KAPITOLA 4

LINQ pro SQL: Dotazování na data

První a nejzřejmější aplikací LINQ je dotazování do externí relační databáze. LINQ pro SQL je součást projektu LINQ, která nabízí možnost dotazování do relační databáze Microsoft SQL Serveru a také objektový model vycházející z dostupných entit. Jinými slovy, můžete definovat množinu objektů, které představují tenkou abstraktní vrstvu nad relačními daty, a do tohoto objektového modelu se dotazovat pomocí dotazů LINQ, které se ve stroji LINQ pro SQL převádějí na odpovídající dotazy SQL. LINQ pro SQL podporuje Microsoft SQL Server od verze SQL Serveru 2000 a Microsoft SQL Server Compact počínaje verzí 3.5.

V LINQ pro SQL lze napsat takovýto prostý dotaz:

```
var query =
   from c in Customers
   where c.Country == "USA"
         && c.State == "WA"
   select new {c.CustomerID, c.CompanyName, c.City };
```

Tento dotaz se převádí na dotaz SQL, jenž se pošle do relační databáze:

```
SELECT CustomerID, CompanyName, City
FROM Customers
WHERE Country = 'USA'
   AND Region = 'WA'
```


Důležité

Dotazy SQL generované v LINQ, které si budeme ukazovat v této kapitole, jsou pouze ilustrativní. Microsoft si vyhrazuje právo nezávisle definovat SQL generované v LINQ a občas v textu použijeme zjednodušené dotazy. Neměli byste se tedy na vypisované dotazy SQL spoléhat.

V tuto chvíli vás možná napadá několik otázek. Jednak, jak lze psát dotaz LINQ pomocí názvů objektů validovaných kompilátorem? Dále, kdy se generuje z dotazu LINQ dotaz SQL? A za třetí, kdy se dotaz SQL provádí? Abyste poznali odpovědi na tyto otázky, musíte pochopit model entit v LINQ pro SQL a také princip odloženého provádění dotazů.

Entity v LINQ pro SQL

Libovolná externí data je nutné popsat příslušnými metadaty, navázanými na definice tříd. Každá tabulka musí mít odpovídající třídu s příslušnými atributy. Tato třída odpovídá řádku dat a popisuje všechny sloupce jako datové členy definovaného typu. Typem může být úplný nebo částečný popis existující fyzické tabulky, pohledu či výstupu z uložené procedury. Uvnitř dotazu LINQ pro projekci či filtrování je možné používat pouze popsaná pole. Výpis 4.1 ukazuje definici krátké a jednoduché entity.

Důležité

Do svého projektu musíte vložit sestavení *System.Data.Linq*, abyste mohli používat třídy a atributy LINQ pro SQL. Atributy používané ve výpisu 4.1 mají své definice ve jmenném prostoru *System.Data.Linq.Mapping*.

Výpis 4.1 Definice entity pro LINQ pro SQL

```
using System.Data.Linq.Mapping;

[Table(Name="Customers")]
public class Customer {
    [Column] public string CustomerID;
    [Column] public string CompanyName;
    [Column] public string City;
    [Column(Name="Region")] public string State;
    [Column] public string Country;
}
```

Typ *Customer* stanovuje obsah řádku a každé pole či vlastnost s atributem *Column* odpovídá sloupci v relační tabulce. Parametr *Name* může obsahovat název sloupce, jenž se odlišuje od názvu datového členu. (V našem příkladě odpovídá název *State* sloupci Region v příslušné tabulce.) Atribut *Table* říká, že třída je entitou reprezentující data v databázové tabulce; vlastnost *Name* může obsahovat název tabulky, který se odlišuje od názvu entity. Pro název třídy se obvykle používá jednotné číslo (jeden řádek) a pro název tabulky (množina řádků) množné číslo.

Abyste mohli sestavovat dotazy LINQ pro SQL do dat zákazníků, potřebujete tabulku Customers. Správným způsobem, jak vytvořit takový typ, je použít obecnou třídu *Table<T>*:

```
Table<Customer> Customers = ...;
// ...
var query =
   from c in Customers
   // ...
```

Poznámka

K vytváření dotazu LINQ do tabulky *Customers* potřebujete třídu implementující rozhraní *IEnumerable<T>*, kde jako *T* bude figurovat typ *Customers*. Nicméně LINQ pro SQL potřebuje implementace rozšiřujících metod v jiné podobě, než v jaké je implementuje LINQ pro objekty, jež jsme viděli v kapitole 3, „LINQ pro objekty". Z tohoto důvodu musí-

te použít objekt s rozhraním *IQueryable<T>*, abyste mohli vytvářet dotazy LINQ pro SQL. Třída *Table<T>* rozhraní *IQueryable<T>* obsahuje. Aby bylo možné vložit rozšíření LINQ pro SQL, musí být ve zdrojovém kódu příkaz *using System.Data.Linq*.

Pro objekt tabulky *Customers* je potřeba založit instanci. K tomu potřebujete instanci třídy *DataContext*, která vytváří most mezi prostředím LINQ a externí relační databází. Principu práce s třídou *DataContext* je nejpodobnější spojení do databáze – ve skutečnosti je povinným parametrem při zakládání instance třídy *DataContext* připojovací řetězec či objekt *Connection*. Metoda *GetTable<T>* vrací odpovídající objekt typu *Table<T>* pro zadaný typ:

```
DataContext db = new DataContext("Database=Northwind");
Table<Customer> Customers = db.GetTable<Customer>();
```

Poznámka

Třída *DataContext* interně používá třídu *SqlConnection* z prostředí ADO.NET. Existující spojení *SqlConnection* můžete vložit do konstruktoru třídy *DataContext* a spojení, které používá instance třídy *DataContext*, můžete číst prostřednictvím vlastnosti *Connection*. Všechny služby související s databázovým spojením, například používání zásobníku spojení (connection pooling, standardně zapnuto), jsou dostupné na úrovni spojení *SqlConnection* a třída *DataContext* je přímo neimplementuje.

Výpis 4.2 ukazuje výsledný kód, když všechny uvedené prvky spojíte dohromady.

Výpis 4.2 Jednoduchý dotaz LINQ pro SQL

```
DataContext db = new DataContext( ConnectionString );
Table<Customer> Customers = db.GetTable<Customer>();

var query =
    from    c in Customers
    where   c.Country == "USA"
            && c.State == "WA"
    select new {c.CustomerID, c.CompanyName, c.City };

foreach( var row in query ) {
    Console.WriteLine( row );
}
```

Proměnná *query* se inicializuje pomocí dotazovacího výrazu, jenž vytváří strom výrazu. Strom výrazu představuje obraz výrazu v paměti, a nikoli odkaz na metodu pomocí delegáta. Když se ve smyčce *foreach* vypisují data získaná dotazem, používá se strom výrazu ke generování odpovídajícího dotazu SQL s pomocí všech metadat a informací, které se nacházejí ve třídách entit a v používané instanci třídy *DataContext*.

Poznámka

Metoda *odloženého provádění*, jež se používá v LINQ pro SQL, převádí strom výrazu na dotaz SQL, jenž je platný v příslušné relační databázi. Dotaz LINQ je funkčně ekvivalentní řetězci s dotazem SQL, přinejmenším se dvěma významnými rozdíly. Jednak je dotaz vázán na objektový model, a nikoli na databázovou strukturu. A dále, jeho reprezentace

je sémanticky logická bez nutnosti používání analyzátoru SQL a bez vazby na určitou konkrétní verzi jazyka SQL. Strom výrazu je také možné před použitím dynamicky vytvářet v paměti, což si ukážeme v kapitole 11, „Uvnitř stromů výrazů".

Návratová data z dotazu SQL, přistupující k řádku *row* ve smyčce *foreach*, se používají k naplnění promítaného anonymního typu zapsaného za klíčovým slovem *select*. V této ukázce se nikde nezakládá instance třídy *Customer*, která se v LINQ používá pouze pro analýzu metadat.

Vygenerovaný dotaz SQL si můžete prohlédnout pomocí metody *GetCommand* třídy *DataContext*. Ve vlastnosti *CommandText* navráceného objektu typu *DbCommand* se nachází vygenerovaný dotaz v jazyce SQL:

```
Console.WriteLine( db.GetCommand( query ).CommandText );
```

Jednodušší je zavolat pro dotaz LINQ pro SQL metodu *ToString*: přepsaná metoda *ToString* vrací tentýž výsledek jako příkaz *GetCommand(query).CommandText*.

```
Console.WriteLine( query );
```

Jednoduchý dotaz LINQ pro SQL ve výpisu 4.2 generuje následující dotaz SQL:

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[City]
FROM [Customers] AS [t0]
WHERE ([t0].[Country] = @p0) AND ([t0].[Region] = @p1)
```

Jiným způsobem, jak sledovat všechny příkazy SQL posílané do databáze, je přiřadit hodnotu do vlastnosti *Log* třídy *DataContext*:

```
db.Log = Console.Out;
```

V následujících pasážích se podrobněji podíváme na generování tříd entit pro LINQ pro SQL.

Externí mapování

Mapování mezi entitami LINQ pro SQL a databázovou strukturou je nutné popsat prostřednictvím informací v metadatech. Ve výpisu 4.1 jste viděli atributy v definici entity, které splňovaly toto pravidlo. Ale je také možné použít externí mapovací soubor XML, který bude popisovat třídy entit a nebude tak nutné pracovat s atributy. Mapovací soubor XML vypadá podobně jako následující ukázka:

```
<Database Name="Northwind">
   <Table Name="Products">
      <Type Name="Product">
         <Column Name="ProductID" Member="ProductID"
                   Storage="_ProductID" DbType="Int NOT NULL IDENTITY"
                   IsPrimaryKey="True" IsDbGenerated="True" />
```

Element *Type* definuje vztah mezi třídou entity a atribut *Member* elementu *Column* udává odpovídající název členu ve třídě entity pro případ, že by byl odlišný od názvu sloupce v tabulce. Standardně není atribut *Member* povinný a předpokládá se, že je stejný jako atribut *Name* elementu *Column*. Tento soubor XML obvykle mívá rozšíření názvu souboru *dbml* a generují jej některé nástroje popsané v kapitole 6, „Nástroje LINQ pro SQL".

Soubor XML lze načíst pomocí instance třídy *XmlMappingSource* generované voláním její statické metody *FromXml* a předáním této instance do odvozeného konstruktoru třídy *DataContext*. Práci s touto syntaxí ukazuje následující příklad:

```
string path = "Northwind.dbml";
XmlMappingSource prodMapping =
   XmlMappingSource.FromXml(File.ReadAllText(path));
Northwind db = new Northwind(
       "Database=Test_Northwind;Trusted_Connection=yes",
       prodMapping
   );
```

Jedním z možných využití této techniky je situace, kdy je potřeba mapovat různé databáze na specifický datový model. Databáze se mohou lišit v tabulkách a názvech polí (například lokalizovaná verze databáze). Obecně byste měli o této možnosti uvažovat tehdy, když potřebujete trochu uvolnit mapovací vazbu mezi třídami entit a fyzickou datovou strukturou v databázi.

Další informace

Podrobný popis syntaxe XML v souboru .dbml je mimo možnosti této knihy. Syntaxi popisují soubory LinqToSqlMapping.xsd a DbmlSchema.xsd, jež se nacházejí v adresáři Microsoft Visual Studio 9.0\Xml\Schemas ve složce Program Files na vašem počítači, máte-li nainstalováno Visual Studio 2008. Nemáte-li ani jeden z těchto souborů, můžete si zkopírovat kód ze stránky dokumentace tohoto produktu na adrese *http://msdn2.microsoft.com/en-us/library/bb386907.aspx* a *http://msdn2.microsoft.com/en-us/library/bb399400.aspx*.

Modelování dat

Množina tříd entit, které LINQ pro SQL potřebuje, je tenká abstraktní vrstva nad relačním modelem. Každá třída entity definuje tabulku dat, do níž se lze dotazovat a kterou lze měnit. Instance měněných entit mohou promítat provedené změny do dat v relační databázi. Možnosti aktualizace dat uvidíte v kapitole 5, „LINQ pro SQL: Správa dat“. V následujících odstavcích se naučíte vytvářet datový model pro LINQ pro SQL.

DataContext

Třída *DataContext* obstarává komunikaci mezi LINQ a externími relačními datovými zdroji. Každá instance má vlastnost *Connection*, jež směřuje na relační databázi. Je typu *IDbConnection*, a proto by neměla být specificky zaměřena na konkrétní databázový produkt. Nicméně implementace LINQ pro SQL podporuje pouze databáze Microsoft SQL Server. Volba mezi konkrétními verzemi SQL Serveru závisí pouze na připojovacím řetězci, předaném do konstruktoru třídy *DataContext*.

Důležité

Architektura LINQ pro SQL podporuje mnoho poskytovatelů dat, aby bylo možné mapovat různorodé základní relační databáze. Poskytovatel je třída s rozhraním *System.Data.Linq.Provider.IProvider*. Toto rozhraní je však deklarováno jako vnitřní a není zdokumentováno. Microsoft podporuje pouze poskytovatele pro Microsoft SQL Server. .NET Framework 3.5 podporuje SQL Server 2000 a SQL Server 2005 ve 32bitové i 64bitové verzi. V čase psaní této knihy je podporován i SQL Server Compact 3.5, avšak pouze ve 32bitové verzi, protože SQL Server Compact 3.5 pro 64bitové platformy zatím není dostupný. (Pravděpodobně se dočká podpory v následujícím vydání.) Další verze SQL Serveru budou patrně rovněž podporovány.

Třída *DataContext* používá informaci v metadatech k mapování fyzické struktury relačních dat, z níž vychází generování kódu SQL. Třídu *DataContext* lze rovněž použít k volání uložené procedury a k trvalému uložení změn v instancích tříd entit do relační databáze.

Třídy pro specializovaný přístup do konkrétní databáze lze ze třídy *DataContext* odvodit. Takové třídy nabízejí snazší přístup k relačním datům, včetně členů, které představují dostupné tabulky. Pole odkazující se na existující tabulky v databázi lze definovat prostou deklarací bez nutnosti konkrétní inicializace, což dokládá následující kód:

```
public class SampleDb : DataContext {
    public SampleDb(IDbConnection connection)
           : base( connection ) {}
    public SampleDb(string fileOrServerOrConnection)
           : base( fileOrServerOrConnection ) {}
    public SampleDb(IDbConnection connection, MappingSource mapping)
           : base( connection, mapping ) {}

    public Table<Customer> Customers;
}
```

Poznámka

Členy tabulky se inicializují automaticky v základním konstruktoru třídy *DataContext*, jenž zkoumá typ za běhu pomocí Reflection, vyhledává příslušné členy a inicializuje je na základě mapovacích metadat.

Třídy entit

Třída entity má dvě role. První je poskytovat metadata pro dotazovací stroj LINQ: pro tyto účely se nezakládá instance třídy entity. Druhou rolí je vytvářet úložiště pro data načtená ze zdroje relačních dat, uchovávat případné změny a zařizovat jejich přenos zpět do zdroje relačních dat.

Třídu entity tvoří libovolná definice referenčního typu s atributem *Table*. Pro tyto účely nelze použít strukturu (hodnotový typ). Atribut *Table* může mít parametr *Name*, který specifikuje název odpovídající tabulky v databázi. Jestliže parametr *Name* neexistuje, použije se standardně název třídy:

```
[Table(Name="Products")] public class Product { ... }
```

Poznámka

I když se běžně používá pojem *tabulka*, nic vám nebrání použít v parametru *Name* namísto tabulky aktualizovatelný pohled. Pohled, jejž aktualizovat nelze, je možné použít také, přinejmenším do chvíle, dokud se nepokusíte aktualizovat data bez použití dané třídy entity.

Uvnitř třídy entity může být libovolný počet členů libovolného typu. Při definici mapování mezi třídou entity a odpovídající tabulkou v databázi hrají roli pouze datové členy či vlastnosti s atributem *Column*:

```
[Column] public int ProductID;
```

Třída entity by měla mít jedinečný klíč. Tento klíč je nezbytný pro jedinečnou identitu (více o tématu později), aby bylo možné identifikovat odpovídající řádky v databázových tabulkách a generovat příkazy SQL pro aktualizaci dat. Jestliže nemáte v tabulce primární klíč, lze zakládat instance tříd entit, avšak tyto instance nebudou modifikovatelné. Logická hodnota *IsPrimaryKey* v atributu *Column* nastavená na *true* říká, že sloupec přísluší primárnímu klíči v tabulce. Jestliže se jako primární klíč používá složený klíč, všechny sloupce vytvářející tento primární klíč budou mít ve svých parametrech nastaveno *IsPrimaryKey=true*:

```
[Column(IsPrimaryKey=true)] public int ProductID;
```

Standardně se sloupec mapuje pomocí stejného názvu, jaký má člen, v němž se používá daný atribut *Column*. Můžete použít i odlišný název a zadat hodnotu parametru *Name*. Například následující člen *Price* odpovídá poli *UnitPrice* v tabulce v databázi:

```
[Column(Name="UnitPrice")] public decimal Price;
```

Chcete-li filtrovat přístup k datům pomocí přístupů ve vlastnostech členů, musíte do parametru *Storage* zadat příslušný člen základního úložiště. Jestliže vložíte parametr *Storage*, LINQ pro SQL obejde veřejný přístup k vlastnosti a bude interagovat přímo se základní hodnotou. Pochopení tohoto principu je velice důležité především v situaci, kdy chcete sledovat pouze úpravy prováděné ve vašem kódu, a nikoli operace čtení/zápisu, které provádí prostředí LINQ. V následujícím kódu se přistupuje k vlastnosti *ProductName* při každé operaci čtení/zápis ve vašem kódu; při spuštění operace LINQ se provede přímé čtení/zápis do datového členu *_ProductName*:

```
[Column(Storage="_ProductName")]
public string ProductName {
    get { return this._ProductName; }
    set { this.OnPropertyChanging("ProductName");
          this._ProductName = value;
          this.OnPropertyChanged("ProductName");
    }
}
```

Vztah mezi relačním typem a typem .NET se zakládá s předpokladem použití výchozího relačního typu, odpovídajícího použitému typu .NET. Kdykoliv potřebujete stanovit jiný typ, poslouží vám parametr *DBType*, který určuje platný typ pomocí platné syntaxe SQL pro daný relační datový zdroj. Tato vlastnost se používá pouze tehdy, když chcete vytvořit databázové schéma počínaje definicemi tříd entit (tento proces si popíšeme v kapitole 5):

```
[Column(DBType="NVARCHAR(20)")] public string QuantityPerUnit;
```

Jestliže hodnotu ve sloupci automaticky generuje databáze (což v SQL Serveru nabízí klíčové slovo IDENTITY), bude patrně potřeba synchronizovat člen třídy entity s generovanou hodnotou, a to vždy, když vložíte instanci entity do databáze. K tomu musíte nastavit parametr *IsDBGenerated* na *true* a budete muset také příslušným způsobem upravit *DBType* – například přidáním modifikátoru *IDENTITY* pro tabulky SQL Serveru:

```
[Column(DBType="INT NOT NULL IDENTITY",
    IsPrimaryKey=true, IsDBGenerated=true)]
public int ProductID;
```

Stojí za zmínku, že existuje také parametr *CanBeNull*. Lze v něm stanovit, že daná hodnota může být null, ale je nutné podotknout, že pokud chcete stanovit podobnou podmínku v databázi vytvořené v LINQ pro SQL, je stále nutné v zadání *DBType* používat klauzuli *NOT NULL*:

```
[Column(DBType="INT NOT NULL IDENTITY", CanBeNull=false,
    IsPrimaryKey=true, IsDBGenerated=true)]
public int ProductID;
```

Další parametry týkající se aktualizace dat jsou *AutoSync*, *Expression*, *IsVersion* a *UpdateCheck*. Podrobnější výklad parametrů *IsDBGenerated*, *IsVersion* a *UpdateCheck* si uvedeme v kapitole 5.

Dědičnost entit

Někdy obsahuje jediná tabulka více typů entit. Představte si například seznam kontaktů – některé mohou být na zákazníky, další na dodavatele a ostatní na zaměstnance společnosti. Z datového pohledu může mít každá taková entita určitá specifická pole. (Například zákazník může mít pole pro slevu, které nehraje roli u zaměstnanců a dodavatelů.) Z pohledu obchodní logiky může každá entita pracovat s odlišnými obchodními pravidly. Nejlepší způsob, jak modelovat tento druh dat v objektově orientovaném prostředí, je využít dědičnost a vytvořit hierarchii specializovaných tříd. LINQ pro SQL umožňuje vytvořit množinu tříd odvozených z jedné základní třídy a mapovat je na tutéž relační tabulku.

Součástí základní třídy hierarchie je atribut *InheritanceMapping*, jenž obsahuje odpovídající odvozené třídy vycházející z hodnoty speciálního *sloupce diskriminátoru*. Parametr *Code* obsahuje možnou hodnotu a parametr *Typ* určuje odpovídající odvozený typ. Sloupec diskriminátoru se definuje nastavením parametru *IsDiscriminator* na *true* v atributech sloupce.

Výpis 4.3 nabízí ukázku hierarchie založené na tabulce Contacts v ukázkové databázi Northwind.

Výpis 4.3 Hierarchie tříd založená na kontaktech

```
[Table(Name="Contacts")]
[InheritanceMapping(Code = "Customer", Type = typeof(CustomerContact))]
[InheritanceMapping(Code = "Supplier", Type = typeof(SupplierContact))]
[InheritanceMapping(Code = "Shipper", Type = typeof(ShipperContact))]
[InheritanceMapping(Code = "Employee", Type = typeof(Contact), IsDefault = true)]
public class Contact {
    [Column(IsPrimaryKey=true)] public int ContactID;
    [Column(Name="ContactName")] public string Name;
    [Column] public string Phone;
    [Column(IsDiscriminator = true)] public string ContactType;
```

```
}

public class CompanyContact : Contact {
    [Column(Name="CompanyName")] public string Company;
}

public class CustomerContact : CompanyContact {
}

public class SupplierContact : CompanyContact {
}

public class ShipperContact : CompanyContact {
    public string Shipper {
        get { return Company; }
        set { Company = value; }
    }
}
```

Základní třídou hierarchie je *Contact*. Je-li kontaktem *Customer*, *Supplier* nebo *Shipper*, odpovídající třídy se odvozují z mezičlánku *CompanyContact*, kde se nachází pole *Company* odpovídající sloupci CompanyName ve zdrojové tabulce. Přechodová třída *CompanyContact* je nezbytná, protože se nelze odkazovat na tentýž sloupec (CompanyName) ve více než jednom poli, i kdyby to bylo v odlišných třídách téže hierarchie. Třída *ShipperContact* obsahuje vlastnost *Shipper*, jež pracuje se stejnou hodnotou *Company*, ale s odlišným sémantickým významem.

Důležité

Tento přístup vyžaduje, abyste veškeré možné datové sloupce v celé hierarchii sloučili do jediné tabulky. Máte-li běžnou databázi, budete mít patrně data pro různé entity uložena v samostatných tabulkách. Můžete nadefinovat pohled a s hierarchií entit v LINQ pro SQL pracovat s jeho pomocí, ale aby bylo možné data aktualizovat, musí být aktualizovatelný i pohled.

Úroveň abstrakce, kterou umožňuje práce s různými třídami entit v téže hierarchii, dobře dokládají ukázkové dotazy ve výpisu 4.4. Dotaz *queryTyped* používá operátor *OfType*, zatímco dotaz *queryFiltered* vychází z běžné podmínky *where*, která odfiltrovává nezákaznické kontakty.

Výpis 4.4 Dotazy s hierarchií tříd entit

```
var queryTyped =
    from c in contacts.OfType<CustomerContact>()
    select c;

var queryFiltered =
    from c in contacts
    where c is CustomerContact
    select c;

foreach( var row in queryTyped ) {
    Console.WriteLine( row.Company );
}
```

```
// Potřebujeme explicitní přetypování, abychom mohli přistupovat ke členům //
CustomerContact.
foreach( CustomerContact row in queryFiltered ) {
    Console.WriteLine( row.Company );
}
```

Dotazy SQL, vygenerované z těchto dotazů LINQ, jsou funkčně totožné s následujícím. (Skutečný dotaz se odlišuje kvůli zobecněnému kódu.)

```
SELECT [t0].[ContactType], [t0].[CompanyName] AS [Company],
   [t0].[ContactID], [t0].[ContactName] AS [Name],
   [t0].[Phone]
FROM [Contacts] AS [t0]
WHERE [t0].[ContactType] = 'Customer'
```

Rozdíl mezi dotazy *queryType* a *queryFiltered* leží v návratovém typu. Dotaz *queryTyped* vrací sekvenci instancí třídy *CustomerContact*, zatímco dotaz *queryFiltered* vrací sekvenci základní třídy *Contact*. V dotazu *queryFiltered* musíme explicitně přetypovat výsledek na typ *CustomerContact*, pokud chceme přistupovat k vlastnosti *Company*.

Shoda jedinečného objektu

Instance třídy entity uchovává v paměti reprezentaci řádku dat v tabulce. Jestliže se pokusíte založit dvě různé entity obsahující stejný řádek z téže instance *DataContext*, dostanete odkaz na tentýž objekt v paměti. Jinými slovy, shoda objektu (stejné odkazy) zachovává shodu dat (tentýž řádek v tabulce) pomocí jedinečného klíče entity. LINQ pro SQL zajišťuje, že se použije tentýž odkaz na objekt, jestliže se zakládá instance entity založená na výsledku dotazu pocházejícího z téže instance *DataContext*. Toto ověřování neprobíhá, jestliže vytváříte instanci entity samostatně nebo v odlišné instanci *DataContext* (bez ohledu na skutečný datový zdroj). Ve výpisu 4.5 vidíte, že *c1* a *c2* se odkazují na stejnou instanci třídy *Contact*, i když pocházejí ze dvou různých dotazů, zatímco *c3* je odlišný objekt, i když je jeho obsah totožný s ostatními.

Poznámka

Jestliže potřebujete znovu načíst data z databáze ve stejné instanci *DataContext*, musíte použít metodu *Refresh* třídy *DataContext*. Více o tomto tématu naleznete v kapitole 5.

Výpis 4.5 Shoda objektů

```
var queryTyped =
    from c in contacts.OfType<CustomerContact>()
    orderby c.ContactID
    select c;

var queryFiltered =
    from c in contacts
    where c is CustomerContact
    orderby c.ContactID
    select c;

Contact c1 = null;
Contact c2 = null;
```

```
foreach( var row in queryTyped.Take(1) ) {
    c1 = row;
}
foreach( var row in queryFiltered.Take(1) ) {
    c2 = row;
}
Contact c3 = new Contact();
c3.ContactID = c1.ContactID;
c3.ContactType = c1.ContactType;
c3.Name = c1.Name;
c3.Phone = c1.Phone;
Debug.Assert( c1 == c2 ); // stejná instance
Debug.Assert( c1 != c3 ); // odlišné objekty
```

Omezení entit

Třídy entit podporují udržování platných vztahů mezi entitami, podobně jako je tomu s cizími klíči v běžném relačním prostředí. Ale třídy entit nemohou pracovat se všemi možnými omezeními pro relační tabulky. Neexistují žádné atributy, které by specifikovaly klíče (jedinečná omezení), spouště a ověřovací výrazy, které lze definovat v relační databázi. Tento fakt hraje roli, když začínáte pracovat s daty pomocí tříd entit, protože nelze zaručit, že databáze aktualizovanou hodnotu přijme. (Může se kupříkladu vyskytnout duplicita v jedinečném klíči.) Avšak vzhledem k tomu, že do instancí entit lze načíst pouze části (řádky) celé tabulky, nejsou stejně tyto druhy kontrol možné bez přístupu do relační databáze.

Vztahy mezi entitami

Vztahy mezi entitami v relačními databázi se modelují na principu cizích klíčů, odkazujících se na primární klíče v určité tabulce. Třídy entit umějí používat stejný princip pomocí atributu *Association*, který umí popisovat obě strany vztahu *jedna-mnoho*, vyjádřeného cizím klíčem.

EntityRef

Začněme principem *vyhledávání* (lookup), což je typická operace používaná ke zjištění zákazníka souvisejícího s jednou objednávkou. Na vyhledávání lze nahlížet jako na přímý překlad do modelu entit se vztahem prostřednictvím cizího klíče mezi sloupcem CustomerID v tabulce Orders a primárním klíčem tabulky Customers. V našem modelu entit bude mít třída entity *Order* atribut *Association* a příslušnou informaci bude uchovávat ve členu *EntityRef<Customer>* (s názvem *_Customer)*, jenž umožňuje odložené načítání odkazu, což uvidíte zakrátko. Definici tohoto vztahu vidíte ve výpisu 4.6.

Výpis 4.6 Association EntityRef

```
[Table(Name="Orders")]
public class Order {
    [Column(IsPrimaryKey=true)] public int OrderID;
    [Column] private string CustomerID;
    [Column] public DateTime? OrderDate;

    [Association(Storage="_Customer", ThisKey="CustomerID", IsForeignKey=true)]
    public Customer Customer {
        get { return this._Customer.Entity; }
        set { this._Customer.Entity = value; }
}
```

```
private EntityRef<Customer> _Customer;
}
```

Je zjevné, že sloupec CustomerID musí být definován ve třídě *Order*, neboť v opačném případě by nebylo možné získat související zákazníka. Parametr *IsForeignKey* říká, že jsme na podřízené straně vztahu nadřízený-podřízený, a parametr *ThisKey* atributu *Association* určuje sloupec „cizího klíče" (což bude v případě, že se klíč skládá z více sloupců, seznam oddělený čárkami), který slouží ke stanovení vztahu mezi entitami. Chcete-li tento detail skrýt ve vlastnostech entity, můžete daný sloupec deklarovat jako soukromý, stejně jako ve dříve uvedené třídě *Order*.

Poznámka

Atribut *Association* má ještě další dva parametry. Jedním je *IsUnique*, jenž musí být nastaven na true, jestliže je v cizím klíči rovněž požadována jedinečnost. V takovém případě je vztah s nadřízenou tabulkou typu jedna-jedna namísto mnoho-jedna. Druhým parametrem je *Name* a používá se pouze k definici názvu omezení, pakliže se má z metadat pomocí metody *DataContext.CreateDatabase* generovat databáze, což si ukážeme v kapitole 5.

Pomocí třídy *Order* můžete v dotazu LINQ zadat vlastnost *Customer* do filtru, aniž by bylo nutné psát spojení mezi entitami *Customer* a *Order*. V následujícím dotazu se používá člen *Country* navázaného zákazníka, čímž se vyfiltrují objednávky pocházející od zákazníků v konkrétní zemi:

```
Table<Order> Orders = db.GetTable<Order>();
var query =
   from o in Orders
   where o.Customer.Country == "USA"
   select o.OrderID;
```

Uvedený dotaz se přeloží do příkazu SQL JOIN následujícím způsobem:

```
SELECT [t0].[OrderID]
FROM [Orders] AS [t0]
LEFT JOIN [Customers] AS [t1]
   ON [t1].[CustomerID] = [t0].[CustomerID]
WHERE [t1].[Country] = "USA"
```

Až dosud jsme používali vztahy mezi entitami pouze kvůli jejich metadatům při vytváření dotazů LINQ. Když dojde k založení instance třídy entity, pracuje odkaz na jinou entitu (například výše uvedená vlastnost *Customer*) pomocí techniky zvané *odložené načítání*. Instance související entity *Customer* se nezaloží a nenačte z databáze do paměti až do chvíle, kdy dojde k přístupu k ní v režimu pro čtení či zápis.

Poznámka

***EntityRef<T>* je třída obálky, jejíž instance se zakládá v objektu kontejneru (třída odvozená ze třídy *DataContext*) a poskytuje platný odkaz pro veškerý přístup k entitě, na niž se odkazujeme. Každá operace čtení i zápisu se filtruje v blocích vlastnosti *get* a *set*, které spouštějí dotaz pro načítání z databáze při prvním přístupu k této entitě, pokud již není v paměti.**

Jinými slovy, ke generování dotazu SQL, který naplní entitu navázanou na objekt *Customer* při práci s vlastností *Country*, vám poslouží následující kód:

```
var query =
    from o in Orders
    where o.OrderID == 10528
    select o;

foreach( var row in query ) {
    Console.WriteLine( row.Customer.Country );
}
```

Proces přístupu k vlastnosti *Customer* zahrnuje ověření, zdali je související entita *Customer* v aktuálním prostředí *DataContext* již načtena do paměti. Jestliže je, přistupuje se k této entitě, v opačném případě se provede následující dotaz SQL, do paměti se načte odpovídající entita *Customer* a poté dojde k přístupu k ní:

```
SELECT [t0].[Country], [t0].[CustomerID], [t0].[CompanyName]
FROM [Customers] AS [t0]
WHERE [t0].[CustomerID] = "GREAL"
```

Řetězec GREAL je hodnota *CustomerID* pro objednávku 10528. Jak vidíte, příkaz SELECT se dotazuje do všech sloupců deklarovaných v entitě *Customer*, i když se ve výrazu, který vyžaduje entitu *Customer*, nepoužívají. (V tomto případě se prováděný kód nikdy neodkazoval na člen *CompanyName*.)

EntitySet

Na druhé straně vztahu stojí tabulka, na niž se odkazuje jiná tabulka prostřednictvím svého primárního klíče. I když v relačním modelu jde o implicitní důsledek omezení cizím klíčem, v modelu entit je potřeba tento vztah explicitně definovat. Jestliže se na tabulku Customers odkazuje tabulka Orders, lze ve třídě *Customer* vytvořit vlastnost *Orders*, která představuje množinu entit *Order* souvisejících s příslušným zákazníkem. Vztah se zadává pomocí instance třídy obálky *EntitySet*<*Order*> nad sekvencí souvisejících objednávek. Typ *EntitySet*<*T*> lze přímo zveřejnit, což je případ kódu ve výpisu 4.7. Zde parametr *OtherKey* atributu *Association* udává název členu souvisejícího typu (*Order*), který určuje vazbu mezi třídou *Customer* a množinou entit *Order*.

Výpis 4.7 Association EntitySet (viditelné navenek)

```
[Table(Name="Customers")]
public class Customer {
    [Column(IsPrimaryKey=true)] public string CustomerID;
    [Column] public string CompanyName;
    [Column] public string Country;

    [Association(OtherKey="CustomerID")]
    public EntitySet<Order> Orders;
}
```

Můžete se rovněž rozhodnout zveřejnit objednávky jako vlastnost, což ukazuje výpis 4.8. V tomto případě stanovuje parametr *Storage* atributu *Association* množinu *EntitySet*<*T*> jako fyzické úložiště. Můžete se rovněž rozhodnout zviditelnit vně třídy *Customer* pouze kolekci *ICollection*<*Order*> namísto množiny *EntitySet*<*Order*>, není to však běžná praxe.

Výpis 4.8 Association EntitySet (skryté)

```
public class Customer {
    [Column(IsPrimaryKey=true)] public string CustomerID;
    [Column] public string CompanyName;
    [Column] public string Country;

    private EntitySet<Order> _Orders;

    [Association(OtherKey="CustomerID", Storage="_Orders")]
    public EntitySet<Order> Orders {
        get { return this._Orders; }
        set { this._Orders.Assign(value); }
}
    public Customer() {
        this._Orders = new EntitySet<Order>();
    }
}
```

V obou modelech deklarace vztahu můžete použít třídu *Customer* v dotazu LINQ a přistupovat k souvisejícím entitám *Order*, aniž by bylo nutné psát spojení. Stačí použít vlastnost *Orders*. Následující dotaz vrací jména zákazníků, kteří učinili více než 20 objednávek:

```
Table<Customer> Customers = db.GetTable<Customer>();
var query =
   from c in Customers
   where c.Orders.Count > 20
   select c.CompanyName;
```

Předchozí dotaz LINQ se přeloží do dotazu SQL podobného tomuto:

```
SELECT [t0].[CompanyName]
FROM [Customers] AS [t0]
WHERE ( SELECT COUNT(*)
        FROM [Orders] AS [t1]
        WHERE [t1].[CustomerID] = [t0].[CustomerID]
        ) > 20
```

V tomto případě se nevytvářejí žádné instance entity *Order*. Vlastnost *Orders* slouží pouze jako zdroj metadat, s jehož pomocí se generuje požadovaný dotaz SQL. Jestliže vrátíte z dotazu LINQ entitu *Customer*, můžete podle potřeby přistupovat k objednávkám určitého zákazníka:

```
var query =
   from c in Customers
   where c.Orders.Count > 20
   select c;

foreach( var row in query ) {
   Console.WriteLine( row.CompanyName );
   foreach( var order in row.Orders ) {
      Console.WriteLine( order.OrderID );
   }
}
```

V předcházejícím kódu využíváme odložené načítání. Pokaždé, když přistupujete k vlastnosti *Orders* určitého zákazníka poprvé (což ukazuje zvýrazněný kód), do databáze se pošle dotaz podobný následujícímu (který používá *@p0* jako parametr pro filtrování hodnot *CustomerID*):

```
SELECT [t0].[OrderID], [t0].[CustomerID]
FROM [Orders] AS [t0]
WHERE [t0].[CustomerID] = @p0
```

Chcete-li načíst veškeré objednávky pro všechny zákazníky do paměti jedním dotazem do databáze, musíte si namísto odloženého načítání vyžádat *okamžité načítání*. Máte pro to dvě možnosti. První přístup ukazuje výpis 4.9, kde vynutíte práci s množinou entit pomocí instance třídy *DataLoadOptions* a zavoláním její metody *LoadWith<T>*.

Výpis 4.9 Použití třídy DataLoadOptions a metody LoadWith<T>

```
DataContext db = new DataContext( ConnectionString );
Table<Customer> Customers = db.GetTable<Customer>();

DataLoadOptions loadOptions = new DataLoadOptions();
loadOptions.LoadWith<Customer>( c => c.Orders );
db.LoadOptions = loadOptions;
var query =
    from   c in Customers
    where  c.Orders.Count > 20
    select c;
```

Druhou možností je výběr nové entity, která explicitně obsahuje vlastnost *Orders* pro daného zákazníka:

```
var query =
   from c in Customers
   where c.Orders.Count > 20
   select new { c.CompanyName, c.Orders };
```

Tyto dotazy LINQ posílají dotaz SQL do databáze a načítají všechny zákazníky, kteří učinili více než 20 objednávek, včetně úplného seznamu objednávek pro každého zákazníka. Příslušný dotaz SQL by mohl vypadat asi jako následující kód:

```
SELECT [t0].[CompanyName], [t1].[OrderID], [t1].[CustomerID], (
   SELECT COUNT(*)
   FROM [Orders] AS [t3]
   WHERE [t3].[CustomerID] = [t0].[CustomerID]
   ) AS [value]
FROM [Customers] AS [t0]
LEFT OUTER JOIN [Orders] AS [t1] ON [t1].[CustomerID] = [t0].[CustomerID]
WHERE (
   SELECT COUNT(*)
   FROM [Orders] AS [t2]
   WHERE [t2].[CustomerID] = [t0].[CustomerID]
   ) > 20
ORDER BY [t0].[CustomerID], [t1].[OrderID]
```

Poznámka

Všimněte si, že zde máme jediný dotaz SQL a stroj LINQ pro SQL výsledky rozděluje – vybírá odlišné entity (Customers a Orders). Díky zachování řazení výsledků podle *CustomerID* může LINQ pro SQL vytvářet entity a vztahy v paměti rychleji.

Poddotaz získaný ze vztahu lze filtrovat. Předpokládejme, že chcete vidět pouze zákazníky, kteří v roce 1997 uskutečnili nejméně pět objednávek, a chcete načíst a vidět pouze tyto

objednávky. K tomu vám poslouží metoda *AssociateWith<T>* třídy *DataLoadOptions*, což demonstruje výpis 4.10.

Výpis 4.10 Použití třídy DataLoadOptions a metody AssociateWith<T>

```
DataLoadOptions loadOptions = new DataLoadOptions();
loadOptions.AssociateWith<Customer>(
    c => from o in c.Orders
        where o.OrderDate.Value.Year == 1997
        select o);
db.LoadOptions = loadOptions;
var query =
     from c in Customers
     where c.Orders.Count > 5
     select c;
```

Dozajista oceníte, že filtrovací podmínka v C# (*o.OrderDate.Value.Yer == 1997*) se překládá na následující výraz SQL:

```
(DATEPART(Year, [t2].[OrderDate]) = 1997)
```

Metoda *AssociateWith<T>* může také řídit počáteční řazení kolekce. Stačí k tomu přidat podmínku řazení do dotazu předávaného do parametru metody *AssociateWith<T>*. Chcete-li například načíst objednávky pro každého zákazníka počínaje nejnovějšími, přidejte řádek *orderby*, zvýrazněný v následujícím kódu:

```
loadOptions.AssociateWith<Customer>(
   c => from o in c.Orders
       where o.OrderDate.Value.Year == 1997
       orderby o.OrderDate descending
       select o);
```

Samotné použití metody *AssociateWith<T>* nezaručuje okamžité načítání. Chcete-li okamžité načítání i filtrování pomocí vztahu, musíte zavolat metody *LoadWith<T>* i *AssociateWith<T>*. Na pořadí těchto volání nezáleží. Můžete například napsat takovýto kód:

```
DataLoadOptions loadOptions = new DataLoadOptions();
loadOptions.AssociateWith<Customer>(
   c => from o in c.Orders
       where o.OrderDate.Value.Year == 1997
       select o);
loadOptions.LoadWith<Customer>( c => c.Orders );
db.LoadOptions = loadOptions;
```

Načítání veškerých dat do paměti pomocí jediného dotazu může být lepší v situaci, kdy jistě víte, že budete přistupovat ke všem načteným datům, protože vás méně zdrží zpoždění při načítání. Ale uvedená technika spotřebuje více paměti a síťové kapacity než při typickém náhodném přístupu ke grafu entit. Při rozhodování o dotazech do svého datového modelu mějte tyto detaily na mysli.

Konzistence grafu

Vztahy mezi entitami jsou obousměrné – dojde-li k aktualizaci na jedné straně, druhou stranu je potřeba udržet v synchronizaci. LINQ pro SQL neprovádí tento druh synchronizace automaticky a je potřeba jej zařídit při implementaci tříd entit. LINQ pro SQL nabízí implementační postup, který používají i nástroje pro generování kódu jako SQLMetal, jenž je součástí sady Microsoft .NET 3.5 Software Development Kit (SDK), či generátor tříd pro

LINQ pro SQL, který přináleží do Visual Studia 2008. Oba tyto nástroje si popíšeme v kapitole 6. Uvedený postup vychází ze třídy *EntitySet<T>* na jedné straně a ze složitého bloku *set* na straně druhé. Zajímají-li vás implementační podrobnosti tohoto postupu, podívejte se na kód generovaný těmito nástroji.

Upozornění na změny

V kapitole 5 uvidíte, že LINQ pro SQL umí sledovat změny v entitách a posílat příslušné změny do databáze. Tento proces probíhá automaticky prostřednictvím algoritmu, který srovnává obsah objektu s jeho původními hodnotami, což vyžaduje kopii každého sledovaného objektu. Spotřeba paměti může být vysoká, ale lze ji optimalizovat, pokud se do služby sledování změn zapojí i entity a dají vědět, když dojde v objektu ke změně.

Implementace upozorňování na změny vyžaduje, aby příslušná entita zveřejnila veškerá svá data ve vlastnostech v rozhraní *System.ComponentModel.INotifyPropertyChanging*. Každý blok *set* ve vlastnosti musí volat metodu *PropertyChanging* třídy *DataContext*. Další detaily naleznete v dokumentaci produktu. Strojově generovaný kód entit (například prostřednictvím programu SQLMetal či ve Visual Studiu 2008) již toto chování obsahuje.

Srovnání relačního modelu a hierarchického modelu

Model entit, používaný v LINQ pro SQL, definuje množinu objektů, které mapují databázové tabulky na objekty, jež lze používat a obsluhovat v dotazech LINQ. Výsledný model představuje změnu v přístupu, kterou jsme si ukázali při popisu vztahů mezi entitami. Posunuli jsme se od relačního modelu (tabulky v databázi) k hierarchickému či grafovému modelu (objekty v paměti).

Hierarchický/grafový model představuje přirozený způsob práce s objekty v programu napsaném v jazycích C# či Microsoft Visual Basic. Když se pokusíte přeložit existující dotaz SQL do dotazu LINQ, narazíte na významnou principiální překážku. V LINQ můžete napsat dotaz používající spojení mezi samostatnými entitami, podobně jako v SQL. Ale lze také napsat dotaz využívající existující vztahy mezi entitami, což jsme dělali pomocí vztahů *EntitySet* a *EntityRef*.

Důležité

Pamatujte si, že SQL nepoužívá při dotazech na data vztahy mezi entitami. Tyto vztahy existují pouze kvůli definici podmínek datové integrity. LINQ neobsahuje princip *referenční integrity*, ale využívá vztahy pro definici možných navigačních cest k datům.

Dotazování na data

Dotaz LINQ pro SQL se posílá do databáze pouze tehdy, když program potřebuje načítat data. Například následující smyčka *foreach* prochází řádky navrácené z tabulky:

```
var query =
    from c in Customers
    where c.Country == "USA"
    select c.CompanyName;
```

```
foreach( var company in query ) {
    Console.WriteLine( company );
}
```

Kód generovaný příkazem *foreach* je totožný s následujícím kódem. Přesný okamžik provádění dotazu odpovídá volání metody *GetEnumerator*:

```
// GetEnumerator posílá dotaz do databáze
IEnumerator<string> enumerator = query.GetEnumerator();
while (enumerator.MoveNext()) {
    Console.WriteLine( enumerator.Current );
}
```

Více smyček *foreach* jednom dotazu znamená odpovídající počet volání metody *GetEnumerator* a tedy odpovídající počet opakovaného provádění téhož dotazu. Chcete-li procházet tatáž data vícekrát, dáte patrně přednost uložení dat do mezipaměti. Pomocí příkazů *ToList* či *ToArray* převedete výsledky dotazu na seznam či pole. Když zavoláte tyto metody, pošle se do databáze dotaz SQL:

```
// ToList() posílá dotaz do databáze var companyNames = query.ToList();
```

Posílat dotaz do databáze vícekrát je potřeba, když s dotazem LINQ mezi iteracemi manipulujete. Můžete mít kupříkladu interaktivní uživatelské rozhraní, které uživateli umožňuje přidat pro každý průchod daty nový filtr. Ve výpisu 4.11 zobrazuje metoda *DisplayTop* pouze několik prvních řádků výsledku; manipulace s dotazem mezi voláními metody *DisplayTop* simuluje uživatelský zásah, který vede vždy k nové podmínce ve filtru.

Více informací

Výpis 4.11 ukazuje velmi prostou techniku manipulace s dotazem, která přidává další omezující podmínky k existujícímu dotazu, reprezentovanému objektem *IQueryable<T>*. Kapitola 11 popisuje metody, jak dynamicky vystavět strom dotazu pružnějším způsobem.

Výpis 4.11 Manipulace s dotazem

```
static void QueryManipulation() {
    DataContext db = new DataContext( ConnectionString );
    Table<Customer> Customers = db.GetTable<Customer>();
    db.Log = Console.Out;

    // všichni zákazníci
    var query =
        from c in Customers
        select new {c.CompanyName, c.State, c.Country };

    DisplayTop( query, 10 );

    // Uživatel přidává
    // k předchozímu dotazu filtr.
    // zákazníci z USA
    query =
        from c in query
        where c.Country == "USA"
        select c;
```

```
    DisplayTop( query, 10 );

    // Uživatel přidává k předchozímu
    // dotazu další filtr.
    // zákazníci z USA, Washington
    query =
    from c in query
    where c.State == "WA"
    select c;

    DisplayTop( query, 10 );
}

static void DisplayTop<T>( IQueryable<T> query, int rows ) {
    foreach( var row in query.Take(rows)) {
        Console.WriteLine( row );
    }
}
```

Důležité

V předchozí ukázce jsme jako parametr metody *DisplayTop* použili typ *IQueryable<T>*. Kdybychom namísto toho použili typ *IEnumerable<T>*, výsledky by vypadaly stejně, ale dotaz poslaný do databáze by neobsahoval klauzuli TOP (*řádky*), která filtruje data přímo v databázi. Při práci s typem *IEnumerable<T>* se používá odlišná množina rozšiřujících metod k rozřešení operátoru *Take* bez generování nového stromu výrazu. Popis rozdílů mezi typy *IEnumerable<T>* a *IQueryable<T>* naleznete v kapitole 2, „Základy syntaxe LINQ".

Běžný dotaz, používaný pro přístup k databázi, je načítání jediného řádku z tabulky, kdy definujeme podmínku zaručující jedinečnost, kupříkladu pomocí klíče záznamu. Typický dotaz vypadá takto:

```
var query =
    from c in db.Customers
    where c.CustomerID == "ANATR"
    select c;
var enumerator = query.GetEnumerator();
if (enumerator.MoveNext()) {
    var customer = enumerator.Current;
    Console.WriteLine( "{0} {1}", customer.CustomerID, customer.CompanyName );
}
```

V tomto případě by bylo kratší a jasnější zapsat svůj záměr pomocí operátoru *Single*. Předchozí dotaz lze přepsat do této kompaktnější formy:

```
var customer = db.Customers.Single( c => c.CustomerID == "ANATR" );
Console.WriteLine( "{0} {1}", customer.CustomerID, customer.CompanyName );
```

Je však důležité si uvědomit, že volání *Single* má odlišnou sémantiku než předchozí, ekvivalentní dotaz. Volání operátoru *Single* generuje dotaz do databáze pouze tehdy, jestliže požadovaná entita (v tomto případě objekt typu *Customer* s *CustomerID* rovno *ANATR*) již není v paměti. Chcete-li načíst data z databáze, musíte zavolat metodu *DataContext.Refresh*:

```
db.Refresh(RefreshMode.OverwriteCurrentValues, customer);
```

Více informací o životním cyklu entity naleznete v kapitole 5.

Projekce

Transformace ze stromu výrazů na dotaz SQL vyžaduje úplné pochopení dotazovacích operací poslaných do stroje LINQ pro SQL. Tato transformace ovlivňuje použití inicializátorů objektů. K projekcím vám poslouží klíčové slovo *select*, což ukazuje následující příklad:

```
var query =
   from c in Customers
   where c.Country == "USA"
   select new {c.CustomerID, Name = c.CompanyName.ToUpper()} into r
   orderby r.Name
   select r;
```

Celý dotaz LINQ se přeloží na tento příkaz SQL:

```
SELECT [t1].[CustomerID], [t1].[value] AS [Name]
FROM ( SELECT [t0].[CustomerID],
                    UPPER([t0].[CompanyName]) AS [value],
                    [t0].[Country]
          FROM [Customers] AS [t0]
       ) AS [t1]
WHERE [t1].[Country] = "USA"
ORDER BY [t1].[value]
```

Jak vidíte, metoda *ToUpper* byla přeložena na volání funkce T-SQL UPPER. Stroj LINQ pro SQL musí proto velmi dobře znát význam veškerých operací ve stromu výrazů. Podívejte se na tento dotaz:

```
var queryBad =
   from c in Customers
   where c.Country == "USA"
   select new CustomerData( c.CustomerID, c.CompanyName.ToUpper()) into r
   orderby r.Name
   select r;
```

V tomto případě voláme konstruktor typu *CustomerData*, který může dělat cokoliv, co zvládne kousek kódu v převodním jazyce (Intermediate Language, IL). Jinými slovy, neexistuje žádná jiná sémantika volání konstruktoru než počáteční přiřazení vytvořené instance. Následkem je skutečnost, že LINQ pro SQL nemůže tuto syntaxi korektně přeložit na ekvivalentní kód SQL a při spuštění tohoto dotazu vypíše výjimku. Parametrizovaný konstruktor je však možné bezpečně použít v závěrečné projekci dotazu, což dokládá následující ukázka:

```
var queryParamConstructor =
   from c in Customers
   where c.Country == "USA"
   orderby c.CompanyName
   select new CustomerData( c.CustomerID, c.CompanyName.ToUpper() );
```

Pokud pouze potřebujete inicializovat objekt, použijte namísto parametrizovaného konstruktoru inicializátory objektu, podobně jako v dalším dotazu:

```
var queryGood =
   from c in Customers
   where c.Country == "USA"
   select new CustomerData { CustomerID = c.CustomerID,
                                    Name = c.CompanyName.ToUpper() } into r
   orderby r.Name
   select r;
```

> **Důležité**
>
> Pro kódování projekcí v LINQ pro SQL vždy používejte inicializátory objektu. Parametrizované konstruktory používejte pouze pro závěrečnou projekci v dotazu.

Uložené procedury a uživatelské funkce

Přístup k datům pomocí uložených procedur a uživatelských funkcí (UDF) vyžaduje definici odpovídajících metod s atributy. Umožňuje to psát dotazy LINQ se silnou typovou kontrolou. Z pohledu LINQ není rozdíl v tom, je-li uložená procedura či UDF napsána v jazyce T-SQL či SQLCLR, musíte však znát určité podrobnosti, máte-li zvládnout rozdíly mezi uloženými procedurami a uživatelskými funkcemi.

> **Poznámka**
>
> Když uvážíme, že mnoho z vás bude automaticky vytvářet specializované třídy odvozené ze třídy *DataContext*, soustředíme se na nejvýznamnější principy, které potřebujete znát pro efektivní používání těchto objektů. Chcete-li ručně vytvářet tyto obálky, podívejte se do dokumentace produktu, kde naleznete podrobný seznam atributů a jejich parametrů.

VLožené procedury

Podívejte se na vloženou proceduru *Customers by City*:

```
CREATE PROCEDURE [dbo].[Customers By City]( @param1 NVARCHAR(20) )
AS BEGIN
    SET NOCOUNT ON;
    SELECT CustomerID, ContactName, CompanyName, City
    FROM Customers AS c
    WHERE c.City = @param1
END
```

Můžete definovat metodu s atributem *Function*, která volá tuto vloženou proceduru pomocí metody *ExecuteMethodCall* třídy *DataContext*. Ve výpisu 4.12 nadefinujeme *CustomerByCity* jako člen třídy odvozené ze třídy *DataContext*:

Výpis 4.12 Deklarace uložené procedury

```
class SampleDb : DataContext {
    // ...
    [Function(Name = "Customers by City", IsComposable = false)]
    public ISingleResult<CustomerInfo> CustomersByCity(string param1) {
        IExecuteResult executeResult =
            this.ExecuteMethodCall(
                    this,
                    (MethodInfo) (MethodInfo.GetCurrentMethod()),
                    param1);
        ISingleResult<CustomerInfo> result =
            (ISingleResult<CustomerInfo>) executeResult.ReturnValue;
        return result;
    }
}
```

Metoda *ExecuteMethodCall* má následující deklaraci:

```
IExecuteResult ExecuteMethodCall( object instance,
                                  MethodInfo methodInfo,
                                  params object[] parameters)
```

Prvním parametrem metody je instance (která není nutná, voláte-li statickou metodu). Druhým parametrem je popis metadat volané metody, který lze získat skrze Reflection, jako ve výpisu 4.12. Třetím parametrem je pole s parametry předávanými do volané metody.

Metoda *CustomersByCity* vrací instanci *ISingleResult<CustomerInfo>*, která obsahuje rozhraní *IEnumerable<CustomerInfo>* a lze ji vypsat ve smyčce *foreach*, jako je například tato:

```
SampleDb db = new SampleDb( ConnectionString );
foreach( var row in db.CustomersByCity( "London" )) {
   Console.WriteLine( "{0} {1}", row.CustomerID, row.CompanyName );
}
```

Z výpisu 4.12 vidíme, že jsme mohli přistupovat k rozhraní *IExecuteResult*, navrácenému metodou *ExecuteMetodCall*, a získat tak potřebné výsledky. Uvedený postup vyžaduje další výklad. Tentýž atribut *Function* používáme pro metodu tvořící obálku uložené procedury i UDF. Rozlišující faktor mezi těmito dvěma konstrukcemi představuje parametr *IsComposable* atributu *Function*: je-li nastaven na *false*, následná metoda tvoří obálku uložené procedury, je-li roven *true*, bude se volat uživatelská funkce.

Poznámka

Název *IsComposable* se odkazuje na kompozitní chování uživatelských funkcí v dotazovacím výrazu. Ukázku uvidíte, až si budeme v další části kapitoly popisovat mapování uživatelských funkcí.

Rozhraní *IExecuteResult* má prostou definici:

```
public interface IExecuteResult : IDisposable {
   object GetParameterValue(int parameterIndex);
   object ReturnValue { get; }
}
```

Metoda *GetParameterValue* umožňuje přistupovat k výstupním parametrům vložené procedury. Tento výsledek je nutné převést na správný typ a do parametru *parameterIndex* vložit ordinální pozici výstupního parametru.

Vlastnost *ReturnValue*, určená pouze pro čtení, slouží k přístupu k návratové hodnotě vložené procedury či UDF. Skalární návratovou hodnotu lze používat po převodu na správný typ: vložená procedura vždy vrací celé číslo, kdežto typ uživatelské funkce se může různit. Ale je-li výsledkem tabulka, používá se pro přístup k jediné výsledkové množině rozhraní *ISingleResult<T>,* zatímco pro přístup k vícenásobným výsledkovým množinám slouží rozhraní *IMultipleResults*.

Vždy je potřeba znát metadata všech možných návratových výsledkových množin a na generická rozhraní, sloužící k návratu dat, aplikovat správný typ. *ISingleResult<T>* je jednoduchá obálka rozhraní *IEnumerable<T>*, které rovněž obsahuje rozhraní *IFunctionResult*, jež má vlastnost *ReturnValue* určenou pouze pro čtení, která funguje jako vlastnost *IExecuteResult.ReturnValue*, kterou jsme již viděli:

```
public interface IFunctionResult {
    object ReturnValue { get; }
}
public interface ISingleResult<T> :
    IEnumerable<T>, IEnumerable, IFunctionResult, IDisposable { }
```

Ukázku použití rozhraní *ISingleResult<T>* jste viděli ve výpisu 4.12. Obálku *CustomersByCity* jsme zapsali poněkud rozvláčně, pro lepší ilustraci vnitřních kroků, jež jsou nezbytné pro přístup k návratovým datům.

Kdykoliv vaše vložená procedura vrací vícenásobné výsledkové množiny, volejte sekvenčně metodu *IMultipleResult.GetResult<T>* se správným typem *T* očekávaného výsledku. Rozhraní *IMultipleResults* také obsahuje rozhraní *IFunctionResults* a nabízí tedy rovněž vlastnost *ReturnValue* (pouze pro čtení).

```
public interface IMultipleResults : IFunctionResult, IDisposable {
    IEnumerable<TElement> GetResult<TElement>();
}
```

Podívejte se na následující vloženou proceduru, která vrací dvě výsledkové množiny s odlišnou strukturou:

```
CREATE PROCEDURE TwoCustomerGroups
AS BEGIN
    SELECT CustomerID, ContactName, CompanyName, City
    FROM Customers AS c
    WHERE c.City = 'London'

    SELECT CustomerID, CompanyName, City
    FROM Customers AS c
    WHERE c.City = 'Torino'
END
```

Výsledek této vložené procedury lze uložit do následujících typů *CustomerInfo* a *CustomerShortInfo*, jež ve své deklaraci nepotřebují žádné atributy:

```
public class CustomerInfo {
    public string CustomerID;
    public string CompanyName;
    public string City;
    public string ContactName;
}

public class CustomerShortInfo {
    public string CustomerID;
    public string CompanyName;
    public string City;
}
```

Deklarace odpovídající části v LINQ pro vloženou proceduru *TwoCustomerGroups* by měla vypadat jako výpis 4.13.

Výpis 4.13 Uložená procedura s více výstupními množinami

```
class SampleDb : DataContext {
    // ...
    [Function(Name = "TwoCustomerGroups", IsComposable = false)]
    [ResultType(typeof(CustomerInfo))]
    [ResultType(typeof(CustomerShortInfo))]
```

```
    public IMultipleResults TwoCustomerGroups() {
        IExecuteResult executeResult =
                this.ExecuteMethodCall(
                    this,
                    (MethodInfo) (MethodInfo.GetCurrentMethod()));
        IMultipleResults result =
            (IMultipleResults) executeResult.ReturnValue;
        return result;
    }
}
```

Výsledkové množiny jsou odlišného typu. Při volání metody *GetResult<T>* musíte zadat správný typ, který potřebuje přinejmenším pro každý navrácený sloupec veřejný člen se stejným názvem. Jestliže zadáte typ s více veřejnými členy, než kolik je sloupců, „chybějící" členy budou obsahovat výchozí hodnotu. Dále, každý návratový typ je nutné deklarovat pomocí atributu *ResultType*, jenž je součástí metody *TwoCustomrGroup*, což dokládá výpis 4.13. V následující ukázce musí první výsledková množina odpovídat typu *CustomerInfo* a druhá výsledková množina typu *CustomerShortInfo*:

```
IMultipleResults results = db.TwoCustomerGroups();
foreach( var row in results.GetResult<CustomerInfo>()) {
   // přístup k instanci CustomerInfo
}
foreach( var row in results.GetResult<CustomerShortInfo>()) {
   // přístup k instanci CustomerShortInfo
}
```

Pamatujte si, že nezáleží na pořadí atributů *ResultType*, ale je potřeba dbát na pořadí volání metody *GetResult<T>*. První výsledková množina se bude mapovat při prvním volání metody *GetResult<T>* atd., bez ohledu na typ použitého parametru. Pokud například otočíte předchozí dvě volání a před voláním *CustomerInfo* budete nejprve volat *CustomerShortInfo*, nedojde k žádné chybě, ale ve druhé výsledkové množině, mapované na typ *CustomerInfo*, bude v položce *ContactName* prázdný řetězec.

Důležité

Pořadí volání metody *GetResult<T>* je důležité a musí odpovídat pořadí návratových množin. Oproti tomu pořadí atributů *ResultType* metody reprezentující uloženou proceduru nemá žádnou relevanci.

Dalším způsobem použití rozhraní *IMultipleResults* je případ, kdy vložená procedura může vracet odlišné typy v závislosti na parametrech. Podívejte se například na tuto uloženou proceduru:

```
CREATE PROCEDURE ChooseResultType( @resultType INT )
AS BEGIN
   IF @resultType = 1
      SELECT * FROM [Customers]
   ELSE IF @resultType = 2
      SELECT * FROM [Products]
END
```

Takováto vložená procedura bude vždy vracet jediný výsledek, ale jeho typ se může při jednotlivých voláních odlišovat. Tento druh uložených procedur nemáme rádi a raději se této

situaci vyhýbáme. Ale musíte-li se s touto situací vypořádat, umožní vám to vložení obou možných atributů *ResultType* do příslušné metody:

```
[Function(Name = "ChooseResultType", IsComposable = false)]
[ResultType(typeof(Customer))]
[ResultType(typeof(Product))]
public IMultipleResults ChooseResultType( int resultType ) {
    IExecuteResult executeResult =
              this.ExecuteMethodCall(
              this,
              (MethodInfo) (MethodInfo.GetCurrentMethod()),
              resultType );
    IMultipleResults result =
       (IMultipleResults) executeResult.ReturnValue;
    return result;
}
```

V jediném potřebném volání metody *GetResult<T>* musíte specifikovat typ, který odpovídá tomu, co bude vložená procedura vracet:

```
IMultipleResults results = db.ChooseResultType( 1 );
foreach( var row in results.GetResult<Customer>()) {
    // přístup k instanci Customer
}
```

Pokud se nacházíte v obdobné situaci, je lepší volání vložené procedury (zde *ChooseResultType)* rozdělit do několika metod, každé pro jeden z možných návratových typů, čímž omezíte nebezpečí nesouladu mezi parametrem a typem výsledku:

```
public IEnumerable<Customer> ChooseCustomer() {
    IMultipleResults results = db.ChooseResultType( 1 );
    return results.GetResult<Customer>();
}
public IEnumerable<Product> ChooseProduct() {
    IMultipleResults results = db.ChooseResultType( 2 );
    return results.GetResult<Product>();
}
```

Než se obrátíme k uživatelským funkcím, chceme se ještě podívat na situaci, kdy voláte vloženou proceduru v dotazu LINQ. Podívejte se na tento kód:

```
var query =
    from c in db.CustomersByCity("London")
    where c.CompanyName.Length > 15
    select new { c.CustomerID, c.CompanyName };
```

Je zjevné, že tento dotaz lze plně převést na dotaz SQL. Ale veškerá data navrácená metodou *CustomerByCity* se předávají ze serveru SQL na klienta, což ukazuje vygenerovaný příkaz SQL:

```
EXEC @RETURN_VALUE = [Customers by City] @param1 = London'
```

Operace filtrování (*where*) a projekce (*select*) provádí LINQ pro objekty, filtruje data převedená na klienta a vypisuje pouze ty řádky, v nichž je vlastnost *CompanyNyme* delší než 15 znaků. Vložené procedury tedy nelze zakomponovat do jednoduchého dotazu SQL. Pro tento druh práce je nutné použít uživatelské funkce.

Uživatelské funkce (UDF)

Aby bylo možné používat uživatelskou funkci, je potřeba ji deklarovat stejně jako vloženou proceduru. Když UDF používáte uvnitř dotazu LINQ, stroj LINQ pro SQL ji musí při sestavování příkazu vzít v úvahu a přidat volání UDF do vygenerovaného kódu SQL. Možnost používá UDF v dotazu LINQ představuje kompozitní charakter – schopnost skládat různé dotazy a operátory do jediného dotazu. Protože pro uložené procedury i uživatelské funkce slouží tentýž atribut *Function*, nastavuje se pro práci s uživatelskou funkcí parametr *IsComposable* na *true* a v případě uložené procedury na *false*. Nezapomeňte, že nezáleží na tom, zdali je UDF napsána v T-SQL či SQLCLR.

Výpis 4.14 nabízí ukázku deklarace LINQ pro skalární UDF *MinUnitPriceByCategory*, definovanou v databázi Northwind:

Výpis 4.14 Skalární uživatelská funkce

```
class SampleDb : DataContext {
    // ...
    [Function(Name = "dbo.MinUnitPriceByCategory", IsComposable = true)]
    public decimal? MinUnitPriceByCategory( int? categoryID) {
        IExecuteResult executeResult =
            this.ExecuteMethodCall(
                this,
                ((MethodInfo) (MethodInfo.GetCurrentMethod())),
                categoryID);
        decimal? result = (decimal?) executeResult.ReturnValue;
        return result;
    }
}
```

Volání UDF v izolovaném výrazu generuje jednoduchý dotaz SQL. UDF lze použít také v dotazu LINQ, jako je tento:

```
var query =
   from c in Categories
   select new { c.CategoryID,
                c.CategoryName,
                MinPrice = db.MinUnitPriceByCategory( c.CategoryID )};
```

Vygenerovaný příkaz SQL se skládá z dotazu LINQ s voláním UDF, což vede k následujícímu dotazu SQL:

```
SELECT [t0].[CategoryID],
       [t0].[CategoryName],
       dbo.MinUnitPriceByCategory([t0].[CategoryID]) AS [value]
FROM [Categories] AS [t0]
```

Existují určité rozdíly v obálkách UDF s hodnotami v tabulce. Podívejte se na následující UDF:

```
CREATE FUNCTION [dbo].[CustomersByCountry] ( @country NVARCHAR(15) )
RETURNS TABLE
AS RETURN
   SELECT CustomerID,
          ContactName,
          CompanyName,
          City
   FROM Customers c
   WHERE c.Country = @country
```

K použití této UDF v LINQ musíte deklarovat metodu *CustomersByCountry*, již vidíte ve výpisu 4.15. Funkce s tabulkovým výstupem má vždy v atributu *Function* nastaven parametr *IsComposable* na *true*, ale namísto metody *DataContext.ExecuteMethodCall* volá *DataContext.CreateMethodCallQuery*.

Výpis 4.15 Uživatelská funkce s hodnotami v tabulce

```
class SampleDb : DataContext {
    // ...
    [Function(Name = "dbo.CustomersByCountry", IsComposable = true)]
    public IQueryable<Customer> CustomersByCountry(string country) {
        return this.CreateMethodCallQuery<Customer>(
            this,
            ((MethodInfo) (MethodInfo.GetCurrentMethod())),
            country);
    }
}
```

UDF s hodnotami v tabulce lze použít v dotazu LINQ podobně jako libovolnou jinou tabulku. Můžete například propojit zákazníky navrácené v uvedené funkci na objednávky, které učinili, což ukazuje následující dotaz:

```
Table<Order> Orders = db.GetTable<Order>();
var queryCustomers =
   from c in db.CustomersByCountry( "USA" )
   join o in Orders
          on c.CustomerID equals o.CustomerID
          into orders
   select new { c.CustomerID, c.CompanyName, orders };
```

Vygenerovaný dotaz SQL bude vypadat asi takto:

```
SELECT [t0].[CustomerID], [t0].[CompanyName],
        [t1].[OrderID], [t1].[CustomerID] AS [CustomerID2],
        (SELECT COUNT(*)
        FROM [Orders] AS [t2]
        WHERE [t0].[CustomerID] = [t2].[CustomerID]
        ) AS [value]
FROM dbo.CustomersByCountry('USA') AS [t0]
LEFT OUTER JOIN [Orders] AS [t1] ON [t0].[CustomerID] = [t1].[CustomerID]
ORDER BY [t1].[OrderID]
```

Kompilované dotazy

Jestliže potřebujete opakovat určitý dotaz mnohokrát, případně s různými hodnotami parametrů, asi budete přemýšlet o problémech s opakovaným sestavováním dotazu. Některé databáze, například SQL Server, zkoušejí parametrizovat přijaté dotazy SQL automaticky a optimalizovat tak kompilaci prováděcího plánu dotazu. Ale program, jenž posílá do SQL Serveru parametrizovaný dotaz, bude mít lepší výkon, protože SQL Server nemusí věnovat čas analýze dotazu, jestliže je dotaz podobný jinému dotazu, který se již prováděl dříve. LINQ odvádí dobrou práci při optimalizaci dotazu, ale pokaždé, když se vyhodnocuje stejný strom dotazu, stroj LINQ pro SQL jej analyzuje a vytváří z něj totožný kód SQL. Toto chování lze optimalizovat pomocí třídy *CompiledQuery*.

Další informace

Vestavěný poskytovatel pro SQL Server posílá do databáze parametrizované dotazy. Pokaždé, když vidíte v kódu SQL prezentovaném v této kapitole konstantní hodnotu, mějte na paměti, že skutečný dotaz SQL, posílaný do databáze, má parametr pro každou konstantu v dotazu. Tato konstanta může být výsledkem výrazu, jenž je nezávislý na vykonávání dotazu. Uvedený druh výrazu se vytváří v prostředí jazyka (v tomto případě C#). Když použijete třídu *CompiledQuery*, analýza stromu dotazu a vytváření odpovídající kódu SQL se neopakuje pokaždé, když LINQ musí provádět totožný dotaz. Možná se divíte, co je tedy tak zásadního na používání třídy *CompiledQuery*. Rico Mariani provedl výkonový test popsaný v tomto blogu: *http://blogs.msdn.com/ricom/archive/2008/01/14/performance-quiz-13-linq-to-sql-compiledquery-cost-solution.aspx*. Výsledkem tohoto testu je, že od dvou zavolání dotazu je použití třídy *CompiledQuery* výkonově výhodné.

Ke zkompilování dotazu můžete použít některou ze statických metod *CompiledQuery.Compile*. Toto volání předává dotaz LINQ jako parametr ve formě stromu výrazu a poté získává delegáta s parametry odpovídajícími prostředí *DataContext*, v němž chcete pracovat, a parametrům dotazu. Výpis 4.16 ilustruje deklaraci a používání kompilovaného dotazu.

Výpis 4.16 Lokální kompilovaný dotaz

```
static void CompiledQueriesLocal() {
    DataContext db = new DataContext( ConnectionString );
    Table<Customer> Customers = db.GetTable<Customer>();

    var query =
        CompiledQuery.Compile(
            ( DataContext context, string filterCountry ) =>
                from c in Customers
                where c.Country == filterCountry
                select new { c.CustomerID, c.CompanyName, c.City } );

    foreach (var row in query( db, "USA" )) {
        Console.WriteLine( row );
    }
    foreach (var row in query( db, "Italy" )) {
        Console.WriteLine( row );
    }
}
```

Jak ukazuje výpis 4.16, metoda *Compile* přebírá výraz lambda, jehož prvním parametrem je instance třídy *DataContext*. Tento parametr určuje spojení, v němž se dotaz bude provádět. V uvedeném případě nepoužíváme uvnitř našeho výrazu lambda tento parametr. Přiřazení výsledku metody *CompiledQuery.Compile* do lokální proměnné je snadné (protože tuto proměnnou deklarujete pomocí deklarátoru *var*), ale tato situace se nevyskytuje příliš často. Častokrát budete potřebovat uložit delegáta navráceného metodou *CompiledQuery.Compile* do instance či statického členu, abyste jej mohli použít vícekrát. K tomu musíte znát správnou syntaxi deklarace.

Kompilovaný dotaz se ukládá do delegáta *Func*, v němž prvním parametrem musí být instance třídy *DataContext* (nebo třídy odvozené z třídy *DataContext*) a poslední parametr

musí tvořit návratový typ dotazu. Mezi nimi mohou být až tři parametry, které budou tvořit parametry zkompilovaného dotazu, a je potřeba je zadat pro každé volání kompilovaného dotazu. Výpis 4.17 ukazuje syntaxi, kterou můžete použít pro vytvoření a následné použití kompilovaného dotazu.

Výpis 4.17 Kompilovaný dotaz vložený do statického členu

```
public static Func< nwind.Northwind, string, IQueryable<nwind.Customer>>
    CustomerByCountry =
        CompiledQuery.Compile(
            ( nwind.Northwind db, string filterCountry ) =>
                from c in db.Customers
                where c.Country == filterCountry
                select c );

static void CompiledQueriesStatic() {
    nwind.Northwind db = new nwind.Northwind( ConnectionString );

    foreach (var row in CustomerByCountry( db, "USA" )) {
        Console.WriteLine( row.CustomerID );
    }

    foreach (var row in CustomerByCountry( db, "Italy" )) {
        Console.WriteLine( row.CustomerID );
    }
}
```

Protože delegát *Func*, uchovávající kompilovaný dotaz, potřebuje ve své deklaraci datový typ výsledku, nelze pro výsledek kompilovaného dotazu použít anonymní typ. Je to možné pouze tehdy, když kompilovaný dotaz ukládáte do lokální proměnné, což jste viděli ve výpisu 4.16.

Různé přístupy k dotazům na data

Když pracujete s entitami LINQ pro SQL, existují pro dotazy na tatáž data dva postupy. Klasický způsob práce s relačním schématem obnáší zápis dotazů s vazbami, podobně jako v SQL. Jiný přístup, který nabízí LINQ pro SQL, znamená průchod grafem. Pro stejné výsledky můžeme při použití odlišných postupů v LINQ dostat odlišné dotazy SQL a různou úroveň výkonu.

Podívejte se na následující dotaz SQL, který počítá celkové množství objednávek produktu (v tomto případě čokolády, což je lokalizovaný název v databázi Northwind):

```
SELECT SUM( od.Quantity ) AS TotalQuantity
FROM [Products] p
LEFT JOIN [Order Details] od
   ON od.[ProductID] = p.[ProductID]
WHERE p.ProductName = 'Chocolade'
```

Přirozený převod na dotaz LINQ vidíte ve výpisu 4.18. Operátor *Single* načítá první řádek a vkládá jej do proměnné *quantityJoin*, která se používá k zobrazení výsledků.

Výpis 4.18 Dotaz s klauzulí Join

```
var queryJoin =
    from   p in db.Products
    join   o in db.Order_Details
```

```
            on p.ProductID equals o.ProductID
            into OrdersProduct
    where p.ProductName == "Chocolade"
    select OrdersProduct.Sum( o => o.Quantity );
var quantityJoin = queryJoin.Single();
Console.WriteLine( quantityJoin );
```

Jak vidíte, asociativní dotaz v LINQ může explicitně vyžadovat spojení mezi položkami *Products* a *Order_Details* získané pomocí rovnosti jejich parametrů *ProductId*. Využijeme-li entit, můžeme implicitně použít vztahu mezi entitami *Products* a *Order_Details*, definovaného ve třídě *Product*, což ukazuje výpis 4.19.

Výpis 4.19 Dotaz s vazbou Association

```
var queryAssociation =
    from p in db.Products
    where p.ProductName == "Chocolade"
    select p.Order_Details.Sum( o => o.Quantity );
var quantityAssociation = queryAssociation.Single();
Console.WriteLine( quantityAssociation );
```

Jednoduché dotazy SQL, generované oběma těmito dotazy LINQ, jsou identické. Dotaz LINQ s klauzulí *join* má explicitnější formu přístupu k datům, kdežto dotaz využívající vazby mezi entitami *Product* a *Order_Details* je v tomto ohledu implicitnější. Používání implicitních vazeb zkracuje dotazy a předchází chybám (protože nelze udělat chybu v podmínce spojení). Na první pohled se může zdát zkrácený dotaz hůře čitelný. Ale vnímání tohoto typu dotazů se zlepší, protože nyní jste zvyklí na delší dotazy a po čase se může dostavit uspokojení z kratších dotazů.

Poznámka

Dotaz SQL, vygenerovaný z dotazů LINQ ve výpisech 4.18 a 4.19, se odlišuje v SQL Serveru 2000 a SQL Serveru 2005 či pozdějších verzích. V SQL Serveru 2005 se použije spojení OUTER APPLY. Jde o výsledek interní implementace poskytovatele, nicméně výsledek je totožný.

Když situaci prozkoumáme hlouběji, vidíme, že načítání jednoho produktu nevyžaduje dotazovací výraz. Můžete aplikovat operátor *Single* přímo na tabulku *Products*, což dokládá výpis 4.20. Zjevně dostanete stejné výsledky. Ale vnitřní proces je výrazně odlišný, protože tento druh přístupu generuje instance entit *Product* a *Order_Details* v paměti, i když je ve svém programu nepoužíváte.

Výpis 4.20 Přístup přes entitu

```
var chocolade = db.Products.Single( p => p.ProductName == "Chocolade" );
var quantityValue = chocolade.Order_Details.Sum( o => o.Quantity );
Console.WriteLine( quantityValue );
```

Jde o operaci o dvou krocích, která posílá do databáze dva dotazy SQL. První načítá entitu *Product*. Druhý přistupuje k tabulce Order Details, načítá *všechny* řádky této tabulky pro požadovaný produkt a v paměti pro něj sčítá všechny hodnoty *Quantity*. Uveďme si vygenerované příkazy SQL:

```
SELECT [t0].[ProductID], [t0].[ProductName], [t0].[SupplierID],
    [t0].[CategoryID], [t0].[QuantityPerUnit], [t0].[UnitPrice],
    [t0].[UnitsInStock], [t0].[UnitsOnOrder], [t0].[ReorderLevel],
    [t0].[Discontinued]
FROM [dbo].[Products] AS [t0]
WHERE [t0].[ProductName] = "Chocolade"

SELECT [t0].[OrderID], [t0].[ProductID], [t0].[UnitPrice], [t0].[Quantity],
    [t0].[Discount]
FROM [dbo].[Order Details] AS [t0]
WHERE [t0].[ProductID] = "Chocolade"
```

Kód používající tento druh přístupu je sice ve srovnání s dotazem kratší, ale jeho výkon bude nižší, pokud potřebujete pouze součet hodnot *Quantity* a není nutné načítat entity *Product* a *Order_Details* do paměti pro další operace.

Dotazy ve výpisu 4.18 a 4.19 nevytvářejí instance *Product* či *Order_Details*, protože ve výstupu vyžadujeme pouze součet pro produkt. Z tohoto úhlu pohledu by byl výkon uvedených dotazů nižší v situaci, v níž bychom již měli požadované instance *Product* a *Order_Details* pro položku Chocolade v paměti, protože dotazy by zbytečně přistupovaly do databáze a načítaly data, která již jsou v paměti. Naopak druhý přístup, kterým získáváme součet hodnot *Quantity*, by mohl být rychlejší, pokud použijete dříve uvedený postup využívající entity. Podívejte se na následující kód:

```
var chocolade = db.Products.Single( p => p.ProductName == "Chocolade" );
var quantityValue = chocolade.Order_Details.Sum( o => o.Quantity );
Console.WriteLine( quantityValue );
var repeatCalc = chocolade.Order_Details.Sum( o => o.Quantity );
Console.WriteLine( repeatCalc );
```

Vyčíslení hodnoty *quantityValue* vyžaduje dotaz do databáze, aby se mohly vytvořit entity *Order_Details*, zatímco výpočet hodnoty *repeatCalc* se provádí na entitách v paměti, aniž by bylo potřeba načítat další data z SQL Serveru.

Poznámka

Chcete-li pochopit chování tohoto kódu, můžete zanalyzovat vygenerované dotazy SQL. V předchozích ukázkách jsme psali do dotazu LINQ funkci *Sum*. Když vygenerovaný dotaz SQL obsahuje agregační operaci SUM, nenačítáme entity do paměti. Když vygenerovaný dotaz SQL neobsahuje tutéž agregační operaci, jakou potřebujeme, agregace bude probíhat v paměti na příslušných entitách.

Poslední myšlenka se týká počtu vygenerovaných dotazů. Možná vás napadne, že při přístupu k datům prostřednictvím entity *Product* jsme vygenerovali dva dotazy, protože máme dva odlišné příkazy – jeden, který přiřazuje do proměnné *chocolade*, a druhý, jenž přiřazuje do proměnné *quantityEntity*. Tato úvaha není zcela správná. I když napíšete jediný dotaz, použití entity *Product* (výsledek volání operátoru *Single*) obnáší samostatný dotaz. Výpis 4.21 generuje stejný výsledek (pokud jde o objekty v paměti a dotazy SQL) jako výpis 4.20.

Výpis 4.21 Přístup přes entitu v jediném příkazu

```
var quantityChocolade = db.Products.Single( p => p.ProductName == "Chang" )
                          .Order_Details.Sum( o => o.Quantity );
Console.WriteLine( quantityChocolade );
```

Hledání lepšího způsobu přístupu k datům ve skutečnosti závisí na celé množině operací, kterou program provádí. Jestliže používáte ve svém kódu rozsáhlým způsobem entity k ukládání dat do paměti, nabídne vám lepší výkon patrně přístup k datům promocí průchodu grafem založeného na přístupu k entitám. Naopak, pokud vždy převádíte výsledky dotazů na anonymní typy a nikdy nemanipulujete s entitami v paměti, dáte patrně přednost přístupu vycházejícímu z dotazů LINQ. Jak už to bývá, správná odpověď zní: „Přijde na to."

Přímé dotazy

Někdy můžete potřebovat přístup k funkcím databáze SQL, které nejsou přístupné prostřednictvím LINQ. Představte si kupříkladu situaci, kdy chcete použít v SQL Serveru výrazy Common Table Expressions (CTE) nebo příkaz PIVOT. LINQ pro tyto operace nemá explicitní konstruktor, dokonce i kdyby určený poskytovatel pro SQL Server tyto funkce uměl kvůli optimalizací určitých dotazů používat. Výpis 4.22 ukazuje, jak můžete použít metodu *ExecuteQuery<T>* třídy DataContext k zaslání dotazu přímo do databáze. *T* v metodě *ExecuteQuery<T>* představuje třídu entity, která reprezentuje navrácený řádek.

Výpis 4.22 Přímý dotaz

```
var query = db.ExecuteQuery<EmployeeInfo>( @"
    WITH EmployeeHierarchy (EmployeeID, LastName, FirstName,
                            ReportsTo, HierarchyLevel) AS
     ( SELECT EmployeeID,LastName, FirstName,
              ReportsTo, 1 as HierarchyLevel
       FROM   Employees
       WHERE  ReportsTo IS NULL

       UNION ALL

       SELECT      e.EmployeeID, e.LastName, e.FirstName,
                   e.ReportsTo, eh.HierarchyLevel + 1 AS HierarchyLevel
       FROM        Employees e
       INNER JOIN  EmployeeHierarchy eh
               ON  e.ReportsTo = eh.EmployeeID
    )
    SELECT   *
    FROM     EmployeeHierarchy
    ORDER BY HierarchyLevel, LastName, FirstName" );
```

Jak vidíte, potřebujeme typ, skrze nějž bychom získali výsledky přímého dotazu. Použili jsme třídu *EmployeeInfo*, jež má tuto deklaraci:

```
public class EmployeeInfo {
   public int EmployeeID;
   public string LastName;
   public string FirstName;
   public int? ReportsTo; // int? odpovídá typu Nullable<int>
   public int HierarchyLevel;
}
```

Názvy a typy členů třídy *EmployeeInfo* musí odpovídat názvům a typům sloupců navrácených vykonaným dotazem. Nezapomeňte, že může-li sloupec vracet hodnotu NULL, je potřeba použít typ povolující hodnotu null, což je případ členu *ReportsTo*, který je deklarován jako *int?* (což odpovídá typu *Nullable<int>*).

Důležité

Sloupce ve výsledných řádcích, které neodpovídají atributům entity, budou ignorovány. Členy entity, které nemají odpovídající sloupce, budou inicializovány pomocí výchozí hodnoty. Jestliže bude třída *EmployeeInfo* obsahovat chybně napsaný název sloupce, tento člen nebude obsahovat hodnoty a nedojde k chybě. Buďte opatrní, a pokud některý sloupec či člen neobsahuje příslušné hodnoty, ověřte soulad názvů.

Metoda *ExecuteQuery* může také přebírat parametry pomocí stejné notace se složenými závorkami jako metody *Console.WriteLine* a *String.Format*, ale chování bude odlišné. V řetězci posílaném do databáze se parametry nenahrazují hodnotami, ale dojde k náhradě za názvy parametrů, které se generují automaticky (*@p0, @p1, @p2...*) a posílají se do SQL Serveru jako argumenty parametrického dotazu.

Kód ve výpisu 4.23 ukazuje volání metody *ExecuteQuery<T>* s příkazem SQL a dvěma parametry. Parametry slouží k filtrování zákazníků, kteří svou první objednávku učinili v rozmezí dvou konkrétních časových okamžiků.

Výpis 4.23 Přímý dotaz s parametry

```
var query = db.ExecuteQuery<CompanyOrders>(@"
        SELECT    c.CompanyName,
                  MIN( o.OrderDate ) AS FirstOrderDate,
                  MAX( o.OrderDate ) AS LastOrderDate
        FROM      Customers c
        LEFT JOIN Orders o
               ON o.CustomerID = c.CustomerID
        GROUP BY  c.CustomerID, c.CompanyName
        HAVING    COUNT(o.OrderDate) > 0
           AND    MIN( o.OrderDate ) BETWEEN {0} AND {1}
        ORDER BY  FirstOrderDate ASC",
    new DateTime( 1997, 1, 1 ),
    new DateTime( 1997, 12, 31 ) );
```

Parametry mají formát *{0}* a *{1}*. Vygenerovaný dotaz SQL je jednoduše nahradí výrazy *@p0* a *@p1*. Výsledky se vracejí v instancích třídy *CompanyOrders*, jež má následující deklaraci:

```
public class CompanyOrders {
   public string CompanyName;
   public DateTime FirstOrderDate;
   public DateTime LastOrderDate;
}
```

Odložené načítání entit

Viděli jsme, že používání průchodu grafem pro dotazy na data je velice komfortní přístup. Ale někdy chcete poskytovateli LINQ pro SQL zabránit, aby automaticky rozhodoval, které entity

se mají načítat z databáze a kdy, a převzít kontrolu nad touto částí procesu. Slouží k tomu vlastnosti *DeferredLoadingEnabled* a *LoadOptions* třídy *DataContext*.

Kód ve výpisu 4.24 provádí stejné volání metody *QueryOrder* za tří různých podmínek, které řídí kód v metodě *DemoDeferredLoading*.

Výpis 4.24 Odložené načítání entit

```
public static void DemoDeferredLoading() {
    Console.Write("DeferredLoadingEnabled=true ");
    DemoDeferredLoading(true);
    Console.Write("DeferredLoadingEnabled=false ");
    DemoDeferredLoading(false);
    Console.Write("Using LoadOptions ");
    DemoLoadWith();
}

static void DemoDeferredLoading(bool deferredLoadingEnabled) {
    nwDataContext db = new nwDataContext(Connections.ConnectionString);
    db.DeferredLoadingEnabled = deferredLoadingEnabled;

    QueryOrder(db);
}

static void DemoLoadWith() {
    nwDataContext db = new nwDataContext(Connections.ConnectionString);
    db.DeferredLoadingEnabled = false;

    DataLoadOptions loadOptions = new DataLoadOptions();
    loadOptions.LoadWith<Order>(o => o.Order_Details);
    db.LoadOptions = loadOptions;

    QueryOrder(db);
}

static void QueryOrder(nwDataContext db) {
    var order = db.Orders.Single((o) => o.OrderID == 10251);
    var orderValue = order.Order_Details.Sum(od => od.Quantity * od.UnitPrice);
    Console.WriteLine(orderValue);
}
```

Volání metody *DemoDeferredLoading(true)* nastavuje vlastnost *DeferredLoadingEnabled* na *true*, což je výchozí podmínka pro instanci třídy *DataContext*. Volání *DemoDeferredLoadin g(false)* vlastnost *DeferredLoadingEnabled* vypíná. Jakýkoliv přístup k souvisejícím entitám nenačítá automaticky data z databáze a součet entit *Order_Details* dává hodnotu 0. Závěrečné volání metody *DemoLoadWith* rovněž vypíná vlastnost *DeferredLoadingEnabled*, ale nastavuje vlastnost *LoadingOptions* třídy *DataContext* a načítá entity *Order_Details* svázané s instancí *Order*. Spuštění metody *DemoDeferredLoading* z výpisu 4.24 dává následující výstup:

```
DeferredLoadingEnabled=true      670,8000
DeferredLoadingEnabled=false     0
použití LoadOptions              670,8000
```

Pamatujte si, že použití vlastnosti *LoadOptions* je možné bez ohledu na nastavení vlastnosti *DeferredLoadingEnabled* a hodí se pro zvýšení výkonu v situaci, představuje-li časné načtení souvisejících entit (namísto odloženého načítání) pro vaši aplikaci výhodu. Použití

vlastnosti *DeferredLoadingEnabled* pečlivě uvažte, protože negeneruje žádnou chybu, ale omezuje možnost procházení dat při průchodu grafem. Musíte si však uvědomit, že vlastnost *DeferredLoadingEnabled* se považuje za rovnu *false* vždy, když je rovněž vypnuta vlastnost *ObjectTrackingEnabled* (o níž pojednává následující část kapitoly).

Odložené načítání vlastností

LINQ pro SQL nabízí mechanismus odloženého načítání, které funguje na úrovni vlastností a načítá data pouze tehdy, když se k dané vlastnosti poprvé přistupuje. Uvedený mechanismus můžete použít, jestliže potřebujete načíst velké množství entit do paměti, což obvykle vyžaduje dostatek prostoru pro všechny vlastnosti třídy, které odpovídají sloupcům v databázové tabulce. Je-li určité pole značně velké a nepřistupuje se k němu v každé entitě, můžete načítání této vlastnosti zpozdit.

Chcete-li pracovat s odloženým načítáním vlastnosti, stačí na deklaraci úložné proměnné pro sloupec v tabulce použít typ *Link<T>*, což dokládá výpis 4.25.

Výpis 4.25 Odložené načítání vlastností

```
[Table(Name = "Customers")]
public class DelayCustomer {
    private Link<string> _Address;

    [Column(IsPrimaryKey = true)] public string CustomerID;
    [Column] public string CompanyName;
    [Column] public string Country;

    [Column(Storage = "_Address")]
    public string Address {
        get { return _Address.Value; }
        set { _Address.Value = value; }
    }
}

public static class DeferredLoading {
    public static void DelayLoadProperty() {
        DataContext db = new DataContext(Connections.ConnectionString);
        Table<DelayCustomer> Customers = db.GetTable<DelayCustomer>();
        db.Log = Console.Out;

        var query =
            from c in Customers
            where c.Country == "Italy"
            select c;

        foreach (var row in query) {
            Console.WriteLine(
                "{0} - {1}",
                row.CompanyName,
                row.Address);
        }
    }
}
```

Dotaz zaslaný do databáze, jenž načítá seznam italských zákazníků, vypadá následovně:

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[Country]
FROM [Customers] AS [t0]
WHERE [t0].[Country] = "Italy"
```

Tento dotaz nenačítá pole *Address*. Když se výsledek dotazu prochází ve smyčce *foreach*, přistupuje se k vlastnosti *Address* aktuálního zákazníka poprvé a tím pádem se do databáze posílá dotaz, který vybírá hodnotu *Address*:

```
SELECT [t0].[Address]
FROM [Customers] AS [t0]
WHERE [t0].[CustomerID] = @p0
```

Typ *Link<T>* by se měl používat pouze tehdy, když je obsah pole mimořádně veliký (což by neměl být případ pole *Address* v našem příkladu) a když se k poli přistupuje jen velmi zřídka. Obecně je dobrým kandidátem typ SQL *VARCHAR(MAX),* pokud se jeho hodnota zobrazuje pouze v detailním formuláři, zobrazovaném na požádání, a nikoli v hlavním seznamu s výsledky dotazu. S typem *Link<T>* můžete pracovat pomocí generátoru tříd LINQ pro SQL, jenž je součástí Visual Studia 2008, a nastavit vlastnost *Delay Loaded* požadované členské vlastnosti na *true*.

Důležité

Typ *Link<T>* je nutné použít pro úložnou proměnnou vlastnosti typu *T*, mapované na sloupec, což je případ ve výpisu 4.25. Typ *Link<T>* nelze použít přímo pro veřejný datový člen mapovaný na sloupec v tabulce (podobně jako všechna ostatní pole); jestliže to uděláte, dojde za běhu k výjimce. Ve verzi RTM jde o běhovou chybu typu *VerificationException*. V budoucích verzích bude patrně použita popisnější výjimka.

Přístup k datům pouze pro čtení pomocí třídy DataContext

Jestliže potřebujete k datům přistupovat výhradně pro čtení, můžete vylepšit výkon zákazem služby *DataContext*, která slouží k úpravám dat:

```
DataContext db = new DataContext( ConnectionString );
db.ObjectTrackingEnabled = false;
var query = ...
```

Vlastnost *ObjectTrackingEnabled* řídí změny sledovací služby, kterou si popíšeme v následující kapitole. Ve výchozím nastavení je vlastnost *ObjectTrackingEnabled* nastavena na *true*.

Důležité

Vypnutí sledování objektů rovněž vypíná funkci odloženého načítání v téže instanci třídy *DataContext*. Chcete-li zvýšit výkon vypnutím funkce sledování objektů, nezapomeňte na vedlejší dopady vypnutí odloženého načítání. Podrobnosti najdete v předchozí stati „Odložené načítání entit“.

Omezení LINQ pro SQL

LINQ pro SQL má určitá omezení týkající se převodu dotazu LINQ na odpovídající příkaz SQL. Z tohoto důvodu nejsou v LINQ pro SQL podporovány určité platné příkazy prostředí LINQ pro objekty. V této části kapitoly se budeme věnovat nejvýznamnějším operátorům, které nelze v dotazu LINQ pro SQL použít.

Další informace

Úplný seznam nepodporovaných metod a typů je součástí dokumentace produktu, „Data Types and Functions (LINQ to SQL)", jež je k dispozici na adrese *http://msdn2.microsoft.com/en-us/library/bb386970.aspx*.

Agregační operátory

Obecný operátor *Aggregate* není podporován. Ale plnou podporu mají specializované operátory, jako je *Count, LongCount, Sum, Min, Max* a *Average.*

Libovolný agregační operátor s výjimkou *Count* a *LongCount* vyžaduje speciální pozornost, aby nedošlo k výjimce, když je výsledek roven *null*. Má-li třída entity člen s typem, jenž nemůže nabývat hodnoty *null*, a vy na něm chcete provádět agregaci, výsledek *null* (když například nedojde k agregaci žádných řádků) způsobí výjimku. Agregovanou hodnotu je potřeba převést na typ povolující *null* a teprve poté jej lze zařadit do agregační funkce, čímž předejdete uvedené výjimce. Příklad nezbytného přetypování vidíte ve výpisu 4.26.

Výpis 4.26 Ošetření hodnot null v agregačních operátorech

```
decimal? totalFreight =
    (from o in Orders
     where o.CustomerID == "NOTEXIST"
     select o).Min( o => (decimal?) o.Freight );
```

Tento převod je nutný pouze tehdy, když deklarujete vlastnosti *Freight* jako *decimal*:

```
[Table(Name = "Orders")]
public class Order {
   [Column] public decimal Freight;
}
```

Jiným řešením je deklarovat vlastnost *Freight* výrazem *decimal?* jako typ povolující *null*, ale není vhodné mít odlišné nastavení povolení hodnot *null* v entitách a v odpovídajících tabulkách v databázi.

Další informace

Podrobnější diskusi o tomto problému naleznete v příspěvku od Iana Grifithse: *http://www.interact-sw.co.uk/iangblog/2007/09/10/linq-aggregates*.

Dělicí operátory

Operátory *TakeWhile* a *SkipWhile* nejsou podporovány. Operátory *Take* a *Skip* podporovány jsou, ale u operátoru *Skip* buďte opatrní, protože vygenerovaný dotaz SQL může být složitý

a neefektivní, má-li se přeskočit hodně řádků, především je-li cílovou databází SQL Server 2000.

Operátory pro elementy

Následující operátory nejsou podporovány: *ElementAt, ElementAtOrDefault, Last* a *LastOrDefault*.

Řetězcové metody

Mnoho metod typu *String* v .NETu je v LINQ pro SQL podporováno, neboť existuje odpovídající metoda v T-SQL. Ale neexistuje podpora metod, které pracují s jazykovou verzí (tedy ty, které přebírají parametry typu *CultureInfo*, *StringComparison* a *IFormatProvider*), a metod, které přebírají či vracejí pole typu *char*.

Metody pro datum a čas

Mezi typem *DateTime* v .NETu a typy *DATETIME* a *SMALLDATETIME* v SQL Serveru jsou určité rozdíly. Interval hodnot a přesnost jsou v .NETU vyšší než v SQL Serveru, což umožňuje korektní zobrazení typů SQL Serveru v .NETu, ale nikoli naopak. Dále, typ *DATETIME* v SQL Serveru nepočítá s časovým pásmem, tudíž nemůže být v LINQ pro SQL podporován. V neposlední řadě nejsou podporovány některé metody .NETu pro datum a čas, převážně kvůli absenci odpovídající funkce v T-SQL.

Nepodporované funkce SQL

LINQ pro SQL nemá syntaxi, která by umožňovala použít operátor SQL *LIKE* a agregaci *STDDEV*.

Uvažování v LINQ pro SQL

Když začnete pracovat s LINQ pro SQL, budete patrně muset změnit způsob smýšlení při psaní dotazů, především pokud se snažíte najít v LINQ odpovídající syntaxi pro dobře známé příkazy SQL. Navíc je možné rozvláčný dotaz LINQ zredukovat, když dojde k vygenerování příslušného dotazu SQL. Této změny si musíte být vědomi a musíte ji plně pochopit, máte-li v LINQ pro SQL pracovat produktivně. V poslední části této kapitoly vám ukážeme, jak uvažovat v LINQ pro SQL.

Klauzule IN/EXISTS

Jedním z nejlepších příkladů syntaktických odlišností T-SQL a LINQ je klauzule NOT IN, kterou lze používat v SQL. Tato klauzule v LINQ neexistuje, což vede k úvaze, zdali je možné v LINQ nějak vyjádřit stejnou myšlenku. Ve skutečnosti neexistuje vždy přímý překlad každého klíčového slova v SQL, ale stejného výsledku se lze dobrat pomocí sémanticky ekvivalentních příkazů, někdy se stejným či lepším výkonem.

Podívejte se na následující kód, jenž vrací všechny zákazníky, kteří nemají v tabulce Orders žádnou objednávku. Dotaz vracející tuto informaci vypadá takto:

```
SELECT *
FROM [dbo].[Customers] AS [t0]
```

```
WHERE [t0].[CustomerID] NOT IN (
    SELECT [t1].[CustomerID]
    FROM [dbo].[Orders] AS [t1]
)
```

Nejde o nejrychlejší způsob, jak docílit požadovaného výsledku. (Naším oblíbeným způsobem je použít klauzuli NOT EXISTS – více za okamžik.) Nicméně LINQ nemá operátor, který by přímo odpovídal operátorům IN či NOT IN, ale nabízí operátor *Contains*, který umožňuje zapsat kód z výpisu 4.27. Zaměřte se na operátor negace (*!*) aplikovaný na predikát *where*, kde neguje následnou podmínku *Contains*.

Výpis 4.27 Použití operátoru Contains k získání výrazu odpovídajícího klauzuli EXISTS/IN

```
public static void DemoContains() {
    nwDataContext db = new nwDataContext(Connections.ConnectionString);
    db.Log = Console.Out;

    var query =
        from c in db.Customers
        where !(from o in db.Orders
                select o.CustomerID)
               .Contains(c.CustomerID)
        select new { c.CustomerID, c.CompanyName };

    foreach (var c in query) {
        Console.WriteLine(c);
    }
}
```

Následující kód je dotaz SQL, vygenerovaný pomocí LINQ pro SQL:

```
SELECT [t0].[CustomerID], [t0].[CompanyName]
FROM [dbo].[Customers] AS [t0]
WHERE NOT (EXISTS(
    SELECT NULL AS [EMPTY]
    FROM [dbo].[Orders] AS [t1]
    WHERE [t1].[CustomerID] = [t0].[CustomerID]
    ))
```

Tento přístup ke generování kódu SQL není v důsledku jen sémanticky ekvivalentní, ale také znamená rychlejší provádění. Jestliže se podíváte na operace čtení/zápis v SQL Serveru 2005, první dotaz (používající NOT IN) vykonává 364 logických načítání z tabulky Orders, zatímco druhý dotaz (používající NOT EXISTS) vyžaduje pouze 5 logických načítání z téže tabulky Orders. Jde o značný rozdíl a v tomto případě je LINQ pro SQL pro vás nejlepší volbou.

Tentýž operátor *Contains* může generovat operátor IN v SQL například tehdy, když jej aplikujete na seznam konstant, viz výpis 4.28.

Výpis 4.28 Použití operátoru Contains se seznamem konstant

```
public static void DemoContainsConstants() {
    nwDataContext db = new nwDataContext(Connections.ConnectionString);

    var query =
        from c in db.Customers
        where (new string[] { "London", "Seattle" }).Contains(c.City)
        select new { c.CustomerID, c.CompanyName, c.City };
```

```
    Console.WriteLine(query);

    foreach (var c in query) {
        Console.WriteLine(c);
    }
}
```

Kód generovaný LINQ pro SQL je snazší na čtení než původní dotaz:

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[City]
FROM [dbo].[Customers] AS [t0]
WHERE [t0].[City] IN ("London", "Seattle")
```

Neintuitivní na dotazu LINQ je to, že musíte zadat operátor *Contains* pro seznam konstant a předávat hledanou hodnotu v parametru, což je přesně naopak než v SQL:

```
where (new string[] { "London", "Seattle" }).Contains(c.City)
```

Po letech práce s SQL nám více vyhovuje hypotetická syntaxe *IsIn*, vypadající asi takto:

```
where c.City.IsIn( new string[] { "London", "Seattle" } )
```

Je však jen otázkou času, kdy si na novou syntaxi zvykneme. Ve skutečnosti odpovídá sémantika operátoru *Contains* přesně pozici parametru. Abychom kód vyjasnili, mohli bychom oddělit deklaraci seznamu konstant od deklarace dotazu a zadat jej do pole *cities*. Kód je pak čitelnější:

```
var cities = new string[] { "London", "Seattle" };
var query =
   from c in db.Customers
   where cities.Contains(c.City)
   select new { c.CustomerID, c.CompanyName, c.City };
```

Poznámka

Vytvoření pole *cities* mimo dotaz namísto jeho vložení přímo do predikátu *where* pouze zvyšuje čitelnost kódu, přinejmenším v LINQ pro SQL. Z výkonového hlediska se v obou případech vytváří pouze jediné pole řetězců. Důvodem je skutečnost, že v LINQ pro SQL definuje dotaz při generování příkazu SQL pouze strom výrazu a pole se vytváří pouze jednou. V LINQ pro SQL, pokud nespustíte tentýž dotaz mnohokrát, je výkon v obou případech stejný (vytvoření objektu uvnitř či vně predikátu). Tím se prostředí liší od LINQ pro objekty. Tam by se podmínka predikátu v klauzuli *where* prováděla pro každý řádek v datovém zdroji.

Redukce dotazů SQL

Každý dotaz LINQ pro SQL je na počátku reprezentován v paměti stromem výrazu. Stroj LINQ pro SQL převádí tento strom na odpovídající dotaz SQL, přičemž prochází strom a generuje odpovídající kód. Teoreticky však může překlad probíhat mnoha způsoby, které vždy dávají stejný výsledek, i když ne všechny překlady jsou stejně čitelné či mají stejný výkon. Současná implementace LINQ pro SQL generuje dobrý kód SQL, upřednostňující výkon před čitelností, i když i čitelnost je často více než přijatelná.

Další informace

Více informací o redukci dotazů v poskytovateli LINQ naleznete v příspěvku Matta Warrena: *http://blogs.msdn.com/mattwar/archive/2008/01/16/linq-building-an-iqueryable-provider-part-ix.aspx*. Implementace poskytovatele dotazů je předmětem kapitoly 12, „Rozšíření LINQ".

Tuto vlastnost LINQ pro SQL popisujeme proto, abyste si byli vědomi skutečnosti, že nepotřebné části dotazu se před odesláním dotazu do SQL Serveru odstraňují. S přihlédnutím k tomuto faktu můžete skládat dotazy LINQ mnoha způsoby – například přidáváním nových predikátů a projekcí k původně rozsáhlému výběru řádků a sloupců, kdy se nemusíte příliš starat o nepotřebné elementy, které v dotazu zbyly.

Dotaz ve výpisu 4.29 se skládá nejprve z dotazu do tabulky Customers, který vybírá pouze zákazníky s názvem společnosti delším než 10 znaků. Tyto společnosti poté filtrujeme podle země a pracujeme s anonymním typem generovaným vnořeným dotazem:

Výpis 4.29 Ukázka redukce dotazu

```
var query =
    from s in (
        from   c in db.Customers
        where  c.CompanyName.Length > 10
        select new { c.CustomerID, c.CompanyName, c.ContactName, c.City,
                     c.Country, c.ContactTitle, c.Address }
    )
    where s.Country == "UK"
    select new { s.CustomerID, s.CompanyName, s.City };
```

Vygenerovaný dotaz SQL má tento tvar:

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[City]
FROM [dbo].[Customers] AS [t0]
WHERE ([t0].[Country] = @p0) AND (LEN([t0].[CompanyName]) > @p1)
```

Zde byla provedena dvě významná zkrácení. Jednak je zde pouze jedna tabulka, s níž pracuje klauzule *FROM*, namísto konstrukce *SELECT ... FROM (SELECT ... FROM)*, která by byla standardním výsledkem překladu původního stromu dotazu. A za druhé, došlo k odstranění nepotřebných polí; součástí projekce SELECT jsou pouze pole *CustomerID*, *CompanyName* a *City*, protože to jsou jediná pole, která uživatel tohoto dotazu LINQ potřebuje. První redukce zlepšuje čitelnost kódu, druhá zvyšuje výkon, protože omezuje velikost dat přenášených z databázového serveru na klienta.

Mísení kódu .NET s dotazy SQL

Dotazy LINQ pro SQL mají tvar stromu výrazu, který se překládá na odpovídající dotaz SQL. Již jsme si řekli, že existují určitá známá omezení LINQ pro SQL při používání celé palety funkcí .NET, které nelze beze zbytku přeložit na odpovídající operace T-SQL. To nemusí nezbytně znamenat, že nelze napsat dotaz s nepodporovanou metodou, ale měli byste si být vědomi toho, že takovou metodu nelze přeložit do T-SQL a bude vykonána lokálně na klientovi. Vedlejším efektem tohoto postupu může být, že části stromu dotazu závislé na metodě .NET, která nemá odpovídající překlad do T-SQL, budou plně pod správou LINQ pro objekty,

což znamená, že se na klienta musejí přenést všechna data, aby mohlo dojít k aplikaci požadovaných operátorů.

Tento efekt si ukážeme na několika příkladech. Podívejte se na dotaz LINQ ve výpisu 4.30.

Výpis 4.30 Dotaz LINQ s nativní manipulací s řetězcem v projekci

```
var query1 =
    from   p in db.Products
    where  p.UnitPrice > 50
    select new {
        ProductName = "** " + p.ProductName + " **",
        p.UnitPrice };
```

Vygenerovaný dotaz SQL obsahuje manipulaci s názvem produktu:

```
SELECT ("** " + [t0].[ProductName]) + " **" AS [ProductName],
   [t0].[UnitPrice]
FROM [dbo].[Products] AS [t0]
WHERE [t0].[UnitPrice] > 50
```

Pokusíme se přenést práci s řetězcem do rozšiřující metody .NET, což ukazuje výpis 4.31.

Výpis 4.31 Rozšiřující metoda pro manipulaci s řetězcem

```
static public class Extensions {
    public static string Highlight(this string s) {
        return "** " + s + " **";
    }
}
```

Ve výpisu 4.32 změníme dotaz LINQ a použijeme metodu *Highlight* definovanou ve výpisu 4.31.

Výpis 4.32 Dotaz LINQ volající v projekci metodu .NET

```
var query2 =
    from p in db.Products
    where p.UnitPrice > 50
    select new {
        ProductName = p.ProductName.Highlight(),
        p.UnitPrice };
```

Výsledek dotazu *query2* z výpisu 4.32 je stejný jako výsledek dotazu *query1* z výpisu 4.30. Ale dotaz SQL, poslaný do databáze, se odlišuje a chybí v něm operace s řetězcem:

```
SELECT [t0].[ProductName] AS [s],
       [t0].[UnitPrice]
FROM [dbo].[Products] AS [t0]
WHERE [t0].[UnitPrice] > 50
```

Pole *ProductName* se vrací pod názvem *s* a použije se jako parametr při volání metody *Highlight*. Pro každý řádek se zavolá metoda .NET *Highlight*. To není problém v případě, že výsledky dotazu *query2* ihned používáme. Ale pokud bychom tutéž operaci vložili do poddotazu, veškeré závislé dotazy nelze přeložit do nativního příkazu SQL. Podívejte se například na dotaz *query3* ve výpisu 4.33.

Výpis 4.33 Dotaz LINQ kombinující nativní a vlastní manipulaci s řetězcem

```
var query3 =
    from a in (
        from p in db.Products
        where p.UnitPrice > 50
        select new {
            ProductName = p.ProductName.Highlight(),
            p.UnitsInStock,
            p.UnitPrice
        }
    )
    select new {
        ProductName = a.ProductName.ToLower(),
        a.UnitPrice };
```

Dotaz SQL, vygenerovaný dotazem *query3* ve výpisu 4.33, je stejný jako dotaz generovaný dotazem *query2* ve výpisu 4.32, i přes přidání další manipulace s řetězcem (*ToLower*) pro pole *ProductName*:

```
SELECT [t0].[ProductName] AS [s],
       [t0].[UnitPrice]
FROM [dbo].[Products] AS [t0]
WHERE [t0].[UnitPrice] > 50
```

Jestliže odebereme volání metody *Highlight* a obnovíme původní práci s řetězcem přímo uvnitř dotazu LINQ, dostaneme opět kompletní nativní dotaz SQL, což ukazuje výpis 4.34.

Výpis 4.34 Dotaz LINQ používající nativní manipulaci s řetězcem

```
var query4 =
    from a in (
            from   p in db.Products
            where  p.UnitPrice > 50
            select new {
                ProductName = "** " + p.ProductName + " **",
                p.UnitPrice
            }
    )
    select new {
        ProductName = a.ProductName.ToLower(),
        a.UnitPrice
    };
```

Dotaz *query4* ve výpisu 4.34 generuje následující dotaz SQL, který nevyžaduje žádné další manipulace v kódu .NET:

```
SELECT LOWER([t1].[value]) AS [ProductName], [t1].[UnitPrice]
FROM (
    SELECT ("** " + [t0].[ProductName]) + " **" AS [value],
           [t0].[UnitPrice]
    FROM [dbo].[Products] AS [t0]
    ) AS [t1]
WHERE [t1].[UnitPrice] > 50
```

Až dosud jsme viděli, že určité dopady na výkon mohou nastat pouze v situaci, kdy použitá metoda .NET nemá odpovídající protějšek v SQL. Ale mohou nastat situace, které stroj LINQ pro SQL nemůže zpracovat, a dojde k výjimce za běhu – například pokud se pokusíme použít výsledek volání metody *Highlight* v predikátu *where*, podobně jako ve výpisu 4.35.

Výpis 4.35 Dotaz LINQ volající metodu .NET v predikátu where

```
var query5 =
    from p in db.Products
    where p.ProductName.Highlight().Length > 20
    select new {
        ProductName = p.ProductName.Highlight(),
        p.UnitPrice
    };
```

V čase provádění způsobí pokus o přístup k výsledkům dotazu *query5* (nebo pokus o vyžádání dotazu SQL) následující výjimku:

```
System.NotSupportedException
Method 'System.String Highlight(System.String)'
has no supported translation to SQL.
```

Jak vidíte, je důležité si uvědomovat, které operátory LINQ pro SQL podporuje, protože kód může za běhu dobře fungovat i selhat, v závislosti na použití těchto operátorů. Je obtížné stanovovat jiné základní pravidlo, než předcházet práci s nepodporovanými operátory. Pokud si myslíte, že určitý dotaz LINQ má kompozitní charakter a lze jej použít jako zdroj pro sestavení dalšího dotazu, jediným bezpečným vodítkem je používat operátory, které LINQ pro SQL podporuje.

Souhrn

V této kapitole jsme probrali funkce LINQ pro SQL pro dotazování na data. LINQ pro SQL vám umožňuje dotazovat se do relační struktury uložené v databázi Microsoft SQL Serveru takovým způsobem, aby bylo možné převádět dotazy LINQ na nativní dotazy SQL a v případě potřeby používat uživatelské funkce a uložené procedury. LINQ pro SQL pracuje s třídami entit, které se mapují na fyzickou strukturu databáze pomocí atributů či externích souborů XML. Uložené procedury či uživatelské funkce lze mapovat na metody třídy reprezentující databázi SQL Serveru. LINQ pro SQL podporuje většinu základních funkcí LINQ, které jsme si ukázali v kapitole 3.