Бесплатная электронная книга

УЧУСЬ

Dapper.NET

Free unaffiliated eBook created from
Stack Overflow contributors.

#dapper

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: dapper-net

It is an unofficial and free Dapper.NET ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Dapper.NET.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с Dapper.NET

замечания

Что такое Даппер?

Dapper - это микро-ORM для .Net, который расширяет ваш `IDbConnection` , упрощая настройку запросов, выполнение и чтение результатов.

Как мне это получить?

- github: https://github.com/StackExchange/dapper-dot-net
- NuGet: https://www.nuget.org/packages/Dapper

Общие задачи

- Основной запрос
- Выполнение команд

Версии

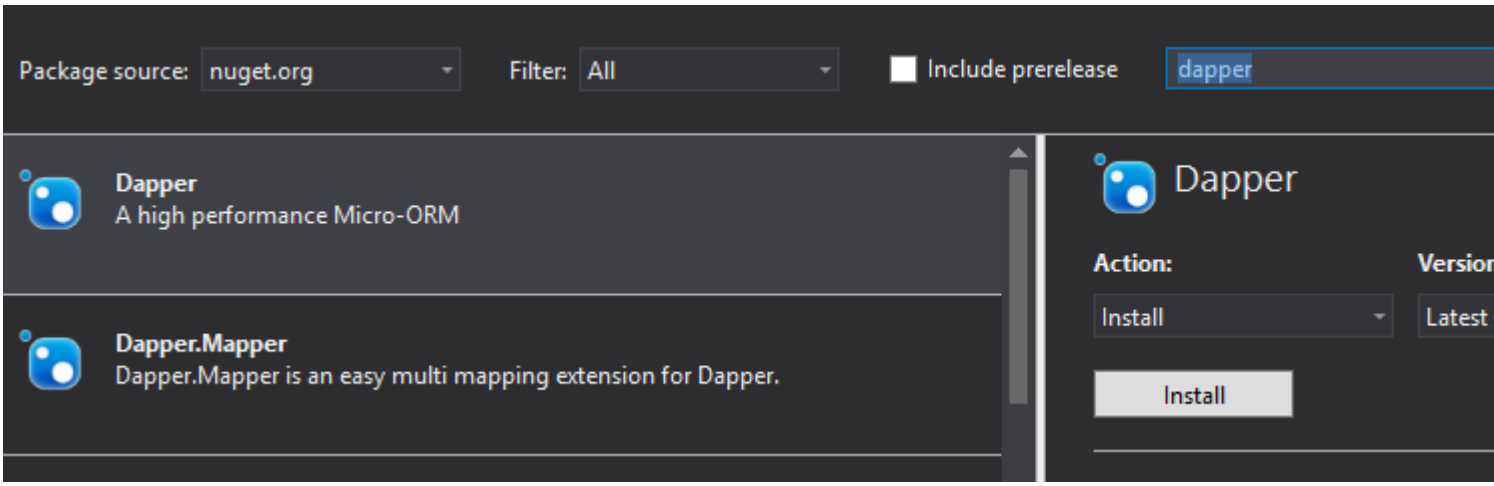
Версия	Заметки	Дата выхода
1.50.0	core-clr / asp.net 5.0 против RTM	2016-06-29
1.42.0		2015-05-06
1.40.0		2015-04-03
1.30.0		2014-08-14
1.20.0		2014-05-08
1.10.0		2012-06-27
1.0.0		2011-04-14

Examples

Установите Dapper из Nuget

Любой поиск в графическом интерфейсе Visual Studio:

Инструменты> Диспетчер пакетов NuGet> Управление пакетами для решения ... (Visual Studio 2015)



Или запустите эту команду в экземпляре Nuget Power Shell, чтобы установить последнюю стабильную версию

```
Install-Package Dapper
```

Или для конкретной версии

```
Install-Package Dapper -Version 1.42.0
```

Использование Dapper в C

```
using System.Data;
using System.Linq;
using Dapper;

class Program
{
    static void Main()
    {
        using (IDbConnection db = new
SqlConnection("Server=myServer;Trusted_Connection=true"))
        {
            db.Open();
            var result = db.Query<string>("SELECT 'Hello World'").Single();
            Console.WriteLine(result);
        }
    }
}
```

Завершение соединения в блоке « `Using` закроет соединение

Использование Dapper в LINQPad

LINQPad отлично подходит для тестирования запросов к базе данных и включает

интеграцию NuGet . Чтобы использовать Dapper в LINQPad, нажмите **F4,** чтобы открыть Свойства запроса, а затем выберите **Добавить NuGet** . Найдите **сеть dapper dot net** и выберите « **Добавить в запрос»** . Вы также захотите нажать « **Добавить пространства имен»** и выделить «Даппер», чтобы включить методы расширения в ваш запрос LINQPad.

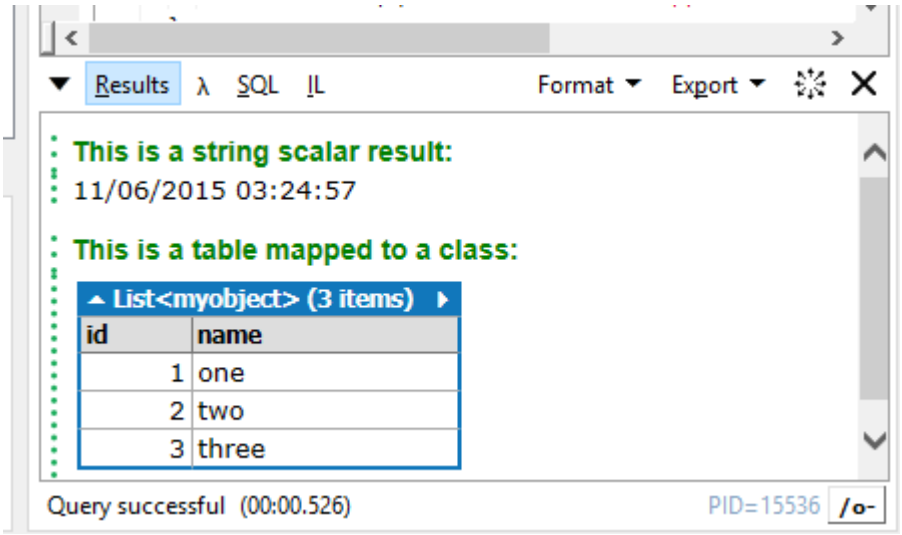
После того, как Dapper включен, вы можете изменить раскрывающийся список «Язык» на « **C # Program»** , отобразить результаты запроса к классам C # и использовать метод .Dump () для проверки результатов:

```
void Main()
{
    using (IDbConnection db = new SqlConnection("Server=myServer;Trusted_Connection=true")){
        db.Open();
        var scalar = db.Query<string>("SELECT GETDATE()").SingleOrDefault();
        scalar.Dump("This is a string scalar result:");

        var results = db.Query<myobject>(@"
        SELECT * FROM (
        VALUES (1,'one'),
            (2,'two'),
            (3,'three')
        ) AS mytable(id,name)");
        results.Dump("This is a table mapped to a class:");
    }
}

// Define other methods and classes here
class myobject {
    public int id { get; set; }
    public string name { get; set; }
}
```

Результаты при выполнении программы будут выглядеть так:



Прочитайте Начало работы с Dapper.NET онлайн: https://riptutorial.com/ru/dapper/topic/2/начало-работы-с-dapper-net

глава 2: Multimapping

Синтаксис

- `public static IEnumerable<TReturn> Query<TFirst, TSecond, TReturn>(this IDbConnection cnn, string sql, Func<TFirst, TSecond, TReturn> map, object param = null, IDbTransaction transaction = null, bool buffered = true, string splitOn = "Id", int? commandTimeout = null, CommandType? commandType = null)`
- `public static IEnumerable<TReturn> Query<TFirst, TSecond, TThird, TFourth, TFifth, TSixth, TSeventh, TReturn>(this IDbConnection cnn, string sql, Func<TFirst, TSecond, TThird, TFourth, TFifth, TSixth, TSeventh, TReturn> map, object param = null, IDbTransaction transaction = null, bool buffered = true, string splitOn = "Id", int? commandTimeout = null, CommandType? commandType = null)`
- `public static IEnumerable<TReturn> Query<TReturn>(this IDbConnection cnn, string sql, Type[] types, Func<object[], TReturn> map, object param = null, IDbTransaction transaction = null, bool buffered = true, string splitOn = "Id", int? commandTimeout = null, CommandType? commandType = null)`

параметры

параметр	подробности
спп	Соединение с базой данных, которое должно быть открыто.
SQL	Команда для выполнения.
типы	Массив типов в наборе записей.
карта	`Func<>` который обрабатывает построение результата возврата.
пары	Объект для извлечения параметров из.
сделка	Транзакция, в которой этот запрос является частью, если таковой имеется.
буферном	Будь или нет буферизация чтения результатов запроса. Это необязательный параметр с по умолчанию значением true. Когда буферизировано верно, результаты буферизуются в `List<T>` а затем возвращаются как `IEnumerable<T>` который безопасен для множественного перечисления. Если буферизация ложна, соединение sql будет открыто до тех пор, пока вы не закончите чтение, что позволит вам обрабатывать одну строку во времени в памяти. Множественные перечисления вызовут дополнительные подключения к базе данных. Хотя буферизованное ложное значение очень эффективно для сокращения использования памяти, если вы поддерживаете только очень мелкие фрагменты возвращенных записей, он имеет значительные накладные

параметр	подробности
	расходы по сравнению с желательным материализацией набора результатов. Наконец, если у вас есть многочисленные параллельные небуферизованные соединения sql, вам нужно рассмотреть головоломку пула соединений, вызывающую блокировку запросов до тех пор, пока соединения не станут доступными.
разделить на	Поле должно разделить и прочитать второй объект из (default: id). Это может быть список с разделителями-запятыми, если в записи содержится более 1 типа.
CommandTimeout	Количество секунд до истечения времени ожидания выполнения команды.
CommandType	Это хранимая процедура или пакет?

Examples

Простое отображение нескольких таблиц

Предположим, у нас есть запрос оставшихся всадников, которым необходимо заполнить класс Person.

название	Родившийся	Резиденция
Даниэль Деннетт	1942	Соединенные Штаты Америки
Сэм Харрис	1967	Соединенные Штаты Америки
Ричард Докинз	1941	Объединенное Королевство

```
public class Person
{
    public string Name { get; set; }
    public int Born { get; set; }
    public Country Residience { get; set; }
}

public class Country
{
    public string Residence { get; set; }
}
```

Мы можем заполнить класс person, а также свойство Residence экземпляром страны, используя перегруженный `Query<>` который принимает `Func<>` который может

использоваться для компоновки возвращаемого экземпляра. `Func<>` может принимать до 7 типов ввода с окончательным общим аргументом, всегда являющимся типом возврата.

```
var sql = @"SELECT 'Daniel Dennett' AS Name, 1942 AS Born, 'United States of America' AS
Residence
UNION ALL SELECT 'Sam Harris' AS Name, 1967 AS Born, 'United States of America' AS Residence
UNION ALL SELECT 'Richard Dawkins' AS Name, 1941 AS Born, 'United Kingdom' AS Residence";

var result = connection.Query<Person, Country, Person>(sql, (person, country) => {
        if(country == null)
        {
            country = new Country { Residence = "" };
        }
        person.Residience = country;
        return person;
    },
    splitOn: "Residence");
```

Обратите внимание на использование `splitOn: "Residence"` который является 1-м столбцом следующего типа класса, который будет заполнен (в данном случае `Country`). Dapper автоматически ищет столбец с именем *Id* для разделения, но если он не найдет его, а `splitOn` не будет предоставлен, `System.ArgumentException` будет `splitOn` полезным сообщением. Поэтому, хотя это необязательно, вам обычно нужно предоставить значение `splitOn` .

Отображение «один ко многим»

Давайте рассмотрим более сложный пример, содержащий отношения «один ко многим». Теперь наш запрос будет содержать несколько строк, содержащих повторяющиеся данные, и нам нужно будет справиться с этим. Мы делаем это с поиском в закрытии.

Запрос слегка меняется, как и классы примеров.

Я бы	название	Родившийся	CountryId	Название страны	BookID	BookName
1	Даниэль Деннетт	1942	1	Соединенные Штаты Америки	1	Мозговые штурмы
1	Даниэль Деннетт	1942	1	Соединенные Штаты Америки	2	Колено
2	Сэм Харрис	1967	1	Соединенные Штаты Америки	3	Моральный пейзаж
2	Сэм	1967	1	Соединенные	4	Пробуждение:

Я бы	название	Родившийся	CountryId	Название страны	BookID	BookName
	Харрис			Штаты Америки		руководство к духовности без религии
3	Ричард Докинз	1941	2	Объединенное Королевство	5	Магия реальности: как мы знаем, что действительно верно
3	Ричард Докинз	1941	2	Объединенное Королевство	6	Аппетит к удивлению: создание учёного

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Born { get; set; }
    public Country Residience { get; set; }
    public ICollection<Book> Books { get; set; }
}

public class Country
{
    public int CountryId { get; set; }
    public string CountryName { get; set; }
}

public class Book
{
    public int BookId { get; set; }
    public string BookName { get; set; }
}
```

Словарь, `remainingHorsemen` всадниках, будет заполнен полностью материализованными экземплярами объектов человека. Для каждой строки результата запроса передаются сопоставленные значения экземпляров типов, определенных в аргументах лямбда, и вам решать, как справиться с этим.

```
            var sql = @"SELECT 1 AS Id, 'Daniel Dennett' AS Name, 1942 AS Born, 1 AS
CountryId, 'United States of America' AS CountryName, 1 AS BookId, 'Brainstorms' AS BookName
UNION ALL SELECT 1 AS Id, 'Daniel Dennett' AS Name, 1942 AS Born, 1 AS CountryId, 'United
States of America' AS CountryName, 2 AS BookId, 'Elbow Room' AS BookName
UNION ALL SELECT 2 AS Id, 'Sam Harris' AS Name, 1967 AS Born, 1 AS CountryId,  'United States
of America' AS CountryName, 3 AS BookId, 'The Moral Landscape' AS BookName
UNION ALL SELECT 2 AS Id, 'Sam Harris' AS Name, 1967 AS Born, 1 AS CountryId,  'United States
```

```
of America' AS CountryName, 4 AS BookId, 'Waking Up: A Guide to Spirituality Without Religion'
AS BookName
UNION ALL SELECT 3 AS Id, 'Richard Dawkins' AS Name, 1941 AS Born, 2 AS CountryId,  'United
Kingdom' AS CountryName, 5 AS BookId, 'The Magic of Reality: How We Know What`s Really True'
AS BookName
UNION ALL SELECT 3 AS Id, 'Richard Dawkins' AS Name, 1941 AS Born, 2 AS CountryId,  'United
Kingdom' AS CountryName, 6 AS BookId, 'An Appetite for Wonder: The Making of a Scientist' AS
BookName";

var remainingHorsemen = new Dictionary<int, Person>();
connection.Query<Person, Country, Book, Person>(sql, (person, country, book) => {
    //person
    Person personEntity;
    //trip
    if (!remainingHorsemen.TryGetValue(person.Id, out personEntity))
    {
        remainingHorsemen.Add(person.Id, personEntity = person);
    }

    //country
    if(personEntity.Residience == null)
    {
        if (country == null)
        {
            country = new Country { CountryName = "" };
        }
        personEntity.Residience = country;
    }

    //books
    if(personEntity.Books == null)
    {
        personEntity.Books = new List<Book>();
    }

    if (book != null)
    {
        if (!personEntity.Books.Any(x => x.BookId == book.BookId))
        {
            personEntity.Books.Add(book);
        }
    }

    return personEntity;
},
splitOn: "CountryId,BookId");
```

Обратите внимание, как аргумент `splitOn` представляет собой список с разделителями-запятыми из первых столбцов следующего типа.

Отображение более 7 типов

Иногда количество отображаемых типов превышает 7, предоставленное Func <>, которое выполняет конструкцию.

Вместо использования `Query<>` с входами аргумента generic type мы предоставим типы для отображения в виде массива, а затем функцию отображения. Помимо первоначальной

ручной настройки и литья значений, остальная часть функции не изменяется.

```
            var sql = @"SELECT 1 AS Id, 'Daniel Dennett' AS Name, 1942 AS Born, 1 AS
CountryId, 'United States of America' AS CountryName, 1 AS BookId, 'Brainstorms' AS BookName
UNION ALL SELECT 1 AS Id, 'Daniel Dennett' AS Name, 1942 AS Born, 1 AS CountryId, 'United
States of America' AS CountryName, 2 AS BookId, 'Elbow Room' AS BookName
UNION ALL SELECT 2 AS Id, 'Sam Harris' AS Name, 1967 AS Born, 1 AS CountryId,  'United States
of America' AS CountryName, 3 AS BookId, 'The Moral Landscape' AS BookName
UNION ALL SELECT 2 AS Id, 'Sam Harris' AS Name, 1967 AS Born, 1 AS CountryId,  'United States
of America' AS CountryName, 4 AS BookId, 'Waking Up: A Guide to Spirituality Without Religion'
AS BookName
UNION ALL SELECT 3 AS Id, 'Richard Dawkins' AS Name, 1941 AS Born, 2 AS CountryId,  'United
Kingdom' AS CountryName, 5 AS BookId, 'The Magic of Reality: How We Know What`s Really True'
AS BookName
UNION ALL SELECT 3 AS Id, 'Richard Dawkins' AS Name, 1941 AS Born, 2 AS CountryId,  'United
Kingdom' AS CountryName, 6 AS BookId, 'An Appetite for Wonder: The Making of a Scientist' AS
BookName";

var remainingHorsemen = new Dictionary<int, Person>();
connection.Query<Person>(sql,
    new[]
    {
        typeof(Person),
        typeof(Country),
        typeof(Book)
    }
    , obj => {

        Person person = obj[0] as Person;
        Country country = obj[1] as Country;
        Book book = obj[2] as Book;

        //person
        Person personEntity;
        //trip
        if (!remainingHorsemen.TryGetValue(person.Id, out personEntity))
        {
            remainingHorsemen.Add(person.Id, personEntity = person);
        }

        //country
        if(personEntity.Residience == null)
        {
            if (country == null)
            {
                country = new Country { CountryName = "" };
            }
            personEntity.Residience = country;
        }

        //books
        if(personEntity.Books == null)
        {
            personEntity.Books = new List<Book>();
        }

        if (book != null)
        {
            if (!personEntity.Books.Any(x => x.BookId == book.BookId))
            {
                personEntity.Books.Add(book);
```

```
            }
        }

        return personEntity;
},
splitOn: "CountryId,BookId");
```

Пользовательские сопоставления

Если имена столбцов запроса не соответствуют вашим классам, вы можете настроить сопоставления для типов. Этот пример демонстрирует сопоставление с использованием `System.Data.Linq.Mapping.ColumnAttribute` а также настраиваемое сопоставление.

> Отображения нужно настраивать только один раз для каждого типа, чтобы они были установлены при запуске приложения или где-то еще, что они только инициализируются один раз.

Предположив тот же запрос, что и пример «один-ко-многим», и классы, реорганизованные для более удобных имен:

```
public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Born { get; set; }
    public Country Residience { get; set; }
    public ICollection<Book> Books { get; set; }
}

public class Country
{
    [System.Data.Linq.Mapping.Column(Name = "CountryId")]
    public int Id { get; set; }

    [System.Data.Linq.Mapping.Column(Name = "CountryName")]
    public string Name { get; set; }
}

public class Book
{
    public int Id { get; set; }

    public string Name { get; set; }
}
```

> Обратите внимание, что `Book` не полагается на `ColumnAttribute` но нам нужно будет поддерживать оператор `if`

Теперь разместите этот код сопоставления где-нибудь в своем приложении, где он выполняется только один раз:

```
Dapper.SqlMapper.SetTypeMap(
    typeof(Country),
```

```
    new CustomPropertyTypeMap(
        typeof(Country),
        (type, columnName) =>
            type.GetProperties().FirstOrDefault(prop =>
                prop.GetCustomAttributes(false)
                    .OfType<System.Data.Linq.Mapping.ColumnAttribute>()
                    .Any(attr => attr.Name == columnName)))
);


var bookMap = new CustomPropertyTypeMap(
    typeof(Book),
    (type, columnName) =>
    {
        if(columnName == "BookId")
        {
            return type.GetProperty("Id");
        }

        if (columnName == "BookName")
        {
            return type.GetProperty("Name");
        }

        throw new InvalidOperationException($"No matching mapping for {columnName}");
    }
);
Dapper.SqlMapper.SetTypeMap(typeof(Book), bookMap);
```

Затем запрос выполняется с использованием любого из предыдущих примеров `Query<>` .

В этом ответе показан более простой способ добавления отображений.

Прочитайте Multimapping онлайн: https://riptutorial.com/ru/dapper/topic/351/multimapping

глава 3: Выполнение команд

Examples

Выполнить команду, которая не возвращает результатов

```
IDBConnection db = /* ... */
var id = /* ... */

db.Execute(@"update dbo.Dogs set Name = 'Beowoof' where Id = @id",
   new { id });
```

Хранимые процедуры

Простое использование

Dapper полностью поддерживает хранимые процедуры:

```
var user = conn.Query<User>("spGetUser", new { Id = 1 },
                            commandType: CommandType.StoredProcedure)
           .SingleOrDefault();
```

Параметры ввода, вывода и возврата

Если вы хотите что-то более интересное, вы можете сделать:

```
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);

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

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

Табличные параметры

Если у вас есть хранимая процедура, которая принимает параметр, оцениваемый по таблице, вам необходимо передать DataTable, который имеет ту же структуру, что и тип таблицы в SQL Server. Ниже приведено определение типа таблицы и процедуры,

использующей ее:

```
CREATE TYPE [dbo].[myUDTT] AS TABLE([i1] [int] NOT NULL);
GO
CREATE PROCEDURE myProc(@data dbo.myUDTT readonly) AS
SELECT i1 FROM @data;
GO
/*
-- optionally grant permissions as needed, depending on the user you execute this with.
-- Especially the GRANT EXECUTE ON TYPE is often overlooked and can cause problems if omitted.
GRANT EXECUTE ON TYPE::[dbo].[myUDTT] TO [user];
GRANT EXECUTE ON dbo.myProc TO [user];
GO
*/
```

Чтобы вызвать эту процедуру из с #, вам необходимо сделать следующее:

```
// Build a DataTable with one int column
DataTable data = new DataTable();
data.Columns.Add("i1", typeof(int));
// Add two rows
data.Rows.Add(1);
data.Rows.Add(2);

var q = conn.Query("myProc", new {data}, commandType: CommandType.StoredProcedure);
```

Прочитайте Выполнение команд онлайн: https://riptutorial.com/ru/dapper/topic/5/выполнение-команд

глава 4: Динамические параметры

Examples

Основное использование

Не всегда возможно аккуратно упаковать все параметры в одном объекте / вызове. Чтобы помочь с более сложными сценариями, dapper позволяет параметру `param` быть экземпляром `IDynamicParameters` . Если вы это сделаете, ваш настраиваемый метод `AddParameters` вызывается в соответствующее время и передает команду для добавления. В большинстве случаев, однако, достаточно использовать ранее существовавшие типы `DynamicParameters` :

```
var p = new DynamicParameters(new { a = 1, b = 2 });
p.Add("c", dbType: DbType.Int32, direction: ParameterDirection.Output);
connection.Execute(@"set @c = @a + @b", p);
int updatedValue = p.Get<int>("@c");
```

Это показывает:

- (необязательная) популяция из существующего объекта
- (необязательно) добавление дополнительных параметров на лету
- передача параметров команде
- получение любого обновленного значения после завершения команды

Обратите внимание, что из-за того, как работают протоколы RDBMS, обычно можно получить только обновленные значения параметров **после того, как** все данные (из операции `Query` или QueryMultiple) были **полностью использованы** (например, на SQL Server обновленные значения параметров находятся в *конце* потока TDS).

Динамические параметры в Dapper

```
connection.Execute(@"some Query with @a,@b,@c", new
{a=somevalueOfa,b=somevalueOfb,c=somevalueOfc});
```

Использование объекта шаблона

Вы можете использовать экземпляр объекта для формирования ваших параметров

```
public class SearchParameters {
  public string SearchString { get; set; }
  public int Page { get; set; }
}

var template= new SearchParameters {
```

```
  SearchString = "Dapper",
  Page = 1
};

var p = new DynamicParameters(template);
```

Вы также можете использовать анонимный объект или `Dictionary`

Прочитайте Динамические параметры онлайн: https://riptutorial.com/ru/dapper/topic/12/динамические-параметры

глава 5: Использование Async

Examples

Вызов хранимой процедуры

```
public async Task<Product> GetProductAsync(string productId)
{
    using (_db)
    {
        return await _db.QueryFirstOrDefaultAsync<Product>("usp_GetProduct", new { id =
productId },
            commandType: CommandType.StoredProcedure);
    }
}
```

Вызов хранимой процедуры и игнорирование результата

```
public async Task SetProductInactiveAsync(int productId)
{
    using (IDbConnection con = new SqlConnection("myConnectionString"))
    {
        await con.ExecuteAsync("SetProductInactive", new { id = productId },
            commandType: CommandType.StoredProcedure);
    }
}
```

Прочитайте Использование Async онлайн: https://riptutorial.com/ru/dapper/topic/1353/использование-async

глава 6: Использование DbGeography и DbGeometry

Examples

Требуется конфигурация

1. установите требуемую сборку `Microsoft.SqlServer.Types` ; они не установлены по умолчанию и доступны от Microsoft здесь как «Типы среды Microsoft® для CLR для Microsoft® SQL Server® 2012» - обратите внимание, что для x86 и x64 существуют отдельные установщики.

2. установить `Dapper.EntityFramework` (или эквивалент сильного имени); это можно сделать с помощью интерфейса IDE «Управление пакетами NuGet ...» или (в консоли диспетчера пакетов):

```
install-package Dapper.EntityFramework
```

3. добавить необходимые переадресации привязки сборки; это потому, что Microsoft отправляет v11 сборок, но Entity Framework запрашивает v10; вы можете добавить следующее в `app.config` или `web.config` в элементе `<configuration>` :

```
<runtime>
  <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
    <dependentAssembly>
      <assemblyIdentity name="Microsoft.SqlServer.Types"
          publicKeyToken="89845dcd8080cc91" />
      <bindingRedirect oldVersion="10.0.0.0" newVersion="11.0.0.0" />
    </dependentAssembly>
  </assemblyBinding>
</runtime>
```

4. скажите «dapper» о доступных доступных обработчиках типов, добавив (где-то в вашем запуске, прежде чем он пытается использовать базу данных):

```
Dapper.EntityFramework.Handlers.Register();
```

Использование геометрии и географии

Когда регистрируются типы обработчиков, все должно работать автоматически, и вы должны иметь возможность использовать эти типы в качестве параметров или возвращаемых значений:

```
string redmond = "POINT (122.1215 47.6740)";
```

```
DbGeography point = DbGeography.PointFromText(redmond,
    DbGeography.DefaultCoordinateSystemId);
DbGeography orig = point.Buffer(20); // create a circle around a point


var fromDb = connection.QuerySingle<DbGeography>(
  "declare @geos table(geo geography); insert @geos(geo) values(@val); select * from @geos",
  new { val = orig });

Console.WriteLine($"Original area: {orig.Area}");
Console.WriteLine($"From DB area: {fromDb.Area}");
```

Прочитайте Использование DbGeography и DbGeometry онлайн:
https://riptutorial.com/ru/dapper/topic/3984/использование-dbgeography-и-dbgeometry

глава 7: Массовые вставки

замечания

У `WriteToServer` и `WriteToServerAsync` есть перегрузки, которые принимают `WriteToServerAsync` IDataReader (см. Примеры), DataTable и DataRow (`DataRow[]`) в качестве источника данных для массовой копии.

Examples

Асинхронная массовая копия

В этом примере используется метод `ToDataReader` описанный здесь. Создание Generic List DataReader для SqlBulkCopy .

Это также можно сделать, используя неасинхронные методы.

```
public class Widget
{
    public int WidgetId {get;set;}
    public string Name {get;set;}
    public int Quantity {get;set;}
}

public async Task AddWidgets(IEnumerable<Widget> widgets)
{
    using(var conn = new SqlConnection("{connection string}")) {
        await conn.OpenAsync();

        using(var bulkCopy = new SqlBulkCopy(conn)) {
            bulkCopy.BulkCopyTimeout = SqlTimeoutSeconds;
            bulkCopy.BatchSize = 500;
            bulkCopy.DestinationTableName = "Widgets";
            bulkCopy.EnableStreaming = true;

            using(var dataReader = widgets.ToDataReader())
            {
                await bulkCopy.WriteToServerAsync(dataReader);
            }
        }
    }
}
```

Массовая копия

В этом примере используется метод `ToDataReader` описанный здесь. Создание Generic List DataReader для SqlBulkCopy .

Это также можно выполнить с помощью методов async.

```
public class Widget
{
    public int WidgetId {get;set;}
    public string Name {get;set;}
    public int Quantity {get;set;}
}

public void AddWidgets(IEnumerable<Widget> widgets)
{
    using(var conn = new SqlConnection("{connection string}")) {
        conn.Open();

        using(var bulkCopy = new SqlBulkCopy(conn)) {
            bulkCopy.BulkCopyTimeout = SqlTimeoutSeconds;
            bulkCopy.BatchSize = 500;
            bulkCopy.DestinationTableName = "Widgets";
            bulkCopy.EnableStreaming = true;

            using(var dataReader = widgets.ToDataReader())
            {
                bulkCopy.WriteToServer(dataReader);
            }
        }
    }
}
```

Прочитайте Массовые вставки онлайн: https://riptutorial.com/ru/dapper/topic/6279/массовые-вставки

глава 8: Несколько результатов

Синтаксис

- public static SqlMapper.GridReader QueryMultiple (это IDbConnection cnn, строка sql, object param = null, транзакция IDbTransaction = null, int? commandTimeout = null, CommandType? commandType = null)
- public static SqlMapper.GridReader QueryMultiple (эта команда IDbConnection cnn, CommandDefinition)

параметры

параметр	подробности
спп	Соединение с базой данных должно быть открыто
SQL	Строка sql для обработки, содержит несколько запросов
пары	Объект для извлечения параметров из
SqlMapper.GridReader	Предоставляет интерфейсы для чтения нескольких наборов результатов из запроса Dapper

Examples

Пример базового множественного результата

Для извлечения нескольких сеток в одном запросе `QueryMultiple` метод `QueryMultiple` . Это позволяет вам *последовательно* получать каждую сетку через последовательные вызовы с возвращенным `GridReader` .

```
var sql = @"select * from Customers where CustomerId = @id
            select * from Orders where CustomerId = @id
            select * from Returns where CustomerId = @id";

using (var multi = connection.QueryMultiple(sql, new {id=selectedId}))
{
   var customer = multi.Read<Customer>().Single();
   var orders = multi.Read<Order>().ToList();
   var returns = multi.Read<Return>().ToList();
}
```

Прочитайте Несколько результатов онлайн: https://riptutorial.com/ru/dapper/topic/8/несколько-результатов

глава 9: Обработка нулей

Examples

null vs DBNull

В ADO.NET правильное обращение с `null` является постоянным источником путаницы. Ключевым моментом в dapper является то, что *вам не нужно* ; он занимается всем этим внутренне.

- Значения параметров , которые являются `null` правильно посланы как `DBNull.Value`
- значения, которые считаются `null` , представлены как `null` или (в случае сопоставления известному типу) просто игнорируются (оставляя их по умолчанию на основе типа)

Он просто работает:

```
string name = null;
int id = 123;
connection.Execute("update Customer set Name=@name where Id=@id",
    new {id, name});
```

Прочитайте Обработка нулей онлайн: https://riptutorial.com/ru/dapper/topic/13/обработка-нулей

глава 10: Обработчики типов

замечания

Обработчики типов позволяют преобразовывать типы баз данных в .Net пользовательские типы.

Examples

Преобразование varchar в IHtmlString

```
public class IHtmlStringTypeHandler : SqlMapper.TypeHandler<IHtmlString>
{
    public override void SetValue(
        IDbDataParameter parameter,
        IHtmlString value)
    {
        parameter.DbType = DbType.String;
        parameter.Value = value?.ToHtmlString();
    }

    public override IHtmlString Parse(object value)
    {
        return MvcHtmlString.Create(value?.ToString());
    }
}
```

Установка TypeHandler

Обработчик вышеуказанного типа можно установить в `SqlMapper` с `AddTypeHandler` метода `AddTypeHandler` .

```
SqlMapper.AddTypeHandler<IHtmlString>(new IHtmlStringTypeHandler());
```

Вывод типа позволяет опустить параметр типового типа:

```
SqlMapper.AddTypeHandler(new IHtmlStringTypeHandler());
```

Существует также перегрузка с двумя аргументами, которая принимает явный аргумент `Type` :

```
SqlMapper.AddTypeHandler(typeof(IHtmlString), new IHtmlStringTypeHandler());
```

Прочитайте Обработчики типов онлайн: https://riptutorial.com/ru/dapper/topic/6/обработчики-типов

глава 11: операции

Синтаксис

- conn.Execute (sql, transaction: tran); // указать параметр по имени
- conn.Execute (sql, parameters, tran);
- conn.Query (sql, transaction: tran);
- conn.Query (sql, parameters, tran);
- await conn.ExecuteAsync (sql, transaction: tran); // Async
- await conn.ExecuteAsync (sql, parameters, tran);
- await conn.QueryAsync (sql, transaction: tran);
- await conn.QueryAsync (sql, parameters, tran);

Examples

Использование транзакции

В этом примере используется SqlConnection, но поддерживается любое IDbConnection.

Также любое IDbTransaction поддерживается из соответствующего IDbConnection.

```
public void UpdateWidgetQuantity(int widgetId, int quantity)
{
    using(var conn = new SqlConnection("{connection string}")) {
        conn.Open();

        // create the transaction
        // You could use `var` instead of `SqlTransaction`
        using(SqlTransaction tran = conn.BeginTransaction()) {
            try
            {
                var sql = "update Widget set Quantity = @quantity where WidgetId = @id";
                var parameters = new { id = widgetId, quantity };

                // pass the transaction along to the Query, Execute, or the related Async
methods.
                conn.Execute(sql, parameters, tran);

                // if it was successful, commit the transaction
                tran.Commit();
            }
            catch(Exception ex)
            {
                // roll the transaction back
                tran.Rollback();

                // handle the error however you need to.
                throw;
            }
        }
    }
```

```
}
```

Ускорить вставки

Обертка группы вставок в транзакции ускорит их в соответствии с этим вопросом / ответом StackOverflow .

Вы можете использовать эту технику или использовать Массовую копию для ускорения выполнения ряда связанных операций.

```
// Widget has WidgetId, Name, and Quantity properties
public void InsertWidgets(IEnumerable<Widget> widgets)
{
    using(var conn = new SqlConnection("{connection string}")) {
        conn.Open();

        using(var tran = conn.BeginTransaction()) {
            try
            {
                var sql = "insert Widget (WidgetId,Name,Quantity) Values(@WidgetId, @Name,
@Quantity)";
                conn.Execute(sql, widgets, tran);
                tran.Commit();
            }
            catch(Exception ex)
            {
                tran.Rollback();
                // handle the error however you need to.
                throw;
            }
        }
    }
}
```

Прочитайте операции онлайн: https://riptutorial.com/ru/dapper/topic/6601/операции

глава 12: Основной запрос

Синтаксис

- public static IEnumerable <T> Query <T> (это IDbConnection cnn, строка sql, object param = null, транзакция SqlTransaction = null, bool buffered = true)
- public static IEnumerable <dynamic> Query (это IDbConnection cnn, строка sql, object param = null, транзакция SqlTransaction = null, bool buffered = true)

параметры

параметр	подробности
спп	Соединение с базой данных, которое должно быть открыто.
SQL	Команда для выполнения.
пары	Объект для извлечения параметров из.
сделка	Транзакция, в которой этот запрос является частью, если таковой имеется.
буферном	Будь или нет буферизация чтения результатов запроса. Это необязательный параметр с по умолчанию значением true. Когда буферизировано верно, результаты буферизуются в `List<T>` а затем возвращаются как `IEnumerable<T>` который безопасен для множественного перечисления. Если буферизация ложна, соединение sql будет открыто до тех пор, пока вы не закончите чтение, что позволит вам обрабатывать одну строку во времени в памяти. Множественные перечисления вызовут дополнительные подключения к базе данных. Хотя буферизованное ложное значение очень эффективно для сокращения использования памяти, если вы поддерживаете только очень мелкие фрагменты возвращенных записей, он имеет значительные накладные расходы по сравнению с желательным материализацией набора результатов. Наконец, если у вас есть многочисленные параллельные небуферизованные соединения sql, вам нужно рассмотреть головоломку пула соединений, вызывающую блокировку запросов до тех пор, пока соединения не станут доступными.

Examples

Запрос для статического типа

Для типов, известных во время компиляции, используйте общий параметр с `Query<T>` .

```
public class Dog
{
    public int? Age { get; set; }
    public Guid Id { get; set; }
    public string Name { get; set; }
    public float? Weight { get; set; }

    public int IgnoredProperty { get { return 1; } }
}

//
IDBConnection db = /* ... */;

var @params = new { age = 3 };
var sql = "SELECT * FROM dbo.Dogs WHERE Age = @age";

IEnumerable<Dog> dogs = db.Query<Dog>(sql, @params);
```

Запрос для динамических типов

Вы также можете запросить динамически, если вы оставите общий тип.

```
IDBConnection db = /* ... */;
IEnumerable<dynamic> result = db.Query("SELECT 1 as A, 2 as B");

var first = result.First();
int a = (int)first.A; // 1
int b = (int)first.B; // 2
```

Запрос с динамическими параметрами

```
var color = "Black";
var age = 4;

var query = "Select * from Cats where Color = :Color and Age > :Age";
var dynamicParameters = new DynamicParameters();
dynamicParameters.Add("Color", color);
dynamicParameters.Add("Age", age);

using (var connection = new SqlConnection(/* Your Connection String Here */))
{
    IEnumerable<dynamic> results = connection.Query(query, dynamicParameters);
}
```

Прочитайте Основной запрос онлайн: https://riptutorial.com/ru/dapper/topic/3/основной-запрос

глава 13: Ссылка на синтаксис параметров

параметры

параметр	подробности
`this cnn`	Основное соединение с базой данных - `this` метод расширения; соединение не должно быть открытым - если оно не открыто, оно автоматически открывается и закрывается.
`<T>` / `Type`	(необязательно) Тип возвращаемого объекта; если неуниверсальный / не - `Type` используется API - , А `dynamic` объект возвращается на строку, имитируя свойство с именем для каждого имени столбца , возвращаемое из запроса (этот `dynamic` объект также реализует `IDicionary<string,object>`).
`sql`	SQL для выполнения
`param`	(необязательно) Параметры для включения.
`transaction`	(необязательно) Операция базы данных для связи с командой
`buffered`	(необязательно) Предпочитаете ли вы предварительно использовать данные в списке (по умолчанию), а также открывать открытый `IEnumerable` над живым читателем
`commandTimeout`	(необязательно) Тайм-аут для использования в команде; если не указано, предполагается `SqlMapper.Settings.CommandTimeout` (если указано)
`commandType`	Тип выполняемой команды; по умолчанию - `CommandText`

замечания

Синтаксис для выражения параметров изменяется между СУБД. Все приведенные выше примеры используют синтаксис SQL Server, то есть `@foo` ; однако `?foo` и `:foo` также должен работать нормально.

Examples

Базовый параметризованный SQL

Dapper позволяет легко следовать наилучшей практике с помощью полностью параметризованного SQL.

Параметры важны, поэтому dapper упрощает правильное использование. Вы просто выражаете свои параметры обычным способом для вашей РСУБД (обычно `@foo` `?foo` или `:foo`) и даете dapper объект, у которого *есть член с именем* `foo` . Самый распространенный способ сделать это - анонимный тип:

```
int id = 123;
string name = "abc";
connection.Execute("insert [KeyLookup](Id, Name) values(@id, @name)",
    new { id, name });
```

И это все. Dapper добавит требуемые параметры, и все должно работать.

Использование вашей объектной модели

Вы также можете использовать существующую объектную модель в качестве параметра:

```
KeyLookup lookup = ... // some existing instance
connection.Execute("insert [KeyLookup](Id, Name) values(@Id, @Name)", lookup);
```

Dapper использует командный текст для определения того, какие члены объекта добавить - обычно он не будет добавлять ненужные вещи, такие как `Description` , `IsActive` , `CreationDate` потому что команда, которую мы выпустили, явно не включает их, - хотя бывают случаи, когда может сделать это, например, если ваша команда содержит:

```
// TODO - removed for now; include the @Description in the insert
```

Он не пытается понять, что выше всего лишь комментарий.

Хранимые процедуры

Параметры хранимых процедур работают точно так же, за исключением того, что dapper не может попытаться определить, что следует / не следует включать - все доступные

рассматриваются как параметр. По этой причине обычно предпочтительны анонимные типы:

```
connection.Execute("KeyLookupInsert", new { id, name },
    commandType: CommandType.StoredProcedure);
```

Ценностная инкрустация

Иногда удобство параметра (с точки зрения обслуживания и выразительности) может быть перевешивается его стоимостью в производительности, чтобы рассматривать его как параметр. Например, когда размер страницы фиксируется настройкой конфигурации. Или значение статуса соответствует значению `enum` . Рассматривать:

```
var orders = connection.Query<Order>(@"
select top (@count) * -- these brackets are an oddity of SQL Server
from Orders
where CustomerId = @customerId
and Status = @open", new { customerId, count = PageSize, open = OrderStatus.Open });
```

Единственным *реальным* параметром здесь является `customerId` - остальные два являются псевдопараметрами, которые фактически не изменятся. Часто РСУБД могут лучше работать, если обнаруживают их как константы. Для этого Dapper имеет специальный синтаксис - `{=name}` вместо `@name` - который применяется *только* к числовым типам. (Это минимизирует любую поверхность атаки из SQL-инъекции). Примером может служить следующее:

```
var orders = connection.Query<Order>(@"
select top {=count} *
from Orders
where CustomerId = @customerId
and Status = {=open}", new { customerId, count = PageSize, open = OrderStatus.Open });
```

Dapper заменяет значения литералами перед выпуском SQL, поэтому СУБД фактически видит что-то вроде:

```
select top 10 *
from Orders
where CustomerId = @customerId
and Status = 3
```

Это особенно полезно, когда RDBMS-системы не просто принимают более правильные решения, а открывают планы запросов, которые предотвращают фактические параметры. Например, если предикат столбца относится к параметру, тогда отфильтрованный индекс с определенными значениями в этих столбцах нельзя использовать. Это связано с тем, что *следующий* запрос может иметь параметр отдельно от одного из указанных значений.

С литеральными значениями оптимизатор запросов может использовать

отфильтрованные индексы, так как он знает, что значение не может измениться в будущих запросах.

Список расширений

Обычный сценарий в запросах базы данных - `IN (...)` где список создается во время выполнения. Для большинства РСУБД не хватает хорошей метафоры для этого - и для этого нет универсального решения для *РСУБД* . Вместо этого dapper обеспечивает нежное автоматическое расширение команд. Все, что требуется, - это заданное значение параметра, которое является `IEnumerable` . Команда, включающая `@foo` , расширена до `(@foo0,@foo1,@foo2,@foo3)` (для последовательности из 4 элементов). Наиболее распространенным применением этого является `IN` :

```
int[] orderIds = ...
var orders = connection.Query<Order>(@"
select *
from Orders
where Id in @orderIds", new { orderIds });
```

Затем автоматически расширяется, чтобы выдать соответствующий SQL для многорядной выборки:

```
select *
from Orders
where Id in (@orderIds0, @orderIds1, @orderIds2, @orderIds3)
```

с параметрами `@orderIds0` т. д. добавляются как значения, взятые из атрибута. Обратите внимание, что тот факт, что он недействителен SQL первоначально, является преднамеренным, чтобы гарантировать, что эта функция не используется ошибочно. Эта функция также корректно работает с подсказкой `OPTIMIZE FOR` / `UNKNOWN` в SQL Server; если вы используете:

```
option (optimize for
    (@orderIds unknown))
```

он правильно расширит это:

```
option (optimize for
    (@orderIds0 unknown, @orderIds1 unknown, @orderIds2 unknown, @orderIds3 unknown))
```

Выполнение операций с несколькими наборами входов

Иногда вы хотите сделать одно и то же несколько раз. Dapper поддерживает это в методе `Execute` если *внешний* параметр (который обычно представляет собой один анонимный тип или экземпляр модели домена) фактически предоставляется как последовательность `IEnumerable` . Например:

```
Order[] orders = ...
// update the totals
connection.Execute("update Orders set Total=@Total where Id=@Id", orders);
```

Здесь dapper просто делает простой цикл для наших данных, в основном так же, как если бы мы это сделали:

```
Order[] orders = ...
// update the totals
foreach(Order order in orders) {
    connection.Execute("update Orders set Total=@Total where Id=@Id", order);
}
```

Это использование становится *особенно* интересным в сочетании с `async` API в соединении, которое явно сконфигурировано для всех «Множественных наборов активных результатов» - в этом использовании dapper автоматически *контактирует* с операциями, поэтому вы не оплачиваете затраты на задержку в строке. Это требует немного более сложного использования,

```
await connection.ExecuteAsync(
    new CommandDefinition(
        "update Orders set Total=@Total where Id=@Id",
         orders, flags: CommandFlags.Pipelined))
```

Обратите внимание, однако, что вы также можете изучить параметры таблицы.

Псевдопозиционные параметры (для поставщиков, которые не поддерживают именованные параметры)

Некоторые поставщики ADO.NET (в первую очередь: OleDB) не поддерживают *именованные* параметры; параметры вместо этого задаются только *положением* , с `?` место-держатель. Dapper не знал бы, какой член использовать для них, поэтому dapper допускает альтернативный синтаксис `?foo?` ; это будет то же самое, что `@foo` или `:foo` в других вариантах SQL, за исключением того, что dapper полностью **заменит** токен параметра `?` перед выполнением запроса.

Это работает в сочетании с другими функциями, такими как расширение списка, поэтому справедливо следующее:

```
string region = "North";
int[] users = ...
var docs = conn.Query<Document>(@"
     select * from Documents
     where Region = ?region?
     and OwnerId in ?users?", new { region, users }).AsList();
```

`.region` `.users` члены `.region` и `.users` , а выданный SQL (например, у 3 пользователей):

```
    select * from Documents
    where Region = ?
    and OwnerId in (?,?,?)
```

Обратите внимание, однако, что dapper **не** позволяет использовать один и тот же параметр несколько раз при использовании этой функции; это необходимо для того, чтобы несколько раз добавить одно и то же значение параметра (которое может быть большим). Если вам нужно ссылаться на одно и то же значение несколько раз, рассмотрите объявление переменной, например:

```
declare @id int = ?id?; // now we can use @id multiple times in the SQL
```

Если переменные недоступны, вы можете использовать двойные имена членов в параметрах - это также сделает очевидным, что значение отправляется несколько раз:

```
int id = 42;
connection.Execute("... where ParentId = $id0$ ... SomethingElse = $id1$ ...",
      new { id0 = id, id1 = id });
```

Прочитайте Ссылка на синтаксис параметров онлайн:
https://riptutorial.com/ru/dapper/topic/10/ссылка-на-синтаксис-параметров

глава 14: Таблицы темпов

Examples

Таблица Temp, которая существует, пока соединение остается открытым

Когда временная таблица создается сама по себе, она останется, пока соединение открыто.

```
// Widget has WidgetId, Name, and Quantity properties
public async Task PurchaseWidgets(IEnumerable<Widget> widgets)
{
    using(var conn = new SqlConnection("{connection string}")) {
        await conn.OpenAsync();

        await conn.ExecuteAsync("CREATE TABLE #tmpWidget(WidgetId int, Quantity int)");

        // populate the temp table
        using(var bulkCopy = new SqlBulkCopy(conn)) {
            bulkCopy.BulkCopyTimeout = SqlTimeoutSeconds;
            bulkCopy.BatchSize = 500;
            bulkCopy.DestinationTableName = "#tmpWidget";
            bulkCopy.EnableStreaming = true;

            using(var dataReader = widgets.ToDataReader())
            {
                await bulkCopy.WriteToServerAsync(dataReader);
            }
        }

        await conn.ExecuteAsync(@"
            update w
            set Quantity = w.Quantity - tw.Quantity
            from Widgets w
                join #tmpWidget tw on w.WidgetId = tw.WidgetId");
    }
}
```

Как работать с временными таблицами

Пункт о временных таблицах состоит в том, что они ограничены областью действия соединения. Dapper автоматически откроет и закроет соединение, если оно еще не открыто. Это означает, что любая временная таблица будет потеряна непосредственно после ее создания, если соединение, переданное Dapper, не было открыто.

Это не будет работать:

```
private async Task<IEnumerable<int>> SelectWidgetsError()
{
  using (var conn = new SqlConnection(connectionString))
  {
```

```
    await conn.ExecuteAsync(@"CREATE TABLE #tmpWidget(widgetId int);");

    // this will throw an error because the #tmpWidget table no longer exists
    await conn.ExecuteAsync(@"insert into #tmpWidget(WidgetId) VALUES (1);");

    return await conn.QueryAsync<int>(@"SELECT * FROM #tmpWidget;");
  }
}
```

С другой стороны, эти две версии будут работать:

```
private async Task<IEnumerable<int>> SelectWidgets()
{
  using (var conn = new SqlConnection(connectionString))
  {
    // Here, everything is done in one statement, therefore the temp table
    // always stays within the scope of the connection
    return await conn.QueryAsync<int>(
      @"CREATE TABLE #tmpWidget(widgetId int);
        insert into #tmpWidget(WidgetId) VALUES (1);
        SELECT * FROM #tmpWidget;");
  }
}

private async Task<IEnumerable<int>> SelectWidgetsII()
{
  using (var conn = new SqlConnection(connectionString))
  {
    // Here, everything is done in separate statements. To not loose the
    // connection scope, we have to explicitly open it
    await conn.OpenAsync();

    await conn.ExecuteAsync(@"CREATE TABLE #tmpWidget(widgetId int);");
    await conn.ExecuteAsync(@"insert into #tmpWidget(WidgetId) VALUES (1);");
    return await conn.QueryAsync<int>(@"SELECT * FROM #tmpWidget;");
  }
}
```

Прочитайте Таблицы темпов онлайн: https://riptutorial.com/ru/dapper/topic/6594/таблицы-темпов

кредиты

S. No	Главы	Contributors
1	Начало работы с Dapper.NET	Adam Lear, balpha, Community, Eliza, Greg Bray, Jarrod Dixon, Kevin Montrose, Matt McCabe, Nick, Rob, Shog9
2	Multimapping	Devon Burriss
3	Выполнение команд	Adam Lear, Jarrod Dixon, Sklivvz, takrl
4	Динамические параметры	Marc Gravell, Matt McCabe, Meer
5	Использование Async	Dean Ward, Matt McCabe, Nick, Woodchipper
6	Использование DbGeography и DbGeometry	Marc Gravell
7	Массовые вставки	jhamm
8	Несколько результатов	Marc Gravell, Yaakov Ellis
9	Обработка нулей	Marc Gravell
10	Обработчики типов	Benjamin Hodgson, Community, Marc Gravell
11	операции	jhamm
12	Основной запрос	Adam Lear, Chris Marisic, Cigano Morrison Mendez, Community, cubrr, Jarrod Dixon, jrummell, Kevin Montrose, Matt McCabe
13	Ссылка на синтаксис параметров	4444, Marc Gravell, Nick Craver
14	Таблицы темпов	jhamm, Rob, takrl