EBook Gratis

APRENDIZAJE Dapper.NET

Free unaffiliated eBook created from
Stack Overflow contributors.

#dapper

Tabla de contenido

Acerca de

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

Capítulo 1: Empezando con Dapper.NET

Observaciones

¿Qué es Dapper?

Dapper es un micro-ORM para .Net que extiende su `IDbConnection` , simplificando la configuración de consultas, la ejecución y la lectura de resultados.

¿Como lo consigo?

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

Tareas comunes

- Consulta Básica
- Comandos de ejecución

Versiones

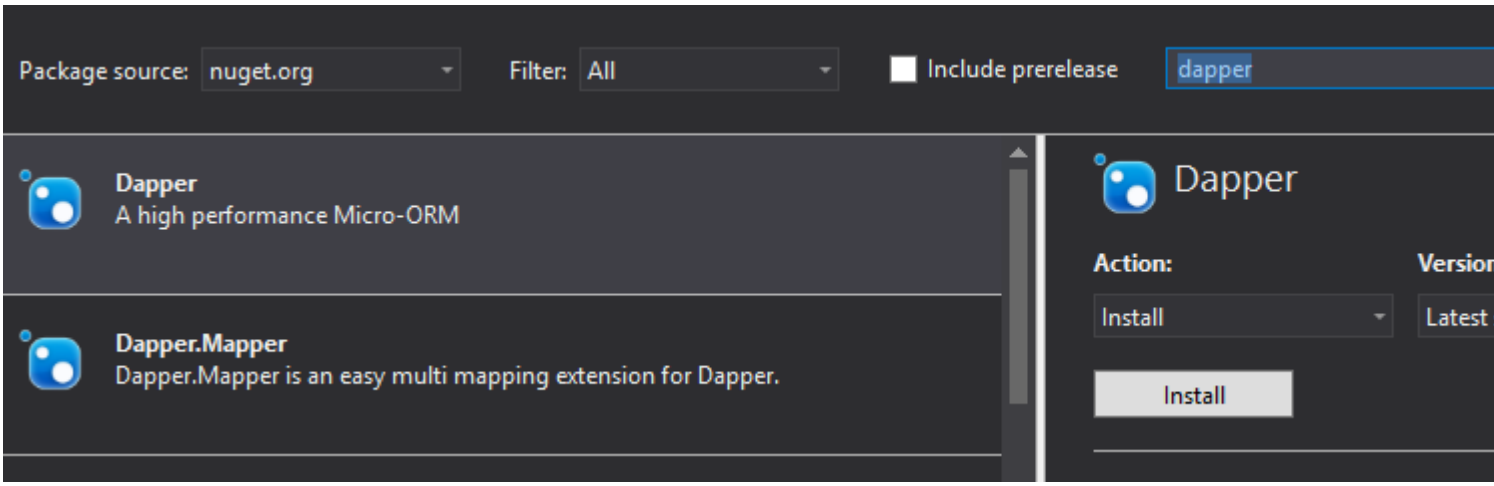
Versión	Notas	Fecha de lanzamiento
1.50.0	core-clr / asp.net 5.0 compilación contra RTM	2016-06-29
1.42.0		2015-05-06
1.40.0		2015-04-03
1.30.0		2014-08-14
1.20.0		2014-05-08
1.10.0		2012-06-27
1.0.0		2011-04-14

Examples

Instala Dapper desde Nuget

O bien buscar en la GUI de Visual Studio:

Herramientas> NuGet Package Manager> Administrar paquetes para la solución ... (Visual Studio 2015)



O ejecute este comando en una instancia de Nuget Power Shell para instalar la última versión estable

```
Install-Package Dapper
```

O para una versión específica.

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

Usando Dapper en 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);
        }
    }
}
```

Envolver la conexión en un bloque de `Using` cerrará la conexión

Usando Dapper en LINQPad

LINQPad es ideal para probar consultas de bases de datos e incluye la integración de NuGet . Para usar Dapper en LINQPad, presione **F4** para abrir las Propiedades de la consulta y luego

seleccione **Agregar NuGet** . Busque **dapper dot net** y seleccione **Agregar a la consulta** . También querrá hacer clic en **Agregar espacios de nombres** y resaltar Dapper para incluir los Métodos de Extensión en su consulta LINQPad.

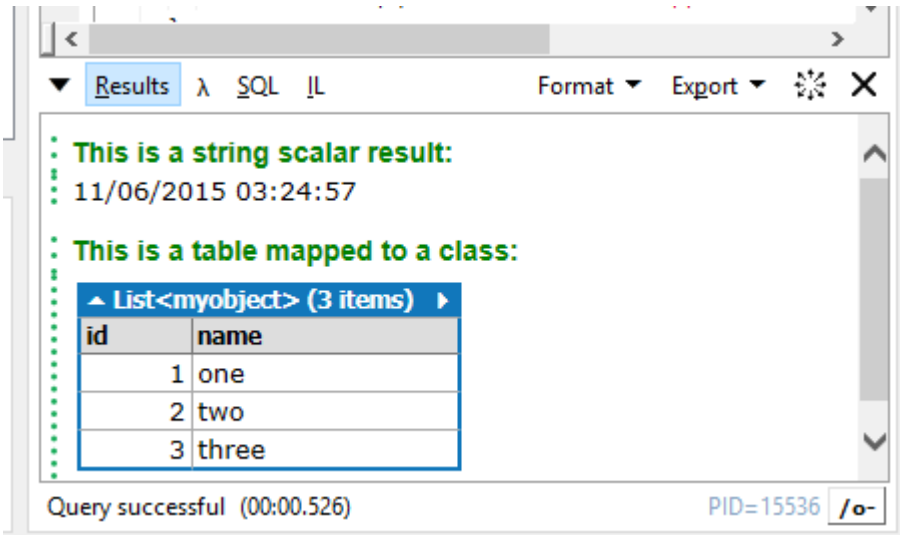
Una vez que Dapper esté habilitado, puede cambiar el menú desplegable de Idioma al **Programa C #** , asignar los resultados de la consulta a las clases C # y usar el método .Dump () para inspeccionar los resultados:

```
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; }
}
```

Los resultados al ejecutar el programa se verían así:



Lea Empezando con Dapper.NET en línea: https://riptutorial.com/es/dapper/topic/2/empezando-con-dapper-net

Capítulo 2: Actas

Sintaxis

- conn.Execute (sql, transaction: tran); // especifique el parámetro por nombre
- conn.Execute (sql, parameters, tran);
- conn.Query (sql, transaction: tran);
- conn.Query (sql, parameters, tran);
- await conn.ExecuteAsync (sql, transaction: tran); // asíncrono
- await conn.ExecuteAsync (sql, parameters, tran);
- await conn.QueryAsync (sql, transaction: tran);
- await conn.QueryAsync (sql, parameters, tran);

Examples

Usando una Transacción

Este ejemplo utiliza SqlConnection, pero se admite cualquier IDbConnection.

También se admite cualquier IDbTransaction desde la IDbConnection relacionada.

```
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;
            }
        }
    }
}
```

Acelerar las inserciones

Ajustar un grupo de inserciones en una transacción los acelerará de acuerdo con esta Pregunta / Respuesta de StackOverflow .

Puede usar esta técnica, o puede usar Copia masiva para acelerar una serie de operaciones relacionadas para realizar.

```
// 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;
            }
        }
    }
}
```

Lea Actas en línea: https://riptutorial.com/es/dapper/topic/6601/actas

Capítulo 3: Comandos de ejecución

Examples

Ejecuta un comando que no devuelve resultados

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

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

Procedimientos almacenados

Uso simple

Dapper soporta completamente los procs almacenados:

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

Parámetros de entrada, salida y retorno

Si quieres algo más elegante, puedes hacer:

```
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");
```

Parámetros de tabla de valores

Si tiene un procedimiento almacenado que acepta un parámetro de valor de tabla, debe pasar un DataTable que tiene la misma estructura que el tipo de tabla en SQL Server. Aquí hay una definición para un tipo de tabla y un procedimiento que lo utiliza:

```
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
*/
```

Para llamar a ese procedimiento desde c #, debe hacer lo siguiente:

```
// 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);
```

Lea Comandos de ejecución en línea: https://riptutorial.com/es/dapper/topic/5/comandos-de-ejecucion

Capítulo 4: Consulta Básica

Sintaxis

- Consulta <T> IEnumerable estática pública (esta IDbConnection cnn, string sql, object param = null, SqlTransaction transaction = null, bool buffered = true)
- Consulta estática pública IEnumerable <dynamic> (este IDbConnection cnn, string sql, object param = null, SqlTransaction transaction = null, bool buffered = true)

Parámetros

Parámetro	Detalles
cnn	Su conexión de base de datos, que ya debe estar abierta.
sql	Comando para ejecutar.
param	Objeto para extraer parámetros de.
transacción	Transacción de la que forma parte esta consulta, si la hubiera.
amortiguado	Ya sea para almacenar o no los resultados de la consulta. Este es un parámetro opcional con el valor predeterminado verdadero. Cuando el búfer es verdadero, los resultados se guardan en una `List<T>` y luego se devuelven como `IEnumerable<T>` que es seguro para la enumeración múltiple. Cuando el búfer es falso, la conexión SQL se mantiene abierta hasta que finalice la lectura, lo que le permite procesar una sola fila en el momento en la memoria. Las enumeraciones múltiples generarán conexiones adicionales a la base de datos. Si bien el búfer falso es altamente eficiente para reducir el uso de memoria, si solo mantiene fragmentos muy pequeños de los registros devueltos, tiene una sobrecarga de rendimiento considerable en comparación con materializar con impaciencia el conjunto de resultados. Por último, si tiene numerosas conexiones de SQL no búfer concurrentes, debe considerar que la inanición de la agrupación de conexiones provoca que las solicitudes se bloqueen hasta que las conexiones estén disponibles.

Examples

Buscando un tipo estático

Para los tipos conocidos en tiempo de compilación, use un parámetro genérico con `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);
```

Consultando por tipos dinámicos

También puede consultar dinámicamente si deja de lado el tipo genérico.

```
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
```

Consulta con parámetros dinámicos

```
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);
}
```

Lea Consulta Básica en línea: https://riptutorial.com/es/dapper/topic/3/consulta-basica

Capítulo 5: Inserciones a granel

Observaciones

`WriteToServer` y `WriteToServerAsync` tienen sobrecargas que aceptan las matrices IDataReader (como se ve en los ejemplos), DataTable y DataRow (`DataRow[]`) como la fuente de los datos para la Copia masiva.

Examples

Copia a granel asincrónica

Este ejemplo utiliza un método `ToDataReader` descrito aquí Creando un DataReader de lista genérico para SqlBulkCopy .

Esto también se puede hacer utilizando métodos no asíncronos.

```
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);
            }
        }
    }
}
```

Copia a granel

Este ejemplo utiliza un método `ToDataReader` descrito aquí Creando un DataReader de lista genérico para SqlBulkCopy .

Esto también se puede hacer utilizando métodos asíncronos.

```
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);
            }
        }
    }
}
```

Lea Inserciones a granel en línea: https://riptutorial.com/es/dapper/topic/6279/inserciones-a-granel

Capítulo 6: Manejo de Nulos

Examples

null vs DBNull

En ADO.NET, el manejo correcto de `null` es una fuente constante de confusión. El punto clave en Dapper es que *no tienes que hacerlo* ; Se trata de todo internamente.

- los valores de parámetro que son `null` se envían correctamente como `DBNull.Value`
- los valores de lectura que son `null` se presentan como `null` , o (en el caso de la asignación a un tipo conocido) simplemente se ignoran (dejando su valor predeterminado basado en el tipo)

Simplemente funciona:

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

Lea Manejo de Nulos en línea: https://riptutorial.com/es/dapper/topic/13/manejo-de-nulos

Capítulo 7: Multimapa

Sintaxis

- `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)`

Parámetros

Parámetro	Detalles
cnn	Su conexión de base de datos, que ya debe estar abierta.
sql	Comando para ejecutar.
tipos	Conjunto de tipos en el conjunto de registros.
mapa	`Func<>` que maneja la construcción del resultado de retorno.
param	Objeto para extraer parámetros de.
transacción	Transacción de la que forma parte esta consulta, si la hubiera.
amortiguado	Ya sea para almacenar o no los resultados de la consulta. Este es un parámetro opcional con el valor predeterminado verdadero. Cuando el búfer es verdadero, los resultados se guardan en una `List<T>` y luego se devuelven como `IEnumerable<T>` que es seguro para la enumeración múltiple. Cuando el búfer es falso, la conexión SQL se mantiene abierta hasta que finalice la lectura, lo que le permite procesar una sola fila en el momento en la memoria. Las enumeraciones múltiples generarán conexiones adicionales a la base de datos. Si bien el búfer falso es altamente eficiente para reducir el uso de memoria, si solo mantiene fragmentos muy pequeños de los registros devueltos, tiene una sobrecarga de rendimiento considerable en comparación con materializar con impaciencia el conjunto de resultados. Por último, si tiene numerosas conexiones de SQL no búfer concurrentes, debe considerar que la inanición de la agrupación de conexiones provoca que las solicitudes se bloqueen hasta que las conexiones estén disponibles.

Parámetro	Detalles
dividido en	El campo del que deberíamos dividir y leer el segundo objeto (por defecto: id). Esta puede ser una lista delimitada por comas cuando más de 1 tipo está contenido en un registro.
commandTimeout	Número de segundos antes del tiempo de espera de ejecución del comando.
commandType	¿Es un proceso almacenado o un lote?

Examples

Mapeo simple multi-mesa

Digamos que tenemos una consulta de los jinetes restantes que necesitan poblar una clase de Persona.

Nombre	Nacido	Residencia
Daniel Dennett	1942	Estados Unidos de America
Sam Harris	1967	Estados Unidos de America
Richard Dawkins	1941	Reino Unido

```
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; }
}
```

Podemos rellenar la clase de persona y la propiedad Residencia con una instancia de País usando una `Query<>` sobrecarga `Query<>` que toma una función `Func<>` que se puede usar para componer la instancia devuelta. El `Func<>` puede tomar hasta 7 tipos de entrada con el argumento genérico final siempre siendo el tipo de retorno.

```
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");
```

Tenga en cuenta el uso del `splitOn: "Residence"` que es la primera columna del siguiente tipo de clase que se completará (en este caso, `Country`). Dapper buscará automáticamente una columna llamada *Id* para dividir, pero si no encuentra una y `splitOn` no se proporciona, se `splitOn` una `splitOn System.ArgumentException` con un mensaje útil. Entonces, aunque es opcional, normalmente tendrá que proporcionar un valor de `splitOn` .

Mapeo uno a muchos

Veamos un ejemplo más complejo que contiene una relación de uno a varios. Nuestra consulta ahora contendrá varias filas que contienen datos duplicados y tendremos que manejar esto. Hacemos esto con una búsqueda en un cierre.

La consulta cambia ligeramente al igual que las clases de ejemplo.

Carné de identidad	Nombre	Nacido	PaísId	Nombre del país	BookId	Nombre del libro
1	Daniel Dennett	1942	1	Estados Unidos de America	1	Tormentas de ideas
1	Daniel Dennett	1942	1	Estados Unidos de America	2	Cuarto de la esquina
2	Sam Harris	1967	1	Estados Unidos de America	3	El paisaje moral
2	Sam Harris	1967	1	Estados Unidos de America	4	Despertar: una guía a la espiritualidad sin religión
3	Richard Dawkins	1941	2	Reino Unido	5	La magia de la realidad: Cómo sabemos lo que es realmente cierto
3	Richard Dawkins	1941	2	Reino Unido	6	Un apetito por la maravilla: la creación de un científico

```
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; }
}
```

Los `remainingHorsemen` diccionario se rellenarán con instancias completamente materializadas de los objetos personales. Para cada fila del resultado de la consulta, se pasan los valores asignados de las instancias de los tipos definidos en los argumentos lambda y depende de usted cómo manejar esto.

```
            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");
```

Observe que el argumento `splitOn` es una lista delimitada por comas de las primeras columnas del siguiente tipo.

Mapeo de más de 7 tipos

A veces, el número de tipos que está asignando excede los 7 proporcionados por la Función <> que hace la construcción.

En lugar de utilizar la `Query<>` con las entradas de argumento de tipo genérico, proporcionaremos los tipos para asignar como una matriz, seguido por la función de asignación. Aparte del ajuste manual inicial y la conversión de los valores, el resto de la función no cambia.

```
            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");
```

Asignaciones personalizadas

Si los nombres de las columnas de consulta no coinciden con sus clases, puede configurar asignaciones para tipos. Este ejemplo muestra la asignación utilizando `System.Data.Linq.Mapping.ColumnAttribute` , así como una asignación personalizada.

> Las asignaciones solo se deben configurar una vez por tipo, así que configúrelas en el inicio de la aplicación o en algún otro lugar para que solo se inicialicen una vez.

Suponiendo de nuevo la misma consulta que el ejemplo Uno a muchos y las clases rediseñadas para obtener mejores nombres como:

```
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; }
}
```

> Observe cómo `Book` no se basa en `ColumnAttribute` pero tendríamos que mantener la instrucción `if`

Ahora coloque este código de mapeo en algún lugar de su aplicación donde solo se ejecute una vez:

```
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);
```

Luego, la consulta se ejecuta utilizando cualquiera de los ejemplos anteriores de `Query<>` .

Una forma más sencilla de agregar las asignaciones se muestra en esta respuesta .

Lea Multimapa en línea: https://riptutorial.com/es/dapper/topic/351/multimapa

Capítulo 8: Parámetros dinámicos

Examples

Uso básico

No siempre es posible empaquetar cuidadosamente todos los parámetros en un solo objeto / llamada. Para ayudar con escenarios más complicados, dapper permite que el parámetro `param` sea una instancia de `IDynamicParameters` . Si hace esto, se llama al método `AddParameters` personalizado en el momento adecuado y se le entrega el comando para que lo agregue. En la mayoría de los casos, sin embargo, es suficiente usar el tipo `DynamicParameters` preexistente:

```
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");
```

Esta espectáculos:

- (opcional) población de un objeto existente
- (opcional) añadiendo parámetros adicionales sobre la marcha
- Pasando los parámetros al comando.
- recuperando cualquier valor actualizado después de que el comando haya finalizado

Tenga en cuenta que, debido a la forma en que funcionan los protocolos RDBMS, generalmente solo es confiable obtener valores de parámetros actualizados **después de que** todos los datos (de una operación `Query` o QueryMultiple`) se hayan consumido por **completo** (por ejemplo, en SQL Server, los valores de parámetros actualizados están al *final)* de la secuencia de TDS).

Parámetros dinámicos en Dapper

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

Usando un objeto de plantilla

Puedes usar una instancia de un objeto para formar tus parámetros

```
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);
```

También puedes usar un objeto anónimo o un `Dictionary`

Lea Parámetros dinámicos en línea: https://riptutorial.com/es/dapper/topic/12/parametros-dinamicos

Capítulo 9: Referencia de sintaxis de parámetros

Parámetros

Parámetro	Detalles
`this cnn`	La conexión de base de datos subyacente - `this` denota un método de extensión; la conexión no necesita estar abierta; si no está abierta, se abre y se cierra automáticamente.
`<T>` / `Type`	(opcional) El tipo de objeto a devolver; si se usa la API no genérica / no `Type` , se devuelve un objeto `dynamic` por fila, simulando una propiedad nombrada por el nombre de columna devuelto por la consulta (este objeto `dynamic` también implementa `IDicionary<string,object>`).
`sql`	El SQL para ejecutar
`param`	(Opcional) Los parámetros a incluir.
`transaction`	(Opcional) La transacción de la base de datos para asociar con el comando.
`buffered`	(opcional) si se debe consumir previamente los datos en una lista (el valor predeterminado), en lugar de exponer un `IEnumerable` abierto sobre el lector en vivo
`commandTimeout`	(opcional) El tiempo de espera para usar en el comando; si no se especifica, se asume `SqlMapper.Settings.CommandTimeout` (si se especifica)
`commandType`	El tipo de comando que se está ejecutando; por defecto a `CommandText`

Observaciones

La sintaxis para expresar parámetros varía entre RDBMS. Todos los ejemplos anteriores utilizan la sintaxis de SQL Server, es decir, `@foo` ; sin embargo `?foo` y `:foo` también deberían funcionar bien.

Examples

SQL básico parametrizado

Dapper facilita el seguimiento de las mejores prácticas mediante SQL totalmente parametrizado.

Los parámetros son importantes, por lo que Dapper hace que sea fácil hacerlo bien. Simplemente expresa tus parámetros de la forma normal para tu RDBMS (generalmente `@foo` `?foo` o `:foo`) y le das a Dapper un objeto que *tiene un miembro llamado* `foo` . La forma más común de hacer esto es con un tipo anónimo:

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

Y eso es. Dapper agregará los parámetros requeridos y todo debería funcionar.

Usando tu modelo de objeto

También puede usar su modelo de objeto existente como parámetro:

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

Dapper usa el texto de comando para determinar qué miembros del objeto agregar: generalmente no agregará elementos innecesarios como `Description` , `IsActive` , `CreationDate` porque el comando que hemos emitido claramente no los involucra, aunque existen casos en los que podría hacer eso, por ejemplo, si su comando contiene:

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

No intenta descubrir que lo anterior es solo un comentario.

Procedimientos almacenados

Los parámetros de los procedimientos almacenados funcionan exactamente igual, excepto que Dapper no puede tratar de determinar qué debe / no debe incluirse, todo lo disponible se trata como un parámetro. Por esa razón, usualmente se prefieren los tipos anónimos:

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

Valor en línea

A veces, la conveniencia de un parámetro (en términos de mantenimiento y expresividad) puede ser superada por su costo en el rendimiento para tratarlo como un parámetro. Por ejemplo, cuando el tamaño de la página está fijado por una configuración. O un valor de estado coincide con un valor `enum` . Considerar:

```
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 });
```

El único parámetro *real* aquí es `customerId` , los otros dos son pseudo parámetros que no cambiarán realmente. A menudo, el RDBMS puede hacer un mejor trabajo si detecta estas constantes. Dapper tiene una sintaxis especial para esto - `{=name}` lugar de `@name` - que *solo se* aplica a los tipos numéricos. (Esto minimiza cualquier superficie de ataque de la inyección de SQL). Un ejemplo es el siguiente:

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

Dapper reemplaza los valores con literales antes de emitir el SQL, por lo que RDBMS realmente ve algo como:

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

Esto es particularmente útil cuando se permite que los sistemas RDBMS no solo tomen mejores decisiones, sino que también abran planes de consulta que impiden los parámetros reales. Por ejemplo, si un predicado de columna es contra un parámetro, entonces no se puede usar un índice filtrado con valores específicos en esas columnas. Esto se debe a que la *siguiente* consulta puede tener un parámetro aparte de uno de esos valores especificados.

Con valores literales, el optimizador de consultas puede hacer uso de los índices filtrados, ya que sabe que el valor no puede cambiar en futuras consultas.

Lista de expansiones

Un escenario común en las consultas de base de datos es `IN (...)` donde la lista aquí se genera en tiempo de ejecución. La mayoría de los RDBMS carecen de una buena metáfora para esto, y no existe una solución universal *de RDBMS cruzados* para esto. En su lugar, Dapper proporciona una suave expansión automática de comandos Todo lo que se requiere es un valor de parámetro suministrado que sea `IEnumerable` . Un comando relacionado con `@foo` se expande a

`(@foo0,@foo1,@foo2,@foo3)` (para una secuencia de 4 elementos). El uso más común de esto sería `IN` :

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

Esto luego se expande automáticamente para emitir el SQL apropiado para la recuperación de varias filas:

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

con los parámetros `@orderIds0` etc. que se agregan como valores tomados del arrray. Tenga en cuenta que el hecho de que originalmente no sea un SQL válido es intencional, para garantizar que esta característica no se use por error. Esta característica también funciona correctamente con la sugerencia de consulta `OPTIMIZE FOR` / `UNKNOWN` en SQL Server; si utiliza:

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

expandirá esto correctamente a:

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

Realizar operaciones contra múltiples conjuntos de entrada

A veces, quieres hacer lo mismo varias veces. Dapper admite esto en el método de `Execute` si el parámetro *más externo* (que generalmente es un solo tipo anónimo o una instancia de modelo de dominio) en realidad se proporciona como una secuencia `IEnumerable` . Por ejemplo:

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

Aquí, Dapper solo está haciendo un simple bucle en nuestros datos, esencialmente como si hubiéramos hecho:

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

Este uso se vuelve *particularmente* interesante cuando se combina con la API `async` en una conexión que está configurada explícitamente para todos los "Conjuntos de resultados activos

múltiples". En este uso, Dapper *canalizará* las operaciones automáticamente, por lo que no está pagando el costo de latencia por fila. Esto requiere un uso un poco más complicado,

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

Sin embargo, tenga en cuenta que es posible que también desee investigar los parámetros con valores de tabla.

Parámetros pseudo-posicionales (para proveedores que no admiten parámetros nombrados)

Algunos proveedores de ADO.NET (en particular: OleDB) no admiten parámetros con *nombre* ; los parámetros se especifican en su lugar solo por *posición* , con el `?` titular de lugar. Dapper no sabría qué miembro usar para estos, por lo que Dapper permite una sintaxis alternativa, `?foo?` ; esto sería lo mismo que `@foo` o `:foo` en otras variantes de SQL, excepto que Dapper **reemplazará** el token de parámetro completamente con `?` Antes de ejecutar la consulta.

Esto funciona en combinación con otras características como la expansión de lista, por lo que lo siguiente es válido:

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

Los miembros `.region` y `.users` se utilizan en consecuencia, y el SQL emitido es (por ejemplo, con 3 usuarios):

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

Sin embargo, tenga en **cuenta** que Dapper **no** permite que se use el mismo parámetro varias veces al usar esta función; esto es para evitar tener que agregar el mismo valor de parámetro (que podría ser grande) varias veces. Si necesita referirse al mismo valor varias veces, considere declarar una variable, por ejemplo:

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

Si las variables no están disponibles, puede usar nombres de miembros duplicados en los parámetros. Esto también hará que sea obvio que el valor se envía varias veces:

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

```
        new { id0 = id, id1 = id });
```

Lea Referencia de sintaxis de parámetros en línea:
https://riptutorial.com/es/dapper/topic/10/referencia-de-sintaxis-de-parametros

Capítulo 10: Resultados Múltiples

Sintaxis

- public static SqlMapper.GridReader QueryMultiple (este IDbConnection cnn, string sql, object param = null, IDbTransaction transaction = null, int? commandTimeout = null, CommandType? commandType = null)
- Public static SqlMapper.GridReader QueryMultiple (este IDbConnection cnn, comando CommandDefinition)

Parámetros

Parámetro	Detalles
cnn	Tu conexión de base de datos, ya debe estar abierta.
sql	La cadena sql a procesar, contiene múltiples consultas.
param	Objeto para extraer parámetros de
SqlMapper.GridReader	Proporciona interfaces para leer múltiples conjuntos de resultados de una consulta Dapper

Examples

Ejemplo de base de resultados múltiples

Para obtener varias cuadrículas en una sola consulta, se `QueryMultiple` método `QueryMultiple` . Esto le permite recuperar cada cuadrícula *secuencialmente a* través de llamadas sucesivas contra el `GridReader` devuelto.

```
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();
}
```

Lea Resultados Múltiples en línea: https://riptutorial.com/es/dapper/topic/8/resultados-multiples

Capítulo 11: Tablas de temperatura

Examples

Tabla temporal que existe mientras la conexión permanece abierta

Cuando la tabla temporal se crea por sí misma, permanecerá mientras la conexión esté abierta.

```
// 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");
    }
}
```

Cómo trabajar con tablas temporales.

El punto sobre las tablas temporales es que están limitadas al alcance de la conexión. Dapper abrirá y cerrará automáticamente una conexión si aún no está abierta. Eso significa que cualquier tabla temporal se perderá directamente después de crearla, si no se ha abierto la conexión pasada a Dapper.

Esto no funcionará:

```
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;");
  }
}
```

Por otro lado, estas dos versiones funcionarán:

```
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;");
  }
}
```

Lea Tablas de temperatura en línea: https://riptutorial.com/es/dapper/topic/6594/tablas-de-temperatura

Capítulo 12: Tipo Handlers

Observaciones

Los controladores de tipos permiten que los tipos de base de datos se conviertan a tipos personalizados .Net.

Examples

Convertir varchar a 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());
    }
}
```

Instalar un TypeHandler

El controlador de tipo anterior se puede instalar en `SqlMapper` utilizando el método `AddTypeHandler` .

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

La inferencia de tipos le permite omitir el parámetro de tipo genérico:

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

También hay una sobrecarga de dos argumentos que toma un argumento `Type` explícito:

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

Lea Tipo Handlers en línea: https://riptutorial.com/es/dapper/topic/6/tipo-handlers

Capítulo 13: Usando DbGeography y DbGeometry

Examples

Configuración requerida

1. instale el ensamblado requerido de `Microsoft.SqlServer.Types` ; no se instalan de forma predeterminada, y están disponibles en Microsoft aquí como "Tipos de CLR de Microsoft® System para Microsoft® SQL Server® 2012"; tenga en cuenta que hay instaladores independientes para x86 y x64.

2. instale `Dapper.EntityFramework` (o el equivalente con nombre `Dapper.EntityFramework`); esto podría hacerse a través de la interfaz de usuario "Administrar paquetes NuGet ..." del IDE, o (en la Consola del Administrador de paquetes):

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

3. agregue las redirecciones obligatorias de ensamblaje requeridas; esto se debe a que Microsoft entrega la v11 de los ensamblajes, pero Entity Framework solicita la v10; puede agregar lo siguiente a `app.config` o `web.config` bajo el elemento `<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. diga a "dapper" sobre los nuevos controladores de tipos disponibles, agregando (en algún lugar de su inicio, antes de que intente usar la base de datos):

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

Usando geometría y geografía

Una vez que se hayan registrado los controladores de tipo, todo debería funcionar automáticamente y debería poder usar estos tipos como parámetros o valores de retorno:

```
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}");
```

Lea Usando DbGeography y DbGeometry en línea:
https://riptutorial.com/es/dapper/topic/3984/usando-dbgeography-y-dbgeometry

Capítulo 14: Utilizando async

Examples

Llamando a un procedimiento almacenado

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

Llamando a un procedimiento almacenado e ignorando el resultado.

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

Lea Utilizando async en línea: https://riptutorial.com/es/dapper/topic/1353/utilizando-async

Creditos

S. No	Capítulos	Contributors
1	Empezando con Dapper.NET	Adam Lear, balpha, Community, Eliza, Greg Bray, Jarrod Dixon, Kevin Montrose, Matt McCabe, Nick, Rob, Shog9
2	Actas	jhamm
3	Comandos de ejecución	Adam Lear, Jarrod Dixon, Sklivvz, takrl
4	Consulta Básica	Adam Lear, Chris Marisic, Cigano Morrison Mendez, Community, cubrr, Jarrod Dixon, jrummell, Kevin Montrose, Matt McCabe
5	Inserciones a granel	jhamm
6	Manejo de Nulos	Marc Gravell
7	Multimapa	Devon Burriss
8	Parámetros dinámicos	Marc Gravell, Matt McCabe, Meer
9	Referencia de sintaxis de parámetros	4444, Marc Gravell, Nick Craver
10	Resultados Múltiples	Marc Gravell, Yaakov Ellis
11	Tablas de temperatura	jhamm, Rob, takrl
12	Tipo Handlers	Benjamin Hodgson, Community, Marc Gravell
13	Usando DbGeography y DbGeometry	Marc Gravell
14	Utilizando async	Dean Ward, Matt McCabe, Nick, Woodchipper