Dapper – micro-ORM

Недавно я рассказывал про легковесную ORM BLToolkit, и при поиске и изучении материала неизбежно наталкивался на сравнение BLT с другими разработками в области мапирования данных на бизнес-объекты. Одним из самых привлекательных вариантов по скорости, а так же по вниманию общественности, оказался Dapper.

Dapper – это даже не легковесная, а микро-ORM система для чтения (в основном) информации из реляционных баз данных. Данная микро-ORM система является разработкой Сэма Сафрона (Sam Saffron) для Stack Overflow, где она работает в связке с Linq2Sql. Для такого большого и посещаемого ресурса как Stack Overflow очень важно быстро получать информацию из базы данных, так как большинство пользователей просматривает ответы, использует их в своей работе, и сравнительно редко пишет. Для записи информации, что требуется значительно реже, до сих пор самым удобным и быстрым остается Linq2Sql.

О системе

Dapper – это по сути один файл с исходным кодом, который надо включить в свой проект. Dapper работает в некотором роде классом помощником, расширяя стандартный интерфейс IDbConnection с помощью extended методов. Т.е. данному фреймворку абсолютно без разницы, с какой базой вы работаете, если соединение с ней построено на указанном интерфейсе.

Запросы Dapper принимает только в текстовом виде, нет поддержки LINQ. Модификация бизнес-классов в основном не требуется, в редких случая потребуется использовать свойства вместо полей класса.

В Dapper нет поддержки маппинга сложных связанных объектов, так же как и в BLToolkit.

Ключевой особенностью Dapper является скорость и на страничке проекта приведены данные измерений времени, которое необходимо для выполнения 500 запросов выборки.

Производительность для операции SELECT для 500 итераций с мапингом - POCO объекты

Реализация	Длительность	Заметки
Hand coded (using a `SqlDataReader`)	47ms	
Dapper `ExecuteMapperQuery<Post>`	49ms	
PetaPoco	52ms	Can be faster
BLToolkit	80ms	
SubSonic CodingHorror	107ms	
NHibernate SQL	104ms	
Linq 2 SQL `ExecuteQuery`	181ms	
Entity framework ExecuteStoreQuery	631ms	

Производительность для операции SELECT для 500 итераций с мапингом - динамические объекты

Реализация	Длительность	Заметки
Dapper `ExecuteMapperQuery (dynamic)`	48ms	
Massive	52ms	
Simple.Data	95ms	

Производительность для операции SELECT для 500 итераций с мапингом – типичное использование (сложный маппинг объектов)

Реализация	Длительность	Заметки
Linq 2 SQL `CompiledQuery`	81ms	Не совсем типичное использование, привлекается много сложносоставных объектов
NHibernate HQL	118ms	
Linq 2 SQL	559ms	
Entity framework	859ms	
SubSonic `ActiveRecord.SingleOrDefault`	3619ms	

Все тесты можно взять со страницы проекта в виде отдельного файла с C# классом.

Ограничения и оговорки

Каждый запрос к базе данных кэшируется, что позволяет быстро материализовывать объекты и распознавать параметры запросов. Текущая реализация кэша основана на ConcurrentDictionary. Объекты попадающие в это хранилище не очищаются во время работы приложения, так что если вы генерируете SQL запросы на ходу **без использования параметров**, возможны проблемы с памятью, так количество уникальных запросов сильно вырастет.

Простота фреймворка означает, что многие возможности которые идут с полновесными ORM опущены. Например нет identity map, нет помощников для обновления/выборки данных и так далее.

Так же Dapper не управляет временем жизни соединения и он предполагает что **соединение уже открыто И используется монопольно**, т.е. не существует других процессов читающих данные в текущем соединении. Если только не задействована опция MARS - Multiple Active Result Sets.

По умолчанию при запросе включено полное кэширование, т.е. произойдет загрузка всех данных в память. Так что будьте осторожны с большими объемами данных.

Установка

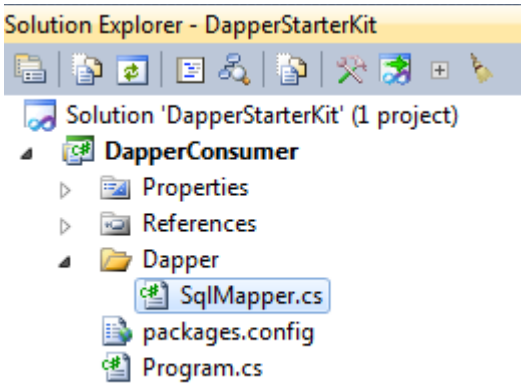
Установить dapper легче всего с помощью NuGet. На данный момент на сайте указано, что рекомендуемая версия 1.7 и выпущена она 5 ноября 2011 года.

```
install-package dapper
```

Результат:

```
PM> install-package dapper
Successfully installed 'Dapper 1.7'.
Successfully added 'Dapper 1.7' to DapperConsumer.
```

И новая папка с файлом в проекте. Что достаточно необычно для распространения через NuGet. Зато все желающие могут почитать код и осознать, насколько много вещей еще можно изучать.



Или как уже было сказано выше, можете просто включить файл с реализацией Dapper в проект.

Подготовка

Проверять удобство работы будем на тех же классах и данных, что были в статье про BLToolkit. Однако все равно наверно лучше указать здесь все необходимые скрипты и классы.

В качестве эксперимента у нас будут выступать некоторые сводные данные по людям, и их адреса. Итак, скрипт для создания данных в базе:

```
set xact_abort on
begin tran

create table Address(
        AddressId uniqueidentifier primary key default newid()
       ,Country   nvarchar(100)
       ,Region    nvarchar(100)
       ,City      nvarchar(100)
       ,Street    nvarchar(100)
       ,Residence nvarchar(100)
)
go

create table Person(
        PersonId     uniqueidentifier primary key default newid()
       ,Name         nvarchar(100) not null
       ,Birth        datetime
```

```
       ,Resident     bit default 0
       ,Gender             char default 'm'
       ,Weight             int
       ,Height             decimal (3,2)
       ,AddressId    uniqueidentifier
       constraint fk_address2person foreign key (AddressId) references Address(AddressId)
)
go

insert into Person values(newid(), 'Adam', '01-01-1980', 0, 'm', 80, 1.80, null)
insert into Person values(newid(), 'Amy', '04-07-1986', 1, 'f', 52, 1.65, null)

insert into Address values(newid(), 'Russia', 'N.Novgorod', 'N.Novgorod','Minina', '1A')
insert into Address values(newid(), 'Russia', 'Msk', 'Msk', 'Lenina', '40 - 123')

update Person
   set Person.AddressId = Address.AddressId
  from Address
 where city = 'Msk' and Name = 'Adam'

update Person
   set Person.AddressId = Address.AddressId
  from Address
 where city <> 'Msk' and Name = 'Amy'

commit
```

Чтобы два раза не вставать создали сразу все таблицы и заполнили их данными.

Сразу создадим классы, которые будем заполнять информацией из базы данных:

```
public class Person {
    public Guid PersonId;
    public string Name;
    public DateTime Birth;
    public bool Resident;
    public string Gender;
    public int Weight;
    public decimal Height;

    public Guid AddressId;
    public Address Address;
}


public class Address {
    public Guid AddressId;
    public string Country;
    public string Region;
    public string City;
    public string Street;
    public string Residence;
}
```

Все, структуры данных готовы, можно приступать к экспериментам. Ах да, еще надо упомянуть про отдельное создание соединения с базой данных, так как Dapper сам не контролирует время жизни соединения. Создадим отдельный класс, в котором будем писать методы с различными вариантами работы фреймворка. Пусть этот класс называется Dapper, и конструктор будет создавать соединение с базой данных.

```
public class Dapper {
    private readonly SqlConnection con;

    public Dapper() {
        var cs = new SqlConnectionStringBuilder {
                                InitialCatalog = "Test",
                                IntegratedSecurity = true,
                                DataSource = @"LETHIATHAN\LCF11CTP3"
                                                    };
        con = new SqlConnection(cs.ConnectionString);
    }
}
```

Думаю, что тут пояснять ничего не требуется уже. Далее шаблоном для каждого метода будет выступать следующая заготовка:

```
    public void MethodName() {
        con.Open();

        // код будет здесь

        con.Close();
    }
```

Базовые возможности

У нас все готово к тому, чтобы начать работу с Dapper:

- Таблицы и данные готовы
- Классы в C# коде есть
- Dapper в проект включен

Простой запрос

Начнем с самого простого, что можно только себе представить. Сделаем выборку полной таблицы Person и заполним соответствующий класс. Для этого воспользуемся методом расширения со строгой типизацией Query, который принимает текст запроса в качестве обязательного параметра.

```
    public void SimpleStrongTypedSelect() {
        con.Open();

        var sql = "Select * from Person";
        var persons = con.Query<Person>(sql);
        foreach (var p in persons)
            Console.WriteLine("{0} {1} {2}", p.Name, p.Birth, p.Resident);

        con.Close();
    }
```

Открываем соединение с базой, пишем текст запроса, вызываем метод расширения для соединения, получаем готовые классы. Профит! Легко и быстро до безобразия. Однако метод Query не так прост, у него много необязательных параметров для тонкой настройки. Полная сигнатура выглядит так:

```
public static IEnumerable<T> Query<T>(
             this IDbConnection cnn,
             string sql,
             dynamic param = null,
             IDbTransaction transaction = null,
             bool buffered = true,
             int? commandTimeout = null,
             CommandType? commandType = null
            )
```

- Sql – текст запроса,
- Param – значения параметров, если они включены в запрос
- Transaction – включение в транзакцию
- Buffered – полное чтение результатов или по мере необходимости
- CommandTimeout – ограничение времени выполнения запроса
- CommandType – тип запроса: запрос, команда

Запрос с параметрами

Далее, немаловажно уметь передавать параметры в запрос. В запросе параметры следуют нотации MSSQL, т.е. начинаются со знака @. **Параметры передаются в виде анонимного класса**, и только в виде него. Имена полей должны соответствовать именам параметрам, причем **регистр имеет значение**. Это правило действует для всех параметров.

```
public void SimpleStrongTypedSelectWithParam() {
    con.Open();

    var sql = "Select * from Person where weight < @Weight";
    var persons = con.Query<Person>(sql, new {Weight = 70});
    foreach (var p in persons)
        Console.WriteLine("{0} {1} {2}", p.Name, p.Birth, p.Resident);

    con.Close();
}
```

Оказалось не сложно, верно?

Простой не типизированный запрос

Можно опустить строгую типизацию и получить результат в виде dynamic типа.

```
public void SimpleDynamicSelectWithParam() {
    con.Open();

    var sql = "Select * from Person where weight < @Weight";
    var persons = con.Query(sql, new {Weight = 70});
    foreach (var p in persons)
        Console.WriteLine("{0} {1} {2}", p.Name, p.Birth, p.Resident);

    con.Close();
}
```

Команда, возвращающая набор данных

Работа с хранимыми процедурами не сильно отличается от работы с простыми запросами. Необходимо указать имя процедуры и что это все же процедура, а не запрос.

```
public void ExecuteSimpleSelectCommand() {
    con.Open();

    var persons = con.Query<Person>("Person_GetOlderThan",
                                    new {someOtherParam = 70, age = 30, weight = 60},
                                    commandType: CommandType.StoredProcedure);
    foreach (var p in persons)
        Console.WriteLine("{0} {1} {2}", p.Name, p.Birth, p.Resident);

    con.Close();
}
```

В данном случае использован строго типизированный запрос, и чтобы не писать множество null, я использовал именованный параметр для указания типа запроса. CommandType взят из пространства имен System.Data.

Как видите и здесь нет ничего сложного, подводных камней нет тоже никаких. Вообще пользоваться Dapper'ом оказалось весьма просто и легко.

Команды без результирующего набора данных

Естественно, что не все команды возвращают набор данных в результате своей работы. Для таких вызовов предназначен другой метод расширяющий интерфейс IDbConnetction – Execute.

Метод Execute в качестве обязательного параметра принимает текст запроса, опциональным является тип запроса. Вообще данная команда предназначена для любого типа операций, которые не возвращают данных.

```
public void ExecuteNonSelectCommand() {
    con.Open();

    var affectedRows = con.Execute("DumbProc",
                                    commandType: CommandType.StoredProcedure);
    Console.WriteLine(affectedRows);

    con.Close();
}
```

Результатом операции будет целое число, говорящее о количестве измененных строк.

Полная сигнатура метода:

```
public static int Execute(
                    this IDbConnection cnn,
                    string sql,
                    dynamic param = null,
                    IDbTransaction transaction = null,
                    int? commandTimeout = null,
                    CommandType? commandType = null
)
```

На этом можно завершить рассказ о базовых возможностях фреймворка.

Продвинутые возможности

К продвинутым возможностям можем отнести выполнение пакетной вставки данных, получение и разбор нескольких запросов за раз, множественный маппинг.

Пакетная вставка данных как пример использования списка параметров

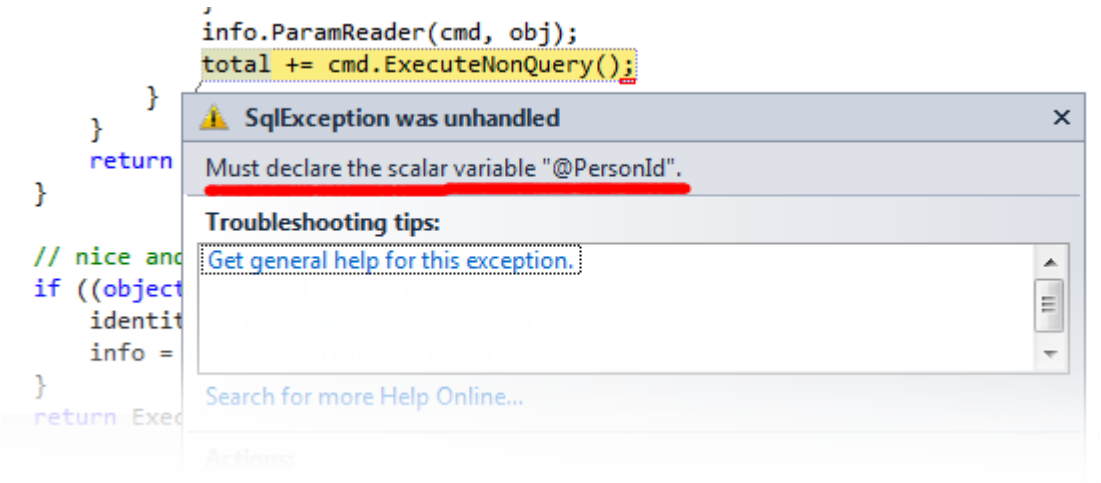
С помощью Dapper можно осуществлять пакетную вставку данных передав список данных. Т.е. у нас есть список людей для внесения в базу данных, и с помощью единственной команды можно это провернуть. Лучше сразу на примере показать:

```
    public void ExecuteNonSelectBulkCommand() {
        con.Open();

        var list = new List<Person>();
        for (var i = 1; i < 5; i++) {
            list.Add(new Person {
                                    PersonId = Guid.NewGuid(),
                                    Birth = new DateTime(1980, i, i),
                                    Name = "Pers " + i,
                                });
        }

        con.Execute("INSERT INTO Person (PersonId, Name, Birth) VALUES(@PersonId, @Name,
@Birth)"
                    , list);

        var persons = con.Query<Person>("Select * from Person");
        foreach (var p in persons)
            Console.WriteLine("{0} {1} {2}", p.Name, p.Birth, p.Resident);

        con.Close();
    }
```

Создаем несколько экземпляров класса Person, и передаем список в качестве параметра в команду Execute. Потом можно визуально проверить результат.

В качестве параметра для пакетной обработки может выступать любая коллекция, которая реализует интерфейс IEnumerable<T>.

Упс!



Ничего же не менялось в коде, и все переменные указаны верно, в чем же дело?

Дело в особенностях реализации Dapper и то, как построен класс Person. На данный момент PersonId является полем, так попробуем его изменить на свойство. Дописываем get, set каждому полю и пробуем снова. Все заработало! Данный момент надо иметь в виду при передаче параметров списком.

Список параметров в запросах

Dapper достаточно интеллектуален и, когда надо, по списку параметров формирует единственный запрос вместо множества.

Пример:

```
public void ListEnumParams() {
    con.Open();

    var sql = "Select * from Person where name in @names";
    var names = new List<string> {"Pers 1", "Pers 2"};
    var persons = con.Query<Person>(sql, new {names});

    foreach (var p in persons)
        Console.WriteLine("{0} {1} {2}", p.Name, p.Birth, p.Resident);

    con.Close();
}
```

В данном случае необходимо список параметров оборачивать в анонимный класс, так как параметры представлены элементарными типами.

Если посмотреть активность по запросам в профайлере, то можно увидеть, что был единственный запрос:

```
exec sp_executesql N'Select * from Person where name in (@names1,@names2)',N'@names1
nvarchar(4000),@names2 nvarchar(4000)',@names1=N'Pers 1',@names2=N'Pers 2'
```

Маппинг сложносоставных методов

Можно научить фреймворк восстанавливать составные типы по сложным запросам в базу данных. Например, нам надо получить всех наших товарищей с заполненными адресами. Для этого не требуется вносить никаких дополнительных данных в исходные классы, все делается на уровне получения данных.

Мы снова воспользуемся Query, однако перегруженным вариантом метода, который принимает несколько шаблонных типов.

```
public void MultiMappings() {
    con.Open();

    var sql = "Select * from Person p join Address a on p.AddressId = a.AddressId";
    var persons = con.Query<Person, Address, Person>(sql, (pers, address) => {
                                                              pers.Address = address;
                                                              return pers;
                                                          });

    foreach (var p in persons)
        Console.WriteLine("{0} {1}", p.Name, p.Address.City);
    con.Close();
}
```

В угловых скобках сначала идут типы, на которые надо постараться замапить ответ, последним типом указывается, какие данные будут результирующими. В параметрах метода надо указать текст запроса и правило, по которому надо создавать сложносоставные объекты. Правилом в данном случае выступает лямбда-функция, но как вы понимаете, в общем случае туда можно передавать Func.

Запускаем.

И опять беда. Дело в том, что запрос в текущем виде вернет результат в виде составного набора полей из двух таблиц. Логическое разделение данных идет по первичным ключам, и Dapper исходит из предположения что они названы «Id». Если это не так, то требуется явно указать поле по которому происходит конкатенация таблиц.

Модифицируем код:

```
public void MultiMappings() {
    con.Open();

    var sql = "Select * from Person p join Address a on p.AddressId = a.AddressId";
    var persons = con.Query<Person, Address, Person>(sql, (pers, address) => {
                                                              pers.Address = address;
                                                              return pers;
                                                          },
                                                    splitOn: "AddressId");

    foreach (var p in persons)
        Console.WriteLine("{0} {1}", p.Name, p.Address.City);

    con.Close();
}
```

В метод Query добавили именованный атрибут **splitOn** с именем поля, по которому идет объединение таблиц.

Мульти запрос

Помимо всего прочего есть возможность сделать запрос из нескольких не связанных между собой таблиц. Может быть удобно, если процедура возвращает несколько наборов данных.

Для обработки таких запросов следует применять метод расширения QueryMultiple, который принимает текст запроса.

```
public void MultiSelect() {
    con.Open();
    var sql = @"Select * from Person
                select * from Address";

    using (var multi = con.QueryMultiple(sql)) {
        // iterated once
        var person = multi.Read<Person>().ToList();
        var address = multi.Read<Address>().ToList();

        foreach (var p in person)
            Console.WriteLine("{0}", p.Name);

        foreach (var a in address)
            Console.WriteLine("{0}", a.City);
    }
```

```
        con.Close();
    }
```

Ключевым моментом здесь является то, что надо сразу материализовать запрос:

```
var person = multi.Read<Person>().ToList();
```

Если не использовать метод ToList(), то при итерации в цикле foreach получите исключение говорящие о том, что возможен единственный проход по списку.

Varchar

Иногда могут быть конфликты в связи с типом параметра, тогда можно задавать их более явно. Например:

```
new DbString { Value = "abcde", IsFixedLength = true, Length = 10, IsAnsi = true }
```

Больше контроля над параметрами процедуры

Можно явно указывать какие параметры в процедуре будут модифицированы, и что является результатом.

```
    public void FancyParameters() {
        var p = new DynamicParameters();
        p.Add("@a", 11);
        p.Add("@b", dbType: DbType.Int32, direction: ParameterDirection.Output);
        p.Add("@c", dbType: DbType.Int32, direction: ParameterDirection.ReturnValue);

        con.Execute("spMagicProc", p, commandType: CommandType.StoredProcedure);

        var b = p.Get<int>("@b");
        var c = p.Get<int>("@c");
    }
```

Заключение

Рассмотрев Dapper можно сказать, что работая с ним, надо будет вести отдельный класс для запросов, с учетом возможных параметров. Очень интересно посмотреть его использование в реальном проекте, как идет работа с запросами, как они хранятся. Насколько гибко они построены. У меня есть некоторые мысли по этому поводу, но они требуют проверки и более основательного подхода и времени.

Dapper оставляет приятное впечатление, особенно с учетом его скорости работы, что немаловажный фактор.

Несмотря на предостережение построения запросов на лету, хотелось бы все же какой-то функционал по этой теме, чтобы можно было строить запросы в духе LINQ и затем переводить их в текст. При определенной сноровке, мне кажется, можно к этому приспособить Linq2Sql, следует поэкспериментировать в деле построения запросов и передачи параметров в запрос. Параметры в L2S именуются порядковыми номерами.

В таблице сравнения скоростей работы с POCO объектами упоминается PetaPoco, который имеет встроенный составитель запросов. Так что следующий на очереди PetaPoco, думаю, что это не менее интересный и быстрый микро-ORM. После этого уже можно будет выбирать, зная потенциальные возможности и недостатки исходя из конкретных задач и исходных данных.

Hard’n’heavy!

Violet Tape