使用Yii来建立博客

此教程描述了使用Yii开发一个演示博客的过程。此博客同时可以在Yii发布包里找到。它详细讲解了开发中的每个步骤，这些步骤可能同样适用于其他Web应用的开发。作为对Yii指南和类参考 的补充, 此教程旨在展示Yii的实际使用，而不是详尽具体的使用说明。

读者不需要预先了解关于Yii的知识，但具备基本的面向对象编程（OOP）和数据库编程知识会使读者更容易理解此教程。

Yii 之初体验

在这一部分里，我们将讲解怎样建立一个程序的骨架作为着手点。为简单起见，我们假设Web服务器根目录是 /wwwroot，相应的URL是 http://www.example.com/。

安装Yii

首先，我们来安装Yii框架。从 www.yiiframework.com 获取一份Yii的拷贝，解压缩到 /wwwroot/yii。再次检查以确保 /wwwroot/yii/framework 目录存在。

提示: Yii框架可以安装在文件系统的任何地方，而不是必须在Web目录中。它的 framework 目录包含了框架的代码，这也是部署Yii应用时唯一一个必要的目录。一个单独的Yii安装可以被用于多个Yii应用。

Yii安装完毕之后，打开浏览器访问URL http://www.example.com/yii/requirements/index.php。它将显示Yii提供的需求检查程序。对我们的Blog应用来说，除了Yii所需的最小需求之外，我们还需要启用 pdo 和 pdo_sqlite 这两个PHP 扩展。这样我们才能访问SQLite数据库。

创建应用骨架

然后，我们使用 yiic 工具在 /wwwroot/blog 目录下创建一个应用骨架。yiic 工具是在Yii发布包中提供的命令行工具。它可以用于创建代码以减少某些重复的编码工作。

打开一个命令行窗口，执行以下命令：

```
% /wwwroot/yii/framework/yiic webapp /wwwroot/blog
Create a Web application under '/wwwroot/blog'? [Yes|No]y
......
```

提示: 为了使用上面提到的 yiic 工具，CLI PHP 程序必须在命令搜索路径内（译者注：即 php.exe 所在的目录必须在PATH环境变量中），否则，可能要使用下面的命令:

```
path/to/php /wwwroot/yii/framework/yiic.php webapp /wwwroot/blog
```

要查看我们刚创建的应用，打开浏览器访问 URL http://www.example.com/blog/index.php。可以看到我们的程序骨架已经有了四个具备完整功能的页面：首页（Home），“关于”页（About），联系页（Contact）和登录页（Login）。

接下来，我们简单介绍一下在这个程序骨架中的内容。

入口脚本

我们有一个入口脚本 文件 /wwwroot/blog/index.php，内容如下：

```
<?php
$yii='/wwwroot/framework/yii.php';
$config=dirname(__FILE__).'/protected/config/main.php';
```

```
// remove the following line when in production mode
defined('YII_DEBUG') or define('YII_DEBUG',true);

require_once($yii);
Yii::createWebApplication($config)->run();
```

这是唯一一个网站用户可以直接访问的脚本。此脚本首先包含了Yii的引导文件 yii.php。然后它按照指定的配置创建了一个应用 实例并执行此应用。

基础应用目录

我们还有一个 应用基础目录 /wwwroot/blog/protected。我们主要的代码和数据将放在此目录下，它应该被保护起来，防止网站访客的直接访问。针对 Apache httpd 网站服务器，我们在此目录下放了一个 .htaccess 文件，其内容如下：

```
deny from all
```

对于其他的网站服务器，请参考相应的关于保护目录以防止被访客直接访问的相关文档。

应用的工作流程

为了帮你理解Yii是怎样工作的，对于我们的程序骨架，当有人访问它的联系页（Contact）时，我们对它的工作流程描述如下：

1. 用户请求此 URL http://www.example.com/blog/index.php?r=site/contact；
2. 入口脚本 被网站服务器执行以处理此请求；
3. 一个 应用 的实例被创建，其配置参数为/wwwroot/blog/protected/config/main.php 应用配置文件中指定的初始值；
4. 应用分派此请求到一个 控制器（Controller） 和一个 控制器动作（Controller action）。对于联系页（Contact）的请求，它分派到了 site 控制器和 contact 动作 (即 /wwwroot/blog/protected/controllers/SiteController.php 中的 actionContact 方法);
5. 应用按 SiteController 实例创建了 site 控制器并执行；
6. SiteController 实例通过调用它的 actionContact() 方法执行 contact 动作；
7. actionContact 方法为用户渲染一个名为 contact 的 视图（View） 。在程序内部，这是通过包含一个视图文件 /wwwroot/blog/protected/views/site/contact.php 并将结果插入 布局 文件 /wwwroot/blog/protected/views/layouts/column1.php 实现的。

需求分析

我们要开发的博客系统是一个单用户系统。系统的所有者可以执行以下操作：

- 登录和退出
- 创建，更新，删除日志
- 发布，撤销发布，存档日志
- 审核和删除评论

其他的访客则可以执行以下操作：

- 阅读日志
- 创建评论

此系统的额外需求包括：

- 系统的首页应显示最新的帖子列表。
- 如果页面中有超过10篇日志，应该以分页的方式显示。
- 系统应该在显示日志的同时显示此日志的评论。
- 系统应可以按照指定的Tag列出相应的日志。
- 系统应展示一个可以表明标签使用频率的标签云。
- 系统应展示一个最新评论列表。
- 系统应可以更换主题。
- 系统应使用 SEO 友好的 URL 。

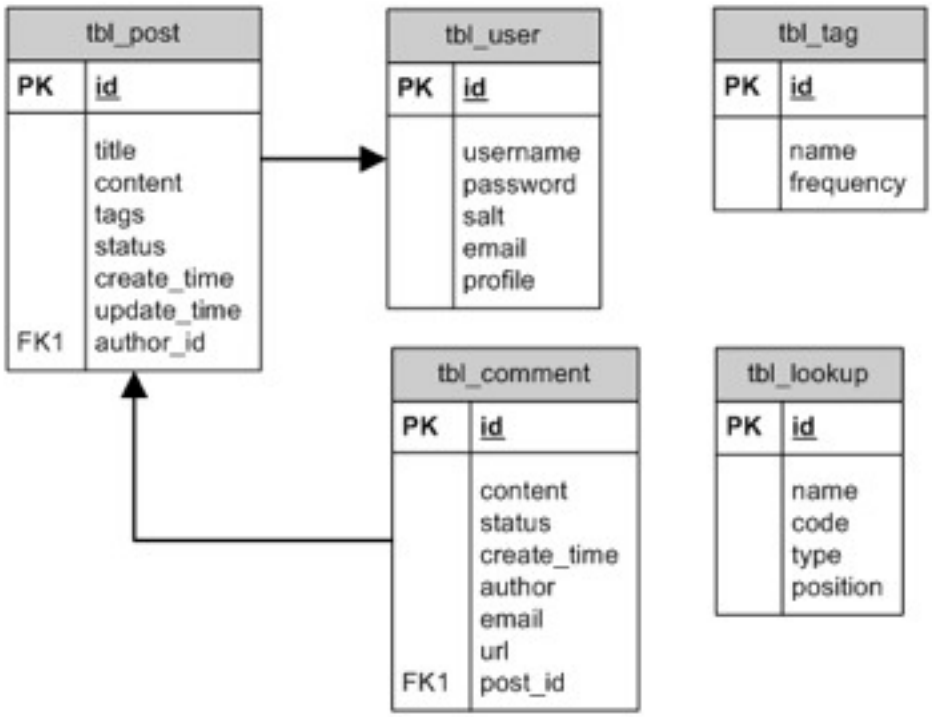
整体设计

基于需求分析，我们决定为我们的博客应用使用如下数据表存储持久数据：

- tbl_user 存储用户信息，包括用户名和密码。
- tbl_post 存储博客日志信息。它由如下几列组成：
  - title: 必填项，日志的标题；
  - content: 必填项，日志的内容，使用 Markdown 格式;
  - status: 必填项，日志的状态，可以是以下值之一：
    - 1, 表示日志在草稿箱里，对外不可见；
    - 2, 表示日志已经对外发布；
    - 3, 表示日志已经过期，在日志列表中不可见（但仍然可以单独访问）。
  - tags: 可选项，用于对日志归类的一个以逗号分隔的词语列表。
- tbl_comment 存储日志评论信息，每条评论关联于一篇日志，主要包含如下几列：
  - name: 必填项，作者名字；
  - email: 必填项，作者 Email；
  - website: 可选项, 作者网站的 URL;
  - content: 必填项，纯文本格式的评论内容；
  - status: 必填项，评论状态，用于表示日志是(2)否(1)已通过审核。
- tbl_tag 存储日志Tag使用频率信息，用于实现标签云效果。此表主要包含以下几列：
  - name: 必填项，唯一的Tag名字;
  - frequency: 必填项，Tag出现在日志中的次数。
- tbl_lookup 存储通用查找信息。它本质上是一个整型数字和文本字符的映射。前者是我们的代码中的数据表现，后者是相应的对最终用户的表现。例如，我们使用整数1表示草稿日志，而使用字符串“草稿”把此状态显示给最终用户。此表主要包含以下几列：
  - name: 数据项的文本表现形式，用于显示给最终用户；
  - code: 数据项的整数表现形式；
  - type: 数据项的类型；
  - position: 同类数据项中，数据项相对于其他数据项的显示顺序。

如下的实体-关系（ER）图展示了上述表的表结构和他们之间的关系。

博客数据库实体-关系图



上述ER图相应的完整SQL语句可以在 博客演示 中找到。在我们的安装包中， 它们位于 /wwwroot/yii/demos/blog/protected/data/schema.sqlite.sql。

信息: 我们对所有表和列的命名使用了小写字母。这是因为不同的 DBMS 通常有不同的大小写敏感处理方式， 我们通过这种方式来避免这种问题。

我们同时对所有的表使用了 tbl_ 前缀。这出于两个目的。第一， 此前缀对这些表提供了一个命名空间以使他们和同一数据库中的其他表共存， 此情况常出现在在共享的主机环境中， 一个数据库常被多个应用使用。第二， 使用表前缀减少了表名中出现DBMS保留字的可能。

我们把博客应用的开发划分为如下几个阶段：

- 阶段 1: 创建一个博客系统的原型。它应该包括大多数所需的功能。
- 阶段 2: 完善日志管理功能。包括日志的创建， 日志列表， 日志显示， 日志更新和删除。
- 阶段 3: 完善评论管理功能。包括评论创建， 评论列表， 审核， 更新以及日志评论的删除。
- 阶段 4: 实现 Portlets。它包括用户菜单， 登录， 标签云和最新评论Portlets。
- 阶段 5: 最终调试和部署。

建立数据库

完成了程序骨架和数据库设计， 在这一节里我们将创建博客的数据库并将其连接到程序骨架中。

创建数据库

我们选择创建一个SQLite数据库。由于Yii中的数据库支持是建立在 PDO 之上的， 我们可以很容易地切换到一个不同的 DBMS (例如 MySQL, PostgreSQL) 而不需要修改我们的应用代码。

我们把数据库文件 blog.db 建立在 /wwwroot/blog/protected/data 中。注意， 数据库文件和其所在的目录都必须对Web服务器进程可写， 这是SQLite的要求。我们可以简单的从博客演示中复制这个数据库文件， 它

位于 /wwwroot/yii/demos/blog/protected/data/blog.db。我们也可以通过执行 /wwwroot/yii/demos/blog/protected/data/schema.sqlite.sql 文件中的SQL语句自己创建这个数据库。

提示: 要执行SQL语句，我们可以使用 sqlite3 命令行工具。它可以在 SQLite 官方网站 中找到。

建立数据库连接

要在我们创建的程序骨架中使用这个数据库，我们需要修改它的应用配置，它保存在PHP脚本 /wwwroot/blog/protected/config/main.php 中。此脚本返回一个包含键值对的关联数组，它们中的每一项被用来初始化应用实例 中的可写属性。

我们按如下方式配置 db 组件，

```
return array(
    'components'=>array(
        'db'=>array(
            'connectionString'=>'sqlite:/wwwroot/blog/protected/data/blog.db',
            'tablePrefix'=>'tbl_',
        ),),);
```

上述配置的意思是说我们有一个 db 应用组件，它的 connectionString 属性应当以 sqlite:/wwwroot/blog/protected/data/blog.db 这个值初始化，它的 tablePrefix 属性应该是 tbl_。

通过这个配置，我们就可以在代码的任意位置使用 Yii::app()->db 来访问数据库连接对象了。注意，Yii::app() 会返回我们在入口脚本中创建的应用实例。如果你对数据库连接的其他可用的方法和属性感兴趣，可以阅读 类参考。然而，在多数情况下，我们并不会直接使用这个数据库连接。而是使用被称为 ActiveRecord 的东西来访问数据库。

我们想对配置中的 tablePrefix 属性再解释一点。此属性告诉 db 连接它应该关注我们使用了 tbl_ 作为数据库表前缀。具体来说，如果一条SQL语句中含有一个被双大括号括起来的标记 (例如 {{post}})，那么 db 连接应该在把它提交给DBMS执行前，先将其翻译成带有表前缀的名字 (例如 tbl_post) 。这个特性非常有用，如果将来我们需要修改表前缀，就不需要再动代码了。例如，如果我们正在开发一个通用内容管理系统 (CMS)，我们就可以利用此特性，这样当它被安装在一个不同的环境中时，我们就能允许用户选择一个他们喜欢的表前缀。

提示: 如果你想使用MySQL而不是SQLite来存储数据，你可以使用位于 /wwwroot/yii/demos/blog/protected/data/schema.mysql.sql 文件 中的SQL语句创建一个名为 blog 的 MySQL 数据库。然后，按如下方式 修改应用配置，

```
return array(
    'components'=>array(
        'db'=>array(
            'connectionString' => 'mysql:host=localhost;dbname=blog',
            'emulatePrepare' => true,
            'username' => 'root',
            'password' => '',
            'charset' => 'utf8',
            'tablePrefix' => 'tbl_',
        ),),);
```

脚手架

创建，读取，更新，删除 (CRUD) 是应用的数据对象中的四个基本操作。由于在Web应用的开发中实现CURD的任务非常常见，Yii 为我们提供了一些可以使这些过程自动化的代码生成工具，名为 *Gii* （也被称为 脚手架） 。

注意: Gii 从 Yii 1.1.2 版开始提供。在这之前，你可能需要使用 yiic shell tool 来实现相同的任务。

下面，我们将阐述如何使用这个工具来实现博客应用中的CRUD操作。

安装 Gii

首先我们需要安装 Gii. 打开文件 /wwwroot/blog/protected/config/main.php ， 添加如下代码:

```
return array(
    'import'=>array(
        'application.models.*',
        'application.components.*',
    ),
    'modules'=>array(
        'gii'=>array(
            'class'=>'system.gii.GiiModule',
            'password'=>'这儿设置一个密码',
        ),),);
```

上面的代码安装了一个名为 gii 的模块， 这样我们就可以通过在浏览器中浏览如下URL来访问 Gii 模块:

```
http://www.example.com/blog/index.php?r=gii
```

我们将被提示要求输入一个密码。输入我们前面在 /wwwroot/blog/protected/config/main.php 中设置的密码， 我们将看到一个页面， 它列出了所有可用的代码生成工具。

注意: 上述代码在生产环境中应当移除。代码生成工具只应当用于开发环境。

创建模型

首先我们需要为每个数据表创建一个模型（Model） 类。模型类会使我们可以通过一种直观的、面向对象的风格访问数据库。稍后我们将会看到这一点。

点击 Model Generator 链接开始使用模型创建工具。

在 Model Generator 页中， 在Table Name一栏输入 tbl_user (用户表的名字)， 然后按下 Preview 按钮。一个预览表将显示在我们面前。我们可以点击表格中的链接来预览要生成的代码。如果一切OK， 我们可以按下 Generate 按钮来生成代码并将其保存在一个文件中。

信息: 由于代码生成器需要保存生成的代码到文件， 它要求Web服务器进程必须拥有对相应文件的创建和修改权限。为简单起见， 我们可以赋予Web服务器进程对整个 /wwwroot/blog 目录的写权限。注意这只在开发机器上使用 Gii 时会用到。

对剩余的其他表重复同样的步骤， 包括 tbl_post, tbl_comment, tbl_tag 和 tbl_lookup。

提示: 我们还可以在 Table Name 栏中输入一个星号 '*' 。这样就可以通过一次点击就对 所有的 数据表生成相应的模型类。

通过这一步， 我们就有了如下新创建的文件：

- models/User.php 包含了继承自 CActiveRecord 的 User 类， 可用于访问 tbl_user 数据表；
- models/Post.php 包含了继承自 CActiveRecord 的 Post 类， 可用于访问 tbl_post 数据表；
- models/Tag.php 包含了继承自 CActiveRecord 的 Tag 类， 可用于访问 tbl_tag 数据表；
- models/Comment.php 包含了继承自 CActiveRecord 的 Comment 类， 可用于访问 tbl_comment 数据表；
- models/Lookup.php 包含了继承自 CActiveRecord 的 Lookup 类， 可用于访问 tbl_lookup 数据表；

实现 CRUD 操作

模型类建好之后， 我们就可以使用 Crud Generator 来创建为这些模型实现CRUD操作的代码了。我们将对 Post 和 Comment 模型执行此操作。

在 Crud Generator 页面中，Model Class 一栏输入 Post (就是我们刚创建的 Post 模型的名字)，然后按下 Preview 按钮。我们会看到有很多文件将被创建。按下 Generate 按钮来创建它们。

对 Comment 模型重复同样的步骤。

让我们看一下通过CRUD生成器生成的这些文件。所有的文件都创建在了 /wwwroot/blog/protected 目录中。为方便起见，我们把它们分组为 控制器（Controller） 文件和 视图（View） 文件：

- 控制器文件：
  - controllers/PostController.php 包含负责所有CRUD操作的 PostController 控制器类；
  - controllers/CommentController.php 包含负责所有CRUD操作的 CommentController 控制器类；
- 视图文件：
  - views/post/create.php 一个视图文件，用于显示创建新日志的 HTML 表单；
  - views/post/update.php 一个视图文件，用于显示更新日志的 HTML 表单；
  - views/post/view.php 一个视图文件，用于显示一篇日志的详情；
  - views/post/index.php 一个视图文件，用于显示日志列表；
  - views/post/admin.php 一个视图文件，用于在一个带有管理员命令的表格中显示日志；
  - views/post/_form.php 一个插入 views/post/create.php 和 views/post/update.php 的局部视图文件。它显示用于收集日志信息的HTML表单；
  - views/post/_view.php 一个在 views/post/index.php 中使用的局部视图文件。它显示单篇日志的摘要信息；
  - views/post/_search.php 一个在 views/post/admin.php 中使用的局部视图文件。它显示一个搜索表单；
  - 还有为评论创建的一系列相似的文件。

测试

我们可以通过访问如下网址测试我们刚生成的代码所实现的功能：

```
http://www.example.com/blog/index.php?r=post

http://www.example.com/blog/index.php?r=comment
```

注意，由代码生成器实现的日志和评论功能是完全相互独立的。并且，当创建一个新的日志或评论时，我们必须输入如 author_id 和 create_time 这些信息，而在现实应用中这些应当由程序自动设置。别担心。我们将在下一个阶段中修正这些问题。现在呢，这个模型已经包含了大多数我们需要在博客应用中实现的功能，我们应该对此感到满意了 ^_^。

为了更好地理解这些文件是如何使用的，我们在下面列出了当显示一个日志列表时发生的工作流程。

1. 用户请求访问这个 URL http://www.example.com/blog/index.php?r=post;
2. 入口脚本 被Web服务器执行，它创建并实例化了一个 应用 实例来处理此请求；
3. 应用创建并执行了 PostController 实例；
4. PostController 实例通过调用它的 actionIndex() 方法执行了 index 动作。注意，如果用户没有在URL中指定执行一个动作，则 index 就是默认的动作；
5. actionIndex() 方法查询数据库，带回最新的日志列表；

6.actionIndex() 方法使用日志数据渲染 index 视图。

用户验证

我们的博客应用需要区分系统所有者和来宾用户。因此，我们需要实现 用户验证 功能。

或许你已经发现了，我们的程序骨架已经提供了用户验证功能，它会判断用户名和密码是不是都为 demo 或 admin。在这一节里，我们将修改这些代码，以使身份验证通过 User 数据表实现。

用户验证在一个实现了 IUserIdentity 接口的类中进行。此程序骨架通过 UserIdentity 类实现此目的。此类存储在 /wwwroot/blog/protected/components/UserIdentity.php 文件中。

提示: 按照约定，类文件的名字必须是相应的类名加上 .php 后缀。遵循此约定，就可以通过一个路径别名（path alias） 指向此类。例如，我们可以通过别名 application.components.UserIdentity 指向 UserIdentity 类。Yii 的许多API都可以识别路径别名 (例如 Yii::createComponent())，使用路径别名可以避免在代码中插入文件的绝对路径。绝对路径的存在往往会导致在部署应用时遇到麻烦。

我们将 UserIdentity 类做如下修改，

```
<?php
class UserIdentity extends CUserIdentity
{
    private $_id;
    public function authenticate()
    {
        $username=strtolower($this->username);
        $user=User::model()->find('LOWER(username)=?',array($username));
        if($user===null)
            $this->errorCode=self::ERROR_USERNAME_INVALID;
        else if(!$user->validatePassword($this->password))//! is here mean true to false, f to t
            $this->errorCode=self::ERROR_PASSWORD_INVALID;
        else
        {
            $this->_id=$user->id;
            $this->username=$user->username;
            $this->errorCode=self::ERROR_NONE;
        }
        return $this->errorCode==self::ERROR_NONE;
    }

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

在 authenticate() 方法中，我们使用 User 类来查询 tbl_user 表中 username 列值（不区分大小写）和提供的用户名一致的一行，请记住 User 类是在前面的章节中通过 gii 工具创建的。由于 User 类继承自 CActiveRecord ，我们可以利用 ActiveRecord 功能 以 OOP 的风格访问 tbl_user 表。

为了检查用户是否输入了一个有效的密码，我们调用了 User 类的 validatePassword 方法。我们需要按下面的代码修改 /wwwroot/blog/protected/models/User.php 文件。注意，我们在数据库中存储了密码的加密串和随机生成的SALT密钥，而不是存储明文密码。 所以当要验证用户输入的密码时，我们应该和加密结果做对比。

```
class User extends CActiveRecord
{
    ......
    public function validatePassword($password)
    {
```

```
        return $this->hashPassword($password,$this->salt)===$this->password;
    }

    public function hashPassword($password,$salt)
    {
        return md5($salt.$password);
    }
}
```

在 UserIdentity 类中，我们还覆盖（Override，又称为重写）了 getId() 方法，它会返回在 User 表中找到的用户的 id。父类 (CUserIdentity) 则会返回用户名。username 和 id 属性都将存储在用户 SESSION 中，可在代码的任何部分通过 Yii::app()->user 访问。

提示: 在 UserIdentity 类中，我们没有显式包含(include)相应的类文件就访问了 CUserIdentity 类，这是因为 CUserIdentity 是一个由Yii框架提供的核心类。Yii 将会在任何核心类被首次使用时自动包含类文件。

我们也对 User 类做了同样的事情。这是因为 User 类文件被放在了 /wwwroot/blog/protected/models 目录，此目录已经通过应用配置中的如下几行代码被添加到了 PHP 的 include_path 中：

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

上面的配置说明，位于 /wwwroot/blog/protected/models 或 /wwwroot/blog/protected/components 目录中的任何类将在第一次使用时被自动包含。

UserIdentity 类主要用于 LoginForm 类中，它基于用户名和从登录页中收到的密码来实现用户验证。下面的代码展示了 UserIdentity 的使用：

```
$identity=new UserIdentity($username,$password);
$identity->authenticate();
switch($identity->errorCode)
{
    case UserIdentity::ERROR_NONE:
        Yii::app()->user->login($identity);
        break;
}
```

信息: 人们经常对 identity 和 user 应用组件感到困惑，前者代表的是一种验证方法，后者代表当前用户相关的信息。一个应用只能有一个 user 组件，但它可以有一个或多个 identity 类，这取决于它支持什么样的验证方法。一旦验证通过，identity 实例会把它自己的状态信息传递给 user 组件，这样它们就可以通过 user 实现全局可访问。

要测试修改过的 UserIdentity 类，我们可以浏览 URL http://www.example.com/blog/index.php ，然后尝试使用存储在 User 表中的用户名和密码登录。如果我们使用了 博客演示 中的数据，我们应该可以通过用户名 demo 和密码 demo 登录。注意，此博客系统没有提供用户管理功能。因此，用户无法修改自己的信息或通过Web界面创建一个新的帐号。用户管理功能可以考虑作为以后对此博客应用的增强。

总结

我们已经完成了阶段1。来总结一下目前为止我们所完成的工作：

1.我们确定了完整的需求；

2.我们安装了Yii框架；

3.我们创建了一个程序骨架；

4.我们设计并创建了博客数据库；

5.我们修改了应用配置，添加了数据库连接；

6.我们为日志和评论生成了实现CRUD操作的代码；

7.我们修改了验证方法以实现通过 tbl_user 表验证身份；

对一个新项目来说，大部分时间将花在对程序骨架的第1至4步操作上。

虽然 gii 工具生成的代码可以对数据库实现完整的 CRUD 操作，但它在实际应用中常需要做一些修改。鉴于此，在下面的两个阶段中，我们的工作就是自定义生成的日志及评论的 CRUD 代码，使他们达到我们一开始的需求。

总体来说，我们首先要修改 模型 类，添加适当的 验证 规则并声明 相关的对象。然后我们要为每个CRUD操作修改其 控制器动作 和 视图 代码。

自定义日志模型

由 yiic 工具生成的 Post 日志模型类主要需要做如下两处修改：

- rules() 方法：指定对模型属性的验证规则；
- relations() 方法：指定关联的对象；

信息: 模型 包含了一系列属性，每个属性关联到数据表中相应的列。属性可以在类成员变量中显式定义，也可以隐式定义，不需要事先声明。

自定义 rules() 方法

我们先来指定验证规则，它可以确保用户输入的信息在保存到数据库之前是正确的。例如，Post 的 status 属性应该是 1, 2 或 3 中的任一数字。yiic 工具其实也为每个模型生成了验证规则，但是这些规则是基于数据表的列信息的，可能并不是非常恰当。

基于需求分析，我们把 rules() 做如下修改：

```
public function rules()
{
    return array(
        array('title, content, status', 'required'),
        array('title', 'length', 'max'=>128),
        array('status', 'in', 'range'=>array(1,2,3)),
        array('tags', 'match', 'pattern'=>'/^[\w\s,]+$/',
            'message'=>'Tags can only contain word characters.'),
        array('tags', 'normalizeTags'),

        array('title, status', 'safe', 'on'=>'search'),
);}
```

在上面的代码中，我们指定了 title, content 和 status 属性是必填项；title 的长度不能超过 128；status 属性值应该是 1 (草稿), 2 (已发布) 或 3 (已存档)；tags 属性应只允许使用单词字母和逗号。另外，我们使用 normalizeTags 来规范化用户输入的Tag，使Tag是唯一的且整齐地通过逗号分隔。最后的规则会被搜索功能用到，这个我们后面再讲。

像 required, length, in 和 match 这几个验证器（validator）是Yii提供的内置验证器。normalizeTags 验证器是一个基于方法的验证器，我们需要在 Post 类中定义它。关于如何设置验证规则的更多信息，请参考 指南。

```
public function normalizeTags($attribute,$params)
{
    $this->tags=Tag::array2string(array_unique(Tag::string2array($this->tags)));
}
```

Code :: Tag::array2string() & string2array()

```
public static function string2array($tags)
{
	return preg_split('/\s*,\s*/', trim($tags),-1, PREG_SPLIT_NO_EMPTY);
}

public static function array2string($tags)
{
	return implode(',', $tags);
}
```

rules() 方法中定义的规则会在模型实例调用其 validate() 或 save() 方法时逐一执行。

注意: 请务必记住 rules() 中出现的属性必须是那些通过用户输入的属性。其他的属性，如 Post 模型中的 id 和 create_time，是通过我们的代码或数据库设定的，不应该出现在 rules() 中。详情请参考 属性的安全赋值（Securing Attribute Assignments）.

作出这些修改之后，我们可以再次访问日志创建页检查新的验证规则是否已生效。

自定义 relations() 方法

最后我们来自定义 relations() 方法，以指定与日志相关的对象。通过在 relations() 中声明这些相关对象，我们就可以利用强大的 Relational ActiveRecord (RAR) 功能来访问日志的相关对象，例如它的作者和评论。不需要自己写复杂的 SQL JOIN 语句。

我们自定义 relations() 方法如下：

```
public function relations()
{
    return array(
        'author' => array(self::BELONGS_TO, 'User', 'author_id'),
        'comments' => array(self::HAS_MANY, 'Comment', 'post_id',
            'condition'=>'comments.status='.Comment::STATUS_APPROVED,
            'order'=>'comments.create_time DESC'),
        'commentCount' => array(self::STAT, 'Comment', 'post_id',
            'condition'=>'status='.Comment::STATUS_APPROVED),
    );
}
```

我们还在 Comment 模型类中定义了两个在上面的方法中用到的常量。

```
class Comment extends CActiveRecord
{
    const STATUS_PENDING=1;
    const STATUS_APPROVED=2;
}
```

relations() 中声明的关系表明：

- 一篇日志属于一个作者，它的类是 User，它们的关系建立在日志的 author_id 属性值之上；
- 一篇日志有多个评论，它们的类是 Comment，它们的关系建立在评论的 post_id 属性值之上。这些评论应该按它们的创建时间排列，且评论必须已通过审核；
- commentCount 关系有一点特别，它返回一个关于日志有多少条评论的一个聚合结果。

通过以上的关系声明，我们现在可以按下面的方式很容易的访问日志的作者和评论信息。

```
$author=$post->author;
echo $author->username;
$comments=$post->comments;
foreach($comments as $comment)
    echo $comment->content;
```

关于如何声明和使用关系的更多详情，请参考 指南.

添加 url 属性+基本无效，与后期URL美化一起看。

日志是一份可以通过一个唯一的URL访问的内容。我们可以在 Post 模型中添加一个 url 属性，这样同样的创建URL的代码就可以被复用，而不是在代码中到处调用 CWebApplication::createUrl 。稍后讲解怎样美化 URL 的时候，我们将看到添加这个属性给我们带来了超拽的便利。

要添加 url 属性，我们可以按如下方式给 Post 类添加一个 getter 方法：

```
class Post extends CActiveRecord
{
    public function getUrl()
    {
        return Yii::app()->createUrl('post/view', array(
            'id'=>$this->id,
            'title'=>$this->title,
        ));}}
```

注意我们除了使用日志的ID之外，还添加了日志的标题作为URL中的一个 GET 参数。这主要是为了搜索引擎优化 (SEO) 的目的，在 美化 URL 中将会讲述。

由于 CComponent 是 Post 的最顶级父类，添加 getUrl() 这个 getter 方法使我们可以使用类似 $post->url 这样的表达式。当我们访问 $post->url 时，getter 方法将会被执行，它的返回结果会成为此表达式的值。关于这种组件的更多详情，请参考 指南。

以文本方式显示状态

由于日志的状态在数据库中是以一个整型数字存储的，我们需要提供一个文本话的表现形式，这样在它显示给最终用户时会更加直观。在一个大的系统中，类似的需求是很常见的。

作为一个总体的解决方案，我们使用 tbl_lookup 表存储数字值和被用于其他数据对象的文本值的映射。为了更简单的访问表中的文本数据，我们按如下方式修改 Lookup 模型类：

```
class Lookup extends CActiveRecord
{
    private static $_items=array();

    public static function items($type)
    {
        if(!isset(self::$_items[$type])) //! here!!!!
            self::loadItems($type);
        return self::$_items[$type];
    }

    public static function item($type,$code)
    {
        if(!isset(self::$_items[$type]))//! here !!!!
            self::loadItems($type);
        return isset(self::$_items[$type][$code]) ? self::$_items[$type][$code] : false;
    }

    private static function loadItems($type)
    {
        self::$_items[$type]=array();
        $models=self::model()->findAll(array(
            'condition'=>'type=:type',
            'params'=>array(':type'=>$type),
            'order'=>'position',
        ));
        foreach($models as $model)
            self::$_items[$type][$model->code]=$model->name;
```

```
    }
}
```

我们的新代码主要提供了两个静态方法： Lookup::items() 和 Lookup::item()。前者返回一个属于指定的数据类型的字符串列表，后者按指定的数据类型和数据值返回一个具体的字符串。

我们的博客数据库已经预置了两个查询类别： PostStatus 和 CommentStatus。前者代表可用的日志状态，后者代表评论状态。

为了使我们的代码更加易读，我们还定义了一系列常量，用于表示整数型状态值。我们应该在涉及到相应的状态值时在代码中使用这些常量。

```
class Post extends CActiveRecord
{
    const STATUS_DRAFT=1;
    const STATUS_PUBLISHED=2;
    const STATUS_ARCHIVED=3;
    ......
}
```

这样，我们可以通过调用 Lookup::items('PostStatus') 来获取可用的日志状态列表（按相应的整数值索引的文本字符串），通过调用 Lookup::item('PostStatus', Post::STATUS_PUBLISHED) 来获取发布状态的文本表现形式。

日志的创建与更新

准备好了 Post 模型，我们现在需要调整控制器 PostController 的动作和视图了。在这一节里，我们首先自定义CRUD操作的访问权限控制；然后我们修改代码实现 创建 与 更新 操作。

自定义访问控制

我们想做的第一件事是自定义 访问控制（Access control），因为 yiic 工具生成的代码并不符合我们的需求。

我们将 /wwwroot/blog/protected/controllers/PostController.php 文件中的 accessRules() 方法修改如下：

```
public function accessRules()
{
    return array(
        array('allow',  // allow all users to perform 'list' and 'show' actions
            'actions'=>array('index', 'view'),
            'users'=>array('*'),
        ),
        array('allow', // allow authenticated users to perform any action
            'users'=>array('@'),
        ),
        array('deny',  // deny all users
            'users'=>array('*'),
        ),
    );
}
```

上面的规则说明：所有用户均可访问 index 和 view 动作，已通过身份验证的用户可以访问任意动作，包括 admin 动作。在其他场景中，应禁止用户访问。注意这些规则将会按它们在此列出的顺序计算。第一条匹配当前场景的规则将决定访问权。例如，如果当前用户是系统所有者，他想尝试访问日志创建页，第二条规则将匹配成功并授予此用户权限。

test:: open wwwroot/blog/index.php?r=post/list //open is access when anonymous status
open wwwroot/blog/index.php?r=post/create //open id deny when anonymous status and login form is display, so login as username:demo, password:demo

自定义 创建 和 更新 操作

创建 和 更新 操作非常相似。他们都需要显示一个HTML表单用于收集用户的输入的信息， 然后对其进行验证， 然后将其存入数据库。主要的不同是 更新 操作需要把从数据库找到的已存在的日志数据重现在表单中。鉴于此， yiic 工具创建了一个局部视图 /wwwroot/blog/protected/views/post/_form.php ， 它会插入 创建 和 更新 视图来渲染所需的HTML表单。

我们先修改 _form.php 这个文件， 使这个HTML表单只收集我们想要的输入：title, content, tags 和 status。我们使用文本域收集前三个属性的输入， 还有一个下拉列表用来收集 status 的输入。此下拉列表的选项值就是可用的日志状态文本。

```
<?php echo $form->dropDownList($model,'status',Lookup::items('PostStatus')); ?>
```

在上面的代码中， 我们调用了 Lookup::items('PostStatus') 以带回日志状态列表。

test :: open wwwroot/blog/index.php?r=post/create //after login you can see three input boxes and one dropdownlist within 3 items// all the three is reading from your DB so check your table frist.

然后我们修改 Post 类， 使它可以在日志被存入数据库前自动设置几个属性 (例如 create_time, author_id)。我们覆盖 beforeSave() 方法如下：

```
protected function beforeSave()
{
    if(parent::beforeSave())
    {
        if($this->isNewRecord)
        {
            $this->create_time=$this->update_time=time();
            $this->author_id=Yii::app()->user->id;
        }
        else
            $this->update_time=time();
        return true;
    }
    else
        return false;
}
```

当我们保存日志时， 我们想更新 tbl_tag 表以反映 Tag 的使用频率。我们可以在 afterSave() 方法中完成此工作， 它会在日志被成功存入数据库后自动被Yii调用。

```
    protected function afterSave()
    {
        parent::afterSave();
        Tag::model()->updateFrequency($this->_oldTags, $this->tags);
    }

    private $_oldTags;

    protected function afterFind()
    {
        parent::afterFind();
        $this->_oldTags=$this->tags;
    }
```

在这个实现中， 因为我们想检测出用户在更新现有日志的时候是否修改了 Tag ， 我们需要知道原来的 Tag 是什么, 鉴于此， 我们还写了一个 afterFind() 方法把原有的 Tag 信息保存到变量 _oldTags 中。方法 afterFind() 会在一个 AR 记录被数据库中的数据填充时自动被 Yii 调用。

Code ::Tag::updateFrequency() and his brothers and sisters.

```
    public function updateFrequency($oldTags, $newTags)
```

```
{
        $oldTags=self::string2array($oldTags);
        $newTags=self::string2array($newTags);
        $this->addTags(array_values(array_diff($newTags, $oldTags)));
        $this->removeTags(array_values(array_diff($oldTags, $newTags)));
}

public function addTags($tags)
{
        $criteria = new CDbCriteria();
        $criteria->addInCondition('name', $tags);
        $this->updateCounters(array('frequency'=>1),$criteria);
        foreach ($tags as $name)
        {
                if(!$this->exists('name=:name',array(':name'=>$name)))
                {
                        $tag=new Tag();
                        $tag->name=$name;
                        $tag->frequency=1;
                        $tag->save();
                }
        }
}

public function removeTags($tags)
{
        if(empty($tags))
                return ;
        $criteria = new CDbCriteria();
        $criteria->addInCondition('name', $tags);
        $this->updateCounters(array('frequency'=>-1), $criteria);
        $this->deleteAll('frequency<=0');
}
```

日志显示

在我们的博客应用中，一篇日志可以显示在一个列表中，也可以单独显示。前者的实现通过 index 操作，而后者是通过 view 操作。在这一节中，我们将自定义这两个操作来适合我们一开始的需求。

自定义 view 操作

view 操作是通过 PostController 中的 actionView() 方法实现的。它的显示是通过 view 视图文件 /wwwroot/blog/protected/views/post/view.php 生成的。

下面是在 PostController 中实现 view 操作的具体代码：

```
public function actionView()
{
    $post=$this->loadModel();
    $this->render('view',array(
        'model'=>$post,
    ));
}

private $_model;

public function loadModel()
{
    if($this->_model===null)
    {
        if(isset($_GET['id']))
        {
```

```
            if(Yii::app()->user->isGuest)
                $condition='status='.Post::STATUS_PUBLISHED
                    .' OR status='.Post::STATUS_ARCHIVED;
            else
                $condition='';
            $this->_model=Post::model()->findByPk($_GET['id'], $condition);
        }
        if($this->_model===null)
            throw new CHttpException(404,'The requested page does not exist.');
    }
    return $this->_model;
}
```

我们的修改主要是在 loadModel() 方法上进行的。在这个方法中，我们通过 id GET参数查询了 Post 表。如果日志未找到或者没有发布，也未存档(当用户为游客（guest）时)，我们将抛出一个 404 HTTP 错误。否则，日志对象将返回给 actionView()，actionView() 又会把日志对象传递给视图脚本用于显示。

提示: Yii 会捕获 HTTP 异常 (CHttpException 的实例) 并通过预置的模板或自定义的错误视图显示出来。由 yiic 生成的程序骨架已经包含了一个自定义的错误视图 /wwwroot/blog/protected/views/site/error.php。如果想进一步自定义此错误显示，我们可以自己修改此文件。

view 脚本中的修改主要是关于调整日志显示格式和样式的。 /wwwroot/blog/protected/views/post/view.php

Code::

```
<?php
$this->breadcrumbs=array(
        $model->title,
);
$this->pageTitle=$model->title;
?>

<?php $this->renderPartial('_view', array(
        'data'=>$model,
)); ?>
```

//余下的Comment 显示，添加表单，不需要在本步添加

自定义 index 操作

和 view 操作类似，我们在两处自定义 index 操作：PostController 中的 actionIndex() 方法和视图文件 /wwwroot/blog/protected/views/post/index.php。我们主要需要添加对显示一个特定Tag下的日志列表的支持；

下面就是在 PostController 中对 `actionIndex() 方法作出的修改:

```
    public function actionIndex()
    {
        $criteria=new CDbCriteria(array(
            'condition'=>'status='.Post::STATUS_PUBLISHED,
            'order'=>'update_time DESC',
            'with'=>'commentCount',
        ));
        if(isset($_GET['tag']))
            $criteria->addSearchCondition('tags',$_GET['tag']);

        $dataProvider=new CActiveDataProvider('Post', array(
            'pagination'=>array(
                'pageSize'=>5,
            ),
            'criteria'=>$criteria,
        ));
```

```
        $this->render('index',array(
            'dataProvider'=>$dataProvider,
        ));
    }
```

在上面的代码中，我们首先为检索日志列表创建了一个查询标准（criteria），此标准规定只返回已发布的日志，且应该按其更新时间倒序排列。因为我们打算在显示日志列表的同时显示日志收到的评论数量，因此在这个标准中我们还指定了要带回 commentCount, 如果你还记得，它就是在 Post::relations() 中定义的一个关系。

考虑到当用户想查看某个Tag下的日志列表时的情况，我们还要为指定的Tag添加一个搜索条件到上述标准中。

test:: webroot/b2/index.php?r=post&tag=yii

使用这个查询标准，我们创建了一个数据提供者（data provider）。这主要出于三个目的。第一，它会在查询结果过多时实现数据分页。这里我们定义分页的页面大小为5。第二，它会按用户的请求对数据排序。最后，它会填充排序并分页后的数据到小部件(widgets)或视图代码用于显示。

完成 actionIndex() 后，我们将 index 视图修改为如下代码。此修改主要是关于在用户指定显示Tag下的日志时添加一个 h1 标题。

```
<?php if(!empty($_GET['tag'])): ?>
<h1>Posts Tagged with <i><?php echo CHtml::encode($_GET['tag']); ?></i></h1>
<?php endif; ?>

<?php $this->widget('zii.widgets.CListView', array(
    'dataProvider'=>$dataProvider,
    'itemView'=>'_view',
    'template'=>"{items}\n{pager}",
)); ?>
```

注意上面的代码，我们使用了 CListView 来显示日志列表。这个小物件需要一个局部视图以显示每一篇日志的详情。这里我们制定了局部视图为 _view，也就是文件 /wwwroot/blog/protected/views/post/_view.php. 在这个视图脚本中，我们可以通过一个名为 $data 的本地变量访问显示的日志实例。

日志管理

日志管理主要是在一个管理视图中列出日志，我们可以查看所有状态的日志，更新或删除它们。它们分别通过 admin 操作和 delete 操作实现。yiic 生成的代码并不需要太多修改。下面我们主要解释这两个操作是怎样实现的。

在表格视图中列出日志

admin 操作在一个表格视图中列出了所有状态的日志。此视图支持排序和分页。下面就是 PostController 中的 actionAdmin() 方法：

```
    public function actionAdmin()
    {
        $model=new Post('search');
        if(isset($_GET['Post']))
            $model->attributes=$_GET['Post'];
        $this->render('admin',array(
            'model'=>$model,
        ));
    }
```

上面的代码由 yiic 工具生成，未作任何修改。它首先创建了一个 search 场景（scenario） 下的 Post 模型。我们将使用此模型收集用户指定的搜索条件。然后我们把用户可能会提供的数据赋值给模型。最后，我们以此模型显示 admin 视图。

下面就是 admin 视图的代码：

```
<?php
$this->breadcrumbs=array(
    'Manage Posts',
);
?>
<h1>Manage Posts</h1>

<?php $this->widget('zii.widgets.grid.CGridView', array(
    'dataProvider'=>$model->search(),
    'filter'=>$model,
    'columns'=>array(
        array(
            'name'=>'title',
            'type'=>'raw',
            'value'=>'CHtml::link(CHtml::encode($data->title), $data->url)'
        ),
        array(
            'name'=>'status',
            'value'=>'Lookup::item("PostStatus",$data->status)',
            'filter'=>Lookup::items('PostStatus'),
        ),
        array(
            'name'=>'create_time',
            'type'=>'datetime',
            'filter'=>false,
        ),
        array(
            'class'=>'CButtonColumn',
        ),
    ),
)); ?>
```

我们使用 CGridView 来显示这些日志。它允许我们在单页显示过多时可以分页并可以按某一列排序。我们的修改主要针对每一列的显示。例如，针对 title 列，我们指定它应该显示为一个超级链接，指向日志的详情页面。表达式$data->url 返回我们之前在 Post 类中定义的 url 属性值。

提示: 当显示文本时，我们要调用 CHtml::encode() 对其中的HTML编码。这可以防止 跨站脚本攻击(cross-site scripting attack).

日志删除

在 admin 数据表格中，每行有一个删除按钮。点击此按钮将会删除相应的日志。在程序内部，这会触发如下实现的 delete 动作。

```
public function actionDelete()
{
    if(Yii::app()->request->isPostRequest)
    {
        // we only allow deletion via POST request
        $this->loadModel()->delete();

        if(!isset($_POST['ajax']))
            $this->redirect(array('index'));
    }
    else
        throw new CHttpException(400,'Invalid request. Please do not repeat this request again.');
}
```

上面的代码就是 yiic 生成的代码，未经任何修改。我们想在此对判断 $_POST['ajax'] 稍作解释。CGridView 小物件有一个非常好的特性：它的排序、分页和删除操作默认是通过AJAX实现的。这就意味着在执行上述操作时，整个页面不会重新加载。然而，它也可以在非AJAX模式下运行（通过设置它的

ajaxUpdate 属性为 false 或在客户端禁用JavaScript）。delete 动作区分两个场景是必要的：如果删除请求通过AJAX提交，我们就不应该重定向用户的浏览器，反之则应该重定向。

删除日志应该同时导致日志的所有评论被删除。额外的，我们应更新相关的删除日志后的 tbl_tag 表。这两个任务都可以通过在 Post 模型类中写一个如下的 afterDelete 方法实现。

```
protected function afterDelete()
{
    parent::afterDelete();
    Comment::model()->deleteAll('post_id='.$this->id);
    Tag::model()->updateFrequency($this->tags, '');
}
```

上面的代码很直观：它首先删除了所有 post_id 和所删除的日志ID相同的那些评论。然后它针对所删日志中的 tags 更新了 tbl_tag 表。

提示: 由于 SQLite 并不真正支持外键约束，我们需要显式地删除属于所删日志的所有评论。在一个支持此约束的DBMS （例如 MySQL， PostgreSQL）中，可以设置好外键约束，这样如果删除了一篇日志，DBMS就可以自动删除其评论。这样的话，我们就不需要在我们的代码中显式执行删除了。

自定义评论模型

对 Comment 模型，我们主要需要自定义 rules() 和 attributeLabels() 方法。attributeLabels() 方法返回一个属性名字和属性标签的映射。由于 yiic 生成的 relations() 代码已经很不错了，我们现在就不需要改动这个玩意了。

Code :: Comment::relations()

```
public function relations()
{
        // NOTE: you may need to adjust the relation name and the related
        // class name for the relations automatically generated below.
        return array(
                'post' => array(self::BELONGS_TO, 'Post', 'post_id'),
        );
}
```

自定义 rules() 方法

我们首先自定义 yiic 工具生成的验证规则。用于评论的验证规则如下:

```
public function rules()
{
    return array(
        array('content, author, email', 'required'),
        array('author, email, url', 'length', 'max'=>128),
        array('email','email'),
        array('url','url'),
    );
}
```

如上所示，我们制定了 author, email 和 content 属性是必填项； author, email 和 url 的长度不能超过 128； email 属性必须是一个有效的 Email 地址； url 属性必须是一个有效的 URL；

自定义 attributeLabels() 方法

然后我们自定义 attributeLabels() 方法以声明每个模型属性显示时的标签（label）文本。此方法在当我们调用 CHtml::activeLabel() 显示一个属性标签时返回一个包含了名字-标签对的数组。

```
public function attributeLabels()
```

```
{
    return array(
        'id' => 'Id',
        'content' => 'Comment',
        'status' => 'Status',
        'create_time' => 'Create Time',
        'author' => 'Name',
        'email' => 'Email',
        'url' => 'Website',
        'post_id' => 'Post',
    );
}
```

提示: 如果属性的标签没有在 attributeLabels() 中定义，则会使用一种算法自动生成一个标签名。例如，将会为属性 create_time 或 createTime 生成标签 Create Time。

自定义存储的流程

由于我们想要记录评论创建的时间，和我们在 Post 模型中的做法一样，我们覆盖 Comment 的 beforeSave ()方法如下：

```
protected function beforeSave()
{
    if(parent::beforeSave())
    {
        if($this->isNewRecord)
            $this->create_time=time();
        return true;
    }
    else
        return false;
}
```

评论的创建与显示

此节中，我们实现评论的创建与显示功能。

为增强用户交互体验，我们打算在用户输入完每个表单域时就提示用户可能的出错信息。也就是客户端输入验证。我们将看到，在Yii中实现这个是多么简单多么爽。注意，这需要 Yii 1.1.1 版或更高版本的支持。

评论的显示

我们使用日志详情页(由 PostController 的 view动作生成)来显示和创建评论，而不是使用单独的页面。在日志内容下面，我们首先显示此日志的评论列表，然后显示一个用于创建评论的表单。

为了在日志详情页中显示评论，我们把视图脚本 /wwwroot/blog/protected/views/post/view.php 修改如下：

```
...这儿是日志的视图...
```

```
<div id="comments">
    <?php if($model->commentCount>=1): ?>
        <h3>
            <?php echo $model->commentCount . 'comment(s)'; ?>
        </h3>

        <?php $this->renderPartial('_comments',array(
            'post'=>$model,
            'comments'=>$model->comments,
        )); ?>
```

```
    <?php endif; ?>
</div>
```

如上所示，我们调用了 renderPartial() 方法渲染一个名为 _comments 的局部视图以显示属于当前日志的评论列表。注意，在这个视图中我们使用了表达式 $model->comments 获取日志的评论。这是有效的，因为我们在 Post 类中声明了一个 comments 关系。执行此表达式会触发一个隐式的 JOIN 数据库查询以获取相应的评论数据。此特性被称为 懒惰的关系查询（lazy relational query）。

Code ::/wwwroot/yii/demos/blog/protected/post/_comments.php.

```
<?php foreach($comments as $comment): ?>
<div class="comment" id="c<?php echo $comment->id; ?>">

        <?php echo CHtml::link("#{$comment->id}", $comment->getUrl($post), array(
                'class'=>'cid',
                'title'=>'Permalink to this comment',
        )); ?>

        <div class="author">
                <?php echo $comment->authorLink; ?> says:
        </div>

        <div class="time">
                <?php echo date('F j, Y \a\t h:i a',$comment->create_time); ?>
        </div>

        <div class="content">
                <?php echo nl2br(CHtml::encode($comment->content)); ?>
        </div>

</div><!-- comment -->
<?php endforeach; ?>
```

评论的创建

要处理评论创建，我们首先修改 PostController 中的 actionView() 方法如下：

```
public function actionView()
{
    $post=$this->loadModel();
    $comment=$this->newComment($post);

    $this->render('view',array(
        'model'=>$post,
        'comment'=>$comment,
    ));
}

protected function newComment($post)
{
    $comment=new Comment;
    if(isset($_POST['Comment']))
    {
        $comment->attributes=$_POST['Comment'];
        if($post->addComment($comment))
        {
            if($comment->status==Comment::STATUS_PENDING)
                Yii::app()->user->setFlash('commentSubmitted','Thank you...');
            $this->refresh();
        }
    }
    return $comment;
}
```

如上所示，我们在渲染 view 前调用了 newComment() 方法。在 newComment() 方法中，我们创建了一个 Comment 实例并检查评论表单是否已提交。如果已提交，我们尝试通过调用 $post->addComment

($comment) 添加日志评论。如果一切顺利，我们刷新详情页面。由于评论需要审核，我们将显示一条闪过信息(flash message)以作出提示。闪过信息通常是一条显示给最终用户的确认信息。如果用户点击了浏览器的刷新按钮，此信息将会消失。

Code :: Post::addComment()

```
public function addComment($comment)
{
        if(Yii::app()->params['commentNeedApproval'])
                $comment->status=Comment::STATUS_PENDING;
        else
                $comment->status=Comment::STATUS_APPROVED;
        $comment->post_id=$this->id;
        return $comment->save();
}
```

此外，我们还需要修改 /wwwroot/blog/protected/views/post/view.php ，

```
<div id="comments">
  ......
  <h3>Leave a Comment</h3>

  <?php if(Yii::app()->user->hasFlash('commentSubmitted')): ?>
    <div class="flash-success">
      <?php echo Yii::app()->user->getFlash('commentSubmitted'); ?>
    </div>
  <?php else: ?>
    <?php $this->renderPartial('/comment/_form',array(
      'model'=>$comment,
    )); ?>
  <?php endif; ?>

</div><!-- comments -->
```

以上代码中，如果有可用的闪过信息，我们就会显示它。如果没有，我们就通过渲染局部视图 /wwwroot/blog/protected/views/comment/_form.php 显示评论输入表单。

客户端验证

为支持评论表单的客户端验证，我们需要对评论表单视图 /wwwroot/blog/protected/views/comment/_form.php 和 newComment() 方法做一些小的修改。

在 _form.php 文件中，我们主要需要在创建 CActiveForm 小物件时设置 CActiveForm::enableAjaxValidation 为 true：

```
<div class="form">

<?php $form=$this->beginWidget('CActiveForm', array(
  'id'=>'comment-form',
  'enableAjaxValidation'=>true,
)); ?>
......
<?php $this->endWidget(); ?>

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

在 newComment() 方法中，我们插入了一段代码以响应 AJAX 验证请求。这段代码检查是否存在一个名为 ajax 的 POST 变量，如果存在，它将通过调用 CActiveForm::validate 显示验证结果。

```
protected function newComment($post)
{
  $comment=new Comment;
```

```
        if(isset($_POST['ajax']) && $_POST['ajax']==='comment-form')
        {
            echo CActiveForm::validate($comment);
            Yii::app()->end();
        }

        if(isset($_POST['Comment']))
        {
            $comment->attributes=$_POST['Comment'];
            if($post->addComment($comment))
            {
                if($comment->status==Comment::STATUS_PENDING)
                    Yii::app()->user->setFlash('commentSubmitted','Thank you...');
                $this->refresh();
            }
        }
        return $comment;
    }
```

评论管理

评论管理包括更新，删除和审核。这些操作是以 CommentController 类的动作实现的。

评论的更新与删除

由 yiic 生成的更新及删除评论的代码大部分都不需要修改。

评论审核

当评论刚创建时，它们处于待审核状态，需要等审核通过后才会显示给访客。审核评论主要就是修改评论的状态（status）列。

我们创建一个 CommentController 中的 actionApprove() 方法如下：

```
    public function actionApprove()
    {
        if(Yii::app()->request->isPostRequest)
        {
            $comment=$this->loadModel();
            $comment->approve();
            $this->redirect(array('index'));
        }
        else
            throw new CHttpException(400,'Invalid request...');
    }
```

如上所示，当 approve 动作通过一个 POST 请求被调用时，我们执行了 Comment 模型中定义的 approve() 方法改变评论状态。然后我们重定向用户浏览器到显示此评论所属日志的页面。

Code :: Comment::approve()

```
    public function approve()
    {
            $this->status=Comment::STATUS_APPROVED;
            $this->update(array('status'));
    }
```

我们还修改了 Comment 的 actionIndex()方法以显示所有评论。我们希望看到等待审核的评论显示在前面。

```
    public function actionIndex()
    {
        $dataProvider=new CActiveDataProvider('Comment', array(
            'criteria'=>array(
                'with'=>'post',
```

```
            'order'=>'t.status, t.create_time DESC',
        ),
    ));

    $this->render('index',array(
        'dataProvider'=>$dataProvider,
    ));
}
```

注意，在上面的代码中，由于 tbl_post 和 tbl_comment 表都含有 status 和 create_time 列，我们需要通过使用表的别名前缀消除列名的歧义。正如 指南 中所描述的，在一个关系查询中，主表的别名总是使用 t。因此，我们在上面的代码中对 status 和 create_time 使用了 t 前缀。

和日志的索引视图（index view）类似，CommentController 的 index 视图使用 CListView 显示评论列表，CListView又使用了局部视图 /wwwroot/blog/protected/views/comment/_view.php 显示每一条评论。

Code :: /wwwroot/yii/demos/blog/protected/views/comment/_view.php.

```
<div class="comment" id="c<?php echo $data->id; ?>">

    <?php echo CHtml::link("#{$data->id}", $data->url, array(
            'class'=>'cid',
            'title'=>'Permalink to this comment',
     )); ?>

     <div class="author">
            <?php echo $data->authorLink; ?> says on
            <?php echo CHtml::link(CHtml::encode($data->post->title), $data->post->url); ?>
        </div>

        <div class="time">
            <?php if($data->status==Comment::STATUS_PENDING): ?>
                <span class="pending">Pending approval</span> |
                <?php echo CHtml::linkButton('Approve', array(
                    'submit'=>array('comment/approve','id'=>$data->id),
                )); ?> |
            <?php endif; ?>
            <?php echo CHtml::link('Update',array('comment/update','id'=>$data->id)); ?> |
            <?php echo CHtml::linkButton('Delete', array(
                'submit'=>array('comment/delete','id'=>$data->id),
                'confirm'=>"Are you sure to delete comment #{$data->id}?",
            )); ?> |
            <?php echo date('F j, Y \a\t h:i a',$data->create_time); ?>
        </div>

        <div class="content">
            <?php echo nl2br(CHtml::encode($data->content)); ?>
        </div>

</div><!-- comment -->
```

创建用户菜单 Portlet

基于需求分析，我们需要三个不同的 portlet （译者注：如果一开始不理解什么是 portlet 没关系，继续往下看就知道了。）：“用户菜单” portlet，“标签云” portlet 和“最新评论” portlet 。我们将通过继承Yii提供的 CPortlet 小物件开发这三个 portlet。

在这一节中，我们将开发第一个具体的 portlet ——用户菜单 portlet，它显示一个只对已通过身份验证的用户可见的菜单。此菜单包含四个项目：

- 评论审核: 一个指向待审核评论列表的超级链接；

•创建新日志: 一个指向日志创建页的超级链接；

•管理日志: 一个指向日志管理页的超级链接；

•注销: 一个可用于注销当前用户的链接按钮。

创建 UserMenu 类

我们创建一个用于表现用户菜单 portlet 逻辑的 UserMenu 类。此类保存在文件 /wwwroot/blog/protected/components/UserMenu.php 中， 其代码如下：

```
Yii::import('zii.widgets.CPortlet');

class UserMenu extends CPortlet
{
    public function init()
    {
        $this->title=CHtml::encode(Yii::app()->user->name);
        parent::init();
    }

    protected function renderContent()
    {
        $this->render('userMenu');
    }
}
```

UserMenu 类继承自 zii 库中的 CPortlet 类。它覆盖了 CPortlet 类的 init() 和 renderContent() 方法。前者设置 portlet 的标题为当前用户的名字；后者通过渲染一个名为 userMenu 的视图生成 portlet 的主体内容。

提示: 注意， 我们必须在首次使用之前通过调用 Yii::import() 显式包含 CPortlet 类。这是因为 CPortlet 是 zii 工程的一部分。zii 工程是 Yii 的官方扩展库。出于性能的考虑， 此工程中的类并未列入核心类。因此，我们必须在首次使用之前将其导入（import）。

创建 userMenu 视图

然后， 我们创建 userMenu 视图， 它保存在 /wwwroot/blog/protected/components/views/userMenu.php:

```
<ul>
    <li><?php echo CHtml::link('Create New Post',array('post/create')); ?></li>
    <li><?php echo CHtml::link('Manage Posts',array('post/admin')); ?></li>
    <li><?php echo CHtml::link('Approve Comments',array('comment/index'))
        . ' (' . Comment::model()->pendingCommentCount . ')'; ?></li>
    <li><?php echo CHtml::link('Logout',array('site/logout')); ?></li>
</ul>
Code:: Comment::getPendingCommentCount()
    public function getPendingCommentCount()
    {
            return $this->count('status='.self::STATUS_PENDING);
    }
```

信息: 默认情况下， 小物件的视图文件应保存在包含小物件类文件的目录的 views 子目录中。文件名必须和视图名称相同。

使用 UserMenu Portlet

是可以把我们新完成的 UserMenu portlet 投入使用的时候了。我们把布局文件 /wwwroot/blog/protected/views/layouts/column2.php 修改如下:

```
.....
<div id="sidebar">
    <?php if(!Yii::app()->user->isGuest) $this->widget('UserMenu'); ?>
```

```
</div>
.....
```

如上所示，我们调用了 widget() 方法创建并执行了 UserMenu 类的实例。由于此 portlet 只应显示给已通过身份验证的用户，我们只在当前用户的 isGuest 属性为 false 时（即用户未登录时）调用 widget() 方法。

测试 UserMenu **Portlet**

让我们来测试一下所作的工作：

1. 打开浏览器输入 URL http://www.example.com/blog/index.php 。核实页面中的侧边栏中没有任何东西显示。
2. 点击 Login 超链接，填写登录表单登录，如果登录成功，核实 UserMenu portlet 显示在了侧边栏中，且其标题为当前用户名。
3. 点击 UserMenu portlet 中的 'Logout'，核实注销成功且 UserMenu portlet 已消失。

总结

我们创建的是一个 portlet，它是高度可复用的。我们可以稍加修改或不作修改就能很容易地把它用在另一个不同的工程中。此外，此 portlet 的设计完美重现了表现与逻辑分离的思想。虽然我们在前面的部分中没有提到这一点，但此实践在一个典型的 Yii 应用中几乎随处可见。

创建标签云 **Portlet**

标签云 显示一个日志标签列表，每个标签都可以通过可视化的方式反映其使用频度。

创建 TagCloud 类

我们在 /wwwroot/blog/protected/components/TagCloud.php 文件中创建 TagCloud 类。此文件内容如下：

```
Yii::import('zii.widgets.CPortlet');

class TagCloud extends CPortlet
{
    public $title='Tags';
    public $maxTags=20;

    protected function renderContent()
    {
        $tags=Tag::model()->findTagWeights($this->maxTags); //findTagWeights 方法如下

        foreach($tags as $tag=>$weight)
        {
            $link=CHtml::link(CHtml::encode($tag), array('post/index','tag'=>$tag));
            echo CHtml::tag('span', array(
                'class'=>'tag',
                'style'=>"font-size:{$weight}pt",
            ), $link)."\n";
        }
    }
}
```

与 UserMenu portlet 不同,TagCloud portlet 不使用视图。它的前端表现是在 renderContent() 方法中完成的。这是因为其前端表现并不含有很多HTML标签。

Code ::Tag::findTagWeights()

```
public function findTagWeights($limit=20)
{
        $models=$this->findAll(array(
                'order'=>'frequency DESC',
                'limit'=>$limit,
```

```
        ));

        $total=0;
        foreach($models as $model)
                $total+=$model->frequency;

        $tags=array();
        if($total>0)
        {
                foreach($models as $model)
                        $tags[$model->name]=8+(int)(16*$model->frequency/($total+10));
                ksort($tags);
        }
        return $tags;
    }
```

我们把每个标签显示为指向带有此标签参数的日志索引页的链接。每个标签链接的文字大小是通过他们与其他标签的相对比重确定的。如果一个标签比其他标签有更高的使用频度，则它会以更大的字体显示。

使用 TagCloud **Portlet**

TagCloud portlet 的使用非常简单。我们把布局文件 /wwwroot/blog/protected/views/layouts/column2.php 修改如下：

```
<div id="sidebar">

    <?php if(!Yii::app()->user->isGuest) $this->widget('UserMenu'); ?>

    <?php $this->widget('TagCloud', array(
                        'maxTags'=>Yii::app()->params['tagCloudCount'],//config/main.php      params里设置 'tagCloudCount'=>20
    )); ?>

</div>
```

创建最新评论 **Portlet**

此节中，我们创建最后一个 portlet，它将显示最新发布的评论列表。

创建 RecentComments 类

我们将 RecentComments类创建在文件 /wwwroot/blog/protected/components/RecentComments.php 中。此文件内容如下：

```
Yii::import('zii.widgets.CPortlet');

class RecentComments extends CPortlet
{
    public $title='Recent Comments';
    public $maxComments=10;

    public function getRecentComments()
    {
        return Comment::model()->findRecentComments($this->maxComments);
    }

    protected function renderContent()
    {
        $this->render('recentComments');
    }
}
```

如上所示，我们调用了 Comment 类中定义的 findRecentComments 方法。此方法代码如下：

```
class Comment extends CActiveRecord
{
    ......
    public function findRecentComments($limit=10)
    {
        return $this->with('post')->findAll(array(
            'condition'=>'t.status='.self::STATUS_APPROVED,
            'order'=>'t.create_time DESC',
            'limit'=>$limit,
        ));
    }
}
```

创建 recentComments 视图

recentComments 视图存储在文件 /wwwroot/blog/protected/components/views/recentComments.php 中。它只是简单的显示由 RecentComments::getRecentComments() 方法返回的每一条评论。

Code :: /wwwroot/blog/protected/components/views/recentComments.php

```
<ul>
    <?php foreach($this->getRecentComments() as $comment): ?>
    <li><?php echo $comment->authorLink; ?> on
        <?php echo CHtml::link(CHtml::encode($comment->post->title), $comment->getUrl()); ?>
    </li>
    <?php endforeach; ?>
</ul>
```

使用 RecentComments **Portlet**

我们修改布局文件 /wwwroot/blog/protected/views/layouts/column2.php 插入最后这个 portlet,

```
<div id="sidebar">

    <?php if(!Yii::app()->user->isGuest) $this->widget('UserMenu'); ?>

    <?php $this->widget('TagCloud', array(
        'maxTags'=>Yii::app()->params['tagCloudCount'],
    )); ?>

    <?php $this->widget('RecentComments', array(
        'maxComments'=>Yii::app()->params['recentCommentCount'],
//config/main.php  params里设置 'recentCommentCount'=>10
    )); ?>

</div>
```

美化 URL

链接着我们博客应用不同页面的 URL 看起来很丑。例如展示日志内容的页面，其 URL 如下：

```
/index.php?r=post/show&id=1&title=A+Test+Post
```

此节中，我们将讲解如何美化这些 URL 并使它们对 SEO 友好。我们的目标是在应用中可以使用如下样式的 URL：

1./index.php/posts/yii： 指向属于标签 yii 的日志列表页；

2./index.php/post/2/A+Test+Post： 指向 ID 为 2，标题为 A Test Post 的日志的日志详情页；

3./index.php/post/update?id=1： 指向 ID 为 1 的日志更新页。

注意在第二个URL格式中，我们在URL中还包含了日志标题。这主要是为了使其对 SEO 友好。据说搜索引擎会在索引URL时重视其中的单词。

要实现我们的目标，我们修改 应用配置 如下，

```
return array(
    ......
    'components'=>array(
        ......
        'urlManager'=>array(
            'urlFormat'=>'path',
            'rules'=>array(
                'post/<id:\d+>/<title:.*?>'=>'post/view',
                'posts/<tag:.*?>'=>'post/index',
                '<controller:\w+>/<action:\w+>'=>'<controller>/<action>',
            ),
        ),
    ),
);
```

如上所示，我们配置了 urlManager 组件，设置其 urlFormat 属性为 path 并添加了一系列 rules （规则）。

urlManager 通过这些规则解析并创建目标格式的URL。例如，第二条规则指明：如果一个 URL /index.php/posts/yii 被请求，urlManager 组件就应负责调度此请求到 路由（route） post/index 并创建一个值为 yii 的 GET 参数 tag 。从另一个角度来说，当使用路由 post/index 和 tag 参数生成URL时，urlManager 组件将同样使用此规则生成目标 URL /index.php/posts/yii。鉴于此，我们说 urlManager 是一个双向的 URL 管理器。

urlManager 组件还可以继续美化我们的URL，例如从URL中隐藏 index.php ，在URL的结尾添加 .html 等。我们可以通过在应用配置中设置 urlManager 的各种属性实现这些功能。更多详情，请参考 指南.

错误日志

生产环境中的 Web 应用常需要具有完善的事件日志功能。在我们的博客应用中，我们想记录它在使用时发生的错误。这些错误可能是程序错误或者是用户对系统的不当使用导致的错误。记录这些错误可以帮助我们完善此博客应用。

为启用错误日志功能，我们修改 应用配置 如下，

```
return array(
    'preload'=>array('log'),
    'components'=>array(
        'log'=>array(
            'class'=>'CLogRouter',
            'routes'=>array(
                array(
                    'class'=>'CFileLogRoute',
                    'levels'=>'error, warning',
                ),
            ),
        ),
    ),
);
```

通过上述配置，如果有错误（error）或警告（warning）发生，其详细信息将被记录并保存到位于 /wwwroot/blog/protected/runtime 目录的文件中。

log 组件还提供了更多的高级功能，例如将日志信息发送到一个 Email 列表，在 JavaScript 控制台窗口中显示日志信息等。更多详情，请参考指南。

最终调整与部署

我们的博客应用快要完成了。在部署之前，我们还想做一些调整。

修改主页

我们要把日志列表页修改为主页。我们将 应用配置 修改如下，

```
return array(
    ......
    'defaultController'=>'post',
    ......
);
```

提示: 由于 PostController 已经声明了 index 作为它的默认动作，当我们访问此应用的首页时，我们将看到由 post 控制器的 index 动作生成的结果页面。

启用表结构缓存

由于 ActiveRecord 按数据表的元数据（metadata）测定列的信息。读取元数据并对其分析需要消耗时间。这在开发环境中应该问题不大，但对于一个在生产环境中运行的应用来说，数据表结构如果不发生变化那这就是在浪费时间。因此，我们应通过修改应用配置启用数据表结构缓存，

```
return array(
    ......
    'components'=>array(
        ......
        'cache'=>array(
            'class'=>'CDbCache',
        ),
        'db'=>array(
            'class'=>'system.db.CDbConnection',
            'connectionString'=>'sqlite:/wwwroot/blog/protected/data/blog.db',
            'schemaCachingDuration'=>3600,
        ),
    ),
);
```

如上所示，我们首先添加了一个 cache 组件，它使用一个默认的 SQLite 数据库作为缓存平台。如果我们的服务器配备了其他的缓存扩展，例如 APC，我们同样可以使用它们。我们还修改了 db 组件，设置它的 schemaCachingDuration 属性为 3600，这样解析的数据表结构将可以在 3600 秒的缓存期内有效。

禁用除错（Debug）模式

我们修改入口文件 /wwwroot/blog/index.php，移除定义了 YII_DEBUG 常量的那一行。此常量在开发环境中非常有用，它使 Yii 在错误发生时显示更多的除错信息。然而，当应用运行于生产环境时，显示除错信息并不是一个好主意。因为它可能含有一些敏感信息，例如文件所在的位置，文件的内容等。

部署应用

最终的部署主要是将 /wwwroot/blog 目录复制到目标目录。下面的检查列表列出了每一个所需的步骤：

1.如果目标位置没有可用的 Yii，先将其安装好。

2.复制整个 /wwwroot/blog 目录到目标位置；

3.修改入口文件 index.php，把 $yii 变量指向新的Yii引导文件。

4.修改文件 protected/yiic.php，设置 $yiic 变量的值为新的 yiic.php 文件位置；

5.修改目录 assets 和 protected/runtime 的权限，确保Web服务器进程对它们有可写权。

今后的增强

使用主题

不需要写任何代码，我们的博客应用已经是 可更换主题(themeable) 的了。要使用主题，我们主要是需要通过编写个性化的视图文件开发主题。例如，要使用一个名为 classic 的使用不同布局的主题，我们需要创建一个布局视图文件 /wwwroot/blog/themes/classic/views/layouts/main.php。我们还需要修改应用配置以显示我们选择的 classic 主题。

```
return array(
    ......
    'theme'=>'classic',
    ......
);
```

国际化

我们也可以把我们的博客应用国际化，这样它就可以通过多种语言显示。这主要包括两方面的工作。

第一，我们创建不同语言的视图文件。例如，针对 PostController 的 index 页面，我们创建了视图文件 /wwwroot/blog/protected/views/post/zh_cn/index.php。当应用的语言被配置为简体中文（语言代码是 zh_cn）时，Yii 将自动使用此视图文件。

第二，我们可以为代码生成的信息创建信息翻译。信息翻译应保存在目录 /wwwroot/blog/protected/messages 中，我们也需要在使用文本字符串的地方调用 Yii::t() 方法把这些字符串括起来。

关于国际化的更多详情，请参考 指南。

通过缓存提高性能

虽然 Yii 框架 非常高效，但 Yii 写的某个应用未必高效。在我们的博客应用中有基础可以提高性能的地方。例如，标签云 portlet 可能是性能瓶颈之一，因为它使用了较复杂的数据库查询和PHP逻辑。

我们可以使用 Yii 提供的成熟的 缓存功能 提高性能。Yii 中最有用的组件之一就是 COutputCache，它会缓存页面显示中的片段，这样生成此片段的代码就不需要在每次收到请求时执行。例如，在布局文件 /wwwroot/blog/protected/views/layouts/column2.php 中，我们可以将标签云 portlet 嵌入到 COutputCache 中:

```
<?php if($this->beginCache('tagCloud', array('duration'=>3600))) { ?>

    <?php $this->widget('TagCloud', array(
        'maxTags'=>Yii::app()->params['tagCloudCount'],
    )); ?>

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

通过以上代码，标签云的显示将由缓存实现，而不需要在每次收到请求时实时生成。缓存内容将在 3600 秒的缓存期内有效。

添加新功能

我们的博客应用现在只有非常基本的功能。要成为一个完整的博客系统，还需要添加更多的功能。例如，日历 portlet，邮件提醒，日志分类，存档日志 portlet 等等。我们把这些功能的实现留给感兴趣的读者。