

Easy, Efficient and Extensible



框架文档

Web Programming

Framework Documentation

Yii 框架文档

本文档根据[官方中文文档](#)及《[Yii 权威指南](#)》整理，本文档在《[Yii 文档协议 \(Terms of the Yii Documentation\)](#)》下发布，本文档可被自由拷贝及修改。本文档由 Illinus 和 Snail 整理，封面由 Snail 设计。

2010/7/18

目 录

开始	1
新特征	1
从 1.0 升级至 1.1	3
Yii 是什么	3
安装	4
建立第一个 Yii 应用	5
基本概念	18
模型-视图-控制器 (MVC)	18
入口脚本	20
应用	20
控制器	23
模型	28
视图	29
组件	32
模块	35
路径别名和命名空间	37
约定	39
开发流程	41
使用表单	42
创建模型	42
创建动作	50
创建表单	52
收集表格输入	56
使用 FORM BUILDER	58
使用数据库	72
数据访问对象 (DAO)	72
ACTIVE RECORD	77
缓存	105
概述	105
数据缓存	106
片段缓存	107
页面缓存	110
动态内容	111
扩展 Yii	113

概述	113
使用扩展	113
一般组件	121
创建扩展	121
使用第三方库	127
测试	128
概述	128
定义 FIXTURE	131
单元测试	133
功能测试	136
专题	139
自动化代码生成	139
网址管理	148
验证和授权	154
主题	165
日志	170
错误处理	174
WEB SERVICE	176
国际化	182
使用可选的模板句法	186
安全措施	189
性能调整	192
常用扩展手册	196
SRBAC 1.1.0.2 使用手册	196

开始

新特征

此页面概述了在每个 Yii 版本中的主要的新特征。

Version 1.1.3

- [增加了在应用配置中配置 widget 默认值的支持](#)

Version 1.1.2

- [增加了一个基于 Web 的代码生成工具 Gii](#)

Version 1.1.1

- 增加了 CActiveForm 简化编写和表单相关的代码，并支持对客户端和服务端无缝和一致的验证。
- 调整了 yiic 工具生成的代码。特别是，现在生成的骨架应用有多个布局；为 crud 页面重新组织了菜单；为 crud 命令生成的 admin 页面增加了搜索和过滤特征；使用 CActiveForm 渲染表单。
- [增加了允许定义全局 yiic 命令。](#)

Version 1.1.0

- [增加了对编写单元和功能测试的支持](#)
- [增加了支持使用 widget 皮肤](#)
- [增加了一个可扩展的 form builder](#)
- 改善了声明安全模型属性的方式。查看 [Securing Attribute Assignments](#).
- 为关联 active record 查询更改了默认的 eager 载入算法以便所有的表被连接到一个单独的 SQL 语句中。
- 更改默认的表别名为 active record 关联的名字。
- [增加对使用表前缀的支持。](#)
- 增加了一个新的扩展集，即 [Zii library](#).
- 在一个 AR 查询中主表的别名确定为 't'

Version 1.0.11

- 增加支持以参数化后的主机名解析和创建 URL
 - [Parameterizing Hostnames](#)

Version 1.0.10

- 增加支持使用 CPhpMessageSource 管理模块信息

- [信息翻译](#)
- 增加支持关联匿名函数作为 `event handlers`
 - [组件事件](#)

Version 1.0.8

- 增加支持同时检索多个缓存值
 - [数据缓存](#)
- 引入了一个新的默认根路径别名 `ext`，它指向含有所有第三方扩展的目录。
 - [使用扩展](#)

Version 1.0.7

- 增加了在跟踪信息中支持显示调用栈
 - [Logging Context Information](#)
- 增加了 `index` 选项到 `AR` 关联，以便被关联的对象可以被使用一个指定字段的值索引
 - [Relational Query Options](#)

Version 1.0.6

- 增加在 `update` 和 `delete` 方法使用命名空间的支持：
 - [命名空间](#)
- 增加支持在关联规则的 `with` 选项中使用命名空间：
 - [Relational Query with Named Scopes](#)
- 增加支持 `profiling SQL executions`
 - [Profiling SQL Executions](#)
- 增加记录额外上下文信息支持
 - [Logging Context Information](#)
- 增加通过设置它的 `urlFormat` 和 `caseSensitive` 选项来定制一个单独的 `URL` 规则：
 - [对用户友好的 URL](#)
- 增加使用一个 `controller action` 显示应用错误：
 - [使用 Action 管理错误](#)

Version 1.0.5

- 增强了 `active record`，支持命名空间。查看：
 - [命名空间](#)
 - [默认命名空间](#)
 - [使用命名空间进行关联查询](#)
- 增强了 `active record`，支持 `lazy loading with dynamic query options`。查看：
 - [动态关联查询选项](#)
- 增强了 [CUrlManager](#)，支持 `URL` 规则中的 `route` 部分。查看：
 - [Parameterizing Routes in URL Rules](#)

从 1.0 升级至 1.1

与 Model Scenarios 相关的改变

- 删除了 `CModel::safeAttributes()`。安全属性被定义为由 `CModel::rules()` 为特定场景指定的规则来验证。
- 改变了 `CModel::validate()`, `CModel::beforeValidate()` 和 `CModel::afterValidate()`。 `CModel::setAttributes()`, `CModel::getSafeAttributeNames()` 参数 'scenario' 被删除。你应当得到和设置模型场景, 通过 `CModel::scenario`。
- 改变了 `CModel::getValidators()` 并删除了 `CModel::getValidatorsForAttribute()`。 `CModel::getValidators()` 现在只返回适用于模型指定场景的验证器。
- 改变了 `CModel::isAttributeRequired()` 和 `CModel::getValidatorsForAttribute()`。 `scenario` 参数被删除。而是使用模型的 `scenario` 属性。
- 删除了 `CHTML::scenario`。 `CHtml` 将使用模型的 `scenario` 属性。

与 Eager Loading for Relational Active Record 相关的改变

- 默认的, 一条 JOIN 语句将被生成并为 `eager` 载入涉及的所有关联执行。若主表有它的 `LIMIT` 或 `OFFSET` 查询选项, 它将被单独首先查询, 然后跟上取回其所有关联对象的另外的 SQL。在版本 1.0.x 之前, 默认的行为是, 若一个 `eager` 载入涉及到 `N HAS_MANY` or `MANY_MANY` 关联, 将有 `N+1` 个 SQL 语句。

与在 Relational Active Record 中表别名相关的改变

- 现在一个关联表的默认别名和对应的关联的名字相同。在版本 1.0.x 之前, 默认情况下 Yii 将自动为每个关联表生成一个表别名, 我们必须使用前缀 `??`。来指向自动生成的别名。
- 在 AR 查询中的主表的别名确定为 `t`。在之前的版本 1.0.x, 它和表的名字相同。This will cause existing AR query code to break if they explicitly specify column prefixes using the table name. 解决办法是替换这些前缀为 `'t.'`。

与 Tabular 输入相关的改变

- 对于属性名字, 使用 `Field[$i]` 不再是有效的, 它们应当类似于 `[$i]Field` 为了支持数组类型的字段 (例如 `[$i]Field[$index]`)。

其他改变

- [CActiveRecord](#) 构造器的签名被改变。第一个参数(属性列表) 被删除。

Yii 是什么

Yii 是一个基于组件、用于开发大型 Web 应用的高性能 PHP 框架。它将 Web 编程中的可重用性发挥到极致，能够显著加速开发进程。Yii（读作“易”）代表简单(easy)、高效(efficient)、可扩展(extendable)。

需求

要运行一个基于 Yii 开发的 Web 应用，你需要一个支持 PHP 5.1.0 （或更高版本）的 Web 服务器。

对于想使用 Yii 的开发者而言，熟悉面向对象编程(OOP)会使开发更加轻松，因为 Yii 就是一个纯 OOP 框架。

Yii 适合做什么？

Yii 是一个通用 Web 编程框架，能够开发任何类型的 Web 应用。它是轻量级的，又装配了很好很强大的缓存组件，因此尤其适合开发大流量的应用，比如门户、论坛、内容管理系统(CMS)、电子商务系统，等等。

Yii 和其它框架比起来怎样？

和大多数 PHP 框架一样，Yii 是一个 MVC 框架。

Yii 以性能优异、功能丰富、文档清晰而胜出其它框架。它从一开始就为严谨的 Web 应用开发而精心设计，不是某个项目的副产品或第三方代码的组合，而是融合了作者丰富的 Web 应用开发经验和其它热门 Web 编程框架（或应用）优秀思想的结晶。

安装

Yii 的安装由如下两步组成：

1. 从 yiiframework.com 下载 Yii 框架。
2. 将 Yii 压缩包解压至一个 Web 可访问的目录。

提示: 安装在 Web 目录不是必须的，每个 Yii 应用都有一个入口脚本，只有它才必须暴露给 Web 用户。其它 PHP 脚本（包括 Yii）应该保护起来不被 Web 访问，因为它们可能会被黑客利用。

需求

安装完 Yii 以后你也许想验证一下你的服务器是否满足使用 Yii 的要求，只需浏览器中输入如下网址来访问需求检测脚本：

```
http: //hostname/path/to/yii/requirements/index.php
```

Yii 的最低需求是你的 Web 服务器支持 PHP 5.1.0 或更高版本。Yii 在 Windows 和 Linux 系统上的 [Apache HTTP 服务器](#) 中测试通过，应该在其它支持 PHP 5 的 Web 服务器和平台上也工作正常。

建立第一个 Yii 应用

为了对 Yii 有个初步认识，我们在本节讲述如何建立第一个 Yii 应用。我们将使用强大的 `yiic` 工具，它用来自动生成各种代码。假定 `YiiRoot` 为 Yii 的安装目录。

在命令行运行 `yiic`，如下所示：

```
% YiiRoot/framework/yiic webapp WebRoot/testdrive
```

注意：在 MacOS、Linux 或 Unix 系统中运行 `yiic` 时，你可能需要修改 `yiic` 文件的权限使它能够运行。此外，也可以这样运行此工具：

```
% cd WebRoot/testdrive
```

```
% php YiiRoot/framework/yiic.php webapp WebRoot/testdrive
```

这将在 `WebRoot/testdrive` 目录下建立一个最基本的 Yii 应用，`WebRoot` 代表你的 Web 服务器根目录。这个应用具有所有必须的目录和文件，因此可以方便地在此基础上添加更多功能。

不用写一行代码，我们可以在浏览器中访问如下 URL 来看看我们第一个 Yii 应用：

```
http://hostname/testdrive/index.php
```

正如我们看到的，这个应用包含三个页面：首页、联系页、登录页。首页展示一些关于应用和用户登录状态的信息，联系页显示一个联系表单以便用户填写并提交他们的咨询，登录页允许用户先通过认证然后访问已授权的内容。查看下列截图了解更多：

My Web Application

[Home](#) [About](#) [Contact](#) [Login](#)

Welcome to *My Web Application*

Congratulations! You have successfully created your Yii application.

You may change the content of this page by modifying the following two files:

- View file: D:\testdrive\protected\views\site\index.php
- Layout file: D:\testdrive\protected\views\layouts\main.php

For more details on how to further develop this application, please read the [documentation](#). Feel free to ask in the [forum](#), should you have any questions.

Copyright © 2010 by My Company.
All Rights Reserved.
Powered by [Yii Framework](#)

联系页

My Web Application

[Home](#) [About](#) [Contact](#) [Login](#)

[Home](#) » [Contact](#)

Contact Us

If you have business inquiries or other questions, please fill out the following form to contact us. Thank you.


*Fields with * are required.*

Name *

Email *

Subject *

Body *

Verification Code
 [Get a new code](#)

Please enter the letters as they are shown in the image above.
Letters are not case-sensitive.

Copyright © 2010 by My Company.
All Rights Reserved.
Powered by [Yii Framework](#)

输入错误的联系页

My Web Application

[Home](#) [About](#) [Contact](#) [Login](#)

[Home](#) » [Contact](#)

Contact Us

If you have business inquiries or other questions, please fill out the following form to contact us. Thank you.

Fields with * are required.

Please fix the following input errors:

- Subject cannot be blank.
- Body cannot be blank.
- The verification code is incorrect.


Name *

Email *

Subject *

Body *

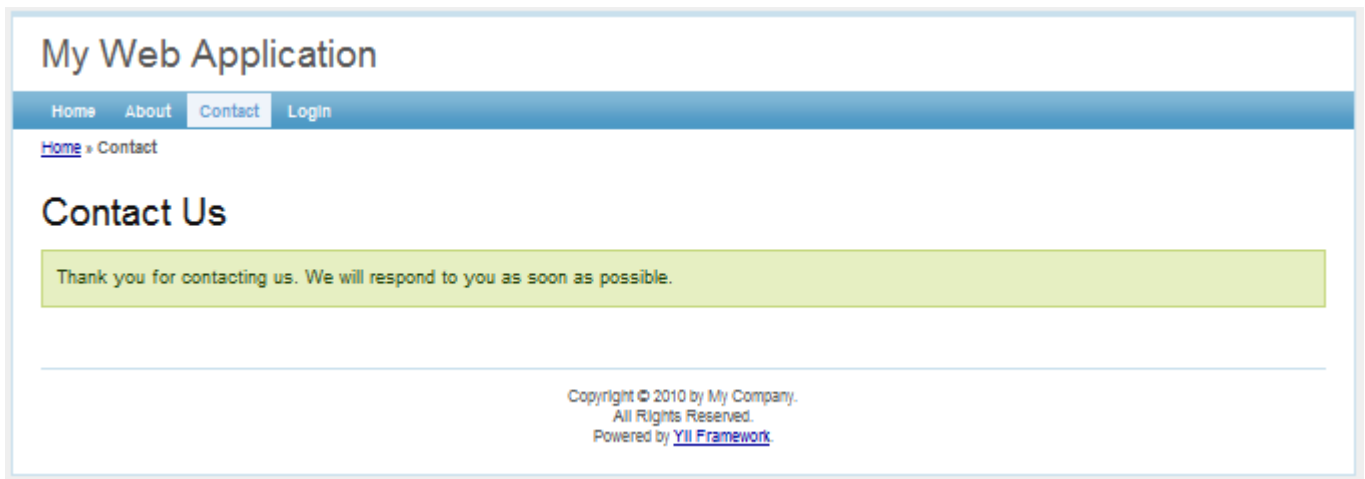
Verification Code

 [Get a new code](#)

Please enter the letters as they are shown in the image above.
Letters are not case-sensitive.

Copyright © 2010 by My Company.
All Rights Reserved.
Powered by [Yii Framework](#)

提交成功的联系页



My Web Application

Home About **Contact** Login

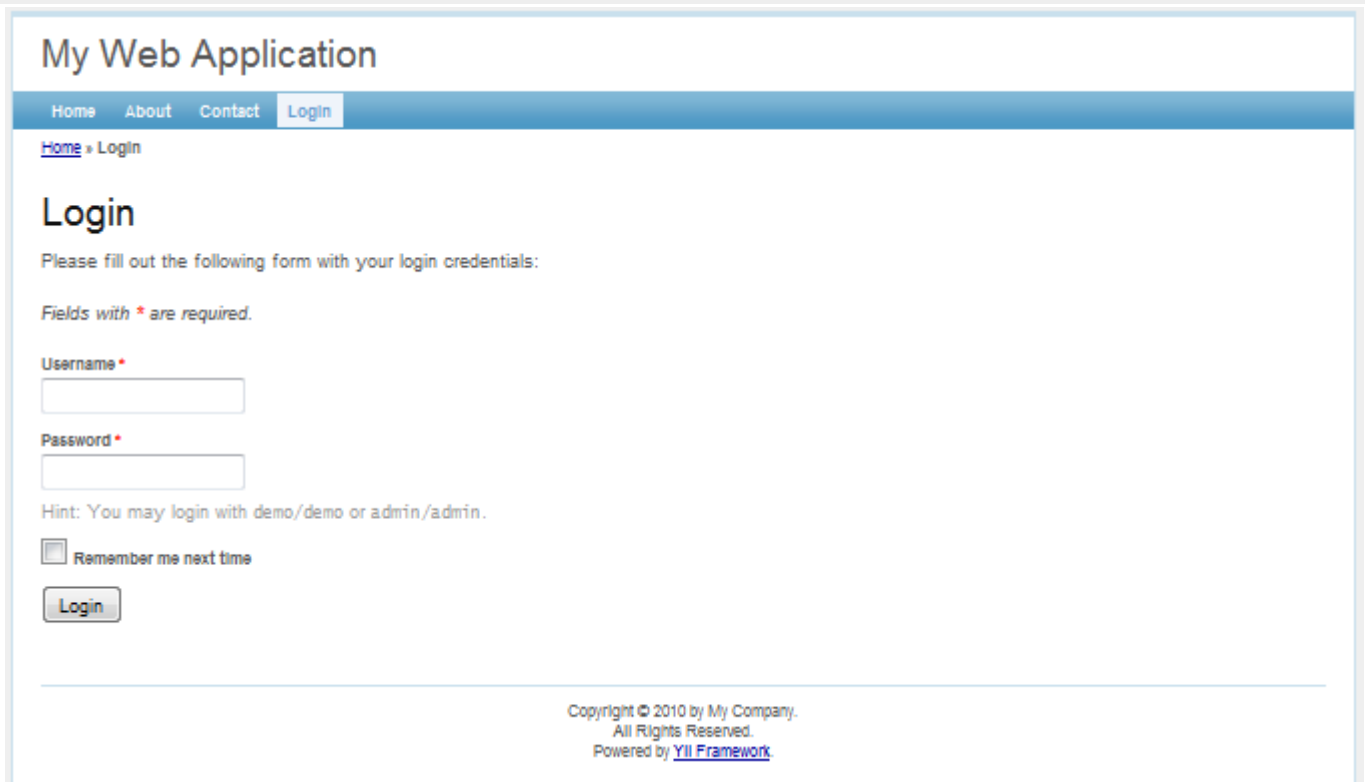
[Home](#) » Contact

Contact Us

Thank you for contacting us. We will respond to you as soon as possible.

Copyright © 2010 by My Company.
All Rights Reserved.
Powered by [Yii Framework](#)

登录页



My Web Application

Home About **Login** Contact

[Home](#) » Login

Login

Please fill out the following form with your login credentials:

Fields with * are required.

Username *

Password *

Hint: You may login with demo/demo or admin/admin.

☐ Remember me next time

Login

Copyright © 2010 by My Company.
All Rights Reserved.
Powered by [Yii Framework](#)

下面的树图描述了我们这个应用的目录结构。请查看[约定](#)以获取该结构的详细解释。

testdrive/

index.php web 应用入口脚本文件

index-test.php 功能测试使用的入口脚本文件

assets/ 包含公开的资源文件

<code>css/</code>	包含 CSS 文件
<code>images/</code>	包含图片文件
<code>themes/</code>	包含应用主题
<code>protected/</code>	包含受保护的应用文件
<code>yiic</code>	<code>yiic</code> 命令行脚本
<code>yiic.bat</code>	Windows 下的 <code>yiic</code> 命令行脚本
<code>yiic.php</code>	<code>yiic</code> 命令行 PHP 脚本
<code>commands/</code>	包含自定义的 ' <code>yiic</code> ' 命令
<code> shell/</code>	包含自定义的 ' <code>yiic shell</code> ' 命令
<code>components/</code>	包含可重用的用户组件
<code> Controller.php</code>	所有控制器类的基础类
<code> Identity.php</code>	用来认证的 ' <code>Identity</code> ' 类
<code>config/</code>	包含配置文件
<code> console.php</code>	控制台应用配置
<code> main.php</code>	Web 应用配置
<code> test.php</code>	功能测试使用的配置
<code>controllers/</code>	包含控制器的类文件
<code> SiteController.php</code>	默认控制器的类文件
<code>data/</code>	包含示例数据库
<code> schema.mysql.sql</code>	示例 MySQL 数据库
<code> schema.sqlite.sql</code>	示例 SQLite 数据库
<code> testdrive.db</code>	示例 SQLite 数据库文件
<code>extensions/</code>	包含第三方扩展
<code>messages/</code>	包含翻译过的消息
<code>models/</code>	包含模型的类文件

LoginForm.php	'login' 动作的表单模型
ContactForm.php	'contact' 动作的表单模型
runtime/	包含临时生成的文件
tests/	包含测试脚本
views/	包含控制器的视图和布局文件
layouts/	包含布局视图文件
main.php	所有视图的默认布局
column1.php	使用单列页面使用的布局
column2.php	使用双列的页面使用的布局
site/	包含 'site' 控制器的视图文件
pages/	包含 "静态" 页面
about.php	"about" 页面的视图
contact.php	'contact' 动作的视图
error.php	'error' 动作的视图(显示外部错误)
index.php	'index' 动作的视图
login.php	'login' 动作的视图

连接到数据库

大多数 Web 应用由数据库驱动，我们的测试应用也不例外。要使用数据库，我们首先需要告诉应用如何连接它。修改应用的配置文件 `WebRoot/testdrive/protected/config/main.php` 即可，如下所示：

```
return array(  
  
    .....  
  
    'components'=>array(  
  
        .....  
  
        'db'=>array(  

```

```
'connectionString'=>'sqlite:protected/data/testdrive.db',  
  
    ),  
  
    ),  
  
    .....  
);
```

上面的代码告诉 Yii 应用需要在需要时将连接到 SQLite 数据库 `WebRoot/testdrive/protected/data/testdrive.db`。注意这个 SQLite 数据库已经包含在我们创建的应用框架中。数据库只包含一个名为 `tbl_user` 的表：

```
CREATE TABLE tbl_user (  
  
    id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,  
  
    username VARCHAR(128) NOT NULL,  
  
    password VARCHAR(128) NOT NULL,  
  
    email VARCHAR(128) NOT NULL  
  
);
```

若你想要换成一个 MySQL 数据库，你需要导入文件 `WebRoot/testdrive/protected/data/schema.mysql.sql` 来建立数据库。

注意：要使用 Yii 的数据库功能，我们需要启用 PHP 的 PDO 扩展和相应的驱动扩展。对于测试应用来说，我们需要启用 `php_pdo` 和 `php_pdo_sqlite` 扩展。

实现 CRUD 操作

激动人心的时刻来了。我们想要为刚才建立的 User 表实现 CRUD (create, read, update 和 delete) 操作，这也是实际应用中最常见的操作。我们无需麻烦地编写实际代码，这里我们将使用 Gii —— 一个强大的基于 Web 的代码生成器。

信息：Gii 自版本 1.1.2 可用。在此之前，可以使用 yiic 来实现相同的功能。更多细节，请参考 [用 yiic shell 实现 CRUD 操作](#)。

配置 Gii

为了使用 Gii，首先需要编辑文件 `WebRoot/testdrive/protected/main.php`，这是已知的应用配置文件：

```
return array(

    .....

    'import'=>array(

        'application.models.*',

        'application.components.*',

    ),

    'modules'=>array(

        'gii'=>array(

            'class'=>'system.gii.GiiModule',

            'password'=>'pick up a password here',

        ),

    ),

);
```

然后，访问 URL `http://hostname/testdrive/index.php?r=gii`。这里我们需要输入密码，它是在我们在上面的配置中指定的。

生成 User 模型

登陆后，点击链接 **Model Generator**。它将显示下面的模型生成页面，

Model Generator

Model Generator

This generator generates a model class for the specified database table.

Fields with * are required. Click on **highlighted fields** to edit them.

Table Prefix

[empty]

Table Name *

tbl_user

Model Class *

User

Base Class *

CActiveRecord

Model Path *

application.models

Code Template *

default (D:\yii\framework\gii\generators\model\templates\default)

Preview

在 Table Name 输入框中，输入 `tbl_user`。在 Model Class 输入框中，输入 `User`。然后点击 Preview 按钮。这里将展示将要生成的新文件。现在点击 Generate 按钮。一个名为 `User.php` 将生成到 `protected/models` 目录中。如我们稍后描述的，`User` 模型类允许我们以面向对象的方式来访问数据表 `tbl_user`。

生成 CRUD 代码

在创建模型类之后，我们将生成执行 CRUD 操作的代码。我们选择 Gii 中的 Crud Generator，如下所示，

CRUD Generator

Crud Generator

This generator generates a controller and views that implement CRUD operations for the specified data model.

Fields with * are required. Click on **highlighted fields** to edit them.

Model Class *

Controller ID *

Base Controller Class *

Code Template *

在 Model Class 输入框中，输入 User。在 Controller ID 输入框中，输入 user (小写格式)。现在点击 Generate 按钮后的 Preview 按钮。CRUD 代码生成完成了。

访问 CRUD 页面

让我们看看成果，访问如下 URL：

```
http://hostname/testdrive/index.php?r=user
```

这会显示一个 `tbl_user` 表中记录的列表。因为我们的表是空的，现在什么都没显示。

点击页面上的 **Create User** 链接，如果没有登录的话我们将被带到登录页。登录后，我们看到一个可供我们添加新用户的表单。完成表单并点击 **Create** 按钮，如果有任何输入错误的话，一个友好的错误提示将会显示并阻止我们保存。回到用户列表页，我们应该能看到刚才添加的用户显示在列表中。

重复上述步骤以添加更多用户。注意，如果一页显示的用户条目太多，列表页会自动分页。

如果我们使用 `admin/admin` 作为管理员登录，我们可以在如下 URL 查看用户管理页：

```
http://hostname/testdrive/index.php?r=user/admin
```

这会显示一个包含用户条目的漂亮表格。我们可以点击表头的单元格来对相应的列进行排序，而且它和列表页一样会自动分页。

实现所有这些功能不要我们编写一行代码！

用户管理页

My Web Application































[Home](#) [About](#) [Contact](#) [Logout \(admin\)](#)

[Home](#) » [Users](#) » Manage

Manage Users

[List User](#) [Create User](#)

Displaying 11-20 of 21 result(s).

Id	Username	Password	Email	
11	test11	pass11	test11@example.com	  
12	test12	pass12	test12@example.com	  
13	test13	pass13	test13@example.com	  
14	test14	pass14	test14@example.com	  
15	test15	pass15	test15@example.com	  
16	test16	pass16	test16@example.com	  
17	test17	pass17	test17@example.com	  
18	test18	pass18	test18example.com	  
19	test19	pass19	test19example.com	  
20	test20	pass20	test20@example.com	  

Go to page: [< Previous](#) [1](#) [2](#) [3](#) [Next >](#)

Copyright © 2010 by My Company.
All Rights Reserved.
Powered by [Yii Framework](#)

新增用户页

My Web Application

[Home](#) [About](#) [Contact](#) [Logout \(admin\)](#)

[Home](#) » [Users](#) » Create

Create User

*Fields with * are required.*

Please fix the following input errors:

- Password cannot be blank.
- Email cannot be blank.

Username *

test

Password *

Password cannot be blank.

Email *

Email cannot be blank.

Create

Operations

List User

Manage User

Copyright © 2010 by My Company.
All Rights Reserved.
Powered by [Yii Framework](#).

基本概念

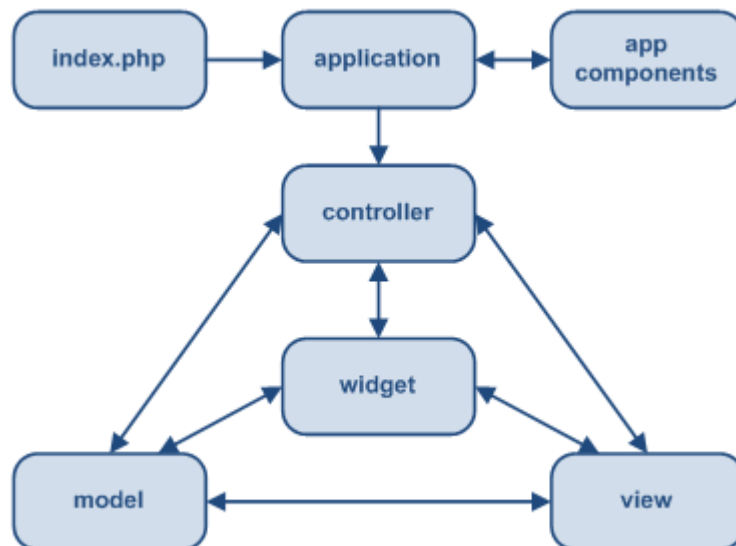
模型-视图-控制器 (MVC)

Yii 实现了 Web 编程中广为采用的“模型-视图-控制器”(MVC)设计模式。MVC 致力于分离业务逻辑和用户界面，这样开发者可以很容易地修改某个部分而不影响其它。在 MVC 中，模型表现信息（数据）和业务规则；视图包含用户界面中用到的元素，比如文本、表单输入框；控制器管理模型和视图间的交互。

除了 MVC，Yii 还引入了一个叫做 `application` 的前端控制器，它表现整个请求过程的运行环境。`Application` 接收用户的请求并把它分发到合适的控制器作进一步处理。

下图描述了一个 Yii 应用的静态结构：

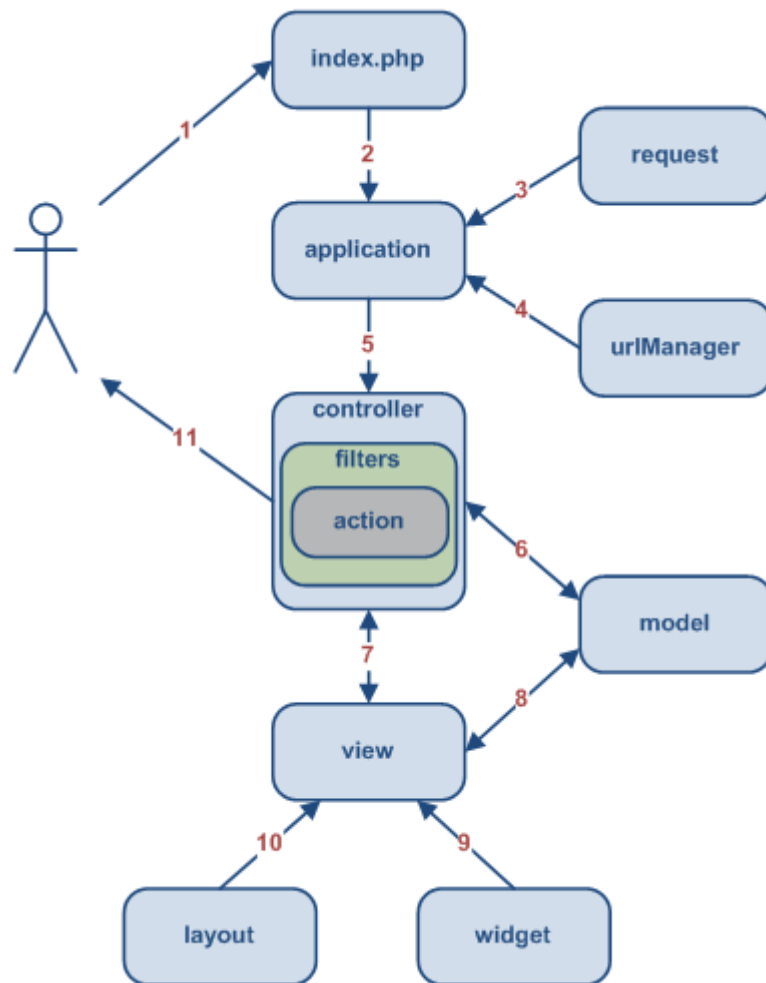
Yii 应用的静态结构



一个典型的处理流程

下图描述了一个 Yii 应用处理用户请求时的典型流程：

Yii 应用的典型流程



1. 用户访问 `http://www.example.com/index.php?r=post/show&id=1`, Web 服务器执行入口脚本 `index.php` 来处理该请求。
2. 入口脚本建立一个 [应用](#) 实例并运行之。
3. 应用从一个叫 `request` 的 [应用组件](#) 获得详细的用户请求信息。
4. 在名为 `urlManager` 的应用组件的帮助下, 应用确定用户要请求的 [控制器](#) 和 [动作](#)。
5. 应用建立一个被请求的控制器实例来进一步处理用户请求, 控制器确定由它的 `actionShow` 方法来处理 `show` 动作。然后它建立并应用和该动作相关的过滤器 (比如访问控制和性能测试的准备工作), 如果过滤器允许的话, 动作被执行。
6. 动作从数据库读取一个 ID 为 1 的 Post [模型](#)。
7. 动作使用 Post 模型来渲染一个叫 `show` 的 [视图](#)。
8. 视图读取 Post 模型的属性并显示之。
9. 视图运行一些 [widget](#)。
10. 视图的渲染结果嵌在 [布局](#) 中。
11. 动作结束视图渲染并显示结果给用户。

入口脚本

入口脚本是在前期处理用户请求的引导脚本。它是唯一一个最终用户可以直接请求运行的 PHP 脚本。

大多数情况下，一个 Yii 应用的入口脚本只包含如下几行：

```
// 部署到正式环境时去掉下面这行

defined('YII_DEBUG') or define('YII_DEBUG',true);

// 包含 Yii 引导文件

require_once('path/to/yii/framework/yii.php');

// 建立应用实例并运行

$configFile='path/to/config/file.php';

Yii::createWebApplication($configFile)->run();
```

这段代码首先包含了 Yii 框架的引导文件 `yii.php`，然后它配合指定的配置文件建立了一个 Web 应用实例并运行。

调试模式

一个 Yii 应用能够根据 `YII_DEBUG` 常量的指示以调试模式或者生产模式运行。默认情况下该常量定义为 `false`，代表生产模式。要以调试模式运行，在包含 `yii.php` 文件前将此常量定义为 `true`。应用以调试模式运行时效率较低，因为它会生成许多内部日志。从另一个角度来看，发生错误时调试模式会产生更多的调试信息，因而在开发阶段非常有用。

应用

应用是指执行用户的访问指令。其主要任务是解析用户指令，并将其分配给相应的控制器以进行进一步的处理。应用同时也是一个存储参数的地方。因为这个原因，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 [名称](#) 和 [默认控制器](#) 属性。

```
array(  
  
    'name'=>'Yii Framework',  
  
    'defaultController'=>'site',  
  
)
```

我们一般将配置保存在一个单独的 PHP 代码里(如 `protected/config/main.php`)。在这个代码里，我们返回以下参数数组，

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

为执行这些配置，我们一般将这个文件作为一个配置，传递给应用的构造器。或者像下述例子这样传递给 `Yii::createWebApplication()` 我们一般在 [入口脚本](#) 里指定这些配置：

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

提示: 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 include()` 把其它 配置文件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，这个目录一般是入口代码所在目录的一个目录：`protected`。这个路径可以通过在 [application configuration](#) 里设置 `basePath` 来改变。

普通用户不应该能够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。`.htaccess` 的文件内容是这样的：

```
deny from all
```

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。一个应用可以有很多应用组件，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。

通过配置 `application` 的 `components` 属性，我们可以自定义一些组件的类及其参数。比如说，我们可以配置 `CMemCache` 组件以使用服务器的内存当缓存：

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

在上述例子中，我们将 `cache` 元素加在 `components` 数组里。这个 `cache` 告诉我们这个组件的类是 `CMemCache`，其 `servers` 属性应该这样初始化。

要调用这个组件，用这个命令：`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。

提示：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 列上：`preload`。

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信息，如网址，[cookie](#)。在几乎每一个方面，通过配置这些核心组件的属性，我们都可以更改 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](#): [CErrHandler](#) - 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：

1. 使用 [CApplication::preinit\(\)](#) 预初始化应用。
2. 建立类自动加载器和错误处理；
3. 注册核心应用组件；
4. 载入应用配置；
5. 用 [CApplication::init\(\)](#) 初始化应用程序。
 - 注册应用行为；
 - 载入静态应用组件；
6. 触发 [onBeginRequest](#) 事件；
7. 处理用户请求：
 - 解析用户请求；
 - 创建控制器；
 - 执行控制器；
8. 触发 [onEndRequest](#) 事件；

控制器

控制器是 [CController](#) 或者其子类的实例。用户请求应用时，创建控制器。 控制器执行请求 `action`，`action` 通常引入必要的模型并提供恰当的视图。 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
{
}
```

路由

控制器和 `actions` 通过 ID 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 `action` 前缀构成 `action method`。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 `route`。一个 `route` 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, `route` 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,通过在应用配置中设置 [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](#)。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 [CWebApplication::catchAllRequest](#) 被指定, 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。此主要用来将 `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, 一个 404 [CHttpException](#) 将被唤起(raised).

一旦 [modules](#) 被使用(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` 开始. 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
{
    public function run()
    {
        // place the action logic here
    }
}
```

要让 controller 知道此 action 的存在, 我们重写 controller 类的 [actions\(\)](#) 方法:

```
class PostController extends CController
{
    public function actions()
    {
        return array(
            'edit'=>'application.controllers.post.UpdateAction',
        );
    }
}
```

```
}  
  
}
```

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/  
  
    controllers/  
  
        PostController.php  
  
        UserController.php  
  
    post/  
  
        CreateAction.php  
  
        ReadAction.php  
  
        UpdateAction.php  
  
    user/  
  
        CreateAction.php  
  
        ListAction.php  
  
        ProfileAction.php  
  
        UpdateAction.php
```

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

`$filterChain` 是 [CFilterChain](#) 的一个实例, `CFilterChain` 代表了与被请求的 `action` 相关的 `filter` 列表. 在此 `filter` 方法内部, 我们可以调用 `$filterChain->run()` 以继续 `filtering` 和 `action` 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
{
    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
    }
}
```

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
{
```

```

.....

public function filters()
{
    return array(
        'postOnly + edit, create',
        array(
            'application.filters.PerformanceFilter - edit, create',
            'unit'=>'second',
        ),
    );
}
}

```

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 应当和不当被应用. 在上面的例子中, `postOnly` 被应用到 `edit` 和 `create` action, 而 `PerformanceFilter` 应被应用到所有的 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.

模型

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 从表单模型和 active record 实现了两种模型，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 [使用表单](#)。

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。

视图

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 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(

    '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。

```
.....header here.....

<?php echo $content; ?>
```

```
.....footer here.....
```

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。要渲染一个视图而不应用任何布局，换用 [renderPartial\(\)](#)。

Widget

`widget` 是 [CWidget](#) 或其子类的实例。它是一个主要用于呈现目的的组件。`widget` 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 `widget` 可以用于渲染一个复杂的日历界面。`widget` 可以在用户界面上更好的实现重用。

要使用一个 `widget`，在一个 `view` 脚本中这样做：

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

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

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

或者

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

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。正如下所示：

```
<?php

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

    'mask'=>'99/99/9999'

));
```

```
?>
```

继承 [CWidget](#) 以及重载它的 [init\(\)](#) 和 [run\(\)](#) 方法，可以定义一个新的 widget:

```
class MyWidget extends CWidget
{
    public function init()
    {
        // this method is called by CController::beginWidget()
    }

    public function run()
    {
        // this method is called by CController::endWidget()
    }
}
```

Widget 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

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。

组件

Yii 应用构建于组件之上。组件是 [CComponent](#) 或其子类的实例。使用部件主要就是涉及访问其属性和唤起/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

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

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()

{

    return $this->_textWidth;

}


public function setTextWidth($value)

{

    $this->_textWidth=$value;

}
```

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一个好处：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。

注意：通过 **getter/setter** 方法和通过定位类里的一个成员变量来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。

我们可以附加一个方法到此 **event**，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)

{

    .....

}
```

这里的 **\$event** 是描述事件(源于 `raiseEvent()` 调用的)的参数。**\$event** 参数是 [CEvent](#) 或其源类(derived)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) {

    .....

}
```

如果我们现在调用了 `onClicked()`, `onClicked` 事件将被唤起(`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, 首先通过调用行为的 [attach\(\)](#) 方法绑定到一个组件是必须的. 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.

$behavior->attach($name, $component);

// test() 是一个方法或者行为

$component->test();
```

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` 的行为被绑定到部件, 我们可以获得行为对象的引用:

```
$behavior=$component->tree;

// 相当于以下:

// $behavior=$component->asa('tree');
```

行为是可以被临时禁止的, 此时它的方法开就会在组件中失效. 例如:


```
$component->disableBehavior($name);
```

```
// 以下语句将抛出一个异常
```

```
$component->test();
```

```
$component->enableBehavior($name);
```

```
// 当前可用
```

```
$component->test();
```

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.

模块

注意：对于模块的支持从版本 1.0.3 开始。

一个模块是一个自足的软件单元,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[应用的基础目录](#)。下面展示一个名为 `forum` 的典型目录结构:

```
forum/
```

```
    ForumModule.php           the module class file
```

```
    components/               containing reusable user components
```

<code>views/</code>	containing view files for widgets
<code>controllers/</code>	containing controller class files
<code>DefaultController.php</code>	the default controller class file
<code>extensions/</code>	containing third-party extensions
<code>models/</code>	containing model class files
<code>views/</code>	containing controller view and layout files
<code>layouts/</code>	containing layout view files
<code>default/</code>	containing view files for DefaultController
<code>index.php</code>	the index view file

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` 指向模块 ID(或模块目录名)。模块类起到存储在模块代码中分享的信息的中心位置的作用。例如，我们可以使用 [CWebModule::params](#) 来存储 模块参数，并 使用 [CWebModule::components](#) 来在模块水平分享[应用组件](#)。

提示：我们可以使用 [Gii](#) 的模块生成器来创建一个新模块的基本骨架。

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

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(
```

```
.....

'modules'=>array(

    'forum'=>array(

        'postPerPage'=>20,

    ),

),

.....

);
```

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水平分享的信息。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;

// or the following if $this refers to the controller instance

// $postPerPage=$this->module->postPerPage;
```

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`。

提示：若一个控制器是 `controllers` 的子目录中，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`。

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 [模块](#),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:

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

[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.

小贴士: 当调用一个通过 Yii 框架定义的类时, 我们不必导入或者包含它.所有的 Yii 核心类都是被预导入的.

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.

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

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.

小贴士: 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.

约定

Yii 遵循约定大于配置原则。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 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）。

[CUrlManager](#) 的帮助下，有可能生成和识别更多的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

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

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

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

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 [模块](#)，每个表示为一个子目录。
- `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

大多数 Web 应用由数据库支撑的。最佳实践是，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 `singular` 或 `plural` 名字，但不要两者均采用。简化起见，我们推荐使用 `singular` 名字。
- 表名字可以使用前缀,如 `tbl_`。 当一个应用的表和其他应用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。

1. 创建目录结构。在前面的章节 [Creating First Yii Application](#) 写的 `yiic` 工具可以帮助我们快速完成这步。
2. 配置 `application`。就是修改 `application` 配置文件。这步有可能会写一些 `application` 组件(例如：用户组件)
3. 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。
4. 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步。
5. 实现 `actions` 和相应的 `views`。这是真正需要我们编写的工作。
6. 在 `controller` 类里配置需要的 `action filters`。
7. 如果需要主题功能，编写 `themes`。
8. 如果需要 `internationalization` 国际化功能，编写翻译语句。
9. 使用 `caching` 技术缓存数据和页面。
10. 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。

使用表单

使用表单

通过 HTML 表单收集用户数据是 Web 程序开发的主要工作.而设计表单,开发者往往需要使用已存在的数据或者默认值来填充表单,验证用户输入,为无效的输入展示恰当的错误信息,然后保存数据到持久存储器.Yii 使用了它的 MVC 架构,大大简化了这个工作流程.

使用 Yii 通常要按以下步骤来处理表单：

1. 创建一个模型类来描述需要被收集的数据字段;
2. 创建一个控制器动作代码来响应提交的表单;
3. 在视图脚本里创建一个表单关联到控制器动作.

在下一节,我们将详细介绍这些步骤的具体实现.

创建模型

在编写我们所需要的表单的 HTML 代码之前,我们先得决定我们要从用户那里获取什么样的数据,这些数据需要遵从怎样的规则.模型类可以用于记录这些信息.模型,如在 [Model](#) 部分被定义的,是用来保存,校验用户输入的中枢位置.

根据用户输入的用途,我们可以创建两类模型.如果用户的输入被收集,使用然后被丢弃了,我们应该创建一个 [form model](#); 如果用户的数据被收集,然后保存到数据库,我们则应该选择使用 [active record](#) 模型.这两种模型共享着定义了表单所需通用接口的基类 [CModel](#).

注意: 本章中我们主要使用表单模型的示例. 它也同样适用于 [active record](#) 模型.

定义模型类

以下我们会创建一个 LoginForm 模型类从一个登录页面收集用户数据.因为登录数据只用于 校验用户而不需要保存,所以我们创建的 LoginForm 是一个表单模型.

```
class LoginForm extends CFormModel
{
    public $username;

    public $password;

    public $rememberMe=false;
```



```
}
```

`LoginForm` 声明了三个属性: `$username`, `$password` 和 `$rememberMe`. 他们用于保存用户输入的用户名, 密码以及用户是否想记住它登录状态的选项. 因为 `$rememberMe` 有一个默认值 `false`, 其相应的选项框在表单初始化显示时是没有被选中的.

说明: 为了替代这些成员变量"属性"(properties)的叫法, 我们使用 *特性*(attributes) 一词来区分于普通属性. 特性是一种主要用于储存来自用户输入或来自数据库的数据的属性.

声明验证规则

一旦用户提交了他的表单, 模型获得了到位的数据, 在使用数据前我们需要确定输入是否是有效的. 这个过程被一系列针对输入有效性的校验规则来校验. 我们应该在返回一个规则配置数组的 `rules()` 方法中指定这一有效的规则.

```
class LoginForm extends CFormModel
{
    public $username;

    public $password;

    public $rememberMe=false;

    private $_identity;

    public function rules()
    {
        return array(
            array('username, password', 'required'),
            array('rememberMe', 'boolean'),
            array('password', 'authenticate'),
        );
    }
}
```

```

public function authenticate($attribute,$params)
{
    $this->_identity=new UserIdentity($this->username,$this->password);

    if(!$this->_identity->authenticate())

        $this->addError('password','Incorrect username or password.');
```

以上的代码中 `username` 和 `password` 都是必填的,`password` 将被校验,`rememberMe` 值的类型是 `boolean`.

每个通过 `rules()` 返回的规则必须遵照以下格式:

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

`AttributeList` 是需要通过规则校验的以逗号分隔的特性名称集的字符串; **校验器(Validator)** 指定了使用哪种校验方式; `on` 参数是规则在何种情况下生效的场景列表;附加选项是用来初始化相应校验属性值的"名称-值"的配对.

在一个校验规则中有三种方法可以指定 **校验器**. 第一, **校验器** 可以是模型类中的一个方法的名称,就像以上例子中的 `authenticate`. 校验方法(`validator method`)必须是以下结构:

```

/**
 * @param string 被验证的属性的名字
 * @param array 指定了校验规则
 */
public function ValidatorName($attribute,$params) { ... }
```

第二, **校验器** 可以是一个校验类的名称.当规则被应用时,一个校验类的实例将被创建用于执行实际的校验. 规则里的附加选项用于初始化实例中属性的初始值.校验类必须继承自 [CValidator](#).

注意: 当为一个 `active record` 指定规则时,我们可以使用名称为 `on` 的特别选项.这个选项可以是 `'insert'` 或者 `'update'` 以便只有当插入或者更新记录时,规则才会生效.如果没有设置,规则 将在任何 `save()` 被调用的时候生效.

第三, `Validator` 可以是一个指向一个预定义的校验类的别名.在以上的例子中, `required` 指向了 [CRequiredValidator](#), 它确保了特性的有效值不能为空.以下是预定义校验别名的一份完整的列表:

- `boolean`: [CBooleanValidator](#) 的别名, 确保属性的值是 [CBooleanValidator::trueValue](#) 或 [CBooleanValidator::falseValue](#).
- `captcha`: [CCaptchaValidator](#) 的别名, 确保了特性的值等于 [CAPTCHA](#) 显示出来的验证码.
- `compare`: [CCompareValidator](#) 的别名, 确保了特性的值等于另一个特性或常量.
- `email`: [CEmailValidator](#) 的别名, 确保了特性的值是一个有效的电邮地址.
- `default`: [CDefaultValueValidator](#) 的别名, 为特性指派了一个默认值.
- `exist`: [CExistValidator](#) 的别名, 确保属性值存在于指定的数据表字段中.
- `file`: [CFileValidator](#) 的别名, 确保了特性包含了一个上传文件的名称.
- `filter`: [CFilterValidator](#) 的别名, 使用一个 `filter` 转换属性.
- `in`: [CRangeValidator](#) 的别名, 确保了特性出现在一个预订的值列表里.
- `length`: [CStringValidator](#) 的别名, 确保了特性的长度在指定的范围内.
- `match`: [CRegularExpressionValidator](#) 的别名, 确保了特性匹配一个正则表达式.
- `numerical`: [CNumberValidator](#) 的别名, 确保了特性是一个有效的数字.
- `required`: [CRequiredValidator](#) 的别名, 确保了特性不为空.
- `type`: [CTypeValidator](#) 的别名, 确保了特性为指定的数据类型.
- `unique`: [CUniqueValidator](#) 的别名, 确保了特性在数据表字段中是唯一的.
- `url`: [CUrlValidator](#) 的别名, 确保了特性是一个有效的路径.

以下我们列出了使用预定义校验器的例子:

```
// username 不为空
array('username', 'required'),

// username 必须大于 3 小于 12 字节
array('username', 'length', 'min'=>3, 'max'=>12),

// 在注册场景中, password 必须和 password2 一样
array('password', 'compare', 'compareAttribute'=>'password2', 'on'=>'register'),

// 在登录场景中, password 必须被校验
array('password', 'authenticate', 'on'=>'login'),
```

安全的特性分配

在一个模型实例被创建之后,我们经常需要使用用户提交的数据填充到(`populate`)它的特性.使用下面的 `massive` 赋值可以方便的完成:

```
$model=new LoginForm;

if(isset($_POST['LoginForm']))

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

最后的语句被称为 *massive assignment*，它赋值每个 `$_POST['LoginForm']` 中的 entry 到相应的 model attribute. 而它相当于如下的赋值方式:

```
foreach($_POST['LoginForm'] as $name=>$value)

{

    if($name is a safe attribute)

        $model->$name=$value;

}
```

决定哪些 attribute 是安全的,这一点非常重要.例如,我们暴露(expose)一个数据表的主键是安全的,然后一个攻击者有机会改变给定记录的主键，因此可以篡改他未被授权的数据.

安全策略在版本 1.0 和 1.1 中是不同的。下面我们分别描述它们.

Safe Attributes in 1.1

在版本 1.1 中，一个 attribute 被认为是安全的,若它出现在一个适用于指定方案的验证规则中.例如,

```
array('username, password', 'required', 'on'=>'login, register'),

array('email', 'required', 'on'=>'register'),
```

在上面, username 和 password attribute 在 login 方案中是必需的, 而 username, password 和 email attribute 在 register 方案中是必需的. 结果, 若我们在 login 方案中执行一个 massive assign, 只有 username 和 password 被 massively assigned 因为只有它们是出现在 login 的验证规则中的 attribute. 换句话说, 若方案是 register, 所有三个 attribute 可被 massively assigned.

```
// in login scenario

$model=new User('login');

if(isset($_POST['User']))
```

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

// in register scenario

$model=new User('register');

if(isset($_POST['User']))

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

为什么我们采取这种策略来决定一个 `attribute` 是否安全?其背后的基本原理是:若一个 `attribute` 已经有一个或多个验证规则来检查它的有效性,还有什么可担心的?

重要的是记住验证规则被用来检查用户输入的数据而不是我们在代码中产生的数据(例如. `timestamp`, 自动产生的主键). 因此, 不要为这些不希望从最终用户提交的 `attributes` 增加验证规则.

有时, 我们想要声明一个 `attribute` 是安全的, 即使我们并没有为它指定任何规则. 一个例子是:一篇文章的 `content` attribute 可以接受任何用户输入(take any user input). 我们可以使用此特殊的安全规则来实现:

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

为了完整性,也有一个 `unsafe` 规则直接声明一个 `attribue` 是不安全的:

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

规则 `unsafe` 很少用到, 它是我们之前定义的 `safe attributes` 的一个例外.

Safe Attributes in 1.0

在版本 1.0 中,决定一个数据项是否是安全的,基于一个名为 `safeAttributes` 方法的返回值和数据项被指定的场景. 默认的,这个方法返回所有公共成员变量作为 [CFormModel](#) 的安全特性,而它也返回了除了主键外, 表中所有字段名作为 [CActiveRecord](#) 的安全特性.我们可以根据场景重写这个方法来限制安全特性 .例如, 一个用户模型可以包含很多特性,但是在 `login` 场景.里,我们只能使用 `username` 和 `password` 特性.我们可以按照如下来指定这一限制 :

```
public function safeAttributes()

{

    return array(
```

```

        parent::safeAttributes(),

        'login' => 'username, password',

    );
}

```

`safeAttributes` 方法更准确的返回值应该是如下结构的：

```

array(

    //这些属性可以在任意场景被大量分配的

    //以下特性并没有被明确的分配

    'attr1, attr2, ...',

    *

    //以下特性只可以在场景 1 中被大量分配的

    'scenario1' => 'attr2, attr3, ...',

    *

    //以下特性只可以在场景 2 中被大量分配的

    'scenario2' => 'attr1, attr3, ...',

)

```

如果模型不是场景敏感的(比如,它只在一个场景中使用,或者所有场景共享了一套同样的安全特性),返回值可以是如下那样简单的字符串.

```

'attr1, attr2, ...'

```

而那些不安全的数据项,我们需要使用独立的赋值语句来分配它们到相应的特性.如下所示:

```

$model->permission='admin';

$model->id=1;

```

触发校验

一旦用户提交的数据到位,我们可以调用 [CModel::validate\(\)](#) 来触发数据校验处理.这个方法 返回了一个指示校验是否成功的值. 而 [CActiveRecord](#) 中的校验可以在我们调用它的 [CActiveRecord::save\(\)](#) 方法时自动触发.

使用属性 `scenario` 我们可以设置一个场景, 此外指出哪个验证规则集合被应用.

验证在一个场景中被执行. 属性 `scenario` 指定了哪个 `scenario` 此 `model` 被使用以及哪个验证集合被使用. 例如, 在 `login scenario`, 我们只想验证用户模型的 `username` 和 `password` 输入; 而在 `register` 场景中,我们需要验证更多输入, 例如 `email`, `address` 等. 下面的例子展示了如何在 `register` 场景中执行验证:

```
// creates a User model in register scenario. It is equivalent to:
```

```
// $model=new User;
```

```
// $model->scenario='register';
```

```
$model=new User('register');
```

```
// populates the input values into the model
```

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

```
// performs the validation
```

```
if($model->validate()) // if the inputs are valid
```

```
    ...
```

```
else
```

```
    ...
```

一个规则适用的场景可以使用规则中的选项 `on` 来指定.若 `on` 没有设置, 意味这此规则适用于所有场景. 例如

```
public function rules()
```

```
{
```

```
    return array(
```

```
array('username', 'password', 'required'),  
  
array('password_repeat', 'required', 'on'=>'register'),  
  
array('password', 'compare', 'on'=>'register'),  
  
);  
  
}
```

第一条规则在所有场景生效,而接下来的两条规则只有在 **register** 场景中生效.

检索校验错误

一旦验证完成,任何存在的错误将存储在对象 **model** 中.我们可以通过调用 [CModel::getErrors\(\)](#) 和 [CModel::getError\(\)](#) 来检索这些信息.二者的不同点是前者返回所有信息,而后者只返回第一条错误信息.

特性标签

当设计一个表单时,我们通常需要为每个输入字段显示标签. 标签告诉了用户他被期望输入哪种信息.尽管我们可以在视图里使用硬性编码,但是如果我们在对应的模型里指定了标签,那么它将提供更强的弹性和更好的 便利性.

[CModel](#) 会默认地返回特性的名称作为特性的标签.而通过重写 [attributeLabels\(\)](#) 方法,可以实现标签的定制.在接下来章节中我们将看到,在模型里指定标签将允许我们创建一个更快捷更强大 的表单.

创建动作

一旦有了 **model**, 我们可以开始编写操作 **model** 的逻辑.我们把这些逻辑放在 **controller action** 里面.用录入登陆表单这个例子来说明, 如下是需要的代码:

```
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，如果无效则重定位到前个页面

if($model->validate())

    $this->redirect(Yii::app()->user->returnUrl);

}

// 显示登陆表单

$this->render('login',array('model'=>$model));
}

```

在上面，我们首先创建一个 `LoginForm` 实例；如果请求是 `POST` 方式（意味着登陆表单被提交），我们以提交的数据 `$_POST['LoginForm']` 填充(populate) `$model`；然后验证输入，如果成功，把用户请求 `url` 定位到相应需要授权的页面。如果验证失败，或者是第一次访问 `login` 页面的，把用户请求 `url` 定位到 `login` 的页面，`login` 页面具体怎么写会在下一个小节里描写。

提示: 在 `login action` 里面，我们用 `Yii::app()->user->returnUrl` 获取之前需要验证的 `url`。组件 `Yii::app()->user` 是一个 [CWebUser](#) (或其子类) 的实例，它主要用来存放用户 `session` 信息的（例如：用户名，状态等）。想要了解更多，看 [身份验证和授权](#) 这章。

大家注意这段在 `login action` 里面的 `php` 语句：

```

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

```

如我们在 [Securing Attribute Assignments](#) 提到，这行代码用用户提交的数据填充此 `model`。 [CModel](#) 里面以 `name-value` 数组形式定义了 `attributes` 属性，每个 `value` 被分配到相应的 `name` 属性上。所以如果 `$_POST['LoginForm']` 给了我们这样的数组，上面的代码将等同于后面的这长串代码（假设每个需要的属性这个数组都提供）：

```

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

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

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

```

提示: 为了让 `$_POST['LoginForm']` 不提供字符串而是数组，根据惯例 `view` 页面的输入字段应该写 `model` 相应的名字。记住是，一个页面输入字段对应 `model`（简称 `C`）里面的一个属性 `C[a]`。例如，我们用 `LoginForm[username]` 去命名页面 `username` 输入字段。

剩下的工作是编写 `login` 视图了，编写里面的 `html` 表单和相应的输入字段。

创建表单

编写 `login` 视图是直截了当的。我们以 `form` 标签开头，`form` 标签的 `action` 属性应该是 `login` 行为之前描述的 URL。然后我们插入在 `LoginForm` 类中声明过的标签和文本框。最后我们插入一个用于用户点击后提交表单的按钮。所有这些都可以使用纯 `HTML` 代码来完成。

Yii 提供了一些辅助器(helper)类来简化视图编写。例如，创建一个文本输入框，我们可以调用 [CHtml::textField\(\)](#)；创建一个下拉菜单，则可调用 [CHtml::dropDownList\(\)](#)。

信息：人们可能不知道在编写类似代码时使用辅助器比使用纯 `HTML` 编写代码好处是什么。例如，如下代码将生成一个当其值被用户改变时可以触发表单提交的文本输入框。

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

否则在任何需要的地方都要写上那笨拙的 `JavaScript` 了。

如下，我们使用 [CHtml](#) 来创建登陆表单。我们假设变量 `$model` 代表 `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>

<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>

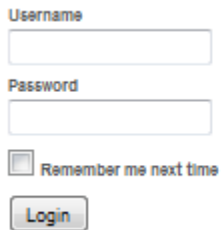
<?php echo CHtml::endForm(); ?>

</div><!-- form -->
```

以上代码生成了一个更动态的表单。例如 [CHtml::activeLabel\(\)](#) 生成了一个关联到指定模型特性的标签。如果这个特性有一个输入错误, 标签 CSS 的 class 将变成改变标签视觉表现到相应 CSS 样式的 `error`。类似的, [CHtml::activeTextField\(\)](#) 为指定的模型特性生成了一个文本输入框, 其 CSS 的 class 也会在发生任何错误时变成 `error`。

如果我们使用了 `yiic` 脚本提供的 CSS 样式文件 `form.css`, 那么生成的表单和如下显示的差不多：

登陆页面



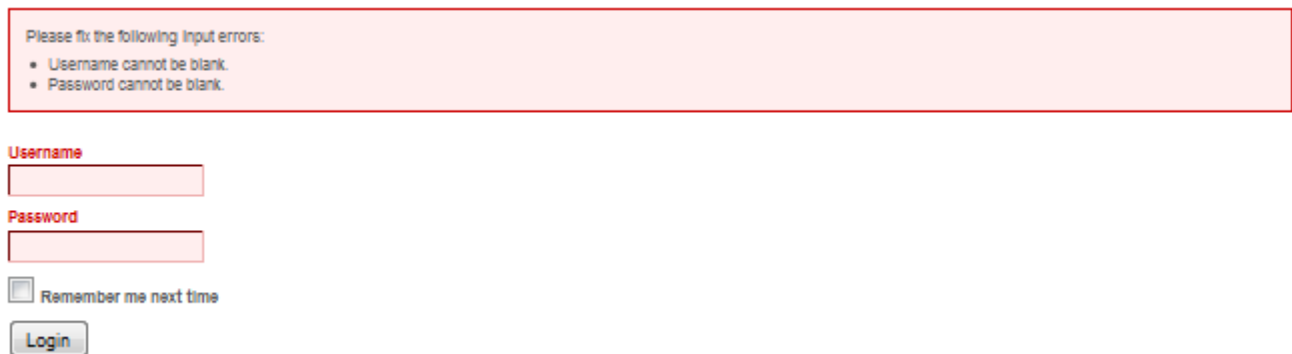
Username

Password

☐ Remember me next time

Login

登陆出错页面



Please fix the following input errors:

- Username cannot be blank.
- Password cannot be blank.

Username

Password

☐ Remember me next time

Login

自版本 1.1.1 开始, 一个新的 widget 称为 [CActiveForm](#) 被提供用来简化表单创建。此 widget 能够无缝和坚固 (consistent) 的支持客户端和服务端端的验证。使用 [CActiveForm](#), 上面的视图代码可以如下编写:

```
<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 -->
```

收集表格输入

有时候我们想按批收集用户输入.也就是,用户可以为多个模型实例输入信息然后一次性提交全部.我们之所以把这个称之为表格输入(*tabular input*) 是因为输入的字段通常出现在一个 HTML 表格里.

要使用表格输入,我们首先需要创建或者使用模型实例填充一个数组,这取决于我们是插入还是更新数据.然后我们从 `$_POST` 变量里取出用户输入的数据,再将他们分配到各个模型中.这和从单模型输入中取出数据有一点微小的差异,那就是我们使用 `$_POST['ModelClass'][$i]` 取出数据而不是 `$_POST['ModelClass']`.

```
public function actionBatchUpdate()
{
    // 批处理模式中,收集用于更新的项

    // 假定每项都是模型类 'Item' 的

    $items=$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) // 所有的项都是有效的

            // ... 在这里干点什么
```

```

    }

    // 显示视图收集表格输入

    $this->render('batchUpdate',array('items'=>$items));
}

```

准备好了动作,我们需要 batchUpdate 视图在一个 HTML 表中显示输入框.

```

<div class="yiiForm">

<?php echo CHtml::beginForm(); ?>

<table>

<tr><th>名称</th><th>价格</th><th>数量</th><th>描述</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><!-- yii 表单 -->

```

注意：在上述代码中,我们使用了 "[*i*]name" 代替了 "name" 来作为 [CHtml::activeTextField](#) 的第二参数.

如果有任何校验错误,那么对应的字段将会自动高亮,就像我们先前提到的单模型输入一样.

使用 Form Builder

当创建 HTML 表单时,经常我们发现我们在写很多重复而且在不同项目中很难重用的视图代码。例如,对于每个输入框,我们需要以一个文本标签和显示可能的验证错误来关联它。为了改善这些代码的重用性,我们可以使用自版本 1.1.0 可用的 form builder 特征。

基本概念

Yii form builder 使用一个 CForm 对象来呈现一个 HTML 表单,包括此表单关联哪些数据模型,在此表单中有哪些种输入框,以及如何渲染整个表单。开发者主要需要创建和配置此 [CForm](#) 对象,然后调用它的渲染方法来显示此表单。

表单输入的规格被组织为一个分层的表单元素。在分层的根部,它是 CForm 对象。根表单对象维护着它的 children 为两个集合: [CForm::buttons](#) 和 [CForm::elements](#)。前者包含按钮元素(例如提交按钮,重设按钮),而后者包含输入元素,静态文本和子表单。子表单是一个包含在另外一个表单 [CForm::elements](#) 中的 [CForm](#) 对象。它可以有自己的数据模型, [CForm::buttons](#) 和 [CForm::elements](#) 集合。

当用户提交一个表单,整个表单输入框中填写的值被提交,包含属于子表单的输入框。[CForm](#) 提供了方便的方法,可以自动赋值输入数据到相应的模型属性并执行数据验证。

创建一个简单的表单

下面,我们展示如何使用 form builder 来创建一个登录表单。

首先,我们编写登录 action 代码:

```
public function actionLogin()
{
    $model = new LoginForm;

    $form = new CForm('application.views.site.loginForm', $model);

    if($form->submitted('login') && $form->validate())

        $this->redirect(array('site/index'));

    else
```



```
$this->render('login', array('form'=>$form));  
  
}
```

上面的代码中，我们使用由路径别名 `application.views.site.loginForm` 指定的规格创建了一个 [CForm](#) 对象。此 [CForm](#) 对象和在[创建模型](#)描述的 `LoginForm` 模型关联。

如代码所示，若表单被提交并且所有的输入经过了验证而没有错误，我们将转向用户的浏览器到 `site/index` 页面。否则，我们使用此表单渲染登录页面。

路径别名 `application.views.site.loginForm` 实际指向 PHP 文件 `protected/views/site/loginForm.php`。此文件应返回一个代表了 [CForm](#) 所需配置的数组，如下所示：

```
return array(  
  
    'title'=>'Please provide your login credential',  
  
    'elements'=>array(  
  
        'username'=>array(  
  
            'type'=>'text',  
  
            'maxlength'=>32,  
  
        ),  
  
        'password'=>array(  
  
            'type'=>'password',  
  
            'maxlength'=>32,  
  
        ),  
  
        'rememberMe'=>array(  

```

```
        'type'=>'checkbox',
    )
),

'buttons'=>array(

    'login'=>array(

        'type'=>'submit',

        'label'=>'Login',

    ),

),

);
```

配置是一个由 name-value 对组成的关联数组，被用来初始化 [CForm](#) 相应的属性。要配置的最重要的属性是 [CForm::elements](#) 和 [CForm::buttons](#)。它们的每一个是一个指定了表单元素列表的数组。在下一小节我们将给出更多细节关于如何配置表单元素。

最后，我们编写登录视图脚本，如下，

```
<h1>Login</h1>

<div class="form">

<?php echo $form; ?>

</div>
```

提示：上面的代码 `echo $form;` 相当于 `echo $form->render();`。这是因为 [CForm](#) 执行 `__toString` 魔术方法。它调用 `render()` 并返回它的结果为代表此表单对象的字符串。

指定表单元素

使用 `form builder`，我们大部分的工作由编写视图脚本代码转为指定表单元素。在这一小节中，我们描述如何指定 [CForm::elements](#) 属性。我们不准备描述 [CForm::buttons](#) 因为它的配置非常类似于 [CForm::elements](#)。

[CForm::elements](#) 属性接受一个数组作为它的值。每个数组元素指定了一个单独的表单元素，这个表单元素可以是一个输入框，一个静态文本字符串或一个子表单。

指定输入元素

一个输入元素主要由一个标签，一个输入框，一个提示文字和一个错误显示组成。它必须和一个模型属性关联。一个输入元素的规格被代表为一个 [CFormInputElement](#) 实例。[CForm::elements](#) 数组中的如下代码指定了一个单独的输入元素：

```
'username'=>array(  
  
    'type'=>'text',  
  
    'maxlength'=>32,  
  
),
```

它说明模型属性被命名为 `username`，输入框的类型为 `text`，它的 `maxlength` 属性为 `32`。我们可以在上面的数组中指定另外的 选项只要它们是 [CFormInputElement](#) 的可写属性。例如，我们可以指定 [hint](#) 选项以便显示一条提示信息，或者我们可以指定 [items](#) 选项若它的输入类型是一个 `list box`，一个下拉列表，一个多选框列表或一个单选按钮列表。

`type` 选项需要注意。它指定输入框的类型。例如，类型 `text` 意味着渲染一个普通的文本输入框；`password` 类型意味着一个密码输入框。[CFormInputElement](#) 识别如下内置的类型：

- `text`
- `hidden`
- `password`
- `textarea`
- `file`
- `radio`
- `checkbox`
- `listbox`
- `dropdownlist`

- checkboxlist
- radiolist

除了这些内置的类型，选项 `type` 也可以使用一个 `widget` 类名字或 `widget` 类的路径别名。 `widget` 类必须扩展自 [CInputWidget](#)。当渲染输入元素时，一个指定 `widget` 类的实例将被创建并渲染。The widget will be configured using the specification as given for the input element.

指定静态文本

很多情况下，一个表单包含一些装饰性的 HTML 代码。例如，一个水平线被用来 分隔表单中不同的部分；一个图像出现在特定的位置来增强表单的视觉外观。 我们可以在 [CForm::elements](#) 集合中指定这些 HTML 代码作为静态文本。要这样做， 我们只要指定一个静态文本字符串作为一个数组元素，在 `CForm::elements` 恰当的位置。例如，

```
return array(  
    'elements'=>array(  
        .....  
        'password'=>array(  
            'type'=>'password',  
            'maxlength'=>32,  
        ),  
  
        '<hr />',  
  
        'rememberMe'=>array(  
            'type'=>'checkbox',  
        )  
    ),  
    .....  
);
```

```
);
```

在上面，我们在 `password` 输入和 `rememberMe` 之间插入一个水平线。

静态文本最好用于文本内容和它们的配置不规则时。若表单中的每个输入元素需要被相似的装饰，我们应当定制表单渲染方法，此章节将简短介绍。

指定子表单

子表单被用来分离一个长的表单为几个逻辑部分。例如，我们可以分离用户注册表单为两部分：登录信息和档案信息。每个子表单和一个数据模型有无关联均可。例如在用户注册表单，若我们存储用户登录信息和档案信息到两个分离的数据表中(表示为两个数据模型)，然后每个子表单 需要一个对应的数据模型关联。若我们存储所有信息到一个数据表中，任意一个子表单都没有数据模型因为它们和根表单分享相同的模型。

一个子表单也呈现为一个 [CForm](#) 对象。要指定一个子表单，我们应当以类型为 `form` 的一个元素配置此 [CForm::elements](#) 属性：

```
return array(  
  
    'elements'=>array(  
        .....  
        'user'=>array(  
  
            'type'=>'form',  
            'title'=>'Login Credential',  
            'elements'=>array(  
  
                'username'=>array(  
                    'type'=>'text',  
                ),  
                'password'=>array(  

```

```
        'type'=>'password',

    ),

    'email'=>array(

        'type'=>'text',

    ),

),

'profile'=>array(

    'type'=>'form',

    .....

),

    .....

),

    .....

);
```

类似于配置一个根表单，我们主要需要为一个子表单指定 [CForm::elements](#) 属性。若一个子表单需要 被一个数据模型关联，我们也可以配置它的 [CForm::model](#) 属性。

有时，我们想要不使用默认的 [CForm](#) 类来呈现一个表单。例如， 此小节将简短展示，我们可以扩展 [CForm](#) 以定制表单渲染逻辑。通过指定输入元素的类型为 `form`，一个子表单将自动被表示为一个对象，它的类和它的父表单相同。若我们指定输入元素的类型类似于 `XYZForm` (一个字符串以 `Form` 结尾)， 然后子表单将被表示为一个 `XYZForm` 对象。

访问表单元素

访问表单元素和访问数组元素一样简单。[CForm::elements](#) 属性返回一个 [CFormElementCollection](#) 对象，它扩展自 [CMap](#) 并允许以类似于一个普通数组的方式来访问它的元素。例如，要访问登录表单中的元素 `username`，我们可以使用下面的代码：

```
$username = $form->elements['username'];
```

要访问 `user` 注册表单中的 `email` 元素，使用

```
$email = $form->elements['user']->elements['email'];
```

因为 [CForm](#) 为它的 [CForm::elements](#) 属性执行数组访问，上面的代码可以简化为：

```
$username = $form['username'];

$email = $form['user']['email'];
```

创建一个嵌套表单

我们已经描述了子表单。我们称一个有子表单的表单为一个嵌套表单。在这一章节，我们使用用户注册表单作为例子来展示如何创建一个关联多个数据模型的嵌套表单。我们假设用户的认证信息存储为一个 `User` 模型，而用户的档案信息被存储为一个 `Profile` 模型。

我们首先创建 `register` action 如下：

```
public function actionRegister()
{
    $form = new CForm('application.views.user.registerForm');

    $form['user']->model = new User;

    $form['profile']->model = new Profile;

    if($form->submitted('register') && $form->validate())
    {
```

```
$user = $form['user']->model;

$profile = $form['profile']->model;

if($user->save(false))

{

    $profile->userID = $user->id;

    $profile->save(false);

    $this->redirect(array('site/index'));

}

}

$this->render('register', array('form'=>$form));

}
```

在上面，我们使用由 `application.views.user.registerForm` 指定的配置创建了表单。在表单被提交且成功验证之后，我们尝试保存 `user` 和 `profile` 模型。我们通过访问相应子表单对象的 `model` 属性来检索 `user` 和 `profile` 模型。因为输入验证已经完成，我们调用 `$user->save(false)` 来跳过验证。为 `profile` 模型也这样做。

接下来，我们编写表单配置文件 `protected/views/user/registerForm.php`:

```
return array(

    'elements'=>array(

        'user'=>array(

            'type'=>'form',

            'title'=>'Login information',
```



```
'elements'=>array(  
  
    'username'=>array(  
        'type'=>'text',  
    ),  
    'password'=>array(  
        'type'=>'password',  
    ),  
    'email'=>array(  
        'type'=>'text',  
    )  
),  
  
'profile'=>array(  
  
    'type'=>'form',  
    'title'=>'Profile information',  
    'elements'=>array(  
  
        'firstName'=>array(  
            'type'=>'text',  
        ),  

```

```
'lastName'=>array(

    'type'=>'text',

),

),

),

),

'buttons'=>array(

    'register'=>array(

        'type'=>'submit',

        'label'=>'Register',

    ),

),

);
```

在上面，当指定每个子表单时，我们也指定它的 [CForm::title](#) 属性。The default form rendering logic will enclose each sub-form in a field-set which uses this property as its title.

最后，我们编写 `register` 视图脚本：

```
<h1>Register</h1>

<div class="form">
```

```
<?php echo $form; ?>

</div>
```

定制表单显示

使用 form builder 最主要的好处是逻辑 (表单配置被存储在一个单独的文件中) 和表现 ([CForm::render](#) 方法) 的分离。这样, 我们可以实现定制表单显示, 通过重写 `CForm::render` 或渲染一个部分视图。两种方法都可以保持表单配置的完整性, 并且可以容易的重用。

当重写 [CForm::render](#) 时, 你主要需要遍历 [CForm::elements](#) 和 [CForm::buttons](#) 并为每个表单元素调用 [CFormElement::render](#) 方法。例如,

```
class MyForm extends CForm
{
    public function render()
    {
        $output = $this->renderBegin();

        foreach($this->getElements() as $element)

            $output .= $element->render();

        $output .= $this->renderEnd();

        return $output;
    }
}
```

```
}
```

可能我们也需要写一个视图脚本 `_form` 以渲染一个视图:

```
<?php

echo $form->renderBegin();

foreach($form->getElements() as $element)

    echo $element->render();

echo $form->renderEnd();
```

要使用此视图脚本, 我们需要调用:

```
<div class="form">

$this->renderPartial('_form', array('form'=>$form));

</div>
```

若一个通用的表单渲染不适用于一个特殊的表单(例如, 表单为特定的元素需要不规则的装饰), 在视图脚本中我们可以这样做:

```
some complex UI elements here
```

```
<?php echo $form['username']; ?>
```

some complex UI elements here

```
<?php echo $form['password']; ?>
```

some complex UI elements here

在最后的方法中，the form builder 看起来并没有带来好处，因为我们仍然需要写很多表单代码。然而，它仍然是有好处的，表单被使用一个分离的配置文件指定，这样可以帮助开发者更专注于逻辑部分。

使用数据库

Working with Database(数据库开发工作)

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。

数据访问对象 (DAO)

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,改变数据库访问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](#): 代表一个 DB 事务处理。

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,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,一个异常将被唤起(例如,错误的 DSN 或无效的用户名/密码)。

```
$connection=new CDbConnection($dsn,$username,$password);

// establish connection. You may try...catch possible exceptions
```

```

$connection->active=true;

.....

$connection->active=false; // close connection

```

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。查看 [PDO documentation](#) 得到完整信息。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`

因为 [CDbConnection](#) 扩展自 [CApplicationComponent](#)，我们也可以使用它作为一个 [application component](#)。要这样做，db(或其他名字) 应用组件在 [application configuration](#) 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

        ),

    ),

)

```

然后我们可以通过已自动被激活的 `Yii::app()->db` 来访问此 DB 连接, 除非我们明确配置 `CDbConnection::autoConnect` 为 `false`. 使用此方法, 这个单一的 DB 连接可以在代码中的多处位置分享.

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);

// 若需要, SQL 语句可以被如下更新:

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

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.

若在 SQL 语句查询过程中出现错误, 一个异常将被唤起.

```
$rowCount=$command->execute(); // execute the non-query SQL

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

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

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

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

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

取出查询结果

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();
```



```
// calling read() repeatedly until it returns false

while(($row=$dataReader->read())!==false) { ... }

// using foreach to traverse through every row of data

foreach($dataReader as $row) { ... }

// retrieving all rows at once in a single array

$rows=$dataReader->readAll();
```

注意: 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).

上面的流程可以使用下面的代码来执行:

```
$transaction=$connection->beginTransaction();

try
{
    $connection->createCommand($sql1)->execute();

    $connection->createCommand($sql2)->execute();

    //.... other SQL executions

    $transaction->commit();
}

catch(Exception $e) // an exception is raised if a query fails
```

```
{  
  
    $transaction->rollBack();  
  
}
```

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"  
  
$sql="INSERT INTO tbl_user (username, email) VALUES(:username,:email)";  
  
$command=$connection->createCommand($sql);  
  
// replace the placeholder ":username" with the actual username value  
  
$command->bindParam(":username",$username,PDO::PARAM_STR);  
  
// replace the placeholder ":email" with the actual email value  
  
$command->bindParam(":email",$email,PDO::PARAM_STR);  
  
$command->execute();  
  
// insert another row with a new set of parameters  
  
$command->bindParam(":username",$username2,PDO::PARAM_STR);  
  
$command->bindParam(":email",$email2,PDO::PARAM_STR);  
  
$command->execute();
```

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

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";

$dataReader=$connection->createCommand($sql)->query();

// bind the 1st column (username) with the $username variable

$dataReader->bindColumn(1,$username);

// bind the 2nd column (email) with the $email variable

$dataReader->bindColumn(2,$email);

while($dataReader->read()!==false)

{

    // $username and $email contain the username and email in the current row

}
```

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}}` 指向表的名称, `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}}';

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

Active Record

虽然 Yii DAO 可以处理事实上任何数据库相关的任务，但很可能我们会花费 90%的时间 用来编写一些通用的 SQL 语句来执行 CRUD 操作（创建，读取，更新和删除）。同时我们也很难维护这些 PHP 和 SQL 语句混合的代码。要解决这个问题，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），其字段作为 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;

$post->title='sample post';

$post->content='post body content';

$post->save();
```

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 处理数据库中的关系。为了简单起见，我们使用本节下面的数据库表作为例子。请注意，如果你使用 MySQL 数据库，在下面的 SQL 中您应该替换 AUTOINCREMENT 为 AUTO_INCREMENT。

```
CREATE TABLE tbl_post (

    id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,

    title VARCHAR(128) NOT NULL,

    content TEXT NOT NULL,

    create_time INTEGER NOT NULL

);
```

注意: 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。下面的应用程序配置提供了一个例子：

```
return array(
```

```
'components'=>array(

    'db'=>array(

        'class'=>'system.db.CDbConnection',

        'connectionString'=>'sqlite:path/to/dbfile',

        // turn on schema caching to improve performance

        // 'schemaCachingDuration'=>3600,

    ),

),

);
```

提示： 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](#)

注意： 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。

提示： 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， 您可以以不同的 [getDbConnection\(\)](#) 来创建不同的 AR 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
```

```
{

    public static function model($className=__CLASS__)

    {

        return parent::model($className);

    }

    public function tableName()

    {

        return 'tbl_post';

    }

}
```

提示: 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(

    'import'=>array(

        'application.models.*',

    ),

);
```

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. 方法 `model()` is declared as such for every AR class (to be explained shortly).

信息: 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,

这样,不是返回一个完整的表名,我们返回去掉了前缀的表名,并把它环绕在双弯曲括号中。

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

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.

信息: 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, and we must quote a column in a query condition if the column contains mixed-case letters.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 类指定哪些字段应当为主键，通过重写 `primaryKey()` 方法，

```
public function primaryKey()  
  
{  
  
    return 'id';  
  
    // For composite primary key, return an array like the following  
  
    // return array('pk1', 'pk2');
```

```
}
```

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;

$post->title='sample post';

$post->content='content for the sample post';

$post->create_time=time();

$post->save();
```

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

{

    public $title='please enter a title';

    .....

}

$post=new Post;
```



```
echo $post->title; // this would display: please enter a title
```

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;

$post->create_time=new CDbExpression('NOW()');

// $post->create_time='NOW()'; will not work because
// 'NOW()' will be treated as a string

$post->save();
```

提示: 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

$post=Post::model()->find($condition,$params);

// find the row with the specified primary key

$post=Post::model()->findByPrimaryKey($postID,$condition,$params);

// find the row with the specified attribute values

$post=Post::model()->findByAttributes($attributes,$condition,$params);
```

```
// find the first row using the specified SQL statement
```

```
$post=Post::model()->findBySql($sql,$params);
```

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 被用来访问类水平的方法(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
```

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

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不限于 `WHERE` 子语句. 例如,

```
$criteria=new CDbCriteria;
```

```
$criteria->select='title'; // only select the 'title' column
```

```
$criteria->condition='postID=:postID';
```

```
$criteria->params=array(':postID'=>10);
```

```
$post=Post::model()->find($criteria); // $params is not needed
```

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(

    'select'=>'title',

    'condition'=>'postID=:postID',

    'params'=>array(':postID'=>10),

));
```

信息: 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

$post=Post::model()->findAll($condition,$params);

// find all rows with the specified primary keys

$post=Post::model()->findAllByPk($postIDs,$condition,$params);

// find all rows with the specified attribute values

$post=Post::model()->findAllByAttributes($attributes,$condition,$params);

// find all rows using the specified SQL statement

$post=Post::model()->findAllBySql($sql,$params);
```

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

$n=Post::model()->count($condition,$params);

// get the number of rows using the specified SQL statement

$n=Post::model()->countBySql($sql,$params);

// check if there is at least a row satisfying the specified condition

$exists=Post::model()->exists($condition,$params);
```

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);

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

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

若我们所见, 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; 若此 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 提供如下方便的类水平的(class-level)方法来实现它:

```
// update the rows matching the specified condition

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

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

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

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

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

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

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

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.

下面类水平的(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

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

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

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

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())
{
    // data is valid and is successfully inserted/updated
}

else
{
    // data is invalid. call getErrors() to retrieve error messages
}
```

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

$post->save();
```

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 [Securing Attribute Assignments](#) 部分和 [Creating Action](#) 部分找到。

```
// assume $_POST['Post'] is an array of column values indexed by column names

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

$post->save();
```

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()`.

信息: 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.

Customization

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.

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();

$transaction=$model->dbConnection->beginTransaction();

try
{
    // find and save are two steps which may be intervened by another request
    // we therefore use a transaction to ensure consistency and integrity

    $post=$model->findByPk(10);

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

    $post->save();

    $transaction->commit();
}
```

```
}  
  
catch(Exception $e)  
{  
  
    $transaction->rollBack();  
  
}
```

命名空间

注意：对于命名空间的支持从版本 1.0.5 开始。思路来源于 Ruby on Rails.

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 CActiveRecord  
  
{  
  
    .....  
  
    public function scopes()  
    {  
  
        return array(  
  
            'published'=>array(  
  
                'condition'=>'status=1',  
  
            ),  
  
            'recently'=>array(  

```



```

        'order'=>'create_time DESC',

        'limit'=>5,

    ),

);

}

}

```

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`，翻译为一个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 `modifier` 来使用。几个命名空间可以连接在一起，则样可以得到一个更加有限制性的查询结果集。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();
```

通常命名空间必须出现在一个 `find` 方法的左边。Each of them provides a query criteria, which is combined with other criterias, including the one passed to the `find` method call. The net effect is like adding a list of filters to a query.

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();
```

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

参数化命名空间

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)
```

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

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：

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

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  
  
{  
  
    public function defaultScope()  
  
    {  
  
        return array(  

```

```

        'condition'=>"language='".Yii::app()->language.'"",

    );

}

}

```

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();
```

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

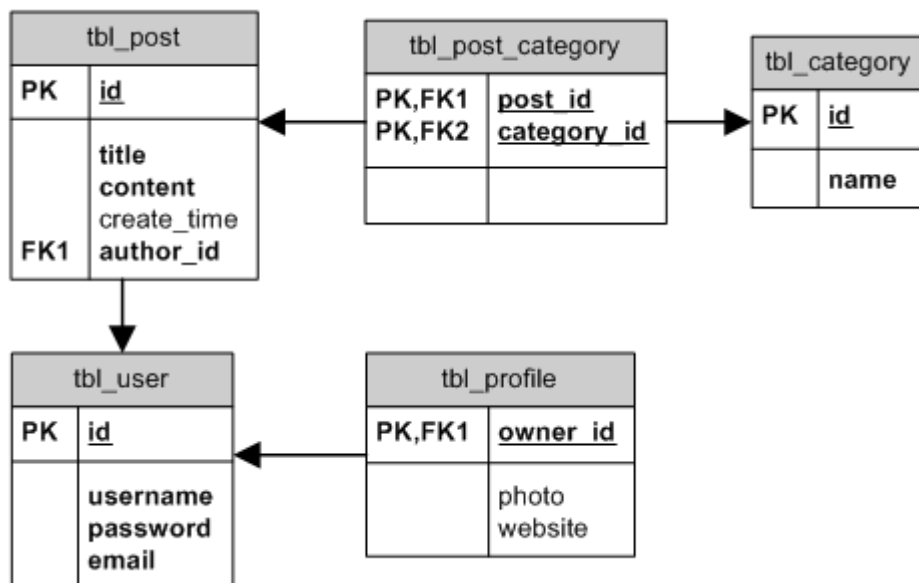
Relational Active Record

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。

要使用关联 AR，推荐在需要被关联的数据表中声明主键-外键约束。约束帮助保持相关数据的一致性和完整性。

在这一节中，我们将以下面这个简单的实体-关系(ER)图所描述的数据库为例，来介绍 如何使用包含关联的 ActiveRecord。

ER Diagram



说明：不同的关系数据库对外键约束的支持有所不同。

SQLite 是不支持外键约束的，但你在创建表时仍然可以声明约束。

声明关联

在使用 AR 进行关联查询之前，我们需要让 AR 知道它和其他 AR 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（例如 `tbl_user` 和 `tbl_post`），一对一（例如 `tbl_user` 和 `tbl_profile`），多对多（例如 `tbl_category` 和 `tbl_post`）。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，例如 `Post` 属于 `User`。
- **HAS_MANY**: 如果数据表 A 和 B 的关系是一对多，那我们就说 A 有多个 B，例如 `User` 有多个 `Post`。
- **HAS_ONE**: 这是‘HAS_MANY’关系中的一个特例，当 A 最多有一个 B，例如一个 `User` 最多只有一个 `Profile`
- **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。在我们的例子中，`tbl_post_category` 就是这个用作关联的表。用 AR 的术语，我们可以将 **MANY_MANY** 解释为 **BELONGS_TO** 和 **HAS_MANY** 的结合。例如 `Post` 属于多个 `Category` 并且 `Category` 有多个 `Post`。

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，可以是下面四个常量之一：

`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', 'author_id'),

    'categories'=>array(self::MANY_MANY, 'Category',

        'tbl_post_category(post_id, category_id)'),

    );

}

}

class User extends CActiveRecord

{

    .....

    public function relations()

    {

        return array(

            'posts'=>array(self::HAS_MANY, 'Post', 'author_id'),

            'profile'=>array(self::HAS_ONE, 'Profile', 'owner_id'),

        );

    }

}
```

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 `tbl_post_category(post_id, category_id)`。

在 `AR` 类中声明关联时，每个关联会作为一个属性被隐含地添加到 `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，若 `$author` 代表一个 `User` `AR` 实例，我们可以使用 `$author->posts` 来访问它的关联的 `Post` 实例。

执行关联查询

进行关联查询最简单的方式就是访问一个 `AR` 实例的关联属性。如果这个属性之前没有被访问过，一个关联查询被初始化，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”(lazy loading)，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()->findByPrimaryKey(10);

// retrieve the post's author: a relational query will be performed here

$author=$post->author;
```

说明: 若一个关联没有关联的实例，对应的属性将是 `null` 或一个空数组。对于 `BELONGS_TO` 和 `HAS_ONE` 关联，结果是 `null`；对于 `HAS_MANY` 和 `MANY_MANY`，结果是一个空数组。注意 `HAS_MANY` 和 `MANY_MANY` 关联返回的数组元素是对象，在尝试访问其属性之前需要遍历此数组。否则你将收到“Trying to get property of non-object”错误。

The lazy loading 方法非常方便使用，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 lazy 方法将执行 `N` 此连接查询。此时我们应当使用所谓的 *eager loading* 方法。

The eager loading 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 the lazy 方法，每个 `Post` 实例中的 `author` 属性已经被关联的 `User` 实例填充，在我们访问此属性之前。不是为每个 `post` 执行一个连接查询，the eager loading 方法在一个单独的连接查询中取出所有的 `post` 以及它们的作者！

我们可以在 [with\(\)](#) 方法中指定多个关联名字 ,the eager loading approach will bring them back all in one shot. 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 eager loading. 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档案和 `posts`.

从版本 1.1.0 开始, eager loading 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',

        )

    );

```

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。字段名字应当消除歧义。
- **condition**: **WHERE** 子语句。默认为空。此选项中的字段名应被消除歧义。
- **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。此选项自版本 1.0.3 可用。
- **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。此选项自版本 1.0.2 可用。
- **order**: **ORDER BY** 子语句。默认为空。此选项中的字段名应被消除歧义。
- **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**。
- **alias**: 关联的数据表的别名。此选项自版本 1.0.1 可用。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 **join together with the primary table and other tables**。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。此选项中的字段名应被消除歧义。注意：选项自版本 1.0.1 可用。
- **index**: the name of the column whose values should be used as keys of the array that stores related objects. 不设置此选项，一个关联对象数组将使用基于 0 的整数索引。此选项只为 **HAS_MANY** 和 **MANY_MANY** 设置。此选项自版本 1.0.7 可用。

此外，下面的选项在 **lazy loading** 中对特定关联是可用的：

- **limit**: limit of the rows to be selected. 此选项不应用于 **BELONGS_TO** 关联。
- **offset**: offset of the rows to be selected. 此选项不应用于 **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.

为字段名消除歧义

当一个字段的名称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'  
  
));
```

注意：自版本 1.1.0 开始，字段消除歧义的行为已经被改变。先前在版本 1.0.x 中，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。Also，在版本 1.0.x，主表的别名是表自身的名字。

动态关联查询选项

从版本 1.0.2 开始，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 `eager loading` 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();
```

从版本 1.0.5 开始，动态查询选项也可以当使用 `lazy loading` 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 `status` 为 1 的 `posts`：

```
$user=User::model()->findByPk(1);
```

```
$posts=$user->posts(array('condition'=>'status=1'));
```

关联查询的性能

如上所述，**eager loading** 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，因为它 **simplifies filtering based on a column in a related table**。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 **eager loading**。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 the eager loading 时，我们也可以动态地设置此选项：

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

注意: 在版本 1.0.x , 默认的行为是 Yii 将产生并执行 N+1 SQL 语句, 若有 N HAS_MANY 或 MANY_MANY 关联。每个 HAS_MANY 或 MANY_MANY 关联有它自己的 SQL 语句。通过在 with() 后调用 together()方法, 我们可以 强制只有一个单独的 SQL 语句被产生并执行。例如,

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

统计查询

注意: 统计查询已自版本 1.0.4 被支持。

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 `lazy loading` 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。

Relational Query with Named Scopes

注意：对命名空间的支持已从版本 `1.0.5` 开始。

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。

缓存

概述

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](#)。注意，该组件从 1.0.1 版本开始提供。
- [CEAcceleratorCache](#): 使用 PHP [EAccelerator 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。
- [CZendDataCache](#): 使用 [uses Zend Data Cache](#) 作为基础缓存媒介。注意，自版本 1.0.4 可用。
- [CFileCache](#): 使用文件来存储缓存数据。特别适用于大块数据(例如页面)。注意，自版本 1.0.6 可用。
- [CDummyCache](#): 代表不执行缓存的虚拟缓存。此组件的目的是简化需要检查缓存有效的代码。例如，在开发中或服务器实际不支持缓存，我们可以使用此缓存组件。当一个实际缓存支持被启用，我们可以转为使用对应的缓存组件。在两种情况下，我们可以使用相同的代码 `Yii::app()->cache->get($key)` 来尝试检索一段数据而无需担心 `Yii::app()->cache` 可能为 null。此组件自版本 1.0.5 可用。

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)。

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。

一些缓存存储, 例如 **MemCache**, **APC**, 支持批模式检索多个缓存值, 这样可以减少检索缓存数据的开销。自版本 **1.0.8** 开始, 一个新的方法 `mget()` 被提供用来利用此特征。万一底层缓存存储不支持此特征, `mget()` 将仍然模拟它。

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。
- [CExpressionDependency](#): 该依赖因指定 PHP 表达式的结果发生改变而改变。此类自版本 1.0.4 可用。

片段缓存

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。

缓存选项

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

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。

依赖

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...

<?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...
```

变化

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。

请求类型

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...

<?php if($this->beginCache($id, array('requestTypes'=>array('GET')))) { ?>

...被缓存的内容...

<?php $this->endCache(); } ?>

...其他 HTML 内容...
```

嵌套缓存

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分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，即使内部缓存中的内容已经过期。

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：

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

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](#)。

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，这意味着它是即是一个 widget 又是一个过滤器。事实上，`widget` 的工作方式和过滤器非常相似：一个 widget (filter) 是在 action 动作里的内容执行前执行，在执行后结束。

动态内容

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\(\)](#) 在你想要的地方插入动态内容。

...其他 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();

}

?>

...其他 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。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。

扩展 Yii

概述

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 widget 时,你正在继承 [CWidget](#) 或者一个已存在的 widget 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).

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

- [应用组件](#)
- [行为](#)
- [widget](#)
- [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](#) 类的组件.
- 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: 模块是一个自足的软件单元,由模型,视图,控制器和其他支持组件组成.在很多方面,模块类似于应用.主要的不同是模块位于应用内部.例如,我们可有一个模块用来提供用户管理功能.

扩展也可以是不属于上述分类中的任何一个的组件.事实上,Yii 是设计的很谨慎,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.

使用扩展

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.

每个扩展都有一个所有扩展中唯一的名称标识.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.

注意: 根路径别名 `ext` 已自版本 1.0.8 可用.之前,我们需要使用 `application.extensions` 指向包含所有扩展的目录. 在下面的描述中,我们假设 `ext` 被定义.若在使用版本 1.0.7 或之前的版本,你需要替换为 `application.extensions`.

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(

    'dataProvider'=>$dataProvider,

));
```

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(

    // 'preload'=>array('xyz',...),

    'components'=>array(

        'xyz'=>array(

            'class'=>'ext.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 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

$component->attachBehavior($name,$behavior);

// test() is a method of $behavior

$component->test();
```

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(

    'components'=>array(

        'db'=>array(

            'class'=>'CDbConnection',

            'behaviors'=>array(

                'xyz'=>array(

                    'class'=>'ext.xyz.XyzBehavior',

                    'property1'=>'value1',

                    'property2'=>'value2',

                ),

            ),

        ),

    ),
```

```
//....

),

);
```

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()

{

    return array(

        'xyz'=>array(

            'class'=>'ext.xyz.XyzBehavior',

            'property1'=>'value1',

            'property2'=>'value2',

        ),

    );

}
```

Widget

Widget 主要用在 [视图](#) 里.假设 **Widget** 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('ext.xyz.XyzClass', array(

    'property1'=>'value1',

    'property2'=>'value2')); ?>

// 组件可以包含主体内容

<?php $this->beginWidget('ext.xyz.XyzClass', array(

    'property1'=>'value1',

    'property2'=>'value2')); ?>

...组件的主体内容...

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

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'=>'ext.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.

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  
{  
    public function filters()  
    {  
        return array(  
            array(  
                'ext.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 `+` 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(  
    'controllerMap'=>array(  
        'xyz'=>array(  
            'class'=>'ext.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。

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

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(  
    'commandMap'=>array(  
        'xyz'=>array(  
            'class'=>'ext.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`.

模块

请参考章节 [模块](#), 了解如何使用一个模块。

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。

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组件,这些描述也适用。

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

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

{

    public function beforeSave($event)

    {

        if($this->owner->isNewRecord)

            $this->owner->create_time=time();

        else

            $this->owner->update_time=time();

    }

}
```

Widget

`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` 如下，让您在使用 `widget` 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，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
{
    protected function registerClientScript()
    {
        // ...publish CSS or JavaScript file here...

        $cs=Yii::app()->clientScript;

        $cs->registerCssFile($cssFile);

        $cs->registerScriptFile($jsFile);
    }
}
```

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。

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](#)。

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
{
    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`，它包含了当前被过滤的 `action` 的相关信息。

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。

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);
    }
}
```

```
}  
  
}
```

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';  
    }  
}
```

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 。

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。

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：

```
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` 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：

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

测试

概述

注意: 这一章描述的测试支持需要 Yii 版本 1.1 或更高的版本。这并不意味着你不能测试使用 Yii 1.0.x 开发的应用程序。有很多伟大的测试框架来帮助你完成这个任务，例如 PHPUnit, SimpleTest.

测试软件开发不可缺少的过程。无论我们是否意识到了这一点，当我们开发一个 **web** 应用时我们总是在测试。例如，当我们编写了一个类，我们可以使用一些 **echo** 或 **die** 语句来展示我们正确执行了一个方法；当我们执行一个包含复杂 **HTML** 表单的网页，我们可能尝试输入一些测试数据来确保页面按照预想的那样和我们互动。更高级的开发者可以编写一些代码来自动化测试过程，当需要测试时，只需要调用代码即可，让计算机为我们执行测试。这就是已知的 *automated testing*，它是这一章的主题。

Yii 提供的测试支持包含 *unit testing* 和 *functional testing*。

单元测试验证一个代码单元如预期的那样工作。在面向对象的编程中，最基本的代码单元是一个类。一个单元测试因此主要需要验证每个类接口方法工作正常。那就是，给出不同的输入参数，测试验证此方法是否返回预期的结果。单元测试主要由编写被测试类的人开发。

功能测试验证一个特征（例如：一个 **blog** 系统中的发布管理）如预期的那样工作。和单元测试相比，功能测试位于一个更高的级别因为一个被测试的特征经常涉及到多个类。功能测试经常由对系统需求非常了解的人开发(可以是开发者或质量工程师)。

Test-Driven Development

下面我们展示在所谓的 [test-driven development\(TDD\)](#) 中的开发周期:

- 创建一个新的测试，包含一个将被实现的特征。测试的预期结果是首次执行失败因为此特征还没有被执行。
- 运行所有测试并确保新的测试失败。
- 编写代码让新的测试通过。
- 运行所有测试并确保它们全部通过。
- Refactor 新编写的代码并确保测试仍然通过。

Repeat step 1 to 5 to push forward the functionality implementation.

建立测试环境

Yii 提供的测试支持需要 [PHPUnit](#) 3.3+ 和 [Selenium Remote Control](#) 1.0+。请参考它们的手册关于如何安装 PHPUnit 和 Selenium Remote Control。

当我们使用 `yiic webapp` 控制台命令来创建一个新的 Yii 应用，它为我们编写和执行新的测试将产生如下文件和目录:

testdrive/

protected/ containing protected application files

tests/ containing tests for the application

fixtures/ containing database fixtures

functional/ containing functional tests

unit/ containing unit tests

report/ containing coverage reports

bootstrap.php the script executed at the very beginning

phpunit.xml the PHPUnit configuration file

WebTestCase.php the base class for Web-based functional tests

如上所示，我们的测试代码将主要位于三个目录中：**fixtures**、**functional** 和 **unit**，目录 **report** 将被用来存储产生的 **code coverage** 报告。

要执行测试（无论是单元测试还是功能测试），我们可以在一个控制台窗口中执行如下命令：

```
% cd testdrive/protected/tests

% phpunit functional/PostTest.php    // executes an individual test

% phpunit --verbose functional        // executes all tests under 'functional'

% phpunit --coverage-html ./report unit
```

在上面，最后的命令将执行 **unit** 目录下所有的测试并产生一个 **code-coverage** 报告到 **report** 目录。注意 [xdebug extension](#) 必须被安装并启用，为了产生 **code-coverage** 报告。

测试引导脚本

让我们看一下 **bootstrap.php** 文件里有哪些东西。这个文件很特殊，因为它类似于 [入口脚本](#)，当我们执行测试集合时它是开始点。

```
$yiit='path/to/yii/framework/yiit.php';
```

```
$config=dirname(__FILE__).'../config/test.php';

require_once($yiit);

require_once(dirname(__FILE__).'../WebTestCase.php');


Yii::createWebApplication($config);
```

在上面，我们首先包含了 Yii 框架中的 `yiit.php`，它初始化一些全局常量并载入必要的测试基础类。然后我们使用 `test.php` 配置文件创建一个 Web 应用实例。若我们检查 `test.php`，我们会发现它继承自 `main.php` 配置文件并增加一个 `fixture` 应用组件，这个组件的类是 [CDbFixtureManager](#)。我们将在下面章节详细描述 fixtures。

```
return CMap::mergeArray(

    require(dirname(__FILE__).'../main.php'),

    array(

        'components'=>array(

            'fixture'=>array(

                'class'=>'system.test.CDbFixtureManager',

            ),

            /* uncomment the following to provide test database connection */

            'db'=>array(

                'connectionString'=>'DSN for test database',

            ),

            */

        ),

    )
```



```
);
```

当我们运行涉及数据库的测试，我们应当提供一个测试数据库以便测试的执行不会干扰正常的开发或应用的实际应用。要这样做，我们只需取消上面 `db` 配置部分的注释，并且以测试数据库的 `DSN` 来填充 `connectionString` 属性。

使用一个引导脚本，当我们运行单元测试时，我们将拥有一个应用实例，它和相应网络请求的那个应用实例几乎完全一样。主要的不同点是它有 `fixture manager` 并且使用的是测试数据库。

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.

为了使用 [CDbFixtureManager](#)，我们在应用配置中如下设置它，

```
return array(

    'components'=>array(

        'fixture'=>array(

            'class'=>'system.test.CDbFixtureManager',

        ),

    ),

);
```

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字相同。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

return array(

    'sample1'=>array(

        'title'=>'test post 1',

        'content'=>'test post content 1',

        'createTime'=>1230952187,

        'authorId'=>1,

    ),

    'sample2'=>array(

        'title'=>'test post 2',

        'content'=>'test post content 2',

        'createTime'=>1230952287,

        'authorId'=>1,

    ),

);
```

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。

提示：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`。

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`：

```
class CommentTest extends CDbTestCase
{
    public $fixtures=array(
```

```
        'posts'=>'Post',  
        'comments'=>'Comment',  
    );  
  
    .....  
}
```

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：

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

注意：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。

```
public function testApprove()
{
    // insert a comment in pending status

    $comment=new Comment;

    $comment->setAttributes(array(

        'content'=>'comment 1',

        'status'=>Comment::STATUS_PENDING,

        'createTime'=>time(),

        'author'=>'me',

        'email'=>'me@example.com',

        'postId'=>$this->posts['sample1']['id'],

    ),false);

    $this->assertTrue($comment->save(false));

    // verify the comment is in pending status

    $comment=Comment::model()->findByPk($comment->id);

    $this->assertTrue($comment instanceof Comment);

    $this->assertEquals(Comment::STATUS_PENDING,$comment->status);

    // call approve() and verify the comment is in approved status

    $comment->approve();

    $this->assertEquals(Comment::STATUS_APPROVED,$comment->status);
}
```

```
$comment=Comment::model()->findByPk($comment->id);

$this->assertEquals(Comment::STATUS_APPROVED,$comment->status);

}
```

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.

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/');

class WebTestCase extends CWebTestCase

{

    /**

     * Sets up before each test method runs.

     * This mainly sets the base URL for the test application.

     */
```

```
protected function setUp()

{

    parent::setUp();

    $this->setBrowserUrl(TEST_BASE_URL);

}

.....

}
```

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` 唯一的区别是前者使用 `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

{

    public $fixtures=array(

        'posts'=>'Post',

    );

    public function testShow()
```

```
{

    $this->open('post/1');

    // verify the sample post title exists

    $this->assertTextPresent($this->posts['sample1']['title']);

    // verify comment form exists

    $this->assertTextPresent('Leave a Comment');

}

.....

}
```

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**。

提示：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。

专题

自动化代码生成

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(  
    .....  
    'modules'=>array(  
        'gii'=>array(  
            'class'=>'system.gii.GiiModule',  
            'password'=>'pick up a password here',  
            // 'ipFilters'=>array(...a list of IPs...),  
  
            // 'newFileMode'=>0666,  
            // 'newDirMode'=>0777,  
        ),  
    ),  
);
```

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生成。

注意：Gii 主要是一个开发工具。因此，它应只被安装于一个开发机器上。因为它可以产生新 PHP 脚本文件到应用中，我们应当注意采取安全措施（例如 password, IP filters）。

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://hostnamepath/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(
    .....
    'urlManager'=>array(
        'urlFormat'=>'path',
        'rules'=>array(

            'gii'=>'gii',

            'gii/<controller:\w+>'=>'gii/<controller>',

            'gii/<controller:\w+>/<action:\w+>'=>'gii/<controller>/<action>',

            ...existing rules...

        ),
    ),
)
```

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

虽然 **Gii** 默认的代码生成器可以生成强大的代码，我们经常想定制它们或创建一个新的来适应我们的口味和需求。例如，我们想要生成的代码是我们喜欢的代码样式，或者我们想让代码支持多语言。适应 **Gii**，所有这些都可以轻松完成。

Gii 可以以两种方式被扩展：定制已存在的代码生成器的代码模板，以及编写新的代码生成器。

代码生成器的结构

一个代码生成器被存储到一个目录中，目录的名字就是生成器的名字。此目录通常由如下内容组成：

<code>model/</code>	the <code>model generator</code> 根目录
<code>ModelCode.php</code>	生成代码的代码模型
<code>ModelGenerator.php</code>	代码生成器的控制器
<code>views/</code>	包含生成器的视图脚本
<code>index.php</code>	默认的视图脚本
<code>templates/</code>	包含代码模板集合的目录
<code>default/</code>	'default' 代码模板集合
<code>model.php</code>	生成 <code>model</code> 类代码的模板

生成器搜索路径

Gii 在 [`GiiModule::generatorPaths`](#) 属性指定的目录中搜索可用的生成器。当需要定制时，我们可以在应用配置中如下设置此属性，

```
return array(  
    'modules'=>array(  
        'gii'=>array(  
            'class'=>'system.gii.GiiModule',  
            'generatorPaths'=>array(  
                'application.gii', // 一个路径别名  
            ),  
        ),  
    ),  
);
```

上面的配置告诉 Gii 在别名是 `application.gii` 的目录中寻找生成器，而不仅仅是在默认的位置 `system.gii.generators`。

两个生成器同名但在不同的搜索路径中，这样也是可以的。这时，在 [GiiModule::generatorPaths](#) 中首先出现的将有优先权。

定制代码模板

这是扩展 Gii 最简单和最常见的方式。我们使用一个例子来解释如何定制代码模板。假设我们想要定制 `the code generated by the model generator`。

我们首先创建一个名为 `protected/gii/model/templates/compact` 的目录。这里的 `model` 意味着我们将要覆盖默认的 `model` 生成器。`templates/compact` 意味着我们将要增加一个新的模板集合 `compact`。

然后我们更改应用配置，增加 `application.gii` 到 [GiiModule::generatorPaths](#)，如上一小节所述。

现在打开 `the model code generator` 页面。点击 `Code Template` 输入框。我们可以看到一个下拉列表，它包含了我们新创建的模板目录 `compact`。然而，若我们选择此模板来生成代码，我们将看到一个错误。这是因为我们还没有把实际的代码模板文件放置到新建的 `compact` 模板集中。

复制文件 `framework/gii/generators/model/templates/default/model.php` 到 `protected/gii/model/templates/compact`。若我们再次尝试使用 `compact` 模板生成，我们应该成功。然而，生成的代码和使用 `default` 模板集生成的代码并没有什么不同。

现在是时候做一些实际的工作了。打开文件 `protected/gii/model/templates/compact/model.php` 来编辑它。记得它将以类似一个视图脚本的方式被使用，意味着它可以含有 `PHP` 表达式和语句。让我们改变此模板以便生成的代码中的 `attributeLabels()` 方法使用 `Yii::t()` 来翻译属性标签：

```
public function attributeLabels()
{
    return array(

<?php foreach($labels as $name=>$label): ?>

        <?php echo "'$name' => Yii::t('application', '$label'),\n"; ?>

<?php endforeach; ?>

    );
}
```

在每个代码模板中，我们可以访问一些预定义的变量，例如上面例子中的 `$labels`。这些变量由相应代码生成器提供。不同的代码生成器在它们的代码模板中提供不同的变量集合。请仔细阅读默认代码模板中的描述。

创建新的生成器

在这个小节中，我们展示如何创建一个新的可以生成 `widget` 类的生成器。

首先我们创建一个名为 `protected/gii/widget` 的目录。在此目录中，我们将创建如下文件：

- `WidgetGenerator.php`: 包含 `WidgetGenerator` 控制器类。这是 `widget` 生成器的入口。
- `WidgetCode.php`: 包含 `WidgetCode` `model` 类。这个类含有代码生成的主要逻辑。
- `views/index.php`: 显示代码生成输入表单的视图脚本。
- `templates/default/widget.php`: 生成 `widget` 类文件的默认代码模板。

创建 `WidgetGenerator.php`

文件 `WidgetGenerator.php` 非常简单。它只包含如下代码：

```
class WidgetGenerator extends CCodeGenerator
{

    public $codeModel='application.gii.widget.WidgetCode';

}
```

在上面的代码中，我们指定生成器将使用路径别名是 `application.gii.widget.WidgetCode` 的模型类。类 `WidgetGenerator` 扩展自 [CCodeGenerator](#) which implements a lot of functionalities, including the controller actions needed to coordinate the code generation process.

创建 WidgetCode.php

文件 `WidgetCode.php` 包含 `WidgetCode` 模型类，这个类有基于用户输入生成一个 `widget` 类的主要逻辑。在这个例子中，我们假设唯一想从用户那里得到的输入是 `widget` 的名字。我们的 `WidgetCode` 看起来如下：

```
class WidgetCode extends CCodeModel
{

    public $className;

    public function rules()
    {

        return array_merge(parent::rules(), array(

            array('className', 'required'),

            array('className', 'match', 'pattern'=> '/^\w+$/'),

        ));

    }

}
```

```
public function attributeLabels()
{
    return array_merge(parent::attributeLabels(), array(

        'className'=>'Widget Class Name',

    ));
}

public function prepare()
{
    $path=Yii::getPathOfAlias('application.components.' . $this->className) . '.php';

    $code=$this->render($this->templatepath.'/widget.php');

    $this->files[]=new CCodeFile($path, $code);
}
}
```

`WidgetCode` 类扩展自 [CCodeModel](#)。类似于一个普通的模型类，在这个类中我们可以声明 `rules()` 和 `attributeLabels()` 来分别验证用户输入和提供属性标签。注意由于基类 [CCodeModel](#) 已经定义了一些规则和属性标签，在这里我们应当将它们和我们新加的规则及标签组合起来。

`prepare()` 方法准备将被生成的代码。它的主要任务是准备一个 [CCodeFile](#) 对象列表，每个对象代表了一个将被生成的代码文件。在我们的例子中，我们只需要创建 [CCodeFile](#) 对象，它代表了被生成的 `widget` 类文件。新的 `widget` 类文件将生成到 `protected/components` 目录中。我们调用 [CCodeFile::render](#) 方法来生成实际的代码。此方法载入代码模板作为一个 PHP 脚本并返回输出的内容作为生成后的代码。

创建 views/index.php

有了控制器 (WidgetGenerator) 和模型 (WidgetCode), 现在我们创建视图 views/index.php。

```
<h1>Widget Generator</h1>

<?php $form=$this->beginWidget('CCodeForm', array('model'=>$model)); ?>

    <div class="row">

        <?php echo $form->labelEx($model, 'className'); ?>

        <?php echo $form->textField($model, 'className', array('size'=>65)); ?>

        <div class="tooltip">

            Widget class name must only contain word characters.

        </div>

        <?php echo $form->error($model, 'className'); ?>

    </div>

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

在上面, 我们主要使用 [CCodeForm](#) widget 显示一个表单。在这个表单中, 我们为WidgetCode 的 className 属性显示一个输入框以收集用户的输入。

当创建此表单时，我们可以利用由 [CCodeForm](#) widget 提供的两个特征。一个是 `input tooltips`。另一个是 `sticky inputs`。

若你尝试过任何默认的代码生成器，你将注意到当设置焦点到一个输入框时，一个提示将显示在这个输入框的后面。这非常容易实现，只需在这个输入框后面编写一个 CSS 类是 `tooltip` 的 `div` 即可。

对于一些输入框，我们想要记得它们最后的有效值，以免去再次输入它们的麻烦。一个例子是收集默认控制器生成器的控制器基类名的输入框。这些 `sticky` 输入框被初始显示为高亮的静态文本。若我们点击它们，它们将转化为一个输入框来收集我们的输入。

为了声明一个输入框是 `sticky`，我们需要做两件事情。

首先，我们需要为对应的模型属性声明一条 `sticky` 验证规则。例如，默认的控制器的生成器有如下规则来声明 `baseClass` 和 `actions` 属性是 `sticky`:

```
public function rules()
{
    return array_merge(parent::rules(), array(

        .....

        array('baseClass', 'actions', 'sticky'),

    ));
}
```

其次，我们需要增加一个 CSS 类 `sticky` 到视图中的输入框的 `div` 中，如下：

```
<div class="row sticky">

    ...input field here...

</div>
```

创建 `templates/default/widget.php`

最后，我们创建代码模板 `templates/default/widget.php`。如我们之前描述的，它被用作一个视图脚本，可以包含 PHP 表达式和语句。在一个代码模板中，我们总是可以使用 `$this` 变量，它指的是代码模型对象。在我们的例子中，`$this` 指的是 `WidgetModel` 对象。这样我们可以通过 `$this->className` 得到用户输入的 `widget` 类名字。

```
<?php echo '<?php'; ?>

class <?php echo $this->className; ?> extends CWidget

{

    public function run()

    {

    }

}
```

这里包含了一个新的代码生成器的创建。我们可以通过 如下 URL 访问这个代码生成器 `http://hostname/path/to/index.php?r=gii/widget`。

网址管理

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，应用程序需求提供一种创造 URL 的方法，以便创建的 URL 可以被应用理解。对于 Yii 应用程序，这些通过 `CUrlManager` 辅助完成。

创建网址

虽然 URL 可被硬编码在控制器的视图文件，但往往可以更灵活地动态创建它们：

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

`$this` 指的是控制器实例；`$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，使用 `createUrl` 创建的 URL 的格式被称为 `get`。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
```

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

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，省去查询字符串和把 `GET` 参数加到路径信息，作为网址的一部分：

```
/index.php/post/read/id/100
```

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以及应用程序可以正确理解新的网址：

```
array(  
    .....  
    'components'=>array(  
        .....  
        'urlManager'=>array(  
            'urlFormat'=>'path',  
        ),  
    ),  
);
```

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

提示：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`。

用户友好的 URL

当用 `path` 作为 URL 格式，我们可以指定某些 URL 规则使我们的网址更用户友好性。例如，我们可以产生一个简短的 `URL/post/100`，而不是冗长的 `/index.php/post/read/id/100`。URL 规则被 `CUrlManager` 用于 URL 创建和解析目的。

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

```
array(  
    .....  
    'components'=>array(  
        .....  
        'urlManager'=>array(  
            'urlFormat'=>'path',  
            'rules'=>array(  
                'pattern1'=>'route1',  
                'pattern2'=>'route2',  
                'pattern3'=>'route3',  
            ),  
        ),  
    ),  
);
```

规则被指定为一个由 模式-路由对(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)。一个规则的路由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)
```

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)。当此条规则被用来解析进来的请求, 这个属性中声明的值被添加到 \$_GET。
- **matchValue**: 当创建一个 URL 时, 是否 GET 参数值应当匹配此规则中的对应的子模式。默认是 null, 意味着使用 [CUrlManager::matchValue](#) 的值。若此属性是 false, 它意味着一条规则将被用来创建一个 URL, 若它的路由和参数名字匹配给定的那个。若此属性被设置为 true, 给定的参数值也必须匹配对应的参数子模式。注意设置此属性为 true 将降低性能。

使用命名参数

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, 以一种特殊令牌格式表现如下:

```
<ParamName:ParamPattern>
```

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。若 ParamPattern 被忽略, 意味着参数应当匹配任意字符除了斜线 /。当生成一个网址 (URL) 时, 这些参数令牌将被相应的参数值替换; 当解析一个网址时, 相应的 GET 参数将被解析后的结果填充。

我们使用一些例子来解释网址工作规则。我们假设我们的规则包括如下三个:

```
array(
    'posts'=>'post/list',
    'post/<id:\d+>'=>'post/read',
    'post/<year:\d{4}>/<title>'=>'post/read',
)
```

- 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。

总之, 当使用 [createUrl](#) 生成 URL 时, 传递给此方法的路由和 GET 参数被用来决定哪条 URL 规则适用。如果关联规则中的每个参数可以在 GET 参数找到的, 将被传递给 [createUrl](#), 如果规则的路由也匹配路由参数, 此规则将用来生成 URL。

如果传递到 [createUrl](#) 的 GET 参数是多于一条规则所需要的, 多出的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，该规则 `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 规则的其他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）。

注：使用的 URL 规则将降低应用的性能。这是因为当解析请求的 URL，`CUrlManager` 尝试使用每个规则来匹配它，直到某个规则可以适用。因此，高流量网站应用应尽量减少其使用的 URL 规则。

参数化路由

自版本 1.0.5 起，我们可以在一条规则的路由部分引用 named parameters。This allows a rule to be applied to multiple routes based on matching criteria. It may also help reduce the number of rules needed for an application, and thus improve the overall performance.

We use the following example rules to illustrate how to parameterize routes with named parameters:

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

In the above, we use two named parameters in the route part of the rules: `_c` and `_a`. The former matches a controller ID to be either `post` or `comment`, while the latter matches an action ID to be `create`, `update` or `delete`. You may name the parameters differently as long as they do not conflict with GET parameters that may appear in URLs.

Using the aboving rules, the URL `/index.php/post/123/create` would be parsed as the route `post/create` with GET parameter `id=123`. And given the route `comment/list` and GET parameter `page=2`, we can create a URL `/index.php/comments?page=2`.

参数化主机名

Starting from version 1.0.11, it is also possible to include hostname into the rules for parsing and creating URLs. One may extract part of the hostname to be a GET parameter. For example, the URL `http://admin.example.com/en/profile` may be parsed into GET parameters `user=admin` and `lang=en`. On the other hand, rules with hostname may also be used to create URLs with paratermized hostnames.

In order to use parameterized hostnames, simply declare URL rules with host info, e.g.:

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

The above example says that the first segment in the hostname should be treated as `user` parameter while the first segment in the path info should be `lang` parameter. The rule corresponds to the `user/profile` route.

Note that `CUrlManager::showScriptName` will not take effect when a URL is being created using a rule with parameterized hostname.

Also note that the rule with parameterized hostname should NOT contain the sub-folder if the application is under a sub-folder of the Web root. For example, if the application is under `http://www.example.com/sandbox/blog`, then we should still use the same URL rule as described above without the sub-folder `sandbox/blog`.

隐藏 index.php

还有一点，我们可以做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` 应用组件。

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  
  
IndexIgnore */*  
  
RewriteEngine on  
  
# if a directory or a file exists, use it directly  
  
RewriteCond %{REQUEST_FILENAME} !-f  
  
RewriteCond %{REQUEST_FILENAME} !-d  
  
# otherwise forward it to index.php
```

```
RewriteRule . index.php
```

然后，我们设定 [urlManager](#) 组件的 [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。

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。

验证和授权

如果网页的访问需要用户权限限制，那么我们需要使用验证（**Authentication**）和授权（**Authorization**）。验证是指核查某人表明的身份信息是否与系统相合。一般来说使用用户名和密码，当然也可能使用别的表明身份方式，录入智能卡，指纹等等。授权是找出已通过验证的用户是否允许操作特定的资源。一般的做法是找出此用户是否属于某个允许操作此资源的角色。

利用 Yii 内置的验证和授权（**auth**）框架，我们可以轻松实现上述功能。

Yii auth framework 的核心一块是一个事先声明的 *user application component*（用户应用部件），实现 [IWebUser](#) 接口的对象。此用户部件代表当前用户存储的身份信息。我们能够通过 `Yii::app()->user` 在任何地方来获取它。

使用此用户部件，可以通过 [CWebUser::isGuest](#) 检查一个用户是否登陆。可以 [login](#)（登陆）或者 [logout](#)（注销）一个用户；可以通过 [CWebUser::checkAccess](#) 检查此用户是否能够执行特定的操作；还可以获得此用户的 [unique identifier](#)（唯一标识）和别的身份信息。

定义身份类

为了验证一个用户，我们定义一个有验证逻辑的身份类。这个身份类实现 [IUserIdentity](#) 接口。不同的类可能实现不同的验证方式（例如：OpenID, LDAP）。最好是继承 [CUserIdentity](#)，此类是居于用户名和密码的验证方式。

定义身份类的主要工作是实现 [IUserIdentity::authenticate](#) 方法。在用户会话中根据需要，身份类可能需要定义别的身份信息

下面的例子，我们使用 [Active Record](#) 来验证提供的用户名、密码和数据库的用户表是否吻合。我们通过重写 `getId` 函数来返回验证过程中获得的 `_id` 变量（缺省的实现则是返回用户名）。在验证过程中，我们还借助 [CBaseUserIdentity::setState](#) 函数把获得的 `title` 信息存成一个状态。


```
class UserIdentity extends CUserIdentity
{
    private $_id;

    public function authenticate()
    {
        $record=User::model()->findByAttributes(array('username'=>$this->username));

        if($record===null)

            $this->errorCode=self::ERROR_USERNAME_INVALID;

        else if($record->password!==md5($this->password))

            $this->errorCode=self::ERROR_PASSWORD_INVALID;

        else
        {
            $this->_id=$record->id;

            $this->setState('title', $record->title);

            $this->errorCode=self::ERROR_NONE;
        }

        return !$this->errorCode;
    }

    public function getId()
    {
        return $this->_id;
    }
}
```

作为状态存储的信息（通过调用 [CBaseUserIdentity::setState](#)）将被传递给 [CWebUser](#)。而后者则把这些信息存放在一个永久存储媒介上（如 `session`）。我们可以把这些信息当作 [CWebUser](#) 的属性来使用。例如，为了获得当前用户的 `title` 信息，我们可以使用 `Yii::app()->user->title`（这项功能是在 1.0.3 版本引入的。在之前的版本里，我们需要使用 `Yii::app()->user->getState('title')`）。

提示：缺省情况下，[CWebUser](#) 用 `session` 来存储用户身份信息。如果允许基于 `cookie` 方式登录(通过设置 [CWebUser::allowAutoLogin](#) 为 `true`)，用户身份信息也可以被存放在 `cookie` 中。确记敏感信息不要存放(例如 `password`)。

登录和注销

使用身份类和用户组件，我们方便的实现登录和注销。

```
// 使用提供的用户名和密码登录用户

$identity=new UserIdentity($username,$password);

if($identity->authenticate())

    Yii::app()->user->login($identity);

else

    echo $identity->errorMessage;

.....

// 注销当前用户

Yii::app()->user->logout();
```

缺省情况下，用户在一段时间不活跃之后将被注销，根据 [session 配置](#)。设置用户组件的 [allowAutoLogin](#) 属性为 `true` 和在 [CWebUser::login](#) 方法中设置一个持续时间参数来改变这个行为。即使用户关闭浏览器，此用户将在设置的时间保持登陆状态。需要注意的是此特征需要用户的浏览器接受 `cookies`。

```
// 保留用户登陆状态时间 7 天

// 确保用户部件的 allowAutoLogin 被设置为 true。

Yii::app()->user->login($identity,3600*24*7);
```

访问控制过滤器

访问控制过滤器是检查当前用户是否能执行访问的 `controller action` 的初步授权模式。这种授权模式基于用户名，客户 IP 地址和访问类型。 It is provided as a filter named as ["accessControl"](#).

小贴士: 访问控制过滤器适用于简单的验证。需要复杂的访问控制，需要使用将要讲解到的基于角色访问控制 (role-based access (RBAC)) 。

在控制器(controller)里重载 [CController::filters](#) 方法设置访问过滤器来控制访问动作(看 [Filter](#) 了解更多过滤器设置信息)。

```
class PostController extends CController
{
    .....

    public function filters()
    {
        return array(
            'accessControl',
        );
    }
}
```

在上面，设置的 [access control](#) 过滤器将应用于 `PostController` 里每个动作。过滤器具体的授权规则通过重载控制器的 [CController::accessRules](#) 方法来指定。

```
class PostController extends CController
{
    .....

    public function accessRules()
    {
        return array(
            array('deny',
                'actions'=>array('create', 'edit'),
```

```

        'users'=>array('?'),
    ),
    array('allow',
        'actions'=>array('delete'),
        'roles'=>array('admin'),
    ),
    array('deny',
        'actions'=>array('delete'),
        'users'=>array('*'),
    ),
);
}
}

```

上面设定了三个规则，每个用个数组表示。数组的第一个元素不是 **allow** 就是 **deny**，其他的是名-值成对形式设置规则参数的。上面的规则这样理解：**create** 和 **edit** 动作不能被匿名执行；**delete** 动作可以被 **admin** 角色的用户执行；**delete** 动作不能被任何人执行。

访问规则是一个一个按照设定的顺序一个一个来执行判断的。和当前判断模式（例如：用户名、角色、客户端 IP、地址）相匹配的第一条规则决定授权的结果。如果这个规则是 **allow**，则动作可执行；如果是 **deny**，不能执行；如果没有规则匹配，动作可以执行。

info|提示：为了确保某类动作在没允许情况下不被执行，设置一个匹配所有人的 **deny** 规则在最后，类似如下：

```

return array(

    // ... 别的规则...

    // 以下匹配所有人规则拒绝'delete'动作

    array('deny',

        'action'=>array('delete'),
    ),
);

```

```
),  
  
);
```

因为如果没有设置规则匹配动作，动作缺省会被执行。

访问规则通过如下的上下文参数设置：

- **actions**: 设置哪些动作匹配此规则。这应该是一个 **action ID** 组成的数组。对比不区分大小写。
- **controllers**: 指定哪些 **controllers** 匹配此规则。这应该是控制器 **ID** 构成的一个数组。对比不区分大小写。此选项自版本 **1.0.4** 可用。
- **users**: 设置哪些用户匹配此规则。当前用户的 **name** 被用来匹配。这里可以使用三种特殊字符：
 - *****: 任何用户，包括匿名和验证通过的用户。
 - **?**: 匿名用户。
 - **@**: 验证通过的用户。
- **roles**: 设定哪些角色匹配此规则。这里用到了将在下一节描述的特征。特别的，若 **CWebUser::checkAccess** 为其中一个角色返回 **true**，此规则被应用。提示，用户角色应该主要被设置成 **allow** 规则，因为角色代表能做某些事情的权限。也要注意，虽然在这里使用了术语 角色，它的值实际上可以是任何认证项目，包括角色，任务和操作。
- **ips**: 设定哪些客户端 **IP** 匹配此规则。
- **verbs**: 设定哪些请求类型(例如: **GET**, **POST**)匹配此规则。对比不区分大小写。
- **expression**: 设定一个 **PHP** 表达式。它的值用来表明这条规则是否适用。在表达式，你可以使用一个叫 **\$user** 的变量，它代表的是 **Yii::app()->user**。这个选项是在 **1.0.3** 版本里引入的。

授权处理结果

当授权失败，即，用户不允许执行此动作，以下的两种可能将会产生：

- 如果用户没有登录，在用户组件中配置了 **loginUrl**，浏览器将重定位网页到此配置 **URL**。默认情况下，**loginUrl** 指向 **site/login** 页面。
- 否则一个错误代码 **403** 的 **HTTP** 异常将显示。

当配置 **loginUrl** 属性，可以用相对和绝对 **URL**。还可以使用数组通过 **CWebApplication::createUrl** 来生成 **URL**。第一个元素将设置 **route** 为登录控制器动作，其他为名-值成对形式的 **GET** 参数。如下，

```
array(  
  
    .....  
  
    'components'=>array(  
  
        'user'=>array(  

```

```
// 这实际上是默认值

    'loginUrl'=>array('site/login'),

),

),

)
```

如果浏览器重定位到登录页面，而且登录成功，我们将重定位浏览器到引起验证失败的页面。我们怎么知道这个值呢？我们可以通过用户部件的 [returnUrl](#) 属性获得。我们因此可以用如下执行重定向：

```
Yii::app()->request->redirect(Yii::app()->user->returnUrl);
```

基于角色的访问控制

基于角色的访问控制（RBAC）提供了一种简单而强大的集中访问控制。请参阅[维基文章](#) 了解更多详细的 RBAC 与其他较传统的访问控制模式的比较。

Yii 通过其 [authManager](#) 应用组件实现一个分级 RBAC 方案。在下面，我们首先介绍用于这模式的主要概念;我们然后描述了如何设定授权数据;最后，我们看看如何利用授权数据，以进行访问检查。

概览

在 Yii 的 RBAC 的一个基本概念是 *authorization item*（授权项目）。一个授权项目是一个做某事的权限（如创建新的博客文章，管理用户）。根据其粒度和目标用户，授权项目可分为 *operations*（操作），*tasks*（任务）和 *roles*（角色）。一个角色由任务组成，一个任务由操作组成，一个操作是权限的最基本单元。例如，我们就可以有一个 **administrator** 角色，包括 **post management** 和 **user management** 任务。**user management** 任务可能包括 **create user**, **update user** 和 **delete user** 操作。为了更灵活，Yii 也可以允许一个角色由其他角色或操作组成，任务包括其他任务，操作包括其他操作。

授权项目通过名称唯一确定。

授权项目可与 *business rule*（业务规则）关联。业务规则是一块 PHP 代码，将在执行关于此项的访问检查时被执行。只有当执行返回 **true**，用户将被视为有此项的权限。举例来说，当定义一个操作 **updatePost**，我们想添加业务规则来检查，用户 ID 是否和帖子作者 ID 一样，以便只有作者自己能够有权限更新发布。

使用认证项，我们可以建立一个 *authorization hierarchy*（授权等级）。在授权等级中如果项目 A 包括项目 B，A 是 B 的父亲(或说 A 继承 B 所代表的权限)。一个项目可以有多个子项目，也可以有多个父项目。因此，授权等级是一个 *partial-order* 图，而不是树型。在此等级中，角色项在最高，操作项在最低，任务项在中间。

一旦有了授权等级，我们可以在此等级中分配角色给应用用户。一个用户，一旦被分配了角色，将有角色所代表的权限。例如，如果我们指定 **administrator** 角色给用户，他将拥有管理员权限 其中包括 **post management** 和 **user management**（和 相应的操作，如 **create user**）。

现在精彩的部分开始。在控制器的行动，我们要检查，当前用户是否可以删除指定的发布。利用 **RBAC** 等级和分配，可以很容易做到这一点。如下：

```
if(Yii::app()->user->checkAccess('deletePost'))  
{  
  
    // 删除此发布  
  
}
```

配置授权管理器

在我们准备定义授权等级和执行访问检查前，我们需要配置 [authManager](#) 应用程序组件。Yii 提供两种类型的授权管理器：[CPhpAuthManager](#) 和 [CDbAuthManager](#)。前者使用的 PHP 脚本文件来存储授权 数据，而后者的数据存储在数据库中。当我们配置 [authManager](#) 应用组件，我们需要指定哪个组件类被使用，此组件的初始值是什么。例如，

```
return array(  
  
    'components'=>array(  
  
        'db'=>array(  
  
            'class'=>'CdbConnection',  
  
            'connectionString'=>'sqlite:path/to/file.db',  
  
        ),  
  
        'authManager'=>array(  
  
            'class'=>'CdbAuthManager',  
  
            'connectionID'=>'db',  
  
        ),  
  
    ),  
  
);
```

然后，我们便可使用 `Yii::app()->authManager` 访问 [authManager](#) 应用组件。

定义授权等级

定义授权等级涉及三个步骤：定义授权项目，建立授权项目间的关系，并分配角色给应用用户。[authManager](#) 应用组件提供了一整套的 API 来完成这些任务。

根据不同项目的类型，调用下列方法之一定义授权项目：

- [CAuthManager::createRole](#)
- [CAuthManager::createTask](#)
- [CAuthManager::createOperation](#)

一旦我们拥有一套授权项目，我们可以调用以下方法建立授权项目关系：

- [CAuthManager::addItemChild](#)
- [CAuthManager::removeItemChild](#)
- [CAuthItem::addChild](#)
- [CAuthItem::removeChild](#)

最后，我们调用下列方法来分配角色项目给各个用户：

- [CAuthManager::assign](#)
- [CAuthManager::revoke](#)

下面我们将展示一个例子是关于用所提供的 API 建立一个授权等级：

```
$auth=Yii::app()->authManager;

$auth->createOperation('createPost','create a post');

$auth->createOperation('readPost','read a post');

$auth->createOperation('updatePost','update a post');

$auth->createOperation('deletePost','delete a post');

$bizRule='return Yii::app()->user->id==$params["post"]->authID;';

$task=$auth->createTask('updateOwnPost','update a post by author himself',$bizRule);

$task->addChild('updatePost');
```



```
$role=$auth->createRole('reader');  
  
$role->addChild('readPost');  
  
$role=$auth->createRole('author');  
  
$role->addChild('reader');  
  
$role->addChild('createPost');  
  
$role->addChild('updateOwnPost');  
  
$role=$auth->createRole('editor');  
  
$role->addChild('reader');  
  
$role->addChild('updatePost');  
  
$role=$auth->createRole('admin');  
  
$role->addChild('editor');  
  
$role->addChild('author');  
  
$role->addChild('deletePost');  
  
$auth->assign('reader','readerA');  
  
$auth->assign('author','authorB');  
  
$auth->assign('editor','editorC');  
  
$auth->assign('admin','adminD');
```

信息：虽然上面的例子看起来冗长和枯燥，这主要是为示范的目的。开发者通常需要制定一些用户接口，以便最终用户可以更直观使用它来建立一个授权等级。

使用业务规则

当我们定义授权等级时，我们可以以一个所谓的“业务规则”来关联一个角色，一个任务或一个操作。 我们也可以在为一个用户指定角色时关联一个业务规则。一个业务规则是一个 **PHP** 代码片段，在我们执行访问检查时被执行。 返回值被用来决定是否角色或赋值适用于当前用户。在上面的例子中，我们关联一个业务规则到任务 `updateOwnPost`。 在业务规则里我们只检查是否当前用户 `ID` 和指定帖子的作者 `ID` 是相同的。在执行访问检查时，数组 `$params` 中的帖子信息由开发者提供。

访问检查

为了执行访问检查，我们得先知道 授权项目的名字。例如，如果要检查当前用户是否可以创建一个发布，我们将检查是否有 `createPost` 操作的权限。然后，我们调用 [CWebUser::checkAccess](#) 执行访问检查：

```
if(Yii::app()->user->checkAccess('createPost'))  
  
{  
  
    // 创建发布  
  
}
```

如果授权规则关联了需要额外参数的业务规则，我们传递参数。例如，要检查如果用户是否可以更新发布，我们将编写

```
$params=array('post'=>$post);  
  
if(Yii::app()->user->checkAccess('updateOwnPost',$params))  
  
{  
  
    // 更新post  
  
}
```

使用缺省角色

注意: 缺省角色的功能是在 1.0.3 版本引入的。

很多 **Web** 应用需要一些很特殊的角色，它们通常需要被分配给几乎每一个用户。例如，我们可能需要为所有注册用户分配一些特殊的权限。假如要象上述方法那样去为每一个用户分配这种角色，我们在维护上将面临很多麻烦。因此，我们采用 **缺省角色** 功能来解决这个问题。

所谓缺省角色指的是被隐式分配给每一个用户（包括注册和非注册的用户）的角色。我们无需明确分配它给一个用户。当我们调用 `CWebUser::checkAccess`，缺省角色将首先被检查，就像它们已经被分配给当前用户一样。

缺省角色必须通过 [CAuthManager::defaultRoles](#) 属性进行声明。例如，下面的应用配置声明了两个缺省角色：`authenticated` 和 `guest`。

```
return array(

    'components'=>array(

        'authManager'=>array(

            'class'=>'CdbAuthManager',

            'defaultRoles'=>array('authenticated', 'guest'),

        ),

    ),

);
```

因为缺省角色实质上是分配给每一个用户的，它通常需要伴随一个业务规则用来确定它是否真正适用某个用户。例如，下面的代码定义了两个角色，**authenticated** 和 **guest**，它们在实质上分别被分配给已通过验证和未通过验证的用户。

```
$bizRule='return !Yii::app()->user->isGuest;';

$auth->createRole('authenticated','authenticated user',$bizRule);

$bizRule='return Yii::app()->user->isGuest;';

$auth->createRole('guest','guest user',$bizRule);
```

主题

Theming 是一个在 Web 应用程序里定制网页外观的系统方式。通过采用一个新的主题，网页应用程序的整体外观可以立即和戏剧性的改变。

在 Yii，每个主题由一个目录代表，包含 **view** 文件，**layout** 文件和相关的资源文件，如图片，**CSS** 文件，**JavaScript** 文件等。主题的名字就是它的目录名字。全部主题都放在在同一目录 **WebRoot/themes** 下。在任何时候，只有一个主题可以被激活。

提示：默认的主题根目录 `WebRoot/themes` 可被配置成其他的。只需要配置 `themeManager` 应用部件的属性 `basePath` 和 `baseUrl` 为你所要的值。

要激活一个主题，设置 Web 应用程序的属性 `theme` 为你所要的名字。可以在 [application configuration](#) 中配置或者在执行过程中在控制器的动作里面修改。

注：主题名称是区分大小写的。如果您尝试启动一个不存在的主题，`yii::app()->theme` 将返回 `null`。

主题目录里面内容的组织方式和 [application base path](#) 目录下的组织方式一样。例如，所有的视图文件必须位于 `views` 下，布局视图文件在 `views/layouts` 下，和系统视图文件在 `views/system` 下。例如，如果我们要替换 `PostController` 的 `create` 视图文件为 `classic` 主题下，我们将保存新的视图文件为 `WebRoot/themes/classic/views/post/create.php`。

对于在 [module](#) 里面的控制器视图文件，相应主题视图文件也将被放在 `views` 目录下。例如，如果上述的 `PostController` 是在一个命名为 `forum` 的模块里，我们应该保存 `create` 视图文件为 `WebRoot/themes/classic/views/forum/post/create.php`。如果 `forum` 模块嵌套在另一个名为 `support` 模块里，那么视图文件应为 `WebRoot/themes/classic/views/support/forum/post/create.php`。

注：由于 `views` 目录可能包含安全敏感数据，应当配置以防止被网络用户访问。

当我们调用 [render](#) 或 [renderPartial](#) 显示视图，相应的 `view` 文件以及布局文件将在当前激活的主题里寻找。如果发现，这些文件将被渲染。否则，就后退到 [viewPath](#) 和 [layoutPath](#) 所指定的默认位置寻找。

提示：在一个主题的视图内部，我们经常需要链接其他主题资源文件。例如，我们可能要显示一个在主题下 `images` 目录里的图像文件。使用当前激活主题的 `baseUrl` 属性，我们就可以为此图像文件生成如下 url，

```
yii: :app()->theme->baseUrl . '/images/FileName.gif'
```

下面的例子是：一个应用有两个主题，`basic` 和 `fancy`，它们的目录结构如下：

```
WebRoot/

    assets

    protected/

        .htaccess

        components/

        controllers/

        models/
```

```
views/

    layouts/

        main.php

    site/

        index.php

themes/

    basic/

        views/

            .htaccess

            layouts/

                main.php

            site/

                index.php

    fancy/

        views/

            .htaccess

            layouts/

                main.php

            site/

                index.php
```

在应用配置中，若我们配置

```
return array(

    'theme'=>'basic',

    .....
```

```
);
```

basic 主题将生效，意味着应用的布局将使用 目录 **themes/basic/views/layouts** 下的视图，站点的 **index** 视图将使用 目录 **themes/basic/views/site** 下的视图。若在主题中没有找到一个视图文件，它 将后退到目录 **protected/views**。

皮肤

注意： 皮肤特征自版本 **1.1.0** 可用。

使用一个主题我们可以快速改变视图的外观，我们可以使用皮肤来系统化的定制视图中的 **widget** 的外观。

一个皮肤是一个由 **name-value pairs** 组成的数组，可被用来初始化一个 **widget** 的属性。 一个 **skin** 属于一个 **widget** 类，一个 **widget** 类可有多由它们名字识别的皮肤。 例如，我们可有 [CLinkPager](#) widget 的一个皮肤，它的名字是 **classic**。

为了使用皮肤特征，我们首先需要改变应用配置来安装 **widgetFactory** 组件：

```
return array(

    'components'=>array(

        'widgetFactory'=>array(

            'class'=>'CWidgetFactory',

        ),

    ),

);
```

然后我们创建所需的皮肤。属于同一个 **widget** 类的皮肤被存储在一个名字和此 **widget** 类相同的单个 **PHP** 脚本文件中。默认地，所有这些皮肤文件存储在目录 **protected/views/skins** 中。 若你要改变为一个别的目录，你可以配置 **widgetFactory** 的 **skinPath** 属性。 例如，我们创建一个名为 **CLinkPager.php** 的文件到目录 **protected/views/skins**，它的内容如下，

```
<?php

return array(

    'default'=>array(
```

```

        'nextPageLabel'=>' ',

        'prevPageLabel'=>' ',

    ),

    'classic'=>array(

        'header'=>' ',

        'maxButtonCount'=>5,

    ),

);

```

在上面,我们 `CLinkPager` widget 创建了两个皮肤: `default` 和 `classic`。前者是我们不明确指定 `CLinkPager` 的 `skin` 属性时使用的皮肤,而后者是其 `skin` 属性被指定为 `classic` 时使用的皮肤。例如,在下面的视图代码中, 第一个 `pager` 将使用 `default` 皮肤而第二个使用 `classic` 皮肤:

```

<?php $this->widget('CLinkPager'); ?>

<?php $this->widget('CLinkPager', array('skin'=>'classic')); ?>

```

若我们使用一个属性初始值列表创建 widget, they will take precedence and be merged with any applicable skin。例如, 下面视图代码将创建一个 `pager`, 它的初始值是 `array('header' =>' ', 'maxButtonCount' =>6, 'cssFile' =>false)`, 它是在视图中的属性初始值和 `classic` 皮肤结合的结果。

```

<?php $this->widget('CLinkPager', array(

    'skin'=>'classic',

    'maxButtonCount'=>6,

    'cssFile'=>false,

));

?>

```

注意皮肤特征无需使用主题。然而, 当使用主题时, `Yii` 将也在主题的视图目录中的 `skins` 目录寻找皮肤。(例如 `WebRoot/themes/classic/views/skins`)。若在以上两个地方都找到了, 主题皮肤有优先权。

若一个 `widget` 使用一个不存在的皮肤。Yii 将仍然照常创建此 `widget` 而不出现任何错误。

信息: 使用皮肤将降低性能，因为 Yii 在一个 `widget` 首次被创建时需要寻找皮肤文件。

日志

Yii 提供了一个灵活和可扩展的日志特征。被记录的信息可以根据日志等级和信息分类来被分类。使用等级和分类过滤器，被选择的信息可以进一步发送到不同的目的地，例如文件，电子邮件，浏览器窗口，等。

信息记录

调用 [Yii::log](#) 或 [Yii::trace](#) 均可记录信息。二者不同点是后者只有当应用处于[调试模式](#)时才记录信息。

```
Yii::log($message, $level, $category);

Yii::trace($message, $category);
```

当记录一条信息时，我们需要指定它的分类和等级。分类是一个格式是 `xxx.yyy.zzz` 的字符串，类似于[路径别名](#)。例如，若一条信息在 [CController](#) 中被记录，我们可以使用分类 `system.web.CController`。信息等级应该是如下值之一：

- `trace`: 这是 [Yii::trace](#) 使用的等级。用于在开发阶段跟踪应用执行的流程。
- `info`: 用于记录普通信息。
- `profile`: this is for performance profile which is to be described shortly.
- `warning`: 用于警告信息。
- `error`: 用于致命的错误信息。

信息发送

使用 [Yii::log](#) 或 [Yii::trace](#) 记录的信息被保存在内存中。通常我们需要显示它们到浏览器中，或者保存在一些持久存储器中，例如文件，`email`。这称为 *信息发送(message routing)*，也就是发送信息到不同的目的地。

在 Yii 中，信息发送由 [CLogRouter](#) 应用组件管理。它管理一整套所谓的 *log routes*。每个 *log route* 代表了一个单独的记录目的地。Messages sent along a log route can be filtered according to their levels and categories.

要使用信息发送，我们需要安装并预载入 [CLogRouter](#) 应用组件。我们也需要使用我们想要的日志 *route* 配置它的 [routes](#) 属性。下面展示一个应用配置：

```
array(  
    .....
```



```
'preload'=>array('log'),

'components'=>array(

    .....

    'log'=>array(

        'class'=>'CLogRouter',

        'routes'=>array(

            array(

                'class'=>'CFileLogRoute',

                'levels'=>'trace, info',

                'categories'=>'system.*',

            ),

            array(

                'class'=>'CEmailLogRoute',

                'levels'=>'error, warning',

                'emails'=>'admin@example.com',

            ),

        ),

    ),

),

),

)
```

在上面，我们有两个 log routes。第一个 route 是 [CFileLogRoute](#)，它保存日志到应用的 runtime 目录中，只有等级是 `trace` 或 `info` 并且分类以 `system.` 开始的信息才被保存。第二个 route 是 [CEmailLogRoute](#)，它发送信息到指定的 email 地址，只有等级是 `error` 或 `warning` 的信息才被发送。

在 Yii 中如下 log routes 可用：

- [CDbLogRoute](#): 保存信息到数据库中。

- [CEmailLogRoute](#): 发送信息到指定的 email 地址。
- [CFileLogRoute](#): 保存信息到应用的 runtime 目录中。
- [CWebLogRoute](#): 在当前网页尾部显示信息。
- [CProfileLogRoute](#): displays profiling messages at the end of the current Web page.

信息: Message routing 发生在当前请求周期的尾部，当 [onEndRequest](#) 事件被唤起时。要明确终止当前请求的处理，调用 [CApplication::end\(\)](#) 代替 `die()` 或 `exit()`，因为 [CApplication::end\(\)](#) 将唤起 [onEndRequest](#) 事件，以便信息可以被恰当的记录。

信息过滤

如我们提及的，在信息被发送到 log route 之前可以根据其等级和分类被过滤。这可以通过设定对应 log route 的 [levels](#) 和 [categories](#) 属性来完成。多个等级或分类应当以逗号连接起来。

因为信息分类的格式是 `xxx.yyy.zzz`，我们将它们视为一个 category hierarchy。特别的，我们说 `xxx` 是 `xxx.yyy` 的上级，而 `xxx.yyy` 是 `xxx.yyy.zzz` 的上级。然后我们可使用 `xxx.*` 代表分类 `xxx` 及其所有的子类和子类的子类。

记录上下文信息

自版本 1.0.6 开始，我们可以指定记录额外的上下文信息，例如 PHP 预定义变量(如 `$_GET`, `$_SERVER`), session ID, 用户名, 等。这通过指定一个 log route 的 [CLogRoute::filter](#) 属性为一个合适的日志过滤器来实现。

框架拥有方便的 [CLogFilter](#) 可以作为在大多数情况下所需的日志过滤器。默认的，[CLogFilter](#) 将记录一条信息 含有一些变量，如 `$_GET`, `$_SERVER` 它们常包含了有价值的系统上下文信息。[CLogFilter](#) 也可以被配置用来使用 session ID, username 等作为每条被记录信息的前缀，当我们检查众多的日志信息时，可以极大的简化全局搜索。

下面的配置展示了如何启用记录上下文信息。注意每个 log route 可以有自己的日志过滤器。默认的，一个 log route 没有日志过滤器。

```
array(
    .....

    'preload'=>array('log'),

    'components'=>array(
        .....

        'log'=>array(

            'class'=>'CLogRouter',

            'routes'=>array(
```

```
array(  
    'class'=>'CFileLogRoute',  
    'levels'=>'error',  
    'filter'=>'CLogFilter',  
),  
    ...other log routes...  
),  
),  
),  
)
```

自版本 1.0.7 开始，Yii 支持记录调用栈信息，这些信息通过调用 `Yii::trace` 得到。默认禁用此特征，因为它降低性能。要使用此特征，只需要在入口文件的开始(在引入 `yii.php` 之前)定义一个名为 `YII_TRACE_LEVEL` 的常量，它是一个大于 0 的整数。Yii 将然后追加文件名和调用栈的行号到每条跟踪信息中。数字 `YII_TRACE_LEVEL` 决定每个调用栈的几层应当被记录。在开发阶段此信息特别有用，因为它可以帮助我们识别触发跟踪信息的位置。

Performance Profiling

Performance profiling 是信息记录的一个特殊类型。它被用来衡量指定代码块需要的时间并找到性能瓶颈在哪里。

要使用 performance profiling，我们需要识别哪个代码块需要被 **profiled**。我们通过插入下面的方法来标记每个代码块的开始和结束：

```
Yii::beginProfile('blockID');  
  
...code block being profiled...  
  
Yii::endProfile('blockID');
```

`blockID` 是一个唯一识别此代码块的 ID。

注意，代码块需要被恰当的嵌套。也就是，一个代码块不能和其他的交叉。它必须是在一个并行的等级或者被其他代码块完整地包围起来。

要展示 profiling 结果，我们需要安装一个 [CLogRouter](#) 应用组件和一个 [CProfileLogRoute](#) log route。这和普通的信息 routing 是相同的。[CProfileLogRoute](#) route 将在每个页面底部展示性能结果。

Profiling SQL Executions

当使用数据库时，Profiling 特别有用，因为 SQL 执行通常是一个应用的主要性能瓶颈。自版本 1.0.6 起，我们可以手工插入 `beginProfile` 和 `endProfile` 语句在恰当的位置以衡量每个 SQL 执行花费的时间。同时 Yii 提供了一个更加系统化的方法来解决这个问题。

通过在应用配置中设置 [CDbConnection::enableProfiling](#) 为 `true`，每个被执行的 SQL 语句将被 profiled。结果可以使用之前描述的 [CProfileLogRoute](#) 容易的显示出来，可以告诉我们哪个 SQL 语句执行花费了多少时间。我们也可以调用 [CDbConnection::getStats\(\)](#) 来检索总的被执行的 SQL 语句数量以及它们总的执行时间。

错误处理

Yii 在 PHP 5 的异常机制基础上提供了一个完整的错误处理框架。当应用被创建来处理一个新来的用户请求，它注册它的 [handleError](#) 方法来处理 PHP 警告和提示；同时它注册它的 [handleException](#) 方法来处理未捕获的 PHP 异常。所以，若一个 PHP warning/notice 或一个未捕获的异常出现于应用执行中，其中一个 error handler 将接管控制并开始必要的错误处理过程。

提示： 错误处理器的注册是应用的构造器中调用 PHP 函数设置 exception handler 和 error handler 实现的。若你不需要 Yii 处理错误和异常，你可以在入口脚本中定义常量 `YII_ENABLE_ERROR_HANDLER` 和 `YII_ENABLE_EXCEPTION_HANDLER` 为 `false`。

默认情况下，[errorHandler](#) (或 [exceptionHandler](#)) 将唤起一个 `onError` 事件 (或 `onException` 事件)。若错误(或 异常)没有被任何事件处理器处理，它将从 [errorHandler](#) 应用组件请求帮助。

唤起异常

在 Yii 中唤起异常和唤起一个普通 PHP 异常没有什么不同。使用下面句法在需要时唤起一个异常：

```
throw new ExceptionClass('ExceptionMessage');
```

Yii 定义了两个异常类: [CException](#) 和 [CHttpException](#)。前者是一个通用异常类，而后者代表了一个应显示给终端用户的异常。后者也使用一个 `statusCode` 属性代表一个 HTTP 状态代码。一个异常的类决定它如何被显示，我们将在下面解释。

提示： 唤起一个 [CHttpException](#) 异常是报告由用户误操作引起的错误的简单方式。例如，若用户在 URL 中提供了一个无效的 post ID，我们可以这样做显示一个 404 错误(页面未找到)：

```
// if post ID is invalid
```

```
throw new CHttpException(404,'The specified post cannot be found.');
```

显示错误

当一个错误被推送到 **CErrorHandler** 应用组件，它选择一个合适的视图来显示错误。若错误需要被显示给终端用户，例如一个 **CHttpException**，它将使用一个名为 **errorXXX** 的视图，其中 **XXX** 代表 **HTTP** 状态代码(例如 400, 404, 500)。若错误是内部错误并只显示给开发者，它将使用一个名为 **exception** 的视图。若是后者，完整的调用栈及错误行信息也将被显示。

信息：当应用处于生产模式，所有错误包括内部错误将使用视图 **errorXXX** 来显示。这是因为一个错误的调用栈可能包含敏感信息。这种情况下，开发者应当依靠错误日志来找出引起错误的真正原因。

CErrorHandler 以以下顺序寻找一个视图的视图文件：

1. **WebRoot/themes/ThemeName/views/system**: 这是当前激活的主题的 **system** 视图目录。
2. **WebRoot/protected/views/system**: 这是一个应用的默认的 **system** 视图目录。
3. **yii/framework/views**: 这是由 **Yii** 框架提供的标准系统视图目录。

因此，若我们想自定义错误显示，我们可以只需要在应用或主题的系统视图目录下创建错误视图文件。每个视图文件是一个普通的 **PHP** 脚本，其中大部分是 **HTML** 代码。更多细节，请参考框架的 **view** 目录下的视图文件。

使用一个 action 处理错误

自版本 1.0.6 开始，**Yii** 允许使用一个 **controller action** 处理错误显示。要这样做，我们应当在应用配置中配置错误处理器，如下：

```
return array(  
    .....  
    'components'=>array(  
        'errorHandler'=>array(  
            'errorAction'=>'site/error',  
        ),  
    ),  
);
```

在上面，我们配置 `CErrorHandler::errorAction` 属性 为路由 `site/error`，这个路由指向 `SiteController` 的 `errorAction`。若需要我们可以使用一个不同的路由。

我们编写 `error action` 如下：

```
public function actionError()
{
    if($error=Yii::app()->errorHandler->error)
        $this->render('error', $error);
}
```

在上面，我们首先从 `CErrorHandler::error` 检索详细的错误信息。若它非空，我们使用错误信息渲染 `error` 视图。`CErrorHandler::error` 返回的错误信息是一个数组，它有如下字段：

- `code`: HTTP 状态代码 (例如 403, 500);
- `type`: 错误类型 (例如 `CHttpException`, `PHP Error`);
- `message`: 错误信息;
- `file`: 出现错误的 PHP 脚本的名字;
- `line`: 出现错误的代码行数;
- `trace`: 错误的调用栈;
- `source`: 错误出现的源代码片段。

提示: 我们检查 `CErrorHandler::error` 是否为空的原因是因为 `error action` 可由终端用户直接请求，若没有错误出现。因为我们传递 `$error` 数组到视图中，它将自动展开到各个变量。结果，在视图中，我们可以直接访问变量，例如 `$code`, `$type`。

信息记录

当错误出现时，错误等级 `error` 总是被记录。若错误由一个 PHP `warning` 或 `notice` 引起，信息被记录到分类 `php`；若错误由一个未捕获的异常，分类将是 `exception.ExceptionClassName` (对于 `CHttpException`，它的 `statusCode` 也将被追加到分类中)。因此可以利用 `logging` 特征来监视应用执行中发生的错误。

Web Service

Web service 是一个软件系统，设计来支持计算机之间跨网络相互访问。对于 Web 应用程序，它通常用一套 API，可以被互联网访问和执行在远端系统主机上的被请求服务。系统主机所要求的服务。例如，以 **Flex** 为基础的客户端可能会援引函数实现在服务器端运行 PHP 的 Web 应用程序。Web service 依赖 **SOAP** 作为通信协议栈的基础层。

Yii 提供 `CWebService` 和 `CWebServiceAction` 简化了在 Web 应用程序实现 Web service。这些 API 以类形式实现，被称为 *service providers*。Yii 将为每个类产生一个 `WSDL`，描述什么 API 是可用的以及它们如何被客户端调用。当一个 API 被客户端调用，Yii 将实例化相应的 *service provider* 并调用被请求的 API 来完成请求。

注: `CWebService` 依靠 `PHP SOAP extension`。请确保您在试用本节中的例子前开启此扩展。

定义 Service Provider

正如我们上文所述，一个 *service provider* 是一个定义了能被远程调用的方法的类。Yii 依靠 `doc comment` 和 `class reflection` 识别 哪些方法可以被远程调用，以及他们的参数和返回值是什么。

让我们以一个简单的股票引用服务开始。这项服务允许客户端请求指定股票的报价引用。我们定义 *service provider* 如下。请注意， 我们通过扩展 `CController` 定义了这个 *provider* 类 `StockController`。这是不是必需的。马上我们将解释为什么这样做。

```
class StockController extends CController
{
    /**
     * @param string the symbol of the stock
     * @return float the stock price
     * @soap
     */
    public function getPrice($symbol)
    {
        $prices=array('IBM'=>100, 'GOOGLE'=>350);

        return isset($prices[$symbol])?$prices[$symbol]:0;

        //...return stock price for $symbol
    }
}
```

在上面的，我们通过在文档注释中的 `@soap` 标签声明 `getPrice` 方法为一个 Web service API。我们依靠文档注释指定输入的参数类型和返回值的类型。其他的 API 可使用类似方式声明。

声明 Web Service Action

已经定义了 `service provider`，我们使他能够通过客户端访问。特别是，我们要创建一个控制器动作暴露这个服务。做到这一点很容易，在控制器类中定义一个 `CWebServiceAction` 动作。对于我们的例子中，我们把它放在 `StockController` 中。

```
class StockController extends CController
{
    public function actions()
    {
        return array(
            'quote'=>array(
                'class'=>'CWebServiceAction',
            ),
        );
    }

    /**
     * @param string the symbol of the stock
     * @return float the stock price
     * @soap
     */
    public function getPrice($symbol)
    {
        //...return stock price for $symbol
    }
}
```



```
}
```

这就是我们需要建立的 Web service! 如果我们尝试访问动作 `http://hostname/path/to/index.php?r=stock/quote` , 我们将看到很多 XML 内容, 这实际上是我们定义的 Web service 的 WSDL 描述。

提示: 在默认情况下, `CWebServiceAction` 假设当前的控制器是 service provider。这就就是为什么我们定义 `getPrice` 方法在 `StockController` 中的原因。

消费 Web Service

要完成这个例子, 让我们创建一个客户端来消费我们刚刚创建的 Web service。例子中的客户端用 php 编写的, 但可以用别的语言编写, 例如 Java, C#, Flex 等等。

```
$client=new SoapClient('http://hostname/path/to/index.php?r=stock/quote');  
  
echo $client->getPrice('GOOGLE');
```

在网页中或控制台模式运行以上脚本, 我们将看到 GOOGLE 的价格 350 。

数据类型

当声明类的方法和属性是远程可访问的, 我们需要指定输入和输出参数的数据类型。以下的原始数据类型可以使用:

- str/string: 对应 xsd:string;
- int/integer: 对应 xsd:int;
- float/double: 对应 xsd:float;
- bool/boolean: 对应 xsd:boolean;
- date: 对应 xsd:date;
- time: 对应 xsd:time;
- datetime: 对应 xsd:dateTime;
- array: 对应 xsd:string;
- object: 对应 xsd:struct;
- mixed: 对应 xsd:anyType.

如果类型不属于上述任何原始类型, 它被看作是由属性组成的复合类型。一个复合型类型被看做一个类, 它的属性当做类的公有成员变量, 在文档注释中被用 `@soap` 标记。

我们还可以使用数组类型, 通过附加 `[]` 在原始或复合型类型的后面。这将指定一个指定类型的数组。

下面就是一个例子，定义 `getPosts` Web API，返回一个 `Post` 对象的数组。

```
class PostController extends CController
{
    /**
     * @return Post[] a list of posts
     * @soap
     */
    public function getPosts()
    {
        return Post::model()->findAll();
    }
}
```

```
class Post extends CActiveRecord
{
    /**
     * @var integer post ID
     * @soap
     */
    public $id;

    /**
     * @var string post title
     * @soap
     */
    public $title;
```

```
public static function model($className= CLASS )
{
    return parent::model($className);
}
}
```

类映射

为了从客户端得到复合型参数，应用程序需要定义从 WSDL 类型到相应 PHP 类的映射。这是通过配置 [CWebServiceAction](#) 的属性 [classMap](#) 完成的。

```
class PostController extends CController
{
    public function actions()
    {
        return array(
            'service'=>array(
                'class'=>'CWebServiceAction',
                'classMap'=>array(
                    'Post'=>'Post', // or simply 'Post'
                ),
            ),
        );
    }
    .....
}
```

```
}
```

拦截远程方法调用

通过实现 [IWebServiceProvider](#) 接口，service provider 可以拦截远程方法调用。

在 [IWebServiceProvider::beforeWebMethod](#) 中，service provider 可以检索当前 [CWebService](#) 实例并通过 [CWebService::methodName](#) 得到当前被请求的方法的名字。它可以返回 `false` 如果远程方法出于某种原因不应被调用（例如：未经授权的访问）。

国际化

Internationalization (I18N) 指的是设计一个软件应用使它可以适用于各种语言和各种地区而无需修改核心代码。对于 Web 应用，这点特别重要因为潜在的用户可能来自世界各地。

Yii 在几个方面提供了 I18N 支持。

- 它为每个可能的语言和变量(variant)提供了本地化数据(the locale data)。
- 它提供了信息和文件翻译服务。
- 它提供了 locale-dependent 日期和时间格式。
- 它提供了 locale-dependent 数字格式。

在下面的子章节中，我们将详细解释上面的每个方面。

区域和语言

Locale 是一个参数集合，它定义了用户的语言，国家和首选项。它通常由一个 ID 定义，包含一个语言 ID 和一个区域 ID。例如，ID `en_US` 代表了 the locale of English and United States。。为了一致性，所有 locale IDs 在 Yii 中被规划为 `LanguageID` 或 `LanguageID_RegionID` 以小写格式(例如 `en`, `en_us`)。

Locale data 被表示为一个 [CLocale](#) 实例。它提供了 locale-dependent 信息，包括货币符号，数字符号，货币格式，数字格式，日期和时间格式，和日期相关的名字。由于语言信息已经由 the locale ID 表示，不由 [CLocale](#) 提供。基于相同的原因，我们经常可以互换使用术语 `locale` 和 `language`。

给出一个 locale ID，使用 `CLocale::getInstance($localeID)` 或 `CApplication::getLocale($localeID)` 可以得到对应的 [CLocale](#) 实例。

信息： Yii 为几乎每种语言和区域配有 locale 数据。数据来自于 [Common Locale Data Repository](#) (CLDR)。对于每个 locale, only a subset of the CLDR data is provided as the original data contains a lot of rarely used information。自版本 1.1.0 开始，用户也可以使用他们自己的定制的 locale 数据。要这样做，配置 [CApplication::localeDataPath](#) 属性为含有定制后 locale 数据的目录。请参考 `framework/i18n/data` 下的 the locale data files under `framework/i18n/data` 创建自定义的 locale data files。

对于一个 Yii 应用，我们区别于它的 [target language](#) 和 [source language](#)。目标语言是应用面向的用户选择的语言，而源语言指的是应用代码编写使用的语言。当二者不同时，需要国际化。

你可以配置 [target language](#) 在 [application configuration](#)，或者在任何国际化发生之前动态的改变它。

提示： 有时，我们想要设置目标语言为用户首选的语言（由用户浏览器首选项指定）。要这样做，我们可以使用 [CHttpRequest::preferredLanguage](#) 检索用户首选的语言 ID。

翻译

最常需要的 I18N 特征可能是翻译，包括信息翻译和视图翻译。前者翻译一条文本信息到想要的语言，而后者翻译一整个文件到想要的语言。

一个翻译请求由需要被翻译的对象，对象的源语言，以及对象需要被翻译到的目标语言组成。在 Yii 中，源语言默认是 [application source language](#)，而目标语言默认是 [application language](#)。若源语言和目标语言相同，无需翻译。

信息翻译

信息翻译通过调用 [Yii::t\(\)](#) 完成。此方法翻译给定的信息从 [source language](#) 到 [target language](#)。

当翻译一条信息时，需要指定它的分类，因为一条信息在不同的分类中翻译可能是不同的。分类 `yii` 被保留用于 Yii 框架核心代码使用的信息。

信息可以包含参数占位符，当调用 [Yii::t\(\)](#) 时，这些占位符由具体值替代。例如，下面的信息翻译请求将替换原始信息中的 `{alias}` 占位符为实际的 `alias` 值。

```
Yii::t('app', 'Path alias "{alias}" is redefined.',
    array('{alias}'=>$alias))
```

注意： 需要被翻译的信息必须是常量字符串。不应当包含改变信息内容的变量（例如 `"Invalid {$message} content."`）。若需要根据一些参数改变信息内容，请使用参数占位符。

翻译后的信息被存储到一个仓库中，称为 *message source*。一个 *message source* 表示为一个 [CMessageSource](#) 或其子类的实例。当 [Yii::t\(\)](#) 被调用时，它将在 *the message source* 中寻找信息，若找到，返回翻译后的信息。

Yii 有如下 *message sources* 类型。你也可以扩展 [CMessageSource](#) 以创建你自己的 *message source* 类型。

- [CPhpMessageSource](#): 信息翻译被存储为 key-value 对到一个 PHP 数组中。原始信息是键，翻译后的信息是值。每个数组代表一个特定信息分类的翻译，并保存到一个单独的 PHP 脚本中，文件名就是分类名。相同语言的 PHP 翻译文件被保存到相同的目录中，以 *locale ID* 命名。所有这些目录放置到由 [basePath](#) 指定的目录中。
- [CGettextMessageSource](#): 信息翻译被保存为 [GNU Gettext](#) 文件。

- [CdbMessageSource](#): 信息翻译被保存到数据库表中。更多信息, 查看 API 文档中的 [CdbMessageSource](#).

一条信息源码被载入为一个 [application component](#)。Yii 预声明了一个应用组件名为 [messages](#) 来存储用户应用使用的信息。默认的, 信息源码的类型是 [CPhpMessageSource](#), 存储 PHP 翻译文件的基础路径是 `protected/messages`。

概括的说, 为了使用信息翻译, 需要如下步骤:

1. 在恰当的位置调用 [Yii::t\(\)](#);
2. 创建 PHP 翻译文件为 `protected/messages/LocaleID/CategoryName.php`。每个文件只是返回一个信息翻译数组。注意, 这里假设你在使用默认的 [CPhpMessageSource](#) 来存储翻译后的信息。
3. 配置 [CApplication::sourceLanguage](#) 和 [CApplication::language](#)。

提示: The `yiic` tool in Yii can be used to manage message translations when [CPhpMessageSource](#) is used as the message source. Its `message` command can automatically extract messages to be translated from selected source files and merge them with existing translations if necessary.

自版本 1.0.10, 当使用 [CPhpMessageSource](#) 管理信息源码, 一个扩展类 (例如一个 `widget`, 一个模块)的信息可以被特别的管理和使用。特别的, 若一条信息属于一个类名字为 `XYZ` 的扩展, 信息分类可以被指定为格式 `XYZ.categoryName`。对应的信息文件假设为 `BasePath/messages/LanguageID/categoryName.php`, `BasePath` 指的是包含扩展类文件的目录。并且当使用 `Yii::t()` 翻译一条扩展信息时, 应当使用如下格式:

```
Yii::t('XYZ.categoryName', 'message to be translated')
```

自版本 1.0.2, Yii 增加了支持 [choice format](#)。Choice format 指的是根据一个给定的数字值选择 `a translated`。例如, 在英语中单词 `'book'` 可以是单数或复数, 根据书的数量。而在其他语言中, 这个单词可以有不同的格式(例如中文) 或有更复杂的复数格式规则 (例如俄语)。Choice format 以一个简单而有效的方式解决了这个问题。

要使用 choice format, 一个翻译后的信息必须由一串 `expression-message` 对组成, 彼此之间由 `|` 分隔。如下:

```
'expr1#message1|expr2#message2|expr3#message3'
```

`exprN` 指的是一个有效的 PHP 表达式, 它被评估为一个 `boolean` 值, 这个 `boolean` 值指示是否对应的信息应当被返回。只有第一个表达式评估为 `true` 对应的信息才被返回。表达式可以包含一个名为 `n` (注意, 不是 `$n`) 的特殊变量, 它将传递数字值作为第一个信息参数。例如, 假设一条翻译后的信息是:

```
'n==1#one book|n>1#many books'
```

当调用 [Yii::t\(\)](#) 时我们传递数字 2 到信息参数数组中, 我们应当得到 `many books` 作为最终翻译后的信息。

作为一个快捷方法, 若一个表达式是一个数字, 它将被对待为 `n==Number`。因此, 上面的翻译后的信息也可以写成:

```
'1#one book|n>1#many books'
```

文件翻译

文件翻译通过调用 [CApplication::findLocalizedFile\(\)](#) 完成。给出被翻译的文件的路径，此方法将在 `LocaleID` 子目录中寻找同名文件。若找到，返回文件路径；否则，返回原始文件路径。

文件翻译主要用于渲染一个视图时。当在一个控制器或 `widget` 中调用一个渲染方法时，视图文件将自动被翻译。例如，若 [target language](#) 为 `zh_cn` 而 [source language](#) 是 `en_us`，渲染一个名为 `edit` 的视图将搜索视图文件 `protected/views/ControllerID/zh_cn/edit.php`。若文件被找到，这个被翻译后的版本将用来渲染；否则，文件 `protected/views/ControllerID/edit.php` 将被用来渲染。

文件翻译也可以被用于其它目的，例如，显示一个翻译后的图像或载入一个 `locale-dependent` 数据文件。

日期和时间格式化

日期和时间在不同的国家和地区常常是不同的。日期和时间格式化的任务就是生成指定格式的日期或时间字符串。Yii 为此提供了 [CDateFormatter](#)。

每个 [CDateFormatter](#) 实例和一个目标 `locale` 相关。要在整个应用中得到此 `target locale` 指定的格式，我们只需访问应用的 [dateFormatter](#) 属性。

[CDateFormatter](#) 类主要提供了两个方法来格式化一个 `UNIX` 时间戳。

- [format](#): 此方法格式化给出的 `UNIX` 时间戳为一个字符串，根据一个自定义的格式 (例如 `$dateFormatter->format('yyyy-MM-dd', $timestamp)`)。
- [formatDateTime](#): 此方法格式化给定的 `UNIX` 时间戳为一个字符串，根据一个预定义在目标 `locale` 数据中的 `pattern` (例如 `short` 日期格式，`long` 时间格式)。

数字格式化

类似于日期和时间，数字在不同的国家和地区通常格式也是不同的。数字格式化包含十进制格式化(decimal formatting)，货币格式化和百分比格式化。Yii 为这些任务提供了 [CNumberFormatter](#)。

要在整个应用中得到和指定地区相应的数字格式，我们可以访问应用的 [numberFormatter](#) 属性。

[CNumberFormatter](#) 提供如下方法来格式化一个 `integer` 或 `double` 值。

- [format](#): 此方法根据自定义的模式将给出的值格式化为一个字符串。(例如 `$numberFormatter->format('#, ##0.00', $number)`)。
- [formatDecimal](#): 此方法使用预定义在目标 `locale` 数据中的十进制格式来格式化给出的数字。

- [formatCurrency](#): 此方法使用在目标 `locale` 数据中预定义的货币模式来格式化给出的数字和货币。
- [formatPercentage](#): 此方法使用在目标 `locale` 数据中预定义的 百分比模式来格式化给出的数字。

使用可选的模板句法

Yii 允许开发者使用他们喜爱的模板句法 (例如 `Prado`, `Smarty`) 来编写控制器或 `widget` 的视图。这通过编写和安装一个 [viewRenderer](#) 应用组件来实现。The view renderer intercepts the invocations of `CBaseController::renderFile`, compiles the view file with customized template syntax, and renders the compiling results.

信息: 推荐只有当编写很少被重用的视图时才使用自定义的模板句法。 否则, 若他人重用此视图, 将不得不在他们的应用中也使用相同的模板句法。

下面, 我们介绍如何使用 [CPradoViewRenderer](#), 它是一个视图渲染器, 允许你使用使用类似于 [Prado 框架 framework](#) 中的模板句法。 对于那些要开发自己视图渲染器的开发者, [CPradoViewRenderer](#) 是个好的参考。

使用 CPradoViewRenderer

要使用 [CPradoViewRenderer](#), 我们只需要如下配置应用:

```
return array(  
  
    'components'=>array(  
  
        .....  
  
        'viewRenderer'=>array(  
  
            'class'=>'CPradoViewRenderer',  
  
        ),  
  
    ),  
  
);
```

默认情况下, [CPradoViewRenderer](#) 将编译视图文件源码并保存结果 PHP 文件到 `runtime` 目录中。 只有当视图文件源码被改变时, PHP 文件才被重新生成。 因此, 使用 [CPradoViewRenderer](#) 只遭受很小的性能损失。

提示: [CPradoViewRenderer](#) 主要引入了一些新模板标签 以使得更容易更快地编写视图, 在视图源码中你仍然可以使用 PHP 代码。

下面, 我们介绍 [CPradoViewRenderer](#) 支持的模板标签。

短 PHP 标签

短 PHP 标签是在一个视图中编写 PHP 表达式和语句的快捷方式。表达式标签 `<%= expression %>` 被翻译为 `<?php echo expression ?>`; 语句标签 `<% statement %>` 被翻译为 `<?php statement ?>`。例如，

```
<%= CHtml::textField($name, 'value'); %>

<% foreach($models as $model): %>
```

被翻译为

```
<?php echo CHtml::textField($name, 'value'); ?>

<?php foreach($models as $model): ?>
```

组件标签

组件标签被用来在一个视图中插入 [widget](#)。它使用如下句法:

```
<com:WidgetClass property1=value1 property2=value2 ...>

    // body content for the widget

</com:WidgetClass>

// a widget without body content

<com:WidgetClass property1=value1 property2=value2 .../>
```

`WidgetClass` 指定 `widget` 类名字或类的 [路径别名](#)，属性初始值可以使用引起来的字符串或一对弯括号引起来的 PHP 表达式。例如，

```
<com:CCaptcha captchaAction="captcha" showRefreshButton={false} />
```

将被翻译为

```
<?php $this->widget('CCaptcha', array(

    'captchaAction'=>'captcha',

    'showRefreshButton'=>false)); ?>
```

注意: `showRefreshButton` 的值被指定为 `{false}` 而不是 `"false"` 因为后者意味着是一个字符串而不是一个 boolean.

缓存标签

缓存标签是使用 [片段缓存](#) 的快捷方式，它的句法如下，

```
<cache:fragmentID property1=value1 property2=value2 ...>

    // content being cached

</cache:fragmentID >
```

`fragmentID` 应当是一个唯一标识被缓存内容的识别符， 属性-值 对被用来配置这个片段缓存。例如，

```
<cache:profile duration={3600}>

    // user profile information here

</cache:profile >
```

将被翻译为

```
<?php if($this->cache('profile', array('duration'=>3600))): ?>

    // user profile information here

<?php $this->endCache(); endif; ?>
```

Clip Tags

类似于缓存标签，clip tags 使用 `CBaseController::beginClip` 和 `CBaseController::endClip` 的快捷方式。句法如下，

```
<clip:clipID>

    // content for this clip

</clip:clipID >
```

clipID 是一个唯一识别 clip content 的识别符。上面的内容翻译如下

```
<?php $this->beginClip('clipID'); ?>

    // content for this clip

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

注释标签

注释标签被用来编写注释，只对开发者可见。注释标签在视图显示给用户时被截去。句法如下，

```
<!---

view comments that will be stripped off

--->
```

安全措施

跨站脚本攻击的防范

跨站脚本攻击(也称为 XSS)出现在当一个网络应用收集来自用户的恶意数据时。攻击者常常向易受攻击的 web 应用注入 JavaScript, VBScript, ActiveX, HTML 或 Flash 来愚弄其他用户并收集他们的信息。举个例子，一个未经良好设计的论坛系统可能不经检查就显示用户所输入的内容。攻击者可以在帖子内容中注入一段恶意的 JavaScript 代码。这样，当其他访客在阅读这个帖子的时候，这些 JavaScript 代码就可以在访客的电脑上运行了。

一个防范 XSS 攻击的最重要的措施之一就是：在显示用户输入的内容之前进行内容检查。比如，你可以对用户的输入进行 HTML 编码处理。但是在某些情况下这种方法就不可取了，因为这种方法禁用了所有的 HTML 标签。

Yii 集成了 [HTMLPurifier](#) 并且为开发者提供了一个很有用的组件 [CHtmlPurifier](#)，这个组件封装了 [HTMLPurifier](#)。它可以将通过彻底的检查、安全和白名单功能来把所审核的内容中的所有恶意代码清除掉，并且确保过滤之后的内容过滤符合标准。

[CHtmlPurifier](#) 组件可以作为一个 [widget](#) 或者 [filter](#) 来使用。 当作为一个 widget 来使用的时候, [CHtmlPurifier](#) 可以对在视图中显示的内容进行安全过滤。 以下是代码示例:

```
<?php $this->beginWidget('CHtmlPurifier'); ?>

//...这里显示用户输入的内容...

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

跨站请求伪造攻击的防范

跨站请求伪造(简称 **CSRF**)攻击,即攻击者在用户浏览器在访问恶意网站的时候,让用户的浏览器向一个受信任的网站发起攻击者指定的请求。 举个例子,一个恶意网站有一个图片,这个图片的 **src** 地址指向一个银行网站:

<http://bank.example/withdraw?transfer=10000&to=someone>。 如果用户在登陆银行的网站之后访问了这个恶意网页,那么用户的浏览器会向银行网站发送一个指令,这个指令的内容可能是“向攻击者的帐号转账 10000 元”。跨站攻击方式利用用户信任的某个特定网站,而 **CSRF** 攻击正相反,它利用用户在某个网站中的特定用户身份。

要防范 **CSRF** 攻击,必须遵守规则: **GET** 请求只允许检索数据而不能修改服务器上的任何数据。 而 **POST** 请求应当含有一些可以被服务器识别的随机数值,用来保证表单数据的来源和运行结果发送的去向是相同的。

Yii 实现了一个 **CSRF** 防范机制,用来帮助防范基于 **POST** 的攻击。这个机制的核心就是在 **cookie** 中设定一个随机数据,然后把它同表单提交的 **POST** 数据中的相应值进行比较。

默认情况下, **CSRF** 防范是禁用的。 如果你要启用它,在[应用配置](#)中设置组件 [CHttpRequest](#) 如下

代码示例:

```
return array(

    'components'=>array(

        'request'=>array(

            'enableCsrfValidation'=>true,

        ),

    ),

);
```

要显示一个表单,请使用 [CHtml::form](#) 而不要自己写 **HTML** 代码。因为 [CHtml::form](#) 可以自动地在表单中嵌入一个隐藏项,这个隐藏项储存着验证所需的随机数据,这些数据可在表单提交的时候发送到服务器进行验证。

Cookie 攻击的防范

保护 cookie 免受攻击是非常重要的。因为 session ID 通常存储在 Cookie 中。如果攻击者窃取到了一个有效的 session ID，他就可以使用这个 session ID 对应的 session 信息。

这里有几条防范对策：

- 您可以使用 SSL 来产生一个安全通道，并且只通过 HTTPS 连接来传送验证 cookie。这样攻击者是无法解密所传送的 cookie 的。
- 设置 cookie 的过期时间，对所有的 cookie 和 session 令牌也这样做。这样可以减少被攻击的机会。
- 防范跨站代码攻击，因为它可以在用户的浏览器触发任意代码，这些代码可能会泄露用户的 cookie。
- 验证 cookie 数据，探测它们是否被更改。

Yii 实现了一个 cookie 验证机制，可以防止 cookie 被修改。启用之后可以对 cookie 的值进行 HMAC 检查。

Cookie 验证在默认情况下是禁用的。如果你要启用它，可以编辑[应用配置](#)中的组件中的 [CHttpRequest](#) 部分。

代码示例：

```
return array(  
    'components'=>array(  
        'request'=>array(  
            'enableCookieValidation'=>true,  
        ),  
    ),  
);
```

一定要使用 Yii 提供的 cookie 验证方案，也需要使用 Yii 内置的 [cookies](#) 集合来访问 cookie，不要直接使用 `$_COOKIE`。

```
// 检索一个名为$name 的 cookie 值  
  
$cookie=Yii::app()->request->cookies[$name];  
  
$value=$cookie->value;  
  
.....  
  
// 设置一个 cookie
```

```
$cookie=new CHttpCookie($name,$value);

Yii::app()->request->cookies[$name]=$cookie;
```

性能调整

网络应用程序的性能受很多因素的影响。数据库存取，文件系统操作，网络带宽是所有潜在的影响因素。Yii 已在各个方面减少框架带来的性能影响。但是在用户的应用中仍有很多地方可以被改善来提高性能。

开启 APC 扩展

启用 [PHP APC extension](#) 可能是改善一个应用整体性能的最简单方式。此扩展缓存和优化 PHP 中间代码并避免时间花费在为每个新来的请求解析 PHP 脚本。

禁用调试模式

禁用调试模式是另一个改善性能容易方式。若常量 `YII_DEBUG` 被定以为 `true`, 这个 Yii 应用将以调试模式运行。调试模式在开发阶段是有用的，但是它影响性能因为一些组件引起额外的系统开销。例如，信息记录器(the message logger) 将为被条被记录的信息记录额外的调试信息。

使用 yiilite.php

当启用 [PHP APC extension](#) 时， 我们可以将 `yii.php` 替换为一个名为 `yiilite.php` 的另一个引导文件来进一步提高 Yii-powered 应用的性能。

文件 `yiilite.php` 包含在每个 Yii 发布中。它是一些常用到的 Yii 类文件的合并文件。在文件中，注释和跟踪语句都被去除。因此，使用 `yiilite.php` 将减少被引用的文件数量并避免执行跟踪语句。

注意，使用 `yiilite.php` 而不开启 APC 实际上将降低性能，因为 `yiilite.php` 包含了一些不是每个请求都必须的类，这将花费额外的解析时间。同时也要注意，在一些服务器配置下使用 `yiilite.php` 将更慢，即使 APC 被打开。最好使用演示中的 `hello world` 运行一个基准程序来决定是否使用 `yiilite.php`。

使用缓存技术

如在 [Caching](#) 章节所述，Yii 提供了几个可以有效提高性能的缓存方案。若一些数据的生成需要长时间，我们可以使用 [data caching](#) 方法来减少数据产生的频率；若页面的一部分保持相对的固定，我们可以使用 [fragment caching](#) 方法减少它的渲染频率；若一整个页面保持相对的固定，我们可以使用 [page caching](#) 方法来节省页面渲染所需的花销。

若应用在使用 [Active Record](#)，我们应当打开 the schema caching 以节省解析数据库模式(database schema)的时间。可以通过设置 [CDbConnection::schemaCachingDuration](#) 属性为一个大于 0 的值来完成。

除了这些应用水平的(application-level)缓存技术，我们也可使用服务器端的(server-level)缓存方案 来提高应用的性能。事实上，我们之前描述的 [APC caching](#) 就属于此项。也有其他的服务器技术，例如 [Zend Optimizer](#), [eAccelerator](#), [Squid](#), to name a few.

数据库优化

从数据库取出数据经常是一个网络应用的主要瓶颈。虽然使用缓存可以减少性能损失，它不能解决根本问题。当数据库包含大量数据而被缓存的数据是无效的，fetching the latest data could be prohibitively expensive without proper database and query design.

在一个数据库中聪明的设计索引。索引可以让 SELECT 查询更快，但它会让 INSERT, UPDATE 或 DELETE 查询更慢。

对于复杂的查询，推荐为它创建一个数据库视图而不是 issuing the queries inside the PHP code and asking DBMS to parse them repetitively.

不要滥用 [Active Record](#)。虽然 [Active Record](#) 擅长以一个 OOP 样式模型化数据，它实际上为了它需要创建一个或几个对象来代表每条查询结果降低了性能。对于数据密集的应用，在底层使用 [DAO](#) 或 database APIs 将是一个更好的选择。

最后但并不是最不重要的一点，在你的 SELECT 查询中使用 LIMIT。这将避免从数据库中取出过多的数据 并耗尽为 PHP 分配的内存。

最小化脚本文件

复杂的页面经常需要引入很多外部的 JavaScript 和 CSS 文件。因为每个文件将引起一次额外的往返一次，我们应当通过联合文件来最小化脚本文件的数量。我们也应当考虑减少每个脚本文件的大小来减少 网络传输时间。有很多工具来帮助改善这两方面。

对于一个 Yii 产生的页面，例外是一些由组件渲染的脚本文件我们不想要更改 (例如 Yii core 组件，第三方组件)。为了最小化这些脚本文件，我们需要两个步骤。

注意： 下面描述的 scriptMap 特征已自版本 1.0.3 起被支持。

首先，通过配置应用组件 [clientScript](#) 的 [scriptMap](#) 属性来声明脚本被最小化。可以在应用配置中完成，也可以在代码中配置。例如，

```
$cs=Yii::app()->clientScript;

$cs->scriptMap=array(

    'jquery.js'=>'/js/all.js',
```

```

    'jquery.ajaxqueue.js'=>'/js/all.js',

    'jquery.metadata.js'=>'/js/all.js',

    .....

);

```

上面的代码所做是映射这些 JavaScript 文件到 URL `/js/all.js`。若这些 JavaScript 文件任何之一需要被一些组件引入，Yii 将引入这个 URL (一次) 而不是各个独立的脚本文件。

其次，我们需要使用一些工具来联合 (和压缩) JavaScript 文件为一个单独的文件，并保存为 `js/all.js`。

相同的技巧也适用于 CSS 文件。

在 [Google AJAX Libraries API](#) 帮助下我们可以改善页面载入速度。例如，我们可以从 Google 的服务器引入 `jquery.js` 而不是从我们自己的服务器。要这样做，我们首先配置 `scriptMap` 如下，

```

$cs=Yii::app()->clientScript;

$cs->scriptMap=array(

    'jquery.js'=>false,

    'jquery.ajaxqueue.js'=>false,

    'jquery.metadata.js'=>false,

    .....

);

```

通过映射(map)这些脚本文件为 `false`，我们阻止 Yii 产生引入这些文件的代码。作为替代，我们在页面中编写如下代码直接从 Google 引入文件，

```

<head>

<?php echo CGoogleApi::init(); ?>

<?php echo CHtml::script(

    CGoogleApi::load('jquery','1.3.2') . "\n" .

```



```
CGoogleApi::load('jquery.ajaxqueue.js') . "\n" .  
CGoogleApi::load('jquery.metadata.js')  
); ?>  
.....  
</head>
```

常用扩展手册

Srbac 1.1.0.2 使用手册

什么是 **srbac**?

Srbac 是一个为 Yii 框架设计的模块。

它设计用来简化 Yii `authManager` 组件的使用, `authManager` 组件实现基于角色的访问控制(R.B.A.C)。

`srbac` 支持的 `authManager` 是 `CdbAuthManger`, `CdbAuthManger` 使用一个数据库来存储认证信息。 `Srbac` 为大多数 RBAC 动作(创建/编辑/删除 认证项目,为用户分配认证项目等)提供了一个图形界面。

Srbac 1.1.x 需要 Yii 版本 1.1.0 或更新的版本。

下载 **srbac**

Srbac 可以在如下地址下载:

Yii 扩展页面: <http://www.yiiframework.com/extension/srbac/>

Google 项目页面: <http://code.google.com/p/srbac/downloads/list>

也可以使用如下命令签出最新的开发代码:

```
svn checkout http://srbac.googlecode.com/svn/trunk/ srbac-read-only
```

安装 **srbac**

要安装 `srbac` 模块,首先解压压缩文件到 Yii 应用的模块目录,然后编辑配置文件如下:

配置数据库组件:

SQLite:

```
'db'=>array(
    'class'=>'CDbConnection',
    'connectionString'=>'sqlite:path/to/database/yourDatabase.db',
),
```

MySQL:

```
'db'=>array(

    'class'=>'CDbConnection',

    'connectionString'=>'mysql:host=localhost;dbname=yourDatabase',

    'username'=>'yourUsername',

    'password'=>'yourPassword',

),
```

配置 `authManager` 组件:

```
'authManager'=>array(

    'class'=>'CDbAuthManager',// Manager 的类型

    'connectionID'=>'db',//使用的数据库组件

    'itemTable'=>'items',// 授权项目表 (默认:authitem)

    'assignmentTable'=>'assignments',// 授权分配表 (默认:authassignment)

    'itemChildTable'=>'itemchildren',// 授权子项目表 (默认:authitemchild)

),
```

配置 `srbac` 模块:

```
'srbac' => array(

    'userclass'=>'User', //可选,默认是 User

    'userid'=>'user_ID', //可选,默认是 userid

    'username'=>'username', //可选,默认是 username

    'debug'=>true, //可选,默认是 false
```

```
'pageSize'=>10, //可选, 默认是 15

'superUser' =>'Authority', //可选, 默认是 Authorizer

'css'=>'srbac.css', //可选, 默认是 srbac.css

'layout'=>

    'application.views.layouts.main', //可选,默认是

        // application.views.layouts.main, 必须是一个存在的路径别名

'notAuthorizedView'=>

'srbac.views.authitem.unauthorized ', // 可选,默认是

    //srbac.views.authitem.unauthorized, 必须是一个存在的路径别名

'alwaysAllowed'=>array(    //可选,默认是 gui

    'SiteLogin','SiteLogout','SiteIndex','SiteAdmin',

    'SiteError', 'SiteContact'),

'userActions'=>array(//可选,默认是空数组

    'Show','View','List'),

'listBoxNumberOfLines' => 15, //可选,默认是 10

'imagesPath' => 'srbac.images', //可选,默认是 srbac.images

'imagesPack'=>'noia', //可选,默认是 noia

'iconText'=>true, //可选,默认是 false

'header'=>'srbac.views.authitem.header', //可选,默认是

    // srbac.views.authitem.header, 必须是一个存在的路径别名

'footer'=>'srbac.views.authitem.footer', //可选,默认是

    // srbac.views.authitem.footer, 必须是一个存在的路径别名

'showHeader'=>true, //可选,默认是 false

'showFooter'=>true, //可选,默认是 false

'alwaysAllowedPath'=>'srbac.components', //可选,默认是 srbac.components
```

```
// 必须是一个存在的路径别名

)
```

查看 `srbac` 属性列表可以得到每个属性的详细信息。

导入 `SbaseController`(for using the auto checking access feature):

```
'import'=>array(

    'application.modules.srbac.controllers.SBaseController',

),
```

现在访问 `/path/to/application/index.php?r=srbac` 你将转向到安装页面。

一个检查被执行,若一切 OK, 你可以开始安装(也可以选择创建一些演示用的授权项目)。

若 `srbac` 已经被安装, 将提示你覆盖之前的安装(这将丢弃所有的表并删除你当前的授权数据)。

一个 'Authorizer' 角色将被创建。(你可以在 `srbac` 配置中修改它的名字)。这是唯一可以管理 `srbac` 的用户(创建,编辑,删除角色,人物,操作以及为用户分配权限)。

注意, 除非你设置 `srbac debug` 属性为 `false`, 否则任何人都可以管理 `srbac`, 同时,除非你分配 `Authorizer` 角色到至少一个用户,否则任何人都可以管理 `srbac`。

在分配 `Authorizer` 角色到一个用户后,设置 `srbac debug` 属性为 `false` 是明智的。同时你也可以删除或重命名 `srbac/views/authitem/install` 目录。

`srbac` 主要的管理页面是:`path/to/application/index.php?r=srbac/authitem/frontpage`。

自动创建和访问检查

自版本 1.02 开始, 你可以为你的控制器自动创建 操作/任务。 操作的命名格式是:
[ModuleId_][Subdirectory.][ContollerId][Action]。例如:

`posts_PostView` : 模块 `posts` , 控制器 `Post`, 动作 `View`

`posts_admin.PostDelete` : 模块 `posts`, 子目录 `admin`, 控制器 `Post`, 动作 `delete`

SiteIndex : 控制器 **Site** , 动作 **Index**.

你也可以创建 2 个任务, 格式是 `[ControllerId]Viewing`, `[ControllerId]Administrating` (例如 `PostViewing`, `PostAdministrating`).

所有操作被分配到 `administrating` 任务, 通过在 `srbac` 配置中编辑 `userActions` 属性, 你可以选择哪些操作被分配到 `viewing` 任务。

若你也想要 `srbac` 在控制器中自动检查访问权限, 你的控制器应当扩展 `srbac` 模块中的 `SBaseController` 类或其子类。

`SbaseController` 重写 `beforeAction($action)` 方法并检查当前用户是否有权限访问当前的 `controller/action`。

要决定一个用户没有权限访问页面后的要做的事情, 你可以重写 `SBaseController` 中的 `onUnauthorizedAccess` 方法。这样你可以显示一个未授权信息页面, 转向到登录页面等。

国际化

若你想要翻译 `srbac` 文本, 你应当在 `Yii` 配置文件中设置目标语言: `'language'=>'fr'`,

然后创建如下包含翻译信息的文件:

`srbac/messages/fr/srbac.php` (你可以复制 `el_gr/srbac.php` 并翻译其中的信息)。

`srbac/views/install/fr/installText.php` (你应当翻译 `srbac/views/install/installText.php`)

若你想要帮助翻译 `srbac`, 请在论坛里联系我:

<http://www.yiiframework.com/forum/index.php?/user/1089-spyros/>

Srbac 属性列表

属性	类型	描述
<code>\$userid</code>	String	用户的 id 属性。默认是 “userid”。
<code>\$username</code>	String	用户的 name 属性。默认是 “username”。
<code>\$userclass</code>	String	用户类。默认是 “User”。
<code>\$debug</code>	Boolean	是否 <code>srbac</code> 处于调试模式。处于调试模式时, <code>srbac</code> 可以被安装, 每个用户可以管理 <code>srbac</code> 并且 missing translations will be marked with a red star *. 默认是 false.
<code>\$pagesize</code>	integer	在授权项目列表上每页显示的数目。默认是 15.
<code>\$superuser</code>	String	<code>srbac</code> 管理员角色的名字。默认是 Authorizer
<code>\$css</code>	String	使用的 css 文件。Srbac 将首先在默认的应用 css 目录 (webroot.css), 然后在 <code>srbac</code> css 目录

		(application.modules.srbac.css) 中寻找该文件。默认是 srbac.css
\$layout	String	当渲染 srbac 视图时所用的布局。默认是一个空字符串，意味着使用应用的 main 布局。
\$notAuthorizedView	String	为访问未授权页面的用户提供的视图。默认是 srbac.views.authitem.unauthorized
\$alwaysAllowed	mixed	总是被允许访问的 action (例如 SiteIndex, SiteLogin, SiteLogout 等)。可以是数组，一个以逗号分隔的字符串，或者字符串“gui”。若没有设置此项或被设置为“gui”，the srbac/components/allowed.php file generated by the always allowed GUI will be used. 默认是 ‘gui’。
\$userActions	mixed	The operations that are assigned to viewing taskby default (e.g. Show, List, View etc). Can be an array or a coma delimited string . 默认是 array().
\$listBoxNumberOfLines	integer	在分配视图 listbox 中的行数。默认是 10。
\$imagesPath	String	srbac 图像目录的别名。默认是 “application.modules.srbac.images”
\$imagesPack	String	使用的图像包。当前有两个选择 tango 和 noia 。若你使用自己的包, 定义\$imagesPath 并把它放在里面 (例如 myImagesPath/myPack)。
\$iconText	Boolean	是否在 icon 旁显示 icon 文本，默认是 false.
\$header	String	The view to render above srbac GUI
\$footer	String	The view to render below srbac GUI
\$showHeader	Boolean	是否显示 header
\$showFooter	Boolean	是否显示 footer
\$alwaysAllowedPath	String	allowed.php 被保存的路径，默认是 “srbac.components”