第 8 章 - 深入模型层

到目前为止本书大多数的讨论都专注于建立页面，处理请求与回应。 但是网页应用程序的业务逻辑大多依赖于它的数据模型。 ~~Symfony~~symfony 的默认模型组件是基于一个对象/关系映射层，也就是我们所知的 Propel 项目 (http://propel.phpdb.org/)。 在 symfony 应用程序中，你不用关注数据库的实际位置，而是通过对象来访问储存在数据库中的数据的。 这保持了高度的抽象和可移植性。

本章解释了如何建立一个数据对象模型和如何通过 Propel 来访问和修改数据。同时也展示了 symfony 是如何整合 Propel 的。

为什么使用 ORM 和抽象层？

数据库是关系型的。 PHP 5 和 symfony 都是面向对象的。 为了在面向对象环境中最有效的访问数据库，需要一个接口用来把对象逻辑转换为关系逻辑。如第一章所述，这个接口就叫做对象-关系映射(ORM)，它是由可以访问数据和保持业务规则的对象组成的。

ORM 最大的优势就是可重用性，它允许数据对象的方法可以被应用程序的其他部分调用，甚至可以从不同的应用程序中调用。ORM 层也可以封装数据逻辑，例如，基于用户的贡献度和如何作出的贡献来计算论坛用户的评分。 当一个页面需要显示例如一个用户的评分，不需要担心如何去计算，只要很简单的调用数据模型的方法即可。如果以后计算方法有所变化，你只需要修改模型的评分方法即可，应用程序的其他部分不需要改变。

使用对象来代替记录，用类来代替表，还有其他好处：他们允许你在对象中增加一个新的读取方法而不需要对应到表的一个列。例如，如果你有一个 client 表，它拥有两个字段分别叫做 first_name 和 last_name，你可能只想要获得一个 Name。在面向对象的世界里，Client 类中增加一个存取方法是非常简单的，如例 8-1 所示。从应用程序的角度来看，Client 类的 FirstName，LastName 和 Name 属性没有什么区别。 只有类本身才能决定属性所对应的数据库的列。

例 8-1 - 在模型类中的存取方法掩盖了实际表结构

```
public function getName()
{
return $this->getFirstName().' '.$this->getLastName();
}
```

所有重复的数据访问函数和数据自身的业务逻辑可以存在对象中。假设你有一

个 ShoppingCart 类，里面有一个 Item（是个对象）。 只要写一个自定义的方法来封装实际的计算过程，就可以在结账时得到购物车的总价。 如例 8-2 所示。

例 8-2 - 存取方法掩盖了数据逻辑

```
public function getTotal()
{
  $total = 0;
foreach ($this->getItems() as $item)
  {
    $total += $item->getPrice() * $item->getQuantity();
  }

return $total;
}
```

在建立数据访问过程的时候还要考虑另外一个要点：数据库厂商所使用的不同的 SQL 语法变种。换用另外一个数据库管理系统（DBMS）会让你不得不重写一部分为以前设计的 SQL 查询。如果用数据库独立语法来建立一个查询，并把实际 SQL 翻译为第三方语言，换数据库系统就不会麻烦了。这就是数据抽象层存在的目的。它强制让你使用特定的语法规则来写查询，同时把它转到到相应的 DBMS 并优化 SQL 语句。

抽象层的主要优势是可移植性，因为他让换用另一种数据库成为可能，甚至可以在项目中期换用。假设你需要为应用程序写一个快速原型，但客户还没有确定哪个数据库最适合他。你能先用 SQLite 建立你的应用程序，然后在客户有了决定后切换到 MySQL，PostgreSQL 或者 Oracle。这只要改变配置文件中一行代码即可。

~~Symfony~~symfony 使用 Propel 来实现 ORM，Propel 使用 Creole 做数据库抽象。这两个第三方组件，都是由 Propel 小组开发的，并且都无缝集合到了 symfony 中，你可以把他们作为框架的一部分。在本章描述的 Propel 和 Creole 的约定和语法规则都被改写过，因此 symfony 的语法与原始语法会有一些不同。

NOTE 在 symfony 项目中，所有的应用程序共享同一个模型。这就是项目层的全局观：依靠通用商业规则重组应用程序。这就是让模型独立于应用程序之外并且模型文件存在项目根目录的 lib/model/目录下的原因。

~~Symfony~~symfony 的数据库设计(schema)

为了创建 symfony 使用的数据对象模型，需要把数据库关系模型翻译为对象数据模型。ORM 需要关系模型的描述来做映射，这就叫做设计。在设计中，你定义表，表之间的关系，和表中列的特性。

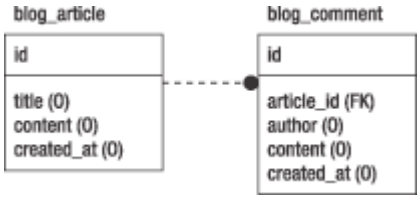
~~Symfony~~symfony 中设计的语法使用了 YAML 格式。schema.yml 文件必须放在 myproject/config/目录下。

NOTE ~~Symfony~~symfony 也接受 Propel 原生的 XML 设计格式，在本章稍后的超越 schema.yml: schema.xml 小节会做描述。

设计示例

如何把数据库结构转换为设计呢？看例子是最好的理解方法。试想一下你有一个 blog 数据库，包含两个表 blog_article 和 blog_comment，结构如图 8-1 所示。

图 8-1 - 一个 blog 数据库表结构



对应的 schema.yml 文件应该看上去如例 8-3 所示。

例 8-3 - schema.yml 示例

```
propel:
  blog_article:
    _attributes: { phpName: Article }
id:
title:        varchar(255)
content:      longvarchar
    created_at:
  blog_comment:
    _attributes: { phpName: Comment }
id:Mutator
    article_id:
author:       varchar(255)
content:      longvarchar
    created_at:
```

注意数据库名字本身（blog）并没有出现在 schemal.yml 文件中。反而在连接名下有数据库描述（本例中是 propel）。这是因为实际的连接设置可以依照应用程序运行的环境来定。例如，当你在开发环境中运行应用程序时，你会访问一个开发数据库（也许是 blog_dev），但是生产数据库也用的是同一个设计文件。在 database.yml 文件中有连接， 本章稍后的"数据库连接"会有介绍。设计不包含任何连接设置的细节，只有连接名用来保持数据库抽象。

基本设计语法

在 schema.yml 文件中，第一个关键字表示的是连接名。它可以包含多个表，每个表都有一些列。根据 YAML 语法，关键字用冒号作结束标记，结构用缩进来表示（一个或多个空格，但不是制表符）。

表可以有特殊的属性，包括 phpName（会生成同名的类）。如果没有设置表的 phpName，symfony 会以表名的驼峰命名法来创建类。

TIP 驼峰命名法把单词之间的下划线去掉了，并把每个单词的首字母大写。blog_article 和 blog_comment 的默认驼峰命名法版本是 BlogArticle 和 BlogComment。这种转换方法约定字的首字母大写， 就像骆驼的驼峰一样。

一个表包含了许多列。列的值可以用三种方式来定义：

- 如果你没有给出定义，symfony 会根据列名和一些约定来猜测最适合的属性，这些在下面的"空列"段落会有描述。例如，在例 8-3 中的 id 列不需要定义。~~Symfony~~symfony 会定义它为自增长的数值类型，表的主键。blog_comment 表的 article_id 会理解为 blog_article 表的外键（结尾是_id 的列被理解为外键，根据列前面部分的名字来确认是和哪张表有关联）。create_at 列会自动设置为 timestamp 类型。 对于这种类型的列，你不需要特别指定他们的类型。这就是为什么说 schema.yml 是非常容易写的原因。
- 如果你只定义了一个属性，那这就是列的类型。~~Symfony~~symfony 支持一些常用的列类型：boolean, integer, float, date, varchar(size), longvarchar (转换过的, 例如, 在 MySQL 下就会转换为 text)和其他。对于 text 内容超过 256 个字符的， 需要使用 longvarchar 类型，这是没有大小限制的（MySQL 中不能超过 65KB）。 注意 date 和 timestamp 类型有通常的 Unix 日期限制并且不能设置早于 1970-01-01。 如果需要设置一个更早的日期（例如，生日）， “unix 日期之前”的日期类型可以使用 bu_date 和 bu_timestamp。
- 如果需要定义其他的列属性（如默认值, 必填属性或者其他），你需要把列属性写为一组 key:value。 这种扩充设计语法会在本章稍后介绍。

列可以有一个 phpName 属性，它是首字母大写的（Id, Title, Content,等）并且在大多数情况下不能覆盖。

表也可以包含详细的外键和索引，以及少量的数据库结构定义。参考本章节后面的“扩展设计语法”部分。

模型类

设计是用来建立 ORM 层的模型类的。为了省时，这些类是通过命令行调用 propel-build-model 来生成的。

```
>symfony propel-build-model
```

输入这个命令后会先分析模型接着在项目的 lib/model/om 目录下生成基础数据模型类：

- BaseArticle.php
- BaseArticlePeer.php
- BaseComment.php
- BaseCommentPeer.php

还有，实际的数据模型类会建立在 lib/model 下：

- Article.php
- ArticlePeer.php
- Comment.php
- CommentPeer.php

你只定义了两个表，但会生成八个文件。这并无不妥， 但应该解释一下。

基础类和自定义类

为什么我们在两个不同的目录保留了两个版本的数据对象模型？

你也许会需要在模型对象中增加自定义方法和属性（试想例 8-1 中的 getName() 方法）。但是由于项目开发需要，你将会需要增加表或者列。当你修改了 schema.yml 文件时，需要重新调用 propel-build-model 来生成对象模型类。如果你的自定义方法写在自动生成的类中，他们会在每一次重新生成的时候被覆盖。

由设计直接生成的 Base 类放在 lib/model/om/目录中。你永远不需要去修改他们，因为每一次新建模型都会完全删除这些文件。

另一方面，自定义对象类会放在 lib/model/目录下，实际上是继承自 Base 类。当对已有的模型调用 propel-build-model 任务时，这些类不会被修改。因此这就是你可以增加自定义方法的地方。

例 8-4 展示了第一次调用 propel-build-model 任务建立的自定义模型类的一个示例。

例 8-4 - lib/model/Article.php 中的模型类文件示例

```
<?php

class Article extends BaseArticle
```

```
{
}
```

它继承了 BaseArticle 类所有的方法，但是修改设计不会影响到这个文件。

用自定义类来扩展基础类的机制可以让你在不知道最终数据库中模型之间的关系的时候开始编程。相关的文件结构会让模型既可以自定义又可以进化。

对象和 Peer 类

Article 和 Comment 是用来显示数据库中记录的对象类。他们赋予了记录的列和相关记录的访问权限。这就是说你可以调用 Article 对象的方法来获取文章的标题， 如例 8-5 所示。

例 8-5 - 在对象类中获得记录列

```
$article = new Article();
...
$title = $article->getTitle();
```

ArticlePeer 和 CommentPeer 都是 peer 类；因此，类包含了静态方法来操作表。他们提供了从表中获得记录的方法。他们的方法通常返回了一个对象或是相关对象类的对象的集合，如例 8-6 所示。

例 8-6 - Peer 类可以用静态方法来获得记录

```
$articles = ArticlePeer::retrieveByPks(array(123, 124, 125));
//$articles 是一个 Article 类的对象数组
```

NOTE 从数据模型的角度来看，不可能有 peer 对象。这就是为什么调用 peer 类的方法会使用::（调用静态方法）而不是通常的->（调用实例方法）。

所以把对象类和 peer 类的基础类和自定义类加起来，数据库设计里的一个表会自动生成四个类。 事实上， 有第五种类生成在 lib/model/map/目录下， 它们包含了关于表运行时所需要的 metadata 信息。 但是也许你永远不需要修改这个类，你完全可以忘了它。

访问数据

在 symfony 中，是通过对象来访问数据的。如果你习惯使用关系模型和使用 SQL 来获取、修改你的数据的话，对象模型方法会让你觉得有些复杂。但是当你尝试过用面向对象方法来访问数据的话，就会喜欢上它的。

但是首先，让我们确信我们说的是同一个词汇。 关系型和对象数据模型有一些相似点，但是他们都有自己的术语：

关系的	面向对象的
表	类
行，记录	对象
字段，列	属性

获得列值

当 symfony 建立模型时，它为每一个在 schema.yml 中存在的表都建立一个基础对象类。每一个类都有一个基于列定义的默认的构造器， 读取方法和设置方法：new，getXXX() 和 setXXX()方法帮助创建对象并给予访问对象属性的权限，如例 8-7 所示。

例 8-7 - 生成对象类方法

```
$article = new Article();
$article->setTitle('My first article');
$article->setContent('This is my very first article.\n Hope you enjoy
it!');

$title   = $article->getTitle();
$content = $article->getContent();
```

NOTE 生成的对象类名为 Article，这是由 blog_article 表的 phpName 定义的。如果在设计中没有定义 phpName，这个类就会取名为 BlogArticle。读取方法和设置方法使用驼峰命名法的变异来定义列名，所以 getTitle()方法会获得 title 列的值。

可以使用 fromArray()方法一次定义多个字段，在生成的每个类对象中都有此方法，如例 8-8 所示。

例 8-8 - fromArray()方法是一个多重设置方法

```
$article->fromArray(array(
  'title'   => 'My first article',
  'content' => 'This is my very first article.\n Hope you enjoy it!',
));
```

获得相关联的数据

在 blog_comment 表中 article_id 列实际上定义了 blog_article 表的一个外键。

每一个 comment 都与一篇文章相对应，同时一篇文章可以有多个 comment。生成的类包含五个方法来把这些对应关系转换成面向对象的方法，如下：

- $comment->getArticle(): 获得相关联的 Article 对象
- $comment->getArticleId(): 获得相关联的 Article 对象的 ID
- $comment->setArticle($article): 定义相关联的 Article 对象
- $comment->setArticleId($id): 通过 ID 定义相关联的 Article 对象
- $article->getComments(): 获得相关联的 Comment 对象

getArticleId()和 setArticleId()方法说明了你可以把 article_id 列作为一个普通列并手动设置对应关系，但这么做并不好。面向对象方法的优点让其他三个方法更容易理解。例 8-9 显示了如何使用生成的设置方法。

例 8-9 - 外键转换为特别的设置方法

```
$comment = new Comment();
$comment->setAuthor('Steve');
$comment->setContent('Gee, dude, you rock: best article ever!);

// 把此 comment 和$article 对象关联
$comment->setArticle($article);

// 另一种语法
// 仅当对象已经存在于数据库中才有意义
$comment->setArticleId($article->getId());
```

例 8-10 展示了生成的获取方法是如何使用的。同时也演示了如何在模型对象中调用关联方法。

例 8-10 - 外键转为特别的 getters

```
// 多对一关系
echo $comment->getArticle()->getTitle();
 => My first article
echo $comment->getArticle()->getContent();
 => This is my very first article.
    Hope you enjoy it!

// 一对多关系
$comments = $article->getComments();
```

getArticle()方法返回了一个 Article 类的对象，从而可以使用 getTitle() 获取方法。这比直接使用 join 要好得多，而仅仅只会多几行代码（从调用

$comment->GetArticleId()开始）。

例 8-10 中的$comments 变量包含了 Comment 类的一个对象数组。你能用 $comments[0]来显示第一个对象或是用 foreach($comments as $comment)来遍历这个对象数组。

NOTE 你现在知道为什么模型对象是以单数命名的了。 在 Comment 对象名字后面增加 s，会在 blog_comment 表中制造一个外键并产生建立 getComments()方法的动作。如果你给模型对象一个复数名字，生成时候会产生一个叫做 getCommentss()的无意义的方法。

保存和删除数据

创建一个新的对象可以通过调用 new 构造器，但修改此对象不会对数据库有任何影响，也就是说这并不对应于 blog_article 表中存在的实际记录。但你可以调用对象的 save()方法把数据保存到数据库中。

```
$article->save();
```

ORM 可以查明对象之间的关系，因此保存$article 对象同时也就保存了相关的 $comment 对象。也就是说它知道保存了的对象在数据库中有关联的数据，当调用 save()的时候，有时候会转换为 INSERT 语句，有时候会使 UPDATE 语句。save()方法会自动设置主键，所以在保存后，你能用$article->getId()得到一个新的主键。

TIP 你能通过调用 isNew()来检查对象是否是新建的。如果你想知道对象是否被修改过是否该保存，可以调用它的 isModified()方法。

如果你读了文章的 comment，也许会后悔把他们发布到互联网上。如果你觉得一些回复者的回复不合适的话，可以很方便的使用 delete()方法来删除评论，如例 8-11 所示。

例 8-11 - 用 delete()方法从数据库删除记录的相关对象

```
foreach ($article->getComments() as $comment)
{
  $comment->delete();
}
```

TIP 在调用 delete()方法后，请求结束之前对象依旧可以访问。要确认是否在数据库中已经把对象删除的话就需要调用 isDeleted()方法了。

通过主键来获得记录

如果你知道特定记录的主键值，可以使用 peer 类的 retrieveByPk()方法来获得相关对象。

```
$article = ArticlePeer::retrieveByPk(7);
```

schema.yml 文件中定义了 id 字段作为 blog_article 表的主键，因此这个语句会返回 id 为 7 的文章。由于你使用了主键，所以只会返回一条记录；$article 变量包含了类 Article 的对象。

有时候，也许包含了多个主键（复合主键）。在这些情况中，retrieveByPK()方法会接受多个参数，每一个对应一个主键。

你也能用 retrieveByPKs()方法，输入一组主键组成的数组作为参数来获得多个对象。

通过 Criteria 获得数据

当你想获得多个记录时，你需要调用 peer 类的 doSelect()方法来获得你想要的对象。例如，调用 ArticlePeer::doSelect()来获得 Article 类的对象。

doSelect()方法的第一个参数是 Criteria 类的一个对象，Criteria 类是一个简单查询定义类，它为了用数据库抽象而没有使用 SQL。

一个空的 Criteria 返回了类的所有对象。 例如，例 8-12 的代码就返回了所有的 article。

例 8-12 - 通过 Criteria 的 doSelect()来获得数据--空的 Criteria

```
$c = new Criteria();
$articles = ArticlePeer::doSelect($c);

// 和下面 SQL 查询结果是一样的
SELECT blog_article.ID, blog_article.TITLE, blog_article.CONTENT,
       blog_article.CREATED_AT
FROM   blog_article;
```

SIDEBAR 化合（hydrating）

调用::doSelect()比使用简单的 SQL 查询强大的多。首先，SQL 会针对使用的 DBMS 而优化。 其次，任何传递给 Criteria 的值都会在整合入 SQL 代码之前被转义， 这能防止 SQL 注入的风险。第三点，此方法返回了一个对象数组而不是一个结果集。ORM 基于数据库结果集自动创建并丢出对象。这个过程叫做化合（hydrating）。

如果遇到一个更复杂的对象选择时，你需要用到 WHERE, ORDER BY, GROUP BY 和其他 SQL 语句。 Criteria 对象有针对所有这些情况的方法和参数。 例如，在例 8-13 中我们建立了一个 Criteria 来取得 Steve 写的所有的 comments， 按照日期排序。

例 8-13 - 通过 Criteria 的 doSelect()来获得记录--有条件的 Criteria

```
$c = new Criteria();
$c->add(CommentPeer::AUTHOR, 'Steve');
$c->addAscendingOrderByColumn(CommentPeer::CREATED_AT);
$comments = CommentPeer::doSelect($c);

// 等同于下面 SQL 语句执行的结果
SELECT blog_comment.ARTICLE_ID, blog_comment.AUTHOR,
blog_comment.CONTENT,
       blog_comment.CREATED_AT
FROM   blog_comment
WHERE  blog_comment.author = 'Steve'
ORDER BY blog_comment.CREATED_AT ASC;
```

把类常量作为参数传递给 add()方法，参考属性名字。 他们的名字都是列名的大写字符版本。例如，要在 blog_article 表中增加 content 列， 用 ArticlePeer:CONTENT 类常量。

NOTE 为什么使用 CommentPeer:AUTHOR 来代替 blog_comment.AUTHOR， 这是因为他会输出 SQL 查询语句？ 假设你需要在数据库中更改 author 字段为 contributor。如果你是用 blog_comment.AUTHOR， 你就需要在每个调用模型的地方都进行修改。 而另外一个， 使用 CommentPeer::AUTHOR, 只要简单的在 schema.yml 文件中修改字段名，设置 phpName 为 AUTHOR 然后重新编译模型即可。

表 8-1 对比了 SQL 语法和 Criteria 对象语法。

表 8-1 - SQL 和 Criteria 对象语法

SQL	Criteria
WHERE column = value	->add(column, value);
WHERE column <> value	->add(column, value, Criteria::NOT_EQUAL);
其他比较操作符	
> , <	Criteria::GREATER_THAN, Criteria::LESS_THAN
>=, <=	Criteria::GREATER_EQUAL,

SQL	Criteria
	Criteria::LESS_EQUAL
IS NULL, IS NOT NULL	Criteria::ISNULL, Criteria::ISNOTNULL
LIKE, ILIKE	Criteria::LIKE, Criteria::ILIKE
IN, NOT IN	Criteria::IN, Criteria::NOT_IN
其他 SQL 关键字	
ORDER BY column ASC	->addAscendingOrderByColumn(column);
ORDER BY column DESC	->addDescendingOrderByColumn(column);
LIMIT limit	->setLimit(limit)
OFFSET offset	->setOffset(offset)
FROM table1, table2 WHERE table1.col1 = table2.col2	->addJoin(col1, col2)
FROM table1 LEFT JOIN table2 ON table1.col1 = table2.col2	->addJoin(col1, col2, Criteria::LEFT_JOIN)
FROM table1 RIGHT JOIN table2 ON table1.col1 = table2.col2	->addJoin(col1, col2, Criteria::RIGHT_JOIN)

TIP 最好的理解生成类的方法的途径就是查看位于 lib/model/om/目录下的 Base 文件。 他们的方法名是很直接的， 但如果你需要了解更多的信息，可以在 config/propel.ini 文件中设置 propel.builder.addComments 参数为 true，然后重建模型。

例 8-14 是多个条件的 Criteria 的另一个实例。 它搜索所有 Steve 的包含"enjoy"这个词的评论，并按照日期排列。

例 8-14 - 通过 Criteria 用 doSelect()获得记录的另一个示例--有条件的 Criteria

```
$c = new Criteria();
$c->add(CommentPeer::AUTHOR, 'Steve');
$c->addJoin(CommentPeer::ARTICLE_ID, ArticlePeer::ID);
$c->add(ArticlePeer::CONTENT, '%enjoy%', Criteria::LIKE);
$c->addAscendingOrderByColumn(CommentPeer::CREATED_AT);
$comments = CommentPeer::doSelect($c);

// 等同于下面 SQL 语句执行的结果
SELECT blog_comment.ID, blog_comment.ARTICLE_ID, blog_comment.AUTHOR,
       blog_comment.CONTENT, blog_comment.CREATED_AT
FROM   blog_comment, blog_article
WHERE  blog_comment.AUTHOR = 'Steve'
       AND blog_article.CONTENT LIKE '%enjoy%'
       AND blog_comment.ARTICLE_ID = blog_article.ID
```

```
ORDER BY blog_comment.CREATED_AT ASC
```

就如同 SQL 是一个很简单的语言却可让你建立非常复杂的查询一样， Criteria 对象也可以处理任意复杂的问题。 但是很多开发者会在转换为面向对象逻辑之前先考虑 SQL， Criteria 对象开始也许会有些难以掌握。 最好的理解方法是从例子和简单的应用程序中学习。 例如 symfony 项目网站，到处都有 Criteria 建立的演示会从各种方面来启发你。

除了 doSelect()方法外， 每一个 peer 类都有一个 doCount()方法用来简单的统计记录的数量，用作给 criteria 传递的参数并返回数字作为结果。 因为没有返回对象， 所以例子中没有化合（hydrating）过程， 因此 doCount()方法比 doSelect()方法快。

Peer 类也提供了 doDelete(), doInsert() 和 doUpdate()方法， 用来作为 Criteria 的参数。 这些方法允许你对数据库执行 DELETE, INSERT 和 UPDATE 查询语句。 可以在你的模型中查看生成的 peer 类来获得更多的关于 propel 方法的细节。

最后，如果你只想返回第一个对象， 调用 doSelectOne()替换掉 doSelect()。 这会让 Criteria 只返回一个结果，好处就是这个方法返回的是一个对象而不是一个对象数组。

TIP 当一个 doSelect()查询返回大量结果的时候， 你也许只会看到他们中的部分结果。 Syfmony 提供了一个叫做 sfPropelPager 的翻页类，可以自动的处理结果的翻页。 看 API 手册 http://www.symfony-project.com/api/symfony.html 来获得更多的信息和使用示例。

直接使用 SQL 查询语句

有时候，你不想得到对象而只是想得到由数据库执行得到的结果。 例如，要获得所有文章最新的建立时间， 取得到所有的文章然后遍历数组是没有意义的。你会倾向于让数据库来处理并只返回结果，因为这会跳过对象化合（hydrating）过程。

另一方面，你不想为了管理数据库直接调用 PHP 命令，因为会失去使用数据库抽象层的优势。这意味着需要绕过 ORM(Propel)而不是数据库抽象层(Creole)。

你需要做如下步骤通过 Creole 查询数据库：

1. 获得数据库连接。
2. 建立查询字符串。
3. 建立一个声明。
4. 从声明执行结果中循环结果集。

如果这对你没帮助， 例 8-15 中的代码也许会让你更清晰一些。

例 8-15 - 用 Creole 来自定义查询

```
$connection = Propel::getConnection();
$query = 'SELECT MAX(%s) AS max FROM %s';
$query = sprintf($query, ArticlePeer::CREATED_AT,
ArticlePeer::TABLE_NAME);
$statement = $connnection->prepareStatement($query);
$resultset = $statement->executeQuery();
$resultset->next();
$max = $resultset->getInt('max');
```

就像 Propel 的选择功能，初次使用 Creole 查询的时候会觉得这需要一些技巧。 再说一次，从已有应用程序的示例和指南会展示给你正确的使用方法。

CAUTION 如果你倾向绕过这个过程并直接访问数据库， 你会面临失去安全性和 Creole 提供的抽象层。Creole 做了对数据库所有可用的转义和安全性的处理。尽管用 Creole 方法会花更长的时间，但它会强迫你使用好的方法，这就保证了应用程序的性能，可移植性和安全性。这对于包含从不信任的来源获得参数（比如 Internet 使用者）的查询来说特别有用。 而直接访问数据库会给你带来 SQL 注入攻击的危险性。

使用特殊日期列

通常，当一个表有一个列叫做 created_at 时， 它通常储存的是记录创建时的日期的时间戳格式。 updated_at 列也一样，因此当每次更新记录时，它本身也会被更新为当前时间。

有个好消息是 symfony 会知道这些列的名字并会自动为你处理。 你不需要手动设置 created_at 和 updated_at 列；它会自动为你更新，如例 8-16 所示。 对于 created_on 和 updated_on 也一样。

例 8-16 - created_at 和 updated_at 列的数据会被自动处理

```
$comment = new Comment();
$comment->setAuthor('Steve');
$comment->save();

// 显示创建日期
echo $comment->getCreatedAt();
  => [date of the database INSERT operation]
```

此外， 日期列的 getter 允许日期格式作为参数：

```
echo $comment->getCreatedAt('Y-m-d');
```

SIDEBAR 数据层重构

当开发一个 symfony 项目时，通常开始在动作中写逻辑代码。 但在控制层中并没有数据库查询和模型处理。 因此所有的数据逻辑都应该转移到模型层。 当你需要在动作中的多个地方执行同一个请求时，想一下如何把相关代码转移到模型中。这会对你保持动作简洁和可读性有很好的帮助。

例如，假设需要一段代码在 blog 中获得指定标签的十个最流行的文章（传递请求参数）。 这个代码在动作中，但不是在模型中。 事实上，如果你需要在模板中列出来， 这个动作应该看上去像这样：

```
public function executeShowPopularArticlesForTag()
{
  $tag = TagPeer::retrieveByName($this->getRequestParameter('tag'));
  $this->foward404Unless($tag);
  $this->articles = $tag->getPopularArticles(10);
}
```

这个动作用 request 参数创建了一个 Tag 类的对象。 因此所有查询数据库所需的代码都位于这个类的 getPopularArticles()方法中。 这使行为更易读，并且让模型代码更易在其他行为中重用。

重构的其中一个方法就是把代码移到一个更合适的位置。如果你经常这么做，你的代码会易于维护并被其他开发者理解。有一个不成文的规则是：动作的代码应该尽量低于 10 行 PHP 代码。

数据库连接

数据模型是和使用的数据库分离的，但是你还是要使用数据库。 要发送请求给项目数据库的话至少要让 symfony 知道数据库名字，访问的代码和数据库的类型。 这些连接设置应该放在位于 config/目录下的 databases.yml 文件中。例 8-17 中的示例。

例 8-17 - 在 myproject/config/databases.yml 的数据库连接配置示例

```
prod:
propel:
param:
host:                mydataserver
```

```
username:           myusername
password:           xxxxxxxxxx

all:
propel:
class:                sfPropelDatabase
param:
      phptype:            mysql     # 数据库类型
hostspec:           localhost
database:           blog
username:           login
password:           passwd
port:               80
      encoding:           utf-8     # 创建表默认的 charset
      persistent:         true      # 是否使用持久连接
```

连接设置是基于环境的。 你可以给 prod, dev 和 test 环境或是任何应用程序中的环境截然不同的设置。 这个配置也可以由每个应用程序通过设置应用程序相关的文件中不同的值而覆盖，例如在 apps/myapp/config/databases.yml 中。 举个例子，你可以使用此方法来给前台和后台应用程序定义不同的安全策略，并定义几个数据库用户拥有不同的数据库权限来实施控制。

对于每一个环境，你能定义很多连接，每一个连接对应了一个同名的设计。 在例 8-17 中，propel 连接对应了例 8-3 中的 propel 设计。

Creole 支持的数据库系统，也就是 phptype 参数允许的值是：

- mysql
- sqlserver
- pgsql
- sqlite
- oracle

hostspec, database, username 和 password 是通常数据库连接设置需要的。他们能写成更短的数据库源名(DSN)。例 8-18 等同于例 8-17 中的 all:。

例 8-18 - 简单的数据库连接方法设置

```
all:
propel:
class:          sfPropelDatabase
param:
dsn:          mysql://login:passwd@localhost/blog
```

如果使用 SQLite 数据库， 必须在数据库文件中设置 hostspec 参数。例如，如

果你把 blog 数据库放在 data/blog.db 中， databases.yml 文件将看上去就像例 8-19 一样。

例 8-19 - SQLite 使用了文件路径作为 HOST 的数据库连接设置

```
all:
propel:
class:          sfPropelDatabase
param:
phptype:  sqlite
database: %SF_DATA_DIR%/blog.db
```

扩展模型

symfony 生成的模型方法是很棒的但通常并不够用。当你实现你自己的业务逻辑时，你需要去扩展它，不论是增加新的方法或是覆盖现有方法。

增加新的方法

你可以在 lib/model/目录下生成的空模型类中增加新的方法。使用$this 来调用当前对象的方法， 并使用 self::来调用当前类的静态方法。 记住，自定义类继承 lib/model/om/目录下 Base 类的方法。

例如，对于在例 8-3 中生成的 Article 对象， 你能增加一个神奇的 __toString()方法让 Article 类显示他自己的标题，如例 8-20 所示。

例 8-20 - 在 lib/model/Article.php 中自定义模型

```
<?php

class Article extends BaseArticle
{
public function __toString()
  {
    return $this->getTitle();  // getTitle()继承自 BaseArticle
  }
}
```

你也可以扩展 peer 类，例如，增加一个新方法来获得按照创建日期排序的所有文章，如例 8-21 所示。

例 8-21 - 在 lib/model/ArticlePeer.php 中自定义模型

```
<?php

class ArticlePeer extends BaseArticlePeer
{
public static function getAllOrderedByDate()
  {
    $c = new Criteria();
    $c->addAscendingOrderByColumn(self:CREATED_AT);
return self::doSelect($c);

  }
}
```

新方法和系统生成的方法使用方法一样， 如例 8-22 所示。

例 8-22 - 使用自定义的模型方法就如使用生成的方法一样

```
foreach (ArticlePeer::getAllOrderedByDate() as $article)
{
  echo $article;       // 会调用 __toString()这个魔术方法
}
```

覆盖现有方法

如果在 Base 类中一些生成的方法和你的需求并不一致， 可以在自定义类中覆盖它们。只要确认你使用相同的方法签名（就是说相同数量的参数）。

例如， $article->getComments()方法返回一个没有排序的 Comment 对象的数组。 如果你想让结果按照创建日期排序过并有最新的 comment 在前面， 只要覆盖 getComments()方法，如例 8-23 所示。 小心原始 getComments()方法（在 lib/model/om/BaseArticle.php）需要一个 criteria 值和一个连接值作为参数，所以你的函数也必须包含它们。

例 8-23 - 在 lib/model/Article.php 覆盖现有的模型方法

```
public function getComments($criteria = null, $con = null )
{
  // 在 PHP5 下，对象是引用传递的， 所以要避免修改原始的，
  // 你必须克隆它
  $criteria = clone $criteria;
  $criteria->addDescendingOrderByColumn(ArticlePeer::CREATED_AT);

return parent::getComments($criteria, $con);
```

}

自定义方法最终调用一个基础父类， 这很好。 不过，你也可以完全绕过父类并返回你希望的值。

使用模型行为

一些模型修改是通用的可以重复使用的。例如，用来让模型对象排序的方法、优化锁定来防止并发对象之间的冲突，都是通用的扩展，可以在许多类中使用。

~~Symfony~~symfony 把这些扩展打包为行为。 行为是外部的给模型类提供了额外的方法的类。 模型类已经包含了钩子， symfony 知道如何用 sfMixer（参考 17 章获得更多的细节）扩展他们。

要在你的模型类中激活行为，你必须修改 config/propel.ini 文件中的一个设置：

```
propel.builder.AddBehaviors = true      // 默认值是 false
```

~~Symfony~~symfony 默认是没有绑定行为的， 但是可以通过插件来安装它。 当行为插件安装好之后，你能用一行代码来设置行为给一个类。 例如， 如果在你的应用程序中安装了 sfPropelParanoidBehaviorPlugin， 你能通过这个行为增加下面这段在文章尾部来扩展文章类：

```
sfPropelBehavior::add('Article', array(
  'paranoid' => array('column' => 'deleted_at')
));
```

重建模型后， 删除 Article 对象时，对象依旧会保留在数据库中，仅仅对 ORM 的查询是不可见的，除非你用 sfPropelParanoidBehavior::disable()把行为禁用了。

在 wiki 查看 symfony 插件列表来搜索 behaviors(http://www.symfony-project.com/trac/wiki/SymfonyPlugins#Propelbehaviorplugins). 每个行为都有自己的文档和安装方法。

扩展设计语法

schema.yml 文件可以是简洁的， 如例 8-3 所示。 但是相关联的模型通常是复杂的。 这就是为什么设计有一个扩展语法来处理几乎每一种情况。

属性

连接和表可以有特殊的属性， 如例 8-24 所示。 它们的定义在_attributes 关

键字下。

例 8-24 - 连接和表的属性

```
propel:
  _attributes:    { noXsd: false, defaultIdMethod: none, package:
lib.model }
  blog_article:
    _attributes: { phpName: Article }
```

你也许想在代码生成前验证你的设计。 要这样做的话， 在连接中需要设置 noXSD 属性为 false。 连接也支持 defaultIdMethod 属性。 如果没有提供，会使用数据库的原生方法生成的 ID，例如，MySQL 的 autoincrement 或者 PostgreSQL 的 sequences。 另外一个可能的值是 none。

package 属性有点像命名空间； 他决定了生成的类的储存路径。 默认是 lib/model/，但是你能通过在 subpackage 组织你的模型来改变它。 例如，如果你不想在同一个目录下放置核心的业务类与数据库定义的类， 可以用 lib.model.business 和 lib.model.stats 包来定义两个设计。

你已经看到过表属性 phpName， 它用来设置生成的类名并映射到相应的表。

包含本地化内容的表（就是不同版本的内容在一张相关联的表中做国际化处理）也使用两个额外的属性（13 章有详细介绍）， 如例 8-25 所示。

例 8-25 - i18n 表属性

```
propel:
  blog_article:
    _attributes: { isI18N: true, i18nTable: db_group_i18n }
```

SIDEBAR 处理多个设计

你可以在每一个应用程序中拥有多个设计。 ~~Symfony~~symfony 会查看 config/目录下文件名结尾是 schema.yml 或者 schema.xml 的文件。 如果你的应用程序有多个表，或者如果一些表不想分享同一个连接，你会觉得这个方法非常有用。

试想一下这两个设计：

```
 // 在 config/business-schema.yml
propel:
   blog_article:
     _attributes: { phpName: Article }
id:
title: varchar(50)
```

```
 // 在 config/stats-schema.yml
propel:
   stats_hit:
     _attributes: { phpName: Hit }
id:
resource: varchar(100)
   created_at:
```

这两个 schemas 共享了同一个连接(propel)， Article 和 Hit 类会生成在同一个 lib/model/目录下。 每件事情处理的都如同写在同一个设计中一样。

你也能让不同的设计使用不同的连接（例如，定义在 databases.yml 中的 propel 和 propel_bis）并在子目录中组织生成的类：

```
 // 在 config/business-schema.yml
propel:
   blog_article:
     _attributes: { phpName: Article, package: lib.model.business }
id:
title: varchar(50)

 // 在 config/stats-schema.yml
 propel_bis:
   stats_hit:
     _attributes: { phpName: Hit, package.lib.model.stat }
id:
resource: varchar(100)
   created_at:
```

许多应用程序使用多个设计。 特别是，一些插件为了防止和你自己的类有冲突打包了他自己的设计（第 17 章有详细介绍）。

列详细资料

基础设计语法提供了两种选择： 让 symfony 根据列名推算出列的特征（给一个空值）或者用于类型关键字定义一个类型。 例 8-26 演示了这些选择。

例 8-26 - 基础列属性

```
propel:
  blog_article:
    id:                 # 让 symnfony 自己来处理
    title: varchar(50)  # 自定一个类型
```

但是你能对列定义更多。 如果这么做了，你需要用一个数组来定义列属性，如例 8-27 所示。

例 8-27 - 复杂的列属性

```
propel:
  blog_article:
id:       { type: integer, required: true,primaryKey: true,
autoIncrement: true }
name:     { type: varchar(50), default: foobar, index: true }
group_id: { type: integer, foreignTable: db_group,foreignReference:
id, onDelete: cascade }
```

列参数有以下几种：

- type：列类型分 boolean, tinyint, smallint, integer, bigint, double, float, real, decimal, char, varchar(size), longvarchar, date, time, timestamp, bu_date, bu_timestamp, blob, and clob.
- required：布尔值。 如果这个列是必须的话就设置为 true。
- default：默认值。
- primaryKey：布尔值。 如果是主键就设为 true。
- autoIncrement：布尔值。 如果是 integer 并需要自动增长的话就设为 true。
- sequence：数据库的序列名用来给需要 autoIncrement 列使用（例如，PostgreSQL 和 Oracle）。
- index：布尔值。 如果想使用简单的索引时设置为 true 或是如果想要为这个列创建 unique 索引。
- foreignTable：一个表名， 用来创建对其他表用的外键。
- foreignReference：相关列的名字，如果是通过 foreignTable 定义的外键
- onDelete：当相关表的字段被删除时候决定作什么操作。 当设为 setnull，则外键列会设置为 null。当设为 cascade，记录会被删除。如果数据库引擎不支持 set 行为， ORM 会模拟。 只在有 foreignTable 和 foreignReference 时有意义。
- isCulture: 布尔值。 如果在本地化的内容表中有 culture 列，则设置为 true。

外键

就如可选择的 foreignTable 和 foreignReference 列属性， 你可以在表的 _foreignKeys:关键字下增加外键。 例 8-28 的设计会在 user_id 列创建一个外键， 来匹配 blog_use 表的 id 字段。

例 8-28 - 外键的另一种语法

```
propel:
  blog_article:
id:
title:   varchar(50)
user_id: { type: integer }
    _foreignKeys:
      -
foreignTable: blog_user
onDelete:     cascade
references:
          - { local: user_id, foreign: id }
```

在有多重引用外键时另一种语法非常有用，他给外键一个名字，如例 8-29 所示。

例 8-29 - 多重引用外键时候的另一种语法

```
    _foreignKeys:
      my_foreign_key:
foreignTable:  db_user
onDelete:      cascade
references:
          - { local: user_id, foreign: id }
          - { local: post_id, foreign: id }
```

索引

作为 index 列属性的另一个选择，你能在表的_indexs:下增加索引。 如果你想要定义唯一索引，你必须用_uniques:替代。 例 8-30 展示了索引的另一个语法。

例 8-30 - 索引和唯一索引的另一种语法

```
propel:
  blog_article:
id:
title:            varchar(50)
    created_at:
    _indexes:
      my_index:       [title, user_id]
    _uniques:
      my_other_index: [created_at]
```

当为多个列建立索引的时候另一种语法是非常有用的。

空列

当遇到一个空值列的时候，symfony 会猜测并增加一个值。 例 8-31 有关于增加一个空列的详细信息。

例 8-31 - 从列名推演出列详细资料

```
// 叫做 id 的空列被当作主键
id:          { type: integer, required: true, primaryKey: true,
autoIncrement: true }

// 叫做 XXX_id 的空列被当作外键
foobar_id:  { type: integer, foreignTable: db_foobar,
foreignReference: id }

// 叫做 created_at, updated_at, created_on 和 updated_on
// 被看作日期并自动设置为 timestamp 类型
created_at: { type: timestamp }
updated_at: { type: timestamp }
```

对于外键， symfony 会查找和列名开头相同 phpName 的表名， 如果找到了，他就会认为这就是 foreignTable。

I18n 表

~~Symfony~~symfony 支持相关的表的内容国际化。 这就意味着当你要把标题国际化时， 他会储存在两个分开的表中： 一个放在常规的列中，另一个放在国际化的列中。

在 schema.yml 文件中，当你把表命名为 foobar_i18n 就意味着是 I18n 表了。例如，在例 8-32 示例中的数据库设计会根据列和表属性自动完成让国际化内容机制工作。 就内部机制来说， symfony 会把它当作例 8-33 所写的一样。 第 13 章会告诉你更多的关于 i18n 的信息。

例 8-32 - 隐含的 i18n 机制

```
propel:
  db_group:
id:
    created_at:

  db_group_i18n:
name:         varchar(50)
```

例 8-33 - 详述的 i18n 机制

```
propel:
```

```
  db_group:
    _attributes: { isI18N: true, i18nTable: db_group_i18n }
id:
    created_at:

  db_group_i18n:
id:       { type: integer, required: true, primaryKey:
true,foreignTable: db_group, foreignReference: id, onDelete:
cascade }
culture:  { isCulture: true, type: varchar(7), required:
true,primaryKey: true }
name:     varchar(50)
```

超越 schema.yml： schema.xml

事实上， schema.yml 格式是 symfony 内部格式。 当你调用一个 propel-命令行时， symfony 实际上会把这个文件转换为 generated-schema.xml 文件， 这是 Propel 实际处理的模型的文件类型。

schema.xml 文件包含了和它 YAML 版本相同的信息。 例如， 例 8-34 就是例 8-3 转换后的 XML 文件。

例 8-34 - 对应例 8-3 的 schema.xml 示例

```
<?xml version="1.0" encoding="UTF-8"?>
<database name="propel" defaultIdMethod="native" noXsd="true"
package="lib.model">
<table name="blog_article" phpName="Article">
<column name="id" type="integer" required="true"
primaryKey="true"autoIncrement="true" />
<column name="title" type="varchar" size="255" />
<column name="content" type="longvarchar" />
<column name="created_at" type="timestamp" />
</table>
<table name="blog_comment" phpName="Comment">
<column name="id" type="integer" required="true"
primaryKey="true"autoIncrement="true" />
<column name="article_id" type="integer" />
<foreign-key foreignTable="blog_article">
<reference local="article_id" foreign="id"/>
</foreign-key>
<column name="author" type="varchar" size="255" />
<column name="content" type="longvarchar" />
<column name="created_at" type="timestamp" />
</table>
```

</database>

schema.xml 格式描述可以在文档中找到，也能在 Propel 项目网站的"Getting Started"章节(http://propel.phpdb.org/docs/user_guide/chapters/appendices/AppendixB-SchemaReference.html)找到。

YAML 格式设计用来让设计的读和写保持简洁，但是代价是复杂的设计不能在 schema.yml 文件中描述出来。另一方面， XML 格式允许所有的 schema 描述，不论它多复杂，不论他是否包含数据库提供商特别的设置，表继承关系等等。

~~Symfony~~symfony 实际上用的是 XML 格式的 schema。 所以如果你的 schema 对于 YAML 语法来说太复杂，如果已有一个 XML 的设计文件，或者你已经对 Propel XML 语法很熟悉了，你可以不使用 symfony YAML 语法。 用你在项目的 config/ 目录下的 schema.xml，来建立模型，开始所有的事情。

SIDEBAR symfony 中的 Propel

本章所述所有的细节都不是针对 symfony 的，而是针对 Propel 的。Propel 是 symfony 所推荐的对象/关系抽象层，但是你可以选择其他的。 无论如何，symfony 和 Propel 结合的更紧密，因为如下原因：

所有的对象数据模型类和 Criteria 类都是自动载入的类。 当你使用它们的时候， symfony 会包含相应的文件并且你不需要手动的去增加相关的文件声明。在 symfony 中，Propel 不需要执行或者初始化。 当一个对象使用 Propel 的时候， 框架会自己初始化它。 一些 symfony 辅助函数使用 Propel 对象作为他的参数来完成高级任务（例如翻页或者过滤）。 使用 Propel 对象可以快速建模和生成应用程序后台（第 14 章有详细介绍）。 通过 schema.yml 文件可以让设计写起来更快。

并且，Propel 与使用的数据库无关，symfony 也是如此。

不要重复建立模型

使用 ORM 的代价是你必须定义数据结构两次： 一次为了数据库，一次为了对象模型。 幸运的是， symfony 提供了从其中一个生成另一个的命令行工具， 这样就可以避免重复工作。

基于已存在的设计建立一个 SQL 数据库结构

如果从写 schema.yml 文件开始写应用程序的话， symfony 能生成一个 SQL 查询并直接从 YAML 数据模型生成数据表。 要使用这些查询语句， 在项目根目录，键入：

```
>symfony propel-build-sql
```

myproject/data/sql/目录下会创建一个 lib.model.schema.sql 文件。 注意生成的 SQL 代码会针对在 propel.ini 文件中定义的 phptype 参数来对数据库系统进行优化。

你可以使用 schema.sql 文件直接建立表。 例如，在 MySQL 中，输入：

```
>mysqladmin -u root -p create blog
>mysql -u root -p blog < data/sql/lib.model.schema.sql
```

生成的 SQL 也对在其他环境中建立数据库有用， 或者改成其他的 DBMS。 如果连接设置在 propel.ini 中定义正确，你甚至能使用 symfony propel-insert-sql 命令去自动完成。

TIP 命令行也提供了一个基于文本文件来填充数据库和数据的机制。 看第 16 章来获得更多的关于 propel-load-data 和 YAML 格式文件的信息。

从已有数据库建立 YAML 数据模型

~~Symfony~~symfony 能使用 Creole 数据库访问层来访问数据库从而生成 schema.yml 文件， 这要归功于内省（introspection）机制（获取数据库表结构的功能）。当你做反向工程时候特别有用，或者如果你期待在有对象模型前先做基于数据库的工作。

为了这个目的，你需要确定项目的 propel.ini 文件指向了正确的数据库和包含了所有的连接设置，然后调用 propel-build-schema 命令：

```
>symfony propel-build-schema
```

从数据库结构生成的全新的 schema.yml 文件会放在 config/目录下。 你能基于这个设计建立模型。

生成设计命令行十分强大，能在你的 schema 中增加一系列的数据库相关的信息。 如 YAML 格式不能处理所有的数据库信息， 你能生成一个 XML 设计来代替。 要生成 XML 文件只要在 build-schema 上加一个 xml 参数就行了：

```
>symfony propel-build-schema xml
```

你会生成一个 schema.xml 文件用来替代生成 schema.yml 文件，这和 Propel 完全兼容， 包含了所有的 vendor 信息。 但是注意，生成的 XML 设计会有些冗长和难以阅读。

SIDEBAR propel.ini 配置

propel-build-sql 和 propel-build-schema 任务不使用定义于 databases.yml 文件中的连接设置。 替代的是，这些任务使用了其他文件中的连接设置，调用存于项目 config/目录下的 propel.ini：

```
propel.database.createUrl = mysql://login:passwd@localhost
propel.database.url       = mysql://login:passwd@localhost/blog
```

这个文件包含了其他的设置用来配置 Propel 生成器来生成和 symfony 兼容的模型类。 多数设置是内部的，除了少数一些外，其他大多用户都不需要去关心：

```
// 在 symfony 中，基类是自动载入的
// 设置这个为 true 使用 include_once
// （对性能有一些负面的影响）
propel.builder.addIncludes = false

// 生成的类默认是没有注释的
// 设置这个为 true 可以给基类增加注释
// （对性能有一些负面的影响）
propel.builder.addComments = false

// 默认不处理行为（Behaviors）
// 设置为 true 可以处理他们
propel.builder.AddBehaviors = false
```

修改 propel.ini 设置后， 别忘了重建模型让所有的修改生效。

总结

~~Symfony~~symfony 使用 Propel 作为 ORM，Creole 作为数据库抽象层。这意味着你需要首先在生成对象模型类前用数据库设计的 YAML 语法来描述数据的关系。然后，在运行时，使用类的方法和 peer 类获得关于记录或者一组记录的信息。你能覆盖他们，用自定义类中增加方法来非常容易的扩展模型。 连接设置定义在 databases.yml 文件中， 用来支持多于一个连接。 命令行包含了特别的任务来避免重复数据定义。

模型层是 symfony 框架中最复杂的。 为什么这么说的一个理由就是数据处理是一个复杂的问题。 对于网站相关的安全性是至关紧要的并不能被忽略的。 另一个理由就是 symfony 更适合企业级的中等到大型的数据库应用。 在这些应用程序中，symfony 模型层提供的自动机制确实省了很多时间，值得去研究和学习。

所以，不要犹豫了，花一些时间来测试这些模型对象和方法让自己完全了解他们。 你的应用程序的可靠性和可扩展性能让你觉得学习的花费很值得。