

第一章：认识 Yii

认识 Yii

在过去几年中，框架迅速发展，几乎在 **Web** 应用开发中，每个人都会涉及到一个新生框架，**Web** 开发框架会帮助你加快你的应用程序发布，你只需迅速的把你的想法在框架的白板上书写功能代码。随着 **Web** 应用的实现具有共同特征，现有的框架方案已经满足这些要求，在今天还有什么理由要从头开始你的下一个 **Web** 应用呢？今天的 **Web** 开发，除程序自身语言外，一个现代化、灵活的和可扩展的框架，几乎是一个至关重要的编程工具，此外，如果语言与框架两个部份有特别的互补性，结果是将一个非常强大的工具包：**Java** 和 **Spring**，**Ruby** 和 **Rails**，**C#**和**.NET**，**PHP** 和 **Yii**。

Yii 是创始人薛强的心血结晶，于 **2008** 年 **1** 月 **1** 日开始开发。在此这前，薛强开发和维护 **PRADO** 框架多年。这些年的经验和从 **PRADO** 项目得到的用户的反馈了解到，用户需要一个更容易、可扩展、更快速的基于 **PHP5** 的框架，以满足应用程序开发人员不断增长的需求。**Yii** 正式发布于 **2008** 年 **10** 月，最初是 **alpha** 版本，其与其他基于 **PHP** 的框架表现相比令人印象深刻，立即引起非常积极的关注。在 **2008** 年 **12** 月 **3** 日 **Yii1.0** 正式发布，并于 **2010** 年 **3** 月 **14** 日，发布最新版本 **1.1.2**。它有一个不断增长的开发团队，并继续在每天 **PHP** 开发人员中得到收益。我们认为，在这本书中包含信息会对你有些帮助，你很快就会明白为什么。

Yii 的名称(是一个缩写，发音为 **Yee** 或 **[ji:]**)代表容易(**easy**)，高效(**efficient**)和可扩展(**extensible**)。**Yii** 是用 **PHP5** 写的一个高性能，基于组件的 **Web** 开发应用框架。**Yii** 可以更容易的创建和维护大规模的网络应用程序。这也将使应用程序更有效和可扩展。让我们快速了解一下这些特性。

Yii 很容易

要运行基于 **Yii** 的 **Web** 应用，你需要它的核心框架和一台支持 **PHP5.1.0** 以上的 **Web** 服务器，如果使用 **Yii** 开发，你需要精通 **PHP** 与面向对象编程 (**OOP**)。你不需要学习什么新的配置和模板语言，建立 **Yii** 应用，主要包括编写和维护自己定义的 **PHP** 类，其中一些将是继承 **Yii** 核心框架组件类。

Yii 集成了许多来自其他著名的 **Web** 程序框架和应用程序中的伟大的想法，这些你可能会很熟悉并易于操作。

Yii 还包含了一个易用的配置约定。这意味着，**Yii** 已经为几乎所有的应用程序编写了合理的默认值，如果你按照约定的规范，你会写更少的代码，花更少的时间来开发应用程序。如果需要 **Yii** 允许你在这些约定基础之上进行定制，本章后面及整书，我们将覆盖到这些默认的约定。

Yii 很高效

Yii 是为开发任何规模的 **Web** 应用提供的一个高性能的基于组件的 **Web** 框架，它鼓励在 **Web** 程序中最大化的代码重用，并可加快开发速度。如前所述，如果你坚持使用 **Yii** 的内置约定，你可以让您的应用在运行中，几乎不需要任何配置手册。

Yii 的另一个目的是帮助你使用 **DRY** 开发，**DRY** (**Don't Repeat Yourself**) 是一种灵活的应用开发。所有 Yii 应用是使用模型-视图-控制器(**MVC**)架构，Yii 强制这种开发模式，通过提供一个放置你的 **MVC** 代码的位置，这最大限度地减少重复，并有助于代码重用性和可维护性。你编写越少的代码，则需要的时间就越少，应用程序将赢得市场。同样，越容易维护你的应用程序，留在市场的时间就越长。

当然，Yii 不仅开发快，他的运行速度也是非常快的，性能是经过优化的。Yii 从一开始就已经做到了开发与性能优化，其结果是，他是一个最快的 **PHP** 开发框架。Yii 的开发团队已经与其他 **PHP** 框架进行了对比与测试，Yii 运行速度在他们之上。这意味着在 Yii 上编写应用程序增加额外的开销是微不足道的。

Yii 可扩展

Yii 是经过精心设计的，让几乎所有的代码都可扩展和可定制，以满足不同任务需要。事实上，在开发 Yii 的应用时，很难用不到 Yii 扩展性，这是一个很主要的行为，可扩展是框架的核心。如果你要将你的代码为作可重用的工具，提供给其他开发者使用，Yii 提供了简单制作步骤和遵循准则，以帮助您建立这样的第三方扩展。并允许你将他贡献到 Yii 扩展列表中。

Yii 最引人注目的是什么。易用性、性能优越、可扩展。Yii 的功能包会帮助你满足今天在 **Web** 应用程序上的那些更高要求。**AJAX** 可用挂件，**Web Service** 包，基于 **MVC** 结构的验证，**DAO** 和 **Active Record** 数据库层，复杂的缓存，分级的角色访问控制，主题化，国际化(**I18N**)，和本地化(**L10N**)，这仅仅是 Yii 的冰山一角。现在从 1.1 版本，核心框架中有一个官方扩展类库叫 **Zii**。这些扩展是由框架的核心团队成员开发与维护。社区用户也可以编写并贡献扩展，Yii 的扩展每天都在增长，有关所有可用的用户贡献的扩展，详见 <http://www.yiiframework.com/extensions/>

MVC 架构

如前所述，Yii 是一个 **MVC** 框架，它提供了一个明确的目录结构，模型，视图，控制器的代码分别放到自己的目录，在我们建立第一个 Yii 应用之前，我们需要定义一些关键语，并查看 Yii 的 **MVC** 架构是如何实现和执行的。

模型

通常在一个 **MVC** 架构中，模型是负责维护状态，因为，它应该封装业务规则，定义数据的状态。在 Yii 中，一个模型可以是 **CModel** 的一个实例或它的子类。通常一个模型类包括数据的属性，可能还会有不同的标签（有些是为了显示给用户时更友好），并且可以设置一些规则进行验证。模型中的数据可能来自数据库的表或一个表单用户输入域。

Yii 实现了两种模型：表单模型(**CFormModel** 类)和 **Active Record** 模型(**CActiveRecord** 类)。他们都继承自同一个基类 **CModel**。**CFormModel** 代表的模型是从 **HTML** 表单中收集的输入，它封装了所有逻辑，如表单的验证和其他业务逻辑，这些是要应用到表单的域上。它将这些数据存储在内存中，或者在一个 **Active Record** 的模型帮助下，存入数据库。

Active Record (AR) 是一种设计模式，用面向对象的方式抽象的访问数据。在 Yii 中，每一个 **AR** 对象的实例可以是 **CActiveRecord** 类或它的子类，它包装了数据库表或视图中的一行记录，并封装了所有逻辑

和访问数据库的细节，如果有大部份的业务逻辑，则必须使用这种模型。数据库表中一行每列字段的值对应 **AR** 对象的一个属性。关于 **AR** 的更多介绍，后面将详细说明。

视图

通常情况下，视图是在数据模型的基础上渲染用户界面，**Yii** 中一个视图就是一个 **PHP** 文件，它包含有关用户界面相关的元素，经常使用 **HTML**，但也可以使用 **PHP** 语句。通常在视图中的 **PHP** 语句应该是非常简单的条件或循环语句，或者使用 **Yii** 的 **UI** 组件，比如 **HTML** 助手的方法或预先建立好的挂件。更复杂的逻辑应该与视图分离放到适当的模型中（如数据直接处理），或控制器中。

控制器

控制器主要是处理一个路由的请求，也负责获得用户的输入，与模型的交互，并指定视图的显示与更新。**Yii** 中一个控制器实例可能是 **CController** 类或它的子类。当一个控制器运行时，它会执行一个请求的操作，然后与需要的模型交互，最后渲染到一个视图上。一个操作，最简单的形式就是一个控制器类方法，方法的名字要以 **action** 开头。

把 MVC 连接一起：Yii 路由请求

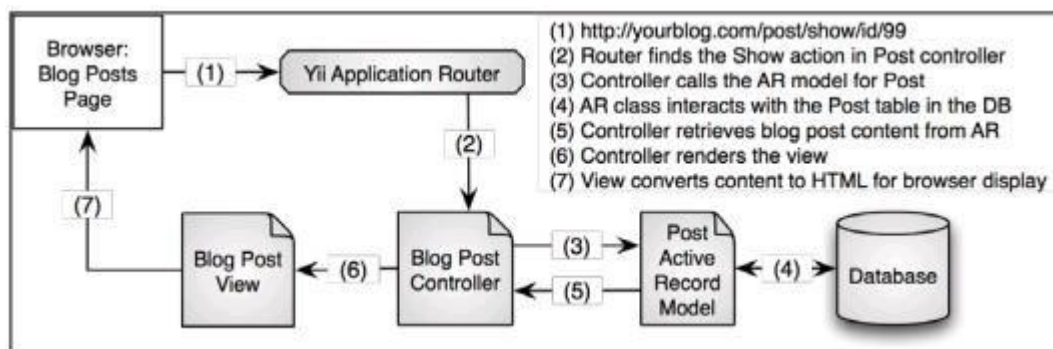
大多数的 **MVC** 实现中，**Web** 请求生命周期通常如下步骤：

- 1.浏览器发送请求到服务器上的 **MVC** 应用。
- 2.调用一个控制器用来处理请求。
- 3.这个控制器与模型交互
- 4.这个控制器调用视图
- 5.视图渲染数据(通常是 **HTML**) 并返回给浏览器显示

Yii 的 **MVC** 实现也不例外，在一个 **Yii** 应用程序中，从浏览器发送处理的请求，首先会传给一个路由。路由分析请求，并确定下一步的处理。在大多数情况下路由会识别控制器中的特定操作方法。这个操作方法将关注传入的请求数据与模型交互和执行其他需要的业务逻辑，最终，这个操作方法将处理的响应数据发送给他对应的视图类。视图将确认符合预期布局设计的数据，并返回到浏览器显示。

Blog 执行例子

为了更好的理解，让我们看一个虚构的例子。假如我们用 **Yii** 建立了自己的一个博客网站，域名是 **yourblog.com**。这个网站与典型的博客网站类似。在首页显示最近发布的文章列表。点击每篇文章的标题，可以显示完整的文章。下面的图片，可以说明 **Yii** 是怎样点击标题，发送请求的。



这个图片是跟踪到用户的点击连接：

<http://yourblog.com/post/show/id/99>

首先，请求发送给路由。路由解析请求 URL 结构中的关键词决定发送到哪。默认情况下，Yii 的 URL 结构如下格式：

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

URL 中的变量 **r** 指的是路由，Yii 分析路由这条路由，以确定适当的控制器和操作方法做进一步处理请求。

现在，你可能已经注意到，前面提到的 URL 地址并不遵循这个默认格式。其实这是一个非常简单的事情，通过配置来使用更友好的格式：

<http://hostname/ControllerID/ActionID>

这个例子我们将继续使用这行简化的格式。该 URL 中 **ControllerID** 指的是控制器的名称。默认情况下第一部份加上单词 **Controller** 就是控制器类的名称，例如：如果你的控制器类名是 **TestController** 则 **ControllerID** 应该是 **Test**。同样 **ActionID** 指的是控制器中定义的操作名称。如果要在控制器中定义操作，那么要将单词 **Action** 加上操作名称，例如：如果您的操作方法名为 **actionCreate()**，那么 **ActionID** 则是 **Create**。

如果 **ActionID** 省略，控制器将采用默认的操作方法，这个方法是 **actionIndex()**。如果 **ControllerID** 也省略了，应用程序将使用默认控制器。Yii 默认的控制是 **SiteController**

回到这个例子，路由将分析下面的 URL，<http://yourblog.com/post/show/id/99>，并会取 URL 路径的第一部份作为 **ControllerID**，第二部份作为 **ActionID**。这将转化为路由请求到 **PostController** 类中的 **actionShow()** 方法。URL 最后部份 (**id/99**) 是一个键/值的查询参数，将在处理过程中，提供给操作方法。在这个例子中，表示所选的博客文章 ID 为 **99**。

actionShow() 方法处理具体的博客文章请求。在这种情况下，它使用查询字符串变量 **id**，来确定具体的要求，他将要求模型来查询 **id=99** 的博客文章。

AR 模型类请求数据库返回数据给控制器，控制器进一步准备数据给视图，视图则渲染到一个需要的 **HTML** 中，并将响应返回给用户的浏览器。

MVC 架构允许我们将模型，控制器与视图分离，这使得开发人员可以很容易改变应用程序，而不影响用户界面(**UI**)，**UI** 设计师也可以自由作出不影响模型的业务逻辑变化。这种分离也可以很容易提供多个相同的模型。例如，你可以在 **yourblog.com** 的 **HTML** 页面或一个 **Flash/Flex RIA** 演示或移动应用程序或 **Web Service** 使用相同的模型代码，使用 **MVC** 约定的分离功能，将使应用程序更容易扩展和维护。

Yii 已经帮你做了很多，并不是简单的分离代码，包括一些命令的约定和建议代码应该放置在同一类目录中。它已经帮助你写好了需要将这些部份关联在一起的代码。这使你可以享受严格的 **MVC** 设计带来的好处，而不必花费时间编写自己所需细节代码。让我们看看它都写了些什么。

对象关系映射和 Active Record

在多数情况下，我们建立的 **Web** 应用程序内部会与数据库关联。如前面的博客发布程序，该程序中博客的文章内容存在数据库的表中。然而，**Web** 应用程序需要的数据，与数据库映射到内存中类定义的属性是对应的。对象关系映射(**ORM**)库提供的，就是映射数据库表为一个对象的类。

ORM 的大部份代码是如何实现数据库表中的字段与对象之间的关系，这是非常繁琐和重复的工作。幸运的是，**Yii** 帮助你们完成了这些繁琐而又乏味的工作，它提供给我们一个 **ORM** 层，这就是 **AR** 模式。

Active Record

正如前面提到的，**AR** 是一种设计模型，用面向对象的方式抽象的访问数据。它将表映射到类。行映射到对象，列则映射到对象的数据。换句话说，每一个 **Active Record** 类的实例代表了数据库表中的一行。然而，一个 **AR** 类不仅仅是数据库表中字段与类中属性的映射关系，他还需要在这些数据上处理一些业务逻辑。最终这个类定义了所有关于怎样对数据库的读写操作。

依靠约定和合理的默认值，**Yii** 的 **AR** 对象将减少开发的时间，不必花费时间在繁琐和重复创建、读取、更新和删除数据的 **SQL** 语句上。它也允许开发人员使用面向对象的方式访问更多的数据库数据。为了说明这一点，在博客系统中的示例代码是用 **AR** 操作对应的数据库表中的 **id**, **id** 是表的主键，**id** 等于 **99**。首先它会检查主键，然后修改标题并将修改结果保存到数据库：

PHP 代码：

```
$post=Post::model()->findPk(99);

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

$post->save();
```

Active Record 完成了我们以前需要编的任何 **SQL** 或其他要处理数据库的事情。

Active Record 不只做了这些。它还无缝的集成了许多 **Yii** 框架的其他方面。有许多表单输入域的 **HTML** 助手捆绑在 **AR** 类的属性上。这样，**Active Record** 提取表单输入值时直接进入模型。并且它支持自动化数据验证，如果验证失败，**Yii** 视图类会显示验证错误给最终用户。我们将在本书提供的许多具体例子再次介绍 **AR**。

视图与控制器

视图和控制器是非常紧密的兄弟。控制器会生成需要显示的可用数据给视图，视图生成页面，并触发事件，发送数据给控制器。

在 **Yii** 中，一个视图文件是由所属的控制器类确定是否渲染它。这样，在视图脚本里，我们可以通过简单的方法 `$this` 引用到控制器实例。这样的实现，使视图和控制器更加紧密。幸运的是，**Yii** 已经完成了所有的细节，因此，我们可以将精力集中具体应用程序的编码上。

Yii 的控制器不仅仅是调用模型和渲染视图。控制器能管理请求前或请求后的操作要求，实现基本的访问控制规则来限制某些操作，管理应用程序范围内的布局和嵌套布局，管理数据的分页和许多幕后的服务。同样，我们要感谢 **Yii** 不需要让我们插手这些零乱工作的细节。

Yii 还有很多内容，要探索它的奥秘，最好的方法是使用它。现在我们已了解一些基础概念和术语，这里是我们的第一个里程碑。在下一章，我们将通过简单的了解 **Yii** 的安装过程和建立一个可运行的应用程序，以更好地说明。

小结

在这一章中，我们介绍了一个高级的 **Web** 应用框架 **Yii**。我们还了解了 **Yii** 的设计理念。初步讨论这些抽象的内容，你是否失去了一点点信心呢？不要担心，这些例子都是有意义的，但，还是要回顾一下这章具体包括的内容：

1. 应用程序开发框架的重要性。
2. **Yii** 如此强大，它的特点是什么？
3. **Yii** 是如何构建 **MVC** 架构的
4. **Yii** 的一个典型的 **Web** 请求周期和 **URL** 的结构
5. **Yii** 中的对象关系映射和 **Active Record**

第二章：入门

很快你就会发现，真正了解 **Yii** 只需要使用它。在这一章中，我们将讲解一个 **Yii** 应用，更深刻的了解上一章所介绍的 **Yii** 的一些概念。遵循 **Yii** 的约定，我们写一个 **Hello, World** 程序试用这个框架。

在这一章中，我们将介绍：

- § **Yii** 框架安装
- § 创建一个新的应用
- § 创建控制器和视图
- § 添加动态内容到视图文件
- § **Yii** 请求路由并将页面链接到一起

在使用 **Yii** 之前，我们首先需要安装框架，现在让我们开始吧。

安装 Yii

在安装 Yii 之前，你必须配置好你的开发环境，如一台支持 PHP5.1.0 以上版本的 Web 服务器。Yii 已经在 Windows 和 Linux 操作系统上的 Apache Web 服务器测试通过。它可能也会运行在其他平台上的支持 PHP5 的 Web 服务器，互联网上公布了很多免费资源，你可能会获得一个配置好 PHP5 的 Web 服务器环境。在这里我们会抛开 Web 服务器和 PHP5 的安装。

Yii 的安装其实非常简单，实际只需要两个步骤：

1. 从 <http://www.yiiframework.com/> 下载 Yii 框架
2. 解压下载文件到 Web 服务器可访问的目录下。

安装完成后，建议你检查一下当前服务器是否已经满足了 Yii 的所有要求。幸运的是，这样做很容易，Yii 自带了一个简单的检查工具。要调用它，在你的浏览器地址栏中输入：

<http://yourhostname/path/to/yii/requirements/index.php>

在下载 Yii 时，可能会有多个版本让你选择。编写本书时 Yii 最稳定的版本是 1.1.2，虽然大部份的示例代码都应该可以运行在 1.1.x 版本上，但可能部份会有些差别。如果你使用的不同这个版本，请使用 1.1.2 版本完成下面的例子。

下面的图片显示的你在屏幕看到的结果，这就是你服务器的配置：

Yii Requirement Checker			
Description			
This script checks if your server configuration meets the requirements for running Yii Web applications. It checks if the server is running the right version of PHP, if appropriate PHP extensions have been loaded, and if php.ini file settings are correct.			
Conclusion			
Your server configuration satisfies the minimum requirements by Yii. Please pay attention to the warnings listed below if your application will use the corresponding features.			
Details			
Name	Result	Required By	Memo
PHP version	Passed	Yii Framework	PHP 5.1.0 or higher is required.
\$_SERVER variable	Passed	Yii Framework	
Reflection extension	Passed	Yii Framework	
PCRE extension	Passed	Yii Framework	
SPL extension	Passed	Yii Framework	
DOM extension	Passed	CWidlGenerator	
PDO extension	Passed	All DB-related classes	
PDO SQLite extension	Warning	All DB-related classes	This is required if you are using SQLite database.
PDO MySQL extension	Passed	All DB-related classes	This is required if you are using MySQL database.
PDO PostgreSQL extension	Warning	All DB-related classes	This is required if you are using PostgreSQL database.
Memcache extension	Warning	CMemCache	
APC extension	Passed	CApcCache	
Mcrypt extension	Passed	CSecurityManager	This is required by encrypt and decrypt methods.
SOAP extension	Passed	CWebService, CWebServiceAction	
GD extension	Passed	CCaptchaAction	
passed failed warning			

使用检查工具，确定服务器没有安装和使用扩展或组件，但它只是给出一个建议，以确保可以确定安装。正如你看到的，下在的检查结果，并非都是 **Passed**（通过）状态，也有部份显示 **Warning**（警告）。当然，你的配置情况可能会略有不同，因此，你的显示结果也会有所不同。其实下面的细节部份没有必要全部能守。但部份也是必要的，根据 **Conclusion**（结论）这个段落的内容：

你的服务器配置满足了 **Yii** 的最低要求。(Your server configuration satisfies the minimum requirements by Yii.)

安装数据库

在本书中，许多的示例中都将使用到数据库，我们会在用到时给出提示。这能让你更好跟着书中的例子操作，建议你安装一个数据库服务。**Yii** 通过 **PHP** 本身可以支持多种数据库，如果你想使用 **Yii** 内置的数据库抽象层，这需要框架的支持。从 **1.1** 版本开始，支持的数据库有：

§ **MySQL 4.1** 或更高版本

§ **PostgreSQL 7.3** 或更高版本

§ **SQLite 2** 和 **3**

§ **Microsoft SQL Server 2000** 或更高版本

§ **Oracle**

现在我们已经安装了 **Yii** 框架，并已经检查了服务器配置符合要求，让我们开始创建一个全新的 **Yii** 框架的 **Web** 应用程序吧。

创建一个新的应用程序

要创建一个新的应用程序，我们将使用框架附带的一个小工具 **yiic**，这是一个命令行工具，可以使你快速的建立一个全新的 **Yii** 应用。你不是必须要用此工具才能创建 **Yii** 应用，但使用它将节省你大量的时间，并保证文件及目录的结构。

要使用此工具创建 **Yii** 应用，需要打开一个 **shell** 窗口，并进入到系统的一个位置来创建应用程序的目录结构。为了这个演示程序,我们将确保如下要求：

§ **Yii** 的安装位置是你已经知道的

§ **WebRoot** 是你的 **Web** 服务器配置的根目录

§ 从你的命令行，进入到 **WebRoot** 目录，并执行以下内容：

```
§ % cd Webroot

§ % YiiRoot/framework/yiic webapp demo

§ Create a Web application under '/WebRoot/demo'? [Yes|No]

§ Yes

§      mkdir /WebRoot/demo

§      mkdir /WebRoot/demo/assets
```



```
$ mkdir /WebRoot/demo/css
```

```
$ generate css/bg.gif
```

```
$ generate css/form.css
```

```
$ generate css/main.css
```

你的应用已经成功创建到了 **/WebRoot/demo** 下。这个 **webapp** 命令的作用是创建一个全新的 **Yii** 应用。它只需要指定一个参数，无论是绝对还是相对路径都会创建应用程序。它所生成的目录及文件只是应用程序的一个骨架。

现在，我们进入刚刚创建的 **demo** 目录，看看都创建了些什么：

SHELL 代码或屏幕回显：

```
% cd demo %  
  
ls -p  
  
assets/    images/    index.php  themes/  
  
css/       index-test.php  protected/
```

下面是更详细的描述：

SHELL 代码或屏幕回显：

```
demo/  
  
index.php      Web 应用程序的入口文件  
  
index-test.php Web 应用程序的测试入口文件  
  
assets/        包含发送的资源文件  
  
css/           包含 CSS 文件  
  
images/        包含图片文件  
  
themes/        包含应用程序主题文件  
  
protected/     包含所有需要保护的应用程序文件
```

随着在命令行执行一条简单的命令，我们已建立了 **Yii** 框架的目录结构和默认所需的文件。但是，这些文件及目录让你看起来有点害怕。刚开始，我们可以暂时忽视其中的一部分。其实这些文件和目录已经是一个可以工作的 **Web** 应用程序了，用 **yiic** 命令创建的应用程序是一个简单的应用程序。其中 **Contact Us**

(联系我们) 页面是一个典型的表单应用, 并且还自动生成了登陆页面和基于 **Yii** 框架的授权及身份验证功能。如果你的 **Web** 服务器支持 **GD2** 图形库的扩展, 你也会在联系表单项中看到一个 **CAPTCHA** (验证码)。

只要你的 **Web** 服务器正在运行, 你可以打开浏览器访问 <http://localhost/demo/index.php>。你将看到 **Web** 应用程序的首页并显示 **Welcome to My Web Application** (欢迎来到我的 **Web** 应用) 和一些帮助信息, 下面的图片主是这个示例首页的样子:



你会发现这个程序的顶部有一个导航栏, 从左到右分别是: **Home**(首页), **About**(关于), **Contact**(联系), **Login**(登录)。让我从左到右点击一遍。这个示例中, **About**(关于)提供的是一个静态页面, **Contact**(联系)前页已经提到, 提供的是一个表单, 同时包括一个 **CAPTCHA**(验证码)输入框。(如果你没有安装 **GD2** 图形库扩展, 将看不到验证码图片)。

点击 **Login**(登录)链接将转到一个显示登录表单的页面。其实这是一个表单验证代码, 通过用户名和密码进行验证。使用 **demo/demo** 或 **admin/admin** 的用户名/密码组合进行登录。你也可尝试输入任意组合, 将得到一个登录失败的信息。成功登录后, 在顶部原来显示 **Login** 的链接变成了 **Logout(username)**(其中 **username** 可能显示 **demo** 或 **admin**, 这取决于你用哪个用户名登录)。很神奇吧, 你不需要任何编码, 就完成了这么多的工作。

Hello, World!

我们将通过自动生成工具, 完成一个简单且有意义的工作。让我们在新的应用上建立一个本章开始提到 **Hello, World!** 程序上。 **Hello, World** 程序在 **Yii** 中是一个简单的 **Web** 程序, 它发送重要信息到我们的浏览器。

在第一章, 认识 **Yii** 中, 我们已经了解到 **Yii** 是一个 **MVC** 结构的框架。一个典型 **Yii** 的 **Web** 应用程序是等待用户通过浏览器传入一个请求后, 解析该请求的信息, 去查找一个对应的控制器, 然后调用该控制器内的操作方法。该控制器可以调用一个特定的视图, 然后将渲染后的内容返回给用户。在处理数据时, 控制器也可以与模型交互来处理创建、读取、更新和删除 (**CRUD**) 等操作。在这个简单的 **Hello, World!** 示例中, 我们只需要一个控制器和视图, 我们不处理任何数据, 这样将不需要模型。让我们开始创建控制器吧。

创建控制器

和最初开创应用程序一样，这里我们还是调用 **yiic** 命令帮助我们创建一个控制器。在这种情况下，我们将使用 **yiic shell** 命令来启动一个可以与 **Yii** 交互的 **shell**。要启动 **shell**，需进入到应用程序 **demo** 目录，如以下命令：

SHELL 代码或屏幕回显：

```
%cd /Webroot/demo
```

然后接着执行 **yiic** 命令：

SHELL 代码或屏幕回显：

```
%YiiRoot/framework/yiic shell

Yii Interactive Tool v1.1

Please type 'help' for help. Type 'exit' to quit.

>>
```

如果你已经进入到了应用程序目录，你也可以调用相对路径 **protected/yiic** 命令，而不用使用 **Yii** 的安装目录。因此，与上面相同效果的启动 **shell** 命令是：

```
% protected/yiic shell
```

你当前显示的是与 **shell** 交互的提示符。你可以输入 **help** 查看 **shell** 为你提供的所有命令列表：

SHELL 代码或屏幕回显：

```
>> help

At the prompt, you may enter a PHP statement or one of the following commands:

- controller
- crud
- form
- help
- model
- module

Type 'help <command-name>' for details about a command.
```

我们看了有几个可选的命令，有一个 **controller** 命令看起来象是我们想要的，可能是用来为应用程序创建一个控制器的命令。我们可以在 **shell** 提示符下进一步了解 **controller** 命令的更多帮助信息。这些信息包括提供的用法说明，参数描述和一些例子。

SHELL 代码或屏幕回显：

```
>> help controller
```

USAGE

```
controller <controller-ID> [action-ID] ...
```

DESCRIPTION

This command generates a controller and views associated with the specified actions.

PARAMETERS

* controller-ID: required, controller ID, e.g., 'post'.

If the controller should be located under a subdirectory,
please specify the controller ID as 'path/to/ControllerID',
e.g., 'admin/user'.

If the controller belongs to a module, please specify
the controller ID as 'ModuleID/ControllerID' or
'ModuleID/path/to/Controller' (assuming the controller is under a subdirectory of that
module).

* action-ID: optional, action ID. You may supply one or several action IDs.

A default 'index' action will always be generated.

EXAMPLES

* Generates the 'post' controller:

```
controller post
```

* Generates the 'post' controller with additional actions 'contact' and 'about':

```
controller post contact about
```

* Generates the 'post' controller which should be located under the 'admin' subdirectory of the base controller path:

```
controller admin/post
```

* Generates the 'post' controller which should belong to the 'admin' module:

```
controller admin/post
```

在最后两个例子中，使用的命令是一样的，但生成控制器所在的目录不同。**Yii** 是能够检测出 **admin** 是一个模块还是一个子目录。

阅读帮助，很明显看出该命令会生成控制器和操作方法及视图文件。由于我们将要做的应用程序主要是显示一条消息，让我们调用 **controller message** 和一个要显示的操作方法：

SHELL 代码或屏幕回显：

```
>> controller message helloWorld
```

```
generate MessageController.php
```

```
mkdir /Webroot/demo/protected/views/message
```

```
generate helloWorld.php
```

```
generate index.php
```

Controller 'message' has been created in the following file:

```
/Webroot/demo/protected/controllers/MessageController.php
```

You may access it in the browser using the following URL:

<http://hostname/path/to/index.php?r=message>

>>

它已经给出了创建成功的提示，**MessageController** 默认创建到了 **protected/controllers/**目录下。

多么简单的一条命令，我们就创建了一个新的控制器，PHP 文件名是 **MessageController.php**，并放到了控制器目录 **protected/controllers/**中。新创建的 **MessageController** 类是继承应用程序的基类 **Controller**，它的位置是 **protected/components/Controller.php**。这类是继承了框架的基础类 **CController**。因此，**MessageController** 继承了 **CController** 默认的所有行为。既然我们指了一个 **ActionID** 的参数是 **helloWorld**，那么相对的也会在 **MessageController** 中也会创建一个 **actionHelloWorld()** 操作方法。**yiic** 工具像在控制器中定义的操作方法一样，也承担了定义视图的代码。此视图文件与方法的 **id** 的名字一样，叫 **helloworld.php**，此视图文件与控制器关联，默认存放在 **protected/views/message/**下。下面的代码是 **MessageController** 类的内容：

PHP 代码：

```
<?php

class MessageController extends Controller
{

    public function actionHelloWorld()

    {

        $this->render('helloworld');

    }

    public function actionIndex() {

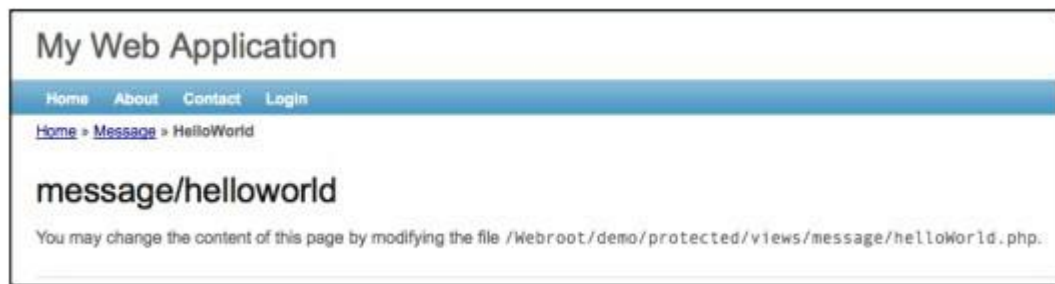
        $this->render('index');

    }

}
```

我们看到，它也增加了一个 **actionIndex()** 方法，并简单渲染一个自动创建的视图文件 **protected/views/message/index.php**。正如第一章中描述的那样，如果只指定了 **controllerID**，但没有指定操作，**Yii** 默认会路由到 **actionIndex()** 方法做进一步处理。**yiic** 太聪明了，知道我们需要一个默认的操作方法。

试着浏览 <http://localhost/demo/index.php?r=message/helloWorld> 地址，你应该看到如下画面：



最后一步

要让应用程序显示 **Hello, World!**，我们需要定制视图文件 **helloWorld.php**。这个很容易就可以做到，编辑 **protected/views/message/helloWorld.php**，修改成如下代码：

PHP 代码：

```
<?php

$this->breadcrumbs=array(

    'Message'=>array(' message/index' ),

    'HelloWorld' ,

);?>

<h1>Hello, World!</h1>
```

保存这段代码，并在再次访问 <http://yourhostname/demo/index.php?r=message/helloWorld>

现在显示问题语的位置改变了，你应该看到如下画面：



我们用很少的代码完成了这个简单的应用，我们仅仅对 **helloWorld** 视图文件修改了一行 **HTML** 代码。

复习一下路由请求

让我们回顾一下运行这个应用程序 Yii 框架是如何分析的：

- 1.通过浏览器输入 Hello, World 地址
<http://yourhostname/demo/index.php?r=message/helloWorld>
- 2.Yii 分析这个 URL，这条路由表标 controllerID 是 message，它将告诉 Yii 应该去请求 MessageController.php 文件，这个文件的位置是 protected/controllers/MessageController.php。
- 3.Yii 还发现，actionID 指定的是 helloWorld，因此，会调用 MessageController 类中的 actionHelloWorld()操作方法。
- 4.actionHelloWorld()方法会渲染 helloWorld.php 视图文件，这个文件的位置是 protected/views/message/helloWorld.php。同时，将这个视图内容返回给浏览器。
- 5.不需要任何配置，这些步骤是连在一起的。按 Yii 的默认约定，整个应用程序的路由请求是无缝的连在一起。当然，如果有必要改变这个流程，Yii 已经给我们提供了这样的接口，但是，如果你使用约定，你将节约配置的时间。

添加动态内容

添加动态内容最简单的方法，就是在视图模板文件中嵌入 PHP 语句。通过视图文件提供的是 HTML 结果，任何普通文本不会改变的。然而，任何在标签之间的代码会作被执行。这是一个典型 HTML 中嵌入 PHP 代码的方法，你可能很熟悉。

添加日期和时间

为我们的页面添加动态内容，并显示日期和时间。再次打开 helloWorld 视图文件，并添加以下代码：

PHP 代码：

```
<h3><?php echo date("D M j G:i:s T Y"); ?></h3>
```

保存，并查看以下 URL 地址：<http://yourhostname/demo/index.php?r=message/helloWorld>

快看，我们已经为应用程序添加了动态内容，每刷新一次页面，将看到显示内容的变化。

不能否认，这并不能令你兴奋，但它确实表明，怎样将 PHP 代码嵌入到我们的视图模板中。

一个更好的添加日期和时间的方法

虽然这种直接嵌入 PHP 代码的做法根本不允许复杂的代码，但 Yii 强烈建议这些语句不改变数据模型，让他们保持简单明了。这将有助于使我们的业务逻辑与视图分离，这是 MVC 架构的一部份。

将数据移到控制器

让我们将创建时间的代码放到控制器中，视图只显示这个时间。我们将代码放到控制器的 `actionHelloWorld()` 方法中，设置一个变量 `$theTime` 保存时间。

1. 首先，让我们改变控制器的操作方法。当前，我们在 `MessageController` 控制中的 `actionHelloWorld()` 中编下如下代码：

```
$this->render('helloWorld');
```

在 `render()` 方法这前，我们添加一段调用时间代码，将时间存入局域变量 `$theTime`。然后我们添加 `render()` 方法的第二个参数，将 `$theTime` 这个变量传给 `render()` 方法：

```
$theTime=date("D M j G:i:s T Y");  
  
$this->render('helloWorld', array('time' => $theTime));
```

调用的 `render()` 方法第二个参数的数据是一个 `array`(数组类型)，`render()` 方法会提取数组中的值提供给视图脚本，数组中的 `key`(键值)将是提供给视图脚本的变量名。在这个例子中，数组的 `key`(键值)是 `time`，`value`(值)是 `$theTime` 则提取出的变量名 `$time` 是供视图脚本使用的。这是将控制器的数据传递给视图的一种方法。

2. 现在让我们在视图中使用这个变量而不使用 `date`(日期函数)。再次打开 `helloWorld` 视图，替换之前添加过的时间代码为：

```
<h3><?php echo $time; ?></h3>
```

3. 保存并查看结果：<http://yourhostname/demo/index.php?r=message/helloWorld>

下面的图片显示了我们这个应用程序的最终看到的结果，**Hello World!**(当然你的日期和时间会有所不同)



我们已经讲解了两种在 **PHP** 视图模板中生成内容的方法。第一种方法把数据逻辑直接在视图文件中创建。第二种方法把数据逻辑放到控制器中，然后提供给视图一个可用变量。两种方法最终的结果是同样的，时间显示在了 **HTML** 页面上。但第二个办法有一个小小的进步，他将业务逻辑的数据与视图分开。**MVC** 架构的目的就是这种分离模式，**Yii** 清晰的目录结构和合理的默认约定使它们连接到了一起。

你有没有注意？

在第一章提到，视图与控制器是非常紧密的兄弟，所以视图文件中的`$this`指的就是渲染这个视图的控制器。

修改前面的示例，在 `MessageController` 中定义一个类的公共属性，而不是局部变量，它是值就是当前的日期和时间。然后在视图中通过`$this`访问这个类的属性。

在前的例子中，我们通过给控制器中 `render()` 方法传递第二个参数，将时间变量传递给视图。第二个参数提取出的变量，可以在视图中使用。但还有另一种方法，我们鼓励你自己去尝试。

把网页连接在一起

典型的网络应用程序展示给用户的页面不会只有一个。虽然我们的示例很简单，但也不会例外。让我们再增加一个页面，与 **Hello, World** 相反的内容，**Goodbye, Yii developer!**，并在两个页面添加互通的超链接。

通常，每个 **Yii** 的 **Web** 应用程序，都对应一个单独的视图文件（尽管不一定总是这样）。因此，我们将创建一个新的视图，并使用一个单独的操作方法来显示这个视图。什么时候添加这个页面呢？我们还需要考虑，是否使用一个单独的控制器。由于 **Hello** 与 **Goodbye** 的页面很相似，所以没有理由将这个应用逻辑放到一个单独的控制器。

连接到新页面

新页面的 URL 如下所示：<http://yourhostname/demo/index.php?r=message/goodbye>

1. 根据 **Yii** 的约定，我们不但需要在控制器中定义操作方法，也需要一个视图名称。因此，打 `MessageController` 控制器文件，在 `actionHelloWorld()` 下面添加 `actionGoodbye()` 操作方法。并在操作方法中添加渲染的视图名称。具体代码如下：

```
2. class MessageController extends CController
```

```
3. {
```

```
4.     ...
```

```
5.
```

```
6.     public function actionGoodbye()
```

```
7.     {
```

```
8.         $this->render('goodbye');
```

```
9.     }
```

```
10.
```

11. ...

```
}
```

12. 下一步，我们要在 **protected/views/message/**目录下创建视图文件，这个视图文件的名字与 **actionID** 相同，叫 **goodbye.php**

13. 打开这个空的模板文件，添加一行代码：

```
<h1>Goodbye, Yi i devel oper!</h1>
```

14. 保存并再次查看：<http://yourhostname/demo/index.php?r=message/goodbye> 应该显示 **goodbye** 的信息。

15. 现在我们需要添加超链接，把两个页面连接起来，若要从 **Hello, World!** 页面添加一个链接到 **goodbye** 页面，我们可以在 **helloWorld** 视图文件中添加一个标签指向 **goodbye**，URL 结构如：

```
<a href="/demo/index.php?r=message/goodbye">Goodbye!</a>
```

请大家记住，这只是个推荐的命名约定。其实视图文件名不必与 **ActionID** 相同，只需要将文件的名字作为第一个参数传递给 **render()** 就可以了。

这当然可以工作，但它仅仅是执行了一个特定 **URL** 结构，这个结构可能在某个时候发生变化，如果 **URL** 结构变化了，那这些链接将会失效。

记得在第一章中，我们介绍 **Blog(博客)** 示例时，使用了一个特别的 **URL** 地址，它的格式更加友好，并且对 **SEO** 优化也不错，如：<http://yourhostname/ControllerID/ActionID> 一个简单的配置，就可以轻易的将现在的 **URL** 结构，改成这种路径格式的。能够改变成这种方法，对应用程序来说非常重要。只要我们避免在应用中对 **URL** 使用硬编辑，那么改变成路径格式将非常简单。

使用 Yii 的 CHtml 得到一些帮助

这种情况可以使用 **Yii** 来处理，**Yii** 提供了许多 **HTML** 助手，可在视图模板中使用。这些 **HTML** 助手都是以静态类的方式存在的。在当前的情况下，如何为应用程序配置 **URL** 结构呢？我们希望能过调用 **CHtml** 的 **link** 方法，**link** 方法需要一个参数是 **controllerID/actionID** 成对出现的字符串，另一个超连接显示的名字。由于所有助手都是静态的，所以我们可以直接调用，而不需要创建它的实例。

1. 使用链接助手，**helloWorld** 视图将会变成：

```
2. <h1>Hello, World!</h1>
```

```
3. <h3><?php echo $time; ?></h3>
```

```
<p><?php echo CHtml::link("Goodbye", array('message/goodbye')); ?></p>
```

4. 保存更改, 并查看: <http://yourhostname/demo/index.php?r=message/helloWorld> 你应该看到这个超链接, 点击它应该转到 **goodbye** 页面。link 助手的第一个参数是链接的显示文本, 第二个参数是一个数组, 值就是 **controllerID/actionID**。结果如下图所示:



5. 我们可以按照同样的方法为 **goodbye** 视图添加链接:

6. `<h1>Goodbye, Yii developer!</h1>`

```
<p><?php echo CHtml::link("Hello", array('message/helloWorld')); ?></p>
```

7. 保存视图文件, 并访问如下 URL 地址:

<http://yourhostname/demo/index.php?r=message/goodbye>

你现在应该能从 **goodbye** 页面看到回到 **Hello, World!** 页面的链接, 下面是 **goodbye** 页面的截图:



小结

在这一章中, 我们创建了一个非常简单的应用, 包括以下几个方面:

- § 如何安装 **Yii** 框架
- § 如何使用 **yiic** 命令创建一个新的应用
- § 如何使用 **yiic** 命令创建一个新的控制器
- § **Yii** 是如何处理你的请求
- § 如何在控制器中创建动态内容, 并将内容显示给浏览器
- § 如何网站内部的超连接

我们已经知道如何将我们的应用程序页面连接在一起。一种方法是在视图文件中添加 **HTML** 的标签以硬编码方法书写 **URL** 结构。另一种（首选）是使用 **Yii** 的 **CHtml** 助手，帮助你建立 **controllerID/actionID** 格式的 **URL** 地址，这样格式的结构，将始终与应用程序的配置关联，即使整个应用程序的 **URL** 地址改变了，我们也不用去修改内部的 **URL**，从而保证了应用程序内部的链接不失效。

通过 **Hello, World!** 这个简单的示例，我们了解到了一些 **Yii** 的配置理念，通过一些默认推荐约定，这个简单的应用（和整个路由的请求过程）的开发是如此的简单和方便。

通过创建这个简单的应用程序，让我们更好的理解怎样使用 **Yii** 框架，它证明了使用 **Yii** 建立应用程序是如此的简单，在下一章，我们将为你介绍一个关于项目的任务管理和问题跟踪的应用程序，这个应用将贯穿本书的其余章节。

第三章：TrackStar(直译：跟踪之星)应用程序

我们可以继续为上面的示例程序添加新的功能，但这样并不会帮助我们了解如何使用框架去开发一个真正的应用程序。为了做到这一点，我们需要建立一个更加紧密的逻辑，与真实的应用非常紧近的程序。

在这一章中，我们将介绍一个项目任务跟踪系统，给它取了个名字叫 **TrackStar**。目前世界上，已经有很多关于项目管理和问题跟踪的应用程序，我们的基本功能将没有什么与众不同。那么，为什么还要建立呢？事实证明，这种基于用户的应用程序有很多的功能，也是很常见的网络应用，这将使我们能够实现两个主要目的：

- § 使用 **Yii** 自带的方便快捷的功能，建立可用的功能并挑战已经存在的其他网络应用。

- § 介绍真实的示例和设计方案，这将帮助你快速建立属于你自己的 **Web** 应用。

介绍 TrackStar

TrackStar 是一个软件开发生命周期（**SDLC**）中问题管理的应用软件。其实主要的目的是帮助保持在整个软件开发过程中所有出现的问题及问题的跟踪。这是一个基础于用户的应用程序，允许管理和创建用户及控制用户权限。一旦用户通过了验证和授权，将可以添加其他用户和管理项目。

TrackStar 可以管理在项目开发中，用户与他人(通常是项目组成员)之间存在问题，如开发任务和一些应用程序的错误(**bug**)，把这些问题，分配给项目组内的其他成员，该项目中的任务将有几个状态如：尚未开始，开始和结束。这样，**TrackStar** 就可以准确的描述一个项目什么时候已经完成，当前正在进行什么，还有什么尚未开始。

创建 User Stories(用户故事)

简单说，**User Stories** 就是应用程序的需求，并且根据应用程序的需求来确定 **User Stories**。**User Stories** 最简单的形式是规定 **User** 可以使用应用程序的哪个部份。它们开始是很简单的，随着程序的复杂性会深入到每一个详细的功能。我们的目标是开始时确定足够的复杂性。如果需要，我们以后将会添加更多的细节和更多的复杂性。

我们之前已经谈到了这个应用程序包括三个主要的功能：用户(**users**)管理，项目(**projects**)管理和问题(**issues**)管理。在应用程序中，这几个主要功能是非常重要的。好了，让我们开始吧。

用户(Users)

TackStar 是一个基于用户的 **Web** 应用，它将有用户类型：

- § 匿名用户

- § 认证用户

一个匿名用户是所有未能通过登录验证的用户。匿名用户只能访问注册或登录，其他受限制的功能需要验证用户。

一个认证用户是所有已经通过登录验证的用户。换句话说，已经认证的用户，他们将可以创建和访问应用程序的主要功能：项目管理和项目的问题管理。

项目(Projects)

在 **TrackStar** 中，项目管理是一个主要的功能。一个项目一般代表公司中高层人员的一个目标，并且由一个或多个人执行。典型的项目可以细分为更具体的任务（或问题），每个小步骤都代表需要完成这个项目的总体目标。

建立一个项目和问题跟踪管理的应用，作为一个例子，它贯穿了整本书。很不幸，我们不能使用它来跟踪我们的程序，因为我们现在还没有开发。但是，假如我们使用一个类似的工具来帮助我们建立项目跟踪，我们可以创建一个项目叫建立 **TrackStart** 项目/问题管理工具。这个工具将项目分解成更细致的项目问题。例如：创建登录页面或设计数据库架构等。

经过身份验证的用户可以创建新的项目。该项目的创建者是这个项目的所有者。项目的所有者可以编辑和删除自己创建的项目，还可以为项目增加成员。除了项目所有者，其他与这个项目关联的用户简称为项目成员，项目成员可以添加新问题以及修改现有的问题。

问题(issues)

项目中的问题可以分为以下三类：

- § 功能(**Features**)：此类别代表真正要实现的功能，例如：“实现登录功能”

- § 任务(**Tasks**)：此类别代表需要做工作，但不是该软件的一个实际的功能。例如，“配置服务器”

- § 错误(**Bugs**)：此类别代表应用程序运行的不是很正常，没有达到预期的功能。例如，“帐户申请一个表单不验证电子邮件的格式”

问题(**issues**)可以有以下三种状态：

- § 尚未开始(**Not yet started**)

- § 已开始

§ 已完成

项目成员不仅可以添加新问题以及修改现有的问题。他们还可以分配问题给自己或其他的项目成员。

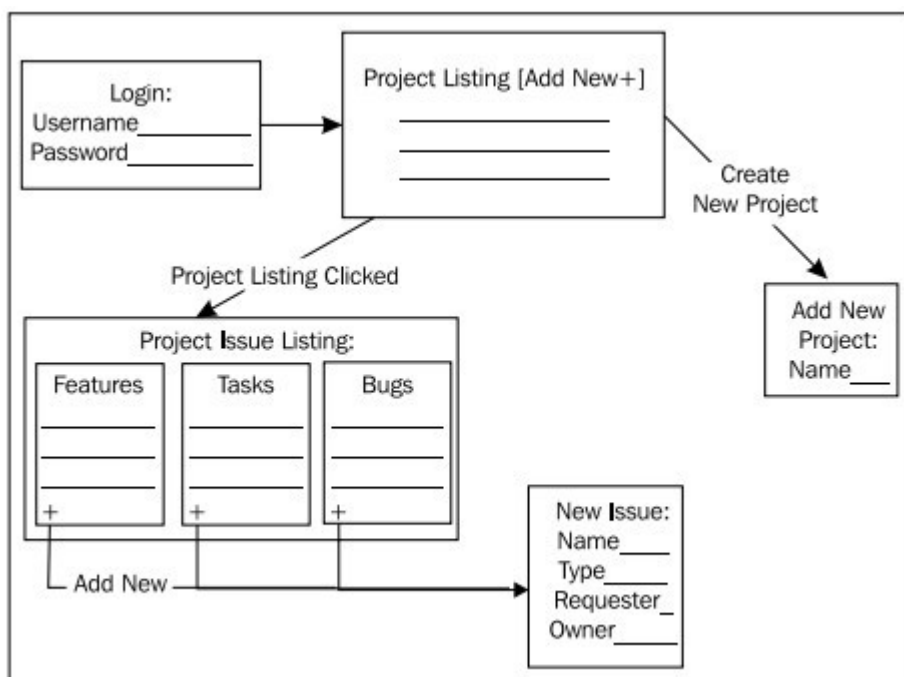
现在，我们了解了这三个主要的模块就可以了，我们将更进一步研究，用户注册的细节以及如果在项目中添加一个任务，我们已经对基本的功能做了一个概述，下面我们将会更加注重细节，因为我们将要实现这些功能。

不过，在我们开始时，我们应该注意一下页面的导航和应用程序的流程。这将帮助我们建立更好的了解总体的布局 and 流程。

导航和页面流程

这是一件能概括应用程序主要页面以及它们是如何结合在一起的工作，这将有助于我们迅速找出所需的 **Yii** 控制器、操作方法和视图，以帮助我们建立期望实现的功能。

下图所显示的是一个基本的想法，描述了这个程序从登录开始到一个项目的细节列表的流程：



当用户第一次访问，必须进登录验证才能访问其他功能。一旦登录成功，用户将看到一个与他相关的项目列表和一个创建新项目按钮，选定一个项目，将转到这个项目的详细信息页面。该项目的详细页面将显示各种类型问题的列表，并且也有添加新问题和编辑其他问题的按钮。

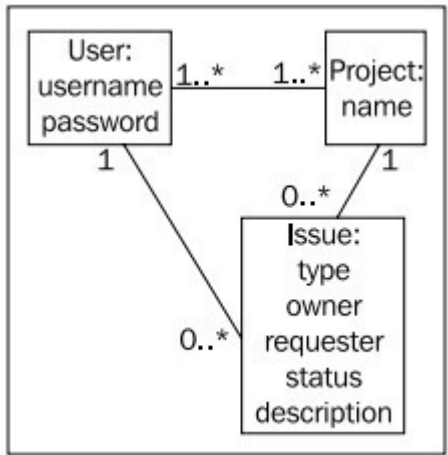
这都是一点非常基本的功能，但这个图片给了我们更多的信息是各各功能怎么样连接在一起，使我们能更好的确定需要的模型、视图和控制器。我们努力的结果是，这张图片可以与他人分享，并使每位参与者都能了解应用程序的流程。以我的经验，当要做一个新的应用程序时，几乎所有人都更喜欢图片而不是繁杂的规则。

定义数据结构

我们下面的工作是需要考虑一些有关的数据的事情，并开始建立规范。如果我们根据主要的功能划分，我们最终会得到一个不错的列表，使用 **Active Record** 建立我们想要的数据库模型。我们根据 **User Stories** 的设计，决定了以下内容：

- § 用户(**User**)
- § 项目(**Project**)
- § 问题(**Issue**)

根据 **User Stories** 的功能设计和工作流程图，我们首先尝试设计所需要的显示数据，如下图：



这是最基本的模型对象，描述了我们主要的数据实体，各自的属性，以及对象之间的一些关系。在 **1..*** 两边的对象，如 **Project**(项目)对象和 **User**(用户)对象之间是多对多的关系。一个用户可以拥有多个项目，一个项目同样也拥有多个用户。同样，一个项目可以有 **0** 个或多个(**Issues**)问题，而与此相关的是 **Issues**(问题)只属于一个项目。此外，用户是所有者（或请求者），可以有多个 **Issues**(问题)，但是 **Issues**(问题)只有一个所有者（也只有一个请求者）。

我们一直在让这些属性保持简单。一个 **User**(用户)只需要用户名和密码，以便可以登录。**(Project)**项目只需要一个名字。

根据我们当前了解，**Issues**(问题)是关联关系最多的。正如之前所说，**Issues**(问题)有一个分类属性，用来区分 **bug**(错误)、**feature**(功能)、**task**(任务)。他们也将有一个状态属性用来说明 **Issues**(问题)的进展情况。用户在系统中刚刚创建 **Issues**(问题) 时，**requester**(请求者就是用户自己)，当把 **Issues**(问题)分配给一个系统中的用户时，他将成为 **Issues**(问题)的所有者。我们还定义了描述属性，用来详细说明这个 **Issues**(问题)。

请注意，我们现在并没有明确的提到关于数据库的结构。事实上，直到现在我们才从数据的角度彻底全面的考虑我们真正需要什么。我们没有使用工具创建并存放这些数据。在操作系统中是否安装了关系数据库？我们是否需要持久化这些数据呢？

对这些问题的回答不需要在计划之初。因为这样可以更好的收集更多我希望和所需要的功能和数据，我们可以与项目相关者讨论要实现的细节，以确保一些想法是我们想要的。项目相关者包括开发人员，产品/项目经理等。他们总会给出一些建议和反馈。

就我们而言，我们实在不能与其他人做这方面的沟通。如果我们能够与你相见，我们一定会与你协商。不幸的是，以书本的这种格式，不允许双向沟通。因为，我们没有其他人来咨询，所以，我们按着之前的方法继续前进。

不过，在我们创建应用程序之前，我们需要了解一下我们的开发方式。我们在开始编码之前，将说明一些开发方式和原则。

确定开发方式

我们将会使用敏捷的开发过程，提高复用性。现代的软件开发中，'Agile'(敏捷)是一个经常提到名词，在开发人员之间有不同的含义。我们在整个开发过程将会集中一个敏捷的方法，包括透明和公开的合作，不断的得到反馈，迅速响应不断变化的需求。

我们不能等到每个细节都确定后再开始编码，我们会逐步的进行。一旦功能的细节确定，我们将开始实现这个功能，其他的功能或细节仍在规划和设计阶段。

在实施中我们会重复的遵循一个规则。我们首先会进行分析和设计，然后试着编写代码、测试代码，并收集反馈意见。设计(**design**)->编码 (**code**)->测试(**test**)->评价(**evaluation**)，遵循这个规则，直到每个人都很满意。一旦大家都很满意，我们就可以将新的功能部署到我们的应用程序中，然后开始收集下一个功能设计，继续遵循这个规则。

自动化软件测试

收集反馈信息是敏捷开发重要的部份。可以从用户、开发团队及所有项目干系人获得反馈。这个开发方式将允许它告诉你在应用程序整合或部署存在的问题。该方法经常由你来编写单元和功能测试得到反馈信息。

单元测试和功能测试

单元测试的编写提供了验证代码是否正确。功能测试的编写提供了对应用程序整体的功能是否正确。

Unit 单元测试

单元测试是软件测试中最小的单位，在面向对象的应用程序中，（如 **Yii** 应用程序）的最小单位是类的接口，公共的方法。单元测试集中在一个单独的类中，而不要求与其它类或对象一起运行。他们的目的是为了验证一个最小单位的代码是否达到预期目的。

功能测试

功能测试重点测试应用程序端对端的功能特性。这个测试相对于单元测试要高一个层次，通常要多个类或对象一起运行。功能测试的目的是验证一个给定的应用程序功能是否可以正常工作。

测试的好处

写单元和功能测试有许多好处。第一可以提供文档。单元测试可以准确迅速地找到一块代码中存在的问题。同样，功能测试文件可以测试应用程序内部的特性是否实现。如果你继续努力的编写这个测试文档，则此文档的变化同样也是应用程序的变化。

测试文档也作为反馈机制，不继地向开发者和项目的其他干系人提供关于应用程序是否达到预期的工作。你每次修改代码后运行你的测试文档并获得即时反馈是否你无意中修改了该系统的其他行为，如果出现问题，立即解决这些问题。这增加了开发人员的信心，让应用程序中的错误减少，使项目更成功。

这样的即时反馈也有助于改进和改善代码的基础设计。开发者可能会改善现有的代码，如果测试文档写的很到位，应用程序的功能改变可以立即得到反馈。编写单元和功能测试文档，可增加开发人员的信息，编写更好的软件，发布更稳定的应用程序及高质量的产品。

测试驱动开发

测试驱动开发(TDD)是一种软件开发方法，它有助于为软件开发创造一个舒适和信心的环境，确保你的测试代码与你的应用程序一起成长，并始终保持最新。它规定在你开始写代码之前先写测试代码。下面是总结的步骤：

- 1.开始写一个失败的测试代码。
- 2.运行测试代码确定它是失败的。
- 3.快速编写你的程序代码，并测试通过。
- 4.再次运行测试代码，以确保它确实通过了。
- 5.重构代码，移除重复的逻辑或改善某些部分，并试着测试通过。

整个开发过程重复这些步骤。

即使再好的意图，如果你迫不及待的写下你的代码之后你可能不会完成测试代码。先写你的测试代码并在书写过程中再编写代码以保证最佳的测试覆盖率。这种深度的覆盖有助于减少项目的压力，虽着应用程序的复杂度建立信心，并不断提供积极的反馈作为补充和作出改变。

为了使用 TDD，我们需要了解 Yii 应用程序中如何测试。

在 Yii 中测试

从 Yii1.1 版本起，Yii 已经紧密结合了 PHPUnit(<http://www.phpunit.de>)和 Selenium Remote Control(<http://seleniumhq.org/projects/remote-control/>)测试框架。特定的测试框架并没有关于 TDD，但非常推荐使用。

当然，你可以使用测试框架测试 **Yii** 的 **PHP** 代码。然而，**Yii** 已经与前面提到的两个框架紧密集成，所以使事情变的更简单。因为简单，所以我们首要目标是使用 **Yii** 的测试功能。

在第 2 章中，当我们使用 **yiic webapp** 命令创建 **Hello, World** 应用程序时，我们注意到，我们创建了许多文件和文件夹。以下是自动测试相关的：

Name of folder	Use/contents
demo/	
protected	This contains protected application files
tests/	This contains tests for the application
fixtures/	This contains database fixtures
functional/	This contains functional tests
unit/	This contains unit tests
report/	This contains coverage reports
bootstrap.php	The script executed at the very beginning of the tests
phpunit.xml	The PHPUnit configuration file
WebTestCase.php	The base class for Web-based functional tests

我们将我们的测试文件放到这个主要目录：**fixtures,functional,unit**。这些报告文件夹用于存储生成的代码覆盖率的报告。

PHP 扩展 **XDebug**，必须安装才能生成报告。推荐使用 **PECL** 安装 **XDebug**。如果详细安装信息，请访问 <http://xdebug.org/docs/install>。如果你想简单的跟随我们的例子，则这些不是必须的。

单元测试

在 **Yii** 中编写单元测试是从框架类 **CTestCase** 中继承一个 **PHP** 类。跟据约定，这个类被命名为 **AbcTest**，其中 **Abc** 可以被替换成测试类的名称。例如，第 2 章的示例程序，如果我们要测试 **Message** 类，则测试类的名称是 **MessageTest**。这个类是保存在 **protected/tests/unit/**下，文件名是 **MessageTest.php**。

测试类中主要包括测试的操作方法，名字是 **testXyz**，其实 **Xyz** 经常与这个测试类相对的类的操作方法名称。

继续 **MessageController** 的例子，如果我们测试 **actionHelloWorld()**方法，则在 **MessageTest** 类中的测试方法是 **testActionHelloWorld()**。

安装 PHPUnit

为了继续往下进行单元测试，您需要安装 **PHPUnit**。应该使用 **Pear** 安装(获得更多关于 **Pear** 的信息，请访问 <http://pear.php.net>)，如 **Mac OS**(苹果操作系统)用户，只需如下两条命令：

SHELL 代码或屏幕回显：

```
% sudo pear channel-discover pear.phpunit.de
```

```
% sudo pear install phpunit/PHPUnit
```

你的配置可能略有不同。了解更多安装过程的信息请访问：

<http://www.phpunit.de/manual/3.0/en/installation.html>

关于 PHPUnit 测试功能使用已经超出了本书范围，建议你花一些时间阅读一下文档

(<http://www.phpunit.de/wiki/Documentation>)，并学习如何编写一个基础本的单元测试。

功能测试

与单元测试一样，功能测试也是编写一个 PHP 类。但它是继承自己 **CWebTestCase** 类，而不是 **CTestCase** 类。跟据约定，如果我们的测试类名叫 **AbcTest**，其中 **Abc** 是被测试的类，则测试类的文件名叫 **AbcTest.php** 只存在 **protected/tests/functional** 中。

为了运行功能测试，你需要安装 **Selenium**。

安装 Selenium

除了 PHPUnit，**Selenium Remote Control Server (Selenium RC)** 是为了运行功能测试所需要的。安装 **Selenium Rc** 非常简单。

1. 从 <http://seleniumhq.org/download/> 下载 **Selenium Remote Control Server (Selenium RC)** zip 文件
2. 解压 zip 文件到你的系统中。

在解压后的目录中，有几个是基础于客户端的目录和一个包含 **Selenium RC Server** 的目录，它的名字有点类似 **selenium-server-1.0.x/**

其中的 **x** 是具体的下载版本。启动服务器也非常简单。只要在命令行下进入服务器文件所在目录，运行以下命令：

SHELL 代码或屏幕回显：

```
% java -jar selenium-server.jar
```

它将在命令行下启动服务。

运行一个简单例子

我们将跟据 **TDD** 方式建立针对 **TaskStar** 应用程序的测试，主要是编写和执行单元测试。但是它现在还不能运行这个功能测试例子。我们在上面的 **Hello, World** 应用中，创建一个 **site** 功能测试文件在 **protected/tests/functional/SiteTest.php**。这个文件中有三个方法。一个用来测试主页 (**Home**)，一个用来测试联系页 (**Contact**)，还有一个用来测试登录 (**Login**) 和注销 (**Logout**)。

在我们运行这个功能测试这前，我们需要修改配置文件。首先，我们需要修改 **protected/tests/WebTestCase.php** 中定义的测试 URL 地址，**Selenium** 将尝试打开这个地址进行测试。找到 **TEST_BASE_URL** 定义的 URL，修改为上一章创建的项目，即以下定义：

PHP 代码：

```
define('TEST_BASE_URL', 'http://localhost/testdrive/index-test.php/');
```

修改为：

PHP 代码：

```
define('TEST_BASE_URL', 'http://localhost/demo/index-test.php/');
```

接下来的变化可能只适用于 **Mac OS**(苹果操作系统)用户。尽管如此，如果你使用的是 **Windows** 系统，但不希望使用 **IE** 浏览器做为测试浏览器，那么你也需要进行修改。修改配置文件 **protected/tests/phpunit.xml** 的 **Selenium** 部份，默认它被配置使用 **IE** 浏览器做为主浏览器。我们可以删除以下突出显示的一行代码，以确保 **Selenium** 运行时只使用 **Firefox**。

SHELL 代码或屏幕回显：

```
<phpunit bootstrap="bootstrap.php"

    colors="false"

    convertErrorsToExceptions="true"

    convertNoticesToExceptions="true"

    convertWarningsToExceptions="true"

    stopOnFailure="false">

    <selenium>

<browser name="Internet Explorer" browser="*iexplore" /> <!-- 这行 -->

        <browser name="Firefox" browser="*firefox" />

    </selenium>

</phpunit>
```

现在，你已经安装了 **PHPUnit**（如未安装请参阅前面的单元测试部分），并且 **Selenium** 服务器也已经运行了（如未运行，请参阅前面的 **Selenium** 部份），接下来我们在命令行下进入到测试目录，运行以下命令进行功能测试：

SHELL 代码或屏幕回显：

```
% cd protected/tests/  
  
% phpunit functional/SiteTest.php
```

这将发生了什么？你将看到浏览器被自动调用，**Selenium** 服务器是使用浏览器访问（如同最终用户访问）我们在 **WebTestCase.php** 中配置的站点。它运行的这个测试方法实际上是自动的模拟网站真实用户的行为。很酷吧。

如果一切正常，最终结果应该显示在命令行窗口，我们执行的测试结果，类似下面的信息：

SHELL 代码或屏幕回显：

```
Time: 19 seconds, Memory: 10.25Mb  
  
OK (3 tests, 10 assertions)
```

开始自动化质量保证测试（**QA** 测试），能够自动完成最终用户的功能测试是一种很好的方式。如果这个项目你有一个独立的 **QA** 团队，你可以向他们演示如何使用这个工具来测试应用程序。如上所述，我们采用测试驱动（**TDD**）的方法，集中编写更多的单元测试，然后最终用户通过浏览器执行功能测试。随后的开发中我们使用一个测试套件来覆盖所有的单元和功能测试。

你运行 **SiteTest.php** 测试时，可能会失败。如果您的测试结果表明有一个位置失败，在 **SiteTest.php** 文件 44 行。这是由于主菜单显示的注销链接的名称，自动生成的测试文件编写的是 **Logout**，而不是 **Logout (demo)**。如果你的功能测试失败，只需更改注销链接，如果你使用的是 **demo/demo** 登录，则应该改成：

```
$this->clickAndWait('Link=Logout (demo)');
```

你好 TDD

让我们简单地温习一下 **Hello World!** 示例，并为这个示例提供了一个 **TDD** 的测试方法。

我们有两个可工作的操作方法，一个显示 **Hello World!** 和一个显示 **Goodbye**。这两个操作方法都在 **MessageController** 类中。

让我们为 **MessageController.php** 添加一个新的操作方法。让这个方法可以返回任何传进来的消息字符串。这听起来很简单，我们只需要在 **MessageController.php** 中添加一个公共的操作方法，可以叫 **repeat()**，并让它完成我们之前所说的功能，这样做对吗？嗯，不太正确，由于我们要采用 **TDD** 方法，所以应该先为这个操作方法编写测试行为。

由于我们要测试这个类的操作方法的行为，所以需要编写一个单元测试。遵从 **Yii** 的默认约定，这个单元测试文件应该放到 **protected/tests/unit/** 目录，并且文件名叫 **MessageTest.php**。让我们放慢步骤，先添加这个类，并让它继承 **Yii** 框架中的单元测试类 **CTestCase**。

创建文件 `protected/tests/unit/MessageTest.php` 并添加如下代码:

PHP 代码:

```
<?php  
  
class MessageTest extends CTestCase  
  
{  
  
}
```

现在, 我们可以在命令行下进行此测试目录, 并执行命令进行测试:

SHELL 代码或屏幕回显:

```
%cd /WebRoot/demo/protected/tests  
  
%phpunit unit/MessageTest.php
```

执行后将显示如下信息:

SHELL 代码或屏幕回显:

```
phpunit unit/MessageTest.php  
  
PHPUnit 3.3.17 by Sebastian Bergmann.  
  
F  
  
Time: 0 seconds  
  
There was 1 failure:  
  
1) Warning(PHPUnit_Framework_Warning) No tests found in class "MessageTest".
```

FAILURES!

Tests: 1, Assertions: 0, Failures: 1.

上面的信息告诉我们，测试失败，没有在 `MessageTest` 类中找到要测试的方法。的确如此，因为我们现在还没有编码。但是我们已经开始了 **TDD** 的第一步，快速编写一个失败的测试用例（尽管有人觉得我们并没有真正的编写测试用例）。

让我们添加一个测试方法，由于我们要写一个用来验证 `MessageController` 类的 `repeat` 操作方法的测试用例，所以这个测试方法的名字叫 `testRepeat`，并添加如下代码：

PHP 代码：

```
class MessageTest extends CTestCase
{
    public function testRepeat()
    {
    }
}
```

如果我们现在重新测试，将看到如下结果：

SHELL 代码或屏幕回显：

OK (1 test, 0 assertions)

当然这是迈向正确的一步。下面我们继续遵循 **TDD**，第二步编写足够的代码可以让测试通过。当然上面的测试用例已经通过了，那是因为此访问并没有测试任何内容，所以并没有用。尽管如此，这是迈向正确的一小步。由于这里并没有真正编写测试用例，下面让我们回到 **TDD** 流程的开始：快速编写一个失败的测试用例。

这一次我们将添加一些具体测试的代码，这些测试代码会测试 `MessageController` 中的 `repeat` 操作方法，为了做到这一点，我们需要：

- 1.在测试方法中，创建一个 `MessageController` 类的实例。
- 2.调用 `repeat` 方法并传给他一串文本。
- 3.验证返回的字符串是否与传入的字符串相同。

现在让我们开始添加代码，`testRepeat()`方法的代码如下：

PHP 代码：

```
public function testRepeat()
```



```

{

    $message = new MessageController('messageTest');

    $yell = "Hello, Any One Out There?";

    $returnedMessage = $message->repeat($yell);

    $this->assertEquals($returnedMessage, $yell);

}

```

我们建一个新的 **MessageController** 类的实例，并提供一个用来验证的 **controllerId** 传给构造函数。然后我们定义一个字符串变量 **\$yell**，并传给 **repeat()** 方法，并将返回结果保存在 **\$returnedMessage** 变量中。

正如我们期望所返回的信息中包含完全相同的字符串。我们使用 **PHPUnit** 的 **API** 方法 **assertEquals()** 用来比较第一个字符串与第二个字符串的结果是否相等(或真或假)。现在这个测试用例已经写完了，让我们试着运行它：

SHELL 代码或屏幕回显：

```
%phpunit unit/MessageTest.php
```

```
PHPUnit 3.4.12 by Sebastian Bergmann.
```

```
E
```

```
Time: 0 seconds, Memory: 10.00Mb
```

```
There was 1 error:
```

```
1) MessageTest::testRepeat
```

```
include(MessageController.php): failed to open stream: No such file or
```

```
directory
```

```
...
```

FAILURES!

Tests: 1, Assertions: 0, Errors: 1.

你可能意识到这又回到了原点，但是，这的确是 TDD 的自然规律性。在正个测试的时间里，我们很快的建立测试用例，并让它先失败再通过。上面的这个错误告诉我们在创建一个 **MessageController** 实例时，并没有在 **classpath** 中找到这个类，所以我们需要在这个测试类中先包含 **MessageController** 这个类。由于这个测试类是应用程序中的一部份，所以我们可以使用 **Yii::import** 语法来包含 **MessageController** 类。

Yii::import 方法允许我们快速包含一个定义的类。它不同与 **include** 和 **require**，因为它很高效。这个定义类开始导入时其实并没有包含，直到它首次被引用时才包含。多次导入同一个定义类也要比 **include_once** 和 **require_once** 快的多。注意：当提到一个 **Yii** 框架内定义类，我们不需要导入或包含它，因为所有 **Yii** 的核心类已经预先导入了。

在 **MessageTest.php** 文件的顶部添加如下代码：

PHP 代码：

```
<?php

Yii::import('application.controllers.MessageController');

class MessageTest extends CTestCase

{
```

保存并运行这个测试，同样也出现了错误，但与之前的错误不同，其实这个错误是告诉我们调用的 **MessageController** 类中没有 **repeat()** 方法。这是对的，因为我们还没有添加这个方法。现在我们创建这个方法并让测试通过。

为 **MessageController** 添加如下方法：

PHP 代码：

```
public function repeat($inputString)

{

    return $inputString;

}
```

保存并再次运行：

SHELL 代码或屏幕回显：

```
% phpunit unit/MessageTest.php
...
OK (1 test, 1 assertion)
```

成功了！我们这个测试通过了。现在我们可以整理一下我们的测试代码，让结构更紧凑些：

PHP 代码：

```
public function testRepeat()
{
    $message = new MessageController('messageTest');
    $this->assertEquals($message->repeat("Any One Out There?"), "Any One Out There?");
}
```

你应该再执行一次测试，以确保它仍然可以通过。

如果你刚刚接触 **TDD**，这一切看起来有点奇怪，特别是要考虑所有细节，有时可能很小的细节只是为了实现这样一个普通的方法。这主要是为了帮助你理解 **TDD** 的流程和节奏。控制这个细节的大小是 **TDD** 的一门艺术。开始的细节可以很小，后面再放大，这样你会感觉很舒服也更加有信心。

提供给开发者几个在测试框架中需要了解的知识。测试结果通过以彩色编码显示。通过测试显示绿色，测试失败显示红色。这种方式也有点像马路上的红绿灯。 **TDD** 常常强调：红(**Red**)，绿(**Green**)，重构(**Refactor**)，重复(**Repeat**)。这是指的基本步骤，如下：

- 1.红色：快速添加一个新的测试用例，然后运行，并看到测试失败。
- 2.绿色：做一个小小的改动，只要能满足测试通过就可以。
- 3.重构：如有必要，删除代码中任何重复的地方，你做这么是否为了让测试能够快速通过，但在其他方面可能也会让你不妥。
- 4.重复：重复第 1 步。

小结

这一章介绍了一个任务跟踪程序 **TrackStar**，我们将会在本书的后面章节讲解如何开发这个程序。我们谈论了这个程序的需求与功能，并用非正式的用户故事的形式提供了一些高层次的需求。然后，我们确定了一些主要的对象，我们将通过这些数据创建应用程序。

我们不仅讨论了将要创建什么，还概述了如何创建。我们讨论了一个基本的敏捷开发思想和方法，并将它应用到了我们自己的程序中。我们还提出了测试驱动开发 (TDD) 为我们将要使用的敏捷开发方法之一。我们不仅讨论了抽象的 TDD 是什么，而且在 Yii 提供的测试框架中如何实现它。

在下一章中，我们终于离开了 **demo** 程序，开始编写这个应用程序 **TaskStar**。

第四章：迭代 1：创建初始 TrackStar 应用

在上一章中，我们介绍了开发方法，这是一个迭代的方法。对我们而言，一次迭代可以看作是一个开发团队从创建项目到测试项目，最终部署到生产环境整个的时间段。开发人员和其他项目干系人决定哪些功能将在这段时间内完成。这本书不能真正的帮我们建立这样一个时间段。因此，我们在每章定义我们自己的迭代。

从现在起，以后的每一章，我们将看作一个新的迭代，并在每章的开始都有一个迭代计划的部份。在这个部份我们将要：

- § 在本次迭代中，把重点放到确定特性和功能。
- § 提供一些必要的前期设计和分析，快速将项目开发任务细化到每一个小项。

迭代计划

在前一章我们了解了什么是任务跟踪系统，我们将要建立这样一个基础 **Web** 的程序。在第二章，我们还了解到如何使用 **yiic** 在命令行下创建一个新的 **Yii** 程序。

在第三章，我们也谈到了数据，但没有特别说明如何处理这些数据。现在就来回答在第三章提出的问题。我们需要持久化数据吗？关系数据库会像文件一样好吗？

一个基于网络的应用程序一般需要存储，检索和操作信息，我们可以有把握地说，我们坚持在这个程序上使用这些性质。同样，存在不同类型的关系数据，我们需要获取与管理这些数据。我们觉得，最好的办法是将数据存储的关系数据库中，**PHP** 开发人考虑到 **Yii** 框架的兼容性，一般使用简单易用，价格低廉的 **MySQL** 数据库。

我们想在首次迭代中，获得一个可以成功连接数据库的应用程序的基本骨架。所以，我们在这次迭代中将关注以下的基本任务：

2

- § 根据 **TaskStar** 创建一个新的 **Yii** 程序
- § 在 **MySQL** 中创建一个新的数据库
- § 配置新的应用程序使它可以连接到新的数据库

在第二章，我们已经看到创建一个新的应用程序是多么的简单，虽然首次迭代时间很短，但在结束时，我们将要做测试和部署 **Web** 应用的工作。

创建应用程序

首先要做的是创建一个最基本 **Yii** 应用程序。在第二章，我们已经了解了，这很容易就可以做到，让我们先确定下列事项：

§ **YiiRoot** 是 **Yii** 框架所在目录

§ **WebRoot** 是你的 **Web** 服务器配置的文档根目录（比如：<http://localhost/>）

下面，在命令行，进入到 **WebRoot** 目录，并执行以下命令：

SHELL 代码或屏幕回显：

```
% cd WebRoot

% YiiRoot/framework/yiiwebapptackstar

Create a Web application under '/Webroot/trackstar'? [Yes|No] Yes
```

上面的执行结果，为我们创建了基本的目录结构和一个 **Yii** 的基本应用程序。访问下面的 **URL**，你应该能够看到新的应用程序：<http://localhost/trackstar/index.php?r=site/index>

连接数据库

现在，我们已经有了一个基本的可运行的应用程序，下面让这个应用程序连接到一个数据库。虽然这个工作主要是配置问题。我们将遵循测试优先的原则，使连接数据库成为我们常规测试的一部份。

测试连接

在第三章中，我们已经了解了 **Yii** 提供的测试框架。所以，我们知道在 **protected/tests/unit/** 下添加一个测试文件，让我们在这个目录下创建一个简单的测试数据库连接的测试文件 **DbTest.php**。并添加如下内容：

PHP 代码：

```
<?php

class DbTest extends CTestCase

{

    public function testConnection()

    {

        $this->assertTrue(true);
    }
}
```

```
}  
  
}
```

在这里，我们添加了一个非常普通的测试代码，`assertTrue()`方法是 **PHPUnit** 中定义的，断言是如果参数为 **true** 将通过，**false** 则失败。因此，在这种情况下，它传的是 **true** 所以这个测试会通过。我们编写测试文件做是为了确保我们新的应用程序工作正常。进入测试目录，并执行这个新的测试：

2

SHELL 代码或屏幕回显：

```
%cd /WebRoot/trackstar/protected/tests  
  
%phpunit t/unit/DbTest.php  
  
...  
  
Time: 0 seconds, Memory: 10.00Mb  
  
OK (1 test, 1 assertion)  
  
...
```

配置测试套件

如果由于某种原因，在你的系统测试失败，你可能需要修改 `protected/tests/bootstrap.php` 中的变量 **\$yiit** 正确指向 `your/YiiRoot/yiit.php`。

相信在新建的 **TrackStar** 应用程序中测试文件已经可以运行了，下面我们可以测试一个数据库连接。

修改 `testConnection` 方法中的 `assertEquals(true)` 语句为如下代码：

PHP 代码：

```
$this->assertNotEquals(NULL, Yii::app()->db);
```

并再次运行测试

SHELL 代码或屏幕回显：

```
%phpunit t/unit/DbTest.php  
  
There was 1 error:  
  
1) DbTest::testConnection
```

```
CDbException: CDbConnection failed to open the DB connection: could not
find driver

...

FAILURES!

Tests: 1, Assertions: 0, Errors: 1.
```

现在有一个失败的测试，这个测试是假设应用程序已经配置了数据库连接组件 **db**。（后面我们将会讨论更多关于应用程序组件问题）。该测试断言，当应用程序与 **db** 连接时，其结果不为空值。

事实上，自动创建的应用程序已经使用了一个数据库，使用 **yiic** 工具生成的应用程序为我们配置了一个 **SQLite** 数据库。你可以查看 **protected/config/main.php** 文件，大约在中间部份有如下声明：

PHP 代码：

```
'db' => array(
    'connectionString' => 'sqlite:' . dirname(__FILE__) . '/../data/yii book.db',
),
```

而你也可以验证 **protected/data/testdrive.db** 是否存在，这个配置是 **SQLite** 数据库所使用。

在我们的例子中，这个测试失败是因为我们的开发环境没有配置 **SQLite** 的驱动。如果你的开发环境有正确的驱动，那么这个测试可能在你的机器上没有失败。我们修改配置使用 **MySQL** 做为数据库服务器，下面让我们简单介绍一下 **Yii** 与普遍使用的数据库。

Yii 和数据库

Yii 提供了与数据库编程的大力支持。**Yii** 数据访问对象(DAO)是建立在 **PHP** 数据对象(PDO)扩展 (<http://php.net/pdo>) 基础之上。这是一个数据库抽象层，应用程序与数据库交互与具体选择的数据库相对独立。所有支持的数据库管理系统(DBMS)使用了统一的接口。通过这种方式，可以保持独立的数据库的代码。应用程序使用 **Yii** 中的 **DAO** 可以很容易地切换使用不同的 **DBMS**。

要建立一个支持 **DBMS** 的连接，可以简单的实例化 **CDbConnection** 类：

PHP 代码：

```
$connection=newCDbConnection($dsn,$username,$password);
```

这里 **\$dsn** 的变量的格式取决于具体被使用程序所采用的 **PDO** 驱动程序。一些常见的格式包括：

§ **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
```

CDbConnection 是继承自 **CApplicationComponent**，这使得它能够被配置为应用程序组件。这意味着我们可以将它添加到应用程序做为组件属性，并在配置文件中自定义类和属性的值。这是我们首选方式。

添加一个数据库连接作为应用程序的组件

当我们刚刚创建基本的程序时，我们指定应用程序类型是一个 **Web** 应用程序。这样实际是指明根据每个请求创建应用程序的单例类 **CWebApplication**，这个单例类例处理整个应用程序范围内的所有请求。其主要的任务是将用户的请求路由到一个适当的控制器中进行下一步处理。关于路由的问题你可以回到第一章了解。这个单例类也可以作为中心地，保存应用程序级别的配置值。

定制应用程序配置，我们通常会提供一个配置文件，当应用程序实例被创建时初始化它的属性，这个主应用程序的配置文件位于 `/protected/config/main.php`。这实际上是一个 **PHP** 文件，包含一个键值对的数组。每个键代表应用程序实例的一个属性，每个值的名称是对应的属性的初始值。如果你打开这个文件，你会看到由 **yiic** 工具为我们配置好的一些设置。

2

在配置中添加一个应用程序组件非常容易，只需要打开主配置文件，然后找到组件的属性。

我们看到，配置文件中已经指定日志和用户的组件。这此将在后面的章节讲到。我们还看到（就像我们前面提到的），有一个 **DB** 组件，配置为使用 **SQLite** 连接到一个 **SQLite** 数据库 `protected/data/testdrive.db`。我们将用一个 **MySQL** 数据库替换这个连接。它的定义如下：

PHP 代码：

```
// application components

'components' => array(

    ...

    'db' => array(

        'connectionString' => 'mysql:host=127.0.0.1;dbname=trackstar_dev',

        'emulatePrepare' => true,

        'username' => 'your_db_user_name',

        'password' => 'your_db_password',
```



```
'charset' => 'utf8',
```

```
),
```

```
),
```

这里假设了本地(127.0.0.1)MySQL 数据库中已经建创建一个名为 **trackstar_dev** 库。其中一个很大的好处是从现在起可以使用这个应用组件，我们可以参考数据库连接作为 **Yii** 主程序的一个属性：**Yii::app()->db** 可以使用在我们应用程序中的任何位置。同样，我们可以将任何组件的配置定义在配置文件中。

我们所有的例子都将使用 **MySQL** 数据库。但是，我们将提供 **DDL** 语句创建表结构，我们尽力让它通用于其它数据库，你可以选择使用与 **Yii** 兼容的数据库。在编写时，**Yii** 有 **Active Record** 支持 **MySQL**, **PostgreSQL**, **SQLite**, **SQL Server**, **Oracle**。

CharSet 属性设置为 **utf8** 字符集用于数据库连接，此属性仅用于 **MySQL** 和 **PostgreSQL**。我们在这里设置，以确保使用正确的 **utf8** 字符支持我们的 **PHP** 应用程序。**emulatePrepare=>true** 这个配置设置一个 **PDO** 属性(**PDO::ATTR_EMULATE_PREPARES**)为 **true**，建议如你使用 **PHP5.1.3** 或更高版本。

因此，我们指定了一个名为 **trackstar_dev** 的 **MySQL** 数据库以及需要连接该数据库的用户名和密码。我们并没有告诉你如何创建这样的 **MySQL**，我们假设你已经有一个喜欢的数据库并知道如何创建一个新的数据库。如果你不确定如何创建一个新的 **trackstart_dev** 数据库以及定义用户名和密码连接数据库，请参阅你的特定数据库文档。

一旦建立了数据库，我们可以再次运行单元测试。

SHELL 代码或屏幕回显：

```
%phpunit t/unit/DbTest.php
```

```
PHPUnit 3.3.17 by Sebastian Bergmann.
```

```
Time: 0 seconds
```

```
OK (1 test, 1 assertion)
```

现在测试已经通过了

小结

我们已经完成了第一个迭代，我们的工作完成了测试应用程序，如果有必要，它可以随时部署。然而，我们的应用程序现在当然做不了很多的事情。我们现在拥有的所有功能，包括一个有效的数据库连接，都是在创建应用程序由 **yiic** 自己动生成的。这当然远远没有达到我们介绍 **TaskStar** 时所描述的功能。但是，我们正在渐渐的实现我们所需要的功能，而首次迭代是实现最终目标的一个伟大的里程碑。

在下一章，我们将终于进入更有意思的功能。我们将开始做一些实际的编码，因为我们需要确实管理我们的项目实体的功能。

第五章：迭代 2：项目(project)的 CRUD

现在，我们已经制定了一个基本的应用程序并配置连接了数据库，现在我们的工作开发一些实用的功能。我们知道项目(**project**)是这个应用程序最根本的组成部份之一。用户首先会在 **TrackStar** 应用程序创建或选择一个已经存在的项目(**project**)并在其中添加任务和问题。出于这个原因，我们想在第二个迭代中集中精力在项目(**project**)这个模块上。

迭代计划

这个迭代相当明了，在这个迭代结束时，我们的应用程序将允许用户创建新的项目(**project**)，在项目(**project**)列表中选择现有的项目(**project**)，更新/编辑现有项目，并删除现有的项目。

为了实现这个目标，我们需要确定具体更细小的任务，下面列出了这次迭代需要完成的所有任务：

- § 设计数据库结构
- § 建立必要的表和所有其他的数据库对象
- § 创建 Yii 的 AR 模型类，允许应用程序能够轻松地与数据库表交互。
- § 创建 Yii 的控制器类，其实中包括的功能：
 - 创建项目(**project**)
 - 取得现有项目(**project**)列表并可以显示
 - 更新现有项目(**project**)的相关数据
 - 删除现有项目(**project**)
- § 创建 Yii 的视图文件，显示的方式和逻辑是：
 - 显示创建新项目(**project**)的表单，并允许创建
 - 显示所有现在项目(**project**)的列表
 - 显示编辑现有项目(**project**)的表单，并允许编辑
 - 在项目列表中添加一个删除按钮，并允许删除项目(**project**)

这些已经足够了，让我们开始吧。我们将很快就把这些任务放到 **TrackStar** 中并管理。现在，我想我们只能先将它们记在记事本上

运行我们的测试套件

在我们进入正式开发之前，我们应该选执行现有的测试套件并确保所有测试全部通过。现在我们只有一个测试，这个测试是在第 4 章中添加的用来验证数据库连接是否有效。所以，不会花太多的时间运行我们的测试套件。打开你的命令提示符，进入 `/protected/tests` 目录，并运行以下单元测试：

SHELL 代码或屏幕回显：

```
%phpunit unit t/  
  
PHPUnit 3.3.17 by Sebastian Bergmann.  
  
Time:  
  
: : 0 seconds  
  
OK (1 test, 1 assertion)
```

随着所有的测试都通过，我们更有信心了。现在我们可以开始进行修改。

创建项目(project)表

早在第 3 章，我们谈论关于一个项目 (**project**) 的基本数据，并在第 4 章，我们决定使用 **MySQL** 关系数据库构建这个应用程序的持久层。现在我们将这个项目 (**project**) 的内容变成一个真正的数据库表。

我们知道，项目 (**project**) 需要有一个名字和描述。我们也将继续保持一些基本信息，跟踪每个记录的创建时间，更新时间以及谁创建的，谁更新的。这些已经足够了，让我们开始达到这个目标。

基于这些所需的属性，如何创建项目 (**project**) 表，如下所示：

SQL 代码：

```
CREATE TABLE tbl_project  
(  
    id INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,  
    name VARCHAR(128),  
    description TEXT,  
    create_time DATETIME,  
    create_user_id INTEGER,  
    update_time DATETIME,  
    update_user_id INTEGER
```

```
);
```

如何使用第三方数据库管理工具，已经超出了本书范围。我们也希望让你跟着使用其他可能使用到的一些其他软件。基于这些原因，我们将简单地提供低级别的数据定义语言(DLL)。创建数据库结构。所以，启动 Yii 支持的数据库服务器，并在 trackstar_dev 数据库中，打开你的数据库编辑器，执行上面的 DLL 语句创建表。

根据你选择使用的数据库，有许多有可用的工具帮助你管理和维护数据库结构，我们建议你使用这些工具，这将使事情变的更加容易。我们实际上是使用 MySQLWorkbench (<http://dev.mysql.com/downloads/workbench/5.1.html>) 进行设计，文档和管理我们的数据库结构。我们也使用 phpMyAdmin (http://www.phpmyadmin.net/home_page/downloads.php) 帮助管理。还有许多类似的工具。花较少时间来熟悉使用这些工具将为你节省很多的时间。

命名规则

你可能已经注意到，我们定义的数据库表名，以及所有的列名都是小写。在我们的开发中，我们将所有表名和列名都使用小写字母。这主要是因为不同的 DBMS 是区分大小写的。举个例子，PostgreSQL 的列名在默认情况下是区分大小写的，但我们在一个查询条件中必须引用一列，如果该列包含大小写字母。使用小写字母将有助于消除这个问题。

你可能还注意到，我们为项目(project)表名使用了一个 tbl_前缀。从 1.1.0 版本起，Yii 提供了使用表前缀的支持。表前缀是一个字符串，它是预先决定表的名称。它通常用在共享主机的环境下，多个应用程序共享同一个数据库的情况，使用不同的表前缀加以区分。例如，一个应用程序使用前缀 tbl_ 而另一个可以使用 yii_。另外，一些数据库管理员把它当成一个命令规则，用来前缀标识数据库对象是什么类型，或使用前缀来进行分组。

在 Yii 中为了采用表前缀支持，必须设置 CDbConnection::tablePrefix 属性为期望的表前缀。然后在整个应用程序的 SQL 语句中，可以使用 {{TableName}} 做为参考表名，其中 TableName 就是表的名称，但不用前缀。例如，如果我们需要修改这个配置，我们仍然可以使用如下代码查询所有项目(project)：

PHP 代码：

```
$sql='SELECT * FROM {{project}}';  
  
$projects=Yii::app()->db->createCommand($sql)->queryAll();
```

但这个问题有点超前了。现在让我们离开我们的配置。重新回到正题。稍后再进入数据库查询。

创建 AR 模型类

现在，我们已经创建了 tbl_project，我们需要创建 Yii 模型类来管理该表中的数据。早在第 1 章，我们介绍了 Yii 的对象关系映射(ORM)层和 Active Record(AR)。现在我们根据应用程序的上下文来看一个具体的例子。

以前，我们使用 **yiic shell** 命令来帮助我们自动生成一些代码。如在第 2 章，我们正是使用 **shell** 命令来创建我们的第一个控制器，还有许多其他的 **shell** 命令可以执行，以帮助自动创建应用程序代码。然而，从 1.1.2 版本起，Yii 有一个新的和更复杂的界面工具 **Gii**。**Gii** 是一个高度可定制和可扩展的基础于 Web 的代码生成平台，把 **yiic shell** 命令提升到了新的高度。我们将使用这个新平台，创建我们的新模型类。

配置 Gii

在我们开始使用 **Gii** 之前，我们必须在应用程序中配置。此时，你可能猜到我们将要编辑我们的主应用程序配置文件 **protected/config/main.php**。是的，要配置 **Gii**，打开这个文件，并添加如下的高亮代码：

PHP 代码：

```
return array(

    'basePath' => dirname(__FILE__) . DIRECTORY_SEPARATOR . '..',

    'name' => 'My Web Application',

    // preloading 'log' component

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

    // autoloading model and component classes

    'import' => array(

        'application.models.*',

        'application.components.*',

    ),

    'modules' => array(

        'gii' => array(

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

            'password' => '[add_your_password_here]',

        ),

    ),

),
```

这为应用程序配置了 **Gii** 模块。我们本书后面章节详细讲解 **Yii** 模块。这里的重点是要确保这些代码添加到了配置文件，并提供你的密码。现在，访问此工具地址：<http://localhost/trackstar/index.php?r=gii>。

下面的屏幕快照显示了验证表单：

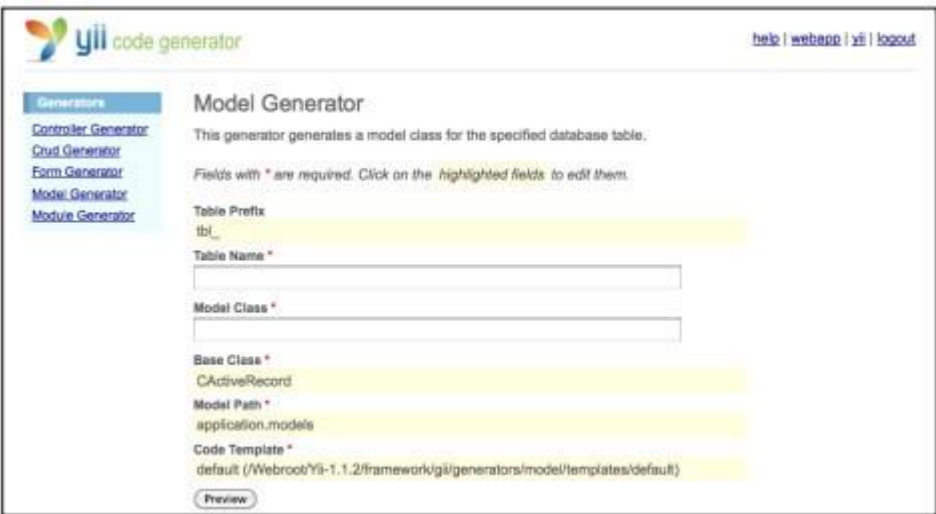


使用 Gii 创建项目(project)AR 类

首先输入您在配置文件中提供的密码，成功后将带你进入 **Gii** 的主菜单页面：



你可能还记得，这些菜单的选项都类似于第 2 章使用 **yiic shell** 命令行工具的帮助信息。因为我们想要创建 **tbl_project** 表的模型，这个模型生成器选项似乎是一个正确的选择。点击链接将带我们进入以下页面：



在创建过程中，表前缀字段域主要用于帮助 **Gii** 确定如何命名 **AR** 类。如果您使用一个前缀，您可以在这里添加。这样，不会使用该前缀来命名新的类。就我们而言，我们使用的是 **tbl_** 前缀，这也恰好是表单的默认值。因此，指定此值将意味着我们新生成的 **AR** 类将被命名为 **Project**，而不是 **tbl_project**。

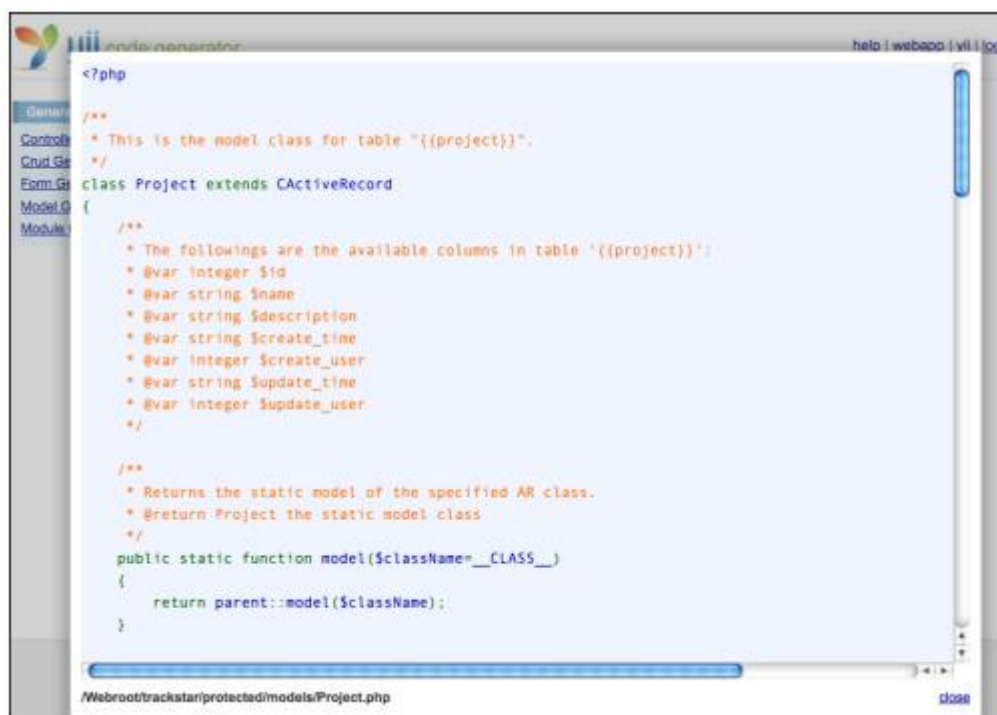
接下来的两个字段域是要求输入表名和我们希望生成的类名。我们表的名称是 **tbl_project**，查看模型类的名称会自动填空。它使用表名，但没有前缀，并以大写字母开头来覆盖模型类的名称。因此，**Project** 为我们模型类的名称，但你也可以自己定义此名称。

接下来的几个字段域是用来进一步进行定制的。**Base Class** 字段用来指定继承哪个模型类。这个类是 **CActiveRecord** 或它的子类。**Model Path** 字段类允许你指定这个模型类文件在应用程序中的位置。默认值是 **protected/models/**(即 **application.models**)。最后一个字段指定使用的生成器的模板。我们可以自定义默认模板以应对其它可能出现的情况，比如所有的模型需要有共同的需求。现在，这些字段的默认值已经可以满足我们的需求了。

点击 **Preview(预览)** 按钮继续，将在页面底部显示如下表格：

<input type="button" value="Preview"/> <input type="button" value="Generate"/>	
Code File	Generate
models/Project.php	new <input checked="" type="checkbox"/>

当你点击 **models/Project.php** 这个链接，可以预览将要生成的代码。下面的截图显示这个预览的样子：

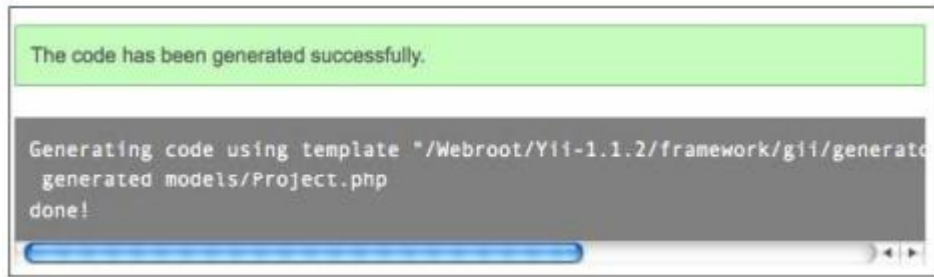


```
<?php
/**
 * This is the model class for table "{{project}}".
 */
class Project extends CActiveRecord
{
    /**
     * The followings are the available columns in table '{{project}}':
     * @var integer $id
     * @var string $name
     * @var string $description
     * @var string $create_time
     * @var integer $create_user
     * @var string $update_time
     * @var integer $update_user
     */

    /**
     * Returns the static model of the specified AR class.
     * @return Project the static model class
     */
    public static function model($className=__CLASS__)
    {
        return parent::model($className);
    }
}
```

它提供了一个可以滚动的弹出窗口，使我们可以预览到整个文件的代码。

好了，关闭这个弹出窗口，点击生成按钮。如果一切顺利，你应该在屏幕底部看到如下画面：



确保 `/protected/models`(或你在 **Model Path** 字段域填写的路径) 是有 **Web** 方式写入的权限。否则，你将收到一个权限错误信息。

Gii 已经为我们创建了一个新的 **AR** 模型类。它的名字是 **Project.php**，默认情况下，放在 `protected/models/` 目录。这个类封装了是我们的 `tbl_project` 表。表中的所有字段访问是通过 **Project AR** 类的属性。

让我们熟悉的方式为新创建的 **AR** 类写一些测试。

测试新生成的代码

一个好的方式是为新代码或功能编写测试。用单元测试的方式来了解 **AR** 类是如何在 **Yii** 中工作的。由于本次迭代的重点是项目 (**projects**) 的创建，读取，更新和删除 (**CRUD**)。我们将针对 **Project AR** 类来编写有关这些操作的测试。在我们新建的 **Project.php** 这个 **AR** 类中，已经包含了这个 **CRUD** 功能的公共方法。所以我们不需要编写这些代码。我们可以只专心来写测试。

创建单元测试文件

首先，需要创建一个新的单元测试文件，让我们创建这个文件在：
protected/tests/unit/ProjectTest.php，并输入如下代码：

PHP 代码：

```
<?php

class ProjectTest extends CDbTestCase
{
    public function testCRUD()
    {
    }
}
```

这个类继承自 **CDbTestCase**，这是 **Yii** 框架中单元测试的基类，最门用来测试与数据库相关的功能。这个特定的基类提供了一些管理操作。下面我们将介绍更多的细节。

我们将使用 `testCRUD()` 方法对 **Project AR** 类进行所有的 **CRUD** 操作测试。我们将开始测试创建一个新项目 (**project**)。

我们并没有真正从事 **TDD**，其原因是我们进行测试的并不是我们所写的代码。我们正在使用 **Yii** 中的 **AR** 类，帮助你熟悉这种测试方法和编写基本的测试。由于不是真正的 **TDD**，我们将不会完全遵循在第 3 章所说的 **TDD** 的步骤。例如，由于我们测试的是 **Yii** 的核心代码，所以我们没有做到先写一个失败的测试。

测试创建(Create)

为 `testCRUD()` 方法添加代码，创建一个 **Project AR** 类，设置它的属性，然后保存它：

PHP 代码：

```
public function testCRUD()
{
    //Create a new project

    $newProject=new Project;

    $newProjectName = 'Test Project 1';

    $newProject->setAttributes(
        array(
            'name' => $newProjectName,
            'description' => 'Test project number one',
            'create_time' => '2010-01-01 00:00:00',
            'create_user_id' => 1,
            'update_time' => '2010-01-01 00:00:00',
            'update_user_id' => 1,
        )
    );

    $this->assertTrue($newProject->save(false));
}
```

此代码首先创建了一个 **Project AR** 类实例，然后，我们使用 **AR** 类的 **setAttributes()** 方法以批量的方式传入数组来设置 **AR** 类的属性。我们看到，类的属性是这个数组的键，具体的数据是这个数组对应键的值。

设置好属性后，我们调用 **Project** 类的 **save()** 方法保存。我们传递给 **save()** 方法一个参数 **false**，用来告诉它绕过属性的数据验证（我们将在数据模型验证部份介绍它，添加自己的验证字段）。然后我们测试返回值，保存成功将返回 **true**。

现在我们切换到命令行下执行这个新的测试，以确保成功：

SHELL 代码或屏幕回显：

```
% cd Webroot/protected/tests

% phpunit unit/ProjectTest.php

PHPUnit 3.3.17 by Sebastian Bergmann.

.

Time:

0:00.00 seconds

OK (1 test, 1 assertion)
```

不错，它通过了。因此，我们已经成功的添加了一个新的项目(**project**)。你可以直接查询数据库来验证。使用你的数据库维护工具，查询项目 (**project**)表返回的内容，你会看到一条新记录的详细属性与 **Project AR** 类的属性匹配。在下面的例子中，我们使用 **MySQL** 命令行查看：

SQL 代码：

```
mysql> select * from tbl_project\G

***** 1. row *****

      id: 1

    name: Test Project 1

description: Test project number one

create_time: 2010-01-01 00:00:00

create_user_id: 1

update_time: 2010-01-01 00:00:00

update_user_id: 1
```

```
1 row in set (0.00 sec)
```

你可能已经注意到，当设置 **Project AR** 类属性时，我们没有指定 **id** 字段。这是因为该列被定义为一个自增长主键。当插入新行时数据库会自动分配个值。一旦插入成功，**AR** 类会自己设置此属性。你可以很容易地获得 **id** 属性，方法如下：

PHP 代码：

```
$newProject->id
```

我们将继续下个测试。

测试读取(Read)

现在，我们已经验证创建测试 **Project AR** 类的 **save()** 方法，让我们继续，添加下面高亮部份的代码，保存在 **testCRUD()** 方法中现有代码的最后。

PHP 代码：

```
public function testCRUD()
{
    //Create a new project

    //READ back the newly created project

    $retrievedProject=Project::model()->findByPk($newProject->id);

    $this->assertTrue($retrievedProject instanceof Project);

    $this->assertEquals($newProjectName,$retrievedProject->name);
}
```

在这里，我们使用静态方法 **model()**，它已经定义在了每一个 **AR** 类中。这个方法返回了 **Project AR** 类的实例，从而进行部访问该类的方法。我们调用 **findByPk()** 方法来查询 **Project** 特定实例。

这种方法（正如你期望的）通过传递主键的值，返回一个相匹配的行。我们传递的是之前新创建的 **Project** 类的实例的自增长属性 **id**。通过这种方式，我们试图当我们保存 **\$newProject** 后，读取插入的这行。我们有两个断言，我们首先确认返回的是否为 **Project AR** 类的实例。然后我们确认项目 (**project**) 的名称是否与读取的项目 (**project**) 名称一致。

让我们再次在命令行下运行如下测试：

SHELL 代码或屏幕回显:

```
% phpunit unit/ProjectTest.php
...
OK (1 test, 3 assertions))) )
```

非常好！我们已经验证了 **CURD** 中的"**R**"是按着我们预期的完成了工作。

接下来让我们快速完成 **Update**(更新)和 **Delete**(删除)的测试。

测试更新(Update)和删除>Delete)

现在，同样在 **testCRUD()** 方法的底部，添加如下代码：

PHP 代码:

```
//Create a new project
...

//READ back the newly created project
...

//UPDATE the newly created project

$updatedProjectName = 'Updated Test Project 1';

$newProject->name = $updatedProjectName;

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

//read back the record again to ensure the update worked

$updatedProject=Project::model()->findByPk($newProject->id);

$this->assertTrue($updatedProject instanceof Project);

$this->assertEquals($updatedProjectName,$updatedProject->name);

//DELETE the project
```

```
$newProjectId = $newProject->id;

$this->assertTrue($newProject->delete());

$deletedProject=Project::model()->findByPk($newProjectId);

$this->assertEquals(NULL, $deletedProject);
```

在这里，我们增加了用来测试项目(**project**)的更新和删除的代码。首先，我们给**\$newProject**实例更新**name**属性，然后保存这个项目 (**project**)。由于这里的 **AR** 实例已经存在，我们知道这个 **AR** 类是更新，而不是插入新记录，当我们调用**->save()**后，我们再读回这行，来确定是否更新了 **name**。

为了测试删除，我们保存**\$newProject**实例的**id**属性值到一个局部变量**\$newProjectId**。然后，我们调用**\$newProject**实例的 **->delete()**方法，你可能猜到了从数据库中删除记录，将销毁 **AR** 实例。然后我们尝试通过这个局部变量作为主键，查询 **Project** 返回这行。由于这条记录已经被删除，我们将得到的结果为 **NULL**。测试断言将使用 **NULL** 比较。

让我们再测运行测试，以确保成功：

SHELL 代码或屏幕回显：

```
% phpunit unit/ProjectTest.php
...
OK (1 test, 8 assertions)
```

因此，我们已经证实了，我们的 **Project AR** 类的 **CRUD** 操作是可以达到预期的工作。

所有测试是真的有必要吗？

当谈到 **TDD** 的软件开发方法时，人们不停地面对测试和时间哪个更重新来作出决定，哪些部份不需要测试。

这些问题必须由你自己来回答。提供足够的测试可使代码质量提高，但很显然测试应用程序中的每一行代码可能有些夸张。

一般的经验规则是，不要为外部库编写测试代码（除非你有特别的理由不信任它）。

我们已经对 **Project AR** 类编写了 **CRUD** 操作的测试，它们背后的是 **Yii** 框架的代码，而不是我们写的代码。我们没有写这些测试，因为我们信任框架的代码，只是想让你了解怎么使用一个 **Active Record**。这项工作只是我们产品中测试套件的一部份，然则，没有必要再去测试我们创建的每一个 **AR** 类，因为，我们不会再对创建的 **AR** 类做这件事情了。

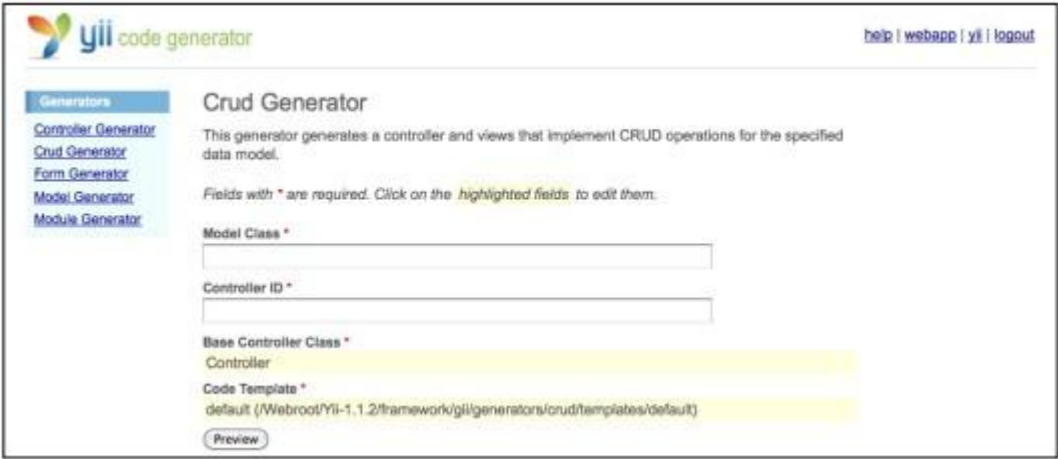
用户允许 CRUD 操作

前面提到的测试向我们介绍了如何使用 **AR** 类。它展示了如何创建新记录，查询现有的记录，更新现有的记录，并删除现有记录。我们花了很多时间测试 **Project** 表的 **AR** 类的一些低级别的操作。但我们的 **TrackStar** 应用还没有公开这些功能给用户 (**user**)。我们真正是需要让用户 (**user**) 来创建，读取，更新和删除项目 (**project**)。现在，我们知道了一些围绕 **AR** 操作的方法，我们可以在控制器中开始编写一些功能代码。幸运的是，我们不用做。

通过脚手架为项目创建 CRUD

让我们再次进入 **Gii** 代码生成工具帮我们生成一个繁琐，耗时又很常见的代码。**Yii** 为我们提供这个明确的方式，在应用程序中开发者使用共同的 **CRUD** 进行数据库操作。如果你已经熟悉了其他框架，你可能知道脚手架这个词。让我们来看看在 **Yii** 中利用它的优势。

访问 <http://localhost/trackstar/index.php?r=gii>，并选择 **Crud Generator** 链接。你将看到如下画面：



The screenshot shows the 'yii code generator' interface. On the left, there's a sidebar with 'Generators' listed: Controller Generator, Crud Generator (selected), Form Generator, Model Generator, and Module Generator. The main area is titled 'Crud Generator' and contains the following text: 'This generator generates a controller and views that implement CRUD operations for the specified data model.' Below this, it says 'Fields with * are required. Click on the highlighted fields to edit them.' There are four input fields: 'Model Class *' (empty), 'Controller ID *' (empty), 'Base Controller Class *' (set to 'Controller'), and 'Code Template *' (set to 'default (Webroot/Yii-1.1.2/framework/gii/generators/crud/templates/default)'). A 'Preview' button is at the bottom left of the form area.

在这里，我们看到两个表单字段域。第一个是要求我们指定模型类，我们希望生成所有的 **CRUD** 操作。在我们的例子中，使用 **Project.php** 这个 **AR** 类。所以在这个字段域中填写 **Project**。当我们输入后，我们注意到，**Controller ID** 字段域自动填充成了 **project**，这是基于 **Yii** 的约定。我们现在保持默认即可。

这两个字段域填写后，点击预览按钮，将会在页面底部看到如下表格：

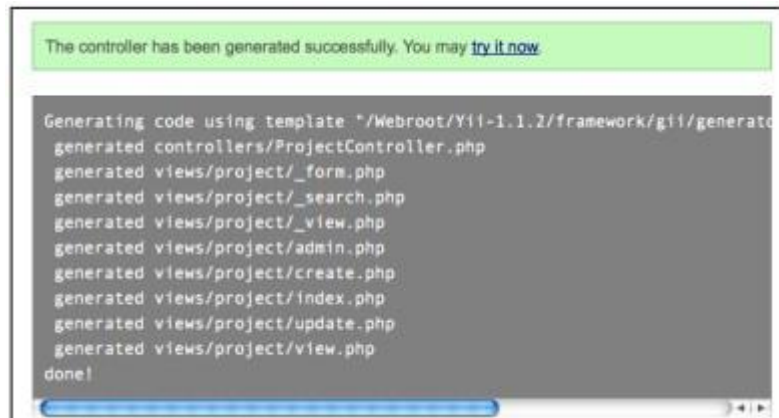


The screenshot shows a table with two columns: 'Code File' and 'Generate'. The table lists the following files:

Code File	Generate <input type="checkbox"/>
controllers/ProjectController.php	new <input checked="" type="checkbox"/>
views/project/_form.php	new <input checked="" type="checkbox"/>
views/project/_search.php	new <input checked="" type="checkbox"/>
views/project/_view.php	new <input checked="" type="checkbox"/>
views/project/admin.php	new <input checked="" type="checkbox"/>
views/project/create.php	new <input checked="" type="checkbox"/>
views/project/index.php	new <input checked="" type="checkbox"/>
views/project/update.php	new <input checked="" type="checkbox"/>
views/project/view.php	new <input checked="" type="checkbox"/>

我们可以看到将要生成很多的文件，其实包括一个 **ProjectContrller.php** 控制器类文件（它的包括了 **CRUD** 的所有操作方法）和许多的视图文件，每一个单独的视图文件对应每一个操作同时提供了可以搜索项目 (**project**) 记录。你当然也可以通过去掉复选框的选择来不生成一些相应文件。然后，对我们而言，我们很喜欢 **Gii** 为我们创建的这些文件。

继续并单击 **Generate(生成)** 按钮。我们应该在页面底部看到如下画面：

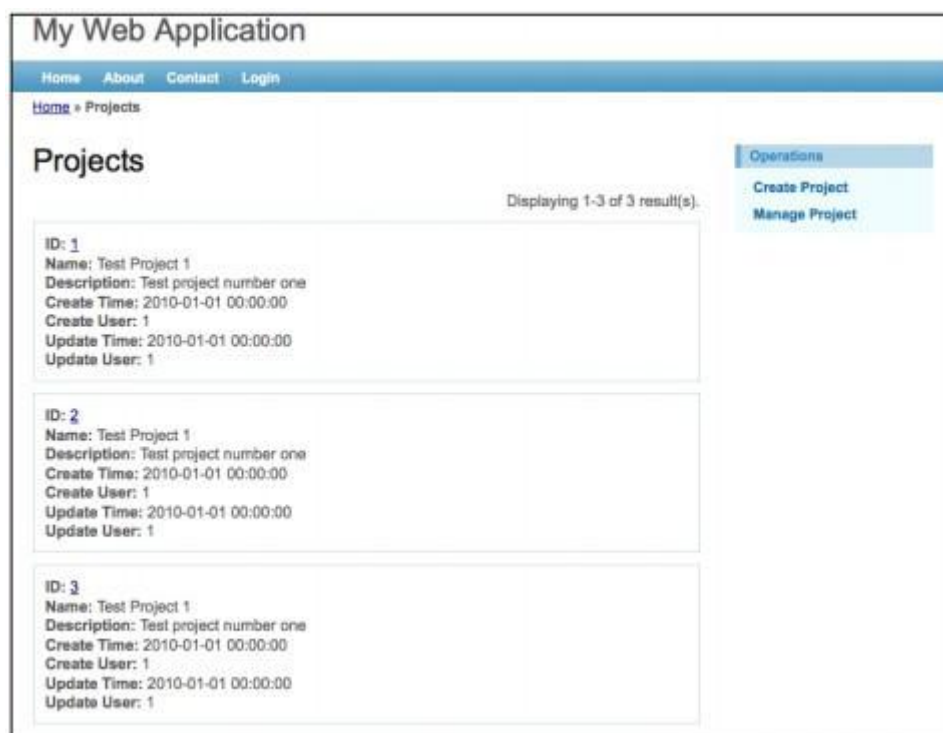


你可能需要确保 **Web** 服务器可以对 **/protected/controllers** 和 **/protected/views** 两个目录有可写权限，否则，你将收到权限错误消息，则不是成功的结果。

现在，我们可以点击 **try it now** 链接到这个页面去测试新功能了。

首先会带你进入项目 (**project**) 的列表页面。这个页面显示了当前系统中所有的项目 (**project**)。你可能现在还看不到任务项目 (**project**)，因为我们还没有创建新的项目 (**project**)。然而，我们的项目 (**project**) 列表页面显示了几个项目 (**project**)，如下图所示：（访问地址：

<http://localhost/trackstar/index.php?r=project>）



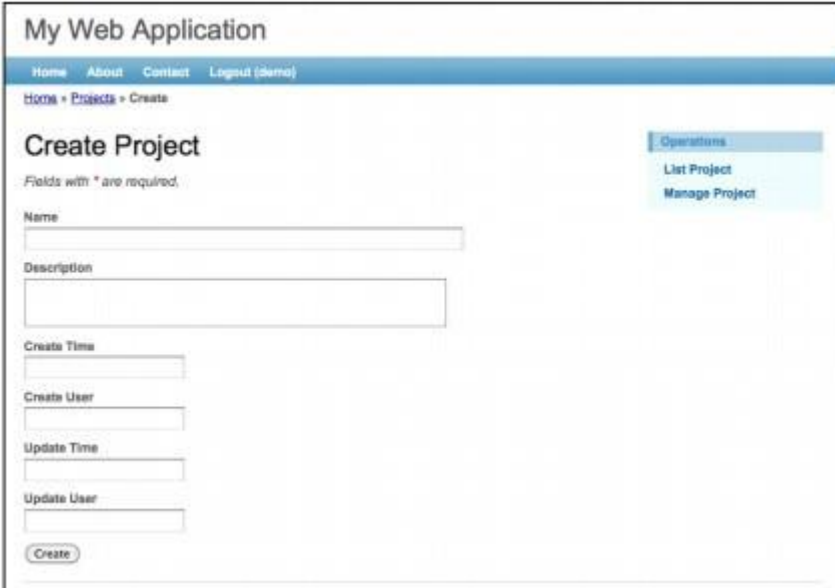
但是，这些项目是从何而来呢？你的项目 **(project)** 列表与截图相比可能多于或少于 3 列，这取决于你在之前执行单元测试时的次数。我们在单元测试中为 **Project AR** 类编写的 **CRUD** 操作，实际上每次运行会在数据库中创建新记录。这些记录被创建出来时，我们还没有完成我们的测试代码，最终的测试代码是删除被创建的记录。在这个特殊情况下，这已经有一些项目 **(project)** 了，这样我们就可以看清它们如何被显示出来的。然而，一般情况下，运行单元测试或功能测试使用开发数据库这种方式很不好。所以在后面，我们将讨论如何改变这些测试，让它们运行在一个单独的测试数据库。现在让我们继续试用新生成的代码。

创建一个新项目(project)

你可能发现在项目 **(project)** 列表页(前面的那张截图)的右侧有一个小导航栏。点击 **Create Project** 链接。你会发现会把你们带到登录页面，而不是一个创建新项目 **(project)** 的表单。其原因是在 **Gii** 中定义了生成代码的规则，只有验证用户（即登录用户）可以创建新的项目 **(project)**。任何匿名用户尝试访问创建新项目 **(project)** 页面都会被重定向到登录页面。好的，让我们使用 **demo** 作为帐号与密码进行登陆。

登陆成功后，你应该重定向到下面的网址：<http://localhost/trackstar/index.php?r=project/create>

此页面显示了一个用来添加新项目 **(project)** 的表单，如下图所示：



The screenshot shows a web application interface titled "My Web Application". At the top, there is a navigation bar with links: Home, About, Contact, and Logout (demo). Below this, a breadcrumb trail reads "Home > Projects > Create". The main heading is "Create Project". A note states "Fields with * are required". The form contains the following fields: "Name" (text input), "Description" (text area), "Create Time" (text input), "Create User" (text input), "Update Time" (text input), and "Update User" (text input). At the bottom left of the form is a "Create" button. On the right side, there is a sidebar with a section titled "Operations" containing two links: "List Project" and "Manage Project".

让我们快速填写此表单来建立一个新项目 **(project)**。虽然这个表单中没有必填字段，让我们在 **Name** 字段填写 **Test Project**，在 **Description** 字段填写 **Test project description**。点击 **Create** 按钮将数据发送给服务器。如果出现错误，会显示错误信息，并且出错字段域会高亮显示。成功保存会重定向到新创建的项目 **(project)** 的具体列表。我们的操作成功了，页面被重定向到

<http://localhost/trackstar/index.php?r=project/view&id=4>，如下图所示：



正如前面提到的，有一件事是关于我们新创建的项目 (**project**) 表单中所有字段域没有标记为必填，我们可以不输入任何数据就能提交成功。但是，我们知道，每个表单至少需要一个必填字段。让我们设置一个必填字段。

为表单添加必填字段域

在 **Yii** 中当表单与 **AR** 模型类交互时，设置一个验证规则来限制字段域的范围。这需要在 **Project AR** 模型类的 **rules()** 方法中，添加一个数组，数组中包含特定的值。

打开 `/protected/models/Project.php` 类，已经看到了公共的 **rules** 方法被定义了，并且在 **rules** 方法中已经存在了一些规则：

PHP 代码：

```
/**
 * @return array validation rules for model attributes.
 */

public function rules()
{
    // NOTE: you should only define rules for those attributes that will receive user inputs.

    return array(

        array('create_user_id, update_user_id', 'numerical', 'integerOnly' => true),

        array('name', 'length', 'max' => 128),

        array('create_time, update_time', 'safe'),

        // The following rule is used by search().
    );
}
```

```
// Please remove those attributes that should not be searched.

array('id', 'name', 'description', 'create_time', 'create_user_id',

      'update_time', 'update_user_id', 'safe', 'on'=>'search'),

);
```

rules()方法返回的是一个规则数组，一般每一个规则格式如下所示：

PHP 代码：

```
Array('Attribute List', 'Validator', 'on'=>'Scenario List', ...additional options);
```

Attribute List(属性列表)是一个字符串，需要验证的类的属性名用逗号分开。**Validator**(验证器)指的是使用什么样的规则执行验证。**on** 这个参数指定了一个 **scenario**(情景)列表来使用这条验证规则。

scenario(情景)允许你限制验证规则应用在特定的上下文中。一种典型的例子是 **insert**(插入)或 **update**(更新)。例如：如果被指定为 **'on'=>'insert'**，这将表明验证规则只适用于模型的插入情景。这同样适用于**'update'**或其它的任何你希望定义的情景。你可以设置一个模型的 **scenario**(情景)属性或能通过构造函数传给一个模型的实例。

如果这里没有设置，该规则将适用于调用 **save()**方法的所有情景。最后，**additional options**(附加选项)是 **name/value**(键值对)出现的用来初始化 **validator**(验证器)的属性。

validator(验证器)可以是模型类中的一个方法或一个单独的验证器类。如果定义为模型类中的方法，它的格式必须是如下的形式：

PHP 代码：

```
/**
 * @param string the name of the attribute to be validated
 *
 * @param array options specified in the validation rule
 */
public function ValidatorName($attribute, $params) { ... }
```

如果我们使用一个 **validator**(验证器)类，则这个类必须继承 **CValidator**。其实有三种方法可以指定 **validator**(验证器)，包括前面提到的一种格式：

- 1.第一种是在模型类中定义验证方法
- 2.第二种是指定一个单独的验证器类（这个类继承 **CValidator**）。
- 3.第三种是你可以使用 **Yii** 框架中现有的验证器，指定预定义的验证器别名即可。

Yii 为你提供了很多预定义的验证器类，同时也指定了别名，用在定义规则时。Yii1.1 版本，预定义的验证器别名的完整列表如下：

- § **boolean**: 它是 **CBooleanValidator** 类的别名，验证属性的值是布尔值(**true** 或 **false**)。
- § **captcha**: 它是 **CCaptchaValidator** 类的别名，验证属性的值等于一个显示的 **CAPTCHA**(验证码)的值。
- § **compare**: 它是 **CCompareValidator** 类的别名，验证属性的值与另一个属性的值相等。
- § **email**: 它是 **CEmailValidator** 类的别名，验证属性的值为有一个有效的 **Email** 地址。
- § **default**: 它是 **CDefaultValidator** 类的别名，验证属性的值为分配的默认值。
- § **exist**: 它是 **CExistValidator** 类的别名，验证属性的值在表中的对应列中存在。
- § **file**: 它是 **CFileValidator** 类的别名，验证属性的值包含上传的文件。
- § **filter**: 它是 **CFilterValidator** 类的别名，用过滤器转换属性的值。
- § **in**: 它是 **CRangeValidator** 类的别名，验证属性值在一个预定义列表中。
- § **length**: 它是 **CStringValidator** 类的别名，验证属性值的长度在一个范围内。
- § **match**: 它是 **CRegularExpressionValidator** 类的别名，验证属性值匹配一个正则表达式。
- § **numerical**: 它是 **CNumberValidator** 类的别名，验证属性值是数字。
- § **required**: 它是 **CRequiredValidator** 类的别名，验证属性值必需有值，不能为空。
- § **type**: 它是 **CTypedValidator** 类的别名，验证属性值是一个指定的数据类型。
- § **unique**: 它是 **CUniquedValidator** 类的别名，验证属性值在表中的对应列中是唯一的。
- § **url**: 它是 **CUrlValidator** 类的别名，验证属性值是一个有效的 **URL**。

因为我们想使项目(**project**)的 **name** 属性字段必填，这个看着好像应该使用 **required** 别名可以满足我们的需要。让我们添加一验证规则指定 **name** 属性的验证器为 **required** 别名。我们将追加到现有的规则中：

PHP 代码：

```
public function rules()
{
    // NOTE: you should only define rules for those attributes that will receive user inputs.

    return array(

        array('create_user_id, update_user_id', 'numerical', 'integerOnly' => true),

        array('name', 'length', 'max' => 128),

        array('create_time, update_time', 'safe'),

        // The following rule is used by search().
```

```
// Please remove those attributes that should not be searched.

array('id', 'name', 'description', 'create_time', 'create_user_id',
update_time', 'update_user_id', 'safe', 'on'=>'search'),

array('name', 'required'),

);

}
```

保存项目(**project**)模型文件,并再次访问:[http://localhost/trackstar/index.php?r=project /create](http://localhost/trackstar/index.php?r=project/create), 我们看到在字段名称旁边有一个红色星号。这表明现在这个字段域是必需填写的。尝试不填写任何信息提交表单,你应该看到一个错误消息。指出这个字段域不能为空,如下图所示:

虽然我们已经做了一些改变,让我们继续为 **Description** 字段添加 **required** 验证。我们只需将 **Description** 字段添加到之前指定的那条规则中,如下所示:

PHP 代码:

```
array('name', 'description', 'required'),
```

这样,我们看到,我们可以在 **Attribute List** (属性列表处) 指定多个字段域,用逗号分隔。这样,我们声明的形式表明,无论是 **name** (名称)或 **description**(描述)属性都是必需有值的。尝试不填写任何信息提交表单,你应该看到一个错误消息。

如果我们在创建表时,规定 **name** 和 **description** 列为 **NOT NULL**,则当我们使用 **Gii** 代码生成器创建模型类型时,这个规则将会帮我们自动生成。它会在表中列的定义的基础上自动添加规则。例如:列包含了 **NOT NULL** 约束将会添加 **required** 验证器,另外,列有长度限制,比如 **name** 列定义为 **varchar(128)**,将会自动添加字符限制的规则。我们注意到,在 **Project AR** 类的 **rules()**方法中, **Gii** 自己动创建了一条规则

```
array('name', 'length', 'max'=>128)
```

读取项目

查看我们新项目(**project**)详细信息:

<http://localhost/trackstar/index.php?r=project/view&id=4>, 那么, 这基本上就是 **CRUD** 中的"**R**", 但是, 要查看整个列表, 我们可以点击右侧的 **List Project** 链接。这让我们回到了项目(**project**)列表页面, 其实包含了我们新创建的项目(**project**)。所以, 我们可以读取所有的项目 (**project**), 以及每个项目的细节。

更新和删除项目

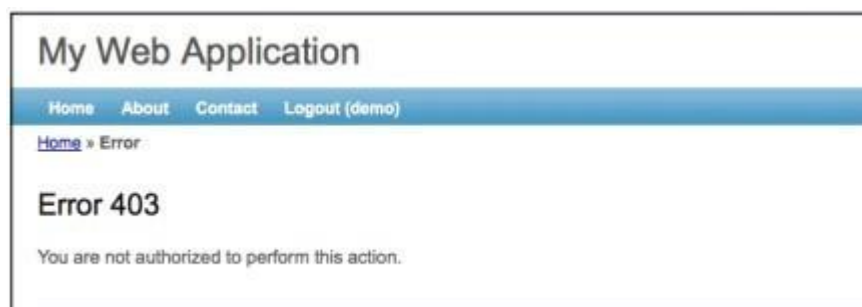
访问一项目的详细信息页面可以在项目的列表中点击每个项目的 **ID** 链接。让我们在项目列表中选择这个新创建的项目 **ID: 4**。点击这个链接将把我们转到此项目的详细页面。本页面的几个操作(**Operations**)动作列表在页面的右侧, 如下图所示:



我们看到了更新项目(**Update Project**)和删除项目(**Delete Project**)的链接分别指的是 **CRUD** 操作中的 "**U**"和"**D**". 我们将会离开这个部份, 你可以去验证这些链接所做的工作。

在 admin 模式中管理项目


最后, 我们讲一下前面没有提到的项目操作——管理项目(**Manage Project**)。点击这个连接, 它可能看到一个授权错误, 如下图所示:











此错误的原因是当我们登陆应用程序创建项目时, 我们使用的是 **demo/demo** 的用户名/密码。由于这些代码是由 **Gii** 生成的, 它限制了管理项目权由管理员可以访问。

进入管理员模式, 仅仅需要使用 **admin/admin** 的用户名/密码组合。来吧, 点击页面顶部导航上的 **Logout(demo)** (注销 **demo** 帐号)。然后重新登陆, 但这次使用管理员帐号。如果使用 **admin** 登陆成功(你可以通过页面顶部导航上的 **Logout(admin)**验证是否成功)。访问一个特定的项目页, 列如:

<http://localhost/trackstar/index.php?r=project/view&id=4>，并尝试再次点击 **Manage Project** 链接。我们现在应该看到类似下图所示的页面：



ID	Name	Description	Create Time	Create User	Update Time	
1	Test Project 1	Test project number one	2010-01-01 00:00:00	1	2010-01-01 00:00:00	 
2	Test Project 1	Test project number one	2010-01-01 00:00:00	1	2010-01-01 00:00:00	 
3	Test Project 1	Test project number one	2010-01-01 00:00:00	1	2010-01-01 00:00:00	 
4	Test Project	Test project description	0000-00-00 00:00:00		0000-00-00 00:00:00	 

我们现在所看到的是一个交互度更高的项目列表页面。它在一个可交互的数据表格中显示了所有项目。每一行都内置了查看、更新和删除这个项目的超链接。点击一列的表头上的链接会将按这一列值进行排序。第二行的一些输入框允许你按关键词搜索这个项目中这个列的值。**Advanced Search**(高级搜索)链接展示了一个搜索表单，并提供了多个搜索条件。下图展示了高级搜索表单：



Advanced Search

ID

Name

Description

Create Time

Create User

Update Time

Update User

在本次迭代中我们基本实现了目标提到的所有功能。但并没有写太多的代码。实际上，是在 **Gii** 的帮助下我们实现目标中没有期望到的基于项目 (**project**) 的搜索功能。虽然基本，但我用很少的代码完成了项目任务跟踪系统中的一个特有的功能。

但是现在尚不是搁置的时候。脚手架的代码并没有真正完全替代应用程序开发。相反，它的会帮助我们建立真正的应用程序。当我们通过了解项目的功能如何工作的所有细节和差别，我们可以依靠自动生成的代码保持项目前进。我们将一样可以根据项目的求要，继续前进，但这些由脚手架自动生成的代码没有完成所有的功能，我们需要一个应用程序中管理项目 (**project**) 的完整的解决方案。

更多与测试有关的--夹具(fixture)

在我们继续为 **TrackStar** 应用程序添加新功能前，我们需要再次配置我们的测试。正如我们前面所讨论的，我们的单元测试实际上实现了在开发环境中为应用程序添加新项目 (**project**)，此外，我们完成了创建项目后并删除的测试，数据库将重复使用与插入相同的 **ID** 标识符。所以当我们继续执行我们的测试，我们将发现我们的项目 **ID** 越来越大（这可能与正常的开发产生混淆）。

这个问题是由于创建新项目时单元测试与 **web** 表单形式针对的是同一个数据库。因此，存在一些潜在的问题。我们需要做的是为测试环境配置一个独立的镜相数据库。它只专注于测试。我们也需要一种方式可以确保我们的测试对针相同的数据总是以相同方式运行。前者是通过配置文件很容易的就可以改变。后者是通过一个使用一个夹具(**fixtures**)完成。

一个测试夹具是指在测试运行中的一个系统状态或上下文。我们想多次运行我们的测试，每次运行我们希望能够让他们可以返回重复的结果。夹具的目的是提供一个众所周知在固定的环境中运行验证。通常情况下，一个夹具的主要工作是确保所有参与测试的对象在测试运行中初始化一个指定的状态。一个典型的夹具例子是使用固定和已知的数据加载一个数据库的表。

在 **Yii** 的 **PHP** 文件中夹具返回一个数组指定初始数据配置。它们通常的命名与它们所代表的数据库中表的名称相同，并保存在 **protected/tests/fixtures/** 目录下。因此，要指定项目的夹具数据，我们将要在此目录下创建一个名叫 **tbl_project.php** 文件。此文件包含固定和已知的用来初始化在 **/tests/unit/ProjectTest.php** 文件中进行测试的数据库中项目表的数据。这个夹具文件是指定在 **ProjectTest.php** 测试文件顶部：

PHP 代码：

```
class ProjectTest extends CDbTestCase
{
    public $fixtures=array
    (
        'projects' => 'Project' ,
    );
}
```

配置 CDbFixtureManager

测试过程中，建立这些类型的数据库夹具可能是一个非常耗时的部份。**Yii** 来了，它提供了 **CdbFixtureManager** 类再次把我们从这个乏味的程过拯救出来。当前我们配置一个应用组件时，它将提供以下功能：

- § 运行所有测试之前，它重置所有相关数据表为一个已知状态数据。
- § 在运行单个测试之前，它能重置指定数据表为一个已知状态数据。
- § 在执行一个测试其间，它提供了访问固定状态数据一部份，即数据行。

要使用夹具管理器，我们需要配置应用程序中的配置文件。这实际上当我们创建初始应用程序时已经创建了。如果你打开了应用程序的指定的测试配置文件 **protected/config/test.php**。你将看到如下的应用组件定义：

PHP 代码:

```
fixture'=>array(

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

),
```

因此应用程序已经配置了并使用了夹具管理器。现在我们需要创建一个新夹具。

创建一个夹具(fixture)

在 Yii 中一个夹具是在一个 PHP 文件中返回一个数组用来初始化特定表的数据行。文件名同表名相同。默认情况下, 这些夹具文件将放置在 **protected/tests/fixtures** 目录下。如果需要可以在应用程序配置中你可以使用 **CDbFixtureManager::basePath** 属性改变这个位置。让我们先看一个为 **tbl_project** 表创建一个夹具的例子。创建一个新文件 **protected/tests/fixtures/tbl_project.php**, 并保存如下内容:

PHP 代码:

```
<?php

return array(

    'project1'=>array(

        'name' => 'Test Project 1',

        'description' => 'This is test project 1',

        'create_time' => '',

        'create_user_id' => '',

        'update_time' => '',

        'update_user_id' => '',

    ),

    'project2'=>array(

        'name' => 'Test Project 2',

        'description' => 'This is test project 2',

        'create_time' => '',
```



```

        'create_user_id' => '',

        'update_time' => '',

        'update_user_id' => '',

    ),

    'project3'=>array(

        'name' => 'Test Project 3',

        'description' => 'This is test project 3',

        'create_time' => '',

        'create_user_id' => '',

        'update_time' => '',    'update_user_id' => '',

    ),

);

```

我们可以看到，我们的夹具数组的键象征我们的数据表。这些键的值是一个 **key=>value** 对的数组代表数据表中的列。我们添加了三行，但如果你喜欢可以添加更多行。为了简单起见，我们仅仅为值不能是 **NULL** 的字段预设了值，这是 **name** 和 **description** 字段。这些数据将足够夹具使用了。

你可能还注意到夹具数据没有指定 **id** 列。因为这列被定义为自增长类型，当我们插入新行时这列的值将会由数据库本身处理。

配置使用此夹具

我们仍然需要告诉我们的单元测试实际使用这个刚刚创建的夹具.我们把它放在单元测试文件中.在这种情况下,我们需要在测试文件 **protected/tests/unit/ProjectTest.php** 的顶部添加我们的夹具声明:

PHP 代码:

```

<?php

class ProjectTest extends CDbTestCase

{

    public $fixtures=array

    (

```

```
'projects' => 'Project',  
  
);  
  
}
```

因此,我们所做的是指定成员变量`$fixtures`,它的值是一个数组,指定用来测试使用的夹具。该数据表示一个夹具的名称与将要用于测试的模型类名称或夹具所指的名称的关系映射(例如,夹具的名称 `projects` 对应模型类 `Project`)。当在使用模型类名称的情况下,则模型类的基础表将被视为夹具对应的表。正如我们前面所述,它是夹具管理器在每次执行测试方法时用来管理这个基本的表并重置数据。

如果你使用的夹具是针对一个表而不是一个 `AR` 类,你需要在表名前加上前缀冒号(例如, `:tbl_project`)来与模型类区分。

夹具的名字允许我们方便的在测试方法里访问夹具数据。例如,我们已经在 `ProjectTest` 类定义了,我们可以通过几下的方式访问我们的夹具数据:

PHP 代码:

```
// return all rows in the 'Project' fixture table  
  
$projects = $this->projects;  
  
// return the row whose alias is 'project1' in the 'Project' fixture table  
  
$projectOne = $this->projects['project1'];  
  
// If our fixture is associated with an active record, return the AR instance representing  
  
// the 'project1' fixture data row  
  
$project = $this->projects('project1');
```

我们将提供更具体的例子,当我们改变我们这些用于实际测试的夹具数据,首先,我们需要改变我们的测试环境。

指定一个测试数据库

正如我们之前提到的,我们需要区分开发数据库与测试数据库,使我们的测试不干涉开发。

Yii 已经为我们指定了一个用于测试的配置文件。我们需要创建另一个数据库,它的名字叫 `trackstar_test`。我们还需要复制当前的 `trackstar_dev` 数据库的结构。这很容易,我们目前只有一个 `tbl_project` 表。请根据之前介绍创建 `tbl_project` 表。一旦创建后,我们可以在指定的配置文件 `protected/config/test.php` 中添加数据库连接信息作为应用程序的组件。你可以从 `main.php` 配置文件中复制 `db` 组件到 `test.php` 文件中。对于 `MySQL` 的用户保持一致即可。我们将要添加以下的高亮显示的代码到测试配置文件中:

PHP 代码:

```
return CMap::mergeArray(

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

    array(

        'components' => array(

            'fixture' => array(

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

            ),

            'db' => array(

                'connectionString' => 'mysql:host=localhost;dbname=trackstar_test',

                'emulatePrepare' => true,

                'username' => '[your db username]',

                'password' => '[your db password]',

                'charset' => 'utf8',

            ),

        ),

    );
```

当我们运行测试会自动加载测试配置文件，而不是主配置文件。实际上它会合并主配置文件到测试配置文件中。如果定义了相同的组件或配置的值，则会优先考虑在测试配置文件定义的。现在，当我们运行单元测试时，我们将操作测试数据库而不是开发数据库，运行我们的测试套件不会对开发进度有负面影响。

使用夹具

现在，我们已经调整了我们的测试环境使用单独的数据库。我们应该采用夹具的优势。当我们最初在单元测试中为 **Project AR** 类编写 **CRUD** 操作时，我们把所有的创建，读取，更新，删除测试都放到了一个测

试方法 `testCRUD()` 中。所有这些离散的测试放到一个大的测试中存在很多的缺点。如果首先创建失败，则整个测试方法将停止执行，读取，更新和删除的测试永远不会运行。理想的情况下，我们应该分离这些测试使彼此不产生依赖。这样做的最主要的原因是我们可以避免需要顺序运行测试方法。如果我们分开了创建测试，和读取测试，有一个读取方法在创建方法之前执行，这将导致测试失败，因为没有读取到创建返回的这行数据。但是，如果我们使用夹具数据，我们就能够避免这个问题。

现在，我们新的测试环境已配置了专用数据库并定义了夹具数据，我们能分离 **CRUD** 单元测试。通过这一些具体的例子来了解如何使用夹具数据。

让我们开始读取测试。打开 **ProjectTest.php** 单元测试文件并添加如下测试方法：

PHP 代码：

```
public function testRead()

{

    $retrievedProject = $this->projects('project1');

    $this->assertTrue($retrievedProject instanceof Project);

    $this->assertEquals('Test Project 1', $retrievedProject->name);

}
```

我们知道，在此测试运行之前，夹具管理器将使用已经定义的夹具数据重置 **trackstar_test** 数据库中的 **tbl_project** 表。在这里，我们只是简单的读取第一行数据，引用该行的别名 **project1**，它将返回一个 **Project AR** 实例基于定义在 **protected/tests/fixtures/tbl_project.php** 中的夹具数据的第一行数据。然后，我们测试刚刚返回的变量是否为一个 **Project** 类的实例，并测试它的名称(**name**)是否与夹具数据中的一致。

同样，我们可以添加单独的 **testCreate()**，**testUpdate()** 和 **testDelete()** 方法。整个测试文件修改后如下所示：

PHP 代码：

```
<?php

class ProjectTest extends CDbTestCase

{

    public $fixtures=array(

        'projects' => 'Project' ,

    );

}
```

```

public function testCreate()

{
    //CREATE a new Project

    $newProject=new Project;

    $newProjectName = 'Test Project Creation';

    $newProject->setAttributes(array(

        'name' => $newProjectName,

        'description' => 'This is a test for new project creation',

        'createTime' => '2009-09-09 00:00:00',

        'createUser' => '1',

        'updateTime' => '2009-09-09 00:00:00',

        'updateUser' => '1',

    ));

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

    //READ back the newly created Project to ensure the creation worked

    $retrievedProject=Project::model()->findByPrimaryKey($newProject->id);

    $this->assertTrue($retrievedProject instanceof Project);

    $this->assertEquals($newProjectName, $retrievedProject->name);

}

public function testRead()

{

```

```

    $retrievedProject = $this->projects('project1');

    $this->assertTrue($retrievedProject instanceof Project);

    $this->assertEquals('Test Project 1', $retrievedProject->name);
}

public function testUpdate()
{
    $project = $this->projects('project2');

    $updatedProjectName = 'Updated Test Project 2';

    $project->name = $updatedProjectName;

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

    //read back the record again to ensure the update worked

    $updatedProject=Project::model()->findByPk($project->id);

    $this->assertTrue($updatedProject instanceof Project);

    $this->assertEquals($updatedProjectName, $updatedProject->name);
}

public function testDelete()
{
    $project = $this->projects('project2');

    $savedProjectId = $project->id;

    $this->assertTrue($project->delete());

    $deletedProject=Project::model()->findByPk($savedProjectId);

    $this->assertEquals(NULL, $deletedProject);
}

```

}

现在，如果其中任何一个测试运行失败，其余的不同操作仍然可以提供更细的反馈。

小结

虽然我们没有在这章进行实际编码，但我们也完成了很多事情。我们创建一个新的数据库表，并看到了 **Yii** 中 **AR** 的操作。我们使用 **Gii** 代码生成器首先创建了一个封装了 **tbl_project** 表的 **AR** 类。然后我们编写了测试，去尝试了解和使用 **AR** 类。

然后我们演示了怎么使用 **Gii** 代码生成工具生成了 **Web** 应用程序的 **CRUD** 功能。有了这个非常棒的工具，我们实现了本次迭代计划的提供的大部分功能。我们修改了提交表单中项目的名称(**name**)和描述(**description**)的验证功能。

最后，我们介绍了 **Yii** 中的测试夹具，并利用夹具的优势对测试环境作出了一些调整。

在接下来的迭代中，我们将通过我们在这章中学习到的知识，继续建立相关的数据模型。

第六章：迭代 3：添加任务（Task）

在上一个迭代中，我们交付了与项目（**Project**）实体相关的基本功能。项目（**Project**）是 **TrackStar** 应用的基础。但是，项目（**Project**）本身用处并不是很大。我们希望使用这个应用来管理问题（**Issue**），而项目（**Project**）只是问题（**Issue**）的基本容器。由于管理项目（**Project**）中的问题（**Issue**）是开发这款应用的主要目的，所以我们将这个迭代主要用于添加一些基本的问题（**Issue**）管理功能。

迭代计划

我们已经有了创建和列出项目（**Project**）的功能，但是这些项目（**Project**）还不能包含任何东西。在这个迭代结束后，我们希望应用程序能提供所有关于问题（**Issue**）或者说任务（**Task**）的 **CRUD**（增删改查）操作（我们会交替使用术语问题（**Issue**）和任务，但是在我们的数据模型中，一个任务实际上只是问题（**Issue**）的一种）。我们还需要将所有对问题（**Issue**）进行的 **CRUD** 操作，限制在一个特定的项目（**Project**）上下文中进行。也就是说，问题（**Issue**）是从属于项目（**Project**）的。在进行任何对问题（**Issue**）的 **CRUD** 操作之前，用户必须首先选定一个已经存在的项目（**Project**）并在该项目（**Project**）之下进行工作。

为了实现上述目标，我们需要识别出所有要在此迭代中完成的细化工作。下面的列表是对这些工作的概括：

- § 设计数据库结构，创建对象来支持问题（**Issue**）
- § 创建 **Yii** 模型类，使应用能够简单地与我们创建的数据库表交互
- § 创建控制器类，来容纳我们需要的功能，包括：
 - 创建新问题（**Issue**）

- 从数据库中取回项目（**Project**）中问题（**Issue**）的列表
 - 更新、编辑问题（**Issue**）
 - 删除问题（**Issue**）
- 创建试图来为这些动作（上述的）生成用户界面

这些信息已经足够我们开始干活了。运行完测试后，我们开始做一些必要的数据库修改。

运行测试套件

在深入到开发工作中之前，运行已经存在的测试套件，总是一个非常好的主意。我们的测试套件在上一个迭代的时候增长了一点。现在我们已经有了对项目（**Project**）进行 **CRUD** 操作和数据库链接的测试。把它们一起再运行一次。打开测试文件夹，`/protected/tests/unit`，然后运行所有单元测试：

SHELL 代码或屏幕回显：

```
%phpunit unit/  
  
PHPUnit 3.3.17 by Sebastian Bergmann.  
  
.....  
  
Time: 0 seconds OK (5 tests, 11 assertions)
```

都通过了。让我们开始进行修改。

设计数据库结构

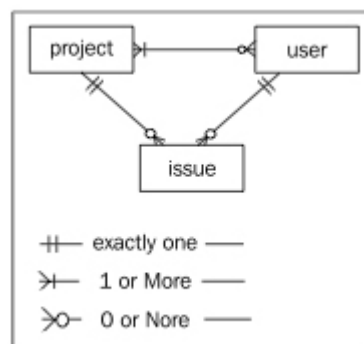
回到第 3 章 **TrackStar** 应用，我们提出了一些关于问题（**Issue**）的初始想法。我们假设它有类型、所有人、请求人（**requester**）、状态和描述等自然属性。还提到了当我们创建了表 **tbl_project** 的时候，会给每个表添加基本的审计历史信息，用于跟踪更新了表的用户、日期和时间。在这个过程中，需求并没有发生改变，所以我们可以继续进行初始的计划。但是，类型、所有人、请求人和状态这些属性本身就是它们自己的实体。为了保持模型的弹性和扩展性，我们会对这些属性分别建模。所有人和请求人都是系统的用户，将会指向 **tbl_user** 表中的一行。在表 **tbl_project** 中，我们已经引入了用户的概念，我们添加了 **create_user_id** 和 **update_user_id** 两列，来跟踪创建了这个项目（**Project**）和对这个项目（**Project**）细节最后一次更新负责的用户标识符。尽管还没有正式引入用户表，这些域已经被设定成数据库中另一个存储用户的表的外键。在表 **tbl_issue** 中，**owner_id** 和 **requestor_id** 也是指向表 **tbl_user** 的外键。

我们可以用同样的方式来为类型和状态属性建模。不过，在需求要求模型具备这些额外的复杂度之前，我们让事情简单一些。表 **tbl_issue** 中的 **type** 和 **status** 列仍将是整数值类型，映射到命名的类型和状态。我们把这些属性建模成问题（**Issue**）实体的 **AR** 模型类的常量值，而不是使用分开的表来完善我们的模型。如果所有的这一切看起来有一点迷惑，请不要担心，在接下来的章节中，这些都会变清楚的。

定义一些关系

因为要引入表 **tbl_user**, 我们回过头来定义一下用户和项目 (**Project**) 之间的关系。第三章引入 **TrackStar** 应用的时候, 我们设定用户 (项目成员) 可以和一个或者多个项目关联。还提到项目可以有許多 (0 个或多个) 用户。因为项目可以有多个用户, 而且用户可以被关联到多个项目, 我们称之为项目和用户之间的多对多关系。在关系型数据库中, 对一个多对多关系建模的最简单方法是, 使用关联或者赋值表 (**assignment table**)。所以, 我们也需要将这个表添加到我们的模型中。

下图勾勒出一个基本的实体关系, 这个关系就是我们需要在用户, 项目 (**Project**), 问题 (**Issue**) 之间建立的模型关系。项目可以有 0 个或者多个用户参与, 一个用户至少要被关联到一个项目, 但是也可以被关联到多个项目。问题 (**Issue**) 属于且仅属于一个项目, 同时项目可以包含 0 个或者多个问题 (**Issue**)。最后, 一个问题 (**Issue**) 只能被分配给一个用户 (或者由一个用户提出)。



建立数据库和关系

那么, 我们需要创建三张新表: **tbl_issue**, **tbl_user** 和我们的关联表 **tbl_project_user_assignment**。为了让您方便一点, 我们提供了基本的数据定义语言 (**DDL**) 语句用于创建表和它们之间的关系。由于基本的用户管理不是这个迭代的一部分, 我们还提供了一点种子数据填充到用户表中, 以使我们可以立即使用它们。请像前几个迭代你做的那样, 创建下列的表和关系。下列的语法假设你使用的是 **MySQL** 数据库:

SQL 代码:

```
DROP TABLE IF EXISTS `tbl_project`;

CREATE TABLE `tbl_project`
(
  `id` INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,
  `name` VARCHAR(128),
  `description` TEXT,
  `create_time` DATETIME,
  `create_user_id` INTEGER,
```

```
    `update_time` DATETIME,  
    `update_user_id` INTEGER  
)  
ENGINE = InnoDB;
```

```
CREATE TABLE IF NOT EXISTS `tbl_issue`  
(  
    `id` INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,  
    `name` VARCHAR(256) NOT NULL,  
    `description` VARCHAR(2000),  
    `project_id` INTEGER,  
    `type_id` INTEGER,  
    `status_id` INTEGER,  
    `owner_id` INTEGER,  
    `requester_id` INTEGER,  
    `create_time` DATETIME,  
    `create_user_id` INTEGER,  
    `update_time` DATETIME,  
    `update_user_id` INTEGER,  
    INDEX (`project_id`)  
)  
ENGINE = InnoDB;
```

```
CREATE TABLE IF NOT EXISTS `tbl_user`  
(  

```

```

`id` INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,

`email` VARCHAR(256) NOT NULL,

`username` VARCHAR(256),

`password` VARCHAR(256),

`last_login_time` DATETIME,

`create_time` DATETIME,

`create_user_id` INTEGER,

`update_time` DATETIME,

`update_user_id` INTEGER

) ENGINE = InnoDB;

```

```

CREATE TABLE IF NOT EXISTS `tbl_project_user_assignment`

(

    `project_id` INT(11) NOT NULL,

    `user_id` INT(11) NOT NULL,

    `create_time` DATETIME,

    `create_user_id` INTEGER,

    `update_time` DATETIME,

    `update_user_id` INTEGER,

    PRIMARY KEY (`project_id`, `user_id`)

) ENGINE = InnoDB;

```

-- The Relationships

```
ALTER TABLE `tbl_issue` ADD CONSTRAINT `FK_issue_project`  
FOREIGN KEY (`project_id`) REFERENCES `tbl_project` (`id`)  
ON DELETE CASCADE ON UPDATE RESTRICT;
```

```
ALTER TABLE `tbl_issue` ADD CONSTRAINT `FK_issue_owner`  
FOREIGN KEY (`owner_id`) REFERENCES `tbl_user` (`id`)  
ON DELETE CASCADE ON UPDATE RESTRICT;
```

```
ALTER TABLE `tbl_issue` ADD CONSTRAINT `FK_issue_requester`  
FOREIGN KEY (`requester_id`) REFERENCES `tbl_user` (`id`)  
ON DELETE CASCADE ON UPDATE RESTRICT;
```

```
ALTER TABLE `tbl_project_user_assignment`  
ADD CONSTRAINT `FK_project_user` FOREIGN KEY (`project_id`)  
REFERENCES `tbl_project` (`id`)  
ON DELETE CASCADE ON UPDATE RESTRICT;
```

```
ALTER TABLE `tbl_project_user_assignment`  
ADD CONSTRAINT `FK_user_project` FOREIGN KEY (`user_id`)  
REFERENCES `tbl_user` (`id`)  
ON DELETE CASCADE ON UPDATE RESTRICT;
```

```
-- Insert some seed data so we can just begin using the database
```

```
INSERT INTO `tbl_user`  
  
(`email`, `username`, `password`)  
  
VALUES  
  
('test1@notanaddress.com', 'Test_User_One', MD5('test1')),  
  
('test2@notanaddress.com', 'Test_User_Two', MD5('test2'));
```

创建 Active Record 模型类

建立好了这些表后，为了能与它们简单地交互，我们需要创建 Yii 的 **AR** 模型类。在第 5 章 迭代 2：项目 **CRUD** 中，使用 **Gii** 代码生成工具创建过 **Project.php** 模型类时，我们就做过这件事。在这里，我们会把步骤再提醒你一次，但是省略了截屏。用 **Gii** 工具的更多细节请参考第 5 章。

创建问题（Issue）模型类

访问 <http://localhost/trackstar/index.php?r=gii> 打开 **Gii** 工具页面，选择模型生成器链接。表前缀保留 **tbl_**。表名（Table Name）填写 **tbl_issue**，模型类（Model Class）格会自动填上 **Issue**。

填写后，点击预览按钮，会得到一个链接，那个链接会打开一个弹出框，里面将展示所有将要生成的代码。然后点击生成按钮来真正创建 **Issue.php** 模型类文件到 **/protected/model** 文件夹。生成代码的完整列表如下：

PHP 代码：

```
/**  
  
 * This is the model class for table "tbl_issue".  
  
 *  
 * The followings are the available columns in table 'tbl_issue':  
  
 * @property integer $id  
  
 * @property string $name
```

```

* @property string $description

* @property integer $project_id

* @property integer $type_id

* @property integer $status_id

* @property integer $owner_id

* @property integer $requester_id

* @property string $create_time

* @property integer $create_user_id

* @property string $update_time

* @property integer $update_user_id

*

* The followings are the available model relations:

* @property User $requester

* @property User $owner

* @property Project $project

*/

class Issue extends CActiveRecord {

    /**
     * Returns the static model of the specified AR class.
     *
     * @return Issue the static model class
     */

    public static function model($className=__CLASS__) {

        return parent::model($className);

    }

```

```
/**
```

```
 * @return string the associated database table name
```

```
 */
```

```
public function tableName() {
```

```
    return 'tbl_issue';
```

```
}
```

```
/**
```

```
 * @return array validation rules for model attributes.
```

```
 */
```

```
public function rules() {
```

```
    // NOTE: you should only define rules for those attributes that
```

```
    // will receive user inputs.
```

```
    return array(
```

```
        array('name', 'required'),
```

```
        array('project_id, type_id, status_id, owner_id, requester_id, create_user_id,  
            update_user_id', 'numerical', 'integerOnly' => true),
```

```
        array('name', 'length', 'max' => 256),
```

```
        array('description', 'length', 'max' => 2000),
```

```
        array('create_time, update_time', 'safe'),
```

```
        // The following rule is used by search().
```

```
        // Please remove those attributes that should not be searched.
```

```
        array('id, name, description, project_id, type_id, status_id, owner_id,
```

```
requester_id, create_time, create_user_id, update_time,  
update_user_id',
```

```
'safe', 'on' => 'search'),
```

```
);
```

```
}
```

```
/**
```

```
 * @return array relational rules.
```

```
 */
```

```
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(  
        'requester' => array(self::BELONGS_TO, 'User', 'requester_id'),  
        'owner' => array(self::BELONGS_TO, 'User', 'owner_id'),  
        'project' => array(self::BELONGS_TO, 'Project', 'project_id'),  
    );  
}
```

```
/**
```

```
 * @return array customized attribute labels (name=>label)
```

```
 */
```

```
public function attributeLabels() {
```

```
    return array(  
        'id' => 'ID',
```



```
'name' => 'Name',  
  
'description' => 'Description',
```

```
'project_id' => 'Project',
```

```
'type_id' => 'Type',  
  
'status_id' => 'Status',  
  
'owner_id' => 'Owner',  
  
'requester_id' => 'Requester',  
  
'create_time' => 'Create Time',  
  
'create_user_id' => 'Create User',  
  
'update_time' => 'Update Time',  
  
'update_user_id' => 'Update User',
```

```
);
```

```
}
```

```
/**
```

```
 * Retrieves a list of models based on the current search/filter conditions.
```

```
 * @return CActiveDataProvider the data provider that can return the models based on  
         the search/filter conditions.
```

```
 */
```

```
public function search() {
```

```
    // Warning: Please modify the following code to remove attributes that
```

```
    // should not be searched.
```

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

```

$criteria->compare('id', $this->id);

$criteria->compare('name', $this->name, true);

$criteria->compare('description', $this->description, true);

$criteria->compare('project_id', $this->project_id);

$criteria->compare('type_id', $this->type_id);

$criteria->compare('status_id', $this->status_id);

$criteria->compare('owner_id', $this->owner_id);

$criteria->compare('requester_id', $this->requester_id);

$criteria->compare('create_time', $this->create_time, true);

$criteria->compare('create_user_id', $this->create_user_id);

$criteria->compare('update_time', $this->update_time, true);

$criteria->compare('update_user_id', $this->update_user_id);

return new CActiveDataProvider(get_class($this), array(

    'criteria' => $criteria,

));

}

}

```

创建用户模型类

这对您来说可能已经像是老生常谈了，所以我们把创建用户 **AR** 模型类的工作留作练习。下一章中，讨论用户权限和认证的时候，这个类会变得非常重要。

表 `tbl_project_user_assignment` 对应的 **AR** 类什么时候创建呢？

虽然可以为这个表创建一个 **AR** 类，但是这并没有必要。**AR** 模型为我们的应用提供了一个对象关系映射 (**ORM**) 层，来帮助我们方便地操作我们的领域对象。然而，**ProjectUserAssignment** 不是一个应用中的领域对象。它只是在关系型数据库中，构建出来，帮助我们建模和管理项目 (**Project**) 和用户之间的多对

多关系。维护一个专门的 **AR** 类来管理这个表，增加了额外的复杂性，我们可以避免在这时候做这样的事。我们可以通过使用 **Yii** 的 **DAO** 来直接管理这个表的插入、更新和删除来避免额外的维护工作和性能负载。

创建问题（Issue）CRUD 操作

现在，**AR** 类已经准备就绪了，我们可以开始创建管理我们项目问题（**Issue**）必须的功能了。由于项目问题（**Issue**）的 **CRUD** 操作是我们这个迭代要实现的主要目标，我们还是要依赖 **Gii** 代码生成工具来帮助我们这些基本的功能。这方面的细节，我们在第 5 章创建项目的时候已经做过了。在这里，为问题（**Issue**）创建相关代码的时候，我们会再提醒你一些基本的步骤。

在 <http://localhost/trackstar/index.php?r=gii> 打开 **Gii** 代码生成器的菜单，选择 **Crud Generator** 链接。在 **Model Class** 表单域中填写 **Issue**。**Controller ID** 中会自动填上 **Issue**。**Base Controller Class** 和 **Code Template** 域可以保留他们的默认值。点击 **Preview** 按钮，可以得到一个 **Gii** 将要创建的代码的列表。下面的截屏显示了这个文件的列表：



Code File	Generate <input type="checkbox"/>
controllers/IssueController.php	new <input checked="" type="checkbox"/>
views/issue/_form.php	new <input checked="" type="checkbox"/>
views/issue/_search.php	new <input checked="" type="checkbox"/>
views/issue/_view.php	new <input checked="" type="checkbox"/>
views/issue/admin.php	new <input checked="" type="checkbox"/>
views/issue/create.php	new <input checked="" type="checkbox"/>
views/issue/index.php	new <input checked="" type="checkbox"/>
views/issue/update.php	new <input checked="" type="checkbox"/>
views/issue/view.php	new <input checked="" type="checkbox"/>

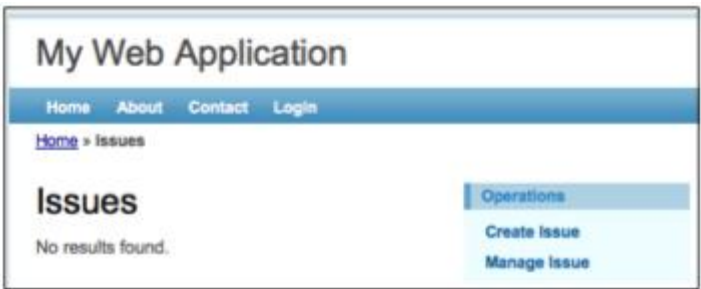
您可以点击每个单独的链接来预览将要生成的代码。如果满意了，就点击 **Generator** 按钮，创建所有这些文件。您将看到下面的成功消息：



The controller has been generated successfully. You may [try it now](#).

使用问题（Issue）CRUD 操作

让我们来试一下吧，点击上个截屏中显示的 **try it now** 链接，或者直接访问 <http://localhost/trackstar/index.php?r=issue>，您将看到跟下面的截图相似的一个画面：



创建一个新的问题（Issue）

由于我们还没有创建过任何一个新的问题（**Issue**），所以列表是空的。那么，让我们来创建一个吧。点击 **Create Issue** 链接（如果您被重定向到登录画面，那么使用 **demo/demo** 或者 **admin/admin** 登录），您将会看到与下面截屏相似的新问题（**Issue**）输入表单：

Create Issue

*Fields with * are required.*

Name *

Description

Project

Type

Status

Owner

Requester

Create Time

Create User

Update Time

Update User

Create

Operations

List Issue

Manage Issue

看这张输入表单，可以注意到，数据库表中的每一列，都有一个对应的输入域，跟数据库表中定义的一样。然而，定义和创建数据库表时，我们就知道，有些域不是直接输入数据的域，而是表达与其他实体之间的关系的。例如，我们应该使用一个下拉列表，里面填好可能的问题（**Issue**）类型，让用户选择，而不是放一个可以随便填写的 **Type** 输入框。对于 **Status** 域来说，也是一样。**Owner** 和 **Requester** 域也应该是下拉列表，里面填充的选择项是那些已经被分派了在此问题（**Issue**）所属项目下工作的用户。此外，所有的问题（**Issue**）管理，都应该发生在某个特定的项目的上下文环境中。因此，**Project** 域甚至根本不应该出现在表单中。最后，**Create Time**，**Create User**，**Update Time** 和 **Update User** 都是应该在表单提交时候计算出来的，而不应该由用户直接来填写。

好，现在我们已经识别出了一些这张初始的表单上要更正的内容。第 5 章中已经提到，由 **Gii** 工具自动生成的 **CRUD** 脚手架代码，只是我们开始的地方。它不可能满足一个应用中所有功能需求。我们肯定会经识别出许多问题（**Issue**）创建过程中需要更正的部分。我们会逐一来修改。

添加类型下拉菜单

我们从添加一个问题（**Issue**）类型下拉菜单开始。

问题（Issue）只有下面三种类型：

§ 错误（Bugs）

§ 功能（Features）

§ 任务（Tasks）

当我们创建一个问题（Issue）时，我们希望看到的是，一个只包含这三个选项的下拉列表域供用户选择。我们将通过让问题（Issue）模型类自己提供一个可选类型的列表，来实现这一点。你可能已经猜到了，在给问题（Issue）模型 AR 类添加这个新功能之前，我们首先要写一个测试。

你应该记得，在第 5 章中，我们添加了一个新的数据库来专门运行我们的测试，叫 **trackstar_test**。我们这么做，是为了确保测试环境不会对开发环境造成不利的影响。所以请先确认你已经用我们先前创建的新表 **tbl_issue** 和 **tbl_user** 更新了测试数据库。

让测试进入“红色区”

我们已经知道，TDD 过程的第一步，是快速地写一个产生失败结果的测试。创建一个新的单元测试文件 **protected/tests/unit/IssueTest.php** 并且添加下面这些代码：

PHP 代码：

```
class IssueTest extends CDbTestCase {

    public function testGetTypes() {

        $options = Issue::model()->typeOptions;

        $this->assertTrue(is_array($options));

    }

}
```

现在打开命令行，运行如下命令执行测试：

SHELL 代码或屏幕回显：

```
phpunit unit/IssueTest.php

PHPUnit 3.3.17 by Sebastian Bergmann.

.E
```

```
Time: 0 seconds
```

```
There was 1 error:
```

```
testGetTypes(IssueTest)
```

```
CException: Property "Issue.typeOptions" is not defined.
```

```
/YiiRoot/framework/base/CComponent.php: 131
```

```
/YiiRoot/yii-read-only/framework/db/ar/CActiveRecord.php: 107
```

```
/Webroot/tasctrak/protected/tests/unit/IssueTest.php: 6
```

```
FAILURES!
```

```
Tests: 1, Assertions: 0, Errors: 1.
```

OK，现在我们已经完成了 TDD 中的第一步。（那就是快速写一个失败的测试）。测试会失败的原因是非常明显的。在模型类中，并不包含方法 `Issue::typeOptions()`。我们需要添加一个。

从“红色区”到“绿色区”

现在打开 AR 模型类，在 `protected/models/Issue.php` 文件夹里，然后给类添加下面的方法：

PHP 代码：

```
/**
 * @return array issue type names indexed by type IDs
 */
public function getTypeOptions() {
    return array();
}
```

我们已经添加了一个简单的方法，取了一个合适的名字，它返回一个数组类型（尽管现在仍旧是空的）。

现在我们再运行一次测试：

SHELL 代码或屏幕回显：

```
phpunit unit/IssueTest.php
```

```
PHPUnit 3.3.17 by Sebastian Bergmann.
```

```
..
```

```
Time: 0 seconds
```

```
OK (1 tests, 1 assertion)
```

Yii 框架的基类使用 PHP 的 `__get` 神奇函数。这允许我们在子类中编写方法，如 `getTypeOptions()`，但是，却可以像使用类属性那样的语法如 `>typeOptions` 去访问这个方法。

所以，现在我们的测试会通过的。我们已经进入“绿色区”。这非常棒！但是我们现在实际上并没有返回任何数值。我们当然不能够基于这个空的数组来添加我们的下拉菜单。因为我们有三种基础的问题（**Issue**）类型，我们使用类常量来将它们映射为整数值，然后我们使用我们的 `getTypeOptions()` 方法来返回用户友好的描述词应用到下来菜单中。

再次回到“红色区”

在将这个添加到 **Issue** 类之前，让我们的测试再次失败。让我们再添加一个断言来质询返回的数组，并且验证它的内容是我们想要的。我们的测试将保证返回数组有三个元素，并且这些值对应着我们的问题（**Issue**）类型：错误、功能和任务。将测试修改为：

PHP 代码：

```
public function testGetTypes() {  
  
    $options = Issue::model()->typeOptions;  
  
    $this->assertTrue(is_array($options));  
  
    $this->assertTrue(3 == count($options));  
  
    $this->assertTrue(in_array('Bug', $options));  
  
    $this->assertTrue(in_array('Feature', $options));  
  
    $this->assertTrue(in_array('Task', $options));  
  
}
```

由于 `getTypeOptions()` 方法，仍然返回一个空数组，我们的断言肯定会失败。所以，我们又回到了红色区。让我们往 `Issue.php` 中添加代码来使这些断言通过吧。

再次回到“绿色区”

在 **Issue** 类的顶部，添加下面三个常量定义：

PHP 代码：

```
const TYPE_BUG=0;

const TYPE_FEATURE=1;

const TYPE_TASK=2;
```

然后，修改方法 **Issue::getTypeOptions()** 返回一个基于这些常量定义的数组。

PHP 代码：

```
public function getTypeOptions()

{

    return array(

        self::TYPE_BUG=>'Bug',

        self::TYPE_FEATURE=>'Feature',

        self::TYPE_TASK=>'Task',

    );

}
```

现在如果我们再运行一次测试，我们所有 5 个断言都回通过，然后我们又回到了绿色区。

SHELL 代码或屏幕回显：

```
phpunit unit/IssueTest.php

PHPUnit 3.3.17 by Sebastian Bergmann.

..

Time: 0 seconds

OK (1 tests, 5 assertions)
```


我们现在让我们的模型类返回了需要的问题（**Issue**）类型，但是我们的表单中还没有一个下拉列表域来使用这些值。现在让我们来添加。

添加问题（Issue）类型下拉选择框

打开新建问题（**Issue**）表单视图，`protected/views/issue/_form.php`，找到与类型对应的表单域：

PHP 代码：

```
<div class="row">

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

    <?php echo $form->textField($model, 'type_id'); ?>

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

</div>
```

这几行需要明确含义。为了理解这个，我们需要参考在 `_form.php` 的顶部的一些代码：

PHP 代码：

```
<?php $form=$this->beginWidget('CActiveForm', array(

    'id' => 'issue-form',

    'enableAjaxValidation' => false,

)); ?>
```

这段代码定义了一个变量 `$form`，它是一个 **Yii** 的 **CActiveForm** 挂件（**Widget**）。第 9 章会介绍挂件（**Widget**）的更多细节。现在，我们只要更好地理解 **CActiveForm**，就能理解这些代码。它可以被认为是一个助手类，提供了一组方法帮助我们创建一个与数据模型类关联的数据输入表单。在这，模型类指的就是问题（**Issue**）模型类。

为了完全理解视图文件中的变量，让我们回顾一下产生了视图文件的控制器代码。你应该记得，将数据从控制器传递到视图的一个方法是明确声明一个数组，数组的键，是在视图文件中可用的变量的名称。由于这是一个新问题（**Issue**）的创建动作，生成表单的控制器方法是 `IssueController::actionCreate()`。这个方法在下面列出来了：

PHP 代码：

```
public function actionCreate() {

    $model=new Issue;
```

```
// Uncomment the following line if AJAX validation is needed
```

```
// $this->performAjaxValidation($model);
```

```
if(isset($_POST['Issue'])) {
```

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

```
    if($model->save())
```

```
    {
        $this->redirect(array('view','id'=>$model->id));
    }
```

```
    $this->render('create',array(
```

```
        'model'=>$model,
```

```
    ));
```

```
}
```

这里我们看到，生成视图时，问题（Issue）模型类的一个实例被传送了，可以通过一个叫`$model`的变量访问到。

好的，现在让我们回到创建新问题（Issue）项表单中负责生成 **Type** 域的代码。第一行是：

PHP 代码：

```
$form->labelEx($model,'type_id');
```

这一行使用 `CActiveForm::labelEx()` 为问题（Issue）模型的一个属性 `type_id` 生成一个 HTML 标签。它接受一个模型类的实例和对应的模型属性名来生成我们要的标签。模型类 `Issue::attributeLabels()` 方法被用来决定标签。找到这个方法，我们看到属性 `type_id` 映射到一个标签 **Type**，正是我们看到的这个表单域的标签。

PHP 代码：

```
public function attributeLabels() {
```

```
    return array(
```

```
        'id' => 'ID',
```

```
        'name' => 'Name',
```

```
        'description' => 'Description',
```

```
'project_id' => 'Project',
```

```
'type_id' => 'Type',
```

```
'status_id' => 'Status',
```

```
'owner_id' => 'Owner',
```

```
'requester_id' => 'Requester',
```

```
'create_time' => 'Create Time',
```

```
'create_user_id' => 'Create User',
```

```
'update_time' => 'Update Time',
```

```
'update_user_id' => 'Update User',
```

```
);
```

```
}
```

下一行代码是：

PHP 代码：

```
<?php echo $form->textField($model, 'type_id'); ?>
```

它使用了 **CActiveForm::textField()** 方法，来给我们的问题（**Issue**）模型属性生成一个文本输入域。所有的为 **type_id** 定义的验证都在类方法 **Issue::rules()** 中，他们会作为表单验证规则应用到输入表单上。

最后一行代码如下：

PHP 代码：

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

它使用了 **CActiveForm::error()** 方法来生成提交时，与特定 **type_id** 相关的属性的验证错误。使用这种方式，错误消息会直接显示在域的下方。

你可以使用 **Type** 域来尝试一下这个验证。因为 **type_id** 列在我们的 **MySQL** 数据库中被定义成了一个整数，**Gii** 生成的问题（**Issue**）模型类在 **Issue::rules()** 方法中，有一个验证规则，来保证这个约束：

PHP 代码：

```
public function rules() {
```

```

// NOTE: you should only define rules for those attributes that
// will receive user inputs.

return array(

    array('name', 'required'),

    array('project_id, type_id, status_id, owner_id, requester_id, create_user_id,
          update_user_id', 'numerical', 'integerOnly' => true),

    array('name', 'length', 'max' => 256),

    array('description', 'length', 'max' => 2000),

    array('create_time, update_time', 'safe'),

    // The following rule is used by search().

    // Please remove those attributes that should not be searched.

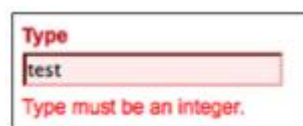
    array('id, name, description, project_id, type_id, status_id, owner_id, requester_id,
          create_time, create_user_id, update_time, update_user_id',

          'safe', 'on' => 'search'),

);
}

```

所以，如果我们试图提交一个字符串值到 **Type** 表单域，我们会看到一个行内错误，在域的右下方，像下面的截图显示的一样：

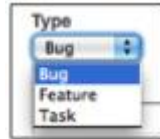


现在我们已经明白了我们有什么，那我们就有了一个更好的基础来改变它。我们需要做的事情是，我们要把这个表单域从一个允许自由输入的文本域变成一个下拉选项类型。可能让你有点惊讶，**CActiveForm** 类有一个 **dropDownList()** 方法，可以根据模型属性生成一个下来列表。所以，让我们使用下面的代码替换 **\$text->textField** 的那一行：

PHP 代码：

```
<?php echo $form->dropDownList($model,'type_id',$model->getTypeOptions()); ?>
```

这个方法将采用相同的模型作为第一个参数，模型属性作为第二个参数。第三个参数确定了下拉的选项。这应该是一个数组，由 **value=>display** 对组成。在 **Issue** 模型中，我们已经创建我们的 **getTypeOptions()** 方法，来返回这个格式的数组，所以我们现在可以直接使用。保存你的工作，然后我们再看一次我们的问题（**Issue**）输入表单。你可以看到一个很好的下拉列表，里面有类型的选项，出现在原来自由输入文本域的位置，就像下面的图显示的一样。



自己动手添加状态下拉菜单

我们在处理问题（**Issue**）状态的时候，要采用相同的方法。就像第 3 章提到的那样，当我们将一个问题（**Issue**）引入到应用中来，它可以是三种状态中的一种：

§ 还没有开始

§ 已经开始了

§ 已经完成

我们将创建状态下拉列表的操作留给读者。在使用了我们处理类型时候相同的方法后，（我们希望你已经测试了第一个方法），并且类型和状态表单域应该已经变成了下拉列表的形式。表单应该看起来像下面的截图一样：

Create Issue

Fields with * are required.

Name *

Description

Project

Type

Bug

Status

Not Yet Started

Not Yet Started

Started

Finished

Requester

Create Time

Create User

Update Time

Update User

Create

Operations

List Issue

Manage Issue

修复所有者和请求者字段域

我们先前就已经注意到了问题（**Issue**）创建表单的另一个问题，所有者和请求者域还是自由填写的文本域。然而，我们知道，这两个域应该是整数，代表的是 **tbl_user** 表的外键。所以，我们还需要为这些域添加下拉菜单。我们将不会使用与创建类型和状态属性时相同的方法，因为问题（**Issue**）的所有者和请求者需要从表 **tbl_user** 中取到。让事情稍微复杂一点，因为并不是系统中的所有用户都在问题（**Issue**）所在的项目（**Project**）中，所以，我们不能使用从整个 **tbl_user** 表中取出的数据来填充下拉表单。我们要将列表限制在一个与这个项目（**Project**）相关联的用户范围内。

这就引出了另一个我们需要解决的问题。像迭代计划中说的那样，我们需要在一个特定项目的上下文中来管理我们的问题（**Issue**）。也就是说，甚至在我们选定了一个特定的项目（**Project**）之前，你都无法看到创建问题（**Issue**）的表单。目前，我们的应用功能并不支持这个工作流程。

让我们逐一来解决这些问题。首先，我们将修改应用，来强制在管理操作与项目（**Project**）相关的问题（**Issue**）之前，必须先选中一个有效的项目（**Project**）。一旦一个项目（**Project**）被选定，我们将确保所有者和请求者下拉列表中的选择项，只能是与这个项目（**Project**）关联起来的用户。

强制选择一个项目（**Project**）上下文

我们想要确保在任何与问题（**Issue**）相关的功能被操作之前，必须处于一个有效的项目（**Project**）上下文中。为了做到这一点，要实现一个过滤器。Yii 中的过滤器是指，通过配置，在一个控制器的动作被执行之前或者之后执行的一小段代码。一个普遍的例子就是，当我们要求执行某个特定的控制器动作之前，用户必须已经登录，那么可以写一个简单的访问过滤器在这个动作执行之前来检查这个要求。另一个例子是，

如果我们想要在某个动作执行后额外记录些什么，或者执行一些审核逻辑，可以编写一个简单的审计过滤器来进行这个动作之后的处理任务。

在这个案例中，我们想要保证一个有效的项目（**Project**）必须在创建一个问题（**Issue**）之前被选中。所以，需要添加一个项目（**Project**）过滤器到 **IssueController** 类中来实现。

实现一个过滤器

一个过滤器可以被定义为一个控制器类的方法，或者单独建立一个类。当使用前者时，过滤器方法的命名必须以 **filter** 开头，并且有一个特定的签名。例如，如果我们想要创建一个过滤器方法，叫 **SomeMethoName**，我们的完整过滤器方法将像这样：

PHP 代码：

```
public function filterSomeMethodName($filterChain)

{

    ...

}
```

另一个创建过滤器的方法是写一个单独的类来实现过滤器的逻辑。使用此种方法时，类必须继承自 **CFilter**，并且至少要重载 **preFilter()** 和 **postFilter()** 两个方法中的一个，具体取决于逻辑应该在动作被调用之前执行还是之后执行。

添加一个过滤器

现在，让我们添加一个过滤器到 **IssueController** 类中，来处理有效的项目（**Project**）。现在先使用最简单的方法，添加一个以 **filter** 开头的方法到类里。因为这个方法的调用是 **Yii** 框架自动执行的，所以，我们很难使用先行测试的流程来实现这个方法。在这个案例里面，我们会稍微打破一下惯例，添加个方法之前，我们不会写测试。

打开 **protected/controllers/IssueController.php** 并且添加下面的方法到类的底部：

PHP 代码：

```
public function filterProjectContext($filterChain)

{

    $filterChain->run();

}
```

好了，现在我们已经定义了一个过滤器，但是却并没有做太多事情。它只是执行了 `$filterChain->run()`，这个操作将继续进行过滤处理，并允许执行使用了这个过滤器的操作方法。这就带来了另一个问题：我们怎样定义指定的操作方法使用这个过滤器？

指定过滤动作

如果我们要指定在哪个动作方法中，应该应用过滤器，需要重载 Yii 框架中控制器的基类 `CController` 的方法 `filters()`。事实上，这个方法已经在 `IssueController` 类中被重载了。是在我们使用 Gii 工具自动生成这个类的时候，就已经完成了。它已经帮我们添加了一个简单的访问控制过滤器 `accessControl`，这个方法在 `CController` 基类中已经被定义过了，用来处理一些基本的验证以保证用户有足够的权限来执行特定的动作。我们将会在下一章中涉及到用户验证和鉴权。现在，我们只需要把这个过滤器添加到过滤器配置数组。为了让我们的新过滤器应用到创建动作，添加如下高亮处代码到 `IssueController::filters()` 方法中：

PHP 代码：

```
/**
 * @return array action filters
 */
public function filters()
{
    return array(
        'accessControl', //perform access control for CRUD operations
        'projectContext + create', //check to ensure valid project context
    );
}
```

`filters()` 方法会返回一个包含了过滤器配置的数组。前面的方法返回了一个配置，指定了 `projectContext` 过滤器，该过滤器是在类中定义的方法，应用到 `actionCreate()` 上。这个配置的语法，允许“+”号和“-”号，用来指定是否要应用到一个方法上。例如，如果我们决定要将这个过滤器应用到除了 `actionUpdate()` 和 `actionView()` 之外的所有动作上，我们可以这样写：

PHP 代码：

```
return array(
    'projectContext - update, view' ,
```



```
);
```

你不能同时使用加号和减号。在任何给出的过滤器配置中，只能使用一个。加号操作符意味着“只在下列动作中使用过滤器”。减号则意味着“在除了下列动作之外的所有动作中应用过滤器”。如果既不使用‘+’，也不使用‘-’，那么过滤器会被应用到所有的动作中。

此时，我们将只限制创建动作。所以，先前用 **+ create** 来配置过滤器，我们的过滤器方法将在任何用户试图创建一个问题（**Issue**）时候被调用。

添加一些过滤器逻辑

好，现在我们已经定义了一个过滤器了，并且，我们配置了在 **IssueController** 类中 **actionCreate()** 方法被调用的时候调用此过滤器。然而，它还是不能提供必须的逻辑。我们希望确保在执行上述动作之前，处于当前项目（**Project**）环境，所以，我们需要在 **\$filterChain->run()** 方法运行之前，把相应的逻辑放入到这个过滤器方法中。

我们将会在控制器类中添加一个项目（**Project**）属性。我们将会在 **URL** 地址中使用一个查询参数来作为项目（**Project**）标识符。动作前过滤器会检查存在的项目（**Project**）属性是否为空。如果不为空，它会使用查询参数作为主键标识符找到项目（**Project**）。如果成功，动作会被执行，如果失败，会抛出一个异常。下面是 **IssueController** 中实现上述功能所必须的代码。

PHP 代码:

```
class IssueController extends CController {  
  
    .....  
  
    /**  
     * @var private property containing the associated Project model instance.  
     */  
  
    private $_project = null;  
  
    /**  
     * Protected method to load the associated Project model class  
     * @project_id the primary identifier of the associated Project  
     * @return object the Project data model based on the primary key
```

```

*/

protected function loadProject($project_id) {

    //if the project property is null, create it based on input id

    if ($this->_project === null) {

        $this->_project = Project::model()->findbyPk($project_id);

        if ($this->_project === null) {

            throw new CHttpException(404, 'The requested project does not exist.');
        }

    }

    return $this->_project;

}

```

```

/**

    * In-class defined filter method, configured for use in the above filters() method

```

** It is called before the createAction() action method is run in order to ensure a proper project context*

```

*/

public function filterProjectContext($filterChain) {

    //set the project identifier based on either the GET or POST input

    //request variables, since we allow both types for our actions

    $projectId = null;

    if (isset($_GET['pid']))

        $projectId = $_GET['pid'];

    else

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

```

```

        $projectId = $_POST['pid'];

        $this->loadProject($projectId);

        //complete the running of other filters and execute the requested action

        $filterChain->run();

    }

    ...

}

```

在这里，在问题（Issue）列表页面点击创建问题（Issue）链接

（<http://hostname/trackstar/index.php?r=issue/list>），尝试去创建一个新的问题（Issue）。

你会遇到一个 **404** 错误页面，而且会看到刚才我们定义的错误信息，请求的项目（Project）是不存在的。

这很好。它说明我们已经恰当地实现了阻止当没有项目（Project）被指定时创建问题（Issue）的运动的代码。要避免这个错误的最简单的方法，是在创建一个新的问题（Issue）时，在 URL 中添加一个参数来指定 pid。让我们来试一下这样做，来提供一个有效的项目（Project）标识符吧，并且处理表单来创建一个新的问题（Issue）。

添加项目 ID

回到第五章，当我们测试和实现项目（Project）的 CRUD 操作时，我们创建了好几个新的项目（Project）。所以，看起来在开发数据库中，你已经有了一个有效的项目 ID 了。如果没有，使用应用再次创建一个新的项目。一旦完成，注意一下项目的 ID，我们需要将这个 ID 写到新建问题（Issue）URL 中。

我们需要修改的链接在问题（Issue）列表页面对应的试图当中：`/protected/views/issue/index.php`。在那个文件的顶部，你将看到新建的链接在菜单中被指定，如下方高亮的代码：

PHP 代码：

```

$this->menu=array(

    array('label'=>'Create Issue', 'url'=>array('create')),

    array('label'=>'Manage Issue', 'url'=>array('admin')),

);

```

为了添加一个查询参数到则个链接中，我们只是简单地添加一个 `name=>value` 键值对到定义 `url` 的数组中。我们为过滤器添加的代码，查询参数的名字叫 `pid`（表示项目 ID）。还有，因为我们使用第一个（`project ID = 1`）项目（**Project**）来举这个例子，我们要将新建问题（**Issue**）这个链接该成如下样子：

PHP 代码：

```
array('label'=>'Create Issue', 'url'=>array('create', 'pid'=>1)),
```

现在当你再查看问题（**Issue**）列表页面的时候，你会看到新建问题（**Issue**）的超链接的 URL 已经带有了一个查询参数：<http://localhost/trackstar/index.php?r=issue/create&pid=1>

查询参数允许过滤器正确地设定项目（**Project**）上下文。所以，现在你再点击超链接，不再会显示 404 页面，而是打开了问题（**Issue**）创建的表单。

修改项目详情页面

在创建新问题（**Issue**）的链接中添加了项目（**Project**）ID 到 URL 中作为参数，是确保我们的过滤器像我们希望的那样工作的很好的第一步。然而，现在我们已经硬编码了那个链接，使得我们总是将新的问题（**Issue**）与项目 ID 为 1 的项目关联起来。当然，这不是我们想要的。我们想要的是创建新问题（**Issue**）的链接成为项目（**Project**）详情页面的一部分。这样，当你从项目（**Project**）列表选定一个项目（**Project**）的时候，就可以知道项目（**Project**）上下文，我们可以自动地把项目（**Project**）的 ID 追加到新建问题（**Issue**）的链接的末尾，现在让我们来实现它。

打开项目（**Project**）详情对应的视图，`/protected/views/project/view.php`，在这个文件的顶部，你将注意到菜单项包含在了 `$this->menu` 这个数组中。我们需要再添加一个创建新问题（**Issue**）的链接到这个列表的末尾，我们这样定义这个链接：

PHP 代码：

```
$this->menu = array(

    array('label' => 'List Project', 'url' => array('index')),

    array('label' => 'Create Project', 'url' => array('create')),

    array('label' => 'Update Project', 'url' => array('update', 'id' => $model->id)),

    array('label' => 'Delete Project', 'url' => '#', 'linkOptions' => array('submit' =>
array('delete', 'id' => $model->id), 'confirm' => 'Are you sure you want to delete this item?')),

    array('label' => 'Manage Project', 'url' => array('admin')),

    array('label' => 'Create Issue', 'url' => array('issue/create', 'pid' => $model->id)),
    //! t; ---注意，这一行

);
```

刚才我们做的，已经将创建新问题（**Issue**）的菜单选项移到了一个特定项目（**Project**）的详情页面中。我们使用了一个与先前相似的超链接，但是这次，我们指定了完整的控制器/动作对（**issue/create**）。而且，这次我们没有将项目的 **ID** 硬编码为 **1**，而是在这个视图文件中使用了 **\$model** 变量，它对应着特定的项目的 **AR** 对象。使用这种方式，不用考虑我们选择的项目，这个变量总是会给出正确的项目 **id** 属性。

移除项目输入表单域

现在，当我们要创建一个新的问题（**Issue**）时，已经正确设定了项目（**Project**）上下文，我们可以从表单中移除项目（**Project**）表单域。但是，我们还是需要有一个项目（**Project**）**ID** 与表单一起提交。因为我们在生成这个表单之前，已经知道了项目（**Project**）的 **ID**，所以我们可以创建动作中设定项目（**Project**）模型的属性。使用这种方法，传给视图文件的 **\$model** 实例将会已经带有恰当的项目（**Project**）**ID** 了。

首先，让我们来修改 **IssueController::actionCreate()** 方法，在其以创建后，就为问题（**Issue**）模型的实例指定一个 **project_id** 属性：

```
public function actionCreate() { $model=new Issue; $model->project_id =  
$this->_project->id; ... }
```

现在表单文件中，**project_id** 属性已经设定了。

打开新问题（**Issue**）表单的视图文件，**/protected/views/issue/_form.php**。移除下列与项目（**Project**）输入域关联的几行代码：

PHP 代码：

```
<div class="row">  
  
    <?php echo $form->labelEx($model , 'project_id'); ?>  
  
    <?php echo $form->textField($model , 'project_id'); ?>  
  
    <?php echo $form->error($model , 'project_id'); ?>  
  
</div>
```

将它们替换为一个隐藏域：

PHP 代码：

```
<div class="row">  
  
    <?php echo $form->hiddenField($model , 'project_id'); ?>  
  
</div>
```

现在当我们提交表单的时候，`project_id` 属性会被正确地设定。尽管我们现在还没有设定我们的所有者和请求者下拉列表，我们已经可以提交表单来创建一个正确设定了项目（**Project**）ID 的问题（**Issue**）了。

回到所有者和请求者下拉域

最终，我们回到我们起初想要做的，将所有者和请求者两个表单域改成下拉选项，里面包含着这个项目（**Project**）的有效成员。为了正确的完成这些，我们需要给一个项目（**Project**）关联一些用户。因为在第 7 章和第 8 章中会介绍用户管理，所以这里，我们将会通过 SQL 语句直接在数据库中添加一些关联。在我们早先的 DDL 语句中，我们已经添加了两个新的测试用户作为我们的种子数据。为了提醒一下，那个 `insert` 语句在下面：

SQL 代码：

```
INSERT INTO `tbl_user`  
  
(`email`, `username`, `password`)  
  
VALUES  
  
('test1@notanaddress.com', 'Test_User_One', MD5('test1')),  
  
('test2@notanaddress.com', 'Test_User_Two', MD5('test2'));
```

这段代码在我们的系统中创建了两个 ID 为 1 和 2 的新用户。让我们手动将这两个用户分派给项目（**Project**）#1。

为了做到这一点，在你的 `trackstar_dev` 和 `trackstar_test` 数据库中，运行下面的 `insert` 语句：

SQL 代码：

```
INSERT INTO `tbl_project_user_assignment` (`project_id`, `user_id`) VALUES (1,1), (1,2);
```

运行了前面所述的 SQL 语句后，我们已经有了两个有效的成员分配给了项目（**Project**）#1。

Yii 中关系型活动记录一个比较神奇的特性是，直接从问题（**Issue**）`$model` 的实例中访问问题（**Issue**）所属项目的有效成员的能力。当我们使用 Gii 工具初始创建我们的问题（**Issue**）模型类时，它足够聪明地查看下面的数据库并且创建相关的关系。可以从 `relations()` 方法中看到，在 `/protected/models/Issue.php` 文件中可以查看。因为我们在创建了恰当的数据库关系后，才创建的这个类，所以，方法应该看起来是这样的：

PHP 代码：

```
/**  
  
 * @return array relational rules.
```

```

*/

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(

        'requester' => array(self::BELONGS_TO, 'User', 'requester_id'),

        'owner' => array(self::BELONGS_TO, 'User', 'owner_id'),

        'project' => array(self::BELONGS_TO, 'Project', 'project_id'),

    );

}

```

就像 **NOTE** 中建议的那样，你的属性名称可能有轻微不同，请按照需要将它们调整好。这个数组配置定义了模型实例的属性，而这几个属性本身也是其他的 **AR** 实例。这里有了这些关系，我们可以相当简单地访问相关 **AR** 实例。例如说，我们想要访问与问题（**Issue**）关联的项目模型类。我们可以通过使用下面的语法来做到这一点：

PHP 代码：

```

//create the model instance by primary key:

```

```

$model = Issue::model()->findbyPk(1);

```

```

//access the associated Project AR instance

```

```

$project = $model->project;

```

现在，因为我们在数据库中定义其他的表和关系之前就已经建立了我们的项目（**Project**）模型类，所以关系还没有定义。然而，现在我们已经定义了一些关系，我们需要将这些关系添加到 **Project::relations()** 方法中。打开项目（**Project**）AR 类文件 **/protected/models/Project.php**，使用下面的代码替换整个 **relations()** 方法：

PHP 代码：

```

/**

 * @return array relational rules.

 */

```

```

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(

        'issues' => array(self::HAS_MANY, 'Issue', 'project_id'),

        'users' => array(self::MANY_MANY, 'User',

                        'tbl_project_user_assignment(project_id, user_id)'),

    );

}

```

有了这些，我们可以相当简单的访问所有的与项目（**Project**）关联的问题（**Issue**）和/或用户：

PHP 代码：

```

//create the Project model instance by primary key:

$model = Project::model()->findByPk(1);

//get an array of all associated Issue AR instances

$allIssues = $model->issues;

//get an array of all associated User AR instance

$allUsers = $model->users;

//get the User AR instance representing the owner of

//the first issue associated with this project

$ownerOfFirstIssue = $model->issues[0]->owner;

```

通常我们需要写复杂的 **SQL join** 语句来访问这样的关系型数据。而使用了 **Yii** 中的关系型 **AR**，将我们从这种复杂漫长痛苦的过程中解放出来。我们现在可以使用优雅精简的面向对象语法来访问这些关系了。

生成数据来填充下拉菜单

现在，我们已经知道了数个方法，都可以生成填充“所有者”和“请求者”下拉选项的数据。我们将使用一个与生成状态和类型下拉数据时相似的方法，并且把逻辑放到模型类中。在这个例子中，项目 **AR** 类是主角，因为一个有效的用户是与一个项目关联的，而不是与一个问题（**Issue**）关联。

因为我们将要添加一个公用方法到我们的项目 **AR** 类中，所以我们可以再一次地使用 **TDD** 方法了。好，让我们快点写一个失败的测试吧。

再一次，请记住我们已经设定了一个 **trackstar_test** 数据库用于测试。如果你一直实践着上面的代码，请确保这个测试数据库的结构和 **trackstar_dev** 数据库保持一致。

打开 `/protected/tests/unit/ProjectTest.php` 文件添加如下测试：

PHP 代码：

```
public function testGetUserOptions()
{
    $project = $this->projects('project1');

    $options = $project->userOptions;

    $this->assertTrue(is_array($options));
}
```

现在运行测试：

SHELL 代码或屏幕回显：

```
>>phpunit unit/ProjectTest.php
```

```
PHPUnit 3.3.17 by Sebastian Bergmann.
```

```
....E
```

```
Time: 0 seconds There was 1 error:
```

```
1) ProjectTest::testGetUserOptions
```

```
CException: Property "Project.userOptions" is not defined....
```

```
FAILURES!
```

```
Tests: 5, Assertions: 10, Errors: 1.
```

好，我们有了一个没有通过的测试。它失败的原因很显然，因为我们测试了一个在项目 **AR** 类中并不存在的方法。所以，让我们来添加它。打开 `/protected/models/Project.php` 文件，添加下列方法到类的底部：

PHP 代码：

```
/**
 * @return array of valid users for this project, indexed by user IDs
 */
public function getUserOptions()
{
    $usersArray = array(); return $usersArray;
}
```

如果我们再运行一次测试，会看到我们又回到了“绿色区”。然而，我们只有一个返回空数组的方法。我们需要的是一个有效的用户数组来填充表单里的下拉选项。让我们修改测试，确保返回的数组包含的元素数量大于 0，来让我们的测试再次变成“红色区”。

将测试改成下面的样子：

PHP 代码：

```
public function testGetUserOptions()
{
    $project = $this->projects('project1');

    $options = $project->userOptions;

    $this->assertTrue(is_array($options));

    $this->assertTrue(count($options) > 0);
}
```

再次运行测试，应该看到如下的错误结果：

SHELL 代码或屏幕回显：

```
There was 1 failure:
```

```
1) ProjectTest::testGetUserOptions
```

Failed asserting that <boolean:false> is true.

所以，让我们再次回到 **Project::getUserOptions()** 方法，返回一些真实的用户。将方法修改成：

PHP 代码：

```
public function getUserOptions()
{
    $usersArray = CHtml::listData($this->users, 'id', 'username');

    return $usersArray;
}
```

这里我们使用了 Yii 的 CHtml 助手类帮助我们创建一个数组包含了 **id=>username** 键值对表达的与项目（**Project**）关联的每个用户。还记得 **Project** 类的 **users** 属性映射到了一个用户 AR 实例的数组上。**CHtml::listData()** 方法可以利用这个列表并且生成一个适合 **CActiveForm::dropDownList()** 使用的有效的数组。现在，只要我们记得用我们两个用户以及与他们关联的项目（**Project**）#1 来填充我们的测试数据库，我们测试将会顺利通过。

添加“用户”和“用户项目关系表”夹具

我们的测试现在通过了，这只是因为我们显式添加了用户，并且我们还显式地添加了条目到项目（**Project**）关联表中。如果有人过来移除了这些条目会怎么样呢？我们需要修复这个脆弱的关系。我们已经知道，测试夹具正是我们需要来确保与我们的测试涉及到的数据库数据可以以一种比较稳定的形式重复运行的东西。我们曾经为我们的项目（**Project**）数据做过相同的事情。我们需要为与 **tbl_user** 和 **tbl_project_user_assignment** 表关联的数据再做一次。

创建一个文件，**/protected/tests/fixtures/tbl_user.php**，并且添加如下代码：

PHP 代码：

```
return array(

    'user1' => array(

        'email' => 'test1@notanaddress.com',

        'username' => 'Test_User_One',

        'password' => MD5('test1'),
```

```

        'last_login_time' => '',

        'create_time' => '',

        'create_user_id' => '',

        'update_time' => '',

        'update_user_id' => '',

    ),

    'user2' => array(

        'email' => 'test2@notanaddress.com',

        'username' => 'Test_User_Two',

        'password' => MD5('test2'),

        'last_login_time' => '',

        'create_time' => '',

        'create_user_id' => '',

        'update_time' => '',

        'update_user_id' => '',

    ),

);

```

这与我们通过显式 **SQL** 手动添加的数据是相同的内容，但是这里它们被表示成了固定的数据。

我们需要为我们的关联表做相同的事情。创建一个新文件，
/protected/tests/fixtures/tbl_project_user_assignment.php 并且添加如下代码：

PHP 代码：

```

return array(

    'user1ToProject1' => array(

        'project_id' => 1,

        'user_id' => 1,

```

```

        'create_time' => '',

        'create_user_id' => '',

        'update_time' => '',

        'update_user_id' => '',

    ),

    'user2ToProject1' => array(

        'project_id' => 1,

        'user_id' => 2,

        'create_time' => '',

        'create_user_id' => '',

        'update_time' => '',

        'update_user_id' => '',

    ),

);

```

这与我们手动添加到 `tbl_project_user_assignment` 表中的数据相同，只是被表达成了固定的数据。

现在我们需要将这些固件添加到单元测试中。打开 `ProjectTest` 文件，
`/protected/tests/unit/ProjectTest.php`，将它添加到文件顶部固件的定义中，如下方高亮代码示意的一样：

PHP 代码：

```

public $fixtures = array(

    'projects' => 'Project',

    'users' => 'User',

    'projUsrAssi gn' => ':tbl_project_user_assignment',

);

```

注意当映射到表 `tbl_project_user_assignment` 时，我们必须添加“: ”。这用来表示这是一个数据库表，而不是一个 **AR** 模型类。

现在这些已经被添加了，每次我们运行测试 **ProjectTest.php** 的时候，我们的 **tbl_user** 和 **tbl_project_user_assignment** 表将被在固件中定义的数据置为惯常的状态。

现在让我们再运行一次跟项目（**Project**）有关的测试：

SHELL 代码或屏幕回显：

```
>> unit/ProjectTest.php

PHPUnit 3.4.12 by Sebastian Bergmann.

..... Time: 0 seconds

OK (5 tests, 12 assertions)
```

我们仍然通过了测试，但是现在他们使用的夹具中的数据了。

现在我们已经有了我们的 **getUserOptions()** 方法了，我们需要实现下拉列表来它显示返回的数据。我们已经添加了一个私有的 **\$_project** 属性到我们的 **IssueController** 类中。这个属性包含了有效的项目（**Project**）上下文。我们需要在视图文件中访问这个相同的项目（**Project**）属性来显示输入表单。所以，我们需要添加一个简单的 **getter** 方法来将这个私有属性给暴露出来。添加下列方法到 **IssueController** 类的底部：

PHP 代码：

```
/**
 * Returns the project model instance to which this issue belongs
 */

public function getProject() {

    return $this->_project;

}
```

现在，打开包含了输入表单的视图文件，**/protected/views/issue/_form.php**，找到 **owner_id** 和 **requester_id** 这两个文字输入表单域元素定义的地方。

替换

PHP 代码：

```
<?php //echo $form->textField($model, 'owner_id'); ?>
```

为以下代码：

PHP 代码:

```
<?php echo $form->dropDownList($model, 'owner_id', $this->getProject()->getUserOptions()); ?>
```

再替换这行

PHP 代码:

```
<?php echo $form->textField($model, 'requester_id'); ?>
```

为以下代码

PHP 代码:

```
<?php echo $form->dropDownList($model, 'requester_id', $this->getProject()->getUserOptions()); ?>
```

现在如果我们再一次查看问题（**Issue**）创建表单，我们看到所有者和请求者两个域已经变成了两个很漂亮的下拉表单域。

做最后一个修改

因为我们已经打开了创建问题（**Issue**）的表单视图文件，让我们来快速地做最后一个修改。创建时间和用户，以及最后修改时间和用户是我们用来记录修改历史和审计用的，不应该暴露给用户。最后，我们将修改应用逻辑，根据问题（**Issue**）的插入动作和更新动作来自动填写这些域。现在，只是将他们对应的表单输入域移除。

简单地将下列代码从文件 `/protected/views/issue/_form.php` 中移除:

PHP 代码:

```
<div class="row">

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

    <?php echo $form->textField($model, 'create_time'); ?>

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

</div>

<div class="row">

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

```
<?php echo $form->textField($model, 'create_user_id'); ?>

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

</div>

<div class="row">

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

    <?php echo $form->textField($model, 'update_time'); ?>

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

</div>

<div class="row">

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

    <?php echo $form->textField($model, 'update_user_id'); ?>

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

</div>
```

下列截图显示了我们的新问题（**Issue**）创建表单现在的样子：

The screenshot shows a web application interface for creating an issue. At the top, there's a navigation bar with links: Home, About, Contact, and Logout (demo). Below this, a breadcrumb trail shows 'Home » Issues » Create'. The main heading is 'Create Issue'. A note states 'Fields with * are required.' The form contains several fields: 'Name' (required), 'Description', 'Type' (a dropdown menu currently showing 'Bug'), 'Status' (a dropdown menu currently showing 'Not Yet Started'), 'Owner' (a dropdown menu currently showing 'Test_User_One'), and 'Requester' (a dropdown menu currently showing 'Test_User_One'). At the bottom of the form is a 'Create' button. On the right side, there's a sidebar with a section titled 'Operations' containing two links: 'List Issue' and 'Manage Issue'.

完成剩下的 CRUD 操作

这一个迭代的目标是实现问题（**Issue**）的所有 **CRUD** 操作。我们已经将创建功能完成了，但是我们仍然需要来完成读，更新和删除问题（**Issue**）的操作。幸运的是，所有的基础已经被 **Gii** 的代码生成功能完成了。然而，因为我们想要问题（**Issue**）的所有操作都在项目（**Project**）的上下文中来完成，我们需要做一些修改来决定我们如何访问这些功能。

列出问题（Issue）

尽管在 **IssueController** 类中已经有了 **actionIndex()** 方法能显示数据库中的问题（**Issue**）列表，我们不需要这个目前已经写好的功能。我们想要一个只列出与特定项目（**Project**）相关的问题（**Issue**）的页面，而不是一个列出数据库中所有的问题（**Issue**）的列表。所以，我们修改应用来让项目（**Project**）页面的一部分显示问题（**Issue**）的列表。因为我们使用了关系型 **AR** 模型，这个修改将会非常简单。

修改 ProjectController

首先，让我们修改 **ProjectController** 类中的 **actionView()** 方法。因为我们想显示一个与项目（**Project**）相关的问题（**Issue**）的列表，我们可以它显示在项目（**Project**）详情页面上。方法 **actionView()** 是显示项目（**Project**）详情的方法。

将那个方法修改成：

PHP 代码：

```

/**
 * Displays a particular model.
 */

public function actionView() {

    $id = $_GET['id'];

    $issueDataProvider = new CActiveDataProvider('Issue', array(

        'criteria' => array(

            'condition' => 'project_id=:projectId',

            'params' => array(

                'projectId' => $this->loadModel($id)->id),

            ),

        'pagination' => array('pageSize' => 1),

    ));

    $this->render('view', array(

        'model' => $this->loadModel($id),

        'issueDataProvider' => $issueDataProvider,

    ));

}

```

这里，我们使用了 **CActiveDataProvider** 框架类来为 **ActiveRecord** 对象提供数据。它将使用关联的 **AR** 模型类从数据库中取数据，并且可以很容易的被 **Zii** 挂件 **CListView** 简单地以列表的形式显示在视图文件中。我们已经使用了 **criteria** 属性来指定条件，它只能取与当前显示的项目相关的问题（**Issue**）。我们已经使用了 **pagination** 属性来限制问题（**Issue**）列表每页只显示一个问题（**Issue**）。我们设定了如此低的一个数字，是为了只要添加两个问题（**Issue**），我们就能快速地看到分页的特性。我们马上演示这个特性。

最后的事情我们要做的就是，将这个数据提供者 **\$issueDataProvider** 添加到 **render()** 方法的参数中，好让视图文件能访问到。

修改项目（Project）视图文件

我们将使用 **Zii** 挂件 **CListView** 来显示我们的问题（**Issue**）列表在项目（**Project**）详情页面上。打开文件 `/protected/views/project/view.php`，然后添加下面代码到文件底下：

PHP 代码：

```
<br>

<h1>Project Issues</h1>

<?php $this->widget('zii.widgets.CListView', array(

    'dataProvider' => $issueDataProvider,

    'itemView' => '/issue/_view'

));

?>
```

这里我们将 **CListView** 的 **dataProvider** 属性设为刚才我们创建的问题（**Issue**）数据提供者。然后我们配置它使用 `/protected/views/issue/_views.php` 文件作为模板为数据提供者中的每个数据生成列表项。这个文件在我们使用 **Gii** 工具创建问题（**Issue**）的 **CRUD** 操作的时候，就已经被创建了。我们只是使用它来在项目（**Project**）详情页面显示问题（**Issue**）。

我们还需要对文件 `/protected/views/issue/_view.php` 做一些修改，我们要为每个问题（**Issue**）指定一个布局模板。将那个文件的整个内容修改成如下的样子：

PHP 代码：

```
<div class="view">

    <b><?php echo CHtml::encode($data->getAttributeLabel('name')); ?></b>

    <?php echo CHtml::link(CHtml::encode($data->name), array('issue/view', 'id' =>
    $data->id)); ?>

    <br />

    <b><?php echo CHtml::encode($data->getAttributeLabel('description')); ?></b>

    <?php echo CHtml::encode($data->description); ?>

    <br />

    <b><?php echo CHtml::encode($data->getAttributeLabel('type_id')); ?></b>
```

```

<?php echo CHtml::encode($data->type_id); ?>

<br />

<b><?php echo CHtml::encode($data->getAttributeLabel('status_id')); ?>:</b>

<?php echo CHtml::encode($data->status_id); ?>

</div>

```

现在，如果我们保存，并且查看我们的项目（Project）详情页面，项目（Project）1（<http://localhost/trackstar/index.php?r=project/view&id=1>），假如你已经创建了几个问题（Issue）在你的项目（Project）下，你将会看到下面的页面：

My Web Application

Home About Contact Logout (demo)

Home » Projects » Test project one

View Project #1

ID	1
Name	Test project one
Description	This is the first test project
Create Time	0000-00-00 00:00:00
Create User	Not set
Update Time	0000-00-00 00:00:00
Update User	Not set

Operations

- List Project
- Create Project
- Update Project
- Delete Project
- Manage Project
- Create Issue

Project Issues

Displaying 1-1 of 1 result(s).

Name: [Test Issue For Project 1](#)

Description: this is a test bug for project 1

Type: 0

Status: 0

因为我们给数据提供者的 **pagination** 属性设定了非常低的值（1），我们可以再添加一个问题（Issue）来展示内建的分页功能。再添加一个问题（Issue），会让问题（Issue）的显示增加链接允许我们按照页面来访问项目（Project）的问题（Issue）列表，像下面的截屏描述的那样：

Project Issues

Displaying 2-2 of 2 result(s).

Name: [Test Feature for project 1](#)

Description: this is a test feature for project 1

Type: 1

Status: 0

Go to page: < Previous 1 2 Next >

做一些最后的调整

现在，在项目（**Project**）详情页面，我们已经可以展示与项目（**Project**）相关的问题（**Issue**）列表了。我们还可以查看问题（**Issue**）的详情，还有更新和删除问题（**Issue**）。所以，**CRUD** 的绝大部分功能都已经就位了。

然而，在我们结束这个迭代之前，还是有一些问题（**Issue**）需要解决。我们可以注意到，问题（**Issue**）列表使用数字 **ID** 来显示类型，状态，所有者和请求者域。我们应该把他们改成对应的文字值。还有，因为问题（**Issue**）已经在某个项目（**Project**）下面了，那么在问题（**Issue**）的列表上，显示项目（**Project**）的 **ID** 就有点多余了。所以，我们可以把这个移除掉。最后，我们需要解决一些其他表单页面的导航链接，使它们总是能回到项目（**Project**）详情页面来开始用户的问题（**Issue**）管理。

我们将在接下来解决。

使状态和类型文字显示出来

先前，我们已经往问题（**Issue**）**AR** 类中添加了公有方法来取得状态和类型选项，来填充问题（**Issue**）创建表单的下拉选项。我们还需要在这个 **AR** 类中添加类似的方法来返回特定标识符的问题（**Issue**）来显示我们的问题（**Issue**）列表。

因为这些方法是问题（**Issue**）**AR** 模型类的公有方法，我们将在使用 **TDD** 方法来实现它。为了让事情进展的更快一点，我们将这两者放到一起来做。而且，我们将 **TDD** 放一下，我们将开始一个更大的步骤。我们可以总是返回一个更紧凑的方法。

首先，我们需要添加一些固件数据来确保我们有一些问题（**Issue**）与一个项目（**Project**）关联起来。我们还要确保我们的问题（**Issue**）测试使用了项目（**Project**）固件数据同时，这个问题（**Issue**）属于这个项目（**Project**）。

首先，为问题（**Issue**）添加一个新的固件数据文件，`/protected/fixtures/tbl_issue.php`，然后添加下列内容：

PHP 代码：

```
return array('issueBug' => array(

    'name' => 'Test Bug 1',

    'description' => 'This is test bug for project 1',

    'project_id' => 1,

    'type_id' => 0,

    'status_id' => 1,
```

```

        'owner_id' => 1,

        'requester_id' => 2,

        'create_time' => '',

        'create_user_id' => '',

        'update_time' => '',

        'update_user_id' => '',

    ),

    'issueFeature' => array(

        'name' => 'Test Bug 2',

        'description' => 'This is test bug for project 2',

        'project_id' => 2,

        'type_id' => 1,

        'status_id' => 0,

        'owner_id' => 2,

        'requester_id' => 1,

        'create_time' => '',

        'create_user_id' => '',

        'update_time' => '',

        'update_user_id' => '',

    ),

);

```

现在我们需要配置我们的 **IssueTest** 类使用一些固件数据。将下列固件数组添加到问题（**Issue**）测试类的顶部：

PHP 代码：

```
public $fixtures=array(
```

```
'projects' => 'Project',  
  
'issues' => 'Issue',  
  
);
```

有了我们的固件数据就位后，我们可以添加两个新的测试到 **IssueTest** 单元测试中，用来测试状态和类型的文字：

PHP 代码：

```
public function testGetStatusText()  
{  
  
    $this->assertTrue('Started' == $this->issues('issueBug')->getStatusText());  
  
}
```

还有这个

PHP 代码：

```
public function testGetTypeText()  
{  
  
    $this->assertTrue('Bug' == $this->issues('issueBug')->getTypeText());  
  
}
```

现在，如果我们运行测试，我们将得到一个失败的提示，因为我们现在还没有添加这两个公有方法到我们的 **AR** 类中：

SHELL 代码或屏幕回显：

```
>>phpunit unit/IssueTest.php  
  
PHPUnit 3.4.12 by Sebastian Bergmann.  
  
..EE  
  
Time: 2 seconds, Memory: 12.25Mb  
  
There were 2 errors:
```

1) IssueTest::testGetStatusText

Exception: Unknown method 'issues' for class 'IssueTest'.

...

2) IssueTest::testGetTypeText

Exception: Unknown method 'issues' for class 'IssueTest'.

...

FAILURES!

Tests: 4, Assertions: 10, Errors: 2.

所以，我们已经得到了我们的失败的测试，让我们来添加必要的代码到 `/protected/models/Issue.php` 文件中，来让测试通过。将下列的公有方法添加到问题（**Issue**）类中来取得当前问题（**Issue**）的状态和类型文字：

PHP 代码：

```
/**
 * @return string the status text display for the current issue
 */
public function getStatusText() {
    $statusOptions = $this->statusOptions;

    return isset($statusOptions[$this->status_id]) ?

        $statusOptions[$this->status_id] :

        "unknown status ({ $this->status_id })";
}
```



```

/**
 * @return string the type text display for the current issue
 */

public function getTypeText() {

    $typeOptions = $this->typeOptions;

    return isset($typeOptions[$this->type_id]) ?

        $typeOptions[$this->type_id] :

        "unknown type ({ $this->type_id })";

}

```

现在我们再次运行测试：

SHELL 代码或屏幕回显：

```

>>phpunit unit/IssueTest.php

....

Time: 1 second, Memory: 12.25Mb

OK (4 tests, 12 assertions)

```

我们已经通过了测试，并且回到了“绿色区”。

添加文字显示到表单

现在我们已经有了两个新的公有方法，可以返回有效的状态和类型的文字以帮助我们显示列表，我们需要利用好它们。修改 `/protected/views/issue/_view.php` 文件中的下列行：

修改下列命令：

PHP 代码：

```

<?php echo CHtml::encode($data->type_id); ?>

```

为：

PHP 代码:

```
<?php echo CHtml::encode($data->getTypeText()); ?>
```

并修改下列命令:

PHP 代码:

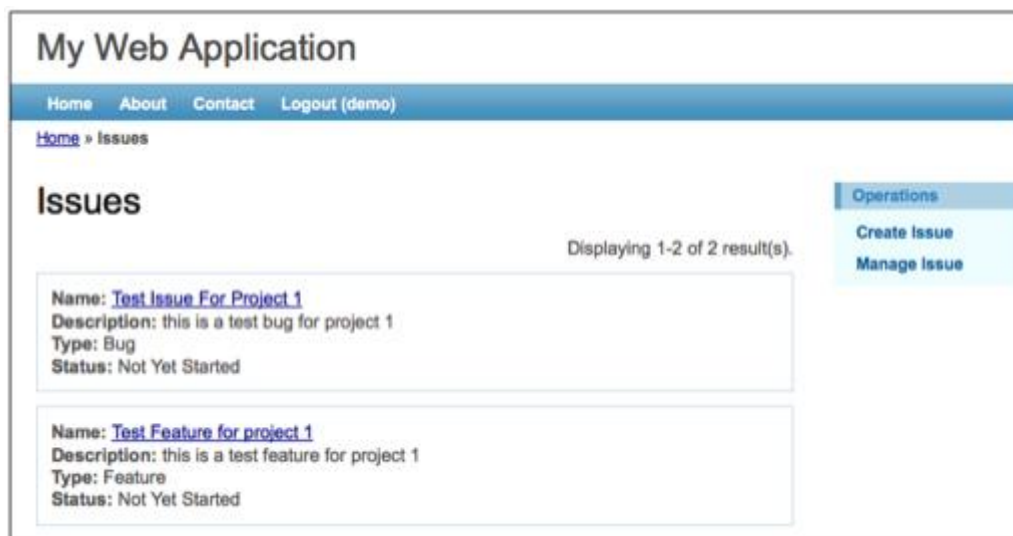
```
<?php echo CHtml::encode($data->status_id); ?>
```

为:

PHP 代码:

```
<?php echo CHtml::encode($data->getStatusText()); ?>
```

在这些修改之后, 我们的问题 (Issue) 列表页面, <http://localhost/trackstar/index.php?r=issue> 不再显示整数值到我们的类型和状态域中。它显示的内容看起来和下面的截屏比较像:



因为我们使用了相同的视图文件来显示问题 (Issue) 列表和我们的项目 (Project) 详情页面, 这些变化也在那边体现了出来。

改变问题 (Issue) 详情视图

我们还需要对问题 (Issue) 的详情页面做一些修更改。现在, 我们打开问题 (Issue) 的详情页面, 它看起来像下面的截屏:

View Issue #1	
ID	1
Name	Test Issue For Project 1
Description	this is a test bug for project 1
Project	1
Type	0
Status	0
Owner	1
Requester	1
Create Time	0000-00-00 00:00:00
Create User	Not set
Update Time	0000-00-00 00:00:00
Update User	Not set

这个页面使用了一个我们还没有修改的视图文件。它显示的还是项目（**Project**）的 **ID**，我们并不需要，而且类型和状态显示的还是整数值，而不是对应的文字值。打开生成这个页面的视图文件，`/protected/views/issue/view.php`，我们注意到它使用了 **Zii** 扩展挂件，**CDetailView**，我们以前并没有见过这个。这个和 **CListView** 是相似的东西，用于显示一个列表，但是它用于显示一个单一数据模型的详细内容，而不是显示一个多项的列表。这个文件中的相关代码显示了这个挂件的使用法：

PHP 代码：

```
$this->widget('zii.widgets.CDetailView', array(

    'data'=>$model,

    'attributes'=>array(

        'id',

        'name',

        'description',

        'project_id',

        'type_id',

        'status_id',

        'owner_id',

        'requester_id',

        'create_time',

        'create_user_id',
```

```

        'update_time',

        'update_user_id',

    ),

));

```

这里，我们将 **CDetailView** 挂件的数据模型设定为问题（**Issue**）模型类，然后在生成的细节视图中设定一个被显示的模型属性的列表。一个属性可以被指定是一个字符串，按照如下格式名字：类型：标签，类型和标签是可选的，或者属性本身就是一个数组。这里只指定了属性的名字。

如果我们指定了一个属性为一个数组，我们可以通过指定一个 **value** 元素来进一步自定义显示。我们将用这个方法使得类型和状态域分别使用 **getTypeText()** 和 **getStatusText()** 模型类方法显示。

让我们使用如下的配置来修改 **CDetailView**:

PHP 代码:

```

$this->widget('zii.widgets.CDetailView', array(

    'data' => $model,

    'attributes' => array(

        'id',

        'name',

        'description',

        array(

            'name' => 'type_id',

            'value' => CHtml::encode($model->getTypeText())

        ), array(

            'name' => 'status_id',

            'value' => CHtml::encode($model->getStatusText())

        ),

        'owner_id',

        'requester_id', ),

```

```
));
```

这里我们移除了几个属性，并不全部显示所有的。`project_id`, `create_time`, `update_time`, `create_user_id` 和 `update_user_id`。我们将在后面来出来这些字段的显示和填充，现在我们只是将它们从细节中移除。

我们还修改了 `type_id` 和 `status_id` 的声明，使用了数组来设定，这样我们可以使用 `value` 元素。我们已经指定使用 `Issue::getTypeText()` 和 `Issue::getStatusText()` 方法来取得这些属性的值。做了这些修改后，问题（**Issue**）详情页面看起来像这个样子。

View Issue #1	
ID	1
Name	Test Issue For Project 1
Description	this is a test bug for project 1
Type	Bug
Status	Not Yet Started
Owner	1
Requester	1

好，离我们想要的东西更进一步了，但是我们还需要做几个修改。

让所有者和请求者的名字显示出来

事情看起来更好了，但是我们还是看到了整数标识符显示在所有者和请求者的位置上，而不是显示真是的用户姓名。我们将使用处理类型和状态时候相似的方法。我们会添加两个公有方法在问题（**Issue**）模型类中，来返回这两个属性的名字。

使用关系型 AR

由于问题（**Issue**）和用户由不同的数据库表表示，并使用一个外键来关联，我们可以直接从视图文件中的 `$model` 访问所有者和请求者。利用 **Yii** 框架中 **AR** 模型的强大特性，可以很容易的在用户模型类实例中显示用户名属性。

我们已经提过了，模型的 `Issue::relations()` 方法是定义关系的地方。如果我们看一下这个方法，我们会看到：

PHP 代码：

```
/**
 * @return array relational rules.
 */
```

```

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(

        'requester' => array(self::BELONGS_TO, 'User', 'requester_id'),

        'owner' => array(self::BELONGS_TO, 'User', 'owner_id'),

        'project' => array(self::BELONGS_TO, 'Project', 'project_id'),

    );

}

```

高亮的代码是与我们需求高度相关的。所有者和请求者属性都被定义为了用户模型类的相关。这些定义指定了用户模型类实例中的属性的值。**owner_id** 和 **requester_id** 确定了他们对应的用户类实例的主键。所以，我们可以像访问其他问题（**Issue**）模型的属性一样访问他们。

所以，要显示用户类实例的所有者和请求者，我们再一次修改我们的 **CDetailView** 配置为：

PHP 代码：

```

$this->widget('zii.widgets.CDetailView', array(

    'data' => $model,

    'attributes' => array(

        'id',

        'name',

        'description',

        array(

            'name' => 'type_id',

            'value' => CHtml::encode($model->getTypeText())

        ), array(

            'name' => 'status_id',

```

```

        'value' => CHtml::encode($model->getStatusText())

    ),

    array(

        'name' => 'owner_id',

        'value' => CHtml::encode($model->owner->username)

    ),

    array(

        'name' => 'requester_id',

        'value' => CHtml::encode($model->requester->username)

    ),

    ),

));

```

做了上述修改后，我们的问题（**Issue**）详情页面看起来好看了不少。请看下面的截图：

View Issue #1	
ID	1
Name	Test Issue For Project 1
Description	this is a test bug for project 1
Type	Bug
Status	Not Yet Started
Owner	Test_User_One
Requester	Test_User_One

做最后的一些导航修改

我们离完成这个迭代之初计划的功能已经非常接近了。唯一剩下的就是要清理一下导航。你可能已经注意到了，仍旧有一些选项可以使得用户导航到全部问题（**Issue**）的列表，或者在项目（**Project**）的范围外，创建一个新的问题（**Issue**）。对于 **TrackStar** 应用的目的来说，与问题（**Issue**）相关的所有操作，应该在一个特定项目（**Project**）的上下文中。早先，我们强制在创建一个新的问题（**Issue**）的时候，指定项目（**Project**）上下文（很好的开始），但是我们还是需要一些修改。

我们注意的一个东西是，应用仍然允许用户导航到所有问题（**Issue**）的列表页面，没有项目（**Project**）的区分。例如，在一个问题（**Issue**）详情页面，如 <http://localhost/trackstar/index.php?r=issue/view&id=1>，我们看到右侧菜单导航有两个链接，列

出问题（Issue）和管理问题（Issue），对应着链接

<http://localhost/trackstar/index.php?r=issue/index> 和

<http://localhost/trackstar/index.php?r=issue/admin>（记得要访问管理页面，你必须登录

admin/admin）。这些页面仍然显示所有项目（Project）的所有的问题（Issue）。所以，我们必须限制这些列表到特定的项目（Project）。

由于这些链接原来是问题（Issue）详情页面的，那个特定的问题（Issue）已经关联了一个项目（Project），我们可以首先修更改这些链接，来传递一个特定的项目（Project）ID，然后把那个项目（Project）ID 传给 `IssueController::actionIndex()` 和 `IssueController::actionAdmin()` 方法。

首先，修改链接。打开文件 `/protected/views/issue/view.php`，找到文件顶部菜单项数组。修改菜单的配置为如下：

PHP 代码：

```
$this->menu = array(

    array('label' => 'List Issue', 'url' => array('index', 'pid' => $model->project->id)),

    array('label' => 'Create Issue', 'url' => array('create', 'pid' => $model->project->id)),

    array('label' => 'Update Issue', 'url' => array('update', 'id' => $model->id)),

    array('label' => 'Delete Issue', 'url' => '#', 'linkOptions' => array('submit' =>
array('delete', 'id' => $model->id), 'confirm' => 'Are you sure you want to delete this item?')),

    array('label' => 'Manage Issue', 'url' => array('admin', 'pid' => $model->project->id))

);
```

这些修改被高亮了。我们已经添加了一个新的查询参数到新的创建问题（Issue）链接，而且也添加到了问题（Issue）列表链接和问题（Issue）管理链接。我们已经知道，我们必须为创建问题（Issue）链接做此项修更改，因为我们之前已经实现了一个过滤器来强制指定一个项目（Project）上下文。我们将来不会进一步修改这个链接了，我们将修更改他们对应的动作方法来利用这个新的查询字符串变量。

由于我们已经配置了一个过滤器来利用查询字符串装载关联的项目（Project），让我们利用这个。我们将要修改过滤器配置，使得我们的过滤器在 `IssueController::actionIndex()` 和 `IssueController::actionAdmin()` 方法调用之前，被调用。修改过滤器方法如下：

PHP 代码：

```
/**

 * @return array action filters

 */
```



```

public function filters() {

    return array(

        'accessControl',

        // perform access control for CRUD operations

        'projectContext + create index admin',

        //perform a check to ensure valid project context

    );

}

```

在这里，关联的项目（**Project**）将会被装载。让我们在 **IssueController::actionIndex()**方法中来使用它。修改方法为：

PHP 代码：

```

public function actionIndex() {

    $dataProvider = new CActiveDataProvider('Issue',

        array(

            'criteria' => array(

                'condition' => 'project_id=:projectId',

                'params' => array(

                    'projectId' => $this->_project->id),

            ),

        ));

    $this->render(

        'index', array(

            'dataProvider' => $dataProvider,

        ));

}

```

这里，我们先前已经做过了，我们只是简单地添加一个条件到模型数据提供者的创建过程中，只取回与项目（**Project**）相关联的问题（**Issue**）。这将会限制问题（**Issue**）的列表中只出现该项目（**Project**）下的问题（**Issue**）。

我们对于管理列表页面，也需要做相同的修改。然而，视图文件 `/protected/views/issue/admin.php` 中使用了模型类 `Issue::search()` 方法来提供问题（**Issue**）的列表。所以，我们实际上需要修改两处地方在这个列表中强制项目（**Project**）上下文。

首先，我们需要修改 `IssueController::actionAdmin()` 方法来设定发送给视图的模型实例的 `project_id` 属性。下列高亮代码是修改的地方：

PHP 代码：

```
public function actionAdmin() {

    $model = new Issue('search');

    if (isset($_GET['Issue']))

        $model->attributes = $_GET['Issue'];

    $model->project_id = $this->_project->id; //&lt; --这个

    $this->render('admin', array(

        'model' => $model,

    ));

}
```

然后，我们需要将我们的标准添加到 `Issue::search()` 模型类方法。下列高亮代码标识出了我们需要进行的修改：

PHP 代码：

```
public function search() {

    // Warning: Please modify the following code to remove attributes that

    // should not be searched.

    $criteria = new CDbCriteria;
```

```

$criteria->compare('id', $this->id);

$criteria->compare('name', $this->name, true);

$criteria->compare('description', $this->description, true);

$criteria->compare('type_id', $this->type_id);

$criteria->compare('status_id', $this->status_id);

$criteria->compare('owner_id', $this->owner_id);

$criteria->compare('requester_id', $this->requester_id);

$criteria->compare('create_time', $this->create_time, true);

$criteria->compare('create_user_id', $this->create_user_id);

$criteria->compare('update_time', $this->update_time, true);

$criteria->compare('update_user_id', $this->update_user_id);

$criteria->condition = 'project_id=:projectID'; //&lt;t;--这行

$criteria->params = array(

    ':projectID' => $this->project_id

);

return new CActiveDataProvider(get_class($this), array(

    'criteria' => $criteria,

));

}

```

有了这些修改，管理页面的问题（**Issue**）列表现在被限制到了特定项目（**Project**）的范围内。

在`/protected/views/issues/`目录中的视图文件里，有多处链接需要包含 `pid` 请求字符串，以使得他们能够正常工作。我们将它们作为练习留给读者，请按照上面三个例子中的方法做适当的修改。随着我们应用的开发进程，我们假设所有的创建新问题（**Issue**）或者显示问题（**Issue**）列表的链接已经被正确处理并包含了 `pid` 请求字符串参数。

小结

我们在这个迭代中，讨论了相当多的主题。基于议题，项目（**Project**）和用户在我们应用中的关系，我们议题管理功能的实现比上个迭代中，项目（**Project**）实体管理的实现要复杂很多。幸运的是，**Yii** 在许多时候，都可以将我们从编写代码来满足这些复杂性的痛苦中解放出来。

具体来说，我们讨论了如下内容：

- § 使用 **Gii** 代码生成工具创建活动记录模型，并且初始实现议题实体之上的 **CRUD** 操作
- § 设计和创建带有显式关系的数据库表
- § 使用带有关系的活动记录
- § 添加下拉菜单型的表单元素
- § 控制器过滤器

我们已经让我们的基础应用进步了许多，并且做完这一切，我们并没有写很多的代码。**Yii** 框架已经做了大多数繁重的工作。现在我们已经有一个可以工作的应用，我们可以用其管理项目（**Project**），并在项目（**Project**）中管理议题。这是我们的应用想要实现的核心功能。我们应该为取得的这些成绩感到自豪。

然而，在这个应用能够被用于生产环境之前，我们还有很长的路要走。缺失的最主要的一块东西，就是用户管理的功能。这将在接下来的两个迭代中涉及到。

在前面很短的时间内，我们完成了大量开发工作。**Trackstar** 应用程序基本功能的基础已经奠定。目前为止，我们已经有能力去管理该项目和其中存在的问题，而这个能力正是 **Trackstar** 应用程序需要实现的首要目标。当然，还有许多事在向我们招手。

请回顾一下第三章，当时我们介绍这一应用程序的时候，我们将它描述为基于角色的应用程序，它允许建立用户帐号，一旦用户获得授权并且通过认证就可以使用一些功能。为了让这个应用程序比单用户系统更有用，我们需要添加基于项目的用户管理能力。下述 **2** 次迭代会完成这一功能。

迭代计划

最初我们使用 **yiic** 命令行工具建立我们的 **Trackstar** 应用程序时，我们注意到(**Yii**)系统已经自动为我们创建了基础登录功能。当前登录页面只能使用 **2** 对帐号/密码进行登录(**demo/demo** 和 **admin/admin**)。你或许还记得，我们必须在登录的情况下，才可以对项目和相关问题进行 **CRUD** 操作。

这些用于认证的基础脚手架代码，确实为我们提供了一个好的开端，但是我们需要进行一定的修改，使之支持更多的用户。同时我们需要向应用程序添加用户 **CRUD** 功能，用来对多用户进行管理。这次迭代聚焦在使用 **User** 表扩展认证模型，和适当添加基本用户数据管理所需功能。

为了达到上述目标，我们需要确定所有在本次迭代将要实现的细节。以下列表中列出了这些细节：

- § 建立一个用于存放所有我们需要功能的 **Controller(控制器)**类：
 - 建立新用户
 - 从数据库获取一列已存在用户信息

- 更新/编辑已存在用户信息
- 删除已存在用户
- § 建立 **View File**(视图文件)和呈现层逻辑, 用来:
 - 显示用于建立新项目的表单
 - 用列表显示所有已存在项目
 - 显示授予某一用户权限, 去编辑一个已存在项目的表单
 - 为项目列表添加一个删除按钮, 实现删除项目
- § 调整新建用户表单, 使之可以被用于外部用户自助注册
- § 修改认证流程, 使用数据库来认证用户登录信息

运行测试套件

在我们添加新功能之前运行一下测试套件绝对是一个好主意。测试套件包括之前每一次迭代, 包括我们为应用添加的代码, 也包括我们所添加过的测试套件。随着测试套件的增多, 我们的应用程序向我们反馈其健壮程度的能力也在不断增长。在我们进行改变前, 请确保一切工作如常。从测试文件包

(`/protected/tests/`)运行一次单元测试:

SHELL 代码或屏幕回显:

```
% phpunit unit/

PHPUnit 3.4.12 by Sebastian Bergmann.

.....

Time: 0 seconds OK (10 tests, 26 assertions)
```

一切都那么美好, 让我们尽情的投入这次迭代。

创建用户 CRUD

由于我们正在建立一个基于用户的 **Web** 应用, 我们必须有添加和管理用户的方式。在第 6 章我们添加了表 `tbl_user` 到我们的数据库。你可能记得, 我们将它作为一个练习(建立与之相关的 **AR** 模型类)留给读者。如果你还没有建立这个类, 你需要现在建立它。

使用 **Gii** 代码创建工具, 新建一个模型类的简要提示。通过 <http://localhost/trackstar/index.php?r=gii> 打开 **Gii** 工具并点击 **Model Generator** 连接, 设置表前缀为 `tbl_`。在 **Table Name** 输入框填入 `tbl_user`, **Model Class** 输入框会自动填入 `User`。

当表单填写完成, 点击 **Preview** 按钮, 可以得到一个可以弹出将要生成代码情况的连接。然后点击 **Generate** 按钮来确定在 `/protected/models` 下生成一个新 `User.php` 模型类文件。

伴随 **User AR** 类的完成和基于以前从 **Gii** 操作中获得的经验，建立 **CRUD** 脚手架就是小菜一碟。以下作为一个简单的回顾：

1

1. 打开 <http://localhost/trackstar/index.php?r=gii>
2. 在可选生成器列表中点击 **Crud Generator** 连接
3. 在 **Model Class** 框输入 **User**, **Controller ID** 框将会自动写入 **user**.

点击 **Preview** 按钮后你可以预览到将要生成的代码，确认后，点击 **Generate** 按钮，所有相关 **CRUD** 文件都会被自动生成在预定好的位置。

当上述被完成后，我们可以在 <http://localhost/trackstar/index.php?r=user> 浏览我们的用户列表。在上一次迭代中，我们手动添加了一些用户进入系统，所以接下来我们可以控制项目，项目中的问题和用户之间的关系了。所以在上述地址我们可以看到一些用户的显示。

下面的截图向我们展示了用户列表页面的样子：

我们也可以通过 <http://localhost/trackstar/index.php?r=user/create> 来访问新建用户表单。如果目前你未登录，你将被引导至登录页面，登录后才可显示新建表单。使用 **demo/demo** 或 **admin/admin** 帐号密码登录后来访问这个表单。

2

从第一次在 **Project** 上使用 **CRUD** 操作，到后来的 **Issue**，我们对于这些功能如何在 **Gii** 代码生成器下执行已经非常熟悉了。那些输入框提供的新建和更新操作是一个不错的开始，不过通常需要进行一些特殊的修改来达到预期的项目需求。在新建用户表单时，这个原则一样适用。每一个 **tbl_user** 表中定义的列，都有一个输入框与之对应，我们并不希望把它们全部暴露给用户。**last_login_time**, **creation_time**, **create_user_id**, **update_time**, **update_user_id** 这些项应该被设置成在表单被提交后由程序自动附值。

审查之前的表更新公共字段

回顾第五，第六章，在我们介绍基于 **Project** 和 **Issue** 的 **CRUD** 功能时，我们也注意到我们的表单提供的输入项远超过了应该暴露给用户的。当我们完成设置时，我们所有的数据库表都拥有相同的新建、更新时间和用户列，每一个自动生成的表单都有相同程度的暴露。在第五章处理创建 **Project** 时，我们完全忽略了这些输入框。同样的，在第六章建立一个新 **Issue** 时，我们只是从表单里移除了这些输入框，当一个新行被添加时，我们并没有设置任何逻辑代码来为这些框设置合适的值。

2

让我们花费一点时间来添加这些逻辑代码。因为我们所有的实体表(**tbl_project**, **tbl_issue**, 和 **tbl_user**)都定义了相同的列，我们将添加所需的逻辑代码到一个公共基类，然后每一个独立的 **AR** 类都继承自这个公共基类。

可能你也想到了，在我们开始添加项目代码之前我们先要写一个测试。我们已经完成了位于 **tests/unit/ProjectTest.php** 的 **ProjectTest::testCreate()** 的测试函数，用来测试新建 **Project**。我们将通过修改这个测试方法来测试更新公共列的操作。

首先应当修改移除 **ProjectTest::testCreate()** 方法里的 **setAttributes()** 方法中，在新建 **Project** 时对这些列的直接赋值：

PHP 代码：

```
$newProject->setAttributes(array(

    'name' =>$newProjectName,

    'description' =>'This is a test for new project creation',

    // - remove - 'createTime' => '2009-09-09 00:00:00',

    // - remove - 'createUser' => '1',

    // - remove - 'updateTime' => '2009-09-09 00:00:00',

    // - remove - 'updateUser' => '1',

));
```

现在我们需要添加用户 **ID** 和移除当保存 **AR** 时的 **false** 参数返回，因为我们现在要触发验证。之所以要在这里触发 **AR** 验证是为了更新这些输入框，我们需要引入验证流。下面的代码显示了全部方法，同时高亮了其中修改部分。

PHP 代码：

```
public function testCreate() {

    //CREATE a new Project $newProject=new Project;

    $newProjectName = 'Test Project Creation';

    $newProject->setAttributes(array('name' =>$newProjectName, //这行

    'description' =>'This is a test for new project creation', //这行

    )); //这行

    //set the application user id to the first user in our users fixture data
```

```

Yii::app()->user->setId($this->users('user1')->id);    //这行

//save the new project, triggering attribute validation

$this->assertTrue($newProject->save());    //这行

//READ back the newly created Project to ensure the creation worked

$retrievedProject = Project::model()->findByPk($newProject->id);

$this->assertTrue($retrievedProject instanceof Project);

$this->assertEquals($newProjectName, $retrievedProject->name);

//ensure the user associated with creating the new project is the same as the applicaiton user
we set

//when saving the project

$this->assertEquals(Yii::app()->user->id, $retrievedProject->create_user_id);    //这行
}

```

新添加的断言是用来测试 **Project** 表的 **create_user_id** 列是否使用了当前用户 **ID** 做更新时的属性。这已经足够证明我们的做法是可行的。如果你以命令行方式运行该命令，你将看到我们所预期的失败提示。失败的原因是我们还未为该字段设置相关逻辑代码。

现在是让这个测试通过的时候了。我们着手新建一个用来存放更新相关公共字段逻辑代码的类。这个新类将成为一个可被我们应用中所有 **AR** 类继承的基类。新建这样一个基类而不是直接将相关逻辑代码添加到 **Project** 模型类的原因是：这部分逻辑代码被 **Issue** 和 **User** 模型类所共同需要。比起复制相同代码到每一个类中，上面的做法将允许我们在一个地方为每一个 **AR** 模型类同时设置那些字段(属性)。我们也将定义这个类为抽象类，这样使得他们不可以被直接实例化。

我们需要在 **protected/models/** 手工新建一个文件 **TrackStarActiveRecord.php**,并按如下形式添加代码：

PHP 代码：

```

<?php

abstract class TrackStarActiveRecord extends CActiveRecord{

    /**

```



```

        * Prepares create_time, create_user_id, update_time and
        * update_user_id attributes before performing validation.

        */

protected function beforeValidate() {

    if ($this->isNewRecord) {

        // set the create date, last updated date

        // and the user doing the creating

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

        $this->create_user_id = $this->update_user_id = Yii::app()->user->id;

    } else {

        // not a new record, so just set the last updated time

        // and last updated user id

        $this->update_time = new CDbExpression('NOW()');

        $this->update_user_id = Yii::app()->user->id;

    }

    return parent::beforeValidate();

}

}

```

在这里我们重载了 **CActiveRecord::beforeValidate()** 方法。这是 **CActiveRecord** 提供的众多可以定制其工作流的事件之一。作为一个小提示，如果在 **AR** 类 **save()** 被调用时不提供一个 **false** 参数，验证将被触发。这一过程将执行所有在 **AR** 类中 **rules()** 里定义的验证细节。一共有 2 个方法允许我们将适当的逻辑置放在验证开始之前和验证结束之后，这个 2 方法为：**beforeValidate()** 和 **afterValidate()**。在这个部分，我们决定在验证执行前对我们审查表后的公共字段进行直接设置。

你可能注意到在之前的代码中使用了 `CDbExpression` 来设置新建和更新时间的值为 **Unix** 时间戳。自 **1.0.2** 版开始，一个属性的值可以设置为 `CDbExpression` 类型，在所属记录被保存前。在保存过程中所需的值将由运行数据库表达式获得。

上述代码中的 **NOW** 是 **MySQL** 专有函数，如果你使用非 **MySQL** 数据库，它可能不会起作用。你可以使用不同的方法来设置这个值。例如，使用 **PHP** 时间函数，并且通过格式化使其符合列时间类型：

```
$this->createTime=$this->updateTime=date('Y-m-d H:i:s', time());
```

无论在我们处理新建(如 **insert**)还是更新(如 **update**)的时候，都设置相关字段的值。同时我们也保证父类中的方法通过 `parent::beforeValidate()` 被执行，使得它可以完成所有工作。

在完成上述功能后，我们需要对 3 个已存在 **AR** 类做修改(**Project.php**, **User.php**, **Issue.php**)，使之继承自新建的抽象类，而不是直接继承自 **ActiveRecord**。例如，如下操作：

PHP 代码：

```
classProjectextendsCActiveRecord
{
```

我们将其改成如下格式：

PHP 代码：

```
classProjectextendsTrackStarActiveRecord
{
```

对其他模型类进行相同修改。修改 **Project** 类后，返回确认测试通过。

都完成后，我们可以针对新建 **Projects**, **Issues**, 和 **Users** 表单移除一些字段的输入框(**Issues** 表单已经在之前迭代中移除)。各自的 **HTML** 结构代码可以在 `protected/views/project/_form.php`, `protected/views/issue/_form.php`, 和 `protected/views/user/_form.php` 中找到。从每一个 **HTML** 代码中移除以下代码：

PHP 代码：

```
<div class="row">

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

<?php echo $form->textField($model, 'create_time'); ?>

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

```

</div>

<div class="row">

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

<?php echo $form->textField($model, 'create_user_id'); ?>

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

</div>

<div class="row">

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

<?php echo $form->textField($model, 'update_time'); ?>

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

</div>

<div class="row">

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

<?php echo $form->textField($model, 'update_user_id'); ?>

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

</div>

```

并且在创建的用户表单中(`protected/views/user/_form.php`), 我们也可以同时移除 `last_login_time` 部分:

PHP 代码:

```

<div class="row">

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

<?php echo $form->textField($model, 'last_login_time'); ?>

```

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

</div>
```

完成上述移除后，我们需要移除 **rules** 方法中的对应验证规则。这些验证规则可以确保用户输入的格式是正确的。因为这些字段无需用户输入，所以我们可以移除对应规则。

在 **User::rules()** 方法中，移除以下 2 项规则：

PHP 代码：

```
array('create_user_id', 'update_user_id', 'numerical', 'integerOnly' => true),

array('last_login_time', 'create_time', 'update_time', 'safe'),
```

在 **Project** 和 **Issue** 中也定义了相似的规则，但非完全一致。确保仅保留需要用户填写的项被施加验证规则。

移除 **last_login_time** 属性是有原因的。允许用户输入这个属性不太合适。这个属性应该仅在成功登录时被更新。当我们打开视图文件并移除相关字段的时候，这一项也被决定移除掉。同时，与之相关的逻辑代码添加工作，将放在我们完成一些其他修改和主题之后。

虽然我们一直在修改 **User** 类的验证规则，但还有另外一个需要修改的地方。我们希望确保每一个用户的 **email** 都和他的 **username** 一样唯一，这一验证应该在表单提交时被执行。针对 **rules()** 进行如下修改

PHP 代码：

```
array('email', 'username', 'unique'),
```

完成后的 **User::rules** 如下：

PHP 代码：

```
public function rules()

{

    //NOTE: you should only define rules for those attributes that
    // will receive user inputs.

    return array(

        array('email', 'required'),

        array('email', 'username', 'password', 'length', 'max' => 256),
```

```

array('email', 'username', 'unique'),

// The following rule is used by search().

// Please remove those attributes that should not be searched.

array('id', 'email', 'username', 'password', 'last_login_time',

create_time, create_user_id, update_time, update_user_id', 'safe', 'on'=>'search'),

);

}

```

上述规则中的特殊申明是对 **Yii** 内部验证器 **CUniqueValidator** 的一个引用。这样的验证确保了对应底层数据库表的类属性唯一性。添加这条验证规则后，当我们输入一个已存在的 **username** 或 **email** 时，会出现错误。当我们在第 6 章新建 **tbl_user** 表时，我们插入了 2 个用户，所以你可以基于这个做下尝试。其中一个用户 **email** 为 **test1@notanaddress.com**，使用这个 **email** 地址添加一个新用户。你会看到如下图的错误消息，同时错误位置会被高亮提示。

添加确认密码输入框

我们需要添加一个新输入框来强制用户确认他输入的密码。这是一次针对用户注册表单的标准练习，同时也可以帮助用户避免在输入这部分重要信息时发生错误。幸运地是，**Yii** 的另外一个内部验证器 (**CCompareValidator**)，可以完成你所期望的工作。他的工作是对比 2 个属性值，并且在不相等时返回一个错误消息。

2

为了使用这一内置验证器，我们需要为我们的模型类添加一个新属性。在 **User** 模型 **AR** 类顶部添加如下代码：

PHP 代码：

```
public $password_repeat;
```

我们在期望对比的属性名后附加 **_repeat**，形成了与之对比的新属性名。对比验证器允许你设置 2 个属性名，或者属性名和确定值进行对比。在没有明确指出对比规则的情况下，默认对比属性名相同，后接 **_repeat** 的属性。这也是我们为何如此命名的用意。现在我们可以按照如下方式在 **User::rules()** 添加验证器：

1

PHP 代码：

```
array('password', 'compare'),
```

我们希望标识所有输入框都为必须填写。当前,只有 **email** 属性被标识为必填。所以我们针对 **User::rules()** 进行修改,将 **username** 和 **password** 添加到验证列表中:

PHP 代码:

```
array('email', 'username', 'password', 'required'),
```

由于我们直接在 **User AR** 类中添加了 **\$password_repeat** 属性,并且它与底层数据库表之间没有对应关系,我们需要告诉模型类允许这个属性在 **setAttributes()** 被调用时被设置。我们的做法是将其添加到 **User** 模型类的安全属性列表中。向 **User::rules()** 数组添加下列代码:

PHP 代码:

```
array('password_repeat', 'safe'),
```

简单介绍一下,当表单提交到 **UserController::actionCreate()** 方法时,会使用下列的代码将 **User** 模型类的属性进行批量转换:

PHP 代码:

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

1

在这里,所有 **\$_POST['User']** 数组里的项会和 **\$model** 类安全属性列表中的做匹配,成功匹配的完成赋值。默认的,除了主键外的所有底层数据库表里的字段都被视为安全的。我们新建的 **\$password_repeat** 不存在对应的 **tbl_user** 表中的列,需要将其直接添加到安全属性列表。

2

我们还需要将密码确认框添加到表单,让我们行动起来。

在 **HTML** 表单中添加如下新代码,打开 **protected/views/user/_form.php**,并且在密码输入框下面添加如下代码块:

PHP 代码:

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

```
<?php echo $form->label($model, 'password_repeat'); ?>
```

```
<?php echo $form->passwordField($model, 'password_repeat', array('size' => 60, 'maxlength' => 256));
```

```
?>
```

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

</div>
```

当所有的表单都被修改后，新建用户表单显示如下图：

现在如果你在，**password** 和 **password Repeat** 输入框里填入不同的值，我们将会看到如下的错误信息：

为密码加密

针对整个新建用户流程的最后一个修改项是在保存之前对用户输入的密码字段进行加密。基于安全标准观点出发，在数据被保存之前，最低限度我们可以对密码运行一次单向加密算法。我们将通过重载 **CActiveRecord** 类下的允许自定义 **AC** 工作流程的方法来把相关逻辑添加到 **User.php AR** 类中去。这次我们将重载 **afterValidate()** 方法，并且在保存数据之前，简单的对密码进行 **MD5** 加密。

打开 **User AR** 类，在类文件底部添加如下代码：

PHP 代码：

```
/**
 * perform one-way encryption on the password before we store it in the database
 */

protected function afterValidate() {
    parent::afterValidate();

    $this->password = $this->encrypt($this->password);
}

public function encrypt($value) {
    return md5($value);
}
```

当修改完成后，整个流程为当所有其他属性都通过验证之后，密码会被进行简单的单向 **MD5** 加密。

在新添一条记录时，上述修改不会有任何问题，但是在更新时，它有可能对已加密值进行重复加密。对此我们有多种处理方法，但是为了保持简单，我们会要求用户每次更新他们的用户信息时提供一个用于认证的密码。

至此，我们有足够的能力为我们的项目添加新用户，通过使用 **Gii** 工具里的 **Crud Generator** 命令添加此项功能，同时也添加了读取，更新，和删除用户的功能。做一个简单的练习，添加一些用户，读取所有的用户列表，更新其中的一些用户资料，最后删除部分用户，确保用户功能按照预期目标工作。另外，在执行删除工作时，请使用 **admin** 登录，而不是 **demo**。

使用数据库进行用户认证

正如我们所知道的，在我们使用 **yiic** 新建项目时，建立基础登录表单和用户认证流程。这个认证方式非常简单。只有在输入 **demo/demo** 或 **admin/admin** 时，认证才可以通过，其他情况下都是失败。这显然不是一个长期解决方案，但是是一个很好的基础。我们通过修改已经存在认证流程使其使用已经插入数据的数据表(这些表已经是我们的模型的一部分)来完成认证。但在我们改变默认的实现方式之前，让我们仔细观察一下 **Yii** 如何实现一个认证模型。

1

介绍 Yii 认证模型

Yii 认证框架的核心是一个叫 **user** 的应用组件，一般来说，是 **IWebUser** 接口的对象实现。这个具体的类是默认通过框架类 **CWebUser** 实现的。这个用户组件封装了所有当前用户在整个应用中的认证信息。这个组件的配置在我们使用 **yiic** 工具自动生成整个项目代码的时候被创建。具体配置信息可以在 **protected/config/main.php** 文件查看，基于 **components** 数组：

2

PHP 代码：

```
'user' => array(  
  
    // enable cookie-based authentication '  
  
    'allowAutoLogin' => true,  
  
),
```

因为它被配置成了一个使用关键词 **'user'** 的应用组件，所以我们可以项目中的任何地方通过 **Yii::app()->user** 进行访问。

2

我们也发现了类属性 **allowAutoLogin** 也在这里被设置好了。这个属性默认值为 **false**，但是当设置为 **true** 时，可以将用户的信息持久的储存在浏览器的 **cookies** 里。储存的信息将在后面的访问中自动用于认证。

这样我们就可以在登录表单显示一个 **Remember Me** 复选框，当用户选中的情况下，再次访问网站都会自动登录进行。

Yii 认证框架定义了一个独立的实体来储存当前认证逻辑。这是一个认证类，而且一般形式上，这个类实现了 **IUserIdentity** 接口。这个类扮演的主要角色是封装了认证逻辑，使得容易进行不同的应用。基于项目需求，我们需要将用户数据的用户名和密码和数据库中储存的值进行匹配，或者允许用户使用他们的 **OpenID credentials** 进行登录，或者整合入一个现有的 **LDAP** 方式。逻辑分离是指将认证方式从登录流程中分离出来，从而允许我们轻松的在不同应用中转换。认证类提供这样程度的分离。

当最初建立我们的应用程序时，处于 `protected/components/UserIdentity.php` 的认证类被同时创建。它继承自 Yii 框架类 **CUserIdentity**，这个框架类是基于用户名密码认证的基础类。让我们仔细观察一下这个类被生成时的代码情况：

PHP 代码：

```
<?php

/**
 * UserIdentity represents the data needed to identity a user.
 * It contains the authentication method that checks if the provided
 * data can identify the user.
 */

class UserIdentity extends CUserIdentity {

    /**
     * Authenticates a user.
     * The example implementation makes sure if the username and password
     * are both 'demo'.
     * In practical applications, this should be changed to authenticate
     * against some persistent user identity storage (e.g. database).
     * @return boolean whether authentication succeeds.
     */

    public function authenticate() {
```

```

$users = array(

    // username => password

    'demo' => 'demo' ,

    'admin' => 'admin' ,

);

if(!isset($users[$this->username]))

    $this->errorCode = self::ERROR_USERNAME_INVALID;

elseif($users[$this->username] !== $this->password)

    $this->errorCode = self::ERROR_PASSWORD_INVALID;

else

    $this->errorCode = self::ERROR_NONE;

return!$this->errorCode;

}

}

```

大量的工作是通过定义在一个认证类中的 **authenticate()** 方法中实现的。这里就是我们存放身份认证代码的地方。这里的实现方式是使用硬编码的用户名/密码对 (**demo/demo** 和 **admin/admin**)。它会将这些值与类属性（定义在父类 **CUserIdentity** 中的属性）的用户名和密码进行匹配，如果匹配失败，将设置并返回一个适当的错误代码。

为了更好的理解，这一部分是如何适应整个端到端认证流程，让我们从登录表单开始本次认证之旅。如果我们进入登录页：<http://localhost/trackstar/index.php?r=site/login>，我们将看到一个允许输入帐号密码，还有之前讨论过的 **Remember Me** 复选框的登录表单。提交这个表单，触发处于 **SiteController::actionLogin()** 方法中的逻辑代码。下面的时序图描述了登录成功时,类之间的交互过程。

这一流程始于将表单提交值设置为表单类模型 **LoginForm** 的属性。**LoginForm->validate()** 方法此时被调用，并使用 **rules()** 方法中设置的规则对设置的属性值进行验证。**rules()** 定义如下：

PHP 代码：

```

publicfunctionrules(){

returnarray(

// username and password are required

array('username, password', 'required'),

// rememberMe needs to be a boolean

array('rememberMe', 'boolean'),

// password needs to be authenticated

array('password', 'authenticate'),

);

}

```

最后一条规则规定 **password** 属性必须使用自定义方法 **authenticate()** 进行验证，这个方法按照如下代码定义在 **LoginForm** 类中：

PHP 代码：

```

/**
 * Authenticates the password.
 * This is the 'authenticate' validator as declared in rules().
 */
publicfunctionauthenticate($attribute, $params){

$this->_identity=newUserIdentity($this->username, $this->password);

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

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

按时序图的顺序，**LoginForm** 类中的密码验证规则调用同一个类中的 **authenticate()** 方法。这个方法创建了一个被使用的认证类实例，在这个案例中是 **/protected/components/UserIdentity.php**，然后调用自身的 **authenticate()** 方法。此方法 **UserIdentity::authenticate()** 定义如下：

PHP 代码：

```

/**
 * Authenticates the password.
 * This is the 'authenticate' validator as declared in rules().
 */

public function authenticate($attribute, $params){

    if(!$this->hasErrors()){

        // we only want to authenticate when no input errors

        $identity = new UserIdentity($this->username, $this->password);

        $identity->authenticate();

        switch($identity->errorCode){

            case UserIdentity::ERROR_NONE: $duration = $this->rememberMe ? 3600 * 24 * 30 : 0; // 30 days

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

            break;

            case UserIdentity::ERROR_USERNAME_INVALID: $this->addError('username', 'Username is incorrect. ');

            break;

            default: // UserIdentity::ERROR_PASSWORD_INVALID

            $this->addError('password', 'Password is incorrect. ');

            break;

        }

    }

}

```

这里实现了使用用户名和密码进行认证。在这次实现中，在输入用户名密码对为 **demo/demo** 或 **admin/admin** 的情况下，这个方法将返回 **true**。等我们登录成功，认证执行完毕，在应用组件中的 **login()** 方法被调用。

正如所说，web 应用程序默认被设置为使用 Yii 框架的 CWebuser 用户组件。它的 login() 方法被放在一个认证类里，并有一个可选参数控制浏览器 cookie 生存时间。在上面的代码中，我们发现它将 30 天设置为勾选 Remember Me 后的生存时间。设置为 0，则取消 cookie 功能。

在用户会话期间，login 方法将认证类中包含的信息持久的储存。默认的储存在 PHP 的 session 中。

当这些都完成之后，LoginForm 中的 validate() 方法将返回 true（成功登录的情况下）。控制器将重定向 URL 到 Yii::app()->user->returnUrl 的值。你可以定义到一个指定页面，如果你希望用户重定向回他们以前的页面，也就是说，任何他们发起登录的页面。默认值为项目的实体 URL。

修改认证的实现方法

现在了解了整个认证过程后，我们现在可以轻松的找出针对表 tbl_user 进行如何的修改，才能验证登录表单提供的用户名和密码。我们可以简单的修改 user 认证类的 authenticate() 方法，来验证存在的用户名和密码和提供的是否匹配。从现在开始，除了认证方法我们的 UserIdentity.php 类中什么都没有，让我们使用如下代码覆盖整个文件的内容：

PHP 代码：

```
<?php

/**
 * UserIdentity represents the data needed to identity a user.
 * It contains the authentication method that checks if the provided
 * data can identify the user.
 */

class UserIdentity extends CUserIdentity {

    private $_id;

    /**
     * Authenticates a user using the User data model.
     *
     * @return boolean whether authentication succeeds.
     */
}
```

```
publicfunctionauthenticate(){

    $user = User::model()->findByAttributes(array('username' =>$this->username));

    if($user === null){

        $this->errorCode = self::ERROR_USERNAME_INVALID;

    }else{

        if($user->password !== $user->encrypt($this->password)){

            $this->errorCode = self::ERROR_PASSWORD_INVALID;

        }else{

            $this->_id = $user->id;

            if(null === $user->last_login_time){

                $lastLogin = time();

            }else{

                $lastLogin = strtotime($user->last_login_time);

            }

            $this->setState('lastLoginTime', $lastLogin);

            $this->errorCode = self::ERROR_NONE;

        }

    }

    return!$this->errorCode;

}

publicfunctiongetId(){

    return$this->_id;

}
```



新代码中的一些地方是需要被指出的。首先，它通过实例化 **User** 模型 **AR** 类来试图取出 **tbl_user** 中的一行，其中的 **username** 和 **UserIdentity** 类的属性相同(还记得那些来自登录框的值嘛？)，因为我们要求了新建用户时用户名的唯一性，所以配置的行最大值为 **1**。如果没有找到匹配行，一个用户名错误的消息将会被显示。如果找到了匹配行，将会对密码进行匹配。因为我们对存入数据库的密码进行了单向加密，所以必须使用我们之前添加至 **User** 类的 **encrypt()** 方法进行加密。如果密码不匹配，将显示一个密码错误的消息。

如果认证成功，在方法提供返回之前，还有一些事发生了。首先，我们为 **UserIdentity** 类添加了新属性用户 **ID**。父类的默认实现是返回 **ID** 对应的用户名。因为我们正在使用数据库，并且有多个独立的主键作为用户标识，我们想确保这个用户 **ID**，是在任何地方需要调用用户 **ID** 时提供的那个。亦即，当 **Yii::app()->user->id** 这行代码被执行，数据库中唯一的用户 **ID** 被返回，而不是用户名。

扩展应用中的用户属性

下面要说的事是，用户认证类的一个属性是取自数据库的最后登录时间。**user** 应用组件 **CWebUser**，通过定义在标识类中的 **ID** 和 **username** 获取它的用户属性，这样形如 **name=>value** 设置在了叫 **identity states** 的数组中。这些是额外的用户属性值，需要被持久保存在用户的会话里。举个例子，我们将属性 **lastLoginTime** 指向数据库中 **last_login_time** 字段的值。如此，在应用程序的任何地方，这个属性可以通过如下获取：

PHP 代码：

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

因为插入新用户时，最后登录时间的值为空，正因为有一个空属性，当用户第一次登录时可以填入适当的时间作为值。我们也尽量将时间格式为更易读。

采用不同的方法储存对应 **ID** 的最后登录时间是因为：**id** 为 **CUserIdentity** 类的明确定义属性。所以与姓名和 **ID** 不同，所有其他需要被持久化保持在会话中的用户属性可以采用相同的方法定义。

当基于 **cookie** 认证被开启（通过设置 **CWebUser::allowAutoLogin** 值为 **true**），用户认证状态将被储存在 **cookie** 中。因此，你不应该使用我们储存用户最后登录时间的方法储存敏感信息（例如，密码）。

当上述改变都完成后，你应该提供正确的储存在表 **tbl_user** 中的用户名和密码对来进行登录。使用之前的 **demo/demo** 和 **admin/admin** 自然是不会工作的。你可以试一下。你可以使用你在本章前面建立的任何帐号进行登录。如果你一直跟随我们的教程，那么我们的数据库相同，以下的帐号密码应该可以工作：

Username: Test_User_One

Password: test1

现在我们已经修改了登录流程，使之使用数据库验证，我们没有进行删除 **Project**, **Issue**, 或用户实体的权限。这是因为，授权检查要求管理员用户才可以进行该操作。目前，我们数据库中也没有管理员用户。不过别担心，这个问题将在下一次迭代中得到解决，很快我们就可以使用这一功能了。

更新用户最后登录时间

正如我们在这一章节前期提到的，我们从新建用户表单中移除了最后登录时间的输入框，但是相关的逻辑并未添加。因为我们使用表 **tbl_user** 来跟踪用户最后登录时间，所以我们需要根据最后一次成功登录信息更新这一字段。因为真正的登录发生在 **LoginForm::login()** 中，让我们在这个方法中设置这个值。为 **LoginForm::login()** 方法添加如下高亮代码：

PHP 代码：

```
/**
 * Logs in the user using the given username and password in the model.
 *
 * @return boolean whether login is successful
 */
public function login() {
    if ($this->_identity === null) {
        $this->_identity = new UserIdentity($this->username, $this->password);

        $this->_identity->authenticate();
    }

    if ($this->_identity->errorCode === UserIdentity::ERROR_NONE) {
        $duration = $this->rememberMe ? 3600 * 24 * 30 : 0; // 30 days

        Yii::app()->user->login($this->_identity, $duration);

        User::model()->updateByPk($this->_identity->id, array('last_login_time' => new
        CDbExpression('NOW()'))); //这行

        return true;
    }

    else

    return false;
}
```



```
}
```

这里我们使用了 `updateByPk()` 方法作为一种高效，处理用户记录更新的方式，并且简单到只需要指出主键和需要更新的 `name=>value` 键值对。

在首页上显示最后登录时间

现在，我们已经将最后登录时间储存进了数据库，同时也将其持久的储存在了会话中，让我们接下来将这一时间显示在用户成功登录后看到的欢迎页面上。因所有的一切都在按预期进行，我们会有一个不错的感觉。

打开用来显示首页默认视图文件：`protected/views/site/index.php`。添加位于欢迎语句下面的高亮代码到文件中：

PHP 代码：

```
<h1>Welcome to <i><?php echo CHtml::encode(Yii::app()->name); ?></i></h1>

<?php if(!Yii::app()->user->isGuest): ?>

<p>

You last logged in on <?php echo date('l, F d, Y, g:i a', Yii::app()->user->lastLoginTime); ?>.

</p>

<?php endif; ?>
```

2

正如我们所见的，让我们删除其他在我们添加的代码之下的自动生成的帮助代码。一旦你完成了修改，并且再次登录，你将看到和下面截图一样的画面(欢迎语句下面有你最后一次成功登录的时间)

小结

这次迭代是我们针对用户管理的 2 次迭代（认证和授权）之一。我们通过一步一步修改新建用户流程，为项目用户建立了管理 CRUD 操作的能力。我们为全部 AR 类建立了一个基类，因此我们可以轻松的管理在所有表中审查出来公共列。我们也通过更新代码实现了将最后登录时间存进数据库的管理。通过这次迭代，我们了解了 CActiveRecord 验证工作流程以及在验证之前或之后进行其它处理。

接下来我们致力于理解 **Yii** 的认证模型，使之符合我们的项目预期：使用数据库中储存的值与用户数据的帐号密码进行匹配。

至此，我们完成了认证部分，我们可以进入 **Yii** 认证和授权框架的第二部分——授权的学习中了。这部分的学习将在下次迭代中介绍。

像我们前面制作的 **TrackStar** 应用程序这样基于用户的 **web** 应用程序中，通常需要针对谁对某一功能提出访问请求进行访问控制。我们所说的用户访问控制是指在一个较高的层次上，当访问请求被提出时一些需要被思考的问题，例如：

- § 请求的提出者是谁？
- § 提出请求的用户是否有足够的权限访问该功能？

上述问题的答案可以帮助应用程序做出适当的响应。

在上次迭代中完成的工作使得应用程序有能力提出第一个问题。我们的基础用户管理应用扩展了应用程序的用户认证流程，使之可以使用数据库内的资料。应用程序现在允许用户建立自己的认证信息，并且在用户登录时使用储存在数据库内的信息进行匹配。在用户成功登录后，应用程序就知道接下来的一系列请求是谁提出的。

本次迭代的核心任务是帮助应用程序回答第二个问题。一旦用户提供了足够的认证信息后，应用程序需要找到一种合适的方法去判断用户是否有足够的权限去执行要求的操作。我们将利用 **Yii** 的用户访问控制中的一些特性来扩展我们的基本授权模型。**Yii** 即提供了一个简单的访问控制过滤器，也提供了一个更先进的 **RBAC**（基于角色的访问控制）体系，来帮助我们完成授权需求。我们将在 **TrackStar** 应用程序中用户访问需求的实现中仔细观察这 2 点。

迭代计划

当我们在第 3 章第一次介绍我们的 **TrackStar** 应用程序时，我们曾提及，这个应用程序拥有 2 个高级别用户类型：匿名用户和认证用户。这是一个小小的区别基于成功登录的用户和未登录的用户。我们也介绍过用户在同一个 **Project** 中扮演不同角色的想法。针对一个 **Project** 我们建立了如下的三类角色模型：

- § **Project Owner**（被赋予所有权限访问此 **Project** 的管理功能）
- § **Project Member**（被赋予一定权限访问此 **Project** 的特性和功能）
- § **Project Reader**（只有读取 **Project** 相关内容的权限，没有任何修改级权限）

这次迭代的重点是实施方法来管理这个应用程序中用户的访问控制权限。我们需要一个方法来建立我们的角色和权限，并且分发给用户，同时制定我们期望对每个角色施加的访问控制规则。

为了这个目标，我们需要标明所有我们将在本次迭代中完成的项目细节。下面的列表是这样一个项目列表：

- § 制定一个策略来强制用户在获得访问任何 **Project** 或 **Issue** 相关功能前，必须登录
- § 建立用户角色并且使之与一个特殊的功能权限对应

- § 制定一个为用户分配角色的机制（包含角色相关的权限）
 - § 确保我们的角色权限结构针对每一个 **Project** 独立存在（也就是说，允许用户在不同的项目拥有不同的权限）
 - § 制定一个让用户和项目，同时也和项目中的角色相关联的机制
 - § 实施适当的认证访问检测，使得应用程序可以针对基于不同用户权限进行允许或拒绝访问操作
- 幸运的是，**Yii** 内部有大量内部功能可以帮助我们完成这一需求。所以，让我们大干一场吧。

运行已存在的测试套件

如常，是时候运动一下了，我们需要运行一次所有已存在的单元测试，来确保测试可以通过：

SHELL 代码或屏幕回显：

```
% cd WebRoot/protected/tests/

% phpunit unit/

PHPUnit 3.4.12 by Sebastian Bergmann.

.....

Time: 3 seconds

OK (10 tests, 27 assertions)
```

一切都没有问题，所以我们可以开始本次旅程了。

访问控制过滤器

我们曾经在第三次迭代的时候介绍过过滤器，当时我们使用它来鉴别项目上下文关系，当处理 **Issue** 相关 **CRUD** 操作时。**Yii** 框架提供的过滤器被叫做 **accessControl**(访问控制)。这个过滤器可以被直接使用在控制器类中，提供一种授权方案来验证用户是否可以使用一个特定的控制器下的行为。实际上，细心的读者应该能想起，我们在第六章使用 **filterProjectContext** 过滤器时，我们曾发现过，访问控制过滤器当时已经存在于 **IssueController** 和 **ProjectController** 类的过滤器列表中，像下面的样子：

PHP 代码：

```
/**
 * @return array action filters
 */

public function filters(){
```

```
returnarray(

    'accessControl',

    // perform access control for CRUD operations

);

}
```

上面的代码包含在由 **Gii** 代码生成器在生成 **Issue** 和 **Project AR** 类的脚手架 **CRUD** 操作时，自动生成的代码里的。

默认的实现方法是设置为允许任何人观看一个已经存在 **Issue** 和 **Project** 项目的列表。然后，它只允许认证用户进行新建和更新操作，进一步限制帐号为 **admin** 的用户享有删除行为。你也许还记得当我们第一次对 **Project** 实施 **CRUD** 操作时，我们必须登录才可以执行操作。同样的情形发生在对 **Issues** 和 **Users** 的操作上。这种机械式的授权和访问模式就是由过滤器 **accessControl** 实现的。让我们仔细观察一下在 **ProjectController.php** 文件中的这种实现方式。

有 2 个相关访问控制实现方法在这个文件中，**ProjectController:filters()**和 **ProjectController::accessRules()**。第一个方法的代码如下：

PHP 代码：

```
/**

 * @return array action filters

 */

publicfunctionfilters()

{

returnarray(

    'accessControl', // perform access control for CRUD operations

);

}
```

下面是第二个方法的代码：

PHP 代码：

```
/**
```

**Specifies the access control rules.*

**This method is used by the 'accessControl' filter.*

**@return array access control rules*

**/*

```
public function accessRules()
```

```
{
```

```
return array(  
    array('allow', // allow all users to perform 'index' and 'view' actions
```

```
    'actions' => array('index', 'view'),
```

```
    'users' => array('*'),
```

```
),
```

```
    array('allow', // allow authenticated user to perform 'create' and 'update' actions
```

```
    'actions' => array('create', 'update'),
```

```
    'users' => array('@'),
```

```
),
```

```
    array('allow', // allow admin user to perform 'admin' and 'delete' actions
```

```
    'actions' => array('admin', 'delete'),
```

```
    'users' => array('admin'),
```

```
),
```

```
    array('deny', // deny all users
```

```
    'users' => array('*'),
```

```
),  
);  
}
```

filters()方法对我们来说已经非常熟悉了。我们在这里申明所有将在这个控制器类里使用的过滤器。在上面的代码中，我们只有一个 **accessControl**，引用了一个由 **Yii** 框架提供的过滤器。这个过滤器调用定义了如何驱动相关访问限制规则的 **accessRules()**方法。

在前面提到的 **accessRules()**方法中，有 **4** 条规则被申明。每一条都以数组形式存在。该数组的第一个元素是 **allow**(通过)或 **deny**(拒绝)。分别标识了获得或拒绝相关访问。剩下的部分由 **name=>value** 键值对组成，申明了规则里剩余的参数。

让我们观察一下前面定义的第一条规则：

PHP 代码：

```
array('allow', // allow all users to perform 'index' and 'view' actions  
  
'actions'=>array('index','view'),  
  
'users'=>array('*'),  
  
),
```

这条规则允许任何用户执行控制器的 **index** 和 **view actions**(行为)。星号`*`特殊字符泛指所有用户（包括匿名，已认证，和其他类型）。

第二条规则被定义成如下形式：

PHP 代码：

```
array('allow', // allow authenticated user to perform 'create' and 'update' actions  
  
'actions'=>array('create','update'),  
  
'users'=>array('@'),  
  
),
```

这条规则允许认证用户（已登录）执行控制器的 **create** 和 **update actions**。`@`特殊字符泛指所有已认证用户。

下面是第三条规则：

PHP 代码：

```
array('allow', // allow admin user to perform 'admin' and 'delete' actions

'actions'=>array('admin','delete'),

'users'=>array('admin'),

),
```

这里申明一个用户名为 **admin** 的特殊用户，被允许执行控制器的 **actionAdmin()** 和 **actionDelete()** **actions**。

第四条规则如下定义：

PHP 代码：

```
array('deny', // deny all users

'users'=>array('*'),

),
```

它拒绝了所有用户执行所有控制器的所有 **action**。

定义访问规则的时候可以使用一些 **context parameters**(内容参数)。前面提到的规则定义了行为和用户来组成规则的内容，下面是一个完整的参数列表：

- § **Controllers(控制器)**：这条规则指定了一个包含多个控制器 **ID** 的数组，来指明哪些规则需要被应用。
- § **Roles(角色)**：这条规则指定了一个将被规则使用的授权列表(包括角色，操作，权限等)。这些是为 **RBAC** 的一些功能服务的，我们将在下一个部分进行讨论。
- § **Ips(IP 地址)**：这条规则指定了一组可以被施加到规则的客户端 **IP** 地址。
- § **Verbs(提交类型)**：这条规则制定了可以被施加到规则的 **HTTP** 请求类型。
- § **Expression(表达式)**：这个规则指定了一个 **PHP** 表达式，这个表达式的值被用来决定这个规则是否应该被使用。
- § **Actions(行为)**：这个规则指定了需要被规则匹配的对 **action ID**(行为 **ID**)的方法。
- § **Users(用户)**：这个规则指定了应该被施加规则的用户。登录当前项目的用户名作为被匹配项。**3** 个特殊字符可以被使用：
 - *****:任何用户
 - **?**:匿名用户
 - **@**:登录用户/认证用户

访问规则按照其被申明的顺序一条一条的被评估。第一条规则与当前模型进行匹配，来判断授权结果。如果当前规则是一个 **allow** 的，这个行为(**action**)可以被执行；如果是一个 **deny** 规则，这个 **action** 无法被执行；如果没有规则和内容匹配，这个 **action** 仍然可以被执行。这是第四条被定义的原因。如果我们不在

我们的认证列表末端定义一条 **deny** 全部用户全部 **action** 的规则，我们就无法完成预期的访问控制。拿第二个规则来举例，所有通过认证的用户可以执行 **create** 和 **update actions**。但是它并不会拒绝匿名用户的请求。它对匿名用户熟视无睹。这里第四条规则确保所有和上面 3 条不匹配的请求都被拒绝掉。

当这些都完成后，修改我们的应用程序来拒绝匿名用户访问所有的 **Project**，**Issue** 和 **user** 相关的功能，绝对小菜一叠。我们需要做的是将用户数组的特殊字符 '*' 替换为 '@'。这将只允许认证用户访问对应控制器的 **actionIndex()** 和 **actionView()** 方法。所有其他 **actions** 都拒绝认证用户访问。

对我们的控制器进行如下修改。打开下面 3 个文件：**ProjectController.php**，**IssueController.php** 和 **UserController.php**，并且如下修改第一条访问控制规则：

PHP 代码：

```
array('allow', // allow only authenticated users to perform 'index' and 'view' actions

'actions'=>array('index','view'),

'users'=>array('@'),

),
```

完成这些修改后，应用程序会在访问 **Project**，**Issue** 或 **User** 的任何功能前要求登录。我们依然允许匿名用户访问 **SiteController** 类的 **action** 方法，这是因为我们的登录方法存在于这个类中。我们必须允许匿名用户访问登录页。

基于角色的访问控制

现在我们已经使用了一个简单的访问控制过滤器(**accessControl**)来 **broad stroke**(泛泛的)限制授权用户访问，我们需要关注我们应用程序所需的更细节化的访问控制。如前所述，用户将在项目中扮演一定的角色。项目中将出现 **owner**(主管)这样的用户类型，可以被视为项目管理员。他们将被赋予所有的项目管理权限。也会有 **member**(成员)这样的用户类型，他们被赋予一部分项目功能权限，为主管权限的一个子集。最后还有一个 **reader**(读者)用户类型，他们只有读没有任何修改的权限。为了完成这样的基于一个用户角色的访问模型，我们将探讨 **Yii** 的 **RBAC** 功能。

在管理已认证用户的访问许可的计算系统安全方面，**RBAC** 是一条已经制定的标准。简单来说，**RBAC** 标准中定义了一个应用程序中的角色。执行某些操作的权限也被定义，然后与角色相关联。然后用户将被分配到一个角色，并且通过这个角色获得与之相关的权限。对于有需要的读者，有大量关于 **RBAC** 概念和标准方面的文档可以阅读。其中一个优秀资源就是 **Wikipedia**：

http://en.wikipedia.org/wiki/Role-based_access_control。我们将专注与 **Yii** 的 **RBAC** 应用。

Yii 中的 **RBAC** 应用是，简单、优雅、和强大的。**Yii** 中 **RBAC** 的基础是一个叫授权项目的 **idea**(想法)。授权项目就是一系列在应用程序中允许去做的事。这些允许可以被归类为 **roles**(角色)，**tasks**(任务)，或者 **operations**(操作)，因此形成权限层级。角色可有任务(或其他角色)组成，任务可以由操作(或其他任务组成)，并且操作是最低权限级别。

例如，在我们的 **TrackStar** 应用程序中，我们需要一个 **owner** 的角色类型。所以我们建立一个角色类型为 **owner** 的授权项目。这个角色将由类似 **user management**(用户管理)和 **issue management**(事务管理)的任务组成。这些任务进一步包含所需的原子操作。例如，用户管理任务可以由新建用户，编辑用户和删除用户等操作组成。权限等级是允许被继承的，来看个例子，如果一个用户被赋予 **owner** 角色，他就继承了新建，编辑，删除的操作权限。

一般来说，你赋予一个用户一定数量的角色，这用户就继承了属于这些角色的权限。这也是 **Yii** 中 **RBAC** 的真谛。当然，在这个例子中，我们可以将用户和任何授权项目想连接，而不是一类角色。这将允许我们灵活的将权限赋予任何级别的用户。如果我们只想赋予删除用户操作给一个特殊用户，而不是全部 **owner**(主管)权限，我们可以将用户简单的与原子操作相连，这使得 **Yii** 中的 **RBAC** 相当灵活。

配置认证管理器

在我们建立授权等级，授予用户角色和访问权限检测之前，我们需要配置授权管理应用组件，**authManager**。这个组件用来存储权限信息，和管理权限与所提供的检测用户是否可以执行相关操作的方法之间的关系。**Yii** 提供了 2 类授权管理器：**CPhpAuthManager** 和 **CDbAuthManager**。**CPhpAuthManager** 使用 **PHP** 脚本文件储存授权信息。**CDbAuthManager**，正如你所猜的，使用数据库储存授权信息。**authManager** 被配置为应用程序组件。只需要指定使用了 2 类的中哪一类，并且设置相关初始化类参数，就可完成授权管理器配置。

1

因为我们已经在 **TrackStar** 应用程序中使用了数据库,这使得,选择 **CDbAuthManager** 应用是更合理的。配置方法：打开位于 **protected/config/main.php** 的主配置文件，将下列代码添加至应用组件数组：

PHP 代码：

```
'authManager'=>array(  
  
    'class'=>'CDbAuthManager',  
  
    'connectionID'=>'db',  
  
),
```

这里建立了一个名为 **authManager** 的新应用组件，指定 **'class'** 类型为 **CDbAuthManager**，并且设置了类属性 **'connectionID'** 为我们的数据库连接组件。至此，我们可以在整个应用程序的任何位置通过 **Yii::app()->authManager** 访问它。

建立 RBAC 使用的数据库表

如前所述，**CDbAuthManager** 类使用数据库表来存储授权信息。它需要一个特定结构。结构文件位置：**YiiRoot/framework/web/auth/schema.sql**。这一简单而优雅的结构由 3 个表组成，**AuthItem**，**AuthItemChild**，和 **AuthAssignment**。表 **AuthItem** 用来储存授权项目的定义信息，包括是一个角色，

任务或操作。表 **AuthItemChild** 用来储存形成授权项目层次的父子关系。最后，**AuthAssignment** 表是用来存储用户和授权项目之间关系的关系表。基本的数据库定义语句如下：

SQL 代码：

```
createtableAuthItem
(
    namevarchar(64)notnull,
    typeintegernotnull,
    descriptiontext,
    bizruletext,
    datatext,
    primarykey(name)
);

createtableAuthItemChild
(
    parentvarchar(64)notnull,
    childvarchar(64)notnull,
    primarykey(parent, child),
    foreignkey(parent)referencesAuthItem(name)ondeletecascadeonupdatecascade,
    foreignkey(child)referencesAuthItem(name)ondeletecascadeonupdatecascade
);

createtableAuthAssignment
(
    itemnamevarchar(64)notnull,
```

```

userid varchar(64) not null,

bizrule text,

data text,

primarykey(itemname, userid),

foreignkey(itemname) references AuthItem(name) on delete cascade on update cascade

);

```

这个结构是直接来自 Yii 框架文件 `/framework/web/auth/schema.sql` 中获取的，并未遵从我们的表命名规则。这些表名是 `CDbAuthManager` 类默认定义的。当然你可以自定义这些表名。简单起见，我们未做修改。

建立 RBAC 授权体系

将上述表添加到我们的 `_dev` 和 `_test` 数据库之后，我们需要将我们的角色和权限添加进去。我们将使用 `authManager` 提供的 API 来完成此操作。我们不会在此建立任何 RBAC 任务。下图展示了我们希望定义的基本等级关系：

图中的等级关系自上而下展示。所以 **Owner** 享有所有列表中的权限，包括从 **Member** 和 **Reader** 二个角色中继承来的。同样的 **Member** 继承了 **Reader** 的权限。现在我们需要做的是在应用程序中建立这一等级关系。入前面提到的，最佳方法是写代码调用 `authManager` API。举个例子，下面的代码新建了一个角色和一个新操作，然后在角色和权限之间建立关系：

1

PHP 代码：

```

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

$role=$auth->createRole('owner');

$auth->createOperation('createProject','create a new project');

$role->addChild('createProject');

```

在上面的代码中，我们首先创建了一个类 `authManager` 的实例，然后我们使用 `createRole()`，`createOperation`，和 `addChild()` 这些 API 接口方法来创建一个新 **owner** 角色和一个叫做 **createProject** 的新操作。然后我们为 **owner** 角色添加权限。这个创建示例只是我们需要的等级关系的一小部分，所有之前被列出的都应该以相似的方式创建。

为了完成构建我们所需的权限等级，我们要书写一段在命令行执行的简单的 **shell** 命令。这将对对我们用来创建最初应用程序的命令行工具 **yiic** 的一个扩展。

编写控制台应用程序命令

2

在第二章我们建立 **HelloWorld** 程序和第四章构建 **TrackStar** 应用程序骨架时我们介绍过 **yiic** 命令行工具。**yiic** 是 **yii** 中以命令行形式执行任务的控制台应用程序。我们曾使用 **webapp** 命令来新建应用程序，并且在第二章我们还使用过 **yiic shell** 命令来新建一个 **controller**(控制器)类。我们也曾用新的 **Gii** 代码生成工具来新建模型类和 **CRUD** 脚手架代码。这些工作也可以用 **yiic** 来完成。作为一个提示，**yiic shell** 命令允许你通过命令行和 **web** 应用程序交互。你可以从包含应用程序脚本的文件包来执行它。同时，内部的一些特殊代码，提供了自动生成 **controllers**(控制器), **views**(视图) and **data models**(数据模型)的功能。

Yii 中的控制台应用程序可以轻松的通过编写自定义命令来扩充，而这正是我们要做的事。我们将通过编写新命令行工具来扩充 **yiic shell** 命令行工具集，这些新命令行工具允许我们使用一致和可重复的方式来构造我们的 **RBAC** 授权等级。

为控制台程序编写新命令是非常容易的。它是继承自 **CConsoleCommand** 类，以最低标准继承了所需的在命令被调用时执行的 **run()** 方法。类的名字应该与所期望的命令名字相同，并且在后面加上 **Command**。在我们的案例中，命令的名字将简单的称为 **rbac**，所以类的名字为 **RbacCommand**。最后，为了使这个命令可以被 **yiic** 调用，我们需要将类文件保存至：**/protected/commands/shell/** 文件夹。

所以，建立名为 **RbacCommand.php** 的新文件，并且添加如下代码：

PHP 代码：

```
<?php

classRbacCommandextendsCConsoleCommand

{

private$authManager;

publicfunctiongetHelp()

{

return<<<EOD

USAGE

rbac
```

DESCRIPTION

This command generates an initial RBAC authorization hierarchy.

EOD;

}

/**

** Execute the action.*

** @param array command line parameters specific for this command*

**/*

public function run(\$args)

{

//ensure that an authManager is defined as this is mandatory for creating an auth heirarchy

if((\$this->_authManager=Yii::app()->authManager)==null)

{

echo "Error: an authorization manager, named 'authManager' must be configured to use this command.\n";

echo "If you already added 'authManager' component in application configuration,\n";

echo "please quit and re-enter the yiic shell.\n";

return;

}

//provide the oportunity for the use to abort the request

echo "This command will create three roles: Owner, Member, and Reader and the following permissions:\n";

echo "create, read, update and delete user\n"; echo "create, read, update and delete project\n";

echo "create, read, update and delete issue\n"; echo "Would you like to continue? [Yes|No] ";

```
//check the input from the user and continue if they indicated yes to the above question
```

```
if(!strncasecmp(trim(fgets(STDIN)), 'y', 1))
```

```
{
```

```
//first we need to remove all operations, roles, child relationship and assignments
```

```
$this->_authManager->clearAll();
```

```
//create the lowest level operations for users
```

```
$this->_authManager->createOperation("createUser", "create a new user");
```

```
$this->_authManager->createOperation("readUser", "read user profile information");
```

```
$this->_authManager->createOperation("updateUser", "update a users information");
```

```
$this->_authManager->createOperation("deleteUser", "remove a user from a project");
```

```
//create the lowest level operations for projects
```

```
$this->_authManager->createOperation("createProject", "create a new project");
```

```
$this->_authManager->createOperation("readProject", "read project information");
```

```
$this->_authManager->createOperation("updateProject", "update project information");
```

```
$this->_authManager->createOperation("deleteProject", "delete a project");
```

```
//create the lowest level operations for issues
```

```
$this->_authManager->createOperation("createIssue", "create a new issue");
```

```
$this->_authManager->createOperation("readIssue", "read issue information");
```

```
$this->_authManager->createOperation("updateIssue", "update issue information");
```

```
$this->_authManager->createOperation("deleteIssue", "delete an issue from a project");
```

```
//create the reader role and add the appropriate permissions as children to this role
```

```
$role=$this->_authManager->createRole("reader");
```

```
$role->addChild("readUser");
```

```
$role->addChild("readProject");
```

```
$role->addChild("readIssue");
```

```
//create the member role, and add the appropriate permissions, as well as the reader role itself, as children
```

```
$role=$this->_authManager->createRole("member"); $role->addChild("reader");
```

```
$role->addChild("createIssue");
```

```
$role->addChild("updateIssue");
```

```
$role->addChild("deleteIssue");
```

```
//create the owner role, and add the appropriate permissions, as well as both the reader and member roles as children
```

```
$role=$this->_authManager->createRole("owner");
```

```
$role->addChild("reader");
```

```
$role->addChild("member");
```

```
$role->addChild("createUser");
```

```
$role->addChild("updateUser");
```

```
$role->addChild("deleteUser");
```

```
$role->addChild("createProject");
```

```
$role->addChild("updateProject");
```

```
$role->addChild("deleteProject");
```

```
//provide a message indicating success

echo "Authorization hierarchy successfully generated.";

}

}

}
```

1

上述代码中的注释可以很好的解释工作流程。我们添加了一个简单的 `getHelp()` 方法，可以帮助其他人快速上手。该命令也与其他 `yiic` 提供的命令保持一致。所有的一切都发生在 `run()` 方法内。该方法确保应用程序中有合法的 `authManager` 应用组件被定义。然后用户将进行最后的选择来决定是否执行。一旦得到用户的确认，它会清除所有 **RBAC** 表中的数据，然后写入新的权限等级资料。这里提到的等级正是我们前面提到的。

不难发现，即使基于如此简单的等级体系，仍然需要写大量的代码。一般的，人们需要制作一个更简单的用户界面来代替授权管理器 **API** 提供的简单的管理角色，任务和操作的接口。处于我们 **TrackStar** 应用程序的考虑，我们可以简单的设置数据库的表，之后运行一次逻辑代码建立基础逻辑关系，然后祈祷我们无需对此做过多修改。这是快速建立 **RBAC** 授权结构的一个不错的解决方案，但是并非一个长期解决方案。

在真实项目里，你可能需要一个拥有更多交互性的不同的工具来管理你的 **RBAC** 关系。**Yii** 扩展库 (<http://www.yiiframework.com/extensions/>) 为之提供了一些打包好的解决方案。

让我们试一试新命令吧。`cd` 到你的应用程序目录，然后执行 **shell** 命令(这里的 **YiiRoot** 代表你安装 **Yii** 框架的位置)：

1

SHELL 代码或屏幕回显：

```
% YiiRoot/framework/yiic shell

Yii Interactive Tool v1.1 (based on Yii v1.1.2)

Please type 'help' for help. Type 'exit' to quit.

>>
```

1

输入 **help**，将显示所有可用命令列表：

SHELL 代码或屏幕回显：

```
At the prompt, you may enter a PHP statement or one of the following commands:
```



```
- controller
- crud
- form
- help
- model
- module
- rbac

Type 'help <command-name>' for details about a command.
```

1

在列表里可以看到我们的 **rbac** 命令。输入 **help rbac** 显示更多信息：

SHELL 代码或屏幕回显：

```
>> help rbac

USAGE

    rbac

DESCRIPTION

    This command generates an initial RBAC authorization hierarchy.
```

上述显示了我们写在 **getHelp()** 方法中的信息。你可以随意对其进行扩充。

是时候启动命令建立所需的等级关系了。

SHELL 代码或屏幕回显：

```
>> rbac

This command will create three roles: Owner, Member, and Reader and the following premissions:

create, read, update and delete user

create, read, update and delete project
```

```
create, read, update and delete issue

Would you like to continue? [Yes|No] Yes

Authorization hierarchy successfully generated.
```

让我们退出 **shell**:

SHELL 代码或屏幕回显:

```
>> exit
```

假设在提示继续时你输入 **Yes**, 所有的权限等级关系都会被建立。

你可能还记得, 我们为测试设置了一个叫 **trackstar_test** 的独立的数据库。因为我们需要在测试数据库里建立这样的关系, 我们将针对测试数据库运行 **yiic shell** 命令。因为测试数据库的连接字符串被定义在测试配置文件 (**/protected/config/test.php**) 中, 我们需要对此文件使用 **yiic shell** 而不是 **main.php**。这非常简单, 因为 **yiic shell** 命令允许你指定不同的配置文件。让我们再次启动 **yiic shell**, 并且指向测试数据库:

SHELL 代码或屏幕回显:

```
% YiiRoot/framework/yiic shell protected/config/test.php

Yii Interactive Tool v1.1 (based on Yii v1.1.2)

Please type 'help' for help. Type 'exit' to quit.

>> rbac

This command will create three roles: Owner, Member, and Reader and the following premissions:

create, read, update and delete user

create, read, update and delete project

create, read, update and delete issue

Would you like to continue? [Yes|No] Yes

Authorization hierarchy successfully generated.

>> exit
```

至此, 测试数据库中也存在了相应的 **RBAC** 授权层次结构。

为用户分配角色

1

目前我们所做的是建立了一个授权层次结构，但并未给用户分配权限。通过给用户分配 **owner**，**member**，或 **reader** 三个角色之一来完成分配权限的工作。例如，如果你想给独立用户 ID 为 1 的用户分配 **member** 角色，我们只需要如下操作：

PHP 代码：

```
$auth=Yii::app()->authManager;  
  
$auth->assign('member',1);
```

一旦建立了关系，检查权限就非常简单了。我们只需要询问应用程序的用户组件，当前用户是否有足够的权限。例如，我们想知道当前用户是否有新建 **Issue** 的权限，我们只需要遵循如下语法结构：

PHP 代码：

```
if(Yii::app()->user->checkAccess('createIssue')){  
  
    //perform needed logic  
  
}
```

在这个例子中，我们给用于 ID 为 1 的用户授予 **member** 角色，并且在我们权限层次结构中，**member** 角色继承到了 **createIssue** 权限，前面提到的 **if** 语句将会返回一个 **true**，如果我们使用 ID 为 1 的用户登录。

在添加一个新用户到项目时我们将此授权逻辑作为业务逻辑代码的一部分来执行。我们将添加一个新的表单来帮助我们为项目添加用户，同时选择适当的角色。但在此之前，我们需要解决另一方面的问题：什么样的用户角色需要被应用到这个应用程序中，即他们需要基于不同的项目申请。

为项目添加 RBAC 角色

现在我们有基础的 **RBAC** 授权模型，但没有将这些关系使用到整个应用程序中。在 **TrackStar** 应用程序中我们需要做的有一些复杂。我们需要针对项目定义角色，而不是针对整个应用程序的全局访问来定义角色。我们需要允许用户在不同项目中担任不同角色。举个例子，一个用户可能在一个项目中为 **reader** 角色，在另外一个项目中为 **member** 角色，在第三个项目中为 **owner** 角色。用户可以存在于多个项目中，并且必须指明他们在每一个项目中的角色。

Yii 内置的 **RBAC** 架构中没有任何符合我们要求的内置功能。**RBAC** 模型倾向于建立角色和权限之间的关系。它对 **TrackStar** 项目一无所知，也不需要知道。为了在我们的授权层次结构中指定额外的维度，我们需要建立一张独立的数据表来维护用户，角色，项目之间的关系。以下为此表的数据库定义语句：

SQL 代码:

```
createtabletbl_project_user_role
(
    project_idINTEGERNOTNULL,
    user_idINTEGERNOTNULL,
    roleVARCHAR(64)NOTNULL,
    primarykey(project_id, user_id, role),
    foreignkey(project_id)referencestbl_project(id),
    foreignkey(user_id)referencestbl_user(id),
    foreignkey(role)referencessAuthItem(name)
);
```

2

所以打开你的数据库编辑器建立这张表，并且请确认它存在于主数据库和测试数据库之中。

添加 RBAC 业务逻辑

尽管之前的数据表将会保存一些基本信息来回答：一个用户是否被授予了一个特殊项目中的角色，但是仍然需要我们的授权层次结构来回答：用户是否有足够的权限来执行相关功能。虽然 Yii 中的 RBAC 模型并不知道我们的 **TrackStar** 项目，但是它有一些强悍的功能可以被利用。当你新建一个授权项目或赋予一个用户授权项目后，你可以适用一小段 PHP 代码(调用 `Yii::app()->user->checkAccess()`)进行检测。如果权限被授予了，上面的代码将在用户获得授权许可之前返回一个 `true`。

对这一功能一个有用的例子是在应用程序中允许用户更新个人资料。一般情况下，应用程序需要确保用户只拥有更新自己资料的权限，而不是其他人的。所以我们需要建立一个叫 `updateProfile` 的授权项目，关联一定的业务逻辑来检查当前用户 ID 和需要更新资料的用户 ID 是否一致。

在我们的例子中，我们将为角色授予关联一个业务规则。当我们为用户授予一个角色时，我们会适用业务规则来检测项目中的关系。`checkAccess()` 方法允许我们传递一个业务规则执行需要的附加数组。我们将适用这一功能传递至当前项目，这样业务规则可以调用 **Project AR** 类中的方法来判断当前用户是否被授予了当前项目中的角色。

我们将要创建的业务规则与每个角色的分配略有不同。例如，当我们给一个用户赋予 **owner** 角色时，将这样写：

PHP 代码:

```
$bizRule='return isset($params["project"]) && $params["project"]->isUserInRole('owner');';
```

member 和 **reader** 语句类似。

在调用 **checkAccess()** 方法后我们将可以访问项目内容。现在举一个例子检查用户访问 **createIssue** 权限，代码如下：

PHP 代码：

```
$params=array('project'=>$project);  
  
if(Yii::app()->user->checkAccess('createIssue',$params))  
{  
  
    //proceed with issue creation logic  
  
}
```

这里的 **\$project** 变量是带有当前项目内容的 **Project AR** 类实例(请明确，我们应用程序中的所有功能都出现在一个项目中)。

实现新 Project AR 类方法

我们已经添加了一张新数据表来存放用户，角色，项目之间的关系，现在我们需要实现用来管理和辨别表中数据的方法。我们添加公共方法到 **Project AR** 类来控制数据的添加和移除，同时用来完成辨别工作。如你所猜，我们会先写一个测试。

首先，让我们添加建立用户，项目和角色之间关系的功能。打开位于 **protected/tests/unit/ProjectTest.php** 的单元测试文件，添加如下测试：

PHP 代码：

```
publicfunctiontestUserRoleAssignment()  
{  
  
    $project = $this->projects('project1');  
  
    $this->assertEquals(1,$project->associateUserToRole());  
  
}
```

然后运行测试：

SHELL 代码或屏幕回显：

```
% cd /Webroot/protected/tests/
```

```
% phpunit unit/ProjectTest.php
```

```
PHPUnit 3.4.12 by Sebastian Bergmann.
```

```
.....E
```

```
Time: 0 seconds
```

```
There was 1 error:
```

```
1) ProjectTest::testUserRoleAssignment
```

```
CException: Project does not have a method named "associateUserRole".
```

```
... FAILURES!
```

```
Tests: 6, Assertions: 13, Errors: 1.
```

因为很明显的原因，测试无法通过。我们需要向 **Project AR** 类添加包含一个角色名、一个用户 **ID** 和已建立的基于角色、用户和项目之间关联的公共方法。打开位于 **protected/models /Project.php** 的文件，并且将下面有足够的逻辑可以让测试通过的代码添加进去：

PHP 代码：

```
/**
```

```
 * creates an association between the project, the user and the user's role within the project
```

```
 */
```

```
public function associateUserRole()
```

```
{
```

```
    return 1;
```

```
}
```

再次运行，测试将返回成功的结果。

SHELL 代码或屏幕回显：

```
% phpunit unit/ProjectTest.php
```

```
PHPUnit 3.4.12 by Sebastian Bergmann.
```

.....

Time: 0 seconds

OK (6 tests, 14 assertions)

现在让我们更新测试，传入角色名和用户 ID:

PHP 代码:

```
public function testUserRoleAssignment()
{
    $project = $this->projects('project1');
    $user = $this->users('user1');
    $this->assertEquals(1, $project->associateUserToRole('owner', $user->id));
}
```

然后修改 `Project::associateUserToRole()` 方法，来加入这些参数，并在表 `tbl_project_user_role` 中插入一行:

PHP 代码:

```
public function associateUserToRole($role, $userId)
{
    $sql = "INSERT INTO tbl_project_user_role (project_id, user_id, role) VALUES (:projectId, :userId, :role)";
    $command = Yii::app()->db->createCommand($sql);
    $command->bindValue(":projectId", $this->id, PDO::PARAM_INT);
    $command->bindValue(":userId", $userId, PDO::PARAM_INT);
    $command->bindValue(":role", $role, PDO::PARAM_STR);
    return $command->execute();
}
```

这里我们使用了 Yii 框架中的 `CDbCommand` 类针对数据库执行 SQL 语句。`CDbCommand` 的一个实例返回自基于数据库连接的 `createCommand()` 方法调用。我们还使用了 `CDbCommand` 类的 `bindValue()` 方法来绑定参数值。这是一个减少 SQL 注入攻击威胁的好方法，同时也帮助我们改进 SQL 语句多次执行的性能。

`CDbCommand::execute()` 方法一般在执行 `insert` 语句前返回受影响的行数。一次成功的 `insert` 将会影响 1 行，所以整数 1 将被返回。测试将执行的返回结果同 1 做比较。如果你一直跟随我们的学习，测试将会通过。但是当你第二次运行，将会发现数据库完整性约束违规的失败，因为它将尝试 2 次插入相同的主键。我们需要花费一些时间来解决这一问题。

当我们在测试中处理数据表时，我们应当已经为该表添加了夹具，使得以可重复和一致性的方式运行我们的测试。

在 `fixtures` 文件夹(`protected/tests/fixtures/`)添加一个叫 `tbl_project_user_role.php` 的新文件，我们只需简单的返回一个空数组：

PHP 代码：

```
<?php  
  
return array()  
  
);
```

接下来，修改位于 `protected/tests/unit/Project-Test.php` 中的 `fixtures` array，使之包含新的夹具：

PHP 代码：

```
public $fixtures = array(  
  
    'projects' => 'Project',  
  
    'users' => 'User',  
  
    'projUserAssign' => 'tbl_project_user_assignment',  
  
    'projUserRole' => 'tbl_project_user_role',  
  
);
```

即使我们并未为我们的夹具添加任何实际数据，夹具管理器将清空我们的 `tbl_project_user_role` 表，也就是说在每次测试前移除所有插入的行。现在我们可以多次运行我们的测试而不用担心任何数据库错误。

当我们改变用户在项目中的角色，或者从一个项目中移除用户时，我们需要移除这样的关联。因此，让我们为此添加一个方法。我们可以使用同一个测试。

让我们修改我们的测试，添加一个调用来移除关联，在添加的后面：

PHP 代码:

```
public function testUserRoleAssignment()
{
    $project = $this->projects('project1');

    $user = $this->users('user1');

    $this->assertEquals(1, $project->associateUserToRole('owner', $user->id));

    $this->assertEquals(1, $project->removeUserFromRole('owner', $user->id));
}
```

再次运行测试，你将看到失败。我们需要在 **Project AR** 类中实现这个新方法。在类的尾部添加下面的方法:

PHP 代码:

```
/**
 * removes an association between the project, the user and the user's role within the project
 */
public function removeUserFromRole($role, $userId){

    $sql = "DELETE FROM tbl_project_user_role WHERE project_id=:projectId AND user_id=:userId AND
    role=:role";

    $command = Yii::app()->db->createCommand($sql);

    $command->bindValue(":projectId", $this->id, PDO::PARAM_INT);

    $command->bindValue(":userId", $userId, PDO::PARAM_INT);

    $command->bindValue(":role", $role, PDO::PARAM_STR);

    return $command->execute();
}
```

只是简单的从数据库中删除了角色，用户和项目之间的关联行。如果成功删除将返回影响的行数为整数 **1**。至此，我们的测试添加并删除了新关联。我们需要再次运行确保一切正常。

SHELL 代码或屏幕回显:

```
% phpunit unit/ProjectTest.php
```

```
PHPUnit 3.4.12 by Sebastian Bergmann.
```

```
.....
```

```
Time: 1 second
```

```
OK (6 tests, 15 assertions)
```

我们已经实现了添加和删除关联的方法。现在我们需要添加函数来判断一个给出的用户是否与项目中的角色有关系。为此我们还需要向 **Project AR** 类添加一个公共方法。

所以，作为一个测试，添加如下的测试方法到 **ProjectTest.php**:

PHP 代码:

```
public function testIsInRole()  
{  
    $project = $this->projects('project1');  
    $this->assertTrue($project->isUserInRole('member'));  
}
```

这个被设计用来测试 **Project::isUserInRole()** 方法的实现。因为我们并未实现这一方法，测试显然会失败。确保结果如下:

SHELL 代码或屏幕回显:

```
% phpunit unit/ProjectTest.php
```

```
PHPUnit 3.4.12 by Sebastian Bergmann.
```

```
.....E
```

```
Time: 0 seconds
```

```
There was 1 error:
```

```
1) ProjectTest::testIsInRole
```

```
CException: Project does not have a method named "isUserInRole".
```

```
...
```

```
FAILURES! Tests: 7, Assertions: 15, Errors: 1.
```

使它通过，需添加如下方法到 **Project AR** 类的底部：

PHP 代码：

```
/**
 * @return boolean whether or not the current user is in the specified role within the context
 * of this project
 */
public function isUserInRole($role)
{
    return true;
}
```

这已经足够使得测试通过了。

SHELL 代码或屏幕回显：

```
% phpunit unit/ProjectTest.php
```

```
...
```

```
OK (7 tests, 16 assertions)
```

下面我们需要实现足够的逻辑来确定是否存在一定的关联。修改 **Project AR** 类中的关系：

PHP 代码：

```
public function isUserInRole($role)
{
    $sql = "SELECT role FROM tbl_project_user_role WHERE project_id=:projectId AND user_id=:userId
    AND role=:role";

    $command = Yii::app()->db->createCommand($sql);

    $command->bindValue(":projectId", $this->id, PDO::PARAM_INT);
```

```

$command->bindValue(":userId", Yii::$app()->user->getId(), PDO::PARAM_INT);

$command->bindValue(":role", $role, PDO::PARAM_STR);

return $command->execute() == 1 ? true : false;

}

```

这里再次执行 SQL 语句直接从我们表中读取。它需要输入角色名和当前用户信息 (`Yii::$app()->user`)，来组成搜索所需的主键。再次运行测试：

SHELL 代码或屏幕回显：

```

% phpunit unit/ProjectTest.php

...

Time: 1 second, Memory: 14.25Mb

There was 1 failure:

1) ProjectTest::testIsInRole

Failed asserting that <boolean:false> is true.

...

FAILURES!

Tests: 7, Assertions: 16, Failures: 1.

```

测试再次失败。失败的原因是 `isUserInRole()` 方法使用了 `Yii::$app()->user->getId()` 来获得当前用户 ID，但实际上不会返回任何东西。我们的测试并不会在调用前直接设置当前用户。让我们编辑逻辑代码来设置当前用户 ID。这样修改测试方法：

PHP 代码：

```

public function testIsInRole()
{
    $user = $this->users('user1');

    Yii::$app()->user->setId($user->id);

    $project = $this->projects('project1');
}

```

```
$this->assertTrue($project->isUserInRole('member'));
```

```
}
```

这里设置了当前用户 ID 为用户夹具信息里 **user1** 的。让我们再次运行测试：

SHELL 代码或屏幕回显：

```
% phpunit unit/ProjectTest.php
```

```
...
```

```
Time: 1 second, Memory: 14.25Mb
```

```
There was 1 failure:
```

```
1) ProjectTest::testIsInRole
```

```
Failed asserting that <boolean: false> is true.
```

```
...
```

```
FAILURES!
```

```
Tests: 7, Assertions: 16, Failures: 1.
```

我们的测试依然是失败的，这是因为表中并不存在相关行，**user1** 并不是此项目的 **owner**。所以让我们在调用 **isUserInRole()** 方法之前先建立关联。

我们可以使用我们之前在测试中建立的添加和移除这些关联的其他方法来建立这样的关系。然而，为了试图保持这个测试的独立于其他测试或 **Project AR** 类中的方法，我们将依靠夹具数据来提供初始化条件。

在我们初次添加夹具文件(**tests/fixtures/tbl_project_user_role.php**)时，我们简单的让其返回一个空数组。作一些修改让其提供一行 **ProjectID** 为 **2**，用户 **ID** 为 **2**，角色为 **member** 的信息：

PHP 代码：

```
return array(  
    'row1' => array(  
        'project_id' => 2,  
        'user_id' => 2,  
        'role' => 'member',  
    ),  
);
```

```
);
```

有了之前的添加和移除使用 **user1** 和 **project1** 为夹具数据的测试经验，我们可以避免不同 **ID** 数据的冲突。

现在我们将使用夹具数据来设置建立 **Project AR** 类时的用户 **ID**。如下修改测试方法：

PHP 代码：

```
public function testIsInRole()
{
    $row1 = $this->projUserRole['row1'];

    Yii::app()->user->setId($row1['user_id']);

    $project=Project::model()->findByPk($row1['project_id']);

    $this->assertTrue($project->isUserInRole('member'));
}
```

这里我们使用了在类的顶部定义的叫 **projUserRole** 的夹具数据来取回针对行数据。然后我们使用这个数据来设置用户 **ID**，同时通过调用 **Project::model()->findByPk** 创建 **Project AR** 类实例。下面我们通过测试来确保用户被授予了 **member** 角色。如果我们运行测试：

SHELL 代码或屏幕回显：

```
% phpunit unit/ProjectTest.php
...
OK (7 tests, 16 assertions)
```

我们的测试再次通过。

我们写并测试添加和移除一个项目中管理的角色，并且通过一个方法来判断一个给定的用户是否与一个项目中的角色相关联。下面我们将写一个最终测试。我们准备写一个端对端应用的测试，来检测我们准备为这个项目添加的而外 **Yii** 的 **RBAC** 结构维度。我们所说实现这一目标是通过添加业务规则到 **Yii RBAC** 授权(每当我们用户和角色关联起来的时候)中。让我们写一个最终方法来测试这个：

打开 **ProjectTest.php** 单元测试文件，添加如下测试方法：

PHP 代码：

```
public function testUserAccessBasedOnProjectRole()
```

```

{

$row1 = $this->projUserRole['row1'];

Yii::app()->user->setId($row1['user_id']);

$project=Project::model()->findByPk($row1['project_id']);

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

$bizRule=' return isset($params["project"]) && $params["project"]->isUserInRole("member");';

$auth->assign('member', $row1['user_id'], $bizRule);

$params=array('project'=>$project);

$this->assertTrue(Yii::app()->user->checkAccess('updateIssue', $params));

$this->assertTrue(Yii::app()->user->checkAccess('readIssue', $params));

$this->assertFalse(Yii::app()->user->checkAccess('updateProject', $params));

}

```

最终测试方法使用了其他的已经存在并且通过测试的 **API** 方法来构成, 所以就不需要经历我们一般的 **TDD** 步骤了。有时, 这一点会引起争论, 比起单元测试更像功能测试, 但我们还是将其放入单元测试类。

我们将使用相同的方式(如同我们之间测试的)来设置用户 **ID** 和通过来自 **tbl_project_user_role.php** 夹具文件中的数据来建立 **Project AR** 类实例。随后我们建立了 **auth** 管理类的实例来授予用户 **owner** 角色。然而, 在我们执行授权操作之前, 我们创建了业务规则。这条业务规则使用 **\$params** 数组, 首先检查是否存在数组中 **project** 元素, 然后调用 **Project AR** 类的 **isUserInRole()** 方法(该方法假设值是数组中的元素)。我们直接传递名称 **owner** 给这个方法, 因为这个角色是我们已经赋予的。最终我们调用了 **Yii RBAC** 相关方法 **Yii::app()->user->checkAccess()** 来确定当前用户是否被授予了基于此项目的我们授权层次结构的 **owner** 角色。

我们检测了用户是否有权限更新 **issue**, **member** 以上的角色拥有这一权限。我们期望返回一个 **true**。我们也制作了一部分断言来测试(和证明)权限的继承。我们期望一个 **member** 角色可以继承 **reader** 的权限。所以我们也测试了一个在我们授权层次结构中为 **reader** 后代角色的用户拥有 **readIssue** 权限。最后, 我们希望拒绝 **owner** 角色的操作。所以, 我们使用测试确保对权限 **updateProject** 返回 **false**。

再次运行测试:

SHELL 代码或屏幕回显:

```
% phpunit unit/ProjectTest.php
```

```
...
```

```
OK (8 tests, 19 assertions)
```

所有的项目测试都通过了。似乎这个方法可以了。

如果我们直接使用代码：

PHP 代码：

```
$auth->assign('member', $row1['user_id'], $bizRule);
```

向 **AuthAssignment** 表插入一行数据，如果我们再次运行测试，我们将会获得一个数据库完整性冲突的警告。简单的说它将试着重复插入相同行，并且会违反我们之前在表中定义的数据完整性约束。为了避免这个，我们需要让夹具管理器也可以控制这张表。与之前一样。在位于 **protected/tests/fixtures/** 的夹具文件夹下建立 **AuthAssignment.php** 文件，使之返回一个空数组。然后修改位于 **ProjectTest.php** 文件顶部的夹具数组，使之包含新建文件：

PHP 代码：

```
public $fixtures=array(  
  
    'projects' => 'Project',  
  
    'users' => 'User',  
  
    'projUserAssign' => 'tbl_project_user_assignment',  
  
    'projUserRole' => 'tbl_project_user_role',  
  
    'authAssign' => 'AuthAssignment',  
  
);
```

现在我们的 **AuthAssignment** 表都会在每次测试前被重置。

在我们完成测试之前，让我们再添加一点来确保项目中没有授权的用户，没有任何权限。因为我们直接设置的关系是基于项目 **id** 等于 **2** 的，让我们使用项目 **id** 等于 **1** 的来测试用户访问权限。在

testUserAccessBasedOnProjectRole()方法尾部加上下面的代码：

PHP 代码：

```
//now ensure the user does not have any access to a project they are not associated with  
  
$project=Project::model()->findByPk(1);  
  
$params=array('project'=>$project);  
  
$this->assertFalse(Yii::app()->user->checkAccess('updateIssue', $params));
```



```
$this->assertFalse(Yii::app()->user->checkAccess('readIssue', $params));  
  
$this->assertFalse(Yii::app()->user->checkAccess('updateProject', $params));
```

在这里我们建立了一个基于 **project_id=1** 的项目实例。因为我们知道用户被没有与该项目关联，所以所有的 **checkAccess()** 调用都会返回 **false**。

添加用户到项目

在上一次迭代中，我们添加为应用程序创建新用户的功能。但是我们并没有方法将用户分配到项目，更深一层，分配用户到项目中的角色。现在我们完成了我们的 **RBAC** 练习，是时候完成这一新功能了。

所需功能的实现包括了几处代码的改变。然而，我们提供了相似的需要改变类型的例子，并且在实现之前的迭代的功能中，包含了所有相关的概念。所以，在这一步我们会比较快，并且只停下来强调几件我们没有提到的事情。介于这样，读者需要在没有帮助的情况下完成改变，并且这是一个亲手练习的好机会。为了进一步鼓励这样的练习，我们将首先列出完成新需求所需要作的所有事。你可以合上书，自己试着做一些，然后再看下面提供的实现方法。

达成目标所需要做的事：

- § 使用测试优先的方法，为 **Project** 模型类添加一个叫 **getUserRoleOptions()** 的 **public static** 方法，用来返回一系列正确的取自 **auth manager** 的 **getRoles()** 方法的可选角色。在添加用户到项目时我们将使用这一列表来填充一个可选角色下拉菜单。
- § 使用测试优先的方法，为 **Project** 模型类添加一个叫 **associateUserToProject(\$user)** 的 **public** 方法，来关联用户到项目。这一方法将直接通过插入数据到 **tbl_project_user_assignment** 表的方式来建立用户和项目之间的关联。
- § 使用测试优先的方法，为 **Project** 模型类添加一个叫 **isUserInProject(\$user)** 的 **public** 方法，来判断用户是否已经与项目关联。我们将在验证提交的规则中使用它，来防止尝试将重复的用户添加到同一个项目中。
- § 添加一个叫 **ProjectUserForm** 新表单模型类，扩展自 **CFormModel** 作为一个新的输入表单模型。为此表单模型类添加 3 个属性：**\$username**、**\$role** 和 **\$project**。同时添加验证规则来确保 **username** 和 **role** 必须被填写，然后 **username** 迟一些将被类自定义方法 **verify()** 验证。**verify()** 方法应该是这样的：
 - 试着用匹配输入的用户名来创建一个 **User AR** 类的实例
 - 如果尝试成功了，将用新方法 **associateUserToProject(\$user)** 将用户关联到一个项目上，此方法是在本章早期将用户和角色关联时添加的。如果没有与用户名匹配的用户被找到，需要设置返回一个错误。（如果需要，看一下 **LoginForm::authenticate()** 如何进行自定义验证的）
- § 在 **views/project** 目录建立一个叫 **adduser.php** 的新 **view** 文件，来显示我们的新添加用户到项目的表单。这个表单只需要 2 个输入：**username** 和由下拉菜单提供的 **role**
- § 在 **ProjectController** 类下添加一个叫 **actionAdduser()** 的新控制器行为方法，然后修改此控制器的 **accessRules()** 方法来确保一个已认证的会员发起访问。新的方法用来负责渲染新 **view** 来显示表单，并且处理表单提交的数据。

再提醒一下，我们非常鼓励读者尝试自己完成上述修改。我们在下面的部分列出了修改的代码。

修改 Project 模型类

我们为 **Project** 类添加了 3 个新 **public** 方法，其中一个为 **static**，因此可以在非实例化的情况下调用：

PHP 代码：

```
/**
 * Returns an array of available roles in which a user can be placed when being added to a project
 */

public static function getUserRoleOptions()
{
    return CHtml::listData(Yii::app()->authManager->getRoles(), 'name', 'name');
}

/**
 * Makes an association between a user and a the project
 */

public function associateUserToProject($user)
{
    $sql = "INSERT INTO tbl_project_user_assignment (project_id, user_id) VALUES
    (:projectId, :userId)";

    $command = Yii::app()->db->createCommand($sql);

    $command->bindValue(":projectId", $this->id, PDO::PARAM_INT);

    $command->bindValue(":userId", $user->id, PDO::PARAM_INT);

    return $command->execute();
}
```

```

/**
 * Determines whether or not a user is already part of a project
 */

public function isUserInProject($user)
{
    $sql = "SELECT user_id FROM tbl_project_user_assignment WHERE project_id=:projectId AND
    user_id=:userId";

    $command = Yii::app()->db->createCommand($sql);

    $command->bindValue(":projectId", $this->id, PDO::PARAM_INT);

    $command->bindValue(":userId", $user->id, PDO::PARAM_INT);

    return $command->execute()==1 ? true : false;
}

```

对于上面的代码没有任何特殊说明。因为它们都是 **Project** 模型类的 **public** 方法，我们 **ProjectTest** 单元测试类中添加了下面 2 个测试方法作为结束：

PHP 代码：

```

public function testGetUserRoleOptions()
{
    $options = Project::getUserRoleOptions();

    $this->assertEquals(count($options), 3);

    $this->assertTrue(isset($options['reader']));

    $this->assertTrue(isset($options['member']));

    $this->assertTrue(isset($options['owner']));
}

public function testUserProjectAssignment(){

```

```
//since our fixture data already has the two users assigned to project 1, we'll assign user 1
to project 2

$this->projects('project2')->associateUserToProject($this->users('user1'));

$this->assertTrue($this->projects('project1')->isUserInProject($this->users('user1')));

}
```

添加新表单模型类

如果在 **login from** 中做的练习一样，我们试着建立一个新的表单模型类用来存放表单的输入和进行集中验证。这是一个非常简单的继承自 **Yii CFormModel** 类的类，它的属性和我们的输入框匹配，也有一个来存放正确的项目信息。我们需要利用这个项目信息来添加用户到项目。整个类的代码如下：

PHP 代码：

```
<?php

/**
 * ProjectUserForm class.
 *
 * ProjectUserForm is the data structure for keeping
 *
 * the form data related to adding an existing user to a project. It is used by the 'Adduser'
action of 'ProjectController'.
 */

class ProjectUserForm extends CFormModel
{
    /**
     * @var string username of the user being added to the project
     */
    public $username;

    /**
     * @var string the role to which the user will be associated within the project
     */
}
```

```

    */

public $role;

/**
 * @var object an instance of the Project AR model class
 */

public $project;

/**
 * Declares the validation rules.
 * The rules state that username and password are required,
 * and password needs to be authenticated using the verify() method
 */

public function rules()
{
    return array(
        // username and password are required
        array('username', 'role', 'required'),

        // password needs to be authenticated
        //array('username', 'verify'),

        array('username', 'exist', 'className' => 'User'),

        array('username', 'verify'),
    );
}

```

```

/**
 * Authenticates the existence of the user in the system.
 * If valid, it will also make the association between the user,
 * role and project * This is the 'verify' validator as declared in rules().
 */

public function verify($attribute, $params)
{
    if(!$this->hasErrors())// we only want to authenticate when no other input errors are present
    {
        $user = User::model()->findByAttributes(array('username'=>$this->username));

        if($this->project->isUserInProject($user)){

            $this->addError('username', 'This user has already been added to the project. ');

        }else{

            $this->project->associateUserToProject($user);

            $this->project->associateUserToRole($this->role, $user->id);

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

            $bizRule='return isset($params["project"]) &&
            $params["project"]->isUserInRole("' . $this->role . '");';

            $auth->assign($this->role, $user->id, $bizRule);

        }

    }

}

```

添加新 action 方法到 Project Controller

我们需要一个控制器 **action** 来处理初始化请求呈现给用户添加用户到项目的表单。我们将这些添加至 **ProjectController** 类，并且命名为 **actionAdduser()**。代码如下：

PHP 代码：

```
public function actionAdduser()

{

    $form=newProjectUserForm;

    $project = $this->loadModel();

    // collect user input data

    if(isset($_POST['ProjectUserForm'])){

        $form->attributes=$_POST['ProjectUserForm'];

        $form->project = $project; // validate user input and set a sucessfull flassh message if valid

        if($form->validate())

        {

            Yii::app()->user->setFlash('success', $form->username . " has been added to the project.");

            $form=newProjectUserForm;

        }

    }

    // display the add user form

    $users = User::model()->findAll();

    $usernames=array();

    foreach($users as $user)

    {

        $usernames[]=$user->username;

    }

    $form->project = $project;
```

```
$this->render('adduser', array('model' => $form, 'usernames' => $usernames));  
}
```

2

对这些我们都比较熟悉。它即处理了初始化的 **GET** 请求来呈现表单也处理了表单提交后的 **POST** 请求。它和我们 **site controller** 内的 **actionLogin()** 方法非常相似。前面代码中的高亮部分是一些我们之前没见过的。如果提交的请求获得成功，它将设置一条 **flash message**(暂存消息)。一条 **flash message** 就是一条暂时储存在 **session** 里的消息。它只在本次和下次请求有效。在这里我们使用了 **CWebUser** 应用程序用户组件中的 **setFlash()** 方法在请求成功时储存一个临时消息。在我们谈论 **view** 之前，我们要看看如何获取这一消息，并且将其呈现给用户。

在前面的代码中，我们创建了一个包含当前系统中可用 **usernames** 的数组。我们将使用这个数组来填充一个叫 **CAutoComplete** 的 **Yii UI Widgets**(部件)，我们将把这个 **widget** 用在 **username** 表单输入框。正如它的名字一样，当我们在表单输入框内输入时，它将根据数组中的元素提供可选建议。

我们对 **ProjectController** 类进行的额外修改是，将这个新的 **action** 方法添加至基本访问规则列表，使得成功登录用户可以访问这个 **action**。

PHP 代码:

```
public function accessRules()  
{  
  
    return array(  
  
        array('allow', // allow all users to perform 'index' and 'view' actions  
  
            'actions' => array('index', 'view', 'adduser'),  
  
            'users' => array('@'),  
  
        ),  
  
    );  
}
```

添加新的 view 文件来呈现表单

我们的新 **action** 方法调用了 **->render('adduser')** 来渲染一个 **view**(视图)文件，因此我们需要建立一个视图文件。完整的 **protected/views/project/adduser.php** 代码如下:

PHP 代码:

```
<?php
```



```

$this->pageTitle=Yii::app()->name . ' - Add User To Project' ;

$this->breadcrumbs=array(

$model->project->name=>array('view','id'=>$model->project->id),

'Add User' ,

);

$this->menu=array(

array('label'=>'Back To Project' ,

'url'=>array('view','id'=>$model->project->id)),

);

?>

<h1>AddUserTo<?phpecho$model->project->name; ?></h1>

<?phpif(Yii::app()->user->setFlash('success')): ?>

<divclass="successMessage">

<?phpechoYii::app()->user->setFlash('success'); ?>

</div>

<?phpendif; ?>

<divclass="form">

<?php$form=$this->beginWidget('CActiveForm'); ?>

<pclass="note">Fieldswith<spanclass="required">*</span>arerequired.</p>

<divclass="row">

<?phpecho$form->labelEx($model,'username'); ?>

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

```

```

        'model' => $model ,

        'attribute' => 'username' ,

        'data' => $usernames ,

        'multiple' => false ,

        'htmlOptions' => array('size' => 25) ,

    )); ?>

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

</div>

<div class="row">

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

    <?php echo $form->dropDownList($model , 'role' , Project::getUserRoleOptions()); ?>

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

</div>

<div class="row buttons">

    <?php echo CHtml::submitButton('Add User'); ?>

</div>

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

</div>

```

这里的大部分我们都已经见过了。我们定义了 **active label** 和 **active form** 元素并且和 **ProjectUserForm** 联系起来。我们使用之前的 **project** 模型类中定义的 **static** 方法填充下拉菜单。我们也在操作菜单面板添加了一个返回项目细节页的连接。

上面代码中的高亮部分是新未见过的。这是针对我们在 **actionAdduser()** 方法中介绍的 **flash message** 方法的一个实例。我们通过使用同一个用户组件的 **hasFlash('success')** 判断是否有相关 **flash message**，来获取由 **setFlash()** 设置的消息。我们会在 **hasFlash()** 方法中传入一个当时设置 **message** 时的名字。这是一个不错的方法向用户显示一些关于他前面提交请求的反馈。

另外一个改变是在项目细节页面添加了一个链接，使得我们可以在应用程序中打开这一表单。下面的一行代码被添加到 **project** 的 **view** 文件: **show.php** 来显示可选链接:

PHP 代码:

```
[<?php echo CHtml::link('Add User To Project', array('adduser', 'id' => $model->projectId)); ?>]
```

2

这样我们就可以访问到新表单了。

将所有的组装到一起

所有的改变都完成后，我们可以在浏览某一个项目细节页是链接至我们的新表单。例如，通过链接:

<http://localhost/trackstar/index.php?r=project/view&id=1> 来浏览 **project id** 等于 **1** 的记录，点击右侧可选菜单里的 **Add User To Project** 超链接，将会显示下面的页面:

你可以利用我们之前建立的表单来添加项目和用户以确保你有足够的数量来操作。然后你可以试着添加用户到项目。当你在 **username** 输入框输入时，会出现智能自动补全的提示。当你尝试添加一个不存在数据库中的用户时，你会收到错误提示。当你添加一个已经在该项目中的用户到该项目时，你会收到另外一个错误提示。当成功添加的时候，你将看到一个短暂的 **flash message** 标识操作成功。

判断授权等级

在本次迭代中需要做的最后一件事是: 为我们已经实现的不同功能模块添加授权判断。在本章的早期我们提出并在后期实现了基于不同角色的 **RBAC** 授权层次结构。除了一处以外，所有的一切都准备好实现通过按照授予用户某一项目的权限来实现允许或拒绝访问相关功能。在发起请求的时候我们还没有实现相关的访问判断。整个应用程序还在使用我们在 **project**, **issue** 和 **user** 控制器中定义的轻量级访问过滤器。我们将只实现其中一个权限的判断，其他的将作为练习留给读者。

我们可以在回顾我们的授权层次结构时发现只有 **project owner** 才可以添加用户到项目。所以，现在我们就来实现这个。我们所做的不是在当前用户是 **project owner** 是才在项目细节页显示添加用户到项目的链接(你可能希望确保已添加最少 **1** 个 **owner**, **1** 个 **member**, **1** 个 **reader** 到 **1** 个项目，这样你才能完成测试)。打开位于: **protected/views/project/view.php** 的 **view** 文件，我们在菜单项中放置了一个添加新用户到项目的链接。将数组元素(添加用户到项目)从菜单数组项中移除，仅当 **checkAccess()** 方法返回 **true** 时，才将其放置在数组的最后。下面的代码展示了菜单项的定义方式:

PHP 代码:

```
$this->menu=array(  
array('label' => 'List Project', 'url' => array('index')),
```

```

array('label'=>'Create Project', 'url'=>array('create')),

array('label'=>'Update Project', 'url'=>array('update', 'id'=>$model->id)),

array('label'=>'Delete Project', 'url'=>'#',

'linkOptions'=>array(

'submit'=>array('delete', 'id'=>$model->id),

'confirm'=>'Are you sure you want to delete this item?'

)

),

array('label'=>'Manage Project', 'url'=>array('admin')),

array('label'=>'Create Issue', 'url'=>array('issue/create', 'pid'=>$model->id)),

);

if(Yii::app()->user->checkAccess('createUser', array('project'=>$model)))

{

$this->menu[] = array(

'label'=>'Add User To Project',

'url'=>array('adduser', 'id'=>$model->id)

);

}

```

这里实现了与我们在本章前期所讨论过相似的方法。我们针对当前用户调用 `checkAccess()` 方法，然后传递给我们需要判断的权限名称。因为我们的角色是基于项目的，所以我们将 `project` 模型实例以数组的形式传递进去。这将允许业务规则执行在授权时所定义的。现在，如果我们以某一项目 `owner` 的身份登录，然后导航至那个项目的细节页，我们将在菜单选项中找到添加用户到这个项目的链接。相对的，如果你以 `member` 或 `reader` 的身份导航至该项目细节页，那个链接将不会显示。

这显然不能阻止精明的用户通过直接导航至该 URL 来获取访问权限。举例说明，如果以 `project #2` 的 `reader` 角色登录应用程序，如果直接导航至 URL:

<http://hostname/trackstar/index.php?r=project/adduser&id=2>，将同样可以获得表单权限。

为了防止这个，我们需要在 `action` 方法内部直接添加访问判断。所以，在 `project` 控制器的 `actionAdduser()` 方法里，我们需要添加判断：

PHP 代码:

```
public function actionAdduser()
{
    $project = $this->loadModel();

    if(!Yii::$app()->user->checkAccess('createUser', array('project'=>$project)))
    {
        throw new \HttpException(403, 'You are not authorized to perform this action.');
```

现在, 当我们直接访问 **URL** 时, 我们的访问将被拒绝, 除非我们使用的是项目 **owner** 这一角色。

我们正准备完成所有功能的访问判断。每一个的实现方式都差不多。

小结

在本次迭代中我们提及了大量内容。首先我们介绍了 **Yii** 提供的基本访问控制过滤器, 它是一个针对控制器里特定 **action** 方法允许和拒绝访问的方法。我们使用它来确保用户在获得任何主要功能访问权限前已经登录。下面我们详细介绍了 **Yii** 的 **RBAC** 模型, 它允许以更复杂的方式来实现访问控制。我们构建了一整套基于应用程序角色的用户授权层次结构。在过程中, 我们介绍了 **Yii** 中控制台程序的写法, 和使用它的好处。接下来我们构建了一个新功能实现添加用户到某一项目, 并且同时赋予用户该项目中的角色。最终, 我们发现了实现所需的访问判断的方法, 在整个应用程序中利用 **RBAC** 层次结构实现允许/拒绝对一些功能的访问。

第九章：迭代 6：添加用户评论

在上两次迭代方法中, 随着用户管理可执行, 我们 **Trackstar** 才真正变得有条理。现在, 我们将应用程序的主要功能实现抛到身后。我们可以开始关注一些 **nice-to-have**(更好) 的功能。首先, 我们要做的就是开发一个让用户可以给主题作出评论的权限。

提供一个主题跟踪工具是用户评论功能的一个重要部分, 其中一个方式是让用户直接留言, 内容来自一个主题的对话。该内容将成为这个主题的一个即时会话, 这也有助于时刻了解所有主题从发起到结束的整个

过程。为了让用户了解这些内容，我们也将使用更多内容来说明使用 **Yii** 挂件和建立一个组件模型（要了解更多关于 **Portlets** 的信息，请参考 <http://en.wikipedia.org/wiki/Portlet>）。

迭代计划

这一节的目标是在 **Trackstar** 程序里实现允许用户阅读主题和发表看法的功能，当用户查看任何项目的细节时，他们应该能够读取以前添加的所有评论，以及关于这个主题上新的评论。我们也希望能够给项目列表页面增加一些主题内容剪辑或者 **portlet**，以便在所有主题的左侧显示一个最新内容的列表。对于了解最近用户的活跃性，并允许用户方便地访问到正在积极讨论的最新主题，提供这样一个区域将是一个不错的方式。

我们将要一步一步完成下面这些高级任务来实现上述目标：

- § 设计并创建一个支持我们工作的数据库；
- § 创建一个 **Yii** 的 **AR** 类关联到我们创建的数据库表；
- § 在主题详情页面添加一个表单以供用户提交评论；
- § 在主题详情页面显示所有与主题相关联的内容列表；
- § 在项目列表页面利用 **Yii** 挂件的列出显示大部分最近内容。

添加模型

跟往常一样，我们可以运行已有的测试程序，我们事先写好的程序达到了我们的预期目标。这时，你应该熟悉了如何做到的。我们把已经完成的源码留给读者，已确保所有的单元都能正常运行。

首先我们要创建一个数据库表，下面是这个表的内容：

SQL 代码：

```
CREATETABLEtbl_comment
(
`id`INTEGERNOTNULLPRIMARYKEYAUTO_INCREMENT,
`content`TEXTNOTNULL,
`issue_id`INTEGER,
`create_time`DATETIME,
`create_user_id`INTEGER,
`update_time`DATETIME,
```

```
`update_user_id` INTEGER  
  
)
```

由于每个评论属于一个具体问题，由 `issue_id` 确定，并由 `create_user_id` 标识的特定用户编写，另外我们还需要定义以下外键关系：

SQL 代码：

```
ALTERTABLE`tbl_comment`ADDCONSTRAINT`FK_comment_issue`FOREIGNKEY(`issue_id`)  
  
REFERENCES`tbl_issue`(`id`);  
  
ALTERTABLE`tbl_comment`ADDCONSTRAINT`FK_comment_author`FOREIGNKEY(`create_user_id`)  
  
REFERENCES`tbl_user`(`id`);
```

如果你是依照步骤来的，请确保该表在 `trackstar_dev` 和 `trackstar_test` 两个数据库里均已创建。

一旦一个数据库表就位后，添加 **AR** 关联类就变得易如反掌了。在先前的章节里我们看过许多次了，我们知道如何正确的实现它。我们简单地使用 **Gii** 代码创建工具里的“**Model Generator**”命令添加一个叫 **Comment** 的 **AR** 类。如果你不明白，可以返回到第 5 和第 6 章查阅如何使用 **Gii** 创建模型类。

既然我们已经创建了问题(**issue**)模型类，我们需要在 **Issue** 模型类中添加一条关联到评论(**comment**)模型。我们还要添加一条关联用于统计查询对应问题(**issue**)的相关评论数量（正是我们在 **Project AR** 类中对问题(**issues**)所做的一样）。**Issue::relations()**方法修改如下：

PHP 代码：

```
publicfunctionrelations()  
  
{  
  
returnarray(  
  
    'requester' =>array(self::BELONGS_TO, 'User', 'requester_id'),  
  
    'owner' =>array(self::BELONGS_TO, 'User', 'owner_id'),  
  
    'project' =>array(self::BELONGS_TO, 'Project', 'project_id'),  
  
    'comments' =>array(self::HAS_MANY, 'Comment', 'issue_id'),  
  
    'commentCount' =>array(self::STAT, 'Comment', 'issue_id'),  
  
);
```

```
}
```

同样的，我们需要改变我们新近创建的 **Comment AR** 类来扩展我们的定制 **TrackStarActiveRecord** 基类，以便它通过我们命名为 **beforeValidate()** 的方法中获益。只需改变类的定义的头部，就像下面这样的：

PHP 代码：

```
<?php

/**
 * This is the model class for table "tbl_comment".
 */

class Comment extends TrackStarActiveRecord
{
```

我们将在 **Comment::relations()** 方法里做最后一次小的变动。关系属性被创建时以作者的用户名命名，我们将 **createUser** 更名为 **author**，这个相关的用户不代表评论作者。这只是一种演变，但是将有助于我们的代码更容易阅读和理解。修改方法如下：

PHP 代码：

```
/**
 * @return array relational rules.
 */

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(

        'author' => array(self::BELONGS_TO, 'User', 'create_user_id'),

        'issue' => array(self::BELONGS_TO, 'Issue', 'issue_id'),

    );
}
```


创建一个评论的 CRUD

一旦我们的 **AR** 类就位，创建一个用于管理实体 **CRUD** 脚手架同样容易。同样的，使用 **Gii** 工具的 **Crud Generator** 命令创建一个名字为 **Comment** 的 **AR** 类。同样的，我们在以前的章节中看到很多次了，所以我们将这个过程留给读者作为练习。同样，如果有需要，可以回到第 5 章和第 6 章如何使用 **Gii** 创建 **CRUD** 代码。尽管不能让 **CRUD** 马上为我们操作评论，但是在某些地方有这样一个脚手架还是相当好使的。

只要我们都已经登录了，我们现在应该能够查看自动生成的评论，通过以下地址访问：

<http://localhost/trackstar/index.php?r=comment/create>

修改脚手架以满足要求

就像之前我们看过的许多次一样，我们不得不对脚手架自动生成的代码进行修改以符合应用程序所要达到的要求，比如，我们的自动生成表单对于数据库里的 **tbl_comment** 表内定义的每一个字段都自动生成了一个用于输入的输入框。

我们实际并不需要所有的字段都生成表单。事实上，我们只需要填写评论的这一个输入框就够了。更何况，我们不希望用户通过输入上述网址来访问，而是通过访问一个问题的详细页面。用户将在问题的详情页面添加评论。我们要建立的页面类似下面的截图。

为了达到这一目标，我们将修改我们的 **Issue** 控制器以处理发布评论的方式，同时修改详情视图，用来显示已有的评论和创建新评论的表单。此外，一个评论应该出现在问题的背景范围之内，我们将给 **Issue** 模型添加一个新的方法用来创建新的评论。

添加新评论

我们来给 **Issue** 模型里的 **public** 方法写一个新的测试，打开 **IssueTest.php** 添加下列代码：

PHP 代码：

```
public function testAddComment()
{
    $comment = new Comment;

    $comment->content = "this is a test comment";

    $this->assertTrue($this->issues('issueBug')->addComment($comment));
}
```

```
}
```

当然，这些还不算完工，我们还需要给 **Issue** 的 **AR** 类添加一个新方法，把下列代码加入 **Issue** 的 **AR** 类：

PHP 代码：

```
/**
 * Adds a comment to this issue
 */
public function addComment($comment)
{
    $comment->issue_id=$this->id;

    return $comment->save();
}
```

此方法可以确保在保存问题的评论之前 **ID** 已经被正确设置。再次运行测试，以确保它现在可以通过。

一旦我们的方式完成，我们可以把注意力集中到 **issue** 的控制器类上，随着我们希望发表新的评论，以把显示其数据回 **IssueController::actionView()** 方法，我们将需要修改该方法。我们还是添加一个受保护的方法来处理表单里 **POST** 请求。首先，修改 **actionView()**，如下：

PHP 代码：

```
public function actionView()
{
    $issue=$this->loadModel();

    $comment=$this->createComment($issue);

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

然后添加下列 **protected** 方法用来创建一个新的评论和处理这个新评论表单的 **post** 请求:

PHP 代码:

```
protected function createComment($issue)
{
    $comment = new Comment;

    if (isset($_POST['Comment']))
    {
        $comment->attributes = $_POST['Comment'];

        if ($issue->addComment($comment))
        {
            Yii::app()->user->setFlash('commentSubmitted', "Your comment has been added.");

            $this->refresh();
        }
    }

    return $comment;
}
```

新的受保护的方法 **createComment()** 是负责处理用户输入的一个新评论的 **POST** 请求。如果这个评论成功创建, 该页面将刷新以显示新的评论。 **IssueController::actionView()** 所作的更改负责新方法的调用, 同时也给这个评论的例子填充显示的数据。

显示表单

现在, 我们需要修改视图。首先我们要创建一个新的视图文件来渲染我们的评论和评论的输入表单, 我们将延续命名规定, 开始在文件名前加一个下划线。在 **protected/views/issue/** 下创建一个新文件命名为 **_comments.php**, 并在该文件内输入下列代码:

PHP 代码:

```
<?php foreach($comments as $comment): ?>

<div class="comment">
```

```

<div class="author">

<?php echo $comment->author->username; ?>

</div>

<div class="time">

on<?php echo date('F j, Y \a\t h:i a', strtotime($comment->create_time)); ?>

</div>

<div class="content">

<?php echo nl2br(CHtml::encode($comment->content)); ?>

</div>

<hr>

</div>

<!-- comment -->

<?php endforeach; ?>

```

该文件将成为输入参数数组的一个实例，并将这些参数一个一个显示出来。我们现在需要为问题详情来修改 **view** 文件。我们这么做，打开 **protected/views/issue/view.php**，在文件末尾添加下列代码：

PHP 代码：

```

<div id="comments">

<?php if($model->commentCount>=1): ?>

<h3>

<?php echo $model->commentCount>1 ? $model->commentCount . ' comments' : 'One comment'; ?>

</h3>

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

```

```

<?phpendi f; ?>

<h3>Leave a Comment</h3>

<?phpif(Yii::app()->user->hasFlash('commentSubmitted')): ?>

<div class="flash-success">

<?phpechoYii::app()->user->getFlash('commentSubmitted'); ?>

</div>

<?phpelse: ?>

<?php$this->renderPartial('/comment/_form',array('model'=>$comment, )); ?>

<?phpendi f; ?>

</div>

```

这里我们利用统计查询特性 **commentCount** 来增强先前的 **Issue AR** 模型类。这让我们对出现的任何特定问题都能快速的作出判断。如果是评论，它将使用 **_comments.php** 视图文件来渲染它们，它将显示我们用 **Gii Crud Generator** 创建的表单。它也会显示一个提示保存成功的一闪而过信息。

我们要做的最后一个修改是评论表单本身。就像我们在前面看到的，表单为 **tbl_comment** 数据表里的每个字段都创建了一个输入框，这不是我们想要给用户的。我们需要的是一个包含用户提交评论内容的简单输入表单。因此，打开包含输入表单的 **view** 文件并向下面这样简单的修改一下，就是这个 **protected/views/comment/_form.php**:

PHP 代码:

```

<div class="form">

<?php$form=$this->beginWidget('CActiveForm', array(

'id' => 'comment-form',

'enableAjaxValidation' => false,

)); ?>

<p class="note">Fields with<span class="required">*</span> are required.</p>

<?phpecho$form->errorSummary($model); ?>

```

```

<div class="row">

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

<?php echo $form->textArea($model, 'content', array('rows' => 6, 'cols' => 50)); ?>

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

</div>

<div class="row buttons">

<?php echo CHtml::submitButton($model->isNewRecord ? 'Create' : 'Save'); ?>

</div>

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

</div>

```

有了这一切，我们可以访问某个问题的页面，如：

<http://hostname/trackstar/index.php?r=issue/view&id=1>

我们看到在页面底部的评论输入表单：

如果我们试图提交没有指定任何内容的评论，我们看到像下面截图所描述一个错误：

然后，如果我们登录一个账号为 **Test User One** 的用户，我们提交我的第一个评论：**My first test comment**，我们会看到下面的显示：

创建一个最新评论的挂件

现在，我们具有了留言的功能，我们将主要注意力放在第二个重要目标上。我们想要显示给用户一个穿插在所有页面左侧的，关于不同问题最新评论的表单。这将提供一个用户活跃程度的快照（译者注：这个快照应该说的是类似于进度条一样的图片，不过是每次访问的时候根据实际情况自动生成的）。我们也想要使用在内容区开辟一小块地方的方式，以此使得它在整个网站所有不同位置重复使用。这是在开发如新闻论坛、天气预报之类的 **web portal** 应用中很常见的风格，在雅虎和 **iGoogle** 也同样使用。这一小段内容

经常在像 **portlets** 被提及，这也是为什么我们在本迭代开始的时候提到开发一个 **portlet** 的原因。另外，你可以输入 <http://en.wikipedia.org/wiki/Portlet> 查看更多相关的信息。

CWidget 介绍

运气不错，**Yii** 已经帮我们完成了它的结构。**Yii** 为这个最终目的提供了一个叫做 **CWidget** 的组件类。一个 **Yii** 的 **widget** 就是这些类中的一个实例（或者是子类的实例），它是一个表面组件典型应用，镶嵌在一个视图文件里用来显示自足，可重用的用户界面样式。我们将使用一个 **Yii widget** 来建立一个最新内容 **portlet** 和在主要项目详情页面显示它所以我们能看见评论活跃度穿插在所有问题与项目联系起来。为了展示重用的方便性，我们将更进一步地在该项目的详细页面显示具体意见列表。

开始添加我们的 **widget**，我们将首先在我们的评论 **AR** 模型类上添加一个新的公共方法以便返回大部分最近添加的内容。正如所料，我们先写一个测试程序。

但是在写这个 **test** 方法以前，让我们更新一下我们的评论固定的数据以便在我们整个测试过程中有几个评论内容供使用。在 **protect/tests/fixtures** 下创建一个新文件叫做 **tbl_comment.php**。打开它添加以下内容：

PHP 代码：

```
<?php

return array(

    'comment1' => array(

        'content' => 'Test comment 1 on issue bug number 1',

        'issue_id' => 1,

        'create_time' => '',

        'create_user_id' => 1,

        'update_time' => '',

        'update_user_id'

    ),

    'comment2' => array(

        'content' => 'Test comment 2 on issue bug number 1',

        'issue_id' => 1,
```

```

        'create_time' => '',
        'create_user_id' => 1,
        'update_time' => '',
        'update_user_id'
    ),
);

```

现在我们有了一致的，可预计的和可重复的评论内容可供利用了。

创建一个新的 **unit** 测试文件 **protect/tests/unit/CommentTest.php** 并添加以下内容：

PHP 代码：

```

<?php

class CommentTest extends CDbTestCase
{
    public $fixtures=array(
        'comments' => 'Comment' ,
    );

    public function testRecentComments()
    {
        $recentComments=Comment::findRecentComments();
        $this->assertTrue(is_array($recentComments));
    }
}

```

当然这个测试会失败，因为我们还没有加入 **Comment::findRecentComments()** 方法到 **Comment** 模型类里。所以，我们现在来添加它。我们将继续添加我们所需的所有方法，而不是只添加可供测试应用成功运行的方法。但如果你是一直跟随下来的，你可以随时转移到你自己的步骤。打开 **Comment.php** 添加以下 **public static** 方法：

PHP 代码:

```
public static function findRecentComments($limit=10, $projectId=null)
{
    if($projectId != null)
    {
        return self::model()->with(array(
            'issue' => array('condition' => 'project_id=' . $projectId))->findAll(
                array(
                    'order' => 't.create_time DESC',
                    'limit' => $limit,
                ));
    }
    else
    {
        //get all comments across all projects

        return self::model()->with('issue')->findAll(array(
            'order' => 't.create_time DESC',
            'limit' => $limit,
        ));
    }
}
```

我们的新方法包含了两个可选参数，一个限制了评论的字数，其它的标记了评论所归属的特定项目的 ID。第二个参数将允许我们使用我们新的挂件以显示一个项目详情页面所有项目的内容。所以，如果输入项目 id 被标记，否则，贯穿所有项目的所有内容都将被返回。

Yii 中更多与 AR 关联查询相关的

上述两个 **AR** 关联查询对我们有点新。我们并没有在这之前使用过这些查询选项。以前我们一直在使用最简单的关联查询方法：

1. 加载 **AR** 实例。
2. 访问定义在 **relations()** 方法中的关联属性。

例如：如果我们想查询项目 **id** 为 **#1** 中的所有问题(**issue**)，我们将执行以下两行代码：

PHP 代码：

```
// retrieve the project whose ID is 1

$project=Project::model()->findPk(1);

// retrieve the project's issues: a relational query is actually being performed behind the scenes here

$issues=$project->issues;
```

这个熟悉的方法采用的是延迟加载。当我们第一次创建该项目的实例时，该查询不返回相关的所有问题（**issue**）。它只是一个初步的查询，当执行 **\$project->issues** 才会进一步确定相关的问题。这被叫做惰性查询，因为它会等待加载问题（**issue**）。

这种方法很方便，也非常有效，尤其是在不需要那些相关问题(**issue**)的情况下。但是，在其他情况下，这种方法效率可能有点低。例如，如果我们查询 **N** 个项目的问题(**issue**)，使用这种惰性方式将执行 **N** 次连接查询。如果 **N** 很大，这可能非常低效。在这种情况下，我们选择另一种方式叫做预加载(**Eager Loading**)

预加载方式的关联 **AR** 实例同时也是主 **AR** 实例的请求。这是通过使用 **with()** 方法与 **find()** 或 **findAll()** 方法两者一起的 **AR** 查询。继续我们的项目实例，我们可以执行以代码，使用预加载方式立即查询所有项目的问题(**issue**)：

PHP 代码：

```
//retrieve all project AR instances along with their associated issue AR instances

$projects = Project::model()->with('issues')->findAll();
```

在这种情况下，**\$projects** 数组中的每一个 **project AR** 实例的 **issues** 属性都被赋值了。这一结果仅使用了一个连接查询。

我们在 **findRecentComments()** 方法中使用了两种关联查询。一种是限制了获得的评论是在一个具体的项目(**project**)中。正如你看到的，我们在预加载问题(**issue**)时指定一个查询条件。让我们看看下面这行代码：

PHP 代码：

```
Comment::model()->with(array('issue'=>array('condition'=>'project_id='.$projectId)))->findAll();
```

这个查询连接了 **tbl_comment** 表和 **tbl_issue** 表。继续 **project id** 等于 **#1** 这个例子，上面的 **AR** 关联查询基本执行的是类似下面的 **SQL** 语句：

SQL 代码：

```
SELECTtbl_comment.*, tbl_issue.* FROMtbl_commentLEFTOUTERJOINTbl_issueON
(tbl_comment.issue_id=tbl_issue.id)WHERE(tbl_issue.project_id=1)
```

我们在 **findAll** 方法的参数数组中指定了一个排序(**order**)和限定(**limit**)的条件来执行 **SQL** 语句。

最后一点需要注意的是，在两个查询中我们是如何处理两个表相同列名的歧义。显示，当两个正在连接的表具有相同的列名，我们要在查询时，区别两者。就这个例子而言，两个表都定义了 **create_time** 列。我们尝试排序的列是在 **tbl_comment** 表中而不是定义在 **issue** 表中。在 **Yii** 的 **AR** 的关联查询中，对主表的别名被固定为 **t** 和，而对关联表的别名，默认情况下与关联名相同。因此，在我们这两个查询中，我们指定 **t.create_time** 表示我们要使用主表的列。如果我们想使用 **issue** 表的 **create_time** 列，我们将要修改例子中的第二个查询为如下这样：

PHP 代码：

```
returnComment::model()->with('issue')->findAll(array(
    'order'=>'issue.create_time DESC',
    'limit'=>$limit,
));
```

完成测试

好了，现在我们完全理解我们的新方法是干什么的了，在此基础上，我们需要一个完整的测试。为了充分测试新方法，我们需要为我们 **fixture**（应该是指文件夹名字）数据做一些修改。打开每一个 **fixture** 里的文件：**tbl_project.php**，**tbl_issue.php**和**tbl_comment.php** 并且确保这些入口已经就位。

在 **tbl_project** 中加入以下代码：

PHP 代码：

```
'project3'=>array(
    'name' =>'Test Project 3',
```

```

'description' => 'This is test project 3' ,

'create_time' => '' ,

'create_user_id' => '' ,

'update_time' => '' ,

'update_user_id' => '' ,

),

```

在 **tbl_issue** 加入以下代码:

PHP 代码:

```

'issueFeature2' => array(

'name' => 'Test Feature For Project 3' ,

'description' => 'This is a test feature issue associated with project # 3 that is completed' ,

'project_id' => 3 ,

'type_id' => 1 ,

'status_id' => 2 ,

'owner_id' => 1 ,

'requester_id' => 1 ,

'create_time' => '' ,

'create_user_id' => '' ,

'update_time' => '' ,

'update_user_id' => '' ,

),

```

最后，在 **tbl_comment** 加入以下代码:

PHP 代码:

```

'comment3' => array(

```

```
'content' =>'The first test comment on the first feature issue associated with Project #3',  
  
'issue_id' =>3,  
  
'create_time' =>'',  
  
'create_user_id' =>'',  
  
'update_time' =>'',  
  
'update_user_id' =>'',  
  
),
```

现在我们总计有三个评论在测试数据库中。其中两个分别与项目**#1**和**#3**相关。

现在我们可以改变以下我们的测试方法了：

- \$ 对所有评论发出请求
- \$ 限制返回的评论为两个
- \$ 限制只返回与**#3**相关的评论。

用下面的方法测试三个脚本

PHP 代码：

```
public function testRecentComments()  
{  
  
    //retrieve all the comments for all projects  
  
    $recentComments = Comment::findRecentComments();  
  
    $this->assertTrue(is_array($recentComments));  
  
    $this->assertEquals(count($recentComments), 3);  
  
  
    //make sure the limit is working  
  
    $recentComments = Comment::findRecentComments(2);  
  
    $this->assertTrue(is_array($recentComments));  
  
    $this->assertEquals(count($recentComments), 2);  
}
```

```
//test retrieving comments only for a specific project

$recentComments = Comment::findRecentComments(5, 3);

$this->assertTrue(is_array($recentComments));

$this->assertEquals(count($recentComments), 1);

}
```

我们还需要确保我们的 **CommentTest** 类为 **comments**, **issues** 和 **projects** 而使用 **fixture** 的数据。确保下面的代码添加到 **CommentTest** 类的开头:

PHP 代码:

```
<?php

class CommentTest extends CDbTestCase
{

    public $fixtures=array(

        'comments' => 'Comment',

        'projects' => 'Project',

        'issues' => 'Issue',

    );

}
```

现在, 如果我们在运行这个测试程序, 我们将得到 6 个要求被通过:

SHELL 代码或屏幕回显:

```
>>phpunit unit/CommentTest.php

PHPUnit 3.4.12 by Sebastian Bergmann.

.

Time: 0 seconds

OK (1 test, 6 assertions)
```

在 Yii 中，提供惰性加载和预加载的见解，我们应该针对如何在 `IssueController::actionView()` 方法里加载 `Issue` 模型来作出调整。直到我们修改了 `issues` 的详细视图用来显示我们的内容和内容作者，当我们在这个方法里调用 `loadModel()`，我们才知道使用预加载提前载入我们的内容除了它们各自的作者将更有效。当这么做完以后，我们可以添加一个简单的复选框来标明这个 `loadModel()` 方法是否是我们不希望加载的内容。

像下面这样修改 `IssueController::loadModel()` 方法：

PHP 代码：

```
public function loadModel($withComments=false)
{
    if($this->_model===null)
    {
        if(isset($_GET['id']))
        {
            if($withComments)
            {
                $this->_model=Issue::model()->with(array(
                    'comments'=>array('with'=>'author')))->findByPk($_GET['id']);
            }
            else
            {
                $this->_model=Issue::model()->findByPk($_GET['id']);
            }
        }

        if($this->_model===null) throw new CHttpException(404,'The requested page does not exist.');
```

```
return $this->_model;
}
```

```
}
```

现在我们可以 `IssueController::actionView()` 修改这个方法的调用，像这样：

PHP 代码：

```
public function actionView()  
  
{  
  
    $issue = $this->loadModel(true);
```

当一切就绪，我们将通过调用一次数据库载入所有的内容，以及它们各自作者的信息。

添加 widget

现在我们准备添加新的挂件，以此使用新的方法来显示我们最近的内容。

正如我们前面提到的，在 **Yii** 里一个挂件是框架类 **CWidget** 或它的子类扩展的一个类。我们将在 **protected/components** 里添加一个新的挂件，作为这个文件夹的内容已被指定在主配置文件自动加载。这样，我们不会有明确导入的类，我们希望在每次需要的时候才使用它。我们将给我们的挂件命名为 **RecentComments**，所有我们需要添加一个相同名字 **php** 文件。添加下面这个新创建的类定义到 **RecentComment**：

PHP 代码：

```
<?php  
  
/**  
 * RecentComments is a Yii widget used to display a list of recent comments  
 */  
  
class RecentComments extends CWidget  
{  
  
    private $_comments;  
  
    public $displayLimit = 5;  
  
    public $projectId = null;  
  
    public function init()
```



```

{

$this->_comments = Comment::model()

    ->findRecentComments($this->displayLimit, $this->projectId);

}

publicfunctiongetRecentComments()

{

return$this->_comments;

}

publicfunctionrun()

{

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

$this->render('recentComments');

}

}

```

主要工作是添加一个新的挂件来覆盖这个基类的 **init()** 和 **run()** 方法。**init()** 方法初始化挂件并且在它的属性被初始化以后调用，**Run()** 方法执行这个挂件。在这种情况下，我们通过调用基于 **\$displayLimit** and **\$projectId properties** 的最新内容初始化这个挂件。**widget** 的执行本身只是呈现其相关的视图文件，这是我们还没有创建一些东西。视图文件，从习惯上是直接放入和挂件同一个文件夹的视图文件夹里，有着和挂件相同的名字，但是但以小写字母开始。按照惯例，创建一个新的文件，路径为 **protected/components/views/renderComments.php**。一旦创建，在该文件里添加如下标记：

PHP 代码：

```

<ul>

<?phpforeach($this->getRecentComments()as$comment): ?>

<divclass="author">

<?phpecho$comment->author->username; ?>addedacomment.

```

```

</div>

<div class="issue">

<?php echo CHtml::link(CHtml::encode($comment->issue->name),
array('issue/view', 'id'=>$comment->issue->id)); ?>

</div>

<?php endforeach; ?>

</ul>

```

这次调用 **RenderComments** 挂件的 **getRecentComments()** 方法是返回内容数组，并用叠加的方式展示了谁添加了评论和在评论上留下了相关问题。

为了按照顺序查看结果，我们需要在一个已有的控制器视图文件里嵌入这个挂件。如前所述，我们想要在项目列表里使用这个挂件，以显示所有项目的最近评论，仍然需要一个特定的项目的详细信息页面，以此为最近的评论显示特定的项目。

让我们开始完成项目列表页面吧。负责展示内容的视图文件是 **protected/views/project/index.php**。打开这个文件在底部添加以下内容：

PHP 代码：

```

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

```

现在如果我们查看这个项目的列表页 <http://localhost/trackstar/index.php?r=project>，我们会看到类似以下内容的截图：

我们现在只是通过调用挂件在页面内嵌入我们的新的评论，这很好，但是我们可以把小挂件更进一步用来显示并在一个与所有在这个应用里其它的 **portlets** 保持一致。我们可以通过 **Yii** 的另一个类的优势来提供给我们，这就是 **CPortlet**。

CPortlet 简介

CPortlet 是 **zii** 的一部分，是将官方的扩展类库打包放在 **Yii** 里。它给所有的 **portlet** 风格的挂件提供了一个优秀的基类。它允许我们渲染一个好的标题和保持一致的 **HTML** 标记，因此所有的挂件可以非常轻松的以同样的方式加入到应用程序中。一旦我们有一个渲染内容的挂件（就像 **RecentComments**），我们可以简单地使用所提供挂件作为 **CPortlet** 的一部分，**CPortlet** 本身就是一个挂件，因为它也是扩展自 **CWidget**。我们可以通过把 **RecentComments** 挂件的调用放置在 **CPortlet** 的 **beginWidget()** 和一个 **endWidget()** 之间，就像下面这样：

PHP 代码:

```
<?php$this->beginWidget('zii.widgets.CPortlet', array(  
  
    'title'=>'Recent Comments',  
  
));  
  
$this->widget('RecentComments');  
  
$this->endWidget(); ?>
```

由于 **CPortlet** 提供了一个标题属性，我们将其设置为对我们的 **portlet** 有用的东西。然后我们使用 **RecentComments** 挂件已渲染的内容来填充到 **portlet** 挂件里。最后的结果如下截图:

这不是自从我们做过什么以前一个巨大变化，但我们已经把内容放到了一个一致的已经被应用于整个站点的容器里。注意右边字段菜单内容块和我们最近创建的内容之间的相似处。我确定这将不会让你感到惊奇，右边菜单块也是包含在 **CPortlet** 容器内的。简单看一下 **protected/views/layouts/column2.php**，这是一个我们最初创建应用时通过 **yiic** 命令自动生成的，留意下面的代码:

PHP 代码:

```
<?php  
  
$this->beginWidget('zii.widgets.CPortlet', array(  
  
    'title'=>'Operations', ));  
  
$this->widget('zii.widgets.CMenu', array(  
  
    'items'=>$this->menu,  
  
    'htmlOptions'=>array('class'=>'operations'),  
  
));  
  
$this->endWidget();  
  
?>
```

如此看来，应用已经利用了 **portlets**。

在另一个页面加入我们的挂件

我们同样的加入 **portlet** 到项目详情页面，并且对那些和项目相关联的内容进行约束。

在 **protected/views/project/view.php** 的底部加入下列代码：

PHP 代码：

```
<?php$this->beginWidget('zii.widgets.CPortlet', array(
    'title'=>'Recent Project Comments',
));
$this->widget('RecentComments', array('projectId'=>$model->id));
$this->endWidget(); ?>
```

还要在项目列表页面加入一些基本相同的东西，除了我们通过添加一个 **name=>value** 的数组对的调用来初始化这个 **RecentComments** 挂件的 **\$projectId** 属性。

现在我们浏览特殊页面的详情，我们将看到类似下面这个截图：

这个截图显示了 **#3** 的项目，其中有一个相关的详细信息页面只有一个问题，关于这个问题的评论所描述的画面。您可能需要添加对这些问题的一些留言以产生类似的效果。我们现在可以在任何地方显示最近的一些评论，使用极少的配置参数易维护的方式。

小结

有了这个迭代，现在我们已经开始使用各种功能让 **Trackstar** 应用功能变得充实，已经达到今天大多数用户的期望。用户之间彼此互相联系的功能对于一个成功的问题管理系统是一个必不可少的组成部分。

当我们创造了这个必要的功能，我们能够深入研究如何写 **AR** 的关系查询。我们还介绍了了一个名为内容挂件的部件和 **Portlet**。这个方法可以让我们用来开发小内容块，并有可能在网站的任何地方使用它们。这个方法大大提高可重用性和一致性，并且便于维护。

在接下来的迭代中，我们将在这里创建基于最近的评论的挂件用来展现内容通过像 **RSS** 一样来产生，以此让用户来跟踪应用程序或项目的活跃情况而无需访问应用程序。

第十章：迭代 7：添加 RSS Web Feed

在上次的迭代中，我们添加了用户对问题(**issue**)进行评论的功能及利用 **portlet** 结构轻松整合列表显示在应用程序中的任何位置。在本次迭代中，我们将利用这一点，实现一个 **RSS** 数据 **feed** 的评论列表。此外，

我们将使用另一个开源框架 **Zend Framework** 中现有的可用的 **feed** 功能。以证明在 **Yii** 应用中整合第三方工具是多么的容易。

迭代计划

本次迭代的目标是使用用户生成的评论内容创建一个 **RSS feed**。我们应该允许用户订阅所有项目 (**project**) 及单独项目的评论 **feed**。幸运的是，我们之前使用挂件功能返回了所有项目最近评论列表或限制为指定的一个项目的数据。所以，我们已经编写了适当的方法来访问所需的数据。这次迭代大部分精力将放在对这些数据以正确 **RSS feed** 格式进行发布，并添加链接允许订阅的用户访问我们的应用程序。

为了实现这些目标，我们将需要完成以下更高级的任务列表：

- § 下载并安装 **Zend Framework** 到 **Yii** 应用程序中。
- § 在一个控制器类中创建一个新的操作，用来响应 **feed** 请求并返回 **RSS** 格式的数据。
- § 改变易于使用的 **URL** 结构。
- § 添加我们新创建的 **feed** 到两个项目 (**project**) 的列表页和每个项目的详细页面

与往常一样，在作出任何更改之前，一定要先运行单元测试套件来保证预期的工作相符。

关于 RSS 内容聚合及 Zend Framework 的背景

Web 的内容聚合已经出现很多久了，但最近才比较流行。**Web** 内容聚合指的是一种发布信息的标准格式，通过这种格式其他网站的读取可以很容易地访问应用程序。许多新闻网站一直以这种格式发布它们的内容，从互联网上网络日志（也称博客）提供内容聚合到几乎每一个网站都有这个功能。当然我们的 **TrackStar** 应用程序也不会例外。

RSS 是一个缩写，全称 **Readily Simple Syndication**。它是一种 **XML** 格式文件用来提供一个 **Web** 内容聚合的标准。还有其他的格式可以使用，但由于绝大多数的网站都使用 **RSS**，我们将着重介绍这种格式。

Zend 是一家“PHP 公司”。其创始人是 **PHP** 语言的核心贡献者，**Zend** 致力于创造新产品用来帮助开发人员改善 **PHP** 应用程序的整个开发生命周期。他们提供的产品和服务会帮助你配置和安装 **PHP**，以及应用程序的开发，部署，生产管理和维护。其中一款有助于应用程序开的产品是 **Zend Framework**。该框架可以作为一个整体来提供一个完整的应用程序，比如可以使用同样的方式开发 **Trackstar** 应用程序，或者使用单独的功能组件库。**Yii** 的灵活之处是让我们可以使用其他的框架组件。我们将使用 **Zend Framework** 中的一个组件，它是 **Zend_Feed**。使用它可以让我们不用写任何低层的代码就能生成一个 **RSS** 格式的 **Web feed**。关于更多的 **Zend_Feed** 信息，请访问

<http://www.zendframework.com/manual/en/zend.feed.html>

安装 Zend 框架

由于我们使用 **Zend** 框架的帮助完成我们的 **RSS** 需求，我们首先需要下载和安装它。要获取最新版本，请访问 <http://framework.zend.com/download/latest>。我们将只采单这个框架的一部份 **Zend_Feed**，所以 **Zend** 框架的最小版本就足够了。

当你展开下载文件后，你应该看到以下文件及目录结构：

SHELL 代码或屏幕回显：

```
INSTALL.txt  
  
LICENSE.txt  
  
README.txt  
  
bin/  
  
library/
```

为了在 **Yii** 应用程序中使用这个框架，我们需要将一些文件移到我们的应用程序的目录结构中。让我们在 **/protected** 目录下创建一个新的目录叫 **vendors** 。然后移动 **Zend** 框架目录下的 **/library/Zend** 目录到新创建的目录下。移动完成了，确保在 **TrackStar** 应用程序中存在 **protected/vendors/Zend/Feed.php** 文件。

使用 Zend_Feed

Zend_Feed 是 **Zend** 框架中的一个小组件，它封装了创建 **Web Feed** 背后复杂的过程，提供了简单易用的接口，这将有助于我们在很短的时间就可以完成这项功能和测试 **RSS** 兼容数据。我们所需要做的是通过 **Zend_Feed** 来格式化我们的评论数据。

我们需要一个地方放置我们需要处理的 **feed** 的请求的位置。我们可以创建一个新的控制器，但是为了简便起见，我们只需要在 **CommentController.php** 文件中添加一个新的操作来处理请求。现在我们只为此方法添加一些功能，然后我们再讨论整个方法。

打开 **CommentController.php** 文件，并添加如下公有方法：

PHP 代码：

```
public function actionFeed()  
{  
  
    if(isset($_GET['pid'])) $projectId = intval($_GET['pid']);  
  
    else $projectId = null;  
  
  
    $comments = Comment::model()->findRecentComments(20, $projectId);  
}
```

```

        //convert from an array of comment AR class instances to an name=>value array for Zend

        $entries=array();

        foreach($comments as $comment)

        {

            $entries[]=array(

                'title'=>$comment->issue->name,

                'link'=>CHtml::encode($this->createAbsoluteUrl('issue/view', array('id'=>$comment->issue->id)
                )),

                'description'=> $comment->author->username . ' says: ' .

                    $comment->content, 'lastUpdate'=>strtotime($comment->create_time),

                'author'=>$comment->author->username,

            );

        }

        //now use the Zend Feed class to generate the Feed

        // generate and render RSS feed

        $feed=Zend_Feed::importArray(array(

            'title'=> 'Trackstar Project Comments Feed' ,

            'link'=> $this->createUrl(''),

            'charset' => 'UTF-8' ,

            'entries' => $entries,

        ), 'rss');

        $feed->send();

```

```
}
```

这一切都非常简单。首先，我们检查输入请求的 **pid** 参数是否存在，这里我们所采取的是一个具体的项目 ID。记住这里我们希望有选择地让数据的内容限制在一个指定的项目的相关评论。接下来，我们用 **findRecentComments** 的方法获得 20 条最近评论列表，这里如果指定项目 ID，则是具体的项目的评论，否则是所有的项目的评论。

你可能还记得，这个方法返回一个 **Comment AR** 类实例的数组。我们遍历这个数，通过 **Zend_Feed** 组件来返回转换成所需格式的数据。**Zend_Feed** 期望的是简单的数组，其本身的每个元素是一个包含评论实体的数组。每个实体是以一个简单的 **name=>value** 对的关联数组。为了符合具体的 **RSS** 格式，我们的每个条目必须至少包含一个 **title**(标题)，**link**(链接)和 **description**(描述)。我们还增加了两个可选的字段，一个名为 **lastUpdate**(最后更新)，这是 **Zend_Feed** 转换为 **RSS** 格式的发布日期，另一个指定为 **author**(作者)。

有一些额外的辅助方法，让我们可以获取正确的数据格式。首先，我们使用控制器的 **createAbsoluteUrl()** 方法，而不仅仅是 **createUrl()** 方法，以创建一个完整的 URL，使用 **createAbsoluteUrl()** 将产生类似下面的链接：<http://localhost/trackstar/index.php?r=issue/view&id=5> 相对于 </index.php?r=issue/view&id=5>。

同时，为了避免出现“**unterminated entity reference**”错误，使用 PHP 的 **DOMDocument::createElement()**，它是由 **Zend_Feed** 用来生成 **RSS** 格式的 **XML**，我们需要使用辅助函数如：**CHTML::encode** 来转换所有使用了 **HTML** 实体的字符。所以我们进行编码这样的链接的 URL 看起来像这样：

<http://localhost/trackstar/index.php?r=issue/view&id=5>

将转换为：

<http://localhost/trackstar/index.php?r=issue/view&id=5>

一旦我们的所有实体都已经被正确填充和格式化，我们可以使用 **Zend_Feed** 的 **importArray()** 方法，它期望一个数据来构造 **RSS** 内容。最后，一旦 **Zend_Feed** 类从输入的数组返回，我们将调用 **Zend_Feed** 类的 **send()** 方法。这将返回正确的 **RSS** 格式的 **XML** 和发送给客户端的 **XML** 文件头。

我们需要修改些配置，在文件 **CommentController.php** 中的类前添加代码。首先我们需要导入 **/vendors/Zend/Feed.php** 文件以及 **Feed/** 目录下的 **Rss.php** 文件。在 **CommentController.php** 文件的顶部添加以下代码：

PHP 代码：

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



```
require_once('Zend/Feed/Rss.php');
```

然后，修改 `CommentController::accessRules()` 方法，允许任何用户访问我们新增方法 `actionFeed()`：

PHP 代码：

```
public function accessRules()
{
    return array(
        array('allow', // allow all users to perform 'index' and 'view' actions
            'actions' => array('index', 'view', 'feed'),
            'users' => array('*'),
        ),
        ...
    );
}
```

这就是全部内容。如果我们访问 <http://localhost/trackstar/index.php?r=comment/feed>，我们能看我们努力的结果。不同的浏览器将显示不同的 RSS 结果，你所看到的可能与截图不同。下面的截图是使用 Firefox 浏览器查看 RSS 的结果。



创建友好的 URL

到目前为止，整个开发过程中，我们使用了 yii 应用程序默认格式的 URL 结构，在第 2 章，我们讨论了这种格式，它采用了查询参数的方式。主要参数 `r` 代表路由，是由 `controllerID/actionID` 对组成，其次是 `action` 方法所需要的一些可选的查询参数。我们新创建的 `feed` 操作也不例外，但这是一个很长，繁琐且不友好的 URL 结构。有更好的方法吗？嗯，事实上是有的。

使用路径格式，我们可以让上面的 URL 看起来更清晰，更明了。消除 URL 中的查询参数字符串，把查询参数放到路径信息中：

以我们的评论 `feed` URL 为例，替换：

<http://localhost/trackstar/index.php?r=comment/feed>

为我们需要的：

<http://localhost/trackstar/index.php/comment/feed/>

更多情况，我们甚至不需要指定入口文件。我们也可以利用 Yii 的路由配置选项删除指定的 `controllerID/actionID` 对。我们的要求将像这样：

<http://localhost/trackstar/commentfeed>

还有，普遍的 `feed` URL 最后都有一个 `.xml` 扩展名。所以，如果我们可以改成下面的这样那就更好了：

<http://localhost/trackstar/commentfeed.xml>

这样将大大简化了 URL 地址，并且也符合各大搜索引擎的收录格式（通常被称为“友好的搜索引擎 URL”）。让我们看看如何使用 Yii 的 URL 管理功能，改变我们的 URL 以符合我们需要的格式。

使用 URL 管理器

在 Yii 中内置的 URL 管理器是一个应用程序组件，它可以在 `protected/config/main.php` 文件中配置。让我们打开这个文件，并在组件数组中添加一个新的 URL 管理器组件的声明：

PHP 代码：

```
'url Manager' => array(  
  
    'url Format' => 'path',  
  
),
```

只要我们使用默认的名字 `urlManager`，我们不需要指定组件的类名，因为它已经在 `CWebApplication.php` 框架类中预定义为 `CUrlManager.php`。

通过简单的添加配置，整个站点的 URL 结构变成了 `'path'` 格式。例如，以前如果我们想查看一个具体的 ID 为 1 的 `issue`，我们会使用下面的 URL：

<http://localhost/trackstar/index.php?r=issue/view&id=1>

但有了这些变化，我们的网址现在看起来像这样：

<http://localhost/trackstar/index.php/issue/view/id/1>

你将会注意到，我们所作出的修改已经影响了整个应用程序。注意到这点，再次访问 **feed** 将使用 <http://localhost/trackstar/index.php/comment/feed/>。我们注意到，所有 **issue** 的联系都被重新格化成了新的 **URL** 结构。这一切都要感谢控制器方法和助手方法来帮助我们生成 **URL**。我们仅修改了配置文件中的 **URL** 格式，整个应用程序都会更改。

我们的 **URL** 看起来很好，但我们仍然有一个指定的入口文件 **index.php**。并且我们还没有追加一个 **.xml** 的后缀。因此，我们将隐藏 **URL** 中 **index.php** 的部份，并且设置请求路由，让它明白请求 **commentfeed.xml** 就是请求 **CommentController.php** 类中的 **actionFeed()** 方法。让我们先解决后者。

配置路由规则

Yii 的 **URL** 管理器允许我们指定规则来定义 **URL** 的解析和创建。定义一个路由规则和模式，该模式用于匹配 **URL**，以确定哪些规则用于解析或创建 **URL**。该模式可以包含命名参数，它使用的语法为 **ParamName:RegExp**（参数名:正则表达式）。当解析一个 **URL**，一个匹配的规则将提取路径信息中的命名的参数放到 **\$_GET** 变量中。当通过应用程序创建一个 **URL** 时，一个匹配的规则将从 **\$_GET** 变量中提取命名的参数并放到路径信息中。如果模式以 **/*** 结尾，它的意思有更多的 **GET** 参数附加到 **URL** 路径信息的中。

指定 **URL** 规则，设置 **CUrlManager** 的 **rules** 属性，**rules** 是一个数组，其中的格式为 **pattern=>route**（模式=>路由）。

举个例子，让我们看看下面两条规则：

PHP 代码：

```
'urlManager' => array(  
    'urlFormat' => 'path',  
    'rules' => array(  
        'issues' => 'issue/index',  
        'issue/<id:\d+>/' => 'issue/view',  
    )  
)
```

上面代码中指定了两条规则。第一条规则说明，如果用户请求的 **URL** 是 <http://localhost/trackstar/index.php/issues>，它应被视为 <http://localhost/trackstar/index.php/issue/index>，并且这条规则也同样适用于创建 **URL**。

第二条规则使用了语法包含了一个命名参数 **ID**，它的意思，例如，如果用户请求的 **URL** 为 <http://localhost/trackstar/index.php/issue/1>，它应被视为 <http://localhost/trackstar/index.php/issue/view?id=1>。并且这条规则也同样适用于创建 **URL**。

路由部分本身也可以指定为一个数组，如指定 **URL** 后缀或是否区分大小写等其它属性。我们将利用这些优势，指定我们的评论 **feed** 规则。

让我们将下列规则添加到我们的 **urlManager** 组件配置中：

PHP 代码：

```
'url Manager' => array(  
  
    'url Format' => 'path',  
  
    'rules' => array(  
  
        'commentfeed' => array('comment/feed', 'url Suffix' => '.xml', 'caseSensitive' => false),  
  
    ),  
  
),
```

在这里，我们使用 **urlSuffix** 属性指定了我们想要的 **.xml** 作为后缀。

现在，我们可以通过下面的 **URL** 访问 **feed**：

<http://localhost/trackstar/index.php/commentFeed.xml>

从 URL 中移除入口文件

现在我们需要从 **URL** 中移除 **index.php**。需要两步完成：

§ 修改 **Web** 服务器配置，重定向所有请求的路由到 **index.php**，但不包括已经存在的文件或目录。

§ 设置 **UrlManager** 的 **showScriptName** 属性为 **false**。

第一步操作是告诉应用程序怎样处理路由请求，第二步操作是告诉应用程序如何创建 **URL**。

由于我们使用的是 **Apache HTTP Server**，第一步我们可以在应用程序根目录创建一个 **.htaccess** 文件，并加入以下指令：

SHELL 代码或屏幕回显：

```
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
```

这种方式只适用于 **Apache HTTP Server**。如果你使用的是不同的 **Web** 服务器，你将需要编写新的重写 (rewrite) 规则文件。另外请注意，这些信息可以放在 **Apache** 的配置文件中以替代 **.htaccess** 文件。

.htaccess 文件创建后，我们现在可以通过以下 **URL** 访问 **feed**，
<http://localhost/trackstar/commentfeed.xml> (或 <http://localhost/trackstar/commentFeed.xml> 因为我们设置了 **caseSensitive** 为 **false**，不区分大小写)

然而，即使这样，如果我们在应用程序中使用一个控制器方法或 **CHTML** 助手方法创建 **URL**，在一个控制器类中执行下面是代码：

PHP 代码：

```
$this->createAbsoluteUrl('comment/feed');
```

它将生成的 **URL** 仍然包括 **index.php**： <http://localhost/trackstar/index.php/commentfeed.xml>

当生成 **URL** 时为了不使用入口文件，我们需要设置 **urlManager** 组件的属性。我们再来修改 **main.php** 配置文件，例如：

PHP 代码：

```
'urlManager' => array(

    'urlFormat' => 'path',

    'rules' => array(

        'commentfeed' => array('site/commentFeed', 'urlSuffix' => '.xml',
        'caseSensitive' => false),

    ),

    'showScriptName' => false,
```

```
),
```

在 URL 中为了处理指定 **project ID** 的评论 **feed**，我们需要添加另外一个规则：

PHP 代码：

```
'url Manager' => array(

    'url Format' => 'path',

    'rules' => array(

        '<pi d: \d+>/commentfeed' => array('site/ commentFeed', 'url Suffix' => '.xml',
        'caseSensi ti ve' => fal se),

        'commentfeed' => array('site/commentFeed', 'url Suffix' => '.xml',
        'caseSensi ti ve' => fal se),

    ),

    'showScriptName' => fal se,

),
```

这条规则同样也使用了语法指定模式，URL 的 **commentfeed.xml** 部分之前指定了一个 **project ID**。通过这条规则，我们可以限制评论的 **feed** 指定到一个具体的 **project**。例如，如果我们只想要 **project #2** 的评论的 **feed**，URL 格式是：<http://localhost/trackstar/2/commentfeed.xml>

添加 feed 链接

现在我们已经创建了更加友好的 **feed** 的 URL 结构。我们需要添加用户可订阅 **feed** 的功能。一种方法是增加以下代码，在页面渲染前添加 **RSS feed** 链接。让我们对项目列表页和具体的项目详细页进行修改。我们首先开始制作项目列表页。这个页面的操作方法是 **ProjectController::actionIndex()**，修改此方法为如下代码：

PHP 代码：

```
public function actionIndex()

{

    $dataProvider=new CActiveDataProvider('Project');

    Yii::app()->clientScript->registerLinkTag(    //这行

        'alternate',    //这行
```

```

        'application/rss+xml', //这行

        $this->createUrl('comment/feed')); //这行

        $this->render('index',

            array('dataProvider'=>$dataProvider,)

        );
    }
}

```

上述高亮代码将在再<head>部份渲染以下 HTML:

PHP 代码:

```

<link rel="alternate" type="application/rss+xml" href="/commentfeed.xml" />

```

在许多的浏览器中，地址栏上将自动生成一个 **RSS feed** 小图标。下面的截图描绘了这个图标像在 **Firefox3.6** 地址栏中的外观:



我们将类似的修改添加到具体的项目详细页，这个页面的操作方法是 **ProjectController::actionView()**，修改此方法为如下代码:

PHP 代码:

```

public function actionView()
{
    $issueDataProvider=new CActiveDataProvider('Issue', array(

        'criteria'=>array(

            'condition'=>'project_id=:projectId',

            'params'=>array(':projectId'=>$this->loadModel()->id),

        ),

        'pagination'=>array(

```

```

        'pageSi ze' =>1,

    ),

));

Yii::app()->cli entScript->regi sterLi nkTag(

    'al ternate',

    'appl i cati on/rss+xml ',

    $thi s->createUrl (' comment/feed' , array(' pi d' =>$thi s->l oadModel ()->i d));

    $thi s->render(' vi ew' , array(

        'model ' =>$thi s->l oadModel (),

        'i ssueDataProvi der' =>$i ssueDataProvi der,

    ));

}

```

除了指定 **project ID**，几乎与添加到 **index** 方法中的代码一样。这将使评论的 **feed** 仅限制在指定项目。现在我们在项目详细页面的地址栏中也显示了 **feed** 订阅图标，点击图标将允许用户订阅这些评论。

小结

本次迭代证明在 **Yii** 框架中融入其他外部框架是多么的容易。我们特地使用了流行的 **Zend Fraemwork** 快速添加网站的 **RSS feed** 符合我们的应用程序的要求来证明这一点。虽然我们仅指定使用了 **Zend_Feed**，但我们真正展示的是如何将 **Zend Framework** 的任何组件融入到我们的应用程序中。这进一步扩大了 **Yii** 的功能，使 **Yii** 应用程序实现更丰富的功能。

我们也了解了使用 **Yii** 中的 **URL** 管理功能来改变整个应用程序的 **URL** 为更加友好的格式。这是提升应用程序外观和感觉的第一步。我们有太多的东西被忽视。在下一迭代中，我们将重点转向风格，主题。

第十一章：迭代 8：美化—设计，布局，主题，国际化(i18N)

在之前的迭代中，我们开始着手美化我们的项目，我们的 **URLs** 开始对用户和爬过我们网站的搜索引擎变得更有魅力（可能指更友好）。在本次迭代中，我们将更多的注意力投向 **Yii** 中页面布局和主题所带来的感观和感觉上。因此我们实现本章将通过将我们应用程序的外观改变的更好看来实现本章的主旨，我们将关注一种可以采取的方法和一种可以用来帮助设计 **Yii** 应用程序前台的工具，而不是自己重新设计。所以本次迭代中将更多的关注如何使你的应用程序更漂亮，而不是花费大量时间去专门设计我们 **TrackStar** 应用程序(的外观)。

迭代计划

这次迭代是基于前端来完成的。我们想制作一个可以被动态重用的网站的新外观。我们并不想通过重写或删除当前设计的方式来实现。同时，我们也会深入 **Yii** 的 **i18N(internationalization = i18N)** 功能，所以我们必须清楚的明白如何使用应用程序适应来自不同地理位置的用户。

下面是一个 **high-level(高级/高优先级)**任务列表，我们需要完成它以实现我们的目标：

- § 通过创建新的布局，**CSS** 和其他配置文件来建立一个新的主题，并为应用程序提供一个新前端设计
- § 使用 **Yii** 的 **i18N** 和定位功能将我们应用程序中的一部分翻译成另外一种新语言

设计布局

你可能注意到的一件事是：我们已经添加了很多功能到我们的应用程序，但没有添加任何实质的导航来连接到这些功能。自默认建立的应用程序开始首页没有发生任何改变。自我们建立应用程序开始，导航项没有发生改变。我们需要改变基本导航来更好的反映现在应用程序中的各个功能。

至此，我们并未完全提及我们的应用程序是如何调用用来呈现内容的 **view** 文件的。我们知道 **view** 文件是用来显示数据和每次页面请求的 **HTML** 回发内容的。当我们建立新 **Controller(控制器)**的 **actions(行为)** 时，我们也会建立相关的 **view** 文件，来承载 **actions** 返回的显示内容。大多数 **view** 文件只针对单一 **action** 存在，没有通用性。然而，有一些，例如主导航菜单，会在网站中很多页面被重复使用。这类 **UI** 组件更适合存放置被我们称为布局文件的文件中。

Yii 中的布局文件，是一种用来装饰其他 **view** 文件的一种特殊的 **view** 文件。一般来说布局会包含修饰部分或者包含其他 **view** 文件的用户界面元素。在使用布局文件渲染一个 **view** 文件时，**Yii** 会将该 **view** 文件装入布局。

定制一个布局

定制一个布局主要分为 **2** 步操作。其一是 **CWebApplication** 的 **\$layout** 属性。默认值指向 **protected/views/layouts/main.php**。这一项为默认设置，可以在位于 **protected/config/main.php** 的主配置文件里进行修改。例如我们在 **protected/views/layouts/newlayout.php** 新建了一个配置文件，并且想将其设置为应用程序级布局文件，我们需要按如下方式修改主配置文件修改 **layout** 属性：

PHP 代码：

```
return array(  
  
    'layout' => 'newlayout',
```

文件名并不包含.php 后缀，并且将 **CWebApplication** 的 **\$layoutPath** 属性默认关联到 **Webroot/protected/views/layouts**(如果你的应用程序路径有所改变，可以使用相同的方式重写该值)。

另外一个需要指定布局的地方是设置 **Controller(控制器)** 类 **\$layout** 属性。这样就允许针对不同 **Controller(控制器)** 以更细的粒度进行设置。这就是在建立最初程序时定制的方法。当使用 **yiic** 工具来建立最初应用程序时，基于 **Webroot/protected/components/Controller.php** 来自动创建一个 **Controller(控制器)**。打开该文件你将看到 **\$layout** 属性被设置为“column1”。在 **Controller(控制器)** 级对布局文件的设置将覆盖 **CWebApplication** 类中的设置。

添加并使用一种布局

在调用 **CController::render()** 方法来使用一个布局文件。也就是说，当你调用 **render()** 方法来渲染一个 **view** 文件时，**Yii** 将会将 **view** 文件的内容嵌入 **controller** 类或应用程序级设置的 **layout** 文件中。你可以通过调用 **CController::renderPartial()** 方法实例来避免应用 **layout** 装饰。

正如前面所说，一个布局文件一般被用来装饰其他 **view** 文件。一个使用布局的例子就是每个页面提供相同的页头和页脚。当 **render()** 方法被调用，后台先针对某个 **view** 文件调用 **renderPartial()**。这一步的输出被保存在一个叫 **\$content** 的变量中，并且在接下来的布局文件中可用。因此，一个简单的布局文件可能是下面的样子：

PHP 代码：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"  
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">  
  
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">  
  
    <head>  
  
        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />  
  
        <meta name="language" content="en" />  
  
    </head>  
  
    <body>  
  
        <div id="header">Some Header Content Here </div>  
  
        <?php echo $content; ?>  
  
        <div id="footer"> Some Footer Content Here</div>
```

```
</body>

</html>
```

让我们动手验证一下。新建一个叫 **newlayout.php** 的文件，并将其放入默认布局文件位置：**/protected/views/layouts/**。将上面的 HTML 添加进去后保存。下面我们将通过修改我们的站点 **controller(控制器)** 来使用这个新布局。打开 **SiteController.php** 然后重载 **layout** 属性设置，添加如下代码：

PHP 代码：

```
class SiteController extends Controller

{

    public $layout='newlayout';
```

这样只有这一个 **controller(控制器)** 会使用 **newlayout.php**。至此，每一次基于 **SiteController** 调用 **render()** 方法都会使用 **newlayout.php** 布局文件。

登录页面就是由 **SiteController** 负责渲染的页面之一。让我们访问一次该页面来确认修改。当我们打开 **http://localhost/trackstar/site/login** (假设我们并未登录)，我们将会看到与下图相关的界面：



The screenshot shows a web form for logging in. At the top, it says "Some Header Content Here". Below that is the title "Login" in a large, bold font. The main text reads "Please fill out the following form with your login credentials:". Below this is a note "Fields with * are required.". There are two input fields: "Username *" and "Password *". Below the input fields is a hint: "Hint: You may login with demo/demo or admin/admin.". There is a checkbox labeled "Remember me next time". Below the checkbox is a "Login" button. At the bottom, it says "Some Footer Content Here".

如果我们简单的注释掉我们之前添加的 **\$layout** 属性，然后刷新登录页面，我们将重新使用原始的 **main.php** 布局，并且页面也变回原来的样子了。

重构 main.php 布局文件

目前为止，我们所有的应用程序页面都使用了 **main.php** 布局文件作为主要的布局标记。在我们着手改动页面布局和设计之前，应该仔细观察一下 **main** 布局文件。下面是该文件的全部内容：

PHP 代码:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">

    <head>

        <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

        <meta name="language" content="en" />

        <!-- blueprint CSS framework -->

        <link rel="stylesheet" type="text/css" href="<?php echo
Yii::app()->request->baseUrl; ?>/css/screen.css"

            media="screen, projection" />

        <link rel="stylesheet" type="text/css" href="<?php echo
Yii::app()->request->baseUrl; ?>/css/print.css"

            media="print" />

        <!--[if lt IE 8]>

            <link rel="stylesheet" type="text/css" href="<?php echo
Yii::app()->request->baseUrl; ?>/css/ie.css"

                media="screen, projection" />

        <![endif]-->

        <link rel="stylesheet" type="text/css" href="<?php echo
Yii::app()->request->baseUrl; ?>/css/main.css" />

        <link rel="stylesheet" type="text/css" href="<?php echo
Yii::app()->request->baseUrl; ?>/css/form.css" />

        <title><?php echo CHtml::encode($this->pageTitle); ?></title>
```

```
</head>

<body>

    <div class="container" id="page">

        <div id="header">

            <div id="logo"><?php echo CHtml::encode(Yii::app()->name); ?></div>

        </div>

        <!-- header -->

        <div id="mainmenu">

            <?php $this->widget('zii.widgets.CMenu',array(

                'items'=>array(

                    array('label'=>'Home', 'url'=>array('/site/index')),

                    array('label'=>'About', 'url'=>array('/site/page','view'=>'about')),

                    array('label'=>'Contact', 'url'=>array('/site/contact')),

                    array('label'=>'Login', 'url'=>array('/site/login'),

                        'visible'=>Yii::app()->user->isGuest),

                    array('label'=>'Logout ('.Yii::app()->user->name.')',

                        'url'=>array('/site/logout'),

                        'visible'=>!(Yii::app()->user->isGuest) ),

                ),

            ); ?>

        </div>

        <!-- mainmenu -->
```

```

        <?php $this->widget('zii.widgets.CBreadcrumbs',

            array( 'links' => $this->breadcrumbs,

        )); ?>

    <!-- breadcrumbs -->

    <?php echo $content; ?>

    <div id="footer"> Copyright &copy; <?php echo date('Y'); ?> by My Company. <br/>

        All Rights Reserved. <br/> <?php echo Yii::powered(); ?>

    </div><!-- footer -->

</div><!-- page -->

</body>

</html>

```

我们将从头开始。最开始的 5 行可能会让你觉得很熟悉。

PHP 代码:

```

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"

"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en"> <head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<meta name="language" content="en" />

```

这几行定义了标准的 **HTML** 文件类型申明，接着一个元素开始，接着是元素开始。在标签内，我们首先使用一个 **meta** 标签来定义标准，并且这是非常重要的，**XHTML** 编译器使用 **UTF-8** 进行编码，接下来的一个标签申明英文为该网页主要书写语言。

介绍 Blueprint CSS 框架

接下来以注释开始的几行可能也对你来说很熟悉。**Yii** 的另一个大优势就是其适当利用了其他拥有优良血统的框架，**Blueprint CSS** 框架就是其中一个例子。

Blueprint CSS 框架在 **yiic** 工具初始化应用程序时就被集成到应用程序中。它被用来帮助标准化的 **CSS** 开发。**Blueprint** 是一个 **CSS** 网格框架。它被用来标准化你的 **CSS**，提供跨浏览器兼容，并且使用标准化 **HTML** 元素来减少 **CSS** 错误。它拥有基于不同屏幕并且面对打印友好的布局，并且帮助快速实现你的设计，通过它提供的一些你需要的 **css** 来帮助 **css** 快速实现。获取更多 **Blueprint framework** 信息，浏览：<http://www.blueprintcss.org/>。

所以，下面几行代码为 **Blueprint CSS** 框架提供所需支持：

PHP 代码：

```
<!-- blueprint CSS framework -->

<link rel="stylesheet" type="text/css" href="<?php echo Yii::app()-
>request->baseUrl; ?>/css/screen.css"

    media="screen, projection" />

<link rel="stylesheet" type="text/css" href="<?php echo Yii::app()-
>request->baseUrl; ?>/css/print.css"

    media="print" />

<!-- [if lt IE 8]>

<link rel="stylesheet" type="text/css" href="<?php echo Yii::app()-
>request->baseUrl; ?>/css/ie.css"

    media="screen, projection" />

<![endif]-->
```

理解 Blueprint 的安装

Yii 并没有对 **Blueprint** 的使用要求。然而，作为项目默认包含的框架，了解它的安装和使用会很有益。

一般安装 **Blueprint** 包含以下步骤：下载框架文件，将其中 3 个 **.css** 文件放入 **Yii** 应用程序的主 **CSS** 文件夹。如果我们小窥一下我们 **TrackStar** 应用程序的主 **css(Webroot/css)** 文件夹，我们将看到以下 3 个文件已经包含进去了：

```
$ ie.css
$ print.css
$ screen.css
```

因此，幸运的，在我们使用 **yiic webapp** 命令行生成应用程序时，基本安装已经完成了。为了使用这个框架的功能，上面的标签需要被放置在每一个页面的标签中。这也是这些申明被放置在布局文件的原因。

接下来的 2 个标签：

PHP 代码：

```
<link rel="stylesheet" type="text/css" href="<?php echo Yii::app()->request->baseUrl; ?>/css/main.css" /> <link rel="stylesheet" type="text/css" href="<?php echo Yii::app()->request->baseUrl; ?>/css/form.css" />
```

定义一些自定义的 **css** 用来提供一些额外的布局申明。你必须始终将你定义的 **css** 置于 **Blueprint** 提供的之下，这样你定义的才可以获得更好的优先级。

设置页面标题

设置不同的并且有意义的页面标题非常重要，可以帮助你的页面在搜索引擎中排序和那些打算收藏该页面到书签的用户。下面的一行定义了页面标题：

PHP 代码：

```
<title><?php echo CHtml::encode($this->pageTitle); ?></title>
```

请牢记 **view** 文件中的 **\$this** 指向渲染该 **view** 文件的 **controller**(控制器)类。**\$pageTitle** 属性被定义在 **Yii** 的 **CController** 基类中，其默认值为 **controller**(控制器)后的 **action**(动作)的名字。这个值很容易在 **controller** 类中重置，或者在 **view** 文件中重置。

定义一个页面的头部

一个惯例网站被设计拥有很多页面都相同的头部，下面的几行定义了主布局文件的头部内容：

PHP 代码：

```
<body>

    <div class="container" id="page">

        <div id="header">

            <div id="logo"><?php echo CHtml::encode(Yii::app()->name); ?></div>

        </div><!-- header -->
```

第一个标有“**container**”类的 **<div>** 标签是 **Blueprint** 框架所必须用来显示网格内容的。

再次申明，Yii 并未要求使用类似 **Blueprint CSS** 框架。只是为了帮助你快速实现你的设计布局。

接下来的三行代码，实现我们在那些页面中看到的顶部内容。它以大写字母的形式应用程序的名字。至此文字‘**My Web Application**’被显示出来了。到这里估计你有点要抓狂了。虽然我们可能会使用 **logo** 图片来做替换，但目前我们只将其替换为该项目的真实名字‘**TrackStar**’。

我们可以将其直接敲入 **HTML** 代码中。但，如果我们以修改应用程序配置的方式实现新名字，这个修改将传播至应用程序中任何一个使用 **Yii::app()->name** 被使用的地方。我确定这个修改对你来说非常容易。打开定义应用程序配置信息的主配置文件 **/protected/config/main.php** 并将‘**name**’属性从：

PHP 代码：

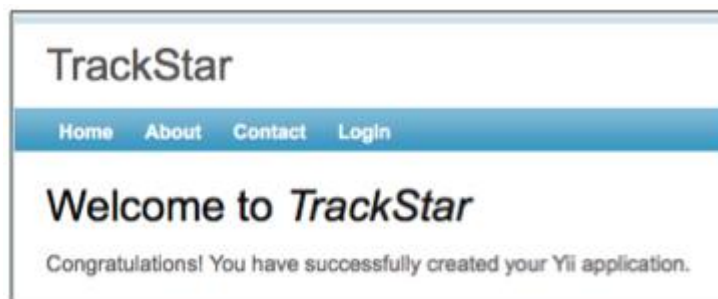
```
'name'=>'My Web Application',
```

改为：

PHP 代码：

```
'name'=>'TrackStar'
```

保存文件，刷新你的浏览器，首页头部应该如下图所示：



在上图中我们能立刻发现的是 **2** 处被修改了。这一现象只发生在首页内容中，**/protected/iews/site/index.php**，因为这里也使用了应用程序的 **name** 属性。因为我们在应用程序的配置文件进行了修改，它影响到了这 **2** 个地方。

显示导航菜单项

在 **web** 应用程序中站点导航经常多个页面中重复，因此在布局放置导航使之非常容易被重用。下面的代码实现了导航：

PHP 代码：

```
<div id="mainmenu">

<?php $this->widget('zii.widgets.CMenu',array(
```

```

        'items' => array(

            array('label' => 'Home', 'url' => array('/site/index')),

            array('label' => 'About', 'url' => array('/site/page', 'view' => 'about')),

            array('label' => 'Contact', 'url' => array('/site/contact')),

            array('label' => 'Login', 'url' => array('/site/login')),

            'visible' => Yii::app()->user->isGuest,

            array('label' => 'Logout (' . Yii::app()->user->name . ')',

                'url' => array('/site/logout'), 'visible' => !Yii::app()->user->isGuest

            ),

        ),

    )); ?>

</div><!-- mainmenu -->

```

在此处我们使用了官方 **Zii** 扩展: **CMenu**。在第 9 章我们介绍过 **Zii**。回忆一下, **Zii** 扩展库是 **Yii** 开发团队所开发的一系列扩展。它随着 **Yii** 框架的安装被安装。在 **Yii** 中使用该扩展是非常容易的, 只需要简单引用所需扩展类文件, 使用 `zii.path.to.ClassName` 这样的代理格式。根 `zii` 被应用程序预先定义, 其他路径是框架的相对文件夹路径。所以, 如上所示 **Zii** 菜单扩展位于你文件系统 **Path-to-your-Yii-Framework/zii/widgets/CMenu.php**, 在应用程序中可以通过 `zii.widgets.CMenu` 的方式来引用。

无需对 **CMenu** 有很多的了解, 我们会发现它使用一个关联数组, 为菜单项提供 **label** 和 **URL** 来形成一个连接, 还有一个布尔值的第三个可选项 **visible**, 用来决定菜单项是否显示。这里的 **'login'** 和 **'logout'** 项就是这样操作的。显而易见的, 我们只希望用户未登录时显示 **'login'** 菜单项。相反的, 只有用户登录后才显示 **'Logout'** 项。**visible** 元素在数组里的使用使得你可以基于用户登录状态动态的控制连接的显示。使用 `Yii::app()->user->isGuest` 到达此效果。以游客状态未登录时返回 **true**, 用户登录后返回 **false**。在你登录以后 **'login'** 项转换为 **'logout'**, 反之亦然。

让我们更新菜单来导航至特定的 **TrackStar** 功能。首先, 除了登录我们并不希望匿名用户获得其他实质功能。所以我们要将登录页面设置为匿名用户的首页。同理, 已登录用户的首页应该是他们的项目列表。我们将通过以下修改达到上述目标:

1. 改变项目列表为默认首页, 而不是现在的 **site/index** 页。
2. 设置默认 **controller**(控制器) **SiteController** 的默认 **action**(行为) 为 **login**。这样任何匿名用户访问顶级域名, **http://localhost/trackstar/**, 将被重定向至登录页。
3. 修改 **actionLogin()** 方法, 如果是已登录用户则重定向至项目列表页。
4. 改变菜单项 **'home'** 为 **'project'**, 并更改 **URL** 地址为项目列表页。

需要做的修改都非常简单。首先，我们修改应用程序配置文件中的`homeUrl`属性。打开 `protected/config/main.php` 添加下面的键值对到返回数组：

PHP 代码：

```
'homeUrl' => '/trackstar/project',
```

这就是修改的全部。

下一个修改，打开 `protected/controllers/SiteController.php`，添加下面一行代码到 `controller`(控制器)类的顶部：

PHP 代码：

```
public $defaultAction = 'login';
```

这样默认 `action` 就被修改为`login`了。现在如果你通过顶级 URL，<http://localhost/trackstar/>，访问你的应用程序，你将会看到登录页面。唯一的问题就是不管你是否登录，你看到的都只能是登录页面。我们将通过第三步修改来修正它。修改位于 `SiteController` 中的 `actionLogin()` 方法，在该方法的开头包含如下代码：

PHP 代码：

```
public function actionLogin()
{
    if(!Yii::app()->user->isGuest)
    {
        $this->redirect(Yii::app()->homeUrl);
    }
}
```

这将会把所有登录用户重定向至应用程序的 `homeUrl` 属性，也就是我们之前设置的项目列表页。

最后，让我们修改 `CMenu` 部件的输入数组中的`Home`菜单项。将如下代码：

PHP 代码：

```
array('label' => 'Home', 'url' => array('/site/index')),
```

修改为：

PHP 代码：

```
array('label' => 'Projects', 'url' => array('/project'))),
```

通过以上替换，我们之前提出的几处修改都已经完成了。如果现在以匿名用户身份访问 **TrackStar** 应用程序，我们将被重定向至登录页。点击 **Projects** 连接还是被重定向至登录页。以匿名用户的身份还是可以访问 **About** 和 **Contact** 页面的。登录后将被重定向至项目列表页。现在点击 **Projects** 连接，我们将能看到项目列表。

建立面包屑导航

回到 **main.php** 布局文件，在 **menu** 部件之后的 3 行代码定义了一个叫 **CBreadcrumbs** 的 **Zii** 扩展：

PHP 代码：

```
<?php $this->widget('zii.widgets.CBreadcrumbs', array(
    'links' => $this->breadcrumbs,
)); ?><!-- breadcrumbs -->
```

这个部件用来显示用来标识当前页面位置的一系列连接，这个位置相对与整个网站中的其他页面。例如，如下形式的连接导航列表：

Projects >> Project 1 > > Edit

表示用户正在浏览 **Edit** 页面，该页面属于 **Project1**。这可以很好的帮助用户回到出发的地方，这里是项目列表，同时也可以很好的看出网站页面之间的继承关系。这也是其被叫做面包屑的原因。很多网站都使用了这种 **UI** 导航元素。

为了使用该部件，我们需要配置它的 **links** 属性，也就是设置被显示的连接。这里的预期值为一个数组，该数组标识了从开始到当前页面的路径关系。利用上面的例子，我们可以如下定义连接数组：

PHP 代码：

```
array(
    'Projects' => array('project/index'),
    'Project 1' => array('project/view', 'id' => 1),
    'Edit',
)
```

面包屑部件会默认自动添加最顶级的 **'Home'** 连接，该连接的值基于应用程序配置中设置的 **homeUrl**。所以上面的面包屑会是如下的样子：

Home >> Projects >> Project 1 >> Edit

因为我们将应用程序的`$homeUrl` 属性设置为项目列表页，所以在这里前 2 个连接是相同的。在布局文件使用渲染该 `view` 文件 `controller` 的`$breadcrumbs` 属性来设置 `link` 属性。你会发现许多是哟个 `Gii` 代码生成工具自动生成的 `contorller` 的 `view` 文件都是这样设置的。列入，如果你打开 `protected/views/project/update.php`，你会在顶部发现如下代码：

PHP 代码：

```
<?php

$this->breadcrumbs=array(

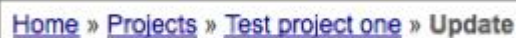
    'Projects'=>array('index'),

    $model->name=>array('view','id'=>$model->id),

    'Update',

);
```

如果我们导航至该页面，我们会在主导航下面看到如下的面包屑导航：



指定被布局所修饰的内容

布局文件中的下一行，就是显示被该布局文件修饰的 `view` 文件内容：

PHP 代码：

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

正如本章早些被提到的，当你在 `controller` 类使用`$this->render()`来显示一个 `view` 文件，布局文件也被隐式执行。这个方法执行的步骤就是，就是在布局文件中的`$content` 中显示对应 `view` 文件的内容。如果再拿项目更新试图文件举例，`$content` 的内容就应该是 `protected/views/project/update.php` 文件渲染的结果。

定义页脚

如同页头部分，一般来说网页都会在不同页面拥有相同的页脚，`main.php` 布局文件的最后几行实现了该功能：

PHP 代码：

```
<div id="footer">

    Copyright &copy; <?php echo date('Y'); ?> by My Company.<br/>

    All Rights Reserved.<br/>

    <?php echo Yii::powered(); ?>

</div><!-- footer -->
```

这里没有什么特殊的，更新为我们网站的相关信息即可。我们可以保留 **Powered by Yii Framework** 这一行来帮助 **Yii** 的推广。所以，简单的修改 **My Company** 为 **TrackStar**，一切都完成了。刷新后页脚如下图所示：



嵌套布局

位于 `protected/layouts/main.php` 中的原始布局文件并不是全部的布局文件。在我们的初始应用程序被建立时，所有的 **controllers**(控制器)都被设计成继承自 `protected/components/Controller.php`。如果我们观察一下该文件，我们将发现 **layout** 属性被直接定义。但是值并不是 **main** 布局文件，而是 **'column1'**。你也许会发现应用程序建立时 `protected/views/layouts/` 文件夹如下：

```
$ column1.php
$ column2.php
$ main.php
```

因此，除非在子类中重写该属性，我们的 **controller** 都将使用 **column1.php** 为主布局文件，而不是 **main.php**。

你也许会问，为什么要大费周章的介绍 **main.php**？因为 **column1.php** 是基于 **main.php** 实现布局的。也就是说并非 **view** 文件可以使用布局文件实现布局，布局文件也可以使用其他布局文件实现布局。这为设计的实现提供了极大的灵活性，同时也减少了 **view** 文件之间的代码重复。让我们仔细观察一下 **column1.php** 看看是如何实现的。

文件内容如下：

PHP 代码：

```
<?php $this->beginContent('/layouts/main'); ?>
```

```

<div class="container">

    <div id="content">

        <?php echo $content; ?>

    </div><!-- content -->

</div>

<?php $this->endContent(); ?>

```

正如我们所见，使用了一些我们没有见过的方法。基类方法中的 `beginContent()` 和 `endContent()` 被用来显示被附加上的特殊视图内容。这里特殊显示的是我们的主布局文件 `'layouts/main'`。 `beginContent()` 方法实际使用了 Yii 内置部件 `CContentDecorator`，该部件主要用于布局文件嵌套。因此介于 `beginContent()` 和 `endContent()` 之间的内容将作为输出内容显示在 `beginContent()` 中指定的 `view` 文件中。如果没有指定，将会使用 `controller` 级的默认布局文件，如果 `controller` 级没有指定将使用应用程序级。

嵌套工作原理如同一个普通的布局文件。当 `column1.php` 被渲染的时候，`view` 文件输出内容将保存至 `$content` 中，最终这个 `$content` 将会传递至 `main.php` 布局文件中。

以渲染登录视图来举例，以下代码出现在 `SiteController::actionLogin()` 方法中：

PHP 代码：

```

$this->render('login');

```

后台执行步骤如下：

- 1.渲染 `/protected/views/site/login.php` 的内容，存储至 `controller` 级布局文件可以使用的 `$content` 中，本例中为 `column1.php`。
- 2.在 `column1.php` 文件中使用了 `main.php` 布局文件，`beginContent()` 和 `endContent()` 之间的内容做渲染，然后通过 `$content` 传递至 `main.php`。
- 3.`main.php` 布局文件，进行渲染，并将结果返回至用户，该结果包含了来自指定 `view` 文件登录页面的内容也包括了嵌套布局文件 `column1.php` 的内容。

另一个在项目中使用的自动生成布局文件为 `column2.php`。你或许一点都不觉得奇怪，该文件使用 2 列布局。我们可以在项目页找到它，在项目子菜单 `Operations` 部件一直显示在右边。下面为该文件内容，我们可以清晰的看到嵌套布局：

PHP 代码：

```

<?php $this->beginContent('/layouts/main'); ?>

```

```

<div class="container">

    <div class="span-19">

        <div id="content">

            <?php echo $content; ?>

        </div><!-- content -->

    </div>

    <div class="span-5 last">

        <div id="sidebar">

            <?php

                $this->beginWidget('zii.widgets.CPortlet', array(

                    'title'=>'Operations',

                ));

                $this->widget('zii.widgets.CMenu', array(

                    'items'=>$this->menu,

                    'htmlOptions'=>array('class'=>'operations'),

                ));

                $this->endWidget();

            ?>

        </div><!-- sidebar -->

    </div>

</div>

<?php $this->endContent(); ?>

```

创建主题

主题提供了一个体系化的解决方案，来设计网页应用程序的布局。**MVC** 结构的最大好处就是就是将展示层从后台分离出来。主题极大的利用了该分离，允许你随时随地轻松且动态更换网站的整体外观。**Yii** 允许使用一种非常简单的应用程序主题，来为 **web** 设计提供极大的自由度。

在 Yii 中创建主题

在 **Yii** 中每一个主题都是由一系列包含 **view** 文件，布局文件，相关资源文件，如：图片，**CSS**，**JavaScript** 文件等的文件夹集合。主题的名字就是该文件包（最外层文件包）的名字。默认的，所有主题都被放置在 **WebRoot/themes** 路径下。当然，该路径可以在应用程序中进行设置。你只需要修改 **themeManager** 应用程序组件的 **basePath** 和 **baseUrl** 属性值就可以完成该修改。

一个主题文件夹下内容组织形式应该与应用程序基本路径相同。例如，所有的视图文件必须位于 **views/** 下，布局视图文件应该位于 **views/layouts/** 下，系统视图文件应当位于 **views/system/** 下。举例说明，如果我们创建了一个主题，命名为 **custom**，并且我们想在该主题下建立新的 **ProjectController** 的 **update view**，我们需要在应用程序的 **themes/custom/views/project/update.php** 建立一个新 **view** 文件。

创建一个 Yii 的主题

让我们动手，给我们的 **TrackStar** 做个小整容手术。我们需要给主题命名，并且在 **Webroot/themes** 下建立拥有该名称的文件夹。我们将会创造一些新东西，并且为我们的新主题起名：**new**。

在 **Webroot/themes/new** 建立新目录，同时在该目录下建立 2 个新文件夹，分别叫 **css/** 和 **views/**。前面的文件夹不是主题系统必须的，但是可以帮助我们组织好 **CSS** 文件。后面的文件夹是必须的，所有针对默认 **view** 文件的修改都在这里进行。我们将对 **main.php** 布局文件进行小小修改，我们需要在 **views/** 路径建立文件夹 **layouts** (请记住文件结构需要同 **Webroot/protected/views/** 相同)。

是时候动手做些事情了。因为我们的 **view** 文件的标记语言已经应用了定义在 **Webroot/css/main.css** 文件中的类和 **ID**，最快的方式就是以此作为起点，然后对其进行一些修改来实现新设计。当然这不是必须的，我们可以在新主题中重建每一个单独的 **view** 文件。但是，为求简单，我们将通过对 **main.css** 文件进行修改来实现新主题，该文件是在自动生成主布局文件 **main.php** 时同时生成的。

动手拉！让我们将上述 2 个文件复制到我们的新主题文件夹。复制 **Webroot/css/main.css** 到 **Webroot/themes/new/css/main.css**，并且也复制 **Webroot/protected/views/layouts/main.php** 到 **Webroot/themes/new/views/layouts/main.php**。

现在，打开新复制得到的 **main.css** 文件，使用以下内容替换：

CSS 代码：

```
body
{
    margin: 0;
```

```
padding: 0;

color: #555;

font: normal 10pt Arial, Helvetica, sans-serif;

background: #d6d6d6 url(background.gif) repeat-y center top;
}
```

#page

```
{

margin-bottom: 20px;

background: white;

border: 1px solid #898989;

border-top: none;

border-bottom: none;

}
```

#header

```
{

margin: 0;

padding: 0;

height: 100px;

background: white url(header.jpg) no-repeat left top;

border-bottom: 1px solid #898989;

}
```

#content

```
{

    padding: 20px;

}


#sidebar

{

    padding: 20px 20px 20px 0;

}


#footer

{

    padding: 10px;

    margin: 10px 20px;

    font-size: 0.8em;

    text-align: center;

    border-top: 1px solid #C9E0ED;

}


#logo

{

    padding: 10px 20px;

    font-size: 200%;

    /* HIDES LOGO TEXT */

    text-indent: -5000px;

}
```

```
#mainmenu
```

```
{  
  
    background: white url (bg2.gif) repeat-x left top;  
  
    border-top: 1px solid #CCC;  
  
    border-bottom: 1px solid #7d7d7d;  
  
}
```

```
#mainmenu ul
```

```
{  
  
    padding: 6px 20px 5px 20px;  
  
    margin: 0px;  
  
}
```

```
#mainmenu ul li
```

```
{  
  
    display: inline;  
  
}
```

```
#mainmenu ul li a
```

```
{  
  
    color: #333;  
  
    background-color: transparent;  
  
    font-size: 12px;  
  
    font-weight: bold;
```

```
text-decoration: none;

padding: 5px 8px;

}


#mainmenu ul li a:hover, #mainmenu ul li a.active

{

color: #d11e1e;

background-color: #ccc;

text-decoration: none;

}


div.flash-error, div.flash-notice, div.flash-success

{

padding: .8em;

margin-bottom: 1em;

border: 2px solid #ddd;

}


div.flash-error

{

background: #FBE3E4;

color: #8a1f11;

border-color: #FBC2C4;

}
```

div.flash-notice

```
{  
  
  background: #FFF6BF;  
  
  color: #514721;  
  
  border-color: #FFD324;  
  
}
```

div.flash-success

```
{  
  
  background: #E6EFC2;  
  
  color: #264409;  
  
  border-color: #C6D880;  
  
}
```

div.flash-error a

```
{  
  
  color: #8a1f11;  
  
}
```

div.flash-notice a

```
{  
  
  color: #514721;  
  
}
```

div.flash-success a

```
{

    color: #264409;

}


div.form .rememberMe label

{

    display: inline;

}


div.view

{

    padding: 10px;

    margin: 10px 0;

    border: 1px solid #C9E0ED;

}


div.breadcrumbs

{

    font-size: 0.9em;

    padding: 10px 20px;

}


div.breadcrumbs span

{

    font-weight: bold;

}
```

```
}
```

```
div. search-form
```

```
{
```

```
padding: 10px;
```

```
margin: 10px 0;
```

```
background: #eee;
```

```
}
```

```
.portlet
```

```
{
```

```
}
```

```
.portlet-decoration
```

```
{
```

```
padding: 3px 8px;
```

```
background: white url (bg2.gif) repeat-x left top;
```

```
}
```

```
.portlet-title
```

```
{
```

```
font-size: 12px;
```

```
font-weight: bold;
```

```
padding: 0;
```

```
margin: 0;
```



```
        color: #fff;
    }

    .portlet-content
    {
        font-size: 0.9em;

        margin: 0 0 15px 0;

        padding: 5px 8px;

        background: #ccc;
    }

    .operations li a
    {
        font: bold 12px Arial;

        color: #d11e1e;

        display: block;

        padding: 2px 0 2px 8px;

        line-height: 15px;

        text-decoration: none;
    }

    .portlet-content ul
    {
        list-style-image: none;

        list-style-position: outside;
```

```
list-style-type: none;

margin: 0;

padding: 0;
}

.portlet-content li

{

padding: 2px 0 4px 0px;
}

.operations

{

list-style-type: none;

margin: 0;

padding: 0;
}

.operations li

{

padding-bottom: 2px;
}

.operations li a

{

font: bold 12px Arial;
```

```

        color: #0066A4;

        display: block;

        padding: 2px 0 2px 8px;

        line-height: 15px;

        text-decoration: none;
    }

    .operations li a:visited

    {

        color: #d11e1e;
    }

    .operations li a:hover

    {

        background: #fff;
    }

```

你可能发现修改中使用到的一些图片文件并不存在。我们添加了 **background.gif** 图片到 **body** 定义中，一个新的 **bg2.gif** 图片到 **#mainmenu** ID 定义中，一个新的 **header.jpg** 图片到 **#header** ID 定义中。可以通过如下连接进行查看和下载：

<http://www.yippyii.com/trackstar/themes/new/css/background.gif>,
<http://www.yippyii.com/trackstar/themes/new/css/bg2.gif>,
<http://www.yippyii.com/trackstar/themes/new/css/header.jpg>

我们需要将上面 3 个图片复制到位于 **Webroot/themes/new/css/** 的 **CSS** 文件夹下。

上面的操作都完成后，我们需要对我们新主题的 **layout** 文件 **main.php** 进行少量调整。首先，我们要修改中的标记语言来引用我们新 **mian.css** 文件，未修改前 **main.css** 文件引用方式如下：

PHP 代码：

```

<link rel="stylesheet" type="text/css" href="<?php echo Yii::app()-
>request->baseUrl; ?>/css/main.css" />

```

引用了基于应用程序(application)请求(request)baseUrl 属性构成的 CSS 文件相对路径。然而我们想使用位于新主题目录下的新 main.css 文件。所以我们可以使用位于 Yii 内置 CThemeManager.php 类的 theme manager(主题管理)应用程序组件。如访问其他应用程序组件一样访问 theme manager。因此我们将使用 theme manager 中的 url 来替换基于请求的 url。所以我们对 /themes/new/views/layouts/main.php 如下修改:

PHP 代码:

```
<link rel="stylesheet" type="text/css" href="<?php echo Yii::app()->theme->baseUrl; ?>/css/main.css" />
```

一旦我们配置应用程序使用新主题(这一步还没有完成),这里的 baseUrl 将会解析为我们主题文件夹的相对路径。

另外一处小修改是,移除页头显示的应用程序标题。因为我们在此处使用了图片,所以无需再使用标题。因此,还是/themes/new/views/layouts/main.php,我们只需修改这里:

PHP 代码:

```
<div id="header">

    <div id="logo"><?php echo CHtml::encode(Yii::app()->name); ?></div>

</div><!-- header -->
```

为下面的样子:

在此我们使用了注释提示我们页头的图片定义的位置。

最后一处修改是针对应用程序中使用的另外 2 个布局文件,我们并未将其复制到新主题目录,他们是: protected/views/layouts/column1.php 和 protected/views/layouts/column2.php。如前面嵌套布局所述,这 2 个布局通过 beginContent() 和 endContent() 调用了 main 布局文件。这些文件是通过 Gii 自动代码生成工具生成的,并且直接指向 protected/views/layouts/ 文件夹。我们需要对 beginContent() 方法进行修改,使之对我们的新主题的布局有效。分别打开 column1.php 和 column2.php,将下面的代码:

PHP 代码:

```
$this->beginContent('application.views.layouts.main');
```

修改为:

PHP 代码:

```
$this->beginContent('/layouts/main');
```

现在，一旦我们配置应用程序使用新主题，它将使用新主题文件夹下的 **main.php** 布局文件。

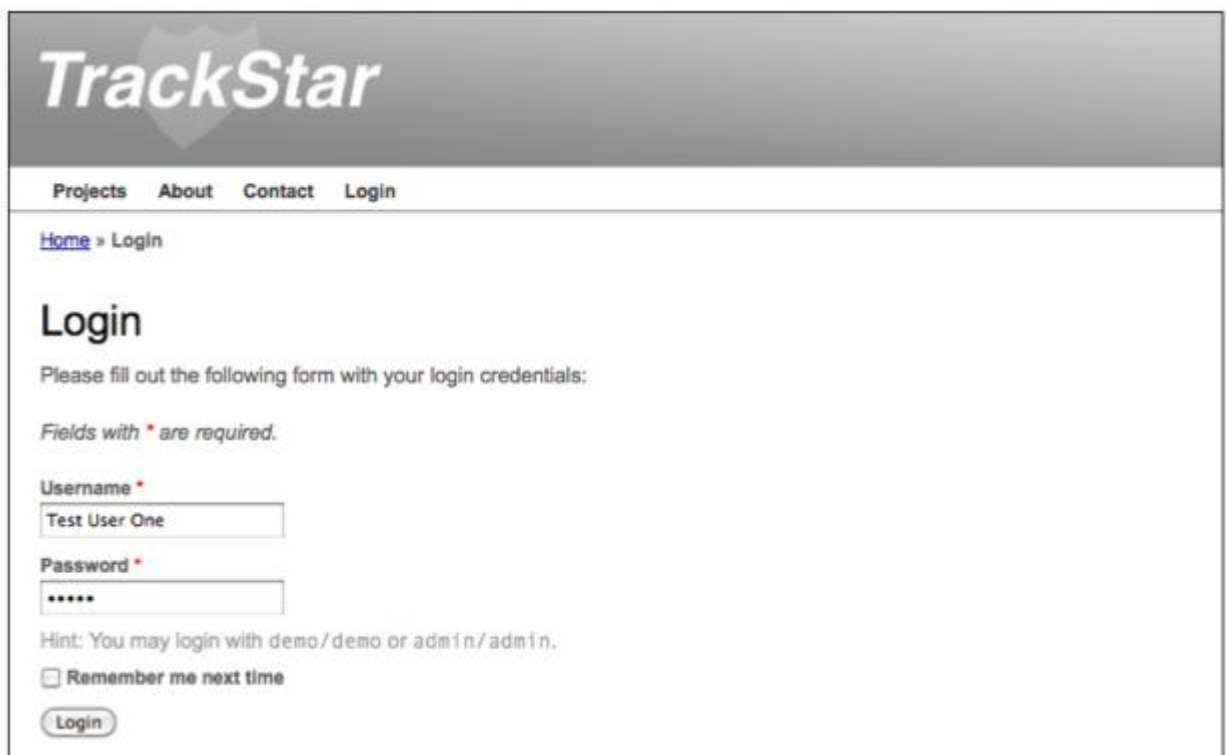
为应用程序配置新主题

好啦，随着我们新主题的建立完成，是时候告诉应用程序使用它了。操作非常之简单。我们只需要修改主配置文件中 **Application(应用程序)** 的 **theme** 属性。这一操作已经非常熟悉了。在 **/protected/config/main.php** 文件的返回数组中添加如下键值对：

PHP 代码：

```
'theme' => 'new',
```

保存之后，名为 **new** 的新主题已经被我们的应用程序使用了，并且我们的应用程序已经有了一个全新的样子。观察一下非登录状态下的首页，也就是登录页面，如下图所示：



当然整体改动非常小。我们用最小的改动来创建了一个新主题。应用程序会优先使用主题文件夹下的 **view** 文件，如果没有找到才会使用默认路径下的。你可以看到换个皮肤如此的简单。你可以随心所欲的做这件事。

将网站翻译至其他语言

在完成本次迭代之前，我们将要讨论一下 **Yii** 中的 **i18n(国际化)** 和 **L10n(本地化)**。国际化就是一个使软件程序自动适应不同语言而不需要修改底层引擎的过程。本地化就是使国际软件适应某一区域或者某一语言的过程。**Yii** 通过如下方面实现对他们支持：

§ Yii 为每一种语言和区域提供本地化数据支持

§ Yii 提供相关服务帮助翻译文字信息和文件

§ Yii 提供语言相关的日期和时间格式化

§ Yii 提供语言相关的数字格式化

定义区域和语言

区域是指一系列定义用户语言，国家，和其他用户界面预设信息(例如，用户所在地)的参数。一般来说是由语言和区域标识组成的 ID。举例说明，区域 ID `en_US` 代表英语语言，所在区域代表美国。约定俗成的，Yii 中所有的区域 ID 都使用小写的语言 ID 或语言 ID_区域 ID(例如，`en`, `en_us`)。

在 Yii 中，本地化数据是 `CLocal` 类或其子类的一个实例。它提供了区域化信息，包括货币和数字符号；货币，数字，日期，时间格式；日期描述名称比如月份，星期几，等等。给定一个区域 ID，就可以通过静态方法 `CLocal::getInstance($localeID)` 或应用程序获得相关 `CLocal` 实例。下面就是一个使用应用程序组件获得 `en_us` 区域 ID 实例的例子：

PHP 代码：

```
Yii::app()->getLocale('en_us');
```

Yii 几乎包含了所有语言和区域的本地化数据。数据都是从通用语言环境数据信息库(CLDLDR) (<http://cldr.unicode.org/>) 并且按照其相关语言 ID 进行命名后储存在 Yii 框架的 `framework/i18n/data/` 目录。因此，按照上面的例子创建一个新的 `CLocale` 实例，用来填充属性的数据来自 `framework/i18n/data/en_us.php` 文件。如果你打开这一文件夹，你会发现大量语言和地区信息文件。

现在，回到我们的例子，如果我们想获取和显示月份的名字以英语按照 **US**(美国)地区标准，我们需要执行以下代码：

PHP 代码：

```
$locale = Yii::app()->getLocale('en_us');  
  
print_r($locale->monthNames);
```

将会产生如下输出：

SHELL 代码或屏幕回显：

```
Array ( [1] => January [2] => February [3] => March [4] => April [5] => May [6] => June [7] =>  
July [8] => August [9] => September [10] => October [11] => November [12] => December )
```

正如我们所说的，意大利语的月份名，使用相同的方式，只是实例化不同的 `CLocale`：

PHP 代码:

```
$locale = Yii::app()->getLocale('it');

print_r($locale->monthNames);
```

将会产生如下输出:

SHELL 代码或屏幕回显:

```
Array ( [1] => gennaio [2] => febbraio [3] => marzo [4] => aprile [5] => maggio [6] => giugno
[7] => luglio [8] => agosto [9] => settembre [10] => ottobre [11] => novembre [12] => dicembre )
```

第一个实例是基于信息文件 `framework/i18n/data/en_us.php`, 后一个是基于 `framework/i18n/data/it.php`。如果有必要, 应用程序的 `localeDataPath` 可以被配置到任何用户自定义时间信息文件。

实现语言切换

或许 `i18n` 最值得期待的功能就是语言切换了。如前面所说的, `Yii` 提供了信息翻译和试图翻译 2 种。前一种将单独文本消息翻译为目标语言的, 后者将整个 `view` 文件翻译成目标语言的。

一个翻译请求由被翻译的对象(一段文字或整个文件), 该对象的语言以及目标语言组成。在 `Yii` 应用程序中源语言和目标语言是不同的。目标语言是我们针对一类目标用户由书写应用程序的源语言翻译而来的语言。目前为止我们的 `TrackStar` 应用程序是由英文书写并且正对英文用户。所以目前为止我们的目标和源语言都是相同的。`Yii` 的国际化功能是在针对 2 种不同语言的情况下进行翻译。

实现消息翻译

消息翻译是通过调用以下应用程序方法:

PHP 代码:

```
t(string $category, string $message, array $params=array ( ), string $source=NULL, string
$language=NULL)
```

这个方法将消息由源语言翻译到目标语言。

当翻译一条消息的时候, `category`(类别)必须指出, 才可以实现同一条消息在不同类别实现不同翻译。类别 `yii` 在 `Yii` 框架核心代码中被预留。

消息可以包含占位符参数, 该参数将在 `Yii::t()` 被调用时被替换。下面的一个例子实现了对错误信息的翻译。在该消息被翻译的时候 `{errorCode}` 占位符需要被 `$errorCode` 的实际值替换:

PHP 代码:

```
Yii::t('category', 'The error: "{errorCode}" was encountered during the last request.',  
array('{errorCode}' => $errorCode));
```

被翻译的消息被存储在一个叫 **message source**(消息源)的信息仓库中。一个消息源代表 **CMessageSource** 或者它子类的实例。一旦 **Yii::t()** 被调用, 将会在消息源中查找消息, 如果找到并将返回结果。

Yii 拥有以下几种消息源:

§ **CPhpMessageSource**: 这个为默认的消息源。可以被翻译的消息以键值对的形式存储在 PHP 数组中。原始消息为 **key**, 被翻译的消息为 **value**。每一个数组代表实际的类别信息, 并且存储在以类别名命名的单独 PHP 脚本文件中。相同语言的 PHP 翻译文件储存在以该语言 ID 命名的文件夹下。所有的文件夹都存储在路径 **basePath** 下。

§ **CGettextMessageSource**: 可翻译被存储在 GNU Gettext 文件中。

§ **CDbMessageSource**: 可翻译信息存储在数据库表中。

一个消息源以应用程序组件的形式被加载。Yii 会预先定义一个叫 **message** 的应用程序组件来存储用于用户应用程序的信息。默认的, 该消息源是一个 **CPhpMessageSource** 类型并且 PHP 翻译文件的存储路径为 **protected/message**。

举一个例子来详细说明, 让我们来翻译登录表单的标签内容到一个杜撰的叫 **Reversish**。**Reversish** 就是将英文单词或短语逆向书写。下面是登录表单的 **Reversish** 翻译关系。

English	Reversish
Username	Emanresu
Password	Drowssap
Remember me next time	Emit txen em rebmemer

我们将使用默认的 **CPhpMessageSource** 来存放翻译消息。所以第一步我们需要做的就是新建一个 PHP 文件来存放我们的翻译。我们将使用语言 ID **'rev'** 并且使用 **'default'** 分类。所以我们需要在消息基本目录下建立如下格式的文件夹 **/localeID/CategoryName.php**。按照这个例子, 我们的新文件应该存放在 **/protected/messages/rev/default.php**, 并且置入如下翻译数组:

PHP 代码:

```
<?php return array(  
  
    'Username' => 'Emanresu',  
  
    'Password' => 'Drowssap',  
  
    'Remember me next time' => 'Emit txen em rebmemer',
```



```
);
```

接下来我们需要做的是将应用程序的目标语言设置为 **Reversish**。我们可以在应用程序配置文件进行设置，这样该设置将会对整个网站起作用。但是我们只想翻译登录表单，所以我们只需要在 **SiteController::actionLogin()** 方法内进行设置，所以该设置只会在渲染登录表单时有效。所以打开该文件，在方法的开头按照如下形式设置目标语言：

PHP 代码：

```
public function actionLogin()

{

    Yii::app()->language = 'rev';
```

最后我们需要做的就是调用 **Yii::t()** 方法来激活翻译。这些表单的标签被定义在 **LoginForm::attributeLabels()** 方法中。按照如下格式重写该方法：

PHP 代码：

```
/**

 * Declares attribute labels.

 */

public function attributeLabels()

{

    return array(

        'rememberMe' => Yii::t('default', 'Remember me next time'),

        'username' => Yii::t('default', 'Username'),

        'password' => Yii::t('default', 'Password'),

    );

}
```

现在如果你访问登录表单，**Reversish** 语言的登录表单将会如图所示显示：

Login

Please fill out the following form with your login credentials:

*Fields with * are required.*

Emanresu *

Drowssap *

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

☐ Emit txen em rebmemer

执行文件翻译

Yii 也提供了根据设置的目标语言 ID 使用不同文件的能力。文件级翻译通过调用应用程序方法 **CApplication::findLocalizedFile()** 来实现。该方法需要一个文件路径并且会使用相同的名字查找文件，而不是使用一个直接输入的目标语言 ID 或者由应用程序输入这个 ID 来进行目录式查找。

让我们试一下。我们所需要的就是创建所需的翻译文件。依旧是翻译登录表单。所以在 **/protected/views/site/rev/login.php** 建立新的 **view** 文件，同时将下面的已经被翻译为 **Reversish** 的内容复制进去：

PHP 代码：

```
<?php

$this->pageTitle='Ni gol';

$this->breadcrumbs=array(

    'Ni gol',

);

?>

<h1>Ni gol</h1>

<p>Slaitnederc nigol ruoy htiw mrof gniwol lof eht tuo lli f esaelp:</p>
```

```

<div class="form">

<?php $form=$this->beginWidget('CActiveForm', array(

    'id'=>'login-form',

    'enableAjaxValidation'=>true, ));

?>

<p class="note">Deriuqer era <span class="required">*</span> htiw sdleif.</p>

<div class="row">

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

    <?php echo $form->textField($model, 'username'); ?>

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

</div>

<div class="row">

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

    <?php echo $form->passwordField($model, 'password'); ?>

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

    <p class="hint">

        <tt>nimda\nimda</tt> ro <tt>omed\omed</tt> htiw nigol yam uoy:tnih

    </p>

</div>

<div class="row rememberMe">

    <?php echo $form->checkBox($model, 'rememberMe'); ?>

    <?php echo $form->label($model, 'rememberMe'); ?>

```

```

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

    </div>

    <div class="row buttons">

        <?php echo CHtml::submitButton('Nigol'); ?>

    </div>

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

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

```

我们已经在 `SiteController::actionLogin()` 中设置了目标语言, 同时调用本地化文件会在 `render('login')` 的时候后台执行。所以, 当这些都完成以后, 我们的登录表单会显示如下图所示的样子:

Home » Nigol

Nigol

Slaitneder nigel ruoy htiw mrof gniwollof eht tuo llif esaelp:

Deriuqer era * htiw sdleif.

Emanresu *

Drowssap *

.nimda\nimda ro omed\omed htiw nigel yam uoy :tnih

☐ Emit txen em rebmemer

Nigol

小结

在本次迭代中, 我们知道了 **Yii** 如何迅速且容易的使我们的设计更完美。我们介绍了布局这个概念, 并且了解了如何在一个应用程序的多个不同页面通过布局实现设计的统一风格。在这里我们同样介绍了 2 个内置部件 **CMenu** 和 **CBreadcrumbs**, 通过这 2 个部件可以轻松的实现 **Yii** 内部导航功能。

接着我们介绍了基于 **Web** 应用程序的主题的概念, 同时介绍了如何在 **Yii** 中实现他们。它允许我们轻松的为 **Yii** 应用程序更换皮肤, 通过它你可以重新设计你的应用程序而无需对后台功能进行重构。

最后我们学习了如果通过 **i18n** 来对网站语言进行切换。我们学习了如何设置应用程序的目标语言信息，通过它我们可以实现本地化设置和语言翻译。

在本章和之前的章节我们都介绍了 **modules**(模块)的概念，但是都没有深入讲解在 **Yii** 的具体实现。这就是我们下一章的重点。

应用 Yii 和 PHP5 进行 web 的敏捷开发

第十二章：迭代 9： 添加管理模块

到目前为止，我们已经给 **TrackStar** 应用程序增加了很多功能。如果你还记得在第 8 章，我们介绍了使用基于用户角色的层次结构来限制用户访问某些功能。这有助于限制对一些单一项目的管理职能的访问。例如：在一个特定的项目中，你可能不希望让所有的成员都有删除权限。我们为用户分配到一个项目内的特殊角色来实现基于角色的访问控制，然后控制这些角色是否有访问该功能的权限。

然而，我们尚未解决的是整个应用程序的管理需求。例如 **TrackStar Web** 应用程序经常需要的有一个非常特殊的用户拥有所有的管理权限。其中一个例子就是让系统的每一个用户拥有 **CURD** 操作，而不仅仅是项目。我们应用程序的系统管理员应该有以下权限：能够登录，删除或者修改用户、项目、问题，管理所有的评论，等等。同时，我们经常建立适用于整个应用程序的额外功能，如能给所有用户发布站点系统信息，管理电子邮件运动，打开/关闭某些应用功能，管理角色的层次，改变站点主题，等等。因为管理员的权限与用户的权限有极大的不同，把这些功能和应用程序分离是一个好想法。我们将通过 **Yii** 模块建立所有的管理功能来完成这个分离。

12.1 迭代计划

在这个迭代中，我们将集中完成以下开发任务：

- 创造一个新的模块来存放管理功能
- 为系统管理员建立系统广播功能，用户可以在项目列表页面查看
- 为应用模块添加一个新课题
- 创建一个新表保持系统消息数据
- 为系统信息生成所有的 **CURD** 操作
- 只允许管理用户使用新模块中的功能
- 在项目列表页面显示新系统信息

12.2 模块

模块类似于一个大型应用程序中的小型应用程序。它有与应用程序类似的结

构，包含模型、视图、控制器及其他支持组件。然而，模块不能作为独立的应用程序，它们必须嵌入某一个应用程序。

Modules 在帮助你的应用程序模块化架构方面很有用。大型的应用程序经常可以分成几个离散的应用程序，这些离散的应用程序也可以用模块来构建。站点功能，如添加一个用户论坛，用户博客或站点管理员功能都是一些例子，说明从主站点功能分离出来的功能，可以让它们单独开发，并且容易在未来的计划中重用。我们将在应用程序中不同的位置建立存放管理功能的模块。

12.3 创建一个模块

使用我们的老朋友，**Gii** 代码生成工具创建一个新模块是很容易的。伴随着我们的 **URL** 的改变，要通过 **http: //localhost/ trackstar / gii** 来访问该工具。打开后，在左边的菜单中选择 **Module Generator** 选项。你将看到下面的画面：



我们需要给模块取一个唯一的名字。由于我们正在创建一个管理模块，我们可以命名为 **admin**。在 **Module ID** 内输入 **admin**，点击 **Preview** 按钮。如下图所示，它会向你展示它将会生成的所有文件，让你在创建之前预览这些文件：

Code File	Generate <input type="checkbox"/>
modules/admin/AdminModule.php	new <input checked="" type="checkbox"/>
modules/admin/components	new <input checked="" type="checkbox"/>
modules/admin/controllers/DefaultController.php	new <input checked="" type="checkbox"/>
modules/admin/messages	new <input checked="" type="checkbox"/>
modules/admin/models	new <input checked="" type="checkbox"/>
modules/admin/views/default/index.php	new <input checked="" type="checkbox"/>
modules/admin/views/layouts	new <input checked="" type="checkbox"/>

然后点击 **Generate** 按钮来生成这些文件。由于 **web** 服务器进程它自动创建文件夹和文件的要求，所以要确保你的 **/protected** 文件夹是可写入的。下面的截图显示一个成功的模块生成：

Preview

The module has been generated successfully.

To access the module, you need to modify the application configuration as follows:

```
<?php
return array(
    'modules'=>array(
        'admin',
    ),
    .....
);
```

```
Generating code using template "/Users/jeff/TrackStar/Yii-1.1.2/framework
generated modules/admin/AdminModule.php
generated modules/admin/components
generated modules/admin/controllers/DefaultController.php
generated modules/admin/messages
generated modules/admin/models
generated modules/admin/views/default/index.php
generated modules/admin/views/layouts
done!
```

让我们更进一步的看看这个自动生成的模板。Yii 中模块是以一个文件夹组织的，文件夹的名字就是该模块的名字。默认情况下，居所有的模块保存在 `/protected/modules` 目录下。每个模块文件夹的结构都与主应用程序非常相似。这个指令为我们所做的是创建 **admin** 模块建立脚手架文件夹结构。因为这是我们的第一个模块，顶层的文件夹 `/protected/modules` 被创建了，然后将 **admin** 文件夹放在里面。下面为我们展示了模块命令行为我们建立的所有文件夹和文件：

Name of folder	Use/contents
admin/	
AdminModule.php	the module class file
components/	containing reusable user components
controllers/	containing controller class files
DefaultController.php	the default controller class file
messages/	stores message translations specific to the module
models/	containing model class files
views/	containing controller view and layout files
default/	containing view files for DefaultController
index.php	the index view file
layouts/	containing layout view files

一个模块必须拥有一个模块继承自 `CWebModule` 或其子类。该模块类的名字由模块 ID (在这里指 `admin`) 和字符串 `Module` 联合生成。模块 ID 的第一个字母大写。所以,在我们的例子中,我们的 `admin` 模块类文件被命名 `AdminModule.php`。模块类主要作为存储模块代码共享信息的中转站。例如,我们可以用 `CWebModule` 的 `params` 属性来存储模块性能的具体参数,并利用其 `components` 属性在模块级分享应用程序的组件。这个模块类在模块中的作用类似于应用程序类对整个应用程序的作用。所以 `CWebModule` 是对模块的就像 `CWebApplication` 是对应用程序的。

12.4 使用一个模块

正如成功建立的信息表明,在我们使用新模块之前,我们需要在主应用程序中设置 `modules` 属性,才可以使用它。在添加 `gii` 模块应用程序之前,允许我们使用 `gii` 代码生成工具。我们对主要配置文件 `protected/config/main.php` 进行改变。以下代码显示了所需的改变:

PHP 代码:

```
'modules'=>array(
    'gii'=>array(
        'class'=>'system.gii.GiiModule',
        'password'=>'iamadmin',
    ),
    'admin',
),
```


保存这个改变后，我们的新admin模块就可以使用了。我们可以先通过访问 `http://localhost/trackstar/admin/default/index` 来看一下。该请求显示的模块的访问页类似于我们的主应用程序页，除了我们需要在路由中添加moduleID。所以我们的路径形式如下：`moduleID/controllerID/actionID`。我们URL请求 `/admin/default/index` 应该解释为admin模块的default控制器的index方法。当我们访问这个页面，我们看到类似下面的画面：

admin/default/index

This is the view content for action "index". The action belongs to the controller "DefaultController" in the "admin" module.

You may customize this page by editing `/Webroot/trackstar/protected/modules/admin/views/default/index.php`

12.5 主题化一个模块

我们立刻发现这一视图（View）似乎没有应用任何布局（Layout）。有人可能会想，控制器渲染视图时用的是`renderPartial()`，而不是`render()`。然而，在检查我们的默认的admin的controller文件，`/protected/modules/admin/controllers/DefaultController.php`时，我们可以看到，事实上它使用的是`render()`方法。因此，我们需要使用一个layout文件(如果有的话)。

问题是在模块中几乎一切都是独立的，包括layout文件的默认路径。网站模块布局的默认路径是`/protected/modules/[moduleID]/views/layouts`，在我们的例子中moduleID应该是admin。我们可以看到，在这个文件夹中没有文件，所以没有应用默认布局。

在这里多讲一点。在上一次迭代中，我们实现了一个名为new的新主题。我们也可以通过这个主题管理我们所有的模块view文件，包括layout view文件。如果我们这样做，我们需要增加一些主题文件夹结构以适应我们的新模块。文件夹的结构与预期的一样。大致上为：

`/themes/[themeName]/views/[moduleID]/layouts/` 为布局文件，

`/themes/[themeName]/views/[moduleID]/[controllerID]/` 为对应controller的视图文件。

为了说得更清楚，我们模拟一次admin模块调用view的过程的Yii决策流程。下面就是admin模块的DefaultController.php文件中 `$this->render('index')` 的渲染过程：

- 1.当`render()`被调用，与`renderPartial()`不同，它会尝试用一个layout文件来修饰指定的视图文件`index.php`。我们的应用程序现在被配置使用名为new的主题，所以要在这个主题文件夹中找到layout文件。我们的新模块的DefaultController类继承自我们的应用程序组件controller.php，所以使用

了column1作为指定的\$layout的属性值。这个属性没有被重写，所以这也是DefaultController的Layout值。最后，当这些都在admin模块中完成后，Yii首先寻找以下layout文件：

/themes/new/views/admin/layouts/column1.php。注意到在这个文件夹结构包含了moduleID。

2.这个文件不存在，所以在模块的默认位置查找。如前所述，每一个模块都有特定的默认布局文件夹。因此，在这种情况下，它将试图找到下面的layout文件：/protected/modules/admin/views/layouts/column1.php.

3.这个文件也不存在，因此它将不会使用layout。现在只是在没有布局时尝试简单的渲染指定试图文件index.php。但是，我们已经为应用程序定义了特定的主题new，所以它将首先寻找以下的视图文件：

/themes/new/ views/admin/default/index.php。

4.这个文件也不存在，那么它将再次查找在这个模块(AdminModule)的控制器(DefaultController.php)的默认位置，即：

/protected/modules/admin/views/default/index.

php。

这就解释了为什么网页<http://localhost/trackstar/admin/default/index>被渲染了却没有任何布局。现在，为了使事情变得完全分离和简单，让我们管理一下在我们的模块的默认位置的视图文件，而不是在主题new内。让我们为admin模块应用我们为原始应用程序设计的主题，就是在使用新主题前的样子。这样的话，我们的admin模块的页面就会和正常应用程序页面有所不同，这将有助于提醒我们，我们是在特殊的admin事务，但我们不需要花任何时间想出一个新设计。

12.6 应用一个主题

首先，让我们为模块设定一个默认的layout值。我们在模块类的init()方法设置我们的模块级配置，模块类位于/protected/ modules/AdminModule.php。打开这个文件，并且添加下面的代码：

PHP代码：

```
class AdminModule extends CWebModule
{
    public function init()
    {
        // 当模块创建时，调用这个方法
        // 你可以在这里用代码来自定义模块或应用程序
```

```

// 导入模块级模型和组件
$this->setImport(array(
    'admin.models.*',
    'admin.components.*',
));
$this->layout = 'main';
}
...

```

这样一来，如果我们在更细化的水平没有特别指定布局文件，像在一个控制器类中，模块的所有视图将会调用位于`/protected/modules/ admin/views/layouts/`下的`main.php`布局文件来修饰。

现在，我们自然需要创建这个文件。从主应用程序复制两个布局文件：`/protected/views/layouts/main.php`和`/protected/ views/layouts/column1.php`，并把他们两个放在`/protected/modules/ admin/views/layouts/`文件夹。复制完之后，我们需要对它们做一些修改。

首先修改`column1.php`文件。在`beginContent()`中去除对`/layouts/main`的引用，修改后代码如下：

PHP代码：

```

<?php $this->beginContent(); ?>
<div class="container">
<div id="content">
<?php echo $content; ?>
</div><!-- content -->
</div>
<?php $this->endContent(); ?>

```

在未指明导入文件时调用`beginContent()`，将会使用模块的默认的布局文件，而我们刚刚将其指定到新复制的`main.php`文件。

现在让我们对`main.php`布局文件做一些修改。我们打算给应用程序头部添加**Admin Console**文本，来强调我们是应用程序的一部分。我们也将修改我们的菜单项，添加一个到管理主页的链接，以及一个回到主站点的链接。我们可以消除这个菜单的**About**和**Content**链接，因为我们并不需要在管理部分重复这些选项。修改后代码如下：

PHP代码：

```

...
<div class="container" id="page">
<div id="header">
<div id="logo">
<?php echo CHtml::encode(Yii::app()->name) . " AdminConsole"; ?></div>
</div><!-- header -->
<div id="mainmenu">
<?php $this->widget('zii.widgets.CMenu', array(
    'items'=>array(
        array('label'=>'Back To Main Site', 'url'=>array('/project')),
        array('label'=>'Admin', 'url'=>array('/admin/default/index')),
        array('label'=>'Login', 'url'=>array('/site/login'),
        'visible'=>Yii::app()->user->isGuest),
        array('label'=>'Logout ('.Yii::app()->user->name.')',
        'url'=>array('/site/logout'), 'visible'=>!Yii::app()->user->isGuest)
    ), )); ?>
</div><!-- mainmenu -->
...

```

文件其余部分不变。现在如果我们访问admin模块页面，我们看到类似下面的画面：



如果我们点击Back To Main Site的链接，我们被带回到最新主题主应用程序。

12.7 限制admin模块的访问

一个你可能已经注意到的问题是：任何用户，包括游客都可以访问我们的新的admin模块。但实际上我们建立admin模块只想将该功能提供给拥有管理员权限的用户。所以我们要解决这个问题。

幸运的是，我们已经在第8章就在应用程序中实现了RBAC访问模型。我们现

在要做的是将其扩大，包括一个新的管理员角色和该角色的管理权限。如果你回忆一下在第八章中，我们使用了一个Yii脚本命令来实现RBAC的结构。我们需要增加命令。所以，打开包含脚本命令的文件，

/protected/commands/shell/RbacCommand.php和添加以下内容：

PHP代码：

```
//给管理员创建一个常规的任务级权限
$this->_authManager->createTask("adminManagement", "access to the
application administration functionality");
//创建站点管理员的角色，并添加适当的权限
$role=$this->_authManager->createRole("admin");
$role->addChild("owner");
$role->addChild("reader");
$role->addChild("member");
$role->addChild("adminManagement");
//确保我们系统内有一个管理员(使它是用户的id号为1)
$this->_authManager->assign("admin", 1);
```

当着修改完成后，我们必须重新运行我们的命令来更新数据库。这样做之后，打开yiic shell，执行rbac的命令：

```
% cd Webroot/trackstar
% protected/yiic shell
>> rbac
```

当我们对RBAC模型修改完后，我们可以添加一个访问检查AdminModule:beforeControllerAction()方法，这样admin模块中没有事物会被执行，除非用户是admin角色：

PHP代码：

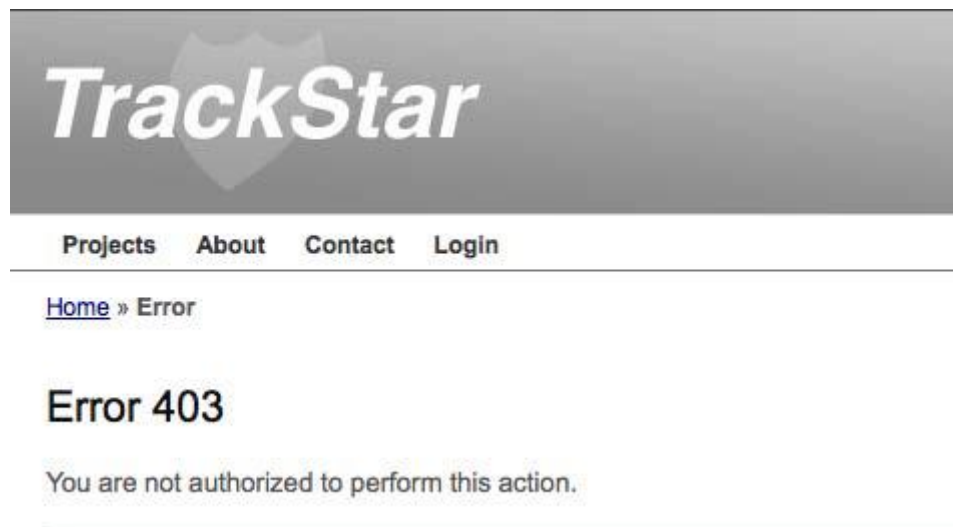
```
public function beforeControllerAction($controller, $action)
{
    if(parent::beforeControllerAction($controller, $action))
    {
        // 在模块控制器执行actions之前，调用这个方法
        // 你可以在这里自定义代码
        if(!Yii::app()->authManager->checkAccess("admin", Yii::app()-
```

```

>user->id) )
{
    throw new CHttpException(403, Yii::t('yii', 'You are not authorized
to perform this action.'));
}
else
{
    return true;
}
}
else
    return false;
}

```

当这些都完成后，如果在admin模块中，一个没有被分配到admin角色的用户试图访问网页，他们将会遇到授权错误页面。例如，如果你在没有登录的情况下，试图访问管理页面，你会遇到以下结果：



这同样适用于任何未被指派admin角色的用户。

现在，我们可以在主应用程序菜单添加admin部分的链接。这样，拥有管路员权限的用户不需要记住的是繁琐的URL就可以进入到管理控制台。作为一个提示，我们的主应用程序菜单位于应用主题的默认布局文件中，
/themes/new/views/layouts/main.php。打开这个文件，并且作如下修改：
PHP代码：

```

<div id="mainmenu">
<?php $this->widget('zii.widgets.CMenu', array(

```

```

'items'=>array(
    array('label'=>'Projects', 'url'=>array('/project')),
    array('label'=>'About', 'url'=>array('/site/page', 'view'=>'about')),
    array('label'=>'Contact', 'url'=>array('/site/contact')),
    array('label'=>'Admin', 'url'=>array('/admin/default/index')),
    'visible'=>Yii::app()->authManager->checkAccess("admin", Yii::app()->user->id),
    array('label'=>'Login', 'url'=>array('/site/login'),
    'visible'=>Yii::app()->user->isGuest),
    array('label'=>'Logout ('.Yii::app()->user->name.')',
        'url'=>array('/site/logout'), 'visible'=>!Yii::app()->user->isGuest)
    ), )); ?>
</div><!-- mainmenu -->

```

现在，使用有admin权限的用户登录应用程序，我们将会看到在顶部的导航中有一个新的链接，它将带我们到站点新添加的admin部分。



12.8 添加系统级的信息

由于一个模块可以被认为是一个小型应用程序，因此，向模块添加功能和向主程序添加功能具有相同的步骤。让我们只为管理员添加一些新功能；这个功能可以使他们第一次登录应用程序时向用户显示的系统级信息。

12.9 创建数据库表

和创建新功能一样，我们需要一个地方来放置我们的数据。我们需要创建一个新的表来存储我们系统级的信息。对于我们而言，我们可以简单一点。下面是我们数据表的定义：

SQL 代码：

```

CREATE TABLE `tbl_sys_message`
(

```

```
`id` INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,  
`message` TEXT NOT NULL,  
`create_time` DATETIME,  
`create_user_id` INTEGER,  
`update_time` DATETIME,  
`update_user_id` INTEGER  
)
```

在主要的 `trackstar_dev` 和我们的 `trackstar_test` 数据库中都创建这个新表。

12.10 创建模型和 CRUD 脚手架

当表建立好以后，下一步就是使用 Gii 代码生成器来建立模型类了。我们将首先使用 **Model Generator** 来创建模型类，然后用 **Crud Generator** 创建脚手架来与该模型快速互动。接下来，引导 Gii 工具表单来创建一个新的模型。

这一次，我们是在模块内容中做的，我们需要明确指定模型的路径。用下面的图中所描述的值填写表单（当然，你的 **Code Template** 路径值应该是具体到您的本地设置）：

Model Generator

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

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

Table Prefix

tbl_

Table Name *

tbl_sys_message

Model Class *

SysMessage

Base Class *

CActiveRecord

Model Path *

application.modules.admin.models

Code Template *

default (/Users/jeff/TrackStar/Yii-1.1.2/framework/gii/generators/model/templates/default)

Preview

现在，我们可以用相同的方式创建的 **CRUD** 脚手架。同样的，以前我们所做的与我们现在正在做的之间的唯一的区别在于模型类的位置是在 **admin** 模块中。从 Gii 工具选择 **Crud Generator** 选项后，按照如下图填写 **Model Class** 和 **Controller**

ID:

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

Model Class *

admin.models.SysMessage

Controller ID *

admin/sysMessage

这里要注意一点，我们的模型类是在 **admin** 模块之中的，我们的控制类以及所有其他与这个代码生成有关的文件也应被放置在管理模块中。

通过点击 **Preview** 按钮后点击 **Generate** 按钮完成文件创建。下图是所有被创建的文件列表：

Code File
modules/admin/controllers/SysMessageController.php
modules/admin/views/sysMessage/_form.php
modules/admin/views/sysMessage/_search.php
modules/admin/views/sysMessage/_view.php
modules/admin/views/sysMessage/admin.php
modules/admin/views/sysMessage/create.php
modules/admin/views/sysMessage/index.php
modules/admin/views/sysMessage/update.php
modules/admin/views/sysMessage/view.php

12.11 将链接添加到我们的新功能

让我们在主 **admin** 部分导航内添加一个新的菜单项链接到我们新建的消息功能。打开含有我们的主菜单导航的文件，`/protected/modules/admin/views/layouts/main.php`，并将在 `menu` 不见添加下列数组项：

```
array('label'=>'System Messages',  
'url'=>array('/admin/sysMessage/index')),
```

由于为新的系统消息功能自动创建 **controller** 和 **view** 文件使用了2列布局文件，我们可以做下列两件事之一：我们可以改变 **controller** 文件让它使用我们现有的单个列布局文件，或者我们可以新增2列的布局文件到我们的 **admin** 模块。后面的会很容易做到，也会更好看，因为所有的 **view** 文件都被设计成拥有一个自菜单项，里面存放了所有 **crud** 功能的连接，而该菜单项被放在右边的一列中。下面是我们所要做的步骤：

1. 从主应用程序中复制2列布局到 **admin** 模块：即复制`/protected`

iews/layouts/column2.php 和/protected/modules/
admin./views/layouts/column2.php。

2.删除新复制的 column2.php 文件中的第一行上的 beginContent() 方法的输入内容/layouts/main。

3.修改 SysMessage()模型类，使之继承自 TrackstarActiveRecord(如果你还记得，这将添加代码自动更新 create_time/user 和 update_time/user 属性)。更新 SysMessageController 控制器类使用新的位于模块文件内的 column2.php 布局文件，而不是主应用程序的。自动生成的代码中 \$layout='application.views.layouts.column2'，但我们需要的修改 \$layout = 'column2'。

4.由于我们继承自 TrackstarActiveRecord，我们可以删除在自动生成系统消息创建的窗体中不必要的字段和模型的类，同时移除与之项目的模型类中的 rules。在 SysMessage::rules()方法中移除一下内容:array('create_user, update_user, 'numerical, 'integerOnly'=>true) 还有 array('create_time, update_time, 'safe')。

最后一步不是必需的，但它的好处是只验证这些用户可以输入字段规则的习惯。

最后一个需要修改的是，我们应该是更新我们简单的访问规则，来实现只有 admin 角色中的用户可以访问这个方法的需求。这也是为什么我们在 AdminModule::beforeControllerAction()方法中使用 RBAC 模型的原因。我们实际上可以将 accessRules 完全删除。但是，我们要通过更新它以实现要求，所以你在下面可以看到，如何使用访问规则的方法。使用以下的代码替换 SysMessageController::accessRules() 方法：

PHP 代码：

```
public function accessRules()
{
    return array(
        array('allow', // 只允许用户有“admin”的角色才可以进入行动
            'actions'=>array('index', 'view', 'create', 'update',
                'admin', 'delete'),
            'roles'=>array('admin'),
        ),
        array('deny', // 否认所有用户
            'users'=>array('*'),
```

```
), );  
}
```

好了，都改完以后，如果使用 `http://localhost/trackstar/admin/sysMessage/create` 访问新消息输入表单，我们将看到类似下图的界



TrackStar Admin Console

[Back To Main Site](#) [Admin](#) [System Messages](#) [Logout \(Test_User_One\)](#)

[Home](#) » [Sys Messages](#) » Create

Create SysMessage

Fields with * are required.

Message *

Create

在表单中填写 `Hello Users! This is your admin speaking...` 后点击 **Submit** 按钮。应用程序将重定向到类似下面的新建消息的清单页面：



TrackStar Admin Console

[Back To Main Site](#) [Admin](#) [System Messages](#) [Logout \(Test_User_One\)](#)

[Home](#) » [Sys Messages](#) » 1

View SysMessage #1

ID	1
Message	Hello Users! This is your admin speaking...
Create Time	2010-07-06 22:22:18
Create User	1
Update Time	2010-07-06 22:22:18
Update User	1

Operations

- [List SysMessage](#)
- [Create SysMessage](#)
- [Update SysMessage](#)
- [Delete SysMessage](#)
- [Manage SysMessage](#)

12.12 向全体用户显示消息

现在我们已经将消息存入在我们的系统，我们将展示给在应用程序的主页的用户。

12.13 在应用程序级导入新模型类

为了在应用程序中随时访问我们新创建的模型，我们应当将其配置为应用程序的一部分。修改 `protected/config/main.php` 文件内容如下：

PHP 代码:

```
// 半自动生成模型和组件类
'import'=>array(
    'application.models.*',
    'application.components.*',
    'application.modules.admin.models.*',
),
```

12.14 选择最近更新的消息

我们将限制只显示一条消息，并且会根据表中的 `update_time` 来选择。我们希望把它添加到主项目列表页面，所以我们需要修改 `ProjectController::actionIndex()` 方法。通过添加以下代码来改变该方法:

PHP 代码:

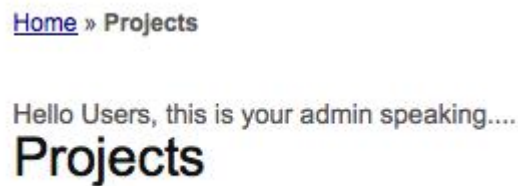
```
public function actionIndex()
{
    $dataProvider=new CActiveDataProvider('Project');
    Yii::app()->clientScript->registerLinkTag( 'alternate', 'application/rss+xml',
    $this->createUrl('comment/feed'));
    //基于 update_time 获得系统最新显示的消息
    $sysMessage = SysMessage::model()->find(array(
    'order'=>'t.update_time DESC',
    ));
    if($sysMessage != null)
        $message = $sysMessage->message;
    else
        $message = null;
    $this->render('index', array(
    'dataProvider'=>$dataProvider,
    'sysMessage'=>$message,
    ));
}
```

现在，我们需要改变我们的视图文件来显示新内容。只需要在 `views/project/index.php` 的 `<h1>Projects</h1>` 上添加下列代码:

PHP 代码:

```
<?php if($sysMessage != null):?>
<div class="sys-message">
<?php echo $sysMessage; ?>
</div>
<?php endif; ?>
```

现在我们只要访问我们的项目列表页面（即我们的应用程序主页），就会看到如下图所显示的内容：



Home » Projects

Hello Users, this is your admin speaking....

Projects

12.15 添加少许的设计调整

好了，这完成了我们想要做的，但是对用户来说，这条消息并不是最完美的，我们可以在我们的 css 主文件中（/themes/new/css/main.css）添加一点片段来做一点修改：

CSS 代码：

```
div.sys-message
{
padding:.8em;
margin-bottom:1em;
border:3px solid #ddd;
background:#9EEFFF;
color:#FF330A;
border-color:#00849E;
}
```

修改完成后，我们的页面消息就更完美了。下图显示了更改以后的消息：

Hello Users, this is your admin speaking....

Projects

可能有人会说，这种设计调整太多了。如果整天盯着这些消息的颜色，用户可能会头痛。我们可以用用一个段小的 **JavaScript** 脚本使消息在五秒后淡出。由于我们只在用户访问该主页我们显示该消息，这样则这可以避免他们盯着看太长时间。

我们将使事情变得更容易，因为 **Yii** 框架包含了强大的 **JavaScript** 框架 **jQuery**。**jQuery** 是一个开源的 **JavaScript** 库，它简化了 **HTML** 文档对象模型（**DOM**）和 **JavaScript** 之间的交互。深入介绍 **jQuery** 的细节不在这本书的范围之内，但访问其文件以稍微熟悉一下其特点，这是非常值得的。由于 **Yii** 包含了 **jQuery**，你可以直接在 **Yii** 的视图文件中使用 **jQuery** 代码，**Yii** 会自动帮你管理好 **jQuery** 库的数据。

我们也将使用该应用程序的辅助组件 **CClientScript** 在生成的网页中来注册 **jQuery JavaScript** 代码。这将确保它被放置在适当的地方，以及也会被适当的标识和格式化。

那么，让我们改变一下之前添加的可以使消息淡出的一个 **JavaScript** 脚本。用下面的代码替换我们刚刚添加到 **views/project/index.php** 中的代码：

PHP 代码：

```
<?php if($sysMessage != null):?>
<div class="sys-message">
<?php echo $sysMessage; ?>
</div>
<?php
    Yii::app()->clientScript->registerScript(
        'fadeAndHideEffect',
        '$(".sys-message").animate({opacity: 1.0}, 5000).fadeOut("slow"); '
    );
endif; ?>
```

现在，如果我们刷新我们的主项目列表页面，我们会看到消息在五秒钟后淡

出。更多的可以轻松地添加到你的网页的，很酷的 jQuery 效果的信息，你可以看看 jQuery 的 API 文档：<http://api.jquery.com/category/effects/>

最后，你可以添加另一个系统信息来确保一切都正常工作。因为该条消息获得了最新的 `update_time` 属性，所以它将成为唯一现实的那一条。

12.16 总结

在本次迭代中，我们介绍了 Yii 模块的概念，并且实践了如何添加 `admin` 模块到网站。我们了解了如何创建一个新的模块，如何使用一个新主题，如何在模块内添加应用程序的功能，甚至如何利用现有的 RBAC 模型在模块内申请一个功能的访问授权。我们还演示了如何使用 jQuery 为我们的应用程序添加 UI 效果。

此外，随着这个管理界面的添加，我们完成了应用程序所有的主要功能。虽然应用程序是非常简单，但我们觉得是时候将它们发布出去了。下一次迭代中我们将聚焦如何将该程序放置到发布环境中。

第十三章：迭代 10：上线/投产准备

尽管我们的应用程序还有很多功能上的不足，(虚构)截止日期的临近和客户对产品投入生产环境感到很焦虑。但在投产之前还有一些事值得做。这正是我们最后一次迭代需要完成的工作。

迭代计划

我们将专注与以下任务，使得我们的应用程序适应生产环境：

- § 启动 Yii 的应用程序日志框架，使得所有致命的错误和事件都被记录下来
- § 启动 Yii 的错误控制框架，使得我们能明白它在开发与生产环境中的不同
- § 启动应用程序数据缓存来帮助增进性能

日志

日志是在应用程序开发的最后一步应该被提起的一个主题。信息，警告，严重错误信息在引起应用程序崩溃时时非常有价值的，在生产环境中它们大多被实际用户使用。

Yii 提供了一个弹性且可扩展的日志功能。日志信息可以依据日志等级和信息类型被分类。通过使用等级和类型过滤器，使得被选中的信息被路由至不同目的地，比如写入磁盘文件，发送至管理员信箱，或者显示在浏览器窗口中。

日志信息

每次请求的时候我们的应用程序都会记录大量信息。当程序被初始化完成后，程序被配置成调试模式，并且在该模式下 **Yii** 框架自身记录日志信息。我们可以查看到该信息，因为默认的这些消息被保存在内存中。因此，它们的生存周期与请求相同。

根目录的 **index.php** 文件中的如下代码，决定了应用程序是否处于调试模式：

PHP 代码：

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

让我们在 **SiteController** 类添加一个小 **action** 来看看被记录的内容，代码如下：

PHP 代码：

```
public function actionShowLog()
{
    echo "Logged Messages: <br><br>";

    var_dump(Yii::getLogger()->getLogs());
}
```

如果我们通过请求：<http://localhost/trackstar/site/showLog> 来调用该 **action**，我们将看到与下面类似的画面：



```
Logged Messages:

array
  0 =>
    array
      0 => string 'Loading "log" application component' (length=35)
      1 => string 'trace' (length=5)
      2 => string 'system.web.CModule' (length=18)
      3 => float 1271216483.7354
  1 =>
    array
      0 => string 'Loading "request" application component' (length=39)
      1 => string 'trace' (length=5)
      2 => string 'system.web.CModule' (length=18)
      3 => float 1271216483.7368
  2 =>
    array
      0 => string 'Loading "urlManager" application component' (length=42)
      1 => string 'trace' (length=5)
      2 => string 'system.web.CModule' (length=18)
      3 => float 1271216483.7379
```

如果我们注释掉在 **index.php** 中定义的全局应用程序条件变量，并刷新页面，我们将看不到日志内容。这是因为系统级调试信息等级事通过 **Yii::trace** 来创建的，只有当应用程序处于该特殊调试模式下才会记录信息。

我们可以通过以下 2 个静态方法中的一个来记录信息：

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

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

好像之前提到的，俩者的唯一区别就是 **Yii::trace** 方法只在调试模式下记录信息。

类型和等级

当记录一个信息，我们需要指定它的类型和等级。类型是表现为 **xxx.yyy.zzz** 格式的类似路径代理的字符串。例如，如果在 **SiteController** 类中记录一条信息，我们可以选择使用

application.controllers.SiteController 作为类型。类型为被记录信息提供了而外的内容。另外当使用 **Yii::log** 为被记录信息指定一个类型时可以同时指定一个等级。等级可以被认为是该消息的缩略。虽然你可以自定义等级，但是一般我们使用以下的一种：

\$ Trace: 这一等级一般被用来基于开发环境的应用程序工作流

\$ Info: 这个是日志的大概内容，而且也是未指定下的默认类型

\$ Profile: 这一等级被用来描述上面提到的性能方面的功能

\$ Warning: 警告信息

\$ Error: 错误信息

添加一个登录日志信息

作为一个例子，让我们为用户登录方法添加一些日志。我们将在该方法的开头添加一些基本调试信息，用来标识该方法正在被执行。然后在登录成功时我们将记录一个信息性的消息，同样的在登录失败时记录一个警告。按照如下代码修改我们的 **SiteController::actionLogin()** 方法：

PHP 代码：

```
/**
 * Displays the login page
 */

public function actionLogin()
{
    Yii::app()->language = 'rev';

    Yii::trace("The actionLogin() method is being requested",
        "application.controllers.SiteController");
```

```
if(!Yii::app()->user->isGuest)

{

    $this->redirect(Yii::app()->homeUrl);

}


$model=new LoginForm;


// if it is ajax validation request

if(isset($_POST['ajax']) && $_POST['ajax']== 'login-form')

{

    echo CActiveForm::validate($model);

    Yii::app()->end();

}


// collect user input data

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

{

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

    // validate user input and redirect to the previous page if valid

    if($model->validate() && $model->login())

    {

        Yii::log("Successful login of user: " . Yii::app()->user->id,

            CLogger::INFO, "application.controllers.SiteController");

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

    }

}
```

```

    }

    else {

        Yii::log("Failed login attempt", "warning", "application.
        controllers.SiteController");

    }

}

// display the login form

//public string findLocalizedFile(string $srcFile, string $srcLanguage=NULL, string
$language=NULL)

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

```

如果我们成功登录(或者进行了一次错误的尝试)，然后访问我们的日志，我们看不到它们(如果你注释掉了调试模式申明，请确保你的应用程序已经开启调试模式)。再一次，原因是默认的，日志会存储在内存中。在请求完成后它们就会消失。这会显得很没价值。我们需要将其导航至持久存储的地方，这样我们才可以在请求之后查看我们生成的日志。

消息路由

如我们所说，默认的，**Yii::log** 和 **Yii::trace** 将信息存储在内存中。一般的，将这些消息显示在浏览器窗口中会使得它们非常有价值，或者存储到一些持久化介质中，或者是发送 **email**，甚至是数据库中。**Yii** 的消息路由功能允许将日志信息路由至不同目的地。

在 **Yii** 中消息路由是由 **CLogRouter** 应用程序组件管理的。它允许你定义一系列消息目的地。

为了利用消息路由的功能，我们需要在 **protected/config/main.php** 中配置 **CLogRouter** 应用程序组件。我们通过设置它的 **routes** 属性为需要的路由地址来完成该动作。

如果我们打开配置文件，我们将看到一些预先设置好的配置信息(是在使用 **yiic** 的 **webapp** 命令生成应用程序时生成的)。下面是默认定义在应用程序中的内容：

PHP 代码：

```

'log'=>array(

    'class'=>'CLogRouter',

```

```

        'routes' => array(

            array(

                'class' => 'CFileLogRoute',

                'levels' => 'error, warning',

            ),

            // uncomment the following to show log messages on web pages

            /*

            array(

                'class' => 'CWebLogRoute',

            ),

            */

        ),

    ),

```

路由应用程序组件被配置为使用框架的 **CLogRouter** 类。你可以使用自定义的类如果你的路由需求无法使用框架基本类来实现，但是在这里，不需要做改变。

在该类定义的下面定义了 **routes** 属性。在这个例子中，只定义了一个路由。它使用了 **Yii** 框架的消息路由类 **CFileLogRoute**。**CFileLogRoute** 消息路由类使用文件系统来保存消息。默认的，文件被保存至 **/protected/runtime/application.log** 中。事实上，如果你一直跟随着我们的教程并且完成了自己的应用程序，你可以打开该文件看看 **yii** 为你保存的消息。只有 **error** 和 **warning** 类型的消息将被路由至该文件。前面代码中注释掉了另外一个路由类 **CWebLogRoute**。如果使用它，消息将会显示在当前页面。下面是一个 **Yii 1.1** 版的消息路由列表：

- § **CDbLogRoute**: 将消息存储到数据库表中
- § **CEmailLogRoute**: 将消息发送至特定的 e-mail 地址
- § **CFileLogRoute**: 将消息保存至应用程序 **runtime** 文件夹下的一个文件中
- § **CWebLogRoute**: 在当前页面的结尾显示消息
- § **CProfileLogRoute**: 在当前页面结尾显示 **profiling** 消息

添加日志至 **SiteController::actionLogin()** 方法中使用了 **Yii::trace** 和 **Yii::log**。当使用 **Yii::trace** 时，日志的等级被自动设置为 **trace**。使用 **Yii::log** 时，如果登录成功，我们将等级设置为 **info**，如果失败则设置为 **warning** 级。让我们修改配置文件，将 **trace** 和 **info** 级消息写入一个叫 **infoMessages** 的新文件，

该文件与 **application.log** 在同一文件夹。同时也配置其将 **warning** 消息显示在浏览器中。为了完成这一操作，我们对配置文件进行如下修改：

PHP 代码：

```
'log'=>array(

    'class'=>'CLogRouter',

    'routes'=>array(

        array(

            'class'=>'CFileLogRoute',

            'levels'=>'error',

        ),

        array(

            'class'=>'CFileLogRoute',

            'levels'=>'info, trace',

            'logFile'=>'infoMessages.log',

        ),

        array(

            'class'=>'CWebLogRoute',

            'levels'=>'warning',

        ),

    ),

),
```

此时，在我们保存修改后，让我们看看改变。首先，进行一次成功登录。应用程序会因此写入 **2** 条消息到 **/protected/runtime/infoMessages.log** 文件(为新建文件)，第一条为 **trace**，后面一条为成功登录。成功登录后，文件内容如下(由于篇幅关系并未全部显示)：

SHELL 代码或屏幕回显：

```
...
```

```
2010/04/15 00:31:52 [trace] [application.controllers.SiteController] The actionLogin() method
is being requested

2010/04/15 00:31:52 [trace] [system.web.CModule] Loading "user" application component

2010/04/15 00:31:52 [trace] [system.web.CModule] Loading "session" application component
2010/04/15 00:31:52 [trace] [system.web.CModule] Loading "db" application component

2010/04/15 00:31:52 [trace] [system.db.CDbConnection] Opening DB connection

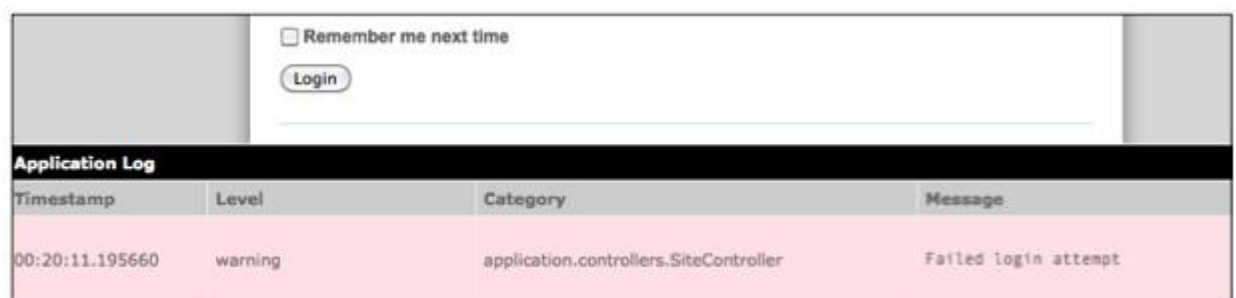
...

2010/04/15 00:31:52 [info] [application.controllers.SiteController] Successful login of user:
1

...
```

哇哦，这里的消息绝对不是 **2** 条。但是其中一定有我们写入的 **2** 条。它们在上面的列表中被加粗显示。现在开始，因为我们将 **trace** 路由至该新建文件，所以基于框架的所有 **trace** 都会出现在这里。这很好的帮助我们弄懂了应用程序中一条请求的生存周期。它比表面上看到的要多的多。在进入生产后，我们应该取消这一设置。在非调试模式，我们将只能看到 **info** 级的日志。不过这一等级的细节对追踪 **bug** 和弄清应用程序正在做什么非常重要。知道它在那里无论是否需要是让人很舒服的。

下面让我们看下登录失败的场景。现在让我们登出，并再次登录，但是这次使用错误的帐号密码使得登录失败，我们将在返回页面的底部看到 **warning** 级的消息，这证明我们的配置起作用了。下图显示了 **warning** 级消息显示的样子：



Application Log			
Timestamp	Level	Category	Message
00:20:11.195660	warning	application.controllers.SiteController	Failed login attempt

当我们使用 **CLogRouter** 来路由消息，日志文件被保存在 **logPath** 属性指定的目录下，并且文件名被指定为 **logFile**。日志路由的另一个好处是自动轮换日志文件。如果日志文件的大小超过了 **maxFileSize**(单位 **KB** 千字节)属性中设置的值，轮换就被执行一次，当前日志文件被重命名为添加后缀 **1**。所有存在的日志文件都会被默认向后更改文件名，例如 **.2** 到 **.3**, **.1** 到 **.2**。 **maxLogFiles** 属性会被用来指定最大存在文件数量。

错误控制

合适的控制随机出现在软件程序中的错误是非常重要的。再次提示，这个主题应该出现在编写我们的应用程序之前，而不是当前的最后一阶段。幸运的是，因为我们使用了一些 **Yii** 自动生成的代码，我们的应用程序已经应用了一些 **Yii** 的差错控制功能。

Yii 提供一个完整的基于 **PHP5** 异常体系的错误控制框架，一个内置的机制通过关键点来控制程序错误。当主 **Yii** 应用程序组件被建立用来接收进入的用户请求，它注册了 **CApplication::handleError()** 方法来控制 **PHP** 的 **warnings** 和 **notices**。它注册了 **CApplication::handleException()** 方法来控制未捕获的 **PHP** 异常。也就是说，在应用程序执行时一个 **PHP warning/notice** 或者一个未捕获的 **PHP** 异常发生了，上面一种错误控制将进行控制，并且执行所需的错误控制流程。

错误控制器的注册在应用程序的构造函数中，通过调用 **PHP** 的 **set_exception_handler** 和 **set_error_handler** 方法来实现。如果你不想让 **Yii** 来控制这种类型的错误和异常，你可以通过定义一个全局常量 **YII_ENABLE_ERROR** 和 **YII_ENABLE_EXCEPTION_HANDLER** 的形式来重写默认行为在主 **index.php** 脚本中未 **false**。

默认的，应用程序将使用框架的 **CErrorHandler** 类作为一项应用程序组件任务来控制 **PHP** 错误和未捕获异常。这一应用程序组件的其中一部分任务就是使用适当的 **view** 文件来显示错误，不管应用程序处于调试模式或生产模式。这将允许你在不同环境下自定义错误信息。在开发模式下显示更多的(冗长的)错误信息是有必要的，可以帮助定位问题。但是允许用户在生产模式下观看该信息可能会引起安全问题。同时，如果你在你的网站里实行多语言化，**CErrorHandler** 将选择最合适的语言来显示错误。

在 **Yii** 中抛出异常的方式与在 **PHP** 中抛出异常相同。使用下面的语法：

PHP 代码：

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

Yii 提供下面 2 个异常类：

§ CException

§ CHttpException

CException 是一个一般的异常类。**CHttpException** 描述了将会显示给最终用户的异常。

CHttpException 同时带有 **statusCode** 属性，来描述 **HTTP** 状态码。使用不同的异常类抛出的异常，显示错误方式不同。

显示错误

如前所述，当一个错误被提交至 **CErrorHandler** 应用程序组件，该组件会决定使用哪个 **view** 文件来显示错误。如果是一个显示给最终用户的错误，类似使用 **CHttpException** 的，默认使用一个名称为 **errorXXX** 的视图文件，其中的 **xxx** 为 **HTTP** 响应的状态码(例如，**400**，**404**，**500**)。如果是一个内部事务并且只向开发人员显示，则使用 **exception** 视图文件。当应用程序处于调试模式，一个完整的堆栈调用和错误行在源文件中的位置将被显示。

遗憾的是，故事并未结束。当应用程序运行在生产模式，所有的错误将被使用 **errorXXX** 视图文件显示出来。这是因为包含堆栈调用的信息很敏感，不应该在向最终用户显示。

当应用程序处于生产模式，开发人员应当通过错误日志来了解更多的错误信息。级别为 **error** 的消息总是会被存储到日志中的。如果一个错误由 **PHP warning** 或 **notice** 引起，消息将会同 **category php** 一起存储起来。如果错误由未捕获异常引起，类型将会是 **exception.ExceptionClassName**，异常类名将会是 **CHttpException** 或 **CException** 或者它们子类中的一个。如前所述，使用日志的一个好处就是用来监控处于生产状态的应用程序中的错误。

默认的，**CErrorHandler** 按照以下的顺序搜索相关 **view** 文件：

- 1.**WebRoot/themes/ThemeName/views/system**:当前主题下的系统视图文件
- 2.**WebRoot/protected/views/system**:某一应用程序的默认系统视图文件
- 3.**YiiRoot/framework/views**:Yii 框架提供的标准系统视图文件夹

所以你可以通过修改基于应用程序或主题下的系统 **view** 文件夹来自定义错误视图显示错误。

Yii 也允许你定义特殊的控制器方法来控制错误的显示。也就是我们如何配置应用程序。我们将通过例子来观察。

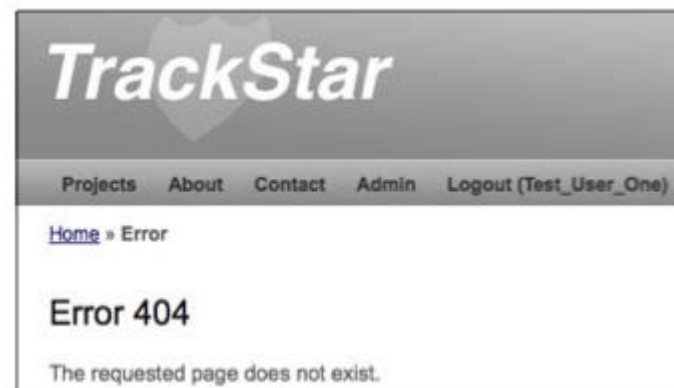
让我们看一些相关的例子。下面一些通过 **Gii** 生成的代码包含了 **Yii** 的错误控制。其中的一个例子就是 **ProjectController::loadModel()** 方法。其代码如下：

PHP 代码：

```
public function loadModel ()
{
    if($this->_model===null)
    {
        if(isset($_GET['id']))
        {
            $this->_model=Project::model()->findbyPk($_GET['id']);
        }
        if($this->_model===null)
        {
            throw new CHttpException(404,'The requested page does not exist.');
```

我们可以看到，以上代码通过输入的 **id** 值来尝试加载一个 **Project AR** 模型类的实例。如果无法定位请求的 **project** 类，则抛出一个 **CHttpException** 异常，让用户明白所请求的页面不存在。我们可以通过一个

不存在的 ID 在我们的浏览器中对其进行测试。我们知道我们的 ID99 的 project 不存在，如果使用 <http://localhost/trackstar/project/view/id/99> 来进行请求，将会看到如下的画面：



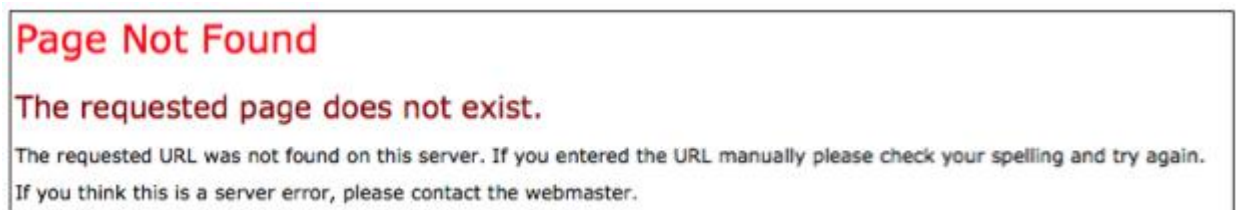
这非常好，因为所有的页面都拥有相同的主题，页头，页脚，等等。这并非默认行为对错误页面的渲染。最初的应用程序被配置为使用一个特殊的控制器方法来控制相关错误。这就是我们提供的在应用程序中控制错误的另一种方法。如果我们观察一下配置文件，我们会看到下面的代码：

PHP 代码：

```
'errorHandler'=>array(  
  
    // use 'site/error' action to display errors  
  
    'errorAction'=>'site/error',  
  
),
```

该配置表示应用程序组件使用 `SiteController::actionError()` 方法来控制所有准备向用户显示的异常。如果我们观察一下该方法，我们会发现它渲染了 `protected/views/site/error.php` 视图文件。这是一个普通的视图文件，所以它也会渲染相关的布局和主题文件。基于以上的原因，当错误发生的时候我们可以给用户提供一个友好的界面。

为了在不修改配置的情况下观察默认行为，让我们临时注销上面的配置代码(在 `protected/config/main.php`)，然后再请求一次不存在的页面。我们将会看到如下反馈：



因为我们没有指定任何自定义错误页面，所以这是 Yii 框架自身的 `error404.php` 文件。

让我们继续，恢复对配置文件的修改，继续使用 `SiteController::actionError()` 方法控制错误。

让我们看一下 **CException** 与 **HTTP** 异常类抛出异常的区别。让我们注销 **HTTP** 相关代码，添加其他异常类，具体代码如下：

PHP 代码：

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

            if($this->_model === null)
            {
                //throw new CHttpException(404, 'The requested page does not exist. ');

                throw new CException('The is an example of throwing a CException');
            }
        }

        return $this->_model;
    }
}
```

现在如果我们请求一个不存在的 **project**，我们看到的情况会大有不同。这次我们看到了一个包含全部堆栈跟踪跟踪错误信息的系统生成的错误页面同时也包含了出错的文件信息。这个画面有点太长不适合展示出来，但是一定会包含一个 **CException** 抛出的，描述信息为 **'The is a example of throwing a CException'** 的源文件和堆栈信息。

因此在调试模式抛出不同的异常类，会有不同的结果。这是一类在开发环境中，为我们提供找到问题根源的信息。让我们在 **index.php** 中暂时注销调试模式设置，来看看在生产模式下的情况：

PHP 代码：

```
// remove the following line when in production mode

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

注销以后，如果通过刷新再次访问不存在的 **project**，将会看到如下图的 **HTTP 500** 错误：



所以没有任何敏感信息会在生产模式暴露给用户。

缓存

缓存数据可以很好的帮助在生产状态下的 **web** 应用程序提高性能。如果这是一个不需要每次请求都发生改变的特殊内容，使用缓存来存储和提供这一内容，可以节约重新生成的时间。

在缓存方面 **Yii** 提供了一些不错的功能。**Yii** 的缓存教程将从配置应用程序缓存组件开始。该组件是众多 **CCache** 子类中的一个，只是存储实现不同。

Yii 提供了很多基于不同存储介质实现缓存的缓存组建类。下面是 **Yii 1.1.2** 所支持的缓存实现列表：

- § **CMemCache**：使用 **PHP memcache** 扩展。
- § **CApcCache**：使用 **PHP APC** 扩展。
- § **CXCache**：使用 **PHP XCache** 扩展。
- § **CEAcceleratorCache**：使用 **PHP EAccelerator** 扩展。
- § **CDbCache**：使用数据库表存储缓存数据。默认的，它将在 **runtime** 文件夹下建立一个 **SQLite3** 数据库。你可以通过 **connectionID** 属性指定数据库。
- § **CZendDataCache**：使用 **Zend Data Cache** 作为优先的缓存媒介。
- § **CFileCache**：使用文件来存储缓存数据。特别适合存储类似页面的大块数据。
- § **CDummyCache**：这里实现一般的缓存接口，但不做任何实际缓存。这一实现的原因是你需要执行缓存，但是你的开发环境不支持缓存。这允许你在实现真正缓存之前持续编码。你不需要为缓存环境而改变编码。

所有的这些组件都扩展自 **CCache**，并且提供相同的接口。也就是说你可以改变不同的缓存策略而不需要改变代码。

配置缓存

如前面所说，在 **Yii** 中使用缓存，需要选择一个缓存组件，然后在应用程序配置文件 (**/protected/config/main.php**) 中进行配置。不同的缓存实现决定了不同的配置。如果要使用 **memcached** 就要使用 **CMemCache** 类，是一个分布式内存对象缓存系统，允许你使用多台主机进行缓存，如果要使用 2 台服务器应该如下配置：

PHP 代码:

```
array(

    .....

    'components'=>array(

        .....

        'cache'=>array(

            'class'=>'system.caching.CMemCache',

            'servers'=>array(

                array('host'=>'server1', 'port'=>12345, 'weight'=>60),

                array('host'=>'server2', 'port'=>12345, 'weight'=>40), ),

            ),

        ),

    );
```

为了使一切对于一直跟着 **TrackStar** 开发的读者相对简单,我们将使用基于 **CFileCache** 的文件系统缓存,我们将看一些例子。在所有的允许对文件系统进行读写操作的开发环境中都可以实现。

如果由于一些原因你无法这样做,但是你又想跟随本教程,请使用 **CDummyCache**。和之前说的一样,它不会存储任何实质数据到缓存中,但是可以使代码正常运行。

CFileCache 提供了一个基于文件系统的缓存结构。当使用它的时候,每一个数据值都会被缓存并存储在一个独立的文件中。默认的,这些文件被存储在 `/protected/runtime/cache` 文件夹下,但是可以通过修改 `cachePath` 属性来轻松改变这一位置。对我们来说默认已经 **ok** 了,所以我们只需要在配置文件 `/protected/config/main.php` 中进行如下配置:

PHP 代码:

```
// application components

'components'=>array(

    ...

    'cache'=>array(

        'class'=>'system.caching.CFileCache',
```

```
    ),  
  
    ...  
  
    ),
```

当上面的都完成之后，我们可以在应用程序的任何位置通过 `Yii::app()->cache` 来访问这一新组件。

使用基于文件的缓存

让我们试试这一新组件吧。还记得我们在上一次迭代中添加的基于管理功能的系统消息吗？比起每一次请求时都从数据库读取相关信息，让我们对该值进行缓存来节约时间，所以无需每一次请求都从数据库获取信息。

让我们在 **SysMessage AR** 模型类添加一个新公共方法来控制最后一次获取系统消息。让我们将该方法设置成 **public** 和 **static**，这样无需实例化 **SysMessage** 就可以调用该方法。在测试中这也是非常有用的。

测试？你可能觉得我们放弃了测试先导的开发模式。实际上不是，让我们进行一下测试。

建立一个新的测试文件，**protected/tests/unit/SysMessageTest.php**，并且为其添加下面的部件定义和测试方法：

PHP 代码：

```
<?php  
  
class SysMessageTest extends CDbTestCase  
{  
  
    public function testGetLatest()  
    {  
  
        $message = SysMessage::getLatest();  
  
        $this->assertTrue($message instanceof SysMessage);  
  
    }  
  
}
```

在命令行中运行该测试，会因为我们没有添加方法而立刻得到错误提示。让我们向 **SysMessage** 类添加如下方法：

PHP 代码：

```

/**
 * Retrieves the most recent system message.
 *
 * @return SysMessage the AR instance representing the latest system message.
 */

public static function getLatest()
{
    //see if it is in the cache, if so, just return it

    if( ($cache=Yii::app()->cache)!=null )
    {
        $key='TrackStar.ProjectListing.SystemMessage';

        if(($sysMessage=$cache->get($key))!==false)

            return $sysMessage;
    }

    //The system message was either not found in the cache, or
    //there is no cache component defined for the application

    //retrieve the system message from the database

    $sysMessage = SysMessage::model()->find(array(

        'order'=>'t.update_time DESC',

    ));

    if($sysMessage != null)
    {
        //a valid message was found. Store it in cache for future retrievals

        if(isset($key))

```

```

        $cache->set($key,$sysMessage,300);

        return $sysMessage;

    }

    else

        return null;

}

```

稍候我们就会谈论一下相关细节。首先让我们使测试通过。即使上面的改动都完成了，如果你再次运行测试，我们面对的仍然是错误。但是这次，错误的原因是我们的方法返回了 **null**，我们的测试为一个非空返回值。返回 **null** 的原因是数据库中没有任何系统信息。请记住我们的测试都是基于 **trackstar_test** 数据库的。**ok**，没其他问题的话，使用固件来解决它。在 **protected/tests/fixtures/tbl_sys_message.php** 添加内容如下的固件文件：

PHP 代码：

```

<?php

return array(

    'message1' => array(

        'message' => 'This is a test message',

        'create_time' => new CDbExpression(' NOW()' ),

        'create_user_id' => 1,

        'update_time' => new CDbExpression(' NOW()' ),

        'update_user_id' => 1,

    ),

);

```

请确保用于测试的 **SysMessageTest** 类被配置使用固件：

PHP 代码：

```

public $fixtures=array(

    'messages' => 'SysMessage',

```

```
);
```

好，现在是时候再次运行测试了，成功是必然的。该方法将会尝试从缓存获取消息。但是因为这是第一次基于测试环境的请求，缓存里没有数据。所以将会从数据库中获取数据然后将结果存储到缓存中方便后续请求使用。

如果我们对默认缓存文件夹(`/protected/runtime/cache/`)做一个列表，会发现一个名字很奇怪的文件(可能会略有不同)：

8b22da6eaf1bf772dae212cd28d2f4bc.bin

如果使用文本编辑器打开，会看到如下内容：

SHELL 代码或屏幕回显：

```
a: 2: {i: 0; 0: 10: "SysMessage": 11: {s: 18: "CActiveRecord_md"; N; s: 19: "18 CActiveRecord_new"; b: 0;
s: 26: "CActiveRecord_attributes"; a: 6: {s: 2: "id"; s: 1: "1"; s: 7: "message"; s: 22: "This is a test
message";

s: 11: "create_time"; s: 19: "2010-07-08
21: 42: 00"; s: 14: "create_user_id"; s: 1: "1"; s: 11: "update_time";

s: 19: "2010- 07-08
21: 42: 00"; s: 14: "update_user_id"; s: 1: "1"; }s: 23: "18CActiveRecord18_related"; a: 0: {}s: 17: "CActiv
eRecord_c";

N; s: 18: "CActiveRecord_pk"; s: 1: "1"; s: 15: "CModel_errors"; a: 0: {}s: 19: "CModel_validators";

N; s: 17: "CModel_scenario"; s: 6: "update"; s: 14: "CComponent_e"; N; s: 14: "CComponent_m"; N; }i: 1; N; }
```

这是一个被序列化的关于我们最新从数据库中取回的 **SysMessage AR** 类实例的缓存值，正是我们希望在那的。这样说来，缓存正常工作。

当运行测试时，针对数据库在测试环境运行应用程序，我们可能会希望将缓存数据配置到不同的位置。如果是这样，你需要在我们测试配置文件 `protected/config/test.php` 添加一个略有不同的缓存组件。例如，如果我们需要指定一个不同的文件夹来存放缓存数据，我们需要向测试配置文件添加如下内容：

```
'cache' => array(

    'class' => 'system.caching.CFileCache',

    'cachePath' => '/Webroot/trackstar/protected/runtime/cache/test',

),
```

这样，当正常使用项目的时候不会对我们的测试数据造成影响。

让我们稍微仔细分析一下上面 `SysMessage::getLatest()` 方法。第一行是用来检测请求的数据是否在缓存中，如果是则返回值：

PHP 代码：

```
//see if it is in the cache, if so, just return it

if( ($cache=Yii::app()->cache)!=null)

{

    $key='TrackStar.ProjectListing.SystemMessage';

    if(($sysMessage=$cache->get($key))!==false)

        return $sysMessage;

}
```

正如我们所说的，我们配置缓存应用程序组件在应用程序中的任何位置都可以通过 `Yii::app()->cache` 来访问。所以，首先代码检查了该组件是否被定义。如果返回 `true`，则通过 `$cache->get($key)` 方法来尝试从缓存获取数据。这或多或少是你所期盼的。其通过一个特殊的 **key** 来从缓存获取数据。该 **key** 是一个特殊的标识字段，用来指向缓存中存储的一部分数据。在我们的系统信息的例子里，我们只需要一次显示一条消息，因此针对一条消息显示的 **key** 会非常简单。**key** 可以是任何字符串，因为它代表缓存里我们需要的一块唯一数据。这里我们选择了具有描述性的字符串 `TrackStar.ProjectListing.SystemMessage` 作为 **key** 来存储和获取我们缓存的系统消息。

当这段代码第一次执行的时候，并没有任何与之相关的数据在缓存中。因此，调用包含这个 **key** 的 `$cache->get()` 将返回 `false`。所以会继续执行下面的代码，使用 **AR** 类从数据库尝试取出适当的系统消息：

PHP 代码：

```
$sysMessage = SysMessage::model()->find(array(

    'order'=>'t.update_time DESC',

));
```

接下来执行下面的代码，首先检查是否从数据库获得任何数据。如果得到了，在返回数据前将其存入缓存，反之返回 `null`。

PHP 代码：

```
if($sysMessage != null)

{
```

```

        if(isset($key))

            $cache->set($key, $sysMessage->message, 300);

        return $sysMessage->message;
    }

    else

        return null;

```

如果一个正确的系统消息被返回，我们就使用 `$cache->set()` 方法将其存入缓存。该方法格式如下：

PHP 代码：

```

set($key, $value, $duration=0, $dependency=null)

```

当将一段数据放入缓存时，必须设置一个唯一的 **key**，用来存储数据。**key** 就是上面提到的唯一的字符串值，**value** 就是希望存入缓存的数据值。可以是任何格式的，唯一的要求是可以被序列化。**duration** 参数是一个指定生存周期的可选项。确保缓存中的数据定时被刷新。默认值为 **0**，也就是说数据永远不会过期，将永远存在缓存中。（实际上，在 **Yii** 中将小于等于 **0** 的持续时间理解为 **1** 年后过期。所以，并非永远，只是一个很长的时间。）

我们以下面的形式调用 `set()` 方法：

PHP 代码：

```

$cache->set($key, $sysMessage->message, 300);

```

我们将 **key** 的值设置为我们之前定义的 **TrackStar.ProjectListing.SystemMessage**，被存储的数据就是 **SystemMessage AR** 类返回的值，也就是表 **tbl_sys_message** 中的 **message** 栏，并且设置过期时间为 **300** 秒。因此数据会每 **5** 分钟从数据库读取一次来做更新。我们没有指定 **dependency**。我们将在下面讨论这一可选参数。

缓存 dependencies(依赖)

dependency 参数允许以选择并且更复杂的形式来决定存储在缓存中的数据是否需要刷新。并非简单的定义一个过期时间，你的缓存策略可能为在特殊用户发出请求后缓存数据变为非法，或者是应用程序的状态，或者是文件系统中的文件最近刚被更新过。该参数允许你指定相关的缓存校验规则。

dependency 是 **CCacheDependency** 或其子类的一个实例。**Yii** 允许下面的缓存依赖：

§ **CFileCacheDependency**：缓存数据将在特殊文件最后更新时间自上次检查后发生改变时变为非法。

§ **CDirectoryCacheDependency**：类似上面的文件缓存依赖，不过该方法将检查指定文件夹下的所有文件和子文件夹。

§ **CDbCacheDependency**：缓存中的数据将在自上次检查后相同语句在数据库中查询结果发生改变后变得非法。

§ **CGlobalStateCacheDependency**：缓存中的数据将在特殊标识全局状态值发生改变后变得非法。全局状态是应用程序中的一个在多页面多请求中持久保存的一个变量。通过调用 **CApplication::setGlobalState()** 来设置。

§ **CChainedCacheDependency**：该项允许你将多个缓存依赖链接起来。当链上的任一缓存依赖发生改变时，缓存中的值变为非法。

§ **CexpressionDependency**：当特殊 **php** 表达式的结果发生改变时缓存中的数据变为非法。

为了提供一个混合的例子，让我们使用一个依赖，使得在表 **tbl_sys_message** 中的数据发生改变后缓存中的数据过期。不是武断的每 5 分钟过期一次，我们希望过期发生在真正需要的时候，也就是，在系统消息表中的某一条消息的 **update_time** 发生改变时。我们将使用 **CDbCacheDependency** 来实现它，因为它正是被设计为基于 **SQL** 语句返回结果来检验缓存合法的。

我们修改我们的 **set** 方法调用，设置持续时间为 0，所以它不会基于时间过期，同时传入一个新的基于特定 **SQL** 语句的依赖实例：

PHP 代码：

```
$cache->set($key, $sysMessage->message, 0,  
  
    new CDbCacheDependency('select id from tbl_sys_message order by update_ time desc'));
```

将持续时间改为 0 并非使用依赖的先决条件。我们可以将持续时间保持为 300。这只会增加一条缓存合法的验证规则。缓存中的数据会按照 5 分钟为最大值来完成校验，但是当小于 5 分钟的情况下，**update_time** 发生了改变也会触发缓存更新。

当上面的都完成后，缓存只会在查询语句返回结果改变时进行更新。但这个例子有一点奇怪，因为我们最初使用缓存就是为了减少数据库查询次数。但是现在我们设置其在每次读取缓存数据前都查询一次数据库。但是如果缓存的数据足够复杂，这样一个简单的 **sql** 语句做缓存刷新判断就有一定价值了。使用什么样的缓存刷新规则，完全基于应用程序的需要。但是 **Yii** 确实提供了多种选择来帮助我们达到各种需求。

为了完成我们对应用程序使用缓存的计划，我们仍然需要对 **ProjectController::actionIndex()** 方法进行更新。非常简单，只需将从数据库读取系统消息更换为调用刚刚新建的方法，将 **ProjectController::actionIndex()** 方法中的：

PHP 代码：

```
$sysMessage = SysMessage::model()->find(array('order' => 't.update_time DESC',));
```

更换为：

PHP 代码：

```
$sysMessage = SysMessage::getLatest();
```

现在在项目列表页显示的系统消息就是被文件缓存后的。

Fagment(片段)缓存

前面的例子示范了缓存的使用。简单的说就是从数据库中读取一段数据，然后存入缓存中。在 **Yii** 中还有其他可行的方法来存储有 **view** 脚本生成的页面的一部分，或者整个页面。

片段缓存用来缓存一个页面的一部分。我们可以在 **view** 脚本中使用片段缓存。我们通过使用 **CController::beginCache()** 和 **CController::endCache()** 方法来实现。这 2 个方法用来在渲染页面中标识内容中需要被缓存的部分。只是作为一个缓存练习的例子，我们需要一个唯一的 **key** 来标识被缓存的内容。一般来说，应该在 **view** 文件中按照如下的形式使用片段缓存：

PHP 代码：

```
...some HTML content...

<?php if($this->beginCache($key)) { ?>

...content to be cached...

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

...other HTML content...
```

如果调用 **beginCache()** 返回 **false**，缓存中的内容将会被自动插入指定位置。反之，如果 **if** 语句返回 **true**，**endCache()** 之前的内容将被缓存。

声明片段缓存可选参数

当调用 **beginCache()** 方法时，我们提供一个数组作为第二个参数，该数组包含了自定义片段缓存的可选项。事实上，**beginCache()** 和 **endCache()** 方法是一个方便的 **COutputCache** 过滤器/部件的包裹器。因此，这里的缓存选项可以为任意 **COutputCache** 的属性。

无可厚非的一个最常用属性就是持续时间，即指定了缓存中的内容保留的时间。它很像我们在之前系统消息缓存中使用的 **duration** 参数。你可以按照如下格式为 **beginCache()** 提供 **duration** 参数：

PHP 代码：

```
$this->beginCache($key, array('duration' => 3600))
```

片段缓存的默认设置与数据缓存不同。如果我们不设置持续时间，默认为 **60** 秒，也就是 **60** 秒后更新。当使用片段缓存时有很多可选参数。具体请通过 **API** 地址查看 **COutputCache** 部分

<http://www.yiiframework.com/doc/guide/caching.fragment>

使用片段缓存

让我们将其运用到我们的 **TrackStar** 应用程序中。我们将再次关注项目列表页。你或许能记得，在页面的底部，为用户留下的针对相关 **project** 中 **issue** 的评论。这个列表指出了谁在某一个 **issue** 留下了评论。无需每次请求时都重新生成，让我们使用片段缓存来实现每 **2** 分钟刷新一次。应用程序会的数据会有一些陈旧，**2** 分钟也并非等待评论列表更新所必须的时间。

为了做到这个，在 **view** 文件 **/protected/views/project/index.php** 进行修改。我们将使用片段缓存来缓存最近评论 **portlet**：

PHP 代码：

```
<?php

$key = "TrackStar.ProjectListing.RecentComments";

if($this->beginCache($key, array('duration'=>120))) {

    $this->beginWidget('zii.widgets.CPortlet', array(

        'title'=>'Recent Comments',

    ));

    $this->widget('RecentComments');

    $this->endWidget();

    $this->endCache();

}

?>
```

代码修改完后，当我们第一次访问项目列表页，将会在缓存中存入我们的评论列表。如果我们在 **2** 分钟内添加一个评论，然后返回项目列表页，我们将无法看到最新添加的评论。不过如果我们不断刷新页面，缓存过期后数据将被刷新，而我们将在列表中看到我们的新评论。

你可以简单的添加一个 **echo time()**；PHP 语句到被缓存的内容中，来看过期时间是否准确。在缓存数据被刷新前，显示的时间不会更新。当使用文件缓存时，请确保你的 **/protected/runtime/** 文件夹是可以被 **web** 服务器进程写入的，当然是在你使用默认缓存储存路径的情况下。

页面缓存

作为片段缓存的扩充，Yii 提供了对整个页面进行缓存的选项。页面缓存和片段缓存很相似。然而，因为页面的内容经常是一个 **view** 文件调用布局文件的结果，所以我们无法简单的在布局文件中调用 **beginCache()** 和 **endCache()** 方法。原因是布局被应用到 **CController::render()** 方法中的 **view** 内容生成时。所以我们将错过从缓存获取数据的机会。

所以，想要缓存整个页面，我们需要跳过页面生成的过程。为了达到这一点，我们可以使用 **COutputCache** 类作为一个我们控制器类中的一个行为过滤器。

让我们拿出一个例子。让我们使用页面缓存的方法来缓存所有项目细节页。在 **TrackStar** 中，通过 URL **http://localhost/trackstar/project/view/id/[id]** 来请求项目细节页，其中 **[id]** 是所请求的项目 **id**。我们想做的是设置一个页面缓存过滤器，按照 **ID** 将页面缓存起来。当缓存内容时我们需要将项目 **id** 混合到 **key** 的值中去。也就是说，我们不希望请求项目 **id** 为 **1** 的项目细节时缺得到项目 **2** 的缓存。幸运的是，**COutputCache** 过滤器已经完成了这一工作。

打开 **protected/controllers/ProjectController.php**，并且将已经存在的 **filters()** 方法修改为如下形式：

PHP 代码：

```
public function filters()
{
    return array( 'accessControl', // perform access control for CRUD operations
        array(
            //cache the entire output from the actionView() method for 2 minutes
            'COutputCache + view',
            'duration' => 120,
            'varyByParam' => array('id'),
        ),
    );
}
```

该 **filter** 配置为利用 **COutputCache** 过滤器来缓存整个被应用程序调用 **ProjectController::actionView()** 方法生成的内容。**+ view** 添加在 **COutputCache** 后，如你所能记得的，是对特殊方法添加过滤器的标准的方式。持续时间被设置为页面生成后的 **2** 分钟。

varyByParam 是我们之前提起过的一种很重要的可选项。为了给你减轻压力，减少编写标识被缓存内容 **key** 的工作量，该功能允许 **key** 被框架自动控制。也就是说，通过指定一系列来自 **GET** 参数中的名称。当我们开始通过 **project_id** 请求 **project** 时，系统将会很好的使用该 **id** 作为唯一 **key** 的一部分来生成缓存内容。通过指定 '**varyByParam**'=>**array('id')**，**COutputCache** 为我们完成了基于输入请求字符串中 **id** 的余下工作。这里有很多可选项，在我们使用 **COutputCache** 来缓存数据时，来作为自动生成 **key** 名字的策略。下面是可以使用的一个列表：

§ **varyByRoute**: 通过将该选项设置为 **true**，具体请求的路由部分将会作为独立标识符的一部分用于生成缓存数据。所以，你可以使用请求 **controller** 和 **action** 的组合来区别缓存内容。

§ **varyBySession**: 通过设置该选项为 **true**，将使用唯一的 **session id** 来区分缓存中的内容。每个用户的 **session** 都是不同的，但是可以用来为缓存服务。

§ **varyByParam**: 如前面所说，这里是用输入的 **GET** 中的参数来区分缓存内容。

§ **varyByExpression**: 给该选项设置 **PHP** 表达式，我们可以使用相应表达式的结果来区分缓存的内容。

所以，当在 **ProjectController** 类中配置了上面的过滤器后，每次针对某个项目的请求内容都会被缓存，并且在 **2** 分钟后刷新缓存。你可以通过先缓存一个项目的细节，然后使用某种方法刷新缓存来查看。所有的更新都会在其生存周期结束后才可以观察到。

缓存全部页面内容是极大提升性能的一种方式，但是对每一个应用程序中的每一个页面进行缓存没有意义。做一个上面 **3** 种方式的总结：数据，片段和页面缓存，在大多数真实的项目中是需要的。我们只简单的了解了一下 **Yii** 中提供的缓存功能。希望这能为你迅速看清 **Yii** 的缓存功能提供助力。

性能调整技巧

在结束本次迭代之前，我们将简要的列出其他可以用来提升基于 **Yii** 开发 **web** 应用程序的领域。

这些内容或多或少引用自 **Performance Tuning** 部分，相关在线手册链接地址：

<http://www.yiiframework.com/doc/guide/topics.performance> 但是在这里重提一下是很有意义的。

启用 APC

启用 **PHP APC** 扩展可能是最简单的增进整个应用程序性能的方式。扩展的缓存和优化控制的 **PHP** 中间代码并且减少了每次请求时花费在 **PHP** 脚本分析上的时间。

禁用调试模式

在本章的早些时候我们提到过这一概念，在这里再提一次也不为过。禁用调试模式是另外一个提高安全性和性能的方式。如果位于主 **index.php** 脚本中的 **YII_DEBUG** 常量被定义为 **true**，**Yii** 应用程序将运行在调试模式下。在调试模式下，许多包括在框架内的组件，将导致额外的损耗。

使用 yiilite.php

当使用 **PHP APC** 扩展，一件可以做的事是使用 **Yii bootstrap**(引导)文件 **yiilite.php** 代替 **yii.php**。这可以很好的增加 **Yii** 应用程序的性能。**yiilite.php** 文件包含在每一个 **yii** 发行版本中。它是一些常用 **Yii** 类文件合并的产物。因此使用 **yiilite.php** 可以减少文件的 **included** 量和 **trace** 语句。

使用 **yiilite.php**，而不启用 **APC** 则会导致性能的下降，这是因为 **yiilite.php** 包含了一些并非每次请求都需要花费时间解析的类。另外一个显而易见的是因为服务器的配置也会引起 **yiilite.php** 使用后变慢，尽管该服务器已经开始 **APC**。最好的鉴别是否适合使用 **yiilite.php** 的方法是针对 **hello world demo** 运行 **benchmark**。

使用缓存技术

正如我们在本章提到的，**Yii** 提供了一些可能会显著提升应用程序性能的缓存解决方案。下面列出了可以使用的缓存系统：

- § 当生成某些数据耗费了大量时间，我们使用数据缓存减少生成频率
- § 如果页面中的一部分相对静态，我们使用片段缓存来减少渲染频率
- § 如果整个页面相对静态，我们使用页面缓存来节约整个页面生存时间

启用 schema 缓存

如果应用程序使用了 **Active Record**，可以在生产环境中启用 **schema** 缓存来减少分析数据库 **schema** 的时间。可以通过设置 **CDbConnection::schemaCachingDuration** 属性的值大于 0 来实现。

值得一提的是这些都是基于应用程序级的缓存技术，我们也可以使用服务器端缓存技术来提升应用程序的性能。上面提到的启用 **APC** 就是这个范畴的。还有一些其他的服务器端技术，例如：**Zend Optimizer**, **eAccelerator**, **Squid**,等。

绝大部分这些只是为了提供一个练习指导，来帮助你做好发布 **Yii** 应用程序的准备。大部分应用程序性能的调整需要更多的练习，并且很多都是在 **Yii** 框架之外的整体性能。起初 **Yii** 就将性能纳入首要任务，并且不断从其他基于 **PHP** 的应用程序中学习(通过 <http://www.yiiframework.com/performance/>来了解更多内容)。当然，任何一个 **web** 应用程序都应当尽量提升性能，但是从一开始就选择 **Yii** 会使你的应用程序在性能上站在一个很高的高度。

小结

在本次最后的迭代中，我们关注了如何对处于生产环境的应用程序中进行维护和性能提升。首先我们学习了 **Yii** 的日志策略，同时也关注了如何按照不同的等级和类别记录和路由信息。然后我们关注了错误控制和 **Yii** 如何利用 **PHP5** 抛出的异常来提供弹性控制。然后我们学习了 **Yii** 中提供的不同缓存策略。我们学习了如何缓存应用程序数据和不同粒度等级的内容。基于特定变量和独立数据库的的数据缓存，页面中一

部分内容的片段缓存，和整个渲染结果页面内容的缓存。最后，我们列出一系列好的尝试，可以用来提升基于 **Yii** 的 **web** 应用程序性能。

不幸的，我们的 **TrackStar** 应用程序并未完成，全功能任务管理系统，以及很多留给读者来实现的概念。但是，一个好的基础已经打好，并且现在已经将 **Yii** 交入你的手中，你可以快速将这些变成功能丰富的应用程序。同时大量提到过的例子的身影会出现在其他你未来的应用程序中。祝你未来的项目开发愉快！