Kostenloses eBook

LERNEN

Dapper.NET

Free unaffiliated eBook created from
Stack Overflow contributors.

#dapper

Inhaltsverzeichnis

Über

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

Kapitel 1: Erste Schritte mit Dapper.NET

Bemerkungen

Was ist Dapper?

Dapper ist ein Mikro-ORM für .Net, das Ihre `IDbConnection` und das Einrichten, Ausführen und Ablesen von Abfragen vereinfacht.

Wie bekomme ich es?

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

Häufige Aufgaben

- Grundlegendes Abfragen
- Befehle ausführen

Versionen

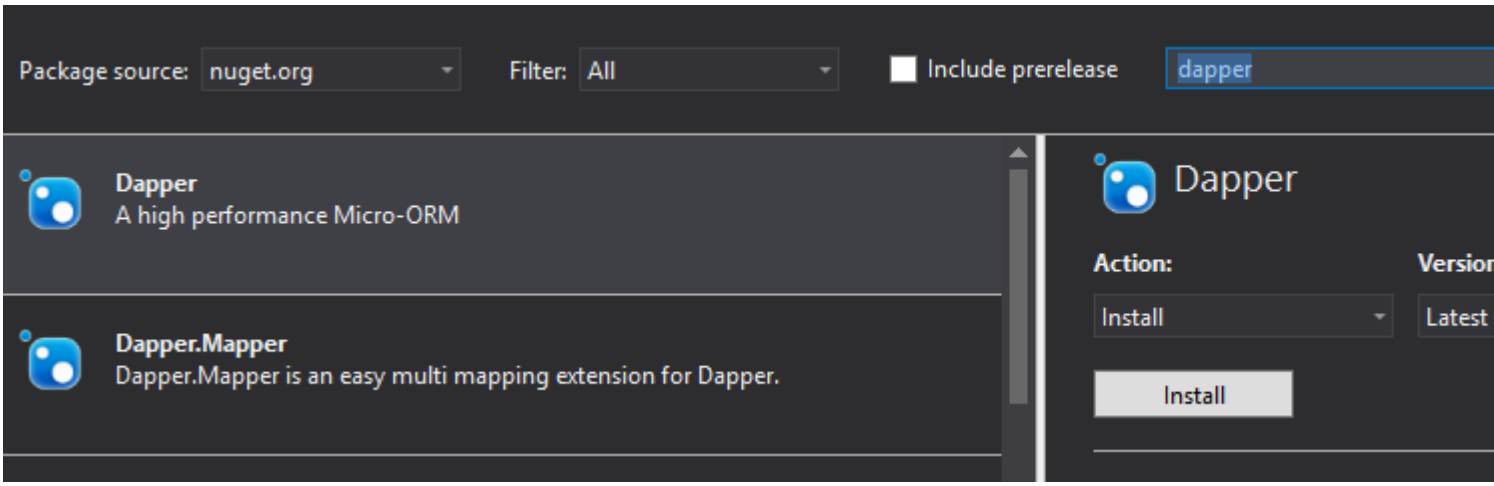
Ausführung	Anmerkungen	Veröffentlichungsdatum
1,50,0	Core-Clr / Asp.net 5.0 Build gegen RTM	2016-06-29
1.42.0		2015-05-06
1,40,0		2015-04-03
1,30,0		2014-08-14
1,20,0		2014-05-08
1.10.0		2012-06-27
1.0.0		2011-04-14

Examples

Installieren Sie Dapper von Nuget

Entweder Suche in der Visual Studio-GUI:

Tools> NuGet Package Manager> Pakete für Lösung verwalten ... (Visual Studio 2015)



Oder führen Sie diesen Befehl in einer Nuget Power Shell-Instanz aus, um die neueste stabile Version zu installieren

```
Install-Package Dapper
```

Oder für eine bestimmte Version

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

Verwenden von 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);
        }
    }
}
```

Durch das Umschließen der Verbindung in einen `Using` Block wird die Verbindung geschlossen

Verwenden von Dapper in LINQPad

LINQPad eignet sich hervorragend zum Testen von Datenbankabfragen und beinhaltet die Integration von NuGet . Um Dapper in LINQPad zu verwenden, drücken Sie **F4** , um die Abfrageeigenschaften zu öffnen, und wählen **Sie dann NuGet hinzufügen** . Suchen Sie nach

dapper dot net und wählen **Sie Zu Abfrage hinzufügen** . Sie möchten auch auf **Namespaces** hinzufügen klicken und Dapper markieren, um die Erweiterungsmethoden in Ihre LINQPad-Abfrage aufzunehmen.

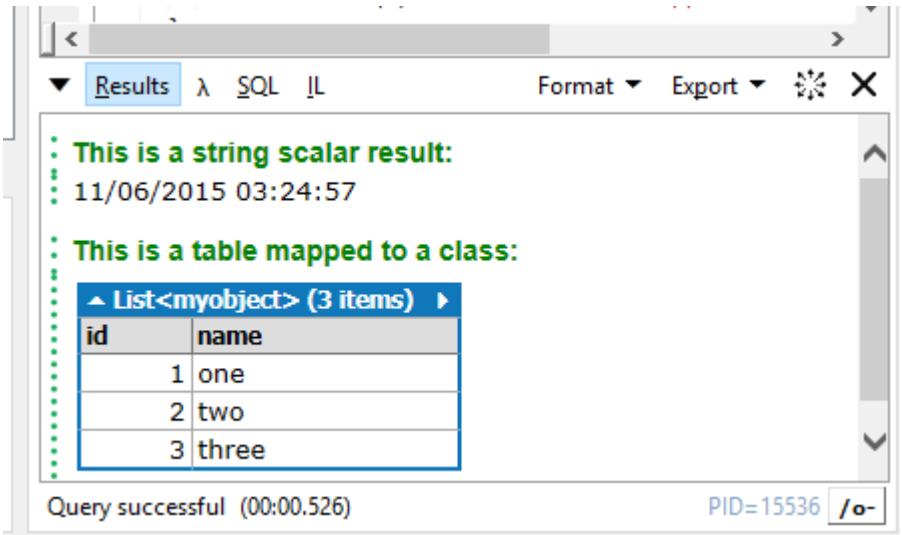
Wenn Dapper aktiviert ist, können Sie die Dropdown-Liste Sprache in **C # -Programm** ändern, Abfrageergebnisse C # -Klassen zuordnen und die .Dump () -Methode verwenden, um die Ergebnisse zu überprüfen:

```
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; }
}
```

Die Ergebnisse beim Ausführen des Programms würden folgendermaßen aussehen:



Erste Schritte mit Dapper.NET online lesen: https://riptutorial.com/de/dapper/topic/2/erste-schritte-mit-dapper-net

Kapitel 2: Async verwenden

Examples

Eine gespeicherte Prozedur aufrufen

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

Aufrufen einer gespeicherten Prozedur und Ignorieren des Ergebnisses

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

Async verwenden online lesen: https://riptutorial.com/de/dapper/topic/1353/async-verwenden

Kapitel 3: Befehle ausführen

Examples

Führen Sie einen Befehl aus, der keine Ergebnisse zurückgibt

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

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

Gespeicherte Prozeduren

Einfache Benutzung

Dapper unterstützt gespeicherte Prozeduren vollständig:

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

Input-, Output- und Return-Parameter

Wenn Sie etwas mehr Lust haben wollen, können Sie:

```
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");
```

Tabellenwertparameter

Wenn Sie über eine gespeicherte Prozedur verfügen, die einen Tabellenwertparameter akzeptiert, müssen Sie eine DataTable übergeben, die dieselbe Struktur wie der Tabellentyp in SQL Server hat. Hier ist eine Definition für einen Tabellentyp und eine Prozedur, die diese verwenden:

```
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
*/
```

Um diese Prozedur in c # aufzurufen, müssen Sie Folgendes tun:

```
// 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);
```

Befehle ausführen online lesen: https://riptutorial.com/de/dapper/topic/5/befehle-ausfuhren

Kapitel 4: Bulk-Einsätze

Bemerkungen

Der `WriteToServer` und der `WriteToServerAsync` haben Überladungen, die IDataReader (in den Beispielen zu sehen), DataTable und DataRow-Arrays (`DataRow[]`) als Datenquelle für die Massenkopie akzeptieren.

Examples

Async-Massenkopie

In diesem Beispiel wird eine `ToDataReader` hier beschriebene Methode eine generische Liste Datareader für SqlBulkCopy erstellen .

Dies kann auch mit nicht asynchronen Methoden erfolgen.

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

Massenkopie

In diesem Beispiel wird eine `ToDataReader` hier beschriebene Methode eine generische Liste Datareader für SqlBulkCopy erstellen .

Dies kann auch mit asynchronen Methoden erfolgen.

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

Bulk-Einsätze online lesen: https://riptutorial.com/de/dapper/topic/6279/bulk-einsatze

Kapitel 5: Dynamische Parameter

Examples

Grundlegende Verwendung

Es ist nicht immer möglich, alle Parameter in einem einzigen Objekt / Aufruf zusammenzufassen. Um bei komplizierteren Szenarien zu helfen, ermöglicht `param` , dass der Parameter `IDynamicParameters` eine `IDynamicParameters` Instanz ist. Wenn Sie dies tun, wird Ihre benutzerdefinierte `AddParameters` Methode zu gegebener Zeit aufgerufen und der Befehl zum Anhängen übergeben. In den meisten Fällen reicht es jedoch aus, den bereits vorhandenen `DynamicParameters` Typ zu verwenden:

```
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");
```

Das zeigt:

- (optional) Population eines vorhandenen Objekts
- (optional) Hinzufügen zusätzlicher Parameter im laufenden Betrieb
- Übergabe der Parameter an den Befehl
- Abrufen aller aktualisierten Werte, nachdem der Befehl abgeschlossen wurde

Beachten Sie, dass aufgrund wie RDBMS Protokolle arbeiten, ist es in der Regel nur zuverlässig jegliche Daten zu erhalten , aktualisiert Parameterwerte **nach** (aus einer `Query` oder QueryMultiple` Betrieb) wurde , sind **vollständig** verbraucht (zum Beispiel auf SQL Server, aktualisiert Parameterwerte am *Ende* des TDS-Streams).

Dynamische Parameter in Dapper

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

Verwenden eines Vorlagenobjekts

Sie können eine Instanz eines Objekts verwenden, um Ihre Parameter zu bilden

```
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);
```

Sie können auch ein anonymes Objekt oder ein `Dictionary`

Dynamische Parameter online lesen: https://riptutorial.com/de/dapper/topic/12/dynamische-parameter

Kapitel 6: Grundlegendes Abfragen

Syntax

- public static IEnumerable <T> Query <T> (diese IDbConnection-cnn, Zeichenfolge sql, object param = null, SqlTransaction-Transaktion = null, bool gepuffert = true)
- public static IEnumerable <dynamic> Query (diese IDbConnection-CNN, Zeichenfolge sql, object param = null, Transaktion SqlTransaction = null, bool buffered = true)

Parameter

Parameter	Einzelheiten
cnn	Ihre Datenbankverbindung, die bereits geöffnet sein muss.
sql	Befehl zum Ausführen
param	Objekt, aus dem Parameter extrahiert werden sollen.
Transaktion	Transaktion, zu der diese Abfrage gehört, sofern vorhanden.
gepuffert	Ob das Lesen der Ergebnisse der Abfrage zwischengespeichert werden soll oder nicht. Dies ist ein optionaler Parameter mit der Standardeinstellung true. Wenn gepuffert wahr ist, werden die Ergebnisse in eine `List<T>` gepuffert und dann als `IEnumerable<T>` , das für mehrere Aufzählungen sicher ist. Wenn gepuffert "false" ist, bleibt die SQL-Verbindung offen, bis Sie mit dem Lesen fertig sind, sodass Sie eine einzelne Zeile zur Zeit im Speicher bearbeiten können. Durch mehrere Aufzählungen werden zusätzliche Verbindungen zur Datenbank hergestellt. Zwar ist gepuffertes false sehr effizient, um die Speicherauslastung zu reduzieren, wenn Sie nur sehr kleine Fragmente der zurückgegebenen Datensätze verwalten, was jedoch einen erheblichen Performance-Overhead im Vergleich zu eifrigem Ergebnis bewirkt. Wenn Sie über zahlreiche gleichzeitige ungepufferte SQL-Verbindungen verfügen, müssen Sie bedenken, dass der Verbindungspool verhungert, dass Anforderungen blockiert werden, bis Verbindungen verfügbar werden.

Examples

Abfrage nach einem statischen Typ

Verwenden Sie für zur Kompilierzeit bekannte Typen einen generischen Parameter mit der `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);
```

Abfrage nach dynamischen Typen

Sie können auch dynamisch abfragen, wenn Sie den generischen Typ weglassen.

```
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
```

Abfrage mit dynamischen Parametern

```
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);
}
```

Grundlegendes Abfragen online lesen: https://riptutorial.com/de/dapper/topic/3/grundlegendes-abfragen

Kapitel 7: Mehrere Ergebnisse

Syntax

- public static SqlMapper.GridReader QueryMultiple (diese IDbConnection-CNN, Zeichenfolge sql, object param = null, IDbTransaction-Transaktion = null, int? commandTimeout = null, CommandType? commandType = null)
- public static SqlMapper.GridReader QueryMultiple (dieser Befehl IDbConnection cnn, CommandDefinition)

Parameter

Parameter	Einzelheiten
cnn	Ihre Datenbankverbindung muss bereits geöffnet sein
sql	Die zu verarbeitende SQL-Zeichenfolge enthält mehrere Abfragen
param	Objekt, aus dem Parameter extrahiert werden sollen
SqlMapper.GridReader	Stellt Schnittstellen zum Lesen mehrerer Ergebnismengen aus einer Dapper-Abfrage bereit

Examples

Beispiel für mehrere Basisergebnisse

Um mehrere Raster in einer einzelnen Abfrage `QueryMultiple` , wird die `QueryMultiple` Methode verwendet. Auf diese Weise können Sie jedes Raster *sequentiell* durch aufeinanderfolgende Aufrufe des zurückgegebenen `GridReader` .

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

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

Mehrere Ergebnisse online lesen: https://riptutorial.com/de/dapper/topic/8/mehrere-ergebnisse

Kapitel 8: Multimapping

Syntax

- `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)`

Parameter

Parameter	Einzelheiten
cnn	Ihre Datenbankverbindung, die bereits geöffnet sein muss.
sql	Befehl zum Ausführen
Typen	Array von Typen im Datensatz.
Karte	`Func<>` , die die Konstruktion des Rückgabeergebnisses übernimmt.
param	Objekt, aus dem Parameter extrahiert werden sollen.
Transaktion	Transaktion, zu der diese Abfrage gehört, sofern vorhanden.
gepuffert	Ob das Lesen der Ergebnisse der Abfrage zwischengespeichert werden soll oder nicht. Dies ist ein optionaler Parameter mit der Standardeinstellung true. Wenn gepuffert wahr ist, werden die Ergebnisse in eine `List<T>` gepuffert und dann als `IEnumerable<T>` , das für mehrere Aufzählungen sicher ist. Wenn gepuffert "false" ist, bleibt die SQL-Verbindung offen, bis Sie mit dem Lesen fertig sind, sodass Sie eine einzelne Zeile zur Zeit im Speicher bearbeiten können. Durch mehrere Aufzählungen werden zusätzliche Verbindungen zur Datenbank hergestellt. Zwar ist gepuffertes false sehr effizient, um die Speicherauslastung zu reduzieren, wenn Sie nur sehr kleine Fragmente der zurückgegebenen Datensätze verwalten, was jedoch einen erheblichen Performance-Overhead im Vergleich zu eifrigem Ergebnis bewirkt. Wenn Sie über zahlreiche gleichzeitige ungepufferte SQL-Verbindungen verfügen, müssen Sie bedenken, dass der Verbindungspool verhungert, dass Anforderungen blockiert werden, bis

Parameter	Einzelheiten
	Verbindungen verfügbar werden.
aufgeteilt auf	Das Feld, aus dem wir das zweite Objekt teilen und lesen sollen (Standard: id). Dies kann eine durch Kommas getrennte Liste sein, wenn in einem Datensatz mehr als ein Typ enthalten ist.
commandTimeout	Anzahl Sekunden vor der Befehlsausführung.
Befehlstyp	Ist es eine gespeicherte Prozedur oder eine Charge?

Examples

Einfaches Multi-Table-Mapping

Nehmen wir an, wir haben eine Abfrage der verbleibenden Reiter, die eine Person-Klasse füllen müssen.

Name	Geboren	Residenz
Daniel Dennett	1942	vereinigte Staaten von Amerika
Sam Harris	1967	vereinigte Staaten von Amerika
Richard Dawkins	1941	Großbritannien

```
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; }
}
```

Wir können sowohl die Personenklasse als auch die Residence-Eigenschaft mit einer Instanz von Country mit einer Überlast- `Query<> Func<>` , die eine `Func<>` verwendet, die zum `Func<>` der zurückgegebenen Instanz verwendet werden kann. Die `Func<>` kann bis zu 7 Eingabetypen aufnehmen, wobei das generische Argument immer der Rückgabetyp ist.

```
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");
```

Beachten Sie die Verwendung des Arguments `splitOn: "Residence"` , bei dem es sich um die 1. Spalte der nächsten zu `splitOn: "Residence"` Klassentypen handelt (in diesem Fall `Country`). Dapper sucht automatisch nach einer Spalte mit dem Namen " *Id",* nach der aufgeteilt werden soll. Wenn jedoch keine `splitOn` wird und `splitOn` nicht `System.ArgumentException` wird, wird eine `System.ArgumentException` mit einer hilfreichen Nachricht ausgelöst. Obwohl es optional ist, müssen Sie normalerweise einen `splitOn` Wert `splitOn` .

One-to-Many-Zuordnung

Schauen wir uns ein komplexeres Beispiel an, das eine Eins-zu-Viele-Beziehung enthält. Unsere Abfrage enthält jetzt mehrere Zeilen, die doppelte Daten enthalten, und wir müssen dies behandeln. Wir machen das mit einem Lookup in einem Abschluss.

Die Abfrage ändert sich geringfügig wie die Beispielklassen.

Ich würde	Name	Geboren	CountryId	Ländername	BookId	BookName
1	Daniel Dennett	1942	1	vereinigte Staaten von Amerika	1	Brainstorms
1	Daniel Dennett	1942	1	vereinigte Staaten von Amerika	2	Spielraum
2	Sam Harris	1967	1	vereinigte Staaten von Amerika	3	Die moralische Landschaft
2	Sam Harris	1967	1	vereinigte Staaten von Amerika	4	Aufwachen: Ein Leitfaden für Spiritualität ohne Religion
3	Richard Dawkins	1941	2	Großbritannien	5	Die Magie der Realität: Wie wir wissen, was wirklich wahr ist

Ich würde	Name	Geboren	CountryId	Ländername	BookId	BookName
3	Richard Dawkins	1941	2	Großbritannien	6	Appetit for Wonder: Die Entstehung eines Wissenschaftlers

```
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; }
}
```

Das `remainingHorsemen` WörterbuchHorsemen wird mit vollständig materialisierten Instanzen der Personenobjekte gefüllt. Für jede Zeile des Abfrageergebnisses werden die zugeordneten Werte von Instanzen der in den Lambda-Argumenten definierten Typen übergeben, und es liegt an Ihnen, wie Sie damit umgehen.

```
            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");
```

Beachten Sie, dass das `splitOn` Argument eine durch Kommas getrennte Liste der ersten Spalten des nächsten Typs ist.

Mapping von mehr als 7 Typen

Manchmal überschreitet die Anzahl der Typen, die Sie abbilden, die Zahl 7, die von der Funktion <> zur Verfügung gestellt wird, die die Konstruktion durchführt.

Anstelle der `Query<>` mit den generischen Typargumenteingaben werden die Typen bereitgestellt, denen ein Array zugeordnet werden soll, gefolgt von der Mapping-Funktion. Abgesehen von der anfänglichen manuellen Einstellung und dem Umwandeln der Werte ändert sich der Rest der Funktion nicht.

```
            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");
```

Benutzerdefinierte Zuordnungen

Wenn die Abfragespaltennamen nicht mit Ihren Klassen übereinstimmen, können Sie Zuordnungen für Typen einrichten. In diesem Beispiel wird die Zuordnung mit `System.Data.Linq.Mapping.ColumnAttribute` sowie eine benutzerdefinierte Zuordnung `System.Data.Linq.Mapping.ColumnAttribute` .

> Die Zuordnungen müssen nur einmal pro Typ eingerichtet werden. Setzen Sie sie also beim Start der Anwendung oder an anderer Stelle so, dass sie nur einmal initialisiert werden.

Nehmen wir wieder dieselbe Abfrage wie das One-to-Many-Beispiel an, und die Klassen werden wie folgt auf bessere Namen umgestaltet:

```
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; }
}
```

> Beachten Sie, dass sich `Book` nicht auf `ColumnAttribute` , wir müssen jedoch die `if ColumnAttribute` beibehalten

Platzieren Sie diesen Mapping-Code nun irgendwo in Ihrer Anwendung, wo er nur einmal ausgeführt wird:

```
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);
```

Dann wird die Abfrage mit einem der vorherigen Beispiele `Query<>` .

Eine einfachere Möglichkeit zum Hinzufügen der Zuordnungen wird in dieser Antwort gezeigt .

Multimapping online lesen: https://riptutorial.com/de/dapper/topic/351/multimapping

Kapitel 9: Parametersyntax-Referenz

Parameter

Parameter	Einzelheiten
`this cnn`	Die zugrunde liegende Datenbankverbindung - `this` bezeichnet eine Erweiterungsmethode; Die Verbindung muss nicht geöffnet sein - wenn sie nicht geöffnet ist, wird sie automatisch geöffnet und geschlossen.
`<T>` / `Type`	(optional) Der Typ des zurückzugebenden Objekts. Wenn die nicht generische / nicht- `Type` API verwendet wird, wird pro Zeile ein `dynamic` Objekt zurückgegeben, das eine Eigenschaft simuliert, die pro Spaltennamen benannt wird, der von der Abfrage zurückgegeben wird (dieses `dynamic` Objekt implementiert auch `IDicionary<string,object>`).
`sql`	Die auszuführende SQL
`param`	(optional) Die einzuschließenden Parameter.
`transaction`	(optional) Die dem Befehl zuzuordnende Datenbanktransaktion
`buffered`	(optional) Gibt an, ob die Daten in einer Liste vorab verbraucht werden sollen (Standardeinstellung), `IEnumerable` ein offenes `IEnumerable` über den Live-Reader `IEnumerable` machen
`commandTimeout`	(optional) Die für den Befehl zu verwendende Zeitüberschreitung. Wenn nicht angegeben, wird `SqlMapper.Settings.CommandTimeout` angenommen (falls angegeben).
`commandType`	Die Art des Befehls, der ausgeführt wird; Der Standardwert ist `CommandText`

Bemerkungen

Die Syntax zum Ausdrücken von Parametern variiert zwischen RDBMS. Alle obigen Beispiele verwenden die SQL Server-Syntax, dh `@foo` . Allerdings sollten `?foo` und `:foo` auch gut funktionieren.

Examples

Basic Parametrisierte SQL

Dapper macht es einfach, Best Practices mithilfe von vollständig parametrisiertem SQL zu befolgen.

Parameter sind wichtig, daher macht dapper es leicht, es richtig zu machen. Sie geben Ihre Parameter einfach wie üblich für Ihr RDBMS aus (normalerweise `@foo` `?foo` oder `:foo`) und geben dapper ein Objekt *mit einem Member namens* `foo` . Dies geschieht am häufigsten mit einem anonymen Typ:

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

Und das ist es. Dapper fügt die erforderlichen Parameter hinzu und alles sollte funktionieren.

Verwenden Sie Ihr Objektmodell

Sie können auch Ihr vorhandenes Objektmodell als Parameter verwenden:

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

Dapper verwendet den Befehlstext, um zu bestimmen, welche Member des Objekts hinzugefügt werden sollen. In der Regel werden keine unnötigen `IsActive` wie `Description` , `IsActive` oder `CreationDate IsActive` , da der von uns ausgegebene Befehl sie eindeutig nicht `IsActive` Dies könnte beispielsweise der Fall sein, wenn Ihr Befehl Folgendes enthält:

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

Es wird nicht versucht herauszufinden, dass das obige nur ein Kommentar ist.

Gespeicherte Prozeduren

Parameter für gespeicherte Prozeduren funktionieren genau gleich, mit der Ausnahme, dass dapper nicht feststellen kann, was nicht eingeschlossen werden soll / soll. Alle verfügbaren Daten werden als Parameter behandelt. Aus diesem Grund werden anonyme Typen normalerweise bevorzugt:

```
connection.Execute("KeyLookupInsert", new { id, name },
```

```
    commandType: CommandType.StoredProcedure);
```

Wert Inlining

Manchmal kann die Bequemlichkeit eines Parameters (in Bezug auf die Wartung und Ausdruckskraft) durch seine Kosten in der Leistung aufgewogen werden, um ihn als Parameter zu behandeln. Zum Beispiel, wenn die Seitengröße durch eine Konfigurationseinstellung festgelegt wird. Oder ein Statuswert wird mit einem `enum` abgeglichen. Erwägen:

```
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 });
```

Der einzige *echte* Parameter hier ist `customerId` - die anderen beiden sind Pseudo-Parameter, die sich nicht wirklich ändern. Häufig kann das RDBMS eine bessere Arbeit leisten, wenn es diese als Konstanten erkennt. Dapper hat hierfür eine spezielle Syntax - `{=name}` anstelle von `@name` - die *nur* für numerische Typen gilt. (Dies minimiert jegliche Angriffsfläche durch SQL-Injection). Ein Beispiel ist wie folgt:

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

Dapper ersetzt Werte durch Literale, bevor SQL ausgegeben wird. Das RDBMS sieht also Folgendes:

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

Dies ist besonders nützlich, wenn RDBMS-Systeme nicht nur bessere Entscheidungen treffen, sondern auch Abfragepläne öffnen können, die durch tatsächliche Parameter verhindert werden. Wenn zum Beispiel ein Spaltenprädikat einen Parameter betrifft, kann ein gefilterter Index mit bestimmten Werten für diese Spalten nicht verwendet werden. Dies liegt daran, dass die *nächste* Abfrage einen Parameter außer einem dieser angegebenen Werte enthalten kann.

Mit Literalwerten kann das Abfrageoptimierungsprogramm die gefilterten Indizes verwenden, da es weiß, dass sich der Wert in zukünftigen Abfragen nicht ändern kann.

Erweiterungen auflisten

Ein häufiges Szenario in Datenbankabfragen ist `IN (...)` bei dem die Liste hier zur Laufzeit generiert wird. Den meisten RDBMS fehlt dafür eine gute Metapher - und dafür gibt es keine universelle *Cross-RDBMS*- Lösung. Stattdessen sorgt dapper für eine sanfte automatische

Befehlserweiterung. Es ist `IEnumerable` ein angegebener Parameterwert erforderlich, der `IEnumerable` . Ein Befehl mit `@foo` wird zu `(@foo0,@foo1,@foo2,@foo3)` (für eine Sequenz von 4 Elementen). Die gebräuchlichste Verwendung davon wäre `IN` :

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

Dies wird dann automatisch erweitert, um die entsprechende SQL für den mehrzeiligen Abruf auszugeben:

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

Die Parameter `@orderIds0` usw. werden als Werte aus dem Arrray hinzugefügt. Beachten Sie, dass die Tatsache, dass SQL ursprünglich nicht gültig ist, beabsichtigt ist, um sicherzustellen, dass diese Funktion nicht versehentlich verwendet wird. Diese Funktion funktioniert auch korrekt mit dem `OPTIMIZE FOR` / `UNKNOWN` in SQL Server. wenn du benutzt:

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

es wird dies richtig erweitern auf:

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

Vorgänge gegen mehrere Eingabesätze ausführen

Manchmal möchten Sie dasselbe mehrmals tun. Dapper unterstützt dies für die `Execute` Methode, wenn der *äußerste* Parameter (normalerweise ein einzelner anonymer Typ oder eine Domänenmodellinstanz) tatsächlich als `IEnumerable` Sequenz `IEnumerable` wird. Zum Beispiel:

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

Hier führt dapper nur eine einfache Schleife mit unseren Daten durch, im Wesentlichen genauso, als ob wir dies getan hätten:

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

Diese Verwendung wird *besonders* interessant, wenn sie mit der `async` API für eine Verbindung

kombiniert wird, die explizit für alle "Multiple Active Results Sets" konfiguriert ist. Bei dieser Verwendung leitet *dapper* die Operationen automatisch weiter, sodass Sie die Latenzkosten pro Zeile nicht bezahlen. Dies erfordert eine etwas kompliziertere Verwendung.

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

Beachten Sie jedoch, dass Sie möglicherweise auch Tabellenwertparameter untersuchen möchten.

Pseudopositionsparameter (für Anbieter, die benannte Parameter nicht unterstützen)

Einige ADO.NET-Anbieter (vor allem: OleDB) unterstützen *benannte* Parameter nicht. Parameter werden stattdessen nur nach *Position angegeben* , mit der `?` Platzhalter. Dapper würde nicht wissen, welches Mitglied für diese verwendet werden soll, also erlaubt Dapper eine alternative Syntax, `?foo?` ; dies würde das gleiche wie sein `@foo` oder `:foo` in anderen SQL - Varianten, außer dass adrett mit dem Parameter - Token vollständig **ersetzen** `?` bevor Sie die Abfrage ausführen.

Dies funktioniert in Kombination mit anderen Funktionen wie der Listenerweiterung, daher gilt Folgendes:

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

Die Mitglieder `.region` und `.users` werden entsprechend verwendet, und die ausgegebene SQL lautet (z. B. mit 3 Benutzern):

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

Beachten Sie jedoch, dass **sich** adrett **nicht** der gleiche Parameter erlauben mehrfach verwendet werden , wenn diese Funktion verwendet wird ; Dies verhindert, dass derselbe Parameterwert (der groß sein kann) mehrmals hinzugefügt werden muss. Wenn Sie mehrmals auf denselben Wert verweisen müssen, sollten Sie eine Variable deklarieren. Beispiel:

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

Wenn keine Variablen verfügbar sind, können Sie doppelte Elementnamen in den Parametern verwenden. Dies macht auch deutlich, dass der Wert mehrmals gesendet wird:

```
int id = 42;
```

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

Parametersyntax-Referenz online lesen:
https://riptutorial.com/de/dapper/topic/10/parametersyntax-referenz

Kapitel 10: Temp-Tabellen

Examples

Temp Tabelle, die vorhanden ist, während die Verbindung offen bleibt

Wenn die temporäre Tabelle von selbst erstellt wird, bleibt sie erhalten, während die Verbindung geöffnet ist.

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

So arbeiten Sie mit temporären Tabellen

Der Punkt bei temporären Tabellen ist, dass sie auf den Umfang der Verbindung beschränkt sind. Dapper öffnet und schließt automatisch eine Verbindung, wenn diese noch nicht geöffnet ist. Das bedeutet, dass eine temporäre Tabelle direkt nach der Erstellung verloren geht, wenn die an Dapper übergebene Verbindung nicht geöffnet wurde.

Das wird nicht funktionieren:

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

Auf der anderen Seite funktionieren diese beiden Versionen:

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

Temp-Tabellen online lesen: https://riptutorial.com/de/dapper/topic/6594/temp-tabellen

Kapitel 11: Transaktionen

Syntax

- conn.Execute (sql, transaction: tran); // Den Parameter anhand des Namens angeben
- conn.Execute (sql, parameters, tran);
- conn.Query (SQL, Transaktion: Tran);
- conn.Query (SQL, Parameter, Tran);
- waitit conn.ExecuteAsync (sql, transaction: tran); // Async
- waitit conn.ExecuteAsync (sql, parameters, tran);
- Erwarten Sie conn.QueryAsync (SQL, Transaktion: Tran);
- Erwarten Sie conn.QueryAsync (SQL, Parameter, Tran);

Examples

Eine Transaktion verwenden

In diesem Beispiel wird SqlConnection verwendet, jedoch wird jede IDbConnection unterstützt.

Auch jede IDbTransaction wird von der zugehörigen IDbConnection unterstützt.

```
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;
            }
        }
    }
}
```

Einsätze beschleunigen

Das Einschließen einer Gruppe von Einfügungen in eine Transaktion beschleunigt diese entsprechend dieser StackOverflow-Frage / Antwort .

Sie können diese Technik verwenden, oder Sie können die Massenkopie verwenden, um eine Reihe von verwandten Vorgängen zu beschleunigen.

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

Transaktionen online lesen: https://riptutorial.com/de/dapper/topic/6601/transaktionen

Kapitel 12: Typ-Handler

Bemerkungen

Type Handlers ermöglichen die Konvertierung von Datenbanktypen in benutzerdefinierte .NET-Typen.

Examples

Konvertierung von 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());
    }
}
```

TypeHandler installieren

Der obige Typ-Handler kann mithilfe der `AddTypeHandler` Methode in `SqlMapper` `AddTypeHandler` werden.

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

Mit der Typinferenz können Sie den generischen Typparameter weglassen:

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

Es gibt auch eine Überladung mit zwei Argumenten, für die ein explizites `Type` erforderlich ist:

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

Typ-Handler online lesen: https://riptutorial.com/de/dapper/topic/6/typ-handler

Kapitel 13: Umgang mit Nullen

Examples

null vs DBNull

In ADO.NET ist die korrekte Handhabung von `null` eine ständige Quelle der Verwirrung. Der Schlüsselpunkt in dapper ist, dass *Sie nicht müssen* ; es befasst sich damit intern.

- Parameterwerte, die `null` sind , korrekt als gesendet `DBNull.Value`
- Gelesene Werte, die `null` sind, werden als `null` oder (bei der Zuordnung zu einem bekannten Typ) einfach ignoriert (wobei der typbasierte Standard beibehalten wird)

Es funktioniert einfach:

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

Umgang mit Nullen online lesen: https://riptutorial.com/de/dapper/topic/13/umgang-mit-nullen

Kapitel 14: Verwenden von DbGeography und DbGeometry

Examples

Konfiguration erforderlich

1. Installieren Sie die erforderliche `Microsoft.SqlServer.Types` Assembly. Sie werden standardmäßig nicht installiert und sind hier von Microsoft als "Microsoft® System CLR-Typen für Microsoft® SQL Server® 2012" verfügbar. Beachten Sie, dass für x86 und x64 separate Installationsprogramme vorhanden sind.

2. installiere `Dapper.EntityFramework` (oder das Äquivalent mit dem starken Namen); Dies kann über die Benutzeroberfläche der IDE "NuGet-Pakete verwalten ..." oder (in der Package Manager Console) erfolgen:

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

3. Fügen Sie die erforderlichen Assembly-Bindungs-Weiterleitungen hinzu. Dies liegt daran, dass Microsoft v11 der Assemblys ausliefert, Entity Framework jedoch nach v10 fragt. Sie können `app.config` oder `web.config` unter dem Element `<configuration>` Folgendes hinzufügen:

```
<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. Informieren Sie "dapper" über die neuen verfügbaren Typhandler, indem Sie Folgendes hinzufügen (irgendwo in Ihrem Startup, bevor es versucht, die Datenbank zu verwenden):

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

Geometrie und Geographie verwenden

Sobald die Typhandler registriert sind, sollte alles automatisch funktionieren, und Sie sollten diese Typen entweder als Parameter oder als Rückgabewerte verwenden können:

```
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}");
```

Verwenden von DbGeography und DbGeometry online lesen:
https://riptutorial.com/de/dapper/topic/3984/verwenden-von-dbgeography-und-dbgeometry

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit Dapper.NET	Adam Lear, balpha, Community, Eliza, Greg Bray, Jarrod Dixon, Kevin Montrose, Matt McCabe, Nick, Rob, Shog9
2	Async verwenden	Dean Ward, Matt McCabe, Nick, Woodchipper
3	Befehle ausführen	Adam Lear, Jarrod Dixon, Sklivvz, takrl
4	Bulk-Einsätze	jhamm
5	Dynamische Parameter	Marc Gravell, Matt McCabe, Meer
6	Grundlegendes Abfragen	Adam Lear, Chris Marisic, Cigano Morrison Mendez, Community, cubrr, Jarrod Dixon, jrummell, Kevin Montrose, Matt McCabe
7	Mehrere Ergebnisse	Marc Gravell, Yaakov Ellis
8	Multimapping	Devon Burriss
9	Parametersyntax-Referenz	4444, Marc Gravell, Nick Craver
10	Temp-Tabellen	jhamm, Rob, takrl
11	Transaktionen	jhamm
12	Typ-Handler	Benjamin Hodgson, Community, Marc Gravell
13	Umgang mit Nullen	Marc Gravell
14	Verwenden von DbGeography und DbGeometry	Marc Gravell