Yii framework 中文手册

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(extensible)。

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。

安装步骤

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。你也可以用 php YiiRoot/framework/yiic.php 来代替 yiic。

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：

首页

My Web Application

Home Contact Login

Welcome, Guest!

This is the homepage of *My Web Application*. You may modify the following files to customize the content of this page:

D:\wwwroot\testdrive\protected\controllers\SiteController.php
: This file contains the `SiteController` class which is the default application controller. Its default `index` action renders the content of the following two files.

D:\wwwroot\testdrive\protected\views\site\index.php
: This is the view file that contains the body content of this page.

D:\wwwroot\testdrive\protected\views\layouts\main.php
: This is the layout file that contains common presentation (such as header, footer) shared by all view files.

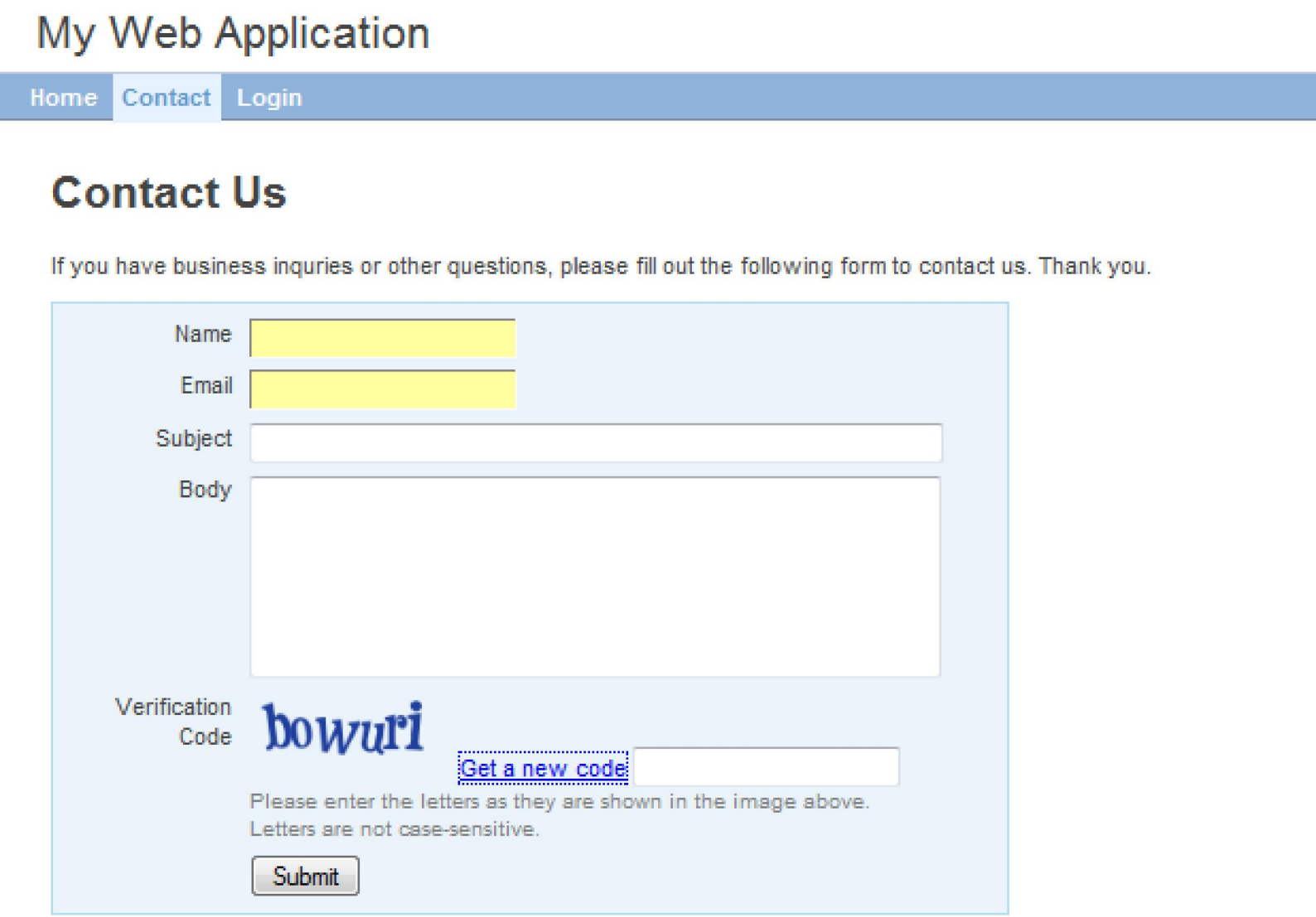
What's Next

- Implement new actions in `SiteController`, and create corresponding views under D:\wwwroot\testdrive\protected\views\site
- Create new controllers and actions manually or using the `yiic` tool.
- If your Web application should be driven by database, do the following:
  - Set up database connection by configuring the `db` component in the application configuration `D:\wwwroot\testdrive\protected\config\main.php`
  - Create model classes under the directory `D:\wwwroot\testdrive\protected\models`
  - Implement CRUD operations for a model class. For example, for the `Post` model class, you would create a `PostController` class together with `create`, `read`, `update` and `delete` actions.

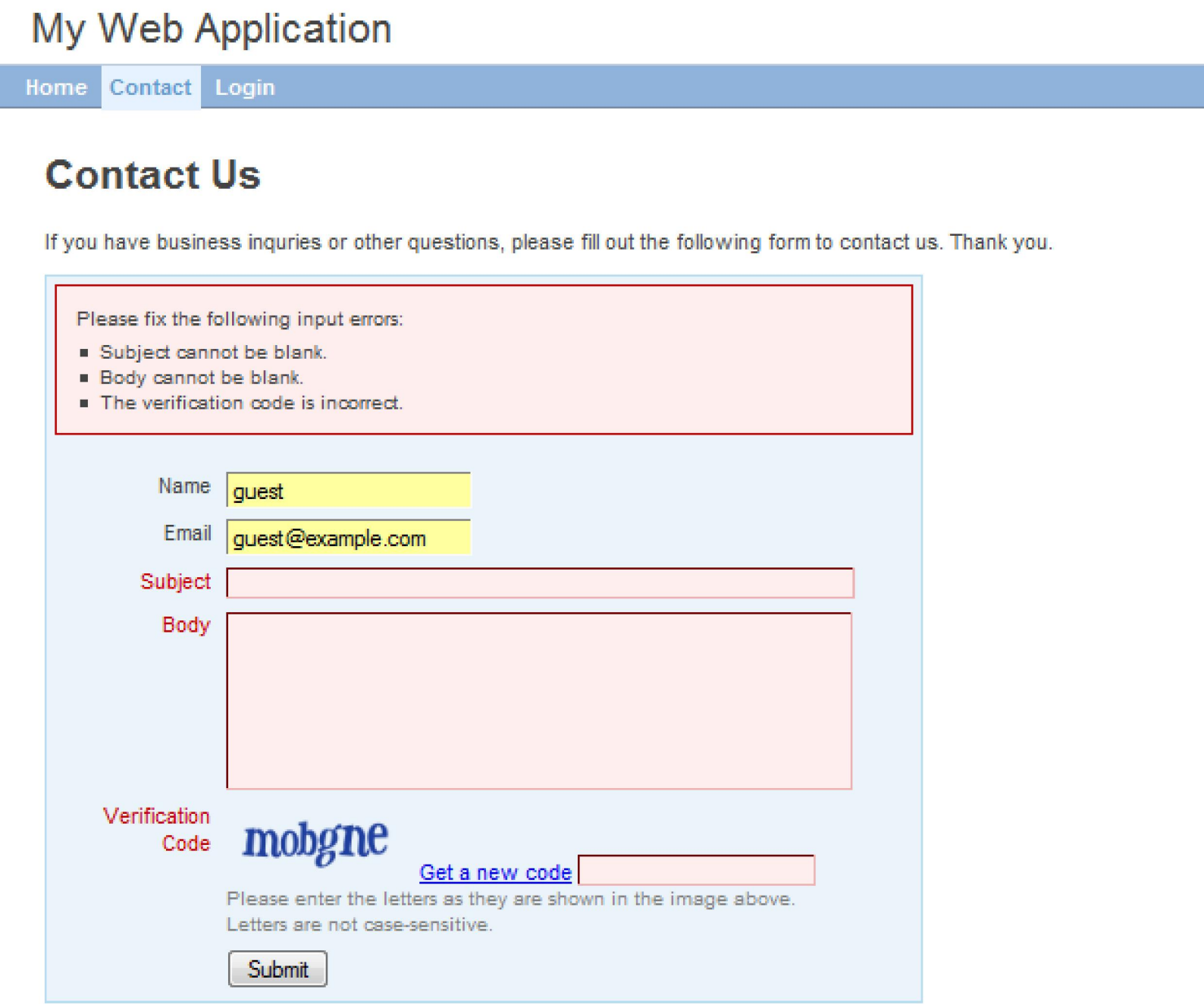
  Note, the `yiic` tool can automate the task of creating model classes and CRUD operations.

If you have problems in accomplishing any of the above tasks, please read Yii documentation or visit Yii forum for help.

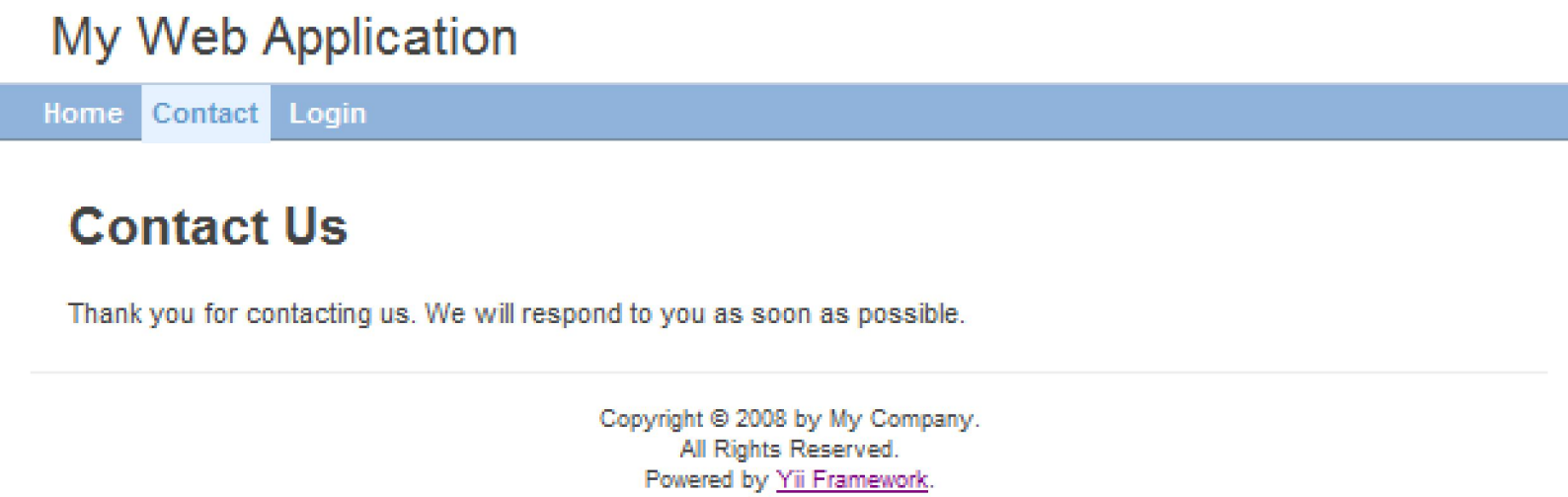
联系页


My Web Application
Home
Contact
Login
Contact Us
If you have business inquries or other questions, please fill out the following form to contact us. Thank you.
Name
Email
Subject
Body
Verification Code
bowuri
Get a new code
Please enter the letters as they are shown in the image above.
Letters are not case-sensitive.
Submit
Copyright © 2008 by My Company.
All Rights Reserved.
Powered by Yii Framework.

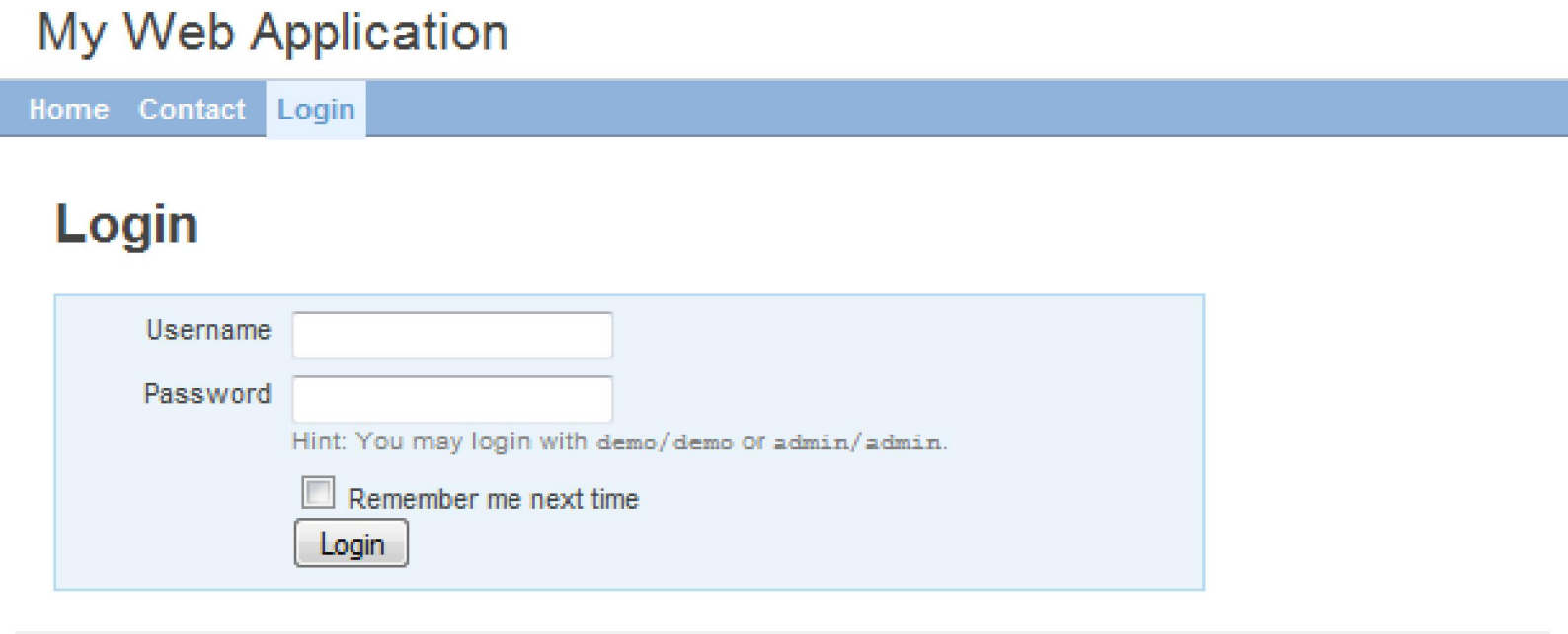
输入错误的联系页


My Web Application
Home
Contact
Login
Contact Us
If you have business inquries or other questions, please fill out the following form to contact us. Thank you.
Please fix the following input errors:
Subject cannot be blank.
Body cannot be blank.
The verification code is incorrect.
Name
guest
Email
guest@example.com
Subject
Body
Verification Code
mobgne
Get a new code
Please enter the letters as they are shown in the image above.
Letters are not case-sensitive.
Submit
Copyright © 2008 by My Company.
All Rights Reserved.
Powered by Yii Framework.

提交成功的联系页



登录页



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
  assets/                包含公开的资源文件
  css/                   包含 CSS 文件
```

```
images/                  包含图片文件
themes/                  包含应用主题
protected/               包含受保护的应用文件
   yiic                  yiic 命令行脚本
   yiic.bat              Windows 下的 yiic 命令行脚本
   commands/             包含自定义的 'yiic' 命令
      shell/             包含自定义的 'yiic shell' 命令
   components/           包含可重用的用户组件
      MainMenu.php       'MainMenu' 挂件类
      Identity.php       用来认证的 'Identity' 类
      views/             包含挂件的视图文件
         mainMenu.php    'MainMenu' 挂件的视图文件
   config/               包含配置文件
      console.php        控制台应用配置
      main.php           Web 应用配置
   controllers/          包含控制器的类文件
      SiteController.php 默认控制器的类文件
   extensions/           包含第三方扩展
   messages/             包含翻译过的消息
   models/               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
   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
      site/              包含 'site' 控制器的视图文件
         contact.php     'contact' 动作的视图
         index.php       'index' 动作的视图
         login.php       'login' 动作的视图
      system/            包含系统视图文件
```

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/source.db',
        ),
    ),
    ......
);
```

在上面的代码中，我们添加了 db 条目至 components 中，指示应用在需要的时候连接到 SQLite 数据库 WebRoot/testdrive/protected/data/source.db。

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。

接下来，我们需要准备一个 SQLite 数据库以使上面的配置生效。使用一些 SQLite 管理工具，我们可以建立一个包含如下模式的数据库：

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

简单起见，我们只在库中建立了一个 User 表。SQLite 数据库文件保存在 WebRoot/testdrive/protected/data/source.db。注意，数据库文件和包含它的目录都要求 Web 服务器进程可写。

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。

我们还是用 yiic 工具来帮助我们生成需要的代码，这个过程通常称为“脚手架”。

```
% cd WebRoot/testdrive
% YiiRoot/framework/yiic shell
Yii Interactive Tool v1.0
Please type 'help' for help. Type 'exit' to quit.
>> model User
   generate User.php

The 'User' class has been successfully created in the following file:
    D:\wwwroot\testdrive\protected\models\User.php

If you have a 'db' database connection, you can test it now with:
```

```
    $model=User::model()->find();
    print_r($model);

>> crud User
   generate UserController.php
   generate create.php
      mkdir D:/wwwroot/testdrive/protected/views/user
   generate update.php
   generate list.php
   generate show.php

Crud 'user' has been successfully created. You may access it via:
http://hostname/path/to/index.php?r=user
```

如上所示，我们使用 yiic 的 shell 命令来和我们刚才建立的应用进行交互。在提示符后面，我们可以输入一个有效的 PHP 语句或表达式来运行并显示。我们还可以完成一些诸如 model 或 crud 之类的任务。model 命令自动生成一个基于 User 表结构的 User 模型类，crud 命令生成实现 User 模型 CRUD 操作的控制器类和视图。

注意: 如果你更改了你的任何代码或配置，请重新输入 yiic shell 以使你的新代码或配置文件生效。还有，确保你使用了正确的 PHP CLI 来运行 yiic，否则你会碰到 "...could not find driver" 之类的错误（即使你确信已经启用了 PDO 和相应的驱动）。这类错误通常是因为 PHP CLI 使用了不恰当的 php.ini。

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 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 新增用户 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 建立 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 Contact Logout

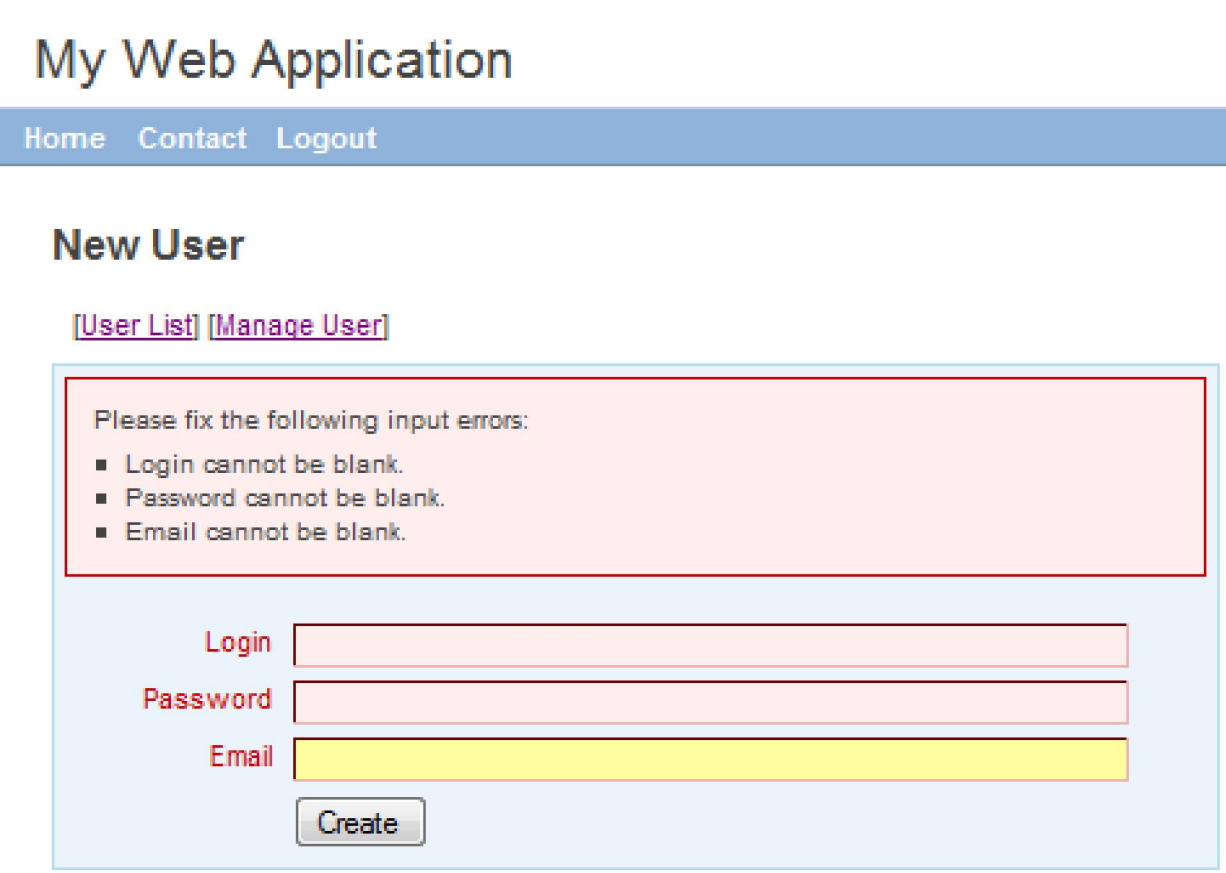
Managing User

[User List] [New User]

Id	Login	Password	Email	Actions
1	test1	pass1	test1@example.com	Update Delete
2	test2	pass2	test2@example.com	Update Delete
3	test3	pass3	test3@example.com	Update Delete
4	test4	pass4	test4@example.com	Update Delete
5	test5	pass5	test5@example.com	Update Delete
6	test6	pass6	test6@example.com	Update Delete
7	test7	pass7	test7@example.com	Update Delete
8	test8	pass8	test8@example.com	Update Delete
9	test9	pass9	test9@example.com	Update Delete
10	test10	pass10	test10@example.com	Update Delete

Go to page: < Previous 1 2 3 Next >

新增用户页



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。

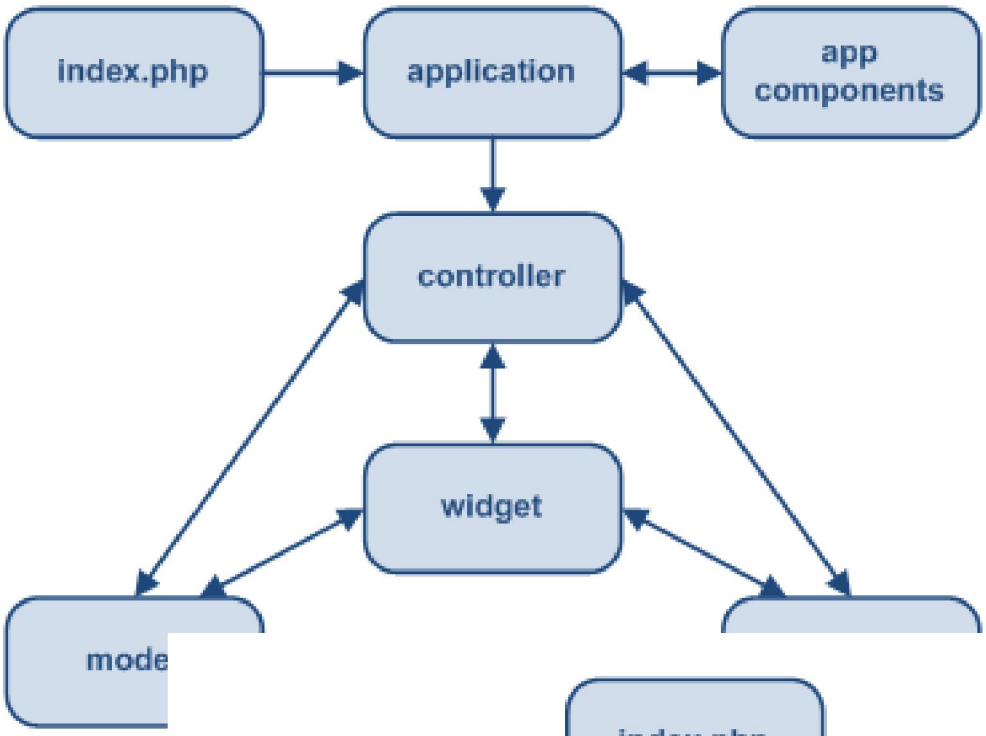
除了 MVC，Yii 还引入了一个叫做 application 的前端控制器，它表现整个请求过程的运行环境。Application 接收用户的请求并把它分发到合适的控制器作进一步处理。

下图描述了一个 Yii 应用的静态结构：

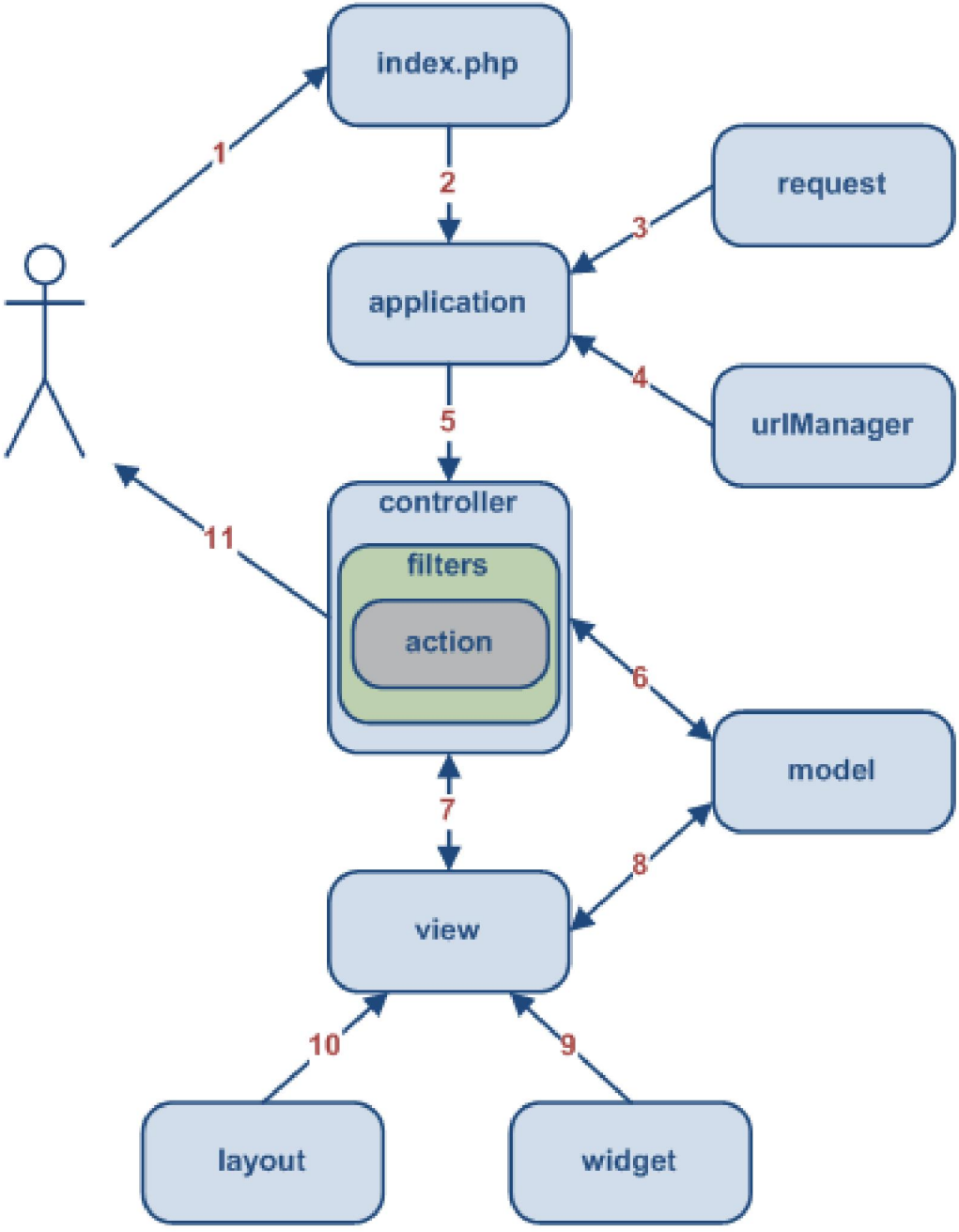
Yii 应用的静态结构

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. 视图运行一些挂件。
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”。

入口脚本将应用创建为一个单例. 应用单例可以在任何位置通过 Yii::app() 来访问.

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(e.g. 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。或者象下述例子这样传递给 Yii::createWebApplication() 我们一般在 entry script 里界定这些配置:

```
$app=Yii::createWebApplication($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
```

应用元件

我们可以很容易的通过元件(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元件，每个元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 部件来解析用户的访问。

通过配置 components property of application, 我们可以个性化一些元件的类及其参数。比如说，我们可以配置 CMemCache 元件以便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(缓存)cache 元素(element)加在 元件 数组里. 这个 缓存（cache） 告诉我们这个元件的类是 CMemCache 其服务 服务器(servers) 属性应该这样初始化。

要调用这个元件，用这个命令：Yii::app()->ComponentID，其中 ComponentID 是指这个元件的 ID。 (比如 Yii::app()->cache).

我们可以在应用配置里，将 enabled 改为 false 来关闭一个元件。当我们访问一个被禁止的元件时，系统会返回一个 NULL 值。

提示: 默认情况下，应用元件是根据需要而创建的。这意味着一个元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元件而下降。有些应用元件，(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元件的 ID 列上：preload。

应用的核心元件

Yii 预先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 元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，通过配置这些核心元件的属性，我们都可以更改 Yii 的默认行为。

下面我们列出 CWebApplication 预先声明的核心元件。

- assetManager: CAssetManager -管理发布私有 asset 文件。
- authManager: CAuthManager - 管理基于角色控制 (RBAC)。
- cache: CCache - 提供数据缓存功能。请注意，您必须指定实际的类（例如 CMemCache, CDbCache ） 。否则，将返回空当访问此元件。

- clientScript: CClientScript -管理客户端脚本(javascripts and CSS)。
- coreMessages: CPhpMessageSource -提供翻译 Yii 框架使用的核心消息。
- db: CDbConnection - 提供数据库连接。请注意，你必须配置它的 connectionString 属性才能使用此元件。
- errorHandler: CErrorHandler - 处理没有捕获的 PHP 错误和例外。
- 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. 建立类自动加载器和错误处理；
2. 注册核心应用组件；
3. 读取应用配置；
4. 用 CApplication::init()初始化应用程序。
    - 读取静态应用组件;
5. 触发 onBeginRequest 事件;
6. 处理用户请求：
    - 解析用户请求；
    - 创建控制器；
    - 执行控制器；

7.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(Route)

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 前辍构成 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。

Users request for a particular controller and action in terms of route. A route is formed by concatenating a controller ID and an action ID separated by a slash. For example, the route post/edit refers to PostController and its edit action. And by default, the URL http://hostname/index.php?r=post/edit would request for this controller and action.

注意: By default, routes are case-sensitive. Since version 1.0.1, it is possible to make routes case-insensitive by setting CUrlManager::caseSensitive to be false in the application configuration. When in case-insensitive mode, make sure you follow the convention that directories containing controller class files are in lower case, and both controller map and action map are using keys in lower case.

Since version 1.0.3, an application can contain modules. The route for a controller action inside a module is in the format of moduleID/controllerID/actionID. For more details, see the 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

- If CWebApplication::catchAllRequest is specified, a controller will be created based on this property, and the user-specified controller ID will be ignored. This is mainly used to put the application under maintenance mode and display a static notice page.
- If the ID is found in CWebApplication::controllerMap, the corresponding controller configuration will be used to create the controller instance.

- If the ID is in the format of 'path/to/xyz', the controller class name is assumed to be XyzController and the corresponding class file is protected/controllers/path/to/XyzController.php. For example, a controller ID admin/user would be resolved as the controller class UserController and the class file protected/controllers/admin/UserController.php. If the class file does not exist, a 404 CHttpException will be raised.

In case when modules are used (available since version 1.0.3), the above process is slighly different. In particular, the application will check if the ID refers to a controller inside a module, and if so, the module instance will be created first followed by the controller instance.

Action

As aforementioned, an action can be defined as a method whose name starts with the word action. A more advanced way is to define an action class and ask the controller to instantiate it when requested. This allows actions to be reused and thus introduces more reusability.

To define a new action class, do the following:

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

In order for the controller to be aware of this action, we override the actions() method of our controller class:

```
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.

Writing class-based actions, we can organize an application in a modular fashion 以模块方式组织程序。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 is a piece of code that is configured to be executed before and/or after a controller action executes. For example, an access control filter may be executed to ensure that the user is authenticated before executing the requested action; a performance filter may be used to measure the time spent in the action execution.

An action can have multiple filters. The filters are executed in the order that they appear in the filter list. A filter can prevent the execution of the action and the rest of the unexecuted filters.

A filter can be defined as a controller class method. The method name must begin with `filter`. For example, the existence of the `filterAccessControl` method defines a filter named `accessControl`. The filter method must be of the signature:

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

where `$filterChain` is an instance of CFilterChain which represents the filter list associated with the requested action. Inside the filter method, we can call `$filterChain->run()` to continue filtering and action execution.

A filter can also be an instance of CFilter or its child class. The following code defines a new filter class:

```
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
    }
}
```

To apply filters to actions, we need to override the CController::filters() method. The method should return an array of filter configurations. For example,

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

The above code specifies two filters: postOnly and PerformanceFilter. The postOnly filter is method-based (the corresponding filter method is defined in CController already); while the PerformanceFilter filter is object-based. The path alias application.filters.PerformanceFilter specifies that the filter class file is protected/filters/PerformanceFilter. We use an array to configure PerformanceFilter so that it may be used to initialize the property values of the filter object. Here the unit property of PerformanceFilter will be initialized as 'second'.

Using the plus and the minus operators, we can specify which actions the filter should and should not be applied to. In the above, the postOnly should be applied to the edit and create actions, while PerformanceFilter should be applied to all actions EXCEPT edit and create. If neither plus nor minus appears in the filter configuration, the filter will be applied to all actions.

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 Working with Form

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

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.

视图有一个当其被渲染(render)时用于用于校验的名称.视图的名称与其脚本名称是一样的.例如:视图 edit 的名称出自一个名为 edit.php 的脚本文件.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.其产生的结果是,在视图脚本里,我们可以直接访问变量 $var1 和 $var2.

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,布局是固定的被提供的.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 CWebApplication::layout 来实现定制 . 通过调用 renderPartial() 可以不依赖布局而渲染视图.

组件

组件是 CWidget 或其子类都实例.它是一个主要用于描述意图的组成部分.组件通常内嵌于一个视图来产生一些复杂却独立的用户界面.例如,一个日历组件可以用于渲染一个复杂的日历界面.组件可以在用户界面上更好的实现重用.

我们可以按如下视图脚本来使用一个组件:

```
<?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 内容的组件.

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

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

继承 CWidget 以及重载它的 init() 和 run() 方法,可以定义一个新的组件:

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

    public function run()
```

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

组件可以像一个控制器一样拥有它自己都视图.默认的,组件的视图文件位于包含了组件文件的 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,组件的视图没有布局文件支持.

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.

部件

YII 应用构建于对象是规范编写的部件之上.部件是 CComponent 或其衍生类的实例.使用部件主要就是涉及访问其属性和挂起/处理它的事件.基类 CComponent 指定了如何定义属性和事件.

部件属性

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
```

我们可以简单的在组件类里公开声明一个公共成员变量来定义组件属性.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.当读取属性时,getTextWidth()被请求,然后它它的返回值编程了属性的值;同样的,当写入属性石,setTextWidth() 被请求.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,挂起事件)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.

部件事件

部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.绑定(分配)一个方法到一个事件将会导致方法在事件被挂起处自动被调用.因此部件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.

部件事件以 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.以下方法定义了一个 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.

我们可以按照下述为这个事件绑定一个方法:

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

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

事件句柄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 或其子类的实例.它至少包含了"是谁挂起了这个事件"的信息.

如果我们现在调用了 onClicked(), onClicked 事件将被挂起(内置的 onClicked()),然后被绑定的事件句柄将被自动调用.

一个事件可以绑定多个句柄.当事件被挂起时,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.

部件行为

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() 方法绑定到一个部件是必须的.然后我们就可以通过部件调用行为了:

```
// $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部件中失效.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部件下是很有可能的.在这种情况下,先绑定的行为则拥有优先权.

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

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(包括根目录别名).

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;
- 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.
- module: 指向当前运行程序模型的目录. 此别名自 1.0.3 版起生效.

通过使用 YiiBase::getPathOfAlias(), 别名可以被转换成他的真实路径. 例如, system.web.CController 可以被转换成 yii/framework/web/CController.

使用别名来插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 核心类都是被预定义的.

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() 创建的一个对应类的实例, 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名命名空间,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.

Conventions（惯例）

Yii主张配置实现惯例。按照惯例，某人可以创建复杂的Yii应用，无需编写和维护复杂的配置。当然，在需要时几乎每一个方面，Yii 仍然可以进行自定义配置。

下面我们描述建议 Yii 开发的惯例。 为了方便起见，我们假设 WebRoot 是 Yii 应用安装目录。

网址（URL）

默认情况下，Yii 确认的网址，格式如下：

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

r GET 变量指 route ，可以通过 Yii 解析为控制器和动作。如果 ActionID 省略，控制器将采用默认的动作(通过 CController::defaultAction 指定);如果 ControllerID 也省略（或 r 变量也没有） ，该应用程序将使用默认的控制器（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（Code）

Yii 建议变量，函数和类类型使用骆驼方式命名，就是不用空格连接每个名字的单词。变量和函数名首字母小写，为了区分于类名称（如：$basePath ，runController（），LinkPager）。私有类成员变量，建议将他们的名字前缀加下划线字符（例如：$ _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，以避免和第三方名称冲突。出于这个原因，所有 Yii 框架类开头的字母"C"。

控制器类名的特别规则是，他们必须附上 Controller 后缀 。控制器的 ID，然后定义为类名称首字母小写和 Controller 结尾。 例如，PageController 类将有 ID page 。这条规则使得应用更加安全。这也使得相关的网址控制器更加简洁（例如/index.php?r=page/index 替代/index.php?r=PageController/index） 。

配置（Configuration）

配置是数组关键值对。每个键代表对象名称属性的配置，每个值相应属性的初始值。举个例子，array('name'=>'My application', 'basePath'=>'./protected')初始 name 和 basePath 属性为其相应的数组值。

一个对象任何写入属性可以配置。如果没有配置，属性将使用它们的默认值。当设定属性，应该阅读相应的文件，以便使初始值设定正确。

文件（File）

文件命名和使用惯例取决于其类型。

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。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（类只能用于一个单一的公共类）可能和公有类存放在同一个文件。

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 脚本的唯一宗旨就是要返回一个关联数组代表配置。

目录（Directory）

Yii 假定默认设置的目录用于各种目的。如果需要的话，他们每个可自定义。

- WebRoot/protected: 这是 application base directory 包括所有安全敏感的 PHP 脚本和数据文件。Yii 有一个默认的别名为 application 代表此路径。这个目录和下面的一切文件目录，将得到保护不被网络用户访问。它可通过 CWebApplication::basePath 自定义。
- WebRoot/protected/runtime: 此目录拥有应用程序在运行时生成的私有临时文件。这个目录必须可被 Web 服务器进程写。它可通过 CApplication::runtimePath 定制。
- WebRoot/protected/extensions: 此目录拥有所有第三方扩展。它可通过 CApplication::extensionPath 定制。
- WebRoot/protected/modules: 此目录拥有所有应用 modules，每个代表作为一个子目录。
- WebRoot/protected/controllers: 此目录拥有所有控制器类文件。它可通过 CWebApplication::controllerPath 定制。
- WebRoot/protected/views:此目录包括所有的视图文件，包括控制视图，布局视图和系统视图。可通过 CWebApplication::viewPath 定制。
- WebRoot/protected/views/ControllerID: 此目录包括某个控制类的视图文件。这里 ControllerID 代表控制类的 ID。可通过 CController::getViewPath 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例外的模板。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，可能被发布来被 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。

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 部件(例如：用户部件)
3. 每种类型的数据都创建一个 model 类来管理。 同样，yiic 可以为我们需要的数据库表自动生成 active record 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

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,用以验证用户输入,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,作为已定义的 Model 的一部分,是用来保存,校验用户输入的核心.

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 模型;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界面的基类 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"的叫法,我们使用 *特性* 一词来区分于普通属性.特性是一种 主要用于储存用户输入数据或数据库读取数据的属性.

声明有效的规则

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;

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

```
    public function authenticate($attribute,$params)
    {
        if(!$this->hasErrors())  // 我们只想校验没有输入错误
        {
            $identity=new UserIdentity($this->username,$this->password);
            if($identity->authenticate())
            {
                $duration=$this->rememberMe ? 3600*24*30 : 0; // 30 天
                Yii::app()->user->login($identity,$duration);
            }
            else
                $this->addError('password','Incorrect password.');
        }
    }
}
```

以上的代码中 username 和 password 都是必填的,password 将被校验.

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 . 校验方法必须是以下结构 :

```
/**
 * @param string 用于校验特性
 * @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.当规则生效时,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:

- 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.
- 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，使用一个过滤器转换特性的形式.
- 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

注意: 自 1.0.2 版起,基于场景的特性分配开始生效.

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归位它的特性.这将大大简化以下繁重的任务:

```
$model=new LoginForm;
if(isset($_POST['LoginForm']))
```

```
$model->setAttributes($_POST['LoginForm'], 'login');
```

以上的是一个繁重的任务,它在 login 场景(第二给参数指定的)中为每个 $_POST['LoginForm'] 数据项分配对应的模型特性.而它和以下的代码效果是一样的:

```
foreach($_POST['LoginForm'] as $name=>$value)
{
    if($name is a safe attribute)
        $model->$name=$value;
}
```

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分配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.

当我们调用 CModel::validate() 方法, 我们可以指定一个场景参数.只有在特定的场景下校验规则才会生效.校验规则会在那些 on 选项没有被设置或者包含了指定的场景名称的场景中生效.如果我们没有指定场景,而调用了 CModel::validate() 方法,只有那些 on 选项没有设置的规则才会被执行.

例如,在注册一个用户时,我们运行以下脚本来执行校验 :

```
$model->validate('register');
```

我们可以按以下在表单模型里声明校验规则:

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

结果是,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.

注意: 自 1.0.2 版起,基于场景的校验开始生效.

检索校验错误

我们可以使用 CModel::hasErrors() 来检查是否有校验错误,如果是,我们可以使用 CModel::getErrors() 来获取错误信息. 上述两者中的任何一个方法都可以用于所有特性或者单独的一个特性.

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 会默认的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.

编写 ACTION

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()
{
    $form=new LoginForm;
    if(isset($_POST['LoginForm']))
    {
        // 收集用户输入的数据
        $form->attributes=$_POST['LoginForm'];
        // 验证用户输入，如果无效则重定位到前个页面
        if($form->validate())
            $this->redirect(Yii::app()->user->returnUrl);
    }
    // 显示登陆表单
    $this->render('login',array('user'=>$form));
}
```

上面写的是，我们编写 LoginForm 实例；如果请求是 POST 方式（意味着登陆表单是 submit），我们产生一个 $form，里面放着提交过来的数据 $_POST['LoginForm'];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。表达式 Yii::app()->user 返回一个 CWebUser 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：

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

真如我们在 Securing Attribute Assignments 提到，这句话只是创建一个 model 存放用户提交来的数据。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）：

```
$form->username=$_POST['LoginForm']['username'];
$form->password=$_POST['LoginForm']['password'];
$form->rememberMe=$_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 view 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)类来简化视图组合.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 $user 代表 LoginForm 的实例.

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

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

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

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

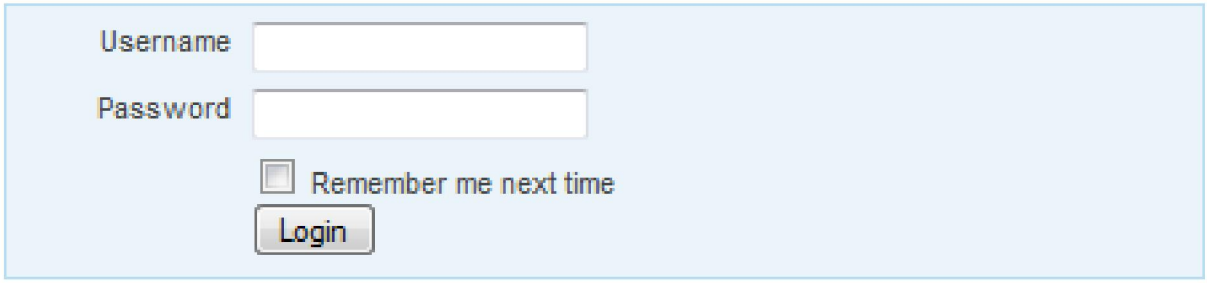
```
<div class="action">
<?php echo CHtml::activeCheckBox($user,'rememberMe'); ?>
记住我?<br/>
<?php echo CHtml::submitButton('Login'); ?>
</div>

</form>
</div><!-- yii表单 -->
```

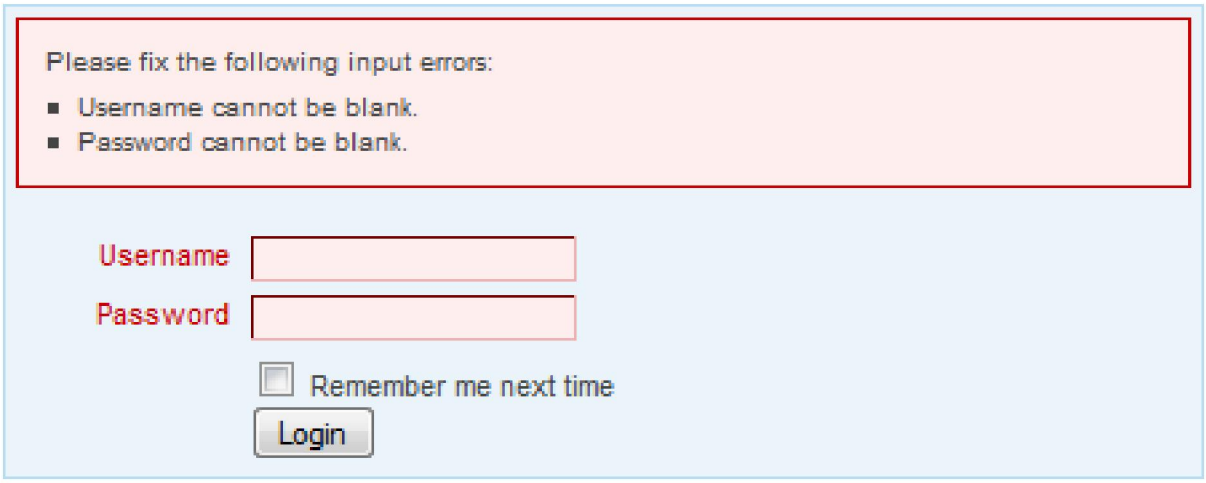
以上代码生成了一个更动态的表单. 例如 CHtml::activeLabel() 生成了一个关联到指定模型特性的标签.如果这个特性有一个输入错误,标签 CSS 的 class 将变成改变标签视觉表现到相应 CSS 样式的 error.类似的,[CHtml::activeTextField() 为指定的模型特性生成了一个文本输入框,其 CSS 的 class 也会在发生任何错误时变成 error.

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



登陆出错页面



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,我们首先需要使用模型实例创建或者填充一个数组,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::form(); ?>
<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($item,"name[$i]"); ?></td>
<td><?php echo CHtml::activeTextField($item,"price[$i]"); ?></td>
<td><?php echo CHtml::activeTextField($item,"count[$i]"); ?></td>
<td><?php echo CHtml::activeTextArea($item,"description[$i]"); ?></td>
</tr>
<?php endforeach; ?>
</table>

<?php echo CHtml::submitButton('Save'); ?>
</form>
</div><!-- yii 表单 -->
```

注意： 在上述代码中,我们使用了 "name[$i]" 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.

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，使得在一个单一的统一的接口可以访问不同的数据库管理系统(DBMS)。使用 Yii 的 DAO 开发的应用程序可以很容易地切换使用不同的数据库管理系统，而不需要修改数据访问代码。Yii 的 Active Record（ AR ），实现了被广泛采用的对象关系映射(ORM)办法，进一步简化数据库编程。按照约定，一个类代表一个表，一个实例代表一行数据。Yii AR 消除了大部分用于处理 CRUD（创建，读取，更新和删除）数据操作的 sql 语句的重复任务。

尽管 Yii 的 DAO 和 AR 能够处理几乎所有数据库相关的任务，您仍然可以在 Yii application 中使用自己的数据库库。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。

Data Access Objects (DAO)

Data Access Objects (DAO) provides a generic API to access data stored in different database management systems (DBMS). As a result, the underlying DBMS can be changed to a different one without requiring change of the code which uses DAO to access the data.

数据访问对象（DAO）针对不同的数据库管理系统提供一个通用的的数据访问 API。因此，使用 DAO 将数据访问移植到其他数据库时，几乎不需要做改动。

Yii DAO is built on top of PHP Data Objects (PDO) which is an extension providing unified data access to many popular DBMS, such as MySQL, PostgreSQL. Therefore, to use Yii DAO, the PDO extension and the specific PDO database driver (e.g. PDO_MYSQL) have to be installed.

Yii DAO mainly consists of the following four classes:

- CDbConnection: represents a connection to a database. 声明数据库连接
- CDbCommand: represents an SQL statement to execute against a database. 声明一个对数据库执行的 SQL 语句
- CDbDataReader: represents a forward-only stream of rows from a query result set. 声明一个从查询结果集中的前瞻性行数据流
- CDbTransaction: represents a DB transaction. 表示一个 DB 事务

In the following, we introduce the usage of Yii DAO in different scenarios.

如下所述，我们将用不同的章节来介绍如何使用 Yii DAO

Establishing Database Connection

To establish a database connection, create a CDbConnection instance and activate it. A data source name (DSN) is needed to specify the information required to connect to the database. A username and password may also be needed to establish the connection. An exception will be raised in case an error occurs during establishing the connection (e.g. bad DSN or invalid username/password).

为了建立一个数据库连接，创建一个 CDbConnection 实体并激活它。需要申明一个连接到数据库的数据源名（DSN）。需要建立连接的用户名和密码。在建立连接过程中的需要申明一个错误异常（例如，错误的数据源或用户名/密码无效）

```
$connection=new CDbConnection($dsn,$username,$password);
// establish connection. You may try...catch possible exceptions
$connection->active=true;
......
$connection->active=false;  // close connection
```

The format of DSN depends on the PDO database driver in use. In general, a DSN consists of the PDO driver name, followed by a colon, followed by the driver-specific connection syntax. See PDO documentation for complete information. Below is a list of commonly used DSN formats:

数据源 DSN 的格式依赖 PDO 数据库驱动建立。一般来说，

- 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`

Because CDbConnection extends from CApplicationComponent, we can also use it as an application component. To do so, configure in a `db` (or other name) application component in the application configuration as follows,

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

We can then access the DB connection via `Yii::app()->db` which is already activated automatically, unless we explictly configure CDbConnection::autoConnect to be false. Using this approach, the single DB connection can be shared in multiple places in our code.

Executing SQL Statements

Albert 2010 年 4 月 26 日 0:37:57

Once a database connection is established, SQL statements can be executed using CDbCommand. One creates a CDbCommand instance by calling CDbConnection::createCommand() with the specified SQL statement:

```
$command=$connection->createCommand($sql);
// if needed, the SQL statement may be updated as follows:
// $command->text=$newSQL;
```

A SQL statement is executed via CDbCommand in one of the following two ways:

- execute(): performs a non-query SQL statement, such as `INSERT`, `UPDATE` and `DELETE`. If successful, it returns the number of rows that are affected by the execution.

- query(): performs an SQL statement that returns rows of data, such as SELECT. If successful, it returns a CDbDataReader instance from which one can traverse the resulting rows of data. For convenience, a set of queryXXX() methods are also implemented which directly return the query results.

An exception will be raised if an error occurs during the execution of SQL statements.

```
$rowCount=$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
```

Fetching Query Results

After CDbCommand::query() generates the CDbDataReader instance, one can retrieve rows of resulting data by calling CDbDataReader::read() repeatedly. One can also use CDbDataReader in PHP's foreach language construct to retrieve row by row.

```
$dataReader=$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();
```

Note: Unlike query(), all queryXXX() methods return data directly. For example, queryRow() returns an array representing the first row of the querying result.

Using Transactions

When an application executes a few queries, each reading and/or writing information in the database, it is important to be sure that the database is not left with only some of the queries carried out. A transaction, represented as a CDbTransaction instance in Yii, may be initiated in this case:

- Begin the transaction.
- Execute queries one by one. Any updates to the database are not visible to the outside world.
- Commit the transaction. Updates become visible if the transaction is successful.
- If one of the queries fails, the entire transaction is rolled back.

The above workflow can be implemented using the following code:

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

```php
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();
}
```

Binding Parameters

To avoid SQL injection attacks and to improve performance of executing repeatedly used SQL statements, one can "prepare" an SQL statement with optional parameter placeholders that are to be replaced with the actual parameters during the parameter binding process.

为了防止 SQL 注入攻击和迅速提升 SQL 语句的执行性能

The parameter placeholders can be either named (represented as unique tokens) or unnamed (represented as question marks). Call CDbCommand::bindParam() or CDbCommand::bindValue() to replace these placeholders with the actual parameters. The parameters do not need to be quoted: the underlying database driver does it for you. Parameter binding must be done before the SQL statement is executed.

```php
// an SQL with two placeholders ":username" and ":email"
$sql="INSERT INTO users(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();
```

The methods bindParam() and bindValue() are very similar. The only difference is that the former binds a parameter with a PHP variable reference while the latter with a value. For parameters that represent large block of data memory, the former is preferred for performance consideration.

For more details about binding parameters, see the relevant PHP documentation.

Binding Columns

When fetching query results, one can also bind columns with PHP variables so that they are automatically populated with the latest data each time a row is fetched.

```
$sql="SELECT username, email FROM users";
$dataReader=$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
}
```

Using Table Prefix

Starting from version 1.1.0, Yii provides integrated support for using table prefix. Table prefix means a string that is prepended to the names of the tables in the currently connected database. It is mostly used in a shared hosting environment where multiple applications share a single database and use different table prefixes to differentiate from each other. For example, one application could use `tbl_` as prefix while the other `yii_`.

从 V1.1.0 开始，Yii 集成支持使用表前缀。表前缀意味在当前连接数据中设置表名。它被经常用于多个应用中共享一个数据库，然后通过使用不同的表前缀来区分他们。例如，一个当其他应用使用 Yii_前缀时，另外一个使用 tbl_作为前缀。

To use table prefix, configure the CDbConnection::tablePrefix property to be the desired table prefix. Then, in SQL statements use `{{TableName}}` to refer to table names, where `TableName` means the table name without prefix. For example, if the database contains a table named `tbl_users` where `tbl_` is configured as the table prefix, then we can use the following code to query about users:

为使用表前缀，配置 CDbConnection::tablePrefix 属性使其。然后，在 SQL 语句中使用 `{{TableName}}` 用于指向不同的表名。当 `TableName` 意味着不使用表前缀。例如，如果一个数据库中当表名为 tbl_name 的表，tbl_被配置为表前缀的时候，然后我们使用如下代码来查询相关用户：

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

Active Record

Although Yii DAO can handle virtually any database-related task, chances are that we would spend 90% of our time in writing some SQL statements which perform the common CRUD (create, read, update and delete) operations. It is also difficult to maintain our code when they are mixed with SQL statements. To solve these problems, we can use Active Record.

通过 Yii DAO 能控制需求虚拟任何一个数据库关系任务

Active Record (AR) is a popular Object-Relational Mapping (ORM) technique. Each AR class represents a database table (or view) whose attributes are represented as the AR class properties, and an AR instance represents a row in that table. Common CRUD operations are implemented as AR methods. As a result, we can access our data in a more object-oriented way. For example, we can use the following code to insert a new row to the Post table:

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

In the following we describe how to set up AR and use it to perform CRUD operations. We will show how to use AR to deal with database relationships in the next section. For simplicity, we use the following database table for our examples in this section. Note that if you are using MySQL database, you should replace AUTOINCREMENT with AUTO_INCREMENT in the following SQL.

```
CREATE TABLE Post (
    id INTEGER NOT NULL PRIMARY KEY AUTOINCREMENT,
    title VARCHAR(128) NOT NULL,
    content TEXT NOT NULL,
    createTime INTEGER NOT NULL
);
```

Note: AR is not meant to solve all database-related tasks. It is best used for modeling database tables in PHP constructs and performing queries that do not involve complex SQLs. Yii DAO should be used for those complex scenarios.

Establishing DB Connection

AR relies on a DB connection to perform DB-related operations. By default, it assumes that the db application component gives the needed CDbConnection instance which serves as the DB connection. The following application configuration shows an example:

```
return array(
    'components'=>array(
        'db'=>array(
            'class'=>'system.db.CDbConnection',
            'connectionString'=>'sqlite:path/to/dbfile',
            // turn on schema caching to improve performance
            // 'schemaCachingDuration'=>3600,
        ),
    ),
```

```
);
```

Tip: Because Active Record relies on the metadata about tables to determine the column information, it takes time to read the metadata and analyze it. If the schema of your database is less likely to be changed, you should turn on schema caching by configuring the CDbConnection::schemaCachingDuration property to be a value greater than 0.

Support for AR is limited by DBMS. Currently, only the following DBMS are supported:

- MySQL 4.1 or later
- PostgreSQL 7.3 or later
- SQLite 2 and 3
- Microsoft SQL Server 2000 or later
- Oracle

Note: The support for Microsoft SQL Server has been available since version 1.0.4; And the support for Oracle has been available since version 1.0.5.

If you want to use an application component other than db, or if you want to work with multiple databases using AR, you should override CActiveRecord::getDbConnection(). The CActiveRecord class is the base class for all AR classes.

如果你想使用一个数据库应用程序组件，或者你想用 AR 来支持多个数据库，你应该替换 CActiveRecord::getDbConnection()。CActiveRecord 类是一个对所有 AR 类的基础类。

Tip: There are two ways to work with multiple databases in AR. If the schemas of the databases are different, you may create different base AR classes with different implementation of getDbConnection(). Otherwise, dynamically changing the static variable CActiveRecord::db is a better idea.

技巧：使用 AR 可以有两种方法工作与多数据库上。数据库中的 schemas 是不同的，你可以用不同的 f getDbConnection()执行类创建不同的 AR 类。否则，动态改变 CActiveRecord::db 静态变量也是一个好主意。

Defining AR Class

To access a database table, we first need to define an AR class by extending CActiveRecord. Each AR class represents a single database table, and an AR instance represents a row in that table. The following example shows the minimal code needed for the AR class representing the Post table.

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

Tip: Because AR classes are often referenced in many places, we can import the whole directory containing the AR class, instead of including them one by one. For example, if all our AR class files are under `protected/models`, we can configure the application as follows:

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

By default, the name of the AR class is the same as the database table name. Override the tableName() method if they are different. The model() method is declared as such for every AR class (to be explained shortly).

默认，AR 类的名称与数据库表明一致。如果他们不同需要改写 tableName()方法。

Info: To use the table prefix feature introduced in version 1.1.0, the tableName() method for an AR class may be overridden as follows,

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

That is, instead of returning the fully qualified table name, we return the table name without the prefix and enclose it in double curly brackets.

Column values of a table row can be accessed as properties of the corresponding AR class instance. For example, the following code sets the `title` column (attribute):

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

Although we never explicitly declare the `title` property in the `Post` class, we can still access it in the above code. This is because `title` is a column in the `Post` table, and CActiveRecord makes it accessible as a property with the help of the PHP `__get()` magic method. An exception will be thrown if we attempt to access a non-existing column in the same way.

Info: In this guide, we name columns using camel cases (e.g. `createTime`). This is because columns are accessed in the way as normal object properties which also uses camel-case naming. While using camel case does make our PHP code look more consistent in naming, it may introduce case-sensitivity problem for some DBMS. For example, PostgreSQL treats column names as case-insensitive by default, and we must quote a column in a query condition if the column contains mixed-case letters. For this reason, it may be wise to name columns (and also tables) only in lower-case letters (e.g. `create_time`) to avoid any potential case-sensitivity issues.

Creating Record

To insert a new row into a database table, we create a new instance of the corresponding AR class, set its properties associated with the table columns, and call the save() method to finish the insertion.

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

If the table's primary key is auto-incremental, after the insertion the AR instance will contain an updated primary key. In the above example, the `id` property will reflect the primary key value of the newly inserted post, even though we never change it explicitly.

If a column is defined with some static default value (e.g. a string, a number) in the table schema, the corresponding property in the AR instance will automatically has such a value after the instance is created. One way to change this default value is by explicitly declaring the property in the AR class:

```
class Post extends CActiveRecord
{
    public $title='please enter a title';
    ......
}

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

Starting from version 1.0.2, an attribute can be assigned a value of CDbExpression type before the record is saved (either insertion or updating) to the database. For example, in order to save a timestamp returned by the MySQL `NOW()` function, we can use the following code:

```
$post=new Post;
$post->createTime=new CDbExpression('NOW()');
// $post->createTime='NOW()'; will not work because
// 'NOW()' will be treated as a string
$post->save();
```

Tip: While AR allows us to perform database operations without writing cumbersom SQL statements, we often want to know what SQL statements are executed by AR underneath. This can be achieved by turning on the logging feature of Yii. For example, we can turn on CWebLogRoute in the application configuration, and we will see the executed SQL statements being displayed at the end of each Web page. Since version 1.0.5, we can set CDbConnection::enableParamLogging to be true in the application configuration so that the parameter values bound to the SQL statements are also logged.

Reading Record

To read data in a database table, we call one of the `find` methods as follows.

```
// find the first row satisfying the specified condition
$post=Post::model()->find($condition,$params);
// find the row with the specified primary key
$post=Post::model()->findByPk($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);
```

In the above, we call the `find` method with `Post::model()`. Remember that the static method `model()` is required for every AR class. The method returns an AR instance that is used to access class-level methods (something similar to static class methods) in an object context.

If the `find` method finds a row satisfying the query conditions, it will return a `Post` instance whose properties contain the corresponding column values of the table row. We can then read the loaded values like we do with normal object properties, for example, `echo $post->title;`.

The `find` method will return null if nothing can be found in the database with the given query condition.

When calling `find`, we use `$condition` and `$params` to specify query conditions. Here `$condition` can be string representing the `WHERE` clause in a SQL statement, and `$params` is an array of parameters whose values should be bound to the placeholders in `$condition`. For example,

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

Note: In the above, we may need to escape the reference to the `postID` column for certain DBMS. For example, if we are using PostgreSQL, we would have to write the condition as `"postID"=:postID`, because PostgreSQL by default will treat column names as case-insensitive.

We can also use `$condition` to specify more complex query conditions. Instead of a string, we let `$condition` be a CDbCriteria instance, which allows us to specify conditions other than the `WHERE` clause. For example,

```
$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
```

Note, when using CDbCriteria as query condition, the `$params` parameter is no longer needed since it can be specified in CDbCriteria, as shown above.

An alternative way to CDbCriteria is passing an array to the find method. The array keys and values correspond to the criteria's property name and value, respectively. The above example can be rewritten as follows,

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

Info: When a query condition is about matching some columns with the specified values, we can use findByAttributes(). We let the $attributes parameters be an array of the values indexed by the column names. In some frameworks, this task can be achieved by calling methods like findByNameAndTitle. Although this approach looks attractive, it often causes confusion, conflict and issues like case-sensitivity of column names.

When multiple rows of data matching the specified query condition, we can bring them in all together using the following findAll methods, each of which has its counterpart find method, as we already described.

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

If nothing matches the query condition, findAll would return an empty array. This is different from find who would return null if nothing is found.

Besides the find and findAll methods described above, the following methods are also provided for convenience:

```
// 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);
```

Updating Record

After an AR instance is populated with column values, we can change them and save them back to the database table.

```
$post=Post::model()->findByPk(10);
$post->title='new post title';
$post->save(); // save the change to database
```

As we can see, we use the same save() method to perform insertion and updating operations. If an AR instance is created using the new operator, calling save() would insert a new row into the database table; if the AR instance is the result of some find or findAll method call, calling save() would update the existing row in the table. In fact, we can use CActiveRecord::isNewRecord to tell if an AR instance is new or not.

It is also possible to update one or several rows in a database table without loading them first. AR provides the following convenient class-level methods for this purpose:

```
// 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);
```

In the above, $attributes is an array of column values indexed by column names; $counters is an array of incremental values indexed by column names; and $condition and $params are as described in the previous subsection.

Deleting Record

We can also delete a row of data if an AR instance has been populated with this row.

```
$post=Post::model()->findByPk(10); // assuming there is a post whose ID is 10
$post->delete(); // delete the row from the database table
```

Note, after deletion, the AR instance remains unchanged, but the corresponding row in the database table is already gone.

The following class-level methods are provided to delete rows without the need of loading them first:

```
// 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);
```

Data Validation

When inserting or updating a row, we often need to check if the column values comply to certain rules. This is especially important if the column values are provided by end users. In general, we should never trust anything coming from the client side.

AR performs data validation automatically when save() is being invoked. The validation is based on the rules specified by in the rules() method of the AR class. For more details about how to specify validation rules, refer to the Declaring Validation Rules section. Below is the typical workflow needed by saving a record:

```
if($post->save())
{
    // data is valid and is successfully inserted/updated
}
else
{
    // data is invalid. call getErrors() to retrieve error messages
}
```

When the data for inserting or updating is submitted by end users in an HTML form, we need to assign them to the corresponding AR properties. We can do so like the following:

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

If there are many columns, we would see a long list of such assignments. This can be alleviated by making use of the attributes property as shown below. More details can be found in the Securing Attribute Assignments section and the Creating Action section.

```
// assume $_POST['Post'] is an array of column values indexed by column names
$post->attributes=$_POST['Post'];
$post->save();
```

Comparing Records

Like table rows, AR instances are uniquely identified by their primary key values. Therefore, to compare two AR instances, we merely need to compare their primary key values, assuming they belong to the same AR class. A simpler way is to call CActiveRecord::equals(), however.

Info: Unlike AR implementation in other frameworks, Yii supports composite primary keys in its AR. A composite primary key consists of two or more columns. Correspondingly, the primary key value is represented as an array in Yii. The primaryKey property gives the primary key value of an AR instance.

Customization

CActiveRecord provides a few placeholder methods that can be overridden in child classes to customize its workflow.

- beforeValidate and afterValidate: these are invoked before and after validation is performed.
- beforeSave and afterSave: these are invoked before and after saving an AR instance.
- beforeDelete and afterDelete: these are invoked before and after an AR instance is deleted.
- afterConstruct: this is invoked for every AR instance created using the `new` operator.
- beforeFind: this is invoked before an AR finder is used to perform a query (e.g. `find()`, `findAll()`). This has been available since version 1.0.9.
- afterFind: this is invoked after every AR instance created as a result of query.

Using Transaction with AR

Every AR instance contains a property named dbConnection which is a CDbConnection instance. We thus can use the transaction feature provided by Yii DAO if it is desired when working with AR:

```
$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();
}
```

Named Scopes

Note: The support for named scopes has been available since version 1.0.5. The original idea of named scopes came from 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.

Named scopes are mainly declared in the CActiveRecord::scopes() method as name-criteria pairs. The following code declares two named scopes, `published` and `recently`, in the `Post` model class:

```
class Post extends CActiveRecord
```

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

Each named scope is declared as an array which can be used to initialize a CDbCriteria instance. For example, the `recently` named scope specifies that the `order` property to be `createTime DESC` and the `limit` property to be 5, which translates to a query criteria that should bring back the most recent 5 posts.

Named scopes are mostly used as modifiers to the `find` method calls. Several named scopes may be chained together and result in a more restrictive query result set. For example, to find the recently published posts, we can use the following code:

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

In general, named scopes must appear to the left of a `find` method call. 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.

Starting from version 1.0.6, named scopes can also be used with `update` and `delete` methods. For example, the following code would delete all recently published posts:

```
Post::model()->published()->recently()->delete();
```

Note: Named scopes can only be used with class-level methods. That is, the method must be called using `ClassName::model()`.

Parameterized Named Scopes

Named scopes can be parameterized. For example, we may want to customize the number of posts specified by the `recently` named scope. To do so, instead of declaring the named scope in the CActiveRecord::scopes method, we need to define a new method whose name is the same as the scope name:

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

Then, we can use the following statement to retrieve the 3 recently published posts:

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

If we do not supply the parameter 3 in the above, we would retrieve the 5 recently published posts by default.

Default Named Scope

A model class can have a default named scope that would be applied for all queries (including relational ones) about the model. For example, a website supporting multiple languages may only want to display contents that are in the language the current user specifies. Because there may be many queries about the site contents, we can define a default named scope to solve this problem. To do so, we override the CActiveRecord::defaultScope method as follows,

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

Now, if the following method call will automatically use the query criteria as defined above:

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

Note that default named scope only applies to SELECT queries. It is ignored for INSERT, UPDATE and DELETE queries.

2010 年 4 月 27 日 1 时 30 分 12 秒 albert

Relational Active Record

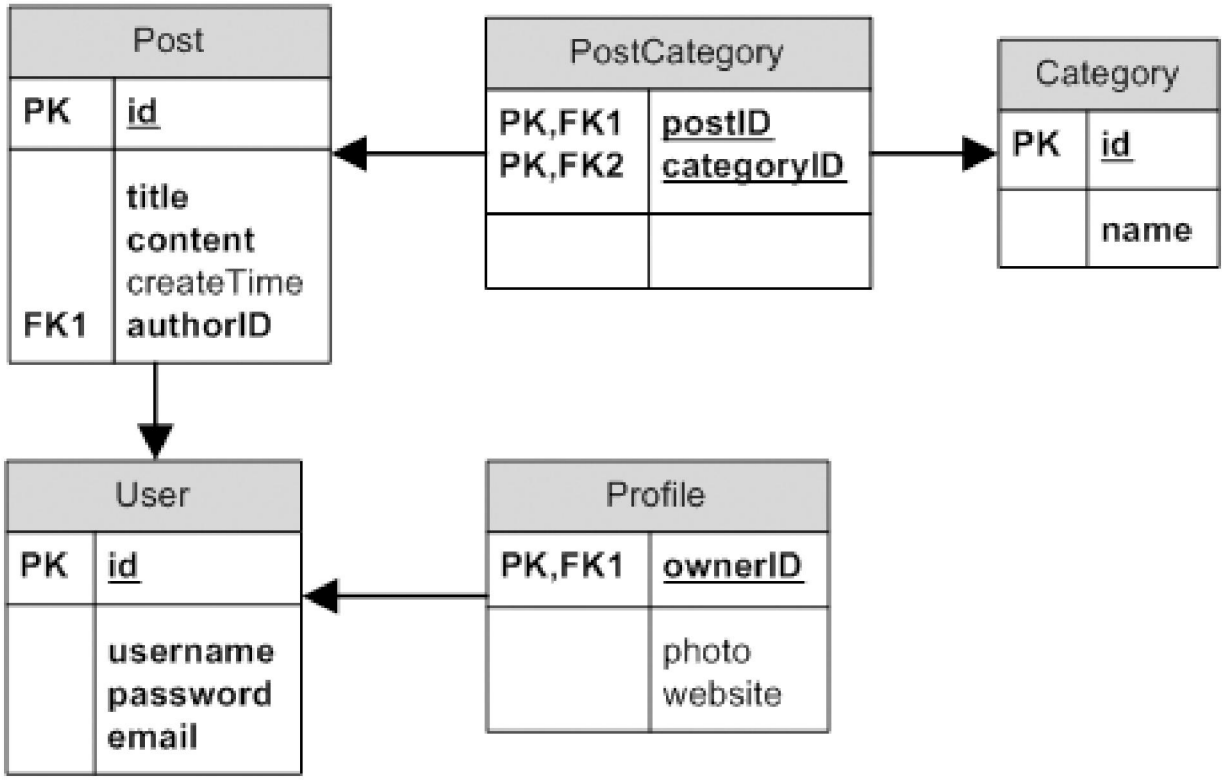
我们已经知道如何通过 Active Record（AR）从单个数据表中取得数据了，在这一节中，我们将要介绍如何使用 AR 来连接关联的数据表获取数据。

在使用关联 AR 之前，首先要在数据库中建立关联的数据表之间的主键-外键关联，AR 需要通过分析数据库中的定义数据表关联的元信息，来决定如何连接数据。

注意：从 1.0.1 版往后，使用关联 AR 不再依赖数据库中的外键约束定义。

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，但允许你在建立数据表时定义外键约束，AR 会利用 DDL 声明中的约束定义获得相应的信息，用来支持关联查询。

MySQL 数据库中的 InnoDB 表引擎支持外键约束，而 MyISAM 引擎不支持。因此我们建议你使用 InnoDB 作为数据库的表引擎。当然你也可以使用 MyISAM，可以通过下面的一个小技巧来实现 关联查询。

~ sql CREATE TABLE Foo (id INTEGER NOT NULL PRIMARY KEY); CREATE TABLE bar (id INTEGER NOT NULL PRIMARY KEY, fooID INTEGER COMMENT 'CONSTRAINT FOREIGN KEY (fooID) REFERENCES

Foo(id)'); ~ 就像上面的例子中的做法，把外键约束的定义写在字段注释中，AR 可以识别这些信息来确定 数据表之间的关联。

如何声明关联

在使用 AR 进行关联查询之前，我们需要告诉 AR 各个 AR 类之间有怎样的关联。

AR 类之间的关联直接反映着数据库中这个类所代表的数据表之间的关联。从关系数据库的角度来说，两个数据表 A，B 之间可能的关联有三种：一对多（例如 User 和 Post），一对一（例如 User 和 Profile），多对多（例如 Category 和 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（B belongs to A），例如 Post 属于 User。
- HAS_MANY: 如果数据表 A 和 B 的关系是多对一，那我们就说 B 有多个 A（B has many A），例如 User 有多个 Post。
- HAS_ONE: 这是'HAS_MANY'关系中的一个特例，当 A 最多有一个的时候，我们说 B 有一个 A（B has one A），例如一个 User 就只有一个 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，PostCategory 就是这个用作关联的表。用 AR 的方式去理解的话，我们可以认为 MANY_MANY 关系是由 BELONGS_TO 和 HAS_MANY 组成的. 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：

~ php 'VarName'=>array('RelationType', 'ClassName', 'ForeignKey', ...additional options) ~

这里的 VarName 是这个关联的名称；RelationType 指定了这个关联的类型，有四个常量代表了四种关联的类型：self::BELONGS_TO，self::HAS_ONE，self::HAS_MANY 和 self::MANY_MANY； ClassName 是这个关系关联到的 AR 类的类名；ForeignKey 指定了这个关联是通过哪个外键联系起来的。后面的 additional options 可以加入一些额外的设置，后面会做介绍。

下面的代码演示了如何定义 User 和 Post 之间的关联。

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

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

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

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

关联查询

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

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

说明: If there is no related instance for a relationship, the corresponding property could be either null or an empty array. For `BELONGS_TO` and `HAS_ONE` relationships, the result is null; for `HAS_MANY` and `MANY_MANY`, it is an empty array.

The lazy loading approach is very convenient to use, but it is not efficient in some scenarios. For example, if we want to access the author information for `N` posts, using the lazy approach would involve executing `N` join queries. We should resort to the so-called *eager loading* approach under this circumstance.

The eager loading approach retrieves the related AR instances together with the main AR instance(s). This is accomplished by using the with() method together with one of the find or findAll methods in AR. For example,

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

The above code will return an array of `Post` instances. Unlike the lazy approach, the `author` property in each `Post` instance is already populated with the related `User` instance before we access the property. Instead of executing a join query for each post, the eager loading approach brings back all posts together with their authors in a single join query!

We can specify multiple relationship names in the with() method and the eager loading approach will bring them back all in one shot. For example, the following code will bring back posts together with their authors and categories:

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

We can also do nested eager loading. Instead of a list of relationship names, we pass in a hierarchical representation of relationship names to the with() method, like the following,

~ php $posts=Post::model()->with(array('author'=>array('profile', 'posts'), 'categories'))->findAll(); ~

The above example will bring back all posts together with their author and categories. It will also bring back each author's profile and posts.

说明: The AR implementation in Yii is very efficient. When eager loading a hierarchy of related objects involving `N` `HAS_MANY` or `MANY_MANY` relationships, it will take `N+1` SQL queries to obtain the needed results. This means it needs to execute 3 SQL queries in the last example because of the `posts` and `categories` properties. Other frameworks take a more radical approach by using only one SQL query. At first look, this approach seems more efficient because fewer queries are being parsed and executed by DBMS. It is in fact impractical in reality for two reasons. First, there are many repetitive data columns in the result which takes extra time to transmit and process. Second, the number of rows in the result set grows exponentially with the number of tables involved, which makes it simply unmanageable as more relationships are involved.

Relational Query Options

We mentioned that additional options can be specified in relationship declaration. These options, specified as name-value pairs, are used to customize the relational query. They are summarized as below.

- select: a list of columns to be selected for the related AR class. It defaults to '*', meaning all columns. Column names should be disambiguated using aliasToken if they appear in an expression (e.g. COUNT(??.name) AS nameCount).
- condition: the WHERE clause. It defaults to empty. Note, column references need to be disambiguated using aliasToken (e.g. ??.id=10).
- order: the ORDER BY clause. It defaults to empty. Note, column references need to be disambiguated using aliasToken (e.g. ??.age DESC).
- with: a list of child related objects that should be loaded together with this object. Note, this is only honored by lazy loading, not eager loading.
- joinType: type of join for this relationship. It defaults to LEFT OUTER JOIN.
- aliasToken: the column prefix placeholder. It will be replaced by the corresponding table alias to disambiguate column references. It defaults to '??.'.
- alias: the alias for the table associated with this relationship. This option has been available since version 1.0.1. It defaults to null, meaning the table alias is automatically generated. This is different from aliasToken in that the latter is just a placeholder and will be replaced by the actual table alias.

In addition, the following options are available for certain relationships during lazy loading:

- group: the GROUP BY clause. It defaults to empty. Note, column references need to be disambiguated using aliasToken (e.g. ??.age). This option only applies to HAS_MANY and MANY_MANY relationships.
- having: the HAVING clause. It defaults to empty. Note, column references need to be disambiguated using aliasToken (e.g. ??.age). This option only applies to HAS_MANY and MANY_MANY relationships. Note: option has been available since version 1.0.1.
- limit: limit of the rows to be selected. This option does NOT apply to BELONGS_TO relation.
- offset: offset of the rows to be selected. This option does NOT apply to BELONGS_TO relation.

Below we modify the posts relationship declaration in the User by including some of the above options:

~ php class User extends CActiveRecord { public function relations() { return array('posts'=>array(self::HAS_MANY, 'Post', 'authorID' 'order'=>'??.createTime DESC', 'with'=>'categories'), 'profile'=>array(self::HAS_ONE, 'Profile', 'ownerID'),); } } ~

Now if we access $author->posts, we would obtain the author's posts sorted according to their creation time in descending order. Each post instance also has its categories loaded.

说明: When a column name appears in two or more tables being joined together, it needs to be disambiguated. This is done by prefixing the column name with its table name. For example, id becomes Team.id. In AR relational queries, however, we do not have this freedom because the SQL statements are automatically generated by AR which systematically gives each table an alias. Therefore, in order to avoid column name conflict, we use a

placeholder to indicate the existence of a column which needs to be disambiguated. AR will replace the placeholder with a suitable table alias and properly disambiguate the column.

缓存

缓存是用于提升网站性能的一种即简单又有效的途径。通过存储相对静态的数据至缓存以备所需，我们可以省去生成这些数据的时间。

在 Yii 中使用缓存主要包括配置和访问缓存组件。如下的应用配置指定了一个使用两台缓存服务器的 memcache 缓存组件：

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

程序运行的时候可以通过 `Yii::app()->cache` 来访问缓存组件。

Yii 提供多种缓存组件以便在不同的媒介上存储缓存数据。比如 CMemCache 组件封装了 PHP memcache 扩展，它使用内存作为存储缓存的媒介；CApcCache 组件封装了 PHP APC 扩展；CDbCache 组件在数据库里存储缓存数据。下面是各种缓存组件的简要说明：

- CMemCache: 使用 PHP memcache 扩展.
- CApcCache: 使用 PHP APC 扩展.
- CXCache: 使用 PHP XCache 扩展. 注意，该组件从 1.0.1 版本开始提供。
- CDbCache: 使用一张数据库表来存储缓存数据。它默认在运行时目录建立并使用一个 SQLite3 数据库，你可以通过设置 connectionID 属性显式地指定一个数据库给它使用。

提示：因为所有这些缓存组件都从同一个基础类 CCache 扩展而来，不需要修改使用缓存的代码即可在不同的缓存组件之间切换。

缓存可以在不同的级别使用。在最低级别，我们使用缓存来存储单个数据，比如一个变量，我们把它叫做 *数据缓存*。往上一级，我们缓存一个由视图脚本生成的页面片断。在最高级别，我们存储整个页面以便需要的时候直接从缓存读取。

接下来我们将阐述如何在这些级别上使用缓存。

注意：按定义来讲，缓存是一个不稳定的存储媒介，它不保证缓存一定存在——不管该缓存是否过期。所以，不要使用缓存进行持久存储（比如，不要使用缓存来存储 SESSION 数据）。

数据缓存

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

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

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

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

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

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

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

为一个要缓存的变量选择 ID 时，确保该 ID 在应用中是唯一的。不必保证 ID 在跨应用的情况下保证唯一，因为缓存组件有足够的智能来区分不同应用的缓存 ID。

要从缓存中删除一个缓存值，调用 delete()；要清空所有缓存，调用 flush()。调用 flush() 时要非常小心，因为它会把其它应用的缓存也清空。

提示：因为 CCache 实现了 ArrayAccess 接口，可以像数组一样使用缓存组件。例如：

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

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

缓存依赖

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

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

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

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

下面是可用的缓存依赖的简要说明：

- CFileCacheDependency: 该依赖因文件的最近修改时间发生改变而改变。
- CDirectoryCacheDependency: 该依赖因目录（或其子目录）下的任何文件发生改变而改变。
- CDbCacheDependency: 该依赖因指定的 SQL 语句的查询结果发生改变而改变。
- CGlobalStateCacheDependency: 该依赖因指定的全局状态值发生改变而改变。全局状态是应用中跨请求、跨 SESSION 的持久变量，它由 CApplication::setGlobalState() 来定义。
- CChainedCacheDependency: 该依赖因依赖链中的任何一环发生改变而改变。

片段缓存(Fragment Caching)

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

要使用片段缓存，在控制器视图脚本中调用 CController::beginCache() 和 CController::endCache() 。这两种方法开始和结束包括的页面内容将被缓存。类似 data caching ，我们需要一个编号，识别被缓存的片段。

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

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

缓存选项(Caching Options)

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

有效期（Duration）

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

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

如果我们不设定期限，它将默认为 60 ，这意味着 60 秒后缓存内容将无效。

依赖(Dependency)

像 data caching ，内容片段被缓存也可以有依赖。例如，文章的内容被显示取决于文章是否被修改。

要指定一个依赖，我们建立了 dependency选项，可以是一个实现 ICacheDependency的对象或可用于生成依赖对象的配置数组。下面的代码指定片段内容取决于 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(Variation)

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

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

- varyByRoute: 设置此选项为 true ，缓存的内容将根据 route 变化。因此，每个控制器和行动的组合将有一个单独的缓存内容。

- varyBySession: 设置此选项为 true ，缓存的内容将根据 session ID 变化。因此，每个用户会话可能会看到由缓存提供的不同内容。
- varyByParam: 设置此选项的数组里的名字，缓存的内容将根据 GET 参数的值变动。例如，如果一个页面显示文章的内容根据 id 的 GET 参数，我们可以指定 varyByParam 为 array('id')，以使我们能够缓存每篇文章内容。如果没有这样的变化，我们只能能够缓存某一文章。

请求类型(Request Types)

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

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

嵌套缓存(Nested Caching)

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

```
...其他 HTML 内容...
<?php if($this->beginCache($id1)) { ?>
...外部被缓存内容...
    <?php if($this->beginCache($id2)) { ?>
    ...内部被缓存内容...
    <?php $this->endCache(); } ?>
...外部被缓存内容...
<?php $this->endCache(); } ?>
...其他 HTML 内容...
```

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

页面缓存

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

页面缓存可以被看作是 片段缓存 (/doc/guide/caching.fragment)一个特殊情况 。由于网页内容是往往通过应用布局来生成，如果我们只是简单的在布局中调用 beginCache()和 endCache()，将无法正常工作。这是因为布局在CController::render()方法里的加载是在页面内容产生之后。

缓存整个页面，我们应该跳过产生网页内容的动作执行。我们可以使用 COutputCache 作为动作 过滤器（/doc/guide/basics.controller#filter）来完成这一任务。下面的代码演示如何配置缓存过滤器：

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

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

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

动态内容(Dynamic Content)

当使用 fragment caching 或 page caching，我们常常遇到的这样的情况整个部分的输出除了个别地方都是静态的。例如，帮助页可能会显示静态的帮助信息，而用户名称显示的是当前用户的。

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

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

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

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

```
<?php $this->endCache(); } ?>
...别的 HTML 内容...
```

在上面的， $callback 指的是有效的 PHP 回调。它可以是指向当前控制器类的方法或者全局函数的字符串名。它也可以是一个数组名指向一个类的方法。其他任何的参数，将传递到 renderDynamic()方法中。回调将返回动态内容而不是仅仅显示它。

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;当你编写一个新的组件时,你正在继承 CWidget 或者一个已存在的组件类.如果扩展代码是由第三方开发者为了复用而设计的,我们则称之为 *extension(扩展)*.

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

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

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

适用扩展通常半酣了以下三步:

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

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

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

应用的部件

使用 应用的部件, 首先我们需要添加一个新条目到 应用配置 的 components 属性, 如下所示:

```
return array(
    // 'preload'=>array('xyz',...),
    'components'=>array(
        'xyz'=>array(
            'class'=>'application.extensions.xyz.XyzClass',
            'property1'=>'value1',
            'property2'=>'value2',
        ),
        // 其他部件配置
    ),
);
```

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

组件

组件 主要用在 视图 里.假设组件类 XyzClass 属于 xyz 扩展,我们可以如下在视图中使用它:

```
// 组件不需要主体内容
<?php $this->widget('application.extensions.xyz.XyzClass', array(
    'property1'=>'value1',
    'property2'=>'value2')); ?>

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

...组件的主体内容...

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

动作

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

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

然后,我们可以通过 路由 test/xyz 来访问.

过滤器

过滤器 也被 控制器 使用.过滤器主要用于当其被 动作 挂起时预处理,提交处理用户请求.假设过滤器的类 XyzClass 属于 xyz 扩展,我们可以在我们的控制器类里重写 CController::filters 方法来使用它:

```
class TestController extends CController
{
    public function filters()
    {
        return array(
            array(
                'application.extensions.xyz.XyzClass',
                'property1'=>'value1',
                'property2'=>'value2',
            ),
            // 其他过滤器
        );
    }
}
```

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

控制器

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

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

然后, 一个在控制里的 a 行为就可以通过 路由 xyz/a 来访问了.

校验器

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

```
class MyModel extends CActiveRecord // or CFormModel
{
    public function rules()
    {
        return array(
            array(
                'attr1, attr2',
                'application.extensions.xyz.XyzClass',
                'property1'=>'value1',
                'property2'=>'value2',
            ),
            // 其他校验规则
        );
    }
}
```

控制台命令

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

```
return array(
    'commandMap'=>array(
        'xyz'=>array(
            'class'=>'application.extensions.xyz.XyzClass',
            'property1'=>'value1',
            'property2'=>'value2',
        ),
        // 其他命令
    ),
);
```

然后,我们就能使用配备了额外命令 xyz 的 yiic 工具了.

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

模块

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

通用部件

使用一个通用 部件, 我们首先需要通过使用

```
Yii::import('application.extensions.xyz.XyzClass');
```

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

Creating Extensions（创建扩展）

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

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

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

Application Component（应用部件）

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

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

Widget（小工具）

widget 应继承 CWidget 或其子类。 A widget should extend from CWidget or its child classes.

最简单的方式建立一个新的小工具是继承一个现成的小工具和重载它的方法或改变其默认的属性值。例如，如果您想为 CTabView 使用更好的 CSS 样式，您可以配置其 CTabView::cssFile 属性，当使用的小工具时。您还可以继承 CTabView 如下，让您在使用小工具时，不再需要配置属性。

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

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

要从零开始创建一个新的小工具，我们主要是需要实现两个方法：CWidget::init 和 CWidget::run。第一种方法是当我们在视图中使用 $this->beginWidget 插入一个小工具时被调用，第二种方法在$this->endWidget 被调用时调用。如果我们想在这两个方法调用之间捕捉和处理显示的内容，我们可以开始 output buffering 在 CWidget::init 和在 CWidget::run 中回收缓冲输出作进一步处理。 If we want to capture and process the content displayed between these two method invocations, we can start output buffering in CWidget::init and retrieve the buffered output in CWidget::run for further processing.

在网页中使用的小工具，小工具往往包括 CSS，Javascript 或其他资源文件。我们叫这些文件 *assets*，因为他们和小工具类在一起，而且通常 Web 用户无法访问。为了使这些档案通过 Web 访问，我们需要用

CWebApplication::assetManager 发布他们，例如上述代码段所示。此外，如果我们想包括 CSS 或 JavaScript 文件在当前的网页，我们需要使用 CClientScript 注册 ：

```
class MyWidget extends CWidget
{
    protected function registerClientScript()
    {
        // ...publish CSS or JavaScript file here...
        $cs=Yii::app()->clientScript;
        $cs->registerCssFile($cssFile);
        $cs->registerScriptFile($jsFile);
    }
}
```

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

Action（动作）

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

Filter（过滤器）

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

```
class MyFilter extends CFilter
{
    protected function preFilter($filterChain)
    {
        // logic being applied before the action is executed
        return true; // false if the action should not be executed
    }

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

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

Controller（控制器）

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

Validator（验证）

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

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

Console Command（控制台命令）

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

```
class MyCommand extends CConsoleCommand
{
    public function run($args)
    {
        // $args gives an array of the command-line arguments for this command
    }

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

Module（模块）

请参阅 modules 一节中关于就如何创建一个模块。

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

Generic Component（通用组件）

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

Using 3rd-Party Libraries(使用第三方库)

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

下面我们用一个例子来说明如何在一个 Yii application 从 Zend framework 使用 Zend_Search_Lucene 部件。

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

第二，在一个 controller 类文件的开始，加入以下行：

```
Yii::import('application.vendors.*');

require_once('Zend/Search/Lucene.php');
```

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

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

```
$lucene=new Zend_Search_Lucene($pathOfIndex);

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

URL Management(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 应用程序可以理解的。对于 Yii 应用程序，这些通过 CUrlManager 辅助完成。

Creating URLs（创建网址）

虽然 URL 可被硬编码在控制器的视图（view）文件，但往往可以很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，URL 以 get 格式使用 createUrl 创建。例如，提供$route='post/read' 和 $params=array('id'=>100) ，我们将获得以下网址：

```
/index.php?r=post/read&id=100
```

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

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 应用元件，以便 createUrl 可以自动切换到新格式和应用程序可以正确理解新的网址：

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

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

createurl 方法所产生的是一个相对地址。为了得到一个绝对的 url ，我们可以用前缀 yii">

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

User-friendly URLs（用户友好的 URL）

当用 path 格式 URL，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，我们可以产生一个短短的 URL/post/100 ，而不是冗长/index.php/post/read/id/100。网址创建和解析都是通过 CUrlManager 指定网址规则。

要指定的 URL 规则，我们必须设定 urlManager 应用元件的属性 rules：

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

这些规则以一系列的路线格式对数组指定，每对对应于一个单一的规则。路线（route）的格式必须是有效的正则表达式，没有分隔符和修饰语。它是用于匹配网址的路径信息部分。还有 route 应指向一个有效的路线控制器。

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。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 参数将通过解析结果来生成。

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 生成网址，路线和传递给该方法的 GET 参数被用来决定哪些网址规则适用。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 ，如果路线的规则也匹配路线参数，规则将用来生成网址。

如果 GET 参数传递到 createUrl 是以上所要求的一项规则，其他参数将出现在查询字符串。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线 post/read 和 GET 参数 array('id'=>100) （可通过$_GET 获得） 。

createurl 方法所产生的是一个相对地址。为了得到一个绝对的 url ，我们可以用前缀 yii">

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。

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 应用程序元件。

我们首先需要配置 Web 服务器，这样一个 URL 没有入口脚本仍然可以处理入口脚本。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 元件的 showScriptName 属性为 false。

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

Faking URL Suffix(伪造 URL 后缀)

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

验证和授权(Authentication and Authorization)

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 （Defining Identity Class）

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)，用户身份信息将被存放在 cookie 中。确记敏感信息不要存放(例如 password) 。

登录和注销（Login and Logout）

使用身份类和用户部件，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，用户将根据 session configuration 完成一序列 inactivity 动作后注销。设置用户部件的 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，此用户将保留用户登陆状态时间为被设置的持续时间之久。前提是用户的浏览器接受 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（Access Control Filter）

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'=>'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: 设置哪个动作匹配此规则。
- users: 设置哪个用户匹配此规则。此当前用户的 name 被用来匹配. 三种设定字符在这里可以用：
    - *: 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: 设定哪个角色匹配此规则。这里用到了将在后面描述的 role-based access control 技术。In particular, the rule is applied if CWebUser::checkAccess returns true for one of the roles.提示，用户角色应该被设置成 allow 规则，因为角色代表能做某些事情。
- ips: 设定哪个客户端 IP 匹配此规则。
- verbs: 设定哪种请求类型(例如：GET, POST)匹配此规则。
- 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（Handling Authorization Result）

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：

- 如果用户没有登录和在用户部件中配置了 loginUrl，浏览器将重定位网页到此配置 URL。
- 否则一个错误代码 401 的 HTTP 例外将显示。

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（Role-Based Access Control）

基于角色的访问控制（ RBAC 的）提供了一种简单而强大集中访问控制。请参阅维基文章（http://en.wikipedia.org/wiki/Role-based_access_control）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 实现通过其 authManager 应用程序组件分级 RBAC 的模式。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（Overview）

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许可（如创造新的博客发布，管理用户）。根据其粒度和 targeted audience， 授权项目可分为 *operations*（行动）,*tasks*（任务）和 *roles*（角色）。角色包括任务，任务包括行动，行动是许可是个原子。 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 行动。为了更灵活，Yii 也可以允许角色包括其他角色和动作，任务包括其他任务，行动包括其他行动。

授权项目通过名称唯一确定。

授权项目可能与 *business rule*（业务规则）关联。业务规则是一块 PHP 代码，将在检查访问此项的相关时被执行。只有当执行返回 true，用户将被视为有权限此项所代表的许可。举例来说，当定义一项行动 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。

使用授权的项目，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，行动项在最底，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（Configuring Authorization Manager）

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授权。当我们配置 authManager 应用部件，我们需要指定哪些部件类和部件的初始值。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 应用部件。

定义授权等级（Defining Authorization Hierarchy）

定义授权等级涉及三个步骤：定义授权项目，建立授权项目关系项目，并分配角色给应用用户。authManager 应用部件提供了一整套的 API 来完成这些任务。

根据不同种类的项目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');
```

请注意，我们给 updateOwnPost 任务关联一个业务规则。在这个业务规则，我们只是检查目前的用户 ID 是否和指定的帖子的作者 ID 一样。当执行访问检查时，发布信息在开发者提供$params 数组中。

info|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。

访问检查（Access Checking）

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 行动的权限。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商业规则，我们同样可以通过他们。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（Default Roles）

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力。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。它们无需象前面所描述的那样去被分配给用户。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被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;';
$auth->createRole('authenticated',$bizRule);

$bizRule='return Yii::app()->user->isGuest;';
$auth->createRole('guest',$bizRule);
```

Theming(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。主题的名字就是他的目录名字。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 view 文件必须位于 views 下 ，布局 view 文件在 views/layouts 下 ，和系统 view 文件在 views/system 下。例如，如果我们要替换 PostController 的 create view 文件为 classic 主题下，我们将保存新的 view 文件为 WebRoot/themes/classic/views/post/create.php。

对于在 module 里面的控制器 view 文件，相应主题 view 文件将被放在 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 view 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 view 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 render 渲染。否则，就后退到 viewPath 和 layoutPath 所指定的预设位置寻找。

baseurl 属性，我们就可以为此图像文件生成如下 url，

```
yii">
```

提示：在一个主题的视图，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'
```

Logging

Yii provides a flexible and extensible logging feature. Messages logged can be classified according to log levels and message categories. Using level and category filters, selected messages can be further routed to different destinations, such as files, emails, browser windows, etc.

Message Logging

Messages can be logged by calling either Yii::log or Yii::trace. The difference between these two methods is that the latter logs a message only when the application is in debug mode.

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

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

When logging a message, we need to specify its category and level. Category is a string in the format of xxx. yyy. zzz which resembles to the path alias. For example, if a message is logged in CController, we may use the category system. web. CController. Message level should be one of the following values:

- trace: this is the level used by Yii::trace. It is for tracing the execution flow of the application during development.
- info: this is for logging general information.
- profile: this is for performance profile which is to be described shortly.
- warning: this is for warning messages.
- error: this is for fatal error messages.

Message Routing

Messages logged using Yii::log or Yii::trace are kept in memory. We usually need to display them in browser windows, or save them in some persistent storage such as files, emails. This is called *message routing*, i.e., sending messages to different destinations.

In Yii, message routing is managed by a CLogRouter application component. It manages a set of the so-called *log routes*. Each log route represents a single log destination. Messages sent along a log route can be filtered according to their levels and categories.

To use message routing, we need to install and preload a CLogRouter application component. We also need to configure its routes property with the log routes that we want. The following shows an example of the needed application configuration:

```
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',
                ),
            ),
        ),
    ),
)
```

In the above example, we have two log routes. The first route is CFileLogRoute which saves messages in a file under the application runtime directory. Only messages whose level is `trace` or `info` and whose category starts with `system`. are saved. The second route is CEmailLogRoute which sends messages to the specified email addresses. Only messages whose level is `error` or `warning` are sent.

The following log routes are available in Yii:

- CDbLogRoute: saves messages in a database table.
- CEmailLogRoute: sends messages to specified email addresses.
- CFileLogRoute: saves messages in a file under the application runtime directory.

- CWebLogRoute: displays messages at the end of the current Web page.
- CProfileLogRoute: displays profiling messages at the end of the current Web page.

Info: Message routing occurs at the end of the current request cycle when the onEndRequest event is raised. To explicitly terminate the processing of the current request, call CApplication::end() instead of `die()` or `exit()`, because CApplication::end() will raise the onEndRequest event so that the messages can be properly logged.

Message Filtering

As we mentioned, messages can be filtered according to their levels and categories before they are sent long a log route. This is done by setting the levels and categories properties of the corresponding log route. Multiple levels or categories should be concatenated by commas.

Because message categories are in the format of `xxx.yyy.zzz`, we may treat them as a category hierarchy. In particular, we say `xxx` is the parent of `xxx.yyy` which is the parent of `xxx.yyy.zzz`. We can then use `xxx.*` to represent category `xxx` and all its child and grandchild categories.

Logging Context Information

Starting from version 1.0.6, we can specify to log additional context information, such as PHP predefined variables (e.g. `$_GET`, `$_SERVER`), session ID, user name, etc. This is accomplished by specifying the CLogRoute::filter property of a log route to be a suitable log filter.

The framework comes with the convenient CLogFilter that may be used as the needed log filter in most cases. By default, CLogFilter will log a message with variables like `$_GET`, `$_SERVER` which often contains valuable system context information. CLogFilter can also be configured to prefix each logged message with session ID, username, etc., which may greatly simplifying the global search when we are checking the numerous logged messages.

The following configuration shows how to enable logging context information. Note that each log route may have its own log filter. And by default, a log route does not have a log filter.

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

```
)
```

Starting from version 1.0.7, Yii supports logging call stack information in the messages that are logged by calling `Yii::trace`. This feature is disabled by default because it lowers performance. To use this feature, simply define a constant named `YII_TRACE_LEVEL` at the beginning of the entry script (before including `yii.php`) to be an integer greater than 0. Yii will then append to every trace message with the file name and line number of the call stacks belonging to application code. The number `YII_TRACE_LEVEL` determines how many layers of each call stack should be recorded. This information is particularly useful during development stage as it can help us identify the places that trigger the trace messages.

Performance Profiling

Performance profiling is a special type of message logging. Performance profiling can be used to measure the time needed for the specified code blocks and find out what the performance bottleneck is.

To use performance profiling, we need to identify which code blocks need to be profiled. We mark the beginning and the end of each code block by inserting the following methods:

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

where `blockID` is an ID that uniquely identifies the code block.

Note, code blocks need to be nested properly. That is, a code block cannot intersect with another. It must be either at a parallel level or be completely enclosed by the other code block.

To show profiling result, we need to install a CLogRouter application component with a CProfileLogRoute log route. This is the same as we do with normal message routing. The CProfileLogRoute route will display the performance results at the end of the current page.

Profiling SQL Executions

Profiling is especially useful when working with database since SQL executions are often the main performance bottleneck of an application. While we can manually insert `beginProfile` and `endProfile` statements at appropriate places to measure the time spent in each SQL execution, starting from version 1.0.6, Yii provides a more systematic approach to solve this problem.

By setting CDbConnection::enableProfiling to be true in the application configuration, every SQL statement being executed will be profiled. The results can be readily displayed using the aforementioned CProfileLogRoute, which can show us how much time is spent in executing what SQL statement. We can also call CDbConnection::getStats() to retrieve the total number SQL statements executed and their total execution time.

Error Handling

Yii provides a complete error handling framework based on the PHP 5 exception mechanism. When the application is created to handle an incoming user request, it registers its handleError method to handle PHP warnings and notices; and it registers its handleException method to handle uncaught PHP exceptions. Consequently, if a PHP warning/notice or an uncaught exception occurs during the application execution, one of the error handlers will take over the control and start the necessary error handling procedure.

Tip: The registration of error handlers is done in the application's constructor by calling PHP functions set_exception_handler and set_error_handler. If you do not want Yii to handle the errors and exceptions, you may define constant YII_ENABLE_ERROR_HANDLER and YII_ENABLE_EXCEPTION_HANDLER to be false in the entry script.

By default, errorHandler (or exceptionHandler) will raise an onError event (or onException event). If the error (or exception) is not handled by any event handler, it will call for help from the errorHandler application component.

Raising Exceptions

Raising exceptions in Yii is not different from raising a normal PHP exception. One uses the following syntax to raise an exception when needed:

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

Yii defines two exception classes: CException and CHttpException. The former is a generic exception class, while the latter represents an exception that should be displayed to end users. The latter also carries a statusCode property representing an HTTP status code. The class of an exception determines how it should be displayed, as we will explain next.

Tip: Raising a CHttpException exception is a simple way of reporting errors caused by user misoperation. For example, if the user provides an invalid post ID in the URL, we can simply do the following to show a 404 error (page not found):

```
// if post ID is invalid
throw new CHttpException(404,'The specified post cannot be found.');
```

Displaying Errors

When an error is forwarded to the CErrorHandler application component, it chooses an appropriate view to display the error. If the error is meant to be displayed to end users, such as a CHttpException, it will use a view named errorXXX, where XXX stands for the HTTP status code (e.g. 400, 404, 500). If the error is an internal one and should only be displayed to developers, it will use a view named exception. In the latter case, complete call stack as well as the error line information will be displayed.

Info: When the application runs in production mode, all errors including those internal ones will be displayed using view errorXXX. This is because the call stack of an error may contain sensitive information. In this case, developers should rely on the error logs to determine what is the real cause of an error.

CErrorHandler searches for the view file corresponding to a view in the following order:

1. WebRoot/themes/ThemeName/views/system: this is the system view directory under the currently active theme.
2. WebRoot/protected/views/system: this is the default system view directory for an application.
3. yii/framework/views: this is the standard system view directory provided by the Yii framework.

Therefore, if we want to customize the error display, we can simply create error view files under the system view directory of our application or theme. Each view file is a normal PHP script consisting of mainly HTML code. For more details, please refer to the default view files under the framework's view directory.

Handling Errors Using an Action

Starting from version 1.0.6, Yii allows using a controller action to handle the error display work. To do so, we should configure the error handler in the application configuration as follows:

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

In the above, we configure the CErrorHandler::errorAction property to be the route site/error which refers to the error action in SiteController. We may use a different route if needed.

We can write the error action like the following:

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

In the action, we first retrieve the detailed error information from CErrorHandler::error. If it is not empty, we render the error view together with the error information. The error information returned from CErrorHandler::error is an array with the following fields:

- code: the HTTP status code (e.g. 403, 500);
- type: the error type (e.g. CHttpException, PHP Error);
- message: the error message;
- file: the name of the PHP script file where the error occurs;
- line: the line number of the code where the error occurs;

- `trace`: the call stack of the error;
- `source`: the context source code where the error occurs.

Tip: The reason we check if CErrorHandler::error is empty or not is because the `error` action may be directly requested by an end user, in which case there is no error. Since we are passing the `$error` array to the view, it will be automatically expanded to individual variables. As a result, in the view we can access directly the variables such as `$code`, `$type`.

Message Logging

A message of level `error` will always be logged when an error occurs. If the error is caused by a PHP warning or notice, the message will be logged with category `php`; if the error is caused by an uncaught exception, the category would be `exception.ExceptionClassName` (for CHttpException its statusCode will also be appended to the category). One can thus exploit the logging feature to monitor errors happened during application execution.

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。在 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 有效和客户端怎么援引。当客户端援引 API，Yii 将实例化相应的 service provider 和调用被请求的 API 来完成请求。

注: CWebService 依靠 PHP SOAP extension 。请确定您是否在试用本节中的例子前允许此扩展。

Defining Service Provider（定义 Service Provider）

正如我们上文所述，service provider 是一个类定义能被远程援引的方法。Yii 依靠 doc comment and class reflection 识别哪些方法可以被远程调用和他们的参数还有返回值。

让我们以一个简单的股票报价服务开始。这项服务允许客户端请求指定股票的报价。我们确定 service provider 如下。请注意，我们定义扩展 CController 的提供类 `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。依靠文档注释指定输入的参数数据类型和返回值。其他的 API 可使用类似方式声明。

Declaring Web Service Action（定义 Web Service 动作）

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。可以做到这一点很容易，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网址 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。这就是因为我们定义 getPrice 方法在 StockController 中。

Consuming Web Service（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 。

Data Types（数据类型）

当定义的方法和属性被远程访问，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，它被看作是复合型组成的属性。复合型类型被看做类，他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。这将定义指定类型的数组。

下面就是一个例子定义 getPosts 网页 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;
```

```
}
```

Class Mapping（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。

```
class PostController extends CController
{
    public function actions()
    {
        return array(
            'service'=>array(
                'class'=>'CWebServiceAction',
                'classMap'=>array(
                    'Post'=>'Post',  // or simply 'Post'
                ),
            ),
        );
    }
    ......
}
```

Intercepting Remote Method Invocation（拦截远程方法调用）

通过实现 IWebServiceProvider 接口，sevice provider 可以拦截远程方法调用。在 IWebServiceProvider::beforeWebMethod ， sevice provider 可以获得当前 CWebService 实例和通过 CWebService::methodName 请求的方法的名字 。它可以返回假如果远程方法出于某种原因不应被援引（例如：未经授权的访问） 。

Internationalization

Internationalization (I18N) refers to the process of designing a software application so that it can be adapted to various languages and regions without engineering changes. For Web applications, this is of particular importance because the potential users may be from worldwide.

Yii provides support for I18N in several aspects.

- It provides the locale data for each possible language and variant.
- It provides message and file translation service.
- It provides locale-dependent date and time formatting.
- It provides locale-dependent number formatting.

In the following subsections, we will elaborate each of the above aspects.

Locale and Language

Locale is a set of parameters that defines the user's language, country and any special variant preferences that the user wants to see in their user interface. It is usually identified by an ID consisting of a language ID and a region ID. For example, the ID `en_US` stands for the locale of English and United States. For consistency, all locale IDs in Yii are canonicalized to the format of `LanguageID` or `LanguageID_RegionID` in lower case (e.g. `en`, `en_us`).

Locale data is represented as a CLocale instance. It provides locale-dependent information, including currency symbols, number symbols, currency formats, number formats, date and time formats, and date-related names. Since the language information is already implied in the locale ID, it is not provided by CLocale. For the same reason, we often interchangeably using the term locale and language.

Given a locale ID, one can get the corresponding CLocale instance by `CLocale::getInstance($localeID)` or `CApplication::getLocale($localeID)`.

Info: Yii comes with locale data for nearly every language and region. The data is obtained from Common Locale Data Repository (CLDR). For each locale, only a subset of the CLDR data is provided as the original data contains a lot of rarely used information. Starting from version 1.1.0, users can also supply their own customized locale data. To do so, configure the CApplication::localeDataPath property with the directory that contains the customized locale data. Please refer to the locale data files under `framework/i18n/data` in order to create customized locale data files.

For an Yii application, we differentiate its target language from source language. The target language is the language (locale) of the users that the application is targeted at, while the source language refers to the language (locale) that the application source files are written in. Internationalization occurs only when the two languages are different.

One can configure target language in the application configuration, or change it dynamically before any internationalization occurs.

Tip: Sometimes, we may want to set the target language as the language preferred by a user (specified in user's browser preference). To do so, we can retrieve the user preferred language ID using CHttpRequest::preferredLanguage.

Translation

The most needed I18N feature is perhaps translation, including message translation and view translation. The former translates a text message to the desired language, while the latter translates a whole file to the desired language.

A translation request consists of the object to be translated, the source language that the object is in, and the target language that the object needs to be translated to. In Yii, the source language is default to the application source language while the target language is default to the application language. If the source and target languages are the same, translation will not occur.

Message Translation

Message translation is done by calling Yii::t(). The method translates the given message from source language to target language.

When translating a message, its category has to be specified since a message may be translated differently under different categories (contexts). The category `yii` is reserved for messages used by the Yii framework core code.

Messages can contain parameter placeholders which will be replaced with the actual parameter values when calling Yii::t(). For example, the following message translation request would replace the `{alias}` placeholder in the original message with the actual alias value.

```
Yii::t('yii', 'Path alias "{alias}" is redefined.',
    array('{alias}'=>$alias))
```

Note: Messages to be translated must be constant strings. They should not contain variables that would change message content (e.g. `"Invalid {$message} content."`). Use parameter placeholders if a message needs to vary according to some parameters.

Translated messages are stored in a repository called *message source*. A message source is represented as an instance of CMessageSource or its child class. When Yii::t() is invoked, it will look for the message in the message source and return its translated version if it is found.

Yii comes with the following types of message sources. You may also extend CMessageSource to create your own message source type.

- CPhpMessageSource: the message translations are stored as key-value pairs in a PHP array. The original message is the key and the translated message is the value. Each array represents the translations for a particular category of messages and is stored in a separate PHP script file whose name is the category name. The PHP translation files for the same language are stored under the same directory named as the locale ID. And all these directories are located under the directory specified by basePath.
- CGettextMessageSource: the message translations are stored as GNU Gettext files.
- CDbMessageSource: the message translations are stored in database tables. For more details, see the API documentation for CDbMessageSource.

A message source is loaded as an application component. Yii pre-declares an application component named messages to store messages that are used in user application. By default, the type of this message source is CPhpMessageSource and the base path for storing the PHP translation files is `protected/messages`.

In summary, in order to use message translation, the following steps are needed:

1. Call Yii::t() at appropriate places;
2. Create PHP translation files as `protected/messages/LocaleID/CategoryName.php`. Each file simply returns an array of message translations. Note, this assumes you are using the default CPhpMessageSource to store the translated messages.
3. Configure CApplication::sourceLanguage and CApplication::language.

Tip: 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.

Starting from version 1.0.10, when using CPhpMessageSource to manage message source, messages for an extension class (e.g. a widget, a module) can be specially managed and used. In particular, if a message belongs to an extension whose class name is `Xyz`, then the message category can be specified in the format of `Xyz.categoryName`. The corresponding message file will be assumed to be `BasePath/messages/LanguageID/categoryName.php`, where `BasePath` refers to the directory that contains the extension class file. And when using `Yii::t()` to translate an extension message, the following format should be used, instead:

```
Yii::t('Xyz.categoryName', 'message to be translated')
```

Since version 1.0.2, Yii has added the support for choice format. Choice format refers to choosing a translated according to a given number value. For example, in English the word 'book' may either take a singular form or a plural form depending on the number of books, while in other languages, the word may not have different form (such as Chinese) or may have more complex plural form rules (such as Russian). Choice format solves this problem in a simple yet effective way.

To use choice format, a translated message must consist of a sequence of expression-message pairs separated by `|`, as shown below:

```
'expr1#message1|expr2#message2|expr3#message3'
```

where `exprN` refers to a valid PHP expression which evaluates to a boolean value indicating whether the corresponding message should be returned. Only the message corresponding to the first expression that evaluates to true will be returned. An expression can contain a special variable named `n` (note, it is not `$n`) which will take the number value passed as the first message parameter. For example, assuming a translated message is:

```
'n==1#one book|n>1#many books'
```

and we are passing a number value 2 in the message parameter array when calling Yii::t(), we would obtain `many books` as the final translated message.

As a shortcut notation, if an expression is a number, it will be treated as `n==Number`. Therefore, the above translated message can be also be written as:

```
'1#one book|n>1#many books'
```

File Translation

File translation is accomplished by calling CApplication::findLocalizedFile(). Given the path of a file to be translated, the method will look for a file with the same name under the `LocaleID` subdirectory. If found, the file path will be returned; otherwise, the original file path will be returned.

File translation is mainly used when rendering a view. When calling one of the render methods in a controller or widget, the view files will be translated automatically. For example, if the target language is `zh_cn` while the source language is `en_us`, rendering a view named `edit` would resulting in searching for the view file `protected/views/ControllerID/zh_cn/edit.php`. If the file is found, this translated version will be used for rendering; otherwise, the file `protected/views/ControllerID/edit.php` will be rendered instead.

File translation may also be used for other purposes, for example, displaying a translated image or loading a locale-dependent data file.

Date and Time Formatting

Date and time are often in different formats in different countries or regions. The task of date and time formatting is thus to generate a date or time string that fits for the specified locale. Yii provides CDateFormatter for this purpose.

Each CDateFormatter instance is associated with a target locale. To get the formatter associated with the target locale of the whole application, we can simply access the dateFormatter property of the application.

The CDateFormatter class mainly provides two methods to format a UNIX timestamp.

- format: this method formats the given UNIX timestamp into a string according to a customized pattern (e.g. `$dateFormatter->format('yyyy-MM-dd',$timestamp)`).
- formatDateTime: this method formats the given UNIX timestamp into a string according to a pattern predefined in the target locale data (e.g. `short` format of date, `long` format of time).

Number Formatting

Like data and time, numbers may also be formatted differently in different countries or regions. Number formatting includes decimal formatting, currency formatting and percentage formatting. Yii provides CNumberFormatter for these tasks.

To get the number formatter associated with the target locale of the whole application, we can access the numberFormatter property of the application.

The following methods are provided by CNumberFormatter to format an integer or double value.

- format: this method formats the given number into a string according to a customized pattern (e.g. `$numberFormatter->format('#,##0.00',$number)`).
- formatDecimal: this method formats the given number using the decimal pattern predefined in the target locale data.
- formatCurrency: this method formats the given number and currency code using the currency pattern predefined in the target locale data.
- formatPercentage: this method formats the given number using the percentage pattern predefined in the target locale data.

Using Alternative Template Syntax

Yii allows developers to use their own favorite template syntax (e.g. Prado, Smarty) to write controller or widget views. This is achieved by writing and installing a viewRenderer application component. The view renderer intercepts the invocations of CBaseController::renderFile, compiles the view file with customized template syntax, and renders the compiling results.

Info: It is recommended to use customized template syntax only when writing views that are less likely to be reused. Otherwise, people who are reusing the views would be forced to use the same customized template syntax in their applications.

In the following, we introduce how to use CPradoViewRenderer, a view renderer that allows developers to use the template syntax similar to that in Prado framework. For people who want to develop their own view renderers, CPradoViewRenderer is a good reference.

Using CPradoViewRenderer

To use CPradoViewRenderer, we just need to configure the application as follows:

```
return array(
    'components'=>array(
        ......,
        'viewRenderer'=>array(
            'class'=>'CPradoViewRenderer',
        ),
    ),
);
```

By default, CPradoViewRenderer will compile source view files and save the resulting PHP files under the runtime directory. Only when the source view files are changed, will the PHP files be re-generated. Therefore, using CPradoViewRenderer incurs very little performance degradation.

Tip: While CPradoViewRenderer mainly introduces some new template tags to make writing views easier and faster, you can still write PHP code as usual in the source views.

In the following, we introduce the template tags that are supported by CPradoViewRenderer.

Short PHP Tags

Short PHP tags are shortcuts to writing PHP expressions and statements in a view. The expression tag `<%= expression %>` is translated into `<?php echo expression ?>`; while the statement tag `<% statement %>` to `<?php statement ?>`. For example,

```
<%= CHtml::textField($name,'value'); %>

<% foreach($models as $model): %>
```

is translated into

```
<?php echo CHtml::textField($name,'value'); ?>

<?php foreach($models as $model): ?>
```

Component Tags

Component tags are used to insert a widget in a view. It uses the following syntax:

```
<com:WidgetClass property1=value1 property2=value2 ...>
    // body content for the widget
</com:WidgetClass>


// a widget without body content
<com:WidgetClass property1=value1 property2=value2 .../>
```

where `WidgetClass` specifies the widget class name or class path alias, and property initial values can be either quoted strings or PHP expressions enclosed within a pair of curly brackets. For example,

```
<com:CCaptcha captchaAction="captcha" showRefreshButton={false} />
```

would be translated as

```
<?php $this->widget('CCaptcha', array(
    'captchaAction'=>'captcha',
    'showRefreshButton'=>false)); ?>
```

Note: The value for `showRefreshButton` is specified as `{false}` instead of `"false"` because the latter means a string instead of a boolean.

Cache Tags

Cache tags are shortcuts to using fragment caching. Its syntax is as follows,

```
<cache:fragmentID property1=value1 property2=value2 ...>
    // content being cached
</cache:fragmentID >
```

where `fragmentID` should be an identifier that uniquely identifies the content being cached, and the property-value pairs are used to configure the fragment cache. For example,

```
<cache:profile duration={3600}>
    // user profile information here
</cache:profile >
```

would be translated as

```
<?php if($this->cache('profile', array('duration'=>3600))): ?>
    // user profile information here
<?php $this->endCache(); endif; ?>
```

Clip Tags

Like cache tags, clip tags are shortcuts to calling CBaseController::beginClip and CBaseController::endClip in a view. The syntax is as follows,

```
<clip:clipID>
    // content for this clip
</clip:clipID >
```

where `clipID` is an identifier that uniquely identifies the clip content. The clip tags will be translated as

```
<?php $this->beginClip('clipID'); ?>

    // content for this clip

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

Comment Tags

Comment tags are used to write view comments that should only be visible to developers. Comment tags will be stripped off when the view is displayed to end users. The syntax for comment tags is as follows,

```
<!---

view comments that will be stripped off

--->
```

Console Applications

Console applications are mainly used by a Web application to perform offline work, such as code generation, search index compiling, email sending, etc. Yii provides a framework for writing console applications in an object-oriented and systematic way.

Yii represents each console task in terms of a command, and a console application instance is used to dispatch a command line request to an appropriate command. The application instance is created in an entry script. To execute a console task, we simply run the corresponding command on the command line as follows,

```
php entryScript.php CommandName Param0 Param1 ...
```

where `CommandName` refers to the command name which is case-insensitive, and `Param0`, `Param1` and so on are parameters to be passed to the command instance.

The entry script for a console application is usually written like the following, similar to that in a Web application,

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

// include Yii bootstrap file

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

// create application instance and run

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

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

We then create command classes which should extend from CConsoleCommand. Each command class should be named as its command name appended with Command. For example, to define an email command, we should write an EmailCommand class. All command class files should be placed under the commands subdirectory of the application base directory.

Tip: By configuring CConsoleApplication::commandMap, one can also have command classes in different naming conventions and located in different directories.

Writing a command class mainly involves implementing the CConsoleCommand::run method. Command line parameters are passed as an array to this method. Below is an example:

```
class EmailCommand extends CConsoleCommand
{
    public function run($args)
    {
        $receiver=$args[0];
        // send email to $receiver
    }
}
```

At any time in a command, we can access the console application instance via Yii::app(). Like a Web application instance, console application can also be configured. For example, we can configure a db application component to access the database. The configuration is usually specified as a PHP file and passed to the constructor of the console application class (or createConsoleApplication in the entry script).

Using the yiic Tool

We have used the yiic tool to create our first application. The yiic tool is in fact implemented as a console application whose entry script file is framework/yiic.php. Using yiic, we can accomplish tasks such as creating a Web application skeleton, generating a controller class or model class, generating code needed by CRUD operations, extracting messages to be translated, etc.

We can enhance yiic by adding our own customized commands. To do so, we should start with a skeleton application created using yiic webapp command, as described in Creating First Yii Application. The yiic webapp command will generate two files under the protected directory: yiic and yiic.bat. They are the *local* version of the yiic tool created specifically for the Web application.

We can then create our own commands under the protected/commands directory. Running the local yiic tool, we will see that our own commands appearing together with the standard ones. We can also create our own

commands to be used when `yiic shell` is used. To do so, just drop our command class files under the `protected/commands/shell` directory.

安全措施 (Security)

跨站脚本攻击的防范

跨站脚本攻击(简称 XSS)，即 web 应用从用户收集用户数据。攻击者常常向易受攻击的 web 应用注入 JavaScript，VBScript，ActiveX，HTML 或 Flash 来迷惑访问者以收集访问者的信息。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，你可以对内容中的 HTML 进行转义处理。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 类。它可以将通过有效的审查、安全和白名单功能来把所审核的内容中的所有的恶意代码清除掉，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，必须谨记一条：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，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(
            '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 seesion 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。
- 在 cookie 有变动的时候验证 cookie 的内容。

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,
        ),
    ),
);
```

一定要使用经过 Yii 验证过的 cookie 数据。使用 Yii 内置的 cookies 组件来进行 cookie 操作，不要使用$_COOKIES。

```
// 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;
```

Performance Tuning

Performance of Web applications is affected by many factors. Database access, file system operations, network bandwidth are all potential affecting factors. Yii has tried in every aspect to reduce the performance impact caused by the framework. But still, there are many places in the user application that can be improved to boost performance.

Enabling APC Extension

Enabling the PHP APC extension is perhaps the easiest way to improve the overall performance of an application. The extension caches and optimizes PHP intermediate code and avoids the time spent in parsing PHP scripts for every incoming request.

Disabling Debug Mode

Disabling debug mode is another easy way to improve performance. An Yii application runs in debug mode if the constant `YII_DEBUG` is defined as true. Debug mode is useful during development stage, but it would impact performance because some components cause extra burden in debug mode. For example, the message logger may record additional debug information for every message being logged.

Using `yiilite.php`

When the PHP APC extension is enabled, we can replace `yii.php` with a different Yii bootstrap file named `yiilite.php` to further boost the performance of an Yii-powered application.

The file `yiilite.php` comes with every Yii release. It is the result of merging some commonly used Yii class files. Both comments and trace statements are stripped from the merged file. Therefore, using `yiilite.php` would reduce the number of files being included and avoid execution of trace statements.

Note, using `yiilite.php` without APC may actually reduce performance, because `yiilite.php` contains some classes that are not necessarily used in every request and would take extra parsing time. It is also observed that using `yiilite.php` is slower with some server configurations, even when APC is turned on. The best way to judge whether to use `yiilite.php` or not is to run a benchmark using the included `hello world` demo.

Using Caching Techniques

As described in the Caching section, Yii provides several caching solutions that may improve the performance of a Web application significantly. If the generation of some data takes long time, we can use the data caching approach to reduce the data generation frequency; If a portion of page remains relatively static, we can use the fragment caching approach to reduce its rendering frequency; If a whole page remains relative static, we can use the page caching approach to save the rendering cost for the whole page.

If the application is using Active Record, we should turn on the schema caching to save the time of parsing database schema. This can be done by configuring the CDbConnection::schemaCachingDuration property to be a value greater than 0.

Besides these application-level caching techniques, we can also use server-level caching solutions to boost the application performance. As a matter of fact, the APC caching we described earlier belongs to this category. There are other server techniques, such as Zend Optimizer, eAccelerator, Squid, to name a few.

Database Optimization

Fetching data from database is often the main performance bottleneck in a Web application. Although using caching may alleviate the performance hit, it does not fully solve the problem. When the database contains enormous data and the cached data is invalid, fetching the latest data could be prohibitively expensive without proper database and query design.

Design index wisely in a database. Indexing can make `SELECT` queries much faster, but it may slow down `INSERT`, `UPDATE` or `DELETE` queries.

For complex queries, it is recommended to create a database view for it instead of issuing the queries inside the PHP code and asking DBMS to parse them repetitively.

Do not overuse Active Record. Although Active Record is good at modelling data in an OOP fashion, it actually degrades performance due to the fact that it needs to create one or several objects to represent each row of query result. For data intensive applications, using DAO or database APIs at lower level could be a better choice.

Last but not least, use `LIMIT` in your `SELECT` queries. This avoids fetching overwhelming data from database and exhausting the memory allocated to PHP.

Minimizing Script Files

Complex pages often need to include many external JavaScript and CSS files. Because each file would cause one extra round trip to the server and back, we should minimize the number of script files by merging them into fewer ones. We should also consider reducing the size of each script file to reduce the network transmission time. There are many tools around to help on these two aspects.

For a page generated by Yii, chances are that some script files are rendered by components that we do not want to modify (e.g. Yii core components, third-party components). In order to minimizing these script files, we need two steps.

Note: The `scriptMap` feature described in the following has been available since version 1.0.3.

First, we declare the scripts to be minimized by configuring the scriptMap property of the clientScript application component. This can be done either in the application configuration or in code. For example,

```
$cs=Yii::app()->clientScript;
$cs->scriptMap=array(
    'jquery.js'=>'/js/all.js',
    'jquery.ajaxqueue.js'=>'/js/all.js',
    'jquery.metadata.js'=>'/js/all.js',
    ......
);
```

What the above code does is that it maps those JavaScript files to the URL `/js/all.js`. If any of these JavaScript files need to be included by some components, Yii will include the URL (once) instead of the individual script files.

Second, we need to use some tools to merge (and perhaps compress) the JavaScript files into a single one and save it as `js/all.js`.

The same trick also applies to CSS files.

We can also improve page loading speed with the help of Google AJAX Libraries API. For example, we can include `jquery.js` from Google servers instead of our own server. To do so, we first configure the `scriptMap` as follows,

```
$cs=Yii::app()->clientScript;
$cs->scriptMap=array(
    'jquery.js'=>false,
    'jquery.ajaxqueue.js'=>false,
    'jquery.metadata.js'=>false,
    ......
);
```

By mapping these script files to false, we prevent Yii from generating the code to include these files. Instead, we write the following code in our pages to explicitly include the script files from Google,

```
<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>
```