【译】微型ORM：PetaPoco - Qi Fei

PetaPoco是一款适用于.Net 和Mono的微小、快速、单文件的微型ORM。

PetaPoco有以下特色：

- 微小，没有依赖项……单个的C#文件可以方便的添加到任何项目中。
- 工作于严格的没有装饰的Poco类，和几乎全部加了特性的Poco类
- Insert/Delete/Update/Save and IsNew 等帮助方法。
- 分页支持：自动得到总行数和数据
- 支持简单的事务
- 更好的支持参数替换，包括从对象属性中抓取命名的参数。
- 很好的性能，剔除了Linq，并通过Dynamic方法快速的为属性赋值
- T4模板自动生成Poco类
- 查询语言是Sql……不支持别扭的fluent或Linq语法（仁者见仁，智者见智）
- 包含一个低耦合的Sql Builder类，让内联的Sql更容易书写
- 为异常信息记录、值转换器安装和数据映射提供钩子。（Hooks for logging exceptions, installing value converters and mapping columns to properties without attributes.）
- 兼容SQL Server, SQL Server CE, MySQL, PostgreSQL and Oracle。
- 可以在.NET 3.5 或Mono 2.6或更高版本上运行
- 在.NET 4.0 和Mono 2.8下支持dynamic
- NUnit单元测试
- 开源（Apache License）
- 所有功能大约用了1500行代码

代码展示：

首先，定义一个Poco类：

```
// Represents a record in the "articles" table
public class article
{
    public long article_id { get; set; }
    public string title { get; set; }
    public DateTime date_created { get; set; }
    public bool draft { get; set; }
    public string content { get; set; }
}
```

接下来，创建一个PetaPoco.Database，来执行查询：

```
// Create a PetaPoco database object
```

```
var db=new PetaPoco.Database("connectionStringName");

// Show all articles
foreach (var a in db.Query<article>("SELECT * FROM articles"))
{
    Console.WriteLine("{0} - {1}", a.article_id, a.title);
}
```

得到一个scalar：

```
long count=db.ExecuteScalar<long>("SELECT Count(*) FROM articles");
```

得到一行记录：

```
var a = db.SingleOrDefault<article>("SELECT * FROM articles WHERE article_id=@0", 123));
```

获取分页数据：

PetaPoco能够自动完成分页请求

```
var result=db.Page<article>(1, 20, // <-- page number and items per page
        "SELECT * FROM articles WHERE category=@0 ORDER BY date_posted DESC", "coolstuff");
```

你将会得到一个PagedFetch对象：

```
public class Page<T> where T:new()
{
    public long CurrentPage { get; set; }
    public long ItemsPerPage { get; set; }
    public long TotalPages { get; set; }
    public long TotalItems { get; set; }
    public List<T> Items { get; set; }
}
```

PetaPoco在背后为我们做了一下处理：

1. 生成并执行一个查询，得到匹配的数据行数
2. 修改原始的查询语句，使其只得到所有匹配数据的一个子集

你现在已经拥有了一个展示单页数据的一切东西和一个分页控制器，他们被封装在一个现成的小对象中。（You now have everything to display a page of data and a pager control all wrapped up in one handy little object!）

Query vs Fetch

Database 对象有两个获取数据的方法：Query 和Fetch。这两个方法非常相似，不同的是Fetch方法返回一个POCO类的List<>，而Query使用 `yield return` 迭代所有数据，这些数据并没有加载到内存中。

不带查询的命令使用Execute 方法执行一个不带查询的命令：

```
db.Execute("DELETE FROM articles WHERE draft<>0");
```

Inserts、Updates 和 Deletes

PetaPoco提供了insert、update和delete操作的帮助。

在插入一条记录时，你需要指定插入的表名和主键：

```
// Create the article
var a=new article();
a.title="My new article";
a.content="PetaPoco was here";
a.date_created=DateTime.UtcNow;

// Insert it
db.Insert("articles", "article_id", a);

// by now a.article_id will have the id of the new article
```

更新记录也一样：

```
// Get a record
var a=db.SingleOrDefault<article>("SELECT * FROM articles WHERE article_id=@0", 123);

// Change it
a.content="PetaPoco was here again";

// Save it
db.Update("articles", "article_id", a);
```

或者你可以传一个匿名类来更新一部分字段。下面的代码只更新article的title字段：

```
db.Update("articles", "article_id", new { title="New title" }, 123);
```

删除：

```
// Delete an article extracting the primary key from a record
```

```
db.Delete("articles", "article_id", a);

// Or if you already have the ID elsewhere
db.Delete("articles", "article_id", null, 123);
```

修饰POCO类

在上面的例子中，必须指明表名和主键是很烦人的，你可以在你的Poco类中附加这些信息：

```
// Represents a record in the "articles" table
[PetaPoco.TableName("articles")]
[PetaPoco.PrimaryKey("article_id")]
public class article
{
    public long article_id { get; set; }
    public string title { get; set; }
    public DateTime date_created { get; set; }
    public bool draft { get; set; }
    public string content { get; set; }
}
```

简化后的insert、update、delete：

```
// Insert a record
var a=new article();
a.title="My new article";
a.content="PetaPoco was here";
a.date_created=DateTime.UtcNow;
db.Insert(a);

// Update it
a.content="Blah blah";
db.Update(a);

// Delete it
db.Delete(a);
```

delete和update的其它方式

```
// Delete an article
db.Delete<article>("WHERE article_id=@0", 123);
```

```
// Update an article
db.Update<article>("SET title=@0 WHERE article_id=@1", "New Title", 123);
```

你还可以告诉POCO忽略某列

```
public class article
{
    [PetaPoco.Ignore]
    public long SomeCalculatedFieldPerhaps
    {
        get; set;
    }
}
```

或许你喜欢一点更详细的描述。和自动映射所有列相比，你可以通过使用类和列的属性来指明哪些列需要映射。

```
// Represents a record in the "articles" table
[PetaPoco.TableName("articles")]
[PetaPoco.PrimaryKey("article_id")]
[PetaPoco.ExplicitColumns]
public class article
{
    [PetaPoco.Column]publiclong article_id { get; set;}
    [PetaPoco.Column]publicstring title { get; set;}
    [PetaPoco.Column]publicDateTime date_created { get; set;}
    [PetaPoco.Column]public bool draft { get; set;}
    [PetaPoco.Column]publicstring content { get; set;}
}
```

它可以结合partial class 很好的工作，把需要绑定的字段放在一个.cs文件中，把计算得到的和别的有用的属性添加到分开的文件中，而不用去考虑DAL。

Hey！是不是已经有装饰POCO数据库的标准属性了呢？

好吧，PetaPoco仅支持少数几个，因为我不想引起混乱。

Hey！稍等……它们不是POCO对象了！

当然，它们打破了严格的POCO概念，但使用它们可以让POCO更容易工作。

T4 模板

Writing all those POCO objects can soon get tedious and error prone... so PetaPoco includes a T4

template that can automatically write classes for all the tables in your your SQL Server, SQL Server CE, MySQL, PostgreSQL or Oracle database.

Using the T4 template is pretty simple. The git repository includes three files (The NuGet package adds these to your project automatically in the folder `\Models\Generated`).

- PetaPoco.Core.ttinclude - includes all the helper routines for reading the DB schema
- PetaPoco.Generator.ttinclude - the actual template that defines what's generated
- Database.tt - the template itself that includes various settings and includes the two other ttinclude files.

A typical Database.tt file looks like this:

```
<#@ include file="PetaPoco.Core.ttinclude" #>
<#
      // Settings
      ConnectionStringName="jab";
      Namespace=ConnectionStringName;
      DatabaseName=ConnectionStringName;
      stringRepoName=DatabaseName+"DB";
      bool GenerateOperations=true;

      // Load tables
      var tables =LoadTables();

#>
<#@ include file="PetaPoco.Generator.ttinclude" #>
```

To use the template:

1. Add the three files to you C# project
2. Make sure you have a connection string and provider name set in your app.config or web.config file
3. Edit ConnectionStringName property in Records.tt (ie: change it from "jab" to the name of your connection string)
4. Save Database.tt.

All going well Database.cs should be generated with POCO objects representing all the tables in your database. To get the project to build you'll also need to add PetaPoco.cs to your project and ensure it is set to compile (NuGet does this for you) .

The template is based on the SubSonic template. If you're familiar with its ActiveRecord templates you'll find PetaPoco's template very similar.

自动的Select语句

当使用PetaPoco时，大多数查询都以"SELECT * FROM table"开头。鉴于我们现在可以从POCO对象的attribute中得到表名，我们没有理由不自动生成Select语句。

如果你运行一个不以select开头的查询， PetaPoco会自动的将它加上：

```
// Get a record
var a=db.SingleOrDefault<article>("SELECT * FROM articles WHERE article_id=@0",123);
```

可以简写：

```
// Get a record
var a=db.SingleOrDefault<article>("WHERE article_id=@0",123);
```

PetaPoco实际上并不生成"SELECT *"，它更准确的得到要查询的列名。

IsNew 和Save 方法

有时你有一个POCO，你想要知道数据库中是否已经存在。因为我们有主键，我们所要做的检查是该属性被设置为别的值还是默认值。

检测是否为新增:

```
// Is this a new record
if(db.IsNew(a))
{
	// Yes it is...
}
```

和它相关联还有一个Save方法，它将根据判断的结果执行Insert或Update。

```
// Save a new or existing record
db.Save(a);
```

事务

事务相当的简单:

```
using (var scope=db.Transaction)
{
	// Do transacted updates here

	// Commit
	scope.Complete();
}
```

事务可以是嵌套的，因此你可以调用其它包含事务的方法，或者被包含在单个的事务中。当所有事务都执行完成了，事务将会提交，否则所有操作都将回滚。

注意：为了使用事务，所有操作都需要相同的PetaPoco Database对象实例。你很可能想到IOC容器中为在每一个http请求或每一个线程共享同一个实体，我喜欢用StructureMap。

Linq从哪儿实现？

没有任何支持。我在Subsonic中使用Linq很长时间，我发现自己下降到使用CodingHorror 来做这些事情，因为：

- 不能用简单的Linq实现
- 在.NET 下工作，但不支持Mono（尤其是Mono 2.6）
- 低效。例如：Subsonic 中activerecord.SingleOrDefault(x=x.id==123)的效率比CodingHorror 低20倍。

Now that I've got CodingHorror all over the place it bugs me that half the code is Linq and half is SQL.

Also, I've realized that for me the most annoying thing about SQL directly in the code is not the fact that it's SQL but that it's nasty to format nicely and to build up those SQL strings.

So...

PetaPoco's SQL Builder

目前已经有很多构建SQL的API，以下是我的版本，它确实很基础！

我的目标是格式化SQL更简单，并且通过适当的参数替换达到防止SQL注入的作用。这不能保证SQL语法的正确，也不支持使用intellisense。

以下是非常基础的写法：

```
var id=123;
var a=db.Query<article>(PetaPoco.Sql.Builder
      .Append("SELECT * FROM articles")
      .Append("WHERE article_id=@0", id)
)
```

很管用吧，参数索引器来通过调用.Append是多么酷啊！【Big deal right? Well what's cool about this is that the parameter indicies are specific to each `.Append` call:】

```
var id=123;
var a=db.Query<article>(PetaPoco.Sql.Builder
      .Append("SELECT * FROM articles")
      .Append("WHERE article_id=@0", id)
      .Append("AND date_created<@0",DateTime.UtcNow)
)
```

你也可以根据条件构建SQL：

```
var id=123;
var sql=PetaPoco.Sql.Builder
      .Append("SELECT * FROM articles")
      .Append("WHERE article_id=@0", id);
```

```
if(start_date.HasValue)
      sql.Append("AND date_created>=@0", start_date.Value);

if(end_date.HasValue)
      sql.Append("AND date_created<=@0", end_date.Value);

var a=db.Query<article>(sql)
```

注意到每个append调用都用到餐厨@0了吗？PetaPoco构建整个列表的参数，将这些参数索引更新到内部。

你也可以使用命名参数，然后他会在传递的参数中找到合适的属性名。

```
sql.Append("AND date_created>=@start AND date_created<=@end",
                        new
                        {
                              start=DateTime.UtcNow.AddDays(-2),
                              end=DateTime.UtcNow
                        }
                  );
```

不管是数字的还是命名的参数，如果任何一个参数不能被推断出来，都会抛出一个异常。

这里还有几个创建SQL的公用方法：

```
var sql=PetaPoco.Sql.Builder()
                  .Select("*")
                  .From("articles")
                  .Where("date_created < @0",DateTime.UtcNow)
                  .OrderBy("date_created DESC");
```

跟踪Sql命令

有些时候能够看到执行的Sql语句会非常有用，PetaPoco为此提供了三个属性：

- LastSQL - 非常明显，不解释
- LastArgs - 传递的参数数组
- LastCommand - SQL语句和参数字符串

在调试器中查看LastCommand属性能够简单的看到执行了那些操作！

OnException Handler Routine

PetaPoco所执行的Sql命令都封装在try/catch语句块中，所有的异常信息都会传递给OnException虚方法。通过记录这些异常（或在这个方法中设置断点），你可以轻松的跟踪那些地方出现了问题。

More

上面的内容展示了最基本的PetaPoco使用方法，想了解更多，请查看这些博客内容。

原文地址：http://www.toptensoftware.com/petapoco/