然后把内容嵌入其间。

```
.....header here.....
<?php echo $content; ?>
```

.....footer here.....

其中的 $content 则储存了内容视图的渲染结果。
当使用 render()时,布局被隐式应用。视图脚本 protected/views/layouts/main.php 是默认的布局文件。这可以通过改变 CWebApplication::layout 进行自定义。要渲染一个不带布局的视图，则需调用 renderPartial() 。

2、小物件

小物件是 CWidget 或其子类的实例。它是一个主要用于表现数据的组件。小物件通常内嵌于一个视图来产生一些复杂而独立的用户界面。例如,一个日历小物件可用于渲染一个复杂的日历界面。小物件使用户界面更加可复用。
我们可以按如下视图脚本来使用一个小物件:

```
<?php $this->beginWidget('小物件类的路径别名'[,'包含属性初始化值的数组']); ?>
...可能会由小物件获取的内容主体...
<?php $this->endWidget(); ?>
```

或者

```
<?php $this->widget('小物件类的路径别名'[,'包含属性初始化值的数组']); ?>
```

后者用于不需要任何 body 内容的组件。
小物件可通过配置来定制它的表现。这是通过调用 CBaseController::beginWidget 或 CBaseController::widget 设置其初始化属性值来完成的。
我们通过传递一个携带这些属性初始化值的数组来实现，该数组的键是属性的名称,而数组的值则是小物件属性所对应的值。如下所示 :

```
<?php
$this->widget('CMaskedTextField',array(
    'mask'=>'99/99/9999'
));
?>
```

继承 CWidget 并覆盖其 init()和 run()方法,可以定义一个新的小物件:

```
class MyWidget extends CWidget
{
    public function init()
    {
        // 此方法会被 CController::beginWidget() 调用
    }
    public function run()
    {
        // 此方法会被 CController::endWidget() 调用
    }
}
```

小物件可以像一个控制器一样拥有它自己的视图。
默认情况下,小物件的视图文件位于包含了小物件类文件目录的 views 子目录之下（protected/components/views）。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,小物件的视图没有布局文件支持。另外，小物件视图中的$this 指向小物件实例而不是控制器实例。

3、系统视图

系统视图的渲染通常用于展示 Yii 的错误和日志信息。
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 抛出一个 404 错误,那么 error404 就会被显示。

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进行自定义。

七、组件

Yii 应用建立于组件之上。组件是 CComponent 或其子类的实例。使用组件主要涉及访问它的属性以及触发或处理它的时间。基类 CComponent 指定了如何定义属性和事件。

1、组件属性

组件的属性就像对象的公共成员变量。它是可读写的。
要定义一个组件属性，我们只需在组件类中定义一个公共成员变量即可。
更灵活的方式是定义其 getter 和 setter 方法，例如：

```
public function getTextWidth()          // 获取 textWidth 属性
{
    return $this->_textWidth;
}
public function setTextWidth($value)      // 设置 TextWidth 属性
{
    $this->_textWidth=$value;
}
```

上述代码定义了一个可写的属性名为 textWidth（名字是大小写不敏感的）。当读取属性时，getTextWidth() 就会被调用，其返回值则成为属性值；相似的，当写入属性时，setTextWidth() 被调用。如果 setter 方法没有定义，则属性将是只读的，如果对其写入则会抛出一个异常。使用 getter 和 setter 方法定义一个属性有一个好处：即当读取或写入属性时，可以执行额外的逻辑（例如，执行验证，触发事件）。
注意: 通过 getter/setter 定义的属性和类成员变量之间有一个细微的差异：属性的名字是大小写不敏感的， 而 类成员变量 是大小写敏感的。

2、组件事件

组件事件是一些特殊的属性，它们使用一些称作 事件句柄（event handlers）的方法作为其值。分配一个方法到一个事件将会引起方法在事件被唤起处自动被调用。因此，一个组件的行为可能会被一种在部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)
{
    $this->raiseEvent('onClicked', $event);
}
```

这里作为事件参数的 $event 是 CEvent 或其子类的实例。
我们可以分配一个方法到此事件，如下所示:

```
$component->onClicked=$callback;
```

这里的 $callback 指向了一个有效的 PHP 回调。它可以是一个全局函数也可以是类中的一个方法。如果是后者，它必须以一个数组的方式提供: array($object,'methodName')。
事件句柄的结构如下：

```
function 方法名($event)
{
```

```
    ......
}
```
这里的 $event 即描述事件的参数（它来源于 raiseEvent() 调用）。$event 参数是 CEvent 或其子类的实例。至少，它包含了关于谁触发了此事件的信息。
事件句柄也可以是一个 PHP 5.3 以后支持的匿名函数。例如：
```
$component->onClicked=function($event) {
    ......
}
```
如果我们现在调用 onClicked()，onClicked 事件将被触发（在 onClicked() 中），附属的事件句柄将被自动调用。
一个事件可以绑定多个句柄。当事件触发时，这些句柄将被按照它们绑定到事件时的顺序依次执行。如果句柄决定组织后续句柄被执行，它会设置 $event->handled 为 true。

3、组件行为

组件已添加了对 mixin 的支持，并可以绑定一个或多个行为。 行为是一个对象，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)。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。 大多数行为可以继承自 CBehavior 。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 CActiveRecordBehavior 继承。
要使用一个行为，它必须首先通过调用此行为的 attach() 方法绑定到一个组件。然后我们就可以通过组件调用此行为方法：
```
// $name 在组件中实现了对行为的唯一识别
$component->attachBehavior($name,$behavior);
// test() 是行为中的方法。
$component->test();
```
已绑定的行为可以像一个组件中的普通属性一样访问。例如，如果一个名为 tree 的行为绑定到了一个组件，我们就可以通过如下代码获得指向此行为的引用。
```
$behavior=$component->tree;
// 等于下行代码：
// $behavior=$component->asa('tree');
```
行为是可以被临时禁止的,此时它的方法就会在组件中失效。例如:
```
$component->disableBehavior($name);
// 下面的代码将抛出一个异常
$component->test();
$component->enableBehavior($name);
// 现在就可以使用了
$component->test();
```
两个同名行为绑定到同一个组件下是有可能的。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。
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。

八、模块

模块是一个独立的软件单元，它包含 模型, 视图, 控制器 和其他支持的组件。在许多方面上，模块看起来像一个 应用。主要的区别就是模块不能单独部署，它必须存在于一个应用里。用户可以像他们访问普通应用的控制器那样访问

模块中的控制器。
模块在一些场景里很有用。对大型应用来说，我们可能需要把它划分为几个模块，每个模块可以单独维护和部署。一些通用的功能，例如用户管理，评论管理，可以以模块的形式开发，这样他们就可以容易地在以后的项目中被复用。

1、创建模块

模块组织在一个目录中，目录名即为模块的唯一 ID。模块目录的结构跟 应用基础目录 很相似。下面列出了一个 fourm 的模块的典型的目录结构：

```
forum/                          模块文件夹
    ForumModule.php             模块类文件
    components/                 包含可复用的用户组件
        views/                  包含小物件的视图文件
    controllers/                包含控制器类文件
        DefaultController.php   默认的控制器类文件
    extensions/                 包含第三方扩展
    models/                     包含模型类文件
    views/                      包含控制器视图和布局文件
        layouts/                包含布局文件
        default/                包含 DefaultController 的视图文件
            index.php           首页视图文件
```

模块必须有一个继承自 CWebModule 的模块类。类的名字通过表达式 ucfirst($id).'Module' 确定，其中的 $id 代表模块的 ID (或者说模块的目录名字)。模块类是存储模块代码间可共享信息的中心位置。例如，我们可以使用 CWebModule::params 存储模块参数，使用 CWebModule::components 分享模块级的 应用组件。

2、使用模块

要使用模块，首先将模块目录放在 应用基础目录 的 modules 文件夹中。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，我们可以使用如下应用配置:

```
return array(
    ......
    'modules'=>array('forum',...),
    ......
);
```

模块也可以在配置时带有初始属性值。做法和配置 应用组件 很类似。例如， forum 模块可以在其模块类中有一个名为 postPerPage 的属性，它可以在 应用配置 中配置如下:

```
return array(
    ......
    'modules'=>array(
        'forum'=>array(
            'postPerPage'=>20,
        ),
    ),
    ......
);
```

模块的实例可通过当前活动控制器的 module 属性访问。在模块实例中，我们可以访问在模块级中共享的信息。例如，为访问上面的 postPerPage 信息，我们可使用如下表达式：

```
$postPerPage=Yii::app()->controller->module->postPerPage;
// 如如$this 引用的是控制器实例，则可以使用下行语句
```

// $postPerPage=$this->module->postPerPage;
模块中的控制器动作可以通过路由“模块 ID/控制器 ID/动作 ID”或“模块 ID/存放控制器类文件的子目录名/控制器 ID/动作 ID”访问。例如，假设上面的 forum 模块有一个名为 PostController 的控制器，我们就可以通过路由 forum/post/create 访问此控制器中的 create 动作。此路由对应的 URL 即 http://www.example.com/index.php?r=forum/post/create。

3、嵌套的模块

模块可以无限级嵌套。这就是说，一个模块可以包含另一个模块，而这另一个模块又可以包含其他模块。我们称前者为 父模块 ，后者为 子模块。子模块必须定义在其父模块的 modules 属性中，就像我们前面在应用配置中定义模块一样。
要访问子模块中的控制器动作，我们应使用路由 父模块 ID/子模块 ID/控制器 ID/动作 ID。

九、路径别名

Yii 中广泛的使用了路径别名。路径别名关联于一个目录或文件的路径。它以点号语法指定，类似于广泛使用的名字空间（namespace）格式：
RootAlias.path.to.target
其中的 RootAlias 是某个现存目录的别名，通过调用 YiiBase::setPathOfAlias(), 我们可以定义新的路径别名。为方便起见，Yii 预定义了以下几个根别名：
system: 表示 Yii 框架目录；
zii: 表示 Zii 库 目录；
application: 表示应用的 基础目录；
webroot: 表示 入口脚本 文件所在的目录。
ext: 表示包含了所有第三方 扩展 的目录。
额外的，如果应用使用了 模块, （Yii） 也为每个模块 ID 定义了根别名，指向相应模块的跟目录。
通过使用 YiiBase::getPathOfAlias(), 别名可以被翻译为其相应的路径。
使用别名可以很方便的导入类的定义。例如，如果我们想包含 CController 类的定义，我们可以调用如下代码
Yii::import('system.web.CController');
import 方法跟 include 和 require 不同，它更加高效。导入（import）的类定义并不会真正被包含进来，直到它第一次被引用。多次导入同样的名字空间也会比 include_once 和 require_once 快得多。
我们还可以使用如下语法导入整个目录，这样此目录下的类文件就会在需要时被自动包含。
Yii::import('system.web.*');
除 import 外， 别名还在其他许多地方指向类。例如，路径别名可以传递给 Yii::createComponent() 以创建相应类的实例。即使类文件在之前从未被包含。
不要将路径别名和名字空间混淆了，名字空间是指对一些类名的一个逻辑组合，这样它们就可以相互区分开，即使有相同的名字。而路径别名是用于指向一个类文件或目录。路径别名与名字空间并不冲突。

十、开发规范

下面我们讲解 Yii 编程中推荐的开发规范。为简单起见，我们假设 WebRoot 是 Yii 应用安装的目录。

1、URL

默认情况下，Yii 识别如下格式的 URL：
http://hostname/index.php?r=ControllerID/ActionID

r 变量意为 路由（route），它可以被 Yii 解析为 控制器和动作。如果 ActionID 被省略，控制器将使用默认的动作（在 CController::defaultAction 中定义）；如果 ControllerID 也被省略（或者 r 变量不存在），应用将使用默认的控制器（在 CWebApplication::defaultController 中定义）。
通过 CUrlManager 的帮助，可以创建更加可识别，更加 SEO 友好的 URL，例如 http://hostname/ControllerID/ActionID.html。

2、代码

Yii 推荐命名变量、函数和类时使用驼峰风格，即每个单词的首字母大写并连在一起，中间无空格。变量名和函数名应该使它们的第一个单词全部小写，以使其区别于类名。对私有类成员变量来说，我们推荐以下划线作为其名字前缀（例如： $_actionList）。
一个针对控制器类名的特殊规则是它们必须以单词 Controller 结尾。那么控制器 ID 就是类名的首字母小写并去掉单词 Controller。例如，PageController 类的 ID 就是 page。这个规则使应用更加安全。它还使控制器相关的 URL 更加简单(例如 /index.php?r=page/index 而不是 /index.php?r=PageController/index)。

3、配置

配置是一个键值对数组。每个键代表了所配置的对象中的属性名，每个值则为相应属性的初始值。
类中任何可写的属性都可以被配置。如果没有配置，属性将使用它们的默认值。当配置一个属性时，最好阅读相应文档以保证初始值正确。

4、文件

命名和使用文件的规范取决于它们的类型。
类文件应以它们包含的公有类命名。例如，CController 类位于 CController.php 文件中。公有类是可以被任何其他类使用的类。每个类文件应包含最多一个公有类。私有类（只能被一个公有类使用的类）可以放在使用此类的公有类所在的文件中。
视图文件应以视图的名字命名。例如， index 视图位于 index.php 文件中。视图文件是一个 PHP 脚本文件，它包含了用于呈现内容的 HTML 和 PHP 代码。
配置文件可以任意命名。配置文件是一个 PHP 脚本，它的主要目的是返回一个体现配置的关联数组。

5、目录

Yii 假定了一系列默认的目录用于不同的场合。如果需要，每个目录都可以自定义。
WebRoot/protected: 这是 应用基础目录，是放置所有安全敏感的 PHP 脚本和数据文件的地方。Yii 有一个默认的 application 别名指向此目录。此目录及目录中的文件应该保护起来防止 Web 用户访问。它可以通过 CWebApplication::basePath 自定义。
WebRoot/protected/runtime: 此目录放置应用在运行时产生的私有临时文件。此目录必须对 Web 服务器进程可写。它可以通过 CApplication::runtimePath 自定义。
WebRoot/protected/extensions: 此目录放置所有第三方扩展。它可以通过 CApplication::extensionPath 自定义。
WebRoot/protected/modules: 此目录放置所有的应用 模块，每个模块使用一个子目录。
WebRoot/protected/controllers: 此目录放置所有控制器类文件。它可以通过 CWebApplication::controllerPath 自定义。
WebRoot/protected/views: 此目录放置所有试图文件，包含控制器视图，布局视图和系统视图。它可以通过 CWebApplication::viewPath 自定义。
WebRoot/protected/views/ControllerID: 此目录放置单个控制器类中使用的视图文件。此处的 ControllerID 是指控制器的 ID 。它可以通过 CController::viewPath 自定义。
WebRoot/protected/views/layouts: 此目录放置所有布局视图文件。它可以通过 CWebApplication::layoutPath 自定义。

WebRoot/protected/views/system: 此目录放置所有系统视图文件。系统视图文件是用于显示异常和错误的模板。它可以通过 CWebApplication::systemViewPath 自定义。
WebRoot/assets: 此目录放置公共资源文件。资源文件是可以被发布的，可由 Web 用户访问的私有文件。此目录必须对 Web 服务器进程可写。它可以通过 CAssetManager::basePath 自定义
WebRoot/themes: 此目录放置应用使用的不同的主题。每个子目录即一个主题，主题的名字即目录的名字。它可以通过 CThemeManager::basePath 自定义。

6、数据库

多数 Web 应用是由数据库驱动的。我们推荐在对表和列命名时使用如下命名规范。注意，这些规范并不是 Yii 所必须的。
㈠数据库表名和列名都使用小写命名。
㈡名字中的单词应使用下划线分割 (例如 product_order)。
㈢对于表名，你既可以使用单数也可以使用复数。但不要 同时使用两者。为简单起见，我们推荐使用单数名字。
㈣表名可以使用一个通用前缀，例如 tbl_ 。这样当应用所使用的表和另一个应用说使用的表共存于同一个数据库中时就特别有用。这两个应用的表可以通过使用不同的表前缀很容易地区别开。

Ⅱ、使用表单

在 Yii 中处理表单时，通常需要以下步骤：

1. 创建用于表现所要收集数据字段的模型类。
2. 创建一个控制器动作，响应表单提交。
3. 在视图脚本中创建与控制器动作相关的表单。

一、创建模型

在编写表单所需的 HTML 代码之前，我们应该先确定来自最终用户输入的数据的类型，以及这些数据应符合什么样的规则。模型类可用于记录这些信息。正如模型章节所定义的，模型是保存用户输入和验证这些输入的中心位置。
取决于使用用户所输入数据的方式，我们可以创建两种类型的模型。如果用户输入被收集、使用然后丢弃，我们应该创建一个表单模型; 如果用户的输入被收集后要保存到数据库，我们应使用一个 Active Record。两种类型的模型共享同样的基类 CModel ，它定义了表单所需的通用接口。

1、定义模型类

例如创建为一个表单模型：

```
class LoginForm extends CFormModel
{
    public $username;
    public $password;
    public $rememberMe=false;
}
```

LoginForm 中定义了三个属性: $username, $password 和 $rememberMe。他们用于保存用户输入的用户名和密码，还有用户是否想记住他的登录的选项。由于 $rememberMe 有一个默认的值 false，相应的选项在初始化显示在登录表单中时将是未勾选状态。
我们将这些成员变量称为特性（attributes）而不是属性（properties），以区别于普通的属性（properties）。特性（attribute）

是一个主要用于存储来自用户输入或数据库数据的属性（propertiy）。

2、声明验证规则

一旦用户提交了他的输入，模型被填充，我们就需要在使用前确保用户的输入是有效的。这是通过将用户的输入和一系列规则执行验证实现的。我们在 rules() 方法中指定这些验证规则，此方法应返回一个规则配置数组。

```
class LoginForm extends CFormModel
{
    public $username;
    public $password;
    public $rememberMe=false;

    private $_identity;

    public function rules()
    {
        return array(
            array('username, password', 'required'), //username 和 password 为必填项
            array('rememberMe', 'boolean'), //rememberMe 应该是一个布尔值
            array('password', 'authenticate'), //password 应被验证（authenticated）
        );
    }

    public function authenticate($attribute,$params)
    {
        $this->_identity=new UserIdentity($this->username,$this->password);
        if(!$this->_identity->authenticate())
            $this->addError('password','错误的用户名或密码。');
    }
}
```

rules() 返回的每个规则必须是以下格式：

array('AttributeList', 'Validator', 'on'=>'ScenarioList', ...附加选项)

其中：

AttributeList（特性列表）是需要通过此规则验证的特性列表字符串，每个特性名字由逗号分隔;

Validator（验证器） 指定要执行验证的种类;

on 参数是可选的，它指定此规则应被应用到的场景列表;

附加选项 是一个名值对数组，用于初始化相应验证器的属性值。

有三种方式可在验证规则中指定 Validator:

第一， Validator 可以是模型类中一个方法的名字，就像上面示例中的 authenticate 。验证方法必须是下面的结构：

```
/**
 * @param string 所要验证的特性的名字
 * @param array 验证规则中指定的选项
 */
public function 验证器名称($attribute,$params) { ... }
```

第二，Validator 可以是一个验证器类的名字，当此规则被应用时，一个验证器类的实例将被创建以执行实际验证。规则中的附加选项用于初始化实例的属性值。验证器类必须继承自 CValidator。

第三，Validator 可以是一个预定义的验证器类的别名。在上面的例子中，required 名字是 CRequiredValidator 的别名，

它用于确保所验证的特性值不为空。下面是预定义的验证器别名的完整列表：
boolean: CBooleanValidator 的别名，确保特性有一个 CBooleanValidator::trueValue 或 CBooleanValidator::falseValue 值。
captcha: CCaptchaValidator 的别名，确保特性值等于 CAPTCHA 中显示的验证码。
compare: CCompareValidator 的别名，确保特性等于另一个特性或常量。
email: CEmailValidator 的别名，确保特性是一个有效的 Email 地址。
default: CDefaultValueValidator 的别名，指定特性的默认值。
exist: CExistValidator 的别名，确保特性值可以在指定表的列中可以找到。
file: CFileValidator 的别名，确保特性含有一个上传文件的名字。
filter: CFilterValidator 的别名，通过一个过滤器改变此特性。
in: CRangeValidator 的别名，确保数据在一个预先指定的值的范围之内。
length: CStringValidator 的别名，确保数据的长度在一个指定的范围之内。
match: CRegularExpressionValidator 的别名，确保数据可以匹配一个正则表达式。
numerical: CNumberValidator 的别名，确保数据是一个有效的数字。
required: CRequiredValidator 的别名，确保特性不为空。
type: CTypeValidator 的别名，确保特性是指定的数据类型。
unique: CUniqueValidator 的别名，确保数据在数据表的列中是唯一的。
url: CUrlValidator 的别名，确保数据是一个有效的 URL。

下面我们列出了几个只用这些预定义验证器的示例：
```
// 用户名为必填项
array('username', 'required'),
// 用户名必须在 3 到 12 个字符之间
array('username', 'length', 'min'=>3, 'max'=>12),
// 在注册场景中，密码 password 必须和 password2 一致。
array('password', 'compare', 'compareAttribute'=>'password2', 'on'=>'register'),
// 在登录场景中，密码必须接受验证。
array('password', 'authenticate', 'on'=>'login'),
```

3、安全的特性赋值

在一个类的实例被创建后，我们通常需要用最终用户提交的数据填充它的特性。这可以通过如下块赋值（massive assignment）方式轻松实现：
```
$model=new LoginForm;
if(isset($_POST['LoginForm']))
    $model->attributes=$_POST['LoginForm'];
```
最后的表达式被称作 块赋值（massive assignment），它将 $_POST['LoginForm'] 中的每一项复制到相应的模型特性中。这相当于如下赋值方法：
```
foreach($_POST['LoginForm'] as $name=>$value)
{
    if($name 是一个安全的特性)
        $model->$name=$value;
}
```
检测特性的安全非常重要，例如，如果我们以为一个表的主键是安全的而暴露了它，那么攻击者可能就获得了一个修改记录的主键的机会，从而篡改未授权给他的内容。
特性如果出现在相应场景的一个验证规则中，即被认为是安全的。例如：
```
array('username, password', 'required', 'on'=>'login, register'),
array('email', 'required', 'on'=>'register'),
```

如上所示， username 和 password 特性在 login 场景中是必填项。而 username, password 和 email 特性在 register 场景中是必填项。于是，如果我们在 login 场景中执行块赋值，就只有 username 和 password 会被块赋值。因为只有它们出现在 login 的验证规则中。另一方面，如果场景是 register ，这三个特性就都可以被块赋值。

```
// 在登录场景中
$model=new User('login');
if(isset($_POST['User']))
    $model->attributes=$_POST['User'];
// 在注册场景中
$model=new User('register');
if(isset($_POST['User']))
    $model->attributes=$_POST['User'];
```

那么为什么我们使用这样一种策略来检测特性是否安全呢？背后的基本原理就是：如果一个特性已经有了一个或多个可检测有效性的验证规则，那我们还担心什么呢？

请记住，验证规则是用于检查用户输入的数据，而不是检查我们在代码中生成的数据（例如时间戳，自动产生的主键）。因此，不要为那些不接受最终用户输入的特性添加验证规则。

有时候，我们想声明一个特性是安全的，即使我们没有为它指定任何规则。例如，一篇文章的内容可以接受用户的任何输入。我们可以使用特殊的 safe 规则实现此目的：

```
array('content', 'safe')
```

还有一个用于声明一个属性为不安全的 unsafe 规则：

```
array('permission', 'unsafe')
```

unsafe 规则并不常用，它是我们之前定义的安全特性的一个例外。

4、触发验证

一旦模型被用户提交的数据填充，我们就可以调用 CModel::validate() 触发数据验证进程。此方法返回一个指示验证是否成功的值。对 CActiveRecord 模型来说，验证也可以在我们调用其 CActiveRecord::save() 方法时自动触发。

我们可以通过设置 scenario 属性来设置场景属性，这样，相应场景的验证规则就会被应用。

验证是基于场景执行的。 scenario 属性指定了模型当前用于的场景和当前使用的验证规则集。例如，在 login 场景中，我们只想验证用户模型中的 username 和 password 输入；而在 register 场景中，我们需要验证更多的输入，例如 email, address, 等。下面的例子演示了如何在 register 场景中执行验证：

```
// 在注册场景中创建一个 User 模型。等价于：
// $model=new User;
// $model->scenario='register';
$model=new User('register'); //给模型类添加参数，该参数就是要触发的验证场景
// 将输入的值填充到模型
$model->attributes=$_POST['User'];
// 执行验证
if($model->validate())     // 如果输入有效
    ...
else
    ...
```

规则关联的场景可以通过规则中的 on 选项指定。如果 on 选项未设置，则此规则会应用于所有场景。例如：

```
public function rules()
{
    return array(
        array('username, password', 'required'),
        array('password_repeat', 'required', 'on'=>'register'),
        array('password', 'compare', 'on'=>'register'),
    );
```

}
第一个规则将应用于所有场景，而第二个将只会应用于 register 场景。

5、提取验证错误

验证完成后，任何可能产生的错误将被存储在模型对象中。我们可以通过调用 CModel::getErrors() 和 CModel::getError() 提取这些错误信息。这两个方法的不同点在于第一个方法将返回 所有 模型特性的错误信息，而第二个将只返回 第一个 错误信息。

6、特性标签

当设计表单时，我们通常需要为每个表单域显示一个标签。标签告诉用户他应该在此表单域中填写什么样的信息。虽然我们可以在视图中硬编码一个标签，但如果我们在相应的模型中指定（标签），则会更加灵活方便。
默认情况下 CModel 将简单的返回特性的名字作为其标签。这可以通过覆盖 attributeLabels() 方法自定义。正如在接下来的小节中我们将看到的，在模型中指定标签会使我们能够更快的创建出更强大的表单。

二、创建动作

有了模型，我们就可以开始编写用于操作此模型的逻辑了。我们将此逻辑放在一个控制器的动作中。对登录表单的例子来讲，相应的代码就是：

```
public function actionLogin()
{
    $model=new LoginForm;
    if(isset($_POST['LoginForm']))
    {
        // 收集用户输入的数据
        $model->attributes=$_POST['LoginForm'];
        // 验证用户输入，并在判断输入正确后重定向到前一页
        if($model->validate())
            $this->redirect(Yii::app()->user->returnUrl); //重定向到之前需要身份验证的页面 URL
    }
    // 显示登录表单
    $this->render('login',array('model'=>$model));
}
```

如上所示，我们首先创建了一个 LoginForm 模型示例；如果请求是一个 POST 请求（意味着这个登录表单被提交了），我们则使用提交的数据 $_POST['LoginForm'] 填充 $model ；然后我们验证此输入，如果验证成功，重定向用户浏览器到之前需要身份验证的页面。如果验证失败，或者此动作被初次访问，我们则渲染 login 视图，此视图的内容我们在下一节中讲解。
提示：在 login 动作中，我们使用 Yii::app()->user->returnUrl 获取之前需要身份验证的页面 URL。 组件 Yii::app()->user 是一种 CWebUser (或其子类) ，它表示用户会话信息（例如 用户名，状态）。
让我们特别留意一下 login 动作中出现的下面的 PHP 语句：

```
$model->attributes=$_POST['LoginForm'];
```

正如我们在 安全的特性赋值 中所讲的，这行代码使用用户提交的数据填充模型。 attributes 属性由 CModel 定义，它接受一个名值对数组并将其中的每个值赋给相应的模型特性。因此如果 $_POST['LoginForm'] 给了我们这样的一个数组，上面的那段代码也就等同于下面冗长的这段 (假设数组中存在所有所需的特性):

```
$model->username=$_POST['LoginForm']['username'];
```

```
$model->password=$_POST['LoginForm']['password'];
$model->rememberMe=$_POST['LoginForm']['rememberMe'];
```

注意: 为了使 $_POST['LoginForm'] 传递给我们的是一个数组而不是字符串， 我们需要在命名表单域时遵守一个规范。具体的，对应于模型类 C 中的特性 a 的表单域，我们将其命名为 C[a] 。例如，我们可使用 LoginForm[username] 命名 username 特性相应的表单域。

现在剩下的工作就是创建 login 视图了，它应该包含一个带有所需输入项的 HTML 表单。

三、创建表单

编写 login 视图是很简单的，我们以一个 form 标记开始，它的 action 属性应该是前面讲述的 login 动作的 URL。然后我们需要为 LoginForm 类中声明的属性插入标签和表单域。最后，我们插入一个可由用户点击提交此表单的提交按钮。所有这些都可以用纯 HTML 代码完成。

Yii 提供了几个助手（helper）类简化视图编写。例如，要创建一个文本输入域，我们可以调用 CHtml::textField()；要创建一个下拉列表，则调用 CHtml::dropDownList()。

例如， 如下代码将生成一个文本输入域，它可以在用户修改了其值时触发表单提交动作。

```
CHtml::textField($name,$value,array('submit'=>''));
```

下面，我们使用 CHtml 创建一个登录表单。我们假设变量 $model 是 LoginForm 的实例。

```
<div class="form">
<?php echo CHtml::beginForm(); ?>

    <?php echo CHtml::errorSummary($model); ?>

    <div class="row">
        <?php echo CHtml::activeLabel($model,'username'); ?>
        <?php echo CHtml::activeTextField($model,'username') ?>
    </div>

    <div class="row">
        <?php echo CHtml::activeLabel($model,'password'); ?>
        <?php echo CHtml::activePasswordField($model,'password') ?>
    </div>

    <div class="row rememberMe">
        <?php echo CHtml::activeCheckBox($model,'rememberMe'); ?>
        <?php echo CHtml::activeLabel($model,'rememberMe'); ?>
    </div>

    <div class="row submit">
        <?php echo CHtml::submitButton('Login'); ?>
    </div>

<?php echo CHtml::endForm(); ?>
</div><!-- form -->
```

上述代码生成了一个更加动态的表单，例如， CHtml::activeLabel() 生成一个与指定模型的特性相关的标签。如果此特性有一个输入错误，此标签的 CSS class 将变为 error，通过 CSS 样式改变了标签的外观。相似的，CHtml::activeTextField() 为指定模型的特性生成一个文本输入域，并会在错误发生时改变它的 CSS class。

我们还可以使用一个新的小物件 CActiveForm 以简化表单创建。这个小物件可同时提供客户端及服务器端无缝的、

一致的验证。使用 CActiveForm, 上面的代码可重写为：

```
<div class="form">
<?php $form=$this->beginWidget('CActiveForm'); ?>

    <?php echo $form->errorSummary($model); ?>

    <div class="row">
        <?php echo $form->label($model,'username'); ?>
        <?php echo $form->textField($model,'username') ?>
    </div>

    <div class="row">
        <?php echo $form->label($model,'password'); ?>
        <?php echo $form->passwordField($model,'password') ?>
    </div>

    <div class="row rememberMe">
        <?php echo $form->checkBox($model,'rememberMe'); ?>
        <?php echo $form->label($model,'rememberMe'); ?>
    </div>

    <div class="row submit">
        <?php echo CHtml::submitButton('Login'); ?>
    </div>

<?php $this->endWidget(); ?>
</div><!-- form -->
```

四、收集表格输入

有时我们想通过批量模式收集用户输入。也就是说，用户可以为多个模型实例输入信息并将它们一次性提交。我们将此称为 表格输入（tabular input） ，因为这些输入项通常以 HTML 表格的形式呈现。
要使用表格输入，我们首先需要创建或填充一个模型实例数组，取决于我们是想插入还是更新数据。然后我们从 $_POST 变量中提取用户输入的数据并将其赋值到每个模型。和单模型输入稍有不同的一点就是：我们要使用 $_POST['ModelClass'][$i] 提取输入的数据而不是使用 $_POST['ModelClass']。

```
public function actionBatchUpdate()
{
    // 假设每一项（item）是一个 'Item' 类的实例，
    // 提取要通过批量模式更新的项
    $items=$this->getItemsToUpdate();
    if(isset($_POST['Item']))
    {
        $valid=true;
        foreach($items as $i=>$item)
        {
            if(isset($_POST['Item'][$i]))
                $item->attributes=$_POST['Item'][$i];
            $valid=$valid && $item->validate();
```

```
            }
            if($valid)   // 如果所有项目有效
                // ...则在此处做一些操作
        }
        // 显示视图收集表格输入
        $this->render('batchUpdate',array('items'=>$items));
}
```

准备好了这个动作，我们需要继续 batchUpdate 视图的工作以在一个 HTML 表格中显示输入项。

```
<div class="form">
<?php echo CHtml::beginForm(); ?>
<table>
<tr><th>Name</th><th>Price</th><th>Count</th><th>Description</th></tr>
<?php foreach($items as $i=>$item): ?>
<tr>
<td><?php echo CHtml::activeTextField($item,"[$i]name"); ?></td>
<td><?php echo CHtml::activeTextField($item,"[$i]price"); ?></td>
<td><?php echo CHtml::activeTextField($item,"[$i]count"); ?></td>
<td><?php echo CHtml::activeTextArea($item,"[$i]description"); ?></td>
</tr>
<?php endforeach; ?>
</table>
<?php echo CHtml::submitButton('Save'); ?>
<?php echo CHtml::endForm(); ?>
</div><!-- form -->
```

注意，在上面的代码中我们使用了 "[$i]name" 而不是 "name" 作为调用 CHtml::activeTextField 时的第二个参数。
如果有任何验证错误，相应的输入项将会自动高亮显示，就像前面我们讲解的单模型输入一样。

III、数据库操作

Yii 提供了强大的数据库编程支持。Yii 数据访问对象(DAO)建立在 PHP 的数据对象(PDO)extension 上，使得在一个单一的统一的接口可以访问不同的数据库管理系统(DBMS)。使用 Yii 的 DAO 开发的应用程序可以很容易地切换使用不同的数据库管理系统，而不需要修改数据访问代码。Yii 的 Active Record（ AR ），实现了被广泛采用的对象关系映射(ORM)办法，进一步简化数据库编程。按照约定，一个类代表一个表，一个实例代表一行数据。Yii AR 消除了大部分用于处理 CRUD（创建，读取，更新和删除）数据操作的 sql 语句的重复任务。
尽管 Yii 的 DAO 和 AR 能够处理几乎所有数据库相关的任务，您仍然可以在 Yii application 中使用自己的数据库。事实上，Yii 框架精心设计使得可以与其他第三方库同时使用。

一、数据访问对象 (DAO)

Yii DAO 基于 PHP Data Objects (PDO) 构建。它是一个为众多流行的 DBMS 提供统一数据访问的扩展，这些 DBMS 包括 MySQL， PostgreSQL 等等。因此，要使用 Yii DAO，PDO 扩展和特定的 PDO 数据库驱动(例如 PDO_MYSQL)必须安装。
Yii DAO 主要包含如下四个类：
CDbConnection: 代表一个数据库连接。
CDbCommand: 代表一条通过数据库执行的 SQL 语句。
CDbDataReader: 代表一个只向前移动的，来自一个查询结果集中的行的流。

CDbTransaction: 代表一个数据库事务。

1、建立数据库连接

要建立一个数据库连接，创建一个 CDbConnection 实例并将其激活。连接到数据库需要一个数据源的名字（DSN）以指定连接信息。用户名和密码也可能会用到。当连接到数据库的过程中发生错误时 (例如，错误的 DSN 或无效的用户名/密码)，将会抛出一个异常。

```
$connection=new CDbConnection($dsn,$username,$password);
// 建立连接。你可以使用 try...catch 捕获可能抛出的异常
$connection->active=true;
.....
$connection->active=false;  // 关闭连接
```

DSN 的格式取决于所使用的 PDO 数据库驱动。总体来说， DSN 要含有 PDO 驱动的名字，跟上一个冒号，再跟上驱动特定的连接语法。可查阅 PDO 文档 获取更多信息。下面是一个常用 DSN 格式的列表。

* 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

由于 CDbConnection 继承自 CApplicationComponent，我们也可以将其作为一个 应用组件 使用。要这样做的话，请在 应用配置 中配置一个 db （或其他名字）应用组件如下：

```
array(
    ......
    'components'=>array(
        ......
        'db'=>array(
            'class'=>'CDbConnection',
            'connectionString'=>'mysql:host=localhost;dbname=testdb',
            'username'=>'root',
            'password'=>'password',
            'emulatePrepare'=>true,  // needed by some MySQL installations
        ),
    ),
)
```

然后我们就可以通过 Yii::app()->db 访问数据库连接了。它已经被自动激活了，除非我们特意配置了 CDbConnection::autoConnect 为 false。通过这种方式，这个单独的 DB 连接就可以在我们代码中的很多地方共享。

2、执行 SQL 语句

数据库连接建立后，SQL 语句就可以通过使用 CDbCommand 执行了。你可以通过使用指定的 SQL 语句作为参数调用 CDbConnection::createCommand() 创建一个 CDbCommand 实例。

```
$connection=Yii::app()->db;   // 假设你已经建立了一个 "db" 连接
// 如果没有，你可能需要显式建立一个连接：
// $connection=new CDbConnection($dsn,$username,$password);
$command=$connection->createCommand($sql);
// 如果需要，此 SQL 语句可通过如下方式修改：
// $command->text=$newSQL;
```

一条 SQL 语句会通过 CDbCommand 以如下两种方式被执行：

execute(): 执行一个无查询 （non-query）SQL 语句，例如 INSERT, UPDATE 和 DELETE 。如果成功，它将返回此执

行所影响的行数。
query(): 执行一条会返回若干行数据的 SQL 语句，例如 SELECT。如果成功，它将返回一个 CDbDataReader 实例，通过此实例可以遍历数据的结果行。为简便起见，（Yii）还实现了一系列 queryXXX() 方法以直接返回查询结果。
执行 SQL 语句时如果发生错误，将会抛出一个异常。

```
$rowCount=$command->execute();     // 执行无查询 SQL
$dataReader=$command->query();     // 执行一个 SQL 查询
$rows=$command->queryAll();        // 查询并返回结果中的所有行
$row=$command->queryRow();         // 查询并返回结果中的第一行
$column=$command->queryColumn(); // 查询并返回结果中的第一列
$value=$command->queryScalar();    // 查询并返回结果中第一行的第一个字段
```

3、获取查询结果

在 CDbCommand::query() 生成 CDbDataReader 实例之后，你可以通过重复调用 CDbDataReader::read() 获取结果中的行。你也可以在 PHP 的 foreach 语言结构中使用 CDbDataReader 一行行检索数据。

```
$dataReader=$command->query();
// 重复调用 read() 直到它返回 false
while(($row=$dataReader->read())!==false) { ... }
// 使用 foreach 遍历数据中的每一行
foreach($dataReader as $row) { ... }
// 一次性提取所有行到一个数组
$rows=$dataReader->readAll();
```

注意: 不同于 query(), 所有的 queryXXX()方法会直接返回数据。例如，queryRow()会返回代表查询结果第一行的一个数组。

4、使用事务

事务，在 Yii 中表现为 CDbTransaction 实例，可能会在下面的情况中启动：

* 开始事务.
* 一个个执行查询。任何对数据库的更新对外界不可见。
* 提交事务。如果事务成功，更新变为可见。
* 如果查询中的一个失败，整个事务回滚。

上述工作流可以通过如下代码实现：

```
$transaction=$connection->beginTransaction();
try
{
    $connection->createCommand($sql1)->execute();
    $connection->createCommand($sql2)->execute();
    //.... other SQL executions
    $transaction->commit();
}
catch(Exception $e) // 如果有一条查询失败，则会抛出异常
{
    $transaction->rollBack();
}
```

5、绑定参数

要避免 SQL 注入攻击 并提高重复执行的 SQL 语句的效率，你可以 "准备（prepare）"一条含有可选参数占位符的 SQL 语句，在参数绑定时，这些占位符将被替换为实际的参数。
参数占位符可以是命名的 (表现为一个唯一的标记) 或未命名的 (表现为一个问号)。调用 CDbCommand::bindParam() 或 CDbCommand::bindValue() 以使用实际参数替换这些占位符。这些参数不需要使用引号引起来：底层的数据库驱动会为你搞定这个。参数绑定必须在 SQL 语句执行之前完成。

```
// 一条带有两个占位符 ":username" 和 ":email"的 SQL
$sql="INSERT INTO tbl_user (username, email) VALUES(:username,:email)";
$command=$connection->createCommand($sql);
// 用实际的用户名替换占位符 ":username"
$command->bindParam(":username",$username,PDO::PARAM_STR);
// 用实际的 Email 替换占位符 ":email"
$command->bindParam(":email",$email,PDO::PARAM_STR);
$command->execute();
// 使用新的参数集插入另一行
$command->bindParam(":username",$username2,PDO::PARAM_STR);
$command->bindParam(":email",$email2,PDO::PARAM_STR);
$command->execute();
```

方法 bindParam() 和 bindValue() 非常相似。唯一的区别就是前者使用一个 PHP 变量绑定参数，而后者使用一个值。对于那些内存中的大数据块参数，处于性能的考虑，应优先使用前者。
6、绑定列
当获取查询结果时，你也可以使用 PHP 变量绑定列。这样在每次获取查询结果中的一行时就会自动使用最新的值填充。

```
$sql="SELECT username, email FROM tbl_user";
$dataReader=$connection->createCommand($sql)->query();
// 使用 $username 变量绑定第一列 (username)
$dataReader->bindColumn(1,$username);
// 使用 $email 变量绑定第二列 (email)
$dataReader->bindColumn(2,$email);
while($dataReader->read()!==false)
{
    // $username 和 $email 含有当前行中的 username 和 email
}
```

7、使用表前缀

要使用表前缀，配置 CDbConnection::tablePrefix 属性为所希望的表前缀。然后，在 SQL 语句中使用 {{TableName}} 代表表的名字，其中的 TableName 是指不带前缀的表名。例如，如果数据库含有一个名为 tbl_user 的表，而 tbl_ 被配置为表前缀，那我们就可以使用如下代码执行用户相关的查询：

```
$sql='SELECT * FROM {{user}}';
$users=$connection->createCommand($sql)->queryAll();
```

二、Active Record

虽然 Yii DAO 可以处理几乎任何数据库相关的任务，但很可能我们会花费 90% 的时间以编写一些执行普通 CRUD（create, read, update 和 delete）操作的 SQL 语句。而且我们的代码中混杂了 SQL 语句时也会变得难以维护。要解决这些问题，我们可以使用 Active Record。

Active Record(AR)是一个流行的对象-关系映射(ORM)技术。每个 AR 类代表一个数据表（或视图），数据表（或视图）的列在 AR 类中体现为类的属性，一个 AR 实例则表示表中的一行。常见的 CRUD 操作作为 AR 的方法实现。因此，我们可以以一种更加面向对象的方式访问数据。例如，我们可以使用以下代码向 tbl_post 表中插入一个新行。

```
$post=new Post;
$post->title='sample post';
$post->content='post body content';
$post->save();
```

注意: AR 并非要解决所有数据库相关的任务。它的最佳应用是模型化数据表为 PHP 结构和执行不包含复杂 SQL 语句的查询。 对于复杂查询的场景，应使用 Yii DAO。

1、建立数据库连接

AR 依靠一个数据库连接以执行数据库相关的操作。默认情况下，它假定 db 应用组件提供了所需的 CDbConnection 数据库连接实例。如下应用配置提供了一个例子：

```
return array(
    'components'=>array(
        'db'=>array(
            'class'=>'system.db.CDbConnection',
            'connectionString'=>'sqlite:path/to/dbfile',
            // 开启表结构缓存（schema caching）提高性能
            // 'schemaCachingDuration'=>3600,
        ),
    ),
);
```

提示: 由于 Active Record 依靠表的元数据（metadata）测定列的信息，读取元数据并解析需要时间。 如果你数据库的表结构很少改动，你应该通过配置 CDbConnection::schemaCachingDuration 属性的值为一个大于零的值开启表结构缓存。如果你想使用一个不是 db 的应用组件，或者如果你想使用 AR 处理多个数据库，你应该覆盖 CActiveRecord::getDbConnection()。CActiveRecord 类是所有 AR 类的基类。

提示: 通过 AR 使用多个数据库有两种方式。如果数据库的结构不同，你可以创建不同的 AR 基类实现不同的 getDbConnection()。否则，动态改变静态变量 CActiveRecord::db 是一个好主意。

2、定义 AR 类

要访问一个数据表，我们首先需要通过集成 CActiveRecord 定义一个 AR 类。每个 AR 类代表一个单独的数据表，一个 AR 实例则代表那个表中的一行。

如下例子演示了代表 tbl_post 表的 AR 类的最简代码：

```
class Post extends CActiveRecord
{
    public static function model($className=__CLASS__)
    {
        return parent::model($className);
    }

    public function tableName()
    {
        return 'tbl_post';
    }
}
```

提示: 由于 AR 类经常在多处被引用，我们可以导入包含 AR 类的整个目录，而不是一个个导入。 例如，如果我们

所有的 AR 类文件都在 protected/models 目录中，我们可以配置应用如下：

```
return array(
   'import'=>array(
        'application.models.*',
   ),
);
```

默认情况下，AR 类的名字和数据表的名字相同。如果不同，请覆盖 tableName()方法。
要使用表前缀功能，AR 类的 tableName() 方法可以通过如下方式覆盖

```
public function tableName()
{
    return '{{post}}';
}
```

这就是说，我们将没有前缀的表名用双大括号括起来，这样 Yii 就能自动添加前缀，从而返回完整的表名。
数据表行中列的值可以作为相应 AR 实例的属性访问。例如，如下代码设置了 title 列 (属性):

```
$post=new Post;
$post->title='a sample post';
```

虽然我们从未在 Post 类中显式定义属性 title，我们还是可以通过上述代码访问。这是因为 title 是 tbl_post 表中的一个列，CActiveRecord 通过 PHP 的__get()魔术方法使其成为一个可访问的属性。如果我们尝试以同样的方式访问一个不存在的列，将会抛出一个异常。
如果一个表没有主键，则必须在相应的 AR 类中通过如下方式覆盖 primaryKey() 方法指定哪一列或哪几列作为主键。

```
public function primaryKey()
{
    return 'id';
    // 对于复合主键，要返回一个类似如下的数组
    // return array('pk1', 'pk2');
}
```

3、创建记录

要向数据表中插入新行，我们要创建一个相应 AR 类的实例，设置其与表的列相关的属性，然后调用 save() 方法完成插入：

```
$post=new Post;
$post->title='sample post';
$post->content='content for the sample post';
$post->create_time=time();
$post->save();
```

如果表的主键是自增的，在插入完成后，AR 实例将包含一个更新的主键。在上面的例子中，id 属性将反映出新插入帖子的主键值，即使我们从未显式地改变它。
如果一个列在表结构中使用了静态默认值（例如一个字符串，一个数字）定义。则 AR 实例中相应的属性将在此实例创建时自动含有此默认值。改变此默认值的一个方式就是在 AR 类中显示定义此属性：

```
class Post extends CActiveRecord
{
    public $title='please enter a title';
    ......
}
$post=new Post;
echo $post->title;  // 这儿将显示: please enter a title
```

记录在保存（插入或更新）到数据库之前，其属性可以赋值为 CDbExpression 类型。例如，为保存一个由 MySQL 的 NOW() 函数返回的时间戳，我们可以使用如下代码：

```
$post=new Post;
$post->create_time=new CDbExpression('NOW()'); //CDbExpression 类就是计算数据库表达式的值
// $post->create_time='NOW()'; 不会起作用，因为
// 'NOW()' 将会被作为一个字符串处理。
$post->save();
```

提示: 由于 AR 允许我们无需写一大堆 SQL 语句就能执行数据库操作， 我们经常会想知道 AR 在背后到底执行了什么 SQL 语句。这可以通过开启 Yii 的日志功能实现。例如，我们在应用配置中开启了 CWebLogRoute，我们将会在每个网页的最后看到执行过的 SQL 语句。 我们也可以在应用配置中设置 CDbConnection::enableParamLogging 为 true，这样绑定在 SQL 语句中的参数值也会被记录。

4、读取记录

要读取数据表中的数据，我们可以通过如下方式调用 find 系列方法中的一种：

```
// 查找满足指定条件的结果中的第一行
$post=Post::model()->find($condition,$params);
// 查找具有指定主键值的那一行
$post=Post::model()->findByPk($postID,$condition,$params);
// 查找具有指定属性值的行
$post=Post::model()->findByAttributes($attributes,$condition,$params);
// 通过指定的 SQL 语句查找结果中的第一行
$post=Post::model()->findBySql($sql,$params);
```

如上所示，我们通过 Post::model() 调用 find 方法。请记住，静态方法 model() 是每个 AR 类所必须的。此方法返回在对象上下文中的一个用于访问类级别方法（类似于静态类方法的东西）的 AR 实例。

如果 find 方法找到了一个满足查询条件的行，它将返回一个 Post 实例，实例的属性含有数据表行中相应列的值。然后我们就可以像读取普通对象的属性那样读取载入的值，例如 echo $post->title;。

如果使用给定的查询条件在数据库中没有找到任何东西， find 方法将返回 null。

调用 find 时，我们使用 $condition 和 $params 指定查询条件。此处 $condition 可以是 SQL 语句中的 WHERE 字符串，$params 则是一个参数数组，其中的值应绑定到 $condation 中的占位符。例如：

```
// 查找 postID=10 的那一行
$post=Post::model()->find('postID=:postID', array(':postID'=>10));
```

注意: 在上面的例子中，我们可能需要在特定的 DBMS 中将 postID 列的引用进行转义。 例如，如果我们使用 PostgreSQL，我们必须将此表达式写为 "postID"=:postID，因为 PostgreSQL 在默认情况下对列名大小写不敏感。

我们也可以使用 $condition 指定更复杂的查询条件。不使用字符串，我们可以让 $condition 成为一个 CDbCriteria 的实例，它允许我们指定不限于 WHERE 的条件。例如：

```
$criteria=new CDbCriteria;
$criteria->select='title';  // 只选择 'title' 列
$criteria->condition='postID=:postID';
$criteria->params=array(':postID'=>10);
$post=Post::model()->find($criteria); // $params 不需要了
```

注意，当使用 CDbCriteria 作为查询条件时，$params 参数不再需要了，因为它可以在 CDbCriteria 中指定，就像上面那样。

一种替代 CDbCriteria 的方法是给 find 方法传递一个数组。数组的键和值各自对应标准（criterion）的属性名和值，上面的例子可以重写为如下：

```
$post=Post::model()->find(array(
    'select'=>'title',
    'condition'=>'postID=:postID',
    'params'=>array(':postID'=>10),
));
```

当一个查询条件是关于按指定的值匹配几个列时，我们可以使用 findByAttributes()。我们使 $attributes 参数是一个以

列名做索引的值的数组。在一些框架中，此任务可以通过调用类似 findByNameAndTitle 的方法实现。虽然此方法看起来很诱人， 但它常常引起混淆，冲突和比如列名大小写敏感的问题。
当有多行数据匹配指定的查询条件时，我们可以通过下面的 findAll 方法将他们全部带回。每个都有其各自的 find 方法，就像我们已经讲过的那样。

```
// 查找满足指定条件的所有行
$posts=Post::model()->findAll($condition,$params);
// 查找带有指定主键的所有行
$posts=Post::model()->findAllByPk($postIDs,$condition,$params);
// 查找带有指定属性值的所有行
$posts=Post::model()->findAllByAttributes($attributes,$condition,$params);
// 通过指定的 SQL 语句查找所有行
$posts=Post::model()->findAllBySql($sql,$params);
```

如果没有任何东西符合查询条件，findAll 将返回一个空数组。这跟 find 不同，find 会在没有找到什么东西时返回 null。
除了上面讲述的 find 和 findAll 方法，为了方便，（Yii）还提供了如下方法：

```
// 获取满足指定条件的行数
$n=Post::model()->count($condition,$params);
// 通过指定的 SQL 获取结果行数
$n=Post::model()->countBySql($sql,$params);
// 检查是否至少有一行复合指定的条件
$exists=Post::model()->exists($condition,$params);
```

5、更新记录

在 AR 实例填充了列的值之后，我们可以改变它们并把它们存回数据表。

```
$post=Post::model()->findByPk(10);
$post->title='new post title';
$post->save(); // 将更改保存到数据库
```

正如我们可以看到的，我们使用同样的 save() 方法执行插入和更新操作。如果一个 AR 实例是使用 new 操作符创建的，调用 save() 将会向数据表中插入一行新数据；如果 AR 实例是某个 find 或 findAll 方法的结果，调用 save() 将更新表中现有的行。实际上，我们是使用 CActiveRecord::isNewRecord 说明一个 AR 实例是不是新的。
直接更新数据表中的一行或多行而不首先载入也是可行的。 AR 提供了如下方便的类级别方法实现此目的：

```
// 更新符合指定条件的行
Post::model()->updateAll($attributes,$condition,$params);
// 更新符合指定条件和主键的行
Post::model()->updateByPk($pk,$attributes,$condition,$params);
// 更新满足指定条件的行的计数列
Post::model()->updateCounters($counters,$condition,$params);
```

在上面的代码中， $attributes 是一个含有以 列名作索引的列值的数组； $counters 是一个由列名索引的可增加的值的数组；$condition 和 $params 在前面的段落中已有描述。

6、删除记录

如果一个 AR 实例被一行数据填充,我们也可以删除此行数据。

```
$post=Post::model()->findByPk(10); // 假设有一个帖子，其 ID 为 10
$post->delete(); // 从数据表中删除此行
```

注意，删除之后， AR 实例仍然不变，但数据表中相应的行已经没了。
使用下面的类级别代码，可以无需首先加载行就可以删除它。

```
// 删除符合指定条件的行
Post::model()->deleteAll($condition,$params);
```

```
// 删除符合指定条件和主键的行
Post::model()->deleteByPk($pk,$condition,$params);
```

7、数据验证

当插入或更新一行时，我们常常需要检查列的值是否符合相应的规则。如果列的值是由最终用户提供的，这一点就更加重要。总体来说，我们永远不能相信任何来自客户端的数据。
当调用 save() 时， AR 会自动执行数据验证。验证是基于在 AR 类的 rules() 方法中指定的规则进行的。关于验证规则的更多详情，请参考 声明验证规则 一节。下面是保存记录时所需的典型的工作流。

```
if($post->save())
{
    // 数据有效且成功插入/更新
}
else
{
    // 数据无效，调用  getErrors() 提取错误信息
}
```

当要插入或更新的数据由最终用户在一个 HTML 表单中提交时，我们需要将其赋给相应的 AR 属性。我们可以通过类似如下的方式实现：

```
$post->title=$_POST['title'];
$post->content=$_POST['content'];
$post->save();
```

如果有很多列，我们可以看到一个用于这种复制的很长的列表。这可以通过使用如下所示的 attributes 属性简化操作。更多信息可以在 安全的特性赋值 一节和 创建动作 一节找到。

```
// 假设 $_POST['Post'] 是一个以列名索引列值为值的数组
$post->attributes=$_POST['Post'];
$post->save();
```

8、对比记录

类似于表记录，AR 实例由其主键值来识别。因此，要对比两个 AR 实例，假设它们属于相同的 AR 类， 我们只需要对比它们的主键值。然而,一个更简单的方式是调用 CActiveRecord::equals()。
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。相应地，主键值在 Yii 中表现为一个数组。primaryKey 属性给出了一个 AR 实例的主键值。

9、自定义

CActiveRecord 提供了几个占位符方法，它们可以在子类中被覆盖以自定义其工作流。
beforeValidate 和 afterValidate：这两个将在验证数据有效性之前和之后被调用。
beforeSave 和 afterSave: 这两个将在保存 AR 实例之前和之后被调用。
beforeDelete 和 afterDelete: 这两个将在一个 AR 实例被删除之前和之后被调用。
afterConstruct: 这个将在每个使用 new 操作符创建 AR 实例后被调用。
beforeFind: 这个将在一个 AR 查找器被用于执行查询（例如 find(), findAll()）之前被调用。
afterFind: 这个将在每个 AR 实例作为一个查询结果创建时被调用。

10、使用 AR 处理事务

每个 AR 实例都含有一个属性名叫 dbConnection ，是一个 CDbConnection 的实例，这样我们可以在需要时配合 AR 使用由 Yii DAO 提供的 事务 功能:

```
$model=Post::model();
$transaction=$model->dbConnection->beginTransaction();
try
{
    // 查找和保存是可能由另一个请求干预的两个步骤
    // 这样我们使用一个事务以确保其一致性和完整性
    $post=$model->findByPk(10);
    $post->title='new post title';
    $post->save();
    $transaction->commit();
}
catch(Exception $e)
{
    $transaction->rollBack();
}
```

11、命名范围

命名范围(named scope)表示一个命名的（named）查询规则，它可以和其他命名范围联合使用并应用于 Active Record 查询。

命名范围主要是在 CActiveRecord::scopes() 方法中以名字-规则对的方式声明。如下代码在 Post 模型类中声明了两个命名范围, published 和 recently。

```
class Post extends CActiveRecord
{
    ......
    public function scopes()
    {
        return array(
            'published'=>array(
                'condition'=>'status=1',
            ),
            'recently'=>array(
                'order'=>'create_time DESC',
                'limit'=>5,
            ),
        );
    }
}
```

每个命名范围声明为一个可用于初始化 CDbCriteria 实例的数组。例如，recently 命名范围指定 order 属性为 create_time DESC ， limit 属性为 5。他们翻译为查询规则后就会返回最近的 5 篇帖子。

命名范围多用作 find 方法调用的修改器。几个命名范围可以链到一起形成一个更有约束性的查询结果集。例如，要找到最近发布的帖子，我们可以使用如下代码：

```
$posts=Post::model()->published()->recently()->findAll();
```

总体来说，命名范围必须出现在一个 find 方法调用的左边。它们中的每一个都提供一个查询规则，并联合到其他规则，

包括传递给 find 方法调用的那一个。最终结果就像给一个查询添加了一系列过滤器。
命名范围也可用于 update 和 delete 方法。例如，如下代码将删除所有最近发布的帖子：
Post::model()->published()->recently()->delete();
注意: 命名范围只能用于类级别方法。也就是说，此方法必须使用 ClassName::model() 调用。

12、参数化的命名范围

命名范围可以参数化。例如，我们想自定义 recently 命名范围中指定的帖子数量，要实现此目的，不是在 CActiveRecord::scopes 方法中声明命名范围，而是需要定义一个名字和此命名范围的名字相同的方法：

```
public function recently($limit=5)
{
    $this->getDbCriteria()->mergeWith(array(
        'order'=>'create_time DESC',
        'limit'=>$limit,
    ));
    return $this;
}
```

然后，我们就可以使用如下语句获取 3 条最近发布的帖子。

```
$posts=Post::model()->published()->recently(3)->findAll();
```

上面的代码中，如果我们没有提供参数 3，我们将默认获取 5 条最近发布的帖子。

13、默认的命名范围

模型类可以有一个默认命名范围，它将应用于所有 (包括相关的那些) 关于此模型的查询。例如，一个支持多种语言的网站可能只想显示当前用户所指定的语言的内容。因为可能会有很多关于此网站内容的查询，我们可以定义一个默认的命名范围以解决此问题。为实现此目的，我们覆盖 CActiveRecord::defaultScope 方法如下：

```
class Content extends CActiveRecord
{
    public function defaultScope()
    {
        return array(
            'condition'=>"language='".Yii::app()->language."'",
        );
    }
}
```

现在，如果下面的方法被调用，将会自动使用上面定义的查询规则：

```
$contents=Content::model()->findAll();
```

注意，默认的命名范围只会应用于 SELECT 查询。INSERT, UPDATE 和 DELETE 查询将被忽略。

三、Relational Active Record（关联查询）

我们已经知道如何通过 Active Record（AR）从单个数据表中取得数据了，在这一节中，我们将要介绍如何使用 AR 来连接关联的数据表获取数据。
在使用关联 AR 之前，首先要在数据库中建立关联的数据表之间的主键-外键关联，AR 需要通过分析数据库中的定义数据表关联的元信息，来决定如何连接数据。

1、如何声明关联

在使用 AR 进行关联查询之前，我们需要告诉 AR 各个 AR 类之间有怎样的关联。
AR 类之间的关联直接反映着数据库中这个类所代表的数据表之间的关联。从关系数据库的角度来说，两个数据表 A，B 之间可能的关联有三种：一对多，一对一，多对多。而在 AR 中，关联有以下四种：
BELONGS_TO: 如果数据表 A 和 B 的关系是一对多，那我们就说 B 属于 A（B belongs to A）。
HAS_MANY: 如果数据表 A 和 B 的关系是多对一，那我们就说 B 有多个 A（B has many A）。
HAS_ONE: 这是‘HAS_MANY’关系中的一个特例，当 A 最多有一个的时候，我们说 B 有一个 A （B has one A）。
MANY_MANY: 这个相当于关系数据库中的多对多关系。因为绝大多数关系数据库并不直接支持多对多的关系，这时通常都需要一个单独的关联表，把多对多的关系分解为两个一对多的关系。用 AR 的方式去理解的话，我们可以认为 MANY_MANY 关系是由 BELONGS_TO 和 HAS_MANY 组成的。
在 AR 中声明关联，是通过覆盖（Override）父类 CActiveRecord 中的 relations()方法来实现的。这个方法返回一个包含了关系定义的数组，数组中的每一组键值代表一个关联：
'VarName'=>array('RelationType', 'ClassName', 'ForeignKey', ...additional options)
这里的 VarName 是这个关联的名称； RelationType 指定了这个关联的类型，有四个常量代表了四种关联的类型：self::BELONGS_TO，self::HAS_ONE，self::HAS_MANY 和 self::MANY_MANY； ClassName 是这个关系关联到的 AR 类的类名；ForeignKey 指定了这个关联是通过哪个外键联系起来的。后面的 additional options 可以加入一些额外的设置，后面会做介绍。
下面的代码演示了如何定义 User 和 Post 之间的关联。

```
class Post extends CActiveRecord {
    public function relations() {
        return array(
                'author'=>array(
                    self::BELONGS_TO,
                     'User',
                     'authorID'
                    ),
                 'categories'=>array(
                    self::MANY_MANY,
                    'Category',
                    'PostCategory(postID, categoryID)'
                    ),
            );
    }
}

class User extends CActiveRecord {
    public function relations() {
        return array(
                'posts'=>array(
                    self::HAS_MANY,
                    'Post',
                    'authorID'
                    ),
                'profile'=>array(
                    self::HAS_ONE,
                    'Profile',
                    'ownerID'
                    ),
```

```
            );
    }
}
```

说明: 有时外键可能由两个或更多字段组成，在这里可以将多个字段名由逗号或空格分隔， 一并写在这里。对于多对多的关系，关联表必须在外键中注明，例如在 Post 类的 categories 关联中，外键就需要写成 PostCategory(postID, categoryID)。

在 AR 类中声明关联时，每个关联会作为一个属性添加到 AR 类中，属性名就是关联的名称。在进行关联查询时，这些属性就会被设置为关联到的 AR 类的实例，例如在查询取得一个 Post 实例时，它的$author 属性就是代表 Post 作者的一个 User 类的实例。

2、关联查询

进行关联查询最简单的方式就是访问一个关联 AR 对象的某个关联属性。如果这个属性之前没有被访问过，这时就会启动一个关联查询，通过当前 AR 对象的主键连接相关的表，来取得关联对象的值，然后将这些数据保存在对象的属性中。这种方式叫做“延迟加载”，也就是只有等到访问到某个属性时，才会真正到数据库中把这些关联的数据取出来。

下面的例子描述了延迟加载的过程：

```
// retrieve the post whose ID is 10
$post=Post::model()->findByPk(10);
// retrieve the post's author: a relational query will be performed here
$author=$post->author;
```

在不同的关联情况下，如果没有查询到结果，其返回的值也不同：BELONGS_TO 和 HAS_ONE 关联，无结果时返回 null; HAS_MANY 和 MANY_MANY, 无结果时返回空数组。

延迟加载方法使用非常方便，但在某些情况下并不高效。例如，若我们要取得 N 个 post 的作者信息，使用延迟方法将执行 N 次连接查询。此时我们应当使用所谓的急切加载方法。

急切加载方法检索主要的 AR 实例及其相关的 AR 实例. 这通过使用 with() 方法加上 find 或 findAll 方法完成。例如，

```
$posts=Post::model()->with('author')->findAll();
```

上面的代码将返回一个由 Post 实例组成的数组. 不同于延迟加载方法，每个 Post 实例中的 author 属性在我们访问此属性之前已经被关联的 User 实例填充。不是为每个 post 执行一个连接查询, 急切加载方法在一个单独的连接查询中取出所有的 post 以及它们的 author!

我们可以在 with()方法中指定多个关联名字。例如, 下面的代码将取回 posts 以及它们的作者和分类:

```
$posts=Post::model()->with('author','categories')->findAll();
```

我们也可以使用嵌套的急切加载。不使用一个关联名字列表, 我们将关联名字以分层的方式传递到 with() 方法, 如下,

```
$posts=Post::model()->with(
    'author.profile',
    'author.posts',
    'categories')->findAll();
```

上面的代码将取回所有的 posts 以及它们的作者和分类。它也将取出每个作者的 profile 和 posts.

急切加载也可以通过指定 CDbCriteria::with 属性被执行, 如下:

```
$criteria=new CDbCriteria;
$criteria->with=array(
'author.profile',
    'author.posts',
    'categories',
);
$posts=Post::model()->findAll($criteria);
```

或

```
$posts=Post::model()->findAll(array(
'with'=>array(
```

```
            'author.profile',
            'author.posts',
            'categories',
        )

);
```

3、关联查询选项

之前我们提到额外的参数可以被指定在关联声明中。这些选项，指定为 name-value 对，被用来定制关联查询。它们被概述如下：
select: 为关联 AR 类查询的字段列表。默认是 '*', 意味着所有字段。查询的字段名字可用别名表达式来消除歧义（例如：COUNT(??.name) AS nameCount）。
condition: WHERE 子语句。默认为空。注意, 列要使用别名引用（例如：??.id=10）。
params: 被绑定到 SQL 语句的参数. 应当为一个由 name-value 对组成的数组（）。
on: ON 子语句. 这里指定的条件将使用 and 操作符被追加到连接条件中。此选项中的字段名应被消除歧义。此选项不适用于 MANY_MANY 关联。
order: ORDER BY 子语句。默认为空。注意, 列要使用别名引用（例如：??.age DESC）。
with: 应当和此对象一同载入的子关联对象列表. 注意, 不恰当的使用可能会形成一个无穷的关联循环。
joinType: 此关联的连接类型。默认是 LEFT OUTER JOIN。
aliasToken：列前缀占位符。默认是“??.”。
alias: 关联的数据表的别名。默认是 null, 意味着表的别名和关联的名字相同。
together: 是否关联的数据表被强制与主表和其他表连接。此选项只对于 HAS_MANY 和 MANY_MANY 关联有意义。若此选项被设置为 false, ...(此处原文出错!).默认为空。此选项中的字段名以被消除歧义。
having: HAVING 子语句。默认是空。注意, 列要使用别名引用。
index: 返回的数组索引类型。确定返回的数组是关键字索引数组还是数字索引数组。不设置此选项, 将使用数字索引数组。此选项只对于 HAS_MANY 和 MANY_MANY 有意义
此外, 下面的选项在延迟加载中对特定关联是可用的:
group: GROUP BY 子句。默认为空。注意, 列要使用别名引用(例如：??.age)。 本选项仅应用于 HAS_MANY 和 MANY_MANY 关联。
having: HAVING 子句。默认为空。注意, 列要使用别名引用(例如：??.age)。本选项仅应用于 HAS_MANY 和 MANY_MANY 关联。
limit: 限制查询的行数。本选项不能用于 BELONGS_TO 关联。
offset: 偏移。本选项不能用于 BELONGS_TO 关联。
下面我们改变在 User 中的 posts 关联声明,通过使用上面的一些选项:

```
class User extends CActiveRecord

{
    public function relations()
    {
        return array(

            'posts'=>array(self::HAS_MANY, 'Post', 'author_id',
                                'order'=>'posts.create_time DESC',
                                'with'=>'categories'),
            'profile'=>array(self::HAS_ONE, 'Profile', 'owner_id'),
        );
    }
}
```

现在若我们访问 $author->posts, 我们将得到用户的根据发表时间降序排列的 posts. 每个 post 实例也载入了它的分类。

4、为字段名消除歧义

当一个字段的名字出现在被连接在一起的两个或更多表中，需要消除歧义(disambiguated)。可以通过使用表的别名作为字段名的前缀实现。
在关联 AR 查询中，主表的别名确定为 t，而一个关联表的别名和相应的关联的名字相同(默认情况下)。 例如，在下面的语句中，Post 的别名是 t ，而 Comment 的别名是 comments:

```
$posts=Post::model()->with('comments')->findAll();
```

现在假设 Post 和 Comment 都有一个字段 create_time , 我们希望取出 posts 及它们的 comments ,排序方式是先根据 posts 的创建时间,然后根据 comment 的创建时间。 我们需要消除 create_time 字段的歧义，如下:

```
$posts=Post::model()->with('comments')->findAll(array(

    'order'=>'t.create_time, comments.create_time'
));
```

默认情况下,Yii 自动为每个关联表产生一个表别名，我们必须使用此前缀 ??. 来指向这个自动产生的别名。 主表的别名是表自身的名字。

5、动态关联查询选项

我们使用 with()和 with 均可使用动态关联查询选项。 动态选项将覆盖在 relations() 方法中指定的
已存在的选项。例如，使用上面的 User 模型， 若我们想要使用急切加载方法以升序来取出属于一个作者的 posts(关联中的 order 选项指定为降序)， 我们可以这样做:

```
User::model()->with(array(
    'posts'=>array('order'=>'posts.create_time ASC'),
    'profile',
))->findAll();
```

动态查询选项也可以在使用延迟加载方法时使用以执行关联查询。 要这样做，我们应当调用一个方法，它的名字和关联的名字相同，并传递动态查询选项 作为此方法的参数。例如，下面的代码返回一个用户的 status 为 1 的 posts :

```
$user=User::model()->findByPk(1);
$posts=$user->posts(array('condition'=>'status=1'));
```

6、关联查询的性能

如上所述，急切加载方法主要用于当我们需要访问许多关联对象时。 通过连接所有所需的表它产生一个大而复杂的 SQL 语句。一个大的 SQL 语句在许多情况下是首选的。然而在一些情况下它并不高效。
考虑一个例子，若我们需要找出最新的文章以及它们的评论。 假设每个文章有 10 条评论，使用一个大的 SQL 语句，我们将取回很多多余的 post 数据， 因为每个 post 将被它的每条评论反复使用。现在让我们尝试另外的方法：我们首先查询最新的文章， 然后查询它们的评论。用新的方法，我们需要执行执行两条 SQL 语句。有点是在查询结果中没有多余的数据。
因此哪种方法更加高效？没有绝对的答案。执行一条大的 SQL 语句也许更加高效，因为它需要更少的花销来解析和执行 SQL 语句。另一方面，使用单条 SQL 语句，我们得到更多冗余的数据，因此需要更多时间来阅读和处理它们。因为这个原因，Yii 提供了 together 查询选项一边我们在需要时选择两种方法之一。默认下， Yii 使用第一种方式，即产生一个单独的 SQL 语句来执行急切加载。我们可以在关联声明中设置 together 选项为 false 以便一些表被连接在单独的 SQL 语句中。例如，为了使用第二种方法来查询最新的文章及它们的评论，我们可以在 Post 类中声明 comments 关联如下,

```
public function relations()
```

```
{
    return array(

        'comments' => array(self::HAS_MANY, 'Comment', 'post_id', 'together'=>false),
    );

}
```

当我们执行急切加载时，我们也可以动态地设置此选项:

```
$posts = Post::model()->with(array('comments'=>array('together'=>false)))->findAll();
```

7、统计查询

除了上面描述的关联查询，Yii 也支持所谓的统计查询(或聚合查询)。 它指的是检索关联对象的聚合信息，例如每个 post 的评论的数量，每个产品的平均等级等。 统计查询只被 HAS_MANY(例如，一个 post 有很多评论) 或 MANY_MANY (例如，一个 post 属于很多分类和一个 category 有很多 post) 关联对象执行。

执行统计查询非常类似于之前描述的关联查询。我们首先需要在 CActiveRecord 的 relations() 方法中声明统计查询。

```
class Post extends CActiveRecord

{
    public function relations()
    {
        return array(

            'commentCount'=>array(self::STAT, 'Comment', 'post_id'),
            'categoryCount'=>array(self::STAT, 'Category', 'post_category(post_id,
category_id)'),
        );
    }
}
```

在上面，我们声明了两个统计查询：commentCount 计算属于一个 post 的评论的数量，categoryCount 计算一个 post 所属分类的数量。注意 Post 和 Comment 之间的关联类型是 HAS_MANY， 而 Post 和 Category 之间的关联类型是 MANY_MANY (使用连接表 PostCategory)。 如我们所看到的，声明非常类似于之间小节中的关联。唯一的不同是这里的关联类型是 STAT。

有了上面的声明，我们可以检索使用表达式 $post->commentCount 检索一个 post 的评论的数量。 当我们首次访问此属性，一个 SQL 语句将被隐含地执行并检索 对应的结果。我们已经知道，这是所谓的 lazy loading 方法。若我们需要得到多个 post 的评论数目，我们也可以使用 eager loading 方法:

```
$posts=Post::model()->with('commentCount', 'categoryCount')->findAll();
```

上面的语句将执行三个 SQL 语句以取回所有的 post 及它们的评论数目和分类数目。使用延迟加载方法， 若有 N 个 post ,我们使用 2*N+1 条 SQL 查询完成。

默认情况下，一个统计查询将计算 COUNT 表达式(and thus the comment count and category count in the above example).

当我们在 relations()中声明它时，通过 指定额外的选项，可以定制它。可用的选项简介如下。

select: 统计表达式。默认是 COUNT(*)，意味着子对象的个数。

defaultValue: 没有接收一个统计查询结果时被赋予的值。例如，若一个 post 没有任何评论，它的 commentCount 将接收此值。此选项的默认值是 0。

condition: WHERE 子语句。默认是空。

params: 被绑定到产生的 SQL 语句中的参数。它应当是一个 name-value 对组成的数组。

order: ORDER BY 子语句。默认是空。

group: GROUP BY 子语句。默认是空。

having: HAVING 子语句。默认是空。

8、关联查询命名空间

关联查询也可以和 命名空间一起执行。有两种形式。第一种形式，命名空间被应用到主模型。第二种形式，命名空间被应用到关联模型。
下面的代码展示了如何应用命名空间到主模型。

```
$posts=Post::model()->published()->recently()->with('comments')->findAll();
```

这非常类似于非关联的查询。唯一的不同是我们在命名空间后使用了 with() 调用。 此查询应当返回最近发布的 post 和它们的评论。
下面的代码展示了如何应用命名空间到关联模型。

```
$posts=Post::model()->with('comments:recently:approved')->findAll();
```

上面的查询将返回所有的 post 及它们审核后的评论。注意 comments 指的是关联名字，而 recently 和 approved 指的是 在 Comment 模型类中声明的命名空间。关联名字和命名空间应当由冒号分隔。
命名空间也可以在 CActiveRecord::relations() 中声明的关联规则的 with 选项中指定。在下面的例子中， 若我们访问 $user->posts，它将返回此 post 的所有审核后的评论。

```
class User extends CActiveRecord
{
    public function relations()

    {
        return array(
            'posts'=>array(self::HAS_MANY, 'Post', 'author_id',
                'with'=>'comments:approved'),
        );
    }

}
```

注意: 应用到关联模型的命名空间必须在 CActiveRecord::scopes 中指定。结果，它们不能被参数化。

Ⅳ、缓存

缓存是用于提升网站性能的一种即简单又有效的途径。通过存储相对静态的数据至缓存以备所需，我们可以省去生成这些数据的时间。
在 Yii 中使用缓存主要包括配置和访问缓存组件。如下的应用配置指定了一个使用两台缓存服务器的 memcache 缓存组件：

```
array(
    ......
    'components'=>array(
        ......
        'cache'=>array(
            'class'=>'system.caching.CMemCache',
            'servers'=>array(
                array('host'=>'server1', 'port'=>11211, 'weight'=>60),
                array('host'=>'server2', 'port'=>11211, 'weight'=>40),
            ),
        ),
    ),
);
```

程序运行的时候可以通过 Yii::app()->cache 来访问缓存组件。
Yii 提供多种缓存组件以便在不同的媒介上存储缓存数据。比如 CMemCache 组件封装了 PHP memcache 扩展，它使用内存作为存储缓存的媒介；CApcCache 组件封装了 PHP APC 扩展；CDbCache 组件在数据库里存储缓存数据。下面是各种缓存组件的简要说明：
CMemCache: 使用 PHP memcache 扩展。
CApcCache: 使用 PHP APC 扩展。
CXCache: 使用 PHP XCache 扩展。
CDbCache: 使用一张数据库表来存储缓存数据。它默认在运行时目录建立并使用一个 SQLite3 数据库，你可以通过设置 connectionID 属性显式地指定一个数据库给它使用。
提示: 因为所有这些缓存组件都从同一个基础类 CCache 扩展而来，不需要修改使用缓存的代码即可在不同的缓存组件之间切换。
缓存可以在不同的级别使用。在最低级别，我们使用缓存来存储单个数据，比如一个变量，我们把它叫做 数据缓存。往上一级，我们缓存一个由视图脚本生成的页面片断。在最高级别，我们存储整个页面以便需要的时候直接从缓存读取。
接下来我们将阐述如何在这些级别上使用缓存。
注意: 按定义来讲，缓存是一个不稳定的存储媒介，它不保证缓存一定存在——不管该缓存是否过期。所以，不要使用缓存进行持久存储（比如，不要使用缓存来存储 SESSION 数据）。

一、数据缓存

数据缓存也就是在缓存中存储一些 PHP 变量，过一会再取出来。缓存基础类 CCache 提供了两个最常用的方法：set() 和 get()。

要在缓存中存储变量 $value，我们选择一个唯一 ID 并调用 set() 来存储它：

```
Yii::app()->cache->set($id, $value);
```

被缓存的数据会一直保留在缓存中，直到因一些缓存策略而被删除（比如缓存空间满了，删除最旧的数据）。要改变这一行为，我们还可以在调用 set() 时加一个过期参数，这样数据过一段时间就会自动从缓存中清除。

```
// 在缓存中保留该值最多 30 秒
Yii::app()->cache->set($id, $value, 30);
```

当我们稍后需要访问该变量时（不管是不是同一 Web 请求），我们调用 get() （传入 ID）来从缓存中获取它。如果返回值为 false，说明该缓存不可用，需要我们重新生成它。

```
$value=Yii::app()->cache->get($id);
if($value===false)
{
    // 因为在缓存中没找到，重新生成 $value
    // 再缓存一下以备下次使用
    // Yii::app()->cache->set($id,$value);
}
```

为一个要缓存的变量选择 ID 时，确保该 ID 在应用中是唯一的。不必保证 ID 在跨应用的情况下保证唯一，因为缓存组件有足够的智能来区分不同应用的缓存 ID。
要从缓存中删除一个缓存值，调用 delete()；要清空所有缓存，调用 flush()。调用 flush() 时要非常小心，因为它会把其它应用的缓存也清空。
提示: 因为 CCache 实现了 ArrayAccess 接口，可以像数组一样使用缓存组件。例如：

```
$cache=Yii::app()->cache;
```

```
$cache['var1']=$value1;  // 相当于: $cache->set('var1',$value1);
$value2=$cache['var2'];  // 相当于: $value2=$cache->get('var2');
```

缓存依赖

除了过期设置，缓存数据还会因某些依赖条件发生改变而失效。如果我们缓存了某文件的内容，而该文件后来又被更新了，我们应该让缓存中的拷贝失效，从文件中读取最新内容（而不是从缓存）。

我们把一个依赖关系表现为一个 CCacheDependency 或它的子类的实例，调用 set() 的时候把依赖实例和要缓存的数据一起传入。

```
// 缓存将在 30 秒后过期
// 也可能因依赖的文件有更新而更快失效
Yii::app()->cache->set($id, $value, 30, new CFileCacheDependency('FileName'));
```

如果我们现在调用 get() 从缓存中获取 $value，缓存组件将检查依赖条件。如果有变，我们会得到 false 值——数据需要重新生成。

下面是可用的缓存依赖的简要说明：

CFileCacheDependency: 该依赖因文件的最近修改时间发生改变而改变。

CDirectoryCacheDependency: 该依赖因目录（或其子目录）下的任何文件发生改变而改变。

CDbCacheDependency: 该依赖因指定的 SQL 语句的查询结果发生改变而改变。

CGlobalStateCacheDependency: 该依赖因指定的全局状态值发生改变而改变。全局状态是应用中跨请求、跨 SESSION 的持久变量，它由 CApplication::setGlobalState() 来定义。

CChainedCacheDependency: 该依赖因依赖链中的任何一环发生改变而改变。

二、片段缓存(Fragment Caching)

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() 。这两种方法开始和结束包括的页面内容将被缓存。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...
<?php if($this->beginCache($id)) { ?>
...被缓存的内容...
<?php $this->endCache(); } ?>
...别的 HTML 内容...
```

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。

1. 缓存选项(Caching Options)

当调用 beginCache()，可以提供一个数组由缓存选项组成的作为第二个参数，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。

2. 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...
<?php if($this->beginCache($id, array('duration'=>3600))) { ?>
...被缓存的内容...
<?php $this->endCache(); } ?>
```

...其他 HTML 内容...
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。

3. 依赖(Dependency)

像 data caching ，内容片段被缓存也可以有依赖。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，我们建立了 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 lastModified 列的值是否变化：

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

4. 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合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。

5. 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 initially requested 表单(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...
<?php if($this->beginCache($id, array('requestTypes'=>array('GET')))) { ?>
...被缓存的内容...
<?php $this->endCache(); } ?>
...其他 HTML 内容...
```

6. 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，意见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...
<?php if($this->beginCache($id1)) { ?>
...外部被缓存内容...
```

```
    <?php if($this->beginCache($id2)) { ?>
    ...内部被缓存内容...
    <?php $this->endCache(); } ?>
...外部被缓存内容...
<?php $this->endCache(); } ?>
```
...其他 HTML 内容...

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，即使内容中的内部缓存已经过期。

三、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 (/doc/guide/caching.fragment)一个特殊情况 。由于网页内容是往往通过应用布局来生成，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（/doc/guide/basics.controller#filter）来完成这一任务。下面的代码演示如何配置缓存过滤器：

```
public function filters()
{
    return array(
        array(
            'system.web.widgets.COutputCache',
            'duration'=>100,
            'varyByParam'=>array('id'),
        ),
    );
}
```

上述过滤器配置会使过滤器适用于控制器中的所有行动。我们可能会限制它在一个或几个行动通过使用插件操作器。更多的细节中可以看过滤器（/doc/guide/basics.controller#filter） 。

提示：我们可以使用 COutputCache 作为一个过滤器，因为它从 CFilterWidget 继承过来 ，这意味着它是一个工具(widget)和一个过滤器。事实上， widge 的工作方式和过滤器非常相似：工具 widget (过滤器 filter)是在 action 动作里的内容执行前执行，在执行后结束。

四、动态内容(Dynamic Content)

当使用 fragment caching 或 page caching，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静态的。例如，帮助页可能会显示静态的帮助信息，而用户名称显示的是当前用户的。

解决这个问题，我们可以根据用户名匹配缓存内容，但是这将是我们宝贵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 功能 。

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...
<?php if($this->beginCache($id)) { ?>
...被缓存的片段内容...
```

```
    <?php $this->renderDynamic($callback); ?>
...被缓存的片段内容...
<?php $this->endCache(); } ?>
...别的 HTML 内容...
```

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。其他任何的参数，将传递到 renderDynamic()方法中。回调将返回动态内容而不是仅仅显示它。

Ⅴ、扩展 Yii

在开发中扩展 Yii 是一个很常见的行为.例如,当你写一个新的控制器时,你通过继承 CController 类扩展了 Yii;当你编写一个新的组件时,你正在继承 CWidget 或者一个已存在的组件类.如果扩展代码是由第三方开发者为了复用而设计的,我们则称之为 extension(扩展)。

一个扩展通常是为了一个单一的目的服务的.在 Yii 中,他可以按照如下分类:

* 应用的部件
* 组件
* 控制器
* 动作
* 过滤器
* 控制台命令
* 校验器: 校验器是一个继承自 CValidator 类的部件。
* 辅助器: 辅助器是一个只具有静态方法的类.它类似于使用类名作为命名空间的全局函数。
* 模块: 模块是一个有着若干个类文件和相应特长文件的包.一个模块通常更高级,比一个单一的部件具备更先进的功能.例如我们可以拥有一个具备整套用户管理功能的模块。

扩展也可以是不属于上述分类中的任何一个的部件。事实上,Yii 是设计得很谨慎的,以至于几乎它的每段代码都可以被扩展和订制以适用于特定需求。

一、使用扩展

使用扩展通常包含了以下三步:

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。

每个扩展都有一个所有扩展中唯一的名称标识。把一个扩展命名为 xyz ,我们也可以使用路径别名定位到包含了 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。

1、应用的部件

使用 应用的部件, 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(
    // 'preload'=>array('xyz',...),
    'components'=>array(
        'xyz'=>array(
            'class'=>'application.extensions.xyz.XyzClass',
            'property1'=>'value1',
```

```
            'property2'=>'value2',
        ),
        // 其他部件配置
    ),
);
```

然后,我们可以在任何地方通过使用 Yii::app()->xyz 来访问部件.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。

2、组件

组件 主要用在 视图 里.假设组件类 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
<?php $this->widget('application.extensions.xyz.XyzClass', array(
    'property1'=>'value1',
    'property2'=>'value2')); ?>

// 组件可以包含主体内容
<?php $this->beginWidget('application.extensions.xyz.XyzClass', array(
    'property1'=>'value1',
    'property2'=>'value2')); ?>

...组件的主体内容...

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

3、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
{
    public function actions()
    {
        return array(
            'xyz'=>array(
                'class'=>'application.extensions.xyz.XyzClass',
                'property1'=>'value1',
                'property2'=>'value2',
            ),
            // 其他动作
        );
    }
}
```

然后,我们可以通过 路由 test/xyz 来访问。

4、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 挂起时预处理,提交处理用户请求。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
{
    public function filters()
    {
        return array(
            array(
                'application.extensions.xyz.XyzClass',
                'property1'=>'value1',
                'property2'=>'value2',
            ),
            // 其他过滤器
        );
    }
}
```

在上述代码中,我们可以在数组的第一个元素离使用加号或者减号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。

5、控制器

控制器 提供了一套可以被用户请求的动作。我们需要在 应用配置 里设置 CWebApplication::controllerMap 属性,才能在控制器里使用扩展:

```
return array(
    'controllerMap'=>array(
        'xyz'=>array(
            'class'=>'application.extensions.xyz.XyzClass',
            'property1'=>'value1',
            'property2'=>'value2',
        ),
        // 其他控制器
    ),
);
```

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。

6、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
{
    public function rules()
    {
        return array(
            array(
                'attr1, attr2',
```

```
                'application.extensions.xyz.XyzClass',
                'property1'=>'value1',
                'property2'=>'value2',
            ),
            // 其他校验规则
        );
    }
}
```

7、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.假设命令控制台 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(
    'commandMap'=>array(
        'xyz'=>array(
            'class'=>'application.extensions.xyz.XyzClass',
            'property1'=>'value1',
            'property2'=>'value2',
        ),
        // 其他命令
    ),
);
```

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。

8、模块

模块通常由多个类文件组成,且往往综合上述扩展类型。因此,你应该按照和以下一致的指令来使用模块。

9、通用部件

使用一个通用 部件, 我们首先需要通过使用
Yii::import('application.extensions.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。

二、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：
*扩展最好是自己自足。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。
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。当您要创建一个主要用于在您自己项目的 component 部件，这些描述也适用。

1、Application Component（应用部件）

一个 application component 应实现接口 IApplicationComponent 或继承 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。

2、Widget（小工具）

widget 应继承 CWidget 或其子类。 A widget should extend from CWidget or its child classes.
最简单的方式建立一个新的小工具是继承一个现成的小工具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，您可以配置其 CTabView::cssFile 属性，当使用的小工具时。您还可以继承 CTabView 如下，让您在使用小工具时，不再需要配置属性。

```
class MyTabView extends CTabView
{
    public function init()
    {
        if($this->cssFile===null)
        {
            $file=dirname(__FILE__).DIRECTORY_SEPARATOR.'tabview.css';
            $this->cssFile=Yii::app()->getAssetManager()->publish($file);
        }
        parent::init();
    }
}
```

在上面的，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，以便他们能够封装成扩展。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小工具，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 $this->beginWidget 插入一个小工具时被调用，第二种方法在$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，我们可以开始 output buffering 在 CWidget::init 和在 CWidget::run 中回收缓冲输出作进一步处理。 If we want to capture and process the content displayed between these two method invocations, we can start output buffering in CWidget::init and retrieve the buffered output in CWidget::run for further processing.
在网页中使用的小工具，小工具往往包括 CSS，Javascript 或其他资源文件。我们叫这些文件 assets，因为他们和小工具类在一起，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 发布他们，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 ：

```
class MyWidget extends CWidget
{
    protected function registerClientScript()
    {
        // ...publish CSS or JavaScript file here...
```

```
            $cs=Yii::app()->clientScript;
            $cs->registerCssFile($cssFile);
            $cs->registerScriptFile($jsFile);
        }
}
```

小工具也可能有自己的视图文件。如果是这样，创建一个目录命名 views 在包括小工具类文件的目录下，并把所有的视图文件放里面。在小工具类中使用$this->render('ViewName') 来 render 渲染小工具视图，类似于我们在控制器里做。

3、Action（动作）

action 应继承 CAction 或者其子类。action 要实现的主要方法是 IAction::run 。

4、Filter（过滤器）

filter 应继承 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
{
    protected function preFilter($filterChain)
    {
        // logic being applied before the action is executed
        return true; // false if the action should not be executed
    }

    protected function postFilter($filterChain)
    {
        // logic being applied after the action is executed
    }
}
```

参数$filterChain 的类型是 CFilterChain，其包含当前被 filter 的 action 的相关信息。

5、Controller（控制器）

controller 要作为扩展需继承 CExtController，而不是 CController。主要的原因是因为 CController 认定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。

6、Validator（验证）

Validator 需继承 CValidator 和实现 CValidator::validateAttribute 方法。

```
class MyValidator extends CValidator
{
    protected function validateAttribute($model,$attribute)
    {
        $value=$model->$attribute;
        if($value has error)
            $model->addError($attribute,$errorMessage);
```

```
    }
}
```

7、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
{
    public function run($args)
    {
        // $args gives an array of the command-line arguments for this command
    }

    public function getHelp()
    {
        return 'Usage: how to use this command';
    }
}
```

8、Module（模块）

请参阅 modules 一节中关于就如何创建一个模块。

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 。

9、Generic Component（通用组件）

开发一个通用组件扩展类似写一个类。还有，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。

三、使用第三方库

Yii 是精心设计的，使第三方库可易于集成，进一步扩大 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 application 从 Zend framework 使用 Zend_Search_Lucene 部件。

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 。 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：

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

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 做到。

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：

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