Basic Handling and Tips of Database Relationships in Entity Framework

From CodeProject

Contents

Introduction

Entity Framework supports the relational database's one-to-many and many-to-many concepts. Usually, this topic isn't easy to explain very clearly and completely, particularly for people who don't have enough database knowledge. This article tries to fit all levels of readers by covering both basic database concept and the practical tips for Entity Framework's handling. If you are good enough on database, you can skip the database section. If you are good enough on Entity Framework, you can skip most of parts and read the conclusion and summary in the end first to see what you can get. This article is based on Entity Framework 4.0 ~ 4.3.1. Version 4.3.1 is the currently released version. Now let us review the basic relational database concepts first so that we can understand better the relevant feature in Entity Framework.

Using the Code

The sample code project includes some typical CRUD operations of DbContext mentioned in this article together with their UnitTest. Do two things before you use the sample code project a) install the database by opening the script file \DbRelationshipInDbContext\db.sql in Sql Server management studio, then click execute. You will need admin privilege for your database login to do so. Make sure your computer has folder "c:\temp\" because database files will be saved in this folder. b) update the data source part in the connection string from the configuration file of the project to point to your database server.

Disclaimer: the sample code is just for the purpose of explaining this article. Practically Implementing a generic persistence layer over Entity Framework is a better way to handle the data access operations.

Relationships in Database

One-to-Many relationship: Entity A can have many Entity B, but Entity B can have only one Entity A, then A and B is one-to-many relationship. For example, a biological father can have many children, but a child can only have one biological father. Therefore, biological father and child are one to many relationship. In relational database, one-to-many relationship doesn't need a matching table but a foreign key to handle. Practically, we can add the primary key in the table of one end as a foreign key in the table of many end. For the sample of biological father and child tables, we can add a column called FatherId into table Child to match the father from table Father. The FatherId in table Father is a primary key; the FatherId in table Child is a foreign key. So, the one-to-many relationship between father and child is well represented by the primary key and foreign key relationship.

Many-to-Many relationship: Entity A can have many Entity B; Entity B can have many Entity A. This relationship usually needs a matching table in a relational database for modeling. For example, Student and Club are many-to-many relationship because a student can join many Clubs; one club can have many students. In a relationship database, a matching table called "StudentClubMatch" to represent this many-to-many relationship. At least, this table should contains columns "StudentId" and ClubId to compose a composite primary key.

Technically, after a matching table is added, a many-to-many relationship has been broken into two one-to-many relationships. For example, due to the matching table StudentClubMatch, the many-to-many relationship of Student and Club has been broken into the following two one-to-many relationships:

Student and StudentClubMatch: one to many because one student can have many StudentClubMatch rows, but one row of StudentClubMatch can have only one student.

Club and StudentClubMatch: one to many because one Club can have many StudentClubMatch rows, but one row of StudentClubMatch can have only one Club.

Database Relationship Handling in Entity Framework

Overview

In order to let Entity Framework know these relationship, you need to create the relationship first in SQL server's or Sql server Express' with the help of diagram feature: below is a relationship diagram created in Sql server Express for student-class relationship and student-club relationship:

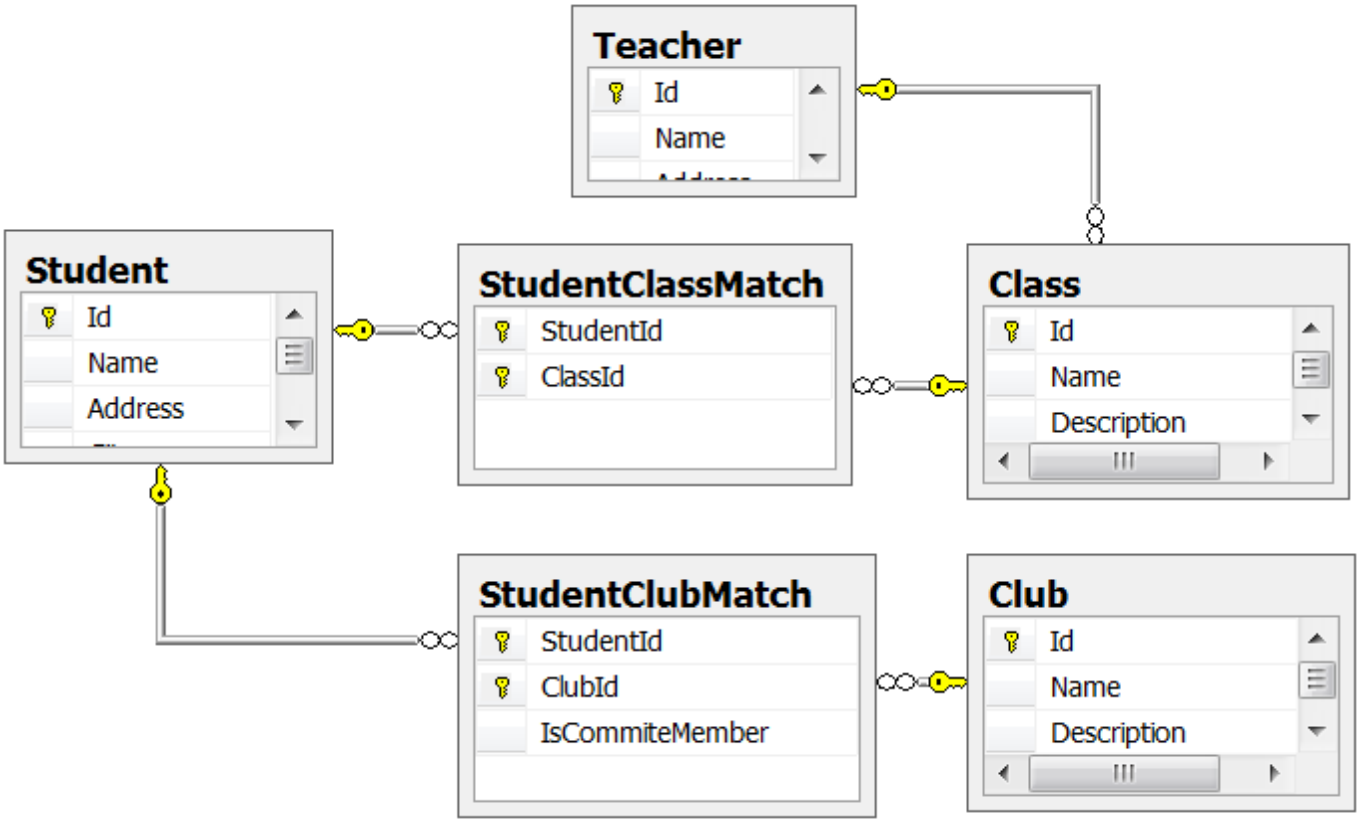

Diagram 1: Database Diagram

Database relationship handling in Entity Framework may not exactly match the handling in database. For example, the type I many-to-many relationship handling are different between database and Entity Framework; you can see this later in this article. Below is a Entity Framework Diagram generated by Entity Framework from above database; you can compare it with above database diagram to see the difference. Entity Framework adds navigation property into entity class to handle the database relationships. Usually there are two types of navigation properties: an entity object for the one end and a collection object for the many end. From Diagram 2, you can see the listed navigation properties for each Entity. For example, property `Class.Teacher` is a navigation property for one end; property `Teacher.Classes` is a navigation property for the many end.

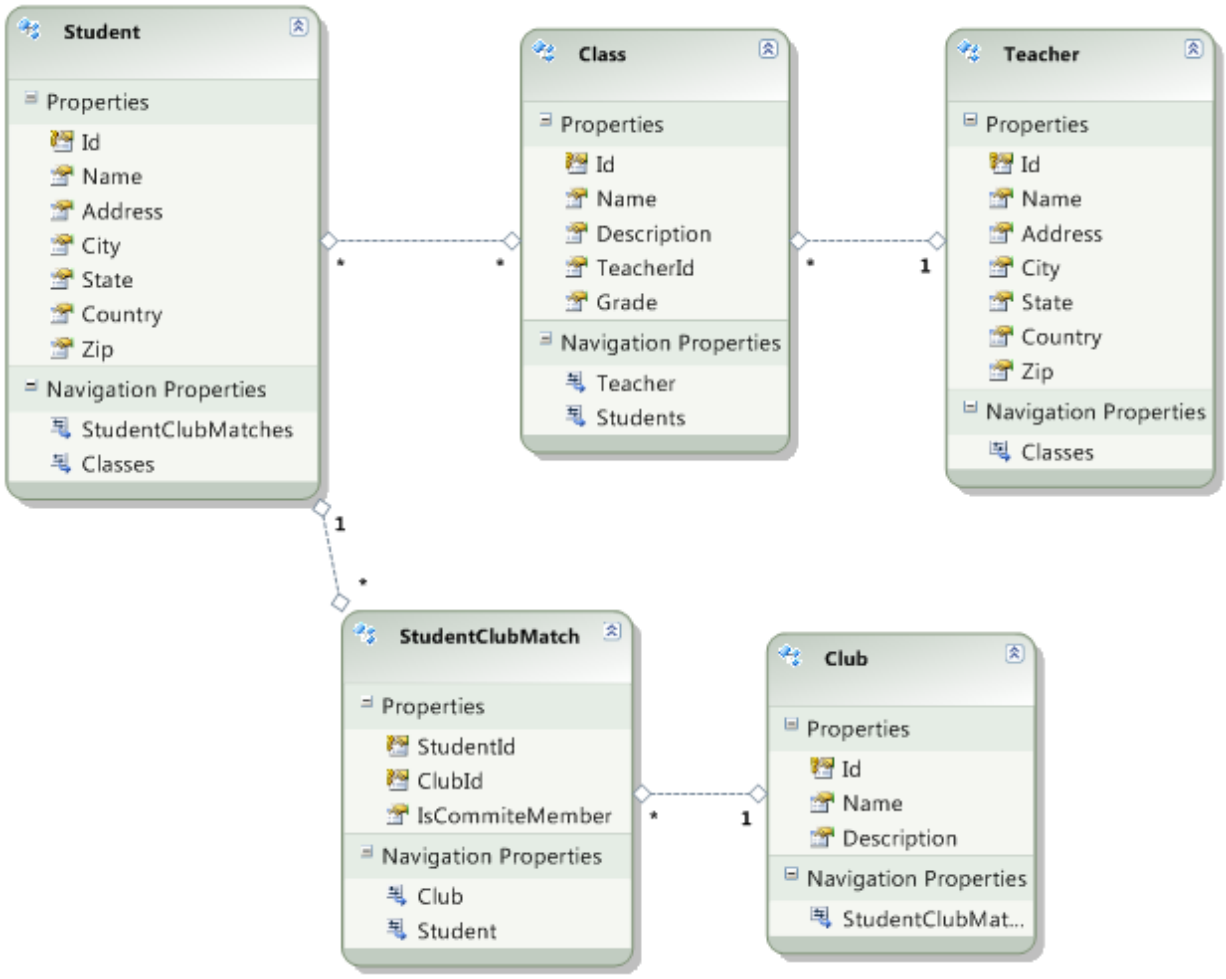

Diagram 2: Entity Framework Diagram

One-to-many Relationship Modeling in Entity Framework

Entity Framework will model this relationship by navigation properties: a foreign key navigation property to represent one end and a collection navigation property to represent the many end. For example, entity class Class has a foreign key navigation property Teacher to represent one teacher, and entity class Teacher has a collection-type navigation property Classes to represent many classes. Comparing the Diagram 1 and 2 above, you can find the database and Entity Framework handle this relationship in the very similar ways.

Many-to-many Relationship Modeling in Entity Framework

In a relationship database, a matching table is used to represent the many-to-many relationship. Entity Framework treats it by two possible ways, depending on whether or not there is any extra column in the matching table besides the two foreign key columns:

a. Type I: True many-to-many modeling. If the matching table has only two foreign key columns, Entity Framework will omit the matching table. For example, the matching table StudentClassMatch in database doesn't show up in the Entity Framework diagram 2 above. Why? Entity Framework can resolve everything without the need of the matching table.

   For this situation, Entity Framework uses collection-type navigation property in each involved entity class to model it. For example, navigation property `Student.Classes` and navigation property `Class.Students` represent the many-to-many relationship between Student and Class.

b. Type II: two one-to-many modeling. If the matching table has extra column besides the two foreign key columns, Entity Framework will need to keep the matching table. Therefore, the many-to-many relationship has been broken into two one-to-many relationships. The matching table StudentClubMatch belongs to this case. The table StudentClubMatch has an extra column IsCommiteMember, therefore, Entity Framework must include this table. In the Diagram 2 above, entity class StudentClubMatch is generated. Now the many-to-many relationship between Student and Club has been broken into two one-to-many relationships: one-to-many between Student and StudentClubMatch, one-to-many between Club and StudentClubMatch.

Lazy Loading in Entity Framework

When lazy loading is enabled in Entity Framework 4.0~4.3, the related object are loaded automatically only when the navigation properties are accessed. This is good because it avoids loading unnecessary things, and then potentially saves a lot of resources.

Ways to Enable Lazy Loading

In `ObjectContext`, set `ObjectContext.ContextOptions.LazyLoadingEnabled` to true.

However, in `DbContext`, you need to set true for both `DbContext.Configuration.LazyLoadingEnabled` and `DbContext.Configuration.ProxyCreationEnabled`. Otherwise the lazy loading won't work. This is why many people found the lazy loading doesn't work in Code first EF 4.3.1 because they don't know `ProxyCreationEnabled` is needed. Why do we need to set `ProxyCreationEnabled` true? The `DbContext` actually generated on the top of `ObjectContext` by the `DbContext` code generator. The generated entity classes are actually POCO entities, which do not have the same relationship requirements as objects that inherit from `EntityObject`, therefore they don't support lazy loading directly. In order to support lazy loading for these generated POCO entities, proxies for these POCO entities need to be created during runtime; the proxy classes are subclasses of these POCO entities.

The default values of `LazyLoadingEnabled` and `ProxyCreationEnabled` varies from `DbContext` and `ObjectContext`, and also varies from different code generators, therefore, it is good to set them up explicitly in your code when you create an instance of `ObjectContext` or `DbContext`.

Load Navigation Properties Explicitly

If the lazy loading is disabled in Entity Framework, you can load navigation properties explicitly in your code by several different ways, such as Function LoadProperty, Load, Include and etc. Following are some sample code to load these navigation property explicitly:

In ObjectContext:

```
_objectContext.LoadProperty(class, c=>c.Teacher);
_objectContext.LoadProperty(teacher, t=>t.Classes); or:
teacher.Classes.Load(); or:
Teacher teacher =
_objectContext.Teachers.Include(o=>o.Classes).FirstOrDefault();
```

In DbContext:

```
_dbContext.ObjectContext.LoadProperty(classObj, c=>c.Teacher);
_dbContext.ObjectContext.LoadProperty(teacher, t=>t.Classes); or
 _dbContext.Entry(classObj).Reference(c => c.Teacher).Load();
_dbContext.Entry(teacher).Collection(t => t.Classes).Load(); or:
Teacher teacher = _dbContext.Teachers.Include(o=>o.Classes).FirstOrDefault();
```

`DbContext` is built on top of `ObjectContext`, it has a property of `ObjectConext` which you can access directly for the feature of `ObjectContext`. For POCO entities in `ObjectContext`, function Load won't work, use `LoadProperty` or `Include` instead. You can pass a full path for Include, or chain up several Include calls. For the details of Include, find and read MSDN online document.

Why Do We Need to Load explicitly if We Have Lazy Loading Feature?

In some cases, we don't want the lazy loading to be enabled in Entity Framework. Two cases are listed below for this situation:

> **a)** If a collection navigation property will be potentially huge, we should avoid lazy loading. Otherwise, one navigation property may consume all of the resource of the computer. For example, in the case of table LogType and Log, one LogType may have million logs. Therefore, we should avoid the lazy loading on LogType's navigation property Logs. Instead, we can load the navigation property explicitly with filters to limit the data size to be retrieved; see the detail later in section Retrieve a subset of the many-end navigation property. Certainly, We can also control this in business layer so that client side won't be able to do this risky action carelessly.
>
> **b)** Serialization: if serialization is needed on the entity, such as WCF application, the lazy loading is good to be turned off to avoid the potential chained up serialization. The chained-up serialization will consume a lot resource and create a big serialized data which we may not want.

Load a Subset of the many-end Navigation Property From Database

Sometimes, we need to load a subset of the the many-end navigation property from database to save resource if the size of the the many-end navigation property is too big. This feature is used when lazy loading is disabled in Entity Framework:

In `ObjectContext`, `EntityCollection.CreateSourceQuery` can achieve this purpose:

```
List<Class> listT = teacher.Classes.CreateSourceQuery().Where(o =>
o.Id<10).ToList();
```

For `DbContext`, use following sampling way:

```
dbCxt.Entry(teacher).Collection(t=>t.Classes).Query().Where(o=>o.Id<10).Load(
);
```

Above ways are good while lazy loading is disabled. If lazy loading is enabled, then there is no need to use above way, we can use the filter functions Where, Select and etc of the collection-type navigation property directly because the whole navigation property has been loaded into memory. However, the sampling ways above are for retrieving subset from database, not from the memory.

Creation, Deletion and Edit in Entity Framework

We will only discuss some typical creation, deletion and edit below, assuming all passed-in parameters are detached entity; also assuming the lazy loading is enabled in Entity Framework. Reading from Entity Framework is kind of easy thanks to the flexible Linq feature; we will skip the Read feature in this article.

Overview (READ THE SECTION BELOW)

Creation

Usually there are two ways to handle the creation of a database relationship entity in Entity Framework:

1. The entity based way: Add the many-end entity into the Entity Framework explicitly, and let Entity Framework resolve the relationship implicitly, such as function AddClassVersion1 in section below.
2. The relationship based way: Create the one-to-many or many-to-many relationship explicitly by updating the navigation properties and let Entity Framework add this entity implicitly, such as functions AddClassVersion2 and AddStudentToClass in the sections below.

Deletion

For deletion, there are three situations which need to handle differently:

1. For one-to-many relationship, use the entity based way to delete an entity, see function DeleteClass below. Removing the relationship by the relationship based way may not delete the entity, depending on the business requirement, such as function RemoveAClassFromTeacher.
2. For Type I many-to-many relationship, use the relationship-based way to delete the relationship entity from the matching table, such as function RemoveStudentFromClass below.
3. For Type II many-to-many relationship, use the entity-based way to delete the relationship entity from the matching table, such as function RemoveStudentFromClub below.

One-to-many Relationship

Below are some sample functions for add/remove/delete/update a class for a teacher in `DbContext`. Teacher and Class have one-to-many relationship; Teacher is in the one end and Class is in the many-end. Here we mainly talk about the many end entity: Class.

```
        // entity based version
        public void AddClassVersion1(Class c)
         {
             _dbContext.Entry(c).State = EntityState.Added;
             _dbContext.SaveChanges();
             if (c.Teacher == null)
                  c.Teacher = _dbContext.Teachers.Single(o => o.Id ==
c.TeacherId);
         }

        // relationship based version
         public void AddClassVersion2(Class c)
         {
```

```
            if (c.Teacher == null)
                 c.Teacher = _dbContext.Teachers.Single(o => o.Id ==
c.TeacherId);
            c.Teacher.Classes.Add(c);
            _dbContext.SaveChanges();
        }

        public void UpdateClass(Class c)
        {
             C.Teacher = _dbContext.Teachers.Single(o => o.Id ==
c.TeacherId);
            _dbContext.Entry(c).State = EntityState.Modified;
            _dbContext.SaveChanges();
        }

       public void RemoveAClassFromTeacher(Class c) //After remove, this
class is still in table
        {
            if (c.Teacher == null) c.Teacher = _dbContext.Teachers.Single(o
=> o.Id == c.TeacherId);
            c.Teacher.Classes.Remove(c);
            c.Teacher = null;
            _dbContext.SaveChanges();
        }

       public void DeleteClass(Class c)
        {
            _dbContext.Entry(c).State = EntityState.Deleted;
            _dbContext.SaveChanges();
        }
```

Comment: There are two versions of functions of adding a Class: AddClassVersion1 and AddClassVersion12. AddClassVersion1 is implemented by the entity based way; AddClassVersion2 is is implemented by the relationship based way. Function `RemoveAClassFromTeacher` just removes the class from a teacher without deleting the Class from Entity Framework. After this action, the Class's TeacherId and Teacher become null. In order to make this function workable, the `Class.TeacherId` must be null allowed in database table, then the auto-generated entity Class will have a type of Nullable<int> for property TeacherId. Function `DeleteClass` completely removes this class from the Entity Framework and database table. Adding, editing and deleting of the one-end entity teacher are simple; we don't have samples here. For deletion of a teacher, we can either delete all of classes of this teacher, or we can keep all the classes of the teacher but set the TeacherId to null for each class, depending on the business requirement.

Even we have lazy loading enabled in Entity Framework, we still load the foreign key navigation property `Class.Teacher` explicitly in functions AddClassVersion1. But we don't need to load `Teacher.Classes` explicitly, which can be loaded correctly by lazy loading. I think this is a glitch in `DbContext`. Namely, Lazy loading for the foreign key navigation property doesn't work for the newly added item, but the lazy loading for the navigation property in the many end works fine, such as `Teacher.Classes`. This glitch occurs only for the newly added item. If you don't load `Class.Teacher` explicitly above for the newly added item, it will be null if it hasn't been loaded somewhere in Entity Framework. However, if it is loaded already somewhere, then

C.Teacher can be automatically resolved by Entity Framework. Whereas, for `ObjectContext`, lazy loading is fine for all types of navigation properties. Below is the code to add a class in `ObjectContext`; this function doesn't need any explicitly loading. This is an entity based version implementation; certainly we can also implement this by relationship based way too.

```
public void AddClass(Class c) {
    _objectContext.Classes.AddObject(c);
    _objectContext.SaveChanges();
}
```

Type I Many-to-Many Relationship

Entity Framework omits the matching table for this case. So, there isn't any matching entity in Entity Framework even though the matching table is in database. Student and Class are this case; StudentClassMatch is a matching table in database, but it isn't in Entity Framework. See Diagram 1 and Diagram 2. Below are some sample functions in DbContext for handling this relationship:

```
public void AddStudentToClass(Student s, Class c)
{
    s.Classes.Add(c);
    c.Students.Add(s);
    _dbContext.SaveChanges();
}

public void RemoveStudentFromClass(Student s, Class c)
{
    s.Classes.Remove(c);
    c.Students.Remove(s);
    _dbContext.SaveChanges();
}
```

Comment: Because there isn't an entity class in Entity Framework for table StudentClassMatch in database. So, the only way we can implement Add and Remove is to use the relationship based way. By updating the navigation properties, then the record will be added or removed from the matching table StudentClassMatch. There is no edit for this case because StudentId and ClassId are composite primary keys, changing any one of them actually will shift current row to another different row in table StudentClassMatch. `ObjectContext` handles this type many-to-many relationship in very similar ways.

Type II Many-to-Many Relationship

In this case, Entity Framework keeps the matching entity due to the extra column in the matching table. Student and Club belongs to this case. See Diagram 1 and 2. This many-to-many relationship is broken into two one-to-many relationships. See our discussion before in database section. Below are some sample functions in `DbContext`:

```
     // entity based version
       public void AddStudentToClubVersion1(Student s, Club c, bool
isCommitteMember)
        {
            //Create and add the matching table record.
            StudentClubMatch scMatch = new StudentClubMatch();
            scMatch.StudentId = s.Id;
            scMatch.ClubId = c.Id;
            scMatch.IsCommiteMember = isCommitteMember;
            _dbContext.Entry(scMatch).State = EntityState.Added;
            _dbContext.SaveChanges();
        }

        // relationship based version
        public void AddStudentToClubVersion2(Student s, Club c, bool
isCommitteMember)
        {
            //Create the matching table record first.
            StudentClubMatch scMatch = new StudentClubMatch();
            scMatch.Student = s;
            s.StudentClubMatches.Add(scMatch);
            scMatch.Club = c;
            c.StudentClubMatches.Add(scMatch);
            scMatch.IsCommiteMember = isCommitteMember;
            _dbContext.SaveChanges();
        }

        public void RemoveStudentFromClub(Student s, Club c)
        {
            StudentClubMatch scMatch =
_dbContext.StudentClubMatches.Single(o=>o.ClubId ==
                                                              c.Id &&
o.StudentId == s.Id);
            _dbContext.Entry(scMatch).State = EntityState.Deleted;
            _dbContext.SaveChanges();
        }

        public void UpdateStudentClubMatch(StudentClubMatch match)
        {
            _dbContext.Entry(match).State = EntityState.Modified;
            _dbContext.SaveChanges();
        }
```

Comment: `AddStudentToClubVersion1` is entity based; `AddStudentToClubVersion2` is relationship based. There are two one-to-many relationships in this case, therefore, in `AddStudentToClubVersion2`, we need to update two sets of one-to-many relationship.

For `UpdateStudentClubMatch`, the only thing we can update is the extra column `IsCommitteMember`. If you change either `StudentId` or `ClubId` for a `StudentClubMatch`, you actually shift to another different record since `StudentId` and `ClubId` are composite primary keys.

How Do We Know What Relationships Exist in Entity Framework?

There are several suggested ways listed below:

1) If you can access Entity Framework .edmx file, click it, you can see the entity relationship diagram in Visual Studio.

2) If you can access database, check the database diagram in Sql Server, which can tell you the relationship in database. The relationship in Entity Framework will vary a little bit for Type I many to many relationship.

3) Or you can check the generated entity classes of the Entity Framework itself. If you see entity type property in one entity class and see the related collection property in another matched entity class, then this is a one-to-many relationship, such as property Class.Teacher and Teacher.Classes. If you see the related collection properties in both matched entity classes, then they are many-to-many relationship, such as properties Class.Students and Student.Classes.

Conclusions

1. When we implement business operations over Entity framework, be sure to get clear firstly what relationship is involved in this operation: no relationship, one-to-many or many-to-many relationship. For each type relationship, we may need to handle it in different ways in Entity Framework. The knowledge of basic database relationship will be very helpful.

2. There are two ways to handle the creation of a database relationship entity in Entity Framework:

a) The entity based way: add the many-end entity into the Entity Framework explicitly, and let Entity Framework resolve the relationship implicitly. **b)** The relationship based way: create the one-to-many or many-to-many relationship explicitly by updating the navigation properties and let Entity Framework add this entity implicitly.

3. Entity Framework handle two types of many-to-many relationship in different ways.

Type I: true many-to-many relationship modeling. For this type, the matching table in database has only two foreign key columns. Entity Framework won't create an entity class for this matching table because of no need. Adding and removing items in the matching table for this type is implemented by the relationship based way by updating the two many-end navigation properties.

Type II: Extra column is in the matching table besides the two foreign key columns. Entity Framework must create an entity class to model this matching table. The relationship is broken into two one-to-many relationships. The add operation for the item in the matching table is

similar to handling two one-to-many relationships either by entity based way or by relationship based way.

4. For deletion, there are three situations which need to handle differently: **a)** For one-to-many relationship, use the entity based way to delete an entity. **b)** For Type I many-to-many relationship, use the relationship-based way to delete the relationship entity from the matching table. **c)** For Type II many-to-many relationship, use the entity-based way to delete the relationship entity from the matching table.

5.The navigation property can be loaded either by lazy loading or manually loading by users, depending on whether or not the lazy loading is enabled in Entity Framework.

6. For DbContext, enabling lazy loading needs to set both DbContext.Configuration.LazyLoadingEnabled and DbContext.Configuration.ProxyCreationEnabled true. However, for ObjectContext, only need to set ObjectContext.ContextOptions.LazyLoadingEnabled true.

7. Loading navigation property explicitly can be done by several ways, such as functions LoadProperty, Load, Include and etc. DbContext and ObjectContext have different ways to handle this; see sample codes before for details. For POCO in ObjectContext, don't use Load but use LoadProperty or Include instead.

8. It is good to turn off lazy loading in Entity Framework in some cases:

a) A potential big size for the many-end navigation property, such as LogType.Logs. **b)** Entity Serialization is needed, such as WCF. To avoid the possible chained-up serialization, turning off lazy loading for this case.

9. For newly added entities, lazy loading in DbContext seems to have a glitch that the one-end navigation property won't be loaded as expected if it hasn't been loaded before. But the many-end navigation property is fine. For existing entities in DbContext, this glitch doesn't occur. For any entity in ObjectContext, this glitch doesn't occur.

10. Entity Framework support nullable property in both regular Entities and POCO entites if the column in database side is null allowed, such as property TeacherId in entity class Class:

```
public Nullable<int> TeacherId { get; set; }
```