目錄

Dapper

> 原文：Dapper Tutorial ，译者：Esofar

什么是Dapper

Dapper是一个简单的.NET对象映射器，在速度方面具有**"King of Micro ORM"**的头衔，几乎与使用原始的ADO.NET数据读取器一样快。ORM是一个对象关系映射器，它负责数据库和编程语言之间的映射。

Dapper通过扩展 `IDbConnection` 提供一些有用的扩展方法去查询您的数据库。

Dapper是如何工作的

它可以分为三个步骤：

- 创建一个 `IDbConnection` 接口对象；
- 编写一个查询SQL来执行CRUD操作；
- 将查询SQL作为 `Execute` 方法的参数传递。

安装

Dapper通过NuGet安装：https://www.nuget.org/packages/Dapper

```
PM> Install-Package Dapper
```

要求

Dapper可以与任何数据库提供者一起工作，因为没有数据库特定的实现。

方法

Dapper会用以下几个方法扩展您的 `IDbConnection` 接口：

- Execute
- Query
- QueryFirst
- QueryFirstOrDefault
- QuerySingle
- QuerySingleOrDefault
- QueryMultiple

```
string sqlInvoices = "SELECT * FROM Invoice;";
string sqlInvoice = "SELECT * FROM Invoice WHERE InvoiceID = @In
voiceID;";
string sp = "EXEC Invoice_Insert";

using (var connection = My.ConnectionFactory())
{
        // 执行普通SQL
    var invoices = connection.Query<Invoice>(sqlInvoices).ToList
();
    // 执行带参数的SQL
    var invoice = connection.QueryFirstOrDefault(sqlInvoice, new
 {InvoiceID = 1});
    // 执行存储过程
    var affectedRows = connection.Execute(sp, new { Param1 = "Si
ngle_Insert_1" }, commandType: CommandType.StoredProcedure);
}
```

参数

执行和查询方法可以用以下几种不同的方式使用参数：

- 匿名类型
- 动态类型
- 列表类型
- 字符串类型

```
// Anonymous
var affectedRows = connection.Execute(sql,
                    new {Kind = InvoiceKind.WebInvoice, Code = "
Single_Insert_1"},
                    commandType: CommandType.StoredProcedure);

// Dynamic
DynamicParameters parameter = new DynamicParameters();

parameter.Add("@Kind", InvoiceKind.WebInvoice, DbType.Int32, Par
ameterDirection.Input);
parameter.Add("@Code", "Many_Insert_0", DbType.String, Parameter
Direction.Input);
parameter.Add("@RowCount", dbType: DbType.Int32, direction: Para
meterDirection.ReturnValue);

connection.Execute(sql,
    new {Kind = InvoiceKind.WebInvoice, Code = "Single_Insert_1"
},
    commandType: CommandType.StoredProcedure);

// List
connection.Query<Invoice>(sql, new {Kind = new[] {InvoiceKind.St
oreInvoice, InvoiceKind.WebInvoice}}).ToList();

// String
connection.Query<Invoice>(sql, new {Code = new DbString {Value =
 "Invoice_1", IsFixedLength = false, Length = 9, IsAnsi = true}}
).ToList();
```

结果

查询方法返回的结果可以映射到以下几种类型：

- 匿名类型
- 强类型
- 多映射
- 多结果

- 多类型

```
string sql = "SELECT * FROM Invoice;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var anonymousList = connection.Query(sql).ToList();
    var invoices = connection.Query<Invoice>(sql).ToList();
}
```

工具

- 异步
- 缓冲
- 事务
- 存储过程

```
// Async
connection.QueryAsync<Invoice>(sql)

// Buffered
connection.Query<Invoice>(sql, buffered: false)

// Transaction
using (var transaction = connection.BeginTransaction())
{
    var affectedRows = connection.Execute(sql,
        new {Kind = InvoiceKind.WebInvoice, Code = "Single_Inser
t_1"},
        commandType: CommandType.StoredProcedure,
        transaction: transaction);

    transaction.Commit();
}

// Stored Procedure
var affectedRows = connection.Execute(sql,
    new {Kind = InvoiceKind.WebInvoice, Code = "Single_Insert_1"
},
    commandType: CommandType.StoredProcedure);
```

Dapper - Execute

描述

`Execute` 是一个可以从 `IDbConnection` 类型的任意对象调用的扩展方法，它可以执行一个或多个命令并返回受影响的行数。此方法通常用于执行：

- 存储过程
- INSERT语句
- UPDATE语句
- DELETE语句

参数

下表显示了 `Execute` 方法的不同参数。

名称	描述
sql	要执行的命令文本。
param	命令参数（默认为 `null` ）。
transaction	需要使用的事务（默认为 `null` ）。
commandTimeout	命令执行超时时间（默认为 `null` ）。
commandType	命令类型（默认为 `null` ）。

案例 - 执行存储过程

单次

执行一次存储过程。

```
string sql = "Invoice_Insert";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var affectedRows = connection.Execute(sql,
        new {Kind = InvoiceKind.WebInvoice, Code = "Single_Inser
t_1"},
        commandType: CommandType.StoredProcedure);

    My.Result.Show(affectedRows);
}
```

多次

执行多次存储过程，为参数数组列表中的每个对象执行一次。

```
string sql = "Invoice_Insert";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var affectedRows = connection.Execute(sql,
        new[]
        {
            new {Kind = InvoiceKind.WebInvoice, Code = "Many_Ins
ert_1"},
            new {Kind = InvoiceKind.WebInvoice, Code = "Many_Ins
ert_2"},
            new {Kind = InvoiceKind.StoreInvoice, Code = "Many_I
nsert_3"}
        },
        commandType: CommandType.StoredProcedure
    );

    My.Result.Show(affectedRows);
}
```

案例 - 执行INSERT语句

单次

执行一次 `INSERT` 语句。

```
string sql = "INSERT INTO Invoice (Code) Values (@Code);";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var affectedRows = connection.Execute(sql, new {Kind = Invoi
ceKind.WebInvoice, Code = "Single_Insert_1"});

    My.Result.Show(affectedRows);
}
```

多次

执行 `INSERT` 语句，为参数数组列表中的每个对象执行一次。

```
string sql = "INSERT INTO Invoice (Code) Values (@Code);";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var affectedRows = connection.Execute(sql,
        new[]
        {
            new {Kind = InvoiceKind.WebInvoice, Code = "Many_Ins
ert_1"},
            new {Kind = InvoiceKind.WebInvoice, Code = "Many_Ins
ert_2"},
            new {Kind = InvoiceKind.StoreInvoice, Code = "Many_I
nsert_3"}
        }
    );

    My.Result.Show(affectedRows);
}
```

案例 - 执行UPDATE语句

单次

执行一次 `UPDATE` 语句。

```
string sql = "INSERT INTO Invoice (Code) Values (@Code);";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var affectedRows = connection.Execute(sql, new {Kind = Invoi
ceKind.WebInvoice, Code = "Single_Insert_1"});

    My.Result.Show(affectedRows);
}
```

多次

执行 `UPDATE` 语句，为参数数组列表中的每个对象执行一次。

```
string sql = "UPDATE Invoice SET Code = @Code WHERE InvoiceID =
@InvoiceID";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var affectedRows = connection.Execute(sql,
        new[]
        {
            new {InvoiceID = 1, Code = "Many_Update_1"},
            new {InvoiceID = 2, Code = "Many_Update_2"},
            new {InvoiceID = 3, Code = "Many_Update_3"}
        });

    My.Result.Show(affectedRows);
}
```

案例 - 执行DELETE语句

单次

执行一次 `DELETE` 语句。

```
string sql = "DELETE FROM Invoice WHERE InvoiceID = @InvoiceID";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var affectedRows = connection.Execute(sql, new {InvoiceID =
1});

    My.Result.Show(affectedRows);
}
```

多次

执行 `DELETE` 语句，为参数数组列表中的每个对象执行一次。

```
string sql = "DELETE FROM Invoice WHERE InvoiceID = @InvoiceID";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var affectedRows = connection.Execute(sql,
        new[]
        {
            new {InvoiceID = 1},
            new {InvoiceID = 2},
            new {InvoiceID = 3}
        });
}
```

Dapper - Query

描述

`Query` 是一个可以从 `IDbConnection` 类型的任意对象调用的扩展方法，它可以执行查询并映射结果。

结果可以映射到：

- 匿名类型
- 强类型
- 多映射（一对一）
- 多映射（一对多）
- 多类型

参数

下表显示了 `Query` 方法的不同参数。

名称	描述
sql	要执行的查询。
param	查询参数（默认为 `null` ）。
transaction	需要使用的事务（默认为 `null` ）。
buffered	是否从缓冲读取查询结果（默认为 `true` ）。
commandTimeout	命令执行超时时间（默认为 `null` ）。
commandType	命令类型（默认为 `null` ）。

案例 - 查询匿名类型

原生SQL查询可以使用 `Query` 方法执行，并将结果映射到动态类型列表。

```
string sql = "SELECT * FROM Invoice;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoices = connection.Query(sql).ToList();

    My.Result.Show(invoices);

    //输出：3 anonymous entity returned
}
```

案例 - 查询强类型

原生SQL查询可以使用 `Query` 方法执行，并将结果映射到强类型列表。

```
string sql = "SELECT * FROM Invoice;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoices = connection.Query<Invoice>(sql).ToList();

    My.Result.Show(invoices);

    //输出：3 invoice(s) returned
}
```

案例 - 查询多映射（一对一）

原生SQL查询可以使用 `Query` 方法执行，并将结果映射到具有一对一关系的强类型列表。

```
string sql = "SELECT * FROM Invoice AS A INNER JOIN InvoiceDetai
l AS B ON A.InvoiceID = B.InvoiceID;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoices = connection.Query<Invoice, InvoiceDetail, Invo
ice>(
            sql,
            (invoice, invoiceDetail) =>
            {
                invoice.InvoiceDetail = invoiceDetail;
                return invoice;
            },
            splitOn: "InvoiceID")
        .Distinct()
        .ToList();

    My.Result.Show(invoices);

    //输出：3 invoice(s) returned (Including InvoiceDetail inform
ation)
}
```

案例 - 查询多映射（一对多）

原生SQL查询可以使用 `Query` 方法执行，并将结果映射到具有一对多关系的强类型列表。

```
string sql = "SELECT * FROM Invoice AS A INNER JOIN InvoiceItem
AS B ON A.InvoiceID = B.InvoiceID;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoiceDictionary = new Dictionary<int, Invoice>();

    var invoices = connection.Query<Invoice, InvoiceItem, Invoic
e>(
            sql,
            (invoice, invoiceItem) =>
            {
                Invoice invoiceEntry;

                if (!invoiceDictionary.TryGetValue(invoice.Invoi
ceID, out invoiceEntry))
                {
                    invoiceEntry = invoice;
                    invoiceEntry.Items = new List<InvoiceItem>()
;
                    invoiceDictionary.Add(invoiceEntry.InvoiceID
, invoiceEntry);
                }

                invoiceEntry.Items.Add(invoiceItem);
                return invoiceEntry;
            },
            splitOn: "InvoiceID")
        .Distinct()
        .ToList();

    My.Result.Show(invoices);

    //输出：3 invoice(s) returned (Including 6 InvoiceItem)
}
```

案例 - 查询多类型

原生SQL查询可以使用 `Query` 方法执行，并将结果映射到不同类型的列表。

```
string sql = "SELECT * FROM Invoice;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoices = new List<Invoice>();

    using (var reader = connection.ExecuteReader(sql))
    {
        var storeInvoiceParser = reader.GetRowParser<StoreInvoic
e>();
        var webInvoiceParser = reader.GetRowParser<WebInvoice>()
;

        while (reader.Read())
        {
            Invoice invoice;

            switch ((InvoiceKind) reader.GetInt32(reader.GetOrdi
nal("Kind")))
            {
                case InvoiceKind.StoreInvoice:
                    invoice = storeInvoiceParser(reader);
                    break;
                case InvoiceKind.WebInvoice:
                    invoice = webInvoiceParser(reader);
                    break;
                default:
                    throw new Exception(ExceptionMessage.General
Exception);
            }

            invoices.Add(invoice);
        }
```

```
    }

    My.Result.Show(invoices);

    //输出：3 invoice(s) returned (StoreInvoice:1, WebInvoice:2)
}
```

Dapper - QueryFirst

描述

`QueryFirst` 是一个可以从 `IDbConnection` 类型的任意对象调用的扩展方法，它可以执行查询并映射第一个结果。

结果可以映射到：

- 匿名类型
- 强类型

参数

下表显示了 `QueryFirst` 方法的不同参数。

名称	描述
sql	要执行的查询。
param	查询参数（默认为 `null` ）。
transaction	需要使用的事务（默认为 `null` ）。
commandTimeout	命令执行超时时间（默认为 `null` ）。
commandType	命令类型（默认为 `null` ）。

First, Single & Default

注意使用正确的方法。 `First` 和 `Single` 的方法是非常不同的。

结果	没有项	有一项	有多项
First	抛异常	当前项	第一项
Single	抛异常	当前项	抛异常
FirstOrDefault	默认值	当前项	第一项
SingleOrDefault	默认值	当前项	抛异常

案例 - 查询匿名类型

执行查询并将第一个结果映射到动态类型列表。

```
string sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID
;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QueryFirst(sql, new {InvoiceID = 1}
);
}
```

案例 - 查询强类型

执行查询并将第一个结果映射到强类型列表。

```
string sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID
;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QueryFirst<Invoice>(sql, new {Invoi
ceID = 1});
}
```

Dapper - QueryFirstOrDefault

描述

`QueryFirstOrDefault` 是一个可以从 `IDbConnection` 类型的任意对象调用的扩展方法，它可以执行查询并映射第一个结果，如果序列不包含任何元素则为默认值。

结果可以映射到：

- 匿名类型
- 强类型

参数

下表显示了 `QueryFirstOrDefault` 方法的不同参数。

名称	描述
sql	要执行的查询。
param	查询参数（默认为 `null` ）。
transaction	需要使用的事务（默认为 `null` ）。
commandTimeout	命令执行超时时间（默认为 `null` ）。
commandType	命令类型（默认为 `null` ）。

First, Single & Default

注意使用正确的方法。 `First` 和 `Single` 的方法是非常不同的。

结果	没有项	有一项	有多项
First	抛异常	当前项	第一项
Single	抛异常	当前项	抛异常
FirstOrDefault	默认值	当前项	第一项
SingleOrDefault	默认值	当前项	抛异常

案例 - 查询匿名类型

执行查询并将第一个结果映射到动态类型列表，如果序列不包含任何元素则为默认值。

```
string sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID
;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QueryFirstOrDefault(sql, new {Invoi
ceID = 1});
}
```

案例 - 查询强类型

执行查询并将第一个结果映射到强类型列表，如果序列不包含任何元素则为默认值。

```
string sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID
;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QueryFirstOrDefault<Invoice>(sql, n
ew {InvoiceID = 1});
}
```

Dapper - QuerySingle

描述

`QuerySingle` 是一个可以从 `IDbConnection` 类型的任意对象调用的扩展方法，它可以执行查询并映射第一个结果，如果序列中没有元素则会引发异常。

结果可以映射到：

- 匿名类型
- 强类型

参数

下表显示了 `QuerySingle` 方法的不同参数。

名称	描述
sql	要执行的查询。
param	查询参数（默认为 `null` ）。
transaction	需要使用的事务（默认为 `null` ）。
commandTimeout	命令执行超时时间（默认为 `null` ）。
commandType	命令类型（默认为 `null` ）。

First, Single & Default

注意使用正确的方法。 `First` 和 `Single` 的方法是非常不同的。

结果	没有项	有一项	有多项
First	抛异常	当前项	第一项
Single	抛异常	当前项	抛异常
FirstOrDefault	默认值	当前项	第一项
SingleOrDefault	默认值	当前项	抛异常

案例 - 查询匿名类型

执行查询并将第一个结果映射到动态类型列表，如果序列中没有元素则会引发异常。

```
string sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID
;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QuerySingle(sql, new {InvoiceID = 1
});
}
```

案例 - 查询强类型

执行查询并将第一个结果映射到强类型列表，如果序列中没有元素则会引发异常。

```
string sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID
;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QuerySingle<Invoice>(sql, new {Invo
iceID = 1});
}
```

Dapper - QuerySingleOrDefault

描述

`QuerySingleOrDefault` 是一个可以从 `IDbConnection` 类型的任意对象调用的扩展方法，它可以执行查询并映射第一个结果，如果序列为空则为默认值。如果序列中有多个元素，则此方法将引发异常。

结果可以映射到：

- 匿名类型
- 强类型

参数

下表显示了 `QuerySingleOrDefault` 方法的不同参数。

名称	描述
sql	要执行的查询。
param	查询参数（默认为 `null` ）。
transaction	需要使用的事务（默认为 `null` ）。
commandTimeout	命令执行超时时间（默认为 `null` ）。
commandType	命令类型（默认为 `null` ）。

First, Single & Default

注意使用正确的方法。 `First` 和 `Single` 的方法是非常不同的。

结果	没有项	有一项	有多项
First	抛异常	当前项	第一项
Single	抛异常	当前项	抛异常
FirstOrDefault	默认值	当前项	第一项
SingleOrDefault	默认值	当前项	抛异常

案例 - 查询匿名类型

执行查询并将第一个结果映射到动态类型列表，如果序列为空则为默认值。如果序列中有多个元素，则此方法将引发异常。

```
string sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID
;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QuerySingleOrDefault(sql, new {Invo
iceID = 1});
}
```

案例 - 查询强类型

执行查询并将第一个结果映射到强类型列表，如果序列为空则为默认值。如果序列中有多个元素，则此方法将引发异常。

```
string sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID
;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QuerySingleOrDefault<Invoice>(sql,
new {InvoiceID = 1});
}
```

Dapper - QueryMultiple

描述

`QueryMultiple` 是一个可以从 `IDbConnection` 类型的任意对象调用的扩展方法，它可以在相同的命令和映射结果中执行多个查询。

```
string sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID
; SELECT * FROM InvoiceItem WHERE InvoiceID = @InvoiceID;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    using (var multi = connection.QueryMultiple(sql, new {Invoic
eID = 1}))
    {
        var invoice = multi.Read<Invoice>().First();
        var invoiceItems = multi.Read<InvoiceItem>().ToList();
    }
}
```

参数

下表显示了 `QueryMultiple` 方法的不同参数。

名称	描述
sql	要执行的查询。
param	查询参数（默认为 `null` ）。
transaction	需要使用的事务（默认为 `null` ）。
commandTimeout	命令执行超时时间（默认为 `null` ）。
commandType	命令类型（默认为 `null` ）。

Dapper - 匿名类型参数

描述

Dapper通过支持匿名类型使其可以简单、安全（SQL注入）的使用参数。

单次

执行一次SQL命令。

```
var sql = "EXEC Invoice_Insert";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var affectedRows = connection.Execute(sql,
        new {Kind = InvoiceKind.WebInvoice, Code = "Single_Inser
t_1"},
        commandType: CommandType.StoredProcedure);

    My.Result.Show(affectedRows);
}
```

多次

执行多次SQL命令。

```
var sql = "EXEC Invoice_Insert";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var affectedRows = connection.Execute(sql,
        new[]
        {
            new {Kind = InvoiceKind.WebInvoice, Code = "Many_Ins
ert_1"},
            new {Kind = InvoiceKind.WebInvoice, Code = "Many_Ins
ert_2"},
            new {Kind = InvoiceKind.StoreInvoice, Code = "Many_I
nsert_3"}
        },
        commandType: CommandType.StoredProcedure
    );
}
```

Dapper - 动态类型参数

描述

在Dapper方法中创建并使用参数。

单次

执行一次SQL命令。

```
var sql = "EXEC Invoice_Insert";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    DynamicParameters parameter = new DynamicParameters();

    parameter.Add("@Kind", InvoiceKind.WebInvoice, DbType.Int32,
 ParameterDirection.Input);
    parameter.Add("@Code", "Many_Insert_0", DbType.String, Param
eterDirection.Input);
    parameter.Add("@RowCount", dbType: DbType.Int32, direction:
ParameterDirection.ReturnValue);

    connection.Execute(sql,
        parameter,
        commandType: CommandType.StoredProcedure);

    int rowCount = parameter.Get<int>("@RowCount");
}
```

多次

执行多次SQL命令。

```
var sql = "EXEC Invoice_Insert";

var parameters = new List<DynamicParameters>();

for (var i = 0; i < 3; i++)
{
    var p = new DynamicParameters();
    p.Add("@Kind", InvoiceKind.WebInvoice, DbType.Int32, Paramet
erDirection.Input);
    p.Add("@Code", "Many_Insert_" + (i + 1), DbType.String, Para
meterDirection.Input);
    p.Add("@RowCount", dbType: DbType.Int32, direction: Paramete
rDirection.ReturnValue);

    parameters.Add(p);
}

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    connection.Execute(sql,
        parameters,
        commandType: CommandType.StoredProcedure
    );

    var rowCount = parameters.Sum(x => x.Get<int>("@RowCount"));
}
```

Dapper - 列表类型参数

描述

Dapper允许您使用列表在IN子句中指定多个参数。

```
var sql = "SELECT * FROM Invoice WHERE Kind IN @Kind;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoices = connection.Query<Invoice>(sql, new {Kind = ne
w[] {InvoiceKind.StoreInvoice, InvoiceKind.WebInvoice}}).ToList(
);
}
```

Dapper - 字符串类型参数

描述

```
var sql = "SELECT * FROM Invoice WHERE Code = @Code;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoices = connection.Query<Invoice>(sql, new {Code = ne
w DbString {Value = "Invoice_1", IsFixedLength = false, Length =
 9, IsAnsi = true}}).ToList();

    My.Result.Show(invoices);
}
```

Dapper - 结果匿名

描述

可以使用扩展方法执行查询并使用动态类型映射结果。

匿名类型结果可以从以下扩展方法映射：

- Query
- QueryFirst
- QueryFirstOrDefault
- QuerySingle
- QuerySingleOrDefault

这些扩展方法可以从IDbConnection类型的任意对象中调用。

案例 - Query

`Query` 方法可以执行查询并将结果映射到动态类型列表。

```
string sql = "SELECT * FROM Invoice;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoices = connection.Query(sql).ToList();
}
```

案例 - QueryFirst

`QueryFirst` 方法可以执行查询并将第一个结果映射到动态类型列表。

```
string sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID
;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QueryFirst(sql, new {InvoiceID = 1}
);
}
```

案例 - QueryFirstOrDefault

`QueryFirstOrDefault` 方法可以执行查询并将第一个结果映射到动态类型列表，如果序列不包含任何元素则为默认值。

```
string sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID
;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QueryFirstOrDefault(sql, new {Invoi
ceID = 1});
}
```

案例 - QuerySingle

`QuerySingle` 方法可以执行查询并将第一个结果映射到动态类型列表，如果序列中没有元素则会引发异常。

```
string sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID
;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QuerySingle(sql, new {InvoiceID = 1
});
}
```

案例 - QuerySingleOrDefault

`QuerySingleOrDefault` 方法可以执行查询并将第一个结果映射到动态类型列表，如果序列为空则为默认值；如果序列中有多个元素，则此方法将引发异常。

```
string sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID
;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QuerySingleOrDefault(sql, new {Invo
iceID = 1});
}
```

Dapper - 结果强类型

描述

可以使用扩展方法执行查询并使用动态类型映射结果。

强类型结果可以从以下扩展方法映射：

- Query
- QueryFirst
- QueryFirstOrDefault
- QuerySingle
- QuerySingleOrDefault

这些扩展方法可以从IDbConnection类型的任意对象中调用。

案例 - Query

`Query` 方法可以执行查询并将结果映射到强类型列表。

```
string sql = "SELECT * FROM Invoice;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoices = connection.Query<Invoice>(sql).ToList();
}
```

案例 - QueryFirst

`QueryFirst` 方法可以执行查询并将第一个结果映射到强类型列表。

```
string sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID
;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QueryFirst<Invoice>(sql, new {Invoi
ceID = 1});
}
```

案例 - QueryFirstOrDefault

`QueryFirstOrDefault` 方法可以执行查询并将第一个结果映射到强类型列表，如果序列不包含任何元素则为默认值。

```
string sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID
;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QueryFirstOrDefault<Invoice>(sql, n
ew {InvoiceID = 1});
}
```

案例 - QuerySingle

`QuerySingle` 方法可以执行查询并将第一个结果映射到强类型列表，如果序列中没有元素则会引发异常。

```
string sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID
;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QuerySingle<Invoice>(sql, new {Invo
iceID = 1});
}
```

案例 - QuerySingleOrDefault

`QuerySingleOrDefault` 方法可以执行查询并将第一个结果映射到强类型列表，如果序列为空则为默认值；如果序列中有多个元素，则此方法将引发异常。

```
string sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID
;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QuerySingleOrDefault<Invoice>(sql,
new {InvoiceID = 1});
}
```

Dapper - 结果多映射

描述

可以使用扩展方法执行查询并将结果映射到具有关系的强类型列表。

关系可以是：

- 一对一
- 一对多

这些扩展方法可以从IDbConnection类型的任意对象中调用。

案例 - 查询多映射（一对一）

`Query` 方法可以执行查询并将结果映射到具有一对一关系的强类型列表。

```
string sql = "SELECT * FROM Invoice AS A INNER JOIN InvoiceDetai
l AS B ON A.InvoiceID = B.InvoiceID;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoices = connection.Query<Invoice, InvoiceDetail, Invo
ice>(
            sql,
            (invoice, invoiceDetail) =>
            {
                invoice.InvoiceDetail = invoiceDetail;
                return invoice;
            },
            splitOn: "InvoiceID")
        .Distinct()
        .ToList();
}
```

案例 - 查询多映射（一对多）

`Query` 方法可以执行查询并将结果映射到具有一对多关系的强类型列表。

```
string sql = "SELECT * FROM Invoice AS A INNER JOIN InvoiceItem
AS B ON A.InvoiceID = B.InvoiceID;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoiceDictionary = new Dictionary<int, Invoice>();

    var invoices = connection.Query<Invoice, InvoiceItem, Invoic
e>(
            sql,
            (invoice, invoiceItem) =>
            {
                Invoice invoiceEntry;

                if (!invoiceDictionary.TryGetValue(invoice.Invoi
ceID, out invoiceEntry))
                {
                    invoiceEntry = invoice;
                    invoiceEntry.Items = new List<InvoiceItem>()
;
                    invoiceDictionary.Add(invoiceEntry.InvoiceID
, invoiceEntry);
                }

                invoiceEntry.Items.Add(invoiceItem);
                return invoiceEntry;
            },
            splitOn: "InvoiceID")
        .Distinct()
        .ToList();
}
```

Dapper - 结果多结果

描述

`QueryMultiple` 是一个扩展方法，可以从IDbConnection类型的任意对象中调用。它可以在同一命令中执行多个查询并映射结果。

```
string sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID
; SELECT * FROM InvoiceItem WHERE InvoiceID = @InvoiceID;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    using (var multi = connection.QueryMultiple(sql, new {Invoic
eID = 1}))
    {
        var invoice = multi.Read<Invoice>().First();
        var invoiceItems = multi.Read<InvoiceItem>().ToList();
    }
}
```

Dapper - 结果多类型

描述

`ExecuteReader` 是一个扩展方法，可以从IDbConnection类型的任意对象中调用。它可以执行查询并将结果映射到不同类型的列表。

```
string sql = "SELECT * FROM Invoice;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoices = new List<Invoice>();

    using (var reader = connection.ExecuteReader(sql))
    {
        var storeInvoiceParser = reader.GetRowParser<StoreInvoic
e>();
        var webInvoiceParser = reader.GetRowParser<WebInvoice>()
;

        while (reader.Read())
        {
            Invoice invoice;

            switch ((InvoiceKind) reader.GetInt32(reader.GetOrdi
nal("Kind")))
            {
                case InvoiceKind.StoreInvoice:
                    invoice = storeInvoiceParser(reader);
                    break;
                case InvoiceKind.WebInvoice:
                    invoice = webInvoiceParser(reader);
                    break;
                default:
                    throw new Exception(ExceptionMessage.General
Exception);
            }

            invoices.Add(invoice);
        }
    }
}
```

Dapper - 异步

描述

Dapper还使用了Async（异步）方法扩展了 `IDbConnection` 接口：

- ExecuteAsync
- QueryAsync
- QueryFirstAsync
- QueryFirstOrDefaultAsync
- QuerySingleAsync
- QuerySingleOrDefaultAsync
- QueryMultipleAsync

我们只在本教程中添加了非异步版本，以便于阅读。

ExecuteAsync

```
var sql = "Invoice_Insert";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var affectedRows = connection.ExecuteAsync(sql,
            new {Kind = InvoiceKind.WebInvoice, Code = "Single_I
nsert_1"},
            commandType: CommandType.StoredProcedure)
        .Result;
}
```

QueryAsync

```
var sql = "Invoice_Insert";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoices = connection.QueryAsync<Invoice>(sql).Result.To
List();
}
```

QueryFirstAsync

```
var sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QueryFirstAsync<Invoice>(sql, new {
InvoiceID = 1}).Result;
}
```

QueryFirstOrDefaultAsync

```
var sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QueryFirstOrDefaultAsync<Invoice>(s
ql, new {InvoiceID = 1}).Result;
}
```

QuerySingleAsync

```
var sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QuerySingleAsync<Invoice>(sql, new
{InvoiceID = 1}).Result;
}
```

QuerySingleOrDefaultAsync

```
var sql = "SELECT * FROM Invoice WHERE InvoiceID = @InvoiceID;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var invoice = connection.QuerySingleOrDefaultAsync<Invoice>(
sql, new {InvoiceID = 1}).Result;
}
```

QueryMultipleAsync

```
var sql = "SELECT * FROM Invoice; SELECT * FROM InvoiceItem;";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    using (var multi = connection.QueryMultipleAsync(sql, new {
InvoiceID = 1 }).Result)
    {
        var invoice = multi.Read<Invoice>().First();
        var invoiceItems = multi.Read<InvoiceItem>().ToList();
    }
}
```

Dapper - 缓冲

描述

- 默认值：True

缓冲查询一次返回整个读取器，这在大多数情况下是理想的。

非缓冲查询与流式传输等效，您只需按需加载对象，这对于一个非常大的查询来减少内存使用情况可能很有用。

```
string sqlInvoices = "SELECT * FROM Invoice;";

using (var connection = My.ConnectionFactory())
{
    var invoices = connection.Query<Invoice>(sqlInvoices, buffer
ed: false).ToList();
}
```

Dapper - 事务

描述

Dapper支持事务和范围事务。

事务

从连接开始一个新事务，并将其传递给事务可选参数。

```
var sql = "Invoice_Insert";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    using (var transaction = connection.BeginTransaction())
    {
        var affectedRows = connection.Execute(sql,
            new {Kind = InvoiceKind.WebInvoice, Code = "Single_I
nsert_1"},
            commandType: CommandType.StoredProcedure,
            transaction: transaction);

        transaction.Commit();
    }
}
```

范围事务

在开始连接之前开始一个新的范围事务。

```
// using System.Transactions;

using (var transaction = new TransactionScope())
{
    var sql = "Invoice_Insert";

    using (var connection = My.ConnectionFactory())
    {
        connection.Open();

        var affectedRows = connection.Execute(sql,
            new {Kind = InvoiceKind.WebInvoice, Code = "Single_I
nsert_1"},
            commandType: CommandType.StoredProcedure);
    }

    transaction.Complete();
}
```

Dapper - 存储过程

描述

在Dapper中使用存储过程非常简单，你只需要指定命令类型。

执行单个

执行一个存储过程。

```
var sql = "Invoice_Insert";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var affectedRows = connection.Execute(sql,
        new {Kind = InvoiceKind.WebInvoice, Code = "Single_Inser
t_1"},
        commandType: CommandType.StoredProcedure);

    My.Result.Show(affectedRows);
}
```

执行多个

执行多个存储过程。

```
var sql = "Invoice_Insert";

using (var connection = My.ConnectionFactory())
{
    connection.Open();

    var affectedRows = connection.Execute(sql,
        new[]
        {
            new {Kind = InvoiceKind.WebInvoice, Code = "Many_Ins
ert_1"},
            new {Kind = InvoiceKind.WebInvoice, Code = "Many_Ins
ert_2"},
            new {Kind = InvoiceKind.StoreInvoice, Code = "Many_I
nsert_3"}
        },
        commandType: CommandType.StoredProcedure
    );

    My.Result.Show(affectedRows);
}
```