



EBook Gratuito

APPENDIMENTO

Dapper.NET

Free unaffiliated eBook created from
Stack Overflow contributors.

#dapper

Sommario

Di.....	1
Capitolo 1: Iniziare con Dapper.NET.....	2
Osservazioni.....	2
Cos'è Dapper?.....	2
Come lo ottengo?.....	2
Attività comuni.....	2
Versioni.....	2
Examples.....	2
Installa Dapper da Nuget.....	2
Usando Dapper in C #.....	3
Utilizzo di Dapper in LINQPad.....	3
Capitolo 2: Esecuzione di comandi.....	5
Examples.....	5
Esegui un comando che non restituisce risultati.....	5
Procedura di archiviazione.....	5
Uso semplice.....	5
Input, Output e Return Parametri.....	5
Parametri valutati a tabella.....	5
Capitolo 3: Gestione dei Null.....	7
Examples.....	7
null vs DBNull.....	7
Capitolo 4: Inserti di massa.....	8
Osservazioni.....	8
Examples.....	8
Copia bulk asincrona.....	8
Bulk Copy.....	8
Capitolo 5: Le transazioni.....	10
Sintassi.....	10
Examples.....	10
Utilizzando una transazione.....	10

Accelerata inserti.....	11
Capitolo 6: Multimapping.....	12
Sintassi.....	12
Parametri.....	12
Examples.....	13
Semplice mappatura multi-tabella.....	13
Mappatura uno-a-molti.....	14
Mappatura di oltre 7 tipi.....	16
Mappature personalizzate.....	17
Capitolo 7: Parametri dinamici.....	19
Examples.....	19
Uso di base.....	19
Parametri dinamici in Dapper.....	19
Utilizzando un oggetto modello.....	19
Capitolo 8: Parametro Sintassi di riferimento.....	21
Parametri.....	21
Osservazioni.....	21
Examples.....	21
SQL parametrico di base.....	21
Usando il tuo modello a oggetti.....	22
Procedura di archiviazione.....	22
Valore Inline.....	23
Elenca espansioni.....	23
Esecuzione di operazioni su più insiemi di input.....	24
Parametri pseudo-posizionali (per provider che non supportano parametri denominati).....	25
Capitolo 9: Query di base.....	26
Sintassi.....	26
Parametri.....	26
Examples.....	26
Interrogazione per un tipo statico.....	26
Interrogazione per i tipi dinamici.....	27
Query con parametri dinamici.....	27

Capitolo 10: Risultati multipli	28
Sintassi.....	28
Parametri.....	28
Examples.....	28
Esempio di risultati multipli di base.....	28
Capitolo 11: Tabelle Temp	29
Examples.....	29
Temp Table che esiste mentre la connessione rimane aperta.....	29
Come lavorare con tabelle temporanee.....	29
Capitolo 12: Tipo gestori	31
Osservazioni.....	31
Examples.....	31
Conversione di varchar in IHtmlString.....	31
Installazione di TypeHandler.....	31
Capitolo 13: Utilizzando Async	32
Examples.....	32
Chiamare una stored procedure.....	32
Chiamare una stored procedure e ignorare il risultato.....	32
Capitolo 14: Utilizzando DbGeography e DbGeometry	33
Examples.....	33
Configurazione richiesta.....	33
Usando la geometria e la geografia.....	33
Titoli di coda	35

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.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

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

Capitolo 1: Iniziare con Dapper.NET

Osservazioni

Cos'è Dapper?

Dapper è un micro-ORM per .Net che estende l' `IDbConnection` , semplificando l'impostazione della query, l'esecuzione e la lettura dei risultati.

Come lo ottengo?

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

Attività comuni

- [Query di base](#)
- [Esecuzione di comandi](#)

Versioni

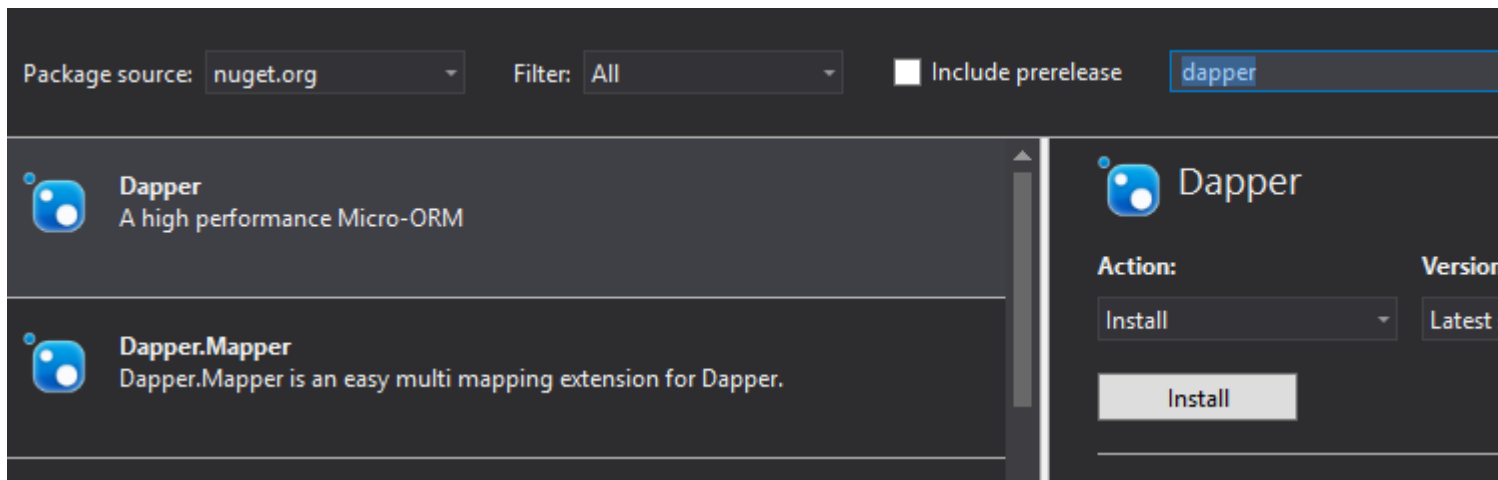
Versione	Gli appunti	Data di rilascio
1.50.0	core-clr / asp.net 5.0 build contro RTM	2016/06/29
1.42.0		2015/05/06
1.40.0		2015/04/03
1.30.0		2014/08/14
1.20.0		2014/05/08
1.10.0		2012-06-27
1.0.0		2011-04-14

Examples

Installa Dapper da Nuget

Cerca nella GUI di Visual Studio:

Strumenti> NuGet Package Manager> Gestisci pacchetti per la soluzione ... (Visual Studio 2015)



Oppure esegui questo comando in un'istanza di Nuget Power Shell per installare l'ultima versione stabile

```
Install-Package Dapper
```

O per una versione specifica

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

Usando Dapper in 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);
        }
    }
}
```

Il wrapping della connessione in un [blocco Using](#) chiuderà la connessione

Utilizzo di Dapper in LINQPad

[LINQPad](#) è ottimo per testare le query del database e include [l'integrazione NuGet](#) . Per utilizzare Dapper in LINQPad premere **F4** per aprire Proprietà query, quindi selezionare **Aggiungi NuGet** . Cerca **net dot dapper** e seleziona **Aggiungi a query** . Dovrai anche fare clic su **Aggiungi spazi**

dei nomi ed evidenziare Dapper per includere i Metodi di estensione nella query LINQPad.

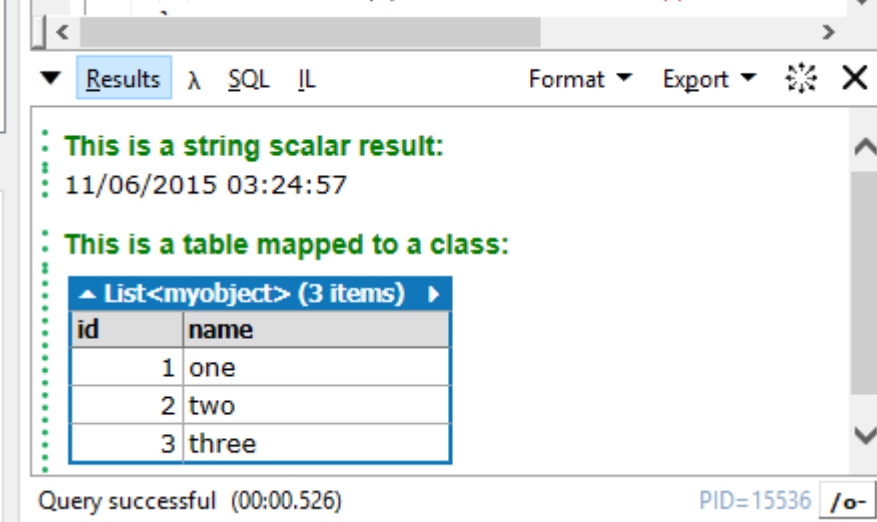
Una volta abilitato Dapper è possibile modificare il menu a discesa Lingua in **C #** , mappare i risultati delle query in classi C # e utilizzare il metodo `.Dump ()` per ispezionare i risultati:

```
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; }
}
```

I risultati durante l'esecuzione del programma sarebbero simili a questi:



Results | SQL | IL | Format | Export

This is a string scalar result:
11/06/2015 03:24:57

This is a table mapped to a class:

List<myobject> (3 items)	
id	name
1	one
2	two
3	three

Query successful (00:00.526) PID=15536

Leggi Iniziare con Dapper.NET online: <https://riptutorial.com/it/dapper/topic/2/iniziare-con-dapper-net>

Capitolo 2: Esecuzione di comandi

Examples

Esegui un comando che non restituisce risultati

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

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

Procedura di archiviazione

Uso semplice

Dapper supporta pienamente i proc memorizzati:

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

Input, Output e Return Parametri

Se vuoi qualcosa di più stravagante, puoi fare:

```
var p = new DynamicParameters();
p.Add("@a", 11);
p.Add("@b",
    DbType.Int32,
    ParameterDirection.Output);
p.Add("@c",
    DbType.Int32,
    ParameterDirection.ReturnValue);

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

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

Parametri valutati a tabella

Se si dispone di una stored procedure che accetta un parametro con valori di tabella, è necessario passare un DataTable che ha la stessa struttura del tipo di tabella in SQL Server. Ecco una definizione per un tipo di tabella e una procedura che lo utilizza:

```

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
*/

```

Per chiamare quella procedura da c #, devi fare quanto segue:

```

// 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);

```

Leggi Esecuzione di comandi online: <https://riptutorial.com/it/dapper/topic/5/esecuzione-di-comandi>

Capitolo 3: Gestione dei Null

Examples

null vs DBNull

In ADO.NET, la gestione corretta di `null` è una costante fonte di confusione. Il punto chiave in dapper è che *non devi*; si occupa di tutto internamente.

- i valori dei parametri `null` vengono inviati correttamente come `DBNull.Value`
- i valori letti che sono `null` sono presentati come `null`, o (nel caso di mappatura ad un tipo noto) semplicemente ignorati (lasciando il loro default basato sul tipo)

Funziona solo:

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

Leggi Gestione dei Null online: <https://riptutorial.com/it/dapper/topic/13/gestione-dei-null>

Capitolo 4: Inserti di massa

Osservazioni

`WriteToServer` e `WriteToServerAsync` hanno sovraccarichi che accettano `IDataReader` (visto negli esempi), `DataTable` e array `DataRow[]` (`DataRow[]`) come origine dei dati per la copia bulk.

Examples

Copia bulk asincrona

Questo esempio utilizza un metodo `ToDataReader` descritto qui [Creazione di un elenco dati generico DataReader per SqlBulkCopy](#).

Questo può anche essere fatto usando metodi non asincroni.

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

Bulk Copy

Questo esempio utilizza un metodo `ToDataReader` descritto qui [Creazione di un elenco dati generico DataReader per SqlBulkCopy](#).

Questo può anche essere fatto usando metodi asincroni.

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

Leggi Inserti di massa online: <https://riptutorial.com/it/dapper/topic/6279/inserti-di-massa>

Capitolo 5: Le transazioni

Sintassi

- `conn.Execute (sql, transaction: tran);` // specifica il parametro per nome
- `conn.Execute (sql, parameters, tran);`
- `conn.Query (sql, transaction: tran);`
- `conn.Query (sql, parameters, tran);`
- attende `conn.ExecuteAsync (sql, transaction: tran);` // Asincrono
- attende `conn.ExecuteAsync (sql, parameters, tran);`
- attende `conn.QueryAsync (sql, transaction: tran);`
- attende `conn.QueryAsync (sql, parameters, tran);`

Examples

Utilizzando una transazione

In questo esempio viene utilizzato `SqlConnection`, ma è supportato qualsiasi `IDbConnection`.

Anche qualsiasi `IDbTransaction` è supportato dalla relativa `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
                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;
            }
        }
    }
}
```

Accelera inserti

Racchiudere un gruppo di inserti in una transazione li accelera in base a questa [domanda / risposta StackOverflow](#) .

È possibile utilizzare questa tecnica, oppure è possibile utilizzare Bulk Copy per accelerare una serie di operazioni correlate da eseguire.

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

Leggi Le transazioni online: <https://riptutorial.com/it/dapper/topic/6601/le-transazioni>

Capitolo 6: Multimapping

Sintassi

- `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)`

Parametri

Parametro	Dettagli
cnn	La tua connessione al database, che deve essere già aperta.
sql	Comando da eseguire.
tipi	Matrice di tipi nel set di record.
carta geografica	<code>Func<></code> che gestisce la costruzione del risultato di ritorno.
param	Oggetto da cui estrarre i parametri.
transazione	Transazione di cui questa query fa parte, se esiste.
tamponata	Indica se bufferizzare o meno i risultati della query. Questo è un parametro facoltativo con il valore predefinito <code>true</code> . Quando il buffer è vero, i risultati vengono memorizzati in un <code>List<T></code> e quindi restituiti come <code>IEnumerable<T></code> che è sicuro per l'enumerazione multipla. Quando il buffer è falso, la connessione sql viene mantenuta aperta fino al termine della lettura, consentendo di elaborare una singola riga alla volta in memoria. Più enumerazioni genereranno ulteriori connessioni al database. Mentre <code>buffered false</code> è altamente efficiente per ridurre l'utilizzo della memoria se si mantengono solo frammenti molto piccoli dei record restituiti, ha un overhead delle prestazioni considerevole rispetto alla materializzazione impaziente del set di risultati. Infine, se si dispone di numerose connessioni SQL simultanee non bufferizzate, è necessario considerare l'inattività del pool di connessioni che causa il blocco delle richieste fino a quando le connessioni diventano disponibili.
dividersi	Il campo dovremmo dividere e leggere il secondo oggetto da

Parametro	Dettagli
	(predefinito: id). Questo può essere un elenco delimitato da virgole quando più di 1 tipo è contenuto in un record.
CommandTimeout	Numero di secondi prima del timeout dell'esecuzione del comando.
CommandType	È un processo memorizzato o un batch?

Examples

Semplice mappatura multi-tabella

Diciamo che abbiamo una query sui restanti cavalieri che devono compilare una classe Person.

Nome	Nato	Residenza
Daniel Dennett	1942	Stati Uniti d'America
Sam Harris	1967	Stati Uniti d'America
Richard Dawkins	1941	Regno Unito

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

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

Possiamo popolare la classe persona e la proprietà Residenza con un'istanza di Paese utilizzando una `Query<>` sovraccarico `Query<>` che accetta un `Func<>` che può essere utilizzato per comporre l'istanza restituita. Il `Func<>` può richiedere fino a 7 tipi di input con l'argomento generico finale che è sempre il tipo restituito.

```
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.Residence = country;
    return person;
});
```

```
},
splitOn: "Residence");
```

Notare l'uso dello `splitOn: "Residence"` argomento `splitOn: "Residence"` che è la prima colonna del prossimo tipo di classe da compilare (in questo caso `Country`). Dappertutto cercherà automaticamente una colonna denominata `Id` da dividere, ma se non ne trova una e `splitOn` non viene fornito un `System.ArgumentException` verrà lanciato con un messaggio utile. Quindi, sebbene sia facoltativo, di solito devi fornire un valore `splitOn`.

Mappatura uno-a-molti

Diamo un'occhiata a un esempio più complesso che contiene una relazione uno-a-molti. La nostra query ora conterrà più righe contenenti dati duplicati e dovremo gestirli. Lo facciamo con una ricerca in una chiusura.

La query cambia leggermente come fanno le classi di esempio.

Id	Nome	Nato	CountryId	Nome del paese	BookID	BookName
1	Daniel Dennett	1942	1	Stati Uniti d'America	1	scambi di idee
1	Daniel Dennett	1942	1	Stati Uniti d'America	2	Camera di gomito
2	Sam Harris	1967	1	Stati Uniti d'America	3	Il paesaggio morale
2	Sam Harris	1967	1	Stati Uniti d'America	4	Svegliarsi: una guida alla spiritualità senza religione
3	Richard Dawkins	1941	2	Regno Unito	5	La magia della realtà: come sappiamo che cosa è veramente vero
3	Richard Dawkins	1941	2	Regno Unito	6	An Appetite for Wonder: The Making of a Scientist

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

```

Il dizionario `remainingHorsemen` sarà popolato con istanze completamente materializzate degli oggetti `Person`. Per ogni riga del risultato della query vengono passati i valori mappati delle istanze dei tipi definiti negli argomenti lambda e sta a te decidere come gestirlo.

```

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

```

Si noti come l'argomento `splitOn` è un elenco delimitato da virgole delle prime colonne del prossimo tipo.

Mappatura di oltre 7 tipi

A volte il numero di tipi che stai mappando supera il 7 fornito da `Func <>` che esegue la costruzione.

Invece di usare la `Query<>` con gli input dell'argomento di tipo generico, forniremo i tipi da mappare come una matrice, seguita dalla funzione di mappatura. A parte l'impostazione manuale iniziale e la trasmissione dei valori, il resto della funzione non 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.Residence == null)
    {
        if (country == null)
        {
            country = new Country { CountryName = "" };
        }
        personEntity.Residence = 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");

```

Mappature personalizzate

Se i nomi delle colonne delle query non corrispondono alle tue classi, puoi configurare i mapping per i tipi. Questo esempio dimostra la mappatura utilizzando

`System.Data.Linq.Mapping.ColumnAttribute` e una mappatura personalizzata.

Le mappature devono essere configurate solo una volta per tipo, quindi impostarle all'avvio dell'applicazione o da qualche altra parte in cui vengono inizializzate solo una volta.

Assumendo di nuovo la stessa query dell'esempio uno a molti e le classi refactored verso nomi migliori come:

```

public class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public int Born { get; set; }
    public Country Residence { 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; }
}

```

Nota come `Book` non si basa su `ColumnAttribute` ma dovremmo mantenere la dichiarazione `if`

Ora posiziona questo codice di mappatura da qualche parte nell'applicazione in cui viene eseguito solo una volta:

```

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);

```

Quindi la query viene eseguita utilizzando uno degli esempi precedenti `Query<>` .

In [questa risposta](#) viene mostrato un modo più semplice per aggiungere i mapping.

Leggi [Multimapping online](https://riptutorial.com/it/dapper/topic/351/multimapping): <https://riptutorial.com/it/dapper/topic/351/multimapping>

Capitolo 7: Parametri dinamici

Examples

Uso di base

Non è sempre possibile impacchettare ordinatamente tutti i parametri in un singolo oggetto / chiamata. Per aiutare con scenari più complicati, dapper consente al parametro `param` di essere un'istanza `IDynamicParameters`. Se si esegue questa operazione, il metodo `AddParameters` personalizzato viene chiamato al momento opportuno e viene consegnato il comando a cui aggiungere. Nella maggior parte dei casi, tuttavia, è sufficiente utilizzare il tipo di `DynamicParameters` preesistente:

```
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");
```

Questo mostra:

- (facoltativo) popolazione da un oggetto esistente
- (facoltativo) aggiunta di parametri aggiuntivi al volo
- passare i parametri al comando
- recuperare qualsiasi valore aggiornato dopo che il comando è terminato

Si noti che a causa di come funzionano i protocolli RDBMS, di solito è solo affidabile ottenere valori di parametri aggiornati **dopo che** tutti i dati (da un'operazione `Query` o `QueryMultiple`) sono stati **completamente** utilizzati (ad esempio, su SQL Server, i valori dei parametri aggiornati sono alla *fine* del flusso TDS).

Parametri dinamici in Dapper

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

Utilizzando un oggetto modello

È possibile utilizzare un'istanza di un oggetto per formare i parametri

```
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);
```

Puoi anche usare un oggetto anonimo o un `Dictionary`

Leggi Parametri dinamici online: <https://riptutorial.com/it/dapper/topic/12/parametri-dinamici>

Capitolo 8: Parametro Sintassi di riferimento

Parametri

Parametro	Dettagli
<code>this cnn</code>	La connessione al database sottostante - <code>this</code> denota un metodo di estensione; la connessione non ha bisogno di essere aperta - se non è aperta, viene aperta e chiusa automaticamente.
<code><T> / Type</code>	(facoltativo) il tipo di oggetto da restituire; se viene utilizzata l'API non generica / non di <code>Type</code> , viene restituito un oggetto <code>dynamic</code> per riga, simulando una proprietà denominata per nome di colonna restituita dalla query (questo oggetto <code>dynamic</code> implementa anche <code>IDictionary<string, object></code>).
<code>sql</code>	L'SQL da eseguire
<code>param</code>	(facoltativo) I parametri da includere.
<code>transaction</code>	(facoltativo) La transazione del database da associare al comando
<code>buffered</code>	(facoltativo) Se pre-consumare i dati in un elenco (predefinito), anziché esporre un <code>IEnumerable</code> aperto sul lettore live
<code>commandTimeout</code>	(facoltativo) il timeout da utilizzare sul comando; se non specificato, si presume <code>SqlMapper.Settings.CommandTimeout</code> (se specificato)
<code>commandType</code>	Il tipo di comando che viene eseguito; il valore predefinito è <code>CommandText</code>

Osservazioni

La sintassi per esprimere i parametri varia tra RDBMS. Tutti gli esempi sopra riportati utilizzano la sintassi di SQL Server, ovvero `@foo`; tuttavia, `?foo` e `:foo` dovrebbero funzionare bene.

Examples

SQL parametrico di base

Dapper semplifica il seguire le migliori pratiche mediante SQL completamente parametrizzato.

I parametri sono importanti, quindi dapper rende facile farlo bene. Devi semplicemente esprimere i tuoi parametri nel modo normale per il tuo RDBMS (solitamente `@foo ?foo o :foo`) e dare a dapper un oggetto che *ha un membro chiamato* `foo`. Il modo più comune per farlo è con un tipo anonimo:

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

E ... questo è tutto. Dapper aggiungerà i parametri richiesti e tutto dovrebbe funzionare.

Usando il tuo modello a oggetti

Puoi anche usare il tuo modello di oggetto esistente come parametro:

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

Dapper usa il comando-testo per determinare quali membri dell'oggetto aggiungere - di solito non aggiunge cose non necessarie come `Description`, `IsActive`, `CreationDate` perché il comando che abbiamo emesso chiaramente non li coinvolge - anche se ci sono casi in cui potrebbe farlo, per esempio se il tuo comando contiene:

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

Non tenta di capire che quanto sopra è solo un commento.

Procedura di archiviazione

I parametri delle stored procedure funzionano esattamente allo stesso modo, ad eccezione del fatto che dapper non può tentare di determinare cosa dovrebbe / non dovrebbe essere incluso - tutto ciò che è disponibile viene trattato come un parametro. Per questo motivo, i tipi anonimi sono generalmente preferiti:

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

Valore Inline

A volte la convenienza di un parametro (in termini di manutenzione ed espressività) può essere superata dal suo costo in termini di prestazioni per trattarlo come un parametro. Ad esempio, quando la dimensione della pagina è fissata da un'impostazione di configurazione. O un valore di stato è abbinato a un valore `enum`. Tenere conto:

```
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 });
```

L'unico parametro *reale* qui è `customerId` - gli altri due sono pseudo-parametri che in realtà non cambieranno. Spesso il RDBMS può fare un lavoro migliore se rileva questi come costanti. Dapper ha una sintassi speciale per questo - `{=name}` invece di `@name` - che *si applica solo* ai tipi numerici. (Ciò minimizza qualsiasi superficie di attacco dall'iniezione SQL). Un esempio è il seguente:

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

Dapper sostituisce i valori con i letterali prima di emettere l'SQL, quindi l'RDBMS vede effettivamente qualcosa del tipo:

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

Ciò è particolarmente utile quando si consente ai sistemi RDBMS non solo di prendere decisioni migliori, ma di aprire piani di query che i parametri attuali impediscono. Ad esempio, se un predicato di colonna è contrario a un parametro, non è possibile utilizzare un indice filtrato con valori specifici su tali colonne. Questo perché la *prossima* query potrebbe avere un parametro diverso da uno di quei valori specificati.

Con valori letterali, Query Optimizer è in grado di utilizzare gli indici filtrati poiché sa che il valore non può cambiare nelle query future.

Elenca espansioni

Uno scenario comune nelle query di database è `IN (...)` dove l'elenco qui viene generato in fase di esecuzione. La maggior parte degli RDBMS non ha una buona metafora per questo - e per questo non esiste una soluzione *cross-RDBMS* universale. Invece, dapper offre una leggera espansione automatica dei comandi. Tutto ciò che è richiesto è un valore di parametro fornito che è `IEnumerable`. Un comando che coinvolge `@foo` viene esteso a `(@foo0,@foo1,@foo2,@foo3)` (per una

sequenza di 4 elementi). L'uso più comune di questo sarebbe `IN` :

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

Questo quindi si espande automaticamente per emettere il codice SQL appropriato per il recupero di più righe:

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

con i parametri `@orderIds0` ecc vengono aggiunti come valori presi da array. Si noti che il fatto che SQL non sia originariamente valido è intenzionale, per garantire che questa funzione non venga utilizzata per errore. Questa funzionalità funziona anche correttamente con `OPTIMIZE FOR / UNKNOWN` query-hint in SQL Server; se usi:

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

lo espanderà correttamente per:

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

Esecuzione di operazioni su più insiemi di input

A volte, vuoi fare la stessa cosa più volte. Dapper supporta questo metodo nel metodo `Execute` se il parametro *più esterno* (che di solito è un singolo tipo anonimo o un'istanza del modello di dominio) viene effettivamente fornito come una sequenza `IEnumerable` . Per esempio:

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

Qui, dapper sta facendo un semplice ciclo sui nostri dati, essenzialmente come se avessimo fatto:

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

Questo utilizzo diventa *particolarmente* interessante quando combinato con l'API `async` su una connessione che è esplicitamente configurata per tutti i "Multiple Active Result Set" - in questo utilizzo, dapper eseguirà automaticamente la *pipeline* delle operazioni, quindi non si paga il costo di latenza per riga. Ciò richiede un utilizzo leggermente più complicato,

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

Si noti, tuttavia, che si potrebbe anche voler esaminare i parametri valutati in tabella.

Parametri pseudo-posizionali (per provider che non supportano parametri denominati)

Alcuni provider ADO.NET (in particolare: OleDb) non supportano i parametri *denominati*; i parametri sono invece specificati solo dalla *posizione*, con il ? posizionare titolare. Dapper non saprebbe quale membro usare per questi, quindi dapper permette una sintassi alternativa, ? ?foo?; questo sarebbe lo stesso di @foo o :foo in altre varianti SQL, ad eccezione del fatto che dapper **sostituirà** completamente il token dei parametri con ? prima di eseguire la query.

Funziona in combinazione con altre funzionalità come l'espansione delle liste, quindi quanto segue è valido:

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

I membri `.region` e `.users` vengono utilizzati di conseguenza e l'SQL rilasciato è (ad esempio, con 3 utenti):

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

Si noti, tuttavia, che dapper **non** consente di utilizzare lo stesso parametro più volte quando si utilizza questa funzione; questo per evitare di dover aggiungere lo stesso valore del parametro (che potrebbe essere grande) più volte. Se è necessario fare riferimento allo stesso valore più volte, prendere in considerazione la dichiarazione di una variabile, ad esempio:

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

Se le variabili non sono disponibili, puoi utilizzare nomi di membri duplicati nei parametri - questo renderà anche ovvio che il valore viene inviato più volte:

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

Leggi Parametro Sintassi di riferimento online: <https://riptutorial.com/it/dapper/topic/10/parametro-sintassi-di-riferimento>

Capitolo 9: Query di base

Sintassi

- `public static IEnumerable <T> Query <T>` (questo `IDcConnection cnn`, stringa `sql`, oggetto `param = null`, `SqlTransaction transaction = null`, `bool buffered = true`)
- `public static IEnumerable <dynamic> Query` (questo `IDcConnection cnn`, stringa `sql`, oggetto `param = null`, `SqlTransaction transaction = null`, `bool buffered = true`)

Parametri

Parametro	Dettagli
<code>cnn</code>	La tua connessione al database, che deve essere già aperta.
<code>sql</code>	Comando da eseguire.
<code>param</code>	Oggetto da cui estrarre i parametri.
<code>transazione</code>	Transazione di cui questa query fa parte, se esiste.
<code>tamponata</code>	Indica se bufferizzare o meno i risultati della query. Questo è un parametro facoltativo con il valore predefinito <code>true</code> . Quando il buffer è vero, i risultati vengono memorizzati in un <code>List<T></code> e quindi restituiti come <code>IEnumerable<T></code> che è sicuro per l'enumerazione multipla. Quando il buffer è falso, la connessione <code>sql</code> viene mantenuta aperta fino al termine della lettura, consentendo di elaborare una singola riga alla volta in memoria. Più enumerazioni genereranno ulteriori connessioni al database. Mentre <code>buffered false</code> è altamente efficiente per ridurre l'utilizzo della memoria se si mantengono solo frammenti molto piccoli dei record restituiti, ha un overhead delle prestazioni considerevole rispetto alla materializzazione impaziente del set di risultati. Infine, se si dispone di numerose connessioni SQL simultanee non bufferizzate, è necessario considerare l'inattività del pool di connessioni che causa il blocco delle richieste fino a quando le connessioni diventano disponibili.

Examples

Interrogazione per un tipo statico

Per i tipi noti al momento della compilazione, utilizzare un parametro generico 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);

```

Interrogazione per i tipi dinamici

Puoi anche eseguire query dinamicamente se non utilizzi il tipo generico.

```

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

```

Query con parametri dinamici

```

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);
}

```

Leggi Query di base online: <https://riptutorial.com/it/dapper/topic/3/query-di-base>

Capitolo 10: Risultati multipli

Sintassi

- `public static IMapper.GridReader QueryMultiple` (questo `IDcConnection cnn`, stringa `sql`, oggetto `param = null`, `IDbTransaction transaction = null`, `int? commandTimeout = null`, `CommandType? commandType = null`)
- `public static IMapper.GridReader QueryMultiple` (questo comando `IDbConnection cnn`, `CommandDefinition`)

Parametri

Parametro	Dettagli
<code>cnn</code>	La tua connessione al database, deve essere già aperta
<code>sql</code>	La stringa <code>sql</code> da elaborare contiene più query
<code>param</code>	Oggetto da cui estrarre i parametri
<code>SqlMapper.GridReader</code>	Fornisce interfacce per la lettura di più set di risultati da una query di Dapper

Examples

Esempio di risultati multipli di base

Per recuperare più griglie in una singola query, viene utilizzato il metodo `QueryMultiple`. Ciò consente quindi di recuperare ogni griglia in *sequenza* attraverso chiamate successive contro il `GridReader` restituito.

```
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();
}
```

Leggi Risultati multipli online: <https://riptutorial.com/it/dapper/topic/8/risultati-multipli>

Capitolo 11: Tabelle Temp

Examples

Temp Table che esiste mentre la connessione rimane aperta

Quando la tabella temporanea viene creata da sola, rimarrà mentre la connessione è aperta.

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

Come lavorare con tabelle temporanee

Il punto sulle tabelle temporanee è che sono limitate allo scopo della connessione. Dapper aprirà e chiuderà automaticamente una connessione se non è già aperta. Ciò significa che qualsiasi tabella temporanea verrà persa direttamente dopo la sua creazione, se la connessione passata a Dapper non è stata aperta.

Questo non funzionerà:

```
private async Task<IEnumerable<int>> SelectWidgetsError()
{
    using (var conn = new SqlConnection(connectionString))
    {
        await conn.ExecuteNonQuery(@"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;");
}
}

```

D'altra parte, queste due versioni funzioneranno:

```

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;");
    }
}
}

```

Leggi Tabelle Temp online: <https://riptutorial.com/it/dapper/topic/6594/tabelle-temp>

Capitolo 12: Tipo gestori

Osservazioni

I gestori di tipi consentono la conversione dei tipi di database in tipi personalizzati Netto.

Examples

Conversione di varchar in 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());
    }
}
```

Installazione di TypeHandler

Il gestore dei tipi sopra può essere installato in `SqlMapper` usando il metodo `AddTypeHandler`.

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

L'inferenza di tipo consente di omettere il parametro di tipo generico:

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

C'è anche un sovraccarico a due argomenti che accetta un argomento `Type` esplicito:

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

Leggi Tipo gestori online: <https://riptutorial.com/it/dapper/topic/6/tipo-gestori>

Capitolo 13: Utilizzando Async

Examples

Chiamare una stored procedure

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

Chiamare una stored procedure e ignorare il risultato

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

Leggi Utilizzando Async online: <https://riptutorial.com/it/dapper/topic/1353/utilizzando-async>

Capitolo 14: Utilizzando DbGeography e DbGeometry

Examples

Configurazione richiesta

1. installare l'assembly `Microsoft.SqlServer.Types` richiesto; non sono installati per impostazione predefinita e sono [disponibili da Microsoft qui](#) come "Tipi di CLR del sistema Microsoft® per Microsoft® SQL Server® 2012". Si noti che esistono programmi di installazione separati per x86 e x64.
2. installa `Dapper.EntityFramework` (o l'equivalente con il nome `Dapper.EntityFramework`); questo potrebbe essere fatto tramite l'interfaccia utente "Gestisci pacchetti NuGet ..." dell'IDE o (nella Console di Gestione pacchetti):

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

3. aggiungere i reindirizzamenti dell'associazione necessari; questo perché Microsoft spedisce v11 degli assembly, ma Entity Framework richiede v10; puoi aggiungere quanto segue ad `app.config` o `web.config` sotto l'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. di "dapper" sui nuovi gestori di tipi disponibili, aggiungendo (da qualche parte nel tuo avvio, prima che provi ad usare il database):

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

Usando la geometria e la geografia

Una volta registrati i gestori di tipi, tutto dovrebbe funzionare automaticamente e dovresti essere in grado di utilizzare questi tipi come parametri o valori di ritorno:

```
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}");
```

Leggi Utilizzando DbGeography e DbGeometry online:

<https://riptutorial.com/it/dapper/topic/3984/utilizzando-dbgeography-e-dbgeometry>

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Dapper.NET	Adam Lear , balpha , Community , Eliza , Greg Bray , Jarrod Dixon , Kevin Montrose , Matt McCabe , Nick , Rob , Shog9
2	Esecuzione di comandi	Adam Lear , Jarrod Dixon , Skivvz , takrl
3	Gestione dei Null	Marc Gravell
4	Inseriti di massa	jhamm
5	Le transazioni	jhamm
6	Multimapping	Devon Burriss
7	Parametri dinamici	Marc Gravell , Matt McCabe , Meer
8	Parametro Sintassi di riferimento	4444 , Marc Gravell , Nick Craver
9	Query di base	Adam Lear , Chris Marisic , Cigano Morrison Mendez , Community , cubrr , Jarrod Dixon , jrummell , Kevin Montrose , Matt McCabe
10	Risultati multipli	Marc Gravell , Yaakov Ellis
11	Tabelle Temp	jhamm , Rob , takrl
12	Tipo gestori	Benjamin Hodgson , Community , Marc Gravell
13	Utilizzando Async	Dean Ward , Matt McCabe , Nick , Woodchipper
14	Utilizzando DbGeography e DbGeometry	Marc Gravell