

# .NET 基础拾遗（6）ADO.NET 与数据库开发基础 - 文章 - 伯乐在线

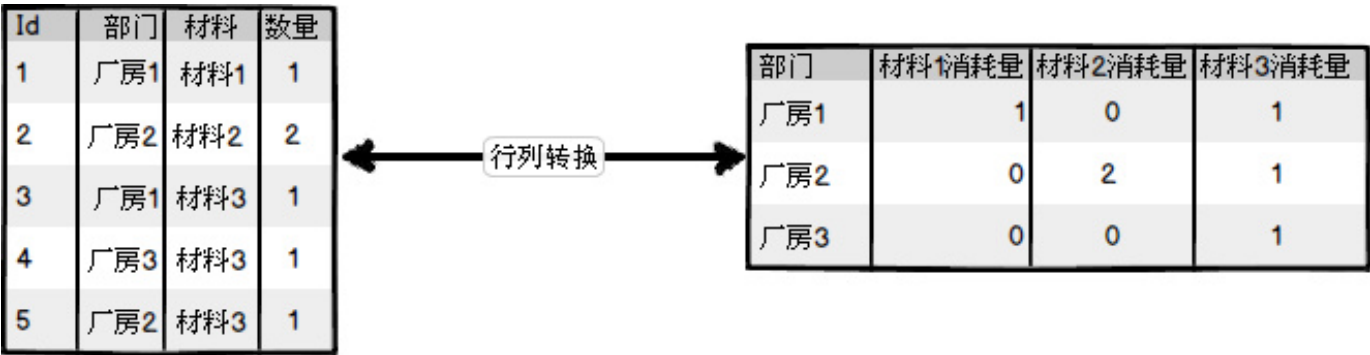


## 一、ADO.NET和数据库程序基础1.1 安身立命之基本：SQL

SQL语句时操作关系型数据库的基础，在开发数据访问层、调试系统工作中十分常用，掌握SQL对于每一个程序员（无论是.NET、Java还是C++等）都非常重要。这里挑选了一个常见的面试题目，来热热身。

常见场景：通过SQL实现单表行列转换

行列转换时数据库系统中经常遇到的一个需求，在数据库设计时，为了适合数据的累积存储，往往采用直接记录的方式，而在展示数据时，则希望整理所有记录并且转置显示。下图是一个行列转换的示意图：



①好了，废话不多说，先建立一张表DeptMaterialDetails：

MySQL

```
CREATE TABLE [dbo].[DeptMaterialDetails](
    [Id] [int] IDENTITY(1,1) NOT NULL,
    [DeptName] [varchar](50) NULL,
    [MaterialName] [varchar](100) NULL,
    [Number] [int] NULL,
CONSTRAINT [PK_DeptMaterialDetails] PRIMARY KEY CLUSTERED
(
    [Id] ASC
)WITH (PAD_INDEX = OFF, STATISTICS_NORECOMPUTE = OFF, IGNORE_DUP_KEY = OFF,
ALLOW_ROW_LOCKS = ON, ALLOW_PAGE_LOCKS = ON) ON [PRIMARY]
) ON [PRIMARY]
```

②填充一些测试数据进该表：

	Id	DeptName	MaterialName	Number
1	1	厂房1	材料1	1
2	2	厂房2	材料2	2
3	3	厂房1	材料3	1
4	4	厂房3	材料3	1
5	5	厂房2	材料3	1
6	6	厂房3	材料1	1
7	7	厂房1	材料1	2
8	8	厂房1	材料2	1
9	9	厂房1	材料3	1

③分析需求，可以发现希望做的是找出具有相同部门的记录，并根据其材料的值累加。经过一番折腾，可以写出如下SQL语句：

MySQL

```
select DeptName as '部门',
SUM(case MaterialName when '材料1' then Number else 0 end) as '材料1消耗',
SUM(case MaterialName when '材料2' then Number else 0 end) as '材料2消耗',
SUM(case MaterialName when '材料3' then Number else 0 end) as '材料3消耗'
from DeptMaterialDetails
group by DeptName
```

执行效果如下图所示，是不是已经完成要求了：

	部门	材料1消耗	材料2消耗	材料3消耗
1	厂房1	3	1	2
2	厂房2	0	2	1
3	厂房3	1	0	1

But，根据上述SQL语句，得到的结果永远只有3种材料的消耗量，如果新增了材料4，那么是不是需要改SQL语句？这时候是不是又想起了在实际开发中时常提到的可扩展性？

④我们可以根据需要动态拼装一个SQL语句，即动态地根据实际材料数目来得到最后的查询语句：

C#

--申明一个字符串用于动态拼接

```
declare <a href='http://www.jobbole.com/members/sql'>@sql</a> varchar(8000)
```

--拼接SQL语句

```
set <a href='http://www.jobbole.com/members/sql'>@sql</a> = 'select DeptName as "部门"
```

--动态地获得材料名，为每个材料构建一个列

```
select <a href='http://www.jobbole.com/members/sql'>@sql</a> = <a href='http://www.jobbole.com/members/sql'>@sql</a> + ',SUM(case MaterialName when '''+temp.Item+''' then Number else 0 end) as [' +temp.Item+' 消耗']
```

```
from (select distinct MaterialName as Item from DeptMaterialDetails) as temp
```

--最终拼上数据源和分组依据

```
select <a href='http://www.jobbole.com/members/sql'>@sql</a> = <a href='http://www.jobbole.com/members/sql'>@sql</a> + 'select DeptName as "部门",
```

```
href='http://www.jobbole.com/members/sql'>@sql</a> + ' from DeptMaterialDetails group by DeptName'
```

--执行SQL语句

```
exec (@sql)
```

执行结果和第一种方式相同，但是需要注意的是：

动态SQL命令的执行效率往往不高，因为动态拼接的原因，导致数据库（查询优化器）可能无法对这样的命令进行优化。此外，这样的SQL命令还受限于字符串的长度（需要事先确定其长度限制），而动态SQL命令的长度往往是根据实际表的内容而改变，因此这类动态SQL命令无法保证100%正常运行。

## 1.2 ADO.NET支持哪几种数据源？

ADO.NET支持的数据源很多，从类别上来划分的话可以大致分为四类。ADO.NET也正是通过如下所示这四个命名空间来实现对这些数据源的支持的：

### ① System.Data.SqlClient

这也许是.NET程序员最常用的了，因为MSSQL你懂的！当然，这不是连接MSSQL的唯一方案，通过OLEDB或者ODBC都可以访问，但是SqlClient下的组件直接针对MSSQL，因此ADO.NET其实是为其专门做了一些优化工作，因此使用MSSQL应该首选 System.Data.SqlClient 命名空间。

### ② System.Data.OracleClient

顾名思义，这个命名空间针对Oracle数据库产品，并且还得搭配Oracle数据库的客户端组件（Oracle.DataAccess.dll）一起使用。

### ③ System.Data.OleDb

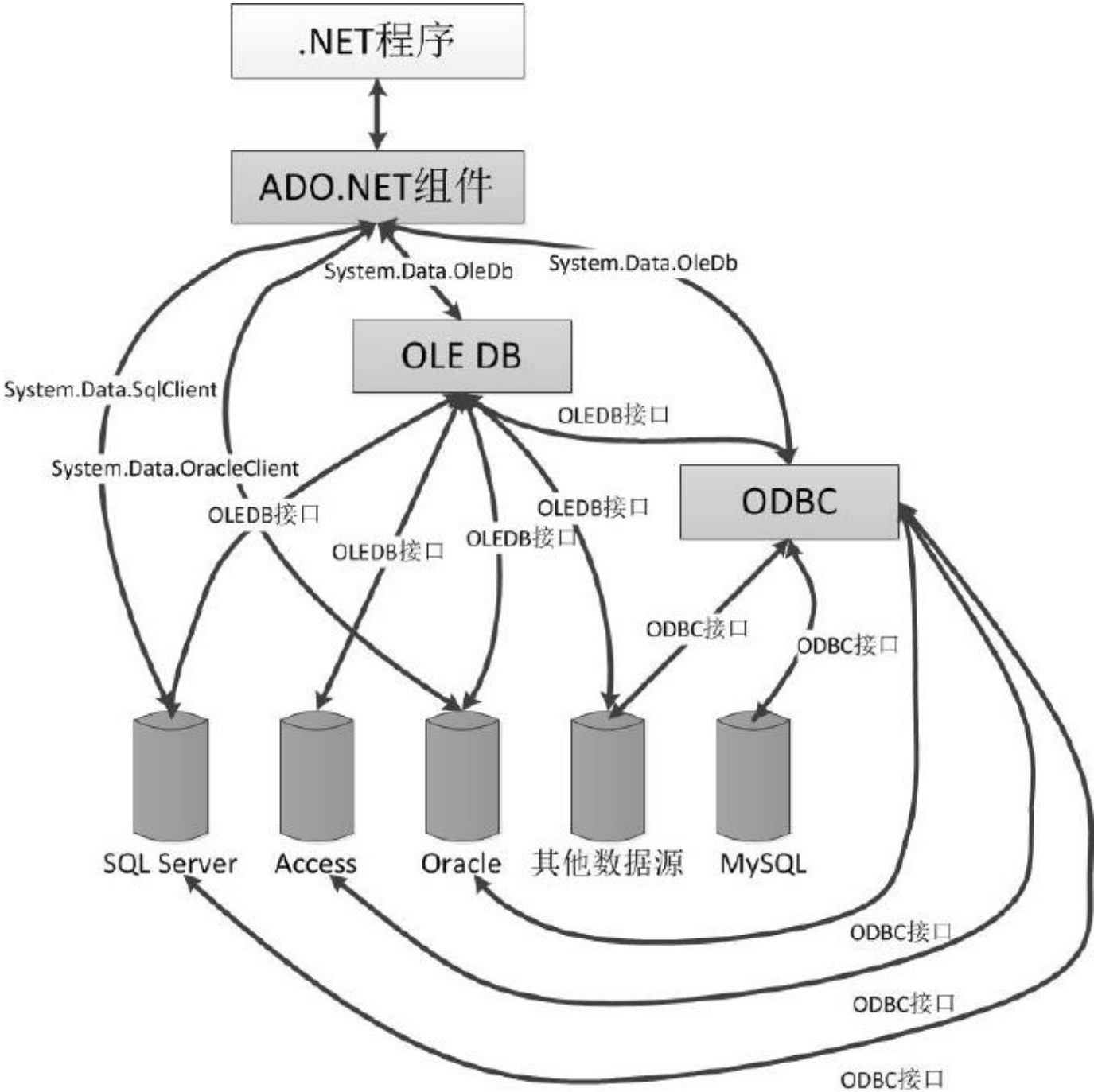
该命名空间下的组件主要针对OLEDB（Microsoft提供的通向不同数据源的低级API）的标准接口，它还可以连接其他非SQL数据类型的数据源。OLEDB是一种标准的接口，实现了不同数据源统一接口的功能。

### ④ System.Data.Odbc

该命名空间下的组件针对ODBC标准接口。

关于ODBC：开放数据库互连（Open Database Connectivity, ODBC）是微软公司开放服务结构（WOSA, Windows Open Services Architecture）中有关数据库的一个组成部分，它建立了一组规范，并提供了一组对数据库访问的标准API（应用程序编程接口）。这些API利用SQL来完成其大部分任务。ODBC本身也提供了对SQL语言的支持，用户可以直接将SQL语句送给ODBC。

总体来说，ADO.NET为我们屏蔽了所有的数据库访问层次，提供了统一的API给我们，使我们无需考虑底层的数据源是具体的DataBase还是另一种标准接口。下图直观地展示了ADO.NET与可能的数据源的连接：



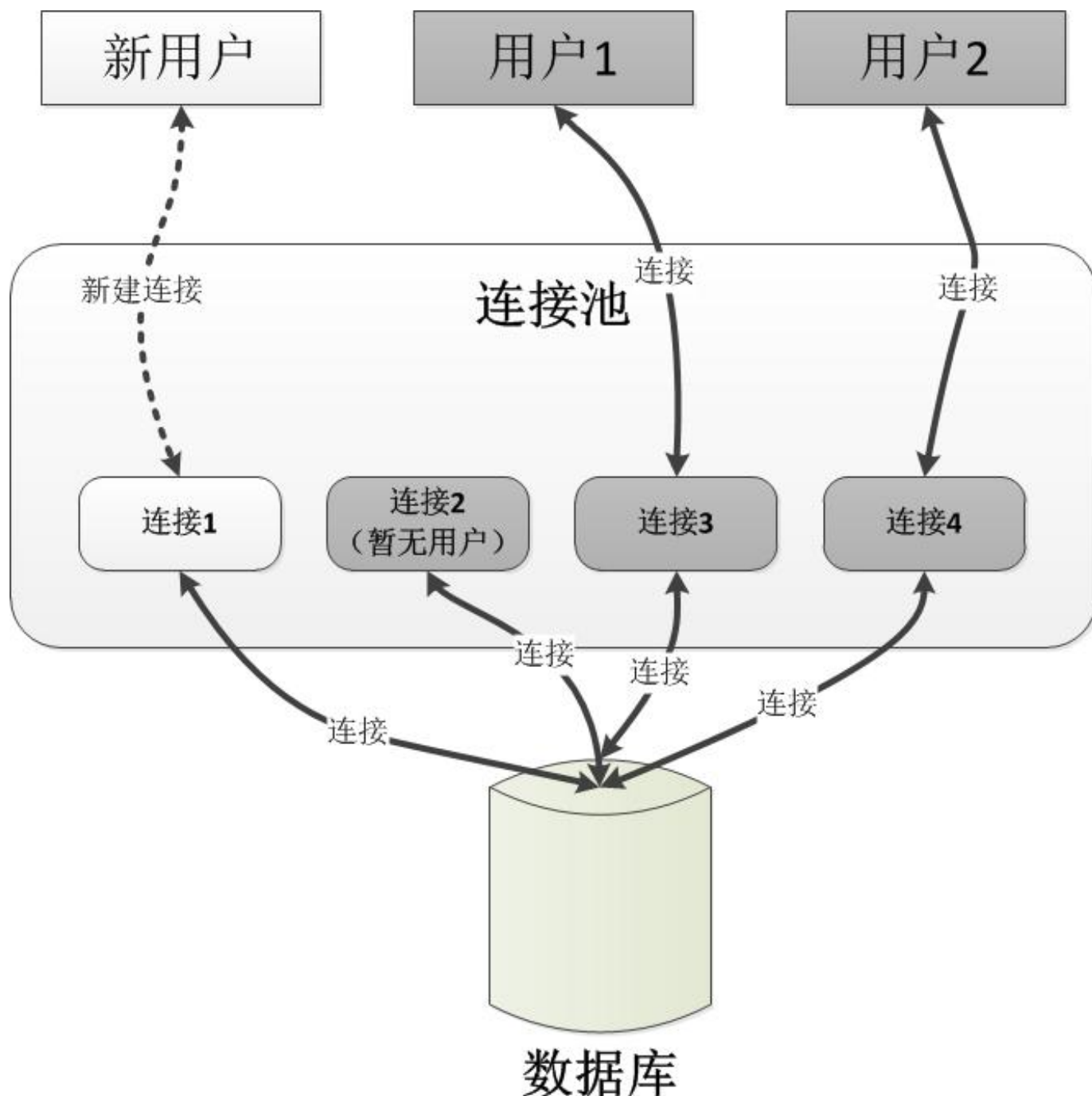
二、ADO.NET和数据库的连接2.1 简述数据库连接池的机制

数据库连接一般都被认为是一个性能成本相对较大的动作，所以针对数据库连接以及读写的优化往往是系统优化的关键点。数据库连接池就是一个非常重要的优化机制。

(1) 数据库连接池的基本概念

数据库连接池，顾名思义就是一个存储数据库连接的缓冲池，由于连接和断开一个数据库的开销很大（想想经典的TCP三次握手和四次挥手），反复连接和断开数据库对于系统的性能影响将会非常严重。而在.NET程序中，有时候是无法预测下一次数据库访问的需求何时到来，所以通常的做法就是在使用完一个连接后就立即关闭它，这就需要ADO.NET的内部机制来维护这个访问池。

下图展示了数据库连接池的机制，在该机制中，当一个用户新申请了一个数据库连接时，当数据库池内连接匹配的情况下，用户会从连接池中直接获得一个被保持的连接。在用户使用完调用Close关闭连接时，连接池会将该连接返回到活动连接池中，而不是真正关闭连接。连接回到了活动链接池中后，即可在下一个Open调用中重复使用。



默认情况下，数据库连接时处于启用状态的。我们也可以通过数据库连接字符串设置关闭数据库连接池，如下面的代码所示：

```
C#  
  
using (SqlConnection connection = new SqlConnection("Server=127.0.0.1;Initial  
Catalog=TestDB;Integrated Security=SSPI;Pooling=false"))  
{  
    connection.Open();  
    // 执行你想要执行的数据库操作  
}
```

其中参数Pooling=false就代表了关闭连接池。当然，我们还可以设置连接池中的最大和最小连接数，参数分别对应Max Pool Size和Min Pool Size。

## (2) 数据库连接的复用

由于数据源和连接参数选择的不同，每个数据库的连接并不是完全通用的。因此，ADO.NET选择通过连接字符串来区分。一旦用户使用某个连接字符串来申请数据库连接，ADO.NET将判断连接池中是否存在拥有相同连接字符串的连接，如果有则直接分配，没有则新建连接。

我们可以看看下面一段代码，三个不同的连接中，第三个复用第一个连接，第二个则无法复用第一个连接：

```
C#  
using (SqlConnection connection = new SqlConnection("Server=127.0.0.1;Initial  
Catalog=TestDB;Integrated Security=SSPI"))  
{  
    // 假设这是启动后第一个数据库连接请求，一个新连接将被建立  
    connection.Open();  
}  
using (SqlConnection connection = new SqlConnection("Server=127.0.0.1;Initial  
Catalog=TestDB1;Integrated Security=SSPI"))  
{  
    // 由于和上一个连接的字符串不同，因此无法复用第一个连接  
    connection.Open();  
}  
using (SqlConnection connection = new SqlConnection("Server=127.0.0.1;Initial  
Catalog=TestDB;Integrated Security=SSPI"))  
{  
    // 连接字符串和第一个连接相同，保存在连接池中的第一个连接被复用  
    connection.Open();  
}
```

### (3) 不同数据源的连接池机制

事实上，ADO.NET组件本身并不直接包含连接池，而针对不同类别机制的数据源指定不同的连接池方案。对于SqlClient、OracleClient命名空间下的组件，使用的连接池是由托管代码直接编写的，可以理解为连接池直接在.NET框架中运行。而对于OLEDB和ODBC的数据源来说，连接池的实现完全依靠OLEDB和ODBC提供商实现，ADO.NET只与其约定相应规范。

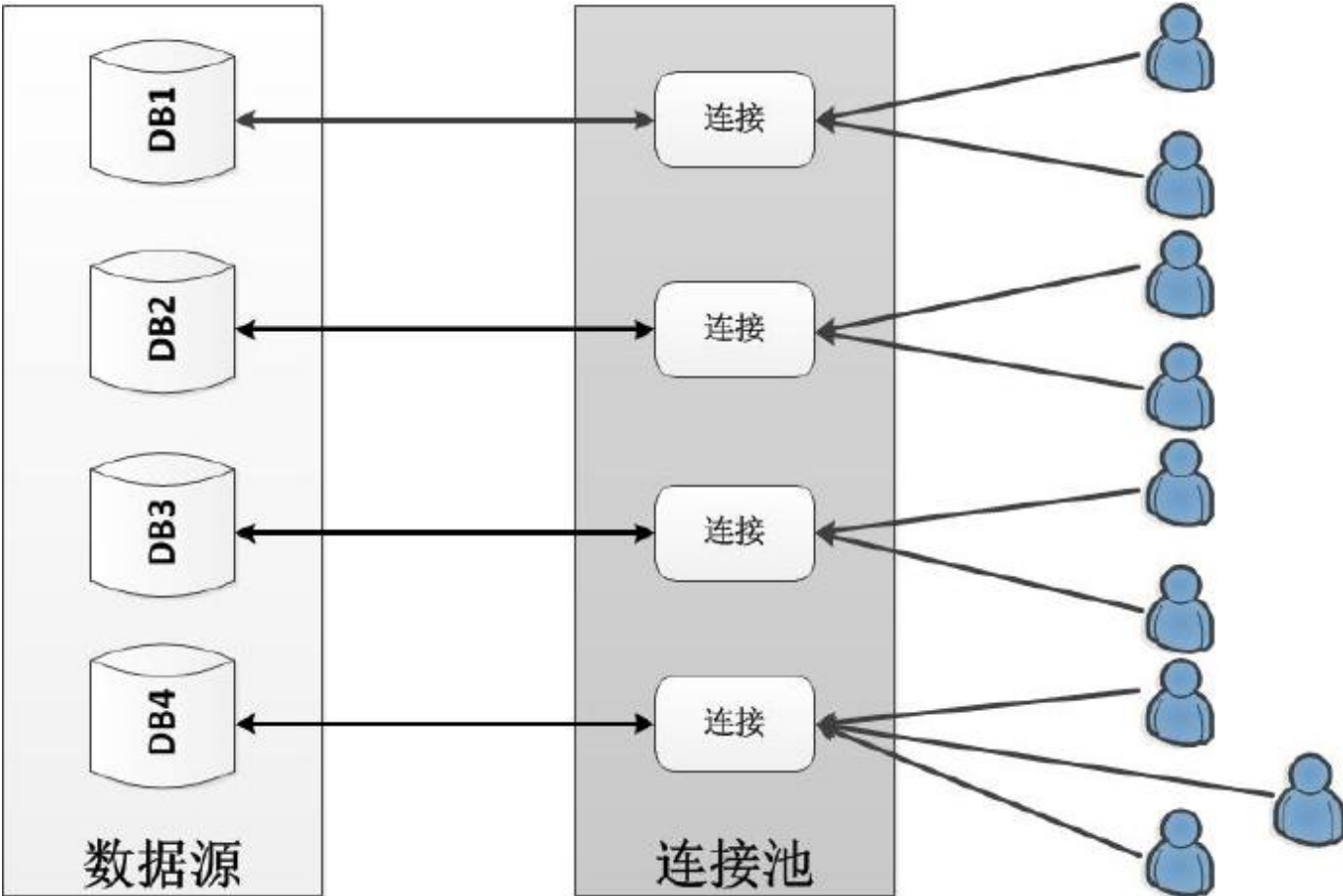
## 2.2 如何提高连接池内连接的重用率

由于只有相同连接字符串才能共享连接，因此经常导致连接池失效的问题，所以需要提高连接池内连接的重用率。

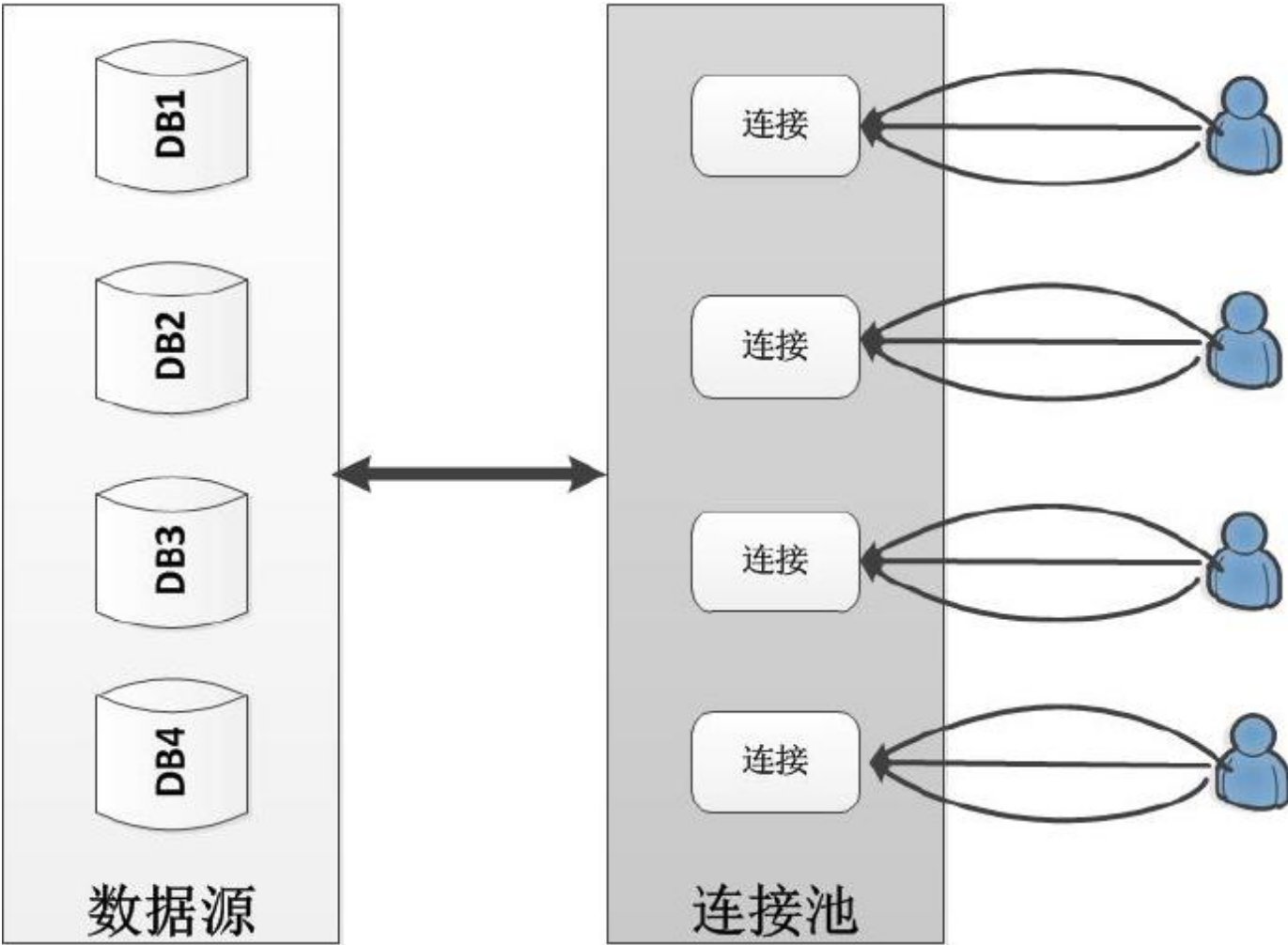
### (1) 连接池重用率低下的原因

由于数据库连接池仅按照数据库连接字符串来判断连接是否可重用，所以连接字符串内的任何改动都会导致连接失效。就系统内部而言，数据库连接字符串中最常被修改的两个属性就是数据库名和用户名/密码。

因此，对于多数据库的系统来说，只有同一数据库的连接才会被共用，如下图所示：



而对多用户的系统而言，只有同一用户的申请才能共用数据库连接，如下图所示：



（2）如何提高数据库连接池重用率

这里提供一种能够有效提高数据库连接池重用率的方式，但是也会带来一点小安全隐患，在进行设计时需要权衡利弊关系，并根据实际情况来指定措施。

### ① 建立跳板数据库

在数据库内建立一个所有权限用户都能访问的跳板数据库，在进行数据库连接时先连接到该数据库，然后再使用 `use databasename` 这样的SQL语句来选择需要访问的数据库，这样就能够避免因为访问的数据库不一致而导致连接字符串不一致的情况。

下面的示例代码演示了这一做法：

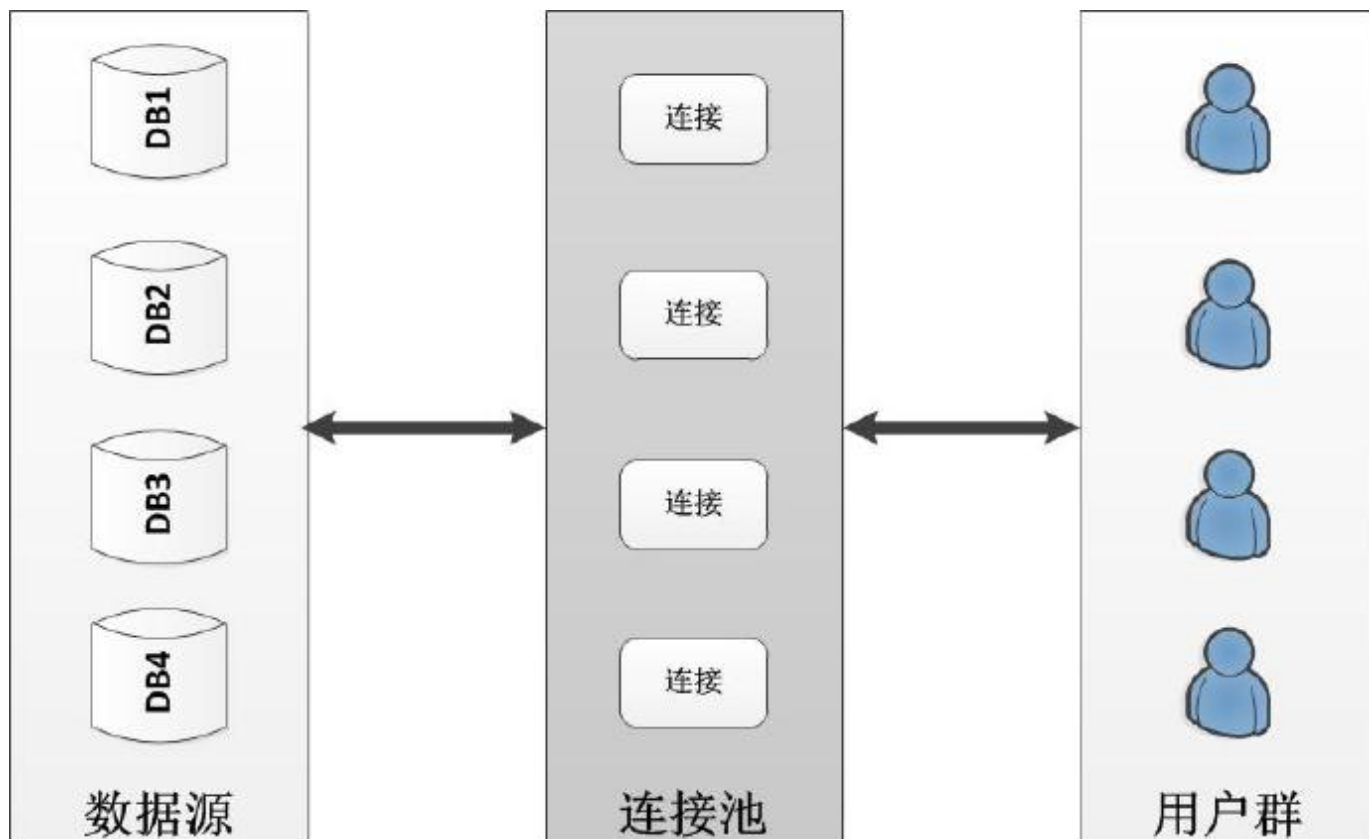
```
C#
1
2
3
4
5
6
7
8
// 假设这里使用Entry数据作为跳板数据库，然后再使用databaseName指定的数据库
using (SqlConnection connection = new
SqlConnection("Server=192.168.80.100;Uid=public;Pwd=public;Database=Entry"))
{
    connection.Open();
    SqlCommand command = connection.CreateCommand();
    command.CommandText = string.Format("USE {0}", databaseName);
    command.ExecuteNonQuery();
}
```

### ② 不使用数据库用户系统来管理系统权限

这样做的结果就是永远使用管理员的账号来连接数据库，而在做具体工作时再根据用户的实际权限，使用代码来限定操作。带来的好处就是：数据库看连接字符串不会因为实际用户的不同而不同。当然，永远使用管理员账号来连接也会相应带来安全隐患！

下图展示了采用了这种方案后数据库连接池的使用情况：





### 三、使用ADO.NET读写数据库3.1 ADO.NET支持访问数据库的方式有哪些？

对于关系型数据库，ADO.NET支持两种访问模式，一种是连接式的访问模式，而另外一种则是离线式的访问模式。

#### (1) 连接式的访问

连接式的访问是指读取数据时保持和数据库的连接，并且在使用时独占整个连接，逐步读取数据。这种模式比较适合从数据量庞大的数据库中查询数据，并且不能确定读取数量的情况。使用XXXCommand和XXXDataReader对象来读取数据就是一个典型的连接式数据访问，这种模式的缺点就是：数据库连接被长时间地保持在打开的状态。

下面的一段示例代码展示了这一读取模式的典型使用，首先是数据访问层的静态方法，该方法返回一个指定SQL命令返回的SqlDataReader独享，该对象关闭时会自动关闭依赖的数据库连接。

```
C#///  
    /// 数据访问层类型  
    /// </summary>  
    public class DataHelper  
    {  
        private static readonly String conn_String = "Server=localhost;Integrated  
Security=true;database=TestDB";  
        /// <summary>  
        /// 使用给定的sql来访问数据库  
        /// 返回SqlDataReader对象，提供连接式访问  
        /// </summary>  
        /// <param name="sql">SQL命令</param>
```

```

/// <returns>SqlDataReader对象</returns>
public static SqlDataReader GetReader(String sql)
{
    SqlConnection con = new SqlConnection(conn_String);
    try
    {
        // 打开连接，执行查询
        // 并且返回SqlDataReader
        con.Open();
        using (SqlCommand cmd = con.CreateCommand())
        {
            cmd.CommandText = sql;
            SqlDataReader dr = cmd.ExecuteReader
                (CommandBehavior.CloseConnection);

            return dr;
        }
    }
    // 连接数据库随时可能发生异常
    catch (Exception ex)
    {
        if (con.State != ConnectionState.Closed)
        {
            con.Close();
        }
        return null;
    }
}

```

其次是调用该方法的入口，使用者将会得到一个连接着数据库的SqlDataReader对象，该对象本身并不包含任何数据，使用者可以通过该对象读取数据库中的数据。但因为是连接方式，读取只能是顺序地逐条读取。

```

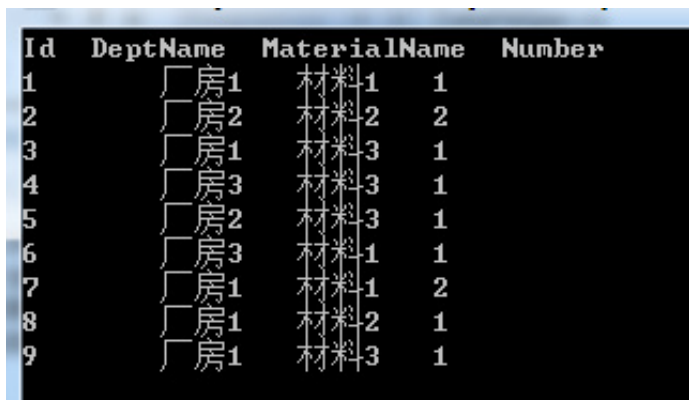
C#/// <summary>
    /// 使用数据库访问层
    /// 连接式读取数据
    /// </summary>
    class Program
    {
        // SQL命令
        private static readonly String sql = "select * from
        dbo.DeptMaterialDetails";
    }

```

```
static void Main(string[] args)
{
    // 使用连接式方法读取数据源
    using (SqlDataReader reader = DataHelper.GetReader(sql))
    {
        // 得到列数
        int colcount = reader.FieldCount;
        // 打印列名
        for (int i = 0; i < colcount; i++)
        {
            Console.Write("{0}    ", reader.GetName(i));
        }
        Console.WriteLine();
        // 顺序读取每一行，并打印
        while (reader.Read())
        {
            for (int i = 0; i < colcount; i++)
            {
                Console.Write("{0}\t",
reader[i].ToString());

            }
            Console.WriteLine();
        }
        reader.Close();
    }
    Console.ReadKey();
}
```

下图是这个示例的执行结果，从数据库中读取了指定表的内容：



Id	DeptName	MaterialName	Number
1	厂房1	材料1	1
2	厂房2	材料2	2
3	厂房1	材料3	1
4	厂房3	材料3	1
5	厂房2	材料3	1
6	厂房3	材料1	1
7	厂房1	材料1	2
8	厂房1	材料2	1
9	厂房1	材料3	1

## (2) 脱机式的访问

脱机式的访问并不是指不连接数据库，而是指一般在读取实际数据时连接就已经断开了。脱机式访问方式在连接至数据库后，会根据SQL命令批量读入所有记录，这样就能直接断开数据库连接以供其他线程使

用，读入的记录将暂时存放在内存之中。脱机式访问的优点就在于不会长期占用数据库连接资源，而这样做的代价就是将消耗内存来存储数据，在大数据量查询的情况下该方式并不适用。例如，使用XXXDataAdapter和DataSet对象就是最常用的脱机式访问方式。

下面的实例代码对上面的连接式做了一些修改，借助SqlDataAdapter和DataSet来实现脱机式访问：

```
C#/// <summary>
    /// 数据访问层类型
    /// </summary>
    public class DataHelper
    {
        private static readonly String conn_String = "Server=localhost;Integrated
Security=true;database=TestDB";
        /// <summary>
        /// 使用给定的sql来访问数据库
        /// 返回DataSet对象
        /// </summary>
        /// <param name="sql">SQL命令</param>
        /// <returns>DataSet对象</returns>
        public static DataSet GetDataSet(String sql)
        {
            SqlConnection con = new SqlConnection(conn_String);
            DataSet ds = new DataSet();
            try
            {
                // 打开连接，执行查询
                // 并且返回DataSet
                con.Open();
                using (SqlDataAdapter sd = new SqlDataAdapter(sql, con))
                {
                    // 这里数据将被批量读入
                    sd.Fill(ds);
                }
                return ds;
            }
            // 连接数据库随时可能发生异常
            catch (Exception ex)
            {
                if (con.State != ConnectionState.Closed)
                {
                    con.Close();
                }
                return ds;
            }
        }
    }
```

```

    }
}

}

/// <summary>
/// 使用数据库访问层
/// 脱机式读取数据
/// </summary>
class Program
{
    //SQL命令
    private static readonly String sql = "select * from
dbo.DeptMaterialDetails";
    static void Main(string[] args)
    {
        DataSet ds = DataHelper.GetDataSet(sql);
        // 打印结果, 这里假设只对DataSet中的第一个表感兴趣
        DataTable dt = ds.Tables[0];
        // 打印列名
        foreach (DataColumn column in dt.Columns)
        {
            Console.Write("{0}    ", column.ColumnName);
        }
        Console.WriteLine();
        // 打印表内容
        foreach (DataRow row in dt.Rows)
        {
            for (int i = 0; i < dt.Columns.Count; i++)
            {
                Console.Write("{0}    ", row[i].ToString());
            }
            Console.WriteLine();
        }
        Console.ReadKey();
    }
}

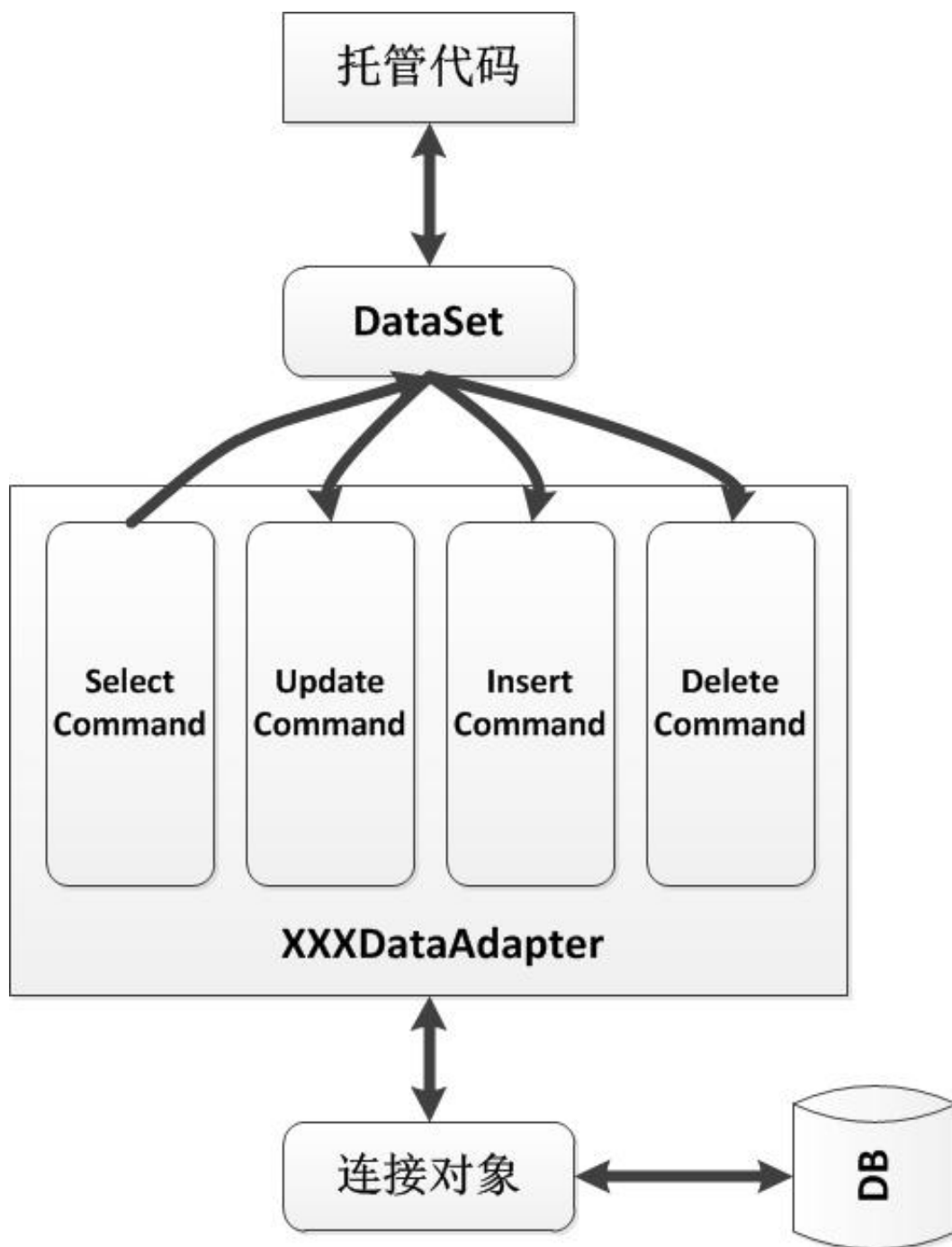
```

由于数据访问类的处理相当赶紧, 调用者轻松就能获得包含数据源的DataSet对象, 这时任何操作都已经和数据源没有联系了。

### 3.2 简述SqlDataAdapter的基本工作机制

ADO.NET提供的XXXDataAdapter类型都使用了非常一致的机制, 并且向使用者提供了统一的接口。一个SqlDataAdapter对象, 在数据库操作中充当了中间适配的角色, 它组织起数据缓存对数据库的所有操作, 进行统一执行。一个SqlDataAdapter对象内实际包含四个负责具体操作的SqlCommand对象, 它们分

别负责查询、更新、插入和删除操作。下图展示了SqlDataAdapter的工作机制：



如上图所示，实际上进行数据操作的是包含在SqlDataAdapter内的四个SqlCommand对象，而当SqlDataAdapter的Update方法被调用时，它会根据DataSet独享的更新情况而调用插入、删除和更新等命令。

### 3.3 如何实现批量更新的功能？

#### (1) 批量更新的概念

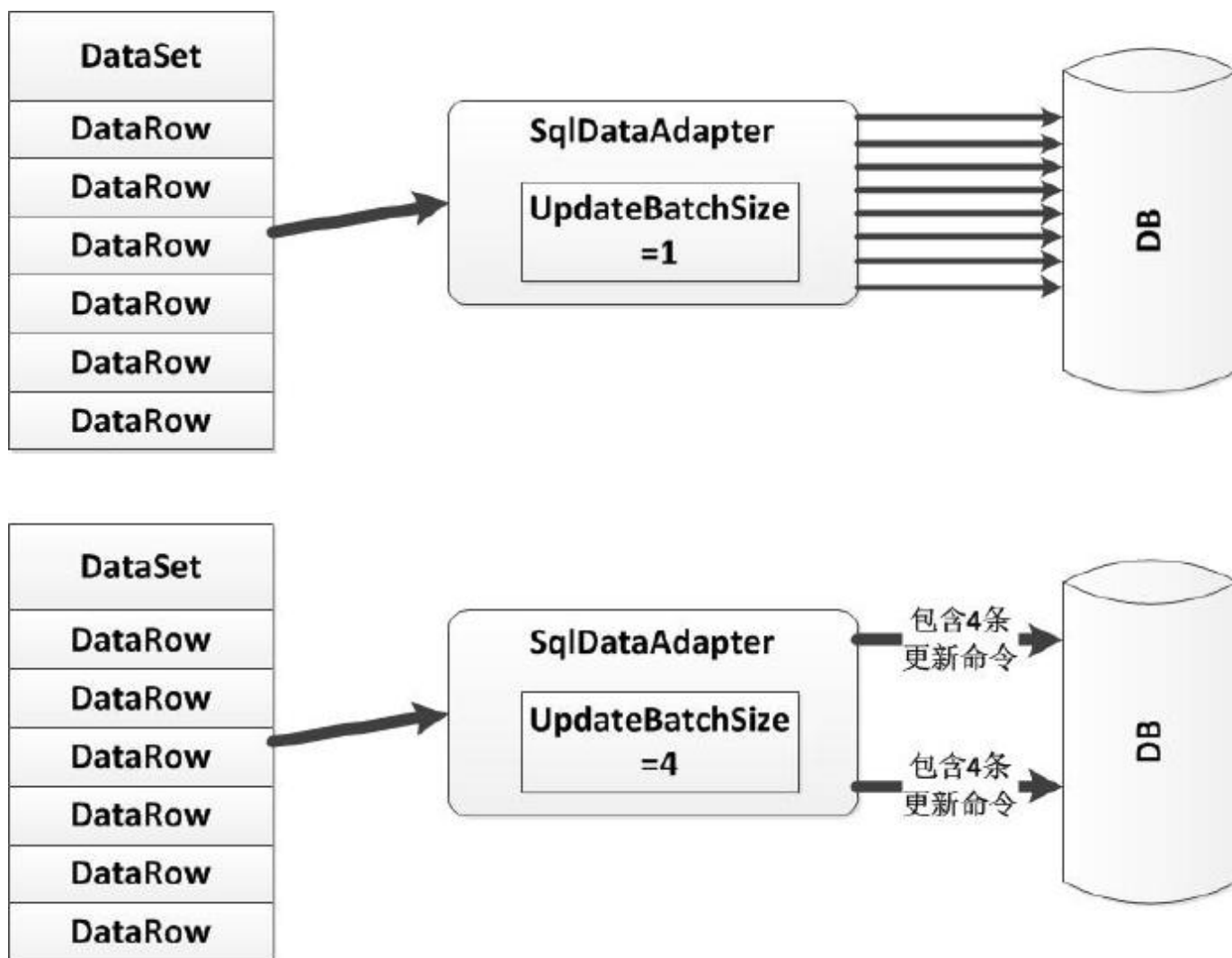
使用XXXDataAdapter更新数据，由于每一行都需要都需要一个从程序集到数据库的往返，在大批量更新的情况下，效率是非常低的。可以考虑使用一次发送多条更新命令的处理方式，这就需要用到UpdateBatchSize属性。在.NET 2.0之后，SqlClient和OracleClient都支持这个属性，这里以SQL Server数据源为例，介绍一下UpdateBatchSize的基本使用。

UpdateBatchSize的值一共有三种：

- ① =0, DbDataAdapter将使用服务器能处理的最大批处理大小;
- ② =1, 禁用批量更新;
- ③ >1, 使用UpdateBatchSize操作批处理一次性发送的量;

当批量更新被允许时, SqlDataAdapter的Update方法将每次发送多条更新命令到数据库, 从而提高性能。

But, 使用批量更新并不意味着SQL的合并或优化。事实上, 批量的意义在于把多个发往数据库服务器的SQL语句放在一个请求中发送。例如, 将UpdateBatchSize设置为20时, 原本每个更新行发送一次更新命令将变为每20个更新行发送一次更新命令, 而每个命令中包含了20个更新一行的命令。下图展示了这一区别:



## (2) 批量更新的使用

下面的示例代码展示了如何使用UpdateBatchSize属性来设置批量更新, 这里更改了DataHelper的Update方法, 在内部设置了UpdateBatchSize属性。

```
C#  
public class DataHelper  
{
```

```

        private static readonly string conn_string = "Server=localhost;Integrated
Security=true;database=TestDB";

        //选择、更新、删除和插入的SQL命令
        static readonly string SQL_SELECT = "SELECT * FROM DeptMaterialDetails";
        static readonly string SQL_UPDATE = "UPDATE DeptMaterialDetails SET
Department=@Department,Item=@Item,Number=@Number where Id=@Id";
        static readonly string SQL_DELETE = "DELETE FROM DeptMaterialDetails where
Id=@Id";

        static readonly string SQL_INSERT = "Insert INTO DeptMaterialDetails
(Department,Item,Number) VALUES (@Department,@Item,@Number)";

        /// <summary>
        /// 得到SqlDataAdapter, 私有方法
        /// </summary>
        /// <param name="con"></param>
        /// <returns></returns>
        private static SqlDataAdapter GetDataAdapter(SqlConnection con)
        {
            SqlDataAdapter sda = new SqlDataAdapter();
            sda.SelectCommand = new SqlCommand(SQL_SELECT, con);
            sda.UpdateCommand = new SqlCommand(SQL_UPDATE, con);
            sda.DeleteCommand = new SqlCommand(SQL_DELETE, con);
            sda.InsertCommand = new SqlCommand(SQL_INSERT, con);
            sda.UpdateCommand.Parameters.AddRange(GetUpdatePars());
            sda.InsertCommand.Parameters.AddRange(GetInsertPars());
            sda.DeleteCommand.Parameters.AddRange(GetDeletePars());
            return sda;
        }
        // 三个SqlCommand的参数
        private static SqlParameter[] GetInsertPars()
        {
            SqlParameter[] pars = new SqlParameter[3];
            pars[0] = new SqlParameter("@Department", SqlDbType.VarChar, 50,
"Department");

            pars[1] = new SqlParameter("@Item", SqlDbType.VarChar, 50, "Item");
            pars[2] = new SqlParameter("@Number", SqlDbType.Int, 4, "Number");
            return pars;
        }
        private static SqlParameter[] GetUpdatePars()
        {
            SqlParameter[] pars = new SqlParameter[4];
            pars[0] = new SqlParameter("@Id", SqlDbType.VarChar, 50, "Id");
            pars[1] = new SqlParameter("@Department", SqlDbType.VarChar, 50,

```



```

    "Department");

    pars[2] = new SqlParameter("@Item", SqlDbType.VarChar, 50, "Item");
    pars[3] = new SqlParameter("@Number", SqlDbType.Int, 4, "Number");
    return pars;
}

private static SqlParameter[] GetDeletePars()
{
    SqlParameter[] pars = new SqlParameter[1];
    pars[0] = new SqlParameter("@Id", SqlDbType.VarChar, 50, "Id");
    return pars;
}

/// <summary>
/// 更新数据库，使用批量更新
/// </summary>
/// <param name="ds">数据集</param>
public static void Update(DataSet ds)
{
    using (SqlConnection connection = new SqlConnection(conn_string))
    {
        connection.Open();
        using (SqlDataAdapter adapter =
            GetDataAdapter(connection))
        {
            // 设置批量更新
            adapter.UpdateBatchSize = 0;
            adapter.Update(ds);
        }
    }
}
}

```

PS：近年来比较流行的轻量级ORM例如Dapper一类的这里就不作介绍了，后续我会实践一下写一个初探系列的文章。另外，数据库中的事务及其隔离级别一类的介绍也会在后续详细阅读《MSSQL技术内幕》后写一个读书笔记，到时分享给各位园友。

#### 参考资料

- (1) 朱毅，《进入IT企业必读的200个.NET面试题》
- (2) 张子阳，《.NET之美：.NET关键技术深入解析》
- (3) 王涛，《你必须知道的.NET》
- (4) 百度百科，[ODBC](#)

拿高薪，还能扩大业界知名度！优秀的开发工程师看过来 -> [《高薪招募讲师》](#)

1 赞 1 收藏 [评论](#)



合作联系

Email: [bd@jobbole.com](mailto:bd@jobbole.com)

QQ: 2302462408 (加好友请注明来意)

更多频道

[小组](#) - 好的话题、有启发的回复、值得信赖的圈子

[头条](#) - 分享和发现有价值的内容与观点

[相亲](#) - 为IT单身男女服务的征婚传播平台

[资源](#) - 优秀的工具资源导航

[翻译](#) - 翻译传播优秀的外文文章

[文章](#) - 国内外的精选文章

[设计](#) - UI, 网页, 交互和用户体验

[iOS](#) - 专注iOS技术分享

[安卓](#) - 专注Android技术分享

[前端](#) - JavaScript, HTML5, CSS

[Java](#) - 专注Java技术分享

[Python](#) - 专注Python技术分享